Grundlagen der Spieleprogrammierung

Teil I: 3D-Graphik

Kapitel 9: Engines, Cg und anderes

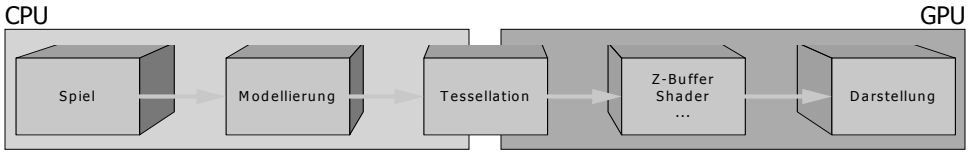
Peter Sturm
Universität Trier

Outline

1. Übersicht und Motivation
2. Mathematische Grundlagen
3. Das Ideal: Photorealistisch (Raytracing, Radiosity)
4. Die Realität: DirectX und OpenGL (Übersicht)
5. Schritt 1: Modellierung und Drahtgitter
6. Schritt 2: Texturen
7. Schritt 4: Licht, Filter, etc.
8. Schritt 5: Fortgeschrittene Techniken (Vertex-, Pixel-Shader, …)
9. 3D-Hardware
10. 3D-Engines im Überblick, Cg von nvidia
11. Spielekonsolen
12. Zusammenfassung und Ausblick

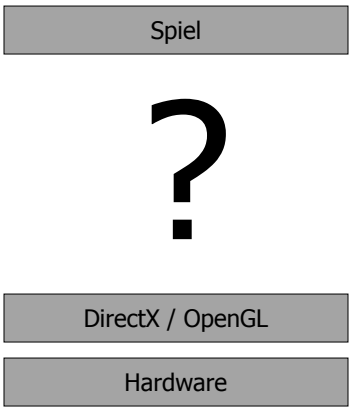
Motivation

- Schlüsselkonzept: Rendering Pipeline
  - Leistungsverteilung auf Hardware und Software
  - CPU (Anwendung) übernimmt Modellierung etc.
  - GPU (Hardware) übernimmt Darstellung und Finish
  - Schnittstelle CPU-GPU: Dreiecke
- DirectX und OpenGL kapseln Hardware-Anteil
  - Fortschreitende Standardisierung
  - Kompensation fehlender HW-Unterstützung



Software-Architekturen

- Unterstützung oberhalb DirectX und OpenGL
  - Keine
  - Sprachbasierte Ansätze
  - Engines
- Spezielle Sprachen
  - Cg von nvidia
- Engines
  - Viele Implementierungen
  - Sprachansatz integrierbar
  - Komplexe Architekturen

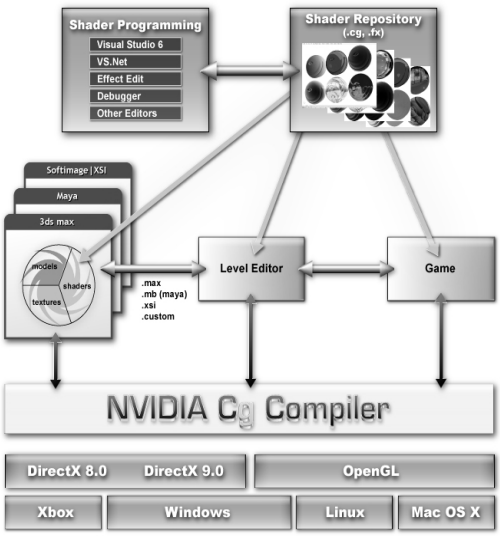


C for graphics: Cg

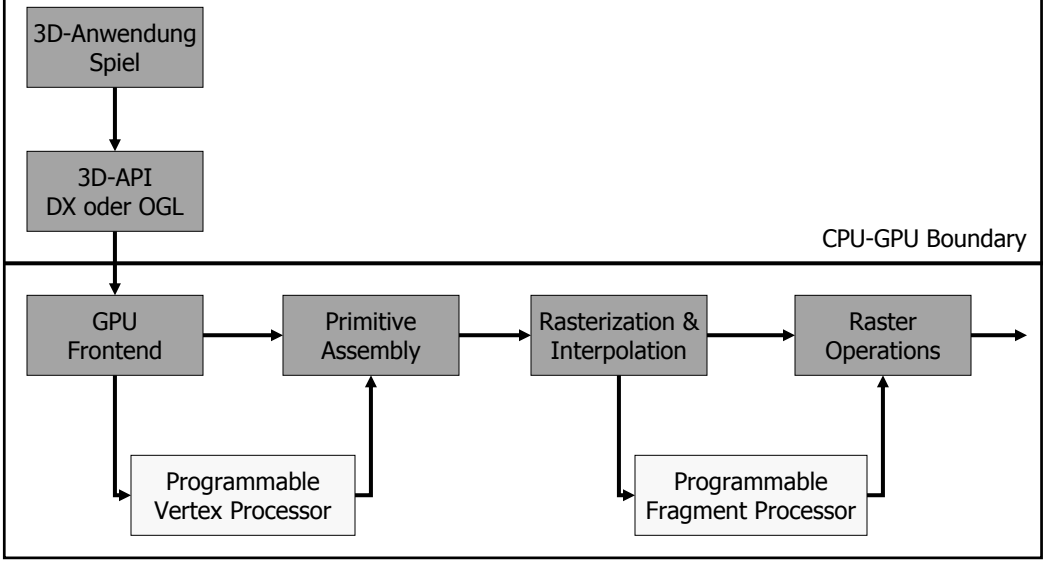
- Open Source
- High-Level Shading Language
  - Ersetzt Assembler für Vertex- und Pixel-Shader
- entstand in Zusammenarbeit mit Microsoft
  - unterstützt DirectX 9 HLSL
- Vorteile
  - Cross-API (DirectX und OpenGL)
  - Cross-Plattform (Windows, Linux, Mac OS)

Funktionsweise

- Entwicklung der gewünschten Shader
- Digital Content Creation (DCC)
  - z.B. mittels Maya, 3ds max etc.
  - Spezielle Editoren
  - Spieleprogrammierung
- Übersetzung durch Cg-Compiler
  - Zum Übersetzungs-zeitpunkt der Anwendung
  - Zur Laufzeit der Anwendung



Programmiermodell



Cg Language Profiles

- Gezielte Einschränkung von Cg auf spezifische Hardware- und Plattformeigenschaften
- Unterstützte Profile
  - DirectX 9 Vertex- und Pixel-Shader (Versionen 2.x)
  - OpenGL ARB Vertex- und Fragment-Programs
  - OpenGL NV30 Vertex- und Fragment-Programs
  - DirextX 8
  - OpenGL NV2X

Cg Programme

- Ersetzen die "Assemblerroutinen" in DirectX
- Operieren auf in den Shadern definierten Ein- und Ausgaben
  - Bindung an Programmvariablen geschieht über spezielle Syntax
- Innerhalb des Codes mehr oder weniger C

Programmeingaben

- Variierende Eingaben
  - Unterschiedliche Daten für jeden Aufruf des Vertex- oder Fragment-Programms
- Uniforme Eingaben
  - Konstanten, Zusatzinformationen
  - Gleichbleibend für jeden Vertex- oder Fragment-Aufrufs innerhalb eines Streams

Variierende Eingaben für Vertex-Programm

- Vordefiniert (in Anlehnung an Fixed-Pipeline)
  - Position im 3D-Raum (Vertex)
  - Normalenvektor an diesem Punkt
  - Texturkoordinaten
- Zusätzliche Werte pro Vertex definierbar

```
struct myinputs {
  float3 myPosition       : POSITION;
  float3 myNormal         : NORMAL;
  float3 myTangent        : TANGENT;
  float  refractive_index : TEXCOORD3;
};

outdata foo(myinputs indata) {
  /* ... */
  // Within the program, the parameters are referred to as
  // "indata.myPosition", "indata.myNormal", and so on.
  /* ... */
}
```

Variierende Ausgaben für Vertex-Programm

- … werden variierende Eingaben für Fragment-Programm
  - Aus- und Eingabebeziehung wird ebenfalls über die Bindings hergestellt
- In allen Profilen definierte Bindings
  - POSITION
  - PSIZE
  - FOG
  - COLOR0, COLOR1
  - TEXCOORD0 – TEXCOOR7
- POSITION muß immer ausgegeben werden

Variierende Ausgaben für Fragmente

- Zwingend
  - COLOR: Bestimmt die endgültige Farbe
  - DEPTH: Modifikation des Tiefenwerts

Cg: Datentypen

- Grundtypen
  - float (32 Bit), half (16 Bit), int, fixed (12 Bit), bool
  - Handles auf ein Texturobjekt
  - Vektoren (1 bis 4 Elemente): z.B. float1 bis float4
  - Matrixen (maximal 4x4): float2x2, float3x2, float4x4
- Structures
- Arrays
  - Assignments kopieren ganzes Feld

Statements und Operatoren

- Kontrollfluß
  - Unterprogramme, if/else, while, for
  - Keine Rekursion
  - Schleifen müssen in manchen Profilen aufrollbar sein (Anzahl der Durchläufe zum Übersetzungszeitpunkt bekannt)
- Unmenge an arithmetischen Operationen
  - Trigonometrische Funktionen
  - Runden, Abschneiden, etc.
  - Multiplikationen auch auf Matrizen, Vektoren, …
  - Kreuzprodukt, Determinante, …
  - Abbildung auf Texturen

Beispiel

Beispiel: lit()

`lit(ndotl, ndoth, m)`	Computes lighting coefficients for ambient, diffuse, and specular light contributions. Returns a 4-vector as follows: • The ***x*** component of the result vector contains the ambient coefficient, which is always 1.0. • The ***y*** component contains the diffuse coefficient which is zero if `(n • l)` < 0; otherwise `(n • l)`. • The ***z*** component contains the specular coefficient which is zero if either `(n • l)` < 0 or `(n • h)` < 0; $(n \bullet h)^m$ otherwise. • The ***w*** component is 1.0. There is no vectorized version of this function

Beispiel: Geometrische Funktionen

`distance(pt1, pt2)`	Euclidean distance between points ***pt1*** and ***pt2***
`faceforward(N, I, Ng)`	***N*** if `dot(Ng,I)` < 0; otherwise, ***-N***.
`length(v)`	Euclidean length of a vector
`normalize(v)`	Returns a vector of length 1 that points in the same direction as vector ***v***.
`reflect(i, n)`	Computes reflection vector from entering ray direction ***i*** and surface normal ***n***. Only valid for 3-component vectors.
`refract(i, n, eta)`	Given entering ray direction ***i***, surface normal ***n***, and relative index of refraction ***eta***, computes refraction vector. If the angle between ***i*** and ***n*** is too large for a given ***eta***, returns (0,0,0). Only valid for 3-component vectors.

3D-Engines

- Komfortabel nutzbare Funktions- bzw. Klassenbibliothek
  - Hohe Abstraktionen
  - Hohe Performanz
- Grundfunktionalität vergleichbar
- Unterschiede bei Features und Leistung
  - Polygonanzahl
  - Runde bzw. organische Formen
  - Transparenzeffekte
  - Realismus bei Wasser, Feuer, Rauch, etc.
- Lizenzkosten bei manchen Engines 10^5 Dollar und mehr

Zusätzliche Eigenschaften

- Partikelsysteme
  - Schöne Explosionen
  - Wird auch von DirectX angesprochen
  - Beispiel: DirectX-Demo Point Sprites
- Physik, Kinetik
  - Schöne Beulen und Verwundungen der Androidenhaut
  - Modellierung physikalischer und physischer Eigenschaften
  - Skin- und Bone-Modelle
  - Realismus
- ...

Zugängliche Engines

- "3D Engine DirectX" liefert 106000 Treffer
- WWW-Seite listet ca. 650 Engines (2001)
- Schwieriger Zugang
  - Viele Engines sind recht komplex
  - Meist wird C++ API von DirectX verwendet (unschön)
- Interessante Beispiele
  - Quake 2 Engine unter GPL
  - Andere?
  - Noch keine für C#? Wer hat Lust ☺

Grundlagen der Spieleprogrammierung

Teil I: 3D-Graphik

Kapitel 10: Spielekonsolen

Peter Sturm
Universität Trier

Outline

1. Übersicht und Motivation
2. Mathematische Grundlagen
3. Das Ideal: Photorealistisch (Raytracing, Radiosity)
4. Die Realität: DirectX und OpenGL (Übersicht)
5. Schritt 1: Modellierung und Drahtgitter
6. Schritt 2: Texturen
7. Schritt 4: Licht, Filter, etc.
8. Schritt 5: Fortgeschrittene Techniken (Vertex-, Pixel-Shader, …)
9. 3D-Hardware
10. 3D-Engines im Überblick, Cg von nvidia
11. Spielekonsolen
12. Zusammenfassung und Ausblick

Motivation

- Eine gewisse Firma aus dem Westen der USA hat mich im Stich gelassen ☹
  - Xbox im Detail wäre bestimmt interessant gewesen
  - Programmiermodell aus unerfindlichen Gründen DirectX sehr ähnlich
  - Unterstützung durch Cg
- Interessant sind generell die Leistungsdaten der Spiele-Hardware

Grundlagen der Spieleprogrammierung

Teil I: 3D-Graphik

Kapitel 11: Zusammenfassung und Ausblick

Peter Sturm
Universität Trier

Outline

1. Übersicht und Motivation
2. Mathematische Grundlagen
3. Das Ideal: Photorealistisch (Raytracing, Radiosity)
4. Die Realität: DirectX und OpenGL (Übersicht)
5. Schritt 1: Modellierung und Drahtgitter
6. Schritt 2: Texturen
7. Schritt 4: Licht, Filter, etc.
8. Schritt 5: Fortgeschrittene Techniken (Vertex-, Pixel-Shader, …)
9. 3D-Hardware
10. 3D-Engines im Überblick, Cg von nvidia
11. Spielekonsolen
12. Zusammenfassung und Ausblick

Rückblick

- Photorealistische Darstellung immer noch um mehrere Größenordnungen zu aufwendig
- Arbeitsteilung zwischen CPU und GPU essentiell
  - Rendering Pipeline
  - Dreiecke sind die zentrale Abstraktion an der Schnittstelle
- Höhersetzen der Grenze innerhalb der Pipeline wird zunehmend schwieriger
  - Vertex- und Pixel-Shader
- "Unendlich" viele Details konnten nicht behandelt werden
  - Zeitproblem
  - Unfähigkeit (daran werde ich noch arbeiten ☺)

Ausblick

- Was wird sich in den nächsten 5 Jahren ändern?