Pixel-Shader in OpenGL

Zum schnellen, schönen Schein

Mit Pixel-Shadern und OpenGL programmieren Sie die neuen GeForce-3 und -4-Grafikkarten und **zaubern Effekte**, wie sie sich bisher nicht in Hardware rendern ließen. Lernen Sie die Techniken der neuen 3D-Spiele kennen.

CARSTEN DACHSBACHER

Ab DirectX 8 unterstützt Direct3D Pixel-Shader. Damit können Sie in einer Art Assemblersprache festlegen, wie die Grafikkarte Texturen ausliest. Davon hängt ab, wie die Farben von den ausgelesenen Texeln und der Beleuchtungsberechnung zusammengefügt werden, um einen gerasterten Pixel zu färben.

Diese Features werden von neuen Grafikkarten wie nVidia-GeForce-3/4 und ATI-Radeon-8500 unterstützt. Allerdings sind Anzahl und Art der verwendeten Befehle beschränkt. Die Funktionsweise erkennen Sie besser, wenn Sie die OpenGL-Extensions von nVidia betrachten, die Ihnen dieser Beitrag vorstellt. Dabei teilt sich die Funktionalität der Pixel-Shader in zwei Aufgabengebiete:

- Sie steuern mit Textur-Shadern, wie und an welcher Koordinate die Texturen ausgelesen werden.
- Die zweite Stufe sind die Register-Combiner, die für das Texture Blending verantwortlich sind. Damit können Sie festlegen, wie aus den ausgelesenen Texeln die endgültige Pixelfarbe wird.

Um diese Features einfach zu handhaben, stellen wir Ihnen eine Bibliothek von nVidia vor, mit der Sie die Einstellungen der Textur-Shader und Register-Combiner in einer Art Pseudo-Programmiersprache vornehmen können: die *nvparse* Bibliothek.

Textur-Shader

Wir gehen von der Hardware einer GeForce-3-Karte bzw. einer ähnlich leistungsfähigen Karte aus. Lediglich die Anzahl der verfügbaren Textur-Units (maximale Anzahl von gleichzeitig adressierbaren Texturen), Register-Combinerb und Textur-Shader-Modi kann im Vergleich zu anderen modernen Grafikkarten variieren. Das Bild unten verdeutlicht, wo die Textur-Shader in der Grafikpipeline zu finden sind.

Um die Textur-Shader zu verwenden, müssen Sie die OpenGL Extensions abfragen und die Zeiger auf die benötigten Funktionen holen. Diese Initialisierungsarbeit finden Sie im Sourcecode zur aktuellen Ausgabe. Die Textur-Shader können nur für alle Textur-Units zusammen aktiviert werden – mit dem Befehl:

```
glEnable
 ( GL_TEXTURE_SHADER_NV );
```

Die Betriebsmodi der einzelnen Textur-Units lassen sich in vier Gruppen unterteilen:

AUF CD

Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Heft Add-ons/Programmierung/PC Underground*.

- Herkömmliche Lookups (Auslesen), wie 1D/2D/Cubemap-Texturen,
- Spezialfälle wie *Pass Through* (Texturkoordinaten in RGB-Werte umwandeln) oder *Cull Fragment*, womit Sie einzelne Pixel beim Zeichnen auslassen können,
- Textur auslesen in Abhängigkeit von den Lookups anderer Texturen,
- Abhängigkeit mit zusätzlichem Skalarprodukt.

In dieser Ausgabe verwenden Sie vorrangig *herkömmliche Lookups* sowie *Textur auslesen*.

Um die Textur-Shader zu verwenden, müssen Sie für je vier Textur-Units die verwendete Instruktion angeben oder deaktivieren. Um eine Instruktion anzugeben, verwenden Sie *glTexEnv[i,f](...)*-Befehle oder *nvparse*. Letzteres Verfahren übergibt Instruktionen in einem String als Pseudocode.

Als einfaches Beispiel aktivieren Sie 2D Texture Mapping über Textur-Shader. Über *glTexEnvi(...)* wählen Sie das Textur-Shader-Environment und setzen als Shader-Operation Texture Mapping:

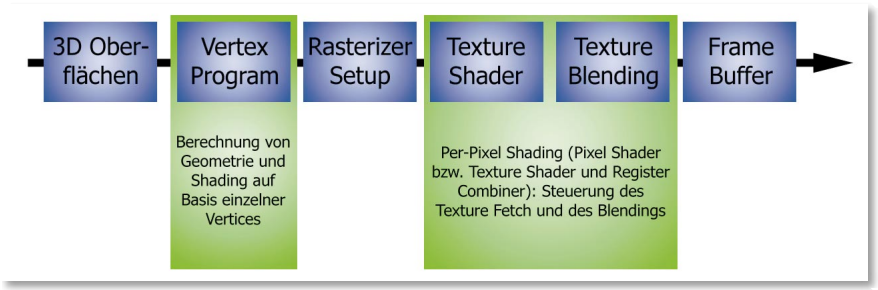
```
glActiveTextureARB
 ( GL_TEXTURE0_ARB );
glTexEnvi
 ( GL_TEXTURE_SHADER_NV,
 GL_SHADER_OPERATION_NV,
 GL_TEXTURE_2D );
```

Alle anderen Textur-Units (hier Unit 1) deaktivieren Sie mit

```
glActiveTextureARB
 ( GL_TEXTURE1_ARB );
glTexEnvi
 ( GL_TEXTURE_SHADER_NV,
 GL_SHADER_OPERATION_NV,
 GL_NONE );
```

Mit den Textur-Shadern bestimmen Sie nur die Art der Adressierung der Texturen. Die Texturen müssen Sie nach wie vor selbst mit dem *glBindTexture(...)*-Befehl setzen.

Damit können Sie das Beispiel ausbauen. Wir wollen auf die erste Textur-Unit eine Verschiebungstextur (DSDT-Textur) legen. Die Werte aus dieser Tex-



DIE KOMPONENTEN der Grafik-Pipeline in einer modernen 3D-Karte

DIE EINFACHEN TEXTUR-SHADER-OPERATIONEN

Operation	*nvparse* Befehl	Zweck
GL_TEXTURE_1D	texture_1d();	1D-Textur auslesen
GL_TEXTURE_2D	texture_2d();	2D-Textur auslesen
GL_TEXTURE_3D	texture_3d();	3D-Textur auslesen
GL_TEXTURE_CUBE_MAP_ARB	texture_cube_map();	Cube-Map-Textur auslesen
GL_CULL_FRAGMENT_NV	cull_fragment(...);	Vergleich, Pixel nicht zeichnen
GL_PASS_THROUGH_NV	pass_through();	STRQ nach RGBA kopieren
GL_OFFSET_TEXTURE_2D_NV	offset_2d(tex?,...)	EnvBump-Mapping
GL_DEPENDENT_AR_TEXTURE_2D_NV	dependent_ar(tex?)	AR-Farbvergleich als ST verwenden
GL_DEPENDENT_GB_TEXTURE_2D_NV	dependent_gb(tex?)	GB-Farbvergleich als ST verwenden

tur können Sie verwenden, um die Textur-Koordinaten der nachfolgenden Unit zu modifizieren. Dabei handelt es sich um den Environment-Bumpmapping-Effekt, wenn die Textur der zweiten Unit die gespiegelte Umgebung des 3D-Objekts enthält. Der Textur-Shader-Befehl hierzu lautet *GL_OFFSET_TEXTURE_2D_NV*.

Wenn Sie einer Textur-Unit eine Operation zuweisen, die einen Input- Wert aus einer vorherigen Unit benötigt (wie es hier der Fall ist), geben Sie diese an:

```
// Unit 1 bekommt als Eingabe
//Daten von Unit 0
glActiveTextureARB
 ( GL_TEXTURE1_ARB );

// BumpEnv Mapping
glTexEnvi( GL_TEXTURE_SHADER_NV,
GL_SHADER_OPERATION_NV,
GL_OFFSET_TEXTURE_2D_NV );

// Input Werte
glTexEnvi(GL_TEXTURE_SHADER_NV,
GL_PREVIOUS_TEXTURE_INPUT_NV,
GL_TEXTURE0_ARB );
```

Eine DSDT-Textur enthält für jeden Texel zwei 8-Bit-Werte, die angeben, wie stark die Verschiebung, also die Modifikation der Textur-Koordinaten ist. Die beiden Komponenten liegen im Wertebereich *[0..255]*, der auf den Bereich *[-1...1]* gemapped wird. Zusammen ergeben die Werte einen 2D-Vektor. Die Modifikation erfolgt nach folgender Formel, wobei *(S/T)* die angegebene Texturkoordinate der Unit *1* und *S'/T'* die neue ist. Die Werte *k(0..3)* sind eine 2x2-Matrix, um den Verschiebungsvektor aus der DSDT-Textur drehen und skalieren zu können.

```
S' = S + k(0) * ds + k(2) * dt
T' = T + k(1) * ds + k(3) * dt
```

Die 2x2-Matrix geben Sie wie folgt an:

```
float mat2d[] =
 { 0.8f, 0.0f, 0.0f, 0.8f };

glTexEnvfv(GL_TEXTURE_SHADER_NV,
GL_OFFSET_TEXTURE_MATRIX_NV,
 mat2d );
```

Einen Überblick über einen Teil der Textur-Shader-Operationen bietet Ihnen die Tabelle oben. Mehr über den dargestellten Teilbereich erfahren Sie auf den Webseiten der Grafikkartenhersteller.

Sie kennen jetzt das Handwerkszeug, um einen Textur-Shader zu aktivieren. Mit der *nvparse*-Bibliothek können Sie mit einem String die Textur-Shader-Operationen elegant beschreiben. Der Parser verarbeitet diesen String, und die Bibliothek übernimmt für Sie die *glTexEnv[i,f](...)*-Aufrufe. Es gibt zwei Schnittstellen zu *nvparse*:

• Der Aufruf zum Parsen (Analysieren) lautet

```
void nvparse
 ( char *program_string );
```

• Die Abfrage, ob Fehler im Programm enthalten sind, heißt

```
const char
**nvparse_get_errors();
```

Ein Textur-Shader-Programm hat einen festen Aufbau. Es beginnt mit einer Kennung, der eine bis vier Instruktionen folgen können – eine für jede Texture Unit. Die Bezeichnungen der einzelnen Instruktionen listet die Tabelle links auf. Die zulässigen Register in einem Programm sind *tex0*, *tex1*, *tex2*, *tex3*, womit Sie auf die Resultate der ausgelesenen Texturen zugreifen. Jedes Register steht für ein RGBA-Quadrupel. die Werte des Registers lassen sich expandieren, wenn Sie jede Komponente durch den Eintrag *2*Komponente-1* ersetzen:

```
expand(tex0)
```

Das obige Beispiel können Sie mit *nvparse* wie folgt angeben:

```
nvparse(
 „!!TS1.0
  texture_2d();
offset_2d(tex0,0.8,0.0,0.0,0.8);
  nop();
  nop();“ );

glEnable( GL_TEXTURE_SHADER_NV );
```

Da *nvparse* mehr Rechenzeit benötigt, sollten Sie es in dieser Form nicht in Ihrer Render-Pipeline stehen lassen. Generieren Sie eine Display-Liste, in der Sie die Aufrufe speichern:

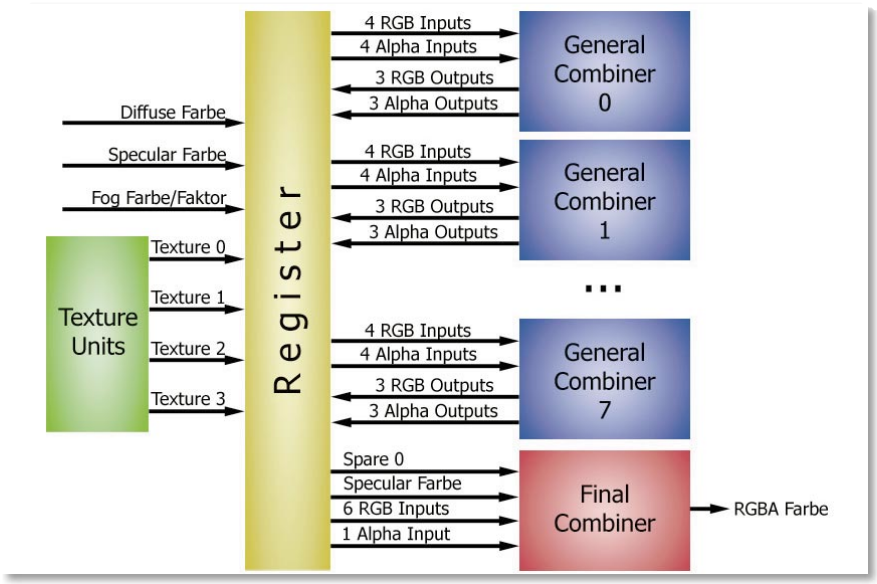
```
GLint setupBumpEnvMap;

setupBumpEnvMap = glGenList( 1 );
glNewList
 ( setupBumpEnvMap, GL_COMPILE);
glEnable( GL_TEXTURE_SHADER_NV);
nvparse( ... );
glEndList();
```

Das in der Liste gespeicherte Setup aktivieren Sie mit:

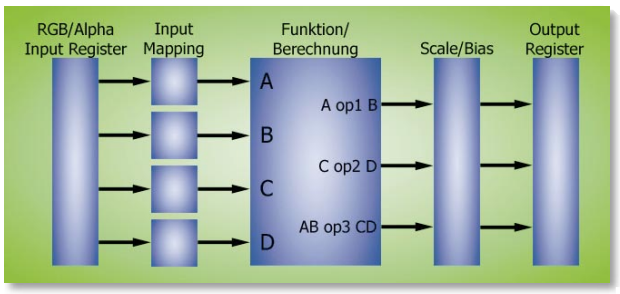
```
glCallList( setupBumpEnvMap );
```

Register-Combiner

Ein Register-Combiner zeigte das vorausgehende Bild beim Textur-Blen-



DIE REGISTER-COMBINER einer Geforce-3-Karte



DAS DIAGRAMM eines General Combiner – nur mit dem RGB-Teil

ding. Hier werden die Farbwerte, die aus den Texturen ausgelesen wurden und aus der Beleuchtungs- und Fog-Berechnung kommen, mit einem oder mehreren General-Combinern zur fertigen Pixelfarbe gemischt.

Die Register, auf die ein General-Combiner zugreifen kann, finden Sie in der Tabelle unten.

Um den Überblick zu behalten, beschränken wir uns auf die *nvparse*-Variante. Bei den Registern können Sie jeweils getrennt auf RGB- (*col0.rgb*) und Alpha-Werte (*col0.a*) zugreifen oder nur mit dem Blauwert (*col0.b*) arbeiten. Auf alle Input-Werte können Sie ein Mapping anwenden. Damit verändern Sie die Eingabewerte komponentenweise. Die Mappings finden Sie in der Tabelle unten.

Sie verwenden die Input Mappings, indem Sie statt des Registers den Namen des Mappings und in Klammern den Registernamen schreiben, z.B. *expand(col0)*. Jeder General-Combiner hat vier Input- und drei Output-Register. Sie können jeweils die Register, das Mapping und die berechnete Funktion wählen. Als Beispiel sehen Sie ein einfaches Programm, dessen Operationen Sie anschließend betrachten:

```
nvparse(
  „!!RC1.0
const0 = ( 0.1,
0.2, 0.3, 0.4 );
 { //Beginn des Ge-
neral Combiner
   rgb { // Beginn
des RGB Teils
    spare0 = col0 * tex0;
    scale_by_two();
   }
   alpha { // Alpha Teil
    spare1 = col1 * const0;
   }
  }
  // Final Combiner
  final_product = spare0*tex0;
  clamp_color_sum();
out.rgb = color_sum()+tex0;“ );

glEnable
 ( GL_REGISTER_COMBINERS_NV );
```

Sie können den General-Combiner fünf verschiedene Berechnungen durchführen lassen, die ersten drei betreffen nur den RGB-Teil. Es folgen fünf Programme in *nvparse*-Notation, wobei das Beispiel die *col0*, *col1*- und *tex0*, *tex1*-Register als Input wählt. Die Resultate geben Sie über die Spare-Register an den nächsten General- oder den Final-Combiner.

• Zweifaches Skalarprodukt (*Dot/Dot/Discard*):

```
spare0 =
 expand(col0) . expand(tex0);
spare1 =
 expand(col1) . expand(tex1);
```

• Skalarprodukt, komponentenweise Multiplikation (*Dot/Mult/Discard*):

```
spare0 =
 expand(col0) . expand(tex0);
spare1 = col1 * tex1;
```

• Komponentenweise Multiplikation, Skalarprodukt (*Mult/Dot/Discard*):

```
spare0 = col0 * tex0;
spare1 =
 expand(col1) . expand(tex1);
```

• Komponentenweise Multiplikation mit Vergleich, wobei *discard* ein internes Temporär-Register darstellt:

```
mux(AB,CD) =
 (Spare0.a < 0.5) ? AB:CD;
discard = col0 * tex0;
discard = col1 * tex1;
spare1 = mux();
```

• Komponentenweise Multiplikation mit Addition:

```
(spare1=discard+spare0)
discard = col0 * tex0;
spare0 = col1 * tex1;
spare1 = sum();
```

Die Resultate der Berechnung können Sie anschließend skalieren und verschieben (*Scale/Bias*), wozu Sie entsprechende Befehle eingeben. Die *Scale/Bias*-Optionen entnehmen Sie der Tabelle auf der nächsten Seite.

Das Ergebnis eines General-Combiners können Sie im nächsten verwenden. Wenn Sie alle Berechnungen durchgeführt haben, setzen Sie die Resultate im Final-Combiner zusammen.

Der Final Combiner

Der Final-Combiner kennt die gleichen Register wie ein General- Combiner, allerdings sind alle Werte hier nur lesbar. Als Input-Werte werden das *spare0*-Register, die Specular-Farbe, sechs weitere RGB-Inputs und ein Alpha-Wert akzeptiert. Als Input-Mappings stehen nur *unsigned(...)* und *unsigned_invert(...)* zur Verfügung. Der Ablauf der Berechnung im Final-Combiner gestattet es Ihnen, Teile auszulassen.

Zunächst können Sie das Final Product berechnen. Dabei multiplizieren Sie zwei beliebige Input-Register. Das Ergebnis steht für weitere Berechnun-

DIE REGISTER EINES GENERAL-COMBINER

Bedeutung	Name	Zugriff (read/write)
Diffuse Farbe	col0	r/w
Specular Farbe	col1	r/w
Farbe aus Textur 0	tex0	r/w
Farbe aus Textur 1	tex1	r/w
Farbe aus Textur 2	tex2	r/w
Farbe aus Textur 3	tex3	r/w
Spare0 (Arbeitsregister)	spare0	r/w
Spare1 (Arbeitsregister)	spare1	r/w
Farb-Konstante	const0	read only
Farb-Konstante	const1	read only
Fog-Farbe und Faktor	fog	read only RGB
Register enthält immer 0	zero	read only
Discard (Internes Register)	discard	write only

DIE INPUT MAPPINGS DER GENERAL-COMBINER BEI INPUT WERT *X*

Name	Beispiel	Funktion	Wertebereich
Signed Identity	tex0	f(x)=x	[-1,1] -> [-1,1]
Unsigned Identity	unsigned(tex0)	f(x)=max(0,x)	[0,1] -> [0,1]
Expand Normal	expand(tex0)	f(x)=2*max(0,x)-1	[0,1] -> [-1,1]
Half Bias Normal	half_bias(tex0)	f(x)=max(0,x)-0.5	[0,1] -> [-.5,.5]
Signed Negate	-tex0	f(x)=-x	[-1,1] -> [1,-1]
Unsigned Invert	unsigned_invert(tex0)	f(x)=1-min(max(0,x),1)	[0,1] -> [1,0]
Expand Negate	-expand(tex0)	f(x)=-2*max(0,x)+1	[0,1] -> [1,-1]
Half Bias Negate	-half_bias(tex0)	f(x)=-max(0,x)+0.5	[0,1] -> [.5,-.5]
Signed Identity	tex0	f(x)=x	[-1,1] -> [-1,1]

gen zur Verfügung. Sie greifen darauf auf ein Register mit dem Namen *final_product* zu.

```
final_product =
col0 * unsigned_invert(tex0);
```

Weiterhin berechnen Sie die *Final Color Sum*. Diese addiert komponentenweise die Werte von *spare0.rgb+col1.rgb*. Da sich der Wertebereich über *[0;2]* erstreckt, können Sie das so genannte *Clamping* aktivieren: Werte größer als *1* werden auf *1* gesetzt. Dazu verwenden Sie den *clamp_color_sum()*-Befehl.

Alle Register, zusammen mit *final_product* und *color_sum*, stehen nun zur Verfügung, um das RGB-Tripel der endgültigen Farbe mit der Final-Combiner-Funktion zu berechnen. Diese Funktion kann zwischen zwei Farben linear interpolieren und die Werte addieren:

```
A * B + ( 1 - A ) * C + D
```

Für *A*, *B*, *C* und *D* nutzen Sie alle Register mit der Ausnahme, dass *A* nicht *color_sum* sein darf. In *nvparse* weisen Sie die resultierende Farbe dem *out*-Register zu. Die Zuweisung kann verschiedene Formen annehmen, die Spezialfälle der obigen Formel darstellen. Verschiedene Beispiele sehen Sie hier, die Mappings für *A*, *B*, *C* und *D* sind dahinter angegeben:

```
// Zuweisung
// A=zero, B=zero, C=zero,D=tex0
out.rgb = tex0;

// Produkt: A=zero, B=egal,
// C=final_product, D=zero
out.rgb = tex0 * final_product;

// Summe: A=zero, B=egal,
// C=tex0, D=final_product
out.rgb = tex0 + final_product

// Interpolation und Summe
// A=tex1.a, B=tex0,
//C=color_sum, D=const1
out.rgb = lerp
(tex1.a, tex0, color_sum)+const1
```

Für den Alpha-Input-Wert führt der Final-Combiner nur ein Mapping durch. Sie können diesen Wert also nicht weiter modifizieren. Die Konstanten *const0* und *const1* können Sie am Anfang des *nvparse*-Programms angeben, diese gelten dann für den Final- und die General-Combiner. Bei einer neuen GeForce-3/4-Karte können Sie die Konstanten für jeden General-Combiner separat spezifizieren.

Das Beispiel-Programm

Das Programm zu dieser Ausgabe erzeugt eine animierte DSDT-Textur. Die Textur-Units werden mit *nvparse* und dem Register- Combiner konfiguriert.

In der Initialisierungsphase wird eine Heightmap, eine 8-Bit-Graustufen-Bitmap, geladen und daraus eine RGB-Textur generiert, die für jeden Pixel der Heightmap die Höhendifferenzen in *x*- und *y*–Richtung in der Grün- und Blaukomponente enthält:

```
// Heightmap Daten
GLubyte heightMap[ 256*256 ];

// RGBA Texture
GLubyte deltaMap[ 256*256*4 ];
memset( delta, 0, 256*256*4 );

for ( j = 0; j < 256; j++ )
  for ( i = 0; i < 256; i++ )
  {
    int ofs = i + (j << 8 );
    h = heightMap[ ofs ];
dx=heightMap[ (ofs+1)&65535 ]-h;
dy=heightMap[(ofs+256)&65535]-h;
    dx += 64;
    dy += 64;
    delta[ ofs * 4 + 1 ] = dx;
    delta[ ofs * 4 + 2 ] = dy;
  }
```

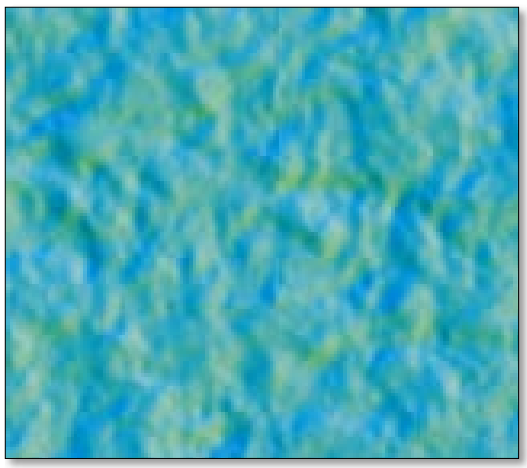
Bevor Sie einen neuen Frame rendern, setzen Sie den Viewport von OpenGL auf eine Größe von 256 x 256 Pixel. Dort zeichnen Sie diese Textur viermal übereinander mit additivem Blending und zeitlich abhängiger Verschiebung der Textur-Koordinaten. Das Blending konfigurieren Sie mit den Register-Combinern. Die vier Textur-Units aktivieren Sie mit:

```
nvpase( „!!TS1.0 texture_2d();
 texture_2d(); texture_2d();
texture_2d();“ );
```

Durch das vierfache Zeichnen mit Blending ergeben sich ständig wechselnde Farbmuster. Die Farben sind grünblau, da in diesen Texture-Farbkanälen die Differenzen der Heightmap gespeichert sind. Diese Farbmuster sollen nur als animerte DSDT-Texturen verwendet werden. Dazu kopieren Sie die RGBA-Daten zunächst in den Speicher und erzeugen daraus die DSDT-Textur:

```
// Speicherbereich
GLubyte deltaMap32[ 256*256*4 ];
glReadPixels( 0, 0, 256, 256,
GL_RGBA, GL_UNSIGNED_BYTE,
deltaMap32 );

// DSDT Bumpmap
GLubyte bumpMap[ 256*256*2 ];

// DSDT Map erzeugen
GLubyte *bumpy = bumpMap;
GLubyte *bumpy32 = deltaMap32;
for ( i = 0; i < 256*256; i++ )
{
  *(bumpy++) = *(bumpy32++);
  *(bumpy++) = *(bumpy32++);
  bumpy32+=2;
}

// und an OpenGL übergeben
glBindTexture( GL_TEXTURE_2D,
 bumpMapTexture );
glTexImage2D( GL_TEXTURE_2D, 0,
 GL_DSDT_NV, 256,256, 0,
 GL_DSDT_NV, GL_UNSIGNED_BYTE,
 bumpMap );
```

Wählen Sie anschließend die DSDT-Map für Textur-Unit *0* und eine beliebige 2D-Textur für Unit *1*, erhalten Sie einen Bump-Mapping-Effekt. Aktivieren Sie die Textur-Shader, wie zu Beginn des Artikels vorgestellt, ist der Effekt aus unserem Beispielprogramm fertig.



DIE DSDT-TEXTUR aus dem Beispielprogramm als Grün-Blau-Textur

Die Anwendungen der Textur-Shader und Register-Combiner sind sehr vielfältig. Vor allem die komplexen Features wie Dot-Product-Bumpmapping erfordern zusätzliche Vertex-Programme. Der Grund: Bestimmte Effekte brauchen Normalen-Vektoren. Diese werden als Farbwerte übergeben, was Vertex-Programme vorbereiten. ET

Links und Literatur:

www.dachsbacher.de/pcu
www.nvidia.com
www.ati.com
www.3dconcept.ch/cgi-bin/showarchiv.cgi?show=2130

DIE *SCALE/BIAS*-OPTIONEN DER GENERAL-COMBINER

Name	Befehl	Funktion
No Scale	–	f(x) = x;
Scale by 1/2	scale_by_one_half();	f(x) = 0.5x
Scale by 2	scale_by_two();	f(x) = 2x
Scale by 4	scale_by_four();	f(x) = 4x
Bias by -1/2	bias_by_negative_one_half();	f(x) = x - 0.5
Bias by -1/2, Scale by 2	bias_by_negative_one_half_ scale_by_two();	f(x) = 2(x-0.5)