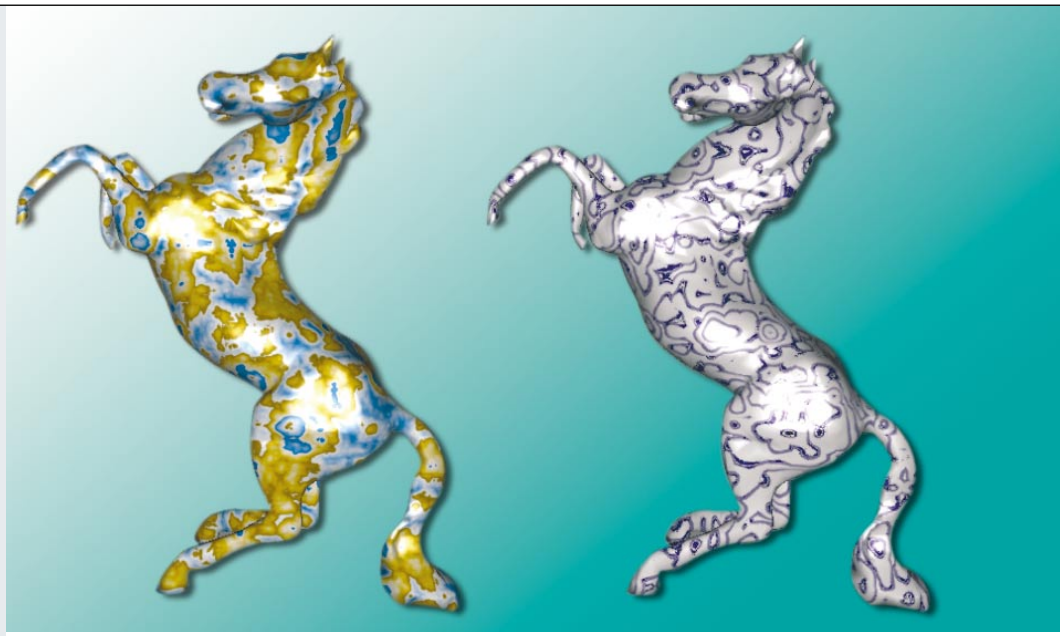
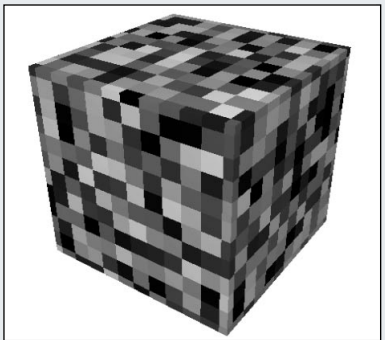


High Level Shader Language Direct3D 9

„Nun lasst uns in 3D sprechen!“

Hochsprachen für die Programmierung von Grafikkarten haben Hochkonjunktur. In dieser Ausgabe erfahren Sie, wie einfach es ist, Microsofts HLSL in Direct3D zu verwenden.

Carsten Dachsbacher



Zufall: Ein 3D Array von Zufallszahlen dient als Grundlage für Noise-Funktionen.

Die Einführung einer programmierbaren Geometrie- und Fragmentverarbeitung bei den Grafikkarten, ersteres ab den GeForce 3 oder Radeon 8500 GPUs und für letzteres ab den Radeon 9700 bzw. GeForce FX GPUs, war ein wichtiger Schritt: Damit genießen Programmierer neue Freiheiten, um eine Vielzahl von Grafikeffekten zu gestalten. Diese Effekte konnte – wenn überhaupt – zuvor nur die CPU berechnen.

Für die Programmierung kamen und kommen noch Assembler artige Sprachen zum Einsatz, die Sie aus bisherigen PC-Underground-Artikeln kennen. Für viele Einsatzzwecke ist aber eine Hochsprache wünschenswert, wie Sie z.B. schon seit langem von Renderman her bekannt ist. Die erste Hochsprache dieser Art ist *nVidias Cg* (C for graphics), die einen Vertex- oder Fragment-Shader aus C ähnlicher Syntax in Assembler-Code übersetzt. Eine solche Hochsprache wird auch direkt in OpenGL 2.0 integriert sein, für Direct3D gibt es das schon: die High Level Shader Language, kurz HLSL. Diese Ausgabe führt Ihnen anhand eines konkreten Beispiels vor, wie Sie in Direct3D ganz einfach HLSL Shader programmieren und einbinden.

Effect Files

Mit Direct3D können Sie so genannte *Effect Files* (Erweiterung *.fx*) definieren. Diese definieren textuell eine oder mehrere Render-Techniken. Eine *fx*-Datei beschreibt vollständig, wie das Rendering eines 3D-Objekts ablaufen soll, d.h. welche Texturen und Texture Mode wie verwendet werden, welche Vertex und Pixel Shader zum Einsatz kommen, und ob einer oder mehrere Renderpasses benötigt werden. Diese Dateien bieten also einen Weg, in abstrakterer Form als mit der Low-Level-Methode zu programmieren. Sie können somit Vertex und Pixel Shader nutzen, um Rendering-Effekte zu kapseln. Die Effekte selbst können Sie dann entweder mit HLSL oder der bereits bekannten Assemblersprache programmieren.

Wie diese Effect Files aufgebaut sind, zeigen wir Ihnen anhand eines Beispiels. Darin werden Sie einige Direct3D Renderstates erkennen, die Sie bisher über explizite Aufrufe einstellen mussten. Als Beispiel soll uns eine an sich wenig spektakuläre Texturierung dienen. Die *fx*-Datei definiert zunächst eine *texture*-Variable (texMap), der Sie später bei der Verwendung von der Applikation aus eine Texture zuweisen. Anschließend definieren Sie *tech0*, das nur einen Renderpass enthält. Die meisten der Renderstate Bezeichner sind selbsterklärend, da ihr Name etwa dem der *SetRenderState(...)*-Konstanten entspricht.

```
texture texMap;
 technique tech0
```

```
{ pass P0
  { // keine Shader:
    fvf   = XYZ | Tex1;...
```

Diesen Effet, wie in der Datei *effect.fx*, verwenden Sie in Ihrem Programm. Um den Effekt zu laden, benötigen Sie ein *ID3DXEFFECT*-Objekt. Den Rest übernimmt D3DX:

```
LPDIRECT3DTEXTURE9 pTexture;...
```

Wie immer sollten Sie prüfen, ob dabei ein Fehler aufgetreten ist. Sollte dies der Fall sein, so können Sie sich eine detaillierte Fehlermeldung ausgeben lassen:

```
if( FAILED( hr ) )
{ char *buf =pBufferErrors-> ...}
```

Jetzt müssen Sie noch eine der potenziell mehreren *Techniken* aus dem Effect File wählen. Dazu können Sie die erste, auf der verwendeten Hardware ausführbaren, Techniken suchen lassen.

```
D3DXHANDLE hTechnique;
pEffect->FindNextValidTechnique
         ( NULL, &hTechnique );
```

Wenn Sie für den ersten Parameter das Handle einer anderen Technik angeben, beginnt die Suche von dort ausgehend. Zuletzt wählen Sie die gefundene Technik aus und setzen die benötigte Texture:

```
pEffect->SetTechnique( hTechnique );
pEffect->SetTexture...
```

Jetzt können Sie die Rendertechnik einsetzen. Da die Applikation nicht weiß, wie viele Renderpasses benötigt werden, fragen Sie deren Anzahl ab und führen sie dementsprechend aus. Das Effect File konfiguriert die Renderstates und arbeitet automatisch.

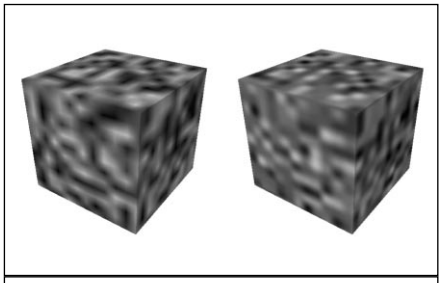
```
// Anzahl der Renderpasses
UINT nPasses;
// Beginn 0: Sichern+Wiederherstellen
pEffect->Begin( &nPasses, 0 );
for ( UINT p = 0; p < nPasses, p ++ )
{  pEffect->Pass( p );
  // Zeichnen:
  pD3Ddevice->DrawPrimitive...
}
pEffect->End();
```

High Level Shader Language

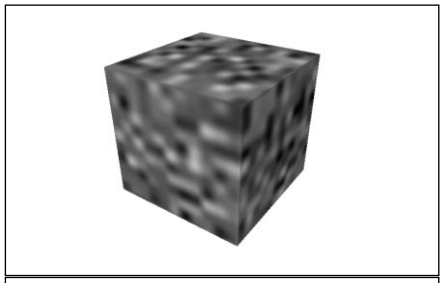
Wie bereits erwähnt, können Sie innerhalb der Effect Files Vertex und Pixel Shader definieren. Diese können Sie, wie das folgende Beispiel zeigt, direkt in der bekannten Assembler Notation angeben:

```
pass P0 {VertexShader = asm { vs_1_1
    dcl_position  v0
    dcl_normal    v1
      mov oPos, v0
      mov oD0, v1 }  ...  }
```

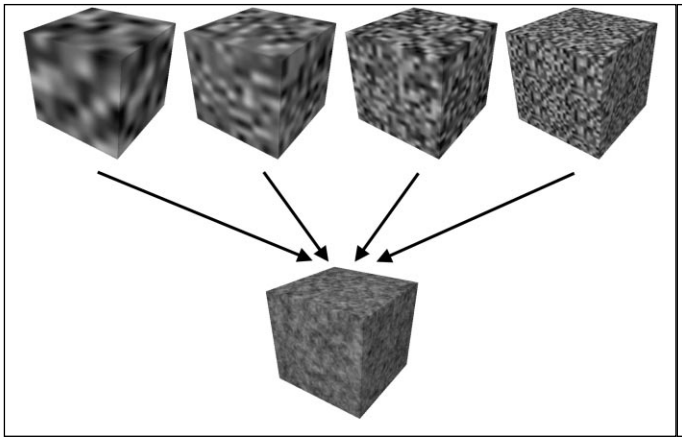
Die zweite Variante, die wir Ihnen in dieser Ausgabe vorstellen wollen, ist die Verwendung der Hochsprache HLSL mit C-ähnlicher Syntax, die im Zuge von DirectX 9 entwickelt wurde. Aufgrund des speziellen Anwendungsgebietes



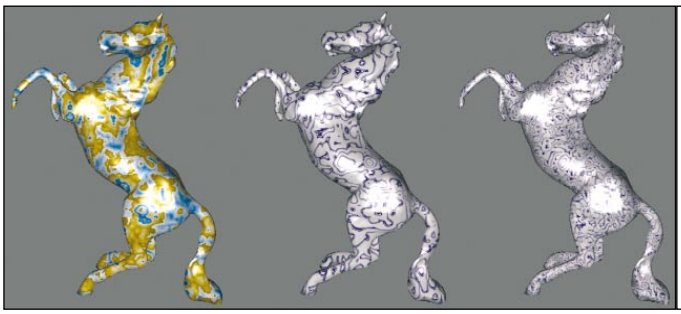
Unterschied: Links sehen Sie die *Lineare* und rechts die vorberechnete kubische Interpolation.



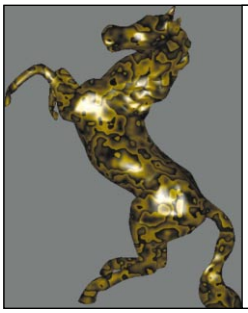
Kubischer Filter: Mit dieser Noise-Funktion erzeugen Sie Texturen.



Summe: Mehrere Noise-Funktionen unterschiedlicher Frequenzen werden summiert.



Marmor: Mit Lookup Textures strukturieren Sie Marmor verschieden, wenn Sie *Gain* und *Lacunarity* variieren.



Lookup Textures in 3D: Holzstrukturen sind ganz einfach zu erzeugen.

sind natürlich entsprechende Datentypen und Objekte definiert, von denen wir Ihnen hier die wichtigsten vorstellen. Die grundlegenden skalaren Datentypen sind *bool* (wahr oder falsch), *int* (32 Bit Integer), und die drei Floating Point Typen *half, float* und *double* mit 16, 32 und 64 Bit Genauigkeit. Dabei müssen Sie aber beachten, dass eine GPU nicht unbedingt alle diese Datentypen unterstützt. Es kann z.B. sein – und damit treten Bereichs- bzw. Genauigkeitsprobleme auf – dass der *int* Typ durch einen *float* emuliert wird.

Ebenso ist die Unterstützung von *half* und *double* Typen nicht gewährleistet. Sie können zwar immer jeden Typ verwenden, aber bedenken Sie die Probleme. Weiterhin sind *vector* und *matrix* Typen definiert, die, wie der Name schon sagt, verwendet werden können, um Vektoren oder Matrizen beliebiger Typen zu deklarieren. Die gebräuchlichsten zusammengesetzten Typen sind aber von vornherein global über *Typedefs* definiert. So bezeichnet z.B. *float3* oder *float4* einen drei bzw. vier Komponentenvektor aus *Floats* und *float4x4* eine 4x4-Matrix, mit der Sie alle notwendigen Transformationen beschreiben können.

Die nächste wichtige Gruppe stellen die *Object Types* dar, zu denen auch die bereits erwähnten Vertex- und Pixel-Shader zählen. Außerdem gehören die so genannten Sampler dazu, die eine Direct3D Sampler Stage beschreiben, also welche Texture wie abgetastet und gefiltert wird. Das folgende Beispiel ist bereits Bestandteil unseres Programms, dessen Aufbau wir schrittweise aufzeigen. Es wird ein Textur-Objekt mit dem Sampler assoziiert und tri-lineares Filterung dafür gewählt:

```
texture noiseTexture;
sampler noiseSampler =
                  sampler_state
{ Texture   = ( noiseTexture );
  MipFilter=LINEAR;MinFilter=LINEAR;
  MagFilter = LINEAR; };
```

Zuletzt benötigen Sie noch Strukturen in Ihrem HLSL-Programm, die Sie wie aus C bekannt mit dem Schlüsselwort *struct* definieren. An dieser Stelle kommen wir auf unser Beispielprogramm. Die Eingabedaten pro Vertex, die Sie beispielsweise mit dem Befehl *DrawPrimitive* von der Applikation zur Grafikpipeline senden und die Ihr Vertex Shader verarbeiten soll, definieren Sie als eine Struktur:

```
struct VERTEX
{ float4 position : POSITION;
  float3 normal   : NORMAL; };
```

Dabei legen Sie die Bezeichner der Daten und den Typ (hier *float3* oder *float4*) fest – die semantische Bindung der Variablen an den Vertex-Datenstrom steht rechts des Doppelpunktes. Der Vertex Shader bearbeitet jeden dieser Vertices und erzeugt die Daten, die an die Rasterisierungsstufe der Grafikkarte weitergegeben werden. Die entsprechenden Daten fassen Sie wiederum in einer Struktur zusammen:

```
struct FRAGMENT
{ // transformierte Koordinaten
  float4 position  : POSITION;
  // Texture Koordinaten
  float3 texture0  : TEXCOORD0;
  float3 texture1  : TEXCOORD1;...
  // zwei Farbwerte
  float4 color     : COLOR0;
  float4 colorSpec : COLOR1; };
```

Als Vertex Shader deklarieren Sie eine Funktion, die als Parameter eine *VERTEX* Struktur entgegen nimmt und eine *FRAGMENT* Struktur zurückliefert:

```
FRAGMENT myVS( VERTEX vertex )
{ FRAGMENT result;
  result.position = ...
  return result. };
```

Genauso verfahren Sie für den Pixel Shader, der eine Struktur ausfüllt, die einen Farbwert enthält, aber auch mehrere enthalten kann:

```
struct FRAGRESULT
{  float4 color : COLOR; };

FRAGRESULT myPS( FRAGMENT frag )
{ FRAGRESULT result;result.color = ...
  return result; }
```

Für die vollständige Liste der semantischen Bindings müssen wir Sie an dieser Stelle an die DirectX-Hilfe verweisen. Ebenso verhält es sich mit dem riesigen Befehlssatz von HLSL, den wir Ihnen hier in Auszügen, sofern er im Beispielprogramm Anwendung findet, vorstellen.

Prozedurale Texturierung

Das Beispielprogramm soll 3D-Objekte prozedural texturieren. Das bedeutet, es kann – durch eine geeignete Berechnungsvorschrift – für jeden Punkt im Raum einen Farbwert berechnen. Diese Form der Texturierung hat natürlich Vor- und Nachteile. Als wichtigste Punkte sprechen dafür, dass Sie ohne explizite Textur-Koordinaten auskommen (Solid Texturing), beliebig große Flächen ohne erkennbare Wiederholungen texturieren können und vor allem eine parametrisierte Texturierung haben, also durch Änderung weniger Parameter das Aussehen der Textur gestalten können. Die Nachteile liegen bei der benötigten Rechenzeit. Deshalb sollten Sie solche Techniken im Allgemeinen nur dort einsetzen, wo es sich auch wirklich lohnt.

Die meisten prozeduralen Texturierungen basieren dabei auf so genannten *Noise*-Funktionen. Solche Funktionen liefern reproduzierbare Pseudozufallszahlen für jeweils gleiche Parameter, sollten bandbegrenzt sein und keine offensichtlich wiederholenden Muster aufweisen. In der Praxis wird oft einfach eine Menge von Zufallszahlen wie ein dreidimensionales Array an ganzzahligen Koordinaten berechnet. Durch eine geeignete Filterung für die Zwischenwerte erhalten Sie eine geglättete Variante, die als Noise-Funktion dienen kann.

Nun ist aber eine dieser Noise-Funktionen alleine nicht sehr spektakulär. Die Summe von Noise-Funktionen (oder auch nur einer Funktion) unterschiedlicher Frequenzen gestattet aber schon sehr interessante Texturen.

Jede der unterschiedlichen Noise-Funktionen wird dabei als Octave bezeichnet, da oft eine verdoppelte Frequenz – wie bei Oktaven in der Musik – zwischen den Noise-Funktionen verwendet wird. Da man allerdings daran nicht gebunden ist, wird ein Parameter, der den Frequenzmultiplikator zweier Oktaven darstellt, eingeführt und mit *Lacunarity* bezeichnet. Die Gewichte, mit denen Sie die Oktaven vor der Summenbildung gestalten, nehmen meist mit zunehmender Frequenz ab.

Der Faktor wird als *Gain* bezeichnet. Solche Texturen können Sie nun in Echtzeit berechnen und darstellen. Die dazugehörigen HLSL-Programme stellen wir Ihnen im Folgenden vor. Als Noise-Funktion dient Ihnen eine 3D-Textur.

Diese sollte aber nicht direkt die Zufallszahlen enthalten, weil die Grafikhardware nur linear filtern kann, für eine gut aussehende geglättete Variante sollten Sie eine kubische Filterung verwenden. Der Trick ist, beispielsweise *16x16x16* Zufallswerte zu berechnen und daraus eine kubisch geglättete *128x128x128* 3D-Textur zu erzeugen. Beim Auslesen der Textur wird zwar wiederum linear interpoliert, aber durch die vorberechnete Glättung werden die Artefakte kaschiert. Den Unterschied sehen Sie!

Um Ihnen HLSL besser zu präsentieren, berechnet der Vertex Shader auch eine Phong-Beleuchtung. Dazu benötigen Sie zunächst eine Reihe von Parametern, die vor dem Rendering von der Applikation mit Werten belegt werden.

```
// Matrix: Object ->Clip Space
float4x4 matWVP;
float4 lightPosition; // Object Space
float4 cameraPosition; // Object Space
float   scale; // Noise Skalierung
float lacunarity; // Lacunarity
float4 amplify; // 4 Oktaven
```

Die Definition der benötigten Strukturen und Sampler haben Sie bereits im vorherigen Abschnitt gesehen, deshalb können wir uns gleich dem Vertex Shader widmen. Als erstes transformieren Sie die Koordinaten der Vertices:

```
result.position = mul(...
```

Anschließend berechnen Sie die normalisierten Vektoren vom Vertex zum Betrachter und zur Lichtquelle.

```
float4 toViewer, ...
```

An der Normalen können Sie den Vektor zur Lichtquelle wie folgt spiegeln:

```
reflect = normalize( ...
```

Die diffuse Beleuchtung berechnen Sie in *NdotL*, die spekulare Beleuchtung durch das Skalar-Produkt (Dot) aus dem Reflexionsvektor und dem Vektor zum Betrachter. Die Vorzeichnenüberprüfungen und die Exponentiation übernimmt der *lit*-Befehl:

```
float NdotL = dot(float4...
```

Den ambienten und diffusen Beleuchtungsanteil speichern Sie getrennt vom spekularen. Die beiden Teile, die getrennt behandelt werden müssen, kombinieren Sie später im Pixel Shader.

```
result.color =litVector....
```

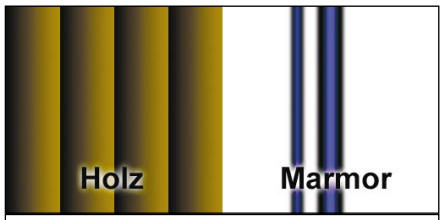
Für das Solid Texturing berechnen Sie jetzt die Positionen, an denen die Noise-Funktionen ausgewertet werden sollen, aus der Object Space Koordinate des Vertex. Die Positionen für vier Oktaven schreiben Sie in die Textur Koordinaten:

```
float4 noisePosition =....
```

Der Pixel Shader liest die Noise Textur an den vier berechneten Positionen aus und muss daraus lediglich noch die Summe bilden. Die Gewichtung der Oktaven (im *amplify* Parameter) und die anschließende Summe ließe sich elegant mit einem Skalarprodukt darstellen. Die Restriktionen der Pixel Shader der Version 1.4 (oder niedriger) verlangen aber die Austeilung in zwei Operationen, wie Sie im Beispiel sehen. Der resultierende Farbwert wird mit dem ambient-diffusen Beleuchtungsanteil multipliziert und der spekulare Anteil hinzuaddiert:

```
float4 octaves;
octaves.x = tex3D...
```

Jetzt müssen Sie nur noch die Vertex und Pixel Shader in den Effekt einsetzen. Dazu verwenden Sie die *Pass*-Beschreibung, wobei Sie die Ziel Vertex und Pixel Shader Version jeweils angeben:



Lookup Textures: Diese Texturen verwendet das Rendering für die letzten beiden Bilder.

```
VertexShader=compile vs_1_1 vsNoise();
```

Die Programmparameter setzen Sie von Ihrer Applikation aus, wobei Sie die Methoden des *ID3DXEFFECT* Interfaces nutzen:

```
D3DXVECTOR4 vec;....
```

Sie erweitern das Programm einfach aber effektvoll, wenn Sie die berechnete Summe der Noise-Werte nicht direkt als Farbwert verwenden, sondern als Textur-Koordinate für eine Lookup-Textur verwenden. So erhalten Sie Holz- und Marmor-Effekte.

Occlusion Query

Lohnt der Aufwand? Wenn ein 3D-Objekt sehr weit vom Betrachter entfernt größtenteils verdeckt ist, können Sie auf einfachere Shader ausweichen. Mit dem Occlusion Query Mechanismus stellen Sie fest, wie viele Pixel tatsächlich in den Framebuffer geschrieben wurden. Beachten Sie, dass der *GetData*-Befehl asynchron arbeitet – es befinden sich einfach zu viele Zwischenstufen in der Grafikpipeline, als dass das Resultat sofort bereitstünde. Sie sollten ein Programm so konzipieren, dass Sie in der *While*-Schleife noch andere Aufgaben erledigen können, oder vor dem Aufruf *GetData* etwas Zeit verstreichen kann. : et