Seminar
Analyse, Planung und Konstruktion
computergraphischer Systeme

Die  Shading Language

Stephan Brumme

7. Januar 2004

Lehrstuhl für
Computergraphische Systeme

Agenda

1. Motivation
2. Das Shader-Konzept
3. Aufbau der OpenGL Shading Language
4. Praxis
5. Schlussfolgerungen

Motivation - I

- Bedarf nach frei programmierbarer Graphik-Pipeline
  - Shader
  - Wahrung von
    - Abwärtskompatibilität
    - Portabilität
- Adäquate Performanz
  - i.d.R. Echtzeit
  - optimale Nutzung vorhandener Ressourcen
- Hardware-unterstützt

© Square, © NVIDIA

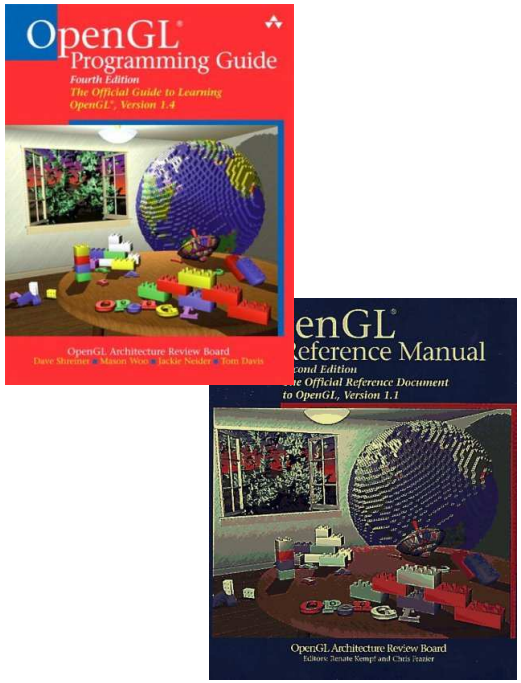
Motivation - II

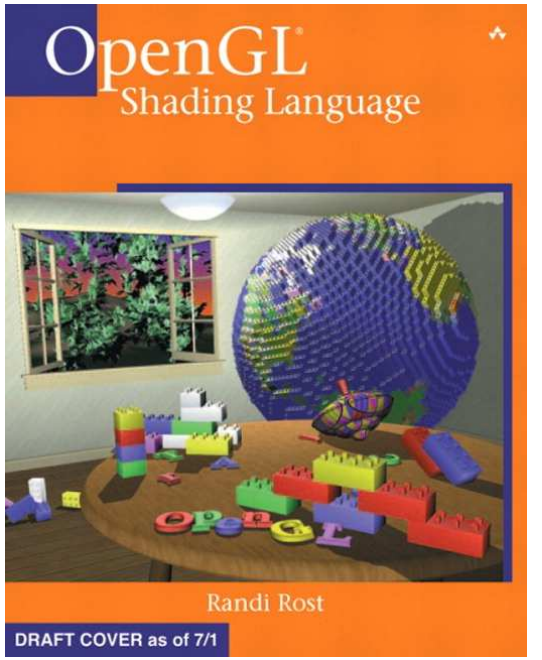
- Möglichkeiten:
  - prozedurale Texturen
  - effizientere Transformationen
  - NPR Effekte
  - Bildoperationen
  - interaktive Visualisierungstechniken
  - …

  → viele OpenGL-Extensions dann überflüssig

Motivation - III

- Grundlegende Änderungen an OpenGL
  - OpenGL 2.0
  - OpenGL Shading Language



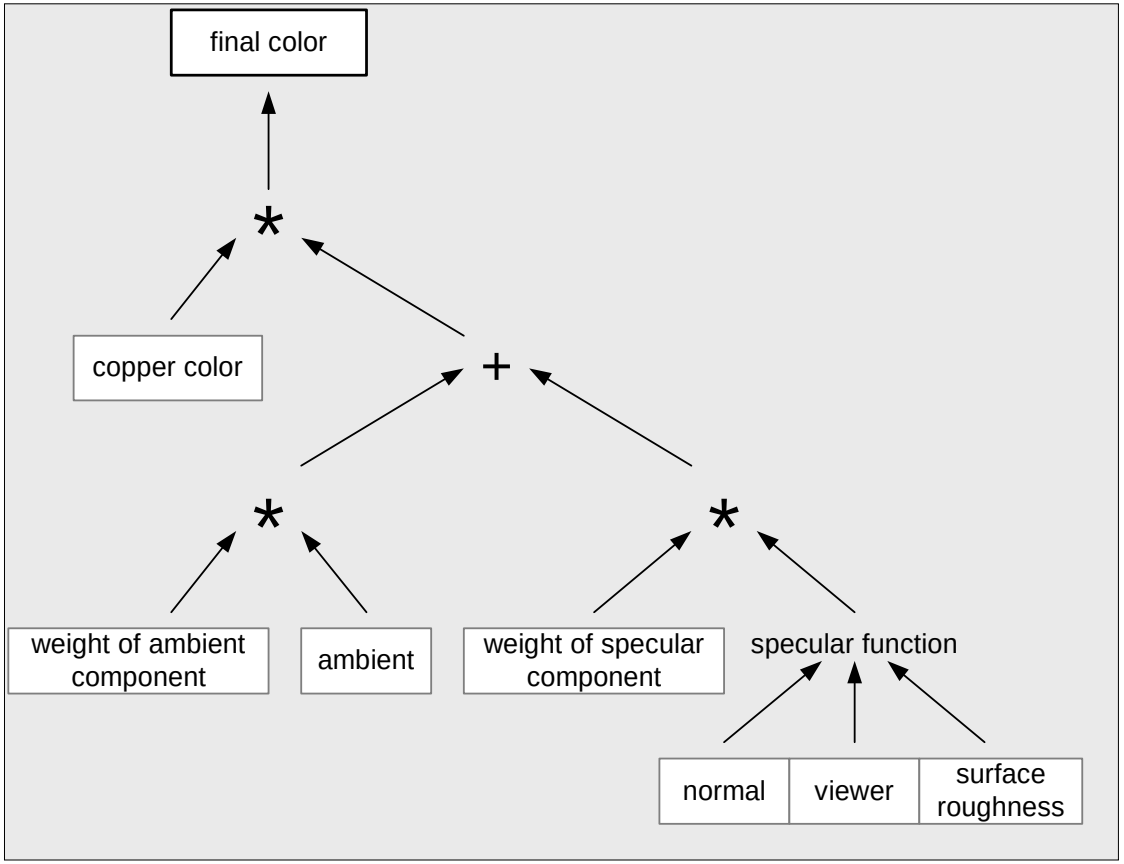


erscheint am 13. Februar 2004

Bilder: Addison-Wesley

Das Shader-Konzept - I

- Shade Trees (Cook, 1984)
  - hierarchische Verkettung von Mikroprogrammen → Shadern




```
float ka=0.5, ks=0.5;
float roughness=0.1;
float intensity;
color copper=(0.8,0.3,0.1);

intensity=ka*ambient()+
ks*specular(normal,viewer,
            roughness);
final_color=intensity*
            copper;
```

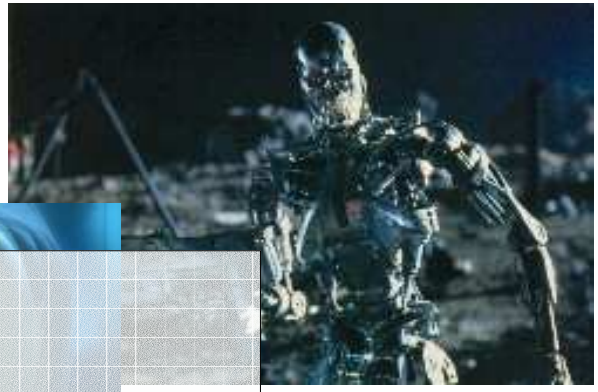
nach: Cook, *Shade Trees*, 1984

Das Shader-Konzept - II

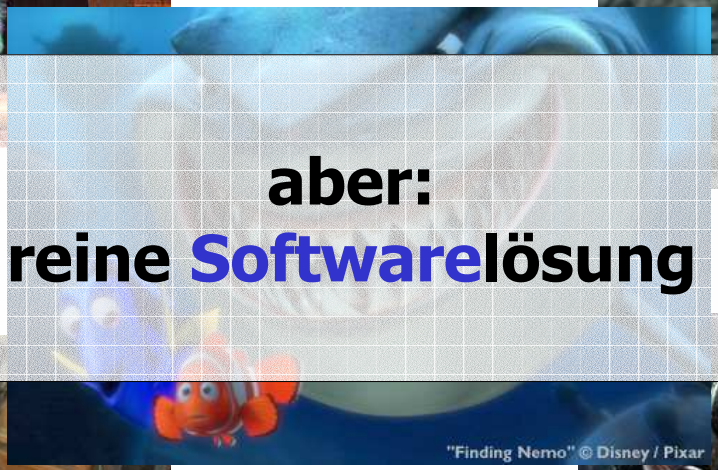
- Shader
  - Mikroprogramme
  - manipulieren zu einem graphischen Basiselement gehörende Daten
    - Basiselemente:
      - Vektor (V), Fragment (F)
    - zugehörige Daten:
      - Position (V),
      - Normale (V),
      - Texturkoordinaten (V/F),
      - Farbe (V/F),
      - benutzerdefinierte Daten (V/F)
      - …

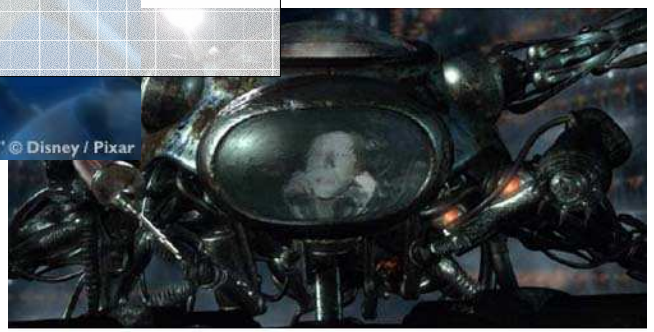
Das Shader-Konzept - III

- Große Verbreitung durch Renderman™ Standard
  - entwickelt von Cook et al. bei Pixar (Reyes-Architektur 1987)



aber:
reine Softwarelösung





Das Shader-Konzept - IV

- Hardwareunterstützung
  - Entwicklung neuer Hardware
    - PixelFlow (Olano, 1995)

    → hohe Kosten, unflexibel

  - Verwendung von reinem Standard-OpenGL
    - Multipass-Rendering von SGI (Peercy, 2000)

    → preiswert, aber Bandbreiten-Problem

    - Quake III Engine

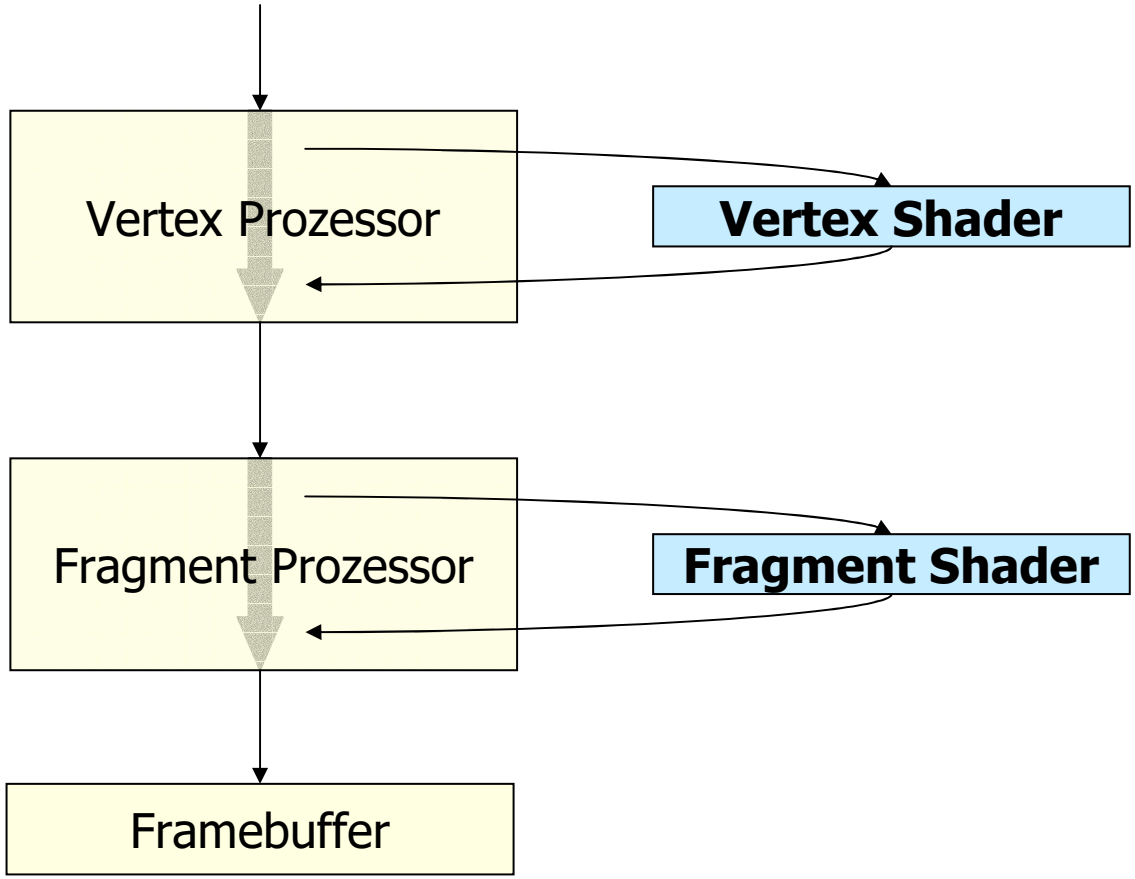
    → preiswert und schnell, kommerziell sehr erfolgreich

    → sehr begrenzte Fähigkeiten, nur Fragment-Shader

Bilder von: Olano, Peercy, id software

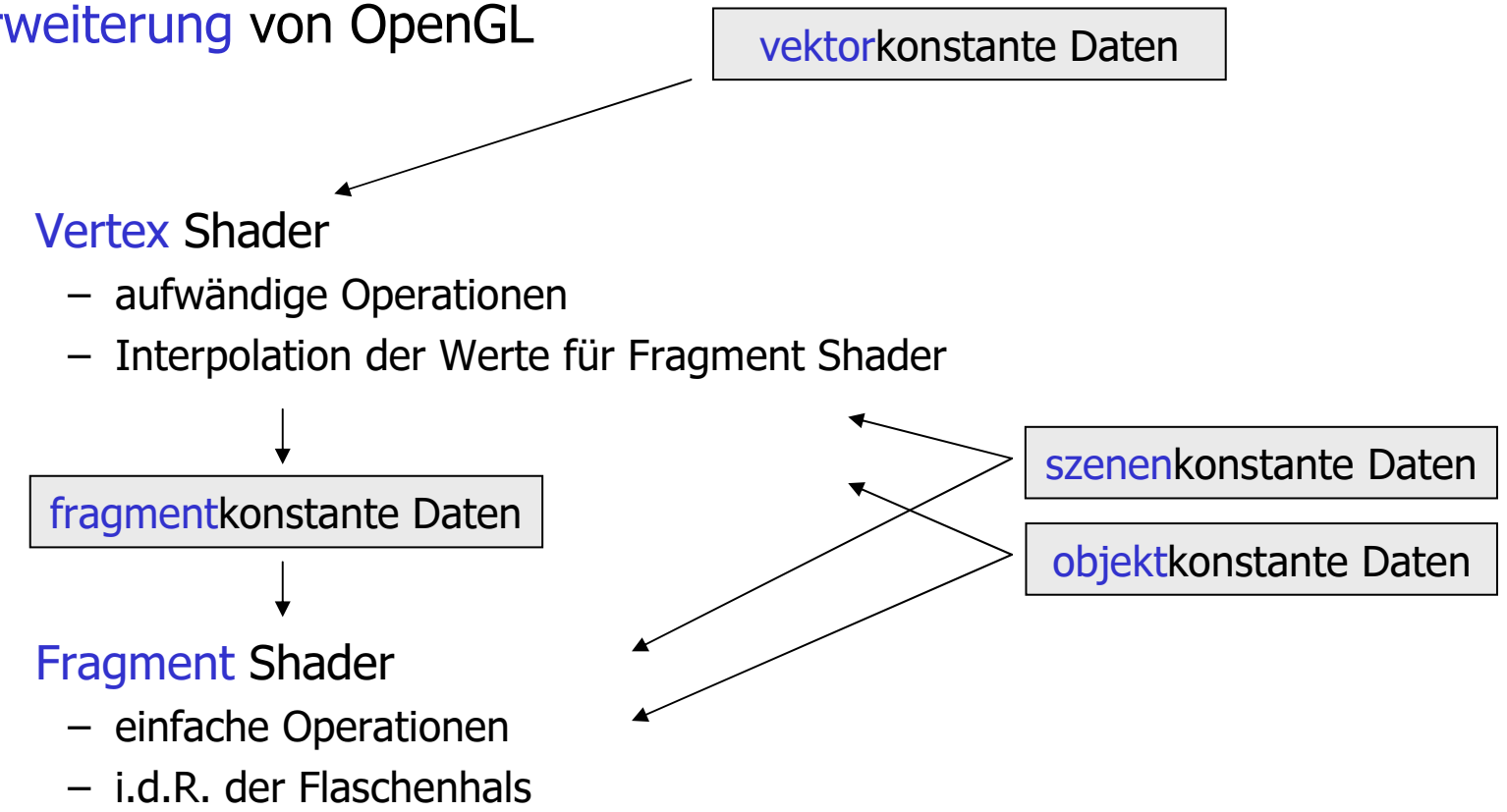
Das Shader-Konzept - V

- Partielle Ersetzung der fixed functionality pipeline



Das Shader-Konzept - VI

- Hardwareunterstützung (Fortsetzung)
  - Erweiterung von OpenGL



Aufbau von GLSL - I

- Hardware-Shader-Programmierung aufwändig
  - bislang nur Assembler-Sprache
  - → Entwicklung einer Hochsprache
  - → Die OpenGL Shading Language (GLSL bzw. glslang)

- Alternativen:
  - Cg von NVIDIA
    - „C for Graphics"
  - HLSL von Microsoft (DirectX 9+)
    - „High Level Shading Language"

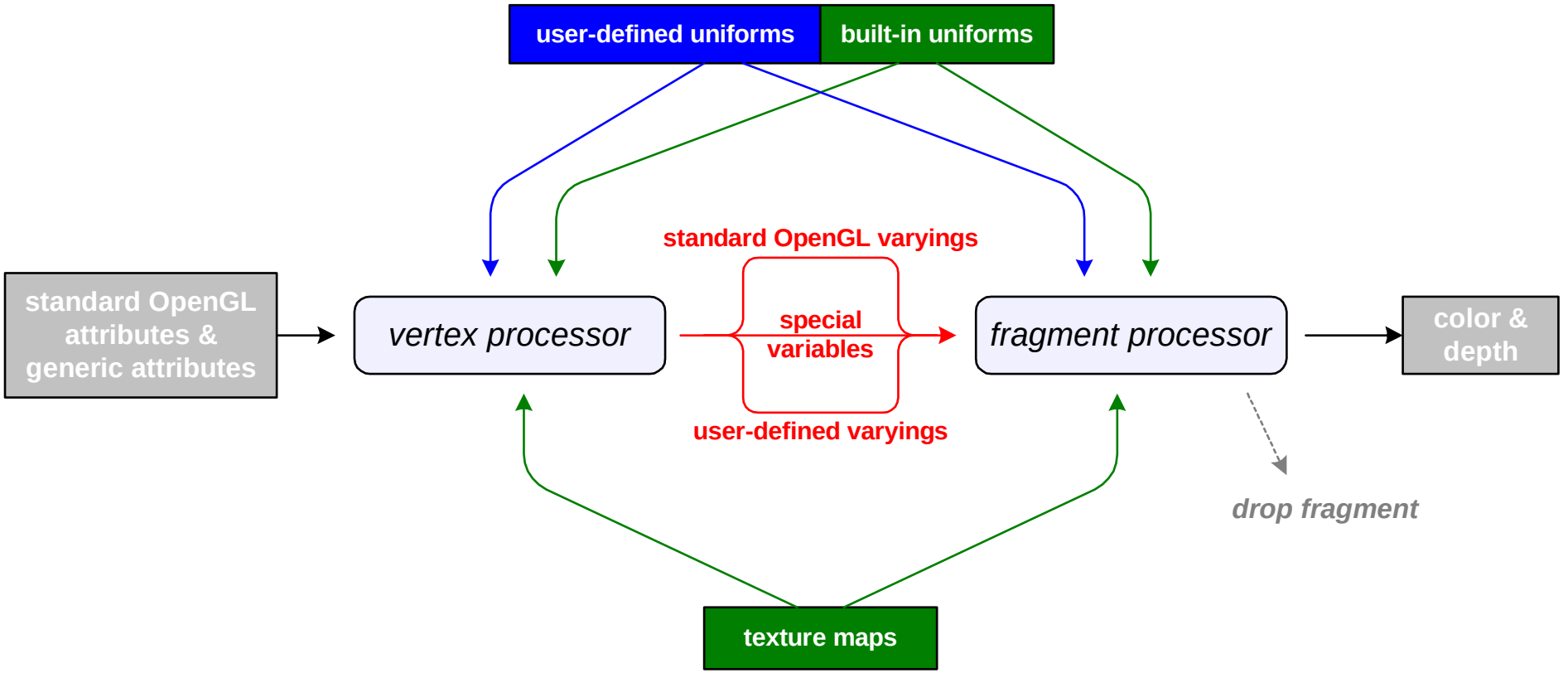
Aufbau von GLSL - II

- Datenfluss:
  - vektorkonstant: Attribute
    - vom Shader nicht veränderbar
    - Standard-OpenGL
      - gl_Color, gl_Normal, …
    - benutzerdefiniert
      - glossiness, refraction, …
  - szenenkonstant und objektkonstant: ***uniform***
    - vom Shader nicht veränderbar
    - Standard-OpenGL
      - gl_ModelViewMatrix, gl_LightSource0, …
    - benutzerdefiniert
      - time, boneWeighting, …

Aufbau von GLSL - III

- Datenfluss (Fortsetzung):
  - Kommunikation Vertex Shader → Fragment Shader: ***varying***
    - vom Shader generiert
    - von Hardware auf Fragmentebene perspektivisch interpoliert
    - Standard-OpenGL
      - gl_FrontColor, gl_TexCoord0, …
    - Spezialwerte
      - gl_Position, gl_PointSize, …
    - benutzerdefiniert
      - thickness, density, …
  - fragmentkonstant:
    - gl_FragColor
    - gl_FragDepth

Aufbau von GLSL - IV



Aufbau von GLSL - V

- Syntax der GLSL
  - stark angelehnt an ANSI C
  - Datentypen:
    - float, int, bool
    - vec{2,3,4}, ivec{2,3,4}, bvec{2,3,4}
    - mat{2,3,4}
    - sampler{1,2,3}D
    - void
  - Funktionen
    - main als Einsprungspunkt
    - Parameter
      - in, out, inout, const
      - Übergabe by-value
    - Eingebaute Funktionen

```
vec4 diffuse(const in vec4 N,
             const in vec4 L,
             in vec4 C)
{
    // compute diffuse lighting
    float a = dot(N,normalize(L));
    if (a > 0)
        C = C*a;
    else
        C = 0;
    return C;
}
```

Aufbau von GLSL - VI

- Syntax der GLSL (Fortsetzung)
  - Strukturierungsmöglichkeiten
    - Sequenz
    - Selektion (if-else)
    - Schleife (for, white, do-while)
    - Sprünge (return, break, continue, discard)
  - Preprozessor
    - #ifdef
    - #pragma
    - Kommentare im C++ Stil

```
vec4 diffuse(const in vec4 N,
             const in vec4 L,
             in vec4 C)
{
    // compute diffuse lighting
    float a = dot(N,normalize(L));
    if (a > 0)
        C = C*a;
    else
        C = 0;
    return C;
}
```

Praxis - I

- Beispiel:

Vertex-Shader

```
// vertex to fragment shader io
varying vec3 N;
varying vec4 I;



// entry point
void main()
{
  // position in eye space
  vec4 P =  gl_ModelViewMatrix * gl_Vertex;
  // position in clip space
  gl_Position = gl_ModelViewProjectionMatrix*
                gl_Vertex;
  // normal transform
  N = gl_NormalMatrix * gl_Normal;
  // incident vector
  I = P - gl_ModelViewMatrix[3];
}
```

Fragment-Shader

```
// vertex to fragment shader io
varying vec3 N;
varying vec4 I;

// globals
uniform float edgefalloff;

// entry point
void main()
{
  float opac = dot(normalize(N),
                   normalize(-I));
  opac = 1 – pow(opac, edgefalloff);

  gl_FragColor =  gl_Color;
  gl_FragColor[3] = opac;
}
```

Praxis - II

- ARB-Erweiterung der OpenGL API
  - Shader-Objekt
    - Quelltext(e)
    - jederzeit kompilierbar

    → bei Programmstart oder unmittelbar vor Benutzung

```
GLhandleARB vs, fs;

// create vertex shader object
vs = glCreateShaderObjectARB
       (GL_VERTEX_SHADER_ARB);
glShaderSourceARB(vs,1,&vsSource,NULL);
glCompileShaderARB(vs);

// create fragment shader object
fs = glCreateShaderObjectARB
       (GL_FRAGMENT_SHADER_ARB);
glShaderSourceARB(fs,1,&fsSource,NULL);
glCompileShaderARB(fs);
```

  - Programm-Objekt
    - Ansammlung von zusammen-gehörigen Shader-Objekten
      - erfordert Linking
    - enhält genau eine main-Funktion

```
GLhandleARB program;

// create program object
program = glCreateProgramObjectARB();

// attach shader objects
glAttachObjectARB(program, vs);
glAttachObjectARB(program, fs);

// link program object and enable it
glLinkProgramARB(program);
glUseProgramObjectARB(program);
```

Praxis - III

- ARB-Erweiterung der OpenGL API (Fortsetzung)
  - uniform
    1. Handle liefern lassen
    2. mittels Handle Wert setzen

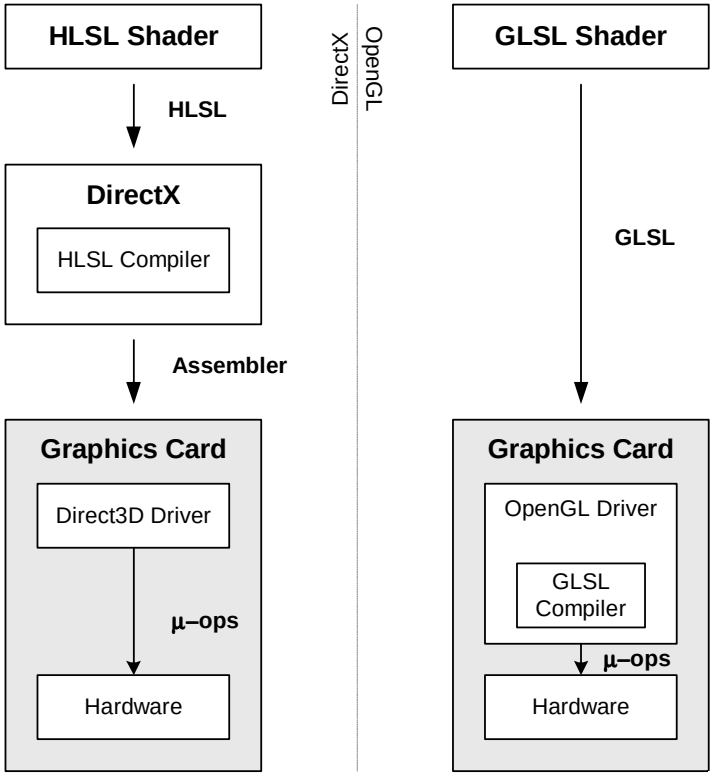
```
Glint hParamFloat, hParamVector;
// get locations of parameters
hParamFloat =
 glGetUniformLocationARB(program, “f”);
hParamVector =
 glGetUniformLocationARB(program, “v”);

// set parameters
glUniform1fARB(hParamFloat, 0.42);
glUniform3fARB(hParamVector,
               0.5, 1.0, 0.0);
```

  - Übermittlung der Attribute auf gewohnte Art und Weise
    - glVertex, glNormal, glColor, …

Schlussfolgerungen - I

- Ähnliche Konzepte von Cg und HLSL
  - Vertex-/Fragmentshader
  - ähnliche Syntax
  - aber Kompilation auf anderer Ebene
    - zusätzliche Schicht in Cg/HLSL
    - daher einfacher zu implementieren
    - jedoch weniger Optimierungs-möglichkeiten durch Treiber

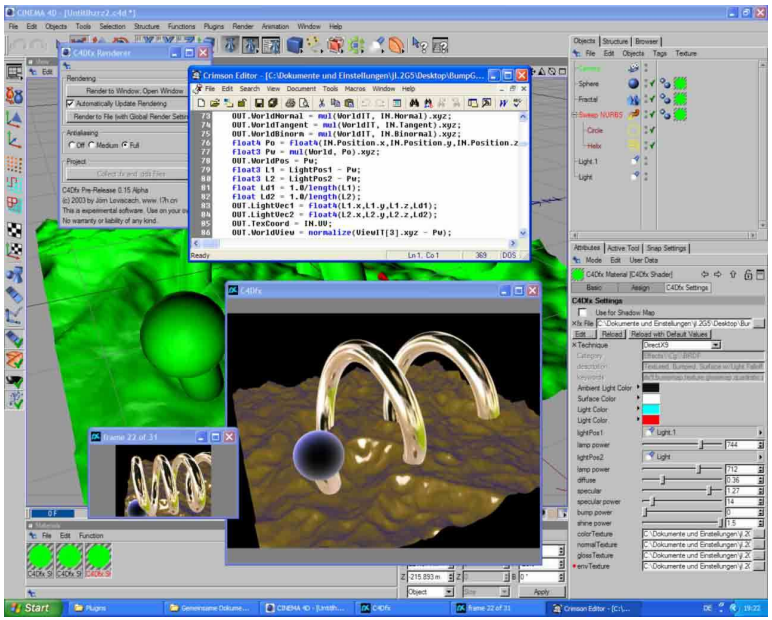


Schlussfolgerungen - II

- Praxisrelevanz GLSL
  - Hardware grundsätzlich in der Lage, GLSL auszuführen
  - → Treiber nicht verfügbar oder nicht ausgereift

- Cg und vor allem HLSL bereits länger im produktiven Einsatz
  - große Menge verfügbarer Shader
  - PlugIns für Vielzahl von HighEnd-Grafikprogrammen

- Massenmarkt (Spiele) kaum durchdrungen



Cg-PlugIn für Cinema4D, © Maxon

Schlussfolgerungen - III

- Zukunft
  - bessere Treiberunterstützung
  - Performance muss stark gesteigert werden
    - schnellere Hardware und bessere Optimierungen
    - erst dann massenmarkttauglich
  - Sprache z.Z. nicht komplett in Hardware abgebildet
    - z.B. noise auf ATI-Karten
  - wichtige Sprachfeatures fehlen
    - switch-Anweisung
  - mangelhafte Unterstützung der Shader-Entwickler
    - vernünftiger Debugger erforderlich

Anhang

- Hauptquellen
  - John Kessenich, Dave Baldwin and Randi Rost, *The OpenGL Shading Language*, version 1.05, 2003.
  - Robert L. Cook, *Shade Trees*, SIGGRAPH 1984, Minneapolis.
  - Robert L. Cook, Loren Carpenter and Edwin Catmull, *The Reyes Image Rendering Architecture*, SIGGRAPH 1987, Anaheim.
  - Tomas Akenine-Möller and Eric Haines, *Real-time Rendering*, 2nd edi-tion, A. K. Peters, 2002.
  - Microsoft Corporation, *High-level shader language*.
  - William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard, *Cg: A system for programming graphics hardware in a C-like language*, SIGGRAPH 2003, San Diego.

- weitere siehe Ausarbeitung

Diskussion