



Computer-Graphik II

Shader

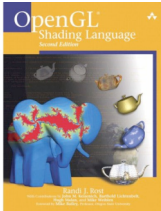


G. Zachmann
 Clausthal University, Germany
cg.in.tu-clausthal.de



Literatur

- Das "Orange Book":
 - Randi J. Rost, et al.:
 "OpenGL Shading Language",
 2nd edition, Addison Wesley.
- Auf der Homepage der Vorlesung:
 - Das Tutorial von Lighthouse3D
 - Mark Olano's "*Brief OpenGL Shading Tutorial*"
 - Der "GLSL Quick Reference Guide"
 - ...

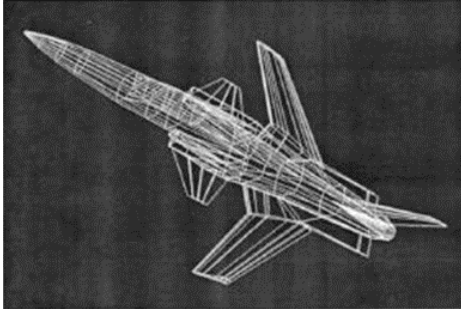


G. Zachmann Computer-Graphik 2 - SS 07

Shader 2

The Quest for Realism


- Erste Generation – Wireframe
 - Vertex-Oper.: Transformation, Clipping und Projektion
 - Rasterization: Color Interpolation (Punkte, Linien)
 - Fragment-Op.: Overwrite
 - Zeitraum: bis 1987



G. Zachmann Computer-Graphik 2 - SS 07 Shader 3

The Quest for Realism

- Zweite Generation – Shaded Solids
 - Vertex-Oper.: Beleuchtungsrechnung & Gouraud-Shading
 - Rasterization: Depth-Interpolation
 - Fragment-Oper.: Depth-Buffer, Color Blending
 - Zeitraum: 1987 - 1992




(Dogfight - SGI)

G. Zachmann Computer-Graphik 2 - SS 07 Shader 4

■ Dritte Generation – Texture Mapping

- Vertex-Oper.: Textur-Koordinaten-Transformation
- Rasterization: Textur-Koordinaten-Interpolation
- Fragment-Oper.: Textur-Auswertung, Antialiasing
- Zeitraum: 1992 - 2000

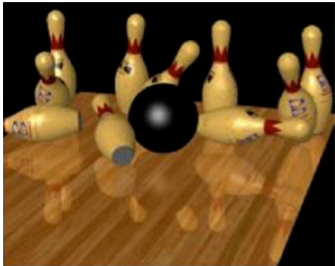




Performertown (SGI)

G. Zachmann Computer-Graphik 2 - SS 07
Shader 5

■ Vierte Generation – Programmierbarkeit

- Vertex-Oper.: eigenes Programm
- Rasterization: Interpolation der (beliebigen) Ausgaben des Vertex-Programms
- Fragment: eigenes Programm
- Zeitraum-Oper.: ab 2000

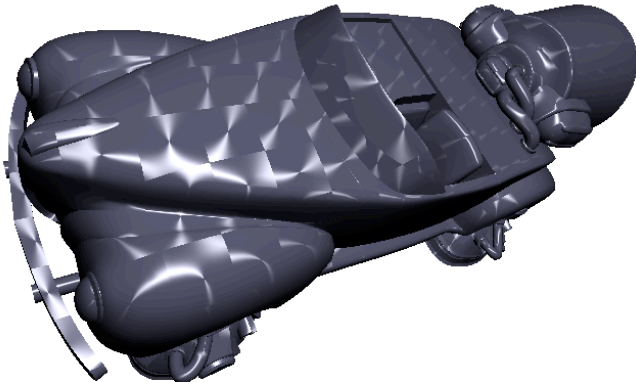



Final Fantasy

G. Zachmann Computer-Graphik 2 - SS 07
Shader 6

Beispiele

- Brushed Steel
- Prozedurale Textur
- Anisotrope Beleuchtung




The image shows a 3D rendered motorcycle, likely a Honda CBR1000RR, with a brushed steel texture. The lighting is anisotropic, creating bright, directional highlights that emphasize the metallic surface's grain and the bike's curves. The background is a plain, light gray.

G. Zachmann Computer-Graphik 2 - SS 07 Shader 7

Schmelzendes Eis:

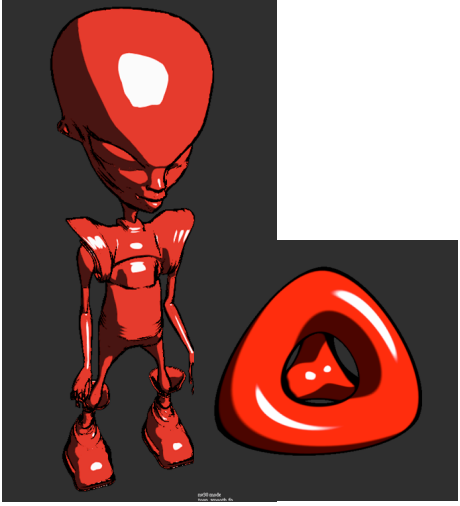
- Prozedurale, animierte Textur
- Bump-mapped environment map



The image shows a 3D rendered scene of a melting ice cube on a dark, reflective surface. The ice is rendered with a procedural, animated texture that shows the melting process. The scene is lit with a spotlight, and the environment map is bump-mapped, creating a realistic effect of light reflecting off the ice and the surface.

G. Zachmann Computer-Graphik 2 - SS 07 Shader 8


- Sog. „Toon Shading“
 - Ohne Texturen
 - Mit Anti-Aliasing
 - Gute Silhouetten ohne zu starker Verdunkelung



G. Zachmann Computer-Graphik 2 - SS 07 Shader 9


- Vegetation & *Thin Film*

Translucence
Backlighting



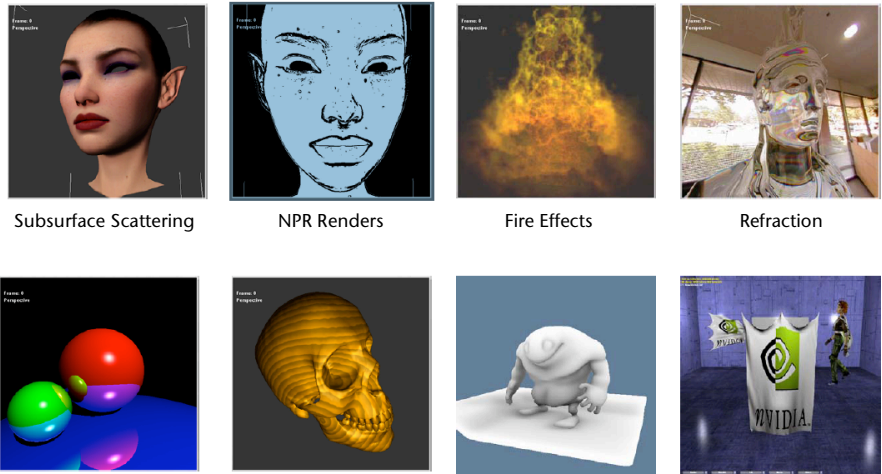

Beispiel von selbstgemachter
Beleuchtungsrechnung; hier:
Simulation von Schillern

G. Zachmann Computer-Graphik 2 - SS 07 Shader 10



<http://ati.amd.com/developer/demos/macs2/index.html>

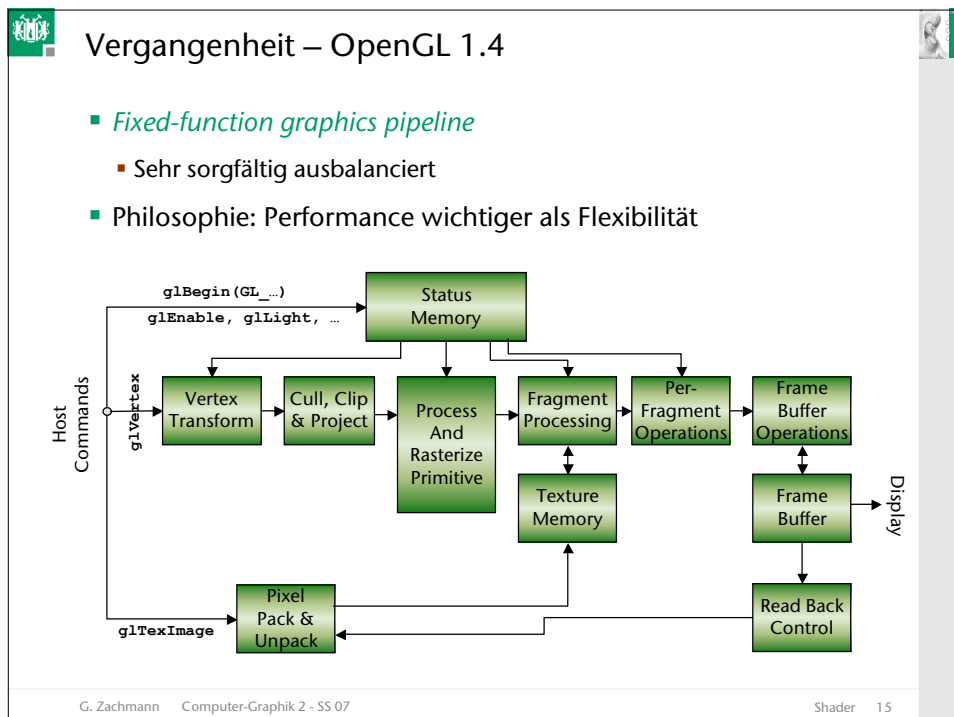
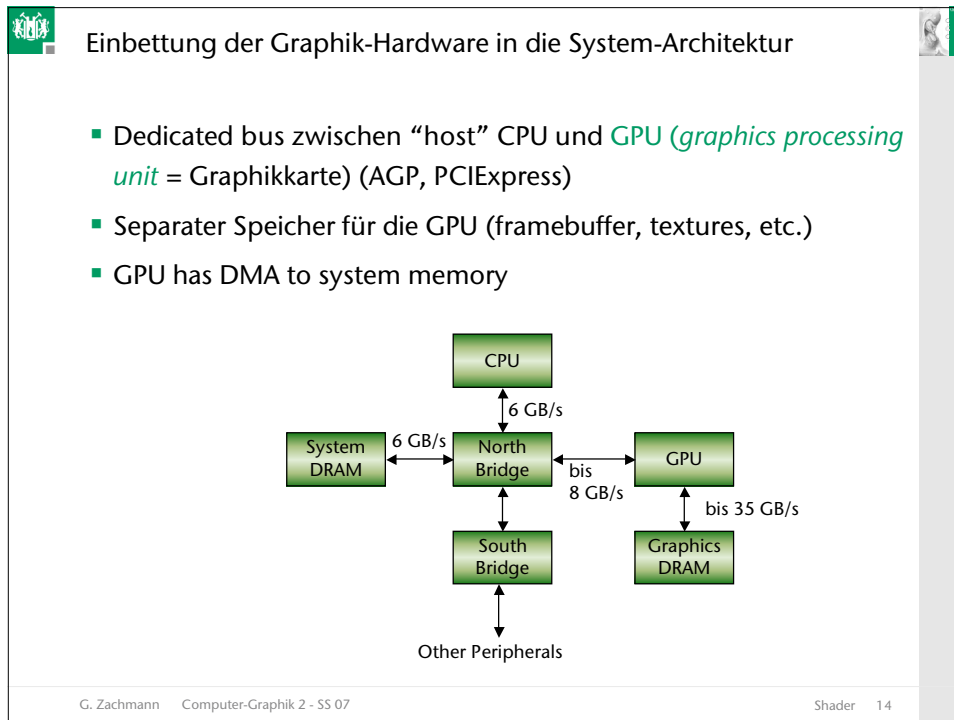
G. Zachmann Computer-Graphik 2 - SS 07 Shader 11



Subsurface Scattering NPR Renders Fire Effects Refraction

Ray Tracing Solid Textures Ambient Occlusion Cloth Simulation

G. Zachmann Computer-Graphik 2 - SS 07 Shader 12



Heute – OpenGL 2.1

- Programmierbare *vertex und fragment processors*
 - Legen offen, was sowieso schon immer da war
- Texturspeicher = allgemeiner Speicher für beliebige Daten

```

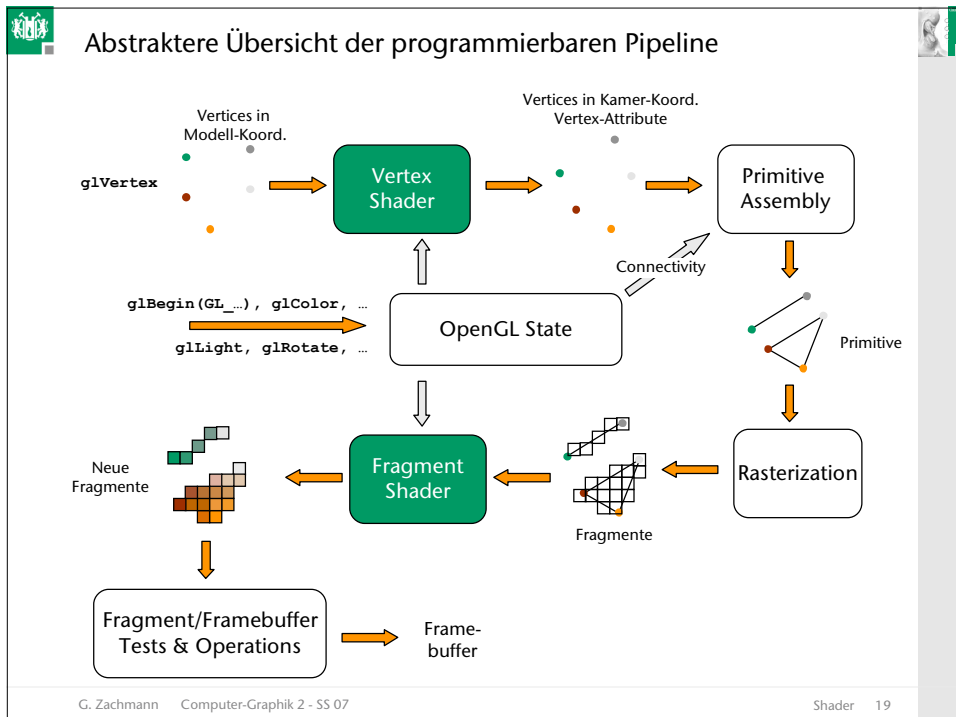
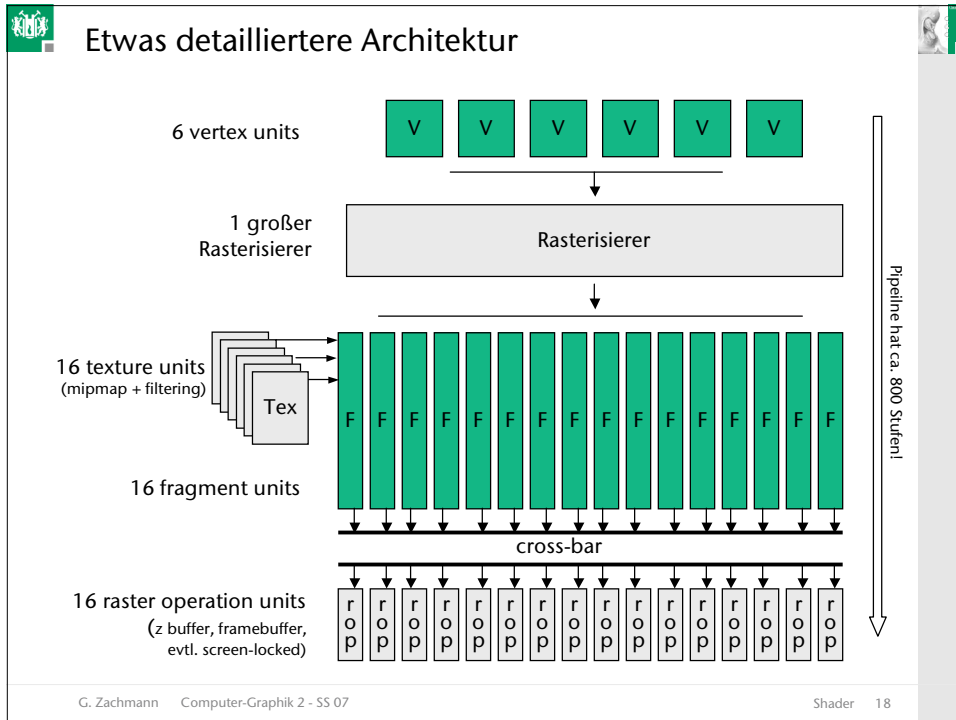
graph TD
    Host[Host Commands] -- glBegin(GL_...), glEnable, glLight, ... --> SM[Status Memory]
    Host -- glVertex --> VP[Vertex Processing]
    Host -- glTexImage --> PPU[Pixel Pack & Unpack]
    SM --> VP
    SM --> CCP[Cull, Clip & Project]
    SM --> PRP[Process And Rasterize Primitive]
    SM --> FP[Fragment Processing]
    SM --> PFO[Per-Fragment Operations]
    VP --> CCP
    CCP --> PRP
    PRP --> FP
    FP --> PFO
    PFO --> FBO[Frame Buffer Operations]
    FBO --> FB[Frame Buffer]
    FB --> RBC[Read Back Control]
    RBC --> PPU
    PPU --> TM[Texture Memory]
    TM --> VP
    TM --> FP
    FB --> RBC
    RBC --> Display[Display]
  
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 16

Bald – OpenGL 3.0

- Große Veränderungen ...

G. Zachmann Computer-Graphik 2 - SS 07 Shader 17



- Hilfsvorstellung:

```

...
foreach tri in triangles
{
    // run the vertex program on each vertex
    v1 = process_vertex( tri.vertex1 );
    v2 = process_vertex( tri.vertex2 );
    v3 = process_vertex( tri.vertex2 );

    // assemble the vertices into a triangle
    assembledtriangle = setup_tri(v1, v2, v3);

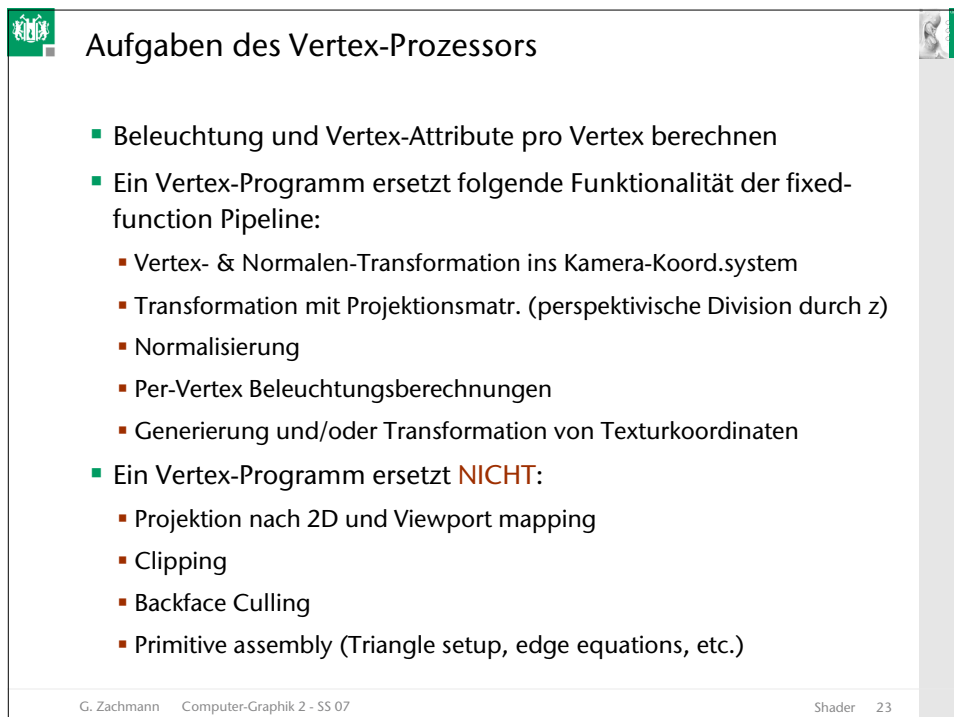
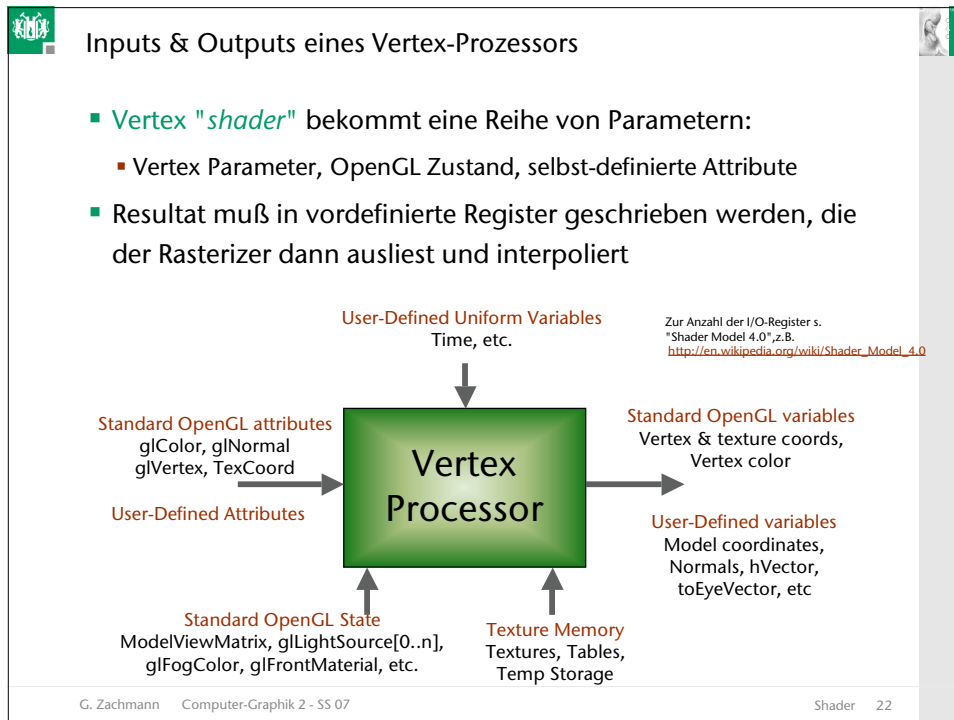
    // rasterize the assembled triangle into [0..many] fragments
    fragments = rasterize( assembledtriangle );

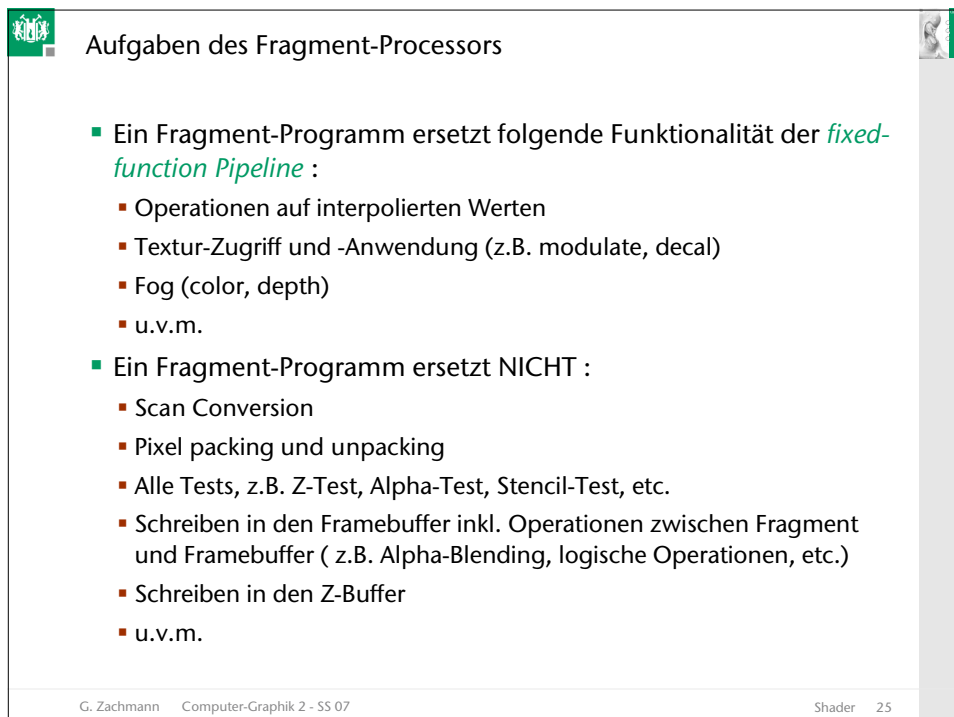
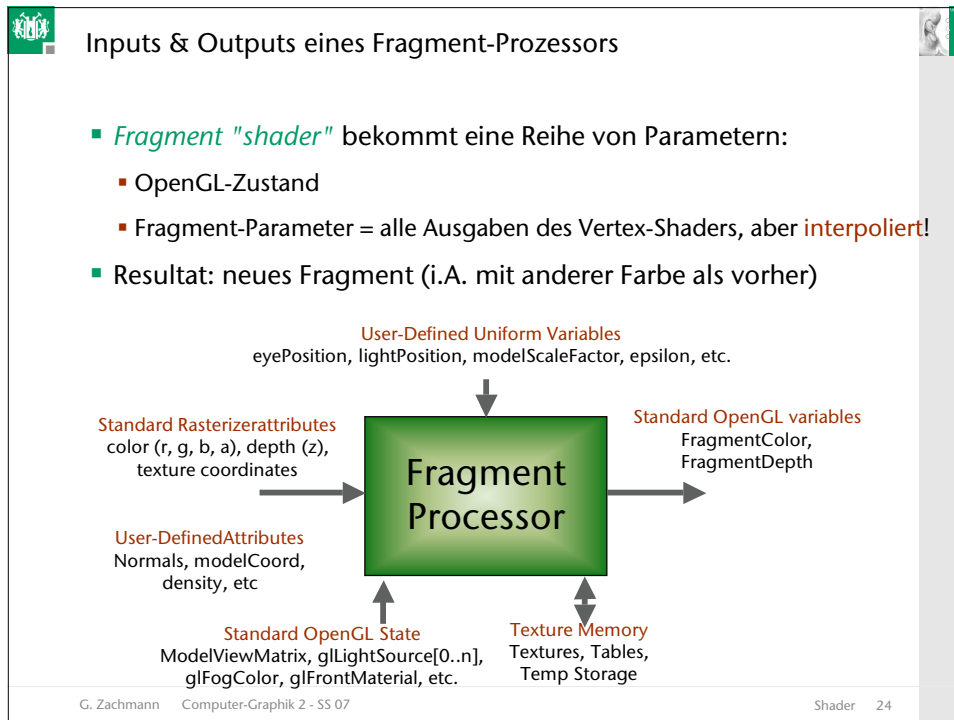
    // run the fragment program on each fragment
    foreach frag in fragments {
        framebuffer[frag.position] = process_fragment( frag );
    }
}
...

```

Fragment vs. Pixel

- Achtung: unterscheide zwischen Pixel und Fragment!
- **Pixel** :=
eine Anzahl Bytes im Framebuffer
bzw. ein Punkt auf dem Bildschirm
- **Fragment** :=
eine Menge von Daten (Farbe, Koordinaten, Alpha, ...), die zum Einfärben eines Pixels benötigt werden
- M.a.W.:
 - Ein Pixel befindet sich am Ende der Pipeline
 - Ein Fragment ist ein "Struct", das durch die Pipeline "wandert" und am Ende in ein Pixel gespeichert wird





Was ein Shader **nicht** kann

- Ein **Vertex-Shader** hat keinen Zugriff auf Connectivity-Info und Framebuffer
- Ein Fragment-Shader
 - hat keinen Zugriff auf danebenliegende Fragmente
 - hat keinen Zugriff auf den Framebuffer
 - kann nicht die Pixel-Koordinaten wechseln (aber kann auf sie zugreifen)

G. Zachmann Computer-Graphik 2 - SS 07 Shader 26

Wie sieht nun echter Shader-Code aus?

Assembly

```

RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxxx, R4.xyz;
MOVR R5.xyz, -R0.xyz;
MOVR R3.xyz, -R3.xyz;
DP3R R3.x, R0.xyz, R3.xyz;
SLTR R4.x, R3.x, {0.000000}.x;
ADDR R3.x, {1.000000}.x, -R4.x;
MULR R3.xyz, R3.xxxx, R5.xyz;
MULR R0.xyz, R0.xyz, R4.xxxx;
ADDR R0.xyz, R0.xyz, R3.xyz;
DP3R R1.x, R0.xyz, R1.xyz;
MAXR R1.x, {0.000000}.x, R1.x;
LG2R R1.x, R1.x;
MULR R1.x, {10.000000}.x, R1.x;
EX2R R1.x, R1.x;
MOVR R1.xyz, R1.xxxx;
MULR R1.xyz, {0.900000, 0.800000, 1.000000}.xyz, R1.xyz;
DP3R R0.x, R0.xyz, R2.xyz;
MAXR R0.x, {0.000000}.x, R0.x;
MOVR R0.xyz, R0.xxxx;
ADDR R0.xyz, {0.100000, 0.100000, 0.100000}.xyz, R0.xyz;
MULR R0.xyz, {1.000000, 0.800000, 0.800000}.xyz, R0.xyz;
ADDR R1.xyz, R0.xyz, R1.xyz;

```

Hochsprache

```

float spec = pow( max(0, dot(n,h)), phongExp);
color cResult = Cd * (cAmbi + cDiff) +
              Cs * spec * cSpec;

```

Einfacher Phong-Shader
ausgedrückt in
Assembly und GLSL

G. Zachmann Computer-Graphik 2 - SS 07 Shader 27

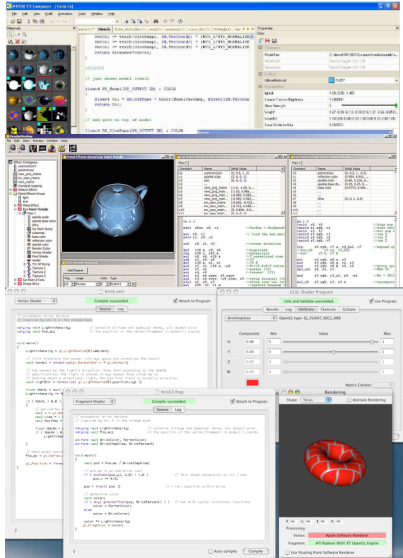
Explosion von GPU-Hochsprachen

- Stanford Shading Language (Vorläufer von Cg)
 - C/Renderman-like
- Cg (Nvidia)
- GLSL ("*glslang*"; OpenGL Shading Language)
- HLSL (Microsoft)
- Alle sind relativ ähnlich zueinander
- Brook, Ashli, ...

G. Zachmann Computer-Graphik 2 - SS 07 Shader 28

GPU IDEs

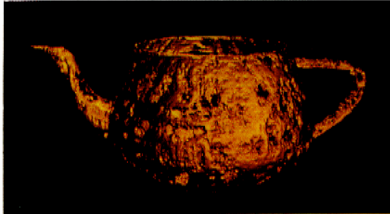
- Ein nicht-triviales Problem ...
- Nvidia: **FX Composer**
 - Kann kein GLSL (?)
- ATI: **RenderMonkey**
- Beide kostenlos, beide nur unter Windows, beide für unsere Zwecke eigtl. schon zu komplex
- Mac: **GLSEditorSample** bzw. meine modifizierte Version
- **Eigene** Testprogramme sind manchmal nicht vermeidbar



G. Zachmann Computer-Graphik 2 - SS 07 Shader 29

RenderMan

- Geschaffen von Pixar in 1988
- Ist heute ein Industriestandard
- Eng an das Ray-Tracing-Paradigma angelehnt
- Mehrere Shader-Arten:
 - Lichtquelle, Oberfläche, Volumen, Displacement



```

surface
dent( float Ka=.4, Kd=.5, Ks=.1, roughness=.25, dent=.4 )
{
  float turbulence;
  point Ni, V;
  float i, freq;

  /* Transform to solid texture coordinate system */
  V = transform("shade",P);

  /* Sum 6 "occures" of noise to form turbulence */
  turbulence = 0; freq = 1.0;
  for( i=0; i<6; i+= 1 ) {
    turbulence += 1/freq * abs( 0.5 - noise( 4*freq*V ) );
    freq *= 2;
  }

  /* Sharpen turbulence */
  turbulence *= turbulence;
  turbulence *= dent;

  /* Displace surface and compute normal */
  P = turbulence * normalize(Ni);
  Ni = faceforward( normalize( calculateNormal(P) ), 1 );
  V = normalize(-V);

  /* Perform shading calculation */
  Oi = 1 - smoothstep( 0.03, 0.05, turbulence );
  Ci = Oi * Cs + (Ka*ambient()) + Ks*specular(Nf,V,roughness);
}

```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 30

Einführung in GLSL

- Fester Bestandteil in OpenGL 2.0 (Oktober 2004)
- Gleiche Syntax für Vertex-Program und Shader-Program
- Plattform-unabhängig
- Rein prozedural (nicht object-orientiert, nicht funktional, ...)
- Syntax basiert auf ANSI C, mit einigen wenigen C++-Features
- Einige kleine Unterschiede zu ANSI-C für saubereres Design

G. Zachmann Computer-Graphik 2 - SS 07 Shader 31

Datentypen

- `float`, `bool`, `int`, `vec{2,3,4}`, `bvec{2,3,4}`, `ivec{2,3,4}`
- Quadratische Matrizen `mat2`, `mat3`, `mat4`
- Arrays – wie in C, aber:
 - nur eindimensional
 - nur konstante Größen (d.h., nur z.B. `float a[4];`)
- Structs (wie in C)
- Datentypen zum Zugriff auf Texturen (später)
- Variablen praktisch wie in C
- Es gibt keine Pointer!

G. Zachmann Computer-Graphik 2 - SS 07 Shader 32

Qualifier (Variablen-Arten)

- **const**
- **attribute:**
 - globale Variable, nur im Vertex-Shader, kann sich pro Vertex ändern
- **uniform:**
 - globale Variable, im Vertex- und Fragment-Shader, gleicher Wert in beiden Shadern, konstant während eines gesamten Primitives
- **varying:**
 - wird vom Vertex-Shader gesetzt (pro Vertex) als Ausgabe,
 - wird vom Rasterizer interpoliert,
 - und vom Fragment-Shader gelesen (pro Pixel)

G. Zachmann Computer-Graphik 2 - SS 07 Shader 33

Operatoren

- grouping: ()
- array subscript: []
- function call and constructor: ()
- field selector and swizzle: .
- postfix: ++ --
- prefix: ++ -- + - !
- binary: * / + -
- relational: < <= > >=
- equality: == !=
- logical: && ^^ [sic] ||
- selection: ? :
- assignment: = *= /= += -=

G. Zachmann Computer-Graphik 2 - SS 07 Shader 34

Skalar/Vektor Constructors

- Es gibt kein Casting: verwende statt dessen Konstruktor-Schreibweise
- Achtung: es gibt keine automatische Konvertierung!
- Es gibt Initialisierung

```

vec2 v2 = vec2(1.0, 2.0);
vec3 v3 = vec3(0.0, 0.0, 1.0);
vec4 v4 = vec4(1.0, 0.5, 0.0, 1.0);
v4 = vec4(1.0); // all 1.0
v4 = vec4(v2, v2); // # components must match
v4 = vec4(v3, 1.0); // dito
v2 = v4; // keep only first components

float f = 1; // error
float f = 1.0; // that's better
int i = int(f); // "cast"
f = float(i);

```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 35

Matrix Constructors

```

vec4 v4; mat4 m4;

mat4( 1.0, 2.0, 3.0, 4.0,
      5.0, 6.0, 7.0, 8.0,
      9.0, 10., 11., 12.,
      13., 14., 15., 16.) // COLUMN MAJOR order!

```

$$\Rightarrow \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

```

mat4( v4, v4, v4, v4 ) // v4 wird spaltenweise eingetragen
mat4( 1.0 ) // = identity matrix
mat3( m4 ) // upper 3x3
vec4( m4 ) // 1st column
float( m4 ) // upper left

```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 36

Zugriff auf Komponenten

- Zugriffsoperatoren auf Komponenten von Vektoren:
 - `.xyzw .rgba .stpq [i]`
- Zugriffsoperatoren für Matrizen:
 - `[i] [i][j]`
 - Achtung: `[i]` liefert die *i*-te **Spalte!**
- Vector components:


```

vec2 v2;
vec4 v4;

v2.x // is a float
v2.x == v2.r == v2.s == v2[0] // comp accessors do the same
v2.z // wrong: undefined for type
v4.rgba // is a vec4
v4.stp // is a vec3
v4.b // is a float
v4.xy // is a vec2
v4.xgp // wrong: mismatched component sets

```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 37

Swizzling & Smearing

- R-values:


```
vec2 v2;
vec4 v4;

v4.wzyx // swizzles, is a vec4
v4.bgra // swizzles, is a vec4
v4.xxxx // smears x, is a vec4
v4.xxx  // smears x, is a vec3
v4.yyxx // duplicates x and y, is a vec4
v2.yyyy // wrong: too many components for type
```
- L-values:


```
vec4 v4 = vec4( 1.0, 2.0, 3.0, 4.0 );

v4.wx = vec2( 7.0, 8.0); // = (8.0, 2.0, 3.0, 7.0)
v4.xx = vec2( 9.0, 3.0); // wrong: x used twice
v4.yz = 11.0;           // wrong: type mismatch
v4.yz = vec2( 5.0 );   // = (8.0, 5.0, 5.0, 7.0)
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 38

Statements und Funktionen

- Flow Control wie in C:
 - `if (bool expression) { ... } else { ... }`
 - `for (initialization; bool expression; loop expr) { ... }`
 - `while (bool expression) { ... }`
 - `do { ... } while (bool expression)`
 - `continue, break`
 - `discard`: nur im Fragment-Shader, wie `exit()` in C, kein Pixel wird gesetzt
- Funktionen:
 - `void main()`: muß 1x im Vertex- und 1x im Fragment-Shader vorkommen
 - `in` = input parameter, `out` = output parameter, `inout` = beides
 - `vec4 func(in float intensity) {`

```
    vec4 color;
    if (intensity > 0.5) color = vec4(1,1,1,1);
    else                  color = vec4(0,0,0,0);
    return( color ); }
```

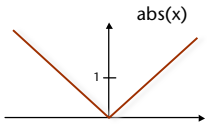
G. Zachmann Computer-Graphik 2 - SS 07 Shader 39

Eingebaute Funktionen

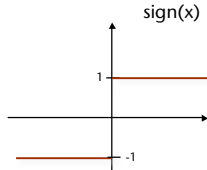
- Trigonometrie: `sin`, `asin`, `radians`, ...
- Exponentialfunktionen: `pow`, `exp`, `log`, `sqrt`, ...
- Sonstige: `abs`, `clamp`, `max`, `sign`, ...
- Alle o.g. Funktionen nehmen und liefern `float`, `vec2`, `vec3`, oder `vec4`, und arbeiten komponentenweise!
- Geometrische Funktionen: `cross(vec3,vec3)`, `mat*vec`, `mat*mat`, `distance()`, `dot()`, `normalize()`, `reflect()`, `refract()`, ...
 - Diese Funktionen nehmen, wenn nichts anderes steht, `float` ... `vec4`
- Vektor-Vergleiche:
 - Komponentenweise: `vec = lessThan(vec, vec)`, `equal()`, ...
 - "Quersumme": `bool = any(vec), all()`

G. Zachmann Computer-Graphik 2 - SS 07 Shader 40

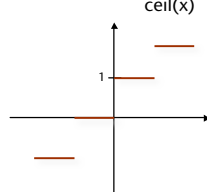
Einige häufige Funktionen



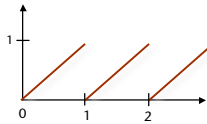
`abs(x)`



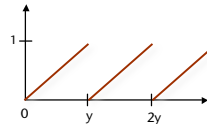
`sign(x)`



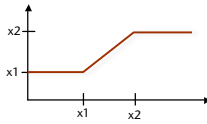
`ceil(x)`



`fract(x)`



`mod(x,y)`



`clamp(x, x1, x2)`

Zur Erinnerung: alle Funktionen arbeiten (komponentenweise) auf `float` ... `vec4` !

G. Zachmann Computer-Graphik 2 - SS 07 Shader 41

The slide contains three graphs and their corresponding code snippets:

- step(t, x):** A graph showing a function that is 0 for $x < t$ and 1 for $x \geq t$. The x-axis is labeled with 0 and t, and the y-axis with 0 and 1.
- smoothstep(t1, t2, x):** A graph showing a smooth S-shaped curve that transitions from 0 to 1 between $x = t1$ and $x = t2$. The x-axis is labeled with 0, t1, and t2, and the y-axis with 0 and 1.
- mix(y1, y2, t):** A graph showing a straight line from $(0, y1)$ to $(1, y2)$. The x-axis is labeled with 0 and 1, and the y-axis with y1 and y2.

```

step(t, x) :=
x <= t ? 0.0 : 1.0

smoothstep(t1, t2, x) :=
t = (x-t1) / (t2-t1);
t = clamp( t, 0.0, 1.0);
return t*t*(3.0-2.0*t);

mix(y1, y2, t) :=
y1*(1.0-t) + y2*t

```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 42

Kommunikation mit OpenGL bzw. der Applikation

- Wie kann man Daten/Parameter an einen Shader übergeben?
Wie kann der Vertex-Shader Daten an den Fragment-Shader ü.g.?
- Geht, aber immer nur in eine Richtung: App. → OpenGL → Vertex-Shader → Fragment-Shader → Framebuffer
- Beide Shader haben Zugriff auf Zustand von OpenGL, z.B. Parameter der Lichtquellen
- Man kann Variablen deklarieren, die von außen gesetzt werden können:
 - Sog. "uniform"-Variablen können sowohl von Vertex- als auch Fragment-Shader gelesen werden
 - Sog. "attribute"-Variablen nur vom Vertex-Shader
- Mittels Texturen können Daten an Shader übergeben werden
 - Interpretation bleibt Shader überlassen

G. Zachmann Computer-Graphik 2 - SS 07 Shader 43

Spezielle vordefinierte Variablen im Vertex-Shader

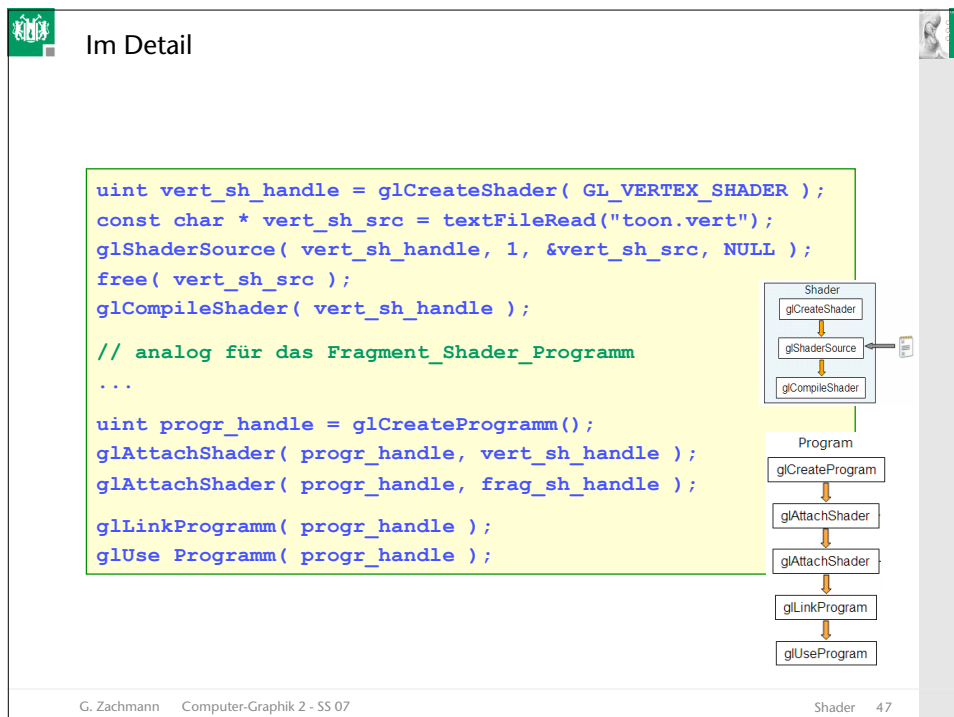
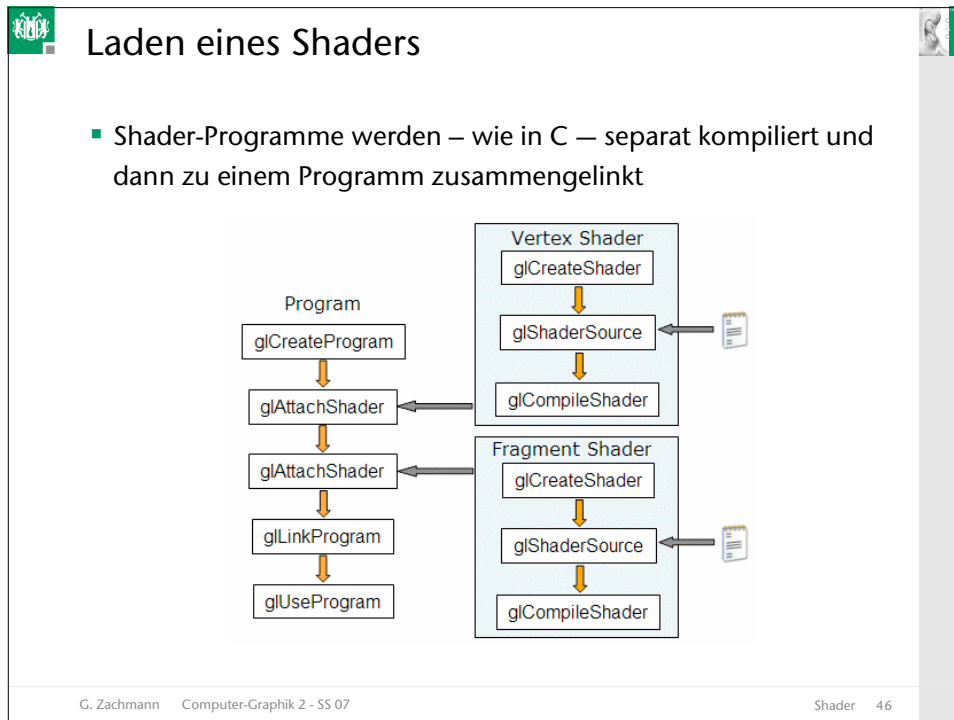
- Output: `gl_Position (vec4)`, ...
 - Diese Variable **muss** vom Shader geschrieben werden!
- Input (*attributes*): `gl_Vertex`, `gl_Normal`, `gl_Color`, `gl_MultiTexCoord0`, ...
 - Alle sind `vec4`
 - Werden gesetzt durch den entsprechenden `gl`-Befehl (`glNormal`, `glColor`, `glTexCoord`; vor `glVertex()`!)
 - Sind read-only
- Output-Variablen:
 - deren Werte werden dann vom Rasterizer interpoliert (über ein Primitiv)
 - `vec4 gl_FrontColor`;
`vec4 gl_TexCoord[]`; ...

G. Zachmann Computer-Graphik 2 - SS 07 Shader 44

Spezielle vordefinierte Variablen im Fragment-Shader

- Input: `gl_Color (vec4)`, `gl_TexCoord[]`
 - Diese werden vom Rasterizer belegt (Interpolation)
 - Read-only
- Spezieller Input: `gl_FragCoord (vec4)`
 - enthält die Pixel-Koordinaten (x,y,z)
- Output: `gl_FragColor (vec4)`, `gl_FragDepth (float)`
 - `gl_FragColor` **muss** vom Shader geschrieben werden!
- Eingebaute Konstanten (für beide Shader):
 - `gl_MaxLights`, ...

G. Zachmann Computer-Graphik 2 - SS 07 Shader 45



Bemerkungen

- Beliebige Anzahl von Shadern und Programmen kann erzeugt werden
- Man kann innerhalb eines Frames zwischen *fixed functionality* und eigenem Programm umschalten (aber natürlich nicht innerhalb eines Primitives, also nicht zwischen `glBegin/glEnd`)
 - Mit `glUseProgram(0)` schaltet man auf *fixed functionality*
- Man kann einen Shader zu mehreren verschiedenen Programmen attachen

G. Zachmann Computer-Graphik 2 - SS 07 Shader 48

Beispiel: Hello_GLSL



A screenshot of a window titled "Hello GLSL" with a standard macOS-style title bar (red, yellow, green buttons). The window content is a solid blue silhouette of a teapot, centered on a white background.

G. Zachmann Computer-Graphik 2 - SS 07 Shader 49

Inspektion der Parameter eines GLSL-Programms

- **Attribut-Variablen:**
 - `glProgramiv()` : liefert die Anzahl aktiver "attribute"-Parameter
 - `glGetActiveAttrib()` : liefert Info über ein bestimmtes Attribut
 - `glGetAttribLocation()` : liefert einen Handle ein Attribut
- **Uniform-Variablen:**
 - `glProgramiv()` : liefert die Anzahl aktiver "uniform"-Parameter
 - `glGetActiveUniform()` : liefert Info zu einem Parameter
- Benötigt man vor allem zur Implementierung von sog. Shader-Editoren

G. Zachmann Computer-Graphik 2 - SS 07 Shader 50

Setzen von "uniform"-Variablen

- Erst `glUseProgram()`
- Dann Handle auf Variable besorgen:


```
uint var_handle = glGetUniformLocation( progr_handle,
                                         "uniform_name" )
```
- Setzen einer uniform-Variable:
 - Für Float:

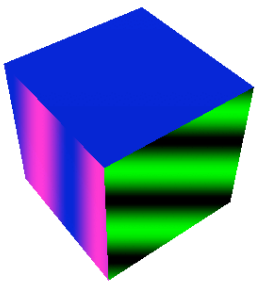

```
glUniform1f( var_handle, f )
```
 - Für Matrizen


```
glUniform4fv( var_handle, count, transpose, float * v)
```

analog gibt es `glUniform{2,3}fv`

G. Zachmann Computer-Graphik 2 - SS 07 Shader 51

Beispiel für uniform-Variablen



G. Zachmann Computer-Graphik 2 - SS 07 Shader 52

Die spezielle Funktion `ftransform`

- Tut genau das, was die fixed-function pipeline in der Vertex-Transformations-Stufe auch tut: einen Vertex von Model-Koordinaten in View-Koordinaten abbilden
- Idiom:


```
gl_Position = ftransform();
```
- Identisch:


```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

G. Zachmann Computer-Graphik 2 - SS 07 Shader 53