workshop

Direct-3D-9 – Teil III

Grafik durch die Pipeline jagen

Direct3D bietet die einheitliche Schnittstelle, um moderne Grafikkarten anzusprechen. Sie programmieren die Grafikpipeline mit den Vertex und Pixel Shader. Dadurch eröffnet sich Ihnen eine Fülle von Grafikeffekten. Wenn Sie 3D-Modelle ins Wavefront-Format konvertierten, helfen Ihnen Modeling-Programme.

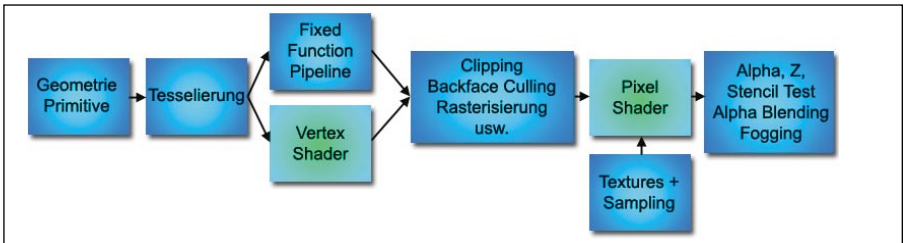
Carsten Dachsbacher

In den ersten beiden Teilen unseres Direct3D-9-Tutorials haben Sie die Grundlagen für stabile und effiziente Direct3D-Programme kennen gelernt. Für imposante 3D-Effekte benötigen Sie aber mehr: zum Einen Daten und 3D-Modelle, zum Anderen die Fähigkeit, moderne Grafikkarten zu programmieren. Und genau um diese beiden Punkte kümmern Sie sich in diesem Artikel!

Im Falle der hardwarebeschleunigten 3D-Grafik, gibt es zwei Wege, wie die Geometrieverarbeitung stattfinden kann. In der ersten, bereits bekannten, Variante arbeiten Sie mit der so genannten *Fixed Function Pipeline*. Dabei handelt es sich um den Teil der GPU, der die herkömmliche Transformations- und Beleuchtungsberechnung durchführt. Die Funktionalität ist hier fixiert. Sie können lediglich die Eingabedaten festlegen, also z.B. Renderstates, Lichtquellen und Materialparameter.

Vertex und Pixel Shader

Die Vertex Shader (im OpenGL Kontext Vertex Programs) können die Fixed Function Pipeline ersetzen. Anstatt Parameter zu setzen, um die Pipeline zu konfigurieren, schreiben Sie ein Vertex Shader Programm, das in der GPU ausgeführt wird. Ein solches Programm verarbeitet jeweils nur einen einzigen Vertex. Sie können damit keine Vertices erzeugen oder eliminieren. Solche Programme setzen Sie z. B. ein, um Koordinaten zu berechnen oder prozedural Blending oder Deformationen zu erzeugen. Weiterhin können Sie damit Farbwerte, Textur-Koordinaten, Nebeleffekte und Punktgrößen berechnen. Die Ausgabedaten bestehen zumindest aus einer Clip-Space Koordinate, d.h. Sie müssen die 3D-Transformation des Vertex vornehmen und optional Farbwerte sowie Textur-Koordinaten und dergleichen berechnen.



Programmierbar: Die Grafikpipeline bietet zwei Stellen, an denen Sie selbst das Geschehen bestimmen können: bei den Vertex und Pixel Shader.

Die Vertex Shader erlauben es Ihnen, eine Reihe von Grafikeffekten zu programmieren, die Sie bisher einzeln pro Bild mit der CPU berechnen mussten. Das Bild zeigt schematisch das Konzept der Vertex Shader (GeForce 3) an. Das Bild *Vertex Shader* verdeutlicht dies.

Unterschiedliche Grafikhardware bietet verschieden leistungsfähige Vertex Shader an. Je nach GPU-Modell und -Generation unterscheiden sich die Anzahl der zur Verfügung stehenden Instruktionen pro Programm, die Zahl der Register und der Befehlsumfang. Die neuesten Grafikkarten bieten inzwischen auch Schleifenbefehle an – bisher konnten Sie nur sequentiell alle Instruktionen abarbeiten!

Mit so genannten Pixel Shader (OpenGL Fragment Programs) programmieren Sie die Rechenwerte pro Pixel (bzw. Fragment), also von Farbwerten. Als solche Erweiterungen wie mit den nVidias Register Combiner eingeführt wurden, konnten Sie noch nicht von Programmierung sprechen – bestenfalls von Konfiguration: Sie konnten gerade einmal mehrere Berechnungseinheiten hintereinander schalten. Auch wenn damals schon die tatsächlichen Einschränkungen der Hardware durch eine Art Assemblersprache verbessert wurden, ist inzwischen ein Punkt erreicht, an dem die Hardware wirklich frei programmierbar ist. Diese Technik führten Karten wie die ATI Radeon 9500/9700 und nVidia GeForce FX GPUs ein, die Pixel Shader der Version 2.0 unterstützen.

Vertex Shader in Direct3D

Direct3D bietet im Gegensatz zu OpenGL (was sich allerdings mit OpenGL 2.0 erübrigen wird) eine einheitliche Schnittstelle, um programmierbare Grafikhardware anzusprechen. Allerdings müssen Sie abfragen, was Ihnen die verwendete GPU bietet. Wie bei jeder vergleichbaren Hardware verwenden Sie die Device Caps, um die entsprechende Information abzufragen. Um zu überprüfen, ob eine bestimmte Vertex Shader Version unterstützt wird, verwenden Sie folgende Zeilen:

```
D3DCAPS9 caps;
pD3D->GetDeviceCaps(
  D3DADAPTER_DEFAULT,
  D3DDEVTYPE_HAL, &caps );
if ( caps.VertexShaderVersion <
              D3DVS_VERSION(1,0) )
//    keine Vertex Shader
```

Wenn keine Vertex Shader unterstützt werden, dann schlägt der obige Test fehl, überprüft aber weiter, ob wenigstens Version 1.0 unterstützt wird. Wenn Sie eine GeForce 3, Radeon 8500 oder neuere Grafikkarte besitzen, wird zumindest Version 1.1 unterstützt, die schon beachtliche Optionen bietet. Mit Vertex Shader 2.0 programmieren Sie zusätzlich Schleifen. Version 3 hingegen gibt es bislang nur auf dem Papier – es gibt noch keine Grafikhardware, die diese unterstützt.

Im Folgenden beschränken wir uns, ohne den Bezug zur Allgemeingültigkeit zu verlieren, auf Vertex Shader 1.1. Wie Sie in Bild *Vertex Shader* bereits gesehen haben, verfügen Sie über eine bestimmte Anzahl von Registern: Ein- und Ausgabe, Temporär-, Konstanten- und Adress-Register. Jedes Register ist ein Vektor, der aus vier Floating Point Zahlen besteht. Mit den Konstanten-Registern können Sie Daten von Ihrer Applikation an den Vertex Shader übergeben. Die Zahl der dazu zur Verfügung stehenden Konstanten-Register ist wieder in der Routine *D3DCAPS9.MaxVertexShaderConst* enthalten. Einen Überblick über die Register erhalten Sie in den beiden Tabellen.

Inzwischen gibt es eine Reihe verschiedener Wege, ein Vertex Shader Programm anzugeben. Die klassische Variante, die auch dieser Artikel verwendet, setzt auf eine Art Assemblersprache. Andere Varianten wären beispielsweise nVidias Cg oder die High-Level Shading Language (HLSL) von Direct3D, in denen Sie in einer C Syntax programmieren. Solche Hochsprachen bieten vor allem den Vorteil, einfache Teile eines Programms wiederzuverwenden und modular zu programmieren. Für komplexere Grafikeffekte werden Sie auch auf diese Variante zurückgreifen.

Nun geht es darum, Ihren ersten Vertex Shader zu programmieren. Den vollständigen Befehlssatz finden Sie am einfachsten, indem Sie in der DirectX9-Hilfe im Index *vertex shader 1_1* eingeben. Zunächst legen Sie für das Programm eine Textdatei an, die Sie dann z.B. in Ihr Visual Studio Projekt einfügen. Ein solches Programm beginnt immer mit der Kennung und Versionsnummer, also in unserem Fall *vs.1.1*.

Wenn Sie Konstanten für Ihr Vertex Shader Programm definieren möchten, tun Sie das gleich im Anschluss. Die folgende Zeile beschreibt das Konstanten Register *c10* mit vier Float Werten:

```
def c10, 0.25, 0.5, 0.75, 1.0
```

Anschließend müssen Sie spezifizieren, welche Eingabedaten Sie verwenden wollen. Zunächst sollen die Vertex Koordinaten genügen, die im ersten Attribut Register für Vertices *(v0)* stehen. Dazu verwenden Sie:

```
dcl_position v0
```

Ihr erstes Programm soll nicht mehr tun, als die Vertices zu transformieren und jedem

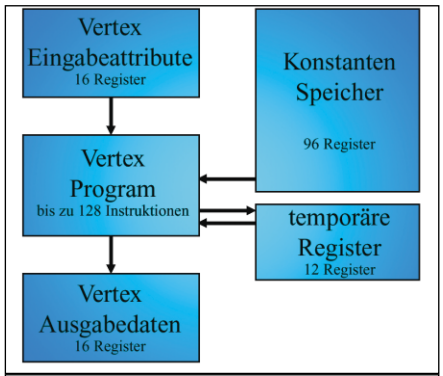
Die Input Register der Vertex Shader 1.1

Name	Typ	Anzahl	Daten	Zugriff
a0	Adressregister	1	Integer	schreiben/adressieren
c#	Float Konstante	mind. 96	4 x Float	definieren/lesen
r#	Arbeitsregister	12	4 x Float	schreiben/lesen
v#	Vertex Parameter	16	4 x Float	lesen

Nur Schreibzugriff auf die Output Register (VS 1.1

Name	Typ	Daten
oD0/oD1	diffuse/spekulare Farbe	je 4 x Float
oFog	Nebel Intensität	1 x Float
oPos	Position*	4 x Float
oPts	Punktgröße	1 x Float
oT#	8 Texture Koordinaten	je 4 x Float

* Alle vier Komponenten müssen von einem Vertex Programm gesetzt werden.



Vertex Shader: Sie sehen den schematischen Aufbau – am Beispiel einer GeForce 3 GPU.

Vertex eine Konstante *Farbe* zu verpassen. Die Transformation erledigen Sie durch eine Multiplikation der entsprechenden 4x4-Matrix, die in den Konstantenregistern *c0-c3* und der Eingabeposition gespeichert ist:

```
m4x4 oPos, v0, c0
```

Der *m4x4* Befehl ist dabei lediglich ein Makro. Tatsächlich werden vier Kreuzprodukte ausgeführt, die genau das Matrix-Vektor-Produkt darstellen. Den eben angesprochenen Farbwert geben Sie den Vertices so mit:

```
mov oD0, c10
```

Nun fragen Sie sich vielleicht, woher kennt der Vertex Shader die Matrix? Hier sind wir am Punkt der Integration des Shaders in die Applikation angelangt. Glücklicherweise bietet die D3DX-Bibliothek wieder eine Reihe von nützlichen Befehlen. Als erstes benötigen Sie eine Variable vom entsprechenden Typ, also einen Zeiger auf ein *IDirect3DVertexShader9*-Interface:

```
LPDIRECT3DVERTEXSHADER9
 pVertexShader = NULL;
```

Mit D3DX können Sie den Vertex Shader kompilieren und in einem Speicherbereich, den ein *D3DXBUFFER*-Objekt verwaltet, ablegen:

```
DWORD flags = 0;
LPD3DXBUFFER pCode;
D3DXAssembleShaderFromFile( „vs.txt“,
  NULL, NULL, flags, &pCode, NULL );
```

Daraus können Sie Direct3D das Vertex Shader Objekt erzeugen lassen und den Speicher wieder freigeben:

Strichzeichnung: Mit diesen Texturen erreichen Sie Hatching-Effekte.

```
pD3DDevice->CreateVertexShader(
  (DWORD*)pCode->GetBufferPointer(),
  &pVertexShader );
pCode->Release();
```

Wenn Sie Ihr Programm weitergeben wollen, aber der Quelltext des Vertex Shaders nicht als Textdatei sichtbar sein soll, können Sie auch das Kompilat in eine Datei schreiben, oder später anderweitig zu Ihrem Programm linken:

```
FILE *f = fopen( „vs.comp“, „wb“ );
fwrite(
  pCode->GetBufferPointer(), 1,
  pCode->GetBufferSize(),    f );
fclose( f );
```

Jetzt müssen Sie beim Rendering nur noch mitteilen, dass Sie den Vertex Shader verwenden wollen. Das geht mit einem einzigen Befehl:

```
pD3DDevice->
 SetVertexShader( pVertexShader );
```

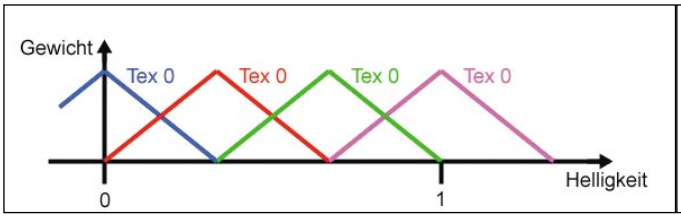
Jetzt verwendet Direct3D den Vertex Shader statt der herkömmlichen Transform&Lighting Stufe der Fixed Function Pipeline solange, bis Sie die obige Funktion mit *null* als Parameter aufrufen.
Als letztes bleibt also die Aufgabe, die benötigten Konstantenregister zu setzen, in unserem Beispiel also die Transformationsmatrix. Diese Matrix muss die Vertices vom Object Space in den World Space und weiter in den Clip Space transformieren. Wenn Sie diese Transformationen einzeln bestimmt haben (wie aus den letzten Ausgaben bekannt), können Sie die benötigte Matrix durch Konkatenation bestimmen:

```
D3DXMATRIX modelViewProjection
  = matWorld * matView * mProjection;
```

Um den Inhalt der Konstanten-Register für den gerade aktuellen Vertex Shader zu definieren, verwenden Sie den *SetVertexShaderConstantF*-Befehl. Damit übergeben Sie einen oder mehrere Vektoren, die aus vier Floats bestehen. Der erste Parameter ist dabei der Index des ersten Konstanten-Registers, das beschrieben wird:

```
pD3DDevice->
 SetVertexShaderConstantF
(0,(float*)modelViewProjection, 4 );
```

Jetzt können Sie wie schon bekannt, z.B. mit Vertex Buffers, rendern, nur die Transformation übernimmt jetzt Ihr Vertex Shader.



Summe gleich eins: Die Gewichte der Hatching-Texturen.

Abwechslung: Ein bekanntes *OBJ*-Modell dargestellt mit Materialparametern, diffuser Beleuchtung und Hatching.

Pixel Shader in Direct3D

Pixel Shader definieren Sie in Direct3D nahezu analog zu Vertex Shaders. Auch hier beschränken wir uns zunächst auf die Assemblersprache und die Pixel Shader Version 1.1. Ein Pixel Shader Programm, für dessen kompletten Befehlssatz wir wieder auf die DirectX-Hilfe verweisen, beginnt wieder mit der Kennung und eventuellen Konstantendefinitionen. Anschließend geben Sie an, welche Texturen Sie auslesen wollen. Wenn Sie die erste Textur-Stage (an der entsprechenden Stelle) auslesen – auch *sample* genannt –, geben Sie den Befehl ein:

```
tex t0
```

Damit steht Ihnen das Resultat, also der ausgelesene Farbwert im Register *t0* zur Verfügung. Weitere typische Instruktionen sind *Addition/Subtraktion* (add/sub), *Multiply-and-Add* (mad) und natürlich *Move* (mov).
Wenn Sie beispielsweise normale Textur auslesen, modulieren und mit der diffusen Farbe nachprogrammieren wollen, tun Sie das mit folgendem Befehl:

```
mul r0, t0, v0
```

Dabei ist *t0* das Register mit der Farbe aus der Textur, *v0* das Eingaberegister der diffusen Farbe und *r0* ein Arbeitsregister, das gleichzeitig auch das Ausgaberegister für den endgültigen Farbwert ist.
Sie aktivieren Pixel Shaders analog wie Vertex Shaders, wobei Sie lediglich den Term *VertexShader* durch *PixelShader* ersetzen.
Im Folgenden zeigen wir Ihnen eine einfache Technik, die beim Rendering den Eindruck von Strichzeichnungen erwecken soll. Die Umsetzung demonstrieren wir Ihnen anhand von einem Vertex und Pixel Shader.

Hatching

Die Technik, mit Strichen zu zeichnen, wird auch Hatching genannt. Natürlich ist es nicht ganz einfach, den Eindruck von Strichzeichnungen per Grafikhardware zu erwecken. Denn dazu müssen Sie recht kompliziert Tex-

tur Koordinaten und viele Texturen generieren. Exemplarisch zeigen wir Ihnen an dieser Stelle, was Sie mit einfachen Mitteln und einem Single Pass Rendering Verfahren ohne großen Aufwand bewirken können.

Das Prinzip ist folgendes: Sie berechnen zunächst eine diffuse Beleuchtung, bilden also ein Skalarprodukt zwischen Normale und Lichtrichtung. Dadurch erhalten Sie einen Wert im Intervall zwischen *-1* und *+1* – negative Werte setzen Sie aber auf Null.

Was Sie noch brauchen, ist eine Reihe von Texturen, die unterschiedlich dunkel schraffierte handgezeichnete Bereiche zeigen. Je nach berechnetem Helligkeitswert soll die entsprechende Textur zum Zeichnen ausgewählt und verwendet werden. Bei vier verwendeten Texturen (so viele sind bei den meisten Grafikkarten in einem Renderpass addressierbar), stünde Texture *0* für das Intervall *[0;0.25]*, Texture *1* für *[0.25;0.5]* usw.

Eine harte Auswahl der Textur würde allerdings keine sehr schönen Ergebnisse liefern, vielmehr ist eine Interpolation von den jeweils zwei nächsten Texturen wünschenswert. Das Bild *Summe gleich eins* zeigt die Gewichtung der einzelnen Texturen – unterschiedlich eingefärbt – in Abhängigkeit von der berechneten Helligkeit. Mit dieser Gewichtung ist sicher gestellt, dass die Summe aller Gewichte gleich eins ist. Das wiederum ermöglicht es, jeweils die Farbwerte der vier Texturen mit dem entsprechenden Gewicht zu multiplizieren und alle aufzusummieren. Und das ist genau das, was unsere Beispiel-Shader tun sollen. Sie müssen dabei beachten, dass es die Striche in den Texturen sind, die wir gewichten wollen, deshalb müssen Sie die Texturen aus dem Bild *Strichzeichnung* invertiert verwenden.

Betrachten Sie nun also die Shader im Einzelnen. Der Vertex Shader benötigt als Eingabe die Vertex Position, die Normale und die folgenden Konstanten. Die Position wird normal transformiert:

```
vs.1.1
def c11, 0.0, 0.33, 0.66, 1.0 ....

dcl_position v0
dcl_normal   v1
m4x4 oPos, v0, c0
```

Nun benötigen Sie, sofern für das 3D-Modell keine Textur-Koordinaten für das Hatching vorhanden sind, eben solche. Das Einfachste ist eine planare Projektion, deren Resultat für alle vier Textur Stage verwendet wird:

```
mul r0, v0.xyzw, c13
mov oT0, r0 ....
```

Anschließend kommt der trickreiche Teil. Dieser berechnet zunächst die diffuse Beleuchtung. Negative Werte setzen Sie Null (max), im Konstantenregister *c4* steht dabei die Lichtrichtung:

```
dp3 r0, v1, c4
max r0, r0, c15
```

An dieser Stelle enthält das *r0*-Register in jeder Komponente die Helligkeit der Oberfläche *[0;1]*. Es gilt, daraus die vier Gewichte zu bestimmen. Dazu wird der Betrag des Abstands des Helligkeitswertes von den Maxima der Gewichtsfunktionen berechnet.

```
sub r0, c11, r0
max r0, r0, -r0
```

Durch entsprechende Skalierung und Inversion erhalten Sie genau die vier Gewichte in den Komponenten von *r0*:

```
mul r0, r0, c14
sub r0, c12, r0
```

Als letzte Aufgabe geben Sie die Daten an die Pixel Shader weiter. Pixel Shader 1.1 können nur sehr eingeschränkt auf Daten zugreifen. Farbwerte können Sie nur in *oD0* und *oD1* übergeben, wobei Sie aber jeweils nur getrennt auf die ersten drei (RGB) oder die letzte Komponente (Alpha) zugreifen können. Deshalb bleibt nur folgendes:

```
mov oD0,   r0.x ....
```

Der Pixel Shader muss nun noch die gewichtete Summe der Texturen berechnen. Sie definieren Konstanten und lesen die Texturen aus:

```
ps.1.1
def c1, 1.0, 1.0, 1.0, 1.0
tex t0 ...
```

Anschließend gewichten Sie den ersten Textur Wert durch Multiplikation mit dem Gewicht aus dem Vertex Shader:

```
mul r0, t0, v0
```

Analog gehen Sie mit den drei weiteren Texturen vor, allerdings verwenden Sie den *Multiply-Add*-Befehl, um gleich die Summe mit den Zwischenergebnis zu erhalten:

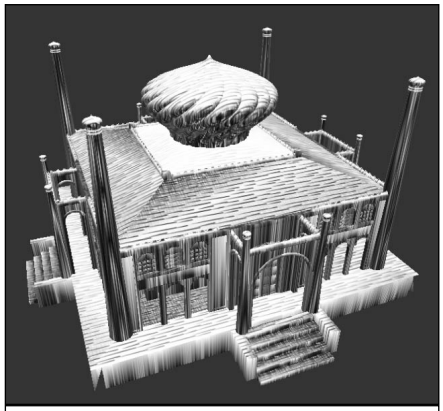
```
mad r0, t1, v0.a, r0...
```

Zuletzt müssen Sie nur noch die Inversion der Texturen umkehren:

```
sub r0, c1, r0
```

Die wichtigsten Tokens im *Wavefront*-Format (.obj)

Token/Syntax	Beschreibung
# text	Kommentarzeile
v float float float	Definition einer Vertex Position, Indizierung beginnt bei 1
vn float float float	Definition einer Normalen, Indizierung ab 1
vt float float	Definition einer Texture Koordinate, Indizierung ab 1
f int int int ...	Polygon definiert durch Vertex Indizes
f int/int int/int ...	Polygon definiert durch Vertices und Texture Koordinaten Indizes
f int/int/int ...	Polygon mit Vertex, Texture und Normalen Index



Palast: Dieses 3D-Modell, dargestellt mit unserem Beispielprogramm, gibt es bei 3D-Cafe.

Datensätze einfach einlesen

Bei der Programmierung von Grafiken bleibt die Frage, wie und woher Ihre Daten für die Modelle kommen. Deshalb enthält unser Beispielprogramm nun eine Klasse, mit der Sie Dateien im *Wavefront*-Format (.obj) lesen können.

Dabei handelt es sich um ein weit verbreitetes Format, das viele Modeling-Programme unterstützen. Es kann sowohl Dreiecksnetze als auch parametrische Flächen speichern. Die *.obj*-Dateien sind Text basiert und daher sehr einfach einzulesen. Eine gute Quelle für Dokumentationen für Dateiformate aller Art finden Sie unter **www.wotsit.org**. Die wichtigsten Tokens finden Sie in der letzten Tabelle aufgelistet.

Viele freie Modelle finden Sie unter: **www.3dcafe.com** : et