

fxPascal

Shader Compiler für OpenGL

Version 0.3

Lars Middendorf (lmid@gmx.de)

01.11.03

Inhaltsverzeichnis

1.Einleitung	3
2.Programmstruktur	3
3.Deklarationen	3
3.1 Lokale Variablen	4
3.2 Program Parameter	4
3.3 Varying Variablen	4
3.4 Datentypen	5
3.5 Funktionen und Prozeduren	5
4.Anweisungen	7
4.1 Zuweisungen	7
4.2 With Anweisung	7
4.3 If Then Else Anweisung	7
4.4 Discard Anweisung	8
4.5 For Anweisung	8
4.6 Ausdrücke	9
5.Vordefinierte Variablen	10
5.1 Variablen in Vertex Programmen	12
5.2 Variablen in Fragment Programmen	12
6. Vordefinierte Funktionen	13
7.Compiler Befehle	29
8. Optimierungen	31
8.1 Konstante Ausdrücke	31
8.2 Funktionsaufrufe	31
8.3 Addition und Multiplikation	32
8.4 Gleiche Teilausdrücke	32
8.5 Optimierung von Anweisungen	33
8.6 Variablen	34
9. fxPascal API	35
10.Beispiel Programme	36
10.1 Einfaches Vertex Programm	36
10.2 Einfaches Fragment Programm	36
10.3 Lightmapping	37
10.4 PerPixel Lichtintensität	38
10.5 Tangent Space Bumpmapping	39
10.6 Volumetrisches Licht	41

Programmieren mit fxPascal

1. Einleitung

Es gibt unter OpenGL die zwei Extensions ARB_vertex_program und ARB_fragment_program. Diese Extensions ermöglichen es den kleinen Programmen anzugeben, die für jeden Vertex oder jedes Fragment an Stelle der festen OpenGL Funktionen ausgeführt werden. Diese Programme werden in einer Art Assembler geschrieben und können fast alle Stufen der OpenGL Renderpipeline ersetzen. Immer leistungsfähigere Hardware erlaubt immer längere Programme direkt auf der GPU ablaufen zu lassen, aber die Entwicklung langer Shader in Assembler ist schwierig, zeitaufwändig und fehleranfällig. fxPascal ist eine Sprache mit Pascal Syntax, die auf diesen beiden Extensions aufsetzt und es ermöglicht Vertex und Fragment Programme auf einer höheren Ebene zu entwerfen. Das Programm besteht aus einer Unit mit den zwei wichtigen Klassen TVertexProgram und TFragmentProgram, die die Übersetzung von fxPascal für die jeweilige Extension durchführen. Die Klassen braucht man allerdings nicht direkt ansprechen, weil es die Funktion „CompileProgram“ gibt, die den übergebenen String auswertet und die richtige Klasse für die Kompilierung benutzt. Außerdem kann man eine Liste mit Symbolen und den zugeordneten Indizes erhalten, damit man anhand eines Parameternamens die Nummer des Parameters in OpenGL ermitteln kann. Dadurch kann man Parameter nur noch als String betrachten und braucht sich nicht mehr um die Zuordnung im Parameter Array von OpenGL zu kümmern. FxPascal ist zwar ein 3-Pass Compiler, aber die Kompilierung wird trotzdem bei dem ersten Syntaxfehler mit einer Exception abgebrochen. Die drei Schritte des Compilers sind in Parsen, Codegenerierung und Optimierung. Die lexikalische Analyse ist im Schritt Parsen enthalten und die semantische Analyse wird kurz bevor der Erzeugung des Programmcodes im selben Durchlauf ausgeführt. Die Schnittstelle zwischen Parsen und Codegenerierung ist ein konkreter Syntaxbaum, der während der semantischen Analyse mit Symbol- und Typeinformationen dekoriert wird. Der generierte Code wird in einem Feld zwischengespeichert, bevor er während der letzten Phase optimiert wird. Während der Optimierung werden Warnungen und Hinweise generiert.

2. Programmstruktur

fxPascal Programme bestehen aus einem Programmkopf, einem Deklarationsteil und einem Anweisungsteil. Alle Programme müssen mit dem reservierten Wort „program“ beginnen. Danach folgt der Programmname und die Uses Klausel:

```
program Name ;
```

```
uses (arb_vertex_program | arb_fragment_program);
```

Mit dem Schlüsselwort „uses“ wird festgelegt, für welche OpenGL Extensions dieses Programm kompiliert werden soll. Nicht alle Extensions unterstützen alle vordefinierten Variablen und Funktionen und die Art und Weise wie bestimmte Anweisungen übersetzt werden, wird auch von der verwendeten Extension beeinflusst. Nach dem Programmkopf folgt der Bereich mit Deklarationen.

Zusätzlich zur Extension kann man auch noch weitere Optionen für das Programm in der uses Klausel angeben. Es sind arb_position_invariant und ati_draw_buffers möglich. Allerdings kann arb_position_invariant nur mit arb_vertex_program und ati_draw_buffers nur mit arb_fragment_program kombiniert werden. Durch arb_position_invariant wird festgelegt, daß das Vertex Programm die Position der Vertex nicht selber berechnet aber exakt der gleichen Stelle wie ohne Vertex Programm entspricht. Mit ati_draw_buffers kommen die 4 neuen Variablen result.color[0..3] hinzu die das gleichzeitige Rendern in mehrere Buffer erlauben.

Hinweis: Der Programmkopf wird von der Funktion CompileProgram ausgewertet um die richtige Klasse für die Kompilierung auszuwählen. Wenn man auf das Schlüsselwort „program“ und die „uses“ Anweisung verzichten möchte, dann kann man selber eine Instanz von TVertexProgram oder TFragmentProgram erstellen und den Parameter „parseheader“ im Constructor auf „false“ setzen.

Zeilenweise Kommentare werden mit „//“ begonnen und Blockkommentare werden in „{,“ und „}“ eingeschlossen und können überall im Programm auftreten.

3. Deklarationen

Variablen Deklarationen haben folgenden Aufbau:

```
Var1, Var2, ... Varn: Datentype;
```

```
Var1, Var2, ... Varn: array[UntereGrenze..ObereGrenze] of Datentype;
```

Die untere Grenze muß 0 betragen.

Im Deklarationsteil gibt es verschiedene Blöcke für verschiedene Arten von Variablen:

3.1 Lokale Variablen

Lokale Variablen sind nur während der Ausführung des Programms gültig und können gelesen und geschrieben werden. Zu Beginn ist ihr Inhalt undefiniert. Ein Block mit Lokalen Variablen wird mit dem Schlüsselwort „var“, eingeleitet.

3.2 Programm Parameter

Programm Parameter Blöcke werden mit dem Schlüsselwort „param“, begonnen und sind zusammen mit den Konstanten die einzigen Blöcke, in denen auch Arrays definiert werden können. Programm Parameter können im Programm selber nur gelesen werden und werden vor der Programmausführung durch OpenGL Befehle gesetzt. Parameter Variablen entsprechen lokalen Programmparametern.

3.3 Varying Variablen

Varying Variablen bilden die Schnittstelle zwischen Vertex und Fragment Programm. Sie werden in Vertex Programmen geschrieben und in Fragment Programmen gelesen. Der Wert der Variablen wird über das Dreieck interpoliert. Intern werden Varying Variablen Texture Koordinaten zugewiesen. Ein Varying Block beginnt mit dem Schlüsselwort „varying“.

3.4 Konstanten

Konstante Variablen haben einen vordefinierten Wert und können nur gelesen werden. Sie können vom Typ vec4, single oder ein Array mit einem dieser beiden Typen als Basis sein. Ein Block mit Konstanten beginnt mit dem Schlüsselwort „const“. Konstanten werden mit folgendem Syntax deklariert:

```
Var1, Var2, ... Varn: Datentype=KonstanterAusdruck;
```

```
Var1, Var2, ... Varn: array[UntereGrenze..ObereGrenze] of Datentype=  
    (KonstanterAusdruck1, KonstanterAusdruck2, ... KonstanterAusdruckn);
```

Auch hier muß die untere Grenze 0 betragen.

Beispiele für Deklarationen:

```
const  
    data: array[0..1] of vec4=(vector(1,2,3,4),vector(1,2,3,4));  
var  
    a,b,c:vec4;  
param  
    lightpos:vec4;  
    paramarray: array[0..10] of single;  
varying  
    texcoord:vec4;
```

3.4 Datentypen

Es gibt nur drei die Datentypen integer, single und vec4. Vec4 ist ein Vektor mit 4 Komponenten und single ist eine einzige Gleitkommazahl. Der Wert einer Variablen vom Type integer single wird in der w Komponente des Vektors gespeichert, weil einige Grafikkarten zwei Operationen zur gleichen Zeit durchführen können, wenn eine davon nur in die letzte Komponente eines Vektors schreibt. Der einzige Unterschied zwischen Integer und Single ist, daß Integer Variablen nur Gannzahlen speichern können.

3.5 Funktionen und Prozeduren

Zusätzlich zu den Variablen können auch eigene Funktionen und Prozeduren im Deklarationsteil eines Programmes angegeben werden. Eine Funktionsdeklaration beginnt mit dem Schlüsselwort „function“, gefolgt von dem Funktionsnamen und der Parameterliste in Klammern. Direkt nach der Parameterliste wird, durch einen Doppelpunkt getrennt, der Datentype des Rückgabewertes angegeben. Bei den Datentypen für Parameter und Rückgabewert sind nur vec4 und single erlaubt. Ein Parameter kann noch zusätzlich als „const“, „var“ oder „out“ deklariert werden. Parameter mit dem Schlüsselwort „const“ können innerhalb der Funktion nicht verändert werden. Um einen Rückgabewert zu definieren wird entweder das Schlüsselwort „var“ oder „out“ benutzt. Der Unterschied zwischen „var“ und „out“ ist, daß „out“ Parameter zu Beginn der Funktion undefiniert sind, weil die übergebene Variable nicht in die funktionslokale Variable kopiert wird. Die folgende Tabelle listet die möglichen Parameterarten auf:

Parameterart	Parameter->lokale Variable	Lokale Variable->Parameter	Veränderbar
(ohne Bez.)	Ja	Nein	Ja
Const	Ja	Nein	Nein
Var	Ja	Ja	Ja
Out	Nein	Ja	Ja

Eigene Prozeduren werden mit dem Schlüsselwort „procedure“ deklariert. Prozeduren haben ähnlich wie Funktionen auch eine Parameterliste aber keinen Rückgabewert. Danach können ein oder mehrere Blöcke mit lokalen Variablen oder lokalen Konstanten folgen, die mit dem Schlüsselwort „var“, bzw. „const“ begonnen werden. Diese funktionslokalen Variablen sind nur innerhalb der Funktion sichtbar. Es ist nicht zulässig Param oder Varying Variablen innerhalb einer Funktion zu definieren. Nach den Deklarationen folgt ein Begin End Anweisungsblock, der die Anweisungen der Funktion beinhaltet.

Syntax für die Deklaration von Funktionen:

```
function Name((const|var|out) Param1,..Paramn:Datentype):ErgebnisType;  
const  
    lokaleKonstante1, lokaleKonstante2,... lokaleKonstanten:Datentype;  
    ...  
var  
    lokaleVariable1, lokaleVariable2,... lokaleVariablen:Datentype;  
    ...  
begin  
    Anweisungen;  
end;
```

Syntax für die Deklaration von Prozeduren:

```
procedure Name((const|var|out) Param1,..Paramn:Datentype);  
const  
    lokaleKonstante1, lokaleKonstante2,... lokaleKonstanten:Datentype;  
    ...  
var  
    lokaleVariable1, lokaleVariable2,... lokaleVariablen:Datentype;  
    ...  
begin  
    Anweisungen;  
end;
```

Innerhalb der Funktion wird automatisch die lokalen Variable Result definiert, der das Funktionsergebnis zugewiesen werden muß. Funktionen können nicht rekursiv aufgerufen werden, weil es keinen Stack gibt und auch lokalen Variablen in Funktionen während der Programmausführung statisch sind. Wenn versucht wird eine Funktion direkt oder indirekt rekursiv aufzurufen, wird die Kompilierung mit einem Fehler abgebrochen. Wenn eine Funktion aufgerufen wird, werden die Funktionsparamter in die lokalen Variablen der Funktion kopiert und der Funktionscode wird direkt eingefügt. Danach wird das Funktionsergebnis aus der lokalen Variable Result gelesen.

Beispiel:

```
function clampvec(x,a,b:vec4):vec4;  
begin  
    result:=min(b,max(a,x));  
end;  
  
begin  
    result.color:=clampvec(Texture2D(0,fragment.texcoord[0]),0,0.5);  
end;
```

Generierter Code:

```
!!ARBfp1.0  
TEMP clampvec$x;  
TEMP temp0;  
TEX clampvec$x,fragment.texcoord[0],texture[0],2D;  
MAX temp0,{0,0,0,0}.www,clampvec$x;  
MIN result.color,{0.5,0.5,0.5,0.5}.www,temp0;  
END
```

4. Anweisungen

Der Anweisungsteil eines Programmes besteht aus dem Schlüsselwort „begin“, den Anweisungen des Programms und dem abschließenden Schlüsselwort „end“, mit Semicolon. Alle Anweisungen zwischen Begin und End werden bei der Programmausführung abgearbeitet. Es gibt mehrere Arten von Anweisungen. Jeder Anweisungsblock „begin“, Anweisungen, „end“, ist auch eine Anweisung.

4.1 Zuweisungen

Syntax:

```
Variable := Ausdruck;
```

Der Ausdruck wird berechnet und der Variablen zugewiesen. Bei der Variablen auf der linken Seite kann man eine Maske angeben, wenn die Variable vom Type vec4 ist. Die einzelnen Komponenten der Maske geben an, welche Komponenten des Vektors geschrieben werden. Die Komponenten dürfen in ihrer Reihenfolgen nicht vertauscht werden.

Beispiel:

```
result.color.rgb:=Texture2D(0,texcoord);  
result.color.a:=Texture2D(1,texcoord);
```

4.2 With Anweisung

Syntax:

```
with Identifier do  
    Anweisung;
```

Die With Anweisung erzeugt keinen Programmcode sondern dient dazu um lange Feldbezeichner zu vereinfachen. In der Anweisung nach dem Schlüsselwort „do“, werden bei der Überprüfung von Variablen Namen zuerst die Variablen gesucht, die mit dem Identifier beginnen. With Anweisungen können auch verschachtelt werden und erhöhen bei langen Bezeichnern die Lesbarkeit des Programms:

Beispiel:

```
with state.matrix do  
    result.position:=MatrixMult4(mvp,vertex.position);
```

4.3 If then else Anweisung

Syntax:

```
if Ausdruck then  
    Anweisung;
```

Oder:

```
if Ausdruck then  
    Anweisung  
else  
    Anweisung;
```

Wenn der Ausdruck 1 ergibt, werden die Anweisungen im then Zweig ausgeführt. Wenn der Ausdruck 0 ergibt, werden nur die Anweisungen im else Zweig ausgeführt, falls ein else Zweig vorhanden ist. Bei allen anderen Werten ist das Ergebnis der Anweisungen undefiniert, aber boolsche Ausdrücke ergeben nur den Wert 0 oder 1. Im Vertex Programm werden beide Anweisungen immer ausgeführt und das Ergebnis anhand des Ausdrucks interpoliert.

Beispiel:

```
if texcoord.x<0.5 then
    Result.color:=vector(0,0,0,0);
```

4.4 Discard Anweisung

Syntax:

```
discard;
```

Die Discard Anweisung ist nur in Fragment Programmen erlaubt und verwirft das aktuelle Fragment. Sinn macht diese Anweisung nur in Verbindung mit der if Anweisung.

Beispiel:

```
if color.alpha<0.5 then
    discard;
```

4.5 For Schleife

Syntax:

```
for Variable:=Startwert to Endwert do
    Anweisung;
```

```
for Variable:=Startwert downto Endwert do
    Anweisung;
```

Die Anweisung wird für jeden Wert der Variable zwischen Start einmal ausgeführt. Die Variable muß vom Typ Integer sein und Start und Ende müssen Integer Konstanten sein. Parameter, Varyings und Konstanten sind nicht erlaubt. Die Variable ist innerhalb der Schleife schreibgeschützt und nach Ende der Schleife undefiniert. Anstelle der einen Anweisung kann auch ein Anweisungsblock mit begin und end stehen. Bei der Version mit to muß der Startwert kleiner als der Endwert sein, weil aufwärts gezählt wird und bei der Variante mit downto wird entsprechend abwärts gezählt. Intern wird die Schleife ausgerollt, weil keine der beiden verwendeten Extensions bedingte Sprünge unterstützt.

Beispiel:

```
for j:=0 to 3 do
    color:=color+texture2d(j,fragment.texcoord[0])/(j+1);
```

Generierter Code:

```
TEX $temp0,fragment.texcoord[0],texture[0],2D;
ADD color,color,$temp0;
TEX $temp0.xyzw,fragment.texcoord[0],texture[1],2D;
MAD color,$temp0.xyzw,{0.5,0.5,0.5,0.5}.w,color;
TEX $temp0.xyzw,fragment.texcoord[0],texture[2],2D;
MAD color,$temp0.xyzw,{0.3333,0.3333,0.3333,0.3333}.w,color;
TEX $temp0,fragment.texcoord[0],texture[3],2D;
```

4.6 Ausdrücke

In Ausdrücken können alle definierten und vordefinierten Variablen benutzt werden. Zusätzlich kann bei jeder Variablen vom Type `vec4` noch eine Swizzle Maske angegeben werden, die 1 oder 4 Zeichen lang ist und angibt, wie die Komponenten des Vektors verwendet werden sollen. Single Variablen erhalten automatisch die Swizzle Maske „w“.

Beispiel:

```
a.xyzwz
```

Hier wird zum Beispiel die z und die w Komponente vertauscht. Statt (x,y,z,w) kann auch (r,g,b,a) für die Komponenten verwendet werden. Wenn die Swizzle Maske nur eine Komponente enthält, dann wird diese Komponente für alle anderen Komponenten verwendet. Die Swizzle Maske entspricht einer Konvertierung in einen anderen Vektortype mit weniger oder gleich viel Komponenten.

Es gilt:

```
a.x=a.xxxx;
```

Beide Datentypen sind zueinander kompatibel. Wenn ein Vektor in eine Single Variable konvertiert wird, dann wird die erste angegebene Komponente aus der Swizzle Maske oder „w“ benutzt, falls keine Komponenten angegeben wurde.

Es gibt die folgenden Operatoren:

```
+, -, *, /,  
=, <, >, <=, >=, >=,  
NOT, AND, OR, XOR
```

Der „/“ Operator arbeitet intern nur auf Skalaren und nicht auf Vektoren. Wenn keine Swizzle Maske mit nur einer Komponente verwendet wird, generiert der Compiler Code um jede Komponente einzeln zu dividieren.

5. Vordefinierte Variablen

Man kann alle Variablen aus den Vertex und Fragment Programme benutzen. Sie haben die gleiche Bedeutung, wie in den ARB Extensions und sind alle vom Type vec4. Für beide Arten von Programmen sind folgenden Variablen definiert:

Materialien

state.material.ambient
state.material.diffuse
state.material.specular
state.material.emission
state.material.shininess
state.material.front.ambient
state.material.front.diffuse
state.material.front.specular
state.material.front.emission
state.material.front.shininess
state.material.back.ambient
state.material.back.diffuse
state.material.back.specular
state.material.back.emission
state.material.back.shininess

Lichter

state.light[0..7].ambient
state.light[0..7].diffuse
state.light[0..7].specular
state.light[0..7].position
state.light[0..7].attenuation
state.light[0..7].spot.direction
state.light[0..7].half
state.lightprod[0..7].ambient
state.lightprod[0..7].diffuse
state.lightprod[0..7].specular
state.lightprod[0..7].front.ambient
state.lightprod[0..7].front.diffuse
state.lightprod[0..7].front.specular
state.lightprod[0..7].back.ambient
state.lightprod[0..7].back.diffuse
state.lightprod[0..7].back.specular

state.lightmodel.ambient
state.lightmodel.scenecolor
state.lightmodel.front.scenecolor
state.lightmodel.back.scenecolor

Texture Koordinaten

state.texgen[0..7].eye.s
state.texgen[0..7].eye.t
state.texgen[0..7].eye.r
state.texgen[0..7].eye.q
state.texgen[0..7].object.s
state.texgen[0..7].object.t
state.texgen[0..7].object.r
state.texgen[0..7].object.q

Nebel

state.fog.color
state.fog.params

Clip Ebenen

state.clip[0..6].plane

Punkte

state.point.size
state.point.attenuation

Matrizen

state.matrix.modelview
state.matrix.modelview.inverse
state.matrix.modelview.transpose
state.matrix.modelview.invtrans
state.matrix.modelview[0..3]
state.matrix.modelview[0..3].inverse
state.matrix.modelview[0..3].transpose
state.matrix.modelview[0..3].invtrans
state.matrix.projection
state.matrix.projection.inverse
state.matrix.projection.transpose
state.matrix.projection.invtrans
state.matrix.mvp
state.matrix.mvp.inverse
state.matrix.mvp.transpose
state.matrix.mvp.invtrans
state.matrix.texture
state.matrix.texture.inverse
state.matrix.texture.transpose
state.matrix.texture.invtrans
state.matrix.texture[0..7]
state.matrix.texture[0..7].inverse
state.matrix.texture[0..7].transpose
state.matrix.texture[0..7].invtrans
state.matrix.program
state.matrix.program.inverse
state.matrix.program.transpose
state.matrix.program.invtrans
state.matrix.program[0..31]
state.matrix.program[0..31].inverse
state.matrix.program[0..31].transpose
state.matrix.program[0..31].invtrans

Auf die einzelnen Zeilen einer Matrix kann mit dem Feld row[0..3] zugegriffen werden.

Program Environment Parameter

program_env[0..31]

Die Environment Parameter können nicht automatisch zugewiesen werden, weil sie von mehreren Programmen unterschiedlich genutzt werden und müssen deshalb über das vordefinierte Array program_env ausgelesen werden.

5.1 Variablen in Vertex Programmen

Die folgenden Variablen gelten nur in Vertex Programmen:

Vertex

vertex.position
vertex.weight
vertex.weight[0..3]
vertex.normal
vertex.color
vertex.color.primary
vertex.color.secondary
vertex.fogcoord
vertex.texcoord
for i:=0 to 7 do
vertex.texcoord[0..7]
vertex.matrixindex
vertex.matrixindex[0,4,8,12]
vertex.attrib[0..15]

Result

result.position
result.color
result.color.primary
result.color.secondary
result.color.front
result.color.front.primary
result.color.front.secondary
result.color.back
result.color.back.primary
result.color.back.secondary
result.fogcoord
result.texcoord
result.texcoord[0..7]

Falls ARB_position_invariant benutzt wird, kann auf result.position Variable nicht zugegriffen werden

5.2 Variablen in Fragment Programmen

Die folgenden Variablen gelten nur in Fragment Programmen:

Fragment

fragment.color
fragment.color.primary
fragment.color.secondary
fragment.texcoord
fragment.texcoord[0..7]
fragment.fogcoord
fragment.position

Result

result.color
result.depth

ATI_daw_buffers

Result.color[0..3]

6. Vordefinierte Funktionen

Deklaration:

```
function abs(x:single):single;overload;  
function abs(x:vec4):vec4;overload;
```

Beschreibung:

Berechnet den Absolutwert der einzelnen Komponenten des Vektors x.

Berechnung:

```
result.x:=abs(x.x);  
result.y:=abs(x.y);  
result.z:=abs(x.z);  
result.w:=abs(x.w);
```

Implementation:

Die abs Funktion existiert als direkter Vertex und Fragment Programm Opcode und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function clamp(x,a,b:single):single;overload;  
function clamp(x,a,b:vec4):vec4;overload;
```

Beschreibung:

Mit der clamp Funktion wird sichergestellt, daß der Rückgabewert im Intervall [a..b] liegt. Falls x kleiner als a ist, wird x auf a gesetzt. Wenn x größer als b ist, wird x auf b gesetzt. Die Funktion arbeitet komponentenweise.

Berechnung:

```
result.x:=min(b.x,max(a.x,x.x));  
result.y:=min(b.y,max(a.y,x.y));  
result.z:=min(b.z,max(a.z,x.z));  
result.w:=min(b.w,max(a.w,x.w));
```

Implementation:

Diese Funktion wird mit Hilfe des Min und Max Opcodes wie in der Berechnung angegeben implementiert. Wenn man nur auf den Bereich von 0..1 beschränken will, ist es bei Fragment Programmen günstiger die sat Funktion zu benutzen.

Deklaration:

```
function cos(x:single):single;
```

Beschreibung:

Berechnet den Cosinus den Winkels x im Bogenmaß. Es wird die erste angegebene Komponente von x oder sonst die w Komponente benutzt.

Berechnung:

```
result.xyzw:=cos(x);
```

Implementation:

In Vertex Programmen wird der Cosinus durch die Taylor Reihe angenähert. Es wird dazu folgender Code erzeugt:

```
MUL temp1.x,{0.1591549430,0.1591549430,0.1591549430,0.1591549430},x.x;
FRC temp1.y,temp1.x;
SLT temp2.x,temp1.y,{0.25,-9.0,0.75,0.1591549430};
SGE temp2.yz,temp1.y,{0.25,-9.0,0.75,0.1591549430};
DP3 temp2.y,temp2,{-1.0,1.0,-1.0,1.0};
ADD temp0.xyz,-temp1.y,{0.0,0.5,1.0,0.0};
MUL temp0,temp0,temp0;
MAD temp1,{24.9808039603,-24.9808039603,24.9808039603,
-24.9808039603},temp0,{0.75,60.1458091736,-0.1458091736,60.1458091736};
MAD temp1,temp1,temp0,{85.4537887573,-85.4537887573,85.4537887573,
-85.4537887573};
MAD temp1,temp1,temp0,{-64.9393539429,64.9393539429,
-64.9393539429,64.9393539429};
MAD temp1,temp1,temp0,{19.7392082214,-19.7392082214,19.7392082214,
-19.7392082214};
MAD temp1,temp1,temp0,{-1.0,1.0,-1.0,1.0};
DP3 result,temp1,-temp2;
```

In Fragment Programmen wird der COS Befehl benutzt.

Deklaration:

```
function cross(x,y:vec4):vec4;
```

Beschreibung:

Berechnet das Kreuzprodukt der Vektoren x und y.

Berechnung:

```
result:=x X y;
```

Implementation:

Die cross Funktion existiert als direkter Vertex und Fragment Programm Opcode (XPD) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function dist(x,y:vec4):single;
```

Beschreibung:

Die dist Funktion berechnet den Abstand der beiden Punkte x und y.

Berechnung:

```
v:=x-y;  
result.xyzw:=sqrt(sqr(v.x)+sqr(v.y)+sqr(v.z));
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
SUB temp,x,y;  
DP3 temp.w,temp,temp;  
RSQ temp.w,temp.w;  
RCP result,temp.w;
```

Deklaration:

```
function dot(x,y:vec4):single; function dot3(x,y:vec4):single;
```

Beschreibung:

Diese beiden Funktionen berechnen beide ein 3 Komponenten Skalarprodukt aus den beiden übergebenen Vektoren x und y.

Berechnung:

```
result.xyzw:=(x.x*y.x) + (x.y*y.y) + (x.z*y.z);
```

Implementation:

Die dot3 Funktion existiert als direkter Vertex und Fragment Programm Opcode (DP3) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function dot4(x,y:vec4):single;
```

Beschreibung:

Diese Funktionen berechnet ein 4 Komponenten Skalarprodukt aus den beiden übergebenen Vektoren x und y.

Berechnung:

```
result.xyzw:=(x.x*y.x) + (x.y*y.y) + (x.z*y.z) + (x.w*y.w);
```

Implementation:

Die dot4 Funktion existiert als direkter Vertex und Fragment Programm Opcode (DP4) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function doth(x,y:vec4):single;
```

Beschreibung:

Diese Funktionen berechnet das homogene Skalarprodukt aus den beiden übergebenen Vektoren x und y.

Berechnung:

```
result.xyzw:=(x.x*y.x) + (x.y*y.y) + (x.z*y.z) + y.w;
```

Implementation:

Die dot4 Funktion existiert als direkter Vertex und Fragment Programm Opcode (DPH) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function exp2(x:single):single;
```

Beschreibung:

Berechnet die x. Potenz von 2. Es sind keine negativen Werte für x erlaubt. Es wird die erste angegebene Komponente oder die w Komponente benutzt.

Berechnung:

```
result.xyzw:=2x;
```

Implementation:

Die exp2 Funktion existiert als direkter Vertex und Fragment Programm Opcode (EX2) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function frac(x:single):single;overload;  
function frac(x:vec4):vec4;overload;
```

Beschreibung:

Die frac Funktion liefert den Nachkommateil der Komponenten des Vektors x zurück. Diese Funktion arbeitet komponentenweise.

Berechnung:

```
result.x:=frac(x.x);  
result.y:=frac(x.y);  
result.z:=frac(x.z);  
result.w:=frac(x.w);
```

Implementation:

Die frac Funktion existiert als direkter Vertex und Fragment Programm Opcode (FRC) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function floor(x:single):single;overload;  
function floor(x:vec4):vec4;overload;
```

Beschreibung:

Die floor Funktion berechnet den größten Integer, der kleiner als x ist und arbeitet komponentenweise.

Berechnung:

```
result.x:=floor(x.x);  
result.y:=floor(x.y);  
result.z:=floor(x.z);  
result.w:=floor(x.w);
```

Implementation:

Die floor Funktion existiert als direkter Vertex und Fragment Programm Opcode (FLR) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function length(x:vec4):single;
```

Beschreibung:

Berechnet die Länge des Vektors x und liefert sie in allen Komponenten des Ergebnisses zurück.

Berechnung:

```
result.xyzw:=sqrt(sqr(x.x)*sqr(x.y)*sqr(x.z))
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
DP3 temp,w,x,x;
RSQ temp.w,temp.w;
RCP result,temp.w;
```

Deklaration:

```
function lightatten(x,y:vec4):single;
```

Beschreibung:

Wenn x die relative Position zum Licht und y die Größe des Lichtes enthält, dann berechnet die Funktion die Lichthelligkeit am Punkt x.

Berechnung:

```
result.xyzw:=1-sqr(x*y);
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
MUL temp,x,y;
DP3 temp,temp,temp;
ADD result,{1,1,1,1},-temp;
```

Deklaration:

```
function log2(x:single):single;
```

Beschreibung:

Berechnet den Logarithmus zur Basis 2 von x. Es wird die erste angegebene Komponente oder die w Komponente benutzt.

Berechnung:

```
result.xyzw:=log2(x);
```

Implementation:

Die log2 Funktion existiert als direkter Vertex und Fragment Programm Opcode (LG2) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function lerp(x,y,f:single):single;overload;  
function lerp(x,y,f:vec4):vec4;overload;
```

Beschreibung:

Die lerp Funktion interpoliert linear von x nach y mit dem Faktor f. Die Funktion arbeitet komponentenweise.

Berechnung:

```
result.x:=x.x*(1-f)+y.x*f;  
result.y:=x.y*(1-f)+y.y*f;  
result.z:=x.z*(1-f)+y.z*f;  
result.w:=x.w*(1-f)+y.w*f;
```

Implementation:

In Vertex Programmen wird die Funktion durch folgende Befehlsfolge implementiert:

```
SUB temp,x,y;  
MAD result
```

In Fragment Programmen gibt es den Befehl LRP, der die gleiche Funktion ausführt.

Deklaration:

```
function mad(x,y,z:single):single;overload;  
function mad(x,y,z:vec4):vec4;overload;
```

Beschreibung:

Addiert z zu dem Produkt aus x und y.

Berechnung:

```
result.x:=x.x*y.x+z.x;  
result.y:=x.y*y.y+z.y;  
result.z:=x.z*y.z+z.z;  
result.w:=x.w*y.w+z.w;
```

Implementation:

Die mad Funktion existiert als direkter Vertex und Fragment Programm Opcode (MAD) und wird auch mit diesem Befehl implementiert. Wenn der Compiler auf einen Ausdruck der Form $x*y+z$ trifft, wird dieser automatisch in einen Aufruf der Funktion mad konvertiert.

Deklaration:

```
function matrix(row0,row1,row2[,row4]:vec4):matrix;
```

Beschreibung:

Gibt die Matrix aus den übergebenen Zeilenvektoren zurück. Diese Funktion kann nur als erstes Argument von matrixmult3 oder matrixmult4 verwendet werden. Es müssen dann immer so viele Zeilen angegeben werden, wie die entsprechende Funktion verlangt.

Berechnung:

```
result.row[0]:=row0;  
result.row[1]:=row1;  
result.row[2]:=row2;  
result.row[3]:=row3;
```

Implementation:

Die Funktion an sich erzeugt keinen Code, sondern bewirkt nur eine Veränderung in der Code Generierung der matrixmult3 oder matrixmult4 Funktion.

Deklaration:

```
function matrixmult3(mat:matrix;x:vec4):vec4;
```

Beschreibung:

Multipliziert die 3*3 Matrix mit dem Vector x.

Berechnung:

```
result.x:=matrix[0].x*x.x + matrix[0].y*x.y + matrix[0].z*x.z;  
result.y:=matrix[1].x*x.x + matrix[1].y*x.y + matrix[1].z*x.z;  
result.z:=matrix[2].x*x.x + matrix[2].y*x.y + matrix[2].z*x.z;
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
DP3 result.x,matrix.row[0],x;  
DP3 result.y,matrix.row[1],x;  
DP3 result.z,matrix.row[2],x;
```

Deklaration:

```
function matrixmult4(mat:matrix;x:vec4):vec4;
```

Beschreibung:

Multipliziert die 4*4 Matrix mit dem Vector x.

Berechnung:

```
result.x:=matrix[0].x*x.x+matrix[0].y*x.y+matrix[0].z*x.z+matrix[0].w*x.w;  
result.y:=matrix[1].x*x.x+matrix[1].y*x.y+matrix[1].z*x.z+matrix[1].w*x.w;  
result.z:=matrix[2].x*x.x+matrix[2].y*x.y+matrix[2].z*x.z+matrix[2].w*x.w;  
result.w:=matrix[3].x*x.x+matrix[3].y*x.y+matrix[3].z*x.z+matrix[3].w*x.w;
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
DP4 result.x,matrix.row[0],x;  
DP4 result.y,matrix.row[1],x;  
DP4 result.z,matrix.row[2],x;  
DP4 result.w,matrix.row[3],x;
```

Deklaration:

```
function max(x,y:single):single;overload;  
function max(x,y:vec4):vec4;overload;
```

Beschreibung:

Die max Funktion gibt jeweils das Maximum einer Komponente der beiden Vektoren x und y zurück.

Berechnung:

```
result.x:=max(x.x,y.x);  
result.y:=max(x.y,y.y);  
result.z:=max(x.z,y.z);  
result.w:=max(x.w,y.w);
```

Implementation:

Die max Funktion existiert als direkter Vertex und Fragment Programm Opcode (MAX) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function min(x,y:single):single;overload;  
function min(x,y:vec4):vec4;overload;
```

Beschreibung:

Die min Funktion gibt jeweils das Minimum einer Komponente der beiden Vektoren x und y zurück.

Berechnung:

```
result.x:=min(x.x,y.x);  
result.y:=min(x.y,y.y);  
result.z:=min(x.z,y.z);  
result.w:=min(x.w,y.w);
```

Implementation:

Die min Funktion existiert als direkter Vertex und Fragment Programm Opcode (MIN) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function normalize(x:vec4):vec4;
```

Beschreibung:

Die normalize Funktion teilt den Vektor x durch seine Länge und gibt den normalisierten Vektor zurück.

Berechnung:

```
result.xyz:=x/length(x);
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
DP3 temp.w,x,x;  
RSQ temp.w;temp.w;  
MUL result,x,temp;
```

Deklaration:

```
function power(x,y:single):single;
```

Beschreibung:

Die power Funktion berechnet die y. Potenz von x. Es wird die erste angegebene Komponente oder die w Komponente benutzt, wenn die Parameter Vektoren sind.

Berechnung:

```
result.xyzw:=xy;
```

Implementation:

Die power Funktion existiert als direkter Vertex und Fragment Programm Opcode (POW) und wird auch mit diesem Befehl implementiert.

Deklaration:

```
function reflect(v,n:vec4):vec4;
```

Beschreibung:

Der Vektor v wird an der Normalen n reflektiert.

Berechnung:

```
result.x:=v.x-2*dot*n.x;  
result.y:=v.y-2*dot*n.y;  
result.z:=v.z-2*dot*n.z;  
result.w:=v.w-2*dot*n.w;
```

Implementation:

Diese Funktion wird mit Hilfe folgender Befehlsfolge implementiert:

```
DP3 temp,v,n;  
MUL temp,temp,n;  
MAD result,temp,{ -2,-2,-2,-2 },v;
```

Deklaration:

```
function sat(x:vec4):vec4;
```

Beschreibung:

Die sat Funktion stellt sicher, daß der zurückgegebene Wert im Bereich von 0..1 liegt. Wenn x kleiner als 0 ist, wird 0 zurückgeliefert. Fall x größer als 1 ist wird 1 zurückgegeben. Die Funktion arbeitet komponentenweise.

Berechnung:

```
result:=clamp(x,0,1);
```

Implementation:

In Vertex Programmen wird diese Funktion, ähnlich wie die clamp Funktion, mit Hilfe der MIN und MAX Befehle umgesetzt. In Fragment Programmen wird wenn möglich der _SAT Zusatz an den entsprechenden Befehl gehängt.

Deklaration:

```
function sin(x:single):single;
```

Beschreibung:

Berechnet den Sinus den Winkels x im Bogenmaß. Es wird die erste angegebene Komponente von x oder sonst die w Komponente benutzt.

Berechnung:

```
result.xyzw:=sin(x);
```

Implementation:

In Vertex Programmen wird der Sinus durch die Taylor Reihe angenähert. Es wird dazu folgender Code erzeugt:

```
MAD temp1.x,{0.1591549430,0.1591549430,0.1591549430,0.1591549430},x.x,
    {-0.25};
FRC temp1.y,temp1.x;
SLT temp2.x,temp1.y,{0.25,-9.0,0.75,0.1591549430};
SGE temp2.yz,temp1.y,{0.25,-9.0,0.75,0.1591549430};
DP3 temp2.y,temp2,{-1.0,1.0,-1.0,1.0};
ADD temp0.xyz,-temp1.y,{0.0,0.5,1.0,0.0};
MUL temp0,temp0,temp0;
MAD temp1,{24.9808039603,-24.9808039603,24.9808039603,
    -24.9808039603},temp0,{0.75,60.1458091736,-0.1458091736,60.1458091736};
MAD temp1,temp1,temp0,{85.4537887573,-85.4537887573,85.4537887573,
    -85.4537887573};
MAD temp1,temp1,temp0,{-64.9393539429,64.9393539429,
    -64.9393539429,64.9393539429};
MAD temp1,temp1,temp0,{19.7392082214,-19.7392082214,19.7392082214,
    -19.7392082214};
MAD temp1,temp1,temp0,{-1.0,1.0,-1.0,1.0};
DP3 result,temp1,-temp2;
```

In Fragment Programmen wird direkt der SIN Befehl benutzt.

Deklaration:

```
function sqr(x:single):single;
```

Beschreibung:

Die sqr Funktion berechnet das Quadrat von x komponentenweise.

Berechnung:

```
result.x:=x.x*x.x;  
result.y:=x.y*x.y;  
result.z:=x.z*x.z;  
result.w:=x.w*x.w;
```

Implementation:

Die sqr Funktion wird mit dem MUL Befehl implementiert.

Deklaration:

```
function sqrt(x:single):single;
```

Beschreibung:

Die sqrt Funktion berechnet die Quadratwurzel der ersten Komponente von x. Wenn keine Komponente angegeben wurde, wird die w Komponente benutzt.

Berechnung:

```
result.xyzw:=sqrt(x);
```

Implementation:

Die sqrt Funktion wird in folgende Befehlsfolge übersetzt:

```
RSQ temp.w,x;  
RCP result,temp.w;
```

Deklaration:

```
function texture1d(TextureID:integer;coord[,bias]:vec4):vec4;  
function texture2d(TextureID:integer;coord[,bias]:vec4):vec4;  
function texture3d(TextureID:integer;coord[,bias]:vec4):vec4;  
function texturecubemap(TextureID:integer;coord[,bias]:vec4):vec4;  
function texturerec(TextureID:integer;coord[,bias]:vec4):vec4;
```

Beschreibung:

Diesen Funktionen lesen jeweils aus einer Texture, dessen Typ im Namen der Funktion angegebenen wird, an der Stelle coord und geben den Farbwert als Vektor zurück. Optional kann auch noch ein LOD Bias für Mipmaps angegeben werden. TextureID ist eine Integer Konstante, die die Nummer der Textureeinheit angibt, aus der gelesen werden soll. Dieser Befehl ist nur im Fragment Programmen verfügbar.

Implementation:

Diese Funktion wird mit dem TEX oder TXB Befehl implementiert, je nachdem ob ein Mipmap Bias angegeben wurde oder nicht.

Deklaration:

```
function normalmap(TextureID:integer;coord[,bias]:vec4):vec4;
```

Beschreibung:

Die Funktion liest aus einer 2D Texture an der Stelle coord eine gepackte Normale und gibt den entpackten Wert als Vektor zurück. Optional kann auch noch ein LOD Bias für Mipmaps angegeben werden. TextureID ist eine Integer Konstante, die die Nummer der Textureeinheit angibt, aus der gelesen werden soll. Dieser Befehl ist nur im Fragment Programmen verfügbar.

Berechnung:

```
result:=SampleTexture2D(textureID,coord,bias)*2-1;
```

Implementation:

Diese Funktion wird mit dem TEX oder TXB Befehl implementiert, je nachdem ob ein Mipmap Bias angegeben wurde oder nicht. Danach wird die Multiplikation und Addition mit dem MAD Befehl durchgeführt.

Deklaration:

```
function texture1dproj(TextureID:integer;coord:vec4):vec4;  
function texture2dproj(TextureID:integer;coord:vec4):vec4;  
function texture3dproj(TextureID:integer;coord:vec4):vec4;  
function texturecubemapproj(TextureID:integer;coord:vec4):vec4;  
function texturerectproj(TextureID:integer;coord:vec4):vec4;
```

Beschreibung:

Diesen Funktionen lesen jeweils aus einer Texture, dessen Typ im Namen der Funktion angegebenen wird, an der Stelle coord.xyz/coord.w und geben den Farbwert als Vektor zurück. Die Texturekoordinaten werden vor dem Lesen durch die w Komponente der Koordinaten geteilt. TextureID ist eine Integer Konstante, die die Nummer der Textureenheit angibt, aus der gelesen werden soll. Dieser Befehl ist nur im Fragment Programmen verfügbar.

Implementation:

Diese Funktion wird mit dem TXP OpCode implementiert.

Deklaration:

```
function vector(x,y,z,w:single):vec4;
```

Beschreibung:

Die Vektor Funktion gibt einen Vektor zurück, dessen Koordinaten als Paramter angegeben werden. Bei den Paramtern wird jeweils nur die erste angegebene Komponente oder ,falls keine Komponente angegeben, die x Komponente benutzt.

Berechnung:

```
result.x:=x;  
result.y:=y;  
result.z:=z;  
result.w:=w;
```

Implementation:

Die einzelnen Komponenten werden mit dem MOV Befehl zusammengesetzt.

7.Compiler Befehle

Compilerbefehle beeinflussen die Generierung des ARB Quelltextes und haben die Form eines Kommentars. Der Befehl wird zwischen „{\$,“ und „}“ eingeschlossen. Es gibt zwei Compilerbefehle.

Optimierungen

```
{$optimize-}  
{$optimize+}
```

Dieser Befehl schaltet die Optimierung am Ende der Code Erzeugung ein oder aus. Standardmäßig sind die Optimierungen eingeschaltet und sollten nur zum Debuggen oder Analysieren von Shadern ausgeschaltet werden. Dieser Compiler Befehl gilt immer für das gesamte Programm.

Register Optimierung

```
{$regcount-}  
{$regcount+}
```

Der Befehl stellt das Verhalten des Compilers bezüglich der Registerbelegung ein und legt fest, ob eher auf kürzere Programme oder weniger Register hin optimiert werden soll. Diese Optimierung betrifft Zwischenergebnisse aus verschiedenen Anweisungen für die sonst ein zusätzliches Register belegt werden müßte. Der Befehl gilt sofort ab der Position an der er auftritt. Standardmäßig ist die Einstellung auf {\$regcount-}. Allgemein wird mit {\$regcount-} besserer Code erzeugt und man sollte diese Einstellung nur ändern, wenn man das Registerlimit erreicht. Es gibt keine Garantie für eine geringere Registerbenutzung, aber mit {\$regcount-} wird die Lebendigkeitsanalyse zusätzlich noch einmal am Anfang der Optimierungsphase durchgeführt.

Beispiel:

```
program Test;  
  
uses ARB_fragment_program;  
  
{ $regcount+ }  
  
var  
    a,b,c:vec4;  
begin  
    a:=sqr(sqr(fragment.color));  
    b:=a*a;  
    c:=b*b;  
    result.color:=c*sqr(sqr(fragment.color));  
end;  
  
!!ARBfp1.0  
TEMP c;  
TEMP temp0;  
MUL temp0.xyzw,fragment.color,fragment.color;  
MUL temp0.xyzw,temp0.xyzw,temp0.xyzw;  
MUL temp0.xyzw,temp0.xyzw,temp0.xyzw;  
MUL c.xyzw,temp0.xyzw,temp0.xyzw;  
MUL temp0.xyzw,fragment.color,fragment.color;  
MUL temp0.xyzw,temp0.xyzw,temp0.xyzw;  
MUL result.color,c.xyzw,temp0.xyzw;  
END
```

Das gleiche Programm mit {\$regcount-} sieht so aus:

```
program Test;

uses ARB_fragment_program;

{$regcount-}

var
    a,b,c:vec4;
begin
    a:=sqr(sqr(fragment.color));
    b:=a*a;
    c:=b*b;
    result.color:=c*sqr(sqr(fragment.color));
end;

!!ARBfp1.0
TEMP temp0;
TEMP temp1;
TEMP temp2;
MUL temp0.xyzw,fragment.color,fragment.color;
MUL temp2.xyzw,temp0.xyzw,temp0.xyzw;
MUL temp2.xyzw,temp2.xyzw,temp2.xyzw;
MUL temp1.xyzw,temp2.xyzw,temp2.xyzw;
MUL temp2.xyzw,temp0.xyzw,temp0.xyzw;
MUL result.color,temp1.xyzw,temp2.xyzw;
END
```

Symbole

```
{$define Symbolname}
{$undefine Symbolname}
```

Mit dem \$define und dem \$undefine Befehl können Symbolnamen zur definiert oder wieder gelöscht werden. Wenn ein Symbol mehrfach definiert wird, wird es trotzdem nur einmal der Liste hinzugefügt. Entsprechend wird das Symbol beim ersten Auftreten des \$undefine Befehls wieder entfernt. Es ist kein Fehler ein Symbol mit \$undefine zu Entfernen, daß nicht zuvor mit \$define definiert worden ist. Bei dem Symbolnamen spiel Groß- und Kleinschreibung kein Rolle. Zusätzlich kann man auch noch eine Symbolliste beim Aufruf der Funktion CompileProgram übergeben um die bedingte Kompilierung von außerhalb zu steuern.

Bedingte Kompilierung

```
{$ifdef Symbolname}
{$else}
{$endif}
```

```
{$ifdef Symbolname}
{$endif}
```

Mit den \$if,\$else und \$endif Befehlen können Blöcke definiert werden, die nur kompiliert werden, falls das angegebene Symbol zuvor definiert worden ist. Diese Blöcke können auch verschachtelt werden, aber zu jeden \$ifdef Befehl muß es einen entsprechenden \$endif Befehl geben.

8.Optimierungen

Der fxPascal Compiler führt verschiedene Optimierungen an Ausdrücken und Anweisungen durch, obwohl der OpenGL Treiber wird das ARB Programm vermutlich auch noch mal kompiliert und optimiert.

8.1 Konstante Ausdrücke

Vektor und skalare Konstanten in Ausdrücken werden direkt ausgerechnet. Dies betrifft die Operatoren +, -, * und /. Eine Division durch einen konstanten Ausdruck wird durch eine Multiplikation mit dem Kehrwert ersetzt. Als konstant deklarierte Array Variablen gelten hierbei jedoch nicht als Konstante, sondern als Programm Parameter mit vorbestimmten Wert. Die Multiplikation benötigt nur einen Befehl im Gegensatz zur Division, die mit mindestens zwei Befehlen implementiert werden muß. Nur wenn der Divisor 1 ist und durch einen skalaren Wert dividiert wird, kann auch die Division mit einem Befehl durchgeführt werden. Ansonsten wird jede Komponente einzeln dividiert.

Beispiel:

```
a:=b/255;
```

Die Division wird durch eine Multiplikation ersetzt:

```
MUL a,b,{0.00390625,0.00390625,0.00390625,0.00390625};
```

8.2 Funktionsaufrufe

Wenn durch das Funktionsergebnis von sqrt, length oder dist dividiert wird, ersetzt der Compiler diese Funktion durch eine andere Funktion, die direkt den Kehrwert von sqrt, length oder dist zurückgibt. Diese Funktionen müssen eine Quadratwurzel ausrechnen und die Berechnung des Kehrwertes einer Wurzel benötigt nur einen Befehl, weil die direkt als OpCode (RSQ) existiert. Außerdem gibt es spezielle Varianten von length und dist, die gleich die quadrierte Länge und die Entfernung zum Quadrat zurückgeben. Der Compiler setzt die Funktionen automatisch ein, wenn zur Quadrierung die Funktion sqr verwendet wird.

Beispiel 1:

```
a:=1/dist(b,c);  
result.color:=a;
```

Die doppelte Berechnung des Kehrwertes wird entfernt:

```
SUB temp0,b,c;  
DP3 temp0.w,temp0,temp0;  
RSQ a,temp0.w;  
MOV result.color,a;
```

Beispiel 2:

```
a.xyz:=a/length(a);
```

Auch bei der Funktion wird die doppelte Kehrwertbildung entfernt:

```
DP3 temp0.w,a,a;  
RSQ temp0,temp0.w;  
MUL a.xyz,a,temp0;
```

8.3 Addition und Multiplikation

Ein Ausdruck der Form $x*y+z$ oder $x+y*z$ wird automatisch in einen Funktionsaufruf der mad Funktion konvertiert, die nur einen Opcode(MAD) benötigt. Wenn ein Wert mit 0 oder 1 multipliziert wird oder 0 addiert oder subtrahiert wird, dann entfernt der Compiler diese Berechnung. Wenn zwei identische Ausdrücke miteinander multipliziert werden, dann wird diese Multiplikation durch ein Aufruf der Funktion „sqr“ ersetzt.

Beispiel:

```
result.color:=a*b+c;
```

Hier werden Addition und Multiplikation zusammengefaßt:

```
MAD result.color,a,b,c;
```

Das gleiche Prinzip wird auch auf mit Subtraktion und Multiplikation angewendet.

8.4 Gleiche Teilausdrücke

Wenn auf beiden Seiten eines Operators der gleiche Teilausdruck steht, dann wird dieser Ausdruck nur einmal ausgewertet.

Beispiele:

```
program Test;

uses ARB_fragment_program;

var
    a,b,c:vec4;
begin
    a:=texture2d(0,fragment.texcoord[0])*fragment.color+
        texture2d(0,fragment.texcoord[0])*fragment.color;
    result.color:=a;
end;

!!ARBfp1.0
TEMP temp1;
TEMP temp2;
TEX temp1.xyzw,fragment.texcoord[0],texture[0],2D;
MUL temp2.xyzw,temp1.xyzw,fragment.color;
MAD result.color,temp1.xyzw,fragment.color,temp2.xyzw;
END

program Test;

uses ARB_fragment_program;

var
    a,b,c,d:vec4;
begin
with fragment do
    a:=texcoord[0]*texcoord[1]+texcoord[0]*texcoord[1];
    result.color:=a;
end;

!!ARBfp1.0
TEMP temp1;
MUL temp1.xyzw,fragment.texcoord[0],fragment.texcoord[0];
ADD result.color,temp1.xyzw,temp1.xyzw;
END
```

8.5 Optimierungen von Anweisungen

Die Anweisungsoptimierung geschieht auf der Ebene der einzelnen ARB Programm Befehle. Daher können auch mehrere Befehle und Anweisungen mit in die Optimierung einbezogen werden. Wenn der Wert einer Zuweisung niemals wieder verwendet wird, dann wird diese Zuweisung und auch die damit verbundenen Funktionsaufrufe entfernt. Die gilt auch für nicht vordefinierte Funktionen. Auch wenn die einzelnen Komponenten einer Variable nacheinander überschrieben werden, wie das z.B. bei Ergebnissen der Funktionen `vector`, `matrixmult3` und `matrixmult4` der Fall ist, gilt die Variable als überschrieben, denn der ursprünglich zugewiesene Wert wird nie verwendet. Der Compiler erkennt auch automatisch, ob ein bestimmtes Ergebnis bereits berechnet worden ist. Wenn sich die Parameter den Befehls zwischen den Befehlen nicht verändert haben, dann wird der alte Wert benutzt. Bei einfache MOV Anweisungen wird versucht in den folgenden Befehlen den zugewiesenen Wert direkt einzusetzen. Wenn dies in allen Fällen gelingt, wird die MOV Anweisung entfernt. Bei if und else Anweisungen werden immer beide Zweige ausgeführt und nacher das richtige Ergebnis mit linearer Interpolation oder dem CMP Befehl zugewiesen. Wenn die Variable sowohl auf der linken als auch auf der rechten Seite der Zuweisung vorkommt und Teil einer Addition ist, kann statt der vollständigen Interpolation auch der MAD Befehl benutzt werden. Im Fragment Programmen gibt es den CMP Befehl, der anstelle der linearen Interpolation in Vertex Programmen verwendet wird. Um einen boolschen Wert, der entweder 0 oder 1 sein kann, mit dem CMP Befehl zu verwenden muß er von 0.5 subtrahiert werden. Wenn in der Bedingung die beiden Operatoren „<“, oder „>“, benutzt worden sind, kann der SLT Befehl unter bestimmten Umständen eingespart werden. Häufig kommt es vor, daß eine Variablen auf der linken Seite einer Zuweisung sowohl im if Zweig als auch im else Zweig vorkommen. In Fragmentprogrammen werden dann zuerst zwei CMP Befehle generiert, die dann während der Optimierung zusammengefaßt werden.

Beispiele:

```
if a.x>0.5 then
  b:=b+a;
```

Diese if Anweisung generiert folgenden Code:

```
SLT temp0,-a.x,-{0.5,0.5,0.5,0.5};
MAD b,a,temp0,b;
```

```
program Test;
```

```
uses ARB_fragment_program;
```

```
var
  a:vec4;
  i:integer;
begin
  a:=vector(0,0,0,0);
  with fragment do
  for i:=1 to 7 do
    a:=texture2d(0,texturecoord[0])*texturecoord[i]+a;
  result.color:=a;
  end;
```

```
!!ARBfp1.0
TEMP temp5;
TEMP temp6;
TEX temp5.xyzw,fragment.texturecoord[0],texture[0],2D;
MAD temp6.xyzw,temp5.xyzw,fragment.texturecoord[1],{0,0,0,0}.xyzw;
MAD temp6.xyzw,temp5.xyzw,fragment.texturecoord[2],temp6.xyzw;
MAD temp6.xyzw,temp5.xyzw,fragment.texturecoord[3],temp6.xyzw;
MAD temp6.xyzw,temp5.xyzw,fragment.texturecoord[4],temp6.xyzw;
MAD temp6.xyzw,temp5.xyzw,fragment.texturecoord[5],temp6.xyzw;
MAD temp6.xyzw,temp5.xyzw,fragment.texturecoord[6],temp6.xyzw;
MAD result.color,temp5.xyzw,fragment.texturecoord[7],temp6.xyzw;
END
```

8.6 Variablen

Alle nicht benutzten Variablen werden nicht in die Deklarationen des ARB Programms mit aufgenommen. Bei Paramtern und Varying Variablen bleibt der Index jedoch bestehen und die Parameter erscheinen trotzdem in der Liste mit den Symbolen. Zusätzlich wird eine Lebendigkeitsanalyse durchgeführt um Variablen einzusparen, die nicht zur gleichen Zeit einen gültigen Wert haben. Dabei wird ein Intefferenzgraph der temporären Variablen aufgebaut um Variablenüberdeckungen schnell prüfen zu können.

Beispiel:

```
program Test;  
  
uses ARB_fragment_program;  
  
var  
    a,b,c:vec4;  
begin  
a:=texture2d(0,fragment.texcoord[0])*fragment.color;  
b:=a*a;  
c:=b*b;  
result.color:=c;  
end;
```

Hier wird nur eine temporäre Variablen benötigt und auch nur benutzt:

```
!!ARBfp1.0  
TEMP temp0;  
TEX temp0,fragment.texcoord[0],texture[0],2D;  
MUL temp0.xyzw,temp0,fragment.color;  
MUL temp0.xyzw,temp0.xyzw,temp0.xyzw;  
MUL result.color,temp0.xyzw,temp0.xyzw;  
END
```

```
program Test;  
  
uses ARB_fragment_program;  
  
var  
    a,b,color,detail:vec4;  
begin  
a:=texture2d(0,fragment.texcoord);  
b:=texture2d(1,fragment.texcoord);  
color:=a*b;  
detail:=texture2d(2,fragment.texcoord);  
result.color:=color*detail;  
end;
```

Auch hier werden nur zwei der vier angegebenen Variablen benutzt.

```
!!ARBfp1.0  
TEMP color;  
TEMP detail;  
TEX detail.xyzw,fragment.texcoord,texture[0],2D;  
TEX color.xyzw,fragment.texcoord,texture[1],2D;  
MUL color.xyzw,detail.xyzw,color.xyzw;  
TEX detail.xyzw,fragment.texcoord,texture[2],2D;  
MUL result.color,color.xyzw,detail.xyzw;  
END
```

9. fxPascal API

Der ganze Compiler befindet sich in der Unit fxpascal.pas. Zur Übersetzung von Programmen gibt es die Funktion CompileProgram, die anhand der uses Anweisung das richtige Compiler Ziel auswählt, die Klasse erstellt und den Quelltext kompiliert. Bei Fehlern wird die Exception ESyntaxError ausgelöst.

```
function CompileProgram(const Source:string; Symbols:TStringList=nil;  
    Warnings:TStringList=nil; Defines:TStringList=nil):string;
```

Parameter:

Source:TStringList	Im Parameter Source wird der Quelltext übergeben. Der Rückgabewert der Funktion enthält dann das ARB Program.
Symbols:TstringList=nil	Als Parameter symbols kann man ein Objekt vom Type TStringList oder nil übergeben. Wenn eine TStringList verwendet wird, dann wird diese Liste mit den Namen aller Programm Parameter gefüllt. Das dem String zugeordnete Objekt ist dann ein Integer, der den Index dieses Parameters im Programm Parameter Array enthält.
Warnings:TstringList=nil	Als Parameter Warnings kann man eine TStringList übergeben die dann mit eventuellen Warnungen und Hints gefüllt wird. Die Zeilennummer der Warnung befindet sich wie bei den Symbolen zugehörigen Objekt.
Defines:TstringList=nil	Um die bedingte Kompilierung zu steuern kann man eine Liste mit definierten Symbolen angeben. Die Symbole in dieser Liste werden so behandelt, als wären sie vor der ersten Programmzeile mit {\$define} definiert worden.

Rückgabewert:

Der kompilierte Quelltext für ARB_vertex_program oder ARB_fragment_program wird in einem String zurückgegeben.

Beispiel für die Benutzung der Funktion:

```
function LoadVertexShader(const filename:String):cardinal;  
var  
    sl:TStringList;  
    s:String;  
begin  
    sl:=TStringList.Create;  
    sl.LoadFromFile(filename);  
    result:=0;  
    try  
        s:=CompileProgram(sl.Text,nil);  
        glGenProgramsARB(1,@result);  
        glProgramStringARB(GL_VERTEX_PROGRAM_ARB,GL_PROGRAM_FORMAT_ASCII_ARB,  
            length(s),pchar(s));  
    finally  
        sl.free;  
    end;  
end;
```

Der erzeugte Quelltext kann direkt an OpenGL übergeben werden.

10. Beispiel Programme

10.1 Einfaches Vertex Programm

Das folgende Vertex Programm multipliziert den Vertex mit dem Produkt aus der Modelview und Projection Matrix. Außerdem werden Farbe und Texture Koordinaten an das Fragment Program oder die normale OpenGL Pipeline weitergegeben. Der With Befehl wird verwendet, damit die Zeile nicht zu lang und einfacher lesbar ist.

```
begin
with state.matrix do
    result.position:=matrixmult4(mvp,vertex.position);
result.color:=vertex.color;
result.texcoord[0]:=vertex.texcoord[0];
end;
```

Diese Vertex Program wird zu folgendem ARB Vertex Program kompiliert:

```
!!ARBvp1.0
DP4 result.position.x,state.matrix.mvp.row[0],vertex.position;
DP4 result.position.y,state.matrix.mvp.row[1],vertex.position;
DP4 result.position.z,state.matrix.mvp.row[2],vertex.position;
DP4 result.position.w,state.matrix.mvp.row[3],vertex.position;
MOV result.color,vertex.color;
MOV result.texcoord[0],vertex.texcoord[0];
END
```

10.2 Einfaches Fragment Programm

Dieses Fragment Programm liest den Farbwert aus der Texture 0 und gibt ihn danach als Ergebnis zurück:

```
begin
result.color:=texture2d(0,fragment.texcoord[0]);
end;
```

Es wird folgender Programmcode erzeugt:

```
!!ARBfp1.0
TEX result.color,fragment.texcoord[0],texture[0],2D;
END
```

Jetzt soll die Texture mit der aktuellen Farbe multipliziert werden:

```
begin
result.color:=texture2d(0,fragment.texcoord[0])*fragment.color;
end;
```

Dadurch wird dieses ARB Programm generiert:

```
!!ARBfp1.0
TEMP temp0;
TEX temp0,fragment.texcoord[0],texture[0],2D;
MUL result.color,temp0,fragment.color;
END
```

10.3 Lightmapping

Beim Lightmapping wird die Lightmap mit der Basistexture multipliziert. Die Texturekoordinaten werden durch eine Varying Variable vom Vertex Programm übermittelt. Dies wird durch folgendes Fragment Programm beschrieben:

```
varying
    texcoord:vec4;
begin
result.color:=texture2d(0,texcoord)*texture2d(1,texcoord);
end;
```

Das dazugehörige ARB Programm sieht so aus:

```
!!ARBfp1.0
ATTRIB texcoord=fragment.texcoord[0];
TEMP temp0;
TEMP temp1;
TEX temp0,texcoord,texture[0],2D;
TEX temp1,texcoord,texture[1],2D;
MUL result.color,temp0,temp1;
END
```

Das Vertex Programm soll nur die Position berechnen und die Texturekoordinaten weitergeben:

```
varying
    texcoord:vec4;
begin
texcoord:=vertex.texcoord[0];
with state.matrix do
    result.position:=matrixmult4(mvp,vertex.position);
end;
```

Der Compiler erzeugt daraus dieses ARB Vertex Programm:

```
!!ARBvp1.0
OUTPUT texcoord=result.texcoord[0];
MOV texcoord,vertex.texcoord[0];
DP4 result.position.x,state.matrix.mvp.row[0],vertex.position;
DP4 result.position.y,state.matrix.mvp.row[1],vertex.position;
DP4 result.position.z,state.matrix.mvp.row[2],vertex.position;
DP4 result.position.w,state.matrix.mvp.row[3],vertex.position;
END
```

Man kann am Vertex und Fragment Programm erkennen, daß Varying Variablen über die Texturekoordinaten übermittelt werden.

10.4 PerPixel Lichtintensität

Die Lichtintensität soll pro Pixel im Fragment Programm berechnet werden. Die Lichtposition und die Lichtgröße werden als Parameter übergeben. Das Vertex Programm berechnet die Vertex Position und die Position des Punktes relativ zu Licht. Außerdem werden noch Texturekoordinaten übergeben, um im Fragmentprogramm aus der Basistexture zu lesen.

Vertex Program:

```
param
    LightPos,LightSize,LightColor:vec4;
varying
    texcoord,lightvec:vec4;
begin
with state.matrix do
    result.position:=matrixmult4(mvp,vertex.position);
    texcoord:=vertex.texcoord[0];
    lightvec:=LightPos-vertex.position;
end;

!!ARBvp1.0
OUTPUT texcoord=result.texcoord[0];
OUTPUT lightvec=result.texcoord[1];
PARAM LightPos=program.local[0];
DP4 result.position.x,state.matrix.mvp.row[0],vertex.position;
DP4 result.position.y,state.matrix.mvp.row[1],vertex.position;
DP4 result.position.z,state.matrix.mvp.row[2],vertex.position;
DP4 result.position.w,state.matrix.mvp.row[3],vertex.position;
MOV texcoord,vertex.texcoord[0];
SUB lightvec,LightPos,vertex.position;
END
```

Fragment Program:

```
param
    LightPos,LightSize,LightColor:vec4;
varying
    texcoord,lightvec:vec4;
begin
result.color:=lightatten(lightvec,LightSize)*Texture2D(0,texcoord)
                *LightColor;
end;

!!ARBfp1.0
ATTRIB texcoord=fragment.texcoord[0];
ATTRIB lightvec=fragment.texcoord[1];
PARAM LightSize=program.local[1];
PARAM LightColor=program.local[2];
TEMP temp0;
TEMP temp1;
MUL temp0,lightvec,LightSize;
DP3 temp0,temp0,temp0;
ADD temp0,{1,1,1,1},-temp0;
TEX temp1,texcoord,texture[0],2D;
MUL temp1,temp0,temp1;
MUL result.color,temp1,LightColor;
END
```

Die Param Listen von beiden Programmen müssen nicht übereinstimmen. Da im Fragment LightPos nicht benutzt wird, entfernt der Compiler die Deklaration. Die Adresse im Parameter Array bleibt jedoch erhalten.

10.5 Tangent Space Bumpmapping

Ähnlich wie im vorherigen Beispiel wird die Lichtintensität pro Pixel berechnet. Allerdings soll zusätzlich noch die Normale aus einer Normalmap für diffuses und specular Licht mit berücksichtigt werden. Es wird davon ausgegangen, daß sich die Tangenten im Vertex Attribut 6 und die Normalen im Vertex Attribut 2 befinden. Die Kamera und Lichtposition befinden sich in den zugehörigen Programm Parametern. Texture 0 ist die Normalmap. Texture 1 soll die diffuse Basistexture enthalten und die Specularmap wird in Texture 2 gespeichert.

Das Vertex Program berechnet die Vektoren von Punkt zum Licht und die Vektoren von der Kamera zum Vertex und bringt sie in den Tangentenraum.

```
param
    LightPos, CamPos:vec4;
varying
    texcoord, LightVec, LightVec2, ViewVec:vec4;
var
    lv:vec4;
begin
    result.position:=MatrixMult4(state.matrix.mvp,vertex.position);
    result.color:=vertex.color;
    texcoord:=vertex.attrib[8];
    lv:=LightPos-vertex.position;
    LightVec:=MatrixMult3(Matrix(vertex.attrib[6],cross(vertex.attrib[6],
                                                    vertex.attrib[2]),vertex.attrib[2]),lv);
    LightVec2:=lv;
    lv:=CamPos-vertex.position;
    ViewVec:=MatrixMult3(Matrix(vertex.attrib[6],cross(vertex.attrib[6],
                                                    vertex.attrib[2]),vertex.attrib[2]),lv);
end;
```

Aus diesem Vertex Program wird folgenden Code generiert:

```
!!ARBvp1.0
OUTPUT texcoord=result.texcoord[0];
OUTPUT LightVec=result.texcoord[1];
OUTPUT LightVec2=result.texcoord[2];
OUTPUT ViewVec=result.texcoord[3];
PARAM LightPos=program.local[0];
PARAM CamPos=program.local[1];
TEMP lv;
TEMP temp0;
DP4 result.position.x,state.matrix.mvp.row[0],vertex.position;
DP4 result.position.y,state.matrix.mvp.row[1],vertex.position;
DP4 result.position.z,state.matrix.mvp.row[2],vertex.position;
DP4 result.position.w,state.matrix.mvp.row[3],vertex.position;
MOV result.color,vertex.color;
MOV texcoord,vertex.attrib[8];
SUB lv,LightPos,vertex.position;
XPD temp0,vertex.attrib[6],vertex.attrib[2];
DP3 LightVec.x,vertex.attrib[6],lv;
DP3 LightVec.y,temp0,lv;
DP3 LightVec.z,vertex.attrib[2],lv;
MOV LightVec2,lv;
SUB lv,CamPos,vertex.position;
DP3 ViewVec.x,vertex.attrib[6],lv;
DP3 ViewVec.y,temp0,lv;
DP3 ViewVec.z,vertex.attrib[2],lv;
END
```

Man kann schön erkennen, daß die XPD Anweisung nicht zweimal generiert wird.

Das Fragment Program erhält die Varying Variablen vom Vertex Program und berechnet dann die Farbwerte der einzelnen Pixel. Der Licht und Halbvektor wird pro Pixel normalisiert. Die normalise Anweisung ersetzt hier die Normalisationscubemap.

```
param
    LightSize,LightColor,LightSpecular:vec4;
varying
    texcoord,LightVec,LightVec2,ViewVec:vec4;
var
    color,halfvec,specular,bump:vec4;
begin
    bump:=NormalMap(0,texcoord);
    color:=sat(dot3(bump,normalize(LightVec)))*Texture2D(1,texcoord)*
        LightColor;
    halfvec:=normalize(LightVec+ViewVec);
    specular:=power(sat(dot(halfvec,bump)),8)*LightSpecular*
        Texture2D(2,texcoord);
    result.color:=(color+specular)*lightatten(LightVec2,LightSize)*bump.w;
end;
```

Das Fragment Program wird zu folgendem Code kompiliert:

```
!!ARBfp1.0
ATTRIB texcoord=fragment.texcoord[0];
ATTRIB LightVec=fragment.texcoord[1];
ATTRIB LightVec2=fragment.texcoord[2];
ATTRIB ViewVec=fragment.texcoord[3];
PARAM LightSize=program.local[0];
PARAM LightColor=program.local[1];
PARAM LightSpecular=program.local[2];
TEMP color;
TEMP halfvec;
TEMP specular;
TEMP bump;
TEMP temp0;
TEMP temp1;
TEX temp0,texcoord,texture[0],2D;
MAD bump,temp0,{2.0,2.0,2.0,1.0},{-1.0,-1.0,-1.0,0.0};
DP3 temp0.w,LightVec,LightVec;
RSQ temp0.w,temp0.w;
MUL temp0.xyz,temp0.w,LightVec;
DP3_SAT temp0,bump,temp0;
TEX temp1,texcoord,texture[1],2D;
MUL temp1,temp0,temp1;
MUL color,temp1,LightColor;
ADD temp1,LightVec,ViewVec;
DP3 temp1.w,temp1,temp1;
RSQ temp1.w,temp1.w;
MUL halfvec.xyz,temp1.w,temp1;
DP3_SAT temp1,halfvec,bump;
POW temp1,temp1.x,{8,8,8,8}.x;
MUL temp1,temp1,LightSpecular;
TEX temp0,texcoord,texture[2],2D;
MUL specular,temp1,temp0;
ADD temp0,color,specular;
MUL temp1,LightVec2,LightSize;
DP3 temp1,temp1,temp1;
ADD temp1,{1,1,1,1},-temp1;
MUL temp1,temp0,temp1;
MUL result.color,temp1,bump.w;
END
```

An diesem längerem Beispiel kann man deutlich die Verinfachung bei der Entwicklung durch den Shader Compiler erkennen.

10.6 Volumetrisches Licht

Mit volumetrischem Licht oder Nebel kann man schöne Effekte erzeugen, weil die Helligkeit eines Pixels von dem Weg durch das Volumen abhängt. Der Einfachheit halber nimmt man meistens als Volumen eine Kugel. Ob es sich um Nebel oder um Licht handelt bestimmt letztlich nur der Blendmodus. Additives Blending erhöht die Helligkeit je nachdem wie lang der Weg durch das Volumen war und kann eine Art Ergänzung zu Lensflares sein. Mit Alpha Blending kann man volumetrisches Nebel erzeugen. Besonders schön wurde volumetrisches Licht in Unreal 1 eingesetzt. Dort wurden allerdings für jede Fläche auf der CPU eine Fogmap berechnet, die ähnlich wie Lightmaps einen Helligkeitswert enthalten. Es ist aber auch möglich diesen Wert für jedes Fragment in einem Fragment Programm zu berechnen. Ein Beispiel gibt es dafür in der Volumetric Lighting II Demo von Humus (<http://esprit.campus.luth.se/~humus/>). Die folgenden fxPascal Programme basieren auf den ARB Programmen aus der Demo.

Das Vertex Programm berechnet nur die Vertex Position und sorgt dafür, daß die Position der Fragmentes interpoliert wird.

```
param
    ViewPos:vec4;
varying
    FragmentPos:vec4;
begin
    result.position:=MatrixMult4(state.matrix.mvp,vertex.position);
    FragmentPos:=vertex.position-ViewPos;
end;
```

Das Fragment Programm ermittelt die Entfernung der Geraden durch Kamera und Fragment und benutzt diese Entfernung um den Helligkeitswert zu bestimmen. Anders als in der VolumetricLightingII Demo wird hier nicht der Kehrwert der Entfernung genommen, sondern die Entfernung zum Quadrat wird von 1 subtrahiert, damit das Licht nicht unendlich weit scheint. Diese Näherung wurde auch schon beim PerPixel Lighting benutzt, damit die Lichter einen festen Radius haben und man die Beleuchtung auf einen bestimmten Radius reduzieren kann. Zum Schluß wird der Wert noch acht Mal mit sich selbst multipliziert damit der Helligkeitsverlauf steiler ist. Eventuell könnte es hier günstiger sein die power Funktion zu nutzen. LightSize ist hier der Kehrwertes der Lightgröße zum Quadrat. LightViewPos ist die Lichtposition minus die Kameraposition. Diese Berechnung könnte man auch im Vertex Programm durchführen. Es ist aber effizienter das nur einmal zu berechnen und dann als Programm Parameter zu übergeben.

```
param
    LightViewPos,LightVolumeColor,LightSize:vec4;
varying
    FragmentPos:vec4;
var
    v1,v2,p:vec4;
begin
    v1:=dot(FragmentPos,FragmentPos);
    v2:=dot(FragmentPos,LightViewPos);
    p:=sat(v2.x/v1.x)*FragmentPos-LightViewPos;
    p:=p*LightSize;
    v1:=sat(1-dot(p,p));
    result.color:=LightVolumeColor*sqr(v1*v1*v1*v1);
end;
```

Die beiden Programme werden zu folgenden ARB Programmen kompiliert:

Vertex Programm:

```
!!ARBvp1.0
OUTPUT FragmentPos=result.texcoord[0];
PARAM ViewPos=program.local[0];
DP4 result.position.x,state.matrix.mvp.row[0],vertex.position;
DP4 result.position.y,state.matrix.mvp.row[1],vertex.position;
DP4 result.position.z,state.matrix.mvp.row[2],vertex.position;
DP4 result.position.w,state.matrix.mvp.row[3],vertex.position;
SUB FragmentPos,vertex.position,ViewPos;
END
```

Fragment Programm:

```
!!ARBfp1.0
ATTRIB FragmentPos=fragment.texcoord[0];
PARAM LightViewPos=program.local[0];
PARAM LightVolumeColor=program.local[1];
PARAM LightSize=program.local[2];
TEMP v1;
TEMP v2;
TEMP p;
TEMP temp0;
DP3 v1,FragmentPos,FragmentPos;
DP3 v2,FragmentPos,LightViewPos;
RCP temp0.w,v1.x;
MUL_SAT temp0.w,temp0.w,v2.x;
MAD p,temp0.w,FragmentPos,-LightViewPos;
MUL p,p,LightSize;
DP3 temp0,p,p;
SUB_SAT v1,{1,1,1,1}.w,temp0;
MUL temp0,v1,v1;
MUL temp0,temp0,v1;
MUL temp0,temp0,v1;
MUL temp0,temp0,temp0;
MUL result.color,LightVolumeColor,temp0;
END
```

Man kann erkennen, daß der SAT Funktionsaufruf in Fragment Programmen direkt an die Anweisung angehängt wird.