3D-Grafiken verbessern durch Shader

Glänzende Effekte

Für die gängigen Computerspiele führen moderne Grafikkarten viele Berechnungen selbstständig aus und rendern 3D-Szenen in akzeptabler Qualität. Eine realistischere Darstellung schaffen Shader-Programme, die auf dem Grafikchip laufen und unter Linux in der Hochsprache Cg programmiert wurden. Oliver Frommel, Stephan Siemen

OpenGL ist der Standard für 3D-Grafik unter Linux [1], [2]. Die Grafikkarte selbst ließ sich bislang damit jedoch nicht programmieren. Die neueste Generation von Grafikkarten erlaubt es, eigene kleine Programme (Shader) zu schreiben, die der Grafikchip (GPU, Graphics Processing Unit) selbst abarbeitet und Modelle dadurch schneller und realistischer rendert.

Alles nur Schein

Grafikkarten sind vor allem auf eine hohe Verarbeitungsgeschwindigkeit ausgelegt. Deshalb benutzen sie vereinfachte physikalische Modelle und Algorithmen, um Licht, Farbeffekte und Texturen zu berechnen. Bei programmierbarer Grafikhardware entscheidet der Entwickler, ob er ein schnelleres Rendering oder lieber eine realistischere Szene erreichen will. Shader bieten ihm beide Möglichkeiten gleichzeitig.

Die Idee der Shader stammt aus den Studios für Animationsfilme. So schuf die Firma Pixar in den späten 80er Jahren für ihr Rendering-Interface Renderman eine eigene Shader-Sprache [3]. Die Anwendung beschränkte sich jedoch auf das relativ langsame Batch-Rendering einzelner Filmframes. Mit einem Shader berechnen die Renderer – egal ob in Echtzeit oder nicht – für jeden Geometriepunkt respektive dargestellten Pixel das Aussehen, statt nur statisch eine einzige Farbe oder Textur zu verwenden. Trotz einfacher Geometrie erscheinen damit gerenderte Objekte mit komplexer Oberflächenstruktur.

Moderne Grafikkarten beherrschen diese Technologie in Echtzeit. Der Programmierer lädt seinen Shader-Code in die Karte und die führt ihn während des Renderings sehr schnell für jeden einzelnen Punkt aus. Musste der Programmierer bislang zu einer Assembler-Sprache greifen, die dem eingesetzten Grafikchip entspricht, gibt es dafür mittlerweile Hochsprachen. Ein Compiler übersetzt den in einer solchen Sprache abgefassten Shader in Code für die jeweilige GPU. Shader beschreiben keine Geometrien oder Objekte, das ist immer noch die Aufgabe von OpenGL. Aber sie beeinflussen, wie die Grafikkarte Transformationen, Licht und Farben verarbeitet.

Die Grafik-Pipeline

Da Shader-Programmierung stark von der Hardware abhängt, ist es nötig, sich mit deren Arbeitsweise zu beschäftigen. Der funktionale Aufbau einer 3D-Grafikkarte wird als Grafik-Pipeline bezeichnet. Schrittweise rechnet sie dreidimensionale Punkte (Vertices), die die Geometrie beschreiben, in farbige Punkte (Pixel) des zweidimensionalen Framebuffers um. Den kann man sich als großes Raster vorstellen, das auf dem Bildschirm erscheint (Abbildung 1).

Vertices enthalten neben den Koordinaten (x, y, z) auch Texturkoordinaten. Pixel bestehen meist aus drei Farbwerten (RGB, Rot-, Grün- und Blau-Anteile) und einem Alphawert (A), der die Transparenz angibt. Vertex-Shader werden von der GPU für jeden Geometriepunkt (Vertex) eines Modells ausgeführt. Pixel-Shader kommen erst in einer späteren Stufe der Pipeline dran, wenn die GPU aus dem Modell bereits ein Rasterbild berechnet hat (siehe Abbildung 1).

OpenGL-Erweiterungen

Seit der Version 1.2 besitzt OpenGL auch Erweiterungen, die es möglich machen, Shader zu programmieren. Die Erweiterungen »ARB_vertex_program()« und

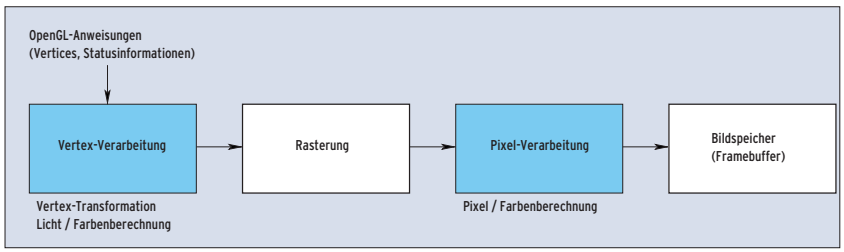

Abbildung 1: Vereinfachte OpenGL-Grafik-Pipeline. Die Teile, die sich bei neueren Grafikkarten frei programmieren lassen, sind blau hinterlegt.

»ARB_fragment_program()« laden den Assembler-Code des Shader auf die GPU. Das Programmieren von Shadern in Assembler ist allerdings abhängig von der genutzten Grafikkarte, also nicht sehr portabel, nicht einmal zwischen verschiedenen Prozessoren desselben Herstellers. Listing 1 zeigt den Assembler-Code eines Vertex-Shaders, der funktional dem Cg-Shader in Listing 3 entspricht.

Die OpenGL-Version 1.5 führt eine eigene Hochsprache zur Shader-Programmierung ein, die OpenGL Shading Language [5]. Die Firma 3DLabs bietet eine Linux-Implementierung an, die aber nur mit den vergleichsweise teuren Karten dieses Herstellers funktioniert. Cg deckt dagegen zurzeit noch ein breiteres Spektrum an Hardware ab. Außerdem ist Cg für jene interessant, die ihre Shader auch unter Direct X einsetzen wollen.

Tools

Programmierbare Shader funktionieren nur mit neueren Grafikkarten. Zurzeit sind das die Karten von Nvidia (Geforce 3 und 4 sowie Geforce FX) und ATI (ab Radeon 8500) und die Boards von 3DLabs.

Cg-Compiler und Runtime

Nvidia bietet den Cg-Compiler als RPM und als Tarball unter [4] zum Download an. Beide Dateien sind etwa 2,5 MByte groß. Der Compiler »cgc« bringt auch Bibliotheken und Include-Dateien für die Runtime-Version mit. Das ermöglicht es dem Programmierer, in eigenen Code Cg-Shader einzubauen, die das System erst zur Laufzeit in passenden GPU-Assembler-Code übersetzt.

NVSDK

Zusätzlich zum Compiler stellt Nvidia das NVSDK bereit [4]. Es enthält viele Beispiele von Cg-Shadern und OpenGL-Programmen. Diese und die nötigen Libraries muss der Entwickler selbst übersetzen, was einfach durch den Aufruf von »make«, erst im Verzeichnis »LIBS/src« und dann in »DEMOS/OpenGL/src« geschieht.

C for Graphics

Um ihre Nähe zur Programmiersprache C zu zeigen, hat Nvidia die Shader-Sprache „C for Graphics“ oder kurz Cg getauft. Die Sprache ähnelt darüber hinaus Microsofts High Level Shading Language (HLSL), die zu Direct X gehört. Der Cg-Compiler ist Open Source und für OpenGL ab Version 1.4 ausgelegt.

In Cg geschriebene Shader lassen sich auf einer ganzen Reihe von Plattformen nutzen: mit OpenGL unter Linux, Windows, Mac OS X und mit Direct X unter Windows und auf der XBox. Die Windows-Versionen der 3D-Pakete 3DSmax, Softimage|XSI (ab Version 3.0) und Maya (ab Version 4.5) unterstützen ebenfalls Cg-Shader. Obwohl es von den letzten beiden auch Linux-Versionen gibt, müssen deren Benutzer auf dieses Feature aber bislang verzichten.

Cg beherrscht sowohl Vertex- als auch Pixel-Shader. Beide benutzen ähnliche mathematische Funktionen, die die Cg-Standardbibliothek zur Verfügung stellt. Für die Pixel-Shader gibt es zusätzlich noch einige Funktionen, um Texturen zu bearbeiten.

Komplexe Datentypen

Die Cg-Sprache besitzt neben den C-Datentypen wie »int« und »float« eigene Datentypen für Matrizen und Vektoren, in denen sie Koordinaten und Farbwerte speichert. Konstruktoren (ähnlich wie in C++) vereinfachen deren Deklaration. Der Grafikchip führt Berechnungen auf solche Datentypen schnell aus, denn er parallelisiert sie in der Art von SIMD (Single Instruction Multiple Data), verarbeitet also beispielsweise vier Elemente eines Vektors zur selben Zeit. Die folgende Anweisung deklariert einen »float«-Vektor mit vier Elementen:

```
float4 vector = float4(2.3, 1.4, 0.1, 5.0);
```

Natürlich lassen sich die Elemente der Vektoren auch einzeln ansprechen. Swizzling heißt die Technik, die einzelne Elemente über ihren Index vertauscht. Der Index der Elemente entspricht den geometrischen Achsen x, y, z und w. Auf den obigen Vektor angewandt ergibt die Anweisung

```
float4 vector2 = vector.yxzw;
```

den Vektor »(1.4, 2.3, 0.1, 5.0)« und

```
float4 vector3 = vector.xxzz;
```

führt zu »(2.3, 2.3, 0.1, 0.1)«.

Hardware-Profile

Nicht alle Grafikkarten unterstützen dieselben Features, so speichert beispielsweise nur die allerneueste Kartengeneration die Farbwerte als Gleitkommazahlen ab, die älteren tun dies nur als Ganzzahlen. So genannte Profile geben die Fähigkeiten der Hardware dem Cg-Compiler bekannt, der auf dieser Grundlage dann ein passendes Programm für die GPU erzeugt.

Kommen Features neuerer Karten in einem Shader zum Einsatz, läuft er nicht mehr auf älterer Hardware. Der Programmierer darf viele Optimierungen nicht benutzen, wenn der Shader rückwärts kompatibel sein soll. Cg bietet die Funktion »cgGLGetLatestProfile()«, um das neueste, von einer GPU unterstützten Profil auszulesen. So lassen sich mehrere Versionen eines Shader in seinen Code einbauen, um zur Laufzeit den passendsten in die GPU zu laden.

Tabelle 1 zeigt die wichtigsten Profile für den Cg-Compiler unter Linux. Zurzeit

Listing 1: Vertex-Shader in GPU-Assembler

```
!!ARBvp1.0
PARAM c1 = { 0, 1, 0, 1 };
PARAM c0 = { 0, 1, 0, 0 };
ATTRIB v16 = vertex.position;
MOV result.position.xy, v16.xyyy;
MOV result.position.zw, c0.yyxy;
MOV result.color.front.primary, c1;
END
```

unterstützt Cg die Nvidia-Grafikkarten am besten. Für alle anderen Karten bleiben nur die ARB-Profile (ARB steht für das Architecture Review Board [1], das den OpenGL-Standard festlegt). Der für diese Profile erzeugte Assembler-Code sollte auf allen Karten laufen, die die entsprechende OpenGL-Shader-Erweiterung unterstützten.

Compiler oder Laufzeitsystem

Es gibt mit Cg zwei Wege, um Shader zu schreiben. Der erste nutzt den Compiler »cgc«, um Cg-Programme in Assembler zu übersetzen. Die erwähnten OpenGL-Erweiterungen »ARB_vertex_program()« und »ARB_fragment_program()« laden den erzeugten Assembler-Code in die GPU. Dieser Weg ist sehr statisch und erlaubt es nicht, den Shader zu verändern, wenn das Programm einmal läuft.

Eine andere Möglichkeit ist es, den Shader-Code zur Laufzeit (Runtime) zu übersetzen und auszuführen. Dafür enthält Cg die Bibliotheken »libCg.so« und »libCgGL.so« und die zugehörigen Include-Dateien »cg.h« und »cgGL.h«. Die einzelnen Schritte, um einen Cg-Shader zur Laufzeit auszuführen, sind:

- Einen »CgContext« erzeugen, der die Shader aufnimmt: »cgCreateContext()«
- Das Shader-Programm übersetzen und zum Kontext hinzufügen: »cgCreateProgram()«
- Das Programm laden: »cgGLLoadProgram()«
- Shader-Parameter setzen: »cgGLSetParameter4fv()«
- Profile setzen: »cgGLEnableProfile()«
- Den Shader mit der OpenGL-Pipeline verbinden: »cgGLBindProgram()«
- Mit OpenGL-Befehlen die Szene zeichnen, während des Zeichnens kann das Programm zwischen verschiedenen Shadern wechseln
- Das Laufzeitsystem führt den Shader an den definierten Stellen der Pipeline selbst aus
- Ressourcen freigeben: »cgDestroyProgram()«, »cgDestroyContext()«

Listing 2: Error-Callback-Funktion

```
01 ...
02
03 void cgErrorCallback(void){
04    CGerror LastError = cgGetError();
05    if (LastError) {
06       printf("%s\n\n", cgGetErrorString(LastError));
07       printf("%s\n", cgGetLastListing(context));
08       printf("Cg error, exiting...\n");
09       exit(0);
10    }
11 }
12
13 ...
14
15 cgSetErrorCallback(cgErrorCallback);
```

Listing 3: Vertex-Shader 1

```
01 struct vs_output
02 {
03     float4 oPosition : POSITION;
04     float4 oColor0 : COLOR0;
05 };
06
07 vs_output main(float2 position : POSITION)
08 {
09     vs_output OUT;
10
11     OUT.oPosition = float4(position,0,1);
12     OUT.oColor0   = float4(0,1,0,1);
13
14     return OUT;
15 }
```

Vertex-Shader

Vertex-Shader führt die GPU in der Pipeline vor den Pixel-Shadern aus. Deshalb verrichten sie in der Grafik-Pipeline andere Aufgaben:

- Die Objektkoordinaten in Positionen auf dem Bildschirm transformieren (Model View Transformation)
- Texturkoordinaten erzeugen
- Farbwerte am jeweiligen Punkt der Geometrie in Abhängigkeit von der Beleuchtung berechnen

Shader-Programme sollten so kurz wie möglich gehalten werden, da sie sehr oft aufgerufen werden, bei Vertex-Shadern für jeden Punkt der Geometrie einmal. Bei vielen Grafikkarten ist die maximale Anzahl von Befehlen in einem Shader beschränkt.

Fehlerbehandlung

Cg, die GPU und das OpenGL-Subsystem nehmen dem Anwendungsprogramm viel Arbeit ab. Der Nachteil ist, dass viel hinter den Kulissen abläuft und Fehler deshalb schwer zu finden sind. Folglich ist es Pflicht, nach jedem Aufruf einer Cg-Funktion zu testen, ob dabei nicht ein Fehler aufgetreten ist. Zusätzlich bringt Cg die Funktion »cgSetErrorCallback()« mit, die eine Callback-Funktion zur Fehlerbehandlung registriert.
Der Programmierer entscheidet dann beim Schreiben der Funktion, wie er die Fehler behandelt. Listing 2 zeigt eine solche Callback-Funktion und auch, wie man sie mit Cg registriert. Dieses Beispiel aus der Cg-Dokumentation gibt die Art des Fehlers aus und beendet das Programm.

Tabelle 1: Cg-Profile

Profile-Name	Grafikkarte
CG_PROFILE_VP20	NV 20+ (Geforce 3, 4)
CG_PROFILE_VP30	NV 30+ (Geforce FX)
CG_PROFILE_ARBVP1	Alle Grafikkarten, die programmierbare Vertex-Shader unterstützten
CG_PROFILE_FP20	NV 20+ (Geforce 3, 4)
CG_PROFILE_FP30	NV 30+ (Geforce FX)
CG_PROFILE_ARBFP1	Alle Grafikkarten, die programmierbare Pixel-Shader unterstützen

Datenübergabe

Die Strukturen »appdata« und »vs_output« im Listing 4 zeigen, wie Shader Ein- und Ausgabedaten übergeben. So muss jeder Vertex-Shader als Eingabevariable die aktuellen Punktkoordinaten von der Grafik-Pipeline bekommen. Das geschieht hier über das Element »position« in der Struktur »appdata«. Der Rückgabewert der »main«-Funktion ist vom selbst definierten Typ »vs_output«, der zwei Felder besitzt: »oPosition« gibt die Position des Bildpunkts an (also die in den Bildraum transformierte Vertex), »oColor0« die Farbe.

Jedes Shader-Programm braucht solche Strukturen, um Daten von und zur Pipeline zu übergeben. Sie sind die Verbindung zu den Werten, die in der Grafik-

Tabelle 2: Wichtige Typen der Cg-Runtime

Cg-Typ	Bedeutung
CGcontext	Laufzeitkontext für Cg-Shader
CGprogram	Referenz zu einem Shader-Programm
CGparameter	Parameter eines Shaders
CGprofile	Beschreibung eines Hardwareprofils
CGerror	Informationen über Fehler der Cg-Runtime

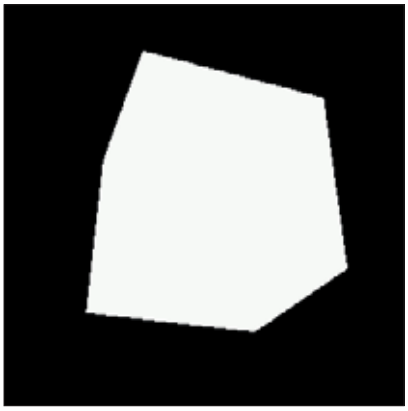

Abbildung 2: Der Einheitswürfel ohne Beleuchtungsberechnung und Shader erscheint nur einfarbig weiß.

Abbildung 3: Der Vertex-Shader aus Listing 3 färbt den Würfel beleuchtungsunabhängig grün ein.

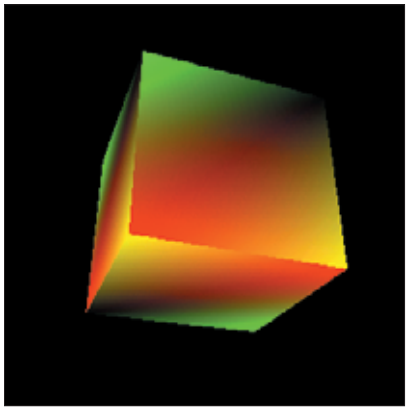

Abbildung 4: Der Vertex-Shader färbt die Oberfläche abhängig von der Oberflächenkoordinate ein.

Pipeline verarbeitet werden. Die Bezeichnungen in Großbuchstaben sind dabei die Namen, unter denen die Werte auf der Pipeline bekannt sind. Die Zeile

```
float4 position : POSITION;
```

weist dem Vektor »position« den Wert zu, der in der Grafik-Pipeline aktuell als »POSITION« geführt wird, nämlich das Ergebnis des letzten »glVertex()«-Aufrufs in OpenGL. Diese automatische Zuweisung heißt in Cg „Binding Semantics".

Pixel-Shader

Was für Vertex-Shader gilt, trifft meist auch auf Pixel-Shader zu. Der wesentliche Unterschied liegt bei den Eingaben und Ausgaben des Shaders. Während das Laufzeitsystem Vertex-Shader für jeden Eckpunkt einer Geometrie ausführt, führt es Pixel-Shader für jeden Pixel eines Bildes aus. Der Pixel-Shader ermittelt die endgültige Farbe eines Pixels. Dafür braucht er die Farbinformationen, die der Vertex-Shader berechnet hat, sowie Informationen über andere Oberflächenparameter, zum Beispiel Texturen. Listing 5 zeigt den Code eines sehr einfachen Pixel-Shaders. Er setzt für jeden Pixel die Farbe, die er einer vorgegebenen Textur entnimmt.

Fazit und Ausblick

Shader ermöglichen es, in eigenen Grafikprogrammen interessante Licht- und Farbeffekte in Echtzeit zu erzeugen. Cg ist zwar nicht die einzige Sprache für die Programmierung von Shadern, aber die erste, die unabhängig von der verwendeten Hardware und über Systemgrenzen hinweg einsetzbar ist. Es bleibt abzuwarten, wie OpenGLs Shader-Sprache angenommen wird, wenn mehr Implementationen verfügbar sind.

Wer mehr über das Programmieren von Shadern mit Cg erfahren möchte, sollte einen Blick in das leider nicht ganz billige Buch „The Cg Tutorial" [6] werfen, das eine sehr umfassende Einführung in Cg gibt. Zum Experimentieren eignet sich Nvidias Cg Labs [7], ein GUI-Programm, das Shader lädt, kompiliert und auf fertige Modelle anwendet. Interessante Artikel, Neuigkeiten und Diskussionen zu Cg-Shadern bietet das zugehörige Online-Forum [8]. ■

Infos

[1] OpenGL: [http://www.opengl.org]
[2] Thomas G. E. Ruge, „Hello 3D-World": Linux-Magazin 04/01
[3] Pixar Renderman: [https://renderman.pixar.com]
[4] Cg Toolkit und NVSDK: [http://developer.nvidia.com/object/cg_toolkit.html]
[5] OpenGL SL: [http://www.opengl.org/developers/documentation/oglsl]
[6] R. Fernando und M. J. Kilgard, „The Cg Tutorial", Addison-Wesley
[7] Cg Labs [http://developer.nvidia.com/object/cg_labs.html]
[8] Cgshaders.org: [http://www.cgshaders.org]

Der Autor

Dr. Stephan Siemen arbeitet als wissenschaftlicher Mitarbeiter an der Essex University (UK). Dort entwickelt er Software zur 3D-Darstellung von Wettersystemen und unterrichtet Studenten in Computergrafik und Programmierung.

Listing 4: Vertex-Shader 2

```
01 struct appdata
02 {
03    float4 position : POSITION;
04    float3 normal : NORMAL;
05    float3 color : DIFFUSE;
06    float3 TestColor : SPECULAR;
07 };
08
09 struct vs_output
10 {
11    float4 oPosition : POSITION;
12    float4 oColor0 : COLOR0;
13 };
14
15 vs_output main(appdata IN,
16             uniform float4 Kd,
17             uniform float4x4 ModelViewProj)
18 {
19    vs_output OUT;
20
21    OUT.oPosition = mul(ModelViewProj, IN.position);
22
23    OUT.oColor0.xyz = Kd.xyz * IN.TestColor.xyz;
24    OUT.oColor0.w = 1.0;
25
26    return OUT;
27 }
```

Listing 5: Pixel-Shader in Cg

```
01 struct ps_output
02 {
03    float4 outColor : COLOR;
04 };
05
06 ps_output main ( uniform sampler2D textur,
07                  float2 uv : TEXCOORD0 ) {
08    ps_output OUT;
09    OUT.outColor = tex2D(textur, uv);
10    return OUT;
11 }
```