# Real-Time Special Effects –

## Neue Möglichkeiten durch benutzerprogrammierbare Grafik-Hardware

Diplomarbeit

**Eingereicht von Jürgen Kett**

Matr.-Nr. 1601974

Prof. Dr.-Ing. Detlef Krömker

Professur für Graphische Datenverarbeitung

*Betreuer:* Dr. Ralf Dörner und Dipl.-Inform. Paul Grimm

11.10.2002

Fachbereich Biologie und Informatik

Johann Wolfgang Goethe-Universität Frankfurt am Main

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Hilfsmittel verwendet und sämtliche Stellen, die benutzten Werke im Wortlaut oder dem Sinn nach entnommen sind, mit Quellenangaben kenntlich gemacht habe.

Hattersheim, den 10.10.2002

## Aufgabenstellung

Viele Spezialeffekte, die vor einigen Jahren noch nächte- oder wochenlange Berechnungen zur Erzeugung brauchten, können heute in Echtzeit generiert werden. Die neueste Generation von Grafikkarten für den Massenmarkt erlaubt es, Grafikroutinen direkt auf der Grafikkarte zu hinterlegen. Dadurch ist es möglich, nicht nur spezialisierte Shader zu schreiben (z.B. für Effekte wie Spiegelungen und Lichtbrechungen), sondern auch physikalische Effekte als Oberflächenmerkmale von 3D-Objekten anzusehen (z.B. Partikelsysteme für Haare oder Kleidung).

Ziel der Arbeit ist es, zwei bis drei Spezialeffekte als Leistungsdemonstratoren für die neuen Technologien umzusetzen. Grundlage für die Arbeit soll eine Studie bilden, die den heutigen Stand auf dem Gebiet der Spezialeffekte widerspiegelt. Darauf aufbauend sollen geeignete Spezialeffekte ausgewählt werden und mittels der DirectX8.0-Technologie prototypisch umgesetzt werden. Abschließend soll eine Bewertung der neuen Technologien und der Entwicklungsumgebungen durchgeführt werden.

## Kurzfassung

Grafik-Hardware ist programmierbar geworden. Graphic Processing Units (GPUs) der neuen Generation wie der GeForce3 von NVIDIA enthalten Prozessoren, die es dem Software-Entwickler erlauben kurze Routinen auf der Grafik-Hardware auszuführen.

Ich gebe in dieser Arbeit einen umfassenden Überblick über die Architekur und Leistungsfähigkeit dieser neuen Chipgeneration, zeige deren Stärken und Schwächen auf und diskutiere Verbesserungsvorschläge.

Als Teil der Arbeit präsentiere ich einige von mir entwickelte Schattierungsverfahren, sowie eine Wassersimulation. Diese Demonstratoren sind darauf ausgerichtet vollständig auf den Prozessoren der neuen Grafikchip-Generation zu laufen.

Als Antwort auf die Mängel der zur Zeit verfügbaren Application Programming Interfaces stelle ich ein alternatives Interface zur Steuerung der neuen GPU-Komponenten vor, das insbesondere die Austauschbarkeit und Kombinierbarkeit von GPU-Programmen unterstützt.

## Abstract

It is possible now to program graphical hardware. Graphic Processing Units (GPUs) of the new generation as GeForce3 of NVIDIA are containing processors which allow the software-developer to carry out short routines on the graphics hardware.

In this thesis I will give a detailed summary of the structure and the capabilities of this new chip generation. Its pros and cons will be considered as well as I will discuss suggestions of improvement.

One part of this paper will be a presentation of some shading methods and a water simulation both developed by myself. These demonstrators are directed to run completely on the processors of the new graphic chip generation.

Refering to the defaults of the Application Programming Interfaces available right now, I will present an alternative Interface to drive the new GPU components. This Interface supports in particular the interchange-ability and the ability of combination of the GPU programs.

# Inhaltverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Die Entwicklungen im Bereich *Realtime Rendering* sind enorm. Immer komplexere Modelle mit immer abwechslungsreicheren Oberflächen finden ihren Platz in interaktiven Anwendungen. Die Qualitätslücke zwischen nicht-zeitkritischen Software-Rendering-Systemen und Echtzeit-Systemen wird zunehmend kleiner.

Grund hierfür sind die Fortschritte, die man in den letzten Jahren im Bereich der Grafik-Hardware gemacht hat, denn ohne den Beschleunigungseffekt dieser Komponenten wäre die Datenflut der Bildbeschreibungen nicht in dieser Geschwindigkeit zu bewältigen. Mit der letzten Grafikchip-Generation erschienen zum ersten Mal Beschleuniger auf dem Massenmarkt, die den gesamten Prozess der Bilderzeugung in Hardware implementierten. Dieser Schritt brachte aber auch einen negativen Aspekt mit sich: Die starre und geschwindigkeitsoptimierte Hardware beschränkte den Programmierer in seiner Flexibilität. Besonders im Bereich der Schattierung von Objekten schrumpfte der Einfluss des Programmierers auf das Setzen einer Hand voll ausgewählter Attribute.

Die neue Grafikchip-Generation soll hier Abhilfe schaffen. Die Hardware wurde um benutzerprogrammierbare Prozessoren erweitert. Auf diesen Prozessoren kann der Programmierer kurze Routinen ablegen, welche den Bildgenerierungsprozess flexibel steuern, ohne dass sich dadurch die Verabeitungsgeschwindigkeit senkt.

Die Hersteller dieser Chips werben damit, dass mit dieser Architektur „unendlich viele Effekte“ erzielbar sind. Forscher und Software-Entwickler sind bereits dabei, basierend auf der Architektur, spezielle High-Level-Sprachen zur Steuerung des Rendering-Prozesses zu entwickeln – Sprachen, wie man sie bisher nur von Software-Rendering-Systemen kannte.

Hat man damit nun tatsächlich beides erreicht: ein hohes Maß an Flexibilität *und* eine hohe Verarbeitungsgeschwindigkeit? Sind die Möglichkeiten der neuen Hardware wirklich so uneingeschränkt, wie die Hersteller behaupten? Ist „Benutzerprogrammierbarkeit“ der Weg, den alle künftigen Hardware-Generationen gehen müssen, um in der Bildqualität mit Software-Rendering-

Systemen weiter aufzuschließen, und was müsste sich an der Technik in Zukunft noch verbessern?
Dies sind die Fragen, mit denen sich diese Arbeit beschäftigt.

## 1.1 Vorgehensweise

Ich habe zunächst eine Grundlagenanalyse zur Klärung der Fragen durchgeführt, welche Anstrengungen bisher unternommen worden waren, um die Lücke zwischen Echt-Zeit-Bildgenerierung und Software-Rendering-Systemen zu schließen und welche Fortschritte man bisher auf diesem Gebiet gemacht hatte.
Um hierfür ein Maß zu haben, war es zunächst notwendig Qualitätsmerkmale zur Bewertung von Rendering-Systemen festzulegen. Neben der Qualität der generierten Bilder spielt hier auch die Benutzbarkeit des Systems eine entscheidende Rolle.
Basierend auf diesen Qualitätsmerkmalen untersuchte ich die Funktionalität und Benutzbarkeit der bisher auf dem Massenmarkt erschienenen Grafikchips und studierte die Forschungsarbeiten der letzten Jahre, die eine Verbesserung der Lage zum Ziel hatten. Inbesondere richtete ich mein Augenmerk darauf, inwieweit man eine Verbesserung durch die Einführung *benutzerprogrammierbarer* Grafik-Hardware zu erzielen versucht hatte.
Um nun Aussagen über Aufbau, Funktionsweise, Leistungsfähigkeit und Benutzbarkeit der neuen Grafikchip-Generation machen zu können, und um letzendlich beurteilen zu können, inwieweit die neue Technik die Lücke zum Software-Rendering zu schließen vermag, führte ich mehrere Studien durch. Zum einen war dies eine ausführliche Architekturstudie zur Klärung von Aufbau und Funktionsweise, zur Klärung der Benutzbarkeit eine Studie über die verschiedenen Interfaces, die zu Steuerung entworfen worden waren bzw. noch immer in Arbeit sind, und zur Klärung der Leistungsfähigkeit eine Studie über die Effekte, die verschiedene Programmierer auf Basis der neuen GPUs bisher realisiert hatten. Um diese Aussagen zu untermauern und ein Gefühl für die neue Technik zu bekommen, entwickelte ich Leistungsdemonstratoren, mit dem Ziel die neuen Möglichkeiten weitestgehend auszunutzen.

Basierend auf den Ergebnissen dieser Studien und den praktischen Erfahrungen mit den Chips, formulierte ich Kritikpunkte und entwickelte Verbesserungsvorschläge – sowohl für die Entwicklung von Interfaces, als auch für die Entwicklung zukünftiger Grafikchips.

## 1.2 Aufbau der Arbeit

Meine Arbeit gliedert sich im Wesentlichen in vier Teile, gleichbedeutend mit den Kapiteln 2 bis 5. Der erste Teil bietet eine kurze Einführung in den Bereich Real-Time-Rendering und präsentiert die Ergebnisse meiner Grundlagenanalyse.
Im zweiten Teil gehe ich auf den architektonischen Aufbau der neuen GPU-Generation ein und beschreibe deren Einbindung in die sie steuernden Application Programming Interfaces. Der dritte Teil enthält die Ergebnisse meiner Studie über die Leistungsfähigkeit der neuen GPU-Generation. Ich orientiere mich hierbei vor allen Dingen an den von mir entwickelten Leistungsdemonstratoren. Im vierten und letzten Teil gebe ich basierend auf den Aussagen der vorherigen Teile eine ausführliche Kritik zur Benutzbarkeit und Leistungsfähigkeit ab und präsentiere mögliche Verbesserungsvorschläge, die zur Entschärfung oder Beseitigung der identifizierten Problembereiche eingesetzt werden könnten.
Ich schließe diese Arbeit mit einer Zusammenfassung der Arbeit und einem Ausblick auf kommende Entwicklungen.

# 2 Grundlagen der Analyse

## 2.1 Realtime Rendering – Bildgenerierung in Echtzeit

Beginnen möchte ich mit einigen Worten zur allgemeinen wissenschaftlichen Umgebung, in der sich diese Arbeit bewegt:
Grafikchips sind Hardwarekomponenten, die die Generierung von Computerbildern beschleunigen – eine Thematik, die in den Forschungsbereich der Grafischen Datenverarbeitung (GDV) fällt. Die GDV, eine Disziplin der Informatik, bietet Verfahren und Möglichkeiten mittels Rechnern, Bilder zu produzieren und zu manipulieren. Einen Überblick über diese Disziplin bieten beispielsweise die Werke [Fole97], [Watt00], sowie [Enca97a] und [Enca97b].

Die GDV lässt sich nach Encarnacao et al. [Enca97a, S.11-15] – ausgehend von Rosenfelds Klassifizierung [Nake72] – in die Teilgebiete *generative Computergrafik*, *Bildverarbeitung* und *Bildanalyse* untergliedern (vgl. Tabelle 1).
.

| Disziplin | Eingabe | Ausgabe |
|---|---|---|
| *Bildverarbeitung* | Bild | Bild |
| *generative Computergrafik* | Bildbeschreibung | Bild |
| *Bildanalyse* | Bild | Bildbeschreibung |

**Tabelle 1: Klassifizierung der Grafischen Datenverarbeitung nach Rosenfeld [Enca97a]**

Bilder und Bildbeschreibungen sind hierbei rechnerinterne Datenformate, und ich werde in den nächsten Abschnitten auf die Unterschiede zwischen beiden Repräsentationsformen für Bilddaten eingehen.
Diese Arbeit beschäftigt sich im Wesentlichen mit der Generierung von Bildern aus Bildbeschreibungen (Rendering), also mit *generativer Computergrafik*. Aber

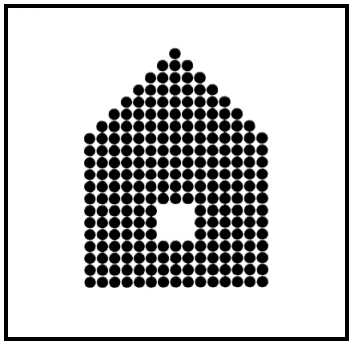
auch die *Bildverarbeitung* (Image Processing) wird an einigen Stellen eine Rolle spielen.

### 2.1.1 Bildbeschreibung und Bild

Ich werde nun die Bedeutung der Begriffe Bildbeschreibung und Bild konkretisieren, denn dies sind die Ein- und Ausgaben des Prozesses, der in dieser Arbeit im Mittelpunkt stehen wird: *die 3D-Bildgenerierung in Echtzeit* (3D Realtime Rendering).

Als Sichtgeräte für Rechner werden heutzutage standardmäßig Rastergeräte verwendet. Bei diesem Gerätetyp setzt sich ein Bild aus einem rechteckigen Raster von Bildpunkten zusammen, ähnlich einem Mosaik. Daher versteht man in der GDV unter einem Bild *eine Menge von n x m Bildpunkten* bzw. *Pixels* (kurz für: „picture elements“) [Enca97a, S.14]. Somit ist die Ausgabe des Bildgenerierungsprozesses ein Raster von Bildpunkten, ein sogenanntes *Rasterbild* oder *Bitmap* (vgl. Abbildung 1).

Zur Definition der Farbe eines Bildpunktes werden üblicherweise die drei Primärfarben Rot, Grün und Blau verwendet (RGB-Modell [Enca97a, S.275-276]). Für jede der Primärfarben wird anhand eines Wertes zwischen 0 und 1, eine Intensität angegeben. Durch Addition der drei Primärfarbintensitäten ergibt sich die darzustellende Farbe (additive Farbmischung). Die darstellbaren Farben werden also als Punkte eines Einheitswürfels im Ursprung des kartesischen Koordinatensystems beschrieben (vgl. Abbildung 2).



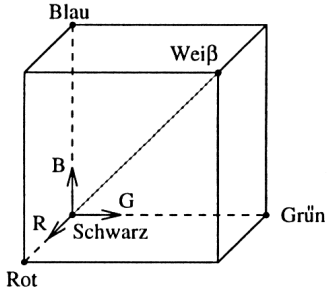



**Abbildung 1: (links) Beispiel für ein Rasterbild [Enca97a]**

**Abbildung 2: (rechts) Der RGB-Einheitwürfel [Enca97a]**

Grauwerte, darstellbar durch gleichgroße Anteile von Rot, Grün und Blau, liegen auf der Hauptdiagonalen des Einheitswürfels mit Schwarz im Ursprung [0,0,0] und Weiß im Punkt [1,1,1].

Die Eingabe des Bildgenierungsprozesses ist wie gesagt eine Bildbeschreibung. Es gibt in der generativen Computergrafik zahlreiche Formen der Bildbeschreibung, und man unterscheidet diese Formen nach den Grundelementen, aus denen sich die Beschreibungen zusammensetzen. Auch ein Rasterbild ist eine Bildbeschreibung: Ein Bild wird mit Hilfe einer Menge von Bildpunkten beschrieben. Die Grundelemente - auch *grafische Primitiva* genannt [Enca97a, S.21] - sind in diesem Fall die Bildpunkte bzw. Pixels. Bilder können aber auch mittels anderer Grundelemente beschrieben werden, beispielweise mittels Geraden bzw. Geradenabschnitten (Vektorgrafik), Kurvenabschnitten (Strichgrafik) oder Flächen (Flächengrafik) [Enca97a, S.21].
Einen guten Überblick über verschiedene Beschreibungsformen gibt [Enca97b].

In der 3D-Bildgenerierung sind die Eingaben Beschreibungen *dreidimensionaler Objekte*. (Ich bevorzuge daher die Begriffe *Objekt-* oder *Szenenbeschreibung* gegenüber dem Wort „Bildbeschreibung“.) Für diesen Bereich werden am häufigsten Beschreibungsformen eingesetzt die *Flächenelemente* als Grundelemente verwenden. Im Speziellen Falle der 3D-Bildgerierung *in Echtzeit* hat sich bis dato das *Dreieck* als Grundelement durchgesetzt. Abbildung 3 zeigt beispielhaft, wie Dreiecke zur Repräsentation von dreidimensionalen Objekten eingesetzt werden können.

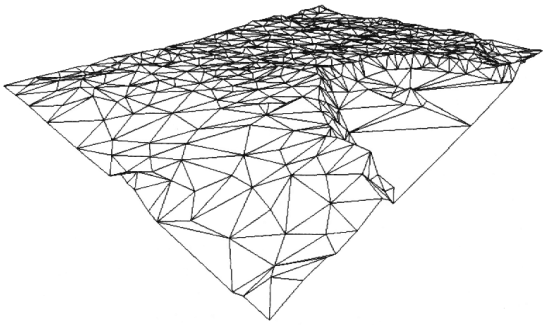


**Abbildung 3: Repräsentation eines Landschaftsausschnittes durch ein Netz von Dreiecken [Möll99]**

Ein Dreieck lässt sich durch Angabe seiner drei *Eckpunkte* (*Vertices*) beschreiben. Ein solcher Punkt wird durch Angabe seiner *x*-, *y*-, und *z*-Koordinaten im dreidimensionalen kartesischen Koordinatensystem angegeben. Die Koordinaten geben den Abstand des Punktes vom Ursprung an (vgl. Abbildung 4).

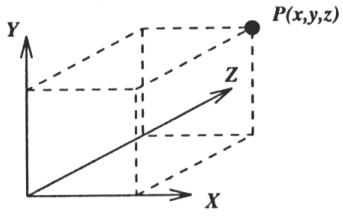



**Abbildung 4: Ein Punkt im dreidimensionalen kartesischen Koordinatensystem [Enca97a]**

Neben solchen geometrischen Daten können einer Bildbeschreibung noch zahlreiche weitere Informationen hinzugefügt werden. Denn bei der Bildgenerierung wird im Allgemeinen nicht nur eine Darstellung der Geometrie erzeugt (Drahtgitter-Darstellung), die Objekte werden meist auch schattiert. Die Berechnung der Schattierung – bildlich gesprochen: das Ausmalen der Dreiecke – erfordert zusätzliche Attibute. Welche Attribute dies im genauen sind, hängt von der verwendeten Schattierungsmethode ab. Verbreitet sind beispielsweise Materialeigenschaften wie Farbe, Transparenz und Grobheit. Die Farbe und Transparenz werden häufig mittels eines einzigen Vektors repräsentiert, der sich

aus einem RGB-Wert und einer zusätzlichen Komponente „Alpha“ zusammensetzt (RGBA-Wert). Diese repräsentiertd die Transparenz der Oberfläche: *0* entspricht völliger Durchsichtigkeit, *1* völliger Undurchsichtigkeit [Möll99, S.85-89]. Neben Materialeigenschaften spielen auch nach zahlreiche andere Attribute eine Rolle. Beispielsweise sind in der Regel Beschreibungen von Lichtquellen und Kameraeigenschaften Teil der Szene.
Attribute werden je nach Bedeutung einmal für jedem Eckpunkt definiert werden (Per-Vertex-Attribute), einmal für jedes Dreieck (Dreiecks-Attribute), einmal für jedes Objekt (Objekt-Attribute) oder einmal für die gesamte Szene (Szenen-Attribute).

Die 3D-Bildbeschreibungen (bzw. Objektbeschreibungen), die in dieser Arbeit eine Rolle spielen, sind also Mengen von Dreiecken, die jeweils durch Angabe der Koordinaten ihrer drei Eckpunkte im dreidimensionalen kartesischen Koordinatensystem definiert werden. Darüber hinaus werden der Beschreibung verschiedene Attribute hinzugefügt, die vor allem nicht-geometrische Eigenschaften repräsentieren. Der Prozess der 3D-Bildgenerierung (3D Rendering) erzeugt aus solchen Bildbeschreibungen zweidimensionale farbige Rasterbilder, die auf den Ausgabegeräten eines Rechners dargestellt werden können.

### 2.1.2 Der Bildgenerierungsprozess

Zum Verständnis dieser Arbeit ist ein grundlegendes Wissen über den Prozess der Bildgenerierung und über die Hardware- und Softwarekomponenten, die diesen Prozess ermöglichen (Grafiksystem), notwendig.
Eine detaillierte und allgemeine Betrachtung dieser Aspekte würde den Rahmen dieser Arbeit sprengen. Es existieren zahlreiche Verfahren, um aus Objektbeschreibungen, wie ich sie im letzten Abschnitt beschrieben habe, Rasterbilder zu erzeugen. Einen Überblick über den Prozess im Allgemeinen und über spezielle Verfahren bieten u.a. die Werke [Fole97], [Blin96], [Watt00] und für den Echtzeit-Bereich [Möll99]. Ich werde mich nun gleich dem speziellen Bereich der Bildgenerierung in *Echtzeit* zuwenden.

#### *2.1.2.1 Zum Begriff „Echtzeit“*

Echtzeit ist eine Forderung an die *Bildgenerierungsrate*. Eine Echtzeit-Anwendung sollte im Minimum ungefähr 15 Bilder pro Sekunde produzieren. Denn dies ist in etwa die Grenze, ab der ein Betrachter eine Animation nicht mehr als eine Abfolge von einzelnen Bildern wahrnimmt, sondern den Eindruck eines flüssigen Bewegtbildes gewinnt [Möll99, S.1]. Das Standardfilmformat liegt bei 24 Bildern pro Sekunde (*frames per second*, kurz: *fps*).

Für Filme können die einzelnen benötigten Bilder (Frames) vorproduziert werden, denn die Inhalte der Frames können beim späteren Abspielen nicht vom Zuschauer beeinflusst werden. Die Bildgenerierungsrate ist also unabhängig von der späteren Abspielrate.

Lässt man aber Interaktivität zu, so dass die Inhalte der Bilder von den Aktionen des Benutzers abhängen, wie beispielsweise bei einem Flugsimulator, dann können diese Bilder erst während der Ausführung des Programms berechnet werden. In diesem Fall ist die Bildgenerierungsrate abhängig von der benötigten Abspielrate. Im Realtime Rendering darf die Generierung eines Bildes also maximal so lange dauern, dass die ungefähre Schranke von 15fps nicht unterschritten wird.

#### *2.1.2.2 Rendering Pipeline*

Ich möchte nun kurz den Prozess vorstellen, der mehr als 15 Bilder pro Sekunde erzeugen soll: Abbildung 5 zeigt eine typische Pipeline zur Bildgenerierung in Echtzeit.

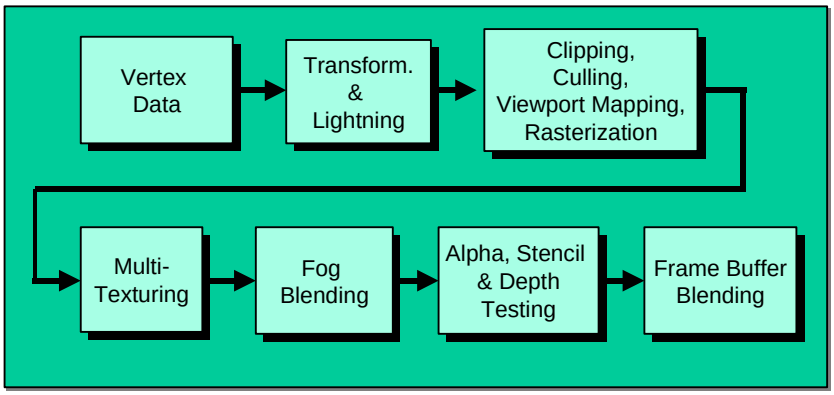



**Abbildung 5: Eine typische Realtime-Rendering-Pipeline**

Am Anfang des Prozesses steht die Bildbeschreibung – eine Menge von Dreieckseckpunkten (Vertices, Singular: Vertex) und deren Attribute wie z.B.:

- Position (*x*-,*y*- und *z*-Koordinate),
- Texturkoordinaten (*u*- und *v*-Koordinaten),
- Vertex-Normale (*x*-, *y*-, und *z*- Koordinaten),
- Materialeigenschaften (RGBA-Farben).

Am Ende des Prozesses befindet sich das generierte Bild im Bildspeicher (Frame Buffer [Enca97a, S.67-70]) und wird anschließend vom Video Controller (Image Display System [Enca97a, S. 49-55]) auf dem Monitor angezeigt.
Die Verarbeitungsschritte, die zur Bildgenerierung durchgeführt werden, sind folgende.

***Geometrische Transformation:***

Der „Transform&Lightning"-Schritt führt mehrere lineare Transformationen durch: Das durch die Daten repräsentierte Objekt wird in die zu generierende Szene eingefügt. Hierzu wird es skaliert, verschoben und ausgerichtet (world transform). Anschließend folgt die Einstellung des Blickwinkels, aus dem die Szene betrachtet werden soll, und das Setzen des Ausschnitts, der aus diesem Winkel zu sehen sein soll (view transform, vgl. Abbildung 6). Schließlich wird die Szene auf eine Ebene (view plane) projiziert [Watt00, S. 149-166], so dass eine zweidimensionale Bildbeschreibung entsteht (projection transform). Die Tiefen-Informationen, diese entsprechen den z-Koordinaten der Vertices nach

dem „view transform“-Prozess, werden aber nicht verworfen, sondern auf den Wertebereich [0,1] abgebildet und gespeichert. 0 entspricht hierbei dem nahesten Punkt, 1 dem am weitesten entfernten. Diese Informationen werden benötigt, um später ermitteln zu können, welche Teile der Bildbeschreibung andere verdecken.
All diese Transformationen werden in der Regel über Matrix-Multiplikationen durchgeführt [Möll99, S.23-64], und die Matrizen werden entsprechend der jeweiligen Transformation häufig mit *world, view* und *projection matrix* bezeichnet. Ich verwende die deutschen Ausdrücke *Welt-*, *Kamera-* und *Projektions-Matrix*.

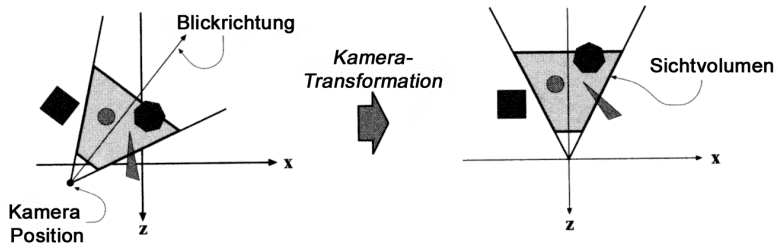



**Abbildung 6: Kamera Transformation (View Transform) [Möll99]**

***Beleuchtungsrechnung:***

Parallel zu den eben beschriebenen Transformationen werden für jeden Vertex sogenannte Beleuchtungsrechnungen durchgeführt (Lightning). Diese Berechnungen bestimmen später die Farbe und Intensität an allen Punkten der Oberfläche. Für solche Beleuchtungsrechnungen werden der Szene eine Anzahl von virtuellen Lichtquellen hinzugefügt. Ein Beleuchtungsmodell beschreibt nun, wie eine Oberfläche mit diesen Lichtquellen interagiert. Das verbreitetste Beleuchtungsmodell ist das von Phong [Phon75]. Er beschreibt das Reflektionsverhalten einer Oberfläche durch die gewichtete Summation einer *ambienten, diffusen* und *spekularen* Komponente. Diese Komponenten berechnen sich wiederum aus den Eigenschaften der Lichtquellen (z.B. Farbe und Intensität des ausgesandten Lichtes) und den Materialeigenschaften der Oberfläche (z.B. Farbe, Mattheit) sowie aus deren Ausrichtung zueinander und relativ zum Betrachter (z.B. Einfallswinkel des Lichtes). Neben diesem sehr einfachen

Verfahren gibt es aber auch noch zahlreiche andere Beleuchtungsmodelle [Watt00], S. 205-221].

***Clipping, Culling, Viewport Mapping und Rasterization:***

Im nächsten Schritt werden alle Teile der Bildbeschreibung, die sich außerhalb des gewählten Szenenausschnittes befinden (View Frustrum Clipping, vgl. Abbildung 7), und i.d.R. alle Dreiecksrückseiten (Culling) entfernt. Die verbleibende Bildbeschreibung wird nun an die Position und die Ausmaße, die sie später auf dem Monitor haben soll, angepasst (Screen Mapping, vgl. Abbildung 8).

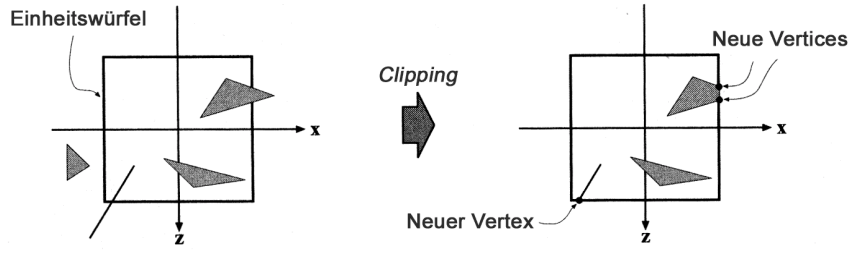



**Abbildung 7: View Frustrum Clipping [Möll99]**

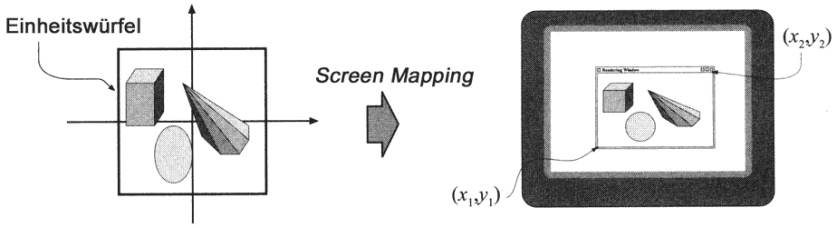



**Abbildung 8: Screen Mapping [Möll99]**

Der folgende Verarbeitungsschritt ermittelt für jedes Dreieck welche Rasterpunkte des Sichtfensters es überdeckt (Rasterization, Scan Conversion [Akel88]). Für jeden überdeckten Rasterpunkt – häufig auch *Fragment* genannt - werden *Texturkoordinaten* und *zwei Farbwerte* berechnet (der spekulare und diffuse Anteil aus der Beleuchtungsrechnung). Dies geschieht durch lineare Interpolation zwischen den jeweiligen Attributen der drei Vertices des Dreiecks, zu dem das

Fragment gehört: Die Texturkoordinaten eines Fragments werden aus den Texturkoordinaten der drei Vertices interpoliert und die Farbwerte des Fragmentes aus den Ergebnissen der Beleuchtungsrechnung (Gouraud Shading [Gour71]).

***Multi-Texturing und Fog Blending:***

Die Multi-Texturing-Stufe berechnet aus diesen interpolierten Daten [Möll99, S.121-122] einen einzelnen RGBA-Wert:

Mittels der interpolierten Texturkoordinaten wird auf Texturen zugegriffen (Texure Sampling [Heck86]). Eine Textur ist hierbei ein *ein-*, *zwei-* oder *dreidimensionales Array von Daten*, typischerweise RGBA-Werte. Der Präfix „Multi“ signalisiert, dass auf mehr als eine Textur pro Pipeline-Durchlauf zugegriffen werden kann. Diese aus den Texturen ermittelten Farbwerte werden mit den beiden anderen aus der Beleuchtungsrechnung stammenden Farbwerten kombiniert. Ich werde diesen Verarbeitungsschritt in Abschnitt 2.3.1 im Detail beschreiben.

In der nächsten Pipeline-Stufe kann dem berechneten RGBA-Wert auf Wunsch eine Nebelkomponente beigemischt werden. Dieser besteht aus einem Nebelfaktor, der die Intensität des Nebels festlegt, und einer Nebelfarbe [Möll99, S.89-93].

***Alpha, Stencil und Depth Testing:***

Es folgen einige Tests, die eines gemeinsam haben: Nicht-Bestehen führt zum Verwerfen des Fragmentes. Der Alpha-Test überprüft, ob der Alpha-Anteil des RGBA-Wertes eines Fragmentes über einem vorgegebenen Mindestwert liegt. Anwendungen für diesen Test sind in [Möll99, S.123] beschrieben.

Der *Stencil-Test* prüft anhand des Inhaltes des *Stencil Buffers* [Open02], ob das Fragment sichtbar ist oder nicht. Ein Stencil Buffer ist ein spezieller Speicher, mit dem Teile des Bildbereiches maskiert werden können. Mit diesem Mechanismus kann beispielsweise der Schattenwurf eines Objektes simuliert werden [Hain01].

Der Tiefentest (Depth Test) überprüft durch einen Blick in den z-Buffer, ob im Frame Buffer nicht bereits ein Fragment vorhanden ist, welches das gerade

betrachtete Fragment überdeckt. Der z-Buffer ist ein Speicher zur Speicherung der Tiefenwerte von Fragmenten [Catm75].

***Frame Buffer Blending:***

Besteht das Fragment all diese Tests, wird dessen RGBA-Wert mit dem bisherigen Inhalt des Frame Buffers kombiniert (Frame Buffer Blending). Das heißt, entweder überschreibt der berechnete Fragmentwert den vorherigen Inhalt des Frame Buffers, oder es werden beide Werte, gewichtet durch den Alpha-Anteil des RGBA-Wertes, miteinander gemischt.

Der soeben beschriebene Prozess läuft so lange ab, bis die gesamte Bildbeschreibung ihren Weg durch die Pipeline genommen hat.

### 2.1.3 Grafiksysteme

Wie lange die Generierung eines Bildes mit Hilfe des oben geschilderten Prozesses dauert hängt einerseits von der zu verarbeitenden Datenmenge ab, andererseits davon, wie schnell ein Rechner die für den Prozess der Bildgenerierung notwendigen Algorithmen ausführen kann. Um eine CPU bei dieser Aufgabe zu unterstützen wurden spezielle Hardwarekomponenten entwickelt.

#### *2.1.3.1 Bildrechner und GPUs*

Abbildung 9 zeigt die Hauptkomponenten eines Rasterdisplaysystems. In dieser Darstellung ist die *CPU* (Central Processing Unit) für die Definition der Bildbeschreibung verantwortlich, während der *Bildrechner* die gesamte Bildgenerierung, -speicherung und -anzeige auf dem Monitor übernimmt.

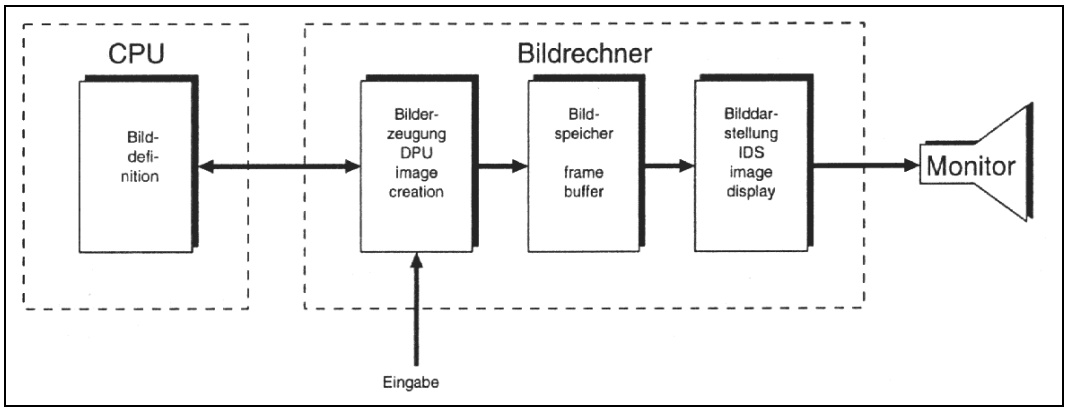



**Abbildung 9: Hauptkomponenten eines Rasterdisplaysystems [Enca97a]**

Der Teil des Bildrechners, der für die Verarbeitung der Vertices verantwortlich ist – dies entspricht den Pipeline-Schritten Transform&Lightning bis Viewport Mapping (vgl. Abbildung 9) – wird *Geometrie-Prozessor* genannt. Den Teil, der für die Fragment-Verarbeitung und das Schreiben in den Frame Buffer verantwortlich ist, nennt man *Display-Prozessor* (DPU).

Durch die Weiterentwicklung dieser Komponenten konnte die Anzahl der in Echtzeit verarbeitbaren Dreiecke und die Qualität der Oberflächenschattierung immer weiter gesteigert werden.

Lange Zeit war solche Grafikhardware, welche den gesamten Prozess der Bildgenerierung unterstützt, nicht für den Massenmarkt erhältlich. Nur Display-Prozessoren waren vollständig implementiert. Beispiele für Grafik-Hardware, die auch die Geometrieverarbeitung beschleunigen, sind die Geometry Engine [Clar82], CHAP [Levi84], Ikonas [Engl86], Indigo Extreme [Harr93], GX4000 [Torb87], Pixel-Planes 5 [Fuch89], DN10000VS [Kirk90], PixelFlow [Moln92], die Reality Engine [Akel93] und InfiniteReality [Mont97].

Für den Massenmarkt erschien zum ersten Mal im Jahre 1999 ein Chip, der einen vollständigen Geometrie-Prozessor sein Eigen nennen konnte (GeForce256). Weitere Chips dieser Machart folgten.

Die in dieser Arbeit behandelten Chips sind die direkten Nachfolger jener Generation von Grafikbeschleunigern, die man zur Unterstreichung ihrer Leistungsfähigkeit in Analogie zur CPU erstmals mit dem Kürzel GPU (Graphics Processing Unit) [NVID02a] versehen hatte.

#### *2.1.3.2 3D-Application Programming Interfaces (3D-APIs)*

Von Rechner zu Rechner variiert die enthaltene Hardware. Dies gilt insbesondere für Grafikhardware in Desktop-Computern. Damit eine Applikation die Vorzüge eines 3D-Hardwarebeschleunigers nutzen kann, muss sie mit seiner Schnittstelle, seinem Treiber, kommunizieren. Die Hardware und somit auch die jeweiligen Treiber unterscheiden sich aber voneinander. Daher muss eine Applikation, die auf verschiedenen Hardwarekonfigurationen laufen soll, auch mit all diesen unterschiedlichen Treibern umgehen können.

Application Programming Interfaces (APIs) wurden dazu entwickelt dieses Problem zu entschärfen. Sie werden als Kommunikationsschicht zwischen Hardware-Treibern und der Applikation geschoben.

Im Falle von 3D-Bildgenerierung spricht man von *3D-Application Programming Interfaces* (3D-APIs). Sie vermitteln zwischen der 3D-Anwendung und der zugrundeliegenden Grafikhardware. Ziel dieser APIs ist es einerseits, Programmierern eine Schnittstelle zur Nutzung der erweiterten Grafikfähigkeiten neuer 3D-Hardware-Beschleuniger zu ermöglichen und andererseits, auch die *Geräteunabhängigkeit* zu sichern. [Enge01, S.3]

Allgemein lässt sich die Funktionsweise einer solchen Grafikschnittstelle wie folgt beschreiben: Die Schnittstelle stellt fest, ob eine bestimmte auszuführende Aufgabe von der Hardware des jeweiligen Rechners unterstützt wird. Ist dies der Fall, sorgt die Schnittstelle durch Kommunikation mit dem Treiber dafür, dass diese Hardware die auszuführende Aufgabe erledigt. Gibt es keine solche Hardware-Unterstützung, dann wird die Aufgabe vollständig auf der CPU ausgeführt.

Bis dato sind die beiden verbreitetsten 3D-APIs zur Entwicklung von Echtzeit-Anwendungen für Desktop-Computer OpenGL [Open02] und DirectX [Micr02a]. DirectX wird in dieser Arbeit eine wichtige Rolle spiele, denn sie ist bisher das einzige API, welches die Möglichkeiten der neuen GPU-Generation weitestgehend auszunutzen weiß. OpenGL wird diesbezüglich vorraussichtlich Ende diesen Jahres mit der Veröffentlichung von OpenGL2.0 nachziehen. Dieser

Schritt ist wichtig, da OpenGL im Gegensatz zu Microsofts DirectX *nicht* ausschließlich für Windows-Systeme geeignet ist.

## 2.2 Überlegungen zur Bewertung von Grafiksystemen

Ich werde in dieser Arbeit eine ausführliche Bewertung über die neue Grafikchip-Generation abgeben. In diesem Abschnitt möchte ich auf die Kriterien und Konzepte eingehen, auf die sich meine Bewertung stützen wird.

Ein grundlegendes Ziel der Forschung im Bereich Real Time Rendering ist es, in der Qualität mit den nicht-zeitkritischen Software-Systemen, wie sie beispielsweise in der Filmproduktion eingesetzt werden, aufzuschließen.
Realtime-Rendering-Hardware soll unter den gegebenen Zeit-Restriktionen eine möglichst hohe Bildqualität ermöglichen. Betrachtet man ein computergeneriertes Bild bzw. eine Animation ganz allgemein als Kommunikationsmedium, dann kann man die Bildqualität gleichsetzen mit dem Grad, in dem es den Bildern gelingt, dem Betrachter genau die Informationen zu vermitteln, zu deren Vermittlung sie geschaffen wurden. Dies mag beispielsweise die Vermittlung eines realistischen Eindrucks von einer räumlichen Umgebung, die visuelle Vermittlung von statistischen Daten oder die Vermittlung der Visionen eines Künstlers sein. Das Gelingen dieses Vorhabens ist einerseits abhängig vom Geschick des Programmierers bzw. Designers und anderseits von den Möglichkeiten, die ihm das Werkzeug „Grafiksystem“ bietet. Ich werde im Folgenden verschiedene Kriterien zur Bewertung von Grafiksystemen vorstellen.

### 2.2.1 Aspekte zur Bewertung der GPU-Funktionalität

#### *2.2.1.1 Detailgrad der Geometrie*

Ein wesentliches Qualitätsmerkmal einer GPU ist die Anzahl der Vertices, die es pro Sekunde verarbeiten kann. Denn je höher diese Rate ist, desto detailliertere geometrische Modelle kann der Programmierer in seiner Applikation einsetzen.

Mit der letzten Grafikchip-Generation hat man in diesem Bereich einen großen Schritt nach vorne gemacht. Denn dadurch, dass die gesamte Geometriepipeline durch Hardware-Komponenten realisiert wurde, konnte die CPU wesentlich entlastet werden. Vorher stand die Anzahl der zu verarbeitenden Vertices im Ressourcenkonflikt mit den anderen Aufgaben der CPU wie der Berechnung künstlicher Intelligenz oder der Kollisionserkennung.

#### *2.2.1.2 Shading*

Der Detailgrad der Geometrie ist aber nur *einer von vielen* qualitätsbildenden Faktoren, und hat er einmal ein gewisses Niveau erreicht, führt eine weitere Verfeinerung kaum zu einer Verbesserung. Dann gleichen die Interpolationsverfahren beim Schattieren (Phong [Phon75] oder Gouraud-Shading [Gour71]) fehlende geometrische Informationen so gut aus, dass der Betrachter die Differenzen beim Erhöhen des geometrischen Feinheitsgrades kaum noch wahrnehmen kann (vgl. Abbildung 10). Und Bildberechnungen, die der Betrachter im Ergebnis nicht wahrnehmen kann, tragen auch nichts zur Informationsvermittlung bei.

**Abbildung 10: Bildgenerierung bei verschieden hoher uniformer Feinheit der Polygonmodelle [Watt00]**

Oben links: *128 Polygone*, oben rechts: *512 Polygone*, unten links: *2048 Polygone*, unten rechts: *8192 Polygone*. Dank des Einsatzes von Interpolationsverfahren (in diesem Fall Phong-Shading) haben die unteren beiden Teekannen eine vergleichbare Qualität, obwohl die rechte aus viermal so vielen Polygonen besteht.

Ein Aspekt, der die Bildqualität stark beeinflusst und mindestens ebenso wichtig ist wie der Detailgrad der Geometrie, ist die *Schattierung der Objekte* (Shading). Hier trennt man in der Regel zwischen zwei Anwendungsgebieten: der *photorealistischen* und *nicht-photorealistischen* Bildgenerierung.

Der erste und um einige Jahre ältere Forschungsbereich beschäftigt sich mit der Simulation in der Natur auftretender Effekte, wie beispielsweise der Interaktion zwischen Licht und Oberflächen. Das Ziel der photorealistischen Bildgenerierung ist es, den Effekt eines traditionellen Photoapparates bzw. einer Filmkamera zu imitieren, d.h. Einzelbilder oder Animationen zu generieren, die der Betrachter nicht mehr von einer fotografierten bzw. gefilmten natürlichen Szene unterscheiden kann [Stro02, S.1].
Ein wesentliches Mittel zur Erreichung dieses Ziels ist die Verwendung von Beleuchtungsmodellen, die sich weitestgehend an physikalischen Gesetzen orientieren [Blin77][Cook82][Kaji85][Cabr87][Hanr93]. Mit diesen Modellen können verschiedenste Materialeigenschaften auf realistische Weise nachgeahmt werden. Um *globale* Beleuchtungseffekte einzufangen – so bezeichnet man Effekte, die auf die gegenseitige Beeinflussung von Objekten untereinander zurückzuführen sind, wie Spiegelungen, Schattenwurf, Lichtbrechung – entwickelte man rekursive Verfahren: Ray Tracing [Kay79] [Whit80] [Kay86] [Watt92, S.219-256] und Radiosity [Gora84] [Cohe85] [Cohe88].

Die nicht-photorealistische Bildgenerierung berücksichtigt darüber hinaus auch Schattierungsverfahren, die nicht die Imitation natürlicher Materialien zum Ziel haben. Von der Imitation künstlerischer Techniken (z.B. Holzschnitt [Kapl00], Zeichnung [Deus98] [Deus00], Malerei [Cohe00] [Curt97]), Drucktechniken [Buch96] bis zu Beleuchtungsmodellen, die für CAD[1]- und CAL[2]- Anwendungen entwickelt wurden (z.B. [Fari96][Gooc98]).

[1] Computer Aided Design

[2] Computer Aided Learning

Eine GPU muss sich also auch daran messen lassen, inwieweit sie die Verwendung von Schattierungsverfahren für all diese unterschiedlichen Anwendungsgebiete in Echtzeit ermöglicht.

#### *2.2.1.3 Animation*

Auch die Fähigkeiten im Bereich Animation sollten ein Kriterium für die Bewertung der Leistungsfähigkeit einer GPU sein. Wie beim Shading, gibt es auch im Bereich der Animation kein ultimatives Verfahren, sondern viele unterschiedlichte Techniken, von der jede für ein bestimmtes Anwendungsgebiet besonders geeignet ist [Watt92, S.337-427]. Auch hier ist daher von Seiten der GPU Flexibilität gefragt.

### 2.2.2 Überlegungen zur Bewertung von GPU-Interfaces

Wie in den letzten Abschnitten beschrieben, besteht ein Grafiksystem aus verschiedenen Hardwarekomponenten sowie verschiedenen Schichten von Softwarekomponenten (Abstraction Layers) – z.B. APIs und Gerätetreiber – die diese Hardware steuern. Die Qualität dieser Steuerungsschnittstellen ist ebenso wichtig wie die Leistungsfähigkeit der Hardware selbst.
Eine Steuerungschnittstelle kann eine sehr dünne und hardware-nahe Schicht sein, aber auch eine sehr abstrakte und auf den Mensch zugeschnittende Oberfläche bieten. Letztere Variante erleichtert dem Programmierer die Arbeit und führt im Allgemeinen zu einer höheren Produktivität und einer höheren Qualität der erstellten Applikationen. Daher ist es auch ein wesentliches Qualitätsmerkmal einer Hardware-Architekur, wie gut sie sich abstrahieren lässt, so dass eine benutzerfreundliche und effektive Steuerung für sie realisiert werden kann.

DirectX und OpenGL sind hardware-nahe Schnittstellen, die bis zum Erscheinen der neuen Chip-Generation eine Steuerung der Hardware im Wesentlichen durch das Setzen von Flags erlaubten. Da die neue Chip-Generation nun aber Programmierbarkeit bietet, ist dieses Verfahren nicht mehr ausreichend. In

Abschnitt 3.3 werde ich besprechen, in welcher Weise man diesem Problem begegnet ist.
Man hat aber auch Anstrengungen unternommen, abstraktere Schnittstellen für die Steuerung von Grafik-Hardware zu entwickeln, die eine benutzerfreundlichere und hardware-unabhängigere Steuerung bieten. Insbesondere versuchte man in den letzten Jahren Interface-Umgebungen, die sich im Bereich Software-Systeme durchgesetzt haben, auf den Echtzeitbereich zu übertragen. Denn hätten Echtzeit-Systeme im Wesentlichen die selbe Interface-Umgebung wie die Software-Systeme, denen man nacheifert, wäre dies bereits ein wesentlicher Schritt zur Verkleinerung der Lücke zwischen beiden Bereichen. Wie ich in Abschnitt 2.3 zeigen werde, waren diese Versuche auf Basis der bisherigen Grafik-Hardware nur in unbefriedigendem Maße erfolgreich. Auch von einiger Bedeutung ist daher die Frage, inwieweit die neue Grafikchip-Generation hierfür bessere Möglichkeiten bietet.

Ein spezielles Konzept, das man seit einigen Jahren auf den Echzeit-Bereich zu übertragen versucht, ist die Shader-Sprache (Shading Language). Eine Shading Language ist eine spezielle Programmiersprache zur Entwicklung von Schattierungsverfahren. Dieses Konzept entstand aus der bereits erwähnten Beobachtung, dass sich die Bildqualität durch flexible und variationsreiche Schattierung enorm steigern läßt. Mit der neuen GPU-Generation versucht man nun zum ersten Mal, dieses Konzept auch für die Echtzeit-Rendering-Systeme durchzusetzen. Inwieweit sie eine solche Abstraktion erlaubt, werde ich später in dieser Arbeit diskutieren.
Um eine Grundlage für die spätere Diskussion zu haben, stelle ich in den nun folgenden Abschnitten zunächst das Konzept der Shader-Sprache und mit dem *RenderMan-Interface*, eine verbreitete Schnittstelle für Software-Systeme vor.

### 2.2.3 Das Konzept „Shader-Sprache“

Shader-Sprachen sind ein Werkzeug zur Entwicklung von Schattierungsverfahren. Von der Kombination verschiedener Texturen bis zur Entwicklung von Beleuchtungsalgorithmen sind alle die Schattierung betreffenden Werkzeuge sind in Shader-Sprachen integriert.

#### *2.2.3.1 Shade Trees*

Entwickelt haben sich Shader-Sprachen aus der Arbeit von Cook [Prou01, Fole89]. Er beschrieb, wie sogenannte *Shade Trees* als ein flexibles und programmierbares Rahmenwerk für Schattierungsberechnungen verwendet werden können [Cook84]. Cooks Ziel war es, die Starrheit existierender Schattierungsmethoden (z.B. Phong-Lightning[Phon75]) aufzubrechen. Sein Ansatz erlaubt die Spezifikation existierender Schattierungsmethoden, bietet zusätzlich aber auch die Möglichkeit, mit neuen Kombinationen zu experimentieren.
Hanrahan und Lawson [Hanr90] heben insbesondere Cooks Entscheidung hervor, die Berechnung der Schattierung in drei konzeptionelle Aufgaben zu unterteilen:

1.) Die Spezifikation der *Lichtquellen* (über sog. *Light Trees*),
2.) Die Beschreibung des *Reflektionsverhaltens* der Oberfläche (über sog. *Shade Trees*)
3.) Die Beschreibung des Einflusses der *Atmosphäre* (über sog. *Atmosphere Trees*)

Cook fasst alle Terme, welche in eine Schattierungsberechnung eingehen, unter dem Begriff *Appearance Paramter* zusammen. Appearance Parameter können nach Cook Geometrie- und Materialeigenschaften, sowie Eigenschaften der Umgebung sein.

Der Zusammenhang zwischen *Shade Trees*, *Light Trees* und *Atmosphere Trees* lässt sich wie folgt beschreiben (vgl. Abbildung 11):

- *Light Trees* spezifizieren die Eigenschaften der Lichtquellen einer Szene.
- Die von den *Light Trees* spezifizierten Lichtquellen-Parameter bilden einen Teil der Basisparameter, auf denen ein *Shade Tree* seine Berechnungen aufbaut.
- Ein *Atmosphere Tree* transformiert die Ausgabe eines Shade Trees. Zielsetzung ist die Berechnung des Effektes, den das zwischen Oberfläche

und Betrachter gelegene Volumen auf das von der Oberfläche reflektierte Licht hat.

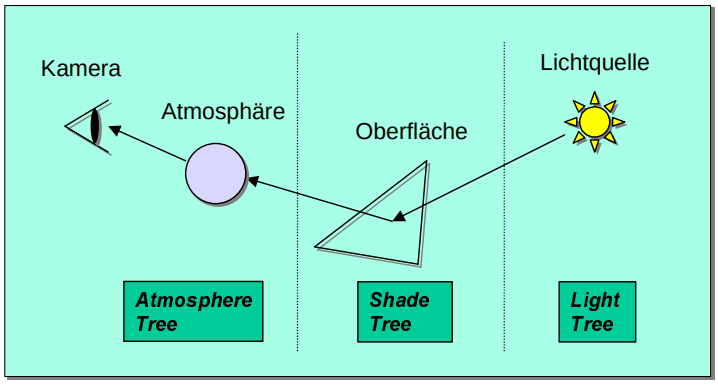



**Abbildung 11: Konzeptioneller Zusammenhang zwischen *Light*, *Shade* und *Atmophere Trees***

Verschiedenen Oberflächen können verschiedene Shade Trees zugewiesen werden, so dass nicht für alle Objekte eines Bildes eine einzige Schattierungsmethode verwendet werden muss. Bei der Bildgenerierung kann, Flächenelement für Flächenelement, eine auf die jeweils zu simulierende Materialeigenschaft zugeschnittene Schattierungs-Methode verwendet werden.

Shade Trees, Light Trees und Atmosphere Trees sind auf identische Weise aufgebaut. Exemplarisch möchte ich den Aufbau eines Shade Trees vorstellen.
Ein Shade Tree ist ein gerichteter azyklischer Graph, dessen Knoten jeweils Appearance Parameter von ihren Kindern erhalten und aus diesen Informationen Appearance Parameter für ihre Eltern produzieren. Jeder Knoten produziert einen oder mehr Parameter als Ausgabe und kann null bis mehrere Parameter als Eingabe verwenden. Ein ausgewiesener Wurzelknoten produziert als Ausgabe das Resultat des Shade Trees.
Die Knoten werden in Post-order-Reihenfolge durchlaufen. Die Blätter des Graphen, also jene Knoten, die keine Eingabeparameter erhalten, erzeugen die Basisparameter, wie beispielsweise Position oder Oberflächennormale [Cook84].
Abbildung 12 zeigt ein Beispiel für die Spezifikation des Phong-Reflektionsmodells [Phon75] mit Hilfe eines Shade Trees.

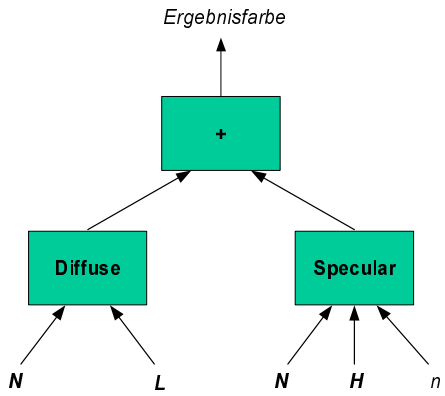



**Abbildung 12: Das Phong-Reflektionsmodell repräsentiert durch einen Shade Tree [Watt92]**

Die Basisparameter sind: ***N*** – Vertex-Normale, ***L*** – Lichtvektor, ***H*** – Half Vector, ***n*** – spekulare Potenz.

Eine Stärke dieses Systems ist es, dass verschiedene Trees auf natürliche Weise miteinander kombiniert werden können. So können beispielsweise die Ausgaben mehrerer Shade Trees als Basisparameter für einen weiteren Shade Tree verwendet werden. Als Anwendungsbeispiel für diesen Mechanismus nennt Cook folgendes Beispiel: Zu simulieren sei eine Oberfläche, die aus einem Konglomerat verschiedener Materialien besteht. Die Schattierung einer solchen Oberfläche kann realisiert werden, indem man zunächst einen Shade Tree für jedes einzelne Material definiert und diese dann zu einem großen Shade Tree zusammenfügt, um so das Zusammenwirken der Materialien zu simulieren.

Zur Beschreibung seiner Graphen schuf Cook eine spezielle Sprache. Einige spezielle Knoten wie *Diffuse*, *Specular*, oder *Square Root* sind bereits Elemente dieser Sprache. Andere Knoten können vom Benutzer definiert und ‚wenn benötigt, dynamisch geladen werden.

#### *2.2.3.2 Pixel Stream Editor*

Perlin erweiterte Cooks Idee von einer flexiblen Bildgenerierungsschnittstelle. Er entwickelte eine Sprache, die eingebettet ist in eine Bildgenerierungsumgebung, dem sogenannten *Pixel Stream Editor* (PSE) [Perl85]. Der PSE operiert auf einem

Pixelarray und berechnet daraus als Ausgabe wiederum ein Array von Pixels – daher auch die Bezeichnung *Pixel Stream* (engl. Pixelstrom). Das vom Programm berechnete Pixelarray kann wiederum als Eingabe für eine weitere Schattierungsprozedur dienen. Den Begriff Pixel verwendet Perlin hierbei recht offen: Ein Pixel kann beliebige Daten enthalten, wie z.B. eine Materialbezeichnung oder eine Oberflächennormale für diesen Punkt. Ein Ausgabepixel muss nicht die selbe Struktur haben wie das Eingabepixel.
Der wesentliche Fortschritt gegenüber Cook war, dass er zur Beschreibung von Schattierungsverfahren eine der Sprache C [Kern78] ähnliche Programmiersprache verwendete. Ein Schattierungsverfahren ist eine in dieser Sprache geschriebene Prozedur, die für jedes Pixel eines Bildes ausgeführt wird (prozedurales Shading).

Dieses System bietet eine andere Sichtweise auf den Rendering-Prozess, als dies bei Cooks Modell der Fall war: In Cooks Modell wird jedes Oberflächenelement genau einmal schattiert. Die Bildgenerierung im PSE ist dagegen ein mehrstufiger Prozess, wobei jede Stufe als Zwischenergebnis ein Bild produziert.

Ein Schwäche dieses Ansatzes ist, dass das Schattieren im großen und ganzen als Prozess betrachtet wird, der nach den Sichtbarkeitsberechnung stattfindet [Hanr90]. Das heißt , nur in der ersten Stufe werden alle vorhergehenden Pipelineschritte (einschließlich der Sichtbarkeitsrechnung) durchlaufen. Das macht es schwierig, globale Beleuchtungsmodelle wie *Radiosity* oder *Ray Tracing* zu implementieren. Denn hierzu sind auch Informationen über Oberflächenfragmente notwendig, die im späteren Bild nicht zu sehen sind. Diese Fragmente werden aber bei der Sichtbarkeitsrechnung aus der Bildbeschreibung entfernt und stehen daher für die darauffolgenden Stufen im PSE nicht zur Verfügung.
Ein Rückschritt zu Cooks Modell ist, dass nicht mehr zwischen der Spezifikation von Lichtquellen, der Spezifikation atmosphärischer Effekte und der Spezifikation des Reflektionsverhaltens unterschieden wird.

#### *2.2.3.3 RenderMan Shading Language*

Die zur Zeit am häufigsten genutzte Shader-Sprache ist *RenderMan Shading Language* von Pixar [Prou01][Pixa02]. Sie ist Teil des *RenderMan-Interfaces* – einem Versuch die Art und Weise, wie Bildbeschreibungen von Modellierungsprogrammen an Rendering-Programme weitergegeben werden, zu standardisieren. Eine detaillierte Beschreibung der Sprache ist in der Spezifikation des RenderMan-Interfaces [Pixa02] zu finden.

Die RenderMan Shading Language, erstmals veröffentlicht 1988, basiert auf den früheren Arbeiten von Cook [Cook84] und Perlin [Perlin85] und versucht Eigenschaften beider Ansätze zu vereinen [Hanr90]. Einerseits wurde die von Cook vollzogene konzeptionelle Trennung von Oberfläche, Umgebung und Lichtquelle weiter vertieft. Andererseits wird Perlins Ansatz vom prozeduralen Shading übernommen. Das heißt, es wird eine flexible Programmiersprache zur Beschreibung von Schattierungsmethoden verwendet. Desweiteren können, wie bei Perlin, neben Oberflächenelementen auch Bildelemente (d.h. Pixels) in mehreren Stufen verarbeitet werden. Hierfür sind sogenannte *Imager Shader* verantwortlich [Pixa00]. Ich werde diese Sprache in den nächsten Abschnitten zusammen mit dem RenderMan-Interface vorstellen.

### 2.2.4 Das RenderMan-Interface

Das RenderMan-Interface [Pixa00] ist eine Sammlung von Prozeduren. *Teil 1* der Spezifikation des RenderMan-Interfaces ist eine Beschreibung dieser Prozeduren. *Teil 2* der Spezifikation ist die *RenderMan Shading Language*, welche die Möglichkeit bietet die vordefinierte Funktionalität des RenderMan-Interfaces zu erweitern.

Im Folgenden beziehe ich mich nicht auf die erstmals veröffentlichte *Version 3.0* von 1988, sondern auf die neueste *Version 3.2* [Pixa00] vom Juli 2000.

#### *2.2.4.1 Architektur des Rahmenwerks*

Das RenderMan-Interface ist zustandsbasiert: Der Programmierer definiert durch das Setzen von Parametern einen Zustand, den sogenannten *Gaphics State*, der

bestimmt, wie ein Oberflächenelement verarbeitet wird [Pixa00]. Der *Graphics State* besteht zwei Teilen: einem *globalen Zustand* (*global state*) und einem *lokalen Zustand* (*current state*).

Der ***globale Zustand*** enthält jene Parameter, die für den gesamten Bildgenerierungsprozess konstant sind. Dies sind beispielsweise Kamera- und Displayparameter[3] sowie die bereits erwähnten *Imager Shader*, mit denen generierte Bilder nachbearbeitet werden können.

Der ***lokale Zustand*** setzt sich aus den Parametern zusammen, die für jedes Oberflächenelement variieren können (Attributes). Die Attributes bestimmen z.B. die Reflektionseigenschaften einer Oberfläche, deren Position im Welt-Koordinatensystem und in die Anzahl und Art der Lichtquellen.

#### *2.2.4.2 Shading in RenderMan*

Eine große Stärke des RenderMan-Interfaces ist meines Erachtens, dass die Schattierung völlig unabhängig von der geometrischen Repräsentation betrachtet wird. Erstellt man im RenderMan-Interface ein Schattierungsverfahren, so gilt dieses für jeden Punkt auf einer Oberfläche, in welcher Repräsentationsform diese auch vorliegen mag.

Es liegt in der Verantwortung des Rendering-Programms, für jeden beliebigen Punkt auf einer Oberfläche die für die Schattierung benötigten Attribute bereitzustellen. Der Programmierer wird mit dieser Aufgabe nicht belastet.

#### *2.2.4.3 Shading Rate*

Die Berechnung der Schattierung wird an einer endlichen Zahl von Punkten auf dem Oberflächenelement durchgeführt. Die Anzahl der Schattierungsberechnungen pro Element wird innerhalb des lokalen Zustandes durch die *Shading Rate* kontrolliert. Die Shading Rate bezieht sich auf die

---

[3] Diese Attribute per Oberflächenelement variabel zu machen widerspräche unserer Vorstellung von der virtuellen Kamera. Es war daher konsequent – RenderMan ist schließlich eine Schnittstelle *für Menschen* – und im Sinne der Benutzerfreundlichkeit diese Werte über die gesamte Bildbeschreibung zu fixieren.

Pixelauflösung des zu genierenden Bildes: Eine Shading Rate von 1 bedeutet zum Beispiel, dass vom Rendering-Programm mindestens eine Kalkulation pro Pixel durchgeführt werden muss, eine Shading Rate von 6, dass für einen Bildbereich von sechs Pixels mindestens eine Kalkulation durchgeführt werden muss. Unabhängig von der Shading Rate wird pro Primitiv aber mindestens eine Berechnung durchgeführt.
Die Mindestanzahl der Schattierungsberechnungen kann also, unabhängig von der geometrischen Repräsentation des zu schattierenden Objektes, beliebig hoch gewählt werden.

#### *2.2.4.4 Shader und Shader-Typen*

Im RenderMan-Interface kann der Programmierer ein Schattierungsverfahren durch die Verwendung spezieller Prozeduren, sogenannter *Shader*, konstruieren.
Fünf Haupttypen dieser Shader werden hierbei unterschieden:

- *Light Source Shader*: Ein Light Source Shader berechnet die Farbe des Lichts, das von einem bestimmten Punkt auf einer Lichtquelle in eine bestimmte Richtung ausgestrahlt wird. Ein Lichtquelle besitzt üblicherweise ein Farbspektrum, eine Intensität, eine direktionale Abhängigkeit und einen Intensitätsabfall mit wachsender Distanz. Eine Lichtquelle kann isoliert existieren oder mit Oberflächenelementen verknüpft sein.

- *Surface Shader*: Jedes Oberflächenelement ist mit einem Surface Shader verknüpft. Er modelliert die optischen Eigenschaften des Materials, aus dem die Oberfläche besteht. Ein Surface Shader berechnet typischerweise das von einem Punkt der Oberfläche in eine bestimmte Richtung reflektierte Licht unter Berücksichtigung der Gesamtheit des einfallenden Lichtes und der Materialeigenschaften an diesem Punkt.

- *Volume Shader*: Dieser Shader-Typ implementiert den Effekt, den ein Volumen auf das es durchquerende Licht hat. Dies kann ein Volumen außerhalb eines Objektes sein (*Atmosphere Shader*, *Exterior Shader*) oder ein

Volumen innerhalb eines Objektes (*Interior Shader*)[4]. Exterior und Interior Shader sind für Rendering Programme geeignet, die Ray Tracing [Kay79] [Whit80] [Kay86] einsetzen – eine Technik, bei welcher der Weg einzelner Lichtstrahlen rekursiv zurückverfolgt wird, um globale Beleuchtungseffekte einzufangen. Atmosphere Shader sind eine Alternative für Rendering-Programme, die nur lokale Beleuchtungseffekte berücksichtigen. Mittels eines Atmosphere Shaders wird der Effekt des Volumens zwischen reflektierender Oberfläche und Betrachter beschrieben. Er berücksichtigt nur Lichtstrahlen, die direkt in Richtung Kamera zielen (sog. *Eye Rays* bzw. *Camera Rays*).

- *Displacement Shader*: Ein Displacement Shader fügt zu einer Oberfläche Unebenheiten hinzu. Er modifiziert die Position und Normale von Oberflächenpunkten[5].

- *Imager Shader*: Dieser Shader-Typ kommt erst zum Einsatz, wenn bereits ein Rasterbild generiert worden ist, und er manipuliert dann die Farben der Pixels. So können zum Beispiel verschiedene Bild-Filterungsverfahren an den Rendering-Prozess angehängt werden (z.B. Farbfilter, Unschärfe und Solarisation).

Eine Sequenz von Shadern ergibt ein Schattierungsmodell, wobei der Weg des Lichtes von der Lichtquelle zum Betrachter durch Hinzu- und Wegnahme von Shadern so komplex oder einfach wie nötig gemacht werden kann.
Hierbei ist zu beachten, dass pro Oberflächenelement maximal je ein Surface Shader, ein Displacement Shader sowie je ein Interior, Exterior und Atmosphere Shader gewählt werden kann [Pixa00]. Pro Primitiv können aber beliebig viele

[4] Per Definiton wird das Volumen, in welches die Oberflächenormale zeigt, als *äußeres* Volumen bezeichnet, das auf der gegenüberliegenden Seite als das innere Volumen [Pixa00].

[5] Der Begriff *Shader* ist in diesem Zusammenhang etwas irritierend, da dieser Shader-Typ nicht Veränderung an der Schattierung eines Objektes, sondern an seiner Geometrie vornimmt. Dies ist ein gutes Beispiel dafür, wie frei die Begriffe *Shader* und *Shading* verwendet werden.

Light Source Shader gleichzeitig aktiv sein– die Anzahl der verschiedenen Lichtquellen in einer Szene ist also nicht beschränkt.
Abbildung 13 zeigt eine Sequenz von fünf Shadern. Die genaue Bedeutung der Ausgabevariablen `Cl`, `Ci` und `Oi` ist Tabelle 2 auf S. 32 zu entnehmen.

### *2.2.4.5 Kommunikation zwischen Shadern*

Die Ausgabevariablen sind entscheidend für die Kommunikation zwischen den Shadern, denn die Ausgabe eines Shaders kann als Eingabe für einen anderen Shader dienen. Diese Vermittlungsaufgabe übernimmt das Rendering-Programm. Dass heisst, ein Shader muss sich nicht um die Benachrichtigung anderer Shader kümmern, falls er Veränderungen vorgenommen hat. Nimmt ein Volume Shader beispielweise Einfluss auf die Farbe des Lichtes, das auf ein Primitiv fällt, dann ist es die Aufgabe des Rendering-Programms dem betreffenden Surface Shader für jeden Aufruf (siehe Abschnitt 2.2.4.3) die korrekten Farben zuzuweisen.

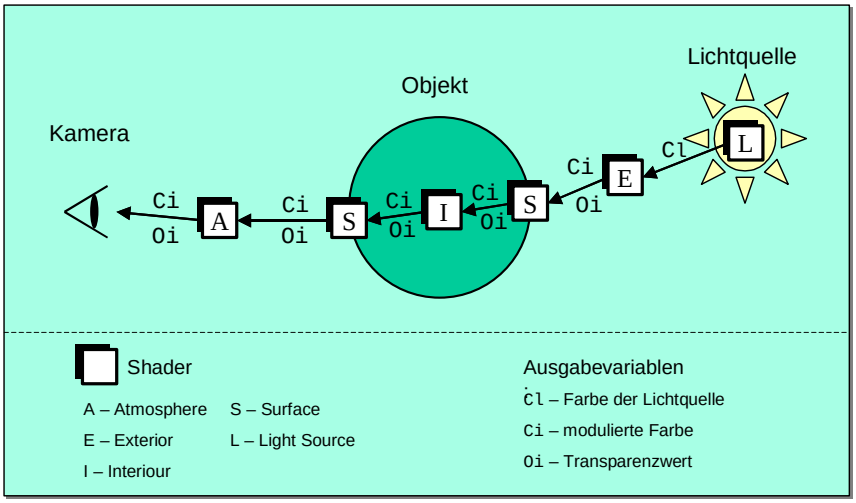



**Abbildung 13: Ein Schattierungsmodell in RenderMan bestehend aus fünf Shadern**

Die Pfeile zeigen den Datenfluss zwischen den einzelnen Komponenten.

### *2.2.4.6 RenderMan Shading Language*

Das RenderMan-Interface verfügt über eine Auswahl von vordefinierten Shadern (siehe Tabelle 3). Der Programmierer hat aber auch die Möglichkeit eigene Shader zu schreiben. Hierzu wurde die anfangs erwähnte RenderMan Shading

Language entwickelt. Sie orientiert sich wie schon Perlins Schattierungssprache stark an der Sprache C [Kern78].

Die Basisdatentypen der Sprache sind: *Float*, *Color*, *Point*, *Vector*, *Normal*, *Matrix*, und *String*. Für jeden dieser Basisdatentypen werden eindimensionale Arrays unterstützt.
Schattierungsberechnungen werden im Allgemeinen an vielen verschiedenen Punkten eines Primitives durchgeführt. In RenderMan wird zwischen Variablen unterschieden, die für alle Punkte eines Primitives konstant sind (uniforme Variablen) und zwischen Variablen, die für jeden Punkt eines Primitives variieren (variierende Variablen). Variablen des ersten Typs erhalten den Bezeichner `uniform`, Variablen des zweiten Typs den Bezeichner `varying`. Dieser Ansatz bietet Optimierungsmöglichkeiten für das Rendering-System: Pixars Referenzimplementierung „*prman*“ berechnet beispielsweise Ausdrücke, die ausschließlich uniforme Variablen enthalten, für jedes Primitiv nur ein einziges Mal. Die Shader teilen sich das Ergebnis dieser Berechnung [Olan98]. Derlei Optimierungsansätze spielen für die Übertragung von prozeduralem Schattieren auf die Echtzeit-Bildgenerierung eine entscheidende Rolle, und ich werde auf diesen Ansatz im entsprechenden Abschnitt (2.3.2) wieder zurückkommen.
Die Sprache unterstützt die üblichen arithmetischen Operatoren (`+`, `-`, `*` und `/`) und die Vektoroperatoren *Kreuzprodukt* (`^`) und *Vektorprodukt* (`.`). Ebenso werden `if`- und `loop`-Anweisungen sowie Funktionsaufrufe unterstützt.
Die Sprache unterscheidet zwei Arten von Prozeduren: *Shader* und *Functions*. Shader sind Prozeduren, die nur vom RenderMan Interface aus aufgerufen werden können. Functions können nur von Shadern und anderen Functions aufgerufen werden.
Ein Shader kann eine Reihe von Instanzvariablen haben. Einige davon können von außen beim Aufruf des Shaders gesetzt werden und diese bilden die Schnittstelle des Shaders. Die Gestaltung dieser Schnittstelle ist Sache des Programmierers. Alle anderen verwendeten Variablen sind innerhalb des Shaders gekapselt und von außen nicht erreichbar (Geheimnisprinzip [Balz98, S.574-576]).

Jeder Shader enthält aber auch eine Reihe vordefinierter Instanzvariablen. Letztere sind automatisch in jedem Shader enthalten und werden vom Rendering-Programm gesetzt. Durch sie wird u.a. die Kommunikation zwischen den Shadern ermöglicht. Welche Variablen vordefiniert sind, ist von Shader-Typ zu Shader-Typ unterschiedlich.
Tabelle 2 zeigt exemplarisch einige der vordefinierten Variablen eines Surface Shaders. Für jeden Shader-Typ sind auch Ausgabevariablen festgelegt, die vom Shader als Ergebnis seiner Berechnungen gesetzt werden sollten (vgl. Tabelle 3).

*Functions* sind sehr ähnlich aufgebaut wie Funktionen in *C*. Sie können jeden der Basistypen als Rückgabewert haben. Für die Eingabeparameter sind zusätzlich auch Arrays erlaubt. Functions sind polymorph. Das heißt, von jeder Function können verschiedene Versionen für verschiedene Parameterlisten existieren, die alle mit dem selben Namen aufgerufen werden können. Rekursive Aufrufe sind nicht erlaubt.

| Name | Datentyp | Beschreibung |
|---|---|---|
| `Cs` | color | Farbe der Oberfläche |
| `Os` | color | Transparenzwert der Oberfläche |
| `P` | point | Position |
| `N` | normal | Shading-Normale |
| `Ng` | normal | geometrische Normale |
| `S, t` | float | Texturkoordinaten |
| `L` | vector | Richtungen eines einfallenden Lichtstrahls* |
| `Cl` | color | Farbe eines einfallenden Lichtstrahls* |
| `Ol` | color | Transparenzwert eines einfallenden Lichtstrahls* |
| `E` | point | Position der Kamera |
| `I` | vector | Richtung des reflektierten Lichtstrahls |
| `Time` | float | Zeitpunkt (des Beginns der Bildgenerierung) |

**Tabelle 2: Eine Auswahl von vordefinierten Surface Shader Variablen [Pixa00]**

* Nur innerhalb von `Illuminance` Statements zugänglich. Das Konstrukt `Illuminace` kontrolliert die Integration des einfallenden Lichtes (Integriert wird über die Fläche einer Halbkugel, zentriert im jeweiligen Oberflächenpunkt.)

| Shader-Typ | erwartete Ausgabe | Datentyp | Beschreibung |
|---|---|---|---|
| Surface | Ci<br>Oi | Color<br>color | modifizierte Farbe<br>modifizierter Transparenzwert |
| *Light Source* | Cl<br>Ol | Color<br>color | ausgestrahlte Farbe<br>ausgestrahlter Transparenzwert |
| *Volume* | Ci<br>Oi | Color<br>color | modifizierte Farbe<br>modifizierter Transparenzwert |
| *Displacement* | P<br>N | Point<br>normal | modifizierte Oberflächenposition<br>modifizierte Normale |
| *Imager* | Ci<br>Oi | Color<br>color | modifizierte Farbe<br>modifizierter Transparenzwert |

**Tabelle 3: Erwartete Ausgabe nach Shader-Typ (RenderMan Interface) [Pixa00]**

Der folgende Quellcode beschreibt einen einfachen Surface-Shader für matte Oberflächen in der RenderMan Shading Language.

Beispiel 1: Ein einfacher Surface-Shader in der *RenderMan Shading Language* [Pixa00].

```
surface
matte(
//Parameter des Shaders
float Ka = 1;         //Intensität ambiente Komponente
float Kd = 0.5);      //Intensität diffuse Komponente
{
// Normale ausrichten, damit sie in das Volumen zeigt,
// in welches das Licht reflektiert wird.
normal Nf = faceforward (normalize(N), I);

// Setzen der Ausgabevariablen Oi und Ci
Oi = Os;
Ci = Os*Cs*(Ka*ambient() + Kd*diffuse(Nf));
}
```

### *2.2.4.7 Diskussion*

Betrachtet man das System der Shader in RenderMan aus softwaretechnischer Sicht zeichnet es sich durch eine geringe *Kopplung* [Somm97, S217-223] zwischen den System-Komponenten aus.

Die Shader sind funktional unabhängig. Zum Beispiel kann ein Surface Shader konstruiert und gewählt werden, ohne die Eigenarten eines Volume Shaders berücksichtigen zu müssen.

Ebenso gering ist die Kopplung zum Gesamtrahmen. Zur Steuerung eines Shaders wird eine Schnittstelle verwendet, eine Liste von ausgesuchten Parametern, deren

Elemente den Instanzvariablen des Shaders entsprechen. Alle anderen Variablen und Ausdrücke, die für die Berechnungen benötigt werden, sind innerhalb des Shaders gekapselt.
Hinzu kommt, dass die Parameter durch den Programmierer beim Aufruf eines Shaders angegeben werden können aber nicht müssen – der Shader initialisiert diese Parameter intern auf Default-Werte. Eine weitere Stärke des Systems ist der hohe Grad an logischer Kohäsion [Somm97, S.217-223] durch die logische Aufteilung in verschiedene Shader-Typen, die jeweils bestimmten Regeln unterworfen sind und von denen jeweils eine bestimmte Ausgabe erwartet wird.
Diese Stärken der Architektur (geringe Kopplung und starke Kohäsion) erzeugen u.a. folgende Vorteile:

- Wiederverwendbarkeit: Einmal geschriebene *Shader* können auf natürliche Weise wiederverwendet werden (dies schafft insbesondere den Rahmen für den Aufbau einer *Shader*-Bibliothek),
- Änderbarkeit: Schattierungsmodelle sind leicht änderbar und erweiterbar. Neue Schattierungsmodelle sind schnell konstruierbar (insbesondere, wenn eine reichhaltige und logisch strukturierte *Shader*-Bibliothek vorhanden ist).

Ich werde auf diese Punkte in der Kritik an der aktuellen Architektur von DirectX (Abschnitt 5.2.1) zurückkommen.

## 2.3 Shading und Shader-Sprachen in Realtime-Rendering-Systemen

Zum Abschluss dieses einführenden Kapitels möchte ich darauf eingehen, wie der Shading auf den bisherigen GPUs realisiert wurde und welche Ansätze verfolgt wurden, um diesen Prozess flexibler zu machen.

### 2.3.1 Multi-Texturing

Zur Zeit ist noch Grafikhardware am verbreitetsten, welche die Steuerung der Bildgenerierung durch das Setzen von Zustandsflags erlaubt. Das heißt, die Rendering Pipeline hat eine Anzahl vordefinierter Konfigurationen, zwischen denen der Programmierer wählen kann. Application Programming Interfaces wie DirectX und OpenGL stellen für diesen Zweck einen Satz spezieller Befehle und Flags zur Verfügung.

Über die Jahre stieg mit der Komplexität der Hardware auch die Komplexität dieser Konstrukte: Neue Befehle und Flags mussten eingeführt werden. Die auf Seite 10 abgebildete Rendering-Pipeline zeigt alle Interface-Komponenten, mit denen GPUs der letzten Generation gesteuert werden.

Für die Steuerung der *Schattierung* sind im Wesentlichen die Komponenten Transform&Lightning und Multi-Texturing zuständig. Wie bereits erwähnt wird im Transform&Lightning-Schritt für jeden Vertex eine Beleuchtungsrechung nach einer Variante von Phongs-Modell [Phon75][Blin77] bestimmt. Der Einfluss des Programmierers beschränkt sich hierbei auf die Definiton einer Liste von Lichtquellen, die in die Beleuchtungsrechnung eingehen, und auf die Festlegung der Materialeigenschaft der zu schattierenden Oberfläche. Diese setzt sich in Phongs Modell aus einer ambienten Farbe, einer spekulare Farbe und einer spekularen Potenz zusammen.

Die Multi-Texturing-Einheit bietet wesentlich mehr Flexibilität: Grafikchips der letzten Generation bieten komplexe Kombinationsverfahren (Texture Combiners seit GeForce256). Zur Steuerung dieser Hardware wird sowohl in OpenGL als auch in DirectX eine Kombinierungs-Pipeline verwendet, die an Cooks *Shade Trees* (s. Abs. 2.2.3.1) erinnert.

Die Pipeline besteht aus mehreren Stufen, sogenannten *Texture Stages* (in DirectX) bzw. *Texture Units* (in OpenGL). Ich verwende den Ausdruck *Texturstufe*. In jeder Texturstufe werden jeweils zwei RGBA-Werte miteinander kombiniert. Die Ausgabe einer Stufe dient als Eingabewert für die nächste Stufe (vgl. Abbildung 14).

Die Eingabewerte stammen, wie bereits in Abschnitt 2.1.2.2 erwähnt, entweder aus Texurzugriffen oder wurden aus den in der Beleuchtungsrechnung ermittelten ambienten und diffusen RGBA-Werten interpoliert.

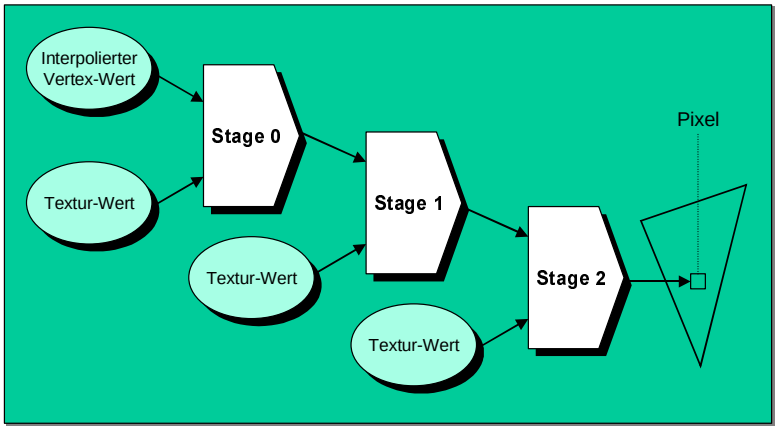



**Abbildung 14: Beispiel für Multi-Texturing [Möll99]**

In *Stage 0* wird das Ergebnis eines Texturzugriffes mit einem interpolierten Wert kombiniert (z.B. diffuse Komponente). Das Ergebnis dieser Kombination wird dann in *Stage 1* und *Stage 2* mit zwei weiteren Texturwerten kombiniert sowie der resultierende Wert dem jeweiligen Pixel zugewiesen.

Der Programmierer kann für jede Stufe zwischen mehreren Operationen wählen, die festlegen auf welche Weise die beiden Eingabewerte miteinander kombiniert werden. Zur Auswahl stehen mathematische Operationen wie Multiplikation oder Addition, aber auch spezielle Operationen wie beispielsweise *Bump Mapping* [Blin78] oder *Alpha Blending* [Möll99, S.123] (die beiden Eingabefarben werden, gewichtet durch ihre Alpha-Werte, miteinander vermischt). Die Anzahl der erlaubten Texturstufen und die zur Verfügung stehenden Operationen variieren von Grafikchip zu Grafikchip. Es folgt ein Beispiel für Multi-Texturing in *DirectX8*.

**Beispiel 2:** Multi-Texturing in DirectX8.

Die Textur einer Mauer (`BrickwallTex`) wird mit einer Schattentextur (`ShadowTex`) kombiniert. In einer Texturstufe darf nur auf genau eine Textur zugegriffen werden, daher sind für diese Operation zwei Stufen notwendig. Die erste Stufe greift auf `BrickwallTex` zu, und die zweite Stufe kombiniert das Ergebnis dieses Zugriffs mit dem Wert aus `ShadowTex`. Der erste Parameter

im Befehl `SetTexture` legt fest, mit welcher Texur eine Stufe verknüpft ist. RGB- und Alpha-Werte werden in jeder Stufe durch separate Operationen kombiniert (`D3DTSS_COLOROP` und `D3DTSS_ALPHAOP`). Die Operation `D3DTSS_DISABLE` in *Texture Stage 2* signalisiert, dass das Ende der *Multi-Texturing*-Pipeline erreicht ist.

```
// Verknüpfe die Textur BrickwallTex mit Texture Stage 0
SetTexture (0, BrickwallTex);
// Verknüpfe die Textur ShadowTex mit Texture Stage 1
SetTexture (1, ShadowTex);

// ----------------- Texture Stage 0 ------------------------
// Wähle BrickwallTex als Quelle für Stage 0
SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);

// Wähle als Ausgabe von Stage 0 das Ergebnis des
// Texturzugriffes auf BrickwallTex
SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);

// ------------------ Texture Stage 1 ----------------------
// Wähle die Ausgabe von Stage 0 als Quelle für Stage 1
SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_CURRENT);
SetTextureStageState(1, D3DTSS_ALPHAARG1, D3DTA_CURRENT);
// Wähle ShadowTex als zweite Quelle für Stage 1
SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_TEXTURE);
SetTextureStageState(1, D3DTSS_ALPHAARG2, D3DTA_TEXTURE);
// Kombiniere die Werte beider Quellen durch komponentenweise
// Multiplikation
SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
SetTextureStageState(1, D3DTSS_ALPHAOP, D3DTOP_MODULATE);

// ------------------ Texture Stage 2 ----------------------
// Signalisiere, dass das Ende der Multi-Texturing-Pipeline
// erreicht ist
SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_DISABLE);
SetTextureStageState(2, D3DTSS_ALPHAOP, D3DTOP_DISABLE);
```

Es ist also schon seit längerer Zeit eine Form von flexiblem Schattieren mit handelsüblichen Grafikkarten und APIs möglich, und die Zahl der mit dieser Technik erzeugbaren Effekte ist nicht zu unterschätzen: Cook [Cook87] beobachtete, dass sich mittels bestimmter Texturen viele globale Effekte simulieren lassen. *Shadow Mapping* oder *Environment Mapping* sind zwei Beispiele für diese Technik.

Unter Verwendung von Texturen als Wertetabellen können einfache Beleuchtungsrechnungen auch per Pixel durchgeführt werden: Eine *Bump Map* erzeugt die Illusion von kleinen Unebenheiten auf der Oberflächen – ein Effekt, der im Ergebnis der Ausgabe eines *Displacement Shaders* (siehe S.29) ähnelt. Es existieren unterschiedliche Varianten dieser Technik, die im Zusammenhang mit *Gouraud Shading* auch *Gouraud Bump Mapping* genannt wird [Erns98]. Desweitern existieren Grafikkarten, die eine Mischung aus Environment Mapping

und einer einfachen Form von Bump Mapping unterstützen (Environment-Mapped Bump Mapping (EMBM) ).
Durch *Gloss Mapping* [Möll99, Kapitel 5.7.3] lassen sich Oberflächen mit matten Stellen versehen: Eine Textur wird verwendet, um pixelweise die spekulare Komponente zu modulieren, so dass einige Stellen weniger Licht reflektieren als andere.

Dennoch ist das Muli-Texturing-System in seiner Flexibilität und Leistungsfähigkeit nicht mit prozeduralem Shading zu vergleichen, wie es die Renderman Shading Language bietet. Im nächsten Abschnitt gehe ich darauf ein, welche Versuche unternommen wurden, um prozedurales Schattieren auch auf Echtzeit-Systemen zu realisieren.

### 2.3.2 Prozedurales Shading – Bisherige Forschung

Die Forschung, die sich mit der Realisierung von prozeduralem Shading in Echtzeitsystemen beschäftigt, ist noch jung. Die Entwicklung der Grafik-Hardware hat gerade erst den Punkt erreicht, da ein solcher Ansatz praktikabel ist [Olan98].
Lastra [Last98] nennt einige Gründe für die Einführung von prozeduralem Shading für Echtzeitsysteme:

- Es ist einfach, Störungen und zufällige Variabilität zu einer Oberfläche hinzuzufügen, um ihr ein realisitischeres Aussehen zu verleihen.
- Es kann einfacher sein, für eine komplizierte Oberfläche einen prozeduralen Shader zu erzeugen, statt die Fehler zu eliminieren, die durch das Umwickeln der Oberfläche mit einer flachen Textur entstehen können.
- Es ist häufig leichter, einen prozeduralen Shader anzupassen, als eine neue Textur zu erzeugen.
- Es ist häufig leichter, Obeflächendetails durch eine prozedurale Schattierungen zu erzeugen, als durch Modifikation der Geometrie.
- Eine prozedural schattierte Oberfläche kann sich mit Faktoren wie Zeit, Entfernung oder Blickwinkel verändern.

#### *2.3.2.1 Multipass Rendering vs. Hardwareprogrammierbarkeit*

Es wurden zwei Ansätze verfolgt, um prozedurales Shading für Echtzeit-Rendering-Systeme zu realisieren:

1. **Flexibilisierung der Grafikhardware durch den Einsatz benutzerprogrammierbarer Prozessoren** [Last95] [Eyles97] [Olan98] [Olan00]: Die Shading-Prozeduren werden direkt auf speziell dafür entwickelten Prozessoren ausgeführt. Innerhalb eines Pipeline-Durchlaufs (Pass) können damit verschiedenste, von Oberflächenpunkt zu Oberflächenpunkt variierende Berechnungen durchgeführt werden.

2. **Multi-Pass-Rendering** [McCo99] [Peer00]: Die Shading-Prozeduren werden auf der CPU ausgeführt und in eine Abfolge von Rendering-Passes übersetzt. Die Grafik-Hardware wird somit als ein CISC-Prozessor aufgefasst: Jede Pipeline-Konfiguration entspricht einer komplexen Instruktion, und ein Pipeline-Durchlauf entspricht der Ausführung dieser Instruktion.

#### *2.3.2.2 OpenGL Shader von SGI*

Der OpenGL Shader [McCo99] von SGI verfolgt letzteren Ansatz. Das System übersetzt Shader-Programme in OpenGL-Anweisungen, die in mehreren Pipeline-Durchläufen (Multipass Rendering) ausgeführt werden. Hierzu wird das Shader-Programm in eine Baumstruktur übertragen, die anschließend dahingehend optimiert wird, dasszur Durchführung möglichst wenig Passes benötigt werden.
Die Shader-Sprache des Systems ist der RenderMan Shading Language sowohl in ihrer Syntax als auch in ihrer Leistungsfähigkeit sehr ähnlich. Allerdings führen bereits einfache Shader-Programme zu unzähligen Rendering-Passes, was einige Nachteilen zur Folge hat.
Erstens besteht mit jedem zusätzlichen Pipeline-Durchlauf die Gefahr des Datenverlustes. Schuld daran ist einerseits die geringe Genauigkeit der Multi-Texturing-Operationen. Denn diese werden normalerweise in Fixpunkt-Arithmetik durchgeführt. Andererseits bestehen auch beim Festhalten von

Zwischenergebnissen Genauigkeitsprobleme, denn der Frame Buffer Probleme speichert Farbwerte im Allgemeinen nur mit 8 Bit pro Kanal.

Zweitens führt Multi-Pass-Rendering zu erheblichen Performance-Verlusten. Denn der OpenGL Shader muss nach jedem Pass Ergebnisdaten aus dem Frame Buffer lesen, um sie für den nächsten Pipeline-Durchlauf verwenden zu können. Dieser Zugriff kostet aufgrund der knappen Bandbreite zwischen GPU und Frame Buffer viel Zeit. Dies gilt insbesondere für PC-Grafikkarten, bei denen sich Frame Buffer und Texturspeicher das Speicherinterface teilen müssen (Unified Memory Architecture).

Aber auch auf einer Octane/MXE von SGI erreicht der OpenGL Shader für relativ einfache Modelle nur Bildgenerieungsraten von 4.6fps bis 12.5fps [McCo99]. Für Echtzeit-Anwendungen ist dieses System somit noch ungeeignet.

**Abbildung 15: Vergleich zwischen einem Software-Rendering-System und dem OpenGL-Shader [SGI02]**

Links: Bildgenerierung mit einem Software-System unter Verwendung der RenderMan Shading Language, rechts: die selbe Szene generiert mit dem OpenGL Shader.

#### *2.3.2.3 PixelFlow Shading System*

Der zweite Ansatz, den man verfolgte, war der Versuch prozedurales Shading direkt auf Hardware-Ebene zu realisieren. Statt auf eine verhältnismäßig inflexible Multi-Texturing-Einheit setzte man auf benutzerprogrammierbare Prozessoren.

Die erste Implementierung, die Shader-Sprachen in Echtzeit durch den Einsatz benutzerprogrammierbarer Prozessoren zu erreichen versuchte, war *Pixel-Planes*

*5* [Fuch89][Rhoa92]. Es erlaubte eine einfache Form von programmierbarem Shading auf Pixel-Ebene. Verwendet wurde eine Low-Level-Sprache mit einfachen Operationen, wie Kopiere, Addiere, Multipliziere und einigen komplexeren Operationen, wie z.B. einer *Perlin-Noise*-Funktion [Perl85]. Die Komplexität der Shader-Programme, die auf der Pixel-Planes-5-Hardware ausgeführt werden konnte war sehr gering, was den praktischen Nutzen des Systems begrenzte [Olan98].

Danach folgten Arbeiten von Molnar, Eyles, Lastra, Olano et al. [Moln92] [Last95] [Eyles97] [Olan98] zu ihrem *PixelFlow Shading System*. Auch dieses System unterstützt prozedurales Shading auf Pixel-Ebene durch die Verwendung von benutzerprogrammierbaren Prozessoren.

Das System setzt sich aus zwei Komponenten zusammen: Dem *Pixel Flow Hardwaresystem*, und der *Pixel Flow Shading Language* (kurz: *pfman*) [Olan98].

Das *Pixel Flow Hardwaresystem* ist eine experimentelle Hardware-Architektur und wurde konstruiert, um zu demonstrieren, dass es möglich ist auf geeigneter Hardware komplexes prozedurales *Shading* in Echtzeit durchzuführen, wie es vorher nur von *Software-Rendering-Systemen* bekannt war [Last95].

Passend zu diesem Konzept wurde als Schnittstelle für die Pixel Flow Hardware mit *pfman* eine der *RenderMan Shading Language* ähnliche Shader-Sprache entworfen. Bis auf wenige Unterschiede ist die Shader-Sprache *pfman* mit ihrem Vorbild, RenderMan, identisch (für eine detaillierte Beschreibung dieser Unterschiede siehe [Olan98], Abschnitt 3).

Mit Hilfe von speziellen Compilern werden in *pfman* geschriebene *Shader* in Programme übersetzt, die auf der *Pixel Flow Hardware* ausgeführt werden können. In [Olan98] sind einige Beispiel-Shader angegeben, die auf diese Weise erzeugt und auf ihrem System bei einer Bildgenierungsrate von mehr als 30fps ausgeführt wurden.

Das Hardwaresystem setzt sich aus mehreren Knoten, den *Pixel Flow Nodes*, zusammen [Last95]. Die Knoten sind durch ein lineares Kommunikationsnetzwerk verbunden, das sich in ein *Geometry Network* und ein *Composition Network* unterteilt [Last95]. Die Pixel Flow Nodes sind im Aufbau identisch (vgl. Abbildung 16), haben aber unterschiedliche Aufgaben: es gibt

*Rendering Nodes*, die für das Rastern der Bildbeschreibung zuständig sind, *Shading Nodes*, die für das Shading zuständig sind, und einen *Frame Buffer Node*, der eine zusätzlich Videokarte enthält, um die Daten an das Ausgabegerät weiterzuleiten.

Ein typisches Pixel Flow System setzt sich aus einem Basissystem (dem sog. Host), einer Anzahl von Rendering Nodes, einer Anzahl von Shading Nodes und einem Frame Buffer Node zusammen [Olan98]. Die Bildgenerierung vollzieht sich abschnittsweise: Der Bildschirm wird in quadratische Regionen von 128x64 Pixels unterteilt, die sukzessive eines nach dem anderen bearbeitet werden. Für jede dieser Regionen ist genau ein *Shading Node* zuständig.

Der Ablauf der Bildgenerierung für eine Bildregion lässt sich verkürzt wie folgt beschreiben: Die Bildbeschreibung wird vom Host geometrisch transformiert und über das *Geometry Network* an die *Rendering Nodes* weitergegeben und von diesen Primitiv für Primitiv in Rasterpunkte zerlegt. Jedem Rasterpunkt der Region werden hierbei die für das *Shading* notwendigen Parameter zugewiesen. Über das *Composition Network* werden diese Informationen an den für die Region zuständigen *Shading Node* weitergegeben. Dieser berechnet daraus die Farbe der Pixels dieser Region und übergibt seine Ergebnisse zur Darstellung an den *Frame Buffer Node*.

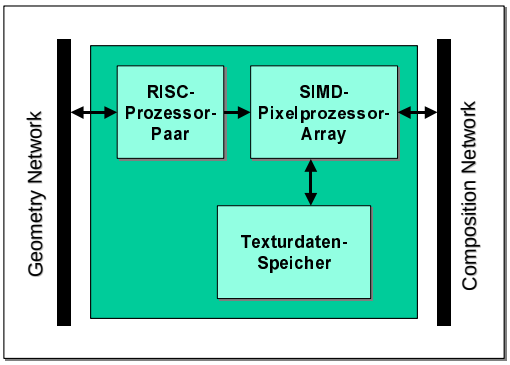



**Abbildung 16: Einfaches Blockdiagramm eines *Pixel Flow Node* [Olan98][Lastra95]**

Ein *Pixel Flow Node* besteht aus drei Hauptkomponenten (vgl Abbildung 16): einem Paar RISC-Grafikprozessoren, einem Array von 128x64 SIMD-Pixel-Prozessoren und einem Speicher für Texturdaten [Olan98].

Die Pixelprozessoren arbeiten parallel, so dass bis zu 128x64 (= 8192) Pixels gleichzeitig verarbeitet werden können. Jeder Pixelprozessor enthält einen lokalen Speicher (256 Byte) und eine 8-Bit-ALU (Arithmetic Logical Unit), die einen Standardsatz von Integer-Instruktionen ausführen kann wie Additionen, Subtraktionen, Multiplikationen und Shifts. Der Texturdatenspeicher kann von den Pixelprozessoren gelesen und beschrieben werden und dient daher als globaler Speicher [Last95].
Zur Steuerung der Pixelprozessoren dienen die beiden RISC-Prozessoren. Sie generieren die Instruktionen für den SIMD-Array. Die RISC-Prozessoren werden wiederum durch eine spezielle Low-Level-Sprache (IGCStream) gesteuert, die auf ihnen ausgeführt wird. Die beiden RISC-Prozessoren verfügen über einen gemeinsamen lokalen Speicher.
Zusammenfassend lässt sich das Programmmodell des Pixel Flow Systems wie folgt beschreiben: Die *High-Level*-Shader-Sprache *pfman*, in welcher der Benutzer seine *Shader* schreibt, wird durch Compiler in mehreren Schritten in Code umgewandelt, der dann auf den RISC-Prozessoren der einzelnen *Pixel Flow Nodes* ausgeführt wird und auf diese Weise die Arbeit des SIMD-Arrays steuert.

Eine Besonderheit des Systems ist die Trennung von Rasterisierungs- und Schattierungsberechnung. Das Rastern und damit auch die Sichtbarkeitsrechnung finden vollständig vor dem Schattieren statt [Olan98] (eine Technik, die auch *Deferred Shading* genannt wird [Whitt81][Deer88]). Das hat den Vorteil, dass beim *Shading* nur Bildpunkte bearbeitet werden, die am Ende auch zu sehen sein werden. So werden Schattierungsberechnungen an Oberflächen, die von anderen verdeckt sind, eingespart. Dieses System hat aber auch einen entscheidenden Nachteil: Die Berücksichtung von Transparenz als Materialeigenschaft ist schwierig. Alle Informationen über eine Oberfläche, die sich hinter einer teilweise transparenten Oberfläche befindet, werden vor dem *Shading* aussortiert.
Neben *Deferred Shading* wurde das System durch die Unterscheidung zwischen uniformen und variierenden Datentypen optimiert (vgl. Abschnitt 2.2.4.6). Uniforme Ausdrücke werden von den RISC-Prozessoren *einmal* berechnet und in deren lokalem Speicher festgehalten. Variierende Ausdrücke können sich von

Pixel zu Pixel unterscheiden und werden daher von den parallel arbeitenden Pixelprozessoren berechnet.
Es wurden noch weitere Optimierungen am System vorgenommen. Die genauen Details aller Optimierungen finden sich in [Olan98].

# 3 Studie der GPU-Architekturen

Nachdem ich nun einen Überblick über das Themengebiet Realtime-Rendering und den bisherigen Stand der Technik gegeben, die Zielrichtung und den Stand aktueller Forschung beschrieben, sowie meine Bewertungskriterien geklärt habe, ist es nun an der Zeit das Objekt der Diskussion vorzustellen: Die neue Grafikchip-Generation.

Es sind inzwischen mehrere verschiedene Vetreter dieser Generation auf dem Massenmarkt erschienen, und ich habe eine Studie über all die verschiedenen Architekturen durchgeführt. Basierend auf dieser Studie, möchte ich in diesem Abschnitt einen Überblick über den Aufbau der Chips und deren Einbindung in Application Programming Interfaces geben.

## 3.1 Die Hardware-Architektur im Überblick

Die NVIDIA Corporation machte den ersten Schritt in Richtung programmierbarer Grafikhardware für den Massenmarkt. Im letzten Jahr stellten sie den Grafik-Chip GeForce3 vor, der über eine benutzerprogrammierbare Architektur verfügt – von NVIDIA *NFiniteFX Engine* getauft.

NVIDIA arbeitete bei der Entwicklung der GeForce3 eng mit der Microsoft Corporation zusammen, so dass fast zeitgleich mit dem Grafikchip eine neue DirectX-Version, DirectX8, erschien, die spezielle Schnittstellen zur Nutzung der neuen Hardware enthält. Um auch eine Nutzung unter OpenGL zu ermöglichen, veröffentlichte NVIDIA einige OpenGL-Extensions [NVID02b].

In den Monaten darauf erschienenen Konkurrenzprodukte der Hersteller ATI [ATI02a] und MATROX [MATR02], die einen leicht verbesserten Aufbau hatten, und auch NVIDIA veröffentlichte mit der *GeForce4 Ti* einen weiteren benutzerprogrammierbaren Chip.

Tabelle 4 listet alle mir bekannten bis dato erschienenen benutzerprogrammierbaren GPUs auf.

| Hersteller | Produkte* |
|---|---|
| NVIDIA | GeForce3,<br>GeForce4 Ti |
| ATI | Radeon8500,<br>Radeon9000,<br>Radeon9700 |
| MATROX | Parthelia-512 |

**Tabelle 4: Veröffentlichte benutzerprogrammierbare GPUs**

* ohne Anspruch auf Vollständigkeit

Alle diese Chips sind sich in ihrem architektonischen Aufbau sehr ähnlich. Dennoch gibt es kein einheitliches Bezeichnungsschema für die einzelnen Komponenten. In den meisten Fällen geben die konkurrierenden Hersteller NVIDIA, ATI und MATROX allen Bauteilen und Techniken völlig unterschiedliche Namen, auch wenn Unterschiede in der Funktionalität marginal oder überhaupt nicht vorhanden sind. Daher habe ich mich dazu entschlossen ein eigenes Bezeichnungsschema zur Beschreibung des allgemeinen Aufbaus der neuen GPU-Generation zu verwenden.

### 3.1.1 Programmierbarkeit auf zwei Ebenen

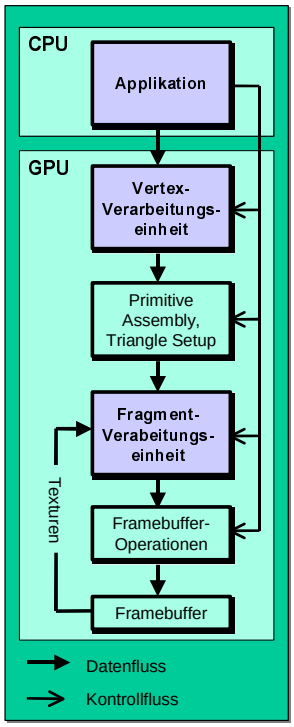



**Abbildung 17: Konzeptionelles Modell des Bildgenerierungsprozesses mit der neuen benutzerprogrammierbaren GPU-Generation**

Die Grafikarten der neuen Generation unterstützen Programmierbarkeit auf zwei Ebenen: auf *Vertex-Ebene* und auf *Pixel-Ebene*. In der Hardwarearchitektur wird dies ermöglicht durch zwei Komponenten (vgl. Abbildung 17): einer programmierbaren *Vertex-Verarbeitungseinheit* und einer programmierbaren *Fragment-Verarbeitungseinheit*.

Die Programmierbarkeit beschränkt sich also nur auf ausgewählte Teile der Rendering-Pipeline. Aufgaben wie

*Triangle Setup*, *Clipping* oder *Sichtbarkeitstests* sind davon ausgeschlossen [Nvid02b]. Diese Entscheidung wurde getroffen, weil man keinen Vorteil in deren Programmierbarkeit sah, und, weil man das Programmmodell so einfach wie möglich halten wollte [Lind01].

> **Definition:** In Analogie zu den Kürzeln CPU (*Central Processing Unit*) und GPU (*Graphic Processing Unit*) werde ich die beiden Einheiten im Folgenden mit **VPU** (*Vertex Processing Unit*) und **FPU** (*Fragment Processing Unit*) abkürzen.

Die Verarbeitungsvorgänge beider Einheiten werden im Wesentlichen durch drei Faktoren beeinflusst:

1. Durch die Attribute, die zusammen mit jedem Datenelement – Vertex oder Fragment – gespeichert sind (variierende Attribute).
2. Durch Werte und Attribute, die für jeden Frame direkt von der Applikation gesetzt werden und die für jedes zu verarbeitende Datenelement konstant sind (uniforme Werte/Attribute).
3. Durch das Verarbeitungs-Programm, das der Programmierer auf der jeweiligen Einheit ablegt (VPU- oder FPU-Programm).

### 3.1.2 Von der Bildbeschreibung zum Bild

Konzeptionell läuft die Bildgenerierung auf einem solchen Chip wie folgt ab:
Die von der Applikation erzeugte Bildbeschreibung erreicht die VPU als Strom von Vertices. Jeder Vertex dieses Stroms durchläuft, einer nach dem anderen, die VPU und wird von dieser verarbeitet. Sie führt hierzu ein benutzerdefiniertes Verarbeitungsprogramm durch, das aus den uniformen und variierenden Eingabeattributen eine Reihe von variierenden Ausgabeattributen berechnet. Diese gibt sie an die folgenden Verarbeitungsschritte weiter.
Die Hardware gruppiert die verarbeiteten Vertices zu Dreiecken (*Primitive Assembly*). Dann folgen die Schritte *Clipping*, *Culling* und *Viewport Mapping* und schließlich das *Triangle Setup*, welches die Dreiecke in Fragmente zerlegt.

Während des *Triangle Setups* werden jedem Fragment Attribute (Farbwerte, Transparenzwerte, Texturkoordinaten, etc.) zugewiesen. Diese werden aus den Attributen der drei Vertices, die zum Dreieck des Fragments gehören, interpoliert.
Jedes Fragment durchläuft nun, eines nach dem anderen, die FPU und bekommt von dieser, basierend auf seinen Attributen, einen RGBA-Wert zugewiesen. Auch hier wird die Verarbeitung in weiten Teilen durch ein benutzerdefiniertes Programm geregelt.
Der daraus resultierende Fragmentstrom wird in den darauffolgenden Schritten weiterverarbeitet (verschiedene Sichtbarkeitstests werden durchgeführt) und mit dem Inhalt des Frame Buffers kombiniert.

### 3.1.3 Merkmale der neuen Architektur

#### *3.1.3.1 Keine Erzeugung von Daten*

Die VPU und FPU operieren auf den Elementen eines Datenstroms, der von anderen Komponenten, z.B. von der Applikation oder vom Triangle Setup, erzeugt wurden. Sie selbst können keine Datenelemente – Vertices oder Fragmente – erzeugen.

#### *3.1.3.2 Keine Kontrollstrukturen*

Eine nicht unwesentliche Einschränkung der VPU- und FPU-Programmmodelle ist das Fehlen jeglicher Kontrollkonstrukte. Ein VPU-Programm wird grundsätzlich vollständig und sequentiell abgearbeitet. Schleifen, Sprünge oder vorzeitige Programmabbrüche sind nicht möglich.
Lediglich die FPU hat die Möglichkeit, Fragmente nach bestimmten Kriterien vor der Verarbeitung auszusortieren und zu verwerfen. Durch diesen Mechanismus können benutzerdefinierte Clipping Planes realisiert werden. Für die VPU existiert kein vergleichbarer Mechanismus.

#### *3.1.3.3 Die isolierte Verarbeitung*

Die Vearbeitung eines Vertices oder Fragmentes ist ein rein lokaler Prozess. Das heißt, jedes Element wird unabhängig von den anderen verarbeitet. Es spielt für das Ergebnis der Verarbeitung keine Rolle in welcher Reihenfolge oder in welcher Form (ob sequentiell oder parallel) die Datenelemente die jeweilige Einheit durchlaufen.

#### *3.1.3.4 Die VPU als Vorprozessor der FPU*

Eine weitere wesentliche Beobachtung ist die Abhängigkeit zwischen VPU und FPU. Die VPU berechnet die Attribute der Vertices. Da die Attribute der Fragmente aus den Attributen der Vertices interpoliert werden, bestimmt die VPU auf indirektem Wege die Eingabewerte für die FPU.
Der Programmierer hat daher die Möglichkeit Werte in der VPU zu berechnen und diese als Ausgabeattribute weiterzugeben, um sie später in der FPU (als interpolierte Fragment-Attribute) zu verwenden.
Dieser Mechanismus kann beispielsweise dazu verwendet werden, um Phong-Shading [Phon75] zu realisieren: Die VPU gibt als eines der Ausgabeattribute die Vertex-Normale weiter. Beim *Triangle Setup* wird dann jedem erzeugten Fragment eine entsprechende interpolierte (nicht normalisierte) Fragment-„Normale“ zugewiesen. Die FPU kann diesen Vektor dann für eine Beleuchtungsrechnung auf Fragment-Ebene verwenden.

#### *3.1.3.5 Begrenzte Programmlänge*

Die Längen der VPU- und FPU-Programme sind begrenzt, und es gibt keine Möglichkeit innerhalb eines Pipeline-Durchlaufes ein zweites VPU und FPU-Programm zu starten.

#### *3.1.3.6 Früher z-Sichtbarkeitstest*

Ein Architekturmerkmal, das sich bei allen Chips der neuen Generation findet, ist der *frühe z-Sichtbarkeitstest* (*z-Occlusion Culling*).

In traditionellen Architekturen wird erst nach der Berechnung der Fragmentfarbe und beim Schreiben in den Frame Buffer ein z-Test durchgeführt. In den neuen Architekturen wird nach dem Triangle Setup und bevor ein Fragment die FPU durchläuft bereits im Groben ein solcher Test durchgeführt, um verdeckte Fragmente frühzeitig auszusortieren und so die FPU zu entlasten.
Es existieren zahlreiche Algorithmen für solche vorgezogenen Sichtbarkeitstests (siehe [Möll00], Kap. 7). Wie genau die betrachteten GPUs diesen Test realisiert haben kann ich nicht sagen, da die Hersteller nur wenige Informationen über den tatsächlichen Aufbau ihrer Chips preisgeben. Grundsätzlich verfolgen beide Hersteller aber wohl den selben Ansatz: *den hierarchischen z-Buffer-Test* [Tom02].
Hierbei wird der z-Buffer in Blöcke aufgeteilt und für jeden Block der größte z-Wert gespeichert (also der Wert des Pixels, der am weitesten im Hintergrund liegt). Beim Erzeugen eines Fragmentes überprüft das Triangle Setup den Wert des z-Buffer-Blocks, der den Bereich abdeckt, in dem das Fragment liegt. Falls dieser Wert kleiner ist, als der Tiefenwert des Fragments, wird es von allen anderen Fragmenten des Bereiches überdeckt, und es wird verworfen.

## 3.2 Analyse der VPU und FPU

Nachdem ich nun den grundlegenden Aufbau der neuen Grafikarten-Generation beschrieben habe, möchte ich ein wenig näher auf die genaue Architektur von VPU und FPU eingehen.
Am ausführlichsten beschreibe ich den GeForce3-Chip von NVIDIA, da dies der erste Vertreter der neuen Generation war, und, weil es der Chip war, der mir für Leistungstests zur Verfügung stand (Kapitel 4). Außerdem weichen seine Konkurrenten nur in einigen Details von seinem Aufbau ab. Diese etwas später erschienenen Chips - *GeForce4 Ti*, *Radeon 8500*, *Radeon 9000* und den *XBox-Chip* - stelle ich im Anschluss kurz vor und gehe auf einige der angesprochenen Abweichungen ein.

### 3.2.1 NVIDIAs GeForce3

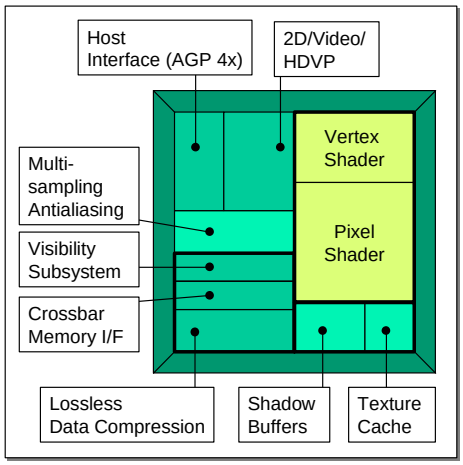



**Abbildung 18: Der Aufbau der GeForce3-GPU [Nvid02a]**

Tabelle 5 zählt einige Eckdaten der GeForce3-GPU auf. Revolutionär ist die Anzahl der Transistoren, aus denen der Chip besteht. Mit 57 Millionen sind das nicht nur doppelt so viel wie beim GeForce256-Chip, sondern auch 20 Prozent mehr als bei Intels neuesten Sprössling Pentium 4 [Tom02].

Um zu vermeiden, dass die hohe Transistoranzahl einen sehr großen Siliziumkern benötigt, wird der GeForce3 im 0.15-Micron-Prozess gefertigt. Durch den 0.15-Micron-Prozess gewährleistet NVIDIA auch, dass der Stromverbrauch gegenüber dem GeForce256 nicht erheblich höher ausfällt [Tom02].

| | |
|---|---|
| Prozesstechnologie: | 0.15 Mikron |
| Transistoranzahl: | 57 Millionen |
| Anzahl der Vertex-Pipelines: | 1 |
| Anzahl der Pixel-Pipelines: | 4 |
| Max Anzahl Texturen pro Pass: | 4 |
| Max. Anzahl FPU-Insruktionen pro Pass: | 12 insgesamt, davon:<br>4 Texture-Sampling-Instruktionen,<br>8 arithmetische Instruktionen |
| Max. Anzahl VPU-Instruktionen pro Pass: | 128 |
| Chiptakt: | 200 MHz * |
| Speichertakt: | 230 MHz DDR * |
| Speichergröße: | 64 MByte * |
| Unterstützte Speicherarten: | SDR oder DDR |
| Anzahl der Speicheradressleitungen: | 128 Bit |

**Tabelle 5: Die wichtigsten Eckdaten der GeForce3 im Überblick [Nvid02a]**

* bezieht sich auf die der GeForce-3-Referenzkarte

Die Komponenten des Chips wurden sowohl räumlich, als auch konzeptionell in Obergruppen zusammengefasst und mit werbeträchtigen Namen versehen (vgl. Abbildung 18). Die GPU und FPU des GeForce3-Chips fasst NVIDIA mit dem Ausdruck „nfiniteFX Engine“ zusammen (Abbildung 18, gelb markierter Bereich) [Nvid01a]. Auf die meisten anderen in Abbildung 18 aufgeführten Techniken werde ich im Rahmen dieser Arbeit überhaupt nicht oder nur am Rande eingehen, da sie keinen direkten Bezug zum Thema haben. Für genauere Informationen zu diesen Techniken verweise ich auf [Nvid02a] und [Tom02]. Der *frühe z-Sichtbarkeitstest* ist Teil des „Visibility Subsystems“ und trägt bei NVIDIA den Namen „z-Occlusion Culling“.

### 3.2.2 Die GeForce3-VPU

Die VPU, von NVIDIA *Vertex Shader* getauft, ist die erste Stufe der "NFiniteFX Engine". Diese beansprucht den Großteil der 32 Millionen Transistoren, die dem GeForce3 gegenüber dem GeForce256 hinzugefügt wurden [Tom02]. Sie besteht aus einer benutzerprogrammierbaren Vertex Engine (VE) (vgl. Abbildung 16).

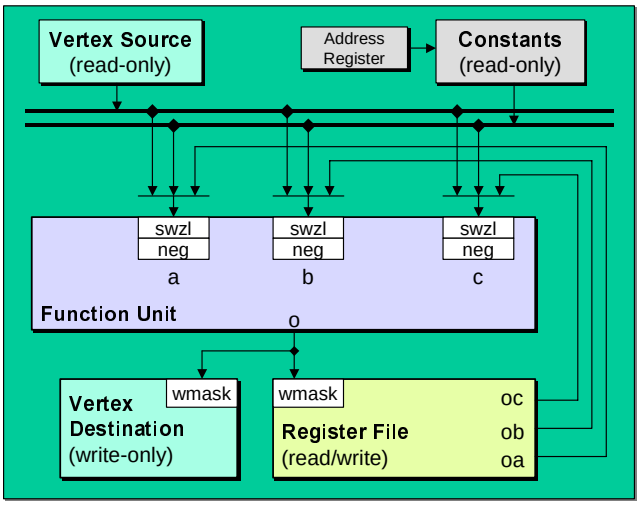



**Abbildung 19: Programmmodell der Vertex Engine [Lind01]**

wmask = *write mask*, swzl = *swizzle*, neg = *negation*.

Der Basisdatentyp des Modells ist ein 4-dimensionaler Vektor, bestehend aus Zahlen vom Typ *IEEE Single Precision Floating Point* [Lind01]. Jeder in der Abbildung eingezeichnete Pfeil repräsentiert einen solchen vierdimensionalen Vektor.

> **Definition:** Ich werde den Basisdatentyp der VPU, {( *x*, *y*, *z*, *w* ) | *x*, *y*, *z*, *w* sind vom Typ „IEEE Single Precision Floating Point"}, im Folgenden wie Lindholm et al. in [Lind01] mit „**Quad-Float**" bezeichnen.

Die Vertex Engine operiert immer nur auf einem einzelnen Vertex und berechnet für diesen aus einer Menge von Eingabe-Attributen (*Vertex Source*, *Constants*) eine Menge von Ausgabe-Attributen (*Vertex Destination*).
Zur Speicherung von Zwischenergebnissen stehen dem Programmierer zwölf *temporäre* Quad-Float-Register zur Verfügung (*Register File*) [Lind01].
Es gibt außerdem ein einzelnes Adressregister `A0`, das mit einer speziellen Instruktion aus dem Instruktionssatz geladen werden kann. Dieses Register kann der Programmierer zum indirekten Lesezugriff auf *Constants-Register* verwenden.

#### *3.2.2.1 Eingabeattribute*

Die Eingabe-Attribute sind in speziellen Read-Only-Registern abgelegt, die vor Beginn der Verarbeitung eines jeden Vertex geladen werden. Sie bilden die Eingangsschnittstelle der Vertex Engine.

Wie bereits erwähnt unterteilen sich die Eingabe-Attribute in zwei Gruppen:

1. in Attribute, die mit dem zu verarbeitenden Vertex verknüpft sind und daher für jeden Vertex variieren können (Abbildung 19: „Vertex Source") und
2. in Attribute, die mit dem gesamten zu verarbeitenden Vertex-Strom verknüpft sind und daher für alle Vertices des Stroms konstant sind (Abbildung 19: „Constants").

**Definition:** In Analogie zur *RenderMan Shading Language* , werde ich im Folgenden die Attribute des ersten Typs (Abbildung Abbildung 19: „Vertex Source“) „**variierend**“ und die des zweiten Typs (Abbildung 19: „Constants“) „**uniform**“ nennen.

Es gibt 16 Quad-Float-Register für variierende Attribute und 96 Quad-Float-Register für uniforme Attribute. Der Programmierer kann diese Register von der Applikation aus mit beliebigen Werten füllen; die Register haben keine vorgeschriebene Bedeutungszuweisung. Tabelle 6 listet einige typische Inhalte für beide Registertypen auf.

Undefinierte Registerkomponenten sind im Fall der ersten drei Komponenten mit 0.0 und im Fall der letzten Komponente mit 1.0 initialisiert. Der Inhalt eines Eingaberegisters bleibt solange erhalten, bis der Host es mit neuen Daten überschreibt. Die Vertex Engine beginnt mit der Arbeit, sobald der Host in das Vertex-Source-Register mit Index 0 schreibt.

| Attributtyp | max. Anzahl | Typische Inhalte |
|---|---|---|
| Variierend | 16 | Position |
| | | Normale |
| | | Materialeigenschaften (diffuse Farbe spekulare Farbe) |
| | | Texturkoordinaten |
| | | Gewicht (für Skeletal Animation) |
| | | Nebelfaktor |
| | | Punktgröße (für Hardware-Sprites) |
| Uniform | 96 | Lichtquellenattribute (Position, Farbe, Abschwächung, etc.) |
| | | Matrizen für geometrische Transformationen |
| | | Matrizen für *Skelett-Animation* |
| | | Zeit |

**Tabelle 6: Typische Eingabeattribute für die Berechungen einer Geforce3-VPU**

### *3.2.2.2 Ausgabeattribute*

Aufgabe der Vertex Engine ist die Berechnung der Ausgabeattribute. Hat sie ein Ausgabeattribut bestimmt, legt sie es in einem der 13 Ausgabe-Register ab (Abbildung 19: „Vertex Destination“).

Diese Register bilden die Ausgangs-Schnittstelle der Vertex Engine. Nach der Abarbeitung der Programmanweisungen werden die hier gespeicherten Werte an die folgenden Pipeline-Schritte weitergegeben.
Die Ausgaberegister dürfen ausschließlich beschrieben werden. D.h. weder das VPU-Programm noch der Host darf auf die Inhalte der Register zugreifen. Wie ich in Kapitel 5 zeigen werde, zieht dieser Sachverhalt einige negative Konsequenzen bezüglich der Nutzbarkeit der Hardware nach sich.

| Registertypen | Anzahl | Inhalte |
|---|---|---|
| HPOS | 1 Quad-Float | Homogene Clip-Space-Position |
| COL | 2 Quad-Float | COL0: häufig diffuse Farbe<br>COL1: häufig spekulare Farbe |
| TEX | 8 Quad-Float | TEX0-TEX7: Texturkoordinaten 0 bis 7 |
| FOGP | 1 Float | Nebelfaktor |
| PSIZ | 1 Float | Punktgröße |

**Tabelle 7:** **Ausgaberegister einer Geforce3-Vertex-Engine [Lind01]**

Im Falle der Register FOGP und PSIZ wird nur die jeweils erste Komponente als Ausgangsschnittstelle verwendet - daher nur 1 Float pro Register.

### *3.2.2.3 Der Weg der VPU-Ausgabeattribute*

Wie aus Tabelle 7 zu entnehmen hat anders als bei den Eingabeattribute jedes Register eine *festgelegte Bedeutung*. Diese feste Bedeutungszuweisung ist im Falle der Aussgaberegister notwendig, da die darin gespeicherten Attribute in den folgenden, festverdrahteten Pipeline-Schritten für vorbestimmte Aufgaben benutzt werden:
Falls der Programmierer Hardware-Sprites verwendet, bestimmt der Inhalt von PSIZ deren Größe. Der Wert in HPOS wird für *Clipping*, *Culling* und *Viewport Mapping* benötigt und wird beim Triangle Setup unter anderem dazu verwendet, jedem Fragment seinen Tiefenwert zuzuweisen.

**Definition:** Aus den Ausgabeattributen TEX0 - TEX7, COL0, COL1 und FOG werden während des *Triangle Setups* für jedes Fragment *F* Fragment-Attribute interpoliert, welche der FPU als Eingabe dienen.
□□□□$_F$ - □□□□$_F$, □□□□$_F$, □□□□$_F$ und □□□□$_F$ seien die entsprechenden interpolierten Fragmentattribute.

COL0$_F$ und COL1$_F$ können in der FPU in verschiedenen arithmetischen Operation zum Einsatz kommen (siehe Tabelle 25). Die Werte TEX0$_F$ - TEX7$_F$ verwendet die FPU i.d.R. zum *Texture Sampling* (vgl. Tabelle 18-Tabelle 20) und Nebelfaktor eines Fragmentes FOGP$_F$ benutzt sie zusammen mit der uniformen Nebelfarbe zum *Fog Blending* (siehe S.13).

### *3.2.2.4 Der Instruktionssatz der GeForce3-VPU*

Die Vertex Engine der GeForce3-VPU verfügt über einen Satz von 17 Instruktionen (vgl. Tabelle 8). Dies sind größtenteils einfache Basisoperationen, aber auch drei spezielle Operationen sind darin enthalten: LIT, DST und ARL.
LIT ist zur beschleunigten Berechnung der spekularen Komponente nach dem Phong-Modell geeignet. DST dient zur Berechnung des Abschwächungsfaktors von Punktlichtquellen, und mit ARL lässt sich unter Verwendung des Adressregisters der Inhalt eines uniformen Registers laden.

| Instruktion | Voller Name | Instruktion | Voller Name |
|---|---|---|---|
| MOV | Move | RCP | Reciprocal |
| MUL | Multiply | RSQ | Reciprocal square root |
| ADD | Add | DP3 | 3 term dor product |
| MAD | Multiply and add | DP4 | 4 term dot product |
| DST | Distance | LOG | Log base 2 |
| MIN | Minimum | EXP | Exp base 2 |
| MAX | Maximum | LIT | Phong lightning |
| SLT | Set on less than | ARL | Load adress register |
| SGE | Set on greater or equal | -- | -- |

**Tabelle 8: Der Instruktionssatz der Geforce3-Vertex-Engine [Lind01]**

Eine Instruktion kann bis zu drei Quad-Floats als Eingabe haben und produziert einen Quad-Float-Wert als Ausgabe. Auf den vier Komponenten der Quad-Floats wird gleichzeitig operiert. Es handelt sich also um SIMD- (*Single Instruction Multiple Data*) Instruktionen. Operationen, die normalerweise einen Skalar *s* und keinen Vektor als Ausgabe produzieren würden – dies sind `DP3` und `DP4` – erzeugen Ausgaben der Form (*s*, *s*, *s*, *s*).
Die Ausgabe einer Instruktion wird entweder in einem der 12 temporären Register oder in einem der 13 Ausgaberegister abgelegt. Hierbei kann über eine Schreibmaske (vgl. Abbildung 19: „`wmask`“) festgelegt werden, welche der Komponenten überschrieben werden dürfen.
Die maximal drei Eingabeparameter einer Instruktion können temporäre Variablen, uniforme Attribute oder variierende Attribute sein. Wie aus der Abbildung 19 zu entnehmen, besteht hier aber eine Beschränkung: Von den *uniformen* und *variierenden Attributen* darf pro Instruktion jeweils nur eines als Eingabeparameter verwendet werden. Will man mehr als ein uniformes bzw. variierendes Attribut als Eingabeparameter für einen Befehl nutzen, müssen die überzähligen Attribute zuvor, über die Instruktion `MOV`, in temporären Variablen zwischengespeichert werden.

Im Instruktionssatz sind keine Anweisungen zur Steuerung des Ausführungspfades wie beispielsweise *if*- oder *goto*-Anweisungen enthalten. Es sind daher weder bedingte noch unbedingte Sprünge möglich, und ein VPU-Programm muss sequentiell, Befehl für Befehl, abgearbeitet werden.
Die Länge eines GeForce3-VPU-Programms ist auf 128 Instruktionen beschränkt.

#### *3.2.2.5 Swizzle, Negation und Schreibmaske*

Es gibt Modifikatoren, die Einfluss auf die Durchführung einer Operation haben können:

- Durch eine *Schreibmaske* kann festgelegt werden, welche der Komponenten eines Zielregisters durch die Ausgabe einer Operation überschrieben werden und welche ihren alten Wert behalten.

- Vor Durchführung einer Instruktion lässt sich jeder Eingabevektor negieren (vgl. Abbildung 19: „neg“)
- und dessen Komponenten lassen sich beliebig gegeneinander austauschen (*Swizzling*, vgl. Abbildung 19: „swzl“).

Das Konzept *Swizzling* ermöglicht unter anderem speichereffizientes Rechnen mit Skalaren (zur Erinnerung: alle Instruktionen operieren ausschließlich auf Quad-Floats, also auf vierdimensionalen Vektoren):

**Beispiel 3:** Das Rechnen mit Skalaren auf einer Vertex Engine.

Sei $s$ ein Skalar, und als Komponente eines Quad-Floats $v$ in einem beliebigen lesbaren Register gespeichert. Sei $I$ nun eine Instruktion, die $s$ als Eingabeattribut verwenden soll. Durch *Swizzling* können vor Ausführung von $I$ alle Komponenten des Quad-Floats $v$ auf $s$ gesetzt werden und es ergibt sich als Resultat ein „Skalar-Vektor“ ($s$, $s$, $s$, $s$). Ohne die Swizzeling-Funktionalität müssten alle Skalare als ebensolche „Skalar-Vektoren“ gespeichert werden und benötigten jeweils ein ganzes Quad-Float-Register statt nur einem viertel.

### 3.2.3 Die GeForce3-FPU

Die FPU des GeForce3, von NVDIA „Pixel Shader“ getauft, berechnet für jedes Fragment einen einzelnen RGBA-Wert. Sie operiert konzeptionell immer nur auf einem Fragment und dessen Eingabeattributen: *Tiefenwert*, $COL0_F$, $COL1_F$, $TEX0_F$ – $TEX7_F$, $FOGP_F$ und *Nebelfarbe*.

#### *3.2.3.1 Das konzeptionelle Modell der GeForce3-FPU*

Die FPU der GeForce3 ist eine Weiterentwicklung der Multi-Texturing-Komponente seines direkten Vorgängers GeForce256. Abbildung 20 und Abbildung 21 zeigen deutlich die starke Ähnlichkeit zwischen der alten und der neuen Komponente. Bei der Entwicklung der Fragment-Verarbeitungseinheit hat man also viel eher auf bereits vorhandene Technik zurückgegriffen, als bei der VPU.

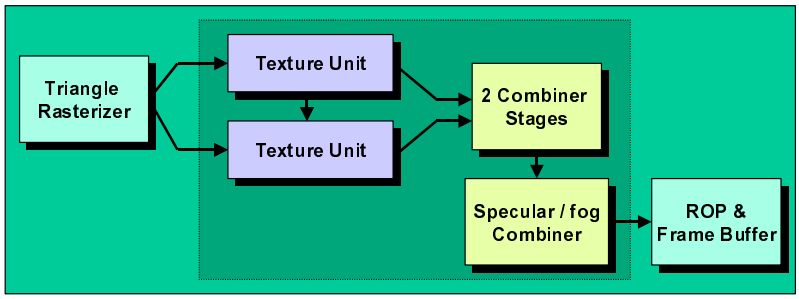



**Abbildung 20: Fragment-Verarbeitungseinheit (bzw. Mutitexturing-Komponente) der GeForce256 [Kirk01]**

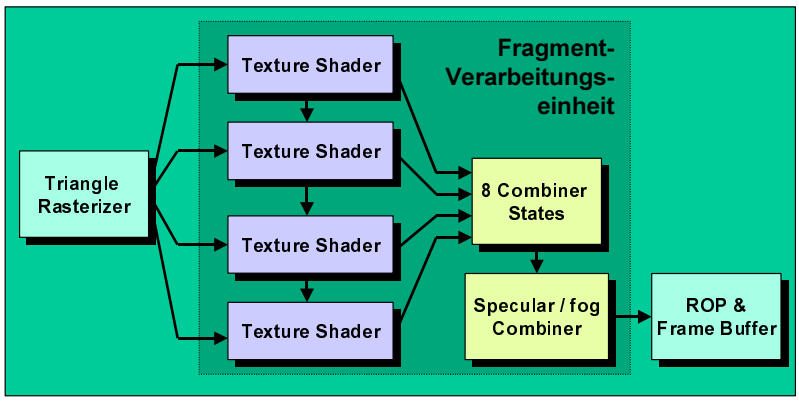



**Abbildung 21: Die Fragment-Verarbeitungseinheit der GeForce3 [Kirk01]**

Im Vergleich zur entsprechenden GeForce2-Komponente (siehe oben) wurden 6 *Combiner States* hinzugefügt und die beiden *Texture Units* durch vier, funktional mächtigere, *Texture Shader* ersetzt.

Die FPU setzt sich aus zwei Hauptkomponenten zusammen:

- den ***Texture Shadern***,
- und den ***Register Combinern*** (in Abbildung 21 gelb markiert).

Dementsprechend ist der Prozess der Fragment-Verarbeitung mit der FPU ein zweistufiger Prozess:

1. Stufe – *Texture Fetching*: Aus gegebenen Texturen und Texturkoordinaten ($TEX0_F$ – $TEX7_F$) werden RGBA-Werte ermittelt. Die *Texture Shader* übernehmen diese Aufgabe [Domi01].

2. Stufe: Aus den beim Texture Fetching ermittelten RGBA-Werten, den Werten $COL0_F$, $COL1_F$ sowie dem Nebelfaktor $FOGP_F$ und der Nebelfarbe wird für ein

einzelner RGBA-Wert berechnet. Hierfür sind die *Register Combiner* zuständig [Spit01].

#### *3.2.3.2 Texture Shaders*

Wie aus Abbildung 21 zu entnehmen, werden die vier *Texture Shader* stufenweise durchlaufen. Jede Stufe ist hierbei mit einer bestimmten Textur und einem bestimmten Texturkoordinatenvektor verknüpft. Mit diesem System kann auf vier verschiedene Texturen pro Pipeline-Durchlauf zugegriffen werden.
Eine Stufe produziert zwei Resultate: einen RGBA-Farbwert, der an die *Register Combiner* weitergegeben wird (*Unit Result*) und einen Wert, der an die nächsten Stufen weitergereicht wird.

Jeder der vier *Texture Shader* führt eines von 23 vordefinierten Programmen aus. Die Programme sind vorwiegend verschiedene *Texture-Sampling-Funktionen* aber es gibt auch einige Spezialfunktionen.
*Tabelle 18*, *Tabelle 19* und *Tabelle 20* auf den Seiten 167-170 listen alle zur Auswahl stehenden Programme auf. Die Namen der Programme stammen aus den von NIVIDIA für ihre Grafikchips entwickelten *OpenGL Extensions*, „NV_texture_shader“ und „NV_texture_shader2“ [Nvid02b].
Neben den Texturen und Texturkoordinaten, kommen bei einigen Programmen benutzerdefinierte Konstanten als zusätzliche Eingaben zum Einsatz. Falls solche Konstanten in ein Programm eingehen, wird dies explizit erwähnt. Die jeweils letzte Spalte der Tabellen (*Unit Result*) gibt den RGBA-Farbwert an, der als Resultat an die *Register Combiners* weitergereicht wird.

Es gibt fünf Typen von *Texture-Shader-Programmen*:

- **Spezial-Funktionen** (gelbe Felder):
    - ***NONE***: Der betreffende *Texture Shader* führt keine Berechnungen durch.
    - ***Fragment Culling***: Das betreffende Fragment wird verworfen und nicht weiterverarbeitet.

- ***Pass Through***: Statt die Texturkoordinate $T \in \{TEX0_F, ..., TEX7_F\}$ zum *Texture Sampling* zu verwenden, wird sie direkt in einen Farbwert konvertiert.
- ***Depth Replace***: Der Tiefenwert des Fragments wird durch einen anderen Wert ersetzt.

- **Standard-Texture-Sampling** (grüne Felder): Standard-Texture-Sampling-Programme für *1D*-, *2D*-, *3D*-Texturen, Rectangle-Texturen[6], und Cube Maps.

- **Bedingtes-Texture-Sampling** (orangene Felder): Für das Texture Sampling aus einer 2D-, oder Rectangle-Textur werden nicht die vorgegebenen Texturkoordinaten benutzt. Das Programm verwendet statt dessen Komponenten aus dem Unit Result der vorherigen Texture-Shader-Stufe.

- **Versatz-Texture-Sampling** (blaue Felder): Vor dem Texture Sampling aus einer 2D- oder Rectangle-Textur werden die Texturkoordinaten zunächst um einen bestimmten Wert versetzt. Dieser Wert wird aus dem Texture-Sampling-Resultat der vorherigen Texture-Shader-Stufe erzeugt. Damit lässt sich ein, dem Bump Mapping ähnlicher, Effekt erzielen[7].

- **Vektorprodukt-Texture-Sampling** (graue Felder): Auch in diesen Programmen werden die vorgegebenen Texturkoordinaten vor dem *Texture Sampling* manipuliert. Die neuen Koordinaten errechnen sich aus dem Vektorprodukt zwischen einer benutzerdefinierten 3x2-Matrix bzw. 3x3-Matrix und einem Vektor, der das Ergebnis eines vorausgehenden Texturzugriffes ist. Der Programmierer kann diese Programme zur Durchführung von Beleuchtungsrechnungen per Fragment oder zu einer

---

[6] *Rectangle-Texturen* sind zweidimensionale Texturen. Im Unterscheid zu (herkömmlichen) 2D-Texturen, müssen die Seitenlängen einer *Rectangle-Textur* - gemessen in Texels - *keine* Zweierpotenzen sein [Nvid02b].

[7] Dies entspricht der Funktionalität des sogenannten „Environment-Mapped Bump Mapping“ (EMBM) der DX6/7/8-Multi-Texturing-Schnittstelle [Nvid02b] (siehe S.37).

qualitativ besseren Variante von Environment-Mapped Bump Mapping verwenden, als jene die die Versatz-Programme bieten.

In vielen dieser Fälle sind nur bestimmte Programme für die Vorgängerstufen zulässig, d.h. einige Programme lassen sich nur im Zusammenschluss mit anderen Programmen verwenden. Um dieser Kontextsensitivität Rechnung zu tragen, habe ich in den Tabellen alle notwendigen Programme und deren Ausgabe in der korrekten Reihenfolge als Block aufgeführt. Vor dem *Programmnamen* und der *Ausgabe* steht ein Index, der die Texture-Shader-Stufe angibt, auf die sich Programm auf Ausgabe beziehen können.

#### *3.2.3.3 Register Combiners*

Die *Register Combiner* berechnen aus den Farbausgaben der *Texture Shader* und aus den übrigen interpolierten Attributen des Fragments – dies sind $COL0_F$, $COL1_F$, $FOGP_F$ und Nebelfarbe – einen einzelnen Farbwert. Dieser entspricht dann der Ausgabe der FPU.

Der Basisdatentyp der Register Combiner ist, wie bei der VPU, ein vierdimensionaler Vektor. Die Vektorkomponenten haben aber eine geringere Präzision und Reichweite: Es handelt sich um 8- oder 9-Bit-Festkomma-Repräsentationen mit einem Wertebereich von [-1,1]. Die Komponenten des Basisdatentyps werden von NVIDIA mit „R“, „G“, „B“ und „A“ (oder „Alpha“) bezeichnet, obwohl die Komponenten auch negative Werte annehmen können[8].

> **Definition 3.2**: Ich werde den Basisdatentyp der Register Combiners im Folgenden „**Signed-RGBA**“ nennen.

[8] Die Komponenten von RGBA-Werten liegen normalerweise im Wertebereich [0,1] ([Watt00], S. 423; [Möll00], S.86).

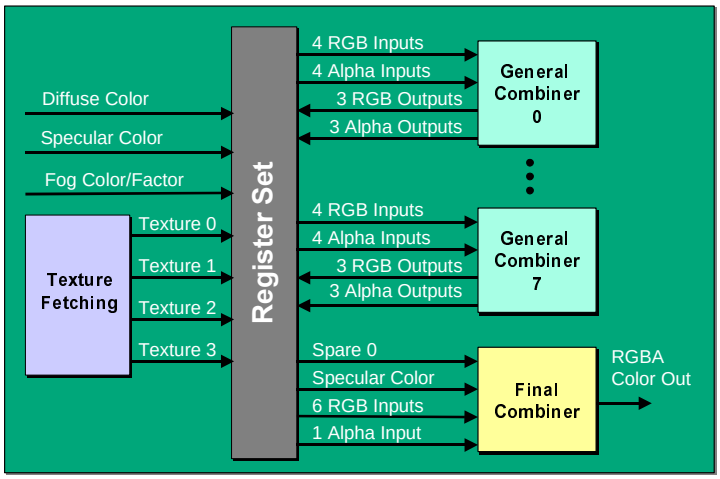



**Abbildung 22: Das Programmmodell des GeForce3 –Register-Combiners [Spit01]**

Die GeForce3-FPU verfügt über acht *Register Combiner*. Die acht Funktionseinheiten arbeiten sequentiell, angefangen mit dem „General Combiner 0“ bis zum „General Combiner 7“ und dem abschließenden „Final Combiner“ (vgl. Abbildung 22). Daher spricht man auch von *Combiner Stages* [Nvid02a] (*Combiner-Stufen*).

Die Kommunikation zwischen den Combiner-Stufen läuft über einen gemeinsamen Speicher („Register Set“). Hier liest ein Combiner die Eingabedaten für seine Berechnungen und legt (mit Ausnahme des „Final Combiners“) die Ergebnisse seiner Berechnungen anschließend auch wieder ab.

Ein „General Combiner“ nimmt vier Signed-RGBA-Werte als Eingabe und erzeugt daraus drei Signed-RGBA-Werte als Ausgabe. Der abschließende „Final Combiner“ berechnet aus vier Signed-RGBA-Werten einen einzelnen RGBA-Wert. Dieser entspricht dann der Ausgabe der FPU und wird an die folgenden Pipeline-Stufen weitergegeben.

Tabelle 9 listet alle Register auf über die ein Register Combiner jeweils verfügen kann. Jeder Combiner hat zwei speziell ihm zugeordnete Register („Konstante Farbe 0“ und „Konstante Farbe 1“), auf die nur er zugreifen kann. Diese beiden Register speichern benutzerdefinierte uniforme Werte.

| Register | Initialisiert mit | General Combiner | | Final Combiner | |
|---|---|---|---|---|---|
| | | Lesen | Schreiben | Lesen | Schreiben |
| Diffuse Farbe | $COL0_F$ | Ja | Ja | Ja | Nein |
| Spekulare Farbe | $COL1_F$ | Ja | Ja | Ja | Nein |
| Texturfarbe 0 | Unit Result 0 | Ja | Ja | Ja | Nein |
| Texturfarbe 1 | Unit Result 1 | Ja | Ja | Ja | Nein |
| Texturfarbe 2 | Unit Result 2 | Ja | Ja | Ja | Nein |
| Texturfarbe 3 | Unit Result 3 | Ja | Ja | Ja | Nein |
| RESERVE 0 | (0, 0, 0, 0) | Ja | Ja | Ja | Nein |
| RESERVE1 | (0, 0, 0, 0) | Ja | Ja | Ja | Nein |
| Nebelfarbe und -faktor | Farbe: Benutzerdefiniert<br>Faktor: $FOGP_F$ | Nur Nebelfarbe | Nein | Ja | Nein |
| Konstante Farbe 0* | Benutzerdefiniert | Ja | Nein | Ja | Nein |
| Konstante Farbe 1* | Benutzerdefiniert | Ja | Nein | Ja | Nein |
| NULL | (0, 0, 0, 0) | Ja | Nein | Ja | Nein |
| Verwerfen | -- | Nein | Ja | Nein | Nein |

**Tabelle 9: Die Register eines Register Combiners [Spit01]**

*Jeder Combiner hat seine eigene „Konstante Farbe 0“ und „Konstante Farbe 1“. Die anderen aufgeführten Register sind global, also für jeden Combiner die selben.

Die in der Tabelle orange markierten Register dienen zur Kommunikation zwischen den Combiner-Stufen, d.h. sie können von den Combinern beschrieben und gelesen werden.

Einige dieser Register speichern zu Beginn der Berechnungen die Eingabeattribute, d.h. die werden mit dern Werten $COL0_F$, $COL1_F$ und den Ausgaben der vier *Texture Shader* (*Unit Result* 0-3) initialisiert. Wegen der Schreibrechte der Combiner können diese Werte während der Berechnungen überschrieben werden, so dass unter Umständen nur der erste Combiner über alle Eingabewerte verfügt. Etwas entschärft wird diese Problematik durch das Vorhandensein zweier zusätzlicher Reserveregister zur Speicherung von Zwischenergebnissen..

*RGB-Anteile* und *Alpha-Anteile* der Ausgaben eines General Combiners werden parallel und in separaten Funktionseinheiten berechnet (vgl. Abbildung 23: RGB-Funktion und Alpha-Funktion). Zur Berechnung des RGB-Anteils stehen alle vier

Komponenten eines Eingabevektors zur Verfügung, zur Berechnung des Alpha-Anteils nur die B- und Alpha-Komponenten.

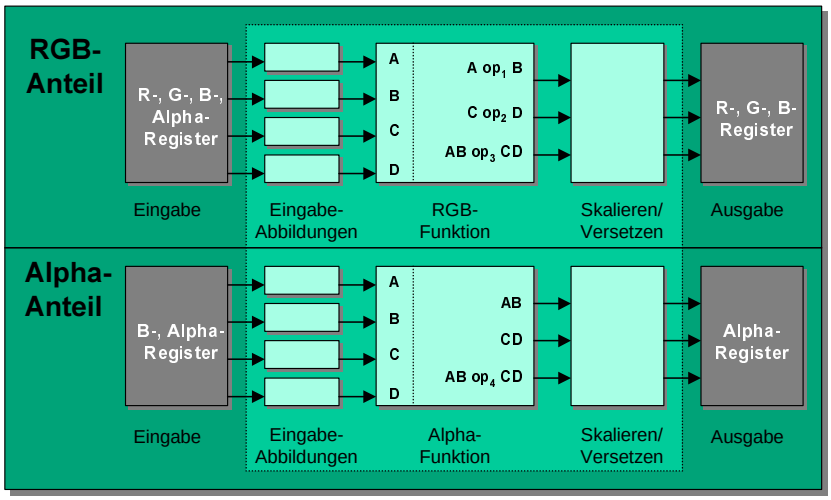



**Abbildung 23: Programmmodell eines General Combiners [Spit01]**

Die möglichen Belegungen der Platzhalter $op_1$, $op_2$, $op_3$, $op_4$ durch Operation kann Tabelle 11 entnommen werden. Eingabe- Ausgabeabbildungen sind in Tabelle 10 aufgeführt.

| Eingabe-Abbildungen | Ausgabe-Abbildungen |
|---|---|
| f(x) = x | F(x) = x |
| f(x) = -x | F(x) = x/2 |
| f(x) = { 0, für x<0<br>x, sonst | f(x) = {-1, für x < -½<br>1, für x > ½<br>2x, sonst |
| f(x) = {-½, für x<0<br>x - ½, sonst | f(x) = {-1, für x < -¼<br>1, für x > ¼<br>4x, sonst |
| f(x) = {-1, für x<0<br>2x - 1, sonst | f(x) = {-1, für x < -½<br>x - ½, sonst |
| f(x) = { 1, für x<0<br>1-x, sonst | f(x) = {-1, für x<0<br>2x - 1, sonst |
| f(x) = { 1, für x<0<br>-2x + 1, sonst | -- |
| f(x) = { ½, für x<0<br>-x + ½, sonst | -- |

**Tabelle 10: Eingabe- und Ausgabeabbildungen eines General Combiners [Spit01]**

Für x gilt: $x \in [-1,1]$.

Zur Transformation der Eingabevektoren einer Funktionseinheit stehen Eingabeabbildungen und zur Transformation der berechneten Ausgaben stehen Ausgabeabbildung zur Verfügung. Tabelle 10 listet alle diese Abbildungen auf.
Die Funktionseinheiten eines General Combiners können vier verschiedene Operationen ausführen:

- Vektorprodukt (geschrieben A•B),
- komponentenweise Multiplikation (geschrieben: AB),
- komponentenweise Addition (geschrieben: A+B),
- Selektion: mux(A,B):= A, wenn RESERVE0 < ½, ansonsten B.

Tabelle 11 zeigt mit welchen Operationskonstellationen die drei Ausgänge (OUT1, OUT2, OUT3) einer Funktionseinheit belegt werden können.

| RGB-Funktionseinheit | | | | | |
|---|---|---|---|---|---|
| OUT1:<br>OUT2:<br>OUT3: | AB<br>CD<br>AB+CD | AB<br>CD<br>mux(AB, CD) | A•B<br>C•D<br>Verwerfen | A•B<br>CD<br>Verwerfen | AB<br>C•D<br>Verwerfen |
| **Alpha-Funktionseinheit** | | | | | |
| OUT1:<br>OUT2:<br>OUT3: | AB<br>CD<br>AB+CD | AB<br>CD<br>mux(AB, CD) | -- | -- | -- |

**Tabelle 11: Mögliche Ausgaben der Funktionseiheiten eines General Combiners [Spit01]**

Nachdem die acht General Combiner, der Reihe Werte aus den Registern entnommen und transformiert haben, berechnet der Final Combiner zum Abschluss aus den Inhalten der Register einen einzelnen RGBA-Wert. Man beachte, dass er als einziger Combiner auf die „Nebelfarbe“ zugreifen kann (vgl. Tabelle 9). Zur Transformation von Ein- und Ausgaben stehen ihm nur zwei Funktionen zur Auswahl, die beide auf den Wertebreich [0,1] abbilden:

```
f(x) =      { 0,   für x<0
              x,   sonst

f(x) =      { 0,   für x<0
              1-x, sonst
```

Den RGB-Anteil der Ausgabe berechnet der Final-Combiner durch den Ausdruck:

```
AB + (1-A)C + D,
```

wobei `A`, `B`, `C` und `D` beliebige lesbare Register (vgl. Tabelle 9) sein können, deren Komponenten durch die oben beschriebenen Eingabeabbildungen transformiert wurden. Der Alpha-Anteil der Ausgabe wird direkt der B- oder Alpha-Komponente eines beliebigen lesbaren Registers entnommen [Spit01].

Welche Leistungsfähigkeit die gerade vorgestellte Architektur der GeForce3 bietet bespreche ich in Kapitel 4. Zunächst möchte ich noch kurz auf die anderen Vertreter der neuen GPU-Generation eingehen.

### 3.2.4 **Andere Massenmarktchips**

#### *3.2.4.1 GeForce3-Variante für die Spielkonsole „XBox“*

Für die Spielekonsole “XBox” von Microsoft hat NVIDIA einen zum GeForce3 fast identischen Chip entwickelt.
Dadurch bieten sich Microsoft und NVIDIA wichtige strategische Vorteile: Für 3D-Spieleprogrammierer wird es nun verhältnismäßig leicht, einen neuen Titel gleich für zwei Plattformen anzubieten: für den PC und für die Spielekonsole XBox. Diese Synergieeffekte könnten außerdem dazu führen, dass Spiele eher in Hinblick auf GeForce3-basierende Grafikkarten optimiert werden als auf Mitbewerber-Chips.

#### *3.2.4.2 GeForce4 Ti*

Der GeForce4 Ti [NVID02] ist der direkte Nachfolger der GeForce3. Er hat im Wesentlichen denselben Aufbau und verfügt über dieselbe Funktionalität wie sein Vorgänger. Man hat aber daran gearbeitet, durch Optimierungen die Performance zu steigern. Die VPU besteht in diesem Chip beispielsweise aus zwei parallel arbeitenden Vertex Engines. Einige funktionale Verbesserungen bietet die FPU. Dort hat man den Satz der Texture Shader Programme erweitert. Diese Erweiterungen bringen aber keine entscheidenden funktionalen Vorteile

gegenüber dem GeForce3-Chip. Für genaue Details hierzu verweise ich auf [NVID20b].

#### *3.2.4.3 Radeon8500 und Radeon9000*

Der Radeon8500-Chip des Herstellers ATI [ATI02] war der erste direkte Konkurrent zur GeForce3, der nicht aus dem Hause NVIDIA stammte. Vor einigen Wochen erschien mit dem Radeon9000 ein zweiter Chip aus dem Hause ATI. Dieser verfügt im Wesentlichen über die selbe Funktionalität wie der Radeon8500-Chip.

Wie die GeForce3 enthalten beide Chips eine benutzerprogrammierbare VPU, bestehend aus zwei Vertex Engines. Diese entsprechen in Aufbau und Funktionalität fast deckungsgleich der GeForce3-Vertex-Engine [Dogg02], weshalb ich nicht weiter auf sie eingehen möchte. Auch im sonstigen Aufbau finden sich meist nur in wenigen Details Unterschiede – ein Fakt der wenig verwunderlich ist, denn die beiden konkurrierenden Unternehmen haben die Weiterentwicklungen des jeweils anderen schon seit Jahren für die eigenen Chips adaptiert [Tom02] (auch wenn man sich anschließend bei der Namensgebung der Techniken scheinbar umso deutlicher voneinander abzusetzen versucht).

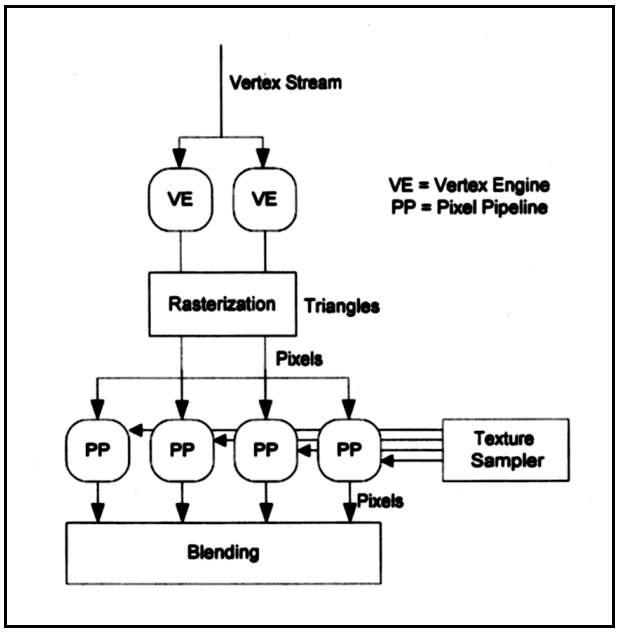



**Abbildung 24: Die programmierbare Pipeline einer Radeon8500-GPU [Dogg02]**

In der Gestaltung ihrer Fragment-Verarbeitungseinheit ist ATI aber einen anderen Weg gegangen als NVIDIA und hat ein etwas flexibleres Modell entwickelt. Statt auf vier kann man auf sechs Texturen pro Pass zugreifen, und die bedingten Texturzugriffe sind nicht wie bei GeForce3/4-FPUs durch fest vorgegebene Programme realisiert (Texture Shaders, Tabelle 18 - Tabelle 20), sondern dem Programmierer wird die Freiheit gegeben, diese Programme selber zu entwickeln.
Dadurch kommt das Programmmodell der Radeon8500/9000-FPU mit einem kleineren Satz an Texturinstruktionen aus. Dies sind Instruktionen zum einfachen Texture Sampling, eine Instruktion zum Weiterreichen der Texturkoordinaten, eine Instruktion zum Setzen des Tiefenwertes, sowie eine Instruktion zum Verwerfen des Fragmentes.
Man kann also sagen, dass NVIDIA bei ihrem Design eher einen CISC-Ansatz und ATI eher einen RISC-Ansatz verfolgte.

ATI realisiert die Möglichkeit des bedingten Texturzugriffes durch ein in zwei Phasen unterteilten Prozess [Mitc02]:
In der ersten Phase schreibt der Programmierer ein Programm zur Berechnung der Texturkoordinaten, in der zweiten Phase verwendet er die berechneten

Texturkoordinaten zum Texture Sampling. Anschließend kombiniert er mit Register Combinern die erhaltenen Farbwerte mit den Farbwerten der übrigen Register. Die erste Phase ersetzt also die Funktionalität der vordefinierten GeForce3/4-Funktionen.

#### *3.2.4.4 Radeon9700 und Matrox Parthelia-512*

Vor einigen Wochen sind bereits Grafik-Chips erschienen, die auf die Funktionalität der kommenden DirectX-Version [Micr02c] ausgerichtet sind. Diese wird voraussichtlich frühestens am Ende diesen Jahres erscheinen, und außer herstellerspezifischen OpenGL-Extensions existieren bis dato auch noch keine anderen Interfaces für diese Chips. Die funktionalen Vorteile dieser GPUs sind somit für Software-Entwickler noch schwer zu nutzen.

Der Parhelia-512-Chip [MATR02] hat gegenüber der GeForce3 eine verbesserte VPU, die über einen erweiterten Instruktionssatz mit Kontrollkonstrukten und über eine größere Anzahl von Registern verfügt sowie längere Programme ausführen kann. Sowohl Schleifen und bedingte Sprünge als auch die Definiton von Prozeduren sind erlaubt. Die Programmlänge der VPU wurde auf 1024 Instruktionen erhöht und die Anzahl der Konstanten auf 256. Die FPU des Chips entspricht weitestgehend der Fragment-Verarbeitungseinheit der GeForce4 Ti.

Der Radeon9700-Chip hat ebenfalls eine solche verbesserte VPU, verfügt zusätzlich aber auch noch über eine stark erweiterte FPU. Sie erlaubt Berechnungen mit einer Präzision von 16 und 32 Bit pro Kanal, und die Programmlänge wurde auf maximal 32 Texturinstruktionen und maximal 64 arithmetische Instruktionen erweitert. Desweiteren ist Zugriff auf bis zu 16 verschiedene Texturen pro Pass möglich. Kontrollstrukturen wurden auf dieser Ebene aber nicht eingeführt.

Diese erweiterten Chips sind schon zur nächsten GPU-Generation zu zählen und sind nicht Gegenstand dieser Arbeit. Dennoch gehe ich am Ende von Kapitel 5 im Rahmen eines Ausblickes auf kommende Entwicklungen noch einmal auf sie ein.

## 3.3 Die Einbindung der neuen Hardware in Application Programming Interfaces

Ich schließe dieses Kapitel mit einem Blick darauf, wie die neue Hardware in Application Programming Interfaces eingebunden wurde.

### 3.3.1 DirectX8

#### *3.3.1.1 Vertex und Pixel Shader*

DirectX8 bietet zur Steuerung der beiden programmierbaren Komponenten zwei separate Interfaces an, eines für die VPU und eines für die FPU. In beiden Fällen wird ein spezielles Programmmodell zur Verfügung gestellt, mit dessen Hilfe der Benutzer Programme zur Steuerung der jeweiligen Verarbeitungseinheit schreiben kann. Diese werden kompiliert und anschließend Befehl für Befehl auf der jeweiligen Einheit ausgeführt.
Ein solches Programm wird normalerweise in einer herkömmlichen Text-Datei gespeichert und anschließend von einem speziellen Compiler in ein Byte-Code-Programm umgewandelt.

Die Namensgebung um die neue Hardware ist etwas irritierend: Das programmierbare Interface für die VPU wurde in DirectX8 (DX8) *Vertex Shader*, das programmierbare Interface für die FPU *Pixel Shader* getauft. Die selben Bezeichnungen werden, wie bereits erwähnt, auch von den Chip-Herstellern verwendet. Bei diesen stehen die Namen „Vertex Shader“ und „Pixel Shader“ aber nicht für Interfacekomponenten, sondern für Hardwarekomponenten, also für die *VPU* und *FPU* selbst [Nvid01a, Nvid01b, Nvid01c].
Die Vermischung von Hardware und Interface in der Namensgebung, soll ein erster Hinweis darauf sein, wie dünn die Abstraktionsschicht ist, die zwischen beiden liegt. In Abschnitt 4.2.1 werde ich die Programmmodelle der beiden Interfaces vorstellen und zeigen, dass diese tatsächlich in weiten Teilen völlig mit den Programmmodellen der Hardwarekomponenten übereinstimmen.

### 3.3.1.2 *Die DX8-Pipeline*

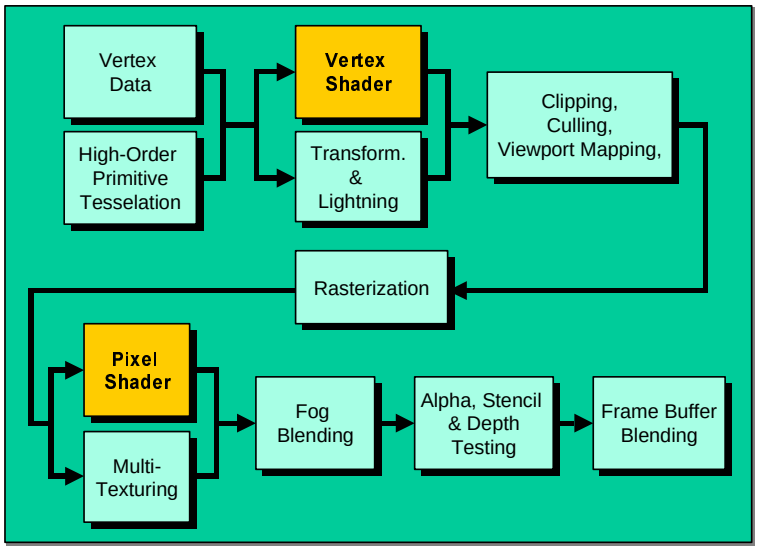



**Abbildung 25: Die DX8-Rendering-Pipeline [Micr02b]**

Abbildung 25 zeigt die Position der beiden neuen Interfacekomponenten in der Bildgenerierungs-Pipeline von DirectX8 (DX8). Der *Vertex Shader* bildet eine Alternative zur *Transform&Lightning*-Komponente (*T&L*), der *Pixel Shader* eine Alternative zur *Multi-Texturing*-Komponente (vgl. Abbildung 5 auf S. 10).

Der Programmierer muss sich jeweils für eine Alternative entscheiden. Das heißt, es dürfen nicht Vertex Shader und T&L bzw. Pixel Shader und Multi-Texturing gleichzeitig verwendet werden. Der Grund hiefür ist leicht ersichtlich: Sowohl die alten Interfaces, als auch die neuen Alternativen steuern jeweils die selbe Hardware-Komponente – nämlich die VPU bzw. die FPU. Man kann also in beiden Fällen lediglich zwischen zwei Möglichkeiten der Steuerung wählen, die zugrundliegende Hardware ist die selbe.

| Geometrieverarbeitung | Shading |
|---|---|
| Transform&Lightning | Multi-Texturing |
| Transform&Lightning | Pixel Shader |
| Vertex Shader | Multi-Texturing |
| Vertex Shader | Pixel Shader |

**Tabelle 12: Mögliche Kombinationen von Interfacekomponenten für die Bildgenerierung in der DX8-Rendering-Pipeline [Micr02b]**

Es ist weiterhin möglich, ausschließlich mit den alten Komponenten zu arbeiten, und auch Kombinationen zwischen Alt und Neu sind erlaubt. Das heißt, bezogen auf diese vier Komponenten, kann der Programmierer zwischen vier unterschiedlichen Pipelines wählen (vgl. Tabelle 12).

Vertex Shader und Pixel Shader sind (zumindest theoretisch) fähig, die Funktionalität der beiden alten Komponenten vollständig zu ersetzen [Micr02b] [Lind02], und bieten darüber hinaus eine flexiblere und mächtigere Architektur. Tabelle 13 listet einige der Funktionalitäten der Shader-Komponenten auf. Wie ich in Kapitel 5 zeigen werde, sind die neuen Komponenten in vielerlei Hinsicht schwer benutzbar, weshalb die alten Interface-Modelle aufgrund ihrer einfachen Bedienbarkeit in vielen Fällen sogar die bessere Alternative darstellen.

| Vertex Shader / VPU | Pixel Shader / FPU |
|---|---|
| Geometrische Transformation (Überführung d. Geometrie vom Model-Space in den View-Space) | Texture Sampling |
| Animation per Vertex (Vertex-Blending, Tweening) | Kombinieren von Texture Samples und interpolierten Farbwerten |
| Beleuchtungsrechnung per Vertex | Beleuchtungsrechnung per Pixel |
| Generierung der Texturkoordinaten per Vertex | |
| andere Vorberechnungen für die FPU | |

**Tabelle 13: Typische Aufgaben der programmierbaren Hardware- und Interfacekomponenten.**

Ich fand aber auch noch andere Gründe, die die Entscheidung, sich nicht von den alten Komponenten getrennt zu haben, stützen:
*Vertex* und *Pixel Shader* sind wie gesagt auf die Funktionalität der neuen Grafikkarten-Generation ausgelegt. Ältere Grafikkarten mit DX7-Funktionalität werden von diesen Komponenten nicht unterstützt. Für die Vertex-Shader-Komponente bietet DX8 zwar eine Software-Emulierung an, das heißt,

Berechnungen, die normalerweise von der VPU ausgeführt würden, laufen vollständig auf der CPU ab. Aber die T&L-Funktionalität von Grafik-Chips bleibt dabei völlig ungenützt. Also gäbe es für diese Fähigkeiten älterer Karten ohne entsprechende Komponente in DX8 keine Unterstützung mehr.
Im Fall der Pixel-Shader-Komponente ist die Lage noch gravierender: Für Pixel-Shader-Programme existiert nämlich kein Software-Emulierungs-Mechanismus. Ohne *Multi-Texturing*-Komponente wären daher Grafikkarten früherer Generationen in DX8 überhaupt nicht mehr zu verwenden.

#### *3.3.1.3 Vertex-Shader-Versionen*

| Version | DX-Version | Unterstützt* |
|---|---|---|
| 0.0 | DX8 | Hardware mit DX7-Transform&Lightning-Funktionalität |
| 1.0 | DX8 | GeForce3, GeForce4 Ti,<br>Radeon 8500, Radeon 9000,<br>Matrox Parthelia-512 |
| 1.1 | DX8, DX8.1 | GeForce3, GeForce4 Ti,<br>Radeon 8500, Radeon 9000 |
| 2.0 | DX9 | Matrox Parthelia-512<br>Radeon 9700 |

**Tabelle 14: Die verschiedenen Vertex-Shader-Versionen**

*ohne Anspruch auf Vollständigkeit

DirectX verfügt zur Zeit über drei verschiedene Vertex-Shader-Versionen (siehe Tabelle 7.1): *Version 0.0*, Version *1.0* und Version *1.1*. Die in der Tabelle aufgeführte *Version 2.0* wird Bestandteil der DX9-API sein und die VPU-Funktionalität des Parthelia-512 und Radeon9700 unterstützen [Micr02c]. Ich habe sie nur der Vollständigkeit halber in die Liste aufgenommen.

Das T&L-Interface wird in DX8 unter *Vertex-Shader-Version 0.0* geführt. Die *Version 1.1* ist auf die Funktionalität der GeForce3-VPU maßgeschneidert. Sie unterscheidet sich von *Version 1.0* nur durch das Vorhandensein eines Adressregisters zur indirekten Adressierung. Es existiert aber bis dato keine Hardware für *Version 1.0*, die nicht gleichzeitig auch *Version 1.1* unterstützt.

#### *3.3.1.4 Pixel-Shader-Versionen*

Fünf verschiedene Pixel-Shader-Versionen sind zur Zeit Bestandteil von DirectX (vgl. Tabelle 7.1): Version 0.0 bis Version 1.4.

Die *Sprachversion 2.0* ist ebenfalls wie auch *Vertex Shader 2.0* Teil der DX9-API [Micr02c] und wird nur der Vollständigkeit halber aufgeführt.

Das Multi-Texturing-Interface wird in DX8 als *Pixel-Shader-Version 0.0* geführt. Die *Versionen 1.0*, *1.1*, *1.2* und *1.3* bauen jeweils aufeinander auf – Programme, die in einer älteren Version geschrieben wurden, sind auch mit den jeweils neueren Versionen kompatibel.

*Pixel Shader 1.4* dagegen ist von den vorherigen Sprachversionen völlig verschieden. Sie wurde für das RISC-Modell der Radeon8500-FPU konzipiert (vgl. Abschnitt 3.2.4.3).

| Version | DX-Version | Unterstützt* |
|---|---|---|
| 0.0 | DX8, DX8.1 | Hardware mit DX7-Multi-Texturing-Funktionalität |
| 1.0 | DX8, DX 8.1 | GeForce3, GeForce4 Ti,<br>Radeon 8500, Radeon 9000<br>Matrox Parthelia-512 |
| 1.1 | DX8, DX8.1 | GeForce3, GeForce4 Ti,<br>Radeon 8500, Radeon 9000<br>Matrox Parthelia-512 |
| 1.2 | DX8.1 | GeForce4 Ti,<br>Radeon 8500, Radeon 9000<br>Matrox Parthelia-512 |
| 1.3 | DX8.1 | GeForce4 Ti,<br>Radeon 8500, Radeon 9000<br>Matrox Parthelia-512 |
| 1.4 | DX8.1 | Radeon 8500, Radeon 9000 |
| 2.0 | DX9 | Radeon 9700 |

**Tabelle 15: Die verschiedenen Pixel-Shader-Versionen**

*ohne Anspruch auf Vollständigkeit

### 3.3.2 OpenGL2.0

Der aktuelle OpenGL 1.4 Standard bietet keine vollständige Unterstützung für benutzerprogrammierbare Hardware. Jedoch ist OpenGL seit jeher eine offene Architektur, die sich durch sogenannte *Extensions* erweitern lässt. NVIDIA, MATROX und ATI bieten solche Extensions zur Steuerung ihrer VPU- und FPU-

Komponenten an. Dadurch begibt sich der Entwickler aber in Hardware-Abhängigkeit. Zumindest für die VPU ist mit Version 1.4 ein Interface in die OpenGL-Architektur aufgenommen worden. Man hat hat hierfür NVIDIAs VPU-Extension verwendet und unterstützt somit die VPU-Funktionalität einer Geforce3.

Mit OpenGL2.0 [Open02b] wird nun aber bald eine Version erscheinen, die eine volle Unterstützung für programmierbare Garfik-Hardware bietet und darüber hinaus eine hardware-unabhängige Shader-Sprache enthalten wird. Diese Shader-Sprache basiert auf der Programmiersprache C und der RenderMan Shading Language. Wie der OpenGL Shader wird sie die Rendering-Pipeline abstrahieren und so die Hardware-Restriktionen vor dem Programmierer weitestgehend verdecken. Integriert in diese High-Level-Sprache werden aber auch Maschinensprachmodelle für die VPU und FPU sein.
Es wird also sowohl eine nachhaltige zukunftsgerichtete Schnittstelle geben, als auch die Möglichkeit speziell auf die aktuelle Grafikchip-Generation optimierte Anwendungen zu entwickeln.

### 3.3.3 Andere Interfaces

Es wurden noch weitere Interfaces zur Steuerung der neuen GPU-Komponenten entwickelt und möchte kurz auch auf deren Eigenarten eingehen. Es ist aber fraglich inwieweit sich diese Interfaces gegenüber den verbreiteten APIs DirectX und OpenGL in der Praxis durchsetzen, zumal beide auch entscheidende Schwächen haben.

#### *3.3.3.1 Cg*

Die NVIDIA Corporation hat vor kurzer Zeit die erste Version einer Sprache veröffentlicht, *Cg* [NVID02c][NVID02d], die sich in die beiden APIs *OpenGL* und *DirectX* einbinden und als Alternative für die dort vorhandenen VPU- und FPU-Interfaces verwenden lässt. Es vor allen Dingen auf die Funktionalität des kommenden NVIDIA-Chips ausgerichtet ist.

Bei der Gestaltung der Sprache hat sich NVIDIA ohnehin sehr an den Eigenarten ihrer eigenen Hardware orientiert, also der GeForce-Serie, so dass meiner Meinung nach eine breite Akzeptanz dieses Modells fraglich ist. Man hat inzwischen zwar den Source Code der Sprache zur Weiterentwicklung veröffentlicht, aber es scheint mir aus firmenpolitischer Sicht nicht wahrscheinlich, dass die Konkurrenten ATI und MATROX Cg-Erweiterungen für ihre eigenen Chips einbringen werden. Aus diesen Gründen werde ich von einer Besprechung dieser Sprache absehen. Im letzten Teil der Arbeit (Kapitel 5), der eine Diskussion über die Benutzbarkeit der neuen Hardware-Generation bietet, werde ich aber noch einige Anmerkungen zu diesem Interface-Modell machen.

#### *3.3.3.2 Stanford Shading Language*

An der Stanford Universität entwickelte man ein System, das ähnlich wie der OpenGL Shader (siehe Abschnitt 2.3.2.2) in einer Shader-Sprache geschriebene Programme auf mehrere GPU-Passes abbildet [Prou01].
Im Unterschied zum OpenGL Shader orientierte man sich mit der Stanford Shading Language aber sehr viel näher an den Architekturen der Grafik-Hardware, insbesondere an der Architektur der neuen Grafikchip-Generation.
Die Sprache ähnelt zwar der RenderMan Shading Language, aber sie verzichtet auf Funktionalitäten, welche die heutige Hardware noch nicht zu bieten hat. Desweiteren unterscheidet sie bei der Durchführung von Operationen zwischen vier verschiedenen „Computation Frequencies“: *Constant*, *Per-Primitive Group*, *Per-Vertex* und *Per-Fragment.* Constant- und Per-Primitive-Group-Operationen werden auf der CPU durchgeführt oder Beeinflussen die Konfiguration der GPU. Per-Vertex-Operationen werden zu VPU-Programmen kompiliert und Per-Fragment-Operationen zu FPU-Programmen.
In [Prou01] werden zwei komplexe Demonstratoren, die mit diesem System erstellt wurden, vorgestellt. Beide laufen auf einem 866-MHz-Pentium-III-System mit einem GeForce-Chip in Echtzeit. Die Stanford Shading Language bietet also ein gutes Verhältnis zwischen Benutzerfreundlichkeit und Hardware einerseits, sowie effizienter Verarbeitung andererseits.

Der entscheidende Nachteil des Systems ist, dass es ausschließlich für Shading konzipiert wurde, aber nicht für Animation. Ich werde auf diesen Nachteil am Ende dieser Arbeit (Kapitel 5) noch etwas genauer eingehen.

# 4 Studie der Leistungsfähigkeit und Benutzbarkeit

Nachdem ich Aufbau und Funktionsweise der neuen GPUs geklärt habe, möchte nun zum zweiten grossen Teil meiner Analyse kommen, die sich mit der Leistungsfähigkeit und der praktischen Benutzbarkeit der Chips beschäftigt.

## 4.1 Vorgehensweise

Ich bin zwei Wege gegangen, um Aussagen über die Leistungsfähigkeit und Benutzbarkeit der neuen GPU-Generation treffen zu können:

- Erstens habe ich eine Studie darüber durchgeführt, welche funktionalen Vorteile die neuen GPUs gegenüber ihren Vorgängern bieten und welche Techniken sich auf ihnen realisieren lassen. Hierbei bin ich themenbezogen vorgegangen. Um ein möglichst vollständiges Bild zu erhalten, habe ich mich bei der Wahl der Themen an Standardwerken der GDV [Fole97] [Watt92] [Watt00] und den wissenschaftlichen Veröffentlichungen der letzten Jahrzehnte (insbesondere der SIGGRAPH- und NPAR- Reihe von ACM [ACM02]) orientiert. Anhand der Studie bereits existierender Demonstratoren, die sich auf den Homepages der verschiedenen Hersteller finden, und anhand eigener Überlegungen habe ich für jeden identifizierten Themenbereich überprüft, inwieweit sich durch die neue GPU-Generation Vorteile ergeben.
- Zweitens habe ich mittels der Entwicklung von Leistungsdemonstratoren getestet, wie schwierig oder leicht die Entwicklung von Applikationen ist, die die Vorteile der neuen Hardware nutzen und wo die Grenzen des Machbaren liegen.

In beiden Fällen habe ich mich nicht direkt an den Funktionalitäten der Hardware-Chips, sondern an der Funktionalität, die das *DirectX8-Interface* bietet, orientiert. Im Speziellen betrachtete ich Vertex-Shader-Version 1.1 und Pixel-Shader-Version 1.1.

Hierfür gab es mehrere Gründe: Zum einen war es mir auf diese Weise möglich, allgemeine Aussagen über alle bis dato erschienenen Chips der neuen Generation zu treffen, denn Vertex Shader 1.1 und Pixel Shader 1.1 sind die funktional

komplexesten Shader-Versionen, die von all diesen Chips unterstützt werden. Es handelt sich sozusagen um eine Bewertung auf dem größten gemeinsamen Nenner. Desweiteren ist DirectX8 mit OpenGL das meist genutzte API für Echtzeit-Anwendungen, insbesondere im Bereich der Spieleprogrammierung, und die aktuelle OpenGL-Version 1.4 bietet im Gegensatz zu DirectX8 bisher keine angemessene Unterstützung für die neuen Techniken. Somit konnte ich nur unter Verwendung von DirectX8 einschätzen, wie gut die neue Technik in der Software-Industrie wohl angenommen werden wird.

Zum Testen meiner Demonstratoren stand mir ein GeForce3-Chip zur Verfügung. Zur Implementierung verwendete ich die Entwicklungsumgebung „Visual C++“. Als Rahmen für meine Programme nutzte ich das „*Common Files Framework*“ – eine Sammlung von Klassen und Methoden, die speziell zur Entwicklung einfacher Demos für das DirectX-Interface von Microsoft bereitgestellt wurden. Sie sind Teil des „*Microsoft DirectX Software Development Kid*“ [Micr02a]. Für eine genaue Beschreibung des *Common Files Framework* verweise ich auf [Micr02a] und [Enge02].

Im nächsten Abschnitt werde ich zunächst die Programmmodelle der beiden verwendeten Shader-Versionen sowie deren Einbindung in das übrige Interface vorstellen. Diese Einführung wird recht ausführlich ausfallen, da ich Teile meiner Kritik an der Nutzbarkeit der neuen GPU-Generation (Kapitel 5) von den Schwächen des DirectX8-Interfaces ausgehend begründen werde.

## 4.2 VPU und VPU-Programmierung in DX8

Ich beschreibe in diesem Abschnitt die Interface-Komponenten Vertex Shader 1.1 und Pixel Shader 1.1 sowie deren Verbindung zum übrigen DirectX8-API. Einen grundsätzlichen Überblick über die Lage der Komponenten in der DX8-Pipeline und deren Aufgaben habe ich bereits in Abschnitt 3.3.1 gegeben. Tabellen mit Instruktionen und Modifikatoren finden sich zum Nachschlagen im Anhang (Tabelle 21-Tabelle 27, Seiten 174-176).

### 4.2.1 Das Programmmodell von Vertex Shader Version 1.1

Das Programmmodell der Vertex-Shader-Version 1.1 [Micr02b] entspricht im Wesentlichen den bereits vorgestellten VPU-Programmmodellen – die Abstraktionsschicht zwischen API- und Hardware-Programmmodell ist also sehr dünn.

Für jeden Registertyp und jede Instruktion gibt es eine direkte Entsprechung im Vertex Shader (vgl. Abbildung 26 und Abbildung 19). Der Basisdatentyp eines Vertex Shaders ist, wie bei den Hardware-Komponenten, *Quad-Float*. Das heißt, alle Daten werden durch einen vierdimensionalen Vektor ($x$, $y$, $z$, $w$) repräsentiert, wobei jede Komponente einer 32-Bit-Fließkommazahl (float) entspricht.

Die erlaubte Programmlänge sowie die Zahl der variierenden Attribute („Per Vertex Input“), der temporären Variablen und der Adressregister ist die selbe wie bei den vorgestellten VPUs. Ebenso sind die Ausgaberegister („Per Vertex Output“) in Typ und Anzahl zu jenen in den VPUs äquivalent (vgl. erste und dritte Spalte in Tabelle 21). Die Schreib- und Leserechte sind, wie aus der Abbildung zu entnehmen, identisch zur VPU. Auf Ausgaberegister kann also weder das Shader-Programm noch die Applikation zugreifen.

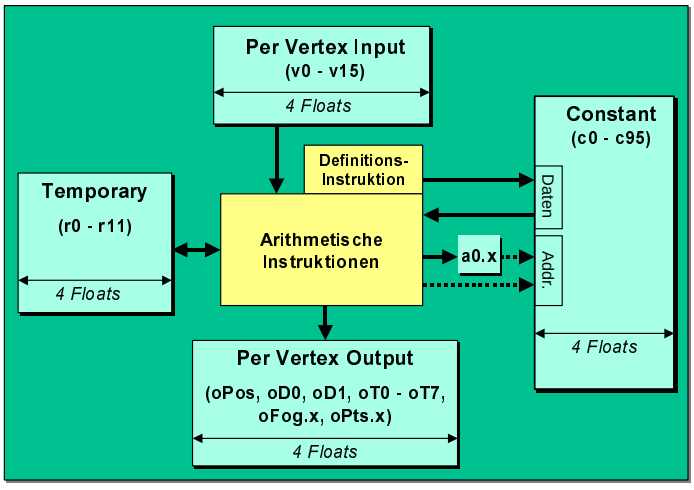



**Abbildung 26: Das Programmmodell eines Vertex Shaders der Version *1.1***

Nicht fest vorgegeben aber ist die verfügbare Anzahl uniformer Werte („Constant“-Register). Wieviele hiervon in einem Vertex-Shader-Programm genutzt werden können, hängt direkt von der Anzahl der entsprechenden Register

auf der zugrundeliegenden VPU ab. Zugesichert wird eine Anzahl von mindestens 96 Registern[9].

#### *4.2.1.1 Vertex-Shader-Register*

Die zweiten Spalte von Tabelle 21 führt die Bezeichnungen der Registertypen in einem Vertex-Shader-Programm auf.
Die vier Komponenten eines Registers werden der Reihenfolge nach mit den Buchstaben *x*, *y*, *z* und *w* benannt. Es wird das folgende Bezeichnungsschema verwendet:

```
Registername.Komponente
```

**Beispiel 4:** Der Ausdruck `r2.y`, bezeichnet die zweite Komponente des Registers `r2`.

#### *4.2.1.2 Vertex-Shader-Instruktionen*

Ein Vertex Shader der Version 1.1 verfügt über einen Satz von 27 verschiedenen Instruktionen. Diese unterteilen sich in eine Versionsinstruktion, eine Definitionsinstruktion sowie 25 arithmetische Instruktionen (17 generelle Instruktionen und 8 Macro-Instruktionen).

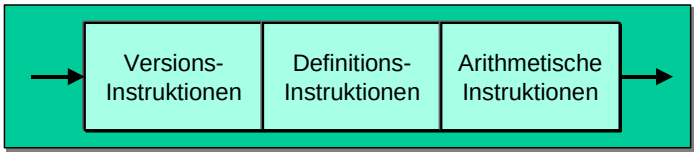



**Abbildung 27: Der Instruktionsfluss eines Vertex Shaders der Version 1.1 [Micr02b]**

Die Versionsinstruktion (`vs.[versionsnummer]`) steht am Anfang eines jeden Pixel-Shader-Programmes, danach folgen die Definitions-Instruktionen (`def`) und im Anschluss daran die arithmetischen Instruktionen und Macro-Instruktionen (vgl. Abbildung 27).

---

[9] Dies entspricht der Anzahl uniformer Register, über welche die GeForce3-VPU verfügt (siehe Kapitel 5).

Die Versionsinstruktion gibt an um welche Vertex-Shader-Version es sich handelt. Programme der Vertex-Shader-Version 1.1 beginnen z.B. mit der Zeile „`vs.1.1`“.

Mit Definitionsinstruktionen kann man den Inhalt von Constant-Register definieren. Der Ausdruck „`def c0, 1.0, 1.0, 1.0, 1.0`“ füllt z.B. das Constant-Register `c0` mit dem Quad-Float (1,1,1,1). Die Versions- und Defintions-Intruktionen sind die einzigen beiden Instruktionstypen, die nicht auf der VPU ausgeführt werden[10]. Hier unterscheidet sich also das Programmmodell des Vertex Shaders vom Programmmodell der zu steuernden VPU.

***Arithmetische Instruktionen***

Die arithmetischen Instruktionen entsprechen im Wesentlichen den Instruktionen des VPU-Programmmodells (vgl. Tabelle 8 mit Tabelle 23). Die einzigen Ausnahmen bilden der nicht vorhandene Befehl zum Laden des Adressregisters – diese Funktionalität übernimmt beim Vertex Shader der Befehl `mov` – und der gegenüber dem VPU-Modell zusätzliche Befehl `nop` („No Operation“).

Auch der Instruktionssatz des Vertex Shaders enthält keinerlei Instruktionen für bedingte Verzweigungen, Schleifen, Sprünge oder Loops, und es besteht ebenfalls keine Möglichkeit, das Programm frühzeitig, d.h. vor Ausführung des letzten Befehls, abzubrechen. Ein Vertex-Shader-Programm wird daher für jeden Vertex auf exakt die selbe Weise, linear und Befehl für Befehl, abgearbeitet.

Für alle Instruktionen gilt die folgende Syntax:

```
Befehlsname Ziel, Quelle1 [, Quelle2 [, Quelle3]]
```

**Beispiel 5:**

```
mov r1, r3
add r1, r2, c3
```

„`Ziel`“, „`Quelle1`“, „`Quelle2`“ und „`Quelle3`“ stehen hierbei für Register des Vertex-Shaders und „`Befehlsname`“ für den Namen einer Instruktion aus dem Befehlsatz. Die mit dem Befehlsnamen verbundene Operation wird mit dem

[10] Ich entnehme dies der Tatsache, dass diese Befehle im Programmmodell (der bisher vorgestellten VPUs) nicht vorkommen (vgl. Tabelle 8), und dass die uniformen Variablen bei diesen grundsätzlich von der Applikation gesetzt werden [Lind01].

Inhalt der Quell-Register als Parameter ausgeführt und das Ergebnis der Berechnung im Zielregister abgelegt.

Wie im VPU-Programmmodell können sowohl auf das Zielregister als auch auf die Quellregister eines jeden Befehls Modifikatoren angewandt werden (vgl. Tabelle 24).

Eine Schreibmaske (Destination Mask) deklariert man durch Anhängen einer Aufzählung der zu überschreibenden Komponenten an den Namen des Zielregisters.

**Beispiel 6:** `mov r1.xz, c2`

Die Schreibmaske führt dazu, dass sich die Instruktion nur auf die *x*-, und *z*-Komponente von `r1` auswirkt, d.h. sie setzt `r1.x = c2.x`, und `r1.z = c2.z`. Die *y*- und *w*-Komponente von `r1` bleibt unverändert.

□

Mit dem Swizzle-Modifikator können die Komponenten eines Quellregisters vor Ausführung einer Instruktion beliebig gegeneinander ausgetauscht werden.

Beispiel 7: Der Ausdruck „`mov r1, c2.zzzy`“ entspricht der Operation:
`r1.x = c2.z, r1.y = c2.z, r1.z = c2.z, r1.w = c2.y.`

Die Negation eines Quellregisters für die Dauer der Instruktion erfolgt durch Voranstellen eines Minus-Zeichens.

Beispiel 8: Der Ausdruck „`mov r1, -c2`“ entspricht der Operation: `r1 = -c2.`

#### *4.2.1.3 Die Verwendung des Adressregisters a0.x*

Für das Setzen des Adressregisters `a0.x` kann ausschließlich über den Befehl `mov` mit Inhalt gefüllt werden. Anschließend kann es nach dem folgenden Schema zur indirekten Adressierung verwendet werden: `c[a0.x + n]` (hierbei sei *n* eine natürliche Zahl).

**Beispiel 9:** Ein Beispiel für die indirekte Adressierung durch Adressregister ist der Ausdruck `c[a0.x + 2]`. Für `a0.x = 1` entspräche dieser Ausdruck dem Registernamen `c3`.

Falls aus der indirekten Adressierung eine Adresse resultieren sollte, die außerhalb des erlaubten Bereiches liegt, führt der Lesezugriff zu unerwarteten Ergebnissen.

### 4.2.2 Pixel Shader 1.1 – Programmmodell

Es folgt nun eine Beschreibung der Pixel-Shader-Version 1.1. Ich habe ein Diagramm für das Programmmodell eines Pixel Shaders der Version 1.1 erstellt (Abbildung 28), das sich für einen direkten Vergleich mit dem entsprechenden Vertex-Shader-Diagramm (Abbildung 26) eignet.

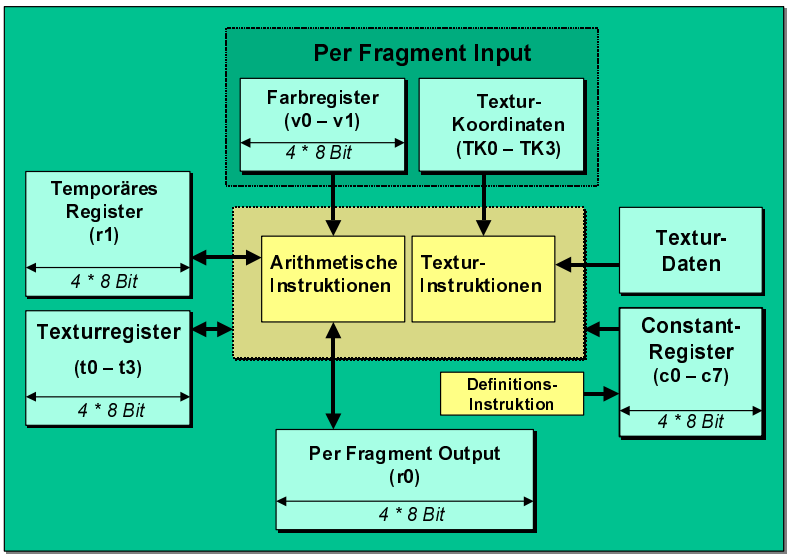



**Abbildung 28: Das Programmmodell eines Pixel Shaders der Version 1.1**

Der Pixel Shader kann auf eine Vielzahl von Datenquellen zurückgreifen. Die variierende Eingabe-Attribute unterteilen sich in zwei Farbwerte (i.d.R. diffuse und spekulare Farbe) und in einen Satz von maximal vier Texturkoordinaten.
Als uniforme Attribute stehen die Texturdaten von maximal vier Texturen sowie sieben Konstanten-Register (Constant Register), die benutzerdefinierte Konstanten speichern, zur Verfügung. Als temporäre Register können die Register

`r0` und `r1` sowie die Texturregister `t0-t3` verwendet werden. Die Ausgabe wird am Ende der Berechnungen in Register `r0` abgelegt.

Die beiden wichtigsten Instruktionstypen sind *arithmetische Instruktionen* und *Texturadressierungsinstruktionen* (kurz: *Texturinstruktionen*). Beide Instruktionstypen können jeweils auf unterschiedliche Datenquellen zugreifen.
Neben Texturinstruktionen und arithmetischen Instruktionen gibt es wie beim Vertex Shader eine Definitions- und eine Versionsinstruktion (`def` und `ps`). Die Versionsinstruktion steht am Anfang eines jeden Pixel-Shader-Programmes und gibt an, um welche Version es sich handelt. Programme der Pixel-Shader-Version 1.1 beginnen z.B. mit der Zeile `ps.1.1`.
Definitions-Instruktionen stehen hinter der Versionsinstruktion, danach folgen die maximal vier Texturinstruktionen und wiederum danach die arithmetischen Instruktionen (vgl. Abbildung 29).

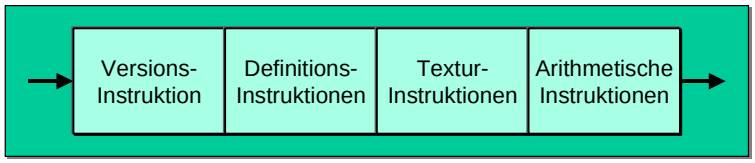



**Abbildung 29: Der Instruktionsfluss eines Pixel Shaders der Version 1.1 [Micr02]**

Ich erwähnte bereits, dass das Interface *Pixel Shader 1.1* auf die Steuerung der GeForce3-FPU ausgerichtet ist. Das Konzept dieses Interfaces ist daher leichter zu verstehen, bringt man es mit der zugrundliegenden Hardwarekomponente , also der GeForce3-FPU, in Bezug. Mit Abbildung 30 habe ich dies versucht: Sie zeigt den Pixel Shader in der DX8-Rendering-Pipeline, wobei ich die Funktionalität dem Instruktionsfluss entsprechend in Texturoperationen und arithmetische Operationen unterteilt habe. Vorbild für diese Darstellung ist Abbildung 21, welche die GeForce3-FPU im Rahmen der sie umgebenden Hardwarekomponenten zeigt.

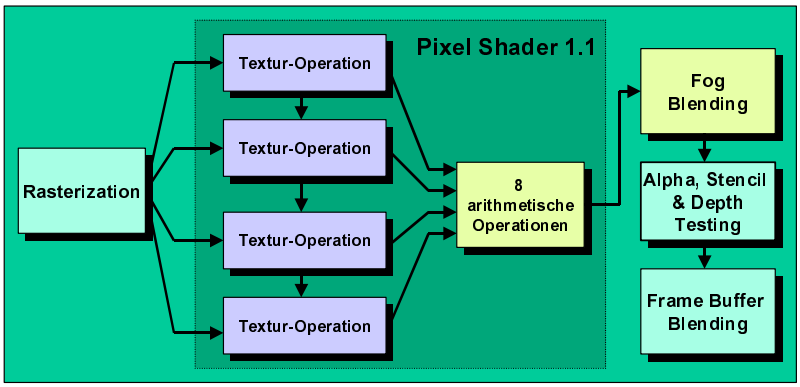



**Abbildung 30: Der Pixel Shader 1.1 in der DX8-Rendering-Pipeline**

In einem Pixel Shader 1.1 sind vier Texturinstruktionen und 8 arithmetische Instruktionen erlaubt. Der Zusammenhang zur Hardware ist offensichtlich: Die Texturinstruktionen steuern die vier *Texture Shader*, und die 8 arithmetischen Instruktionen steuern die 8 *General Combiner* der FPU. Der *Final Combiner* (Fog/Specular-Combiner) der FPU, dessen Aufgabe u.a. darin besteht, der berechneten Fragmentfarbe Nebel hinzuzufügen, wird nicht vom Pixel Shader, sondern von der DX8-*Fog-Blending*-Komponente gesteuert (vgl. Abbildung 25). Dies erklärt auch, wieso, anders als bei der GeForce3-FPU, weder Nebelfaktor noch Nebelfarbe als Eingangsattribute auftauchen (vgl. Abbildung 28).

### *4.2.2.1 Pixel Shader 1.1 - Texturinstruktionen*

***Syntax und Instruktionssatz***

Die vier Komponenten eines Registers werden der Reihenfolge nach entweder mit den Buchstaben *x*, *y*, *z* und *w* oder mit den Buchstaben *r*, *g*, *b* und *a* benannt.
Die meisten Texturoperationen sind nach folgendem Schema aufgebaut:

```
Befehlsname Ziel [, Quelle]
```

`Ziel` und `Quelle` dürfen hierbei ausschließlich Texturregister sein. Einzige Ausnahmen sind die Operation `texkill`, die kein Zielregister sondern nur ein Quellregister als Parameter hat, sowie die Operation `texm3x3vspec`, die zusätzlich als dritten Parameter ein *Konstanten-Register* erwartet.

Eine Texturinstruktion hat als Datenquellen für ihre Berechnungen die *Texturdaten einer Textur*, die mit dieser Textur verknüpften *Texturkoordinaten* sowie die Inhalte von *Textur*- und *Konstanten-Registern* (vgl. Abbildung 28).
Als Attribute einer Texturinstruktion treten aber wie gesagt nur Textur- und Konstanten-Register auf. Es stellt sich die Frage, wie dann einer Instruktion mitgeteilt wird auf welcher Textur und mit welchen Texturkoordinaten sie operieren soll. Diese Zuweisung geschieht indirekt[11]:
Jedes der vier Texturregister, t0 – t3, ist mit einer sogenannten *Texture Stage* (Texturstufe) verknüpft (vgl. Abschnitt 2.3.1: Multi-Texturing). Hierbei wird Register t*i* (mit $i \in \{0,1,2,3\}$) assoziiert mit Texture Stage *i*. Und *Textur Stage i* wiederum ist verknüpft mit einer bestimmten Textur – ich nenne diese im Folgenden T*i* – und den zugehörigen Texturkoordinaten, die ich im Folgenden mit TK*i* bezeichnen werde. Das Ziel-Register hat nun eine doppelte Funktion: Einerseits legt es fest, auf welche Textur sich eine Operation bezieht, andererseits wird im Zielregister letztendlich das Ergebnis der Operation abgelegt.

Die enge Beziehung zwischen Pixel-Shader-Version.1.1 und GeForce3-FPU zeigt sich auch im Aufbau und in der Funktionalität der Texturinstruktionen. Statt einer alle Texturinstruktionen im Einzelnen zu beschreiben, ist es daher einfacher deren Funktionalität durch eine Gegenüberstellung mit den bereits besprochenen Texture-Shader-Programmen der GeForce3-FPU zu erklären (siehe Tabelle 22, S. 172).

Wie bei den Vertex-Shader-Programmen gibt es einige zusammengesetzte Befehle, die nur im Block funktionieren. Für Zweier-Blöcke übernimmt `texm3x2pad` die Funktionalität von `DOT_PRODUCT_NV`, für Dreier-Blöcke `texm3x3pad`. Die Unterscheidung zwischen 1D-, 2D-, 3D- und Rectangle-Texturen sowie Cube Maps wird außerhalb des Pixel-Shader-Programms bei der Definition der Textur festgelegt. Die beiden Texture-Shader-Programme

[11] Diese Assoziation wird nicht innerhalb des Pixel-Shader-Programms festgelegt, sondern außerhalb (vgl. Abschnitt 5.2.1.6).

`_DIFFUSE_CUBE_MAP_NV` und `NONE` haben keine Entsprechung im Pixel-Shader-Progammmodell.

In Pixel-Shader-Version 1.1 gibt es einen Modifikator, den man auf die Quellregister einer Texturinstruktion anwenden kann: „Skalieren mit Vorzeichen“ („Signed Scaling“):

```
Quellregister_bx2
```

„`_bx2`“ subtrahiert ½ von jeder Kompontente und multipliziert sie anschließend mit 2. Der vorher auf den Wertebereich [0,1] beschränkte Inhalt des Quellregisters erstreckt sich dadurch für die Dauer der Operation über den Bereich [-1,1].

#### *4.2.2.2 Pixel Shader 1.1 - Arithmetische Instruktionen*

Die arithmetischen Instruktionen sind nach folgendem Schema aufgebaut:

```
Befehlsname Ziel [, Quelle1 [,Quelle2 ,[Quelle3]]]
```

`Ziel` kann Register `r0` oder `r1` oder eines der vier Texturregister `t0-t3` sein. Als Quellen stehen alle Registertypen zur Verfügung.
Die arithmetischen Instruktionen steuern die 8 General Combiner der GeForce3. An dieser Stelle hat man sich aber weiter von der zugrundeliegenden Hardware-Architektur getrennt als irgendwo sonst. Die GeForce3-Combiner berechnen jeweils aus bis zu vier Eingabevektoren drei Ausgabevektoren. Die arithmetischen Instruktionen eines Pixel Shaders berechnen grundsätzlich immer nur eine Ausgabe (vgl. Tabelle 25). Man überzeuge sich aber davon, dass sich alle Operationen leicht auf die Funktionalität der General Combiner abbilden lassen. Als Beispiel hierfür zeige ich, wie sich der Befehl `lrp` mit einem *General Combiner* realisieren lässt:

**Beispiel 10:** Der zu realisierende Ausdruck ist: `z = q1 * q2 + (1 - q1) * q3`. Seien `A:=q1`, `B:=q2`, `C:=q1` und `D:=q3` die vier Eingabevektoren des Combiners. Durch die entsprechende Eingabeabbildung (siehe Tabelle 10) transformiert man `C` in `C':=1-q1`. Anschließend setzt man:

`OUT1 = A*B`,

`OUT2 = C'*D`,

`OUT3 = A*B+C'*D`, und anschließend `z:= OUT3`.

Die Eigenschaft der General Combiner, Alpha und RGB-Anteile separat zu berechnen, so dass für beide Anteile verschiedene Funktionen verwendet werden können, wurde im Pixel-Shader-Programmmodell durch die Möglichkeit der „Instruktionspaarung“ (Instruction Pairing) berücksichtigt. Die eine Instruktion des Paares berechnet den *RGB-Anteil*, was durch den Suffix „`.rgb`“ am Zielregister angezeigt wird, die andere Instruktion berechnet den *Alpha-Anteil*, angezeigt durch den Suffix „`.a`“ am Zielregister. Ein „`+`“-Zeichen verknüpft die beiden Instruktionen zum Instruktionspaar.

**Beispiel 11:**
```
mul r1.rgb, r0, r0
+add r1.a, r0, r0
```

### *4.2.2.3 Lesemasken, Schreibmasken und Modifikatoren*

Auch die Eingabe- und Ausgabeabbildungen der Register Combiner haben im Pixel-Shader-Programmmodell ihre Äquivalente – die Instruktions-Modifikatoren (Instruction Modifiers) (vgl. Tabelle 26). Zwar werden die Modifikatoren der Register Combiner, wie im

Beispiel 10 gezeigt, bereits implizit zur Realisierung des Instruktionssatzes genutzt, aber es gibt auch die Möglichkeit, sie explizit anzugeben. Hinzu kommen Lese- und Schreibmasken (vgl. Tabelle 27). Letztere habe ich schon im Zusammenhang mit dem Prinzip der „Instruktionspaarung“ (Beispiel 11) vorgestellt.

### 4.2.3 Die Verbindung der Vertex und Pixel Shader zum restlichen API

#### *4.2.3.1 Die Einbindung von Vertex- und Pixel-Shader-Code in die DX8-API*

Bevor ein Vertex- oder Pixel-Shader-Programm genutzt werden kann, muss es zunächst kompiliert werden. DX8 verfügt über integrierte Compiler, die dies erledigen. Hierzu muss der Programmtext entweder als ein Array vom Typ „char“ vorliegen oder in einer Text-Datei abgelegt sein. Es gibt aber auch die Möglichkeit, einen externen Compiler zu verwenden und den bereits kompilierten Code in das DX8-Interface einzubinden [Micr02b].
Letztere Variante bietet die Möglichkeit, Shader-Programmmodelle zu entwickeln, die von der DX8-Syntax abweichen. Beispielsweise stellt NVIDIA im „Developer“-Bereich ihrer Homepage einen solchen Compiler zur Verfügung [Nvid02e, „NVASM“], und ich habe diesen zum Erstellen meiner Demonstrationsprogramme verwendet, da er eine etwas benutzerfreundliche Entwicklungsumgebung bietet. Denn er erlaubt das Ersetzen von Registerbezeichnungen durch benutzerdefinierte Ausdrücke, was die Lesbarkeit des Programmcodes erleichtern kann.

Mit Hilfe des Befehls „`CreateVertexShader`“ bzw. „`CreatePixelShader`“ wird aus dem kompilierten Shader-Programm ein Vertex- bzw. Pixel-Shader-Interface erzeugt, und der Programmierer erhält eine Referenz (ein Handle) auf dieses Interface, über die er die Aktivitäten des Shaders steuern kann.

#### *4.2.3.2 Die Steuerung eines DX8-Vertex-Shader-Interfaces*

Per-Vertex-Daten sind üblicherweise in einem speziellen Speicher-Konstrukt, dem sogenannten „Vertex Buffer“, gespeichert. Welches Format die darin gespeicherten Per-Vertex-Daten haben, legt der Programmierer vor dem Füllen des Vertex Buffers fest. Ein Beispiel für ein solches Per-Vertex-Datenformat könnte sein:

1. dreidimensionaler Positionsvektor,
2. dreidimensionale Normale und
3. zweidimensionaler Texturkoordinatenvektor.

Dem Vertex Shader muss nun dieses Datenformat mitgeteilt werden, damit die Vertex-Register bei der Verarbeitung eines Vertex jeweils mit den gewünschten Per-Vertex-Daten gefüllt werden können. Dies geschieht mittels der sogenannten *Vertex-Shader-Deklaration*. Diese Deklaration wird der Instruktion „`CreateVertexShader`“ als Parameter übergeben.[12]
Für das oben angegebene Beispiel-Format könnte eine solche Deklaration wie folgt aussehen:

```
DWORD dwDecl[] = {
D3DVSD_STREAM(0),
// Erstes Attribut
D3DVSD_REG (0, D3DVSDT_FLOAT4),
// Zweites Attribut
D3DVSD_REG (1, D3DVSDT_FLOAT3),
// Drittes Attribut
D3DVSD_REG (2, D3DVSDT_FLOAT2),
D3DVSD_END()
};
```

Entscheidend ist der Ausdruck `D3DVSD_REG`: Der erste Wert innerhalb der Klammer gibt an, welches der Vertex-Register das jeweilige Attribut enthalten wird – für den Wert „0“ ist dies beispielweise das Register `v0`. Der zweite Wert gibt an, welches Format dieser Wert im Vertex Buffer hat – im Fall des ersten

[12] Anmerkung: Es ist also nicht möglich, die bijektive Abbildung zwischen Vertex-Attributen und Vertex-Registern nachträglich zu ändern. Es muss dann ein neues Vertex-Shader-Interface erzeugt werden.

Attributes ist dies, meinem Beispiel-Format entsprechend, ein vierdimensionaler aus Floats bestehender Vektor.

Es gibt zwei Möglichkeiten, um uniforme Attribute in den Konstanten-Registern des Vertex Shaders abzulegen:

1. Konstanten-Register können innerhalb der *Vertex-Shader-Deklaration* (siehe vorheriger Abschnitt) mit Werten belegt werden. Die genaue Syntax dieser Methode werde ich an dieser Stelle aussparen und verweise auf [Micr02b].

2. Die Zuweisung kann durch den Befehl „`setVertexShaderConstant`“ erfolgen. Der Befehl erhält zwei Parameter: Der erste gibt die Nummer des zu überschreibenden Registers an, der zweite den Wert, den es speichern soll [Micr02b]. Der eindeutige Vorteil dieser Variante ist, dass der Inhalt eines Konstanten-Registers von Frame zu Frame geändert werden kann.

**Beispiel 12:** Das Konstanten-Register c0 soll mit dem Wert (1, 1, 1, 1) belegt werden.

```
// Wert definieren
D3DXVECTOR4 value (1.0f, 1.0f, 1.0f, 1.0f);
// Konstanten-Register c0 mit Wert überschreiben
setVertexShaderConstant( 0, &value );
```

Die Aktivierung von Shadern geschieht über den Befehl „`setVertexShader`, wobei als Attribut die Referenz auf den zu aktivierenden Shader übergeben werden muss [Micr02b]. Gibt man als Parameter stattdessen eine Referenz auf ein *Per-Vertex-Datenformat* an, dann wird der Standard-Shader – die aus DX7 bekannte T&L-Komponente – aktiviert [Micr02b].

#### *4.2.3.3 Die Steuerung eines DX8-Pixel-Shader-Interfaces*

Die einzigen Register, die von außerhalb des Pixel Shaders gesetzt werden können, sind die Konstanten-Register. Die übrigen Eingabe-Register des Shaders werden mit den beim Triangle Setup interpolierten Fragment-Attributen gefüllt.

Die Wertzuweisung erfolgt über den Befehl „`setPixelShaderConstant`“. Analog zum Befehl „`setVertexShaderConstant`“ erwartet die Instruktion zwei Parameter, wobei der erste Parameter die Nummer des zu überschreibenden Registers angibt, der zweite den Wert, den dieses speichern soll [Micr02b].

Die Aktivierung eines Pixel-Shaders geschieht über den Befehl „`setPixelShader`“, wobei als Attribut die Referenz auf den zu aktivierenden Shader übergeben werden muss [Micr02b]. Gibt man als Parameter den Wert „NULL“ an, wird die aus DX7 bekannte Multi-Texturing-Komponente aktiviert [Micr02b].

## 4.3 Die Themenbereiche im Überblick

Ich stelle nun in kombinierter Form die Ergebnisse meiner praktischen Arbeit und meiner themenbezogenen Studie über die Vorteile der neuen Grafikhardware vor. Eine auf diesen Erfahrungen basierende, zusammenfassende Bewertung der Benutzbarkeit und Funktionalität folgt in Kapitel 5.

Die Aufgaben der Vertex Shader 1.1 und Pixel Shader 1.1. lassen sich in drei disjunkte Bereiche unterteilen:

- Transformation und Deformation der Geometrie (Vertex Shader)
- Schattierung (Vertex und Pixel Shader)
- Clipping (Vertex Shader und Pixel Shader)

### 4.3.1 Transformation und Deformation der Geometrie

Die Transformation der Geomerie in das Projektions-Koordinatensystem ist eine Standardaufgabe, die jedes Vertex-Shader-Programm leisten muss. Darüber hinaus können auch noch weitere Transformationen eingesetzt werden, um beispielsweise die zu verarbeitende Repräsentation zu deformieren. Diese Transformation können von Frame zu Frame variiert werden, um Animationen zu erzeugen.

Eine solche Variation wird erzielt, indem ein zeit-abhängiger Wert in den Konstanten-Registern abgelegt wird, auf dessen Basis der Vertex Shader seine Transformationsberechnungen durchführt.
Die Möglichkeiten dieser vertex-shader-basierten Animationen sind allerdings durch die folgende Restriktion eingeschränkt: Ein interaktives Verhalten zwischen Vertices innerhalb eines Vertex Shaders zu realisieren ist nicht möglich. Schuld daran ist der lokale Charakter der Verarbeitung: Jeder Vertex wird separat verarbeitet und kann nicht auf die Attribute anderer Vertices zugreifen. Dennoch gibt es eine Vielzahl verbreiteter Techniken, die mit Vertex Shadern realisiert werden können:

- **Lineare globale Transformation.** Die „Welt-Matrix“, welche die Position, Größe und Ausrichtung des Modells in der Szene bestimmt, kann Frame für Frame variiert werden. Diese Art der Animation wurde auch schon von früheren Hardware-Generationen unterstützt.
- **Nicht-lineare globale Deformation:** Diese Animationstechnik geht auf A. H. Barr [Barr84] zurück. Statt für jeden Vertex dieselbe Transformation durchzuführen, wird der Grad der Transformation von Vertex zu Vertex in Abhängigkeit von den jeweiligen Vertex-Attributen variiert. Dies führt zu einer Deformation des Objektes.
- **Schlüsselbild-Interpolation** (auch „Tweening“ oder „Morphing“ genannt): Statt alle Bilder einer Animation wie bei einem Zeichentrickfilm von Hand zu erzeugen, wird zwischen einer Anzahl an Schlüsselbildern interpoliert. Der Animateur legt diese Schlüsselbilder fest, und ein Algorithmus erzeugt durch Interpolation die dazwischen liegenden Bilder.
- **Skelett-Animation** (auch „Matrix Skinning“ genannt): Eine Technik, die für die Animation von Figuren mit mehreren Gliedmaßen geeignet ist. Für die zu animierende Figur werden Rotations-Matrizen und Per-Vertex-Gewichte definiert. Die Matrizen repräsentieren Gelenke, und die Per-Vertex-Gewichte geben den Einfluss an, den ein bestimmtes Gelenk auf den Vertex hat. Der Vertex Shader berechnet die Position des Vertices durch gewichtete Summation der verschiedenen Matrix-Rotationen.

- **Prozedurale Deformation:** Ich werde den Einsatz von prozeduraler Deformation anhand der Simulationen von Wasser und Gras demonstrieren. Auch zur Simulation von Stoffen oder der Bewegungsabläufe verschiedener Tiere (z.B. einer Schlange [Watt92, S.426-427] oder Qualle [NVID02f]) kann diese Technik eingesetzt werden.
- **Betrachter-abhängige Deformation:** [Mart00] beschreibt Verfahren und Anwendungsgebiete für Deformationen, die in Abhängigkeit von Position und Blickwinkel des Betrachters berechnet werden. Diese Verfahren können mit der aktuellen Funktionalität der VPU problemlos auf dem Grafikchip ausgeführt werden.

### 4.3.2 Schattieren

Anders als die Transformation, die nur eines Vertex-Shader-Programmes bedarf, ist das Schattieren eine Aufgabe, für die grundsätzlich beide Einheiten benötigt werden. Man kann Shading mit Vertex und Pixel Shadern vereinfacht als zweistufigen Prozess betrachten: Der Vertex Shader berechnet in der ersten Stufe alle die Schattierung betreffenden Vertex-Attribute. Aus diesen werden dann die Fragment-Attribute interpoliert, welche der Pixel Shader in der zweiten Stufe zur Berechnung der Fragmentfarben verwendet. Neben den programmierbaren Einheiten haben aber auch zahlreiche andere Interface-Komponenten einen Einfluss auf die Schattierung. Beispielsweise ist es entscheidend, ob Backface-Culling, also die Entfernung der Dreiecksrückseiten, aktiviert ist. Die Realisierung einer bestimmten Schattierung ist also das Ergebnis des Zusammenspiels zahlreicher GPU-Komponenten.

Die neue Grafikchip-Generation bietet, sowohl im photorealistischen wie im nicht-photorealistischen Bereich, mehr Möglichkeiten als ihre Vorgänger. Ich werde eine Auswahl davon in den folgenden Abschnitten vorstellen.

Einige dieser erweiterten Möglichkeiten sind jedoch nicht unwesentlichen Restriktionen unterworfen: Beispielsweise kann, bei einer Beleuchtungsrechnung per Pixel, in den meisten Fällen nur eine Lichtquelle pro Pipeline-Durchlauf berücksichtigt werden.

### 4.3.3 Clipping

Zu guter Letzt können mit Vertex und Pixel Shadern auch Clipping-Ebenen definiert werden. Mit Clipping ist hierbei *nicht* das Zuschneiden der Geometrie auf den Sichtbereich gemeint (*View Frustrum Clipping*, siehe S.12), denn diese Aufgabe wird völlig unabhängig von den DX8-Shadern gesteuert. Der Programmierer kann aber darüber hinaus zusätzliche Clipping Ebenen definieren (User Clipping Planes). Ermöglicht wird diese Funktionalität im Wesentlichen durch die Pixel-Shader-Instruktion `texkill`.

Mit diesen zusätzlichen Clipping Ebenen kann der Pogrammierer Objekte gezielt zuzuschneiden. Diese Technik kann beispielsweise bei der Erzeugung von planaren Reflektion nützlich sein [Möll99, S.1661-165].

Ein Versuch, alle Effekte und Techniken, die mit den neuen Hardware-Komponenten in diesen Aufgabenbereichen realisierbar sind, vorzustellen, würde den Rahmen dieser Arbeit sprengen. Es geht mir in den nun folgenden Abschnitten vielmehr darum, anhand einer Auswahl aus den angesprochenen Themengebieten einen Überblick über die Leistungsfähigkeit zu geben.

## 4.4 Realistischere lokale und globale Reflektionen

### 4.4.1 Anisotropisches Reflektionsverhalten

Zur Simulation der Interaktion zwischen Lichtquellen und Oberflächen auf Vertex-Ebene stand bei der vorherigen GPU-Generation ausschließlich eine Variante des Phong-Beleuchtungsmodells [Phon75] zur Verfügung. VPU und FPU erlauben es dem Programmierer, auch alternative lokale Beleuchtungsmodelle zu verwenden, ohne auf Hardware-Beschleunigung verzichten zu müssen.

Der interessanteste Aspekt für photorealistische Bildgenerierung ist meines Erachtens die Fähigkeit, in vereinfachter Form *physikalisch*-basierte Beleuchtungsmodelle realisieren zu können, wie sie beispielweise von Blinn [Blin77], Cook und Torrance [Cook82], Kajiya [Kaji85], Cabral et al. [Cabr87], oder Hanrahan und Kreuger [Hanr93] entwickelt wurden.

Der Vorteil dieser physikalisch-basierten Modelle ist, dass sie zur Simulation einer Vielzahl verschiedener Materialien geeignet sind, die vorher nur unzureichend repräsentiert werden konnten – denn das Phong-Modell produziert grundsätzlich ein plastikartiges Erscheinungsbild ([Watt00], S. 219]. Das Modell von Cook und Torrance eignet sich beispielsweise gut zur Simulation metallischer Oberflächen [Watt00, S. 219]. Abbildung 31 zeigt eine Teekanne, die mittels einer vereinfachten Form dieses Modells auf einem GeForce3-Chip in Echtzeit schattiert wurde.

**Abbildung 31: Schattierung auf der GeForce3-GPU mittels einer Variante des Cook-Torrance-Modells kombiniert mit Environment-Mapping [NVID02e].**

[Cabr87] und [Hanr93] beschreiben *anisotropische Beleuchtungsmodelle*, die sich besonders zur Simulation von Oberflächen eignen, die über Mikrostrukturen mit ungleichförmiger Ausrichtung verfügen. Anisotropische Modelle werden der Art und Weise, wie beispielsweise das Licht von aufgerautem bzw. geschliffenem Metall oder Haarsträhnen reflektiert wird, gut gerecht. Und auch diese Modelle sind in vereinfachter Form durch den Einsatz von Texturen als Funktionstabellen [Heid99][McCo01] auf den neuen GPUs realisierbar.

Um diese Fähigkeiten zu demonstrieren, habe ich einen entsprechenden Shader programmiert. Abbildung 32 zeigt ein mit diesem Shader schattiertes Objekt. Die Realisierung des Effektes ist denkbar einfach: Der Vertex Shader berechnet für jeden Vertex die diffuse und spekulare Komponente. Die Ergebnisse beider Berechungen werden als *u*- und *v*-Koordinaten in eines der Texturkoordinatenregister abgelegt. Die FPU verwendet diese Koordinaten, um

auf eine Textur zuzugreifen. Diese Textur enthält eine Funktionstabelle der BRDF (Bidirectional Reflectance Distribution Function) des zu simulierenden Materials, und man erhält auf diese Weise den Intensitätswert der Reflektion.

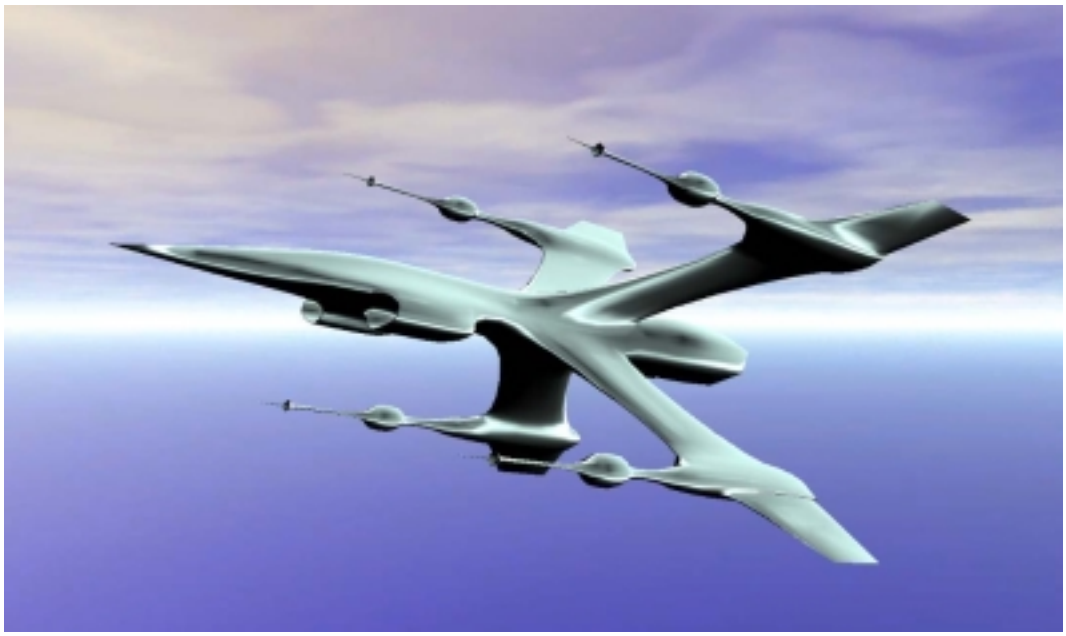


**Abbildung 32: Der Anisotropic Shader im Einsatz (Screenshot)**

### 4.4.2 Bump Environment Mapping

#### *4.4.2.1 Beleuchtungsrechnung per Fragment*

Statt per Vertex können mit der neuen Hardware und DX8-Shadern Beleuchtungsrechnungen auch per Fragment durchgeführt werden. So kann beispielsweise Phongs Beleuchtungsmodell auf Fragment-Ebene berechnet werden. Basis für diese Berechnung bilden aus Vertex-Normalen interpolierte Fragment-Normalen (Phong Shading [Phon75]). Die im vorherigen Punkt angesprochenen physikalisch-basierten Beleuchtungsmodelle lassen sich auf dieser Ebene aber nicht realisieren.

Ein besonderer Aspekt der Beleuchtungsrechung auf Fragment-Ebene ist die Möglichkeit, Bump Mapping [Blin78] durchzuführen. Frühe Hardware-

Generationen erlaubten dagegen nur eine sehr einfache und qualitativ schlechtere Bump-Mapping-Variante [Micr02a].

Die Interaktion zwischen Objekten kann im Allgemeinen nur mit globalen Beleuchtungsmodellen simuliert werden. Diese sind für Echtzeit-Anwendungen aber zu aufwendig. Einige optische Effekte globaler Natur können aber in Echtzeit imitiert werden.
Ein gutes Beispiel hierfür ist Bump Environment Mapping, also die Kombination von Bump Mapping [Blin78] und Environment Mapping [Blin76][Gree86]. Mit der letzten Hardware-Generation war es bereits möglich spiegelnde Oberflächen durch den Einsatz von Environment Mapping [Blin76][Gree86] nachzuahmen. Jetzt lässt sich dieser Effekt mit Bump Mapping kombinieren, so dass unebene, perfekt spiegelnde Oberflächen simuliert werden können. Auf dieselbe Weise wie Spiegelung kann mit der neuen Hardware auch Lichtbrechung simuliert werden. Durch Kombination mit anderen Schattierungsmethoden (im Allgemeinen durch einen zweiten Pipeline-Durchlauf) lassen sich mit diesen Effekten verschiedenste Materialeigenschaften nachahmen.
Ich habe einen solchen Effekt implementiert (vgl. Abbildung 33). Abbildung 34 zeigt, wie durch Bump Environment Mapping multipliziert mit einem Orange-Ton ein Goldeffekt erzeugt werden kann.

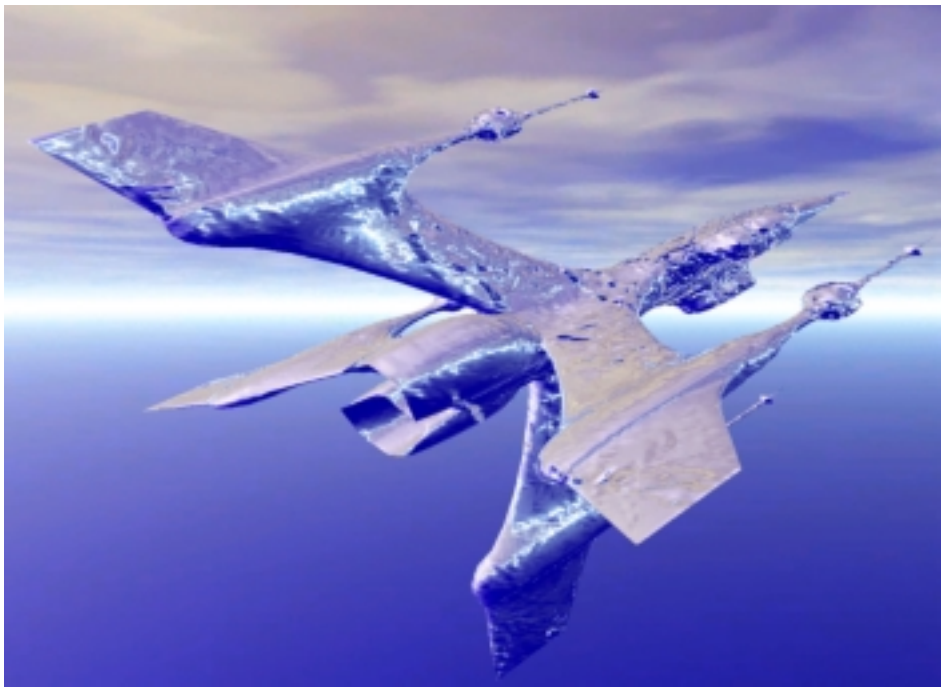


**Abbildung 33: Bump Environment Mapping (Screenshot)**

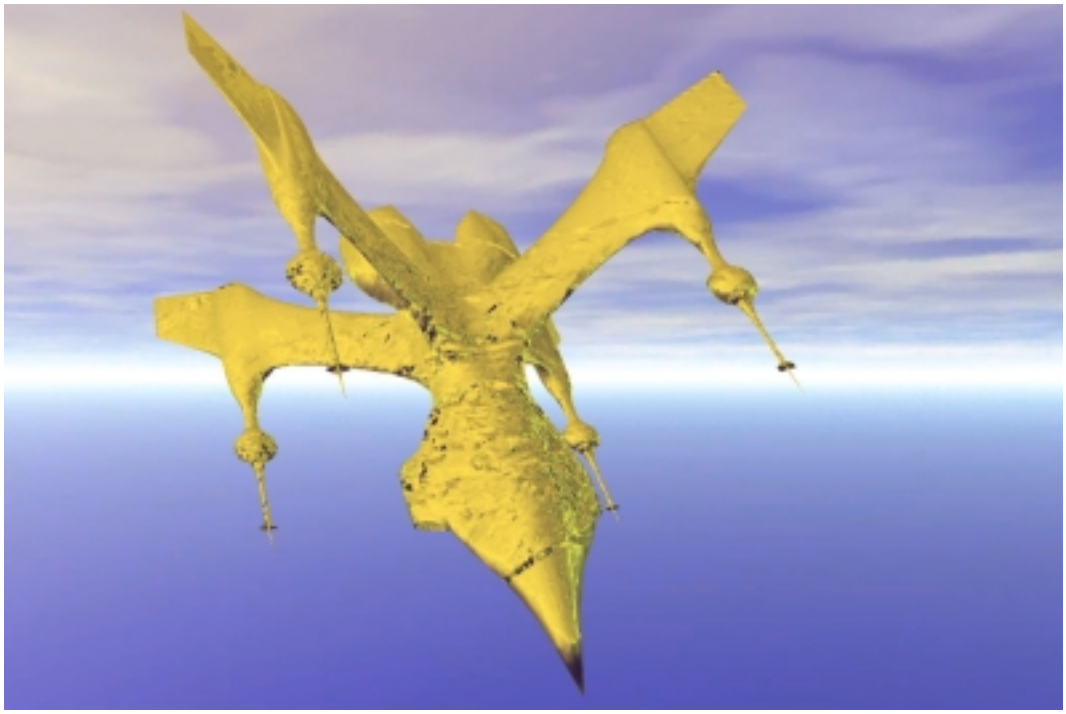


**Abbildung 34: Bump Environment Mapping multipliziert mit einem Orangeton zur Erzeugung eines Gold-Effektes**

#### *4.4.2.2 Realisierung von Bump Environment Mapping*

Für das Bump Environment Mapping sind zwei Texturen nötig: eine Normal Map und eine Environment Map.
In den RBG-Komponenten der Normal-Map-Elemente sind x-,y-, und z-Koordinaten von Oberflächennormalen gespeichert. Diese repräsentieren die Beschaffenheit der zu schattierenden Oberfläche. In der Environment Map ist die zu reflektierende Umgebung enthalten.
Ich habe in allen meinen Demonstratoren einen mit einer kubischen Textur (Cube Map) versehenen Würfel zur Simulierung der Umwelt verwendet. Der Betrachter befindet sich grundsätzlich in der Mitte dieses Würfels. Als Environment Map konnte ich daher die Cube Map dieses Würfels verwenden.

Bump Environment Mapping wird im Wesentlichen innerhalb des Pixel Shaders.verwirklicht. In Version 1.1 ist hierfür der folgende Funktionsblock verantwortlich (vgl. Tabelle 22):

1.) Lies Normale aus Normal Map (Textur `t0`)

```
tex t0
```

2.) Überführe Normale in den Tangenteraum des Fragmentes, reflektiere an der transformierten Normale den Augen-Vektor `(t1.w, t2.w, t3.w)` und nutze die Koordinaten des reflektierten Vektors als Texturkoordinaten für einen Zugriff auf die Environment Map (Textur `t3`).

```
texm3x3pad t1, t0
texm3x3pad t2, t0
texm3x3vspec t3, t0
```

Der zweite Funktionsblock spricht auf dem Geforce3-Chip das entsprechende Texture-Shader-Programm an.
Die zur Überführung der Normalen in den Tangentenraum benötigten Basisvektoren – Normale, Tangente und Binormale – werden vom Vertex Shader

in den entsprechenden Texturkoordinatenregistern (`oT1`, `oT2` und `oT3`) abgelegt. Hierzu müssen aber mindestens zwei dieser Vektoren als Per-Vertex-Attribute vorliegen. Den dritten Basisvektor kann die VPU per Kreuzprodukt aus den anderen beiden Vektoren berechnen.

#### *4.4.2.3 Probleme und mögliche Verbesserungen*

Mit diesem Beispiel habe ich die Verwendung der vordefinierten Texture-Shader-Programme (vgl. Tabelle 18-Tabelle 20) demonstriert. Der eindeutige Nachteil dieser Programme ist ihre Inflexibilität. Der Programmierer kann nichts an ihrer Arbeitsweise ändern. Desweitern werden für den Effekt alle vier erlaubten Texturinstruktionen benötigt, weshalb die Verwendung von weiteren Texturen nur mit zusätzlichen Passes möglich ist.
Dies ist insbesondere für einfaches Bump Mapping, das ebenfalls über vier Texturinstruktionen realisiert wird, problematisch. Denn die Bump-Mapping-Funktion berücksichtigt immer nur eine Lichtquelle. Bump Mapping mit mehreren Lichtquellen ist daher nur mit zusätzlichen Pipeline-Passes realisierbar.

## 4.5 Nicht-photorealistische Beleuchtungsmodelle

Die Flexibilisierung durch die Eigenschaft der Programmierbarkeit wirkt sich insbesondere auf den Einsatz von nicht-photorealistischen Schattierungsverfahren aus. Solche Verfahren mussten bisher weitestgehend auf der CPU ausgeführt werden, da die vorherige Generation, wie bereits erwähnt, nur Phong-Lightning unterstütze. Es wurden in den letzten Jahren zahlreiche, nicht-photrealistische Schattierungsverfahren für Echtzeit-Anwendungen entwickelt, die im Wesentlichen darauf basieren, dass zur Bestimmung der Schattierung auf Texturen zugegriffen wird, wobei die Texturkoordinaten mittels besonderer Formeln berechnet werden (siehe z.B. [Lake00] [Maju02] [Marko97]). Da die Berechnung der Texturkoordinaten nun auf der GPU ausgeführt werden kann – es besteht ja die Möglichkeit auf die Texturkoordinaten der Objekte zuzugreifen und sie durch Schreiben in die oT-Register zu setzen – profitieren insbesondere diese Modelle von den Fähigkeiten der programmierbaren VPU.

Um die Fähigkeiten im Bereich nicht-photorealistischer Bildgenerierung auszutesten und zu demonstrieren, habe ich mich, angeregt durch die Vorschläge in [Lake00], für die Erstellung eines Schraffur-Shaders und eines Cartoon-Shaders entschieden.

### 4.5.1 Schraffuren (Hatching)

Der Schraffur-Shader legt über das dargestellte Objekt eine Textur von Schraffuren verschiedener Helligkeiten, wobei die Helligkeit in Abhängigkeit von einer Beleuchtungsrechnung bestimmt wird. Der Effekt des Shaders ist in Abbildung 35 zu sehen.

**Abbildung 35: Der Hatching Shader im Einsatz (Screenshot)**

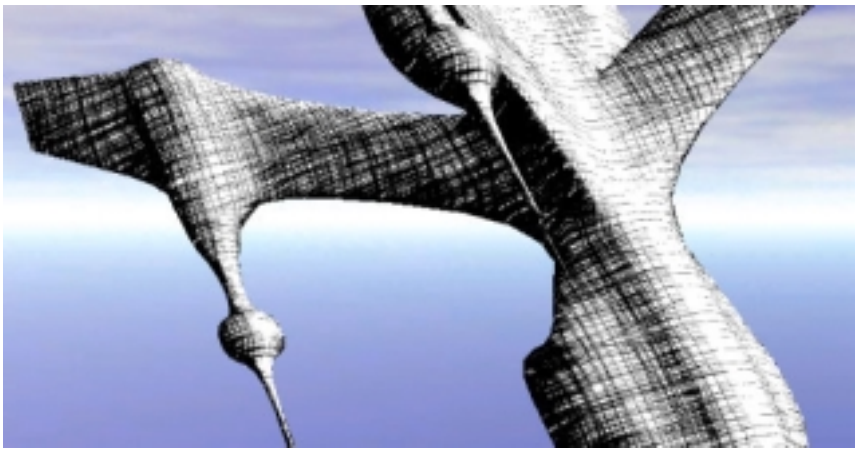


**Abbildung 36: Mit dem Hatching Shader schattiertes Objekt (Detail)**

#### *4.5.1.1 Realisierung*

Ich habe sechs Schraffuren verschiedener Strichdichte und -dicke erzeugt (vgl. Abbildung 37). Aus diesen Texturen setzt sich durch Übereinanderlegen die spätere Schattierung zusammen.

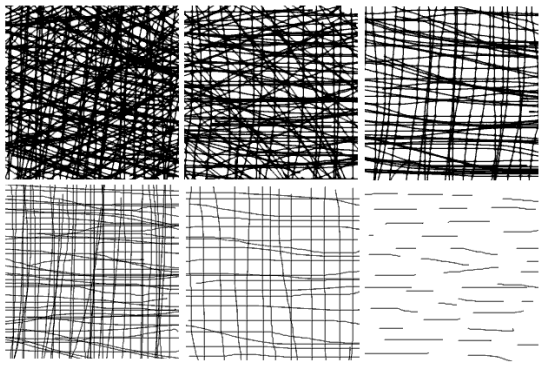


**Abbildung 37: Schraffur-Texturen verschiedener Intensität**

Pixel-Shader-Version 1.1 kann nur auf vier Texturen gleichzeitig zugreifen. Um möglichst wenige Texturstufen zu verbrauchen, verteilte ich die sechs Schraffuren über die RGB-Kanäle zweier Texture Maps (vgl. Abbildung 38). Dies war problemlos möglich, da ich ausschließlich monochrome Schraffuren verwendete und daher ein Kanal pro Textur zur Repräsentation aussreichte.

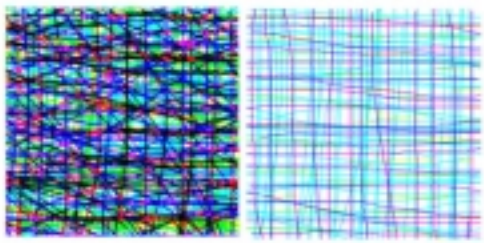


**Abbildung 38: Sechs Schraffuren verteilt auf die RGB-Kanäle von zwei Texture Maps**

Ein Vertex Shader berechnet für jeden Vertex $V$ die Lichtintensität $I_V$ unter dem Einfluss einer gerichteten Lichtquelle. Diese Beleuchtungsrechnung erfolgt mittels Lamberts Gesetz: $I_V = L_V \bullet N_V$. Hierbei sei $L_V$ die Richtung, in der vom Vertex aus betrachtet die Lichtquelle liegt, und $N_V$ sei die Vertex-Normale.

Anhand des so berechneten Intensitätswertes bestimmt der Vertex Shader das Mischungsverhältnis zwischen den verschiedenen Schraffuren. Ich habe mich hierbei für eine recht einfache Regel entschieden: Jeder Schraffur wird ein bestimmter Schwellenwert zugeordnet. Überschreitet $I_V$ diesen Schwellenwert, wird die entsprechende Schraffur nicht angezeigt, andernfalls ja. Eine andere Variante wäre es gewesen, die Schraffuren in Abhängigkeit von den Schwellenwerten stufenweise ein- bzw. auszublenden, aber meiner Meinung nach unterstreicht das plötzliche Auftauchen und Verschwinden von Linien die Grobheit der Technik.
Als Vertex-Shader-Code sieht der gerade beschriebene Vorgang wie folgt aus:

> `r0` enthalte die Normale $N_V$, `c0` den Lichtvektor $L_V$ und sei `c10` := (0.15, 0.3, 0.45, 0.0) und `c11` := (0.5, 0.65, 0.8, 0.0).

```
// Berechnung der Intensität
dp3    r0,          c0,    r0
// Bestimmung des Mischungsverhältnisses
slt    oD0.xyz,     r0,    c10
slt    oD1.xyz,     r0,    c11
```

Die im obigen Code-Ausschnitt verwendeten Grenzwerte (die Komponenten von `c10.xyz` und `c11.xyz`) beziehen sich ihrer Reihenfolge nach jeweils auf die dunkelste bis zur hellsten Schraffur. Die aus der Berechung erhaltenen Vektoren werden in die Ausgaberegistern `oD0` und `oD1` abgelegt und gelangen auf diese Weise in die Eingaberegister `v0` und `v1` des Pixel Shaders.
Im Pixel Shader wird nun mittels der berechneten Mischvektoren die endgültige Schattierung berechnet. Das Programm multipliziert jede Schraffur mit ihrem Gewicht (0 oder 1) und addiert anschließend die erhaltenen Werte. Mittels einer Vektorprodukt-Operation (`dp3`) lassen sich diese Berechnungen für jeweils drei Schraffuren parallel ausführen. Dies ist wegen der auf acht arithmetische Instruktionen beschränkten Programmlänge sogar eine Notwendigkeit:

> `v0.xyz` und `v1.xyz` enthalten die Gewichte, Texture Stage 0 `und` Texture Stage 1 seien mit den beiden in Abbildung 38 dargestellten Texture Maps verknüpft.

```
tex t0       // Hole Schraffur 1-3
tex t1       // Hole Schraffur 4-6
```

```
// Berechne Mischungsverhältnis
dp3_sat      r0,    1-t0,  v0
dp3_sat      r1,    1-t1,  v1
add_sat      r0,    r0,    r1

// Invertiere das Ergebnis
mov          r0,    1-r0
```

#### *4.5.1.2 Zusätzliche Informationen*

Damit der Benutzer die Feinheit der Schraffuren regulieren kann, skaliert der Vertex Shader die Texturkoordinaten mit inem *benutzerdefinierten* Faktor. Desweiteren wählte ich, um eine Schraffur ohne Sprünge zu erhalten, zum Texture Sampling den Mirror-Modus [Möll99, S.103], durch den die Textur beim Verlassen des Wertebereichs an der jeweiligen Kante gespiegelt wird.

### 4.5.2 Cartoon-Schattierung

Zeichner von Cartoons verringern bewusst den Anteil visueller Details, um die für die humoristische Botschaft bzw. die Geschichte entscheidenden Informationen zu betonen.

Der Verzicht auf Dreidimensionalität in der Schattierung ist hierfür ein mögliches Mittel. Helligkeitsabstufungen zwischen stärker und weniger beleuchteten Bereichen werden auf ein Minimum reduziert, nicht selten bis auf zwei Stufen: einem schattierten und einem beleuchteten Bereich. Die Grenze zwischen beiden Bereichen ist eine harte Kante, die den Konturen des Objektes bzw. der Figur folgt [Lake00].

Ich habe ein solches Schattierungsverfahren implementiert. Dieses unterscheidet drei Helligkeitsabstufungen. Ich achtete aber darauf, dass sich die Anzahl der Abstufungen durch marginale Veränderungen im Code beliebig verändern läßt. Abbildung 39 zeigt ein mit diesem Verfahren schattiertes Objekt

**Abbildung 39: Der Cartoon Shader im Einsatz (Screenshot)**

#### *4.5.2.1 Realisierung*

Die Unterscheidung zwischen helleren und dunkleren Bereichen geschieht wie bei der Schraffur-Schattierung durch Berechnung eines Intensitätswertes über Lamberts Gesetz: $I_V = L_V \bullet N_V$ (vgl. Abschnitt 4.5.1.1). Der so berechnete Intensitätswert wird als Texturkkoordinate für den Zugriff auf eine eindimensionale Textur verwendet.

Die eindimensionale Textur enthält die verschiedenen Helligkeitsabstufungen (vgl. Abbildung 40). Um den Inhalt dieser Textur flexibel kontrollieren zu können, wird sie nicht aus einer Datei geladen, sondern in einem Initialisierungsschritt generiert.

**Abbildung 40: Dreistufige Cartoon-Shading-Textur**

Die FPU greift mittels der interpolierten Intensitätskoordinate auf die Textur zu und kombiniert den so erhaltenen Wert durch Multiplikation mit der Materialfarbe des Objektes.

## 4.6 Imager Shader

Der *Imager Shader* ist ein Shader-Typ im RenderMan-Interface, der für die Bildbearbeitung auf Pixel-Ebene zuständig ist (siehe Abschnitt 2.2.4.4). In Analogie hierzu verwende ich für Bildbearbeitungs-Shader im DX8-Interface denselben Ausdruck.
Die erweiterte Funktionalität der FPU eignet sich gut für Bildbearbeitungprozesse (Image Processing). Das generierte Bild wird, statt in den Frame Buffer, in eine Textur geschrieben und gelangt so, in einem zweiten Pass, als Textur in die FPU. Auf diese Weise ist es möglich, verschiedene Linseneffekte zu erzeugen, wie Bewegungsunschärfe oder Tiefenunschärfe. Auch einfache Scharf- und Weichzeichnungsfilter können realisiert werden, oder, wie ich im nächsten Abschnitt zeigen werde, selbst nicht-photorealistische Effekte wie Schraffieren (Hatching) oder Punktieren (Stippling).

### 4.6.1 Realisierung von Imager Shadern

Zur Realisierung dieser Shader musste ich ein geeignetes Rahmenwerk schaffen:
Zunächst muss das Bild, das bearbeitet werden soll, generiert werden. In einem ersten Pipeline-Durchlauf wird also aus der Szene ein Bild erzeugt und dieses in einer Textur abgelegt. Der Imager Shader kommt nun im zweiten Pipeline-Durchlauf zum Einsatz: Man legt die Textur auf ein Rechteck, das parallel zur Sichtebene ausgerichtet ist und genau den Sichtbereich ausfüllt. Nun kann die FPU die Bildbearbeitung durchführen.

Ich beschreibe im Folgenden mehrere einfache, von mir realisierte Imager Shader. Weitere einfache und leicht realisierbare Shader wären Farb- oder Helligkeits-Invertierer sowie primitive Weichzeichnungs- und Scharfzeichnungsfilter (siehe [NVID02f]). Wegen der stark beschränkten Funktionalität der FPU sind komplexere Effekte nur mit zusäzlichen Passes möglich.

### 4.6.2 Gray Scale Imager Shader

**Abbildung 41: Grayscale Imager Shader kombiniert mit Phong Lightning (Screen Shot)**

Der folgende Pixel Shader transformiert ein Farb-Bild in ein Graustufen-Bild (vgl. Abbildung 41), indem er zunächst jeden Kanal mit 0.33 multipliziert – alle Kanäle werden also gleich gewichtet – und anschließend die Resultate zu einem einzelnen Intensitätswert addiert.

`t0` enthalte das zu bearbeitende Bild.

```
ps.1.1
def    c0,    0.33,  0.33,  0.33,  0.00

tex    t0     // Texture Sampling

// Multipliziere die RGB-Komponenten der Textur t0
// mit dem Gewicht 0.33 und addiere die erhaltenen Werte
dp3    r0,    t0,    c0
```

### 4.6.3 Monochrom Imager Shader

Ein in der Drucktechnik angewandtes Verfahren ist der monochrome Druck: Das Bild wird abstufungslos in unbedruckte und bedruckte Bereiche eingeteilt. Zur Realisierung einer solchen Schattierung berechne ich wie im letzten Abschnitt einen Grauwert *g*. Durch einen Vergleich mit einem benutzerdefinierten

Schwellenwert $s$ wird der Bildbereich in schwarze ($g \leq s$) und weiße ($g > s$) eingeteilt. Der folgende Shader führt dies aus:

c0 enthalte den Schwellenwert und t0 das zu bearbeitende Bild

```
ps.1.1
def c1,       0.0, 0.0, 0.0, 0.0
def c2, 1.0, 1.0, 1.0, 1.0

tex t0        // Texture Sampling

dp3_sat r0, t0, c0         // Berechnung des Helligkeitswertes
add_sat r0, r0, c3         // Setzen des Vergleichswertes

// Wenn Vergleich erfolgreich, dann “Weiß”, sonst “Schwarz”
cnd r0, r0.a, c2, c1
```

**Abbildung 42: Ein monochrom schattiertes Bild (Screenshot)**

### 4.6.4 Schraffur- und Woodcut-Imager-Shader

Ich habe in Abschnitt 4.5.1 bereits einen Schraffur-Shader vorgestellt. Die Linien jenes Shaders vollziehen die Form des schraffierten Objektes nach. Ich stelle nun einen von mir entwickelten Schraffur-Shader vor, der ein bereits generiertes Bild

schraffiert. Der Vorteil dieses Verfahrens ist, dass auch Texturen in der Schraffur berücksichtigt werden(vgl. Abbildung 43 und Abbildung 44).

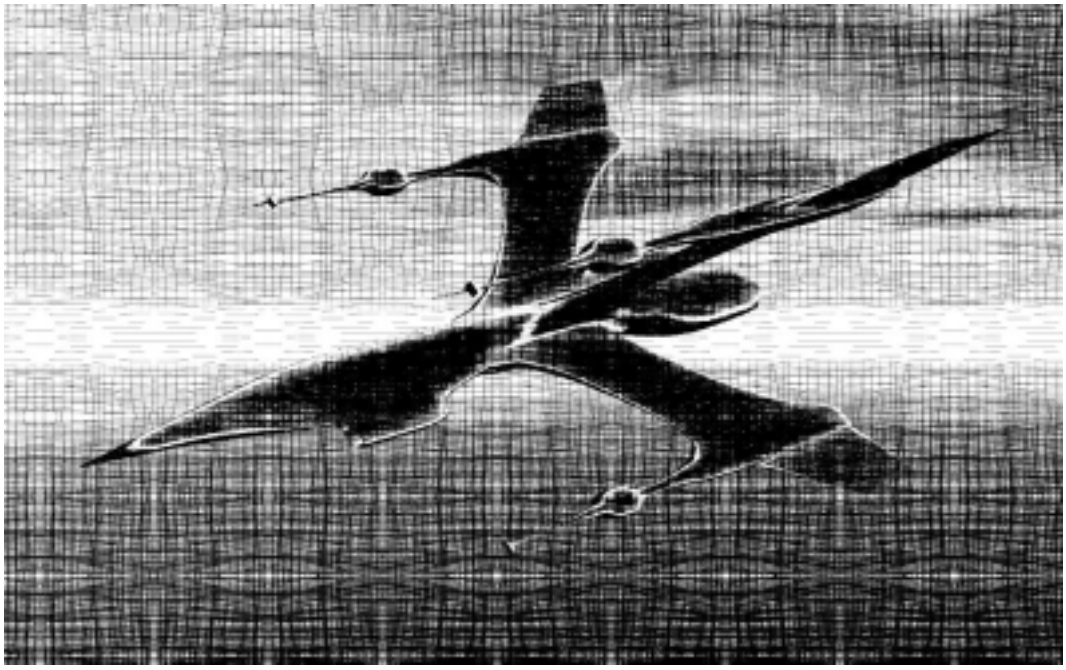


**Abbildung 43: Schraffur-Schattierung eines Bildes (1) (Screenshot)**

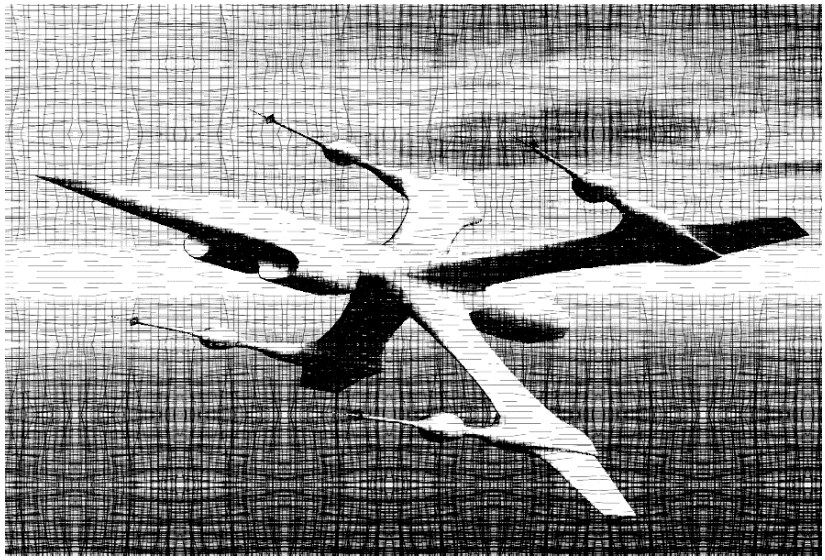


**Abbildung 44: Schraffur-Schattierung eines Bildes (2) (Screenshot)**

#### *4.6.4.1 Realisierung*

Zur Realisierung habe ich die in Abbildung 37 abgebildeten Schraffur-Texturen verwendet. Um die Schraffur ausreichend fein zu gestalten, werden die Texturen von der VPU durch einen benutzerdefinierten Wert skaliert.

Die FPU berechnet wie in den vorherigen beiden Beispielen einen Grauwert *g*. Dieser Grauwert wird dazu verwendet, zwischen den verschiedenen Schraffur-Texturen zu wählen. Was so einfach klingen mag, hat sich in der Durchführung als äußerst schwierig herausgestellt. Schuld daran ist die auf acht arithmetische Instruktionen begrenzte Programmlänge und die geringe Auswahl an Instruktionen und Modifikatoren. Wie im letzten Beispiel mit der cnd-Instruktion zu arbeiten, vebraucht zu viele Instruktionen. Eine Schraffur mit sechs Abstufungen ist mit dem Pixel Shader 1.1 nur möglich, wenn man weite Teile der Berechnungen parallel durchführt.

Das folgende Programm zeigt, wie ich dies durch die Verwendung der SIMD-Instruktionen `add` und `dp3` realisiert habe:

t0 enthalte das zu schraffierende Bild, t1 die Schraffuren 1-3 und t2 die Schraffuren 4-6

```
def c0,  0.33,  0.33,  0.33, 0.33
def c1, -0.15, -0.3,  -0.45, 0.0
def c2, -0.6,  -0.75, -0.9,  0.0

tex t0
tex t1
tex t2

dp3 r1, t0, c0      // Grauwert ermitteln

// Mischen der Schraffuren 1-3
add_sat      r0, r1, c1
dp3_sat      r0, 1-r0, 1-t1

// Mischen der Schraffuren 4-6
add_sat      r1, r1, c2
dp3_sat      r1, 1-r1, 1-t2

// Kombination der beiden Mischungen
add r0, r0, r1
```

Die Schraffuren werden bei steigendem Grauwert stufenweise und ihrer Helligkeit nach ausgeblendet. Das heißt, in ganz dunklen Bereichen ist eine Überlagerung aller Schraffuren zu sehen und in ganz hellen Bereichen nur die hellste Schraffur.

Dies gelingt für jeweils eine der Texturen durch Subtraktion der verschiedenen Grenzwerte (Register c1 und c2) vom Grauwert *g*. Durch diese Operation fallen einige Komponenten unter den Wert 0, die im Anschluss durch den Modifikator „_sat“ auf 0 gesetzt werden. Alle nun mit 0 belegten Komponenten repräsentieren die auszublendenden Schraffur. Der dp3-Befehl multipliziert nun die berechneten Komponenten mit den Schraffuren und addiert die so erhaltenen Werte.
Durch abschließende Addition werden die beiden Teilmischungen miteinander kombiniert.

#### *4.6.4.2 Woodcut Imager Shader*

Der obige Shader erzeugt Schraffuren mit Graustufen. Beseitigen lässt sich dieser Effekt durch einen abschließenden Vergleich:

r0 enthalte das Resulat des Schraffur-Shaders im letzten Abschnitt.

```
def c3, 0.0, 0.0, 0.0, 0.0, -0.5
def c4, 0.0, 0.0, 0.0, 0.0, 0.0
def c5, 1.0, 1.0, 1.0, 1.0, 1.0

// Wenn r0 > 0, dann „Schwarz“, sonst „Weiss“
mov r0.a, c3
cnd r0, r0.a, c4, c5
```

Wegen seiner Grobheit habe ich diese Variante Woodcut-Imager-Shader getauft.

#### *4.6.4.3 Mögliche Erweiterungen*

Wie schon beim Schraffur-Shader in Abschnitt 4.5.1 lassen sich durch die Verwendung anderer Texturen eine Vielzahl von Varianten erzeugen. Der Woodcut-Imager-Shader ist beipielsweise auch für Stippling [Stro02, S.60-71] geeignet. Hierzu müssen die Schraffur-Texturen einfach durch Rasterpunkt-Texturen verschiedener Punktgröße ersetzt werden.

## 4.7 Prozedurale Animation

### 4.7.1 Simulation von Wasser

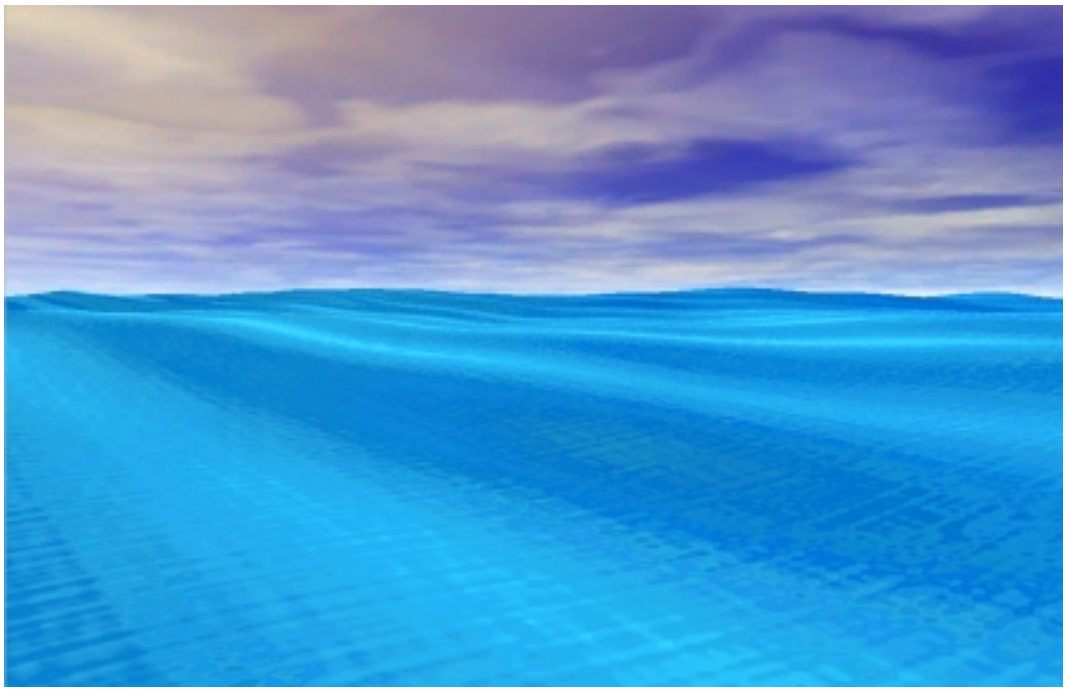


**Abbildung 45: Wassersimulation (Screenshot)**

Ziel dieses Demonstrators war es, zu zeigen, wie verschiedene Techniken zu einem einzigen Effekt kombiniert werden können. Eine Wassersimulation ist hierfür bestens geeignet, da sowohl Animationen als auch Shading angewandt werden müssen.

Eine Restriktion, der ich mich beim Entwurf unterworfen habe, war es, dass bis auf Vorberechnungen die gesamte Simulation auf den Prozessoren des Grafik-Chips abzulaufen hatte. Abbildung 45 zeigt einen Screenshot der Wassersimulation.

#### *4.7.1.1 Forschung zum Thema Wassersimulation*

Das Verhalten eines Volumens von Flüssigkeit kann durch einen Satz von Gleichungen beschrieben werden, die von Navier und Stokes im frühen 18. Jahrhundert entwickelt wurden [Fost01]. Es gibt zahlreiche Forschungsarbeiten,

die sich in den letzten zwanzig Jahren mit der Simulation von Flüssigkeiten beschäftigt haben, z.B. [Four86] [Peac86] [Mast87] [Fost00] [Fost01] [Tess01].
Die meisten der in diesen Arbeiten vorgeschlagenen Modelle sind jedoch für eine Realisierung auf FPU und VPU zu komplex. Bewährte statistische Modelle [Mast87] [Tess01] benötigen zu ihrer Umsetzung eine diskrete Fourier-Transformation, und diese ist auf der VPU schon alleine aufgrund der begrenzten Programmlänge nicht für ausreichend viele Stellen durchführbar. Für andere Techniken [Fost00] [Fost01] ist eine Kommunikation zwischen Partikeln erforderlich, die auf der VPU wegen der isolierten Verarbeitung der Vertices nicht gegeben ist. In einfacher Form realisierbar sind aber die frühen Modelle [Four86] und [Peac86], die in ihrer Simulation nur die Wasseroberfläche berücksichtigen und diese mit zeitabhängigen parametrischen Funktionen animieren.

#### *4.7.1.2 Realisierung*

Ich habe mich ausgehend von [Four86], [Peac86] und [Tess01] für ein sehr triviales Modell entschieden: Die Wasserbewegung wird durch eine Überlagerung von Sinus-Wellen verschiedener Ausbreitungsrichtungen, Amplituden und Frequenzen erzeugt.
Die Schattierung der Wasseroberfläche setzt sich aus zwei Komponenten zusammen: Einem Blau-Ton für die Farbe des Wassers, gemischt mit einer Reflektion der Umwelt. Letztere habe ich durch Environment Mapping realisiert.
Ein wesentlicher Aspekt meiner Simulation ist die Verwendung einer *Bump Map* zur Simulation der hochfrequentiven Wellen. Diese ebenfalls durch Deformation der Geomtrie zu realisieren, hätte ein wesentlich feineres Dreiecksgitter erfordert und damit die Performance der Simulation stark verringert.

#### *4.7.1.3 Deformation der Geometrie*

Zur Deformation wählte ich eine Überlagerung von vier Sinuswellen. Der Grund hierfür war, dass die VPU wegen ihres SIMD-Charakters (die meisten Instruktionen werden parallel auf allen vier Komponenten eines Quad-Floats ausgeführt) vier Sinus-Wellen in wesentlichen Teilen parallel berechnen kann.

Da der Vertex Shader, der ja für Deformationen zuständig ist, in seinem Instruktionssatz keine Kreisfunktionen zur Verfügung stellt, musste ich die Sinus-Funktion durch eine Taylor-Approximation realisieren. Die dafür notwendigen Konstanten habe ich auf der CPU berechnet und in den Konstanten-Registern der VPU abgelegt.

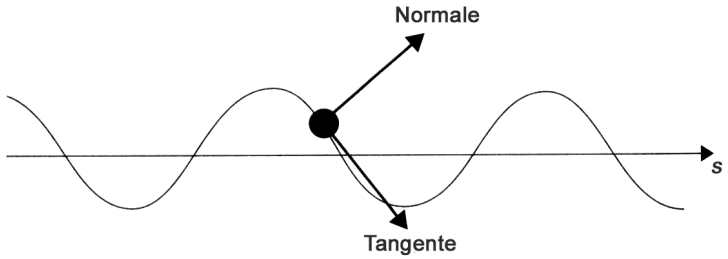



**Abbildung 46: Die Normale und Tangente an einem Punkt einer Sinus-Kurve [Pare02]**

Um Bump Environment Mapping realisieren zu können, müssen mit der Position der Vertices auch die Vertex-Normale und die Vertex-Tangente transformiert werden. Diese müssen proportional zur Veränderung der Steigung im betrachteten Punkt transformiert werden (vgl. Abbildung 46). Man benötigt also neben der Sinus-Funktion auch eine Approximation der Cosinus-Funktion.

**Satz 1:** Die Taylor-Reihe für die Sinus- und Cosinus-Funktion.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} + R_n$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + \frac{(-1)^{n-1} x^{2n-2}}{(2n-2)!} + R_n$$

Ich habe mich für eine Approximation bis zum siebten bzw. achten Glied der Sinus- bzw. Cosinus-Reihe entschieden. Um den Fehler gering zu halten, werden die Werte, die für x eingesetzt werden sollen, auf den Bereich [-π, π] abgebildet. Dies realisiere ich über die Funktion `frc`, die als Ausgabe den *gebrochenen Anteil* von Fliesskommazahlen zurückgibt. Anschließende Substraktion von 0.5 und Multiplikation mit 2π ergibt das gewünschte Ergebnis.

Beide Approximationen (vgl. Satz 1) lassen sich sehr gut gleichzeitig berechnen:

Es gelte:

c1 := ( -1/3!, 1/5!, -1/7!, 1 ),
c2 := ( -1/2!, 1/4!, 1/6!, 1/8! )

1.) Berechnung der Potenzen (`r1` enthalte *x*):

```
mul    r2, r1, r1          // x^2
mul    r3, r2, r1          // x^3
mul    r4, r2, r2          // x^4
mul    r5, r3, r2          // x^5
...
```

2.) Berechnung der Approximationen:

```
mad    r9, c1.y, r3, r1    // x-(x^3)/3!
mad    r9, c1.z, r5, r9    // x-(x^3)/3!+(x^5)/5!
mad    r9, c1.w, r7, r9    // x-(x^3)/3!+(x^5)/5!+(x^7)/7!

mov    r10,   c1.z                 // 1
mad    r10,   c2.x, r2,    r10    // 1-(x^2)/2!
mad    r10,   c2.y, r4,    r10    // 1-(x^2)/2!+(x^4)/4!
mad    r10,   c2.z,  r6,   r10    // ...
mad    r10,   c2.w,  r8,   r10
```

Die auf diese Weise erhaltenen Sinus- und Cosinus-Approximationen habe ich nun dazu verwendet, die Position, die Tangente und die Normale eines Vertex zu transformieren.

#### *4.7.1.4 Generierung der Normal Map*

Zur Realisierung der hochfrequentiven Wellen benötigte ich die in Abschnitt 4.4.2 beschriebene Bump-Environment-Mapping-Funktionalität der GeForce3.

Vorraussetzung für die oben beschriebenen Funktionen ist das Vorhandensein einer *Normal Map*. Die Generierung der Normal Map für die Wassersimulation ist recht aufwendig. Mehrere Pipeline-Durchläufe sind hierzu notwendig. Der Grund ist: Wie auch bei der Deformation soll eine Überlagerung von Sinuswellen realisiert werden, und das bedeutet, dass sich der Inhalt der Normal Map mit der Zeit verändern muss.

Eine Möglichkeit dies zu realisieren wäre es, die Normal Map außerhalb der DX8-Shader für jeden Frame neu zu erzeugen, was aber die CPU belasten würde. Ich werde nun zeigen, wie man in drei Schritten die Generierung einer animierten Normal Map auch auf der VPU und FPU der GeForce3 durchführen kann.

1.) Als einmaligen Initialisierungsschritt habe ich auf der CPU mittels einer Sinus-Funktion eine Textur mit Graustufen gefüllt. Die Sinusfunktion habe ich hierzu so skaliert und verschoben, dass sie Werte im Bereich von 0 bis 1 liefert. Man nennt eine solche Textur auch *Height Map*, denn die Werte repräsentieren Höhen: 0 sei das niedrigste Niveau, 1 das höchste.

2.) Im nächsten Schritt wird durch Überlagerung mehrerer dieser Sinus-Texturen eine weitere Height Map erzeugt (vgl. Abbildung 47). (Man kann mit der Rendering Pipeline Texturen erzeugen, indem man statt in den Frame Buffer in eine Textur schreibt.) Dazu habe ich alle Texturstufen mit der Sinus-Textur verknüpft und mit einem Vertex- und Pixel-Shader-Gespann die Überlagerung der vier Texturen erzeugt. Entscheidend ist hierbei die Fähigkeit des Vertex Shaders, Texturkoordinaten zu manipulieren. Durch Skalierung der Texturkoordinaten war es möglich, Sinuswellen verschiedener Frequenzen zu erzeugen, und durch deren Rotation konnte ich die Ausrichtung der jeweiligen Welle festlegen. Durch zeitabhängige Verschiebung der Koordinaten realisierte ich dann die Animation.
Die Überlagerung der Texturen wird vom Pixel Shader zu einer einzigen Textur zusammengefasst. Er addiert hierzu die Werte aller vier Sinus-Texturen. Es ist aber darauf zu achten, dass der Wertebereich [-1,1] nie verlassen wird, weshalb die Werte vor den Berechnungen immer erst entsprechend skaliert werden müssen.

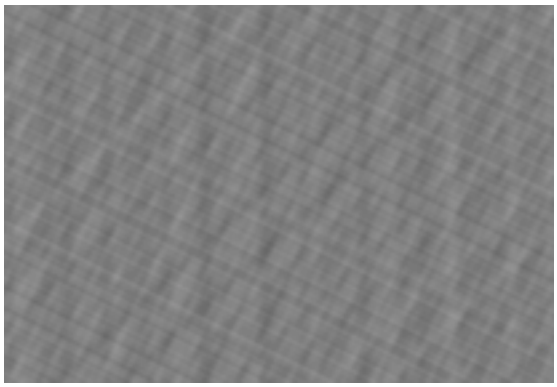


**Abbildung 47: Die Height Map der Wassersimulation**

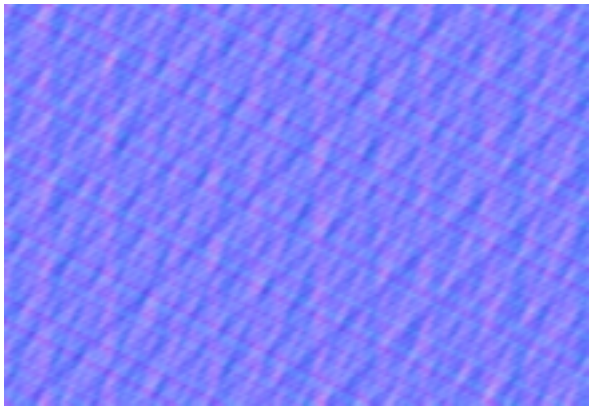


**Abbildung 48: Aus der obigen Height Map generierte Normal Map**

3.) Die so erhaltene Textur eignet sich leider noch nicht für die Bump-Environment-Mapping-Funktion der GeForce3-FPU, denn für diese wird, wie in Abschnitt 4.4.2.2 beschrieben, eine Normal Map benötigt – eine Textur, bei der die RGB-Kanäle eines jeden Texturelementes den Koordinaten einer Normale entsprechen (vgl. Abbildung 48). Um aus der im zweiten Schritt generierten Height Map eine Normal Map zu erzeugen, ist ein weiterer Pipeline-Durchlauf notwendig.

Wie bereits dargelegt, repräsentiert eine Height Map das Profil einer Oberfläche. Die erzeugte Normal Map sollte also für jeden Punkt dieses Profils eine entsprechende Oberflächennormale enthalten. Hierzu wird die Steigung in diesem Punkt benötigt. Eine vereinfachte Steigungsberechnung für einen Punkt *P* lässt sich mit den DX8-Shadern realisieren: Man vergleiche einfach die Niveaus der vier ihn umgebenden Profilpunkte (vgl. Abbildung 49).

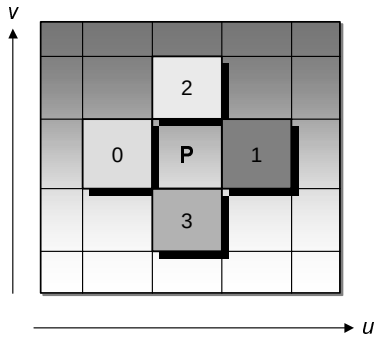



**Abbildung 49: Berechnung der Steigung im Punkt P**

Um mit der FPU gleichzeitig auf diese vier Punkte zugreifen zu können, muss die Height Map mit allen vier Texturstufen verknüpft werden, wobei die Texturkoordinaten für jede Texturstufe von der VPU um jeweils genau ein Texturelement (Texel) in die jeweilige Richtung versetzt werden. Hat man auf diese Weise die Steigung für die *u*- und *v*-Dimension bestimmt, ist die Berechnung der Normalen ein Leichtes.

Die im letzten Schritt erzeugte Normal Map kann nun für die Bump-Environment-Mapping-Funktion der FPU verwendet werden.
Das Deformationsprogramm erzeugt auf der VPU die hierfür notwendige Basis (bestehend aus Normale, Tangente und Binormale), und das in Abschnitt 4.4.2.2 beschriebene Pixel-Shader-Programm führt die Berechnungen für den Bump-Environment-Mapping-Effekt durch. Auf diese Weise werden die in Abbildung 50 erkennbaren Kerben in der Wasserfläche erzeugt.

**Abbildung 50: Ausschnitt der Wasserfläche von oben (Screenshot)**

Ein Vorteil meines Verfahrens ist, dass durch sie auf einfache Weise eine unendlich große Wasserfläche simuliert werden kann. Dies gelingt, indem man die Position und Ausrichtung der Wasserfläche mit der Rotation und der Vor- und Rückwärtsbewegung der Kamera verknüpft. Denn der Vertex Shader passt die Deformation der Geometrie automatisch an die neue Position im Weltkoordinatensystem an. Mit statistischen Modellen ist dies nur unter erheblichen Aufwand zu realisieren.

### 4.7.2 Simulation von Gras

Neben der Wassersimulation habe ich einen zweiten Demonstrator für prozedurale Animation entwickelt, der die Bewegung von Gras im Wind simuliert. Die Grasbewegung wird hierbei im Vertex Shader durch die Ausführung einer einfachen Rotation in Windrichtung vollzogen (vgl. Abbildung 51).

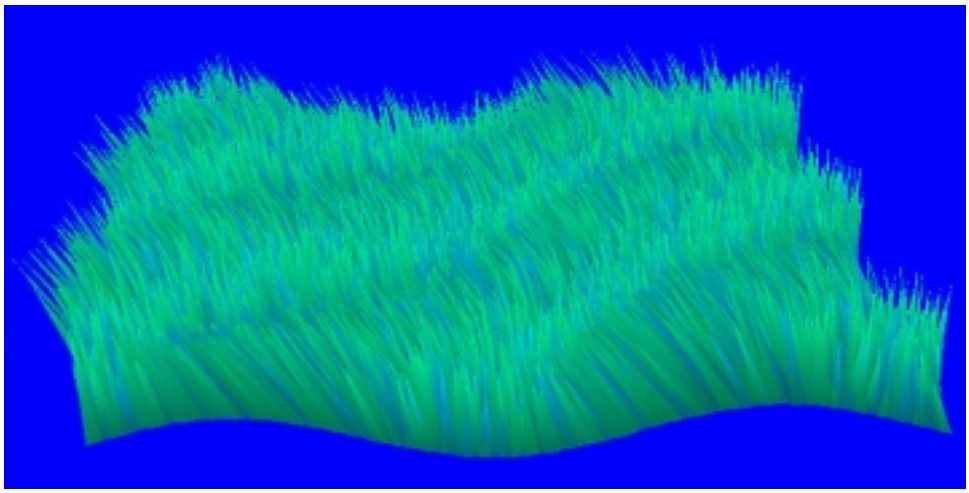


**Abbildung 51: Grassimulation**

#### *4.7.2.1 Realisierung*

Die Schwierigkeit besteht darin, im Vertex Shader jene Vertices zu selektieren, die rotiert werden müssen. Denn Vertices, die sich in der Höhe des Erdbodens befinden, dürfen sich selbstverständlich nicht an der Rotation beteiligen.
Gelöst habe ich dieses Problem durch die Einführung des Per-Vertex-Attributes „relative Höhe“. Hat ein Vertex eine relative Höhe von Null, dann befindet es sich auf der Höhe oder unterhalb des Edbodens und wird nicht rotiert. Bei Werten größer als Null nimmt der Rotationswinkel mit steigender relativer Höhe zu, denn längere Grashalme sind elastischer und bieten dem Wind mehr Angriffsfläche.
Zur Simulation von Windschwankungen verknüpfte ich den Rotationswinkel mit einer zeit- und positions-abhängigen Sinus-Funktion, die der Vertex Shader wie bei der Wassersimulation durch eine Taylor-Approximation berechnet.

# 5 Bewertung und Schlussfolgerungen

Basierend auf den, in den letzten beiden Kapiteln vorgestellten Ergebnissen meiner Analyse zu Architektur und Leistungsfähigkeit der neuen GPU-Generation, möchte ich nun eine abschließende Bewertung abgeben. Wie in Abschnitt 2.2 angekündigt, setzt sich diese Bewertung aus einer Kritik der Funktionalität und der praktischen Benutzbarkeit der neuen Technik zusammen.
Im Anschluss diskutiere ich, ausgehend von identifizierten Problembereichen, Alternativen für die Entwicklung zukünftiger Grafik-Hardware und –Interfaces.

## 5.1 Bewertung der Funktionalität

### 5.1.1 Vorteile und Grenzen der programmierbaren Geometrieverarbeitung

Die größte Innovation, welche die neuen GPU-Architekturen gebracht haben, ist die *Programmierbarkeit der Geometrieverarbeitung*. Dieser Teil der Rendering-Pipeline wurde vorher entweder auf dem Host (also als Software-Implementierung) oder auf einer nichtprogrammierbaren GPU-Komponente, der Transform&Lightning-Einheit, durchgeführt [Micr02b]. Betrachten wir also zunächst die Vorteile, die durch die neue Vertex-Verarbeitungs-Einheit entstehen.

#### *5.1.1.1 CPU-Entlastung und effizientere Verarbeitung*

Lässt man für einen Augenblick den Aspekt der Benutzerprogrammierbarkeit außer Acht, so sind es zwei Vorteile, welche die hardwarebeschleunigte Vertex-Verarbeitung mit sich bringt:
Erstens bedeutet sie eine Entlastung für die CPU. Denn dies ist die Hardwarekomponente, die die Vertex-Verarbeitung ansonsten alleine durchführen müsste. Ihr bleiben somit mehr Ressourcen für die Erledigung anderer Aufgaben, beispielsweise im Bereich der künstlichen Intelligenz und der Kollisionserkennung.
Zweitens liegt die Vertex-Verarbeitung nun in der Hand einer Komponente, die speziell für diesen Bereich gestaltet wurde, und nicht wie die CPU auch zur

Erledigung einer Vielzahl anderer Aufgaben geeignet sein muss. Die VPU konnte daher an vielen Stellen optimiert werden, was zu einer effizienteren Verarbeitung führte. Die unabhängige Verarbeitung der Vertices zur Ausnutzung der parallelen Natur der Aufgabe, sowie die Verwendung von SIMD-Instruktionen sind nur zwei Beispiele für solche Optimierungen [Lind01].

#### *5.1.1.2 Komplexe Geometrie* und *flexibles Schattieren*

Beide Aspekte zusammen bewirken, dass deutlich mehr Vertices pro Sekunde transformiert und beleuchtet werden können als dies ohne Hardware-Beschleunigung der Fall wäre [Tom02]. Daher können in interaktiven Anwendungen komplexere geometrische Modelle verwendet werden, und das kann einen deutlichen Einfluss auf die Qualität der generierten Bilder haben. Dieser Vorteil gilt aber sowohl für die neuen programmierbaren VPUs als auch für die alten nichtprogrammierbaren VPUs der ersten Generation. Diese haben aber einen entscheidenden Nachteil:

Wie ich in *Abschnitt 2.2.1.2* begründet habe, lässt sich die Bildqualität durch den Einsatz flexibler und dem Anwendungsbereich angepasster Schattierungsmethoden enorm steigern. Und hier liegt die Grenze der Leistungsfähigkeit der ersten Generation von Vertex-Verarbeitungseinheiten: Zwar war es durch deren Unterstützung möglich, deutlich komplexere geometrische Modelle zu verwenden, aber zur Beleuchtungsrechnung per Vertex stand lediglich eine einzige Methode zur Verfügung und zwar eine Variante von Phongs Beleuchtungsmodell [Phon75]. Wollte der Programmierer ein anderes Beleuchtungsmodell verwenden, musste er auf die Vorteile der Hardwarebeschleunigung verzichten und damit auch die Komplexität der Modelle niedriger halten. Die Komplexität der Geometrie und Flexibilität der Schattierung standen also miteinander im Konflikt.

Mit diesem Konflikt räumen die benutzerprogrammierbaren Vertex-Prozessoren nun auf: Wie ich im letzten Kapitel demonstriert habe, kann der Programmierer durch das Schreiben von Vertex Shadern verschiedenste Beleuchtungsmodelle realisieren *und* hochdetaillierte geometrische Modelle verwenden. Wie im letzten Kapitel demonstriert, lassen sich so physikalisch-basierte Beleuchtungsmodelle

einsetzen (siehe Abschnitt 4.4), die gut zur Simulation des Reflektionsverhaltens anisotropischer Materialien (z.B. geschliffenes Metall, Haarsträhnen) geeignet sind. Und auch eine Vielzahl nicht-photorealistischer Beleuchtungsmodelle kann nun vollständig auf der Grafik-Hardware implementiert werden.

#### *5.1.1.3 Generierung und Transformation von Texturkoordinaten*

Einen wesentlichen Anteil an den gerade gelobten flexiblen Schattierungsmöglichkeiten der neuen GPU-Generation hat die Fähigkeit, Texturkoordinaten unter Hardware-Beschleunigung zu *generieren* oder zu *transformieren*. Beispielsweise machen all die von mir realisierten Shader in irgendeiner Form von dieser Eigenschaft Gebrauch..

Die Berechnung und Transformation von Texturkoordinaten war zuvor Sache der CPU. Für Oberflächen, die nur aus wenigen Vertices bestehen, ist die Belastung hierbei verhältnismäßig gering: Für ein einfaches Quadrat bestehend aus zwei Dreiecken sind gerade einmal sechs Texturkoordinaten zu transformieren (bzw. nur vier, wenn man mit Indizes arbeitet). Nun ist es aber auch möglich, Koordinatentransformationen für komplexe Oberflächen und Objekte, die sich aus vielen Vertices zusammensetzen, ohne die geringste Belastung der CPU, in hoher Geschwindigkeit durchzuführen.

Ein wesentlicher Anwendungsbereich für die Fähigkeit der Texturkoordinaten-Berechnung ist *Environment Mapping* [Blin76][Gree86], denn dort sind Texturkoordinaten abhängig vom Blickwinkel zu berechnen. Da dieser in interaktiven Anwendungen im Allgemeinen variieren kann, müssen diese Berechnungen für jeden Frame durchgeführt werden. Da nun die CPU von dieser Aufgabe befreit ist, kann diese Technik massiver und unter Verwendung komplexerer Modelle eingesetzt werden.

Desweitern ist es möglich, die Texturkoordinaten auf der VPU linear zu transformieren. Wie in Abschnitt 4.7.1.4 anhand der Generierung einer Normal Map zur Simulation hochfrequentiver Wellen gezeigt, kann die zeitbedingte Skalierung, Verschiebung und Rotation von Texturen auch für Animationseffekte eingesetzt werden.

#### *5.1.1.4 Flexiblere Möglichkeiten der Animation*

Auch die Animation der Geometrie findet auf der neuen Hardware flexible Unterstützung. Dadurch, dass auf die Position eines Vertex innerhalb eines VPU-Programmes beliebig Einfluss genommen werden kann, können Objekte zeitabhängig deformiert werden.

In beschränktem Maße wurde dies bereits von GPUs der letzten Generation unterstützt. Wie beim Shading standen hierfür aber nur eine Auswahl vorgefertigter Algorithmen zur Verfügung. Mit der programmierbaren VPU lassen sich die Algortihmen nun besser an den Anwendungsbereich anpassen, und darüber hinaus können eine Vielzahl alternativer Animationstechniken auf der VPU ausgeführt werden (siehe Abschnitt 4.3.1).

Gerade im Bereich der Animation ist aber der Nachteil der isolierten Verarbeitung spürbar: Da Vertices unabhängig voneinander bearbeitet werden, ist die Hardware nicht geeignet für Animationstechniken, die auf Interaktionen zwischen Vertices bzw. Partikeln aufbauen, wie z.B. verhaltensbasierte prozedurale Animation [Reyn87]. Wie ich am Beispiel einer Wassersimulation gezeigt habe, kann diese Restriktion die Möglichkeiten nicht unwesentlich einschränken, da es häufig nicht möglich ist, Interaktionen zwischen Objekten durch lokale Berechnungen hinreichend zu simulieren.

Verstärkt wird diese Problem durch die Unfähigkeit, von der Applikation aus auf die Berechnungsresultate der VPU zugreifen zu können (siehe S.55). Denn ohne ein präzises Wissen darüber, wie ein Obejekt deformiert wurde, ist eine Kollsionserkennung und -behandlung in vielen Fällen nicht möglich. Daher werden Programmierer meines Erachtens bei vielen Anwendungsgebieten die flexiblere CPU der VPU vorziehen.

Dennoch besteht die Möglichkeit, bestimmte Animationsaufgaben an die VPU abzutreten. Und, wie die Wasser-Simulation demonstrierte, lassen sich, passt man die Anforderungen an die Möglichkeiten der VPU an, großflächige Animationen realisieren, die sonst nur unter großer Belastung der CPU durchführbar wären.

### 5.1.2 Vorteile und Grenzen der neuen Fragment-Verarbeitung

Weit weniger Fortschritt hat man bei der Weiterentwicklung der Fragment-Verarbeitungseinheiten gemacht. Ich habe in *Abschnitt 3.2.3* gezeigt, dass es sich bei den Fragment-Verarbeitungseinheiten der neuen GPU-Generation im Wesentlichen um eine funktionale Erweiterung der alten Multi-Texturing-Komponenten handelt. Die grundlegende Architektur der Komponente ist dieselbe geblieben, so dass man beispielsweise schon die Fragment-Verarbeitung einer GeForce256, direkter Vorgänger der GeForce3, mit dem Attribut „programmierbar“ hätte versehen können.
Zwar ist ATI mit der Entwicklung ihrer FPUs etwas weiter gegangen und bietet die Möglichkeit bedingte Texturzugriffe tatsächlich zu programmieren, anstatt wie NVIDIA den Programmierer nur zwischen verschiedenen vordefinierten Programmen wählen zu lassen. Aber auch hier ist, was die Fuktionalität der Kombinierungsfunktionen und verfügbaren Register angeht, die Verwandtschaft zur alten Multi-Texturing-Komponente noch deutlich spürbar.

Ein Aspekt, der die Berechnungskraft der FPU sehr stark einschränkt, ist die geringe Genauigkeit: Zur Speicherung von Zwischen- und Endergebnissen stehen im Allgemeinen nur 8 oder 9 Bit pro Kanal zur Verfügung. Mehrfach hintereinander ausgeführte Multiplikationen – beispielsweise um eine Zweierpotenz eines Wertes zu berechnen – können daher schnell zu sichtbaren Artefakten führen.

Dennoch hat es auch in der Fragment-Verarbeitung einen nicht zu unterschätzenden Innovationsschub gegeben:
Es kann pro Pass auf eine größere Anzahl von Texturen zugegriffen werden als zuvor, die Möglichkeit des bedingten Texturzugriffes wurde erweitert und auf jedem Fragment können mehr und komplexere Berechnungen ausgeführt werden.
Da auf der FPU Vektorprodukte berechnet werden können, ist es nun möglich die Beleuchtungsrechnung auf Fragment-Ebene durchzuführen, um beispielsweise Bump Mapping zu realisieren. Aber, wegen der stark begrenzten Instruktionsanzahl und des stark beschränkten Instruktionssatzes, ist pro Pass im

Allgemeinen nur die Interaktion mit *einer* Lichtquelle berechenbar. Von einer generellen Verschiebung der Beleuchtungsrechnung auf die Fragment-Ebene ist man daher noch weit entfernt.

Erwähnenswert ist auch, dass kaum einer der von Lastra et al. [Last95] genannten Vorteile, die ihrer Meinung nach für die Einführung von prozeduralem Shading auf Pixel-Ebene sprächen (siehe Abschnitt 2.3.2), von den heutigen FPU-Programmmodellen gewährleistet werden: Es gibt in der FPU weder eine Funktion zum Hinzufügen von Störungen (Perlin Noise [Perl85]) noch die Möglichkeit auf einfache Weise prozedurale Texturen zu generieren. Solche Dinge sind wie bisher nur durch aufwendige Multi-Pass-Techniken realisierbar.

### 5.1.3 Vorteile des Gesamtsystems

In Abschnitt 2.3.2.2 habe ich mit dem OpenGL-Shader ein System vorgestellt, das Shader-Programme auf mehrere aufeinanderfolgende Pipeline-Passes abbildet (Multi-Pass-Rendering). Multi-Pass-Rendering hat zwei entscheidende Probleme:
1.) Mit jedem Pass geht auf Grund der geringen Genaugigkeit des Frame Buffers (8-Bit) ein Datenverlust einher, der nach einigen Durchläufen zu sichtbaren Artefakten führen kann.
2.) Die Bandbreite zwischen GPU und Frame Buffer ist knapp, und da mit jedem Pass in den Frame Buffer geschrieben und aus ihm gelesen wird, sinkt dadurch die Gesamtperformance.

Diese Problematik wird durch die nun flexiblere Hardware entschärft:
Durch die Programmierbarkeit der VPU und der gewachsenen Funktionalität der FPU steigt die Zahl der Pipeline-Konfigurationen. Abstrahiert man, wie Peercy et al. [Peer00], die Grafikpipline als SIMD-Prozessor, so dass jede Pipeline-Konfiguration als eine SIMD-Instruktion und jeder Rendering-Pass der Ausführung einer solchen Instruktion entspricht, dann haben die Programmiermarkeit hat dieser Gewinn an Flexibilität den Effekt, dass dieser „Instruktionssatz“ nun wesentlich komplexer ist. Das heißt, für viele Berechnungen sind nun „Instruktionen“ vorhanden sind, die vorher aus mehreren

„Instruktionen“ zusammengesetzt werden mussten. Das bedeutet, dass im Schnitt weniger Passes notwendig sind als zuvor, was zu einer Verringerung des Bandbreiten und Genauigkeitsproblems führt.

Alles in allem hat die neue GPU-Generation gegenüber ihren Vorgängern an Funktionalität wesentlich hinzugewonnen. Wie im letzten Kapiel demonstriert und beschrieben, lässt sich dieser Bonus direkt ummünzen in den vermehrten Einsatz flexibler Schattierung und Animation. Es können also abwechslungsreichere und komplexere Szenarien generiert werden als zuvor, und – vielleicht noch wichtiger – es ist wesentlich besser möglich, die Mittel an den jeweiligen Verwendungszweck anzupassen, ohne auf Hardwarebeschleunigung verzichten zu müssen.

## 5.2 Über die Nutzbarkeit der neuen Komponenten

In diesem Abschnitt diskutiere ich, wie gut die gerade beschriebenen funktionalen Vorteile der neuen GPUs in der Praxis nutzbar sind. In Abschnitt 2.2.2 habe ich begründet, das für diesen Punkt entscheidend ist, wie gut sich die Hardware abstrahieren lässt, so dass auf deren Basis ein benutzerfreundliches und leistungsfähiges Interface entwickelt werden kann.

Beginnen werde ich meine Diskussion mit einer Bewertung des DX8-Interfaces, und zwar aus den folgenden Gründen:
1.) DirectX ist mit OpenGL das meist genutze 3D-API und daher ist eine Aussage über die Nutzbarkeit der VPU- und FPU-Interface-Komponenten auch gleichzeitig ein Gradmesser dafür, inwieweit zu erwarten ist, dass die im letzten Abschnitt beschriebenen Vorteile der neuen Hardware in der Praxis auch verwendet werden.
2.) Viele der Mängel, die ich den DX8-Shadern nachweisen werde, resultieren aus dem Aufbau der zugrundeliegenden Hardware-Architekturen. Das heißt, diese Mängel lassen sich direkt in eine Kritik an der Nutzbarkeit der aktuellen GPUs, und in Forderungen an kommende GPU-Generationen übersetzen.

3.) Mit einer Kritik am Interface anzufangen, heißt auch die Anforderungen des Benutzers und nicht die der Hardware in den Mittelpunkt zu stellen. Und für eine Bewertung der Benutzbarkeit ist eine Analyse der Benutzeranforderungen unabdingbar.

Ich werde also in diesem Kapitel den umgekehrten Weg gehen und anders als bei der Vorstellung der neuen Technik beim Interface beginnen, Mängel und Wünschenswertes ermitteln und erst danach, die ermittelten Mängel- bzw. Wunschliste im Gepäck, auf die Schwächen der Hardware eingehen.

### 5.2.1 Die Mängel des DX8-Interfaces

Den Ausgangspunktpunkt für die nun folgende Diskussion sollen, wie gesagt, die Anforderungen des Programmierers an ein optimales 3D-Application Programming Interface bilden. Als Ideal für ein solches Interface soll, wie in Abschnitt 2.2.2 angekündigt, das RenderMan Interface mit seiner Shading Language dienen. Ich setze also der Einfachheit halber die Benutzeranforderungen mit den Eigenschaften des RenderMan Interfaces gleich.
Neben dem Vergleich mit RenderMan werden folgende Qualitätskriterien bei der Dikussion des DX8-Interfaces eine Rolle spielen:

- Die Nachhaltigkeit des Interfaces,
- Die Entwicklungs-Effizienz (wie arbeitsaufwendig ist die Entwicklung mit dem Interface),
- Die Änderbarkeit, Verständlichkeit und Übertragbarkeit der Interface-Programme.

#### *5.2.1.1 Nachhaltigkeit*

Eines der Argumente, die für die Einführung einer programmierbaren DX8-Pipeline sprachen, war, dass es eine bessere Erweiterbarkeit ermöglichen sollte [Micr02a][Lind01]: Künftige Hardwareentwicklungen sollten in das bestehende Programmmodell durch die Bereitstellung neuer Befehle oder Ressourcen eingebunden werden können. Der Vorteil eines solchen Konzeptes, also der

stückweisen Erweiterung des Programmmodells, ist, dass dem Programmierer eine gewohnte Umgebung geboten wird, und er nicht mit jeder Neuerung die Grundkonzepte neu erlernen muss.
Im Falle des Vertex Shaders ist dies bisher weitestgehend gelungen: Die beiden existierenden (programmierbaren) Versionen 1.0 und 1.1 bauen konsequent aufeinander auf.
Bei den Pixel Shadern zeigt sich dagegen ein anderes Bild: Mit Version 1.4 erschien ein Interface, dass ein grundlegend anderes Programmmodell als seine Vorgängerversionen hatte. Wie ich in Abschnitt 4.2.2 zeigte, ist dies ein Resultat der starken Hardwarenähe dieser Modelle.
Hinzu kommt, dass es keinerlei Rückfallmechanismen zwischen den Versionen gibt. Das heißt, es ist mit Pixel-Shader höherer Versionsnummern nicht möglich einen Grafikchip zu steuern, der nur die Funktionalität einer früheren Version unterstützt. Versucht man beispielsweise einen GeForce3-Chip mit einem Shader-Programm der Version 1.2 anzusprechen, erhält man grundsätzlich eine Fehlermeldung, egal, ob man nur Instruktionen verwendet, die auch schon in Version 1.1 vorhanden waren.

#### *5.2.1.2 Die Nachteile der Assembler-Sprache*

Ich möchte nun auf die Nachteile eingehen, die die Verwendung einer Assemblersprache mit sich bringt, und warum die Einführung von Kontrollstrukturen eine wesentliche Bereicherung des Modells bedeuten würden.
Zur Veranschaulichung werde ich anhand eines Beispiel-Shaders das DX8-Shader- und RenderMan-Programmmodell gegenüberstellen (vgl. Tabelle 16).

**DX8-Shader**

```
;VERTEX SHADER VERSION 1.1
vs.1.1

;Transformation und Ausgabe der
;Position
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

;Transformiere Vertex-Normale
dp3 r0.x, v3, c5
dp3 r0.y, v3, c6
dp3 r0.z, v3, c7

;Normalisiere Vertex-Normale
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w

;Berechne Position im
;Weltkkordinatensystem
dp4 r1.x, v0, c11
dp4 r1.y, v0, c12
dp4 r1.z, v0, c13
dp4 r1.w, v0, c14

;e = Vektor von Position zum Auge
add r2, c10, -r1

;Normalisiere e
dp3 r2.w, r2, r2
rsq r2.w, r2.w
mul r2, r2, r2.w

;Berechne h = l + e
add r1, r2, c4

;Normalisiere h
dp3 r1.w, r1, r1
rsq r1.w, r1.w
mul r1, r1, r1.w

;l dot n
dp3 oT0.x, r0, c4

;h dot n
dp3 oT0.y, r1, r0
```

```
;PIXEL SHADER VERSION 1.1
ps.1.1

;Zugriff auf Textur 0
tex t0

;Ausgabe des Ergebnisses
mov r0, t0
```

**RenderMan Surface Shader**

```
// SURFACE SHADER
Surface anisotropic ()
{
   // Berechne Normale
   normal Nf =
       faceforward (normalize (N), I);

   C = 0;

   illuminance (P)
   {
      // Berechne h = l + e
      vector Ln = normalize(L);
      vector H = normalize
                 (Ln - normalize (I));

      // Berechne Texturkoordinaten
      float Ts = Ln . Nf;
      float Tt = H . Nf;

      // Zugriff auf Textur
      C += color texture
                 ("aniso.bmp", Ts, Tt);
   }
   // Ausgabe des Ergebnisses
   Ci = C;
}
```

**Tabelle 16: Eine Gegenüberstellung von DX8- und RenderMan-Shadern anhand eines Beispielprogramms**

Zunächst fällt auf, dass der Shader-Code in DirectX8 wesentlich länger ausfällt. Dies liegt im Wesentlichen daran, dass ein Programmierer von DX8-Shadern nur

auf den vorgegebenen Satz an einfachen Basisinstruktionen zurückgreifen kann. Bei RenderMan hingegen gibt es neben Basisoperationen auch komplexe eingebettete Funktionen, und darüber hinaus kann der Programmierer eigene Funktionen definieren (siehe Abschnitt 2.2.4.6).

Ein weiterer Schwachpunkt der DX8-Shader ist, dass die Register immer mit ihrer Adresse angesprochen werden müssen – die Deklaration von Variablen, wie in RenderMan, ist nicht möglich.

Beide Punkte erschweren die Lesbarkeit und Verständlichkeit des Programmcodes. Es ist aber möglich, diesen Nachteil durch den Einsatz von Kommentaren weitestgehend auszugleichen.

Eine negative Konsequenz des Programmierens mit der VPU- und FPU-Maschinensprache ist auch, dass die Optimierung des Programmcodes vollständig im Verantwortungsbereich des Programmierers liegt. Der Compiler übernimmt das Programm so wie es ist und nimmt keinerlei Optimierungen vor. Dies gilt sowohl für im Programm sichtbare Verbesserungsmöglichkeiten wie beispielsweise die Minimierung der Instruktionsanzahl (z.B. durch Zusammenfassen einer `mul`- und einer `add`-Instruktion zu einer `mad`-Instruktion) als auch für Effekte, die aus dem Programm-Code nicht ersichtlich sind: NVIDIA empfiehlt beispielweise das Ausgaberegister für die Position, `oPos`, im Vertex-Shader-Programm so früh wie möglich zu setzen, da dies die Performance steigern würde.

Zusammengefasst wirken sich all diese Aspekte negativ auf die Entwicklungs-Effizienz aus.

#### *5.2.1.3 Optimierungs-Probleme aufgrund fehlender Kontrollkonstrukte*

Viel gravierender als die oben aufgeführten Probleme ist das Fehlen von Kontrollkonstrukten zur Steuerung des Ausführungspfades.

Zwar ist es möglich, durch die Operationen `slt` und `sge` (vgl. Tabelle 23) im Vertex Shader Bedingungen abzufragen, diese haben auf den Ausführungspfad aber keinen Einfluss. Es werden grundsätzliche alle Operationen eines Vertex-Shader-Programms ausgeführt. Daher besteht beispielsweise nicht die Option, die Komplexität des Shaders flexibel zu gestalten, so dass, abhängig von bestimmten

Rahmenbedingungen, in einigen Fällen ein langes und in anderen Fällen ein kürzeres Programm ausgeführt werden könnte. Vorstellbar wäre beispielsweise – in Analogie zum Levels-Of-Detail-Konzept [Hopp96] – eine stufenweise Verringerung der Schattierungsqualität in Abhängigkeit von der Entfernung zum Betrachter, um eine Optimierung der Performance zu erreichen.

Ein wirkliches Problem ergibt sich aber in einem anderen Bereich: Die beiden in Tabelle 16 gegenübergestellten Shader ähneln sich zwar in ihrer Funktionalität, wesentlich allgemeiner aber ist der RenderMan-Shader. Denn er ist für jede in RenderMan erzeugbare Lichtquellenkonstellation gültig. Diese Allgemeingültigkeit erhält er durch das „illuminance“-Konstrukt [Pixa00, S.127-128].

Der gegenübergestellte DX8-Shader dagegen ist spezialisiert auf die Schattierung unter Einfluss einer einzelnen Punktlichtquelle. Soll die zu schattierende Oberfläche beispielsweise von *zwei* Lichtquellen beleuchtet werden, dann muss hierfür ein anderer Shader geschrieben werden. In DX8 gibt es keine Möglichkeit einen einzigen, für alle möglichen Lichtquellenkonstellationen gültigen Vertex Shader zu entwickeln.

Die drei in DirectX üblichen Lichtquellentypen sind *gerichtete Lichtquellen*, *Punktlichtquellen* und *Spotlights* (vgl. Abbildung 52). Hierbei können Punktlichtquellen als Verallgemeinerung von gerichteten Lichtquellen und Spotlights wiederum als Verallgemeinerung von Punktlichtquellen angesehen werden: Gerichtete Lichtquellen haben als Attribute Farbe und Richtung, Punktlichtquellen haben eine Position (aus der sich die Richtung bestimmen lässt) und zusätzlich einen Abschwächungsfaktor, Spotlights zusätzlich zwei Winkel, welche die Ausdehnung des inneren und äußeren Lichtkegels festlegen.

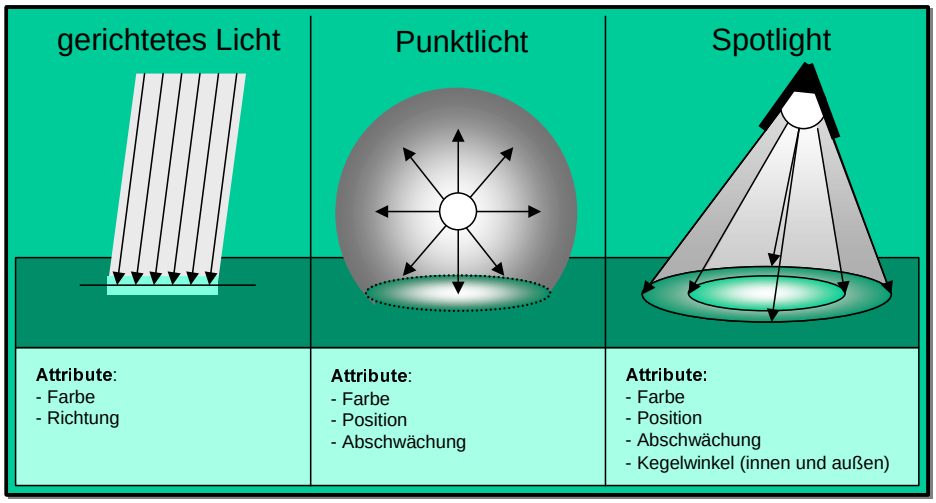



**Abbildung 52: Lichtquellentypen in DirectX [Micr02b]**

Daher wäre es eine theoretische Alternative, einen Shader für eine ausreichend große Anzahl von Spotlights zu schreiben und diesen dann für alle Beleuchtungsumgebungen zu verwenden. Ein solcher Shader muss aber gewährleisten, dass

1. nicht vorhandene Lichtquellen keinen Einfluss auf das Endergebnis haben und
2. bei Verwendung von Punkt- und gerichtetem Licht, *nicht* vorhandene Lichtquelleneigenschaften wie Position, Abschwächungsfaktor oder Kegelwinkel durch entsprechende Werte ersetzt werden.

Das Problem an diesem Ansatz ist, dass für eine kleine Anzahl an Lichtquellen ein hoher Anteil an unnötigen Operationen durchgeführt werden. Denn ein Vertex Shader führt aufgrund fehlender Kontrollkonstrukte grundsätzlich immer alle Programminstruktionen aus. Da die Ausführungsgeschwindigkeit proportional zur Länge der Programme ist, führt dies zu vermeidbaren Performanceeinbußen. Ebenso vermeidbar sind die zusätzlichen Berechnungen, die eventuell anfallen, da grundsätzlich vom attributreichsten Typ Spotlight ausgegangen wird. Denn je weniger Attribute zu berücksichtigen sind, desto weniger Berechnungen sind notwendig. Verwendete man beispielsweise nur gerichtete Lichtquellen, könnte man sich u.a. die Berechnung des Lichtvektors pro Vertex sparen, denn dieser ist bei gerichtete Lichtquellen über die gesamte Szene konstant.

Eine optimale Performance ist nur erreichbar, wenn der Programmierer für jede verwendete Anzahl und Kombination von Lichtquellen einen speziellen Shader entwickelt und in der Applikation eine Abfrage startet, die je nach Szenario den richtigen Shader aktiviert.

#### *5.2.1.4 Das fehlende Baukasten-Prinzip*

Zum Vergleich habe ich in Tabelle 17 die RenderMan- und DX8-Shader-Typen mit ihren jeweiligen Aufgaben aufgelistet.

| Interface | Shader-Typ | zu bearbeitendes Element | Aufgabe |
|---|---|---|---|
| RenderMan | *Displacement Shader* | Punkt auf einer Oberfläche | Bearbeitung der Oberflächenstruktur |
| | *Surface Shader*<br>*Atmosphere Shader*<br>*Light Source Shader* | Punkt auf einer Oberfläche | Beleuchtungsrechnung |
| | *Imager Shader* | Pixel | Bildbearbeitung (Filterung, Linseneffekte) |
| DirectX8 | *Vertex Shader* | Vertex | Vertex-Verarbeitung |
| | *Pixel Shader* | Fragment | Fragment-Verarbeitung |

**Tabelle 17: RenderMan- und DX8-Shader-Typen im Vergleich**

In RenderMan findet sich eine Typisierung nach Shading-Aufgaben: *Volume*, *Surface* und *Light Source Shader* bilden ein Kollektiv zur Simulation der Interaktion zwischen Lichtquellen und Umgebung. Mit *Displacement Shadern* lassen sich Oberflächen mit Unebenheiten versehen, die im geometrischen Ausgangsmodell nicht vorhanden waren. Mit dem *Imager Shader* stellt RenderMan eine Schnittstelle zu Erstellung von Linseneffekten zur Verfügung wie beispielsweise Bewegungsunschärfe oder Tiefenschärfe. Die verschiedenen Shader-Typen ergänzen sich, aber auch jeder Shader für sich genommen ergibt einen bestimmten optischen Effekt.

Diese Aufteilung reflektiert die Sichtweise, dass sich die endgültige Schattierung einer Oberfläche aus der Überlagerung verschiedener Effekte zusammensetzt, die jeweils isoliert betrachtet und modelliert werden können. Dieses Baukastenprinzip übernahm man im Wesentlichen von Cooks Modell [Cook84], der erkannt hatte,

dass durch die Kombination verschiedener einfacher Schattierungsverfahren, komplexe Effekte erzielt werden können (vgl. Abschnitt 2.2.3.1).

Eine solches Konzept fehlt bei DX8: Eine Untergliederung nach Aufgaben, wie beispielsweise die Trennung von Animation und Shading ist nicht vorhanden. Statt dessen wird die Aufteilung in Vertex und Pixel Shader bestimmt durch den Typ des zu verarbeitenden Datenelements – Vertex oder Fragment. Die Shader-Typen repräsentieren Verarbeitungseinheiten der Rendering-Pipeline.
Daraus resultiert, dass Vertex und Pixel Shader *nur in Zusammenarbeit funktionieren* – der Eine ist auf die Funktionalität des Anderen angewiesen. Das in Tabelle 16 aufgeführte Beispiel verdeutlicht dies: Obwohl der größte Teil der Arbeit vom Vertex Shader geleistet wird, ist für den gewünschten Effekt der Zugriff auf eine Textur notwendig, und diese Aufgabe kann nur der Pixel Shader leisten.
Eine getrennte Betrachtung und Modellierung von Vertex und Pixel Shadern ist daher nicht sinnvoll. Dies käme der unabhängigen Entwicklung der letzten beiden Befehlszeilen vom übrigen Programm des RenderMan Shaders in Tabelle 16 gleich. Die Textur, auf die in diesen Zeilen zugegriffen wird, enthält eine Funktionstabelle, und die Texturkoordinaten müssen daher auf dieselbe Weise berechnet werden wie es im oberen Teil des RenderMan Shaders der Fall ist. Das Programmfragment ergibt also nur in diesem Kontext einen Sinn.

Auch ist es nicht möglich, innerhalb eines Pipeline-Durchlaufs mehrere Shader gleichen Typs miteinander zu kombinieren.
Die Kombination von Shadern kann allerdings durch das Hintereinanderschalten mehrerer Pipeline-Durchläufe realisiert werden, was ich in Abschnitt 4.7.1.4 anhand der Generierung einer animierten Normal Map demonstiert habe. Wie ich aber im nächsten Abschnitt zeigen werde hat auch diese Technik aufgrund von Schwächen im Programmmodell ihre Grenzen.

#### *5.2.1.5 Kein direkter Zugriff auf Vertex-Shader-Resultate*

Ein Punkt, der zu einer starken Abwertung des Gesamtsystems führt, ist die Restriktion, dass der Programmierer *keinen Zugriff auf die Ausgaberegister* des Shaders hat. Dem Programmierer stehen als greifbares Resultat nur die am Ende erzeugten Pixel-Daten zur Verfügung.

Für die *Kombination von Shadern* bedeutet dies: Es ist (auf einfache Weise) nicht möglich in einem weiteren Pipeline-Durchlauf die Resultate des vorherigen Vertex Shaders als Eingabe für den nächsten zu verwenden.

Ein Einsatzgebiet für einen solchen Mechanismus wären rekursive Berechnungen: Das nächste Bild in Abhängigkeit von den Pixel-Daten seines Vorgängers zu berechnen ist eine häufig genutzte Technik (Image Feedback) [McCu02, S.489-540]. Eine solche Endlosschleife voneinander abhängiger Berechnungen könnte auch basierend auf den *Vertex-Shader-Resultaten* des jeweils vorhergehenden Pipeline-Durchlaufs durchgeführt werden, um beispielweise Animationen zu erzeugen.

Desweiteren wäre das Hintereinanderschalten mehrerer Vertex-Shader gerade auch zur Entschärfung der Ressourcen-Problematik ein wirksames Mittel: Ist ein Problem zu komplex, um es innerhalb eines Vertex-Shader-Programms zu berechnen, teilt man es auf zwei oder mehrere Programme auf.

Eine effiziente Zugriffsmöglichkeit auf Vertex-Shader-Resultate böte also eindeutige Vorteile. Im Bereich der Animation führt das Fehlen dieser Alternative zu einer besonderen Problematik, denn sie bewirkt, dass sich die Kombination von Shadern durch Multi-Pass-Rendering nicht mit Animationen vertragen::

Der Vertex-Shader transformiert zwar die Geometrie innerhalb der Pipeline, aber er tastet *nicht* die Daten des Ursprungsmodells an. Diese bleiben im Vertex Buffer in ihrer ursprünglichen Form erhalten. Es ist daher beispielsweise nicht möglich einen Vertex und Pixel Shader für eine Skelett-Animation zu schreiben, und diesen über Multi-Pass-Rendering mit einem Morphing-Shader zu kombinieren. Denn letzterer hat kein Wissen über die Transformationen, die sein Vorgänger durchgeführt hat.

Genauso unmöglich ist es, auf diese Weise einen Animations-Vertex-Shader mit einem Shader zur Berechnung der Schattierung zu verknüpfen.
Multi-Pass-Rendering funktioniert nur dann korrekt, wenn die Transformationen der Geometrie mit jedem Pipeline-Durchlauf aufs Neue durchgeführt wird. Soll beispielsweise eine Figur unter Verwendung eines Vertex Shaders animiert werden *und* über mehrere Pipeline-Durchläufe schattiert werden, muss ein Vertex Shader die Animation für jeden dieser Durchläufe berechnen, was sich bei komplexen Animationsprogrammen sehr negativ auf die Performance niederschlagen kann.

Die einzige Möglichkeit, dennoch auf Vertex-Shader-Resultate zuzugreifen, besteht darin, diese in kodierter Form in den Pixel-Daten des Ausgabebildes zu speichern. Ein solches Verfahren wäre durch das Vergeben von Vertex-Indices durchführbar. Der Nachteil aber ist, dass diese Daten, um verwendet werden zu können, vor dem nächsten Pipeline-Durchlauf erst wieder dekodiert werden müssen. Bei *n* transformierten Vertices sind hierfür auch *n* Dekodierungsberechnungen notwendig. Außerdem ist nach jedem Kodierungsschritt ein weiterer Pipeline-Durchlauf zur eigentlichen Bildgenerierung notwendig.

#### *5.2.1.6 Die starke Kopplung zur übrigen Interface Umgebung und deren Folgen*

In Abschnitt 2.2.4.7 verwies ich bereits auf einige Vorteile des RenderMan Interfaces. Es sind im Wesentlichen die folgenden drei Konzepte, die die Programmierung mit der RenderMan Shading Language erleichtern:

- **Zusicherung:** Die Menge der Attribute, die einem RenderMan-Shader als Quelle für die Berechnungen dienen, unterteilen sich in *vordefinierte* und s*hader-spezifische* Werte. Die vordefinierten Attribute sind eine ausgewählte Menge von Variablen, auf die jeder Shader für seine Berechnungen zurückgreifen kann. Das Vorhandensein dieser Attribute ist dem Shader *zugesichert* – das Rendering Programm ist für deren Initialisierung und Aktualisierung verantwortlich [Pixa00, Kapitel 11.7].

- **Absicherung:** Die shader-spezifischen Attribute bilden das Interface eines Shaders. Ihnen können innerhalb des Shaders Default-Werte zugewiesen werden, so dass sie in jedem Fall initialisiert sind, wenn das Programm seine Arbeit beginnt.
- **Abkapselung:** Alle Werte, die nicht für das Interface vorgesehen sind, können *abgekapselt* innerhalb des Shaders berechnet und gespeichert werden. Dies gilt sowohl für uniforme, wie für variierende Variablen.

Diese Art der Zusicherung, Absicherung und Abkapselung fördern aus Sicht des Shader-Programmierers die Entwicklung und Lesbarkeit von Shadern. Andererseits ist es aus Sicht des Entwicklers der Applikation möglich, Shader zu benutzen, ohne mehr als ihre Schnittstelle zu kennen. So können Shader gleichen Typs (Surface, Volume, etc.) problemlos gegeneinander ausgetauscht werden, denn es sind lediglich die *shader-spezifischen* Attribute, die gegebenenfalls angepasst werden müssen.

Bei DX8-Shadern zeigt sich ein völlig anderes Bild: Es gibt keinerlei vordefinierte Attribute. Weder die variierenden, noch die uniformen Eingaberegister hat eine feste Bedeutungszuweisung. Es liegt in der Verantwortung des Programmierers bestimmte Register mit einer festgelegten Bedeutung zu versehen, damit seine Shader weitestgehend austauschbar bleiben und leichter zu lesen sind. Das Verwenden von Shadern eines anderen Programmierers ist nicht möglich, ohne zuvor ein genaues Studium des jeweiligen Programms durchzuführen und die Registerbezeichnungen an das eigene Schema anzupassen.

Auch sind die Shader in DX8 weit weniger abgekapselt als in RenderMan, denn essentielle Teile der Funktionalität liegen in den Händen der Umgebung.
Ein Beispiel hierfür ist die Berechnung *uniformer* Attribute: Die einzige Shader-Instruktion, die eine Schreiberlaubnis bei *Konstanten-Registern* hat, ist die Defintionsinstruktion. Diese ist aber nur für das Setzen von Werten geeignet, die

keiner Vorberechnungen bedürfen. Will der Programmierer einen uniformen Wert innerhalb des Shaders berechnen, dann muss er:

1. eine der rar gesäten temporären Register zur Speicherung verwenden und
2. in Kauf nehmen, dass der Wert, obgleich er für alle Datenelemente konstant ist, für jeden Vertex bzw. jedes Fragment aufs Neue berechnet wird.

Bei *n* zu verarbeitenden Datenelementen, würden damit *n* unnötige Berechnungen durchgeführt.

Um eine optimale Performance zu erreichen, müssen daher uniforme Berechnungen außerhalb des Shaders getätigt werden und ihm anschließend über die Befehle „`setVertexShaderConstant`“ bzw. „`setPixelShaderConstant`“ übermittelt werden.

Und noch ein weiterer Aspekt verhindert die Abkapselung. Wie in Kapitel 4 mehrfach demonstriert, können Texturen auch als Funktionstabellen verwendet werden, z.B. zur Realisierung von anisotropischen oder nicht-photorealistischen Beleuchtungsmodellen. In diesem Fall ist die Funktionsweise des Shaders eng mit dem Inhalt der verwendeten Textur verbunden.

Dennoch kann der Programmierer Texturen ausschließlich außerhalb des Shaders deklarieren und mit Texturstufen verknüpfen sowie den benötigten Texture-Sampling-Modus einstellen, der bestimmt, welches Filterungsverfahren [Möll99, S. 106-119] verwendet und wie mit Texturkoordinaten verfahren wird, die außerhalb des Werte-Bereiches [0,1] liegen (Wrap, Mirror, Clamp, etc [Möll99, S.103]).

Bei DX8-Shadern ist also es im Gegensatz zu RenderMan Shadern schwierig, die Steuerungsschnittstelle auf ein Minimum von Attributen zu reduzieren. DX8 Shader sind in hohem Grade abhängig von der übrigen DX8-Umgebung, da ein Teil der zur Realsierung eines Effektes notwendigen Berechnungen außerhalb des Shaders durchgeführt werden muss.

Auch die Austauschbarkeit von Shadern und insbesondere die Verwendung von Shadern anderer Programmierer gestaltet sich durch die gerade aufgeführten

Sachverhalte als schwierig. Ich kann aus eigener Erfahrung sagen, dass ein nicht selbstentwickelter Shader kaum zu gebrauchen ist, falls nicht auf verständliche Weise dokumentiert wurde, welche Attribute, Vorberechnungen und Texturen ein Shader benötigt, und mit welchen Registern bzw. Textur-Stufen diese zu verknüpfen sind.

#### *5.2.1.7 Die Abhängigkeit von Feinheit der Geometrie und Anzahl der Vertex-Shader-Berechnungen*

In RenderMan aber wird die Mindestanzahl an Shading-Berechnungen pro Bildfläche durch die sogenannte *Shading Rate* (vgl. Abschnitt 2.2.4.3) geregelt. Das heißt, man hat hier die Möglichkeit, die Anzahl der Berechnungen unabhängig von der geometrischen Repräsentation zu regulieren.
Dagegen ist in DX8, da ein Vertex Shader keine Vertices erzeugen kann, die Anzahl seiner Berechnungen abhängig vom Detailgrad der Geometrie des zu verarbeitenden Modells.
Das bedeutet, dass der Programmierer beim Erstellen geometrischer Modelle genau bedenken muss, wie fein das Vertex-Netz sein muss, um den gewünschten Schattierungs- oder Animationseffekt zu erzielen. Deformationen wie beispielsweise bei der von mir entwickelten Wassersimulation erfordern häufig einen bestimmten Mindest-Zergliederungsgrad.

Im Bereich Shading besteht die Möglichkeit diese Abhängigkeit, über die Durchführung der Beleuchtungsrechnung per Fragment zu reduzieren. Da aber die Funktionalität der DX8-Pixel-Shader noch sehr eingeschränkt ist, und daher der Einfluss mehrerer Lichtquellen im Allgemeinen nur im Vertex Shader berechnet werden kann, ist eine solche Unabhängigkeit nur in Ausnahmefällen möglich.

Ein Ausweg, der sich für diese Problematik abzeichnet, ist die Unterstützung der Zerlegung grafischer Primitiva höherer Ordnung (High-Order-Primitive Tesselation [Nark95]): Das Modell wird durch Primitva höherer Ordung beschrieben und erst bei der Bildgenerierung in Dreiecke zerlegt. Leider hat die

entsprechende DX8-Interface-Komponente [Micr02c] noch viele Schwächen, so dass beispielsweise keine stufenlose Zerlegung möglich ist.

### 5.2.2 Konsequenzen für die Hardware- und Interface-Entwicklung

Ich möchte nun darlegen, welche der gerade beschriebene Probleme sich durch Veränderungen an der Hardware vermeiden ließen, und welche Konsequenzen diese Veränderungen für die Interface-Entwicklung hätten. Die nun folgende Diskussion mündet im Wesentlichen in zwei Forderungen:

- Die Forderung nach einem effektiven Zugriff auf VPU-Resultate.
- Die Forderung nach der Einführung von Kontrollstrukturen.

#### *5.2.2.1 Das Problem der begrenzten Ressourcen*

Die in Abschnitt 5.2.1 aufgeführten Schwächen des Programmmodells – die Ressourcenknappheit, die Verwendung einer Assemblersprache, das Fehlen von Kontrollstrukturen – liegen alle begründet in der Hardwarearchitektur von VPU und FPU. Eine Verbesserung der Hardware-Programmmodelle gäbe den Entwicklern von Interfaces die Möglichkeit, weniger maschinennahe Abstraktionen zu erstellen.
High-Level-Sprachen wie C haben den allgemein bekannten Vorteil, dass sie eine problemorientiertere und damit effizientere Entwicklung ermöglichen als Maschinensprachen. Zum Beispiel können Optimierungen und Registeradressierung in die Hand des Compilers gelegt werden und der Programmierer kann seine Programme durch die Verwendung von Funktionen besser strukturieren. Aber auf der schmalen Basis stark begrenzter Registeranzahlen und Programmlängen lässt sich nur schwer eine High-Level-Sprache verwirklichen.

Dennoch hat man mit der Sprache *Cg* Versuche in diese Richtung unternommen – mit unbefriedigendem Resultat:

Man näherte sich zwar optische der Sprache C an, indem man man Befhele in ihrer Syntax an das Vorbild anpasste, aber unter diesem dünnen Lack stößt man sogleich auf das bereits bekannte VPU- und FPU-Programmmodell. Als Beleg hierfür möchte ich kurz die for- und while-Konstrukte von *Cg* vorstellen.
In [NVID02d, S.10] heißt es hierzu sinngemäß: Die aktuelle Vertex- und Fragment-Hardware unterstützt keine Verzweigungsinstruktionen, so dass for- und while-Schleifen nur verwendet werden können, wenn der Compiler sie auflösen kann, und die erforderliche Instruktionszahl nicht die Beschränkungen der Hardware überschreitet.
Ich halte eine solche Form der Abstraktion für kontraproduktiv: Es ist bildlich gesprochen ein wenig so, als setze man den Programmierer auf einer weiten Ebene aus, aber um ihn herum befinde sich ein Glaskasten in der Größe einer Besenkammer – für ihn unsichtbar, so dass er immer erst gegen die Wände laufen müsste, um seinen wahren Freiheitsgrad zu erfahren.

Die einzige Möglichkeit dem Problem der Hardware-Restriktionen konsequent zu begegnen, ist es, den Weg von SGIs OpenGL Shader (siehe Abschnitt 2.3.2.2) zu gehen und die gesamte Rendering-Pipeline zu abstrahieren. Mit diesem Ansatz kann ein System entwickelt werden, das High-Level-Programme auf mehrere Pipeline-Passes abbildet. Die Flexibiltät der neuen Hardware bietet, wie in Abschnitt 5.1.3 begründet, die Möglichkeit die Anzahl der im Schnitt benötigten Passes stark zu reduzieren und somit die von mir erwähnten Probleme von Multi-Pass-Rendering-Systemen (siehe S.39) zu reduzieren.
Die Stanford Shading Language [Prou01] (siehe Abschnitt 3.3.3.2) verfolgt diesen Ansatz. Sie bietet eine benutzerfreundliche Schnittstelle, welche die Restriktionen der Hardware vom Software-Entwickler durch Abstraktion der gesamten Rendering-Pipeline weitestgehend fern hält. Dieses System hat aber einen entscheidenden Nachteil: Es nutzt nicht die Fähigkeiten, die die neuen GPUs im Bereich der *Animation* zu bieten haben (siehe Abschnitt 4.3.1). Denn, da die Hardware den Zugriff auf VPU-Resultate verbietet (s. S.55), bleibt die in Abschnitt 5.2.1.5 beschriebene Unverträglichkeit von Animation und Multi-Pass-

Rendering bestehen. Es gibt keine Möglichkeiten VPU-Animations-Programme auf mehrere Passes aufzuteilen.
Eine Abstraktion, die die Hardware-Restriktionen verdecken soll, muss daher auf die Animations-Fähigkeiten der neuen GPUs verzichten und Berechnungen zur Deformation der Geometrie auf der CPU ausführen.

#### *5.2.2.2 Forderung: Zugriff auf VPU-Resultate*

Eine Lösung für dieses Problem ergäbe sich, erlaubte man in zukünftigen GPU-Generationen den Applikationen einen effektiven Zugriff auf VPU-Resultate. Damit wäre auch die in Abschnitt 5.1.1.4 beschriebene Schwierigkeit mit der Kollisionserkennung beseitigt.
Bestünde eine effiziente Zugriffsmöglichkeit auf VPU-Resulate, dann könnte die Ausgabe eines VPU-Durchlaufes als Eingabe für den nächsten dienen. So könnte ein Vertex die VPU in einer Schleife mehrere Male hintereinander durchlaufen, wobei mit jedem Schleifendurchlauf ein anderes Programm ausgeführt würde – eine Technik, die ich VPU-Multi-Pass-System getauft habe. (Allerdings ist die Vorraussetzung für diesen Aufbau, dass die Eingabe- und Ausgaberegister der Verarbeitungseinheit in Anzahl und Bedeutungszuweisung identisch sind.)
Mit einem solchen System wäre die Programmlänge theoretisch unbegrenzt, und damit problemlos eine Abstraktion entwickelbar, die VPU-Animations- oder Shading-Programme auf mehrere VPU-Passes abbildete.
Dadurch könnte auch die von mir kritisierte Kopplung von Shading und Animation aufgehoben werden: Im ersten VPU-Pass berechnet ein Animations-Programm die Deformation eines Objektes, und basierend auf dessen Berechnungen, berechnet im zweiten VPU-Pass ein anderes Programm die Schattierung.

Ein ähnliches System wäre natürlich auch auf Fragment-Ebene denkbar (FPU-Multi-Pass-System), aber da über den Frame-Buffer bereits jetzt auf die Resultate einer FPU zugegriffen werden kann, wäre in diesem Bereich diese Technik erst interessant, wenn die FPU intern über eine größere Genauigkeit verfügte (16-Bit pro Kanal oder mehr) als der Frame-Buffer.

#### *5.2.2.3 Forderung: Einführung von Kontrollstrukturen*

In Abschnitt 5.2.1.3 habe ich das Fehlen von Kontrollstrukturen bei DX8-Shadern beklagt. Dieser Mangel geht ebenfalls direkt auf die VPU- und FPU-Programmmodelle zurück. Kontrollstrukturen, ob auf Vertex- oder Fragment-Ebene, böten die Möglichkeit, wesentlich allgemeinere VPU- bzw. FPU-Programmen zu entwickeln. Denn so könnte ein einziges VPU- bzw. FPU-Programm gleich mehrere Konstellationen von Eingangsbedingungen abdecken. Als Anwendungsbeispiel habe ich bereits die entfernungsabhängige Reduktion der Shader-Komplexität genannt, ähnlich dem Level-of-Detail-Konzept.

Ein VPU-Multi-Pass-System kombiniert mit Kontrollstrukturen würde eine einfache Lösung der *Lichtquellenproblematik* (siehe Abschnitt 5.2.1.3) ermöglichen. Durch eine *for*-Schleife könnte man ein Äquivalent zu RenderMans „*illuminace*-Konstrukt“[13] [Pixa00, S.127-128] realisieren: Das Programmfragment für die Beleuchtungsrechnung würde einmal für jede Lichtquelle ausgeführt und die Einzelergebnisse jeder Schleife miteinander kombiniert. Durch *if*-Abfragen könnte man für die verschiedenen Lichtquellentypen (vgl. Abbildung 52) unterschiedliche, an die jeweiligen Lichtquellen-Attribute angepasste Programmfragmente erstellen.
Das VPU-Multi-Pass-System würde hierbei garantieren, dass die Anzahl der Lichtquellen nicht durch die Programmlänge begrenzt wäre. Denn eine High-Level-Abstraktion der Hardware könnte, falls die VPU-Ressourcen nicht die Berechnung innerhalb eines Passes ausreichten, die verbleibenden Berechungsschritte automatisch auf weitere VPU-Passes verteilen.
Deweiteren könnten *Abbruchbedingungen* die Anzahl der durchgeführten Berechnungsschritte, von der aktuellen Bildgenerierungsrate abhängig machen.
Kontrollstrukturen wären also, in Kombination mit der Möglichkeit beliebig viele VPU-Passes durchzuführen, ein sehr mächtiges Werkzeug.

[13] vgl. Tabelle 16, rechte Spalte

### 5.2.3 Ein alternatives Interface

Als Mittelweg zwischen dem hardware-nahen und schlecht benutzbaren DirectX8-Interface und der abstrakten benutzerfreundlichen Stanford Shading Language, die aber nicht die volle Funktionalität der neuen Hardware auszunutzen weiß, habe ich ein alternatives Interface entworfen. Beim Entwurf dieses Interfaces ging es mir vor allen Dingen darum, die abgekapselte Entwicklung von GPU-Programmen, sowie die Austauschbarkeit, Kombinierbarkeit und Wiederverwendbarkeit dieser Programme zu fördern.

#### *5.2.3.1 Das Problem des DX8-Interfaces*

Die wesentlichen Grenzen des DX8-Shader-Programmmodells resultieren aus den Beschränkungen der zu steuernden Hardware: der Anzahl der Register, der Programmlänge, der Komplexität der Instruktionen und der Funktionalität der Instruktionen.

Dass eine Flexibilisierung in all diesen Bereichen auch die Entwicklung besserer Interfaces erlauben würde, erwähnte ich bereits.. Aber viele der von mir in Abschnitt 5.2.1 aufgeführten Kritikpunkte lassen sich nicht allein auf die Schwächen der Grafik-Hardware schieben, sondern auf die Art und Weise, wie die neuen Komponenten in die DX8-Umgebung eingebettet wurden.

Es gibt keinen Mechanismus, der die im Pixel- bzw. Vertex-Shader-Programm definierten auszuführenden Aufgaben auf die verschiedene Systemteile verteilt (vgl. Abbildung 53). Denkbar wäre z.B. ein System, dass zwar *einen Teil* der Aufgaben auf der VPU bzw. FPU, *andere Teile* aber auf anderen GPU-Komponenten oder der CPU ausgeführt.[14]

Statt dessen läuft ein Vertex-Shader-Programm vollständig auf der VPU und ein Pixel-Shader-Programm vollständig auf der FPU ab (immer angenommen, dass solche Hardware-Komponenten vorhanden sind).

---

[14] Dieser Aussage ist ein Vereinfachung: Es gibt sowohl im Vertex- als auch im Pixel-Shader-Programmmodell Versions- und Definitionsanweisungen, die auf der CPU ausgeführt werden (s. Abs. 4.2.1.2). Diese Anweisungstypen haben aber für die Funktionalität eines Vertex- bzw. Pixel-Shader-Programmes nur eine untergeordnete Bedeutung.

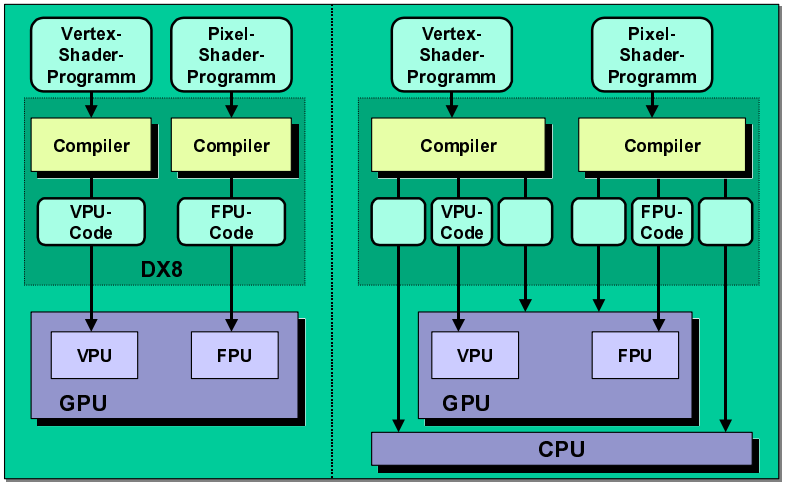



**Abbildung 53: Zusammenhang zwischen DX8-Shadern und VPU/FPU**

Links: Vertex- und Pixel-Shader-Programme werden in DX8 vollständig auf VPU und FPU ausgeführt .

Rechts: Alternativ wäre auch eine Verteilung der Aufgaben auf verschiedene Hardware-Komponenten denkbar gewesen.

Dieser Sachverhalt zieht unter anderem die beiden folgende Konsequenzen nach sich:

1.) Die Funktionalität der Programmmodelle von *Vertex* und *Pixel Shader* entspricht der Funktionalität der zugrundeliegenden VPU- und FPU-Komponenten. (Ich habe in Abschnitt 4.2 gezeigt, dass dies der Fall ist.)

2.) Im DX8-API sind die Programmierung der VPU und FPU und die Verwendung der übrigen Hardware-Komponenten strikt voneinander getrennte Aufgaben. Alle Steuerungsanweisungen, die sich auf andere Hardwarekomponenten beziehen, aber dennoch einen großen Einfluss auf die Arbeit der VPU und FPU haben, liegen außerhalb der Shader-Programme.

Der erste Punkt ist jener, den man durch den Einsatz einer vermeintlichen High-Level-Sprache zu kaschieren versucht. Dem zweiten Punkt scheint dagegen in den neu entwickelten Sprachen wie Cg [Nvid02], keine besondere Bedeutung beigemessen zu werden.

Wie ich im folgenden zeigen werden, lassen sich viele der aufgezeigten Schwächen des DX8-Interfaces durch einen Ansatz, wie ich ihn in die rechte Hälfte von Abbildung 53 zeigt, beseitigen.

#### *5.2.3.2 Grundlegende Entscheidungen*

Beginnen möchte ich zunächst einmal mit der Beseitigung eines Namensmissgriffs – nämlich mit der Bezeichnung „Shader“ für Programme, die auch zur Realisierung von Animationstechniken eingesetzt werden sollen. Ich verwende stattdessen als Überbegriff für alle Programme, die zur Steuerung der VPU und FPU dienen, die allgemeinere Bezeichnung „*Rendering Program*“.
Das zugehörige Interface habe ich dementsprechend Rendering Program Interface oder kurz RP-Interface getauft.

Auch im RP-Interface darf zu jedem Zeitpunkt nur ein Rendering Program aktiv sein. Dies ist ein Tribut an die aktuelle Hardwarearchitektur: In der VPU werden Animation und Schattierungsberechnungen gemeinsam durchgeführt. Theoretisch wäre es denkbar einen Compiler zu entwickeln, der aus einem Animations-Programm und einem Shading-Programm ein einziges VPU-Programm zusammensetzt. Die Knappheit der Ressourcen verurteilt einen solchen Ansatz jedoch zum Scheitern. Erstens verliert der Programmierer den Überblick, über die verfügbaren Ressourcen, und zweitens müssten zu viele temporäre Register für die Speicherung der Zwischenergebnisse (z. B transformierte Normale, transformierte Position, etc.) verwendet werden.

#### *5.2.3.3 Zusammenführung von eng gekoppelten und logisch zusammenhängenden Komponenten*

Ich habe in Abschnitt 5.2.1.4 die Trennung von Vertex und Pixel Shadern kritisiert und in Abschnitt 5.2.1.6 die starke Kopplung zwischen DX8-Shadern und anderen Interface-Komponenten. Die Entwicker von Cg bieten in ihrer Sprache die Möglichkeit VPU- und FPU- Programmierung in einem Programm zu vereinen. Dieser Schritt allein ist aber nicht konsequent genug. Konsequent wäre es, auch den zweiten Schritt zu tun und Teile der Steuerungsfunktionen (wie

beispielsweise das Setzen der Konstante-Register), sowie Teile anderer Interfaces, die mit dem Effekt von FPU und VPU eng gekoppelt sind (wie beispielweise das Festlegen von Texturfilterungs-Modi) zu einem einzigen Modell zu verschmelzen. So erhielte man eine einzige Komponente, die funktional abgeschlossen wäre und über eine minimale Schnittstelle verfügte. Dies ist der Ansatz, dem ich im RP-Interface nachgehe.
Die Kanditaten für eine Zusammenlegung, habe ich bereits in meiner Kritik aufgezählt (s. Abschnitt 5.2.1.6). Dies sind:

- **uniforme Berechnungen**: Berechnungen, die notwenig sind, um den Wert eines Konstanten-Registers zu ermitteln.
- **Vertex-Shader-Deklaration**: Die Zuweisung der Vertex-Buffer-Inhalte an Vertex-Register,
- **Texturgenerierung**: Die Generierung von Texturen, die für FPU-Berechnungen notwendige Funktionstabellen speichern.
- **Definition der Texturstufen**: Die Zuweisung einer Textur an eine bestimmte Texturstufe und die Festlegung der Texture-Sampling-Modi

In einem Rendering Program sind all diese Funktionen zusammen mit den eigentlichen VPU- und FPU-Programmmodellen verknüpft. Auf diese Weise kann die Schnittstelle zur Umgebung auf ein Minimum reduziert werden.

Als weitere Funktionalität wird die Steuerung des Triangle Setups in das Rendering Program aufgenommen. Denn diese Hardware-Komponente ist für die Bestimmung der Fragment-Attribute aus den Ausgabewerten der VPU verantwortlich. Für die Arbeit der FPU ist es entscheidend, ob die Fragment-Attribute durch Interpolation berechnet werden (Gouraud Shading) oder nicht (Flat Shading).

#### *5.2.3.4 Einführung von Rendering-Programm-Typen*

Um die Kombinierbarkeit und Wiederverwendbarkeit zu fördern enthält das RP-Interface verschiedene *Typen* von Rendering Programs.

Die Grundidee ist folgende: Einem Programm-Typ wird eine klar definierte Aufgabe im Rendering-Prozess zugewiesen, und jedem Programm-Typ steht eine Auswahl von vordefinierten Attributen zur Verfügung, die zur Erfüllung seiner Aufgabe essentiell sind. Die verbleibenden Register können zur Speicherung programm-spezifischer Attribute genutzt werden.
Teile der Vertex- und Konstanten-Register werden also in Abhängigkeit vom Programm-Typ mit einer festgelegten Bedeutung versehen, so dass der Programmierer sich bei diesen Registern darauf verlassen kann, dass in Register *A* das Attribut *A* gespeichert ist. Auf der anderen Seite weiß der Programmierer der Applikation bei der Verwendung eines bestimmten Programm-Typs, welche Attribute in jedem Fall von ihm bereitgestellt werden müssen, damit das Programm korrekt arbeiten kann. Desweiteren kann er ein Programm relativ problemlos gegen ein anderes gleichen Typs austauschen, da er nur die programm-spezifischen Attribute anpassen muss.
An dieser Stelle spielt der Effekt, der die im letzten Abschnitt beschriebene Zusammenlegung eine wichtige Rolle. Denn dadurch, dass uniforme Berechnungen nun innerhalb des Rendering-Programms durchgeführt werden können, kann die Anzahl der vordefinierten und programm-spezifischen Attribute auf ein Minimum reduziert werden.

Für die Typisierung von Rendering Programs, also die Bildung von Äquivalenzklassen, habe ich zwei Kriterien ermittelt:

1. der Effekt bzw. der Aufgabe des Programms,
2. die vom Programm benötigten Attribute.

In RenderMan findet sich, eher pure Einteilung nach Aufgaben. Die Liste der zur Verfügung stehenden vordefinierten Attributen wurde der Aufgabe angepasst, aber nicht umgekehrt. Das macht die Verwendung von RenderMan sehr intuitiv.
Aber einen einzigen Programm-Typ für den gesamten Bereich der Objekt-Lichtquellen-Interaktion (Surface Shader) könnte in Echtzeitanwendungen, aufgrund der knappen Register-Anzahl und der begrenzen Bandbreite, zum Problem werden: Beispielsweise benötigen die Per-Fragment-

Beleuchtungsverfahren, in aller Regel für jeden Vertex *eine Tangente* als Attribut (vgl. Abschnitt 4.4.2.2). Für Beleuchtungsrechnungen, die ausschließlich per Vertex durchgeführt werden, ist dieses variierende Attribut nicht erforderlich, und kann daher eingespart werden. Dem Programmierer sollte Raum für solche Optimierungen gegeben werden, und daher eine Einteilung in Programm-Typen vor allen Dingen *ausgehend* von den benötigten Attributen (insbesondere den variierenden) durchgeführt werden.
Eine andere Möglichkeit diesem Problem zu begegnen, wäre die Ausführung einer automatischen Voranalyse, bei der nur die Attribute an den Grafikchip weitergereicht werden, auf die das Rendering-Programm am Ende auch zugreift. Aber ich bevorzuge den anderen Weg, da damit der Erstellung und Speicherung von unnötigen Daten auf dem Host weitestgehend vorgebeugt wird.

Ich habe mich bei der Typisierung für den Aufbau einer Programm-Typen-Hierarchie nach dem Vorbild des *Vererbungsmechanismus* in der objektorientierten Programmierung entschieden. Abbildung 54 zeigt die Wurzel dieser Hierarchie.

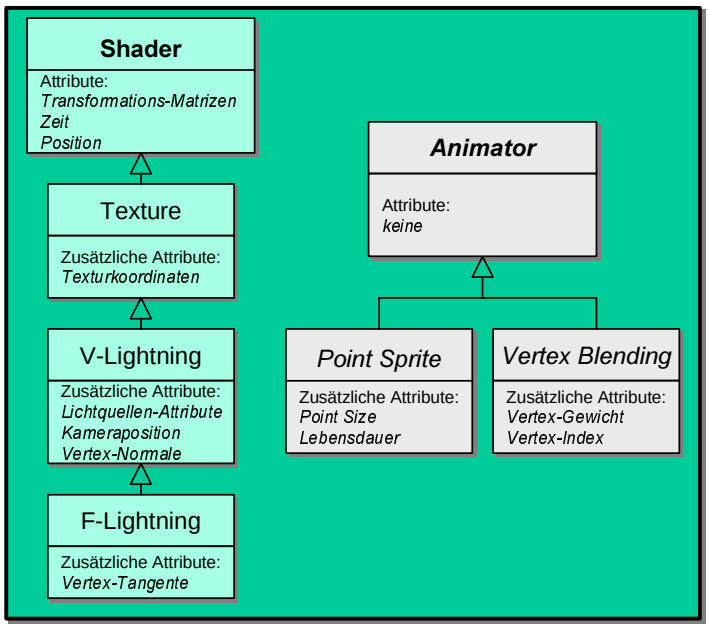



**Abbildung 54: Eine Rendering-Program-Typisierung**

Zunächst unterscheidet das Interface zwischen *Shading* und *Animation*, repräsentiert durch die Ober-Typen „Shader“ und „Animator“. Diese Trennung ist eine Konsequenz aus der von mir gemachten Beobachtung, dass Vertex Shader, die zu Animationszwecken eingesetzt werden, sich nicht über Multi-Pass-Rendering mit beliebigen anderen DX8-Shadern kombinieren lassen, sondern ausschließlich mit DX8-Shadern, welche die Geometrie auf dieselbe Weise transformieren. Die Unterscheidung hat den folgenden Vorteil: Ist ein Rendering Program vom Typ „Shader“ kann der Programmierer es bedenkenlos mit anderen Shadern kombinieren.
Damit dies wirklich garantiert ist, und ein Programmierer einen Shader nicht für Animationen missbraucht, wird bei diesem Programm-Typ die Transformation der Geometrie in das Projektions-Koordinatensystem *automatisch* durchgeführt, und der Programmierer hat *keinen Schreibzugriff* auf das Ausgaberegister für die Position (`HPOS`).

Ein wesentlicher Aspekt meines Modells ist die *Zweifachvererbung:* Aus der Verbindung von *einem Shader* und *einem Animator* können kombinierte Typen gebildet werden, die sowohl schattieren als auch animieren.

**Beispiel 13:** Ein *F-Lightning-Animator* beispielsweise benötigt die Attribute eines F-Lightning-Shaders, kann darüber hinaus aber auch für Animationen eingesetzt werden. Der DX8-Shader zur Simulation der Wasserbewegungen (siehe Abschnitt 4.7.1) wäre ein Vertreter dieses Programm-Typs.

Ein solcher Mechanismus ist notwendig, denn auch in meinem Modell darf wie bereits erwähnt zu jedem Zeitpunkt nur ein Programm aktiv sein, und ein animiertes Objekt ohne Schattierungsberechnung wäre grundsätzlich schwarz. So gesehen handelt es sich bei den Animatoren, um bei den Bezeichnungsschemata der objektorientierten Programmierung zu bleiben, um *abstrakte* Typen[15] – sie dienen nur zur Vererbung von Eigenschaften, nicht zur direkten Anwendung.

[15] Deshalb sind in Abbildung 54 (S. 153) deren Namen der UML-Notation entsprechend kursiv gedruckt.

Daher ist ein *Animator* in der Praxis, in jedem Fall ein Kind der Oberklasse „Shader“ und verfügt über dessen Attribute.

Das System ist selbstverständlich noch erweiterbar. Bei der Implementierung meiner in Abschnitt 4.6 vorgestellten Demonstratoren habe ich beispielweise aus dem Typ „Texture“ einen neuen Typ „Imager Shader“ abgeleitet. Dieser hat zwar dieselbe Attributliste wie seine Oberklasse, aber aufgrund der speziellen Aufgabe eines Imager Shaders, nämlich der Bildbearbeitung, erschien mir eine weitere Differenzierung sinnvoll.

#### *5.2.3.5 Zusicherung und Absicherung*

Um die Programmierung der Rendering Programs und deren Anwendung weiter zu erleichtern, werden vordefinierte uniforme Attribute vom RP-Interface mit Default-Werten versehen, oder, falls möglich grundsätzlich automatisch gesetzt. Letzteres ist beispielsweise beim Attribut „Zeit“ möglich.
Desweiteren kann der Programmierer innerhalb des Rendering Programs Default-Werte für seine programm-spezifischen Attribute angeben, so dass der Benutzer des Shaders bei dessen Initialisierung nicht gezwungenermaßen all diese Werte angeben muss.

#### *5.2.3.6 Einführung eines □□□□□□□□□□□-Konstruktes*

Zur Beseitigung der in Abschnitt 5.2.1.3 beschriebenen Lichtquellenproblematik wird wie in RenderMan ein `illuminance`-Konstrukt eingeführt.
Dieses Konstrukt funktioniert wie ein for-Schleife durch die Anzahl der verschiedenen Lichtquellen. Der Programmierer des Shaders kann nur innerhalb dieses Konstruktes auf die Lichtquellenattribute zugreifen.
Zur Optimierung der Performance kann der Programmierer für alle drei Lichtquellentypen (`directed`, `point` und `spot`) separate Programmfragmente schreiben.

```
Illuminance{
        case (directed):
        …
        case (point):
        …
        case (spot):
        …
        }
```

Realisiert werden kann ein solches Modell auf der aktuellen GPU-Generation durch eine Runtime-Kompilierung. Damit diese Kompilierung nicht für jeden Frame durchgeführt werden muss, schlage ich das folgende einfache Modell vor:
Dem Programmierer der Applikation steht ein Lichtquellen-Array zur Verfügung, in das er die zur Zeit aktiven Lichtquellen einfügt. Veränderungen an den Attributen der Lichtquellen erfordern keine Neukompilierung. Nur falls der Programmierer eine Lichtquelle *hinzufügt*, *entfernt* oder durch eine Lichtquelle eines *anderen* Typs *ersetzt*, muss das Rendering Program erneut kompiliert werden. Es sollte für den Programmierer kein Problem sein über weite Strecken mit einer einzigen Zusammensetzung aus Lichtquellentypen auszukommen, da er ja, wie gesagt, deren Attribute (Position, Farbe, Strahlungsrichtung, etc.) beliebig verändern kann.
Mit diesem Mechanismus lässt sich auch das Problem der begrenzten VPU- und FPU-Programmlänge und Ressorcen besser regeln: Das Array dient gleichzeitig als Prioritätenliste. Bei der Runtime-Kompilierung wird überprüft, ob die Einflüsse der angegebenen Lichtquellen innerhalb der erlaubten Grenzen berechnet werden können. Falls nicht, werden Lichtquellen niedrigerer Priorität weggelassen. Das ist zwar unschön, aber besser als ein Fehler bei der Kompilierung des VPU– bzw. FPU-Codes, wie es der Cg-Compiler handhabt.
Aus der Tatsache, dass die Sprache Cg ebenfalls Runtime-Kompilierung einsetzt (aber für andere Zwecke), und dass die Entwickler der Sprache den Einsatz dieses Mechanismus ausdrücklich empfehlen [Nvid02d, S.29] entnehme ich, dass ein solches Modell in der Praxis realisierbar wäre.

#### *5.2.3.7 Möglicher Programmaufbau*

Ein möglicher Aufbau für ein Rendering Program wäre:

```
<Programm-Typ> <Name> (<programmspezifische uniforme Attribute>)
{
        <Default-Werte der programmspezifischen uniformen Attribute>
        <Register-Zuweisung der programmspezifischen variierenden Attribute>

        // Initialisierung:
        <Initialisierung und Zuweisung der Texturen>
        <Setzen der Texture-Sampling-Modi>

        // Berechnung:
        <Berechnung der uniformen Variablen>
        <Vertex-Programm>
        <Interpolations-Modus>
        <Pixel-Programm>
}
```

#### *5.2.3.8 Demonstration der Vorteile des Ansatzes*

Ich habe für die Entwicklung meiner Demonstrationsprogramme ein Rahmenwerk entwickelt, das sich an dem eben skizzierten Modell orientiert. Dieses implementiert aber nur einige Konzepte, des von mir gerade skizzierten RP-Interfaces.

Mir war es bei der Umsetzung vor allen Dingen wichtig die Vorzüge der Abkapselung, Zusicherung und Absicherung zu demonstrieren. Ich habe aber keinen eigenen Compiler entwickelt, so dass die Entwicklung einer eigenen Syntax und insbesondere die Einführung des Illuminance-Kontruktes noch offen sind.

Das Rahmenwerk baut auf dem DirectX8-Interface auf und nutzt die Vertex- und Pixel-Shader-Interfaces. Es ist folgendermaßen augebaut:

Eine Klasse „RenderingProgram“ dient als Oberklasse für alle Rendering-Program-Typen. Sie enthält Methoden zur Generierung von Vertex und Pixel Shadern aus vorkompilierten Dateien sowie eine virtuelle Methode „setActive()“, mit der ein Rendering Program aktiviert wird.

Von dieser Klasse werden per Vererbungsmechanismus die verschiedenen Programm-Typen abgeleitet. Sie nutzen die Methode SetActive() zur Durchführung aller Berechnungen, die von Frame zu Frame variieren können, und zum Setzen aller Flags. Dies kann z.B. die Berechnung von uniformen Attributen sein, die Definition von Texturstufen oder die Aktivierung eines Vertex oder Pixel Shaders.
Im Konstruktor der abgeleiteten Klassen werden Berechnungen durchgeführt, die zur einmaligen Initialisierung notwendig sind. Zum Beispiel die Berechnung einer Textur, das Setzen der Vertex-Shader-Deklaration oder die Erzeugung von Vertex oder Pixel Shadern. Die Parameter des Konstruktors entsprechen den programm-spezifischen Attributen.

**Beispiel 14:** Der von mir entwickelte Bump Environment Shader erhält als programmspezifische Attribute eine Height Map und eine Environment Map. Diese beiden Texturen werden in der SetActive-Methode mit den im Vertex- und Pixel-Shader-Programm verwendeten Texturstufen verknüpft. Der Programmierer der Applikation muss sich daher nicht um den internen Aufbau der Vertex und Pixel Shader kümmern und der Entwickler des Rendering-Programms weiss, dass die korrekte Verküpfung zwischen Textur und Texturstufe gewährleistet ist.

Das RP-Interface ist für die Kommunikation zwischen Applikation und Rendering Program zuständig. Es setzt alle vordefinierten uniformen Attribute wie beispielsweise Transformationsmatrizen, Lichtquellenattribute, Materialattribute, Zeit und häufig verwendete Konstanten. All diese uniformen Attribute haben Default-Werte. Der Programmierer kann aber auf einige Attribute durch die Methode „SetAttribute“ Einfluss nehmen.

Außerdem sorgt das RP-Interface dafür, dass den globalen Flags der Rendering-Pipeline beim Wechseln des Rendering Programs Default-Werte zugewiesen werden. So werden Seiteneffekte verhindert, die entstünden, wenn das vorherige Rendering Program Veränderungen an den Flags vorgenommen hätte. Dem Programmierer des Rendering Programs wird also eine bestimmte Konfiguration der Rendering Pipeline und das Vorhandensein verschiedener vordefinierter Attribute zugesichert.

Das vom mir implementierte Programm „Shader Demonstration“ nutzt dieses Rahmenwerk. Auf einfache Weise kann in diesem Programm in den Menüs „Object Shader“ und „Imager Shader“ zwischen verschiedenen Schattierungsmethoden gewählt werden (vgl. Abbildung 55).

Diese einfache Austauschbarkeit findet sich auch im Programm-Code: Von den Rendering Program werden unter Angabe der programm-spezifischen Attribute Instanzen erzeugt. Ist dies erledigt, kann das Rendering Program zu einem beliebigen Zeitpunkt durch den Befehl „setActive()“ aktiviert und zur Bildgenerierung verwendet werden:

```
setActive(RenderingProgram P1);
drawPrimitives(VertexBuffer1);
setActive(RenderingProgram P2);
drawPrimitives(VertexBuffer2);
…
```

Ich habe die verwendeten Rendering Programs zu Bibliotheken zusammengefügt und konnte sie auf diese Weise bequem importieren.


**Abbildung 55: Das Programm Shader-Demonstration**

## 5.3 Fazit

### 5.3.1 Was hat die neue GPU-Generation gebracht?

Mit der neuen GPU-Generation hat man versucht, die Inflexibilität seiner Vorgänger zu beseitigen, indem man festverdrahtete Hardware-Komponenten durch benutzerprogrammierbare Prozessoren ersetzte. Die neue Architektur erlaubt Programmierbarkeit auf zwei Ebenen: auf Vertex- und auf Fragment-Ebene. Die neue Vertex-Verarbeitungseinheit ersetzt die ehemalige Transform&Lightning-Komponente, und die neue Fragment-Verarbeitungseinheit, die ehemalige Multi-Texturing-Komponente früherer Hardware-Architekturen.

Dieser Schritt hat keine negativen Auswirkungen auf die Verarbeitungsgeschwindigkeit, und bringt damit eine eindeutige Verbesserung gegenüber den alten Architekuren: Durch den Aspekt der Programmierbarkeit können nun physikalisch-basierte und nicht-photorealistsiche Beleuchtungsverfahren, verschiedenste Animationstechniken und einfache Formen der Bildbearbeitung und Texturgenerierung vollständig auf der Grafik-Hardware ausgeführt werden. Solche Aufgaben konnten zuvor nur mit CPU-Untertützung berechnet werden. Die neuen Grafikchips bringen daher eine Entlastung für die CPU und es ist nun möglich massiven Gebrauch von den oben genannten Techniken zu machen, da sie nicht mehr in Konkurrenz mit den anderen Aufgaben der CPU stehen.

Desweiteren führt die gewonnene Flexibilität dazu, dass zur Realisierung vieler Effekte weniger Pipeline-Durchläufe benötigt werden. Dies entschäft zum einen die Problematik der knappen Bandbreite zwischen Grafik-Chip und Frame Buffer. Zum anderen können präzisere Resultate erzielt werden, da mit jedem Pass weniger die potentielle Gefahr des Datenverlustes sinkt, der aufgrund der gleichbleibend geringen Präzision der Fragment-Verarbeitungseinheiten von 8-Bit oder 9-Bit pro Kanal besteht. Auch diese Aspekte führen zu einer potentiellen Erhöhung der Bildqualität.

Die Leistungsfähigkeit der neuen Hardware hat aber auch ihre Grenzen. Da die nicht auf die Resultate von VPU-Programmen zugegriffen werden kann, muss

sich die Geometrie-Verarbeitung grundsätzlich in der gegebenen Schranke von 96 erlaubten Instruktionen bewegen. Dies führt, zusammen mit der Unfähigkeit bei der Verarbeitung eines Vertex auf die Attribute anderer Vertices zugreifen zu können und dem eingeschränkten Instruktionssatz, in dem jegliche Kontrollstrukturen fehlen, dazu, dass viele in der Grafischen Datenverarbeitung verbreitete Algortihmen und Verfahren wie beispielsweise die diskrete Fourier-Transformation nicht auf der GPU ausgeführt werden können.

Die größten funktionalen Schwächen sind noch bei der programmierbaren Fragment-Verarbeitungseinheit zu finden. Diese ist eher als eine Erweiterung der alten Multi-Texturing-Komponente anzusehen, denn als neuer Baustein, was sich beispielsweise an der bereits erwähnten gleichbleibend niedrigen Präzision von 8- oder 9-Bit pro Kanal zeigt. Die Programmlänge für diese Komponente ist so beschränkt, dass eine Beleuchtungsrechung per Fragment nur in Einzelfällen möglich ist. Die Generierung von prozeduralen Texturen ist wie bisher nur durch mehrere Rendering-Passes realisierbar und komplexere Funktionen wie „Perlin Noise“ [Perl85] oder Exponentialfunktionen stehen nicht zur Verfügung.

Der Programmierer muss sich also in erster Linie an den Gegebenheiten der Hardware orientieren und kann erst in zweiter Linie an die Anforderungen des zu realisierenden Effektes denken.

Diese einzelnen funktionalen Schwächen sind aber vergleichsweise gering einzustufen gegenüber einer anderen Problematik: Die verbesserte Funktionalität der neuen Hardware-Generation ist in der Praxis nur schwer nutzbar. Denn man hat beim Design der Hardware nicht ausreichend darauf geachtet, dass sich diese auch zu einer brauchbaren Benutzerschnittstelle abstrahieren lässt. Die Mängel des DirectX8-Interfaces spiegeln diesen Sachverhalt wider.

Als Hauptursache für diesen Misstand habe ich in diesem Kapitel zwei Punkte ermittelt: Die fehlende Möglichkeit auf die Resultate der VPU zugreifen zu können, und das Fehlen einfacher Kontrollstrukturen im VPU-Programmmodell.

Dennoch lassen sich nicht alle Schwächen von DirectX8 auf Hardware-Mängel zurückführen. Ich habe mit der Präsentation eines alternativen Interfaces gezeigt, durch welche Maßnahmen die abgekapselte Entwicklung, sowie die Austauschbarkeit und Kombiniertbarkeit von GPU-Programmen gefördert werden

könnten. Durch das von mir vorgestellte System, wäre der Aufbau von GPU-Programm-Bibliotheken und somit eine hoher Grad an Wiederverwendbarkeit möglich.

### 5.3.2 Die kommende GPU-Generation und die Zukunft der API-Entwicklung

Die kommene GPU-Generation wird einige der von mir ermittelten Schwächen beseitigen oder zumindest entschärfen. Der vor einigen Wochen erschienene Radeon9700-Chip (s. Abschnitt 3.2.4.4) enthält eine Vertex-Verarbeitungseinheit, die über Kontrollstruktur-Instruktionen verfügt: Sowohl Schleifen und bedingte Sprünge, als auch die Definiton von Prozeduren sind erlaubt. Außerdem enthält der Chip eine Fragment-Verarbeitungseinheit, die eine Präzision von 16 und 32 Bit pro Kanal erlaubt, und deren Programmlänge auf maximal 32 Texturinstruktionen und maximal 64 arithmetische Instruktionen erweitert wurde.
Damit hat man einen wesentlichen Fortschritt im funktionalen Bereich erzielt und bietet durch die Einführung von Kontrollstrukturen die Möglichkeit, allgemeingültigere VPU-Programme zu entwerfen. Dies ist insbesondere zur Entschärfung der Lichtquellenproblematik interessant.
Auf Anfrage teilte man mir mit, dass auf die Resulate der VPU aber weiterhin kein Zugriff erlaubt ist. Damit bleibt das von mir beschriebene VPU-Multi-Pass-System, das eine elegante Abstraktion ohne funktionale Verluste erlauben würde, unrealisierbar.
Will man die Entwicklung eines Interfaces unterstützen, dass einerseits die volle Funktionalität der GPU auszunutzen weiß, andererseits aber auch eine abstrakte Oberfläche zu bieten hat, wie man sie von Software-Rendering-Systemen kennt, dann sollte man diesen Mangel in Zukunft beseitigen. Inwieweit der nächste Chip von NVIDIA in dieser Hinsicht einen Ausweg bietet, ist noch offen. Spätestens der übernächste Produktzyklus aber wird, darf man den Ankündigungen von NVIDIA Glauben schenken [NVID02g], einen Chip hervorbringen, der die Leistungsfähigkeit des Radeon9700-Chip insbesondere was die FPU betrifft noch weit übertreffen wird.

Es ist eine große Herausforderung für die API-Entwickler, für all diese verschiedenen, sich ständig weiterentwickelnden programmierbaren Grafikchip-Generationen ein einheitliches Interface anzubieten. Noch gibt es keine befriedigende Lösung für dieses Problem. DirectX8 ist zu hardware-nah und bietet daher nur wenig Nachhaltigkeit, Cg ist zu sehr auf die Produkte von NVIDIA zugeschnitten, und die Stanford Shading Language ist zwar ein gut funktionierendes hardware-unabhängiges Interface, aber es nutzt nicht die volle Funktionalität der aktuellen Hardware. Außerdem ist noch unklar, wie man mit der Funktionalität der kommenden GPU-Generation umgehen wird. Erweitert man beispielsweise den Befehlssatz für Per-Vertex-Operationen um Kontrollstrukturen, ist eine Kompabilität mit der aktuellen Hardware nicht mehr gewährleistet. Tut man es nicht, verzichtet man auf Teile der verbesserten Funktionalität.
OpenGL2.0 (s. Abschnitt 3.3.2) könnte hier in Zukunft Abhilfe schaffen: Einerseits eine hardware-unabhängige und nachhaltige High-Level-Sprache bereitszustellen und andererseits innerhalb dieser Sprache auch die Möglichkeit zu geben, mit hardware-nahen Programmmodellen arbeiten zu können, ist meines Erachtens ein vielversprechender Ansatz.

In nächster Zeit wird es meines Erachtens noch nicht gelingen, die Interface-Entwicklung von der Hardware-Entwicklung vollständig loszukoppeln. Doch entwickelt sich die Grafik-Hardware weiterhin so rapide weiter, dann wird man dieses Ziel mit der zunehmenden Flexibilisierung der GPU-Prozessoren und der anderen Hardware-Komponenten in nicht allzu ferner Zukunft erreichen.

# 6 Zusammenfassung

Ich habe in dieser Arbeit einen allgemeinen Überblick über die Architekur und Leistungsfähigkeit der neuen GPU-Generation gegeben, deren Stärken und Schwächen aufgezeigt und Verbesserungsvorschläge diskutiert.

Meine Analyse hat ergeben, dass die neue Hardware deutlich funktionale Vorteile gegenüber ihren Vorgängern bietet. Sie ermöglicht eine wesentlich größere Flexibilität in der Schattierung von Objekten, ohne dabei an Verarbeitungsgeschwindigkeit einzubüßen, so dass die Komplexität der verwendeten Geometrien gleichbleibend hoch gewählt werden kann.
Ich habe anhand einiger Leistungsdemonstratoren dargestellt, wie die Möglichkeiten der Hardware ausgenutzt werden können. Unter anderem habe ich eine Wasser-Simulation entwickelt sowie einige nicht photorealistische Beleuchtungs- und Schattierungsverfahren. Alle diese Demonstratoren laufen vollständig auf der Grafik-Hardware und können daher in Anwendungen eingesetzt werden, ohne die CPU zu belasten.
Dennoch hat sich gezeigt, dass das Versprechen, eine „unendliche Anzahl“ von Effekten sei mit der neuen Technik realisierbar, zu hochgegriffen ist. Schuld daran sind die zahlreichen Restriktionen, denen die Programmmodelle der GPU-Prozessoren unterworfen sind: die Programmlängen, die Anzahl der verfügbaren Register und die Funktionalität des Befehlssatzes sind stark beschränkt.
Aber einen noch gravierenderen Mangel habe ich in dieser Arbeit nachgewiesen: Die funktionalen Vorteile, welche die neue Hardware bietet, sind in ihrem vollen Umfang in der Praxis nur schwer nutzbar. Denn die nun programmierbare Hardware stellt auch höhere Anforderungen an die sie steuernden Interfaces. Statt wie bisher im Wesentlichen mit Befehlen zum Setzen verschiedener Flags arbeiten zu können, müssen diese den Programmierern eine Möglichkeit bieten, auch die Programmmodelle der GPU-Prozessoren zu steuern. Die große Schwäche der GPU-Programmmodelle ist aber, dass sie sich nur schwer abstrahieren lassen, so dass als Konsequenz zwei Arten von Interfaces entwickelt wurden: Die einen orientieren sich direkt an den GPU-Programmmodellen und

den anderen Hardwarekomponenten und sind deshalb in vielerlei Hinsicht für eine effiziente Entwicklung von Applikationen ungeeignet. Die anderen garantieren durch eine dicke Abstraktionsschicht Hardware-Unabhängigkeit und bieten ein benutzerfreundliches Interface, verzichten dadurch aber auf einen Teil der Funktionalität.
Ich habe als Mittelweg zwischen beiden Extremen ein alternatives Interface entworfen, das nicht auf Teile der Funktionalität verzichtet und dennoch eine abgekapselte Entwicklung von GPU-Programmen ermöglicht sowie die Austauchbarkeit und Kombinierbarkeit dieser Programme erleichtert. Die von mir im Ansatz vollzogene Implementierung des Systems demonstriert die Vorzüge dieses Ansatzes.
Dennoch ist auf lange Sicht eine weniger hardware-nahe Schnittstelle zu bevorzugen, welche eine nachhaltige Oberfläche bietet, die weitestgehend unabhängig von den Weiterentwicklungen der Grafik-Hardware ist. OpenGL2.0 könnte hierfür in Frage kommen.

Solange sich aber noch kein neuer Standard abzeichnet, werden die Software-Entwickler meines Erachtens nicht von den gewohnten hardware-nahen Interfaces wie DirectX8 und OpenGL1.4 abrücken. Wegen der schlechten Benutzbarkeit in diesen Interfaces ist es daher zumindest fraglich, ob die Funktionalität der neuen GPUs breite Verwendung finden wird. Denn der Zeitdruck, unter dem Software-Unternehmen in aller Regel stehen, wird für die Akzeptanz einer nur unter erheblichem Aufwand voll ausschöpfbaren Technik nicht förderlich sein. Meiner Einschätzung nach wird man die neuen Möglichkeiten vor allen Dingen zur Erzielung vereinzelter Spezialeffekte verwenden. Damit bliebe die Hardware aber unter ihren Möglichkeiten, da sie tatsächlich in fast allen Bereichen der Bildgenerierung funktionale Vorteile bietet. Zu einer stärkeren Nutzung der neuen Technik könnte aber die Tatsache führen, dass Microsofts Spielekonsole „X-Box“ einen Grafikchip der neuen Machart enthält.
Trotz dieser Problematik ist man mit der Einführung benutzerprogrammierbarer Hardware unzweifelhaft den richtigen Weg gegangen und hat mit diesem Schritt die Lücke zwischen Echtzeit-Bildgenierungs- und Software-Rendering-Systemen

weiter verkleinert. Und die kommende Grafikchip-Generation mit ihren flexibleren und besser abstrahierbaren GPU-Programmmodellen wird aller Vorraussicht nach einen Großteil der in dieser Arbeit kritisierten Mängel relativieren.

# Anhang

## Texture-Shader-Instruktionen

| | Programm | Effekt | Unit Result |
|---|---|---|---|
| 1 | TEXTURE_<br>1D | Texture Sampling:<br>Ermittelt (*r*, *g*, *b*, *a*) durch Zugriff<br>auf *1D-Textur* über die Koordinate (*s*/*q*). | (*r*, *g*, *b*, *a*) |
| 2 | TEXTURE_<br>2D | Texture Sampling:<br>Ermittelt (*r*, *g*, *b*, *a*) durch Zugriff<br>auf *2D-Textur* über die Koordinaten (*s*/*q*, *t*/*q*). | (*r*, *g*, *b*, *a*) |
| 3 | TEXTURE_<br>3D | Texture Sampling:<br>Ermittelt (*r*, *g*, *b*, *a*) durch Zugriff auf<br>*3D-Textur* über die Koordinaten (*s*/*q*, *t*/*q*, *r*/*q*) | (*r*, *g*, *b*, *a*) |
| 4 | TEXTURE_<br>RECTANGLE_NV | Texture Sampling:<br>Ermittelt (*r*, *g*, *b*, *a*) durch Zugriff auf<br>*Rectangle-Textur*<br>über die Koordinaten (*s*/*q*, *t*/*q*) | (*r*, *g*, *b*, *a*) |
| 5 | TEXTURE_<br>CUBE_MAP_ARB | Texture Sampling:<br>Ermittelt (*r*, *g*, *b*, *a*) durch Zugriff<br>auf *Cube Map* über die Koordinaten (*s*, *t*, *r*) | (*r*, *g*, *b*, *a*) |
| 6 | NONE | Generiert (0, 0, 0, 0) als Ausgabe | (0, 0, 0, 0) |
| 7 | PASS_THROUGH_NV | Ausgabe der Texturkoordinaten:<br>Konvertiert Texturkoordinaten (*s*, *t*, *r*, *q*) direkt in einen RGBA-Wert: (*r*, *g*, *b*, *a*) ergibt sich aus (*s*, *t*, *r*, *q*) durch Klemmen der Komponenten in den Wertebereich [0,1]. | (*r*, *g*, *b*, *a*) |
| 8 | CULL_FRAGMENT_NV | Verwirft das Fragment, falls eine der Komponenten *s*, *q*, *r*, *t* kleiner als Null ist. | (0, 0, 0, 0) |

**Tabelle 18: Texture-Shader-Programme (1) –**
**Unbedingte (einstufige) Programme [Nvid02b][Domi01]**

Grüne Felder - Standard-Texture-Sampling, gelbe Felder – Spezialfunktionen.

| | Programm | Effekt | Unit Result |
|---|---|---|---|
| 9 | *i*: TEXTURE_2D<br>*i*+1: OFFSET_TEXTURE_2D_NV | **Versetzen der Texturkoordinaten:**<br>Seien $k_0$, $k_1$, $k_2$, $k_3$ benutzerdefinierte Konstanten.<br>- Bestimmt ($r_i$, , $g_i$, $b_i$, $a_i$), durch Zugriff auf eine *2D-Textur* (Textur von Stufe *i*) mit Koordinaten ($s_i$, $t_i$).<br>- Bestimmt *ds*, *dt* durch Klemmen von $r_i$, $g_i$ in den Bereich [-1,1],<br>- Berechnet:<br>$s' := s_{i+1} + k_0 * ds + k_2 * dt$<br>$t' := t_{i+1} + k_1 * ds + k_3 * dt$<br>- Bestimmt (*r*, *g*, *b*, *a*), durch Zugriff auf eine *2D-Textur* (Textur von Stufe *i+1*) mit Koordinaten (*s'*, *t'*). | *i* : (0, 0, 0, 0)<br>*i*+1: (*r*, *g*, *b*, *a*) |
| 10 | *i*: TEXTURE_2D<br>*i*+1: OFFSET_TEXTURE_2D_SCALE_NV | **Versetzen und Skalieren der Texturkoordinaten:**<br>Seien $k_{scale}$, $k_{bias}$ benutzerdefinierte Konstanten.<br>- Die selben Berechnungen wie bei (9).<br>- Berechnet zusätzlich: $m := k_{scale} * b_i + k_{bias}$<br>- und (*r'*,*g'*,*b'*):= (*m***r*, *m***g*, *m***b*). | *i* : (0, 0, 0, 0)<br>*i*+1: (*r'*,*g'*,*b'*,*a*) |
| 11 | *i*: TEXTURE_2D<br>*i*+1: OFFSET_TEXTURE_RECTANGLE_NV | Identisch zu (9), nur dass am Ende auf eine *Rectangle-Textur* (Textur von Stufe *i*+1) zugegriffen wird. | *i* : (0, 0, 0, 0)<br>*i*+1: (*r*, *g*, *b*, *a*) |
| 12 | *i*: TEXTURE_2D<br>*i*+1: OFFSET_TEXTURE_RECANGLE_SCALE_NV | Identisch zu (10), nur dass am Ende auf eine Rectangle-Textur (Textur von Stufe *i*+1) zugegriffen wird. | *i* : (0, 0, 0, 0)<br>*i*+1: (*r'*,*g'*,*b'*,*a*) |
| 13 | *i+1*: DEPENDENT_AR_TEXTURE_2D_NV | **Bedingter Texturzugriff:**<br>Resultat der vorherigen Stufe sei: ($r_0$, $g_0$, $b_0$, $a_0$).<br>- Bestimmt (*r*, *g*, *b*, *a*), durch Zugriff auf eine *2D-Textur* (Textur von Stufe *i+1*) mit Koordinaten ($r_0$, $a_0$). | (*r*, *g*, *b*, *a*) |
| 14 | *i+1*: DEPENDENT_GB_TEXTURE_2D_NV | **Bedingter Texturzugriff:**<br>Resultat der vorherigen Stufe sei: ($r_0$, $g_0$, $b_0$, $a_0$).<br>- Bestimmt (*r*, *g*, *b*, *a*), durch Zugriff auf eine *2D-Textur* (Textur von Stufe *i+1*) mit Koordinaten ($g_0$, $b_0$). | (*r*, *g*, *b*, *a*) |

**Tabelle 19: Texture-Shader-Programme (2) – Bedingte (mehrstufige) Programme [Nvid02b][Domi01]**

blaue Felder : Versatz-Texture-Sampling, rote Felder: bedingtes Texture Sampling. Die Werte vor den Doppelpunkten geben die Stufe an, in der das Programm ausgeführt werden kann. Sei $i \in \{1, 2\}$, und seien ($s_j$, $g_j$, $b_j$, $a_j$) die Texturkoordinaten, welche mit Stufe $j$ ($j \in \{1, 2, 3, 4\}$) verknüpft sind.

| | Programm | Effekt | Unit Result |
|---|---|---|---|
| 15 | *i*: DOT_PRODUCT_NV<br>*i*+1: DOT_PRODUCT_<br>TEXTURE_<br>2D_NV | Resultat der vorherigen Stufe sei: $(r_0, g_0, b_0, a_0)$.<br>- Berechnet $s' := (s_i, t_i, r_i)^T \bullet (r_0, g_0, b_0)^T$,<br>$t' := (s_{i+1}, t_{i+1}, r_{i+1})^T \bullet (r_0, g_0, b_0)^T$,<br>- Ermittelt $(r, g, b, a)$ durch Zugriff auf eine *2D-Textur* (Textur von Stufe *i*) mit den Koordinaten $(s', t')$. | *i*: (0, 0, 0, 0)<br>*i*+1: $(r, g, b, a)$ |
| 16 | *i*: DOT_PRODUCT_NV<br>*i*+1: DOT_PRODUCT_<br>TEXTURE_<br>RECTANGLE_NV | Identisch zu (15), nur dass am Ende auf eine *Rectangle-Textur* (Textur von Stufe 2) zugegriffen wird. | *i*: (0, 0, 0, 0)<br>*i*+1: $(r, g, b, a)$ |
| 17 | 1: DOT_PRODUCT_NV<br>2: DOT_PRODUCT_NV<br>3: DOT_PRODUCT_<br>TEXTURE_<br>3D_NV | Resultat der vorherigen Stufe sei: $(r_0, g_0, b_0, a_0)$.<br>Sei $E := (e_x, e_y, e_z)$ eine benutzerdefinierte Konstante.<br>- Berechnet $s' := (s_1, t_1, r_1)^T \bullet (r_0, g_0, b_0)^T$,<br>$t' := (s_2, t_2, r_2)^T \bullet (r_0, g_0, b_0)^T$,<br>$r' := (s_3, t_3, r_3)^T \bullet (r_0, g_0, b_0)^T$,<br>- Berechnet $R := (r_x, r_y, r_z)$ durch Spiegelung von $E$ an $(s', t', r')$,<br>- Bestimmt $(r_3, g_3, b_3, a_3)$, durch Zugriff auf eine *3D-Textur* (Textur von Stufe 3) mit Koordinaten $(r_x, r_y, r_z)$. | 1: (0, 0, 0, 0)<br>2: (0, 0, 0, 0)<br>3: $(r_3, g_3, b_3, a_3)$ |
| 18 | 1: DOT_PRODUCT_NV<br>2: DOT_PRODUCT_NV<br>3: DOT_PRODUCT_<br>TEXTURE_<br>CUBE_MAP_NV | Identisch zu (3), nur dass am Ende auf eine *Cube Map* (Textur von Stufe 3) zugegriffen wird. | 1: (0, 0, 0, 0)<br>2: (0, 0, 0, 0)<br>3: $(r_3, g_3, b_3, a_3)$ |
| 19 | 1: DOT_PRODUCT_NV<br>2: DOT_PRODUCT_NV<br>3: DOT_PRODUCT_<br>CONST_EYE_<br>REFLECT_<br>CUBE_MAP_NV | Resultat der vorherigen Stufe sei: $(r_0, g_0, b_0, a_0)$.<br>Sei $E := (e_x, e_y, e_z)$ eine benutzerdefinierte Konstante.<br>- Berechnet $s' := (s_1, t_1, r_1)^T \bullet (r_0, g_0, b_0)^T$,<br>$t' := (s_2, t_2, r_2)^T \bullet (r_0, g_0, b_0)^T$,<br>$r' := (s_3, t_3, r_3)^T \bullet (r_0, g_0, b_0)^T$,<br>- Berechnet $R := (r_x, r_y, r_z)$ durch Spiegelung von $E$ an $(s', t', r')$,<br>- Bestimmt $(r_3, g_3, b_3, a_3)$ durch Zugriff auf *Cube Map* (Textur von Stufe 3) mit Koordinaten $(r_x, r_y, r_z)$. | 1: (0, 0, 0, 0)<br>2: (0, 0, 0, 0)<br>3: $(r_3, g_3, b_3, a_3)$ |
| 20 | 1: DOT_PRODUCT_NV<br>2: DOT_PRODUCT_NV<br>3: DOT_PRODUCT_<br>REFLECT_<br>CUBE_MAP_NV | Identisch zu (5), nur für den Vektor $E$ gilt:<br>$E := (q_1, q_2, q_3)$. | 1: (0, 0, 0, 0)<br>2: (0, 0, 0, 0)<br>3: $(r_3, g_3, b_3, a_3)$ |
| 21 | 1: DOT_PRODUCT_NV<br>2: DOT_PRODUCT_<br>DIFFUSE_<br>CUBE_MAP_NV<br>3: DOT_PRODUCT_<br>CONST_EYE_<br>REFLECT_<br>CUBE_MAP_NV | Resultat der vorherigen Stufe sei: $(r_0, g_0, b_0, a_0)$.<br>Sei $E := (e_x, e_y, e_z)$ eine benutzerdefinierte Konstante.<br>- Berechnet $s' := (s_1, t_1, r_1)^T \bullet (r_0, g_0, b_0)^T$,<br>$t' := (s_2, t_2, r_2)^T \bullet (r_0, g_0, b_0)^T$,<br>$r' := (s_3, t_3, r_3)^T \bullet (r_0, g_0, b_0)^T$,<br>- Berechnet $R := (r_x, r_y, r_z)$ durch Spiegelung von $E$ an $(s', t', r')$,<br>- Bestimmt $(r_2, g_2, b_2, a_2)$ durch Zugriff auf Textur 2 mit Koordinaten $(s', t', r')$,<br>- Bestimmt $(r_3, g_3, b_3, a_3)$ durch Zugriff auf Textur 3 mit Koordinaten $(r_x, r_y, r_z)$. | 1: (0, 0, 0, 0)<br>2: $(r_2, g_2, b_2, a_2)$<br>3: $(r_3, g_3, b_3, a_3)$ |
| 22 | 1: DOT_PRODUCT_NV<br>2: DOT_PRODUCT_<br>DIFFUSE_<br>CUBE_MAP_NV<br>3: DOT_PRODUCT_<br>REFLECT_<br>CUBE_MAP_NV | Identisch zu (7), nur für den Vektor $E$ gilt:<br>$E := (q_1, q_2, q_3)$. | |

| | | | |
|---|---|---|---|
| 23 | *i*: DOT_PRODUCT_NV<br>*i*+1: DOT_PRODUCT_<br>DEPTH_<br>REPLACE_NV | Resultat der vorherigen Stufe sei: $(r_0, g_0, b_0, a_0)$.<br>- Berechnet $s' := (s_i, t_i, r_i)^T \bullet (r_0, g_0, b_0)^T$,<br>$t' := (s_{i+1}, t_{i+1}, r_{i+1})^T \bullet (r_0, g_0, b_0)^T$,<br>- Ersetzt die *Window-Space*-Tiefe des Fragments durch den Wert $s'/t'$. | 1: (0, 0, 0, 0)<br>2: (0, 0, 0, 0) |

**Tabelle 20: Texture-Shader-Programme (3) – Bedingte (mehrstufige) Vektorprodukt-Programme [Nvid02b][Domi01]**

graue Felder : Vektorprodukt-Texture-Sampling, gelbes Feld: Spezialfunktion (Tiefenwert-Ersetzen). Die Werte vor den Doppelpunkten geben die Stufe an, in der das Programm ausgeführt werden kann. Sei $i \in \{1, 2\}$, und seien $(s_j, t_j, r_j, q_j)$ die Texturkoordinaten, welche mit Stufe $j$ ($j \in \{1, 2, 3, 4\}$) verknüpft sind.

| Type | Name | VPU-Äquivalent | Zugriff | Anzahl |
|---|---|---|---|---|
| Address | a0.x | A0 | Nur Schreiben | 1 Float |
| Constant | C*n* | Constants | Nur Lesen | 96 Quad-Floats |
| Temporary | R*n* | Register File | Lesen/Schreiben | 12 Quad-Floats |
| Vertex Input | V*n* | Vertex Source | Nur Lesen | 16 Quad-Floats |
| Output | OPos | HPOS | Nur Schreiben | 1 Quad-Float |
| | OD*n* | D*n* | | 2 Quad-Float |
| | OT*n* | TEX*n* | | 8 Quad-Float |
| | oFog.x | FOGP | | 1 Float |
| | oPts.x | PSIZ | | 1 Float |

**Tabelle 21: Die Registertypen eines Vertex Shaders [Micr02]**

| Texturinstruktion | Texture-Shader-Instruktion |
|---|---|
| tex t*i* | TEXTURE_1D<br>oder<br>TEXTURE_2D<br>oder<br>TEXTURE_3D<br>oder<br>TEXTURE_RECTANGLE_NV<br>oder<br>TEXTURE_CUBE_MAP_ARB |
| texbem t*i*, t*j* | OFFSET_TEXTURE_2D_NV<br>oder<br>OFFSET_TEXTURE_RECTANGLE_NV |
| texbeml t*i*, t*j* | OFFSET_TEXTURE_2D_SCALE_NV<br>oder<br>OFFSET_TEXTURE_RECANGLE_SCALE_NV |
| texcoord t*i* | PASS_THROUGH_NV |
| texkill t*j* | CULL_FRAGMENT_NV |
| texreg2ar ti, t*j* | DEPENDENT_AR_TEXTURE_2D_NV |
| texreg2gb ti, t*j* | DEPENDENT_GB_TEXTURE_2D_NV |
| texm3x2pad t(*I+1*), t*i*<br>texm3x2tex t(*I+2*), t*i* | DOT_PRODUCT_NV<br>DOT_PRODUCT_TEXTURE_2D_NV<br>oder<br>DOT_PRODUCT_NV<br>DOT_PRODUCT_TEXTURE_RECTANGLE_NV |
| texm3x3pad t1, t0<br>texm3x3pad t2, t0<br>texm3x3tex t3, t0 | DOT_PRODUCT_NV<br>DOT_PRODUCT_NV<br>DOT_PRODUCT_TEXTURE_3D_NV<br>oder<br>DOT_PRODUCT_NV<br>DOT_PRODUCT_NV<br>DOT_PRODUCT_TEXTURE_CUBE_MAP_NV |
| texm3x3pad t1, t0<br>texm3x3pad t2, t0<br>texm3x3vspec t3, t0 | DOT_PRODUCT_NV<br>DOT_PRODUCT_NV<br>DOT_PRODUCT_REFLECT_CUBE_MAP_NV |
| texm3x3pad t1, t0<br>texm3x3pad t2, t0<br>texm3x3spec t3, t0, c*k* | DOT_PRODUCT_NV<br>DOT_PRODUCT_NV<br>DOT_PRODUCT_CONST_EYE_REFLECT_CUBE_MAP_NV |
| -- | DOT_PRODUCT_DEPTH_REPLACE_NV |
| -- | DOT_PRODUCT_DIFFUSE_CUBE_MAP_NV |
| -- | NONE |

**Tabelle 22: Gegenüberstellung der Texturinstruktionen und Texture-Shader-Programme der GeForce3-FPU**

| Instruktion | Effekt |
|---|---|
| add d, s0, s1 | Komponentenweise Addition zweier Register.<br>d = s0 + s1 |
| dp3 d, s0, s1 | Vektorprodukt der ersten drei Komponenten von s0 und s1:<br>d.x = d.y = d.z = d.w = s0.x*s1.x + s0.y*s1.y + s0.z*s1.z |
| dp4 d, s0, s1 | Vektorprodukt von s0 und s1<br>d.x = d.y = d.z = d.w = s0.x*s1.x +s0.y*s1.y + s0.z*s1.z + s0.w*s1.w |
| dst d, s0, s1 | d.x = 1, d.y = s0.y * s1.y, d.z = s0.z, d.w = s1.w |
| expp d, s0 | *Für s0.w ≥ 0:* Komponente s0.w dient als Basis für verschiedene Berechnungen:<br>d.x = $2^{\lfloor s0.w \rfloor}$<br>d.y = s0.w – ⌊s0.w⌋<br>d.z = $2^{s0.w}$<br>d.w = 1<br>*Für s0.w < 0:* Undefiniertes Resultat |
| lit d, s0 | Berechnet Beleuchtungskoeffizienten für Phong-Lightning:<br>d.x = 1<br>d.y = (s0.x > 0) ? S0.x : 0<br>d.z = (s0.x > 0 && s0.y > 0) ? s0.y ^ s0.w : 0<br>d.w = 1 |
| logp d, s0 | Berechnet Logarithmus von s0 (10 Bit Präzision):<br>d.x = d.y = d.z = d.w =<br>(s0.w != 0) ? log(abs(s0.w))/log(2) : -∞ |
| mad d, s0, s1, s2 | Komponentenweise Multiplikation zweier Werte<br>d.x = s0.x*s1.x + s2.x<br>d.y = s0.y*s1.y + s2.y<br>d.z = s0.z*s1.z + s2.z<br>d.w = s0.w*s1.w + s2.w |
| max d, s0, s1 | Vergleicht komponentenweise und wählt die jeweils größere Komponente:<br>d.x = (s0.x >= s1.x) ? s0.x : s1.x<br>d.y = (s0.y >= s1.y) ? s0.y : s1.y<br>d.z = (s0.z >= s1.z) ? s0.z : s1.z<br>d.w = (s0.w >= s1.w) ? s0.w : s1.w |
| min d, s0, s1 | Vergleicht komponentenweise und wählt die jeweils größere Komponente:<br>d.x = (s0.x < s1.x) ? s0.x : s1.x<br>d.y = (s0.y < s1.y) ? s0.y : s1.y<br>d.z = (s0.z < s1.z) ? s0.z : s1.z<br>d.w = (s0.w < s1.w) ? s0.w : s1.w |
| mov d, s0 | Setzt d = s0 |
| mul d, s0, s1 | Komponentenweise Multiplikation zweier Werte<br>d.x = s0.x * s1.x<br>d.y = s0.y * s1.y<br>d.z = s0.z * s1.z<br>d.w = s0.w * s1.w |
| Nop | No Operation |
| rcp d, s0 | Berechnet den Kehrwert von s0<br>d.x = d.y = d.z = d.w =<br>(s0.w == 0) ? ∞ : 1/s0.w |
| rsq d, s0 | Berechnet den Kehrwert der Quadratwurzel von s0<br>d.x = d.y = d.z = d.w =<br>(s0.w == 0) ? ∞ : 1/sqrt(abs(s0.w)) |

| | |
|---|---|
| sge d, s0, s1 | Bedingungstest:<br>d.x = (s0.x >= s1.x) ? 1 : 0<br>d.y = (s0.y >= s1.y) ? 1 : 0<br>d.z = (s0.z >= s1.z) ? 1 : 0<br>d.w = (s0.w >= s1.w) ? 1 : 0 |
| slt d, s0, s1 | Bedingungstest:<br>d.x = (s0.x < s1.x) ? 1 : 0<br>d.y = (s0.y < s1.y) ? 1 : 0<br>d.z = (s0.z < s1.z) ? 1 : 0<br>d.w = (s0.w < s1.w) ? 1 : 0 |
| sub d, s0, s1 | Komponentenweise Subtraktion<br>d = s0 - s1 |

**Tabelle 23: Vertex-Shader-Instruktionen [Micr02b]**

Der Buchstabe d bezeichnet das Zielregister, die Buchstaben s0, s1, und s2 bezeichnen die Quellregister.

| Name | Syntax | Beschreibung |
|---|---|---|
| Schreibmaske | `Zielregister.[x][y][z][w]` | Nur die im Suffix angegebenen Komponenten des Zielregisters werden von der Ausgabe der Instruktion überschrieben. |
| Swizzle | `Quellregister.{x,y,z,w}{x,y,z,w}{x,y,z,w}{x,y,z,w}` | Die vier Komponenten eines Quellregisters können beliebig gegeneinander ausgetauscht werden. |
| Negation | `-quellregister` | Negiert das Quellregister für diese Instruktion |

**Tabelle 24: Instruktions-Modifikatoren (Vertex Shader) [Micr02b]**

| Instruktion | Effekt |
|---|---|
| `Add z, q1, q2` | Addiert zwei RGBA-Werte:<br>`z = q1 + q2` |
| `Sub z, q1, q2` | Subtrahiert zwei RGBA-Werte:<br>`z = q1 – q2` |
| `Mul z, q1, q2` | Multipliziert zwei RGBA-Werte:<br>`z = q1 * q2` |
| `Mad z, q1, q2, q3` | Multipliziert zwei RGBA- Werte and addiert einen dritten RGBA-Wert:<br>`z = q1 * q2 + q3` |
| `Lrp d, q1, q2, q3` | Lineare Interpolation:<br>`z = q1 * q2 + (1 - q1) * q3` |
| `Dp3 d, q1, q2` | Vektorprodukt der ersten drei Komponenten:<br>`z = q1.r*q2.r + q1.g*q2.g + q1.b*q2.b` |
| `Mov z, q` | Setzt: `z = q` |
| `Cnd z, r0.a, q1, q2` | Bedingungstest:<br>`z = ( r0.a > 0.5) ?  q1 : q2` |

**Tabelle 25: Die arithmetischen Instruktionen eines Pixel Shaders der Version 1.1 [Micr02b]**

`z` – Zielregister; `q, q1, q2, q3` – Quellregister.

| Modifikator | Effekt |
|---|---|
| `instr z, q_bias` | q'= q – 0.5 |
| `instr z, 1-q` | q'= 1 – q |
| `intsr z, -q` | q'= -q |
| `instr z, q_x2` | q'= 2 * q |
| `instr z, q_bx2` | q' = (q - 0.5) * 2 |
| `instr_x2 z, q` | Verdoppelt das Ergebnis der Instruktion |
| `instr_x4 z, q` | Vervierfacht das Ergebnis der Instruktion |
| `instr_d2 z, q` | Halbiert das Ergebnis der Instruktion |
| `instr_sat z, q` | Werte < 0 werden auf 0 gesetzt, so dass alle Werte im Bereich [0,1] liegen. |

**Tabelle 26: Modifikatoren für arithmetische Instruktionen [Micr02b]**

instr – arithmetische Instruktion, z – Zielregister, q – Quellregister, q' – Quellregister nach der Modifikation. Beachte, dass alle Werte die nach der Modifikation außerhalb des Wertebereichs [-1, 1] liegen, auf –1 (bei Werten < -1) bzw. auf 1 (bei Werten > 1) gesetzt werden.

| Modifikator | Effekt |
|---|---|
| `instr z, q.xyz`<br>bzw.<br>`instr z, q.rgb` | `q' = (x, y, z, z)`<br>bzw.<br>`q' = (r, g, b, b)` |
| `instr z, q.xyw`<br>bzw.<br>`instr z, q.rga` | `q' = (x, y, w, w)`<br>bzw.<br>`q' = (r, g, a, a)` |
| `instr z.rgb, q` | Nur die RGB-Komponenten von z werden von der Instruktion überschrieben. |
| `instr z.a, q` | Nur die Alpha-Komponente von z wird von der Instruktion überschrieben. |

**Tabelle 27: Lese- und Schreibmasken für arithmetische Instruktionen [Micr02b]**

instr – arithmetische Instruktion, z – Zielregister, q – Quellregister, q' – Quellregister nach der Modifikation.

# Literaturverzeichnis


[ACM02] ACM Homepage. ACM, 2002. Internet: http://www.acm.org (Zugriff: 08.04.2002).

[Akel88] K. Akeley, T. Jermoluk. *High-Performance Polygon Rendering.* In Proceedings of SIGGRAPH 1988, vol. 22, no. 4, Seiten 239-246, ACM, August 1988.

[Akel93] K. Akeley. *Reality Engine Graphics*. In Proceedings of SIGGRAPH 1993, Seiten 109-116, ACM, August 1993.

[Arvo90] J. Arvo. *Backward Ray Tracing*. In *Proceedings of* SIGGRAPH 1986: Development in Ray Tracing course notes, ACM, August 1986.

[ATI02] ATI Homepage. Internet: http://www.ati.com (Zugriff: 08.04.2002).

[Balz98] H. Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung,* Unternehmensmodellierung. Heidelberg, Berlin: Spektrum, Akademischer Verlag, 1998.

[Barr84] A. H. Barr. *Global and Local Deformations of Solid Primitives*. In Proceedings of SIGGRAPH 1984, Seiten 21-30, ACM, 1984.

[Blin76] J.F. Blinn, M.E. Newell. *Texture and reflection in computer generated images*. Communications of the ACM, 19 (10), Seiten 362-367, ACM, 1976.

[Blin77] J.F. Blinn. *Models of Light Reflection for Computer Synthesized Pictures.* In . In Proceedings of SIGGRAPH 1977, Seiten 192-198, ACM, Juli 1977.

[Blin78] J.F. Blinn. *Simulation of wrinkled surfaces*. In Proceedings of SIGGRAPH 1978, 12(3), Seiten 286-292, ACM, 1978.

[Blin96] J.F. Blinn. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, San Francisco, Morgan Kaufmann Publishers Inc., 1996.

[Booc99] G. Booch, J. Rumbaugh et al., *The Unified Modeling Language User Guide*. Reading, MA, Addison-Wesley Longman, 1999.

[Buch96] J.W. Buchanan. Special Effects with Half-Toning. Computer Graphics Forum, 15(3), Seiten 97-108, 1996.

[Cabr87] B. Cabral, N. Max, R. Springmeyer. *Biderectional reflection functions from surface bump maps.* In Proceedings of SIGGRAPH 1987, Seiten 273-281, ACM, 1987.

[Catm78] E. Catmull. *Computer Display of Curfed Surfaces*. Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures, Seiten 11-17, Los Angeles, Mai 1975.

[Clar82] J.H. Clark. *The Geometry Engine: A VLSI Geometry System for Graphics*. In Proceedings of SIGGRAPH 1982, vol. 16, Seiten 127-133, ACM, Juli 1982.

[Cohe85] M.F. Cohen, D.P. Greenberg. *A Radiosity Solution for Complex Environments*. In Proceedings of SIGGRAPH 1985, 19(3), Seiten 31-40, ACM, 1985.

[Cohe88] M.F. Cohen, S.E. Chen, J.R. Wallace, D.P. Greenberg. *A Progressive Refinement Approach to Fast Radiosity Image Creation.* In Proceedings of SIGGRAPH 1988, 22(4), Seiten 75-84, ACM, 1988.

[Cohe00] J.M. Cohen, J.F. Hughes, R.C. Zeleznik. Harold: A World Made of Drawings. In Proceedings of NPAR 2000, Seiten 83-90, ACM, 2000.

[Cook82] R. L. Cook, K. E. Torrance. *A reflectance model for computer graphics.* In ACM Transactions on Graphics, 1(1), ACM, Januar 1982.

[Cook84] R. L. Cook. *Shade Trees* In Proceedings of SIGGRAPH 1984, Seiten 223-231, ACM, Juli 1984.

[Cook87] R. L. Cook, L.Carpenter, E. Catmull. *The Reyes Image Rendering Architecture*. In Proceedings of SIGGRAPH 1987, Seiten 95-102, ACM, 1987.

[Curt97] C.J. Curtis, S.E. Anderson, J.E. Seims, K.W. Fleischer, D.H. Salesin. Computer-Generated Watercolor. In Proceedings of SIGGRAPH 1997, Seiten 421-430, ACM, 1997.

[Deer88] M.Deering, S. Winner, B. Schediwy, C. Duffy, N. Hunt. *The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics.* In Proceedings of SIGGRAPH 1988, Seiten 21-30, ACM, August 1988.

[Deus98] O. Deussen. *Pixel-Oriented Rendering of Line Drawings*. In T. Strothotte, *Computational Visualization: Graphics, Abstraction, and Interactivity*. Kapitel 6, Seiten 105-119, Berlin: Springer-Verlag, 1998.

[Deus00] O. Deussen, T. Strothotte. Computer-Generated Pen-and-Ink Illustration of Trees. In Proceedings of SIGGRAPH 2000, Seiten 13-18, ACM, 2000.

[Dogg02] M. Dogget. *Programmability Features of Graphics Hardware*. ATI Technologies Inc., April 2002. Internet: http://www.ati.com (Zugriff: 08.04.2002).

[Enca97a] J. Encarnacao, W. Straßer, R. Klein. *Graphische Datenverarbeitung 1: Gerätetechnik, Programmierung und Anwendung graphischer Systeme*. 4. Auflage, München, R. Oldenburg Verlag GmbH, 1997.

[Enca97b] J. Encarnacao, W. Straßer, R. Klein. *Graphische Datenverarbeitung 2: Gerätetechnik, Programmierung und Anwendung graphischer Systeme*. 4. Auflage, München , R. Oldenburg Verlag GmbH, 1997.

[Enge01] Wolfgang Engel, Amir Geva: *Direct3D Spieleprogrammierung*. 1. Auflage, Düsseldorf, Sybex-Verlag GmbH, 2001.

[Engl86] N. England. *A Graphics System Architecture for Interactive Appilication-Specific Display Functions*. IEEE Computer Graphics and Applications, 6(1), Seiten 60-70, Januar 1986.

[Erns98] I. Ernst, H. Russeler, H. Schulz, O. Wittig. *Gouraud Bump Mapping*. In Proceedings of the 1998 Eurgraphics/SIGGRAPH Workshop on Graphics Hardware, Lisbon, Portugal, Seiten 47-53, ACM, August 1998.

[Eyle97] J. Eyles, S. Molnar, J. Poulton, T. Greer. A. Lastra, N. England, L. Westover. *PixelFlow: The Realization*. In Proceedings of SIGGGRAPH 1997/Eurographics Workshop on Graphics Hardware, Los Angeles, Kalifornien, Seiten 57-68, ACM, August 1997

[Fari96] G.E. Fari. *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Guide*. Boston: Academic Press, 1996.

[Fole97] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes. *Computer Graphics: Principle and Practice.* 2. Auflage, Reading MA, USA, Addison-Wesley, Juli 1997.

[Fost00] N. Foster, D. Metaxas. *Modeling Water for Computer Animation*. In Communications of the ACM, Vol. 43, No. 7, Seiten 61-67, 2000.

[Fost01] N. Foster, R. Fedkiw. *Practical Animation of Liquids*. In Proceedings of SIGGRAPH 2001, Seiten 23-30, ACM, August 2001.

[Four86] A. Fournier, W. Reeves. *A Simple Model of Ocean Waves*. In Proceedings of SIGGRAPH 1986, 20(4), Seiten 75-84, ACM, 1986.

[Gooc98] A. A. Gooch, B. Gooch, P. Shirley, E. Cohen. A Non-Photorealistic Lightning Model for Automatic Technical Illustration. In Proceeding of SIGGRAPH 1998, Seiten 447-452, ACM, 1998.

[Gora84] C. Goral, K.E. Torrance, D.P. Greenberg. *Modelling the Interaction of Light between Diffuse Surfaces*. In Proceedings of SIGGRAPH 1984, 18(3), Seiten 212-222, ACM, 1984.

[Gour71] H. Gouraud. *Continuous shading of curved surfaces.* IEEE Transactions on Computers, vol. C-20(6), Seiten 623-629, Juni 1971.

[Gree86] N. Greene. *Environment Mapping and Other Applications of World Projections.* In IEEE Computer Graphics and Applications, 6 (11),. Seiten 21-9, 1986.

[Hain01] E. Haines, T.Möller. *Real-Time Shadows*. GDC 2001 Proceedings, Internet: http://www.gdconf.com/archives /proceedings/ 2001/haines.pdf (Zugriff: 08.04.2002).

[Hanr90] P. Hanrahan, J. Lawson. *A Language for Shading and Lightning Calculations*. In Proceedings of SIGGRAPH 1990, Seiten 289-298, 1990.

[Hanr93] P. Hanrahan, W. Kreuger. *Reflection from layered surfaces due to sub-surface scattering*. In Proceedings of SIGGRAPH 1993, Seiten 165-174, ACM, 1993.

[Harre93] C. Harrell, F. Fouladi. *Graphics Rendering Architecture for a High-Performance Desktop Workstation*. In Proceedings of SIGGRAPH 1993, Seiten 93-100, ACM, August 1993.

[Heck86] P.S. Heckbert. *Survey of texture mapping*. IEEE Computer Graphics and Applications, 6 (11), Seiten 56-67, 1986.

[Heck90] P.S. Heckbert. *Adaptive Radiosity Textures for Bidirectional Ray Tracing*. In Proceedings of SIGGRAPH 1990, Seiten 145-154, ACM, 1990.

[Heid99] W. Heidrich, H.-P. Seidel. *Realistic, Hardware-accelerated Shading and Lightning*. In Proceedings of SIGGRAPH 1999, ACM, Seiten 165-174.

[Hopp96] H. Hoppe. *Progressive meshes*. In Proceedings of SIGGRAPH 1996, S.99-108, ACM, 1996.

[Kaji85] J. T. Kajiya. *Anisotropic Reflection Models*. In Proceedings of SIGGRAPH 1985, S.15-21, ACM, 1985.

[Kaji86] James T. Kajiya. *The Rendering Equatation*. In ACM Computer Graphics 20(4), Seiten 143-149, ACM, August 1986.

[Kay79] D.S. Kay, D. Greenberg. *Transparency for Computer Synthesised Objects*. In Proceedings of SIGGRAPH 1979, Seiten 158-164, ACM, 1979.

[Kay86] D.S. Kay, J.T. Kajiya. *Ray Tracing Complex Scenes*. In ACM Computer Graphics 20(4), In Proceedings of SIGGRAPH 1986, Seiten 169-278, ACM, 1986.

[Kern78] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978

[Kirk90] D. Kirk, D. Voorhies. *The Rendering Architecture of the DN10000VS*. In Proceedings of SIGGRAPH 1990, vol. 24, Seiten 299-307, ACM, August 1990.

[Kirk01] D. Kirk. *GeForce3 Architecture Overview.* NVIDIA Corporation, 2001. Internet: www.nvidia.com/developer (Zugriff: 08.04.2002)

[Lake00] A. Lake, C. Marshall, M. Harris, M. Blackstein. *Stylized Rendering Techniques For Scalable Real-Time 3D Animation*. In Proceedings of NPAR 2000, Seiten 13-20, ACM, 2000.

[Last95] A. Lastra, S. Molnar, M.Olano, Y. Wang. *Real-Time Programmable Shading*. In Proceedings of SIGGRAPH 1995, Seiten 59-66, ACM, April 1995.

[Levi84] A. Levinthal, T. Porter. *Chap – A SIMD Graphics Processor*. In Proceedings of SIGGRAPH 1984, vol. 18, Seiten 77-82, ACM, Juli 1984.

[Lind01] E. Lindholm, M. J. Kilgard, H. Moreton: *A User-Programmable Vertex Engine*. In Proceedings of SIGGRAPH 2001, Seiten 149-158, ACM, August 2001.

[Lind02] E. Lindholm (NVIDIA Corporation). *Vertex Programs for Fixed Function.* Internet: www.nvidia.com/developer (Zugriff: 08.04.2002)

[Maju02] A. Majumder, M. Gopi. *Hardware Accelerated Real Time Charcoal Rendering*. In Proceedings of SIGGRAPH 2002, Seiten 59-66, ACM Press, 2002.

[Mark97] L. Markosian, M.A: Trychin, S.J. Bourdev, L.D., D.Goldstein, J.F. Hughes. *Real-Time Nonphotorealistic Rendering*. In Proceedings of SIGGRAPH 1997, Seiten 415-420, ACM, 1997.

[Mark02] Bill Mark. *NVIDIA Programmable Graphics Technology*. SIGGRAPH 2002, San Antonio, NVIDIA Corporation, 2002. www.nvidia.com (Zugriff: 08.04.2002).

[Mart00] D. Martin, S. Garcia, J.C. Torres. *Observer Dependent Deformations in Illustration*. In Proceedings of NPAR 2000, Seiten 75-82, ACM, 2000.

[Mast87] G.A. Mastin, P.A. Watterberg, J.F. Mareda. *Fourier Synthesis of Ocean Scenes*. IEEE Computer Graphics an Applications, 7(3), Seiten 16-23, 1987.

[MATR02] MATROX Graphics Homepage. Internet : http://www.matrox.com/mga/home.htm (Zugriff: 08.04.2002).

[McCo99] M.D. McCool, W. Heidrich. *Texture Shaders*. SIGGRAPH 1999/Eurographics Workshop on Graphics Hardware (August 1999), ACM, Seiten 117-126.

[McCo01] M.D. McCool, J. Ang, A. Ahmad. *Homomorphic Factorizations of BRDFs for High-Performance Rendering*. SIGGRAPH Proceedings, Seiten 171-178, 2001.

[McCu02] M. McCuskey. *Special Effects Programming with DirectX*. USA, Premier Press, 2002.

[Micr02a] Microsoft Corporation, Inc. *DirectX Home*. Internet: www.microsoft.com/directX (Zugriff: 10.04.2002).

[Micr02b] Microsoft Corporation, Inc. *DirectX 8.1 SDK*. Internet: http://msdn.microsoft.com/library/ (Zugriff: 08.04.2002).

[Micr02c] Microsoft Corporation, Inc. *Microsoft Releases Beta Version 1 of DirectX 9.0 to Partners.* Pressemitteilung vom 02.05.2002. Internet: http://www.microsoft.com/presspass/ (Zugriff: 08.04.2002).

[Micr02d] Microsoft Corporation, Inc. *DirectX 7 SDK*. Internet: http://msdn.microsoft.com/library/ (Zugriff: 08.04.2002).

[Mitc02] J.L. Mitchell. Pixel Shading with DirectX8.1 and the ATI RADEON 8500. ATI Research, 2002. Internet: http://mirror.ati.com/developer/techpapers.html (Zugriff: 08.04.2002).

[Möll99] Tomas Möller, Eric Haines. *Real-time Rendering*. Natick, MA, A K Peters Ltd., 1999

[Moln92] S. Molnar, J. Eyles, J. Poulton. *PixelFlow: High-speed Rendering Using Image Composition*. In Proceedings of SIGGRAPH 1992. Seiten 231-240, ACM, 1992.

[Mont97] J. Montrym, D. Baum, D. Dignam, C. Migdal. *InfiniteReality: A Real-Time Graphics System*. In Proceedings of SIGGRAPH 1997, Seiten 293-302, ACM, August 1997.

[Nake72] Nake, F., Rosenfeld, A. *Graphics Languages*. Amsterdam, Noth Holland Pub, Co., 1972.

[Nark95] A. Narkhede, D. Manocha. *Fast Polygon Triangulation Based on Seidel's Algorithm*. Editor: A. Paeth. Graphic Gems V, ,Boston, AP Professional, Seiten 394-397, 1995.

[NVID01a] NVIDIA Corporation. GeForce3 – Product Overview 06.01.v1. NVIDIA Corporation, 2001. Internet: http://www.nvidia.com (Zugriff: 08.04.2002).

[NVID01b] NVIDIA Corporation. Technical Brief – NVIDIA nfiniteFX Engine: ProgrammableVertex Shaders. NVIDIA Corporation, 2001. Internet: http://www.nvidia.com (Zugriff: 08.04.2002).

[NVID01c] NVIDIA Corporation. Technical Brief – NVIDIA nfiniteFX Engine: Programmable Pixel Shaders. http://www.nvidia.com (Zugriff: 12.05.2002). Erstellt am: 01.03.2001.

[NVID01d] NVIDIA Corporation. Technical Brief – Shadow Buffers. NVIDIA Corporation, 2001. Internet: http://www.nvidia.com (Zugriff: 08.04.2002).

[NVID02a] *Nvidia Home*. NVIDIA Corporation., 2002. Internet: http://www.nvidia.com (Zugriff: 08.04.2002).

[NVID02b] *NVDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2002. Internet: www.nvidia.com/developer (Zugriff: 08.04.2002).

[NVID02c] *Cg Language Specification*. NVIDIA Corporation, Juni 2002. Internet: www.nvidia.com/developer (Zugriff: 25.06.2002).

[NVID02d] *Cg Toolkit – Version 1.0*. NVIDIA Corporation, Juni 2002. Internet: www.nvidia.com/developer (Zugriff: 08.04.2002).

[NVID02e] Nvidia Developer Page. NVIDIA Corporation, 2002. Internet: www.nvidia.com/developer (Zugriff: 08.04.2002).

[NVID02f] NVIDIA Effects Browser. NVIDIA Corporation, 2002. Internet: www.nvidia.com/developer (Zugriff: 08.04.2002).

[NVID02g] NVIDIA “CineFX” Architecture. NVIDIA Corporation, 2002. Internet: www.nvidia.com/developer (Zugriff: 08.04.2002).

[Olan98] M. Olano, A. Lastra. *A Shading Language on Graphics Hardware: The PixelFlow Shading System*. In Proceedings of SIGGRAPH 1998, Seiten 159-168, ACM, Juli 1998.

[Olan00] M. Olano, J.C. Hart, W. Heidrich, M. McCool, B. Mark, K. Proudfoot. *Approaches for procedural shading on graphics hardware: Course notes*. In Proceedings of SIGGRAPH 2000, ACM, Juli 2000.

[Open02] OpenGL Homepage. Internet: http://www.opengl.org (Zugriff: 08.04.2002).

[Open02b] OpenGL 2.0 – Proposals. Internet: http://www.3dlabs.com/support/developer/ogl2/index.htm (Zugriff: 08.04.2002).

[Pare02] R. Parent. *Computer Animation – Algorithms and Techniques*. San Diego: Academic Press, San Francisco: Morgan Kaufmann Publishers, 2002.

[Peac86] D. Peachey. *Modeling Waves and Surf*. In Proceedings of SIGGRAPH 1986, Seiten 65-74, ACM, August1986.

[Peer00] M. S. Peercy, M. Olano, J. Airey, P. J. Ungar. *Interactive Multi-Pass Programmable Shading* In Proceedings of SIGGRAPH 2000, Seiten 425-432, ACM, Juli 2000.

[Perl85] K. Perlin. *An Image Synthesizer*. In Proceedings of SIGGRAPH 1985, Seiten 287-296, ACM, Juli 1985.

[Phon75] B. Phong. *Illumination for computer-generated pictures*. Communications of the ACM, 18 (6), Seiten 311-317, ACM,1975.

[Pixa00] Pixar Animation Studios. *The RenderMan Interface Specification: Version 3.2*. Juli 2000. Internet: http://www.pixar.com (Zugriff: 08.04.2002).

[Prau01] E. Praun, H. Hoppe, M. Webb, A. Finkelstein. *Real-time Hatching*. In Proceedings of SIGGRAPH 2001, Seiten 581-586, ACM, 2001.

[Prou01] K. Proudfoot, W. R. Mark, S. Tzvetkov, P. Hanrahan. *A Real-Time Procedural Shading System for Programmable Graphics Hardware.* In Proceedings of SIGGRAPH 2001, Seiten 159-170, ACM, August 2001.

[Reyn82] C.W. Reynolds. *Flocks, Herds, and Schools: A Distributed Behavioural Model.* In Proceedings of SIGGRAPH 1987, Seiten 25-34, ACM, 1987.

[Rhoa92] J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, A. Varshney. *Real-time Procedural Textures*. In Proceedings of SIGGRAPH 1992, Seiten 95-100, ACM, März 1992.

[SGI02] OpenGL Shader. SGI, 2002. Internet: http://www.sgi.com/software/shader/ (Zugriff: 08.04.2002).

[Shne98] B. Shneidermann. *Designing the User Interface*. 3. Auflage. Reading, MA, Addison Wesley, 1998.

[Sega92] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. Haeberli. *Fast Shadows and Lightning Effects using Texture Mapping*. In Proceedings of SIGGRAPH 1992, Seiten 249-252, ACM, Juli 1992.

[Somm97] I. Sommerville. *Software Engineering*. 5. Auflage. Harlow, England, Addison Wesley, 1997.

[Stro02] T. Strothotte, S. Schlechtweg. *Non-Photorealistic Computer Graphics – Modeling, Rendering, and Animation*. Morgan Kaufmann Publishers, San Francisco, 2002.

[Tess01] J. Tessendorf. *Simulating Ocean Water*. In Proceedings of SIGGRAPH 2001, Course notes, ACM, 2001.

[Tom02] Tom's Hardware Guide. Internet: http://www6.tomshardware.com/graphic/01q1/010227/geforce3-03.html (Zugriff: 08.04.2002).

[Torb87] J. Torborg. *A parallel processor architecture for graphics arithmetic operations*. In Proceedings of SIGGRAPH 1987, vol. 21, Seiten 197-204. ACM, Juli 1987.

[Upst90] S. Upstill. *The RenderMan Companion – A Programmer's Guide to Realistic Computer Graphics*. Reading, Massachusetts, Addison-Wesley Publishing Ltd, 1990.

[Watt92] Alan Watt, Mark Watt. *Advanced Animation and Rendering Techniques*, Harlow, Addison-Wesley Publishing Ltd , 1992.

[Watt00] Alan Watt. *3D Computer Graphics*., Harlow, 3. Auflage, Addison-Wesley Publishing Ltd, 2000.

[Whit80] T. Whitted. *An Improved Illumination Model for Shaded Display*. Communications of the ACM, 26(6), Seiten 342-349, ACM, 1980.

[Whit81] T. Whitted, D. M. Weimer. *A Software Test-Bed for Development of 3-D Raster Graphics Systems*. In Proceedings of SIGGRAPH 1981, Seiten 271-277, ACM, August 1981.

[Wynn01] C. Wynn. *Vertex Programs*. Game Developers Conference 2001. www.nvidia.com (Zugriff: 08.04.2002).