AUF CD

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Praxis/Programmierung/PC Underground*.

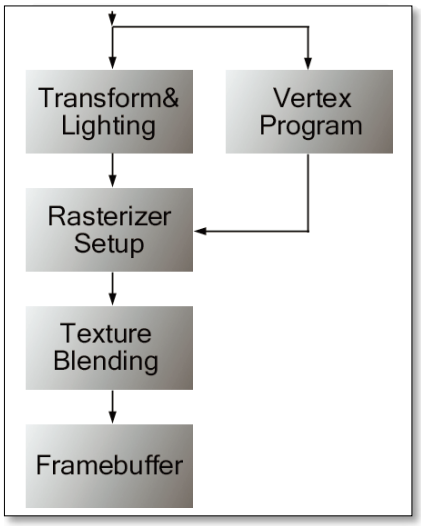
Vertex-Shader in OpenGL

Assemblierte Schönheit

Mit Vertex-Shadern machen Sie Ihrer Grafikkarte Beine und bestimmen selbst, wie Vertices **transformiert und beleuchtet** werden.

CARSTEN DACHSBACHER

Ein unaufhaltsamer Trend bei moderner Grafik-Hardware ist eine immer größere Geschwindigkeit bei der Berechnung und Darstellung von 3D-Grafik. Das Rendering von virtuellen Szenen läuft immer nach demselben Schema, der Grafik-Pipeline, ab, wie Sie im Bild unten sehen können.



JEDER VERTEX muss die Grafik-Pipeline durchlaufen.

Die Grafik-Pipeline transformiert die Geometrie abhängig von der Lage der Objekte und der Kamera und berechnet anschließend die Beleuchtung. Das Resultat sind im allgemeinen Dreiecke, deren Eckpunkte (Vertices) mit Attributen wie Textur-Koordinaten, Farb- und Transparenz-Werten ausgestattet sind. Der Rasterizer, der Teil der Grafik-Hardware, der das Zeichnen verantwortet, bekommt diese Daten und rendert entsprechend in den Framebuffer. Neuere nVidia-Hardware und die Radeon-Karten von ATI gestatten Programmierern, die Transform- and Lighting-Stufe der Grafik-Pipeline mit einer eigenen Assembler-Sprache zu gestalten. Diese Schnittstelle heißt in OpenGL *Vertex Programs* und in DirectX 8 *Vertex-Shader*. (Vertex Shader passt nicht ganz, da sich das Wort *Shading* eigentlich auf Pixel und nicht auf Vertices bezieht.)

In diesem Artikel werden Sie diese Assembler Sprache kennenlernen und erfahren, wie Sie sie mit nVidia-Grafikkarten (GeForce) und OpenGL ab Version 1.2 einsetzen können. Die aktuelle OpenGL Version ist 1.3.

Als Beispiel dienen zwei von vielen Einsatzgebieten: Sie schreiben eine eigene Beleuchtungsberechnung und verwenden die *Vertex Programs*, um so genannte Billboards auszurichten. Billboards sind Polygone, die immer senkrecht zur Blickrichtung liegen, also zum Betrachter hinzeigen.

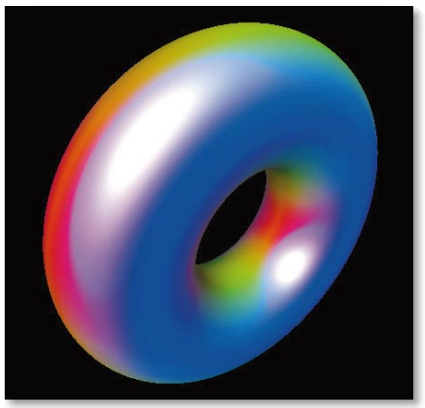
Das Vertex-Programm

Mit Vertex-Programmen haben Sie die volle Kontrolle über die Transform- and Lighting-Stufe der Grafik Pipeline. Damit können Sie komplexe Operationen mit den Vertices in der GPU (Graphic Processing Unit) Ihrer Grafikkarte ausführen lassen. So entlasten Sie die CPU des Rechners, die Sie damit für andere Aufgaben physikalisch oder für Simulationen freimachen. Sie geben der CPU also mehr Zeit für physikalische Berechnungen wie Partikel-Bewegungen.

Sie verwenden Vertex-Programme, um die Beleuchtung zu berechnen, für *Skinning*- und *Blending*-Techniken, also das Überblenden von Bewegungsabläufen bei der Animation von Charakteren, und um Texture-Koordinaten zu generieren. Außerdem können Sie beliebige Texture-Matrix-Berechnungen durchführen oder die Vertices durch weitere Rechenschritte modifizieren.

Sie schreiben Ihr Vertex-Programm in einer speziellen, mächtigen SIMD-Assemblersprache (Single Instruction Multiple Data). Als Eingabedaten dienen eine Reihe von Variablen, deren Inhalt Sie von außen festlegen können, und ein nicht transformierter, nicht beleuchteter Vertex inklusive einiger Attribute.

Die Ausgabe muss die transformierten Koordinaten enthalten und optional die Beleuchtung, die Texture-Koordinaten, Fog-Koordinaten (Nebel) und Point



DIE BELEUCHTUNG UND FARBE dieses Torus wurde von einem Vertex-Programm berechnet.

Sizes (die Größe der Punkte beim Rendering von *GL_POINTS*). Ein Vertex-Programm bearbeitet also immer nur einen Vertex. Es werden keine zusätzlichen Vertices erzeugt oder gelöscht, es gibt keine topologischen Informationen zu benachbarten Vertices, die vielleicht zusammen ein Dreieck bilden könnten (Nachbarschaftsinformation).

Die Vertex-Attribute

Die Vertex-Attribute sind 16 Register, die aus je vier Float-Werten bestehen, also ein Vektor sind. Sie enthalten jeweils die Daten des Vertex, der transformiert werden soll und mit einem *read-only*-Attribut versehen ist. Eine Instruktion eines Vertex-Programms darf jeweils nur eines dieser Register enthal-

DIE VERTEX-ATTRIBUTE (REGISTER)

Register	Name	normale Belegung
v[0]	v[OPOS]	Vertex-Koordinate
v[1]	v[WGHT]	Vertex Weight (für Blending)
v[2]	v[NRML]	Normale
v[3]	v[COL0]	primäre Farbe
v[4]	v[COL1]	sekundäre Farbe
v[5]	v[FOGC]	Fog-Koordinate
v[6]	—	—
v[7]	—	—
v[8]	v[TEX0]	Textur-Koordinate 0
v[9]	v[TEX1]	Textur-Koordinate 1
v[10]	v[TEX2]	Textur-Koordinate 2
v[11]	v[TEX3]	Textur-Koordinate 3
v[12]	v[TEX4]	Textur-Koordinate 4
v[13]	v[TEX5]	Textur-Koordinate 5
v[14]	v[TEX6]	Textur-Koordinate 6
v[15]	v[TEX7]	Textur-Koordinate 7

ten, aber sie darf zweimal dasselbe Register verwenden. Die Register finden Sie in der Tabelle.

Sie können auf zwei Arten darauf zugreifen: mit der Indizierung durch eine Zahl von 0 bis 15 oder durch die Kürzel in der zweiten Spalte, die in der dritten Spalte beschrieben sind. Die Beschreibungen bezeichnen die normale Belegung der Register. Beispielsweise befindet sich die Normale eines Vertex im Register *v[2]*. Da Sie dies als Programmierer beliebig festlegen können, müssen diese Bezeichnungen nicht verbindlich sein. Sie können diese Vertex-Attribute-Register mit beliebigen Werten pro Vertex füllen, also mit Indizes, Vektoren oder anderen Parametern.

Das Ergebnis des Vertex-Programms wird in die 15 Vertex-Result-Register geschrieben. Darin ist die Information enthalten, die die Rasterizer-Einheit der Grafik-Hardware anschließend für das Rendering enthält.

Die Result-Register haben jeweils eine Bezeichnung, an die Sie sich halten müssen: So befinden sich die tranformierten Koordinaten immer im Register *o[HPOS]*

DIE VERTEX-RESULT-REGISTER

Register-name	Beschreibung	Interpretation
o[HPOS]	homogene Koordinaten	(x,y,z,w)
o[COL0]	primäre Farbe (vorne)	(r,g,b,a)
o[COL1]	sekundäre Farbe (vorne)	(r,g,b,a)
o[BFC0]	primäre Farbe (hinten)	(r,g,b,a)
o[BFC1]	sekundäre Farbe (hinten)	(r,g,b,a)
o[FOGC]	Fog Koordinaten	(x,y,z,w)
o[PSIZ]	Point Size	(x,y,z,w)
o[TEX0]	Texture Koordinaten Set 0	(s,t,r,q)
o[TEX1]	Texture Koordinaten Set 1	(s,t,r,q)
o[TEX2]	Texture Koordinaten Set 2	(s,t,r,q)
o[TEX3]	Texture Koordinaten Set 3	(s,t,r,q)
o[TEX4]	Texture Koordinaten Set 4	(s,t,r,q)
o[TEX5]	Texture Koordinaten Set 5	(s,t,r,q)
o[TEX6]	Texture Koordinaten Set 6	(s,t,r,q)
o[TEX7]	Texture Koordinaten Set 7	(s,t,r,q)

DIE BEFEHLE DER VERTEX-PROGRAMME

Befehl	Beschreibung
MOV dest, src0	Kopiert den Inhalt von *src0* nach *dest*
MUL dest, src0, src1	Komponentenweise Multiplikation
ADD dest, src0, src1	Komponentenweise Addition
MAD dest, src0, src1, src2	Addiert die Werte von *src2* zu dem Multiplikationsergebnis von *rc0* und *src1*
RCP dest, src0.C	Berechnet das Reziproke zu einer Komponente *C* von *src0*
RSQ dest, src0.C	Berechnet die inverse Wurzel zu einer Komponente von *src0*
DP3 dest, src0, src1	Skalarprodukt zweier Vektoren/3 Komponenten *(x, y, z)*
DP4 dest, src0, src1	Skalarprodukt zweier Vektoren/4 Komponenten *(x, y, z, w)*
MIN dest, src0, src1	Komponentenweises Minimum bilden
MAX dest, src0, src1	Komponentenweises Maximum bilden
SLR dest, src0, src1	Komponentenweiser Vergleich auf *kleiner als*. Ist eine Komponente von *src0* kleiner als die von *src1*, dann wird die entsprechende Komponente in *dest* auf *1.0* — sonst *0.0* gesetzt.
SGE dest, src0, src1	Komponentenweiser Vergleich auf *größer gleich* (siehe *SLR*)
EXP dest, src0.C	Berechnet $2^{src0.C}$
LOG dest, src0.C	Berechnet Logarithmus zur Basis *2 von src0.C*
ARL A0.x, src0.C	Laden des Adressregisters
LIT dest, src0	Beleuchtungsberechnung
src0.x	Skalarprodukt für diffuse Beleuchtung *(N*L)*
src0.y	Skalarprodukt für spiegelnde (specular) Beleuchtung *(N*H)*
src0.w	Phong Exponent, Resultat: Koeffizient für ambiente *(dest.x)*, diffuse *(dest.y)* und spiegelnde *(specular) (dest.z)* Beleuchtung
DST dest, src0.C, src1.D	Distance Vector: *src0.C* = d^2, *src1.D* = *1/d*, Resultat: *dest* = $(1, d, d^2, 1/d)$

Die Vertex-Programme

Ein Vertex-Programm besteht aus bis zu 128 SIMD-Instruktionen. Die Befehle sind nach einem festen Schema aufgebaut, wobei eckige Klammern jeweils einen optionalen Teil ausweisen:

```
Opcode dst, [-]src0 [,[-]src1
[,[-]src2]]; #Kommentar
```

dst ist das Zielregister *src0*, *src1* und *src2* sind Quellregister. Der Inhalt jedes Quellregisters kann auf Wunsch negiert werden, zum Beispiel beim *MOV*-Befehl, der den Inhalt eines Registers in ein anderes kopiert:

```
MOV R1, R2 oder MOV R1,
-R2
```

Weiterhin können Sie die Komponenten der Register vertauschen:

```
MOV R1, R2.wzyx
```

Dabei passiert Folgendes:

```
R1.x = R2.w
R1.y = R2.z
...
```

Bei *MOV R1.xw, R2* werden nur die *x*- und *w*-Komponente von *R1* mit den entsprechenden Werten von *R2* gefüllt. Die *y*- und *z*-Komponente bleiben unberührt. Eine Liste der Befehle finden Sie in der Tabelle.

SIMD-Befehle verfügen nicht über Verzweigungsbefehle. Für diese Operationen brauchen Sie die verschiedenen Berechnungszweige, die Sie mit den Maskierungen von *SLR* und *SGE* multiplizieren und addieren.

Für jeden Vertex rufen Sie ein Vertex-Programm auf. Sie können dabei auf die Attribute sowie Result-Register zurückgreifen. Weiterhin gibt es 12 temporäre Register *R1* bis *R11* (*read/write*), jeweils aus vier Floats bestehend, und die Programmparameter, die *96* Register mit vier Floats (*c[0]* bis *c[95]*) aufweisen. Die Programmparameter sind *read-only*, die Sie außerhalb des Renderns (also vor oder nach dem *glBegin/glEnd*-Befehlspaar) modifizieren können.

Es ist notwendig, dass Sie zur Transformation der Koordinaten die jeweils gültige *Modelview* und *Projection Matrix* von OpenGL kennen. Dazu nutzen Sie das *Tracking*-Verfahren. Damit legen Sie fest, dass die Vektoren der Matrizen in bestimmten Programmparametern und Registern gespeichert sind. Folgende Zeilen legen dies fest:

```
glTrackMatrixNV
( GL_VERTEX_PROGRAM_NV, 4,
GL_MODELVIEW, GL_IDENTITY_NV );
glTrackMatrixNV
( GL_VERTEX_PROGRAM_NV, 20,
GL_MODELVIEW, GL_INVERSE_NV );
```

Diese Zeilen besagen, dass die Register *c[4]*, *c[5]*, *c[6]* und *c[7]* die Modelview *Matrix* enthalten und *c[20]* bis *c[23]* die Inverse dieser Matrix. Andere Werte speichern Sie mit dem folgenden Befehl:

```
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
16, 1, 2, 3, 4);
```

Damit enthält das Register *c[16]* den Vektor *(1, 2, 3, 4)*.

Ein Vertex-Programm bekommt in OpenGL einen Integer-Wert als Bezeichnung zugeordnet. Diesen Wert erhalten Sie – ähnlich wie bei der Verwaltung von Texturen – durch den Befehl

```
glGenProgramsNV
( int n, int *ids )
```

Ein Vertex-Programm speichern Sie im Quelltext mit einem String und übergeben diesen mit

```
glLoadProgramNV( enum target,
int id, int length,
const char *program )
```

an OpenGL:

```
const unsigned char program[] =
{...};
int vertexProgram;
glGenProgramsNV( 1, &vertexPro-
gram );
glBindProgramNV( GL_VERTEX_PRO-
GRAM_NV, vertexProgram );
glLoadProgramNV( GL_VERTEX_PRO-
GRAM_NV, vertexProgram, strlen(
program ), program );
```

Sobald Sie nun mit *glEnable(GL_VERTEX_PROGRAM_NV)* die Vertex-Programme aktiviert haben, nutzen Sie die gesamte *Transform*- und *Lighting*-Stufe von OpenGL für Ihr eigenes Programm. Jetzt müssen Sie nur noch spezifizieren, welche Daten in den Vertex-Attribut-Registern gespeichert werden sollen. Dazu brauchen Sie zwei Befehle. Zuerst aktivieren Sie einen Stream von Daten mit

```
glEnableClientState( GL_VERTEX_
ATTRIB_ARRAY0_NV );
```

Die zu übermittelnden Daten übergeben Sie mit dem Befehl unten. Dabei ist der erste Parameter der Index des Streams, den Sie soeben aktiviert haben. Der zweite Parameter gibt die Anzahl der Komponenten an. Bei folgendem Beispiel werden die *x*-, *y*- und *z*-Komponenten von *v[HPOS]* mit den Koordinaten aus *vertexArray* gefüllt:

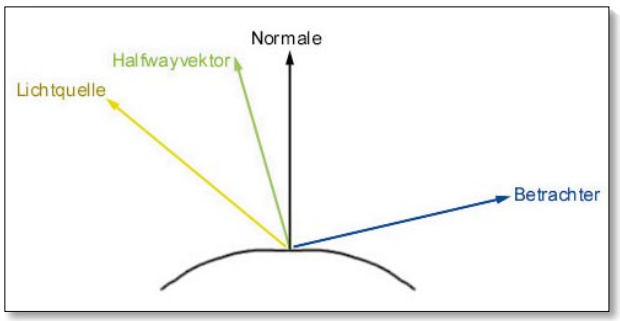
```
glVertexAttribPointerNV
( 0, 3, GL_FLOAT,
sizeof(VERTEX3D), &vertexArray);
```

Weiterhin gibt es eine spezielle Variante von Vertex-Programmen, die so genannten Vertex-State-Programs. Diese dürfen die Parameter-Register modifizieren, müssen aber explizit – von Ihrem Programm – ausgeführt werden.

```
// upload
int vertexStateProgram;
glGenProgramsNV( 1, &vertexState-
Program );
glLoadProgramNV
( GL_VERTEX_STATE_PROGRAM_NV,
vertexStateProgram,
strlen(stateProgram),
stateProgram );

// ausführen
float nulldata[4] = {0.0f, 0.0f,
0.0f, 0.0f};
glExecuteProgramNV(
GL_VERTEX_STATE_PROGRAM_NV, ver-
texStateProgram,  (float*)nullda-
ta );
```

Die speziellen OpenGL-Befehle sind Erweiterungen des ursprünglichen OpenGL Standards, und Sie müssen überprüfen, ob sie zur Verfügung stehen. Dazu suchen Sie nach der *NV_vertex_program*-Erweiterung und holen sich mit *wglGetProcAdress(...)* die Adressen der neuen Befehle. Die entsprechenden Aufrufe und Konstanten finden Sie im Beispiel-Programm auf der Heft CD.



DIE VEKTOREN im Phong-Beleuchtungsmodell.

Das erste Vertex-Program

Los geht's mit Ihrem ersten Vertex-Programm: Dieses soll einen Vertex von seinen angegebenen Koordinaten (mit *glVertex*) ins Koordinatensystem der Betrachterkamera transformieren. Kommentare innerhalb des Vertex-Programms, das immer mit der Kennung *!!VP 1.0* beginnt, kennzeichnen Sie mit einem #-Symbol. Für die Tranformation benötigen Sie die Modelview- und die Projection-Matrix, die in den Parameter-Registern gespeichert werden.

```
const unsigned char simpleSha-
der[]=
{
„!!VP1.0 \
# Transformation Objectspace-
>Worldspace
DP4 R0.x,v[OPOS],c[0]; \
DP4 R0.y,v[OPOS],c[1]; \
DP4 R0.z,v[OPOS],c[2]; \
DP4 R0.w,v[OPOS],c[3];\
# Transformation Worldspace->Ca-
meraspace
DP4 R1.x,R0,c[4]; \
DP4 R1.y,R0,c[5];\
DP4 R1.z,R0,c[6];\
DP4 R1.w,R0,c[7];\
# und speichern
MOV o[HPOS],R1;\
# Farbwert einfach durchreichen
MOV o[COL0], v[COL0];\
END“
};

glEnable(GL_VERTEX_PROGRAM_NV);
glBindProgramNV(GL_VERTEX_PRO-
GRAM_NV, 1);
glLoadProgramNV(GL_VERTEX_PRO-
GRAM_NV, 1, strlen(simpleShader),
simpleShader );

glTrackMatrixNV(GL_VERTEX_PRO-
GRAM_NV, 0, GL_MODELVIEW,
GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PRO-
GRAM_NV, 4, GL_PROJECTION,
GL_IDENTITY_NV);
```

Als nächstes fügen Sie eine eigene Beleuchtungsberechnung in das Vertex-Programm ein. Als Beispiel wollen wir Ihnen hier eine bunt eingefärbte Oberfläche mit Beleuchtung mit dem Phongmodell vorstellen.

Dazu benötigen Sie weitere Parameter:

```
glTrackMatrixNV

(GL_VERTEX_PRO-
GRAM_NV, 8,
GL_MODELVIEW,
GL_INVERSE_TRANSPOSE_NV );
// Licht Richtung

glProgramParameter4fNV
(GL_VERTEX_PROGRAM_NV, 32, 0, 0,
1, 1);
// Halfspace Vektor H
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV, 33,  0,
0, 1, 1 );
// diffus-ambienter Koeffizient
Oberfläche
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
   35,  0.8, 0.2, 0, 0 );
// Farbe der Highlights
glProgramParameter4fNV
(GL_VERTEX_PROGRAM_NV,
 36, 1.0, 1.0, 1.0, 1.0 );
// Phongexponent
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
 38,  30.0, 0, 0, 0 );
```

Die Farbe der Objektoberfläche wird schön bunt, indem Sie den Betrag der Normalen als Farbe verwenden. Den Betrag berechnen Sie mit einem kleinen Trick per *MAX*-Befehl:

```
MOV R4, v[NRML];
MAX R4, R4, -R4;
```

Jetzt berechnen Sie Schritt für Schritt die Beleuchtung. Zunächst transformieren Sie die Normale mit der inversen Modelview-Matrix in dasselbe Koordinatensystem wie die Lichtrichtung:

```
DP3 R2.x, c[8], v[NRML];
DP3 R2.y, c[9], v[NRML];
DP3 R2.z, c[10], v[NRML];
```

Anschließend berechnen Sie die Skalarprodukte der Lichtrichtung bzw. des Halfspace-Vektors und der Normalen und schließen die Vorbereitung der Beleuchtungsberechnung ab, indem Sie in *R3.z* den Phong-Exponenten speichern.

```
DP3 R3.x, c[32], R2;
DP3 R3.y, c[33], R2;
MOV R3.w, c[38].x;
LIT R4, R3;
```

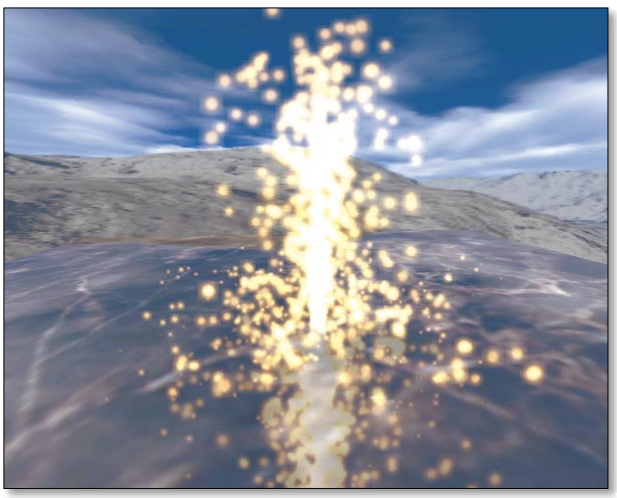
Jetzt können Sie den resultierenden Farbwert an Hand des berechneten Shadings bestimmen:

```
MAD R5, c[35].x, R4.y, c[35].y;
# R5.x = Diffuse*(N*L)+Ambient
MUL R6.xyz, c[36], R4.z;
# R6 = SpecularFarbe*Koeff.
MAD o[COL0].xyz, R4, R5.x, R6;
# Farbe = R4*(Amb+Diff)+Specular-
Farbe
```

Modifikation der Vertex-Koordinaten

Unser zweites Anwendungsbeispiel erledigt nicht nur die Transformation und Beleuchtung, sondern modifiziert die Lage der Vertices selbst. Es soll so genannte Billboards (Polygone), die immer zum Betrachter hinzeigen, automatisch ausrichten.

Für jedes Billboard verwenden Sie ein Quadrat, also vier Vertices, und zeichnen diese mit *GL_QUADS*. Mit den Billboards wollen wir eine Partikelfontäne darstellen.



DIE PARTIKEL dieser Fontäne werden mit Billboards dargestellt.

Für einen Partikel ist jeweils nur der Ort bekannt. Es ist also die Aufgabe des Vertex Programms, aus der Position ein Quadrat im Raum zu platzieren. Die benötigten Vektoren, die vom Betrachter aus nach rechts und nach oben zeigen, sind die *Right*- und *Up*-Vektoren der Modelview-Matrix. Da die Vertex-Programme keine neuen Vertices erzeugen können, legen Sie von vornherein eine Liste von Vertices, für jeden Partikel vier Stück, an. Für jeden Knoten speichern Sie zusätzlich zur Koordinate weitere Daten. Zunächst legen Sie die Programmparameter fest:

```
glTrackMatrixNV
(GL_VERTEX_PROGRAM_NV, 0,
GL_MODELVIEW, GL_IDENTITY_NV);
glTrackMatrixNV
(GL_VERTEX_PROGRAM_NV, 4,
GL_MODELVIEW_PROJECTION_NV,
GL_IDENTITY_NV);
// texture koordinaten
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
  24,  0, 0, 0, 0 );
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
  25,  1, 0, 0, 0 );
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
  26,  1, 1, 0, 0 );
glProgramParameter4fNV
( GL_VERTEX_PROGRAM_NV,
  27,  0, 1, 0, 0 );
```

Rufen Sie vor dem Zeichnen das Vertex-State-Programm auf. Dieses verwendet die *Up*- und *Right*-Vektoren, um die vier Vektoren zu bilden, die das Billboard aufspannen:

```
!!VSP1.0
MOV R0.xyz,c[0];
MOV R1.xyz,c[1];
ADD c[20], -R0, -R1; #links oben
ADD c[21],  R0, -R1; #rchts oben
ADD c[22],  R0,  R1; #rchts unt.
ADD c[23], -R0,  R1; #lnks unten
END
```

Die Speicherung der Vertices erfolgt mit der Struktur, wobei pro Partikel vier Vertices notwendig sind:

```
typedef struct
{
  VERTEX3D pos;
  VERTEX3D vdata;
}BILLBOARDVERTEX;
BILLBOARDVERTEX *particleVertex;
```

In *vdata.x* speichern Sie für jeden Vertex, ob es sich um den ersten, zweiten, dritten oder vierten Vertex des Billboards handelt, die Werte *0*, *1*, *2* oder *3*. Diesen Index benötigen Sie im Vertex-Programm, um die oben berechneten Vektoren und die Texturkoordinaten zu indizieren. In *vdata.y* speichern Sie einen Faktor für die Größe des Partikels und in *vdata.z* seine Helligkeit.

Diese Daten wird das zweite Vertex Programm verwenden. In *pos* speichern Sie die Koordinate, wobei Sie für jeden Vertex eines Billboards dieselbe Koordinate verwenden. Diese Daten übermitteln Sie wie folgt:

```
glEnableClientState
( GL_VERTEX_ATTRIB_ARRAY0_NV );
glEnableClientState
( GL_VERTEX_ATTRIB_ARRAY1_NV );

glVertexAttribPointerNV
( 0, 3, GL_FLOAT, sizeof
(BILLBOARDVERTEX),
&particleVertex[0].pos );
glVertexAttribPointerNV
( 1, 3, GL_FLOAT,
sizeof(BILLBOARDVERTEX),
&particleVertex[0].vdata );

// zeichnen
glDrawElements
( GL_QUADS, nParticles*4,
GL_UNSIGNED_INT, particleIndex );

glDisableClientState
( GL_VERTEX_ATTRIB_ARRAY0_NV );
glDisableClientState
( GL_VERTEX_ATTRIB_ARRAY1_NV );
```

Dieses Vertex-Programm übernimmt die Arbeit:

```
!!VP1.0
ARL A0.x,v[1].x;
#Index laden
MUL R0, c[A0.x+20], v[1].y;
#Vektor indizieren
#und Vektor skalieren
ADD R1, R0, v[0];
#auf die Koordinate addieren

DP4 o[HPOS].x,R1,c[4];
#Koordinate transformieren
DP4 o[HPOS].y,R1,c[5];
DP4 o[HPOS].z,R1,c[6];
DP4 o[HPOS].w,R1,c[7];

MOV R1, v[1];
MUL o[COL0], v[COL0],R1.z;
#helligkeit der farbe
MOV o[TEX0].xy,c[A0.x+24];
#Texture-Koordinate kopieren

END
```

Das Resultat des Vertex-Programms sehen Sie im nebenstehenen Bild. Die optischen Spielereien erreichen Sie mit einer spiegelnden Fläche. Der wolkige Hintergrund bereichert den lebendigen Eindruck. Die vollständige Implementation inklusive der Partikelroutine finden Sie auf der Heft-CD. ET

Lesen Sie zur Programmierung von Grafikkarten: http://developer.nvidia.com