Entwicklung eines Compilers für eine auf Cg basierende Sprache zur Programmierung von Grafikkarten

Diplomarbeit

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Frank Richter[1]
Betreuer: Prof. Dr. habil. Wolfram Amme

21. März 2011

[1] Matrikel 68278, Kontakt: frank.richter@gmail.com

Zusammenfassung

In dieser Arbeit wurde eine Sprache zur Programmierung von Grafikprozessoren sowie ein diese Sprache übersetzender Compiler entwickelt.

Schwerpunkt ist die Besonderheit des Compiler, mehrerer Ausgabeprogramme aus einem Eingabeprogramm zu erzeugen. Die verschiedenen Ausgabeprogramme zielen dabei auf verschiedene Funktionseinheiten eines Grafikprozessors ab. Die Aufspaltung in die Ausgabeprogramme wird so vorgenommen, das im Zusammenspiel der verschiedenen Programme auf den verschiedenen Funktionseinheiten die Semantik des Eingabeprogramms umgesetzt wird. Für die Aufspaltung selbst wurden Kriterien entworfen, die den speziellen Aufbau und Datenfluss auf Grafikprozessoren berücksichtigen und ausnutzen.

Die entwickelte Programmiersprache ist eine Hochsprache, die in wesentlichen Zügen auf der Sprache Cg basiert, im Vergleich aber signifikant vereinfacht wurde und in der syntaktisch bewusste einige eigene Wege eingeschlagen wurden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Zielstellung

Ziel dieser Arbeit ist es, eine Sprache und einen Compiler zum Zweck der Programmierung von 3D-Grafikprozessoren (“GPU”, “Graphics Processing Unit”) zu entwickeln. GPUs sind aus verschiedenen Funktionseinheiten aufgebaut; Sprache und Compiler sollen die GPU-Programmierung vereinfachen, in dem die Aufteilung eines Programms in mehrere Teilprogramme für verschiedene Funktionseinheiten automatisch vorgenommen wird.

Weiterhin soll die entwickelte Sprache die Programmierung von GPUs syntaktisch unterstützen, wie es auch bei es auch bei existierend Sprachen zu diesem Zweck üblich ist: durch Vektortypen, Unterstützung von Verknüpfungen u.ä. von ganzen Vektoren sowie vordefinierte Funktionen für typische Vektoroperationen.

Der entwickelte Compiler soll einer modularen Standardarchitektur folgen: ein „Front-End“ soll Eingaben übernehmen und das Programm in eine interne Darstellung überführen, auf welcher ein „Kern“ weitere Arbeitsschritte vornimmt; abschliessend soll ein „Back-End“ das Programm in eine Zieldarstellung ausgeben. Die vorgenommenen „Arbeitsschritte“ sollen nicht nur der typische Optimierungsschritt sein, sondern in diesem speziellen Compiler auch die „Aufspaltung“ des Programms in die Teile für die verschiedenen GPU-Funktionseinheiten vornehmen.

Für die interne Darstellung des zu übersetzenden Programms soll eine Zwischencoderepräsentation entworfen werden, auf der die nötigen Arbeitsschritte – also „Aufspaltung“ und Optimierungen – vorgenommen werden können und aus der die Ausgabe in die Zieldarstellung generiert werden kann.

1.2 Gliederung

Zuerst wird eine Einführung in Grundlagen der 3D-Grafik vorgenommen. Diese sind nötig um die Absicht und Besonderheit des entwickelten Compilers zu verstehen. Auch in den weiteren Kapiteln vorkommende, fachspezifische Begriffe werden dort erklärt.

Im Kapitel „Sprachspezifikation“ wird die Sprache spezifiziert, in der die Programme geschrieben werden sollen. Es werden allgemeine Anforderungen an die Sprache gestellt sowie auf spezielle Aspekte der vorzunehmenden Auftrennung von Programmen eingegangen. Unter Berücksichtigung dieser Anforderungen und der speziellen Aspekte wird die Sprachsyntax sowie eine Auswahl vordefinierter Funktionen spezifiert.

Im darauf folgenden Kapitel wird schliesslich auf die Implementierung des Compilers selbst eingegangen. Insbesondere werden die verwendete Zwischencoderepräsentation für die Weitergabe des Programmen zwischen den verschiedenen Arbeitsschritten des Compilers sowie das eigentliche „Ziel“ dieser Arbeit, die Komponente zur Auftrennung eines Programms, beschrieben. Auch der umgesetzte Generator für die Programm-Ausgabe sowie vorgenommene Optimierungen werden erläutert.

Abschliessend, im Kapitel „Evaluation“, wird die Leistung des Compilers anhand praxisnaher Beispielprogramme evaluiert.

Kapitel 2

Einführung 3D-Grafik

Dieses Kapitel beschreibt den Ablauf der Berechnung einer 3D-Grafik in Echtzeit. Das Augenmerk liegt vor allem darauf, Grundlagen der Grafikkartenprogrammierung – und damit verbundenen Besonderheiten der verwendeten Sprache – zu erklären, Zweck des entwickelten Compilers, dessen „Zielhardware“ Grafikkarten sind, zu verdeutlichen, und Gründe für dessen Funktionsweise zu liefern.

Diese Einführung erfasst nur einen kleinen Ausschnitt des Themengebietes „3D-Grafik“ und seiner Unterkategorie „Echtzeit-3D-Grafik“. Selbst die ausgewählten Sachverhalte werden nur vereinfacht dargestellt. Eine umfassende Einführung sowie viele weitere Aspekte des Themenkomplexes „3D-Grafik“ bietet zum Beispiel [Wat02].

2.1 Echtzeit-3D-Grafik

Um „Echtzeit-3D-Grafik“ handelt es sich, wenn eine 3D-Grafik mit dynamischen Komponenten (Animation, Änderung der Betrachterposition) dargestellt werden soll, typischerweise verbunden mit einer Manipulation durch einen menschlichen Benutzer (der z.B. die Betrachterposition kontrolliert). Unter anderem Szenarien von „virtuellen Realitäten“ fallen darunter (z.B. Computerspiele). Das „Echtzeit“ im Namen bedeutet, dass die dargestellte Grafik mit möglichst geringer, idealerweise nicht wahrnehmbarer Verzögerung dargestellt wird. Das vom Kino bekannte Prinzip, dass mit einem schnellen Bildwechsel aus statischen Bildern die Illusion von Bewegung erzeugt werden kann, findet genau so auch der Echtzeit-3D-Grafik Anwendung: Computerspiele stellen typischerweise 30 oder 60 Bilder pro Sekunde dar um Animationen überzeugend zu präsentieren. Dabei wird jedes dieser Bilder neu

berechnet – eine der Herausforderungen der Echtzeit-3D-Grafik ist also, für komplexe Szenarien Bilder in wenigen Millisekunden zu berechnen.

2.2 3D-Objekte und Szenen

In einer Echtzeit-3D-Grafik darzustellende Objekte („Modelle“) werden als *Dreiecksnetz* gespeichert. Dieses besteht aus einer Menge *Eckpunkte* (“Vertex” bzw. “Vertices”) sowie von diesen aufgespannte *Dreiecke*.

Definition 1. *Ein* 3D-Modell *wird als* ***Dreiecksnetz*** *gespeichert. Zu diesen Dreiecken wird eine Menge von* Eckpunkten *gespeichert.*

Definition 2. *Ein* ***Vertex*** *ist ein Eckpunkt eines Dreiecksnetzes.*

Zu jedem Eckpunkt ist dessen *Position*, eine Koordinate im dreidimensionalen Raum, gespeichert. Praktisch sind zu jedem Eckpunkt auch noch weitere Daten gespeichert („Vertexattribute“), die bei der Berechnung der Oberflächenerscheinung des Objekts eine Rolle spielen. Typischerweise verwendete Vertexattribute sind eine Oberflächennormale (als Vektor mit drei Komponenten) sowie Koordinaten für ein Oberflächen-„Muster“[1] (als Vektor mit zwei Komponenten). (Man kann sich vorstellen, dass ein Verbundtyp die Vertexattribute definiert; die Menge der Vertices eines Modells wäre eine Reihung von Werten dieses Typs, und ein Element enthält entsprechend die Attributwerte für ein Vertex.)

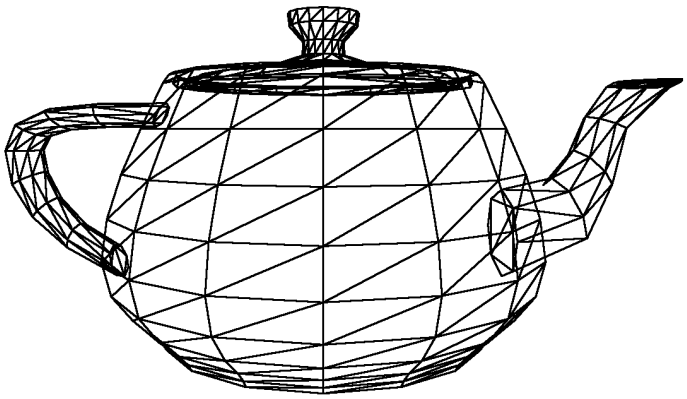

Abbildung 2.1: „Drahtgitter“-Darstellung eines 3D-Objektes. Die einzelnen Eckpunkte und Dreiecke sind gut erkennbar.

[1] siehe Abschnitt 2.3

Definition 3. ***Vertexattribute*** *beschreiben die zu einem Vertex gespeicherten Daten. Es ist immer mindestens ein Attribut „Raumkoordinate“ vorhanden, es können aber eine beliebige Anzahl von Attributen verwendet werden. Attributtypen sind Vektoren mit ein bis vier Komponenten.*

*In einem 3D-Modell besitzt jedes Vertex die gleichen Attribute, aber verschiedene Attribut*werte.

Selten wird ein alleinstehendes 3D-Objekt dargestellt – stattdessen wird praktisch immer mit aus mehreren Objekten bestehenden 3D-„*Szenen*“ gearbeitet. In solchen Szenen ist es auch nicht unüblich, das selbe Objekt mehrfach, aber an unterschiedlichen Position und/oder mit unterschiedlichen Drehungen darzustellen (z.B. eine Küchen-Szene mit identischem Geschirr an verschiedenen Stellen). 3D-Objekte müssen also positioniert usw. werden. Dazu dienen *Transformationen* (Abbildungen). Ein 3D-Objektes wird mit einer Transformation transformiert, indem diese auf die Raumkoordinate jedes Vertices des Modells angewendet wird.

Definition 4. ***Transformationen*** *sind Abbildungen von Raumkoordinaten auf Raumkoordinaten und dienen dazu, 3D-Objekte zu verschieben, zu rotieren oder zu skalieren.*

Eine Transformation wird auf ein Objekt angewendet, indem auf die Raumkoordinaten aller Vertices dieselbe Transformation angewendet wird.

Eine der bestimmenden Gründe, warum Echtzeit-3D-Grafik eingesetzt wird, ist die Möglichkeit eines „freien Bewegens“ durch eine Szene (z.B. könnte die beispielhafte Küche Teil einer „virtuellen Hausbesichtigung“ sein). Mit anderen Worten, der Blickpunkt und die Blickrichtung des Betrachters, oder auch *Kamera*, soll veränderbar sein. Auch dies wird unter Verwendung einer Transformation realisiert.

Transformationen bilden weiterhin die Grundlage der für Echtzeit-3D-Szenen typischen Dynamik von Szenen; Objekte werden animiert, in dem die für die Positionierung verwendete Transformation über die Zeit verändert wird.

Schliesslich wird eine 3D-Szene in der Regel auf einem zweidimensionalen Ausgabegerät – typischerweise einem Computermonitor – dargestellt. Es muss also eine Herrunterrechnung auf die zweidimensionale Fläche, eine *Projektion*, vorgenommen werden; dabei muss, für korrekten räumlichen Eindruck, die *perspektivische Verzerrung*

berücksichtigt werden. Auch diese Projektion und Verzerrung werden grundsätzlich durch Transformationen realisiert[2]. Als letzten Schritt müssen die – jetzt auf einer zweidimensionalen Fläche vorliegenden – Dreiecke der Objekte in Monitor-Pixel „umgewandelt“, also *gerastert*, werden. Für jedes dieser Pixel wird schliesslich die Farbe bestimmt, mit der es auf dem Monitor dargestellt wird[3].

2.3 Darstellung und Beleuchtung von Oberflächen

Beleuchtung und Schattenwurf von 3D-Objekten geben einem menschlichen Betrachter wichtige Hinweise zur Form einzelner Objekte sowie zur Positionierung von Objekten zueinander. Aus Lichtreflektionen der Oberfläche – Art, Farbe und Zusammenspiel mit Strukturen und Mustern – lassen sich Materialeigenschaften und damit das dargestellte Material ableiten.

Beleuchtungs- und Oberflächenberechnungen (“*Shading*”) sind bei der Darstellung von 3D-Objekten also äusserst wichtig. Allerdings können diese auch Zeitaufwendig sein, insbesondere da in der Echtzeit-3D-Grafik die Szenen in der Regel immer auch eine dynamische Komponente haben. Insbesondere Beleuchtungsberechnungen hängen von der Kameraposition und/oder der Objektposition ab und müssen, wenn die Kamera oder Objekte sich bewegen, für jedes darzustellende Bild neu berechnet werden. Aus der Anforderung der Echtzeit-3D-Grafik, für eine überzeugende Bewegungsdarstellung Bilder genügend schnell zu berechnen, ergibt sich auch für die Beleuchtungs- und Oberflächenberechnung, dass diese möglichst schnell von statten gehen soll.

Grundsätzlich wird ein Farbwert pro Rasterpunkt benötigt. Dazu wird aus der Koordinate des Rasterpunktes die ursprüngliche Koordinate im dreidimensionalen Raum berechnet. Für diesen Punkt in Raum können dann Oberflächeneigenschaften bestimmt werden (z.B. Farbe) und zusammen mit Informationen über vorhandene Lichtquellen kann die vom Betracher wahrgenommene „Lichtreflektion“ berechnet werden.

Eine Berechnung für jeden Rasterpunkt ist zwar genau, aber in manchen Fällen zu aufwändig. Eine Annäherung ist, solche Beleuchtungs- und Oberflächenberechnungen

[2] Mit Besonderheiten bei der perspektivischen Verzerrung

[3] Streng genommen werden diese Berechnungen nicht für Pixel sonder für *Fragmente* vorgenommen – ein Pixel kann mehrere Fragmente „überdecken“, z.B. zum Zwecke der Kantenglättung. Zur Anschaulichkeit und Verständlichkeit ist es jedoch ausreichend, über „Pixel“ zu reden.

nur für die Eckpunkte des Dreiecks zu berechnen und das Ergebnis dieser Berechnung linear über die Pixel des Dreiecks zu *interpolieren*.

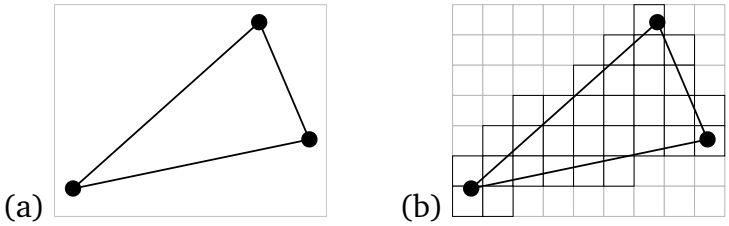

Abbildung 2.2: Rasterung eines Dreiecks:
(a) Nach Transformation in das Koordinatensystem des Monitors.
(b) Vom Dreieck überlagerte Pixel.

Für eine überzeugende Darstellung einer Oberfläche sind deren Details wichtig, z.B. feine Muster in der Oberflächenfarbe. Es liegt nahe, jedem Eckpunkt eines Dreiecks eine Farbe zuzuweisen; aber oftmals sollen die darzustellenden Details kleiner als die Dreiecke, aus denen ein Modell aufgebaut wurde, sein. Zu diesem Zweck werden zweidimensionale Bilder[4] über die Oberfläche „gelegt“. Dazu wird eine Abbildung auf die 3D-Raumkoordinate auf angewendet um eine 2D-Bildkoordinate, an der das Bild ausgelesen wird, zu erhalten. Z.B. kann ein „Blatt Papier“ durch zwei Dreiecke dargestellt werden, aber die Verwendung eines Oberflächenbildes erlaubt es Text usw. darzustellen, was weit detaillierter ist, als es eine einfache Färbung der Dreieckseckpunkte erlauben würde.

An diesem Beispiel erkennt man, dass man nicht immer alle Teile von Beleuchtungs- und Oberflächenberechnungen für jeden Eckpunkt berechnen und dann linear interpolieren kann – dementsprechen lassen sich diese Teile nur sinnvoll berechnen, wenn die Rechnung für jeden Pixel individuell vorgenommen wird.

Andererseits gibt es bei der Dreiecksdarstellung auch Arbeitsschritte, die nur für jeden Eckpunkt berechnet werden können – die Transformation eines Dreiecks vor der Rasterung zum Beispiel: eine Berechnung für jeden Pixel ist schlicht unmöglich, da das Dreieck noch gar nicht gerastert wurde.

In der Praxis werden praktisch immer „Mischformen“ bei der Beleuchtungs- und Oberflächenberechnungen verwendet: einige Werte oder Zwischenergebnisse werden für jeden Eckpunkt berechnet; diese werden dann, bevor sie als Eingabe für eine für ein Pixel auszuführende Berechnung dienen, interpoliert.

[4] „Texturen“

2.4 Verarbeitungsschritte auf 3D-Grafikprozessoren

Moderne Grafikchips (genannt “Graphics Processing Unit”, kurz “GPU”) sind auf Echtzeit-Darstellung von 3D-Grafiken spezialisierte Hardware. Deren logischer Aufbaum, schematisch Abbildung 2.3 dargestellt, ist auf die Darstellung von Dreiecksnetzen ausgerichtet.

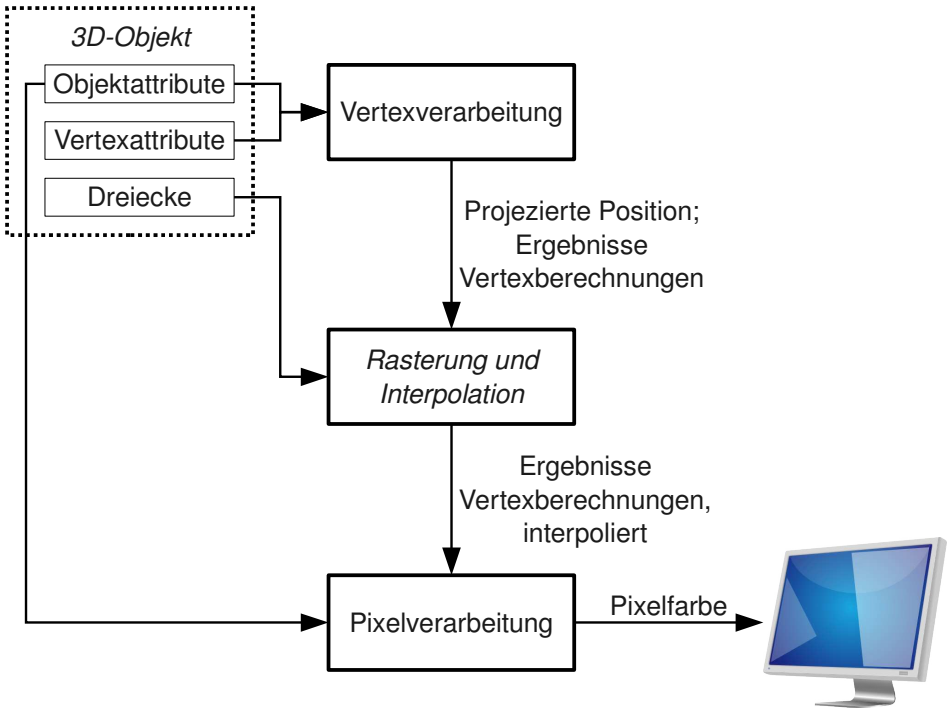

Abbildung 2.3: Schematische Darstellung der 3D-Grafik-Pipeline

Der erste Schritt bei der Darstellung eines 3D-Objekts die Ausführung der für jedes Vertex erforderlichen Berechnungen („Vertexberechnungen“). Dies sind immer mindestens die Anwendung der Transformationen für ein Objekt auf die individuellen Raumkoordinaten der Eckpunkte; in der Regel werden auch, wie oben genannte, Teile der Beleuchtungs- und Oberflächenberechnung vorgenommen.

Die Ergebnisse der Vertexberechnungen sind eine Pixelkoordinate sowie weitere „Zwischenergebnisse“ der Beleuchtungs- und Oberflächenberechnung. Analog zu Vertexattributen liefert jede Vertexberechnung die gleiche Anzahl von Ergebnis„variablen“, aber deren *Werte* unterscheiden sich von Vertex zu Vertex.

Die Pixelkoordinaten der Vertices eines Dreiecks werden von einer speziellen Funktionseinheit verwendet um das Dreieck zu rastern: ausgegeben werden alle Pixel, die vom Dreieck überdeckt werden.

Aus den Ausgaben der Vertexberechnungen – die oben erwähnten „Zwischenergebnisse“ – werden die Eingaben für die Pixelberechnungen abgeleitet. Dies geschieht durch *lineare Interpolation* zwischen den Ergebniswerten der Vertexberechnung für die drei Vertices, aus denen das gerade gerasterte Dreieck besteht. Interpoliert wird auf der Ebene von Ergebnis„variablen“ – die Werte der ersten Ergebnis„variablen“ für jeden Eckpunkt wird bestimmt, zwischen diesen drei Werten wird interpoliert; daraufhin werden die Werte der zweiten Ergebnis„variablen“ für jeden Eckpunkt bestimmt und diese interpoliert; usw. Die Gewichtungsfaktoren für die Interpolation bestimmen sich aus der Position des Pixels, für deren Berechnungen die Eingaben interpoliert werden, auf dem gerasterten Dreieck. Die Faktoren werden von der Funktionseinheit zur Rasterung der Dreiecke mitberechnet.

Abbildung 2.4 zeigt für einen einfachen Fall (horizontale Pixel-Reihe) die Gewichtungsfaktoren für die Ergebnis„variablen“ zweier Eckpunkte eines Dreiecks. Erkennbar ist, dass die Gewichtungsfaktoren des ersten Eckpunkts abnehmen je näher man dem zweiten Eckpunkt kommt; umgekehrt steigern sich dessen Gewichtungsfaktoren in dessen Richtung.

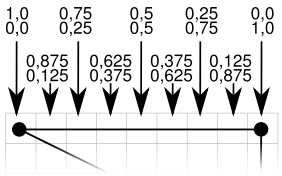

Abbildung 2.4: Gewichtungsfaktoren für die Interpolation zwischen Attributwerten zweier Vertices.
Oberer Faktor wird auf Werte des linken, unterer auf Werte des rechten Vertices angewendet. Gewichtungsfaktor für drittes Vertex kann vernachlässigt werden (ist 0 für alle gezeigten Pixel).

Abbildung 2.5 zeigt beispielhaft das Ergebnis einer Interpolation für ein Pixel im Dreieck aus Abbildung 2.2. Im Bild (a) ist das Beispielpixel hervorgehoben. v_1, v_2 und v_3 sind die Eckpunkte des Dreiecks. f_1, f_2 und f_3 sind die Gewichtungsfaktoren für die Ausgaben der Vertexberechnungen. In Bild (b) wurden diese Faktoren auf eine Ergebnis„variable“ Grauwert angewendet. Die Dreieckseckpunkte wurden mit den Werten dieser Ergebnis„variable“ beschriftet. Für das Beispielpixel wurde der interpolierte Grauwert angegeben.

Definition 5. *Die Eingaben der Pixelberechnungen werden aus den Ausgaben der Vertexberechnungen durch* ***lineare Interpolation*** *abgeleitet.*

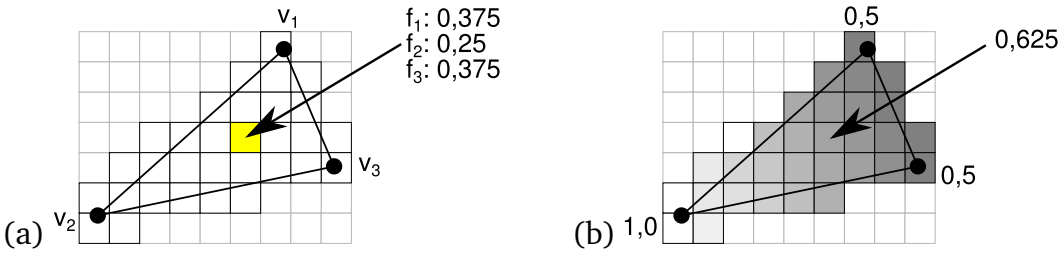

Abbildung 2.5: (a) Gewichtungsfaktoren für die Interpolation der Ausgaben der Vertexberechnungen für ein Dreieck.
(b) Für jeden Eckpunkt vorliegenden Werte und Interpolationsergebnis für einen Grauwert.

Interpoliert wird zwischen den Ausgaben der Vertexberechnungen für die Eckpunkte des gerasterten Dreiecks. Die Gewichtungsfaktoren für die Interpolation bestimmen sich aus der Position des Pixels im Dreieck.

Schliesslich werden die für jeden Pixel vorzunehmenden Berechnungen ausgeführt[5]. Ausgabe dieses Schrittes ist ein Farbwert[6] der für die Darstellung des Pixels auf dem Monitor verwendet wird.

Tatsächlich sind den Pixelberechnungen nur die interpolierten Ausgaben der Vertexberechnung zugänglich – auf für jedes Vertex vorliegenden Eingaben oder Berechnungsergebnisse kann nicht einzeln zugegriffen werden (d.h. bei der Pixelberechnung können nicht Werte wie „Position des ersten Dreieckseckpunktes“ ausgelesen werden).

Die Vertex- wie auch Pixelberechnungen sind unabhängig – d.h. die für ein Vertex oder Pixel ausgeführten Berechnungen hängen nicht von den Werten anderer Vertices oder Pixel ab. (Tatsächlich wird ein „Zugriff“ auf andere Vertices oder Pixel gar nicht erlaubt.) Dies macht die Eckpunkt- und Pixelberechnungen gut parallelisierbar – die jeweiligen Funktionseinheiten der GPUs sind dementsprechend aufgebaut: sie führen gleichzeitig die Berechnungen für mehrere Vertices und Pixel aus. Dazu kommt, das Eckpunkt- und Pixelberechnungen – dem Wesen von 3D-Objekten geschuldet – viel mit Vektoren arbeiten. Die Recheneinheiten der Grafikkarten sind also in der Regel Vektoreinheiten.

Eine wichtige Eigenschaft ist die *Programmierbarkeit* von Vertex- und Pixelberechnungen – die vorgenommenen Berechnungen werden also durch von einem Programmierer bereitgestellte Programme bestimmt.

[5] „Pixelberechnungen“
[6] Eigentlich ein Farbwert (Rot-Grün-Blau-Tripel) sowie eine Deckkraft („Alpha“), für die Darstellung von transparenten Oberflächen

Die GPUs selbst, als Mikroprozessoren, verarbeiten in einem Binärcode vorliegende Programme; dieser Code wird jedoch von den Programmierschnittstellen „versteckt“ und Programme werden in einer Hochsprache gegeben[7]. Die Sprachen werden „Shadingsprachen“ genannt.

Vertex- und Pixelverarbeitung sind zwei logisch getrennte Verarbeitungsschritte, nehmen verschiedenartige Eingabedaten entgegen und werden auf zwei logisch verschiedenenen Funktionseinheiten vorgenommen. Die beiden hauptsächlich verwendetet Programmierschnittstellen für GPUs – OpenGL ([OGL4]) und DirectX ([D3D10]) – spiegeln den Aufbau der 3D-Grafikprozessoren wider und erfordern, dass für jede Funktionseinheit explizit ein passendes Programm bereitgestellt werden muss: zur Programmierung einer GPU müssen zwei Programme – eines, das die Schritte der Vertexverarbeitung beschreibt („Vertexprogramm“), ein anderes für die Beschreibung der Pixelverarbeitung („Pixelprogramm“) – sowie eine „Schnittstellendefinition“ (Ausgaben der Vertexverarbeitung bzw. Eingaben für die Pixelverarbeitung) erstellt werden. Dabei muss darauf geachtet werden, dass diese drei Teile aufeinander abgestimmt sind: eine unpassende Schnittstellendefinition etwa führt meist zu einer falschen Interpretation der Eingaben der Pixelverarbeitung und damit zu unbrauchbaren Ergebnissen.

2.5 Daten eines 3D-Objektes

Bei der Darstellung von 3D-Objekten verwendete Daten, Eingabedaten wie auch Zwischenergebnisse von Berechnungen, liegen in verschieder „Granularität“ vor:

- Einige Eingabedaten liegen für das *Objekt* vor. Dies sind vor allem die Transformationen – verschiedene Objekte haben verschiedene Transformationen, aber es wäre in der Regel unnötig, jedem Eckpunkt ein einem Dreiecksnetz eine Transformation zuzuweisen. Auch einige „Material“eigenschaften – z.B. ein auf die Oberfläche gelegtes Bild – variieren typischerweise von Objekt zu Objekt. Es kann bei diesen Daten von „Objektattributen“ gesprochen werden.

- Für jedes *Vertex* liegen natürlich die Raumkoordinaten der Vertices vor, aber auch die Werte der Vertexattribute – wie u.a. die Oberflächennormale (bei Beleuchtungsberechnungen verwendet).

[7] GLSL bei OpenGL, HLSL bei DirectX

- Für jedes *Pixel* können *Eingabe*daten nicht sinnvoll vorliegen – schliesslich ist ja nicht bekannt, wie ein Dreieck schlussendlich gerastert wird. Allerdings liegen Zwischenergebnisse von Berechnungen per Pixel vor – die interpolierten Ausgaben der Eckpunktberechnung sowie Berechnungen in der Pixelverarbeitung selbst, z.B. das Auslesen eines Bildes zur Färbung der Oberfläche.

Je nachdem, auf welchen Eingabedaten bzw. Operanden eine Berechnung operiert, muss diese verschieden oft ausgeführt werden. Berechnungen, die Operanden verwenden, die in der *Pixelverarbeitung* bestimmt wurden (z.B. aus einem Oberflächenmuster ausgelesene Farbe), müssen für jedes Pixel ausgeführt werden; demgegenüber müssen Berechnungen, die von Daten abhängen, die bloss für jedes *Vertex* vorliegen (z.B. Raumkoordinate) auch nur für jedes *Vertex* ausgeführt werden.

Nun treten praktisch viel mehr Pixel- als Vertex-Berechnungen auf – Abbildung 2.2 zeigt beispielhaft, dass ein durch *drei* Vertices definiertes Dreieck eine *Vielzahl* von Pixeln überdeckt wird.

Eine Folge daraus ist, dass es erstrebenswert ist, möglichst viele Berechnungen in der Vertexverarbeitung vorzunehmen, deren Ergebnisse interpolieren zu lassen und nur in der Pixelverarbeitung auszurechnen, was wirklich nur dort ausgerechnet werden kann: da, auf die gesamte Darstellung einer 3D-Grafik gesehen, in der Regel viel weniger Berechnungen vertexweise als pixelweise ausgeführt werden, verringert die Ausführung einer Operation in der Vertexverarbeitung statt in der Pixelverarbeitung den gesamten Berechnungsaufwand.

2.6 Beispiel

Ein Shadingprogramm, geschrieben in Cg, soll als praktisches Beispiel der Programmierbarkeit von 3D-Grafikprozessoren dienen. Es soll gezeigt werden, wie die Funktionsweise von GPUs sowie einige Konzepte, die in den vorherigen Abschnitten genannt wurden, sich in Shadingprogrammen äussern. Das Beispielprogramm ist in Abbildung 2.6 aufgelistet.

2.6.1 Programmaufbau

Das Beispielprogramm ist zwar als *ein* Quelltext gegeben, trotzdem sind die Programme für die Vertexberechnungen und Pixelberechnungen logisch getrennt – in diesem Fall,

```
struct VertexOutput {
      float4 Position : POSITION;
      float2 TexCoord;
      float3 litColor;
};

void vertex_main (
      in varying float4 Position,
      in varying float2 TexCoord,
      in varying float3 Normal,
      in uniform float4x4 ModelViewProj : state.matrix.mvp,
      in uniform float3 LightColor,
      in uniform float3 LightDirObj,
      out VertexOutput output)
{
      output.Position = mul (ModelViewProj, Position);
      float3 ambient = float3 (0.4);
      output.litColor = LightColor * dot (LightDirObj, Normal);
      output.litColor += ambient;
      output.TexCoord = TexCoord;
}

void pixel_main (in VertexOutput interpolatedVertexOutput,
      in uniform sampler2D Texture,
      out float4 outColor : COLOR)
{
      float3 surface =
            tex2D (Texture, interpolatedVertexOutput.TexCoord).rgb;
      float3 lighting =
            interpolatedVertexOutput.litColor;
      outColor.rgb = surface * lighting;
      outColor.a = 1;
}
```

Abbildung 2.6: Ein Programm-Paar in Cg.

in dem sie durch zwei getrennte Funktionen definiert wurden (`vertex_main` für die Vertexberechnungen bzw. `pixel_main` für die Pixelberechnungen).

Der Strukturtyp `VertexOutput` beschreibt die Ergebnisse der für jedes Vertex vorgenommenen Berechnungen. Gleichzeitig dient er als Typ eines Eingabeparameter des Programms für die Pixelberechnungen, dient also damit auch als „Schnittstellenbeschreibung“ zwischen Vertex- und Pixelprogrammen. Insbesondere sei daran erinnert dass diese Ergebnisse der Vertexberechnungen vor der Eingabe an das Pixelprogramm interpoliert werden.

2.6.2 Eingaben und Ausgaben

Das Beispielprogramm benutzt verschiedenartige Eingaben, sowohl Objektattribute wie auch Werte von Vertexattributen.

Zu den Objektattributen (im Quelltext umrandet), also sich innerhalb eines 3D-Objekts nicht verändernde Werte, gehört zunächst eine *Transformation* (`ModelViewProj`). Es wird erwartet, dass die verschiedenen typischen Transformationen (Objektposition und -rotation, Betrachterposition und -blickrichtung, perspektivische Projektion) zu einer einzigen Transformation konkateniert wurden.

`LightColor`, `LightDirObj` sind Eigenschaften einer Lichtquelle (Farbe bzw. Richtung, aus der Licht scheint); `Texture` (Eingabe des Pixelprogramms) ist ein „Muster“, aus welchem die Oberflächenfarbe abgeleitet wird. Diese Objektattribute bestimmen also Beleuchtung und Aussehen der Oberfläche.

Vertexattribute (hell hinterlegt) sind `Position` (Raumkoordinate), `TexCoord` (2D-Koordinate auf dem Oberflächenmuster) und `Normal` (Oberflächennormale).

Die Eingabe `interpolatedVertexOutput` des Pixelprogramms sind als „für jedes Pixel vorliegend“ (dunkel hinterlegt) markiert. Wie verträgt sich das mit der in Abschnitt 2.5 gemachten Aussage, Eingabedaten könnten nicht sinnvoll für jedes Pixel vorliegen? Die für jedes Pixel vorliegenden Eingabeparameter des Pixelprogramms sind keine „direkten“ Eingaben, wie sie etwa die Werte der Objekt- oder Vertexattribute sind, sondern *abgeleitete* Eingaben – aus den Ergebnissen der Vertexberechnung interpolierte Werte.

Zwei Ausgaben haben eine besondere Bedeutung: `output.Position` des Vertexprogramms und `outColor` des Pixelprogramms. `output.Position` ist die in Monitorkoordinaten transformierte Position eines Vertices. Die Rasterung

der Dreiecke wird anhand dieser Positionen der Dreieckseckpunkte vorgenommen. (In Cg markiert "`: POSITION`" die besondere „Rolle“ dieses Ausgabewertes des Vertexprogramms.)

Die zweite „besondere“ Ausgabe ist der Ausgabeparameter `outColor` des Pixelprogramms. Durch diesen wird die auf dem Monitor darzustellende Pixelfarbe bestimmt. ("`: COLOR`" ist die Cg-Syntax, um den Parameter als „Pixelfarbe“ zu kennzeichnen.) Darüber hinaus wären weitere Ausgaben nicht sinnvoll (so kann ein Monitorpixel nur einen Farbwert besitzen).

2.6.3 Berechnungen

Das Beispielprogramm ist zwar kurz, setzt aber einige der in den Abschnitten 2.2 und 2.3 genannten Konzepte um.

So wird auf das dargestellte Objekt eine *Transformation*(`ModelViewProj`) angewendet und das Ergebnis als Ausgabekoordinate – also als Bildschirmkoordinate für ein Vertex – verwendet (Zuweisung von `output.Position`).

Als einfache Oberflächen- und Beleuchtungsberechnung wird für jedes Vertex ein Beleuchtungswert berechnet, diese werden über die Bildschirmpixel der Dreiecke interpoliert, und schliesslich noch mit einem Oberflächenmuster „eingefärbt“.

Der Beleuchtungswert ist die Ausgabe `output.litColor`. Die Beleuchtungsintensität für die gegebenene Lichtrichtung abhängig von der Oberflächenrichtung (`dot(...)`-Operation) wird mit der Lichtfarbe (`LightColor`) multipliziert, woraus sich die „Beleuchtung“ der Oberfläche an dem Vertex, für den die Vertexberechnungen vorgenommen werden, ergibt. `ambient` ist eine Annäherung von „indirektem Licht“ und wird zu der Gesamtbeleuchtung addiert[8].

Die Interpolation dieses Beleuchtungswertes wird nicht explizit angegeben – diese wird implizit von der GPU während der Dreiecksrasterung vorgenommen. Das Pixelprogramm „sieht“ nur den aus den Ergebnissen der Vertexberechnungen interpolierten Eingabewert.

Der Eingabeparameter `TexCoord` wird von Vertexprogramm unverändert wieder ausgegeben (Zuweisung zu `output.TexCoord`) – ein Pixelprogramm kann keine

[8] Solche Annäherungen werden verwendet, um schattierte Bereiche aufzuhellen, was von menschlichen Betrachtern u.U. als „besser aussehend“ wahrgenommen wird.

Werte von Vertexattributen als Eingaben annehmen, alle solchen Eingaben müssen also im Vertexprogramm explizit „durchgeschleift“ werden. Dabei ist zu beachten, dass, wie alle Ausgaben der Vertexberechnung, auch zwischen `output.TexCoord`-Werten von mehreren Vertices interpoliert wird, bevor es als Eingabe für das Pixelprogramm dient.

Schliesslich wird, im Pixelprogramm, die Oberfläche mit einem „Muster“ versehen. Der Farbwert des Musters, `surface`, wird aus dem Bild `Texture` an den Koordinaten `interpolatedVertexOutput.TexCoord` ausgelesen (`tex2D (...)`-Operation). Die implizite Interpolation von `interpolatedVertexOutput.TexCoord` ist also wünschenswert, da dadurch die Bildkoordinaten, und damit die ausgelesene Musterfarbe, über die Dreieckspixel variieren. (Ohne eine solche Variation – also nur ein Auslesen an einigen wenigen verschiedenen Koordinaten – wäre die Darstellung des Musters nicht sehr detailliert.)

Die Einfärbung der Oberflächenfarbe mit dem Beleuchtungswert wird durch eine einfache Multiplikation der Oberflächenfarbe mit dem Beleuchtungswert erreicht. Damit ergibt sich die ausgegebene Pixelfarbe (`outColor`).

2.6.4 Darstellung

Abbildung 2.7 zeigt ein 3D-Modell, welches mit dem Beispielprogramm aus Abbildung 2.6 dargestellt wurde. Zur Verdeutlichung des Einflusses der verschiedenen Verarbeitungsschritte wurde nicht nur das Endergebnis, sondern auch einige Zwischenergebnisse visualisiert. Abbildung 2.8 zeigt vergrößerte Ausschnitte der Bilder aus Abbildung 2.6.

Bild (a) stellt das 3D-Modell in Drahtgitterdarstellung da. Dies lässt erkennen, wo sich die Vertices des Modells befinden.

Bild (b) zeigt die Anteile der Berechnungen mit „Objektattributen“ (speziell wurde der Wert von `ambient` ausgegeben).

Bild (c) nimmt die Ergebnisse der Vertexberechnungen hinzu (`output.litColor`). In der Vergrößerung lässt sich die Interpolation dieses, als Ausgaben der Vertexberechnungen vorliegenden, Wertes über die Pixel der Dreiecke erkennen.

Bild (d) zeigt das Ergebnis der „kompletten“ Berechnung (`outColor`). Gut erkennbar sind die Details aus dem Oberflächenmuster, wodurch jedes Pixel quasi eine individuelle Farbe erhält: diese Details „maskieren“ die Interpolation des

Beleuchtungswertes, diese ist praktisch nicht auszumachen.

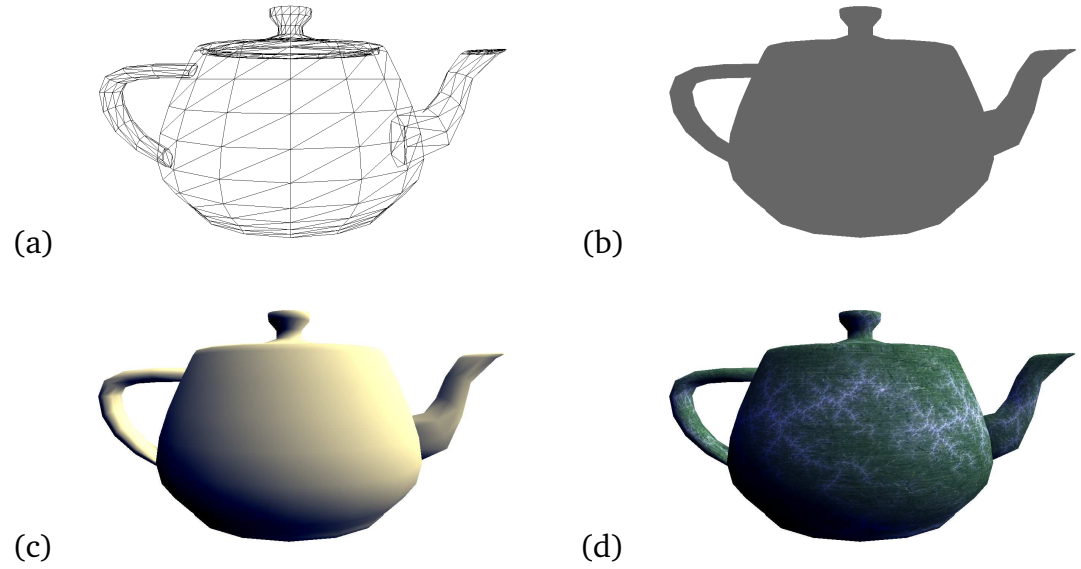

Abbildung 2.7: 3D-Modell, mit Programm aus 2.6 dargestellt
(a): Drahgitterdarstellung
(b): Nur die Anteile aus den Objektattributen
(c): Anteile aus Vertexberechnungen hinzugenommen
(d): Anteile aus Pixelberechnungen hinzugenommen

2.7 Zusammenfassung

Bei der Echtzeit-3D-Grafik werden von der GPU als Dreiecksnetze vorliegende 3D-Modelle dargestellt. Diese Modelle bestehen aus *“Vertices”*, die die Eckpunkte von Dreiecken bilden. Bei der Darstellung eines Modelles wird zuerst die *Vertexverarbeitung* vorgenommen; dabei werden die Raumkoordinaten der Vertices transformiert und weitere Berechnungen auf den Vertices zugeordneten Daten vorgenommen („Vertexattribute“). Als nächsten Schritt der Darstellung werden Dreiecke bei der *Rasterung* auf Pixel des Ausgabegerätes abgebildet. Bei dieser *Pixelverarbeitung* dienen als Eingaben die Ausgaben der Vertexverarbeitung, allerdings *interpoliert*. Es werden weitere Berechnung vorgenommen, um Details hinzuzufügen, für die eine Berechnung per Vertex zu „grob“ wäre, wie das „aufziehen“ von Bilddaten auf das 3D-Modell.

Die Verarbeitungsschritte „Vertexverarbeitung“ und „Pixelverarbeitung“ sind beide programmierbar, jedoch nur getrennt – ein Programmier muss also für jede der Funktionseinheiten ein eigenes Programm schreiben.

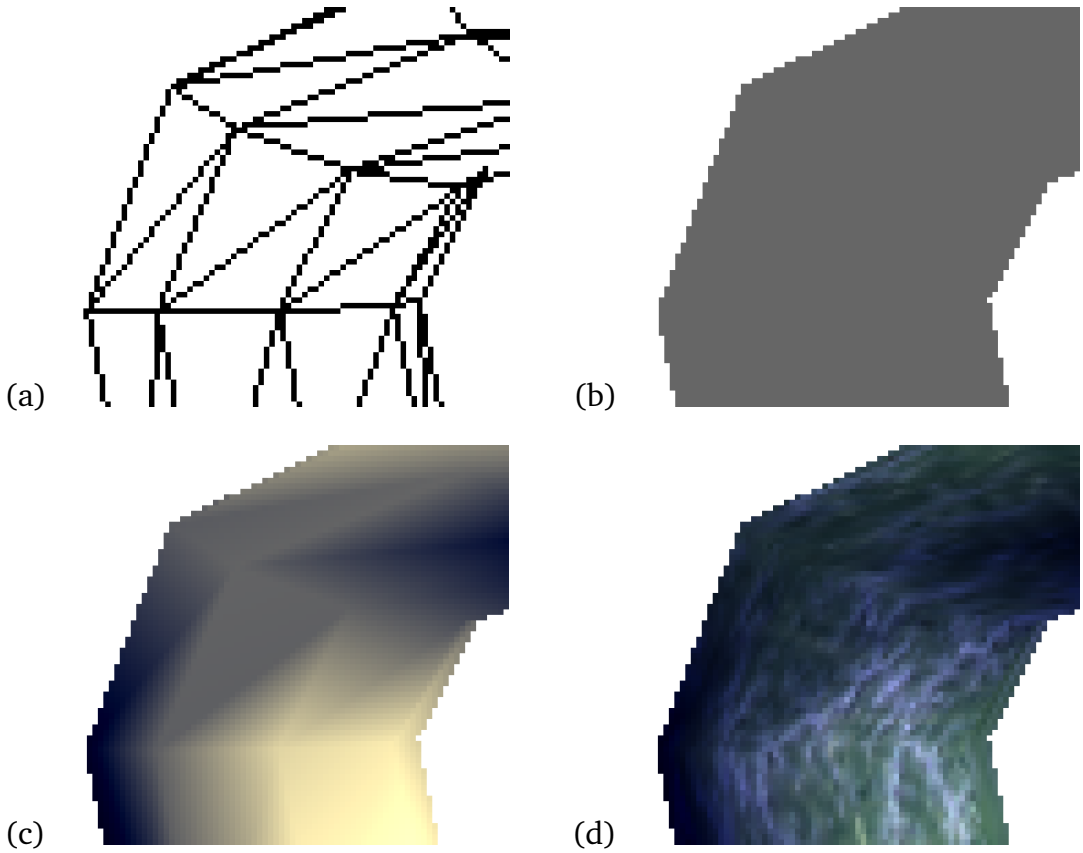

Abbildung 2.8: Vergrößerungen aus Abbildung 2.7

Kapitel 3

Sprachspezifikation

Dieses Kapitel beschreibt konkret eine Ein-Programm-Shadingsprache. Zuerst wird auf Eckpunkte der Syntax eingegangen, danach kommen allgemeinere Aspekte (z.B. wie die Mischung von Operationen verschiedener Berechnungsfrequenzen in verschiedenen Ausführungszweigen) zur Sprache.

3.1 Anforderungen an die Sprache

Auf höherer Ebene soll die Sprache folgende Anforderungen erfüllen:

- *C-ähnliche Syntax*: Um ein schnelles Einarbeiten und Verständnis zu ermöglichen soll sich die Syntax an den verbreiteten Shadingsprachen Cg, DirectX HLSL und OpenGL GLSL orientieren. Diese wiederum sind C-verwandt. Weitere syntaktische Anleihen können auch von passenden Stellen aus anderen Shadingsprachen übernommen werden.

- *Moderne Entwicklungen berücksichtigen*: Die neuesten der C-ähnlichen Sprachen sind Java und C#. Wenn angebracht sollen Konzepte aus diesen übernommen werden.

- *Vereinfachungen für Shading*: Die Sprache soll Elemente besitzen, die das Erstellen von Shading-Programmen vereinfachen.

- *Keine Bindung an bestimmte Architektur*: Die Sprache selbst soll keine bestimmte darunterliegende Hardwarearchitektur voraussetzen; auch keine Architektur-

Klasse wie z.B. „Grafikprozessoren“[1]. Auch soll die Sprache insgesamt weniger hardwarenah als C sein.

Aus diesen Anforderungen leiten sich die folgenden spezifischen Kriterien ab:

- *Syntax*: Die Syntax soll im Wesentlichen auf C basieren. Es sollen Elemente ausgelassen werden, die eher unintuitiv sind und selten praktisch verwendet werden, wie die Oktalnotation bei Integerliteralen. Syntaxerweiterungen sollten Inkonsistenzen oder Auslassungen betreffen - so gibt es bei C z.B. nur bei der Definition einer Array-Variable die Möglichkeit, dieser mehrere Elemente auf einmal zuzuweisen. Bei einer regulären Zuweisung ist dies bei C nicht möglich, soll aber in der neuen Sprache ermöglicht werden.

- *Zeichensatz*: ANSI C basiert auf dem ASCII-Zeichensatz. Mit Unicode gibt es es einen Zeichensatz, der alle weltweit geschriebenen Sprachen umfasst; dieser hat auch in den Definitionen von Programmiersprachen Einzug gehalten und erlaubt z.B. Bezeichner aus anderen Sprachen als Englisch zu deklarieren. Aus technischer Sicht gibt es keinen Grund, nicht Unicode als Basis für den Zeichensatz zu benutzen.

- *Überdeckung von Bezeichnern*: Bei C# können Bezeichner nicht andere Bezeichner überdecken. Dies verhindert eine Klasse von Fehlern, bei der durch Überdeckung eine andere Variable benutzt wird, als der Programmierer angenommen hat. Das Verbot der Überdeckung soll übernommen werden.

- *Shadingspezifische Elemente*: Es sollen Vektor- und Matrixtypen vorhanden sein.

- *Keine Bindung an bestimmte Architektur*: Cg besitzt syntaktische Elemente, um bestimmte Variablen an von der Grafikhardware bereitgestellte Werte zu binden. Auch gibt es einen Mechanismus zur Überladung von Funktionen für verschiedene Zielhardware. Solche Elemente sollen *nicht* in der neuen Sprache enthalten sein.

- *Maschinenferne*: Dynamische Verweise und Zeiger werden nicht unterstützt. Statische Verweise (z.B. für Ausgabevariablen bei Funktionen) sollten aber unterstützt werden.

[1] Sinnvollerweise wird eine von-Neumann-Architektur minimal vorausgesetzt.

3.2 Programmumgebung

Ein Shadingprogramm läuft praktisch immer im Kontext einer Anwendung, die Parameter direkt oder indirekt (wie Vertexdaten) an das Shadingprogramm übergibt. Bei den Parametern der Hauptfunktion des Shadingprogramms, welche von diesem selbst nicht definiert werden, wird deshalb davon ausgegangen, das diese von der umgebenden Anwendung spezifiziert werden.

3.3 Spezifikation

Anmerkung: Die Syntaxbeschreibung erfolgt auf der Ebene lexikalischer Einheiten. Deren Regeln sind in 3.3.8 beschrieben.

Die Notation ist im Wesentlichen die erweiterte Backus-Naur-Form. Die komplette rechte Seite einer Regel ist eingerückt. Besteht eine Regel aus mehreren Zeilen, so ist jede Zeile eine alternative Ableitung.

3.3.1 Programm

programm:

[*programm_statements*(3.3.1)] *EOF*(3.3.1)

programm_statements:

funktion_definition(3.3.5.1) [*programm_statements*(3.3.1)]

typ_definition(3.3.4.7) ‘**;**’ [*programm_statements*(3.3.1)]

dekl_var(3.3.6.1) ‘**;**’ [*programm_statements*(3.3.1)]

dekl_konst(3.3.6.2) ‘**;**’ [*programm_statements*(3.3.1)]

Ein *Programm* besteht aus jeweils keinen bis mehreren Typdefinitionen, Variablen-, Konstanten- und Funktionsdeklarationen. Der Gültigkeitsbereich dieser Deklarationen ist global.

Das Terminal *EOF* markiert das Ende der Eingabe.

3.3.1.1 Eintrittsfunktion

Der Eintrittspunkt der Ausführung eines Programms ist eine Funktion (3.3.5) namens „`main`“. Ist eine solche Funktion nicht definiert, so kann das Programm nicht ausgeführt werden.

Die Eintrittsfunktion muss den Rückgabetyp `void` besitzen.

Es müssen genau zwei Ausgabeparameter vom Typ `float4` (siehe *typ_vektor_float*(3.3.4.3)) deklariert werden. Der erste Ausgabeparameter nimmt die transformierte und projizierte Position entgegen. Der zweite Ausgabeparameter nimmt die auszugebende Pixelfarbe entgegen.

Es können beliebig viele Eingabeparameter beliebigen Typs für die Eintrittsfunktion deklariert werden. Die Werte dieser Parameter entstammen der Programmumgebung.

3.3.2 Blöcke

block:

typ_definition(3.3.4.7) [*block*(3.3.2)]
dekl_var(3.3.6.1) ‘**;**’ [*block*(3.3.2)]
dekl_konst(3.3.6.2) ‘**;**’ [*block*(3.3.2)]
kommando(3.3.2) [*block*(3.3.2)]

kommando:

ausdruck(3.3.3) ‘**;**’
‘`return`’ [*ausdruck*(3.3.3)] ‘**;**’
verzweigung(3.3.7.1)
schleife_for(3.3.7.2)
schleife_while(3.3.7.3)
‘`{`’ [*block*(3.3.2)] ‘`}`’

Ein *Block* besteht aus Typdefinitionen, Variablen- und Konstantendeklarationen sowie keinem bis mehreren Kommandos. Der Gültigkeitsbereich dieser Deklarationen endet mit dem Ende des Blockes. Kommandos werden zur Laufzeit in Reihenfolge ihres Auftretens ausgeführt.

Ein `return`-Kommando führt zum sofortigen Verlassen der umschließenden Funktion. Bei Funktionen mit einem anderen Rückgabetyp als `void` muss ein *Ausdruck* mit angegeben werden, wobei der Rückgabewert der Funktion auf den Wert des Ausdrucks gesetzt wird.

3.3.3 Ausdrücke

ausdruck:

asdr_basis(3.3.3.1)
asdr_zuweisung(3.3.3.2)
asdr_logisch_oder(3.3.3.4)
asdr_ternaer(3.3.3.7)

Der Ausdruck höchster Präzedenz ist die Zuweisung.

3.3.3.1 Basisausdruck

asdr_basis:

‘(’ *ausdruck*(3.3.3) ‘)’ [*attribut_liste_oder_array_element*(3.3.3.1)]
asdr_konst_bool(3.3.3.8) [*attribut_liste_oder_array_element*(3.3.3.1)]
funktion_aufruf(3.3.5.2) [*attribut_liste_oder_array_element*(3.3.3.1)]
BEZEICHNER(3.3.8.1) [*attribut_liste_oder_array_element*(3.3.3.1)]
NUMERIC(3.3.8.2) [*attribut_liste_oder_array_element*(3.3.3.1)]

attribut_liste_oder_array_element:

‘.’ *attribut*(3.3.3.9) [*attribut_liste_oder_array_element*(3.3.3.1)]
‘[’ *ausdruck*(3.3.3) ‘]’ [*attribut_liste_oder_array_element*(3.3.3.1)]

Basisausdrücke sind die Ausdrücke niedrigster Präzedenz. Dies sind neben Bezeichnern für Variablen und Konstanten auch Funktionsaufrufe und geschachtelte Ausdrücke. Weiterhin können Zugriffe auf Attribute (siehe *attribut*(3.3.3.9)) bzw. Arrayelemente spezifiziert werden.

3.3.3.2 Zuweisung

asdr_zuweisung:

BEZEICHNER(3.3.8.1) [*attribut_liste_oder_array_element*(3.3.3.1)] ‘=’ *ausdruck*(3.3.3)

Ein Zuweisungsausdruck setzt sich aus einer „linken Seite“, einer Variable oder einem Arrayelement bzw. einem Swizzle-Attribut (siehe 3.3.4.3) davon, sowie einer „rechten Seite“, dem direkt folgenden, zuzuweisenden *ausdruck*(3.3.3), zusammen.

Dem Ausdruck der linken Seite wird der Wert des Ausdrucks auf der rechten Seite zugewiesen.

Der Typ des zugewiesenen Ausdrucks muss zuweisungskompatibel (siehe unten) zur linken Seite der Zuweisung sein.

Ein Zuweisungsausdruck selbst hat den Wert der linken Seite nach der Zuweisung.

Zuweisungskompatibilität Ein Ausdruck ist *zuweisungskompatibel* zu einem bezeichneten Wert wenn Wert und Ausdruck den selben Typ besitzen oder eine implizite Typumwandlung möglich ist.

Ein Ausdruck ist *verlustfrei zuweisungskompatibel* zu einem bezeichneten Wert wenn Wert und Ausdruck den selben Typ besitzen oder eine implizite Typumwandlung ohne Präzisionsverlust möglich ist.

Siehe auch 3.3.4.8.

3.3.3.3 Binäre Ausdrücke

Ein *binärer Ausdruck* wendet eine arithmetische, bitweise, logische oder vergleichende Verknüpfung (bestimmt durch den Operator) auf die Werte zweier Ausdrücke an.

Werden zwei Vektoren verknüpft entspricht dies einer Verknüpfung der individuellen Komponenten an der gleichen Stelle der Vektoren (`x` mit `x`, `y` mit `y`, ...). Zwei verknüpfte Vektoren müssen die gleiche Komponentenanzahl besitzen.

Bei der Auswertung wird zunächst der linke, dann der rechte Operand ausgewertet, und anschließend die Verknüpfung angewendet.

3.3.3.4 Logische Operatoren

asdr_logisch_oder:
    asdr_logisch_und(3.3.3.4) { '||' *asdr_logisch_und*(3.3.3.4) }
asdr_logisch_und:
    asdr_gleichheit(3.3.3.5) { '&&' *asdr_gleichheit*(3.3.3.5) }

Beide Operanden müssen vom Typ `bool` sein.

'&&' logisch UND-verknüpft die Operanden.

'||' logisch ODER-verknüpft die Operanden.

3.3.3.5 Vergleichsoperatoren

asdr_gleichheit:
    asdr_vergleich(3.3.3.5) { ('!=' | '==') *asdr_vergleich*(3.3.3.5) }
asdr_vergleich:
    asdr_add(3.3.3.5) { ('>' | '>=' | '<' | '<=') *asdr_add*(3.3.3.5) }

'>' wertet aus ob der linke Operand größer als der rechte Operand ist.

'>=' wertet aus ob der linke Operand größer oder gleich dem rechten Operanden ist.

‘<’ wertet aus ob der linke Operand kleiner als der rechte Operand ist.

‘<=’ wertet aus ob der linke Operand kleiner oder gleich dem rechten Operanden ist.

‘==’ wertet aus ob der linke Operand gleich dem rechten Operand ist.

‘!=’ wertet aus ob der linke Operand nicht gleich dem rechten Operanden ist.

Beide Operanden müssen von einem Integer- oder Fließkommatyp (*typ_num*(3.3.4.2)) oder einem dazu zuweisungskompatiblen Typ sein.

Sind beide Operanden vom Typ `unsigned int` oder `int`, so findet ein Vergleich von Ganzzahlwerten statt.

Ist ein Operand vom Typ `int` und ein Operand vom Typ `int`, `unsigned int` oder verlustfrei zuweisungskompatibel zu `int` oder `unsigned int`, so findet ein Vergleich von Ganzzahlwerten statt.

Ist ein Operand von einem Fließkommatyp und ein Operand von einem Integer- oder Fließkommatyp oder zuweisungskompatibel zu dem Fließkommatyp des anderen Operanden, so findet ein Vergleich von Fließkommawerten statt.

Arithmetische Operatoren

asdr_add:

asdr_mult(3.3.3.5) { (‘+’ | ‘−’) *asdr_mult*(3.3.3.5) }

asdr_mult:

asdr_unaer(3.3.3.6) { (‘*’ | ‘/’ | ‘%’) *asdr_unaer*(3.3.3.6) }

‘+’ addiert rechten und linken Operanden.

‘−’ subtrahiert den rechten von dem linken Operanden.

‘*’ multipliziert rechten und linken Operanden.

‘/’ dividiert den linken durch den rechten Operanden.

‘%’ ist der Divisionsrest der Division des linken Operators durch den rechten Operator.

Beide Operanden müssen von einem Integer- oder Fließkommatyp (*typ_num*(3.3.4.2)) oder einem dazu zuweisungskompatiblen Typ sein.

Sind beide Operanden vom Typ `unsigned int`, so ist der Typ des ausgewerteten Ausdrucks `unsigned int`.

Ist ein Operand vom Typ `int` und ein Operand vom Typ `int`, `unsigned int` oder verlustfrei zuweisungskompatibel zu `int`, so ist der Typ des ausgewerteten Ausdrucks `int`.

Ist ein Operand von einem Fließkommatyp und ein Operand von einem Integer- oder Fließkommatyp oder zuweisungskompatibel zu dem Fließkommatyp des anderen Operanden, so ist der Typ des ausgewerteten Ausdrucks von einem Fließkommatyp.

3.3.3.6 Unäre Ausdrücke

asdr_unaer:

[('~' | '–' | '!')] *asdr_basis*(3.3.3.1)

Ein *unärer Ausdruck* transformiert den Wert des Operanden.

Ein unärer Operator, auf einen Vektor angewendet, entspricht der Anwendung auf die individuellen Komponenten.

'–' negiert den Operanden. Der Operand muss von einem Integertyp oder Fließkommatyp sein. Ist der Operand von einem Integertyp, so ist der Typ des unären Ausdrucks `int`. Ist der Operand von einem Fließkommatyp, so ist der Typ des unären Ausdrucks vom selben Typ.

'~' bitweise invertiert den Operanden. Der Operand muss von einem Integertyp sein. Der Typ des unären Ausdrucks ist vom selben Typ.

'!' logisch invertiert den Operanden. Der Operand muss vom Typ `bool` sein. Der Typ des unären Ausdrucks ist vom Typ `bool`.

3.3.3.7 Ternärer Ausdruck

asdr_ternaer:

ausdruck(3.3.3) '?' *ausdruck*(3.3.3) '**:**' *ausdruck*(3.3.3)

Ein ternärer Ausdruck setzt sich aus einem *ausdruck*(3.3.3) (der „Bedingung") sowie zwei weiteren Ausdrücken, dem *Wahr-Ausdruck* und dem *Falsch-Ausdruck*, zusammen. Die *Bedingung* muss ein boolescher Ausdruck sein. Ergibt sich dieser Ausdruck zu `true`, so wird der *Wahr-Ausdruck* (dem `?` folgend) ausgewertet, und der Wert des ternären Ausdrucks ergibt sich zu dem Wert des *Wahr-Ausdrucks*. Ergibt sich die *Bedingung* zu `false`, so wird der *Falsch-Ausdruck* (dem **:** folgend) ausgewertet, und der Wert des ternären Ausdrucks ergibt sich zu dem Wert des *Falsch-Ausdrucks*.

Wahr- und *Falsch-Ausdruck* müssen vom gleichen Typ sein.

3.3.3.8 Boolesche Werte

asdr_konst_bool:

(‘`true`’ | ‘`false`’)

`true` (wahr) und `false` (unwahr) sind Werte vom Typ `bool` (siehe *typ_bool*(3.3.4.1)).

3.3.3.9 Attribute

attribut:

BEZEICHNER(3.3.8.1)

Einige Variablen (und Konstanten) besitzen über den Wert hinaus weitere Eigenschaften, die über Attribute abgefragt werden können.
Attributbezeichner sind dabei keine reservierten Schlüsselwörter, d.h. sie können auch als Variablenbezeichner, Funktionsbezeichner etc. verwendet werden.

```
float4x4 matrix (...);
// Erlaubt:
float4x4 inverted = // 'inverted' hier Variablenbezeichner
  matrix.inverted;  // 'inverted' hier Attribut von 'matrix'
```

Vektorattribute werden in 3.3.4.3 beschrieben.
Matrixattribute werden in 3.3.4.4 beschrieben.
Arrayattribute werden in 3.3.4.6 beschrieben.

3.3.4 Typen

typ_basis:

typ_num(3.3.4.2)
typ_bool(3.3.4.1)
typ_vektor(3.3.4.3)
typ_matrix(3.3.4.4)
typ_sampler(3.3.4.5)
BEZEICHNER(3.3.8.1)

typ:

typ_basis(3.3.4)
typ_array(3.3.4.6)

3.3.4.1 Boolescher Typ

typ_bool:
‘bool’

Der Typ `bool` spezifiziert den booleschen Typ mit den möglichen Werten `true` (wahr) oder `false` (unwahr).

3.3.4.2 Numerische Typen

typ_num:
[‘unsigned’] ‘int’
‘float’

Der Typ `int` spezifiziert vorzeichenbehaftete Ganzzahlwerte.
Der Typ `unsigned int` spezifiziert vorzeichenlose Ganzzahlwerte.
Genauigkeit und Wertebereich sind jeweils implementierungsabhängig.
Der Typ `float` spezifiziert Fließkommazahlen.

3.3.4.3 Vektortypen

typ_vektor:
typ_vektor_int(3.3.4.3)
typ_vektor_float(3.3.4.3)
typ_vektor_bool(3.3.4.3)
typ_vektor_int:
[‘unsigned’] (‘int1’ | ‘int2’ | ‘int3’ | ‘int4’)
typ_vektor_float:
(‘float1’ | ‘float2’ | ‘float3’ | ‘float4’)
typ_vektor_bool:
(‘bool1’ | ‘bool2’ | ‘bool3’ | ‘bool4’)

Der *Basistyp* eines Vektors ist ein Integer-, Boolescher oder Fließkommatyp.

Die Zahl *N* nach dem Basistyp muss eine Ganzzahl mit Wert zwischen 1 und 4 inklusive sein und gibt die Anzahl der Vektorkomponenten an. Werte von Vektortypen sind damit *N*-Tupel von Werten des Basistyps.

Vektorkonstruktoren Vektorkonstruktoren sind besondere, vorgegebene Funktionen, die gleiche Bezeichner wie Vektortypen haben (siehe auch *funktion_aufruf* (3.3.5.2)).

Wird ein Vektor aus *einem* Basisausdruck konstruiert, so haben alle Komponenten diesen Wert. Der *Basisausdruck* muss als Typ einen Basistyp besitzen.

Wird ein Vektor aus *mehreren* Basis- oder Vektorausdrücken oder einem Vektorausdruck konstruiert, so wird der ersten Komponente der Wert des ersten Basisausdrucks bzw. der ersten Vektorausdruckskomponente zugewiesen, der zweiten Komponente der Wert des nächsten Basisausdrucks bzw. der nächsten Vektorausdruckskomponente usw. Allen Komponenten muss ein Wert zugewiesen werden. Die Anzahl aller Basisausdrücke und Komponenten der Vektorausdrücke zusammen muss exakt der Komponentenanzahl entsprechen. Alle verwendeten Vektortypen müssen vom gleichen Basistyp wie der zu konstruierende Vektor sein.

```
float3 a = float3 (1.0);           // a = (1.0, 1.0, 1.0)
float3 b = float3 (1.0, 2.0, 3.0);
float4 c = float4 (b, 0.5);        // c = (1.0, 2.0, 3.0, 0.5)
float3 d = float3 (1.0, 2.0);      // Fehler
                                   // (nicht alle Komponenten gegeben)
```

Vektorattribute Einzelne Komponenten können entweder mit Array-Syntax oder den Attributen `x`, `y`, `z`, `w` bzw. `r`, `g`, `b`, `a` angesprochen werden. `x`/`r` spezifiert die erste Komponente, `y`/`g` die zweite Komponente, `z`/`b` die dritte Komponente, `w`/`a` die vierte Komponente. Ansprechen einer nicht existierenden Komponente führt zu einem Fehler.

Weitere Attribute sind die „Swizzle“-Attribute (*SWIZZLE_RGBA* (3.3.8.1), *SWIZZLE_XYZW* (3.3.8.1)). Das Ergebnis eines Swizzles ist vom Typ eines Vektors des Basistyps, mit der Anzahl der Komponenten entsprechend der Anzahl der Komponenten im „Swizzle“. Die erste Komponente des Wertes entspricht der Komponente des Ursprungsvektors, die an erster Stelle des Swizzles identifiziert wird, usw.

```
float4 sampleVec = float4 (1.0, 2.0, 3.0, 4.0);
float  a = sampleVec.x;    // a = 1.0
float2 b = sampleVec.wz;   // b = (4.0, 3.0)
float4 c = sampleVec.xxxx; // c = (1.0, 1.0, 1.0, 1.0)
float4 d = sampleVec.zwyy; // d = (3.0, 4.0, 2.0, 2.0)
float4 e = sampleVec.bgra; // e = (3.0, 2.0, 1.0, 4.0)
float4 f = sampleVec.xyba; // Fehler (XYZW/RGBA gemischt)
```

```
float  g = b.z;            // Fehler (Komponente existiert nicht)
```

3.3.4.4 Matrixtypen

typ_matrix:

typ_matrix_float(3.3.4.4)

typ_matrix_int(3.3.4.4)

typ_matrix_bool(3.3.4.4)

typ_matrix_float:

('float1x1'|'float2x1'|'float3x1'|'float4x1')
('float1x2'|'float2x2'|'float3x2'|'float4x2')
('float1x3'|'float2x3'|'float3x3'|'float4x3')
('float1x4'|'float2x4'|'float3x4'|'float4x4')

typ_matrix_int:

['unsigned'] ('int1x1'|'int2x1'|'int3x1'|'int4x1')
['unsigned'] ('int1x2'|'int2x2'|'int3x2'|'int4x2')
['unsigned'] ('int1x3'|'int2x3'|'int3x3'|'int4x3')
['unsigned'] ('int1x4'|'int2x4'|'int3x4'|'int4x4')

typ_matrix_bool:

('bool1x1'|'bool2x1'|'bool3x1'|'bool4x1')
('bool1x2'|'bool2x2'|'bool3x2'|'bool4x2')
('bool1x3'|'bool2x3'|'bool3x3'|'bool4x3')
('bool1x4'|'bool2x4'|'bool3x4'|'bool4x4')

Der *Basistyp* einer Matrix ist ein Integer-, Boolescher Typ oder Fließkommatyp. Dem Basistyp folgt eine Zeichenkette der Form $N \times M$. N und M sind vom Typ Ganzzahl, können jeweils Werte zwischen 1 und 4 inklusive annehmen, und geben die Spalten- und Zeilenzahl der Matrix an. Ein Matrixtyp spezifiziert eine $N \times M$-Matrix von Werten des Basistyps.

Matrixkonstruktoren Matrixkonstruktoren sind besondere, vorgegebene Funktionen die gleiche Bezeichner wie Matrixtypen haben (siehe auch *funktion_aufruf* (3.3.5.2)).

Eine Matrix wird aus einem oder mehreren Vektorausdrücken zeilenweise konstruiert. Den Zeilen werden jeweils die Werte der Vektorausdrücke zugewiesen.

Die Komponentenanzahl der Vektoren muss dabei *N* sein. Die Anzahl der gegebenen Vektoren muss *M* sein.

```
int2 v1 = int2 (1, 2);
int2 v2 = int2 (3, 4);
int2x2 m1 = int2x2 (v1, v2); // m1 = (1, 2, 3, 4);
int3x2 m2 = int3x2 (v1, v2); // Fehler: v1, v2 besitzen
                             // nicht 3 Komponenten
int3x2 m3 = int3x2 (int3 (v1, 0),  // m3 = (1, 2, 0,
                    int3 (v2, 0)); //       3, 4, 0)
```

Matrixattribute Einzelne Zeilen können über das Attribute `row` angesprochen werden. Dieses verhält sich wie ein Array mit *M* Elementen des Vektortyps *BasistypN*.

Einzelne Spalten können über das Attribut `col` angesprochen werden. Dieses verhält sich wie ein Array mit *N* Elementen des *BasistypM*.

Das Attribut `transposed` ist eine Matrix vom Typ *BasistypM*x*N* und hat als Wert die transponierte Ursprungsmatrix.

Matrizen mit gleichem *N* und *M* besitzen das Attribut `inverted`. Es ist eine Matrix vom Typ *BasistypN*x*M* und hat als Wert die invertierte Ursprungsmatrix. Das Ergebnis ist undefiniert, wenn die Ursprungsmatrix nicht invertierbar ist.

```
float2x2 m1 = float2x2 (float2 (0, 2),
                        float2 (0.2, 0));
int2 r1 = m1.row (1);       // r1 = (0.2, 0)
int2 c1 = m1.col (1);       // c1 = (2, 0)
float2x2 t = m1.transposed; // t = (0, 0.2, 2, 0)
float2x2 i = m1.inverted;   // i = (0, 0.5, 5, 0)
```

3.3.4.5 Samplertypen

typ_sampler:

(‘sampler1D’ | ‘sampler2D’ | ‘sampler3D’ | ‘samplerCUBE’)

Samplertypen repräsentieren Textureinheiten der Hardware. Texturen verschiedener Dimensionalität müssen in Shadingprogrammen verschiedenartig angesprochen werden; dies bedingt die mehrfachen Samplertypen, die jeweils einen speziellen Texturtyp reflektieren.

Samplertypen sind *undurchsichtig* – Samplerwerte können auf der „rechten Seite“ in Zuweisungen zu Variablen exakt gleichen Samplertyps, als Funktionsparameter usw. verwendet werden, aber nicht in Verknüpfungsoperationen u.ä.: Samplerwerte können

im Wesentlich nur mit den vordefinierten Texturfunktionen (3.4.1.6) zum Auslesen von Texturdaten sinnvoll verwendet werden.

3.3.4.6 Arraytypen

typ_array:

typ(A.1.2) '[' ']'

Der Typ der Elemente im Arraytyp wird durch den *Basistyp* spezifiziert (siehe auch *typ*(A.1.2); insbesondere ist der Basistyp nicht auf *typ_basis*(3.3.4) beschränkt sondern kann selbst ein Arraytyp sein). Die Anzahl der Elemente in individuellen Variablen oder Konstanten eines Arraytyps ergibt sich aus der Anzahl der Elemente des zugewiesenen Array-Wertes.

Elemente einer Array-Variable oder -Konstante können mit `Bezeichner[`*Index*`]` angesprochen werden. Das erste Element wird mit dem Wert 0 für *Index* angesprochen. Liegt *Index* ausserhalb des gültigen Bereiches $(0 \ldots N-1)$ so ist der ausgelesene Wert undefiniert.

Arraykonstruktoren Arraykonstruktoren sind besondere, vorgegebene Funktionen die gleiche Bezeichner wie Arraytypen haben (siehe auch *funktion_aufruf*(3.3.5.2)).

Ein Array wird aus keinem, einem oder mehreren Werten des Elementtyps konstruiert. Die Anzahl der Elemente entspricht der Anzahl der dem Konstruktor übergebenen Ausdrücke.

```
int[] a = int[] (1, 2, 3); // 'a' besitzt 3 Elemente
int e1 = a[1];             // e1 = 2
int e2 = a[4];             // e2 ist undefiniert
```

Arrayattribute Arrays besitzen ein Attribut `length` vom Typ `unsigned int` dessen Wert die Anzahl der Elemente im Array ist.

```
int[] a = int[] (1, 2, 3);
unsigned int l = a.length; // l = 3
```

3.3.4.7 Typdefinitionen

typ_definition:

'`typedef`' *typ*(A.1.2) *BEZEICHNER*(3.3.8.1)

typ_definition(3.3.4.7) deklariert einen neuen Typ mit dem angegeben Bezeichner der ein Alias für den durch ‚*typ*‘ bezeichneten Typen ist.

Die Sichtbarkeit des Typbezeichners beginnt hinter der *typ_definition*(3.3.4.7) und reicht bis zum Ende des Gültigkeitsbereichs in dem die Definition vorgenommen wurde.

Der Bezeichner darf keinen anderen Bezeichner der umgebenden Gültigkeitsbereiche überdecken.

3.3.4.8 Typumwandlung

Implizite Typumwandlungen Ausdrücke eines numerischen Typs können in einen anderen numerischen Typ umgewandelt werden. Wird ein Ausdruck einen numerischen Typs einer Variable oder einem formalen Funktionsparameter eines anderen numerischen Typs zugewiesen, findet eine *implizite* Typumwandlung statt (Zuweisungskompatibilität).

Ist *verlustfreie* Zuweisungskompatibilität verlangt, wird eine implizite Typumwandlung nur vorgenommen, wenn kein Präzisionsverlust auftritt (nach nachfolgenden Definitionen). Tritt ein Präzisionsverlust auf, ist das Programm ungültig.

`int` und `unsigned int` werden als immer untereinander ohne Präzisionsverlust zuweisbar angenommen.

Eine Zuweisung von `int` oder `unsigned int` an `float` wird immer als ohne Präzisionsverlust zuweisbar angenommen.

Eine Zuweisung von `float` zu `int` oder `unsigned int` wird immer mit Präzisionsverlust zuweisbar angenommen.

Explizite Typumwandlungen Ausdrücke können explizit in einen anderen Typ umgewandelt werden (syntaktisch ein Funktionsaufruf mit Typbezeichner als Funktionsbezeichner). Umwandlungen mit Präzisionsverlust sind dabei erlaubt. Ein Ausdruck einer expliziten Typumwandlung hat den Typ in den explizit umgewandelt wurde.

3.3.5 Funktionen

3.3.5.1 Definition

funktion_definition:
    funktion_kopf(3.3.5.1) ‘{’ *block*(3.3.2) ‘}’
funktion_kopf:
    funktion_typ(3.3.5.1) *BEZEICHNER*(3.3.8.1) ‘(’ [*funktion_param_formal_liste*(3.3.5.1)] ‘)’
funktion_typ:
    typ(A.1.2)
    ‘`void`’
funktion_param_formal_liste:
    funktion_param_formal(3.3.5.1) { ‘`,`’ *funktion_param_formal*(3.3.5.1) }
funktion_param_formal:
    [‘`in`’] [‘`out`’] *typ*(A.1.2) *BEZEICHNER*(3.3.8.1) [‘=’ *ausdruck*(3.3.3)]

funktion_definition(3.3.5.1) deklariert eine Funktion mit dem gegebenen Bezeichner.

Die Sichtbarkeit des Funktionsbezeichners beginnt hinter dem ‘)’ von *funktion_definition*(3.3.5.1) und reicht bis zum Ende des Programms.

Der Bezeichner darf keinen anderen Bezeichner des globalen Gültigkeitsbereiches überdecken. Ausnahme: Mehrere Funktionen können den gleichen Bezeicher besitzen; es handelt sich hierbei um *überladene* Funktionen (siehe 3.3.5.3).

funktion_typ(3.3.5.1) spezifiziert den Typ des Rückgabewertes der Funktion. Der spezielle Typ `void` gibt an, dass die Funktion keinen Wert zurückgibt. Weder ist der Typ `void` in irgendeinen anderen Typ umwandelbar, noch ist eine Umwandlung von irgendeinem Typen nach `void` möglich.

funktion_param_formal(3.3.5.1) beschreibt die Parameter, welche die Funktion annimmt. Parameter können Eingabe- und/oder Ausgabeparameter sein: auf *Eingabeparameter* kann nur lesend zugegriffen werden, ein Schreibzugriff ist nicht erlaubt. *Ausgabeparameter* können beschrieben werden und jede Änderung wirkt sich auch sofort auf den zugeordneten aktuellen Parameter aus. Ausgabeparameter können auch gelesen werden, allerdings ist ihr Wert vor dem ersten Beschreiben undefiniert. *Ein- und Ausgabeparameter* können jederzeit gelesen werden, jeder Schreibzugriff wirkt sich aber auch hier auf den zugeordneten aktuellen Parameter aus.

3.3.5.2 Aufruf

funktion_aufruf:

typ(A.1.2) ‘(’ [*funktion_param_aktuell*(3.3.5.2)] ‘)’

funktion_param_aktuell:

ausdruck(3.3.3) { ‘,’ *ausdruck*(3.3.3) }

Funktionen werden mit *funktion_aufruf*(3.3.5.2) aufgerufen. Zunächst werden die gegebenen Werteparameter(*funktion_param_aktuell*(3.3.5.2)) den Funktionsparametern der Funktion über ihre Stelle zugeordnet. Basierend auf der Anzahl der Parameter und den Typen der Parameterausdrücke wird eine Überladung der Funktion ausgewählt. Den Funktionsparametern werden die entsprechenden Werte der Parameterausdrücke zugewiesen. Die Parameterausdrücke werden von links nach rechts ausgewertet. Die Programmausführung springt dann zu dem ersten Kommando des Funktionsblockes. Nach Beendigung von dessen Ausführung wird dem Wert des Aufrufausdrucks der Rückgabewert der Funktion zugewiesen. Die Programmausführung fährt mit dem ursprünglichen Funktionsaufruf unmittelbar folgendem Kommando fort.

Als Bezeichner für die aufzurufende Funktion können nicht nur Bezeichner benutzerdefinierter und vordefinierter Funktionen sondern auch *Typbezeichner* dienen. Im Fall von Vektor-, Matrix- oder Arraytypen sind dies die speziellen Konstruktoren (siehe 3.3.4.3, 3.3.4.4 und 3.3.4.6). Im Fall von numerischen Typen wird eine explizite Typumwandlung (siehe 3.3.4.8) veranlasst.

3.3.5.3 Überladene Funktionen

Es können mehrere Funktionen mit identischem Bezeichner, aber unterschiedlicher Signatur, deklariert werden. Die Signatur einer Funktion wird aus Position und Typ jedes Parameters bestimmt. Werden zwei Funktionen mit identischem Bezeichner und Signatur deklariert ist das Programm ungültig.

Die aufzurufende Überladung einer Funktion wird über die Signatur bestimmt. Existiert eine Überladung, deren Signatur genau mit den übergebenen aktuellen Parametern übereinstimmt wird diese ausgewählt. Andernfalls werden aus den Überladungen mit gleicher Parameteranzahl, aber unterschiedlichen Typen „Kandidaten“ für den Aufruf ausgewählt. Eine Funktion ist „Kandidat“, wenn an jeder Stelle der Signatur der Typ identisch mit dem übergebenen Typ oder in diesen

verlustfrei umwandelbar ist. Gibt es genau einen Kandidaten wird dieser ausgewählt und die Programmausführung mit diesem fortgesetzt. Gibt es keinen oder mehrere Kandidaten ist das Programm ungültig.

3.3.6 Konstanten und Variablen

3.3.6.1 Variablendeklarationen

dekl_var:

typ(A.1.2) *dekl_var_bzch_init*(3.3.6.1) { ‘,’ *dekl_var_bzch_init*(3.3.6.1) }

dekl_var_bzch_init:

BEZEICHNER(3.3.8.1) [‘=’ *ausdruck*(3.3.3)]

dekl_var(3.3.6.1) deklariert eine oder mehrere neue Variablen des gegebenen Typs und den gegebenen Bezeichnern.

Die Sichtbarkeit der Bezeichner beginnt nach der jeweiligen Deklaration und reicht bis zum Ende des umgebenden Blockes.

Der Bezeichner darf keinen anderen Bezeichner des umgebenden Gültigkeitsbereiches überdecken.

Der Wert von einer deklarierten Variable ist anfänglich undefiniert.

Wird ein initialer Ausdruck angegeben, so entspricht dies semantisch einer Zuweisungsoperation hinter *dekl_var_bzch_init*(3.3.6.1), aber vor der Deklaration der nächsten Variable. Der Typ des initialen Ausdrucks muss zuweisungskompatibel zum Typ der Variable sein.

3.3.6.2 Konstantendeklarationen

dekl_konst:

‘`const`’ *typ*(A.1.2) *dekl_konst_bzch_init*(3.3.6.2) { ‘,’ *dekl_konst_bzch_init*(3.3.6.2) }

dekl_konst_bzch_init:

BEZEICHNER(3.3.8.1) ‘=’ *ausdruck*(3.3.3)

dekl_konst(3.3.6.2) deklariert eine oder mehrere neue Konstanten des gegebenen Typs und den gegebenen Bezeichnern.

Die Sichtbarkeit der Bezeichner beginnt nach der jeweiligen Deklaration und reicht bis zum Ende des umgebenden Blockes.

Der Bezeichner darf keinen anderen Bezeichner des umgebenden Gültigkeitsbereiches überdecken.

Die Konstante wird mit dem Wert des initialen Ausdrucks initialisiert. Der Typ des initialen Ausdrucks muss zuweisungskompatibel zum Typ der Konstante sein.

3.3.7 Ablaufsteuerung

3.3.7.1 Verzweigungen

verzweigung:

‘`if`’ ‘`(`’ *ausdruck*(3.3.3) ‘`)`’ ‘`{`’ *block*(3.3.2) ‘`}`’ [‘`else`’ ‘`{`’ *block*(3.3.2) ‘`}`’]

‚*ausdruck*‘ ist ein boolescher Ausdruck, die *Bedingung*.

Wenn die Bedingung zu `true` ausgewertet wird, werden die Kommandos des ersten Blockes (*Wahr-Block*) ausgeführt, ansonsten die Kommandos des Blockes im `else`-Zweig (*Falsch-Block*).

Der `else`-Zweig ist optional.

Im Gegensatz zu vielen C-artigen Sprachen erfordert die Syntax für *Wahr-Block* und *Falsch-Block* „echte“ Blöcke (und nicht, wie sonst üblich, eine einzelne Anweisung *kommando*(3.3.2) die eine Blockanweisung sein kann). Diese Anforderung wurde davon motiviert, eine Art Fehler zu vermeiden: die „Erweiterung“ eines aus einer einzelnen Anweisung bestehenden Verzweigungszweiges“ mit einer weiteren Anweisung, allerdings ohne die einzelne, ursprüngliche Anweisung in einen Block einzufassen. (Das Ergebnis dieses Fehlers kann eine nicht gewollte, unbedingte Ausführung einer Anweisung sein.)

3.3.7.2 `for`-Schleifen

schleife_for:

‘`for`’ ‘`(`’ *schleife_for_kopf*(3.3.7.2) ‘`)`’ ‘`{`’ *block*(3.3.2) ‘`}`’

schleife_for_kopf:

[*ausdruck*(3.3.3)] ‘`;`’ [*ausdruck*(3.3.3)] ‘`;`’ [*ausdruck*(3.3.3)]

Der erste Ausdruck ist die *Initialisierung*. Der zweite Ausdruck ist die *Schleifenbedingung*. Der dritte Ausdruck ist der *Zählausdruck*.

Bei der Ausführung wird zunächst die *Initialisierung* ausgeführt.

Die *Schleifenbedingung* wird ausgewertet. Ist das Ergebnis `true`, so werden die Kommandos des Schleifenblockes und anschließend der *Zählausdruck* ausgeführt. Dieser Ablauf wird wiederholt bis eine Auswertung der *Schleifenbedingung* das Ergebnis `false` hat.

Wie bei Verzweigungsblöcken muss auch der Schleifenblock ein „echter“ Block sein.

3.3.7.3 `while`-Schleifen

schleife_while:

‘`while`’ ‘`(`’ *ausdruck*(3.3.3) ‘`)`’ ‘`{`’ *block*(3.3.2) ‘`}`’

‚*ausdruck*‘ist ein boolescher Ausdruck, die *Bedingung*.

Die Bedingung wird ausgewertet. Ist das Ergebnis `true`, so werden die Kommandos des Schleifenblocks ausgeführt. Dieser Ablauf wird wiederholt bis eine Auswertung der Bedingung das Ergebnis `false` hat.

Wie bei Verzweigungsblöcken muss auch der Schleifenblock ein „echter“ Block sein.

3.3.8 Lexikalische Einheiten

Anmerkung: Da diese Regeln die lexikalischen Einheiten selbst beschreiben gibt es hier *keine* weiteren Trennzeichen zwischen Terminalen!

3.3.8.1 Bezeichner

Number:

Unicode-Ziffer

Letter:

Unicode-Buchstabe

Comp_XYZW:

(‘`x`’ | ‘`y`’ | ‘`z`’ | ‘`w`’)

Comp_RGBA:

(‘`r`’ | ‘`g`’ | ‘`b`’ | ‘`a`’)

SWIZZLE_XYZW:

Comp_XYZW(3.3.8.1) [*Comp_XYZW*(3.3.8.1) [*Comp_XYZW*(3.3.8.1) [*Comp_XYZW*(3.3.8.1)]]]

SWIZZLE_RGBA:

Comp_RGBA(3.3.8.1) [*Comp_RGBA*(3.3.8.1) [*Comp_RGBA*(3.3.8.1) [*Comp_RGBA*(3.3.8.1)]]]

BEZEICHNER:

(*Letter*(3.3.8.1) | '_') { (*Letter*(3.3.8.1) | *Number*(3.3.8.1) | '_') }

Whitespace:

Unicode-Separator

WS:

Whitespace(3.3.8.1) [*WS*(3.3.8.1)]

Bezeicher ergeben sich aus ein oder mehreren Buchstaben, Ziffern und dem Zeichen '_'. Das erste Zeichen darf keine Ziffer sein.

Der Unterteilung in Buchstaben und Ziffern liegen Unicode-Kategorien zugrunde (siehe [Uni09], Abschn. 4.5).

Zwei Bezeichner sind identisch wenn deren Zeichensequenzen *kanonisch äquivalent* nach Unicode sind (siehe [Uni09], Kap. 3, Def. D70)[2].

Lexikalische Einheiten können mit einem oder mehreren Zeichen der Unicode-Kategorie „Separator" getrennt werden.

3.3.8.2 Numerische Literale

Digit_Dec:

'0' .. '9'

Digits_Dec:

Digit_Dec(3.3.8.2) [*Digits_Dec*(3.3.8.2)]

Digits_Dec_Sign:

['–'] *Digits_Dec*(3.3.8.2)

[2] Verschiedene Glyphen können auf verschiedene Weisen in Unicode codiert werden: z.B. *Ä* mit einem einzelnen Zeichen (U+00C4 "Latin capital letter A with diaresis") oder als Kombination von zwei Zeichen (U+0041 "Latin capital letter A" und U+0308 "Combining diaresis"). Bei einem einfachen Stringvergleich würden diese beiden Darstellungen als verschieden angesehen – für einen menschlichen Leser wären sie aber semantisch äquivalent. Die Unicode-Regeln für kanonische Äquivalenz legen fest, wie solche Glyphen dargestellt werden sollen, damit aus semantisch äquivalente Zeichenketten auch „binär" gleiche Zeichenketten resultieren.

Digits_Dec_Frac:

‘.’ *Digits_Dec*(3.3.8.2)

Num_Float_Exp:

(‘e’ | ‘E’) *Digits_Dec_Sign*(3.3.8.2)

Num_Float_Mantissa:

(*Digits_Dec*(3.3.8.2) (‘.’ | *Digits_Dec_Frac*(3.3.8.2)) | *Digits_Dec_Frac*(3.3.8.2))

Num_Float:

Num_Float_Mantissa(3.3.8.2) [*Num_Float_Exp*(3.3.8.2)]

Digit_Hex:

(*Digit_Dec*(3.3.8.2) | ‘A’ .. ‘F’ | ‘a’ .. ‘f’)

Digits_Hex:

Digit_Hex(3.3.8.2) [*Digits_Hex*(3.3.8.2)]

Num_Hex:

‘0x’ *Digits_Hex*(3.3.8.2)

NUMERIC:

(*Num_Float*(3.3.8.2) | *Num_Hex*(3.3.8.2))

Ganzzahlen können in Dezimal- und Hexadezimalnotation angegeben werden.

Fließkommazahlen können als Dezimalbruch oder in Exponentialschreibweise angegeben werden.

3.3.8.3 Kommentare

COMMENT:

‘//’ beliebige Zeichen (‘\r’ | ‘\n’)

ML_COMMENT:

‘/*’ beliebige Zeichen ‘*/’

COMMENT(3.3.8.3) ist ein einzeiliger Kommentar. Alle Eingabezeichen bis zum nächsten Zeilenumbruch werden ignoriert.

ML_COMMENT(3.3.8.3) ist ein mehrzeiliger Kommentar. Alle Eingabezeichen, inklusive Zeilenumbrüchen, zwischen Start- (/*) und nächster Endmarkierung (*/) werden ignoriert.

3.4 Standardumgebung

3.4.1 Vordefinierte Funktionen

3.4.1.1 Skalarprodukt

```
<VektorN> dot (<VektorN> a, <VektorN b);
```

`dot` berechnet das Skalarprodukt zweier Vektoren. *<VektorN>* muss dabei ein Vektor eines numerischen Typs sein (siehe 3.3.4.3).

3.4.1.2 Vektorprodukt

```
float3 cross (float3 a, float3 b);
int3 cross (int3 a, int3 b);
int3 cross (unsigned int3 a, unsigned int3 b);
```

`cross` berechnet das Vektorprodukt zweier Dreikomponentenvektoren. (Das Vektorprodukt zweier Vektoren existiert nur für dreidimensionale Vektorräume[3].)

3.4.1.3 Matrixmultiplikation

```
<MatrixLxN> mul (<MatrixLxM> a, <MatrixMxN> b);
<VektorN> mul (<VektorM> a, <MatrixMxN> b);
<VektorL> mul (<MatrixLxM> a, <VektorM> b);
```

`mul` multipliziert zwei Matrizen miteinander. Der Basistyp aller übergebenen Matrizen muss dabei gleich sein. Nur numerische Basistypen sind erlaubt.

Als Sonderfall ist es möglich, einen Vektor an Stelle eines der Matrixparameter zu übergeben. Das Ergebnis ist äquivalent zu einer Matrixmultiplikation bei der aus dem Vektor eine einzeilige bzw. einspaltige Matrix erstellt wurde. Das Ergebnis ist eine einzeilige bzw. einspaltige Matrix, diese eine Zeile bzw. Spalte wird als Vektor zurückgegeben. Der Basistyp der übergebenen Matrix und des Vektors muss dabei gleich sein. Nur numerische Basistypen sind erlaubt.

3.4.1.4 Normalisierung

```
float<N> normalize (<VektorN> v);
```

[3] „[…] so ist das Vektorprodukt eine Spezialität des $\mathbb{R}^3$, für die es in anderen Dimensionen nichts Entsprechendes gibt.“ – aus [Koe85], Kap. 7, §1.1

`normalize` gibt den zu `v` gehörenden Einheitsvektor zurück. Das Ergebnis ist undefiniert wenn `v` die Länge 0 hat. *<VektorN>* muss dabei ein Vektor eines numerischen Typs sein.

3.4.1.5 Euklidische Länge

```
float length (<VektorN> v);
```

`length` berechnet für den Vektor `v` die euklidische Länge $|v| = \sqrt{\sum {v_c}^2}$ (c: Vektorkomponenten, also z.B. x, y, z bei $N = 3$).

<VektorN> muss dabei ein Vektor eines numerischen Typs sein.

3.4.1.6 Texturfunktionen

```
float4 tex1D (sampler1D tex, <Vektor1> coord);
float4 tex2D (sampler2D tex, <Vektor2> coord);
float4 tex3D (sampler3D tex, <Vektor3> coord);
float4 texCUBE (samplerCUBE tex, <Vektor3> coord);
```

Die Texturfunktionen `tex1D`, `tex2D` usw. veranlassen das Auslesen der `tex` entsprechenden Textureinheit mit den Koordinaten `coord`.

`coord` muss dabei ein Vektor eines numerischen Typs und der angegebenen Komponentenzahl sein.

Texturen verschiedener Dimensionalität müssen in Shadingprogrammen verschiedenartig angesprochen werden; dies bedingt die verschiedenen Texturfunktionen, die jeweils einen speziellen Texturtyp reflektieren.

3.4.1.7 Minimum

```
<Basistyp> min (<Basistyp> a, <Basistyp> b);
<VektorN> min (<VektorN> a, <VektorN> b);
```

`min` gibt das Minimum zweier numerischer Werte bzw. das komponentenweise Minimum zweier numerischer Vektoren zurück (analog der Funktion binärer Ausdrücke, siehe 3.3.3.3).

3.4.1.8 Maximum

```
<Basistyp> max (<Basistyp> a, <Basistyp> b);
<VektorN> max (<VektorN> a, <VektorN> b);
```

`max` gibt das Maximum zweier numerischer Werte bzw. das komponentenweise Maximum zweier numerischer Vektoren zurück (analog der Funktion binärer Ausdrücke, siehe 3.3.3.3).

3.4.1.9 Potenz

```
float pow (float basis, float exp);
```

`pow` gibt den für $basis^{exp}$ berechneten Wert zurück.

3.5 Zusammenfassung

Die spezifizierte Sprache ist vergleichsweise einfach, berücksichtigt jedoch in Syntax und vordefinierten Funktionen die Besonderheiten des Anwendungsbereiches „Shading“ und ist dafür praktisch einsetzbar.

Kapitel 4

Implementierung

In diesem Kapitel wird auf die Implementierung des Compilers selbst eingegangen. Neben Scanner und Parser werden vor allem die verwendete Zwischencoderepräsentation für die Weitergabe des Programms zwischen den verschiedenen Arbeitsschritten des Compilers sowie die Komponente zur Auftrennung eines Programms – das eigentliche „Ziel“ dieser Arbeit – beschrieben.

4.1 Compiler-Aufbau

Der Aufbau entspricht grösstenteils der Compiler-Standardarchitektur (Abbildung 4.1): das *Front-End* generiert nach Syntax- und Semantikanalyse eine Repräsentation des Programms in einem *Zwischencode*. Ein „*Middle-End*“ verarbeitet diese Zwischencoderepräsentation weiter, unter anderem werden Optimierungen vorgenommen. Im letzten Schritt wird im *Back-End* aus der optimierten Zwischencoderepräsentation der tatsächliche Zielcode generiert.

Besonderheit dieses Compilers ist der Schritt *Auftrennung VP/PP* des „Middle-Ends“. Hier wird in der Zwischencoderepräsentation untersucht, welche Operationen während der Vertexberechnung und welche während der Pixelberechnung ausgeführt werden müssen. Mit diesen Informationen kann das Programm entsprechend in ein Vertex- und ein Pixelprogramm aufgeteilt werden. Da zur Laufzeit auch ein „Übergeben“ von Ausgaben des Vertexprogramms an Eingaben des Pixelprogramms stattfindet (die implizite Interpolation von Ausgaben der Vertexberechnung), wird auch eine „Schnittstelle“ generiert, die die Ausgaben der Vertexberechnungen auf Eingaben der Pixelberechnungen abbildet.

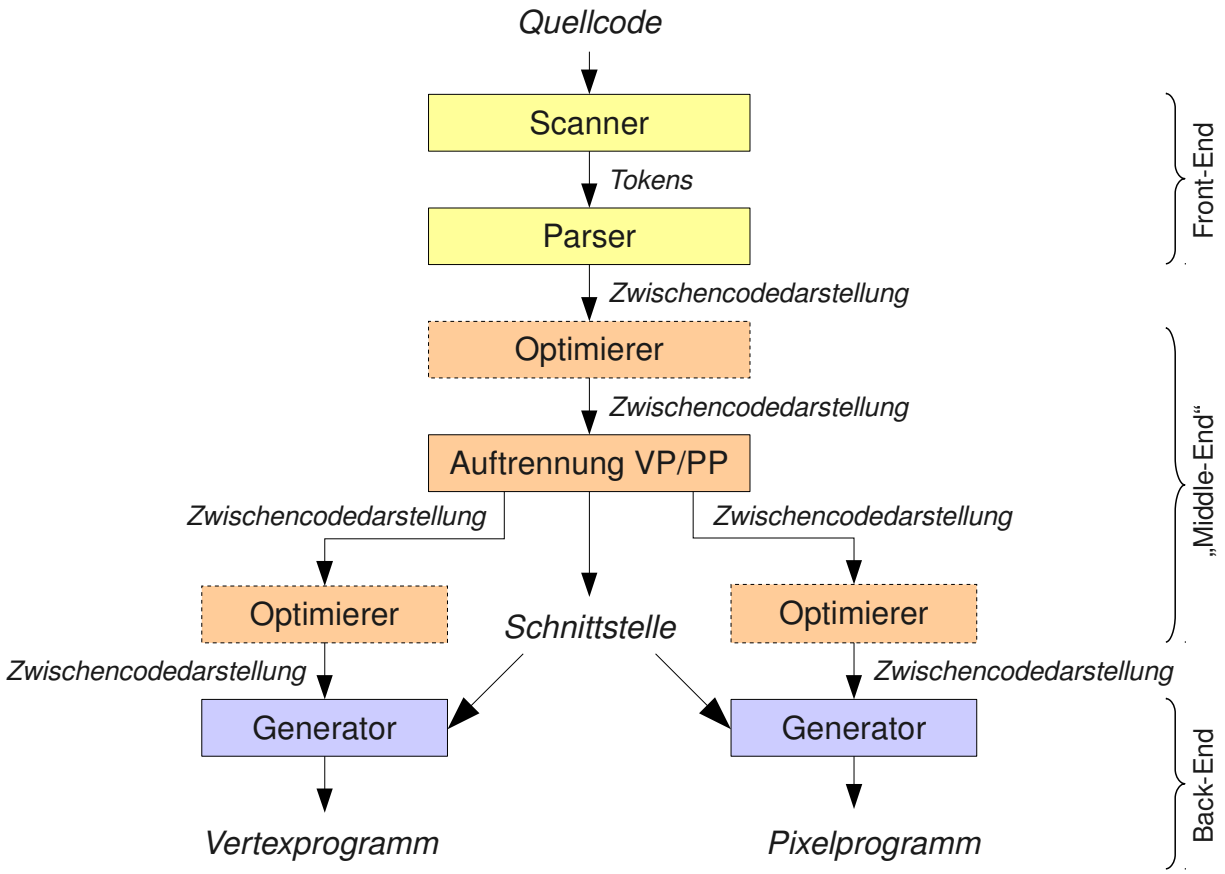

Abbildung 4.1: Schematischer Aufbau des Compilers

Die Programme werden vom Aufspalter in Zwischencode ausgegeben und noch einmal optimiert. Abschließend werden ein Vertex- und ein Pixelprogramm im gewünschten Zielcode ausgegeben[1].

4.2 Implementierungsdetails

Als *Programmiersprache*, in der die hier beschriebene Implementierung verfasst ist, wurde C++ gewählt. Gründe dafür sind:

- Die Flexibilität der Sprache und deren reichhaltige Standardbibliothek,
- hohe Portabilität (C++-Compiler sind für praktisch jede Plattform verfügbar),
- eine reichhaltige Palette an von Dritten hergestellter Bibliotheken,

[1] Diese Implementierung benutzt den gleichen Generator für beide Programme, prinzipiell könnten diese jedoch mit verschiedenen Generatoren ausgegeben werden.

- die einfache Verwendbarkeit in anderen Programmiersprachen (direkt oder über eine C-kompatible Schnittstelle).

Um eine Wiederverwendung des Compilers zu vereinfachen wurde dieser im Wesentlichen als eine *Bibliothek* realisiert; eine Kommandozeilenversion des Compilers setzt auch auf diese Bibliothek auf.

Zur Sicherstellung der fortwährend korrekten Funktionsweise aller Module des Compilers wurden entwicklungsbegleitend jeweils *Tests* der Module geschrieben (Black-Box und White-Box); Ausführen der Tests war regelmässiger Teil des Entwicklungsprozesses.

4.3 Scanner

Der *Scanner* wandelt die als Byte-Strom vorliegende Eingabe in eine Folge von „Tokens“. Ein Token ist eine der in Abschnitt 3.3.8 aufgezählten lexikalischen Einheiten, ein bekanntes Symbol (Operatoren etc.) oder Schlüsselwort. Leerzeichen (“Whitespace”) und Kommentare werden bereits vom Scanner ignoriert (d.h. für diese werden keine Tokens produziert).

Der Scanner folgt einer typischen Implementierung wie sie z.B. in [Wir05] beschrieben ist. Auf einige beachtenswerte Aspekte wird im folgenden eingegangen.

Eingabe. Da die Spezifikation von Unicode als Eingabe ausgeht, arbeitet der Scanner entsprechend auf der Basis von Unicode-kodierten Zeichen. Der Byte-Strom der Eingabe wird also in einem Schritt noch vor dem Scanner in einen „Unicode-Strom“ umkodiert[2].

Schlüsselwörter. Erkennt der Scanner einen Bezeichner, wird auch geprüft, ob es sich um ein Schlüsselwort handelt. Ist dies der Fall wird im Token eine dem Schlüsselwort eindeutig zugeordnete ID gespeichert.

Eine Ausnahme allerdings bilden die Schlüsselwörter für Vektor- und Matrixtypen (Abschnitte 3.3.4.3, 3.3.4.4). Diese entsprechen jeweils dem Muster *typ*N bzw. *typ*N*x*M (mit $N \in 1 \ldots 4, M \in 1 \ldots 4$). Da eine eigene ID für jeden Vektor- oder Matrix insgesamt 20 weitere IDs pro Basis-Typ nach sich ziehen würde – wobei später noch jeder ID wiederum die ursprünglichen Werte für N und M nochmals zugeordnet werden müssten

[2] Verwendet wird dazu die Bibliothek ICU, `http://site.icu-project.org/`

– wird im generierten Token vermerkt, ob es sich um einen Vektor- oder Matrixtyp handelt. Dazu überprüft der Scanner, ob ein Bezeichner den angegebenen Muster für Vektor- bzw. Matrixschlüsselwörtern entspricht. Weiterhin werden bereits N bzw. M aus den Bezeichnern extrahiert und ebenfalls vermerkt.

Abbildung 4.2 stellt die vom Scanner gesammelten Daten als Strukturdefinition dar.

```
// Token-Objekt
struct Token
{
  // Typ/Symbol/Schlüsselwort-ID dieses Tokens
  TokenType typeOrID;
  // Originale Zeichenkette des Tokens
  UnicodeString tokenString;
  // Art (Normal, Vektor, Matrix) von Typ-Schlüsselwörtern
  TypeClassification typeClass;
  // Für Vektoren: Komponentanzahl; Für Matrizen: Spaltenzahl
  int dimension1;
  // Für Matrizen: Anzahl Zeilen
  int dimension2;
};
```

Abbildung 4.2: Daten eines vom Scanner ausgegebenen Tokens

Weiterhin schreibt die Spezifikation vor, dass zwei Bezeicher als identisch betrachtet werden, wenn sie kanonisch äquivalent im Sinne von Unicode sind. Der Scanner speichert zu diesem Zweck alle Bezeichner in einer normalisierten Form. Diese ist vom Unicode-Standard vorgegeben: [Uni09], Annex #15.

4.4 Parser

Der *Parser* untersucht den vom Scanner gelieferten Strom von Tokens auf syntaktische Strukturen. Es wird überprüft, ob der Token-Strom gültig im Sinne der in der Sprachspezifikation gegebenen Sprache ist – sonst liegt ein *Syntaxfehler* vor.

Während der Überprüfung werden auch syntaktische Elemente – grundsätzlich Terminale wie Bezeichner oder numerische Werte – extrahiert. Diese werden bei der *semantischen* Verarbeitung benötigt.

4.4.1 Aufbau des Parsers

Der Parser ist nach der Methode des rekursiven Abstiegs (beschrieben in [Wir05]) handprogrammiert. Der Aufbau spiegelt im Wesentlichen die Struktur der Regeln wieder – viele haben ein direktes Gegenstück in einer Methode des Parsers.

Umgang mit Mehrdeutigkeiten: An einigen Stellen der Grammatik gibt es Mehrdeutigkeiten. Werden beim Parsen eines ‚`programm_statements`' (3.3.1) die Tokens ‚`typ BEZEICHNER`' (A.1.2, 3.3.8.1) erkannt, so kann es sich entweder um die Regel ‚`funktion_definition`' (3.3.5.1) oder um ‚`dekl_var`' (3.3.6.1) handeln. Andere Fälle von Mehrdeutigkeiten sind ‚`dekl_var`' (3.3.6.1) oder ‚`kommando`' (3.3.2) in ‚`block`' (3.3.2) und ‚`funktion_aufruf`' (3.3.5.2) oder ‚`BEZEICHNER`' (3.3.8.1) in ‚`asdr_basis`' (3.3.3.1).

Solche Mehrdeutigkeiten lassen sich entweder in der Implementierung des Parsers oder durch Abändern der Grammatik lösen.

Bei der Parser-Lösung werden einfach weitere Tokens betrachtet. Im Falle eines ‚`programm_statements`' (3.3.1) wird auch das nächste Token nach ‚`typ`' und ‚`BEZEICHNER`' überprüft: handelt es sich um ‚`(`', ist die anzuwendende Regel ‚`funktion_definition`' (3.3.5.1); handelt es sich um ‚`=`', ‚`,`' oder ‚`;`' ist die anzuwendende Regel ‚`dekl_var`' (3.3.6.1); andere Tokens sind ein Syntaxfehler. Eine Pseudo-Code-Version der Implementierung ist in Abbildung 4.3 aufgelistet.

In den Implementierungen der Regeln ‚`block`' (3.3.2) und ‚`asdr_basis`' (3.3.3.1) wurde analog verfahren.

Bei der Lösung von Mehrdeutigkeiten durch Abändern der Grammatik muss eine Regel erstellt werden, die mit der mehrdeutigen Token-Sequenz beginnt. Dahinter werden als Alternativen neue Regeln angefügt, die aus den „Resttokens“ der ursprünglich mehrdeutigen Regeln bestehen müssen.

Allerdings würde damit die Lesbarkeit der Grammatik eingeschränkt. Für die Regel ‚`ausdruck`' (3.3.3) wurde bei der Implementierung dieser Ansatz verfolgt, die Spezifikation der Grammatik in Kapitel 3 dahingehend aber *nicht* geändert. Stattdessen sind die in der Implementierung des Parser verwendeten „Variationen“ der Grammatik in Anhang A.1 beschrieben.

Für die anderen Regeln wurde das „Vorausschauen“ von Tokens gewählt, da es in diesen Fällen einfach zu implementieren war und die Grammatik besser lesbar bleibt.

Semantische Verarbeitung: Die semantische Verarbeitung wird von einer weiteren Komponente – hier “semantic handler” genannt – vorgenommen. Dieses Komponente übernimmt die verschiedenen Aspekte der semantischen Verarbeitung, von der Verwaltung der Symboltabelle bis zu einer geeigneten internen Repräsentation von Ausdrücken. Die in der Komponente gespeicherten Informationen sind in einem

```
Lexer.Token currentToken;

void ParseProgramStatements (const Scope scope)
{
  while (true)
  {
    int beyondType;
    // Prüfen, ob Token ein Typ (vordefiniert oder Typ-Alias) ist
    bool isType = IsType (scope, beyondType);
    // 'const': Eindeutig Konstantendeklaration
    if (currentToken.typeOrID == Lexer.KeywordConst)
    {
      ParseConstDeclare (scope);
    }
    /* Ist das Token ein Typ und das folgende Token
       (durch 'Peek()' vorausgesehen) ein Bezeichner, so kann
       eine Funktionsdeklaration _oder_ eine Variablendeklaration
       vorliegen.
       (Ein Token 'void' wäre eindeutig eine Funktionsdeklaration.)
     */
    else if ((isType
        && (Peek (beyondType).typeOrID == Lexer.Identifier))
      || (currentToken.typeOrID == Lexer.KeywordVoid))
    {
      /* Folgt dem Bezeichner ein '(', so liegt eine
         Funktionsdeklaration vor */
      if ((currentToken.typeOrID == Lexer.KeywordVoid)
          || (Peek (beyondType+1).typeOrID == Lexer.ParenL))
      {
        ParseFuncDeclare (scope);
      }
      /* Ansonsten: Variablendeklaration */
      else
      {
        ParseVarDeclare (scope); /* Enthält Test, ob ein Token '=',
                                    ',' oder ';' folgt */
      }
    }
    // 'typedef': Eindeutig Typdefinition
    else if (currentToken.typeOrID == Lexer.Keyword.Typedef)
    {
      ParseTypedef (scope);
    }
    else
      /* Unbekanntes Token - aufrufende Methode wird Fehler
         auslösen (erwartet "end of file"-Token) */
      break;
  }
}
```

Abbildung 4.3: Beispiel einer Parsing-Methode mit Auflösung von Mehrdeutigkeiten

Rückkanal dem Parser zugänglich; so werden diese zum Beispiel benutzt um festzustellen, ob ein Bezeichner ein Typ-Alias oder eine Funktion identifiziert. Die Ausgabe des “semantic handlers” ist eine Zwischencoderepräsentation des Programms.

Verglichen mit dem „klassischen“ Ansatz der semantischen Verarbeitung – Parser erzeugt Abstract Syntax Tree (AST) als ersten Schritt, semantische Analyse erzeugt

Zwischencoderepräsentation im zweiten Schritt – finden sich kleine Teile der semantischen Analyse im Parser; die Erstellung eines ASTs wird übergangen, der “semantic handler” nimmt mit den vom Parser übergebenen Informationen die restliche semantische Analyse vor und erzeugt sofort eine Zwischencoderepräsentation (beschrieben in Abschnitt 4.5).

Die Erstellung eines AST wurde übergangen, da diese als unnötig angesehen wurde: eine direkte Ausgabe der Zwischencoderepräsentation wurde als einfacher umzusetzen und für die weiteren Arbeitsschritte im Compiler als ausreichend angesehen. Insbesondere Optimierungen sind in der gewählten Zwischencoderepräsentation (ZCR) einfacher vorzunehmen als auf einem AST. Auch werden in der ZCR wichtige semantische Informationen, wie Typen von Werten, Funktionssignaturen etc. gespeichert, es tritt also kein Informationsverlust im Vergleich zu einem AST auf.

Weiterhin wurde die Implementierung so gestaltet, dass die Komponent des “semantic handler” relativ einfach austauschbar ist: aus „Parser-Sicht“ ist die interne Arbeitsweise und Art der Ausgabe des “semantic handler” irrelevant. Sollte also notwendig werden, dass der Compiler einen AST des Programms erstellen soll, so wäre es prinzipiell möglich, einen “semantic handler” zu schreiben der genau dies tut.

Fehlerbehandlung: In der Implementierung wird die Fehlerbehandlung bei der syntaktischen wie auch semantischen Verarbeitung über *Ausnahmen* realisiert. Die Parser-Komponente selbst fängt dabei Ausnahmen ab, um zu gewährleisten, dass möglichst viel eines Programms verarbeitet wird, um möglichst viele potentielle Fehler aufzudecken (wie in [Wir05] empfohlen): tritt z.B. eine Ausnahme während der Verarbeitung eines Block-Kommandos auf, setzt der Parser die Verarbeitung nach dem nächsten Semikolon – also mit dem nächsten Kommando – fort (sofern kein Ende des Blockes festgestellt wird). Die abgefangenen Ausnahmen werden jedoch nicht verworfen, sondern an ein Objekt zur Fehlerbehandlung übergeben um eine Nachricht für den Benutzer darzustellen.

4.5 Zwischencoderepräsentation

4.5.1 Vorlagen der Zwischencoderepräsentation

Als Vorlagen für die hier vorgestellte Zwischencoderepräsentation dienten das “LLVM Instruction Set”, SafeTSA und “SIMPLE” des McCAT Compiler-Projektes.

LLVM [LA04]: LLVM ist ein *Framework* für Compiler. Insbesondere will es ermöglichen, Optimierungen eines Programms über dessen ganzen „Lebenszyklus“ (inklusive Link- und Laufzeit) zu ermöglichen.

Das *LLVM Instruction Set* ist der ausgegebene „Objektcode“. Das Programm wird – ähnlich Maschinen- oder Bytecode – als eine Folge von einfachen Instruktionen auf Registern repräsentiert. Allerdings gibt es eine unbeschränkte Anzahl von typisierten Registern. Typumwandlungen sind immer explizit. Die Instruktionen sind in SSA-Form.

SafeTSA [Amm03]: Eine Art Objektcode, hauptsächlich zur Benutzung als „mobiler“ Code, d.h. zur Übertragung von Programmcode über Netzwerke wie das Internet. Das Design von SafeTSA ist inhärent sicher. Bösartige Manipulationen von Programmen, die zu problematischem Verhalten wie die Benutzung von undefinierten Werten oder Aliasing von Werten eines anderen Typs führen, sind nicht möglich bzw. durch eine einfache Verifizierung feststellbar.

Die Instruktionen basieren auf der SSA-Form. Entsprechend gibt es eine beliebige Zahl von Registern. Register sind in mehrere Sätze organisiert, ein Satz pro Typ. Instruktionen können nur auf einen spezifischen Registersatz zugreifen. Verschiedene Instruktionsblöcke besitzen eigene Registersätze.

SIMPLE [HDE+93]: Als „echte“ Compiler-Zwischencoderepräsentation für einen C-Compiler entwickelt stellt sie Programme auf sehr hoher Ebene dar. Ausdrücke sind nicht in der SSA-Form, allerdings „vereinfacht“ auf zwei Operanden und einfache Strukturzugriffe. Symboltabelle und Typinformationen sind erhalten. Typumwandlung, und andere in C implizite Verhalten, müssen explizit ausgedrückt werden.

Zusammenfassung: Die gewählte Zwischencoderepräsentation ist größtenteils ein „Querschnitt“ aus den obigen Repräsentationen (allerdings auch mit eigenständig entwickelten Aspekten, wie die Behandlung von Arrays). Die meisten Eigenschaften teilt

die Zwischencoderepräsentation mit dem LLVM Instruction Set; chronologisch wurde dieses jedoch als letzte Repräsentation betrachtet. Aus SafeTSA und SIMPLE stammen deshalb grundsätzliche Aspekte der Zwischencoderepräsentation – einfache Statements, SSA-Form, separate Registersätze.

Anzumerken ist, dass die Shadingsprache keine zufälligen Speicherzugriffe oder Zeiger/Referenzen erlaubt. Im Umfang ist sie teilweise beschränkt – es gibt keine Strukturtypen –, besitzt aber als „Eigenheit“ Vektortypen. LLVM, SafeTSA und SIMPLE wurden für „Maschinen“ entwickelt, die die Verwendung von Zeigern erlauben. Entsprechend stellen sie Lösungen für Probleme, wie die Aliasing[3]-Analyse oder typsichere Speicherzugriffe, bereit, die mit der hier spezifizierten Shadingsprache nicht vorkommen. Die genannten Repräsentationen sind hier nicht mit ihren vollständigen Fähigkeiten in Hinsicht auf diese Aspekte beschrieben.

4.5.2 Aufbau

Bei der Gestaltung der *Zwischencoderepräsentation* sollten folgende Rahmenbedingungen erfüllt werden:

- *Eignung als Zwischenrepräsentation zwischen verschiedenen Arbeitsschritten des Compilers*. Um die Komplexität des Compilers und den damit verbundenen Implementierungsaufwand klein zu halten sollte *ein* Format zur Zwischenrepräsentation für den compilerinternen Austausch verwendet werden. (Andere Compiler verwenden verschiedene Formate zwischen Verarbeitungsschritten, siehe z.B. [HDE+93].)

- Dies schliesst auch *Eignung für Auftrennung* und *Eignung für Optimierungen* ein.
  Die Auftrennung sollte mit möglichst feiner „Granularität“ passieren - bei komplexen Ausdrücken soll es möglich sein, dass verschiedene Teilausdrücke in verschiedene Teilprogramme ausgegeben werden können.
  Es sollten auch genug Information enthalten sein, um verschiedene Optimierungsalgorithmen umzusetzen. Idealerweise sollte die Zwischencoderepräsentation sowohl die Umsetzung der Aufspaltung als auch der Optimierungen möglichst unterstützen und vereinfachen.

[3] Zugriff auf ein Datum über mehrere verschiedene Zeiger

- Dafür sollte der *Aufbau* möglichst *einfach* sein, damit Programme in der Zwischencoderepräsentation unkompliziert traversiert und manipuliert werden können.

Die Optimierbarkeit wird unterstützt, in dem der Zwischencoderepräsentation die *"Single Static Assignment"-Form* (SSA, siehe [AWZ88] und [RWZ88]) für Ausdrücke zugrunde liegt. Der gewünschte einfache Aufbau äußert sich darin, dass Befehle eines Programmes in einer einfachen Reihung gespeichert werden – ohne Sprungmarken oder ähnliches. Verzweigungen und Schleifen werden durch „komplexe" Befehle realisiert (die intern wiederum Reihungen von Befehlen enthalten).

Eine Auftrennung auf der Ebene der Befehle der Zwischencoderepräsentation ermöglicht es, bei komplexen Ausdrücke u.ä. für einzelne Rechenschritte zu entscheiden, in welches Teilprogramm die Ausgabe erfolgen soll. Damit wurde die gewünschte „Granularität" – Aufteilung von Teilausdrücken – erreicht.

Das „Grundelement" der Zwischencoderepräsentation ist eine „Sequenz". Eine Sequenz besteht aus „Operationen", die auf „Registern" arbeiten. Register sind *typisiert*, es gibt getrennte Registersätze, ein Satz für jeden verwendeten Typ. Der Aufbau einer Sequenz ist in Abbildung 4.4 schematisch dargestellt.

Die in Operationen gespeicherte Registeridentifikation verweist auf den zugehörigen Registersatz, Typinformationen bleiben also erhalten. Verwendet werden diese Informationen bei der Aufspaltung[4], Optimierung[5] und Codegenerierung[6], und ein Abspeichern der Registertypen „entlastet" diese Komponenten von der Bestimmung des Typs eines Registerinhalts.

Der wesentliche, von der SSA-Form entliehene, Aspekt ist, dass ein Register nur von *einer* Operation der Sequenz beschrieben werden darf (im Weiteren „Register-Zuweisungs-Bedingung" genannt).

Im Gegensatz zu einer „reinen" SSA-Form gibt es jedoch keine ϕ-Operation. Stattdessen werden ϕ-Operation bereits „aufgelöst" gespeichert: soll z.B. bei einer Verzweigung eine Variable in einem Zweig verändert werden, so werden in beiden Zweigen Zuweisungen zum entsprechenden Register generiert. Ein Zweig enthält die Zuweisung des neuen Wertes, der andere die Zuweisung des alten Wertes.

[4] Die „Aufteilbarkeit" einer Operation hängt teilweise von den Datentypen der Operanden ab, siehe 4.6.6

[5] Speziell bei der Konstantenfaltung.

[6] Insbesondere da die gewählte Zielsprache – Cg – selbst eine Hochsprache ist.

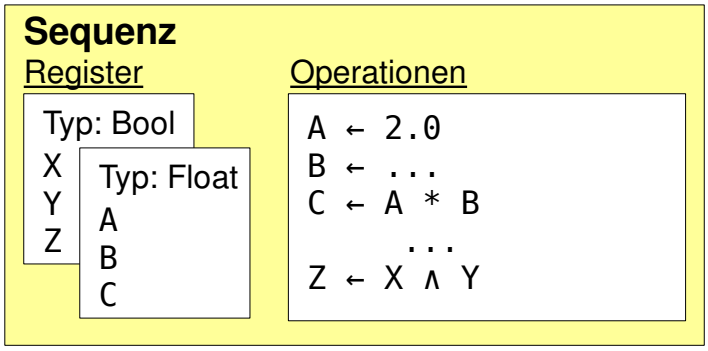

Abbildung 4.4: Teile einer Sequenz

Gewählt wurde dieser Ansatz, um Transformationen von Programmen, insbesondere das Entfernen einzelner Operationen, zu vereinfachen: Das spätere „Aufspalten“ eines Programms ist praktisch eine Erzeugung von mehreren Ausgabeprogrammen aus einem Eingabeprogramm, wobei Operationen des Eingabeprogramms teils kopiert, teils ausgelassen werden. Durch ein Auflösen der ϕ-Operationen können komplexe Operationen (Sequenzschachtelungen, Verzweigungen, Schleifen) trivial aus einem Ausgabeprogramm ausgelassen werden, ohne dass später folgende Operationen betrachtet werden müssten – bei expliziten ϕ-Operationen müsste zumindest für diese noch überprüft werden, ob sie noch gültig sind (d.h. die Verzweigung o.ä., auf die sich eine ϕ-Operation bezieht, existiert noch im Ausgabeprogramm).

Anzumerken ist, dass Sequenzschachtelungen, Verzweigungen und Schleifen in der Sequenz als *eine* Operation gespeichert werden (welche auf die inneren Sequenzen selbst nur Verweisen). Damit verletzt dieses Auflösen nicht die oben gegebenen „Register-Zuweisungs-Bedingung“. (Bei Verzweigungen wird einem Register, auch wenn ihm in beiden Zweigsequenzen ein Wert zugewiesen wird, zur Laufzeit nur einmal ein Wert zugewiesen, da nur eine Zweigsequenz ausgeführt wird. Wertzuweisungen in Schleifenblöcken kann man als Zuweisungen an „temporäre“ Register betrachten, wobei die Zuweisung an das tatsächliche Zielregister erst einmalig nach dem letzten Schleifendurchlauf stattfindet.)

Die Sichtbarkeit von Registern ist auf die Sequenz, in der sie deklariert wurden, beschränkt. Insbesondere gibt es Sequenzoperationen, die andere Sequenzen einschachteln (Verzweigungen, Schleifen, Sequenzschachtelung). Aus solch eingeschachtelten Blöcken kann *nicht* implizit auf die Register aus dem umgebenden Block zugegriffen werden.

Stattdessen werden zu eingeschachtelten Blöcken eine Zuordnung zwischen

Registern aus dem umgebenden Block („extern“) und Registern des eingeschachtelten Blockes („lokal“) gespeichert. Wie genau die Abbildung von Werten von „externen“ an „lokale“ Register stattfindet ist ein Implementierungsdetail, dass dem Generator obliegt; das Verhalten muss einem „umleiten“ von Lese- oder Schreibzugriffen von den angegebenen lokalen auf die zugeordneten externen Registern entsprechen.

Damit wird sichergestellt, dass für eine Sequenz alle Informationen zu von den Operationen verwendeten Registern vorhanden sind. Verarbeitungsschritte, die die Verwendung oder Inhalte von Registern betrachten können, wegen garantierter „lokaler“ Definition von Registern, Gültigkeitsbereiche u.ä. ignorieren; Sequenzen können unabhängig voneinander betrachtet und bearbeitet werden, da ein Verwalten von „umgebenden“ Sequenzen, um möglicherweise Informationen von dort verwendeten Registern zu erhalten, nicht nötig ist.

Abbildung 4.5 zeigt eine Sequenz, in der die zweite Operation eine Sequenzschachtelung ist. Der Sequenzoperation sind neben einem Verweis auf die auszuführende Sequenz auch Registerzuordnungen von Registern der äusseren (einschachtelnden) zu Registern der inneren (eingeschachtelten) Sequenz.

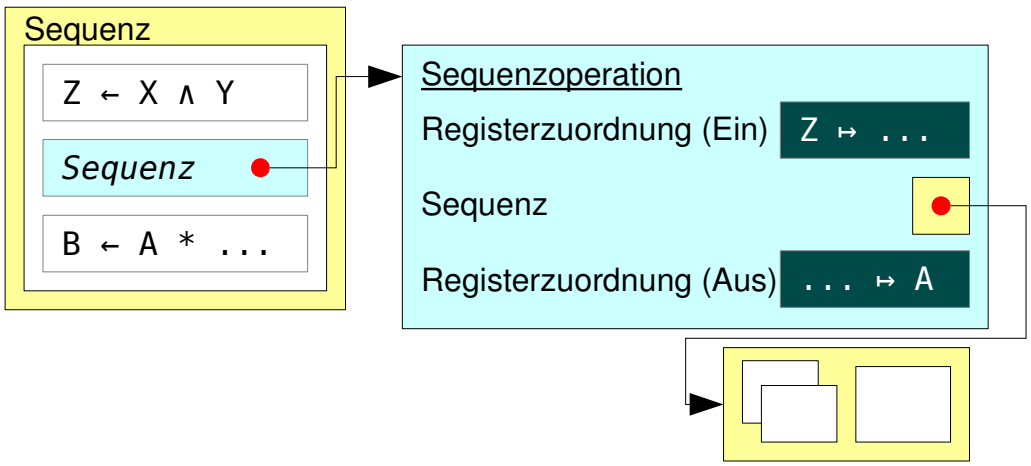

Abbildung 4.5: Schema einer Sequenzschachtelung

Funktionen: Eine Funktion der Zwischencoderepräsentation besteht aus einem eindeutigem Bezeichner, einer Liste von Eingabeparametern, einer Liste von Ausgabeparametern und einer Sequenz mit den eigentlichen Funktionsoperationen. Ein eventueller Rückgabewert ist nur ein weiterer Ausgabeparameter. Abbildung 4.6 ist eine schematische Abbildung einer Funktion.

Bei überladenen Funktionen muss die auszuführende Variante der Funktion in der Zwischencoderepräsentation explizit angegeben werden. Aus diesem Grund wird in der Zwischencoderepräsentation jede Überladung einer Funktion durch einen eindeutigen Bezeichner identifiziert.

Parameter werden mit einem Mechanismus, der der Behandlung „externer“ Register in einem eingeschachteltem Block ähnelt, übergeben. Die Parameterlisten enthalten zu jedem Eingabeparameter ein lokales Register, in dem die Sequenz den Wert des Parameters „erwartet“. Analog wird jedem Ausgabeparameter ein Register zugeordnet, in dem bei Verlassen der Funktion der zurückzugebende Wert liegt. (Die genaue Umsetzung dieses Verhaltens ist ein Implementierungsdetail, dass dem Generator obliegt.)

Ein Parameter, der gleichzeitig Ein- wie auch Ausgabeparameter ist, wird „verdoppelt“, d.h. es wird daraus ein nur-Eingabe- sowie auch ein nur-Ausgabe-Parameter generiert. Bei Funktionsaufrufen werden den beiden Parametern auch entsprechend verschiedene Register zugeordnet.

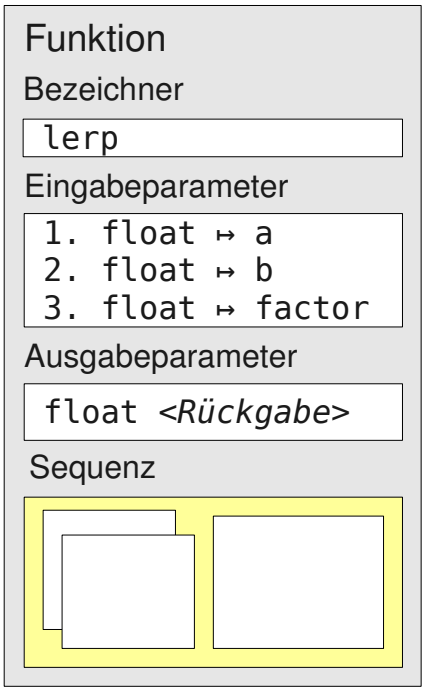

Abbildung 4.6: Schema einer Funktionsbeschreibung
für eine Funktion deklariert mit `float lerp (float a, float b, float factor)`.

Globale Variablen: Echte globale Variablen sind nicht vorgesehen. Sie werden nachgebildet, in dem in einer Funktion gelesene globale Variablen auf spezielle, „versteckte“ Eingabeparameter abgebildet werden. Geschriebene globale Variablen werden auf spezielle Ausgabeparameter abgebildet. Nur in der Eintrittsfunktion werden globalen Variablen tatsächlich „eigene“ Register zugewiesen. Im Prinzip sind „globale“ Variablen „versteckte“ lokale Variablen in der Eintrittsfunktion.

Damit müssen globale Variablen von Optimierungsschritten u.ä. nicht besonders berücksichtigt werden. Insbesondere müssen keine „Seiteneffekte“ von Funktionen ermittelt werden: manipuliert eine Funktion eine „globale“ Variable, so erhält sie in der Zwischencoderepräsentation einen weiteren Ausgabeparameter. Betrachtet man den Aufruf dieser Funktion so wird nur eine weitere Variable beschrieben.

Optimierungsschritte wie Konstantenfaltung oder die Entfernung unnötiger Operationen, aber auch der Auftrennungs-Schritt werden vereinfacht, da nur „lokale“ Variablen berücksichtigt werden müssen; gleichzeitig werden als global deklarierte Variablen von solchen Verarbeitungsschritten korrekt behandelt.

Behandlung von Arrays: Arrays werden als „ein“ Wert behandelt – eine Zuweisungsoperation kopiert immer ein ganzes Array. Das Lesen einzelner Elemente geschieht mit Hilfe der Operation „Extraktion eines Arrayelements“ (`getelem`).

Zum Schreiben eines Elements gibt es die Operation „Änderung eines Arrayelements“ (`setelem`): diese kopiert alle Elemente, bis auf das zu ändernde Element, eines Arrays in das Zielarray; dort wird am gegebenen Index der zu schreibende Wert abgelegt. Dieser Ansatz wurde gewählt, weil er sehr gut in das „SSA-Prinzip“ passt. Bei der direkten Umsetzung eines Zugriffs auf Array-Elemente (Ausdrücke wie eine Zuweisung „$a[i] = x$“) ist es schwierig, sicherzustellen, dass jedes Element von a wie verlangt nur einmal beschrieben wird (insbesondere bei Schleifen); es müsste für jedes Array-Element individuell „verfolgt“ werden, ob es beschrieben wurde. Ein solches Verfolgen wird weiterhin schwieriger, sobald die Array-Größe nicht bekannt ist – die spezifizierte Sprache sieht dies vor. Das betrachten eines Arrays als „einen“ Wert macht es hingegen einfach, die Bedingung „nur eine Zuweisung“ einzuhalten und zu überprüfen.

Für manche Optimierungen ist es trotzdem von Vorteil, die einzelnen Elemente eines Arrays zu verfolgen – sind diese z.B. Konstanten können die Werte an einer Konstantenfaltung teilnehmen. Solche Möglichkeiten der Optimierungen bleiben

bestehen: sind Größe und Elementwerte eines Arrays bekannt, kann ein Optimierer diese wie individuelle Register durch die Arrayoperationen hindurch verfolgen, oder sogar ein Array auf individuelle Register „aufteilen“.

4.5.3 Sequenz-Operationen

Dieser Abschnitt zählt alle möglichen Operationen in einer Sequenz auf. Eine Operation greift für die Eingabe auf kein, ein oder mehrere *Quellregister* zu. Hat die Operation ein Ergebnis, wird dieses in ein *Zielregister* geschrieben.

Für jede Operation wird ein einfaches Beispielprogramm gegeben; diesem wird der generierte Zwischencode gegenüber gestellt.

4.5.3.1 Einfache Operationen

Bei den Textdarstellungen der einfachen Operationen ist das erste Argument das Zielregister der Operation, alle nachfolgenden Argumente sind die Eingaberegister.

Zuweisungsoperation: Kopiert Inhalt eines Registers in ein anderes.

Beispiel und Zwischencode:

```
void main (float y)
{
        float x;
        x = y;
}
```

```
Funktion :   main
Eingabeparameter :   float y → m_y_B0
Registersatz float:  m_y_B0 v_x
Operationen :   mov  v_x, m_y_B0
```

Konstantenoperation: Diese weist dem Zielregister eine Boolesche, Integer- (vorzeichenlos oder vorzeichenbehaftet) oder Fließkommakonstante zu.

Beispiel und Zwischencode:

```
void main ()
{
        float x;
        x = 1.0;
}
```

```
Funktion :   main
Registersatz float:  v_x
Operationen :   mov  v_x, 1
```

Typumwandlungsoperation: Liest das Eingaberegister, wandelt dessen Wert in den Ziel-Typ und schreibt den umgewandelten Wert in das Zielregister. Kann zwischen Integer- (vorzeichenlos oder vorzeichenbehaftet) und Fließkommawerten umwandeln.

Beispiel und Zwischencode:

```
void main (float y)
{
        int x;
        x = int (y);
}
```

Funktion : `main`
Eingabeparameter : `float` y → *m_y_B0*
Registersatz `float`: *m_y_B0*
Registersatz `int`: *v_x*
Operationen : `cast int` *v_x, m_y_B0*

Arithmetische Operation: Die Inhalte zweier Eingaberegister werden durch eine arithmetische Operation verknüpft und das Ergebnis in das Zielregister geschrieben. Die Eingaberegister und das Zielregister müssen vom gleichen Typ sein – Integer- (vorzeichenlos oder vorzeichenbehaftet) und Fließkommawerte.

Beispiel und Zwischencode:

```
void main (float x,
           float y)
{
  float a = x+y;
  float b = x-y;
  float c = x*y;
  float d = x/y;
  float e = x%y;
}
```

Funktion : `main`
Eingabeparameter : `float` x → *m_x_B0*
`float` y → *m_y_B0*
Registersatz `float`: *m_x_B0 m_y_B0 v_a v_b v_c v_d v_e*
Operationen : `add` *v_a, m_x_B0, m_y_B0*
`sub` *v_b, m_x_B0, m_y_B0*
`mul` *v_c, m_x_B0, m_y_B0*
`div` *v_d, m_x_B0, m_y_B0*
`mod` *v_e, m_x_B0, m_y_B0*

Vergleichsoperation: Die Inhalte zweier Eingaberegister werden miteinander verglichen (gleich, ungleich, größer, größer gleich, kleiner oder kleiner gleich) und das Ergebnis in das Zielregister geschrieben. Die Eingaberegister müssen vom gleichen Typ sein – Integer- (vorzeichenlos oder vorzeichenbehaftet) und Fließkommawerte. Das Zielregister muss vom Typ Boolean sein.

Beispiel und Zwischencode:

```
void main (float x,
           float y)
{
  bool a = x == y;
  bool b = x != y;
  bool c = x > y;
  bool d = x >= y;
  bool e = x < y;
  bool f = x <= y;
}
```

Funktion : `main`
Eingabeparameter : `float` x → *m_x_B0*
`float` y → *m_y_B0*
Registersatz `bool`: *v_a v_b v_c v_d v_e v_f*
Registersatz `float`: *m_x_B0 m_y_B0*
Operationen : `cmp eq` *v_a, m_x_B0, m_y_B0*
`cmp ne` *v_b, m_x_B0, m_y_B0*
`cmp ge` *v_c, m_x_B0, m_y_B0*
`cmp gt` *v_d, m_x_B0, m_y_B0*
`cmp lt` *v_e, m_x_B0, m_y_B0*
`cmp le` *v_f, m_x_B0, m_y_B0*

Logische Operation: Die Inhalte zweier Eingaberegister werden durch logisch UND oder logisch ODER verknüpft und das Ergebnis in das Zielregister geschrieben. Die Eingaberegister und das Zielregister müssen vom Typ Boolean sein.

Beispiel und Zwischencode:

```
void main (bool x,
           bool y)
{
  bool a = x && y;
  bool b = x || y;
}
```

```
Funktion :              main
Eingabeparameter :      bool x → m_x_B0
                        bool y → m_y_B0
Registersatz bool:      m_x_B0 m_y_B0 v_a v_b
Operationen :           and  v_a, m_x_B0, m_y_B0
                         or  v_b, m_x_B0, m_y_B0
```

Unäre Operation: Unäre Operationen sind Vorzeichenumkehrung, logisches NICHT und bitweise Invertierung.

Die Vorzeichenumkehrung kehrt das Vorzeichen des Wertes des Eingaberegisters um und schreibt das Ergebnis in das Zielregister. Die Eingabe- und das Zielregister müssen vom gleichen Typ sein – Integer- (vorzeichenlos oder vorzeichenbehaftet) und Fließkommawerte.

Logisches NICHT invertiert den Wert des Eingaberegisters und schreibt das Ergebnis in das Zielregister. Die Eingabe- und das Zielregister müssen vom gleichen Typ Boolean sein.

Bitweise Invertiertung wird auf das Eingaberegister angewendet und schreibt das Ergebnis in das Zielregister. Die Eingabe- und das Zielregister müssen von einem Integer-Typ (vorzeichenlos oder vorzeichenbehaftet) sein.

Beispiel und Zwischencode:

```
void main (bool x,
           float y,
           int z)
{
  bool a = !x;
  float b = -y;
  int c = ~z;
}
```

```
Funktion :              main
Eingabeparameter :      bool x → m_x_B0
                        float y → m_y_B0
                        int z → m_z_B0
Registersatz bool:      m_x_B0 v_a
Registersatz float:     m_y_B0 v_b
Registersatz int:       m_z_B0 v_c
Operationen :           not  v_a, m_x_B0
                        neg  v_b, m_y_B0
                        inv  v_c, m_z_B0
```

4.5.3.2 Vektor- und Matrix-Operationen

Vektor-Erstellung: Nimmt als Eingabe ein bis vier Register, je nach der Komponentenanzahl des Zielregisters. Die Eingaberegister müssen alle den Basistyp

des Zielregisters besitzen. Sie werden der Reihe nach den Vektorkomponenten im Zielregister zugeordnet.

Beispiel und Zwischencode:

```
void main (int a,
           int b)
{
  int2 x = int2 (a, b);
}
```

Funktion : `main`
Eingabeparameter : `int a` → *m_a_B0*
`int b` → *m_b_B0*
Registersatz `int`: *m_a_B0 m_b_B0*
Registersatz `int2`: *v_x*
Operationen : `makevec` *v_x*, *m_a_B0*, *m_b_B0*

Extraktion einer Vektorkomponente: Nimmt neben einem Eingaberegister auch eine Integer-Konstante *N* im Bereich 0...3 entgegen. Das Eingaberegister muss von einem Vektortyp sein. Das Zielregister muss vom Basistyp des Vektors sein. Aus dem Eingabevektor wird die Komponente Nummer *N* extrahiert und in das Zielregister geschrieben.

Beispiel und Zwischencode:

```
void main (int2 a)
{
  int x = a.y;
}
```

Funktion : `main`
Eingabeparameter : `int2 a` → *m_a_B0*
Registersatz `int`: *v_x*
Registersatz `int2`: *m_a_B0*
Operationen : `extract` *v_x*, *m_a_B0*, 1

Matrix-Erstellung: Nimmt als Eingabe ein bis sechzehn Register, je nach den Dimensionen des Zielregisters. Die Eingaberegister müssen alle den Basistyp des Zielregisters besitzen. Sie werden der Reihe nach den Elementen im Zielregister zugeordnet: zuerst das Element der ersten Spalte in der ersten Zeile, als nächstes das Element der zweiten Spalte in der ersten Zeile, usw.

Beispiel und Zwischencode:

```
void main ()
{
  float2x2 m =
    float2x2 (1.0, 0.0,
              0.0, 1.0);
}
```

```
Funktion :  main
Registersatz float: i_tmp0 i_tmp1 i_tmp2 i_tmp3
Registersatz
float2x2: v_m
Operationen :        mov  i_tmp0, 1
                     mov  i_tmp1, 0
                     mov  i_tmp2, 0
                     mov  i_tmp3, 1
                makematrix  v_m, i_tmp0, i_tmp1, i_tmp2, i_tmp3
```

4.5.3.3 Array-Operationen

Array-Erstellung: Nimmt als Eingabe eine variable Anzahl von Registern, die Anzahl der Eingaberegister bestimmt die Länge des Arrays. Die Eingaberegister müssen alle den Basistyp des Zielregisters besitzen. Sie werden der Reihe nach den Array-Elementen im Zielregister zugeordnet.

Beispiel und Zwischencode:

```
void main ()
{
  float[] a = float[](1.0, 2.0, 3.0);
}
```

```
Funktion :  main
Registersatz float: i_tmp1 i_tmp2 i_tmp3
Registersatz float[]: v_a
Operationen :       mov  i_tmp1, 1
                    mov  i_tmp2, 2
                    mov  i_tmp3, 3
                makearray  v_a, i_tmp1, i_tmp2, i_tmp3
```

Extraktion eines Arrayelements: Verwendet zwei Eingaberegister: ein Arrayregister sowie als Index ein Register mit einem vorzeichenlosen Integer. Aus dem Array wird das Element mit dem gegebenen Index extrahiert und in das Zielregister geschrieben. Das Zielregister muss den Basistyp des Array-Registers besitzen.

Beispiel und Zwischencode:

```
void main (float[] x)
{
  float a = x[0];
}
```

```
          Funktion :  main
    Eingabeparameter :  float[] x → m_x_B0
  Registersatz float:  v_a
Registersatz float[]:  m_x_B0
         Registersatz
    unsigned int:  x_tmp1
       Operationen :      mov  x_tmp1, 0
                       getelem  v_a, m_x_B0, x_tmp1
```

Änderung eines Arrayelements: Verwendet drei Eingaberegister: neben einem Arrayregister und dem Indexregister (vorzeichenloser Integer) weiterhin ein Register vom Basistyps des Arrays (der „neue Wert“).
Das Zielregister muss den gleichen Typ wie das Arrayregister besitzen.
Aus dem Eingabearray werden alle Elemente *außer* das Element des gegebenen Indexes in das Zielarray kopiert; dort wird stattdessen der „neue Wert“ abgelegt. Das Zielregister muss den Typ des Arrayregisters besitzen.

In Pseudocode ausgedrückt:

```
// Eingaben: Array, Index, NeuerWert
// Ausgabe: Zielarray
for element = 0 to Array.length-1
{
  if (element == Index)
    ZielArray[element] = NeuerWert;
  else
    ZielArray[element] = Array[element];
}
```

Beispiel und Zwischencode:

```
void main (float[] x)
{
  float[] a = x;
  a[0] = 1.2;
}
```

```
          Funktion :  main
    Eingabeparameter :  float[] x → m_x_B0
  Registersatz float:  i_tmp0
Registersatz float[]:  m_x_B0 v_a v_a.1
         Registersatz
    unsigned int:  x_tmp2
       Operationen :      mov  v_a, m_x_B0
                          mov  i_tmp0, 1.2
                          mov  x_tmp2, 0
                       setelem  v_a.1, v_a, x_tmp2, i_tmp0
```

Arraylänge: Nimmt im Eingaberegister ein Array entgegen. Schreibt in das Zielregister, welches vom Typ „vorzeichenloser Integer“ sein muss, die Anzahl der Elemente des Arrays.

Beispiel und Zwischencode:

```
void main (float[] x)
{
  unsigned int a = x.length;
}
```

Funktion :	`main`
Eingabeparameter :	`float[]` x → *m_x_B0*
Registersatz `float[]`:	*m_x_B0*
Registersatz `unsigned int`:	*v_a*
Operationen :	`arraylen` *v_a*, *m_x_B0*

4.5.3.4 Komplexe Operationen

Sequenzschachtelung: Eine Operation, die auf eine weitere, innere Sequenz verweist, welche beim Ausführen der Operation abgearbeitet wird. Neben dem Verweis auf eine Sequenz werden auch Listen von „importierten“ und „exportierten“ Namen zu der Operation gespeichert. Jedem Namen ist weiterhin ein Register in der inneren Sequenz zugeordnet.

Für den Zugriff auf Register der „umgebenden“ Sequenz wird von dieser beim Einfügen der Operation eine Zuordnung von „importierten“ und „exportierten“ Namen der inneren Sequenz zu Registern der umgebenden Sequenz vorgenommen.

Eine Sequenzschachtelung gilt als *eine* Operation in der umgebenden Sequenz. Solange die Zuordnung von „exportierten“ Namen der eingeschaltelten Sequenz zu Registern der umgebenden Sequenz korrekt ist (es wird kein Register verwendet, das bereits beschrieben wurde), bleibt die Register-Zuweisungs-Bedingung erfüllt.

Beispiel und Zwischencode:

```
void main ()
{
  float x = 1.0, y;
  {
    float z = 2.0;
    y = x + z;
  }
  y = y*0.5;
}
```

```
          Funktion :  main
Registersatz float:  i_tmp3 v_x v_y v_y.1
      Operationen :  mov  v_x, 1
                     ┌──────────────────────────────┐
                     │ Import :  v_x → m_x_B0       │
                     │     mov   v_z, 2             │
                     │     add   m_y_B0, m_x_B0, v_z│
                     │ Export :  v_y ← m_y_B0       │
                     └──────────────────────────────┘
                     mov  i_tmp3, 0.5
                     mul  v_y.1, v_y, i_tmp3
```

Verzweigung: Eine auf der Sequenzschachtelung basierende Operation. Eingaben sind zwei Sequenzen (eine “if”- und eine “else”-Sequenz) sowie ein Register vom Typ Boolean mit dem Wert der Bedingung. Ist dieser „wahr“, wird die “if”-Sequenz ausgeführt, sonst die “else”-Sequenz. Es müssen immer beide Sequenzen gegeben werden, Sequenzen können aber leer sein.

Ein beschriebenes „exportiertes“ Register muss in beiden Untersequenzen beschrieben werden: würde in einer Verzweigung in nur einer Sequenz ein Register beschrieben werden, würde bei Ausführung des „anderen“ Pfades entweder zur Laufzeit das Register einen undefinierten Wert besitzen, oder es müsste anderweitig überschrieben werden, was die Register-Zuweisungs-Bedingung verletzt. Um diese Probleme zu vermeiden wird verlangt, dass in der „anderen“ Sequenz eine Zuweisung (typischerweise vom „alten“ Wert der dem Register entsprechenden Variable) vorgenommen werden muss.

Da eine Verzweigung als *eine* Operation in der umgebenden Sequenz gilt, wird ein Register, auch wenn es in beiden Blöcken beschrieben wird, von der umgebenden Sequenz aus gesehen bloß von einer Operation beschrieben (eben der Verzweigung). Damit bleibt die Register-Zuweisungs-Bedingung erfüllt.

Abbildung 4.7 zeigt schematisch eine Verzweigungsoperation in einer Sequenz. Zuerst wird der Bedingungswert `Z` berechnet. Die Verzweigungsoperation selbst enthält dazu noch Verweise auf die “if”- und “else”-Sequenzen, zu denen es jeweils Zuordnungen von „importierten“ und „exportierten“ Namen der umgebenden Sequenz zu Registern der verwiesenen Sequenz gibt.

Beispiel und Zwischencode:

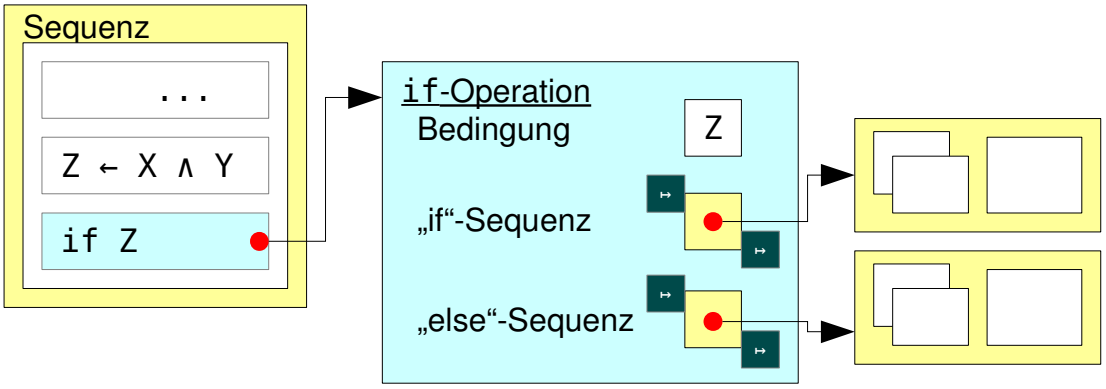

Abbildung 4.7: Schema einer Verzweigungsoperation.
symbolisiert die Registerzuordnungen aus Abb. 4.5.

```
void main (bool a)
{
  float x = 1.0;
  if (a)
  {
    x = 2.0;
  }
}
```

Funktion : `main`
Eingabeparameter : `bool a` → *m_a_B0*
Registersatz `bool`: *m_a_B0*
Registersatz `float`: *v_x v_x.1*
Operationen : `mov` *v_x*, 1
`if` *m_a_B0*:

`mov`	*m_x_B0*, 2
Export :	*v_x.1* ← *m_x_B0*

`else` :

Import :	*v_x* → *m_x_B0*
`mov`	*m_x_B0.1*, *m_x_B0*
Export :	*v_x.1* ← *m_x_B0.1*

While-Schleife: Eine auf der Sequenzschachtelung basierende Operation. Eingaben sind eine Sequenz sowie zwei Register vom Typ Boolean, jeweils mit einem Wert der Bedingung: das erste Register enthält die Bedingung *vor* der ersten Ausführung der Sequenz, das zweite Register die Bedingung *nach* einer Ausführung der Sequenz.

Werte, die sich von Schleifendurchlauf zu Schleifendurchlauf ändern, werden ähnlich behandelt: es muss jedem solchen Wert ein lokales Register im Schleifenkörper zugeordnet werden. Die Schleifenoperation selber erhält eine Abbildung von einem Paar „externer“ Register (Wert vor der ersten Ausführung sowie Wert nach einer Ausführung)

zu den lokalen Registern als weitere Eingabe. Durch diese Verwendung von Paaren von Eingaben ist es möglich, Werte in einem Schleifendurchlauf zu beschreiben und im nächsten Durchlauf wieder als Eingabe für eine Operation zu verwenden: bei ein Quellcode wie `int i = 0; while (...) { i = i + 1; }` müssten, ohne Registerpaare, bei `i = i + 1` entweder Eingabe-`i` wie auch Ergebnis-`i` auf das gleiche Register umgesetzt werden – welches entsprend mehrfach zugewiesen wird – oder aber verschiedene Register, wobei Eingabe-`i` immer den anfänglichen Wert `0` besitzen würde. Registerpaare erlauben es, dass im ersten Durchlauf der anfängliche Wert `0` für Eingabe-`i` verwendet wird, in anschliessenden Durchläufen aber der Wert des Ergebnis-`i` des letzten Durchlaufs.

Ein Register kann also in mehreren Schleifendurchläufen beschrieben werden. Da aber eine Schleife als *eine* Operation in der umgebenden Sequenz gilt, wird auch ein mehrmals beschriebenes Register, von der umgebenden Sequenz aus gesehen, bloß von einer Operation beschrieben (eben der Schleife). Damit bleibt die Register-Zuweisungs-Bedingung erfüllt.

Es gibt keine spezielle Operation für `for`-Schleifen, diese werden auf `while`-Schleifen abgebildet.

Beispiel und Zwischencode:

```
void main (unsigned int x)
{
  unsigned int i = 0;
  while (i < x)
  {
    i = i+1;
  }
}
```

Funktion : `main`
Eingabeparameter : `unsigned int x` → *m_x_B0*
Registersatz `bool`: *v_$cond1 v_$cond1.1*
Registersatz `unsigned int`: *m_x_B0 v_i v_i.1*
Operationen :
`mov` *v_i*, 0
`cmp lt` *v_$cond1*, *v_i*, *m_x_B0*
`while` *v_$cond1*/*v_$cond1.1*:

Import :	*v_i*/*v_i.1* → *m_i_B0*
Import :	*m_x_B0* → *m_x_B1*
`mov`	*i_tmp0*, 1
`add`	*m_i_B0.1*, *m_i_B0*, *i_tmp0*
`cmp lt`	*m_$cond1_B0*, *m_i_B0.1*, *m_x_B1*
Export :	*v_$cond1.1* ← *m_$cond1_B0*
Export :	*v_i.1* ← *m_i_B0.1*

4.5.3.5 Funktions-Operationen

Funktionsaufruf: Diese Operation nimmt einen Funktionsbezeichner, eine Liste Eingabe-Register und eine Liste Ausgabe-Register entgegen. Die Eingaberegister werden der Position nach auf die Funktionsparameter abgebildet.

Der „Bezeichner“ ist eine prinzipiell beliebig wählbare Zeichenkette, die die Funktion bezeichnet. Dem Code-Generator muss vorher eine Funktionsbeschreibung übergeben worden sein, die durch den Bezeichner identifiziert werden kann.

Beispiel siehe nächster Absatz.

Funktionsrücksprung: Damit wird die Funktion verlassen. Es wird eine Liste von Registern entgegen genommen. Diese werden, der Position nach, den Ausgabeparametern der Funktion zugewiesen.

Beispiel und Zwischencode:

```
float sqr (float x)
{
  return x*x;
}

void main (float a)
{
  float b = sqr (a);
}
```

```
Funktion :  sqr$1F
Eingabeparameter :  float x → m_x_B0
Ausgabeparameter :  float $retval ← v_$retval1
Registersatz float: m_x_B0 v_$retval1
Operationen :     mul  v_$retval1, m_x_B0, m_x_B0
                  return  v_$retval1

Funktion :  main
Eingabeparameter :  float a → m_a_B0
Registersatz float: m_a_B0 v_b
Operationen :    call sqr$1F, m_a_B0, v_b
```

Vordefinierte Funktion: Diese Operationen führen vordefinierte Funktionen (siehe 3.4.1) aus und arbeiten analog zu Funktionsaufrufen. Es wird ein Zielregister für den Rückgabewert und eine Liste Eingabe-Register entgegen genommen. Die Eingaberegister werden wieder der Position nach auf die Funktionsparameter abgebildet.

Beispiel und Zwischencode:

```
void main (sampler2D tex, float2 coord)
{
  float4 color = tex2D (tex, coord);
}
```

Funktion :	`main`
Eingabeparameter :	`sampler2D tex` → *m_tex_B0*
	`float2 coord` → *m_coord_B0*
Registersatz `float2`:	*m_coord_B0*
Registersatz `float4`:	*v_color*
Registersatz `sampler2D`:	*m_tex_B0*
Operationen :	`tex2D` *v_color*, *m_tex_B0*, *m_coord_B0*

4.5.4 Beispiel

Abbildung 4.8 zeigt ein einfaches Programm in der Sprache, Abbildung 4.9 die generierte Zwischencoderepräsentation.

```
void main (in float4 inPosition,
           in float4x4 ModelViewProj,
           out float4 outPosition, out float4 outColor)
{
        // Transformierte Koordinaten
        outPosition = mul (ModelViewProj, inPosition);

        // Ausgabe: rot
        outColor = float4 (1.0, 0.0, 0.0, 1.0);
}
```

Abbildung 4.8: Quelltext eines Programms.

Funktion :	`main`
Eingabeparameter :	`float4 inPosition` → *m_inPosition_B0*
	`float4x4 ModelViewProj` → *m_ModelViewProj_B0*
Ausgabeparameter :	`float4 outPosition` ← *m_outPosition_B0*
	`float4 outColor` ← *m_outColor_B0*
Registersatz `float`:	*i_tmp1 i_tmp2 i_tmp3 i_tmp4*
Registersatz `float4`:	*m_inPosition_B0 m_outColor_B0 m_outPosition_B0*
Registersatz `float4x4`:	*m_ModelViewProj_B0*
Operationen :	`mmul` *m_outPosition_B0*, *m_ModelViewProj_B0*, *m_inPosition_B0*
	`mov` *i_tmp1*, 1
	`mov` *i_tmp2*, 0
	`mov` *i_tmp3*, 0
	`mov` *i_tmp4*, 1
	`makevec` *m_outColor_B0*, *i_tmp1*, *i_tmp2*, *i_tmp3*, *i_tmp4*

Abbildung 4.9: Zu 4.8 generierte Zwischencoderepräsentation.

4.6 Auftrennung

Ziel des Compilers soll es sein, als Eingabe *ein* Shadingprogramm – mit einer Hauptfunktion usw., nicht nur „ein Quelltext“ – entgegenzunehmen und daraus Programme zu erzeugen, die für die programmierbaren Grafikchip-Einheiten zur Vertex- bzw. Pixelverarbeitung genutzt werden können.

Diese Aufgabe wird vom „*Splitter*“ wahrgenommen. Er trennt ein als Zwischencoderepräsentation vorliegendes Programm in zwei Ausgabeprogramme, ein *Vertex-* und ein *Pixelprogramm*, auf.

Es muss für jede individuelle Sequenzoperation entschieden werden, ob diese während der Pixelverarbeitung ausgeführt werden muss oder während der Vertexverarbeitung ausgeführt werden kann. Dabei ist zu berücksichtigen, dass einige Operationen nur in einem der Verarbeitungsschritte sinnvoll sind – z.B. sollte ein Auslesen eines Bildes zur Anwendung eines Oberflächenmusters sinnvollerweise in der Pixelverarbeitung vorgenommen werden.

Die Entscheidung, in welchem der beiden Ausgabeprogramm eine Sequenzoperationen ausgeführt werden soll, wird mit Hilfe von „Berechnungsfrequenzen“ (siehe unten), die sich jeder Operation zuordnen lassen, getroffen.

Da einige Operation in der Vertex-, andere in der Pixelverarbeitung ausgeführt werden sollen, müssen bei komplexeren Ausdrücken Zwischenergebnisse von der Vertex- zur Pixelverarbeitung übergeben werden. Es muss also die „Schnittstelle“ zwischen den Verarbeitungseinheiten generiert werden. Dabei ist zu beachten, dass nur Werte vom Vertex- zum Pixelprogramm übertragen werden können; eine Übertragung in die andere Richtung ist nicht möglich.

Auch muss die von der GPU vorgenommene implizite Interpolation von Ergebnissen der Vertexverarbeitung in Betracht gezogen werden. D.h. der Compiler muss die Operationen des Programms so aufteilen, dass die Interpolation bei der Übergabe von Werten von der Vertex- zur Pixelverarbeitung nicht die Berechnungen des Programms „verfälscht“. (Würde z.B. der Ausdruck x^2 in der Vertexberechnung berechnet und interpoliert in der Pixelverarbeitung verwendet werden, so käme dies einer lineare Approximation einer quadratischen Kurve gleich – der Wert des Ausdruck wäre ungenauer, also „verfälscht“.)

Dies wird durch eine Überprüfung auf „Interpolierbarkeit“ (siehe Abschnitt 4.6.6)

einer Operation erreicht.

4.6.1 Berechnungsfrequenzen

In Abschnitt 2.5 wurde aufgezählt, welche Arten von Daten bei der Darstellung eines 3D-Objektes auftreten. Dabei wurde zwischen Daten unterschieden, die zwischen verschiedenen Objekten variieren, Daten, die zwischen verschiedenen Vertices eines Objekt variieren, und Daten, die zwischen verschiedenen Pixeln eines gerasterten Dreiecks variieren. Auch wurde dargelegt, dass Berechnungen auf den verschiedenen Arten von Daten verschieden häufig sind – es treten in der Regel mehr Pixelberechnungen als Vertexberechnungen und mehr Vertexberechnungen als Objektberechnungen auf.

Aus diesen Beobachtungen lässt sich der Begriff der „Berechnungsfrequenz“[7] formulieren.

Definition 6. *Die **Berechnungsfrequenz eines Ausdrucks** beschreibt, wie oft sich der Wert des gegebenen Ausdrucks, auf alle Ausführungen eines Programmes gesehen, ändert.*

Der Term „Frequenz“ wird verwendet, da man die Anzahl der ausgeführten Berechnungen auf eine Zeiteinheit bezogen – hier die „Lebenszeit“ eines Programmes – betrachtet. Zwar sind sowohl die Berechnungsanzahl wie auch die „Lebenszeit“ abstrakte Größen, lassen trotzdem aber Vergleiche zu: zur Veranschaulichung sei die Zeiteinheiten das einmalige Darstellen eines 3D-Objektes. Die Frequenz einer in der Vertexberechnung ausgeführten Operation ist niedriger als die Frequenz einer in der Pixelberechnung ausgeführten Operation, da bei der Darstellung des Objekts in der Regel mehr Pixelberechnungen als Vertexberechnungen vorgenommen werden. Dieses Verhältnis bleibt auch erhalten, wenn man verschiedene Zeiteinheiten wählt – sei dies zwei 3D-Objekte, zehn 3D-Objekte oder N 3D-Objekte.

Die kleinstmögliche Frequenz besitzen also statische Konstanten.

Die höchstmögliche Frequenz besitzen Ausdrücke, die bei jeder Auswertung einen anderen Wert liefern. *(Beispiel: Werte aus einer externen Datenquelle wie einem Zufallsgenerator.)*

Für die Berechnungen der Echtzeit-3D-Grafik sind vor allem folgende Berechnungsfrequenzen bedeutend:

[7] Verwendet und benannt wurden Berechnungsfrequenzen zuerst in [PMTH01]

- *Objekt-Frequenz*, also Berechnungen auf Objektattributen (Daten, die für das Objekt vorliegen) bzw. davon abgeleiteten Werten.
- *Vertex-Frequenz*, also Berechnungen auf Werten von Vertexattributen (Daten, die von Vertex zu Vertex verschieden sind) bzw. davon abgeleiteten Werten.
- *Pixel-Frequenz*, also Berechnungen auf Eingaben der Pixelberechnung bzw. davon abgeleiteten Werten.

4.6.2 Formulierte Frequenz

Zwar soll die Berechnungsfrequenz von Operationen soweit möglich automatisch bestimmt werden, trotzdem muss eine Frequenz gewählt werden, in der alle Operationen vorerst formuliert werden.

Betrachtet man eine „niedrige“ Frequenz wie die Vertexfrequenz, so erkennt man, dass sich damit Berechnungen einer höheren Frequenz (also Pixel) schlecht darstellen lassen. Betrachtet man allerdings Berechnungen mit Pixelfrequenz, so lassen sich unter Umständen Berechnungen niedrigerer Frequenz ableiten. Shadingprogramme sollten also grundsätzlich in „Pixelfrequenz“ formuliert werden.

4.6.3 Bestimmung der Berechnungsfrequenz

Die Berechnungsfrequenz ist zwar eine Eigenschaft von Operationen, allerdings muss zu Registern, Zwischenwerten u.ä. gespeichert werden, mit welcher Frequenz die zuweisende Operation ausgeführt wurde, da diese „Zuweisungsfrequenz“ eines Wertes beeinflusst, mit welcher Frequenz eine Operation ausgeführt werden muss, die den Wert als Eingabe verwendet.

Register: Zu jedem Register wird gespeichert, welche Frequenz die Operation hatte, die den Registerwert bestimmt hat (d.h. in welches der Ausgabeprogramme diese Operation ausgegeben wurde).

Arrays: Wird auf Elemente eines Arrays nur mit statisch bekannten Indizes zugegriffen kann jedes Element als einzelnes Register gesehen werden (und entsprechend kann jedem Element zugeordnet werden, welche Frequenz die Operation hatte, die den Elementwert bestimmt hat). Da beim Zugriff mit nicht statisch bekannten Indizes

jedoch nicht bestimmt werden kann, welche Berechnungsfrequenz die Operation hatte, die einem Element einen Wert zugeweisen hat (da ja auch nicht bekannt ist, auf welche Elemente genau zugegriffen wird), so muss für alle Elemente die höchste, bei der Zuweisung irgendeines Elements möglicherweise verwendete Berechnungsfrequenz angenommen werden.

Ausdrücke: Die notwendige Berechnungsfrequenz für einen Ausdrucks hängt von der grössten gemeinsamen Frequenz, in der die verwendeten Operanden berechnete wurden, sowie der verwendeten Verknüpfung ab. Die Abschnitte 4.6.4 und 4.6.6 beschreiben wie die Berechnungsfrequenz für einen Ausdrucks bestimmt werden kann. Diese Berechnungsfrequenz eines Ausdrucks wird, wie oben genannt, zu den Ergebnisregistern (sofern vorhanden) der Operation gespeichert.

Programmeingaben: Bei Eingabeparametern aus der Umgebung kann eine Frequenz nicht automatisch abgeleitet werden. Es wird davon ausgegangen, dass die umgebende Anwendung dem Compiler mitteilt, mit welcher Frequenz ein Eingabeparameter geändert wird.

4.6.4 Berechnungsfrequenzen in der Aufspaltung

Für die Aufspaltung relevante Frequenzen sind die Objekt-, Vertex- und Pixelfrequenz.

Werte von Objektattributen, also Eingaben, die sich nicht während aufeinanderfolgenden Ausführungen eines Programmes ändern, können als *dynamische Konstanten* angesehen werden. Deren Werte stammen aus der umgebenden Anwendung.

Werte von Vertexattributen sind den verschiedenen Vertices zugeordnete Eingaben. Auch diese sind Anwendungsdaten.

Eingaben der Pixelberechnung sind die interpolierten Ausgaben des Vertexprogramms (siehe Abschnitt 4.6.5 unten). Insbesondere besitzen GPUs keinen Mechanismus, der der Anwendung erlauben würden, direkt Eingaben für jedes Pixel vorzugeben.

„Echte“, statische Konstanten werden vom Splitter als „Objekteingaben“ betrachtet.

Für jedes Register wird verfolgt in welchen Teilprogrammen es berechnet wurde. Dabei kann ein Register in mehreren Teilprogrammen verfügbar sein, wenn es im Vertexprogramm berechnet wurde, aber später zur Übertragung an das Pixelprogramm markiert wird (Abschnitt 4.6.5).

Die Teilprogramme, in denen ein Register vorliegt, beeinflussen, in welchen Teilprogrammen Operationen ausgeführt werden können, die das Register als Operand verwenden: Eine Operation muss mindestens in derjenigen Frequenz ausgeführt werden, welche dem Maximum der Frequenzen entspricht, in denen die Operanden ausgeführt wurden: eine Operation seltener auszuführen, als sich die Operanden ändern könnten, ist offensichtlich nicht sinnvoll. So muss z.B. eine Addition von zwei Werten in der Pixelberechnung ausgeführt werden, wenn einer der Summanden in der Pixelberechnung berechnet wurde.

Vorraussetzung für die Ausführung einer Operation in der Vertexberechnung ist damit, dass beide Operanden als Werte von Vertexattributeb vorliegen bzw. von solchen abgeleitet wurden. (Darüber hinaus muss die Operation das Kriterium der *Interpolierbarkeit*, beschrieben in Abschnitt 4.6.6), erfüllen).

4.6.4.1 Ausgegebene Programme

Der Splitter kategorisiert Operationen auch als „in der Objektberechnung ausgeführt“, generiert jedoch *kein* separates Programm für die Objektberechnung. Stattdessen werden Operationen der Objektberechnung sowohl vom Vertex- als auch vom Pixelprogramm ausgeführt. Zwei Annahmen liegen diesem Zugrunde: zuerst, dass nur relativ einfache Operationen in der Objektberechnung vorgenommen werden, es also kein Nachteil durch eine mehrfache Ausführung in Vertex- und Pixelprogramm entsteht. Die zweite Annahme ist, dass durch weitere Optimierungsschritte wie die „Entfernung unnötiger Operationen“ einige Operationen der Objektberechnung entfernt werden.

4.6.5 Schnittstelle Vertex-/Pixelprogramm

Da ein Teil der Operationen des ursprünglichen Programms dem Vertexprogramm zugeordnet wird, ein anderer Teil aber dem Pixelprogramm, müssen Ergebnisse des einen Programms zum anderen übertragen werden.

Stellt der Splitter die Notwendigkeit des Übertragens für einen Wert fest, wird das Register, welches den Wert enthält, aufgezeichnet. Diese Liste der zu übertragenden

Werte („Schnittstelle“ in Abbildung 4.1) wird später dem Codegenerator übergeben. Dieser kümmert sich um die „technischen“ Details wie die Zuordnung von für die Übertragung notwendigen Ressourcen und entsprechende Abbildung in der Zieldarstellung.

4.6.6 Interpolierbarkeit

Programme der Shadingsprache sind grundsätzlich so formuliert, als würden Operationen in der Pixelberechnung ausgeführt (siehe Abschnitt 4.6.2). Das vom Splitter zu lösende Problem ist damit die Bestimmung von Operationen, die in der Vertexberechnung ausgeführt werden können. Das Hauptkriterium diese Entscheidung ist die *Interpolierbarkeit* einer Operation.

4.6.6.1 Praktische Grundlage

Die praktische Grundlage der Interpolierbarkeit ist die in Abschnitt 2.4 beschriebene lineare Interpolation von Ausgaben von Vertexberechnungen, um die Interpolationsergebnisse als Eingabe der Pixelberechnungen zu nutzen.

Das Kriterium „Interpolierbarkeit“ beschreibt, welche während einer Pixelberechnung vorgenommenen Operationen auch während einer Vertexberechnung vorgenommen werden und anschliessend interpoliert werden können, ohne dass sich das Ergebnis ändert.

4.6.6.2 Ableitung und Definition

Vom Vertexprogramm ausgegebene Werte werden, bevor sie wieder dem Pixelprogramm als Eingabe dienen, *linear interpoliert*: Eine lineare Interpolation – die Funktion sei „lerp“ genannt – zwischen zwei Werten a und b mit Faktor f ($0 \leq f \leq 1$) wird durch $\text{lerp}(a, b, f) = a \cdot (1 - f) + b \cdot f$ berechnet. Der Faktor f „gewichtet“ zwischen den beiden Werten: der Faktor 0 gibt also den Wert a zurück ($\text{lerp}(a, b, 0) = a$), der Faktor 1 gibt den Wert b zurück ($\text{lerp}(a, b, 1) = b$).

Angenommen, ein Wert soll x von der Vertex- an die Pixelberechnung übergeben werden. Die Ergebnisse der Vertexberechnung sind verschiedene Werte für verschiedene Vertices, ein Wert x_1 für das erste Vertex, ein Wert x_2 für das zweite Vertex usw.

Die Eingabe für die Pixelverarbeitung ist das Ergebnis einer implizierten, bei der Rasterung berechneten Interpolation $\text{lerp}(x_1, x_2, f)$[8]. f wird von der GPU berechnet und variiert mit jedem Pixel. (Für das Pixel, auf das das erste Vertex abgebildet wurde, ist $f = 0$; für das Pixel, auf das das zweite Vertex abgebildet wurde, ist $f = 1$; Pixel zwischen diesen „Randpixeln“ haben entsprechend Werte von f dazwischen – siehe Abbildung 2.4).

Betrachtet sei eine Rechenoperation $\alpha = g(x)$, die während der Pixelberechnung ausgeführt wird. x sei dabei eine Ausgabe der Vertexberechnung, also aus Ausgaben x_1 und x_2 der Vertexberechnung für zwei verschiedene Vertices interpoliert – $x = \text{lerp}(x_1, x_2, f)$ mit beliebigem f. Ein Berechnen der Operation während der Pixelberechnung entspricht einem Berechnen von $\alpha_1 = g(\text{lerp}(x_1, x_2, f))$. Ein Berechnen der Operation während der Vertexberechnung und anschliessende Interpolation entspricht einem Berechnen von $\alpha_2 = \text{lerp}(g(x_1), g(x_2), f)$. Die Operation $g(x)$ kann auch in der Vertexberechnung ausgeführt werden, wenn das Ergebnis dieser Berechnung interpoliert werden kann ohne das Endergebnis in der Pixelberechnung zu verändern, also $\alpha_1 = \alpha_2$ für beliebige f gilt.

Also sei $g(x)$ *interpolierbar* genannt, wenn $g(\text{lerp}(x_1, x_2, f)) = \text{lerp}(g(x_1), g(x_2), f)$ gilt.

Augenscheinlich muss g eine lineare Funktion sein.

Für binäre Rechenoperationen lautet die Bedingung $\text{lerp}(g(x_1, y_1), g(x_2, y_2), f) = g(\text{lerp}(x_1, x_2, f), \text{lerp}(y_1, y_2, f))$. Rechenoperationen mit mehr Operanden müssen nicht betrachtet werden, da die Shadingsprache höchstens binäre Rechenoperationen vorsieht bzw. „komplexere“ Rechenoperationen auf binäre Operationen heruntergebrochen werden können (siehe auch 4.6.6.4).

4.6.6.3 Allgemein

Da Rechenoperationen auf Vektoren komponentweise ausgeführt werden, ist die „Interpolierbarkeit“ ohne weiteres auch auf Vektoroperationen anwendbar.

Die geforderte Linearität von $g(x)$ führt dazu, dass bloss Operationen auf `float`-Werten problemlos interpolierbar sind. Integer-Werte sind nicht interpolierbar, da bei deren Interpolation nicht-Integer-Werte als Zwischenergebnisse entstehen können.

[8] Tatsächlich muss bei der Rasterung von Dreiecken zwischen drei Werten interpoliert werden. Dieser Abschnitt betrachtet konkret bloss Interpolation zwischen zwei Werten, die Ergebnisse gelten aber auch bei Interpolation zwischen drei Werten.

Diese müssen (zwangsweise) gerundet[9] werden – diese Rundung ist aber keine stetige Funktion.

Zu Beachten ist, dass Interpolierbarkeit nur Relevanz für die Entscheidung hat, ob eine Operation, bei der mindestens ein Operand in der Vertexberechnung berechnet wurde bzw. Wert eines Vertexattributes ist, während der Vertex- statt der Pixelberechnung ausgeführt werden kann. Operationen allein auf Konstanten oder (praktisch konstanten) Objektattributen sind immer auch in der Vertexberechnung ausführbar - das Ergebnis einer solchen Operation kann nicht zwischen den Berechnungen für verschieden Vertices variieren. Insbesondere können damit auch Operationen auf Integer-Werten, oder Operationen, die nachfolgend als „nicht interpolierbar“ klassifiziert werden, in der Vertexberechnung ausgeführt werden.

4.6.6.4 Interpolierbarkeit einfacher Operationen

Arithmetische Operationen: Die Summe oder Differenz von zwei linearen Funktionen ist wieder eine lineare Funktion, Addition und Subtraktion sind also uneingeschränkt interpolierbar.

Multiplikation und Division sind interpolierbar, wenn mindestens ein Operand höchstens in der Objektberechnung – also nicht in der Vertex- oder Pixelberechnung – bestimmt wurde. Wurden beide Operanden in der Vertex- oder Pixelberechnung bestimmt, so ist eine Multiplikation oder Division keine lineare, sondern eine *quadratische* Funktion[10], und damit nicht interpolierbar. In der Objektberechnung bestimmte Operanden können aber praktisch als konstant betrachtet werden, die Multiplikation oder Division mit einem der Objektberechnung bestimmten Operanden ist also eine lineare Funktion und damit interpolierbar.

Die Modulo-Operation ist unstetig und daher nicht interpolierbar.

Logische Ausdrücke, Vergleichsoperationen: Diese Operationen sind ebenfalls unstetig und somit nicht interpolierbar.

Unäre Ausdrücke: Vorzeichenumkehrung ist offensichtlich interpolierbar.

[9] Oder Nachkommastelle abgeschnitten usw.

[10] Ersichtlich durch ein Beispiel: seien a und b in der Vertex- und/oder Pixelberechnung bestimmt, also $a = \text{lerp}(\ldots, x, f)$ und $b = \text{lerp}(\ldots, y, f)$, so ist das Produkt $a \cdot b = \cdots + \cdot f \cdot x \cdot f \cdot y = \cdots + f^2 \cdot x \cdot y$, also quadratisch.

Logisches NICHT ist, wie die anderen logischen Ausdrücke, nicht interpolierbar.

Bitweises invertieren ist nur für Integer-Werte sinnvoll und damit nicht interpolierbar.

Eingebaute Funktionen: Skalarprodukt, Vektorprodukt und Matrixmultiplikation lassen sich alle mit arithmetischen Basisoperationen darstellen. Da in diesen Darstellungen jeweils auch die Multiplikation enthalten ist, ergeben sich die oben genannten Beschränkungen: diese Operationen sind nur interpolierbar, wenn mindesten ein Operand in der Objektberechnung bestimmt wurde.

Potenzierung ist im Allgemeinen keine stetige Funktion und damit nicht interpolierbar.

Die Berechnung von „Normalisierung“ und „Euklidische Länge“ erfordert das Ziehen einer Wurzel bzw. Potenzieren mit $\frac{1}{2}$. Damit sind diese Funktionen ebenfalls nicht interpolierbar.

Minimum und Maximum sind im Allgemeinen keine stetigen Funktion und damit nicht interpolierbar.

Die Texturfunktionen sind prinzipbedingt nicht interpolierbar.

4.6.7 Ablauf der Auftrennung

4.6.7.1 Prinzipieller Ablauf

Die Auftrennung wird auf des Basis von Sequenzen vorgenommen. Für jede Sequenzoperation wird nacheinander entschieden, in welchen Ausgabeprogrammen es berechnet sie kopiert werden soll. Für das Ergebnisregister wird vermerkt, in welchen Ausgabeprogrammen es berechnet wurde – diese Information wird für die „Auftrennung“ nachfolgender Operationen verwendet.

Die Auftrennung beginnt mit der Eintrittsfunktion des Programms. Für deren Eingabeparameter muss bereits bekannt und gegeben sein, in welchen Ausgabeprogrammen sie verwendet werden können.

4.6.7.2 Auftrennung einfacher Operationen

Für einfache Operationen entscheidet die oben genannte „Interpolierbarkeit“, in welche Ausgabeprogramme eine Operation ausgegeben wird.

4.6.7.3 Auftrennung von Flusskontrolloperationen

Neben mathematischen Ausdrücken kann ein aufzuspaltendes Programm auch Operationen der Flusskontrolle enthalten, die ebenso auf mehrere Ausgabeprogramme aufgespaltet werden müssen.

Grundsätzlich richten sich die Ausführungsfrequenzen von den Ablaufsteuerungsoperationen nach den Frequenzen der „eingebetteten“ Operationen. Im Gegensatz zu „einfachen“ Operationen wird eine Ablaufsteuerungsoperationen meist in mehrere oder alle Ausgabeprogramme übernommen, allerdings mit anderen „eingebetteten“ Operationen.

Sequenzschachtelung: Eine Sequenzschachtelungsoperation wird prinzipiell zu allen generierten Teilprogrammen hinzugefügt, allerdings mit unterschiedlichen Sequenzen. Diese sind selbst das Ergebnis einer Aufspaltung der ursprünglichen Sequenz.

Verzweigung: Eine Verzweigung besteht konzeptionell aus zwei inneren Sequenzen (für die jeweiligen Verzweigungsblöcke) und einem bool’scher Bedingungswert, nach dessen Wert abhängig verzweigt wird.

Die beiden inneren Verzweigungssequenzen werden selbst aufgespalten.

Der bool’scher Bedingungswert kann entweder in der Objektberechnung oder in der Pixelberechnung bestimmt worden sein: in der Objektberechnung, wenn er ein Objektattribut ist bzw. allein aus Objektattributen berechnet wurde. In der Pixelberechnung, wenn er aus Werten berechnet wurden, die in anderen Berechnungsfrequenzen bestimmt wurden. Aufgrund der Nicht-Interpolierbarkeit von Vergleichs- und Logikoperationen kann ein Bedingungswert nicht in der Vertexberechnung bestimmt worden sein.

Selbst wenn der Bedingungswert nur in der Pixelberechnung vorliegt, kann die Verzweigungsoperation nicht allein in der Pixelberechnung ausgeführt werden: die Aufspaltung der Verzweigungsblöcke kann auch in in der Objektberechnung bzw. der Vertexberechnung auszuführende Sequenzen resultieren. Diese müssen als „normale“ innere Sequenzen, d.h. ohne Verzweigung, zu den Objekt- bzw. Vertexberechnungen hinzugefügt werden. In diesen Berechnungen werden also immer die Vertexteile beider Verzweigungsblöcke ausgeführt. Zwischenergebnisse müssen an das Pixelprogramm übertragen und dort ausgewählt werden.

Von den Verzweigungsblöcken beschriebene Register sind immer nur in demselben Programm, in dem der Bedingungswerts bestimmt wurde, verfügbar: Selbst wenn beide Verzweigungen bloss Operationen für die Vertexberechnung enthalten, sind durch die Auswahl nach einer in der Pixelberechnung vorliegenden Bedingung die berechneten Wert in der Pixelberechnung verfügbar.

Schleife: Eine Schleifenoperation besteht aus einer inneren Sequenz (dem Schleifenrumpf) sowie einem booleschem Bedingungswert, der Schleifenbedingung.

Wie bei der Verzweigung kann der boolescher Bedingungswert nur entweder in der Objektberechnung oder in der Pixelberechnung bestimmt worden sein.

Bei der Bestimmung, in welcher Frequenz einzelne Operanden im Schleifenrumpf vorliegen, ergibt sich das Problem, dass die Teilprogramme, in denen ein Register verfügbar ist, von der Anzahl der Schleifendurchläufe abhängt: z.B. kann eine Multiplikation eines Registers mit einem in der Vertexberechnung vorliegendem Wert im ersten Durchlauf eine interpolierbare Operation sein, im zweiten Durchlauf aber nicht mehr, wenn dem Register ein bloss in der Vertexberechnung verfügbares Ergebnis aus dem vorherigen Durchlauf zugewiesen wurde. Um eine „stabile“ Verfügbarkeit von Registern, die von vorherigen Schleifendurchläufen abhängen, zu ermitteln, werden zwei Schleifendurchläufe[11] simuliert und die damit bestimmten Verfügbarkeiten für das Aufspalten des Schleifenrumpfes verwendet.

Teile des Rumpfes können in der Objektberechnung oder der Vertexberechnung ausgeführt werden, wenn die Operationen nicht von vorhergegangenen Schleifendurchläufen abhängen. Ein Ausführen von Operationen in der Objektberechnung oder der Vertexberechnung, die eine solche Abhängigkeit besitzen, wäre problematisch: es müsste bekannt sein, wieviele Werte vom Objekt- bzw. Vertexprogramm zum Pixelprogramm übertragen werden sollen. Dies hängt von der Anzahl der Schleifendurchläufe ab, welche im Allgemeinen nicht zur Übersetzungszeit bestimmt werden kann. Ausführen von „echten“ Schleifen ist also nur komplett in der Objektberechnung oder der Pixelberechnung möglich.

Funktionsaufruf: Der Rumpf einer Funktion ist eine Sequenz, kann also prinzipiell in einen Teil für die Objektberechnung, einen Teil für die Vertexberechnung und einen Teil

[11] Zwei Durchläufe sind ausreichend, da ein Register nach höchstens zwei Schritten in der höchsten Frequenz verfügbar ist – von Objekt auf Vertex bzw. von Vertex auf Pixel.

für die Pixelberechnung aufgespalten werden. Das Hauptproblem besteht dabei darin, dass die Verfügbarkeit der Funktionsparameter je nach Aufruf variieren können.

Dies wird gelöst, in dem für eine Funktion bei der Aufspaltung mehrere Varianten generiert werden: bei einem Funktionsaufruf wird die Funktion aufgespalten, mit den jeweiligen Verfügbarkeiten der übergebenen aktuellen Parameter. Wird die gleiche Funktion nochmals aufgerufen, aber mit einer anderen „Signatur“ von Verfügbarkeiten, so wird die Funktion in neue Varianten aufgespalten usw.

Die Verfügbarkeiten von Ausgabeparametern hängen von den Verfügbarkeiten der Eingabeparameter ab, können aber nach dem Aufspalten einer Variation bestimmt werden.

Das Übertragen von Zwischenwerten einer Funktionsvariation vom Teil der Vertexberechnung zum Teil der Pixelberechnung geschieht über generierte Ausgabe- bzw. Eingabeparameter; die eigentliche Übertragung geschieht schlussendlich in der Eintrittsfunktion.

Für Rekursionen ergibt sich folgendes Problem: um die korrekten Verfügbarkeiten der Ausgabeparameter zu bestimmen muss eine Funktion zunächst aufgespalten werden. Im Falle einer Rekursion wird möglicherweise aber genau die angetroffene Variante gerade selbst aufgespalten – die Verfügbarkeiten sind also noch nicht bekannt, der Versuch einer Aufspaltung der angetroffenen Variante würde über kurz oder lang zum exakt selben Problem führen.

Dieses Problem wird vermieden, in dem rekursive Funktionen besonders behandelt werden: im Wesentlichen wird angenommen, dass bei einem festgestelltem rekursiven Aufruf die Ausgabeparameter nur in der Pixelberechnung verfügbar sind. Mit dieser konservativen Annahme kann der Rumpf einer rekursiven Funktion aufgespalten werden.

4.6.7.4 Auftrennung von Array-Operationen

Array-Operationen sind nur interpolierbar, wenn die Länge des Arrays *statisch konstant* bekannt (also auch kein Objektattribut) ist und bei der Array-Extraktion bzw. -Änderung der verwendete Index in der Objektberechnung verfügbar ist. Dementsprechend können Array-Operationen nur in der Vertexberechnung ausgeführt werden, wenn wenigstens auch dass Array, mit dem gearbeitet wird, eine statisch konstante Länge besitzt. Bei Änderungen von Arrayelement muss ausserdem der zugewiesene Wert in

der Vertexberechnung berechnet worden sein.

4.6.7.5 Andere Operationen

Zuweisungen, Konstantenoperationen, Typumwandlung, die Erstellung eines Vektors sowie die Extraktion einer Komponente können trivialerweise mit derjenigen Frequenz berechnet werden, die der höchsten der Frequenzen, in denen die Eingaben berechnete wurden, entspricht.

4.6.8 Beispiel

Ein einfaches Beispiel soll die Ausgabe des Splitters demonstrieren.

4.6.8.1 Eingabeprogramm

Als Eingabe dient das einfache Shading-Programm aus Abbildung 4.10. Es basiert auf dem Beispiel in Abbildung 2.6 – dessen Vertex- und Pixel-Teil wurden hier zusammengeführt.

Das Programm berechnet eine Beleuchtungsintensität für eine Lichtquelle mit konstanter Richtung des Lichteinfalls (also eine Lichtquelle im Unendlichen) und gegebener Farbe. Diese Beleuchtungsintensität mit der “ambient”-Farbe (eine Annäherung von „Streulicht“) zu einer Gesamtintensität addiert. Diese wird dann mit einer aus einer Textur ausgelesenen Oberflächenfarbe moduliert.

In der Abbildung wurde (manuell) markiert, in welchen Frequenzen die Eingabeparameter vorliegen, sowie die Frequenz, in der die einzelnen Ausdrücke ausgeführt werden müssten.

4.6.8.2 Auftrennung

Die für die Ausdrücke anzuwendenen Aufspaltungsregeln sind:

`outPosition` : `Position` liegt ist der Wert eines Vertexattributs vor, der Gesamtausdruck muss daher mindestens in der Vertexberechnung berechnet werden. `ModelViewProj` ist ein Objektattribut, die Matrixmultiplikation ist interpolierbar, der Gesamtausdruck kann daher auch in der Vertexberechnung berechnet werden und muss nicht in der Pixelberechnung berechnet werden.

```
void main ( in float4 Position,
      in float2 TexCoord,
      in float3 Normal,
      in float4x4 ModelViewProj,
      in float3 LightColor,
      in float3 LightDirObj,
      in sampler2D Texture,
      out float4 outPosition,
      out float4 outColor)
{
      outPosition = mul (ModelViewProj, Position);
      float3 ambient = float3 (0.4);
      float3 diffuse = LightColor * dot (LightDirObj, Normal);
      float3 litColor = diffuse + ambient;
      outColor = tex2D (Texture, TexCoord)
          * float4 (litColor, 1.0);
}
```

Abbildung 4.10: Ein Programm in der Shading-Sprache.
Markierung der Operationen bzw. Werte gibt an, mit in welchem Berechnungsschritt diese ausgeführt bzw. berechnet werden. (Objekt , Vertex , Pixel)

ambient : Konstante, wird daher als Objektattribut betrachtet.

diffuse : Das Skalarprodukt zwischen `LightDirObj` und `Normal` muss, da `Normal` der Wert eines Vertexattributes ist, mindestens in der Vertexberechnung berechnet werden. `LightDirObj` ist ein Objektattribut, das Skalarprodukt ist also interpolierbar, kann daher auch in der Vertexberechnung berechnet werden und muss nicht in der Pixelberechnung berechnet werden.

Ebenso ist die Multiplikation von `LightColor` und dem Skalarprodukt interpolierbar, der Gesamtausdruck kann also in der Vertexberechnung berechnet werden und muss nicht in der Pixelberechnung berechnet werden.

litColor : Da `diffuse` in der Vertexberechnung vorliegt, `ambient` ein Objektattribut ist, und weiterhin Additionen uneingeschränkt interpolierbar sind, kann der Gesamtausdruck in der Vertexberechnung berechnet werden und muss nicht in der

Pixelberechnung berechnet werden.

`outColor` : Da `litColor` in der Vertexberechnung bestimmt wurde wird auch das Erstellen eines `float4`-Vektors aus `litColor` in der Vertexberechnung vorgenommen.

Das Auslesen des Oberflächenmusters mit `tex2D` kann nur in der Pixelberechnung erfolgen. Damit muss auch die Multiplikation des Texturwertes mit dem `float4`-Vektor in der Pixelberechnung erfolgen.

4.6.8.3 Ausgabe

Abbildungen 4.11 und 4.12 enthalten das Ergebnis nach Aufspaltung in Zwischencoderepräsentation. (Vor oder nach der Aufspaltung wurden keine Optimierungen angewendet.)

Die Aufspaltung wirkt sich auf alle Aspekte der Programme aus:

- *Eingabeparameter:* Im Vertexprogramm sind alle Eingabeparameter des Originalprogramms erhalten, auch `Texture`, welches nur im Pixelprogramm ausgelesen wird.

  Bei den Eingabeparametern des Pixelprogramms fehlen die als Werte von Vertexattributen vorliegenden Eingaben – diese werden stattdessen vom Vertexprogramm weitergegeben („Übertragene Register“). Allerdings gibt es auch hier „unnötige“ Eingaben, wie `ModelViewProj`.

- *Ausgabeparameter:* Dies sind die vom jeweiligen Verarbeitungsschritt zu berechnenden Ausgaben – die projizierte Vertex-Position im Falle des Vertexprogramms bzw. die finale Pixelfarbe im Falle des Pixelprogramms.

- *Übertragene Register:* Die „Schnittstelle“ zwischen Vertex- und Pixelprogramm. Dabei enthält *m_TexCoord_B0* den Wert des Eingabeparameters `TexCoord`, der als Wert eines Vertexattributs vorliegt, aber nur im Pixelprogramm verwendet wird. *i_tmp12* das Ergebnis der Beleuchtungsberechnung – der rechte Faktor in der abschliessenden Multiplikation nach `outColor` – welches im Pixelprogramm mit der aus dem Oberflächenmuster ausgelesenen Farbe multipliziert werden soll.

Der Aufspalter hat diese Register zur Übertragung ausgewählt, da deren Werte in der Vertexberechnung bestimmt wurden, aber im Pixelprogramm verwendet werden.

- *Operationen:* Der Splitter hat die in der Vertexberechnung und per in der Pixelberechnung auszuführenden Operationen genau so aufgeteilt, wie es nach der manuellen Klassifizierung im Eingabeprogramm zu erwarten war. Die in der Objektberechnung zu berechnenden Operationen wurden, wie in Abschnitt 4.6.4 beschrieben, sowohl in das Vertex- wie auch das Pixelprogramm ausgegeben. In diesem Fall ist dies nur die Zuweisungen von *v_ambient* und *i_tmp11*. Diese Werte werden im Pixel-Programm nicht verwendet, die Operationen selbst bleiben aber in diesem Fall wegen nicht vorgenommener Optimierungen dort erhalten.

Abbildung 4.13 visualisiert die Anteile der Berechnungen der verschiedenen Berechnungsschritte an der finalen Pixelfarbe. Dafür wurde mit Hilfe des Compilers das Eingabeprogramm in ein Vertex- und Pixelprogramm in der Shadingsprache Cg übersetzt. Diese Ausgabeprogramme sind semantisch äquivalent zum Beispielprogramm aus Abbildung 2.6 – dementsprechend sind auch die Bilder identisch zu denen in Abbildung 2.7.

Abbildung 4.13(a) zeigt den Anteil der Objektberechnung (`ambient`) und zeigt dementsprechend eine durchgängige Färbung.

Abbildung 4.13(b) zeigt wurde der Anteil der Vertexberechnung (`litColor`) hinzugenommen. Es ist deutlich erkennbar, dass die Farbe über das Modell stark variiert und damit eine Beleuchtung aus der rechten oberen Richtung suggeriert.

Im Abbildung 4.13(c) wurde schließlich auch der Anteil der Pixelberechnung hinzugenommen. Die Farbveränderung aus der „Beleuchtung“ ist noch klar erkennbar, aber das Auslesen einer Oberflächenfarbe aus einer Textur fügt weitere visuelle Details hinzu.

Funktion : `main`
Eingabeparameter : `float4 Position` → *m_Position_B0*
`float2 TexCoord` → *m_TexCoord_B0*
`float3 Normal` → *m_Normal_B0*
`float4x4 ModelViewProj` → *m_ModelViewProj_B0*
`float3 LightColor` → *m_LightColor_B0*
`float3 LightDirObj` → *m_LightDirObj_B0*
`sampler2D Texture` → *m_Texture_B0*
Ausgabeparameter : `float4 outPosition` ← *m_outPosition_B0*
Registersatz `float`: *i_tmp1 i_tmp10 i_tmp11 i_tmp3 i_tmp8 i_tmp9*
Registersatz `float2`: *m_TexCoord_B0*
Registersatz `float3`: *i_tmp4 m_LightColor_B0 m_LightDirObj_B0 m_Normal_B0 v_ambient v_diffuse v_litColor*
Registersatz `float4`: *i_tmp12 m_Position_B0 m_outPosition_B0*
Registersatz `float4x4`: *m_ModelViewProj_B0*
Registersatz `sampler2D`: *m_Texture_B0*
Operationen :

```
     mov  i_tmp1, 0.4
 makevec  v_ambient, i_tmp1, i_tmp1, i_tmp1
     mov  i_tmp11, 1
    mmul  m_outPosition_B0, m_ModelViewProj_B0, m_Position_B0
     dot  i_tmp3, m_LightDirObj_B0, m_Normal_B0
 makevec  i_tmp4, i_tmp3, i_tmp3, i_tmp3
     mul  v_diffuse, m_LightColor_B0, i_tmp4
     add  v_litColor, v_diffuse, v_ambient
 extract  i_tmp8, v_litColor, 0
 extract  i_tmp9, v_litColor, 1
 extract  i_tmp10, v_litColor, 2
 makevec  i_tmp12, i_tmp8, i_tmp9, i_tmp10, i_tmp11
```

Übertragene Register : *m_TexCoord_B0 i_tmp12*

Abbildung 4.11: Vertexprogramm aus 4.10 nach der Aufspaltung.

Funktion :	`main`
Eingabeparameter :	`float4x4 ModelViewProj` → *m_ModelViewProj_B0*
	`float3 LightColor` → *m_LightColor_B0*
	`float3 LightDirObj` → *m_LightDirObj_B0*
	`sampler2D Texture` → *m_Texture_B0*
Ausgabeparameter :	`float4 outColor` ← *m_outColor_B0*
Registersatz `float`:	*i_tmp1 i_tmp11*
Registersatz `float2`:	*m_TexCoord_B0*
Registersatz `float3`:	*m_LightColor_B0 m_LightDirObj_B0 v_ambient*
Registersatz `float4`:	*i_tmp12 i_tmp7 m_outColor_B0*
Registersatz `float4x4`:	*m_ModelViewProj_B0*
Registersatz `sampler2D`:	*m_Texture_B0*
Operationen :	`mov` *i_tmp1*, 0.4
	`makevec` *v_ambient*, *i_tmp1*, *i_tmp1*, *i_tmp1*
	`mov` *i_tmp11*, 1
	`tex2D` *i_tmp7*, *m_Texture_B0*, *m_TexCoord_B0*
	`mul` *m_outColor_B0*, *i_tmp7*, *i_tmp12*

Abbildung 4.12: Pixelprogramm aus 4.10 nach der Aufspaltung.

(a)  (b)  (c) 

Abbildung 4.13: Programm aus 4.10
(a) nur Anteil der Objektberechnungen an Pixelfarbe
(b) zusätzlich mit Anteil der Vertexberechnungen
(c) vollständige Pixelfarbe

4.7 Optimierer

Die Aufgabe eines *Optimierers* ist es, ein gegebenes Programm so umzuschreiben, dass es zur Laufzeit schnellstmöglich ausgeführt wird. Im Allgemeinen erstrecken sich mögliche Optimierungen von Vereinfachungen, wie ein Entfernen von unbenötigten Operationen und ein Berechnen von Ausdrücken zur Übersetzungszeit, bis hin zu komplexen Umsortierungen oder Zusammenfassung von Operationen, z.B. um “multiply-add”-Operation besser auszunutzen.

Ein Optimierer, der Redundanzen u.ä. entfernt, erlaubt als Nebeneffekt auch, dass die vorhergehenden Arbeitsschritte eines Compilers – im vorliegenden Compiler insbesondere der Splitter – nicht selbst optimalen Zwischencode erzeugen müssen. Die Implementierungen jener Arbeitsschritte kann damit einfacher gehalten werden.

4.7.1 Aufbau

Der Optimierer im vorliegenden Compiler nimmt als Eingabe ein komplettes Programm entgegen und liefert als Ausgabe entsprechend auch ein komplettes Programm.

Das Arbeiten auf ganzen Programmen erlaubt es prinzipiell, funktionsübergreifende Optimierungen vorzunehmen, von einem Entfernen unbenötigter Funktionen über ein Einbetten von Funktionsaufrufen bis zu einem Entfernen einzelner, nicht benötigter Funktionsparameter.

Allerdings beschränken sich die implementierten, unten beschriebenen Optimierungen auf Sequenzen von Operationen; diese Sequenzoptimierungen werden auf alle Funktionen eines Programms unabhängig voneinander angewendet.

Die einzelnen Sequenzenoptimierungen werden nacheinander angewendet. Bei Bedarf kann eine Optimierung mehrmals angewendet werden – z.B. kann die Konstantenfaltung einen Ausführungszweig einer Verzweigung auswählen; in diesem Fall wird eine erneute Blockauflösung veranlasst, um die dadurch eingefügte Sequenzschachtelungsoperation zu vereinfachen.

4.7.2 Blockauflösung

Die *Blockauflösung* ersetzt alle Sequenzschachtelungsoperationen einer Sequenz durch die in der inneren Sequenz enthaltenen Operationen. Dabei werden die von den Operationen verwendeten bzw. veränderten Register angepasst: Register, die

einem importiertem oder exportiertem Namen zugeordnet sind, werden durch ihre Gegenstücke in der umgebenden Sequenz ersetzt. Andere Register werden umbenannt um Kollisionen zu vermeiden.

Die von der Blockauflösung vorgenommen Änderungen an einer Sequenz haben keine Auswirkungen auf das Laufzeitverhalten des Programms. Dieser Optimierungsschritt dient vor allem als „Hilfsschritt“, um die Implementierungen anderer Optimierungsschritte zu vereinfachen: einerseits können Sequenzschachtelungsoperationen als „nicht vorhanden“ angenommen werden, andererseits kann ein Auflösen von Blöcken – z.B. wegen der statischen Auswahl eines Verzweigungsblockes – einfach vorgenommen werden, da die nötige „Hauptarbeit“ später vom Blockauflösungsschritt vorgenommen wird.

Abbildungen 4.14 bis 4.16 vergleichen ein einfaches Programm vor und nach der Anwendung der Blockauflösung.

```
void main()
{
  float a, b;
  a = 3.0;
  {
    b = a;
  }
}
```

Abbildung 4.14: Blockauflösung: Quellcode.

```
          Funktion :  main
Registersatz float:  v_a v_b
      Operationen :  mov  v_a, 3
                     Registersatz float:  m_a_B0 m_b_B0
                                 Import :  v_a → m_a_B0
                                    mov  m_b_B0, m_a_B0
                                 Export :  v_b ← m_b_B0
```

Abbildung 4.15: Blockauflösung: Zwischencoderepräsentation unoptimiert.

```
          Funktion :  main
Registersatz float:  v_a v_b
      Operationen :  mov  v_a, 3
                     mov  v_b, v_a
```

Abbildung 4.16: Blockauflösung: Zwischencoderepräsentation optimiert.

4.7.3 Konstantenfaltung

Konstantenfaltung berechnet das Ergebnis einer Sequenzoperation zur Übersetzungszeit wenn alle Eingabeoperanden zur Übersetzungszeit bekannte Konstanten sind. Berücksichtigt werden Zuweisungs-, Cast-, arithmetische, logische, unäre und Vergleichsoperationen, alle eingebauten Funktionen (mit Ausnahme von Texturfunktionen) sowie die Operationen „Arraylänge“ und „Extraktion eines Arrayelements“. Weiterhin werden Verzweigungsoperationen, deren Bedingung ein konstanter Wert ist, durch eine Schachtelung der entsprechenden Sequenz ersetzt (die andere Sequenz wird verworfen). Auch Schleifenoperationen, deren Schleifenbedingung den konstanten Wert „falsch“ besitzt, werden verworfen.

In der Implementierung der Konstantenfaltung wird eine Zuordnung zwischen Registern und konstanten Werten verwendet; ist einem Register kein solcher Wert zugeordnet besitzt es keinen bekannten konstanten Wert. Eine solche Zuordnung wird erstellt, wenn eine Operationen ein konstantes Ergebnis hat. Dies sind zuvorderst Konstantenoperationen. Bei Operationen, die einer der oben genannten Arten entsprechen, wird überprüft, ob alle Operanden konstante Werte sind. Ist dies der Fall wird das Ergebnis der Operation vom Compiler berechnet und die Operation durch eine einfache Konstantenoperation ersetzt.

Abbildungen 4.17 bis 4.19 vergleichen ein einfaches Programm vor und nach der Anwendung der Konstantenfaltung. Anzumerken ist, dass die Addition vom Compiler ausgerechnet und durch eine Konstantenoperation ersetzt wird; die Konstantenoperationen der Quelloperanden bleiben jedoch erhalten - deren Entfernung ist die Aufgabe der „Entfernung unnötiger Operationen“, welche aber in diesem Beispiel nicht durchgeführt wurde.

```
void main()
{
  float a = 2.0;
  float b = 3.0;
  float c = a+b;
}
```

Abbildung 4.17: Konstantenfaltung: Quellcode.

```
          Funktion :  main
Registersatz float:  v_a v_b v_c
       Operationen :  mov  v_a, 2
                      mov  v_b, 3
                      add  v_c, v_a, v_b
```

Abbildung 4.18: Konstantenfaltung: Zwischencoderepräsentation unoptimiert.

```
          Funktion :  main
Registersatz float:  v_a v_b v_c
       Operationen :  mov  v_a, 2
                      mov  v_b, 3
                      mov  v_c, 5
```

Abbildung 4.19: Konstantenfaltung: Zwischencoderepräsentation optimiert.

4.7.4 Entfernung unnötiger Operationen

Die „*Entfernung unnötiger Operationen*“ entfernt unnötige Sequenzoperationen; eine Operation ist „unnötig“, wenn keines der von ihr geschriebenen Register jemals benutzt wird.

In der Implementierung werden die Operationen einer Sequenz *von hinten* betrachtet. Es wird eine Menge von „ausgelesenen“ Registern gespeichert. Bei jeder Operation wird zunächst überprüft, ob wenigstens eins der Ausgaberegister ein solches „ausgelesenes“ Register ist. Ist dies nicht der Fall, wird die Operation verworfen. Andernfalls wird die Operation beibehalten und alle ihre Eingaberegister als „ausgelesen“ markiert.

Die Menge der „ausgelesenen“ Register benötigt eine anfängliche Menge von Registern, die als benutzt angenommen werden – ohne diese würden alle Operationen einer Sequenz verworfen werden. Dies sind immer die Ausgabeparameter der Funktion, die die zu optimierende Sequenz enthält. Für Sequenzen, die z.B. in Verzweigungen eingeschachtelt sind, dient die Menge der ausgelesenen Register der umgebenden Sequenz als anfängliche Menge benutzter Register.

Abbildungen 4.20 bis 4.22 vergleichen ein einfaches Programm vor und nach der Anwendung der Entfernung unnötiger Operationen: die Ergebnisse einiger Operationen im Programm 4.20 werden offentsichtlich nicht verwendet. Im optimierten Ausgabeprogramm 4.22 wurden diese Operationen entfernt, und nur die Berechnungen, die für die Werte der Ausgabeparameter nötig sind, sind erhalten.

```
void main(out float4 pos, out float4 color)
{
  float a = 1.0, b = 2.0;
  pos = float4 (a+b);
  float c = a / b;
  color = float4 (b);
  float4 prod = pos*color*c;
}
```

Abbildung 4.20: Entfernung unnötiger Operationen: Quellcode.

Funktion :	`main`
Ausgabeparameter :	`float4 pos` ← *m_pos_B0*
	`float4 color` ← *m_color_B0*
Registersatz `float`:	*i_tmp2 v_a v_b v_c*
Registersatz `float4`:	*i_tmp6 i_tmp7 m_color_B0 m_pos_B0 v_prod*
Operationen :	`mov` *v_a*, 1
	`mov` *v_b*, 2
	`add` *i_tmp2*, *v_a*, *v_b*
	`makevec` *m_pos_B0*, *i_tmp2*, *i_tmp2*, *i_tmp2*, *i_tmp2*
	`div` *v_c*, *v_a*, *v_b*
	`makevec` *m_color_B0*, *v_b*, *v_b*, *v_b*, *v_b*
	`mul` *i_tmp6*, *m_pos_B0*, *m_color_B0*
	`makevec` *i_tmp7*, *v_c*, *v_c*, *v_c*, *v_c*
	`mul` *v_prod*, *i_tmp6*, *i_tmp7*

Abbildung 4.21: Entfernung unnötiger Operationen: Zwischencoderepräsentation unoptimiert.

Funktion :	`main`
Ausgabeparameter :	`float4 pos` ← *m_pos_B0*
	`float4 color` ← *m_color_B0*
Registersatz `float`:	*i_tmp2 v_a v_b*
Registersatz `float4`:	*m_color_B0 m_pos_B0*
Operationen :	`mov` *v_a*, 1
	`mov` *v_b*, 2
	`add` *i_tmp2*, *v_a*, *v_b*
	`makevec` *m_pos_B0*, *i_tmp2*, *i_tmp2*, *i_tmp2*, *i_tmp2*
	`makevec` *m_color_B0*, *v_b*, *v_b*, *v_b*, *v_b*

Abbildung 4.22: Entfernung unnötiger Operationen: Zwischencoderepräsentation optimiert.

4.8 Code-Generator

Der *Code-Generator* übersetzt ein in der Zwischencoderepräsentation vorliegendes Programm in eine *Zieldarstellung*. Diese „Darstellung“ kann wieder eine Hochsprache sein, prinzipiell kann ein Code-Generator aber auch Assembler-Quelltext oder eine Binärformat ausgeben.

4.8.1 Eingaben und Aufgaben

Der *Code-Generator* erhält als Eingabe eine Liste von Funktionsbeschreibungen. Eine Funktionsbeschreibung besteht aus dem eindeutigen Bezeichner, einer Liste von Parametern (getrennt nach Ein- und Ausgabeparametern) und einer Sequenz mit den eigentlichen Operationen. Eine Funktion aus der Liste ist als „Eintrittsfunktion“ markiert.

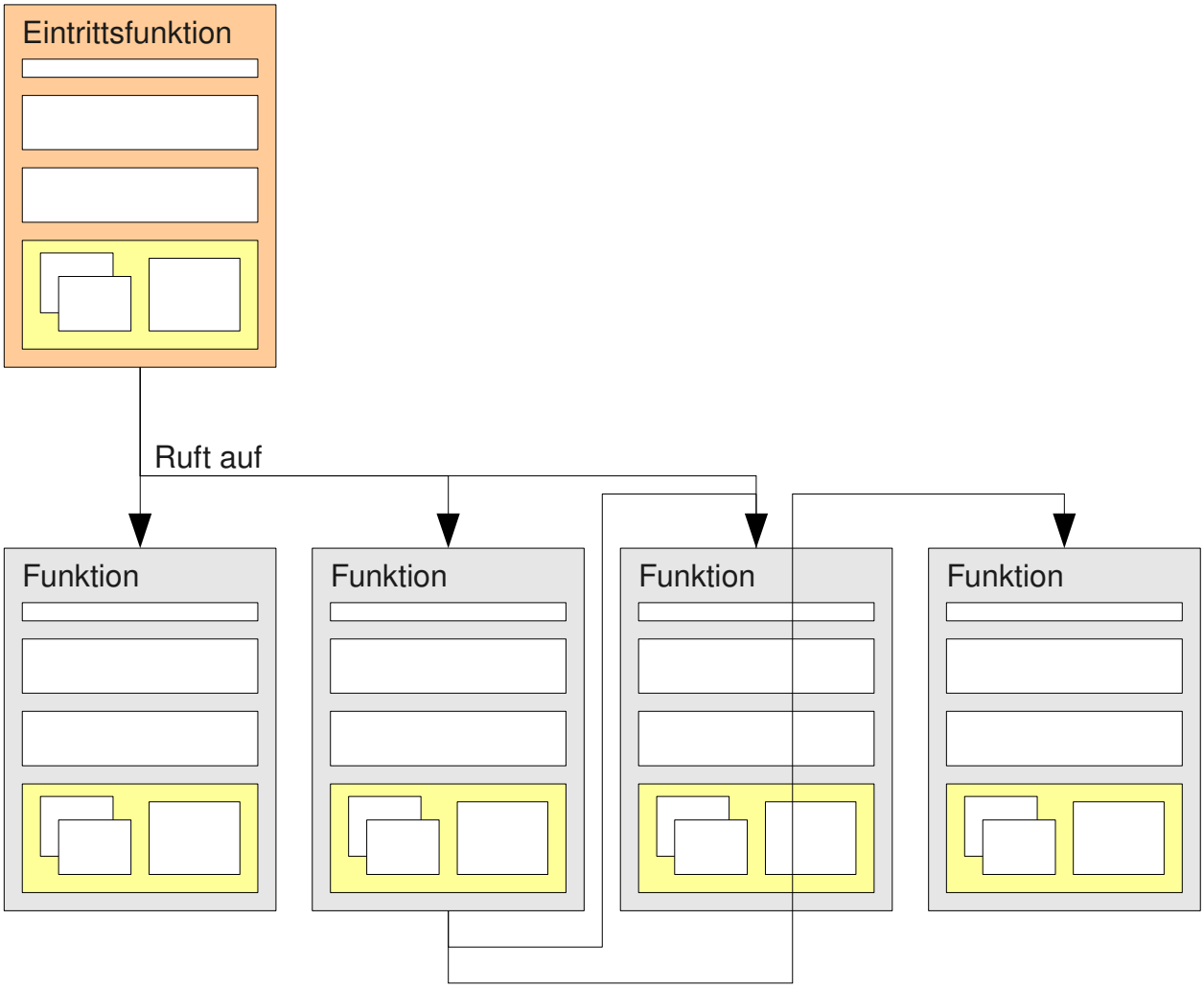

Abbildung 4.23: Zusammensetzung eines Programms in der Zwischencoderepräsentation

Die Aufgaben des Code-Generators bestehen aus:

- Nötige Umformungen für die Zieldarstellung – z.B. Schleifen ausrollen oder „Einbettung“ von Funktionen.
- Übertragung der Funktionsbeschreibungen in eine entsprechende Deklaration in der Zieldarstellung.
- Übersetzung der Sequenzoperationen in die Zieldarstellung.
- Ressourcenallokation, wenn nötig (z.B. bei begrenzter Registerzahl in der Zieldarstellung).
- Generierung von „Schnittstellen-Anweisungen“ wie z.B. bei der Übergabe von Parametern an Funktionen oder die Behandlung von Werten „vor dem ersten

Durchlauf, nach dem ersten Durchlauf“ wie sie bei Schleifen auftreten.

4.8.2 Generator für Cg

In der vorliegenden Implementierung wurde als Zieldarstellung die Sprache Cg ([MGAK03], [NVI10]) gewählt.

Cg als Hochsprache kennt selbst Konstrukte wie Funktionen und Schleifen. Umformungen durch den Codegenerator sind also nicht nötig.

Jede Funktion wird also direkt auf eine Cg-Funktion abgebildet.

Jedem Register der Zwischencoderepräsentation wird eine Variable zugeordnet – eine Registerallokation ist unnötig, diese wird später vom Cg-Compiler selbst vorgenommen. „Einfache“ Sequenzoperation (arithmetische Operationen u.ä.) lassen sich auf einen einzelnen Cg-Befehl übertragen. Weiterhin werden die meisten eingebauten Funktionen und Attribute direkt von Cg unterstützt (Ausnahme ist das Matrix-Attribut `inverted`).

Kompliziertere Operationen – Verzweigungen, Schleifen, Funktionsaufrufe – resultieren in mehreren Statements, obwohl es sich bei dem zusätzlichen „Aufwand“ meist nur um Zuweisungen zwischen Variablen handelt, wie z.B. die Auswahl des Bedingungsregisters basierend zwischen den Wert vor und nach dem ersten Schleifendurchlauf.

Die spezifizierte Sprache unterstützt Zeichenketten aus Unicode-Buchstaben und -Ziffern als Bezeichner; Cg nur eine Untermenge von ASCII. Für die Ausgabe als Cg-Code werden Bezeichner in eine Darstellung in der akzeptieren Zeichenmenge umgewandelt[12].

4.9 Zusammenfassung

Grundsätzlich folgt der Aufbau des Compilers der Standardarchitektur; die Abweichnung ist der Verarbeitungsschritt der „Auftrennung“ in mehrere Programme.

Die Auftrennung nutzt dabei Eigenschaften von GPUs aus, deren Aufbau auf den Ablauf des Echtzeit-3D-Renderings abgestimmt ist. Speziell werden Operationen in der Vertex- statt Pixelberechnung ausgeführt, sofern das Ergebnis zur ursprünglichen

[12] Als Kodierung wurde Punycode ([RFC3492]) gewählt da dies die aus ASCII bestehenden Teile eines Bezeichners gut lesbar lässt.

Operation mathematisch äquivalent ist (unter Berücksichtigung der von der GPU vorgenommen Interpolation von Ausgaben der Vertexberechnung).

Auch hervorzuheben ist die verwendete Zwischencoderepräsentation, die als Übergabeformat zwischen allen Verarbeitungsschritten von der semantischen Analyse bis zur Codegenerierung dient. Weiterhin ist sie darauf ausgerichtet, die Implementierung von Optimierungsschritten möglichst zu vereinfachen.

Kapitel 5

Evaluation

In diesem Kapitel sollen Verhalten und Ausgabe des Compilers bewertet werden.

Um möglichst praxisnahe Fälle zu betrachten wurden die Programme, anhand derer die Bewertungen vorgenommen werden, aus existierenen Cg-Programmen „erstellt“; dazu wurden die als separate Vertex- und Pixelprogramme vorliegenden Quellen in Programme der zu übersetzenden Shadingsprache umgewandelt[1] und anschliessen vom Compiler übersetzt. Die Ausgabe des Compilers, die „generierten Programme“, sind wiederum Cg-Programme. Zur Überprüfung der syntaktischen Korrektheit wurden die generierten Programme mit dem Cg-Compiler `cgc` übersetzt (alle generierten Programme wurden fehlerfrei übersetzt, sind also syntaktisch korrekt). Schliesslich wurden, zur Effizienzbewertung, ursprüngliche wie auch generierte Cg-Programme durch den Cg-Compiler in eine Assembler-artige Shadingsprache übersetzt und diese jeweiligen Ausgaben miteinander verglichen.

Die Ausgabe-Programme des entwickelten Compilers können, da diese ja selbst Cg-Programme sind, mit der Eingabe prinzipiell direkt verglichen werden. Praktisch werden Vergleiche dadurch erschwert, dass die ausgebenen Programme prinzipbedingt drastisch anders formatiert sind, Variablen andere Namen als in den Eingabeprogrammen besitzen usw.

Folgende Aspekt wurden im Vergleich von Eingabe- und Ausgabeprogrammen betrachtet:

- Grundsätzliche *Semantische Äquivalenz* der beiden Programme. Diese ist von höchster Wichtigkeit.

[1] Wegen der großen syntaktischen Ähnlichkeiten konnte dies größtenteils durch Kopieren von Quelltextblöcken vorgenommen werden.

- *Effizienz* des Ausgabeprogrammes, Wirksamkeit von Optimierungen: das Ausgabeprogramm sollte keine unnötigen Operationen enthalten und es sollten möglichst wenige Werte vom Vertexprogramm ausgegeben werden.

Die meisten der verwendeten Eingabe-Cg-Programme stammen aus dem Buch "The Cg Tutorial" ([RF03]). Die Programme wurden ausgewählt, da sie einerseits Einführungs-Charakter besitzen – sie sind vergleichsweise einfach aufgebaut – aber andererseits ohne Weiteres praktisch einsetzbar sind.

5.1 Beispielprogramm

Das erste betrachtete Programm ist das in der Einführung verwendete Beispielprogramm (Abbildung 2.6): es berechnet für jedes Vertex des Modells einen Beleuchtungswert; dieser wird über die Pixel der ausgebenen Dreiecke (implizit) interpoliert. Weiterhin wird für jedes Pixel eine „Materialfarbe“ aus einem Oberflächenmuster gelesen und mit dem Beleuchtungswert moduliert.

```
1  struct VertexOutput {
2    float4 Position : POSITION;
3    float2 TexCoord;
4    float3 litColor;
5  };
6
7  void vertex_main (
8    in varying float4 Position,
9    in varying float2 TexCoord,
10   in varying float3 Normal,
11   in uniform float4x4 ModelViewProj,
12   in uniform float3 LightColor,
13   in uniform float3 LightDirObj,
14   out VertexOutput output)
15 {
16   output.Position = mul (ModelViewProj, Position);
17   float3 ambient = float3 (0.4);
18   output.litColor = LightColor * dot (LightDirObj, Normal);
19   output.litColor += ambient;
20   output.TexCoord = TexCoord;
21
22 }
23
24 void pixel_main (in VertexOutput interpolatedVertexOutput,
25   in uniform sampler2D Texture,
26   out float4 outColor : COLOR)
27 {
28   float3 surface =
29     tex2D (Texture, interpolatedVertexOutput.TexCoord).rgb;
30   float3 lighting =
31     interpolatedVertexOutput.litColor;
32   outColor.rgb = surface * lighting;
33   outColor.a = 1;
34 }
```

Programm 1: Ursprüngliches Programm in Cg (Vertex- und Pixelprogramm)

```
1  void main (in float4 Position,
2             in float2 TexCoord,
3             in float3 Normal,
4             in float4x4 ModelViewProj,
5             in float3 LightColor,
6             in float3 LightDirObj,
```

```
7             in sampler2D Texture,
8             out float4 outPosition,
9             out float4 outColor)
10 {
11         outPosition = mul (ModelViewProj, Position);
12
13         float3 ambient = float3 (0.4);
14         float3 litColor = LightColor * dot (LightDirObj, Normal);
15         litColor = litColor + ambient;
16
17         float3 surface = tex2D (Texture, TexCoord).rgb;
18         outColor.rgb = surface * litColor;
19         outColor.a = 1;
20 }
```

Programm 2: Eingabeprogramm für den Compiler in der Shadingsprache

```
1  struct V2F
2  {
3    float2 m_TexCoord_B0_;
4    float3 v_litColor1_cqa;
5  };
6
7  void vertex_main_ (out V2F v2f, varying in float4 iPosition_, varying in float2
       iTexCoord_, varying in float3 iNormal_, uniform in float4x4 iModelViewProj_,
       uniform in float3 iLightColor_, uniform in float3 iLightDirObj_, out float4
       ooutPosition_ : POSITION)
8  {
9    float3 v_ambient_ = float3 (0.4, 0.4, 0.4);
10   ooutPosition_ = mul (iModelViewProj_, iPosition_);
11   float i_tmp3_ = dot (iLightDirObj_, iNormal_);
12   float3 i_tmp4_ = float3 (i_tmp3_, i_tmp3_, i_tmp3_);
13   float3 v_litColor_ = iLightColor_ * i_tmp4_;
14   v2f.v_litColor1_cqa = v_litColor_ + v_ambient_;
15   v2f.m_TexCoord_B0_ = iTexCoord_;
16 }
17
18
19 void fragment_main_ (in V2F v2f, uniform in float4x4 iModelViewProj_, uniform in
       float3 iLightColor_, uniform in float3 iLightDirObj_, varying in sampler2D
       iTexture_, out float4 ooutColor_ : COLOR)
20 {
21   float4 m_outColor_B0_;
22   float4 i_tmp7_ = tex2D (iTexture_, v2f.m_TexCoord_B0_);
23   float3 v_surface_ = float3 (i_tmp7_.x, i_tmp7_.y, i_tmp7_.z);
24   float3 i_tmp12_ = v_surface_ * v2f.v_litColor1_cqa;
25   float3 i_tmp13_ = i_tmp12_;
26   float4 m_outColor_B01_dua = float4 (i_tmp13_.x, i_tmp13_.y, i_tmp13_.z,
         m_outColor_B0_.w);
27   ooutColor_ = float4 (m_outColor_B01_dua.x, m_outColor_B01_dua.y,
         m_outColor_B01_dua.z, 1);
28 }
```

Programm 3: Ausgabe des Compilers in Cg

Bewertung

Semantik: Sowohl das ursprüngliche Cg-Programm wie auch die Ausgabe des Compilers sind semantisch äquivalent.

In den Vertexprogrammen (`vertex_main` in Programm 1, `vertex_main_` in Programm 3) werden die Ausgabeposition (`output.Position` bzw. `ooutPosition_`) sowie der Beleuchtungswert (`output.litColor` bzw. `v_litColor1_cqa_`) berechnet. Die Zuweisung der Bildkoordinate des Musters (`TexCoord` bzw. `iTexCoord_`) an eine Ausgabe des Vertexprogramms (`output.TexCoord` bzw. `v2f.m_TexCoord_B0_`) wurde für das

Ausgabeprogramm vom Compiler automatisch generiert. Die Ausgabe des generierten Vertexprogramms (die Struktur `V2F`) entspricht der Ausgabe des ursprünglichen Programms 1 (dort die Struktur `VertexOutput`), allerdings wird die transformierte Vertex-Koordinate beim generierten Vertexprogramm (`vertex_main_` in Programm 3) als Ausgabeparameter `ooutPosition_` anstatt als Element der Struktur `V2F` ausgegeben.

Im ursprünglichen Pixelprogramm (`pixel_main` in Programm 1) wie auch generierten Pixelprogramm (`fragment_main_` in Programm 3) werden die gleichen Operationen vorgenommen: ein Auslesen eines Farbwertes aus dem Oberflächenmuster (`surface` bzw. `v_surface_`) und eine Modulierung der Oberflächenfarbe mit dem Beleuchtungswert (`outColor` bzw. `ooutColor_`).

Effizienz: Der generierte Cg-Code (Programm 3) enthält einige unnötige Zuweisungen. Diese werden von Eigenschaften der Zwischencoderepräsentation verursacht: Für eine “Swizzle”-Operation[2] werden mehrere Zwischencodeoperationen zum Extrahieren einzelner Komponenten generiert; mit diesen wird der Ergebnisvektor konstruiert. Zeile 17 im Eingabeprogramm 2 ist eine solche “Swizzle”-Operation, Zeile 23 in Programm 3 ist das Ergebnis der von dieser Operation verursachten Kombination von Zwischencodeoperationen.

Die Zuweisung von Teilkomponenten eines Vektors (Zeile 19 im Eingabeprogramm 2) trägt eine Schwäche des Compilers bei solchen Zuweisungen zu Tage: die unveränderten Komponenten des zu verändernden Vektors werden extrahiert und bei der Konstruktion des Ergebnisvektors benutzt. Grundsätzlich nötig, um den Wert der unveränderten Komponenten zu erhalten, führt dieses Verhalten bei Vektoren mit undefinierten Komponenten dazu, dass undefinierte Werte für den neuen Ergebnisvektor verwendet werden. Im Ausgabeprogramm 3, Zeile 26, ist der Wert `m_outColor_B0_.w` ein solcher uninitialisierter Wert.

Verwendung undefinierter Werte ist mindestens unnötig, aber auch störend: der Cg-Compiler mahnt dies an. Vermieden könnte diese Verwendung werden, in dem der Compiler oder Code-Generator verfolgt, welche Register „undefinierte“ Werte enthalten und mit Hilfe dieser Informationen den generierten Zwischencode bzw. Ausgabecode anpassen (z.B. durch Auslassung von Zuweisungen von undefinierten Werten oder ein

[2] Auswahl/Umordnung von Vektorkomponenten, siehe Abschnitt 3.3.4.3

Ersetzen von undefinierten Vektorkomponenten durch einen beliebigen Wert).

Die Funktionen des generierten Programms werden mit Parametern deklariert, die von den Funktionen selbst nicht verwendet werden (z.B. Parameter `iLightColor_` der Funktion `fragment_main_`). Dies ist ein Ergebnis der Aufspaltung: diese kopiert die Eingabeparameter einer aufzuspaltenden Eingabefunktionen einfach vollständig in alle aufgespalteten Ausgabefunktionen. Abhilfe würde ein Optimierungsschritt schaffen, der für jeden Parameter einer Funktion überprüft, ob dieser tatsächlich verwendet wird und entsprechend unbenutzte Parameter entfernt.

Praktisch führen die unnötigen Operationen allerdings nicht zu Nachteilen: der Cg-Compiler nimmt selbst Optimierungen auf zu übersetzenden Programmen vor; dadurch führt ein Übersetzen des Ursprungsprogramms sowie des generierten Programms mit dem Cg-Compiler zu identischen Ergebnisprogrammen.

5.2 Einfache Beleuchtung für jedes Pixel

Dieses Programm berechnet nur einen Beleuchungswert und verwendet diesen als Oberflächenfarbe. Im Gegensatz zum vorherigen Programm wird diese Beleuchtungsberechnung im Pixelprogramm vorgenommen. Weiterhin ist die Beleuchtungsberechnung selbst komplexer.

```
1  // This is C5E2v_fragmentLighting from "The Cg Tutorial" (Addison-Wesley, ISBN
2  // 0321194969) by Randima Fernando and Mark J. Kilgard.  See page 124.
3
4  void C5E2v_fragmentLighting(float4 position : POSITION,
5                              float3 normal   : NORMAL,
6
7                          out float4 oPosition : POSITION,
8                          out float3 objectPos : TEXCOORD0,
9                          out float3 oNormal   : TEXCOORD1,
10
11                     uniform float4x4 modelViewProj)
12 {
13   oPosition = mul(modelViewProj, position);
14   objectPos = position.xyz;
15   oNormal = normal;
16 }
```

Programm 4: Beleuchtungsberechnung im Pixelprogramm: Vertexprogramm

```
1  // This is C5E3f_basicLight from "The Cg Tutorial" (Addison-Wesley, ISBN
2  // 0321194969) by Randima Fernando and Mark J. Kilgard.  See page 125.
3
4  void C5E3f_basicLight(float4 position  : TEXCOORD0,
5                        float3 normal    : TEXCOORD1,
6
7                    out float4 color     : COLOR,
8
9                uniform float3 globalAmbient,
10               uniform float3 lightColor,
11               uniform float3 lightPosition,
12               uniform float3 eyePosition,
13               uniform float3 Ke,
14               uniform float3 Ka,
15               uniform float3 Kd,
16               uniform float3 Ks,
```

```
17                uniform float  shininess)
18 {
19   float3 P = position.xyz;
20   float3 N = normalize(normal);
21
22   // Compute emissive term
23   float3 emissive = Ke;
24
25   // Compute ambient term
26   float3 ambient = Ka * globalAmbient;
27
28   // Compute the diffuse term
29   float3 L = normalize(lightPosition - P);
30   float diffuseLight = max(dot(L, N), 0);
31   float3 diffuse = Kd * lightColor * diffuseLight;
32
33   // Compute the specular term
34   float3 V = normalize(eyePosition - P);
35   float3 H = normalize(L + V);
36   float specularLight = pow(max(dot(H, N), 0), shininess);
37   if (diffuseLight <= 0) specularLight = 0;
38   float3 specular = Ks * lightColor * specularLight;
39
40   color.xyz = emissive + ambient + diffuse + specular;
41   color.w = 1;
42 }
```

Programm 5: Beleuchtungsberechnung im Pixelprogramm: Pixelprogramm

Die Eingabe-Cg-Programme verwenden keine Struktur, um die Ausgaben des Vertexprogramms bzw. Eingaben des Pixelprogramms zu beschreiben, sondern eine explizite Zuordnung von Eingabe- bzw. Ausgabeparametern zu „Plätzen“ für interpolierte Werte – Schlüsselwörter wie `COLOR` oder `TEXCOORD0` sorgen dafür, dass eine interpolierte Ausgabe des Vertexprogramms dem Parameter des Pixelprogramms als Eingabe dient, der mit dem gleichen Schlüsselwort markiert wurde (z.B. ist die Vertexprogrammausgabe `objectPos` der Pixelprogrammeingabe `position` zugeordnet).

```
1  void main(in float4 position,
2            in float3 normal,
3
4            out float4 oPosition,
5            out float4 color,
6
7            float4x4 modelViewProj,
8            float3 globalAmbient,
9            float3 lightColor,
10           float3 lightPosition,
11           float3 eyePosition,
12           float3 Ke,
13           float3 Ka,
14           float3 Kd,
15           float3 Ks,
16           float  shininess)
17 {
18   oPosition = mul(modelViewProj, position);
19
20   float3 P = position.xyz;
21   float3 N = normalize(normal);
22
23   // Compute emissive term
24   float3 emissive = Ke;
25
26   // Compute ambient term
27   float3 ambient = Ka * globalAmbient;
28
29   // Compute the diffuse term
30   float3 L = normalize(lightPosition - P);
31   float diffuseLight = max(dot(L, N), 0);
32   float3 diffuse = Kd * lightColor * diffuseLight;
33
34   // Compute the specular term
```

```
35    float3 V = normalize(eyePosition - P);
36    float3 H = normalize(L + V);
37    float specularLight = pow(max(dot(H, N), 0), shininess);
38    if (diffuseLight <= 0) { specularLight = 0; }
39    float3 specular = Ks * lightColor * specularLight;
40
41    color.xyz = emissive + ambient + diffuse + specular;
42    color.w = 1;
43  }
```

Programm 6: Eingabeprogramm für Compiler in der Shadingsprache

```
1   struct V2F
2   {
3     float3 m_normal_B0_;
4     float3 i_tmp8_;
5     float3 i_tmp17_;
6   };
7
8   void vertex_main_ (out V2F v2f, varying in float4 iposition_, varying in float3
        inormal_, uniform in float4x4 imodelViewProj_, uniform in float3
        iglobalAmbient_, uniform in float3 ilightColor_, uniform in float3
        ilightPosition_, uniform in float3 ieyePosition_, uniform in float3 iKe_,
        uniform in float3 iKa_, uniform in float3 iKd_, uniform in float3 iKs_,
        uniform in float ishininess_, out float4 ooPosition_ : POSITION)
9   {
10    ooPosition_ = mul (imodelViewProj_, iposition_);
11    float3 v_P_ = float3 (iposition_.x, iposition_.y, iposition_.z);
12    v2f.i_tmp8_ = ilightPosition_ - v_P_;
13    v2f.i_tmp17_ = ieyePosition_ - v_P_;
14    v2f.m_normal_B0_ = inormal_;
15  }
16
17
18  void fragment_main_ (in V2F v2f, uniform in float4x4 imodelViewProj_, uniform in
        float3 iglobalAmbient_, uniform in float3 ilightColor_, uniform in float3
        ilightPosition_, uniform in float3 ieyePosition_, uniform in float3 iKe_,
        uniform in float3 iKa_, uniform in float3 iKd_, uniform in float3 iKs_,
        uniform in float ishininess_, out float4 ocolor_ : COLOR)
19  {
20    float3 v_emissive_ = iKe_;
21    float3 v_ambient_ = iKa_ * iglobalAmbient_;
22    float3 i_tmp14_ = iKd_ * ilightColor_;
23    float3 i_tmp29_ = iKs_ * ilightColor_;
24    float3 i_tmp32_ = v_emissive_ + v_ambient_;
25    float4 m_color_B0_;
26    float3 v_N_ = normalize (v2f.m_normal_B0_);
27    float3 v_L_ = normalize (v2f.i_tmp8_);
28    float i_tmp10_ = dot (v_L_, v_N_);
29    float v_diffuseLight_ = max (i_tmp10_, 0);
30    float3 i_tmp15_ = float3 (v_diffuseLight_, v_diffuseLight_, v_diffuseLight_);
31    float3 v_diffuse_ = i_tmp14_ * i_tmp15_;
32    float3 v_V_ = normalize (v2f.i_tmp17_);
33    float3 i_tmp19_ = v_L_ + v_V_;
34    float3 v_H_ = normalize (i_tmp19_);
35    float i_tmp21_ = dot (v_H_, v_N_);
36    float i_tmp24_ = max (i_tmp21_, 0);
37    float v_specularLight_ = pow (i_tmp24_, ishininess_);
38    bool c_tmp28_ = v_diffuseLight_ <= 0;
39    float v_specularLight1_1wa;
40    if (c_tmp28_)
41    {
42      v_specularLight1_1wa = 0;
43    }
44    else
45    {
46      v_specularLight1_1wa = v_specularLight_;
47    }
48    float3 i_tmp30_ = float3 (v_specularLight1_1wa, v_specularLight1_1wa,
          v_specularLight1_1wa);
49    float3 v_specular_ = i_tmp29_ * i_tmp30_;
50    float3 i_tmp33_ = i_tmp32_ + v_diffuse_;
51    float3 i_tmp34_ = i_tmp33_ + v_specular_;
52    float3 i_tmp35_ = i_tmp34_;
53    float4 m_color_B01_cqa = float4 (i_tmp35_.x, i_tmp35_.y, i_tmp35_.z, m_color_B0_
          .w);
54    ocolor_ = float4 (m_color_B01_cqa.x, m_color_B01_cqa.y, m_color_B01_cqa.z, 1);
55  }
```

Programm 7: Ausgabe des Compilers in Cg

Bewertung

Semantik: Das Vertexprogramm des generierten Cg-Programms (Funktion `vertex_main_` in Programm 7) ist ähnlich einfach wie das ursprüngliche Vertexprogramm 4. Allerdings unterscheiden sich die Ausgaben des Vertexprogramms: im ursprünglichen Programm werden die Werte `position` und `normal` „durchgeschleift“. Im generierten Programm wird dies mit zwar `inormal_` getan, statt dem Wert `iposition_` oder `v_P_` werden aber Zwischenergebnisse von Berechnungen ausgegeben. Die Ursache dafür ist die Arbeitsweise des Aufspalters: Zwischenwerte werden als Ausgabe des Vertexprogramms bzw. Eingabe des Pixelprogramms markiert, sobald eine nicht interpolierbare Operation mit dem Zwischenwert vorgenommen wird. Die Subtraktionen auf Zeilen 30 und 35 im Programm 6 sind interpolierbar, werden also in das Vertexprogramm ausgegeben. Die Funktion `normalize` allerdings nicht und muss in das Pixelprogramm ausgegeben werden – die Argumente von `normalize`, Werte `i_tmp7_` und `i_tmp16_`, werden also aus dem Vertexprogramm ausgegeben um als Eingabe des Pixelprogramms zu dienen.

Im ursprünglichen Programm 5 hat die Verzweigung nur einen `if`-Zweig; im generierten Programm 7 wurde ein `else`-Zweig hinzugefügt, da in der Zwischencoderepräsentation der Variable `specularLight` des Eingabeprogramms wegen der Zuweisung ein neues Register zugeordnet wurde, und der hinzugefügte `else`-Zweig sicherstellt, dass dieses neue Register den „alten“ Wert der Variable erhält, sollte die Bedingung für die Verzweigung nicht eingetreten sein.

Generell finden sich alle Operationen der ursprünglichen Programme 4 und 5 auch im Ausgabeprogramm 7, wenn auch, bedingt durch die Arbeitsweise des Compilers und Aufspalters, teilweise in anderer Reihenfolge und mit expliziten Zuweisungen von Zwischenwerten zu Variablen.

Effizienz: Wie im vorherigen Programm gibt es, aus den gleichen Gründen, wieder einige unnötige Zuweisungen, unbenutzte Funktionsparameter und eine Verwendung eines undefinierten Wertes.

Im Vergleich zu den ursprünglichen Programmen 4 und 5 wurden zwei Operationen aus dem Pixel- in das Vertexprogramm „verschoben“; vergleicht man die Übersetzungsergebnisse des Cg-Compilers von ursprünglichem und generiertem

Programm so enthält das Ergebnis-Vertexprogramm eine zusätzliche Operation, das Ergebnis-Pixelprogramm allerdings zwei Operationen weniger – das vom Compiler generierte Programm ist also leicht effizienter.

5.3 Einfache Beleuchtung für jedes Vertex

Diese Programm verwendet die gleiche Beleuchtsberechnung wie im vorherigen Programm; allerdings soll die Berechnung, wie im ersten Beispielprogramm, für jedes Vertex berechnet werden und über die Ausgabepixel interpoliert werden.

```
1  // This is C5E1v_basicLight from "The Cg Tutorial" (Addison-Wesley, ISBN
2  // 0321194969) by Randima Fernando and Mark J. Kilgard.  See page 111.
3
4  void C5E1v_basicLight(float4 position  : POSITION,
5                        float3 normal    : NORMAL,
6
7                    out float4 oPosition : POSITION,
8                    out float4 color     : COLOR,
9
10                uniform float4x4 modelViewProj,
11                uniform float3 globalAmbient,
12                uniform float3 lightColor,
13                uniform float3 lightPosition,
14                uniform float3 eyePosition,
15                uniform float3 Ke,
16                uniform float3 Ka,
17                uniform float3 Kd,
18                uniform float3 Ks,
19                uniform float  shininess)
20 {
21   oPosition = mul(modelViewProj, position);
22
23   float3 P = position.xyz;
24   float3 N = normal;
25
26   // Compute emissive term
27   float3 emissive = Ke;
28
29   // Compute ambient term
30   float3 ambient = Ka * globalAmbient;
31
32   // Compute the diffuse term
33   float3 L = normalize(lightPosition - P);
34   float diffuseLight = max(dot(N, L), 0);
35   float3 diffuse = Kd * lightColor * diffuseLight;
36
37   // Compute the specular term
38   float3 V = normalize(eyePosition - P);
39   float3 H = normalize(L + V);
40   float specularLight = pow(max(dot(N, H), 0), shininess);
41   if (diffuseLight <= 0) specularLight = 0;
42   float3 specular = Ks * lightColor * specularLight;
43
44   color.xyz = emissive + ambient + diffuse + specular;
45   color.w = 1;
46 }
```

Programm 8: Beleuchtungsberechnung für jedes Vertex: Vertexprogramm

```
1  // This is C2E2f_passthru from "The Cg Tutorial" (Addison-Wesley, ISBN
2  // 0321194969) by Randima Fernando and Mark J. Kilgard.  See page 53.
3
4  struct C2E2f_Output {
5    float4 color : COLOR;
6  };
7
8  C2E2f_Output C2E2f_passthru(float4 color : COLOR)
9  {
10   C2E2f_Output OUT;
11   OUT.color = color;
12   return OUT;
13 }
```

Programm 9: Beleuchtungsberechnung für jedes Vertex: Pixelprogramm

Die Berechnungen zur Beleuchtung stimmen mit denen aus Programm 9 überein, finden sich aber hier im Vertexprogramm. Das Pixelprogramm gibt nur den Beleuchtungswert aus.

```
void main (float4 position,
           float3 normal,

           out float4 oPosition,
           out float4 color,

           float4x4 modelViewProj,
           float3 globalAmbient,
           float3 lightColor,
           float3 lightPosition,
           float3 eyePosition,
           float3 Ke,
           float3 Ka,
           float3 Kd,
           float3 Ks,
           float  shininess)
{
  oPosition = mul(modelViewProj, position);

  float3 P = position.xyz;
  float3 N = normal;

  // Compute emissive term
  float3 emissive = Ke;

  // Compute ambient term
  float3 ambient = Ka * globalAmbient;

  // Compute the diffuse term
  float3 L = normalize(lightPosition - P);
  float diffuseLight = max(dot(N, L), 0);
  float3 diffuse = Kd * lightColor * diffuseLight;

  // Compute the specular term
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(N, H), 0), shininess);
  if (diffuseLight <= 0) { specularLight = 0; }
  float3 specular = Ks * lightColor * specularLight;

  color.xyz = emissive + ambient + diffuse + specular;
  color.w = 1;
}
```

Programm 10: Eingabeprogramm für Compiler in der Shadingsprache

Dieses Programm ist mit dem Programm 6 praktisch identisch.

```
struct V2F
{
  float3 i_tmp7_;
  float3 v_N_;
  float3 i_tmp16_;
};

void vertex_main_ (out V2F v2f, varying in float4 iposition_, varying in float3
    inormal_, uniform in float4x4 imodelViewProj_, uniform in float3
    iglobalAmbient_, uniform in float3 ilightColor_, uniform in float3
    ilightPosition_, uniform in float3 ieyePosition_, uniform in float3 iKe_,
    uniform in float3 iKa_, uniform in float3 iKd_, uniform in float3 iKs_,
    uniform in float ishininess_, out float4 ooPosition_ : POSITION)
{
  ooPosition_ = mul (imodelViewProj_, iposition_);
  float3 v_P_ = float3 (iposition_.x, iposition_.y, iposition_.z);
  v2f.v_N_ = inormal_;
  v2f.i_tmp7_ = ilightPosition_ - v_P_;
  v2f.i_tmp16_ = ieyePosition_ - v_P_;
}
```

```
18 void fragment_main_ (in V2F v2f, uniform in float4x4 imodelViewProj_, uniform in
       float3 iglobalAmbient_, uniform in float3 ilightColor_, uniform in float3
       ilightPosition_, uniform in float3 ieyePosition_, uniform in float3 iKe_,
       uniform in float3 iKa_, uniform in float3 iKd_, uniform in float3 iKs_,
       uniform in float ishininess_, out float4 ocolor_ : COLOR)
19 {
20   float3 v_emissive_ = iKe_;
21   float3 v_ambient_ = iKa_ * iglobalAmbient_;
22   float3 i_tmp13_ = iKd_ * ilightColor_;
23   float3 i_tmp28_ = iKs_ * ilightColor_;
24   float3 i_tmp31_ = v_emissive_ + v_ambient_;
25   float4 m_color_B0_;
26   float3 v_L_ = normalize (v2f.i_tmp7_);
27   float i_tmp9_ = dot (v2f.v_N_, v_L_);
28   float v_diffuseLight_ = max (i_tmp9_, 0);
29   float3 i_tmp14_ = float3 (v_diffuseLight_, v_diffuseLight_, v_diffuseLight_);
30   float3 v_diffuse_ = i_tmp13_ * i_tmp14_;
31   float3 v_V_ = normalize (v2f.i_tmp16_);
32   float3 i_tmp18_ = v_L_ + v_V_;
33   float3 v_H_ = normalize (i_tmp18_);
34   float i_tmp20_ = dot (v2f.v_N_, v_H_);
35   float i_tmp23_ = max (i_tmp20_, 0);
36   float v_specularLight_ = pow (i_tmp23_, ishininess_);
37   bool c_tmp27_ = v_diffuseLight_ <= 0;
38   float v_specularLight1_1wa;
39   if (c_tmp27_)
40   {
41     v_specularLight1_1wa = 0;
42   }
43   else
44   {
45     v_specularLight1_1wa = v_specularLight_;
46   }
47   float3 i_tmp29_ = float3 (v_specularLight1_1wa, v_specularLight1_1wa,
         v_specularLight1_1wa);
48   float3 v_specular_ = i_tmp28_ * i_tmp29_;
49   float3 i_tmp32_ = i_tmp31_ + v_diffuse_;
50   float3 i_tmp33_ = i_tmp32_ + v_specular_;
51   float3 i_tmp34_ = i_tmp33_;
52   float4 m_color_B01_cqa = float4 (i_tmp34_.x, i_tmp34_.y, i_tmp34_.z, m_color_B0_
         .w);
53   ocolor_ = float4 (m_color_B01_cqa.x, m_color_B01_cqa.y, m_color_B01_cqa.z, 1);
54 }
```

Programm 11: Ausgabe des Compilers in Cg

Bewertung

Semantik: Im generierten Cg-Programm 11 findet sich die Beleuchtungsberechnung im Pixelprogramm, statt im Vertexprogramm wie es in den ursprünglichen Programmen 8 und 9 der Fall ist.

Verantwortlich dafür ist die Arbeitsweise des „Splitters“. Wie in Abschnitt 4.6.2 beschrieben, werden die Eingabeprogramme so betrachtet, als würden sie Pixelberechnungen formulieren. Nur unter gewissen Bedingungen („Interpolierbarkeit“, siehe Abschnitt 4.6.6) werden Operationen in die Vertexberechnung verschoben. Diese Bedingungen sind im Eingabeprogramm nicht erfüllt – der Compiler muss also die Mehrzahl der Operationen in die Pixelberechnungen ausgeben.

Um ein generiertes Programm zu erhalten, dass mit dem ursprünglichen Programmen übereinstimmt, wäre im Compiler ein Mechanismus (Schlüsselwort o.ä.) nötig, um explizit anzugeben, dass eine Programmoperation in der Vertex- statt in der

Pixelberechnung ausgeführt werden soll. Der Splitter müsste entsprechend angepasst werden, um für solche markierten Operationen die „Interpolierbarkeit“ als gegeben anzunehmen.

Abgesehen davon, dass die Beleuchtungsberechnung vom Splitter in das „falsche“ Teilprogramm ausgegeben wurde, stimmen die Operation selbst mit denen der ursprünglichen Programme überein.

Effizienz: Das Eingabeprogramm 10 entspricht genau dem vorherigem Eingabeprogramm, dementsprechend treffen die dort gemachten Effizienzbetrachtungen auch hier zu.

5.4 Beleuchtung unter Verwendung zweier Lichtquellen

Dieses Programm verwendet die gleichen Beleuchtungsberechnung wie die vorherigen zwei Programmen, allerdings für zwei Lichtquellen.

```
1  // This is C5E4v_twoLights from "The Cg Tutorial" (Addison-Wesley, ISBN
2  // 0321194969) by Randima Fernando and Mark J. Kilgard.  See pages 128-132.
3
4  // From page 128
5  struct Material {
6    float3 Ke;
7    float3 Ka;
8    float3 Kd;
9    float3 Ks;
10   float shininess;
11 };
12
13 // From page 129
14 struct Light {
15   float3 position;
16   float3 color;
17 };
18
19 // From page 131
20 void C5E5_computeLighting(Light  light,
21                           float3 P,
22                           float3 N,
23                           float3 eyePosition,
24                           float  shininess,
25
26                       out float3 diffuseResult,
27                       out float3 specularResult)
28 {
29   // Compute the diffuse lighting
30   float3 L = normalize(light.position - P);
31   float diffuseLight = max(dot(N, L), 0);
32   diffuseResult = light.color * diffuseLight;
33
34   // Compute the specular lighting
35   float3 V = normalize(eyePosition - P);
36   float3 H = normalize(L + V);
37   float specularLight = pow(max(dot(N, H), 0),
38                             shininess);
39   if (diffuseLight <= 0) specularLight = 0;
40   specularResult = light.color * specularLight;
41 }
42
43 // From page 132 (corrected)
44 void C5E4v_twoLights(float4 position : POSITION,
45                      float3 normal   : NORMAL,
46
```

```
47                  out float4 oPosition : POSITION,
48                  out float4 color     : COLOR,
49
50              uniform float4x4 modelViewProj,
51              uniform float3   eyePosition,
52              uniform float3   globalAmbient,
53              uniform Light    lights[2],
54              uniform Material material)
55  {
56    oPosition = mul(modelViewProj, position);
57
58    // Calculate emissive and ambient terms
59    float3 emissive = material.Ke;
60    float3 ambient = material.Ka * globalAmbient;
61
62    // Loop over diffuse and specular contributions for each light
63    float3 diffuseLight;
64    float3 specularLight;
65    float3 diffuseSum  = 0;
66    float3 specularSum = 0;
67    for (int i = 0; i < 2; i++) {
68      C5E5_computeLighting(lights[i], position.xyz, normal,
69                           eyePosition, material.shininess,
70                           diffuseLight, specularLight);
71      diffuseSum += diffuseLight;
72      specularSum += specularLight;
73    }
74
75    // Now modulate diffuse and specular by material color
76    float3 diffuse = material.Kd * diffuseSum;
77    float3 specular = material.Ks * specularSum;
78
79    color.xyz = emissive + ambient + diffuse + specular;
80    color.w = 1;
81  }
```

Programm 12: Beleuchtung mit zwei Lichtquellen: Vertexprogramm

```
1  // This is C2E2f_passthru from "The Cg Tutorial" (Addison-Wesley, ISBN
2  // 0321194969) by Randima Fernando and Mark J. Kilgard.  See page 53.
3
4  struct C2E2f_Output {
5    float4 color : COLOR;
6  };
7
8  C2E2f_Output C2E2f_passthru(float4 color : COLOR)
9  {
10   C2E2f_Output OUT;
11   OUT.color = color;
12   return OUT;
13 }
```

Programm 13: Beleuchtung mit zwei Lichtquellen: Pixelprogramm

Die Beleuchtungsberechnung wurde in eine Funktion verschoben. Die Parameter der Lichtquellen werden als Arrays übergeben.

```
1  void C5E5_computeLighting(float3 light_position,
2                            float3 light_color,
3                            float3 P,
4                            float3 N,
5                            float3 eyePosition,
6                            float  shininess,
7
8                            out float3 diffuseResult,
9                            out float3 specularResult)
10 {
11   // Compute the diffuse lighting
12   float3 L = normalize(light_position - P);
13   float diffuseLight = max(dot(N, L), 0);
14   diffuseResult = light_color * diffuseLight;
15
16   // Compute the specular lighting
17   float3 V = normalize(eyePosition - P);
18   float3 H = normalize(L + V);
19   float specularLight = pow(max(dot(N, H), 0),
20                             shininess);
21   if (diffuseLight <= 0) { specularLight = 0; }
```

```
22    specularResult = light_color * specularLight;
23  }
24
25  void main(float4 position,
26            float3 normal,
27
28            out float4 oPosition,
29            out float4 color,
30
31            float4x4 modelViewProj,
32            float3   eyePosition,
33            float3   globalAmbient,
34            float3[] lights_position,
35            float3[] lights_color,
36            float3 material_Ke,
37            float3 material_Ka,
38            float3 material_Kd,
39            float3 material_Ks,
40            float material_shininess)
41  {
42    oPosition = mul(modelViewProj, position);
43
44    // Calculate emissive and ambient terms
45    float3 emissive = material_Ke;
46    float3 ambient = material_Ka * globalAmbient;
47
48    // Loop over diffuse and specular contributions for each light
49    float3 diffuseLight;
50    float3 specularLight;
51    float3 diffuseSum  = 0;
52    float3 specularSum = 0;
53    int i;
54    for (i = 0; i < 2; i = i + 1) {
55      C5E5_computeLighting(lights_position[i], lights_color[i],
56                           position.xyz, normal,
57                           eyePosition, material_shininess,
58                           diffuseLight, specularLight);
59      diffuseSum = diffuseSum + diffuseLight;
60      specularSum = specularSum + specularLight;
61    }
62
63    // Now modulate diffuse and specular by material color
64    float3 diffuse = material_Kd * diffuseSum;
65    float3 specular = material_Ks * specularSum;
66
67    color.xyz = emissive + ambient + diffuse + specular;
68    color.w = 1;
69  }
```

Programm 14: Eingabeprogramm für Compiler in der Shadingsprache

Die die Shadingsprache keine Verbundtypen kennt mussten diese in einzelne Variablen „ausgepackt“ werden. Die Größe der Eingabearrays wurde als Compiler-Option spezifiziert (hier nicht sichtbar).

```
1   struct V2F
2   {
3     float4 m_position_B0_;
4     float3 m_normal_B0_;
5   };
6
7   void vertex_C5E5_computeLighting1vF31vF31vF31vF31vF31F2vF32vF3002200_v5b4a (in
        float3 ilight_position_, in float3 ilight_color_, in float3 iP_, in float3 iN_
        , in float3 ieyePosition_, in float ishininess_, out float3 odiffuseResult_,
        out float3 ospecularResult_)
8   {
9   }
10
11  void vertex_main_ (out V2F v2f, varying in float4 iposition_, varying in float3
        inormal_, uniform in float4x4 imodelViewProj_, uniform in float3 ieyePosition_
        , uniform in float3 iglobalAmbient_, uniform in float3 ilights_position_[2],
        uniform in float3 ilights_color_[2], uniform in float3 imaterial_Ke_, uniform
        in float3 imaterial_Ka_, uniform in float3 imaterial_Kd_, uniform in float3
        imaterial_Ks_, uniform in float imaterial_shininess_, out float4 ooPosition_ :
         POSITION)
12  {
13    ooPosition_ = mul (imodelViewProj_, iposition_);
14    v2f.m_position_B0_ = iposition_;
```

```
15    v2f.m_normal_B0_ = inormal_;
16  }
17
18
19  void fragment_C5E5_computeLighting1vF31vF31vF31vF31vF31F2vF32vF3002200_z7b4a (in
        float3 ilight_position_, in float3 ilight_color_, in float3 iP_, in float3 iN_
        , in float3 ieyePosition_, in float ishininess_, out float3 odiffuseResult_,
        out float3 ospecularResult_)
20  {
21    float3 i_tmp0_ = ilight_position_ - iP_;
22    float3 v_L_ = normalize (i_tmp0_);
23    float i_tmp2_ = dot (iN_, v_L_);
24    float v_diffuseLight_ = max (i_tmp2_, 0);
25    float3 i_tmp6_ = float3 (v_diffuseLight_, v_diffuseLight_, v_diffuseLight_);
26    odiffuseResult_ = ilight_color_ * i_tmp6_;
27    float3 i_tmp9_ = ieyePosition_ - iP_;
28    float3 v_V_ = normalize (i_tmp9_);
29    float3 i_tmp11_ = v_L_ + v_V_;
30    float3 v_H_ = normalize (i_tmp11_);
31    float i_tmp13_ = dot (iN_, v_H_);
32    float i_tmp16_ = max (i_tmp13_, 0);
33    float v_specularLight_ = pow (i_tmp16_, ishininess_);
34    bool c_tmp20_ = v_diffuseLight_ <= 0;
35    float v_specularLight1_1wa;
36    if (c_tmp20_)
37    {
38      v_specularLight1_1wa = 0;
39    }
40    else
41    {
42      v_specularLight1_1wa = v_specularLight_;
43    }
44    float3 i_tmp21_ = float3 (v_specularLight1_1wa, v_specularLight1_1wa,
          v_specularLight1_1wa);
45    ospecularResult_ = ilight_color_ * i_tmp21_;
46  }
47
48  void fragment_main_ (in V2F v2f, uniform in float4x4 imodelViewProj_, uniform in
        float3 ieyePosition_, uniform in float3 iglobalAmbient_, uniform in float3
        ilights_position_[2], uniform in float3 ilights_color_[2], uniform in float3
        imaterial_Ke_, uniform in float3 imaterial_Ka_, uniform in float3
        imaterial_Kd_, uniform in float3 imaterial_Ks_, uniform in float
        imaterial_shininess_, out float4 ocolor_ : COLOR)
49  {
50    float3 v_emissive_ = imaterial_Ke_;
51    float3 v_ambient_ = imaterial_Ka_ * iglobalAmbient_;
52    float3 v_diffuseSum_ = float3 (0, 0, 0);
53    float3 v_specularSum_ = float3 (0, 0, 0);
54    float3 i_tmp51_ = v_emissive_ + v_ambient_;
55    float4 m_color_B0_;
56    float3 v_diffuseSum1_0sa = v_diffuseSum_;
57    float3 v_specularSum1_dua = v_specularSum_;
58    int v_i1_xga = 0;
59    bool v_cond11_lja9h = true;
60    float3 v_diffuseLight_;
61    float3 v_specularLight_;
62    while (v_cond11_lja9h)
63    {
64      unsigned int x_tmp25_ = unsigned int (v_i1_xga);
65      float3 i_tmp24_ = ilights_position_[x_tmp25_];
66      unsigned int x_tmp27_ = unsigned int (v_i1_xga);
67      float3 i_tmp26_ = ilights_color_[x_tmp27_];
68      v_i1_xga = v_i1_xga + 1;
69      v_cond11_lja9h = v_i1_xga < 2;
70      float3 i_tmp28_ = float3 (v2f.m_position_B0_.x, v2f.m_position_B0_.y, v2f.
            m_position_B0_.z);
71      fragment_C5E5_computeLighting1vF31vF31vF31vF31vF31F2vF32vF3002200_z7b4a (
            i_tmp24_, i_tmp26_, i_tmp28_, v2f.m_normal_B0_, ieyePosition_,
            imaterial_shininess_, v_diffuseLight_, v_specularLight_);
72      v_diffuseSum1_0sa = v_diffuseSum1_0sa + v_diffuseLight_;
73      v_specularSum1_dua = v_specularSum1_dua + v_specularLight_;
74    }
75    float3 v_diffuse_ = imaterial_Kd_ * v_diffuseSum1_0sa;
76    float3 v_specular_ = imaterial_Ks_ * v_specularSum1_dua;
77    float3 i_tmp52_ = i_tmp51_ + v_diffuse_;
78    float3 i_tmp53_ = i_tmp52_ + v_specular_;
79    float3 i_tmp54_ = i_tmp53_;
80    float4 m_color_B01_cqa = float4 (i_tmp54_.x, i_tmp54_.y, i_tmp54_.z, m_color_B0_
          .w);
81    ocolor_ = float4 (m_color_B01_cqa.x, m_color_B01_cqa.y, m_color_B01_cqa.z, 1);
82  }
```

Programm 15: Ausgabe des Compilers in Cg

Bewertung

Semantik: Auch in diesem Programm wird, wie im vorherigen Programm, der Großteil der Berechnungen in das Pixelprogramm ausgegeben; Grund ist auch die Arbeitsweise des Compilers.

Die Funktion `C5E5_computeLighting` des ursprünglichen Programms 12 (Zeilen 20 bis 41) findet ihre Entsprechung im generierten Programm 15 in der Funktion der Zeilen 53 bis 80. Diese Funktionen führen äquivalente Berechnungen durch, der „Kern“ von ursprünglichem und generiertem Programm stimmen also überein.

Die Iterationen über die Lichtquellen sind im generierten Programm 15 (Zeilen 92 bis 108) als `while`-Schleife ausgeführt, es ist aber einfach erkennbar, dass diese genauso oft und mit den gleichen Werten für die Schleifenvariable `i` durchlaufen wird wie es im ursprünglichen Programm 12 (Zeilen 67 bis 73) der Fall ist.

Effizienz: Auffällig im generierten Programm 15 ist die leere, nicht verwendete Funktion der Zeilen 7 bis 9: hier hat der Splitter bei der Aufspaltung der Funktion `C5E5_computeLighting` des ursprünglichen Programms 12 eine Variante der Funktion für die Verwendung bei der Vertexberechnung (erkennbar am Präfix „`vertex_`“ des Funktionsbezeichners) ausgegeben. Diese wurde aber letztendlich nicht verwendet: Grund dafür ist, dass die ursprüngliche Funktion in der Schleife verwendet wird; Schleifen werden aber, wie in Abschnitt 4.6.7 beschrieben, nie in die Vertexberechnung ausgegeben – damit wird auch die Variante der Funktion für die Vertexberechnung nicht verwendet.

Solche unnötigen Funktionen könnten durch einen zusätzlichen Optimierungsschritt verhindert werden, der untersucht, welche Funktionen eines Programms bzw. eines Teilprogramms tatsächlich verwendet werden und nicht verwendete Funktionen verwirft.

Praktisch, nach einer Übersetzung des generierten Programms mit dem Cg-Compiler, folgen aus dieser „überflüssigen“ Funktion keine Nachteile: der Cg-Compiler entfernt die nicht verwendete Funktion.

Wie beim vorherigen Beispielprogramm finden sich auch hier die Berechnungen

im generierten Programm 15 im Teilprogramm zur Pixelberechnung statt, wie im ursprünglichen Programm 12, in der Vertexberechnung. Grund ist auch hier die Arbeitsweise des Compilers. Damit ist auch die „Lösung“ dieses Problems die Gleiche.

5.5 Zusammenfassung

Der Vergleich von „realistischen“ Cg-Programmen mit der Ausgabe des Compilers, nachdem die Programme von Cg in die Shadingsprache übersetzt wurden, zeigt, dass die Übersetzung prinzipiell semantisch übereinstimmende Programme erzeugen kann.

Eine Schwäche des Compilers ist dessen Korrektheit – in dem Sinne, das Operationen nur für die Vertexberechnung ausgegeben werden, wenn eine Interpolation der Ergebnisse der Vertexberechnung die gleichen Ergebnisse wie eine Ausführung in der Pixelberechnung hätte. Damit wurden im direkten Vergleich bei einigen Programmen Operationen, die ursprünglich in der Vertexberechnung ausgeführt wurden, in die Pixelberechnungen „verschoben“.

Weiterhin gibt es weiteren Bedarf für Optimierungen, von einer Entfernung nicht verwendeter Funktion über die Vermeidung unnötiger Zuweisungen hin zu einer Entfernung nicht verwendeter Funktionsparameter. Zwar hat eine Ausgabe in die Sprache Cg den Vorteil, dass der Cg-Compiler selbst noch Optimierungen vornimmt und damit die fehlenden Optimierungen des Shadingsprachen-Compilers dadurch wenig ins Gewicht fallen, bei anderen Zielsprachen muss dies aber nicht der Fall sein – weitergehende Optimierungen wären in solchen Fällen vorteilhaft.

Kapitel 6

Ausblick

Die übersetzte Sprache ist eine zwar vergleichsweise einfache, aber grundsätzlich praktisch nutzbare Shadingsprache, die es erlaubt, Shadingprogramme zu schreiben, ohne den Hardwareaufbau aus verschiedenen Funktionseinheiten berücksichtigen zu müssen.

Die vom Compiler ausgegebenen Programme können grundsätzlich praktisch eingesetzt werden. Die Evaluation hat jedoch gezeigt, das der Compiler-„Prototyp“ einigen Anwendungsfällen nicht gerecht wird: der in Abschnitt 4.6 beschriebene „Splitter“ arbeitet konservativ in dem Sinne, dass eine Operation nur in das Ausgabe-Vertexprogramm „verschoben“ wird, wenn das Ergebnis mathematisch äquivalent zu einer Berechnung im Ausgabe-Pixelprogramm wäre. Praktisch sollen aber manchmal bewusst Ergebnisse approximiert werden, in dem Operationen während der Vertexberechnung ausgeführt werden (aus Geschwindigkeitsgründen bei komplexen oder oftmals verwendeten Programmen). Die Sprache sollte deswegen noch um ein Schlüsselwort o.ä. erweitert werden welches ein manuelles Bestimmen der Berechnungseinheit für eine Operation erlaubt.

Viele Programmiersprachen unterstützen Verbundtypen. Dies wäre eine sinnvolle Spracherweiterung um das Erstellen von komplexeren Programmen zu vereinfachen. Auch praktisch nützlich wären die von anderen Shadingsprachen wie Cg, GLSL usw. meist bereitgestellten Typen mit unterschiedlichen Genauigkeiten (wie “half”- oder “double precision”-Fließkommazahlen). Diese lassen den Programmierer, je nach Anforderungen, zwischen Genauigkeit und Berechnungsgeschwindigkeit abwägen.

Auch bieten sie anderen Shadingsprachen noch weitere „Komfortfunktionen“ für oft verwendete Berechnungen (z.B. für die lineare Interpolation von Werten oder

Vektoren). Diese können zwar prinzipiell mit den vorhandenen Operationen und eingebauten Funktionen nachgebildet werden. Ein zur Verfügung stellen als eingebaute Funktion wäre jedoch für Benutzer der Sprache komfortabler.

Denkbar ist es auch, den Compiler so zu erweitern, dass Programme aus „üblichen“ Shadingsprachen mit getrennten Vertex- und Pixelprogrammen – wie Cg – akzeptiert werden. Die Ausgabe des Compilers wären funktional äquivalente Vertex- und Pixelprogramme, allerdings mit einer potenziell besseren Verteilung von Operationen auf Verarbeitungseinheiten.

Bei der Implementierung besteht noch Verbesserungspotential im Aufspalter. Vor allem rekursive Funktionen werden nur im Pixelprogramm ausgeführt - dies stellt zwar ein korrektes Ergebnis sicher, ist aber nicht in allen Fällen die optimale Lösung.

Zur weiteren praktischen Verwendung des Compilers ist eine Integration in eine “3D-Engine“ (also Bibliothek für 3D-Visualisierungen), wie es einige unter Open-Source-Lizenzen gibt, anstrebenswert.

Anhang A

A.1 Vom Parser verwendete Grammatik-Regeln

Die in Kapitel 3 angegebene Grammatik enthält Mehrdeutigkeiten und eignet sich in dieser Form nicht direkt zur Implementierung; sie wurde vor allem auf Verständlichkeit hin ausgerichtet.

Der im Compiler implementierte Parser basiert auf Varianten der Regeln, die um Mehrdeutigkeiten bereinigt wurden. Bereinigte Versionen der Regeln, die damit von der Grammatik in Kapitel 3 abweichen, sind hier aufgeführt.

A.1.1 Ausdrücke

ausdruck:

asdr_logisch_oder(3.3.3.4) [(*asdr_suffix_ternaer*(A.1.1.2) | *asdr_suffix_zuweisung*(A.1.1.1))]

Der Ausdruck höchster Präzedenz ist die Zuweisung.

A.1.1.1 Zuweisung

asdr_suffix_zuweisung:

'=' *ausdruck*(3.3.3)

Ein Zuweisungsausdruck setzt sich aus einem *ausdruck*(3.3.3) (der „linken Seite“) sowie einem direkt folgenden *asdr_suffix_zuweisung*(A.1.1.1) (die „rechte Seite“) zusammen. Dem Ausdruck der linken Seite wird der Wert des Ausdrucks auf der

rechten Seite zugewiesen. Der Ausdruck auf der linken Seite muss eine Variable oder ein Arrayelement bzw. von diesen ein Swizzle-Attribut (siehe 3.3.4.3) bezeichnen.

Der Typ des zugewiesenen Ausdrucks muss zuweisungskompatibel (siehe unten) zur linken Seite der Zuweisung sein.

Ein Zuweisungsausdruck selbst hat den Wert der linken Seite nach der Zuweisung.

Anfang des Zuweisungsausdrucks siehe ausdruck*(A.1.1)*.

A.1.1.2 Ternärer Ausdruck

asdr_suffix_ternaer:

‘`?`’ *ausdruck*(3.3.3) ‘`:`’ *ausdruck*(3.3.3)

Ein ternärer Ausdruck setzt sich aus einem *ausdruck*(3.3.3) (der „Bedingung“) sowie einem direkt folgenden *asdr_suffix_ternaer*(A.1.1.2) zusammen. Die *Bedingung* muss ein boolescher Ausdruck sein. Ergibt sich dieser Ausdruck zu `true`, so wird der *Wahr-Ausdruck* (dem `?` folgend) ausgewertet, und der Wert des ternären Ausdrucks ergibt sich zu dem Wert des *Wahr-Ausdrucks*. Ergibt sich die *Bedingung* zu `false`, so wird der *Falsch-Ausdruck* (dem `:` folgend) ausgewertet, und der Wert des ternären Ausdrucks ergibt sich zu dem Wert des *Falsch-Ausdrucks*.

Wahr- und *Falsch-Ausdruck* müssen vom gleichen Typ sein.

Anfang des ternären Ausdrucks siehe ausdruck*(A.1.1)*.

A.1.2 Typen

typ:

typ_basis(3.3.4) [*typ_suffix_array*(A.1.2.1)]

A.1.2.1 Arraytypen

typ_suffix_array:

‘`[`’ ‘`]`’ [*typ_suffix_array*(A.1.2.1)]

Glossar

Dreiecksnetz 3D-Modell, Menge von Eckpunkten (Vertices) und Dreiecken. 4

GPU “Graphics Processing Unit”, auf Darstellung von 3D-Grafik spezialisierte Prozessoren. 1

Interpolation Arbeitsschritt der GPU. Interpoliert zwischen Ausgaben der Vertexberechnungen, Interpolationsergebnisse dienen als Eingaben für Pixelberechnungen. 9

Objektattribut Einem 3D-Modell zugeordnete Eigenschaften (z.B. Transformationen, Material). 11

Pixelberechnungen Arbeitsschritt der GPU. Berechnungen, die für jedes gerasterte Pixel eines Dreiecks vorgenommen werden. 10

Rasterung Arbeitsschritt der GPU. Bestimmt Monitorpixel, die von einem Dreieck überdeckt werden. 8

Shading Berechnung der Farbe eines beleuchteten Punktes auf einer 3D-Oberfläche. 6

Shadingsprache Sprachen zur Programmierung von Eckpunkt- und Pixelberechnungen auf GPUs. 11

Textur Bilddaten, bei der Pixelverarbeitung ausgelesen. Typischerweise auf Oberfläche von Modellen „gespannt“. 7

Transformation Auf Raumkoordinaten von 3D-Modellen angewendete Abbildungen. Ermöglichen Verschiebungen, Rotation, Skalierung von Modellen. 5

Vertex Eckpunkt eines Dreiecksnetzes. Mehrzahl: Vertices. 4

Vertexattribut Beschreibt weitere Daten, die jedem Vertex eines 3D-Modells zugeordnet sind. Attributtypen sind Vektoren mit ein bis vier Komponenten. 4

Vertexattributwert Werte von Vertexattributen. Jedem Vertex ist für jedes Attribut ein Wert zugeordnet. 5

Vertexberechnungen Arbeitsschritt der GPU. Berechnungen, die für jedes Vertex des dargestellten 3D-Modells vorgenommen werden. 8

Literaturverzeichnis

[Amm03] Amme, Wolfram. *Effiziente und sichere Codegenerierung für mobilen Code*. Habilitationsschrift, Friedrich-Schiller-Universität Jena, 2003.

[AWZ88] Alpern, B., Wegman, M. N. und Zadeck, F. K. *Detecting equality of variables in programs*. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 1–11. ACM, New York, NY, USA, 1988. doi:http://doi.acm.org/10.1145/73560.73561.

[D3D10] Microsoft Corporation. *Direct3D 10 Graphics*, August 2009. URL `http://msdn.microsoft.com/en-us/library/ee415646(v=VS.85).aspx`, abgerufen 14.02.2010.

[HDE+93] Hendren, Laurie J., Donawa, C., Emami, Maryam, Gao, Guang R., Justiani und Sridharan, B. *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations*. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, Seiten 406–420. Springer-Verlag, London, UK, 1993. URL `http://portal.acm.org/citation.cfm?id=645670.665360`.

[Koe85] Koecher, Max. *Lineare Algebra und analytische Geometrie*. Springer-Verlag, Berlin, 2. Auflage, 1985.

[LA04] Lattner, Chris und Adve, Vikram. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar 2004.

[MGAK03] Mark, William R., Glanville, R. Steven, Akeley, Kurt und Kilgard, Mark J. *Cg: a system for programming graphics hardware in a C-like language*. In

SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, Seiten 896–907. ACM, New York, NY, USA, 2003. doi:http://doi.acm.org/10.1145/1201775.882362. URL http://www.cs.utexas.edu/users/billmark/papers/Cg/cgpaper.pdf.

[NVI10] NVIDIA Corporation. *Cg Toolkit - GPU Shader Authoring Language*, 2010. URL http://developer.nvidia.com/object/cg_toolkit.html, abgerufen 06.07.2010.

[OGL4] The Khronos Group. *The OpenGL Specification, Version 4.0*, März 2010. URL http://www.opengl.org/registry/, abgerufen 13.02.2010.

[PMTH01] Proudfoot, Kekoa, Mark, William R., Tzvetkov, Svetoslav und Hanrahan, Pat. *A real-time procedural shading system for programmable graphics hardware*. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, Seiten 159–170. ACM, New York, NY, USA, 2001. doi:http://doi.acm.org/10.1145/383259.383275. URL http://graphics.stanford.edu/projects/shading/pubs/sig2001/.

[RF03] Randima Fernando, Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. URL http://developer.nvidia.com/object/cg_tutorial_home.html.

[RFC3492] Costello, A. *Punycode: A bootstring encoding of unicode for internationalized domain names in applications (idna)*, 2003. URL http://www.ietf.org/rfc/rfc3492.txt.

[RWZ88] Rosen, B. K., Wegman, M. N. und Zadeck, F. K. *Global value numbers and redundant computations*. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Seiten 12–27. ACM, New York, NY, USA, 1988. doi:http://doi.acm.org/10.1145/73560.73562.

[Uni09] The Unicode Consortium. *The Unicode Standard, Version 5.2*, 2009. URL http://www.unicode.org/versions/Unicode5.2.0/, abgerufen 01.12.2009.

[Wat02] Watt, Alan. *3D-Computergrafik*. Pearson Studium, München, 3. Auflage, 2002.

[Wir05] Wirth, Niklaus. *Compiler Construction*. Addison-Wesley, Zürich, November 2005. URL `http://www-old.oberon.ethz.ch/WirthPubl/CBEAll.pdf`, "a slightly revised version of the book published by Addison-Wesley in 1996"; abgerufen 14.02.2010.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Seitens des Verfassers bestehen keine Einwände die vorliegende Diplomarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Ort, Datum, Unterschrift

Zum vorliegenden PDF-Dokument

Das Ihnen vorliegende PDF-Dokument wurde am 20. Mai 2021aus den Original-Quellen erzeugt. Die Neu-Erzeugung war nötig um eine korrekte PDF/A-Datei zu erhalten.

Inhaltlich wurden keine Änderungen vorgenommen. Allerdings hat sich, aufgrund von Veränderungen in den verwendeten LaTeX-Paketen, teilweise das Layout und damit die Seitenzahl geändert.

Die Original-Quellen sowie das daraus 2011 erzeugt PDF können hier eingesehen werden: `https://github.com/res2k/shader1/tree/master/thesis`

Frank Richter