workshop

Vertex- und Fragment-Programme mit OpenGL

Beleuchtung – Punkt für Punkt

Mit Geometrie- und Textur-Verarbeitung erzielen Sie Bumpmapping-Effekte. Neueste Grafikkarten verfügen über programmierbare Fragment Shaders, um die Beleuchtung für jeden Pixel zu berechnen.

Carsten Dachsbacher

Da moderne Grafikkarten frei programmierbar sind, lassen sich realistische Beleuchtungseffekte in Echtzeit darstellen. Den ersten Schritt in dieser Richtung stellen die Vertex-Programme dar. Diese von Direct3D-Experten auch als Vertex Shaders bezeichnete Technik führte nVidia mit der GeForce-Grafikkarten-Serie ein. Auf Pixelbasis stehen seit längerem die Texture Shaders und Register Combiners (nVidia) bzw. Pixel/Fragment Shaders (ATI/Direct3D) zur Verfügung.

Die neueste Grafikkarten-Generation wie ATI Radeon 9500/9700 und nVidia GeForce FX ist frei programmierbar auf der Fragment-(Pixel-) Stufe (also im Rasterisierungsteil der Grafik-Pipeline). Sie können in einer Art Assembler-Sprache programmieren, ähnlich den Vertex-Programmen, mit einem Hochsprachen-Compiler wie nVidias Cg oder der HLSL (High Level Shading Language) von DirectX9.

In OpenGL sind Hersteller übergreifende Standards für Vertex- und Fragment- Programme festgelegt worden. Wir berechnen für jeden Pixel – statt nur für jeden Vertex – die Beleuchtung (Per-Pixel-Lighting) mit dem vollständigen Phong- Beleuchtungsmodell und führen Bumpmapping durch. Wer nicht über die neueste Grafikkarte verfügt, erfährt, wie er

Die Attribute für das Vertex-Programm

Attribut	Bedeutung
`position`	Vertex-Koordinate
`texcoord[0]`	Textur-Koordinate
`texcoord[1]`	Binormale
`texcoord[2]`	Tangente
`normal`	Normale

Pixel Lighting mit dem Beispielprogramm: Wagen Sie den Schritt von der Praxis der aufregenden Ego-Shooter in die Theorie, die diese virtuellen Welten erschafft.

ohne Fragment-Programme, nur mit Vertex-Programmen Bumpmapping-Effekte rendern kann.

Das Phong-Beleuchtungsmodell

Phong ist das am häufigsten eingesetzte Beleuchtungsmodell in der Computergrafik. Es wurde 1975 von Phong Bui-Toung (für nicht perfekte Reflektoren) entwickelt und dient dazu, die Farbe eines Oberflächenpunkts zu bestimmen.

Dazu benötigen Sie dessen Normale *N*, den Vektor zur Lichtquelle *L*, den Vektor der reflektierten Lichtrichtung *R* und den Vektor zum Betrachter *V*. Die Grafik auf der folgenden Seite verdeutlicht den Zusammenhang der vorkommenden Vektoren. Weiterhin fließen in die Formel die Farbe der Lichtquelle *I* und die Eigenfarbe der Oberfläche *O* ein. Weitere Oberflächenparameter sind das ambiente, diffuse und spekular reflektierte Licht, gegeben durch die Koeffizienten *k(a)*, *k(d)* und *k(s)*. Der Attenuation Faktor *f_att* gibt die Abnahme der Intensität der Lichtquelle in Abhängigkeit zum Abstand an.

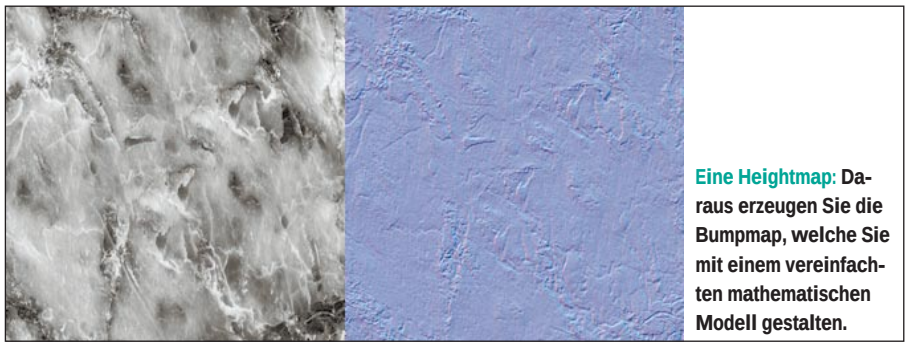
```
F=k(a)·I·O + f_att·I·
        [ k(d)·O·(N dot L) +
        gloss·k(s)·(R·V)^n ]
```

Die Formel enthält Farbvektoren und berechnet RGB-Komponenten. Der Koeffizient *n* dient dazu, Glanzlichter zu modellieren. Größere Werte für *n* resultieren in kleineren schärferen Glanzlichtern. Der *gloss*-Faktor modelliert Unregelmäßigkeiten in der spekularen Reflexion, um glänzende und nicht glänzende Stellen auf einer metallischen Oberfläche darzustellen.

Bumpmapping

Beim Phong-Modell hängen viele Parameter entweder von den Oberflächeneigenschaften wie der Farbe ab oder sind durch die Lage des Objekts in der Szene relativ zur Lichtquelle und zum Betrachter bestimmt (wie durch *L*, *R* und *V*). Sie können daher nur in die Beleuchtungsberechnung eingreifen, indem Sie die Normale verändern. Genau das geschieht beim Bumpmapping. Das Verfahren speichert die jeweilige Oberflächen-Normale, codiert per RGB-Farbwert in einer Textur, und berechnet so die Beleuchtung.

Wir setzen voraus, dass Sie die Position der Lichtquelle im *Object Space*, also relativ zum Koordinatensystem, in dem die Object Vertices definiert sind, bestimmt haben. In einer Textur für Ihr 3D-Objekt ist die Normale gespeichert, die Sie zur Beleuchtungsberechnung verwenden. Das heißt, jedem Punkt der Oberfläche ist ein eindeutiger (unikater) Texel der Textur zugeordnet. Dieser Texel enthält die Normale in codierter Form. Er hängt von der Beschaffenheit der Oberfläche ab.



Eine Heightmap: Daraus erzeugen Sie die Bumpmap, welche Sie mit einem vereinfachten mathematischen Modell gestalten.

Da diese Methode schwierig umzusetzen ist, verwenden wir fürs Bumpmapping eine andere Technik, bei der jedem Vertex nicht nur eine Koordinate, sondern ein Tangent Space (ein eigenes Koordinatensystem) zugeordnet wird. Dieses begnügt sich mit den drei Vektoren Tangente, Binormale und Normale.

Wählen Sie Tangente und Binormale so, dass sie entsprechend den Vektoren des Texture-Mappings verlaufen. Solche Tangent Spaces können Sie mit 3D-Programmen aufspannen. Ein Tangent Space muss der Anforderung genügen, dass die Normale *n* der Z-Achse entspricht. So berechnen Sie den Tangent Space für einen Vertex in Abhängigkeit von *n*:

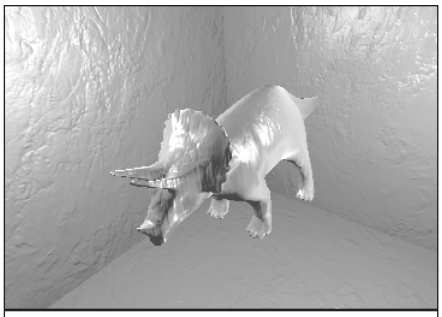
```
VECTOR up = { 0.0, 0.0, -1.0 };

// X->Kreuzprodukte bilden
binormal = n  X up;
tangente = bi X n;
```

Für jeden Vertex speichern Sie die drei Vektoren (in normalisierter Form). Für das Bumpmapping wird nun jeder Vektor von der Vertex-Position zur Lichtquelle, z.B. mit einem Vertex Programm, in dessen Tangent Space transformiert, was drei Skalarprodukten entspricht:

```
// Vektor vertex -> lichtquelle
toLight =
 lightPosition - vertexPosition;
tangentLight.x =
  binormale dot toLight;
tangentLight.y =
  tangente dot toLight;
tangentLight.z =
  normale dot toLight;
```

Die drei Komponenten des *tangentLight*-Vektors speichern Sie in einer Textur-Koordinaten. Diese und somit der Vektor zur Lichtquelle wird beim Rendering der Dreiecke zwischen den Eckpunkten interpoliert und steht für jeden Pixel zur Verfügung. Allerdings bewirkt die lineare Interpolation der Komponenten, dass die Länge des Vektors nicht konstant ist.



Pixel-Phong-Beleuchtung: Gestaltung ohne Eigenfarbe der Oberflächen

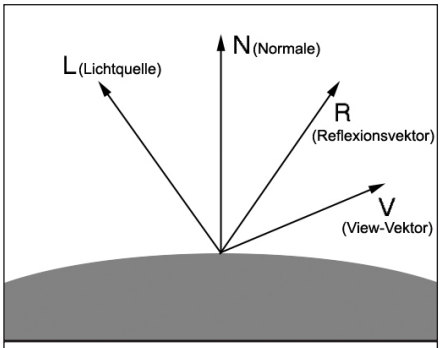
Fürs vollständige Phong-Beleuchtungsmodell benötigen Sie die Richtung zur Kamera im Tangent Space, die Sie analog berechnen.

Der Vorteil des Tangent Space Bumpmapping liegt in der Texturierung der Objekte. In der Bumpmap Texture sind die Normalen codiert, die für eine Fläche mit der Normalen *(0,0,1)* gültig sind. Durch die Tangent-Space-Transformation können Sie ein beliebiges Mapping dieser Textur auf das 3D-Objekt verwenden. Solche Bumpmap-Texturen werden meist aus *Heightmaps* erzeugt. Eine Heightmap ist ein Graustufen-Bitmap, wobei die Graustufe eines Texels dessen Höhe repräsentiert. Mit geeigneten Tools wie von nVidia (**http://developer.nvidia.com/view.asp?IO=map_generator**), können Sie daraus eine Bumpmap erzeugen.

Programmierbare Grafik-Pipeline

Um den Tangent Space und die Beleuchtung zu berechnen, benötigen Sie die OpenGL Extensions für die Vertex- bzw. Fragment-Programme: *GL_ARB_VERTEX_* und *GL_ARB_FRAGMENT_PROGRAM*. Die Spezifikationen aller OpenGL Extensions finden Sie unter **http://oss.sgi.com/projects/ogl-sample/registry/**. Beide Erweiterungen nutzen dieselbe Schnittstelle, um den Assembler Code eines *Program*, gespeichert in einem String, zu übergeben und zu nutzen. Die Funktionszeiger laden Sie im Beispielprogramm mit dem *wglGetProcAddress(...)*-Befehl.

Die Beispiel Szene: **Per Pixel-Phong-Beleuchtung ohne Gloss Mapping erzeugen Sie eindrucksvolle Schattierungen.**



Vier Vektoren: **1975 hat Phong Bui-Toung sein Beleuchtungsmodell für nicht perfekte Reflektoren entwickelt.**

Als erstes fordern Sie immer einen Identifier für Ihr Vertex- oder Fragment-Programm an:

```
GLuint programID;
glGenProgramsARB(1, &programID);
```

Ob es sich hier um ein Vertex- oder Fragment-Programm handelt, bestimmt bei den folgenden Befehlen das Target *GL_VERTEX_PROGRAM_ARB* oder *GL_FRAGMENT_PROGRAM_ARB*.
Im nächsten Schritt erweitern Sie das Programm. Sie übergeben den String mit dem Programmcode, den Sie z.B. aus einer Textdatei vorher eingelesen haben:

```
glBindProgramARB(
  GL_VERTEX_PROGRAM_ARB,
  programID );

char *programCode = „...“;

glProgramStringARB(
  GL_VERTEX_PROGRAM_ARB,
  GL_PROGRAM_FORMAT_ASCII_ARB,
  strlen( programCode ),
                programCode );
```

Um abzufragen, ob ein Fehler in Ihrem Code enthalten ist, liefert Ihnen die folgende Methode das Offset des Fehlers oder den Wert *-1*, falls alles korrekt war:

```
int ep;
glGetIntegerv(
  GL_PROGRAM_ERROR_POSITION_ARB,
  &ep );
```

Vertex-Programme

Ein Vertex-Programm verarbeitet immer nur einen Vertex. Ihr Einsatzgebiet reicht von einer Koordinaten-Transformation zu Vertex Blending für Animationen, Beleuchtungs- und Fog-Berechnungen und mehr.
Als Eingabedaten stehen die Vertex-Attribute wie Koordinate, Normale, Textur-Koordinaten, Farbe usw. zur Verfügung. Weiterhin nutzen Sie OpenGL States wie Materialeigenschaften, Lichtquellen und Parameter. Letztere setzen sich aus mindestens 96 4-Komponenten-Vektoren pro OpenGL-Kontext, ebenso vielen pro Vertex-Programm und weiteren im Code definierten Konstanten zusammen. Sie rechnen mit mindestens zwölf 4-Komponenten-Vektoren und einem Adressregister. Die Operationen umfassen Addition, Substraktion, Skalar- und Kreuzprodukte, Vergleiche, Minimum-, Maximum- sowie Absolutwert-Bildung, zusätzlich Skalar-Operationen wie Potenzierung, Logarithmen, Reziproke und Reziproke-Wurzel-Bildung.
Den Aufbau der Vertex-Programme stellen wir anhand eines einfachen Beispiels vor. Es soll die Koordinaten eines Vertex transformieren, den normalisierten Vektor zur Lichtquelle im Tangent Space berechnen und in der Vertex-Farbe speichern. Die Vertex-Attribute übergeben Sie mit den üblichen OpenGL-Befehlen wie *glVertex3f(...)* oder *glTexCoord3f(...)*.
Die Position der Lichtquelle im Object Space speichern Sie als Parameter. Diesen übergeben Sie folgendermaßen von Ihrem Programm aus:

```
glProgramEnvParameter4fARB(
 GL_VERTEX_PROGRAM_ARB, 0, 1.0f,
           1.0f,  1.0f,  1.0f );
```

Der zweite Parameter bezeichnet die Speicherstelle. Jedes Vertex-Programm beginnt mit der Kennung *!!ARBvp1.0*. Für Programmparameter können Sie Aliasnamen vergeben. Die Position der Lichtquelle ist in einem solchen Parameter gespeichert. Um darauf zuzugreifen, verwenden Sie *program.env[1]* oder führen den Alias *lightPosition* ein:

```
!!ARBvp1.0

PARAM lightPosition =
              program.env[1];
```

Aliasnamen für Vertex Attribute definieren Sie folgendermaßen:

```
ATTRIB binormal =
      vertex.texcoord[ 1 ];
ATTRIB tangent  =
      vertex.texcoord[ 2 ];
ATTRIB normal   = vertex.normal;
```

Aliasnamen für Ausgabewerte definieren Sie analog mit

```
OUTPUT tangentLightNormalized
                 = result.color;
```

Auf alle Werte können Sie auch ohne die Aliases zugreifen. Vertex-Attribute erreichen Sie mit *vertex._*, Ausgabewerte mit *result._*. Temporäre Variablen für die Berechnung definieren Sie mit

```
TEMP toLight, tangentLight,
temp, invLen;
```

Jetzt geht es an den Programmcode. Transformieren Sie die Vertex-Koordinaten. Dann berechnen Sie den Vektor von der Vertex-Koordinaten zur Lichtquelle und speichern diesen in *toLight*. Diesen Vektor transformieren Sie mit drei Skalarprodukten in den Tangent Space (gespeichert in *tangentLight*):

```
# Transformation mit
# Modelview+Projection Matrix
PARAM mvp[4] =
          { state.matrix.mvp };
DP4 result.position.x, mvp[0],
                vertex.position;
DP4 result.position.y,
       mvp[1], vertex.position;
DP4 result.position.z, mvp[2],
                vertex.position;
DP4 result.position.w, mvp[3],
                vertex.position;

ADD toLight,  lightPosition,
               -vertex.position;

DP3 tangentLight.x,
              binormal, toLight;
DP3 tangentLight.y,
              tangent,  toLight;
DP3 tangentLight.z,
              normal,   toLight;
```

Sie können durch Angabe von *.x*, *.y* etc. entweder den Schreibzugriff im Zielregister auf

diese Komponente beschränken oder im Falle eines Quellregisters diese Komponente vervielfachen. Es ist auch *Swizzling* möglich: Vektor-Operanden können nicht nur negiert werden, sondern deren Komponenten lassen sich auch beliebig anordnen und vervielfachen. Bei Skalaroperationen müssen Sie die verwendete Vektorkomponente spezifieren, wie Sie dies bei der Normalisierung des Lichtvektors sehen. Kommentare im Programmcode beginnen mit einem Rautezeichen, mit *END* wird das Programm abgeschlossen:

```
# quadrierte Länge des Vektors
DP3 temp, tangentLight,
                    tangentLight;
# 1/sqrt(länge)
RSQ inverseLength, temp.x;
# normalisiert Vektor berechnen
MUL tangentLightNormalized,
    tangentLight, inverseLength;
END
```

Mit diesem Vertex-Programm können Sie den diffusen Teil der Phong- Beleuchtungsmodells mit Bumpmapping berechnen, wenn Ihre Grafikkarte die *GL_EXT_texture_env_combine*-Erweiterung unterstützt. Dazu wählen Sie für die erste Textur-Stage eine Bumpmap-Textur. Konfigurieren Sie das Textur-Environment so, dass ein Skalarprodukt zweier Vektoren (codiert als Farben) durchgeführt wird. Die Parameter für *glTexEnvi(GL_TEXTURE_ENV, ?, ?)* sind:

```
GL_TEXTURE_ENV_MODE:
         GL_COMBINE_EXT
GL_COMBINE_RGB_EXT :
       GL_DOT3_RGBA_EXT
```

Operanden sind der interpolierte Lichtvektor im Tangent Space (gespeichert in der Farbe):

```
GL_SOURCE0_RGB_EXT :
               GL_PREVIOUS_EXT
GL_OPERAND0_RGB_EXT:
                  GL_SRC_COLOR
```

und die Normale aus der Bumpmap:

```
GL_SOURCE1_RGB_EXT: GL_TEXTURE
GL_OPERAND1_RGB_EXT:GL_SRC_COLOR
```

Aktivieren Sie die Vertex-Programme vor dem Rendering mit der Eingabe

```
glEnable( GL_VERTEX_PROGRAM_ARB )
```

Für eine ganz genaue Berechnung normalisieren Sie die Vektoren. Das gelingt mit den Textur-Einheiten, wenn Sie Normalizing Cube Maps verwenden.

Fragment-Programme

Mit den Fragment-Programmen berechnen Sie die Beleuchtung in Floating-Point-Genauigkeit. Ein Fragment-Programm ersetzt Texturierung, Farbberechnung und das Fogging der OpenGL-Pipeline. Weiterhin können Sie andere Operationen durchführen, die bisher spezielle Erweiterungen übernommen haben, wie Tiefenvergleiche für Depth Map Shadows oder Dependent Texture Lookups für Environment Bump Mapping. Für diese Aufgaben greifen Sie auf einen, dem Vertex-Programm sehr ähnlichen, Befehlssatz zu. Als wichtige neue Instruktionen nutzen Sie das Auslesen von Texturen, das Fragment Killing (bedingtes Nichtzeichnen eines Fragments) und die Option, den Tiefenwert eines Fragments zu modifizieren.

Ein Fragment-Programm besitzt mindestens zehn Eingabe-Attribute, auf die Sie mit *fragment._* zugreifen, 24 Programmparameter, 16 temporäre Register und kann mindestens vier Texture Indirections, 48 ALU-Instruktionen (Arithmetic Logic Unit) und 24 Textur-Instruktionen durchführen. Diese Vielzahl gewährt zahlreiche neue Grafikeffekte.

Syntax und Semantik entsprechen denen der Vertex-Programme, auch was die Aliasnamen angeht. Als Beispiel dient das Fragment-Programm, das das Phong- Beleuchtungsmodell auswertet. Dieses benötigt außer dem Licht- noch den Betrachtervektor, der zusätzlich im Vertex-Programm berechnet wird. Alle Eingabewerte sehen Sie in der Tabelle unten#.

Das Programm beginnt wieder mit einer Kennung *ARBfp1.0* und Ihren Alias-Definitionen entsprechend der Tabelle. Unser Beispielprogramm benötigt einige temporäre Variablen, die Sie dem Quelltext entnehmen.

Der Programmcode beginnt damit, dass Sie per *TEX*-Befehl die Texturen auslesen. Diese sind die Farbe der Oberfläche, die Bumpmap und der Gloss-Faktor. Die Parameter sind Zielregister, Textur-Koordinatenregister, Textur-Stage und zuletzt der Textur-Modus, um auf 2D-, 3D- und Cubemap-Texturen zugreifen zu können:

```
TEX surfaceColor, texCoord0,
                  texture[0], 2D;
TEX bumpNormal,
      texCoord0, texture[1], 2D;
TEX glossFactor,
      texCoord0, texture[2], 2D;
```

Jetzt müssen Sie die Normale aus dem RGB-Wert decodieren, also den Wertebereich der Komponente *[0,1]* auf *[-1,1]* strecken – mit den Konstanten *(2,2,2,2)* und *(1,1,1,1)* – und anschließend normalisieren:

```
MAD bumpNormal, bumpNormal,
                      two, -one;
```

Ebenfalls normalisieren Sie den Betrachter- und Lichtvektor, um akkurat rechnen zu können. Bei der Normalisierung des Lichtvektors erhalten Sie als Zwischenergebnis dessen Länge, mit der Sie die Abnahme der Lichtintensität berechnen können. Eine quadratische Abnahme können Sie mit nur zwei Instruktionen berechnen:

```
(cAtt=(1.0,0.0,0.1,0.0)):
MAD att, distance.z,
              cAtt.z, cAtt.x;
RCP att, att.x;
```

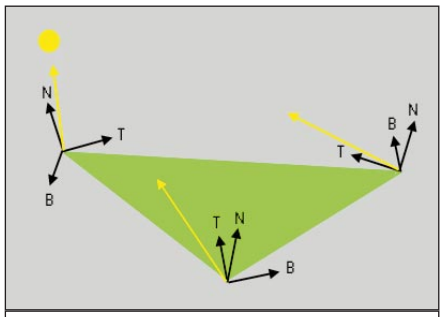
Den Reflexionsvektor berechnen Sie mit

```
DP3 temp, bumpNormal,
                  lightVector;
MUL temp,  temp,  bumpNormal;
```

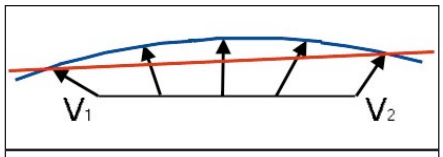
Jetzt haben Sie alle Parameter und Koeffizienten für das Phong-Modell bestimmt und können es auswerten.Mit den Skalarprodukten

```
# N dot L
DP_SAT diffuse,bumpNormal,lightVector;
```

und der Kombination aller Zwischenergebnisse beenden Sie das Programm. : et



Tangent Space: **Dieses mathematische Modell legt die Normale in die Z-Achse.**



Lineare Interpolation: **Aus den Vektorkomponenten resultiert eine Längenänderung.**

Die Eingabewerte der Fragment-Programme

Eingabewert	Bedeutung
program.env[0]	ambientes Licht
program.env[1]	diffuses Licht
program.env[2]	spekulares Licht
program.env[3]	Phong Exponent
fragment.texcoord[0]	Textur-Koordinate
fragment.texcoord[1]	Lichtvektor L
fragment.texcoord[2]	Betrachtervektor V