



Computer-Graphik I Shader-Programmierung

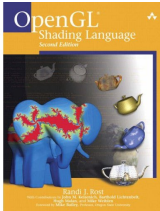


G. Zachmann
Clausthal University, Germany
cg.in.tu-clausthal.de

Literatur

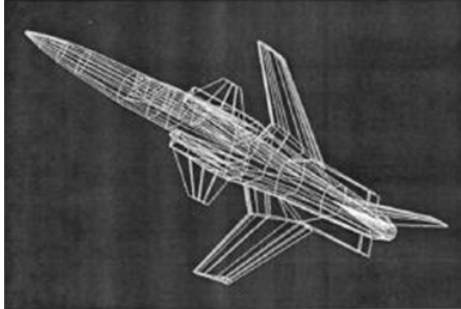
- Das "Orange Book":
 - Randi J. Rost, et al.:
"OpenGL Shading Language",
2nd edition, Addison Wesley.
- Auf der Homepage der Vorlesung:
 - Das Tutorial von Lighthouse3D
 - Mark Olano's "*Brief OpenGL Shading Tutorial*"
 - Der "GLSL Quick Reference Guide"
 - ...



G. Zachmann Computer-Graphik 1 – WS 11/12 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 1 – WS 11/12 Shader 3

The Quest for Realism


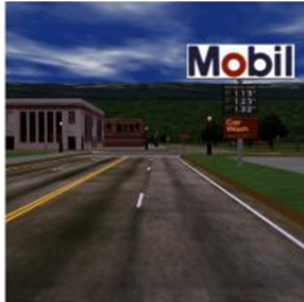
- Zweite Generation – Shaded Solids
 - Vertex-Oper.: Beleuchtungsrechnung & Gouraud-Shading
 - Rasterization: Depth-Interpolation
 - Fragment-Oper.: Depth-Buffer, Color Blending
 - Zeitraum: 1987 - 1992



(Dogfight - SGI)

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 4

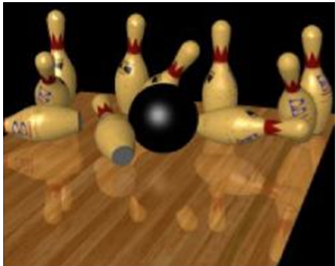

- Dritte Generation – Texture Mapping
 - Vertex-Oper.: Textur-Koordinaten-Transformation
 - Rasterization: Textur-Koordinaten-Interpolation
 - Fragment-Oper.: Textur-Auswertung, Antialiasing
 - Zeitraum: 1992 - 2000

Perfomertown (SCI)

G. Zachmann Computer-Graphik 1 – WS 11/12
Shader 5

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

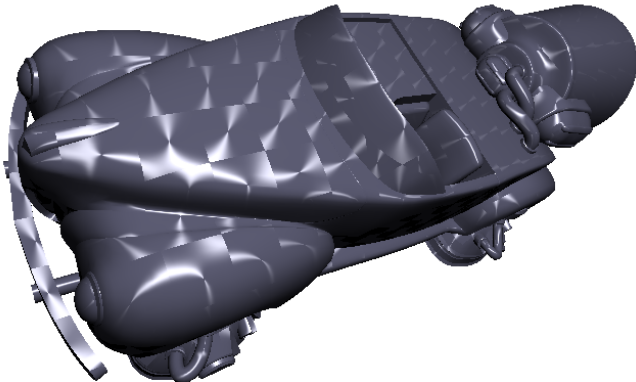



Final Fantasy

G. Zachmann Computer-Graphik 1 – WS 11/12
Shader 6

Beispiele

- Brushed Steel:
 - Prozedurale Textur
 - Anisotrope Beleuchtung




A 3D rendered motorcycle with a brushed steel texture. The surface is highly reflective and shows anisotropic lighting effects, where the reflection is elongated and oriented along the direction of the brush strokes. The motorcycle is shown from a side-rear perspective, highlighting the texture on the fuel tank, seat, and rear fender.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 7

Beispiele


- Schmelzendes Eis:
 - Prozedurale, animierte Textur
 - Bump-mapped environment map



A 3D rendered scene showing melting ice on a 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 distorted, wavy effect on the ice surface.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 8

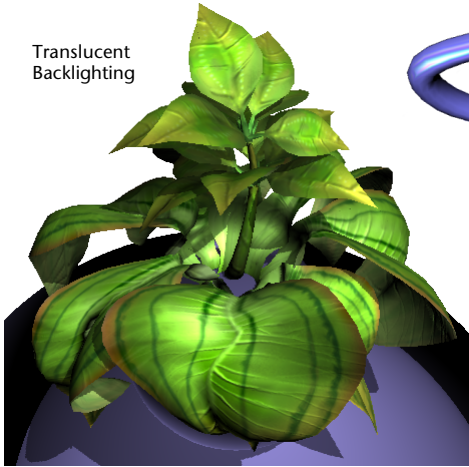
- Sog. „Toon Shading“
 - Ohne Texturen
 - Mit Anti-Aliasing
 - Gute Silhouetten ohne zu starker Verdunkelung



G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 9

- Vegetation & *Thin Film*

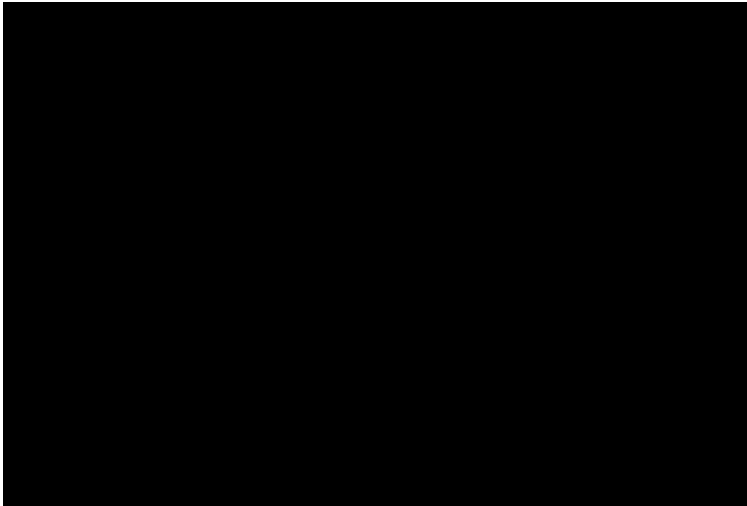
Translucent Backlighting




Beispiel von selbstgemachter Beleuchtungsrechnung; hier: Simulation von Schillern

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 10

Animusic's Pipe Dream



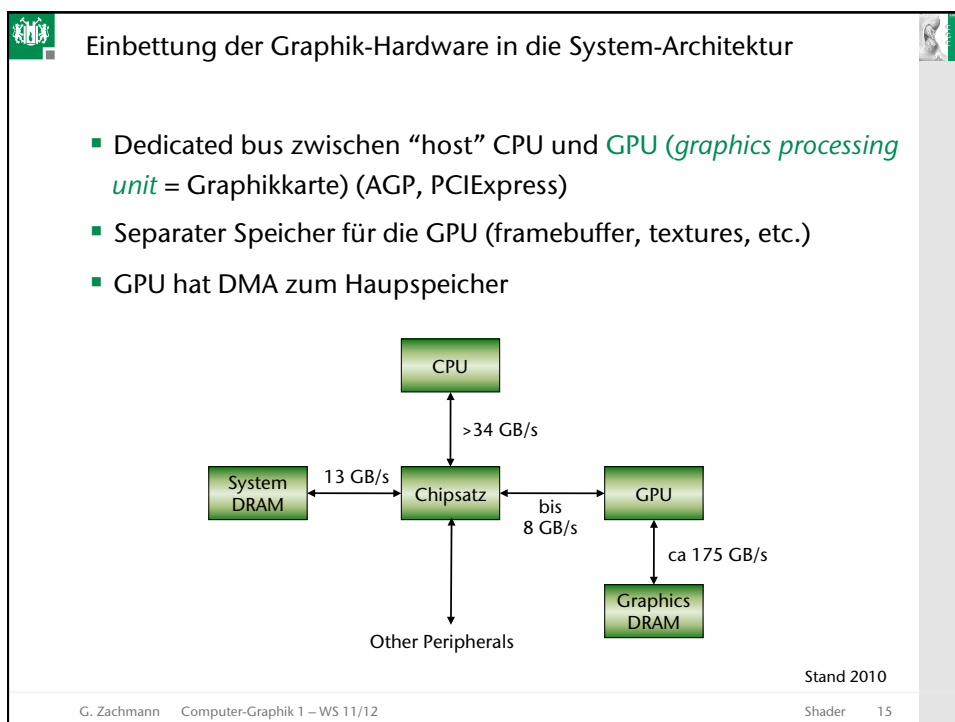
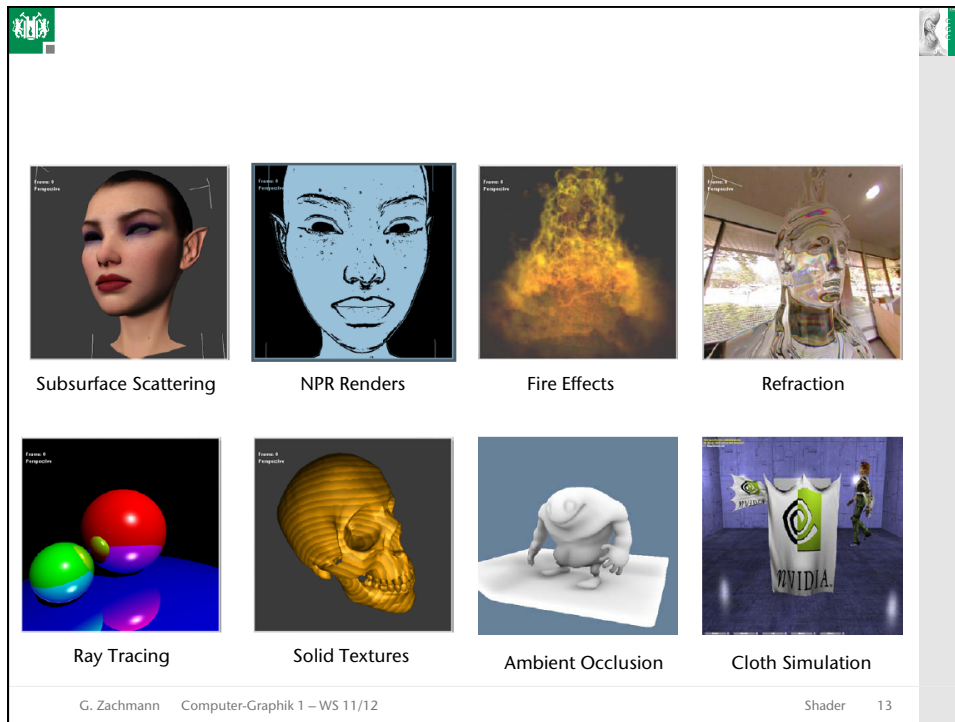
<http://ati.amd.com/developer/demos.html>

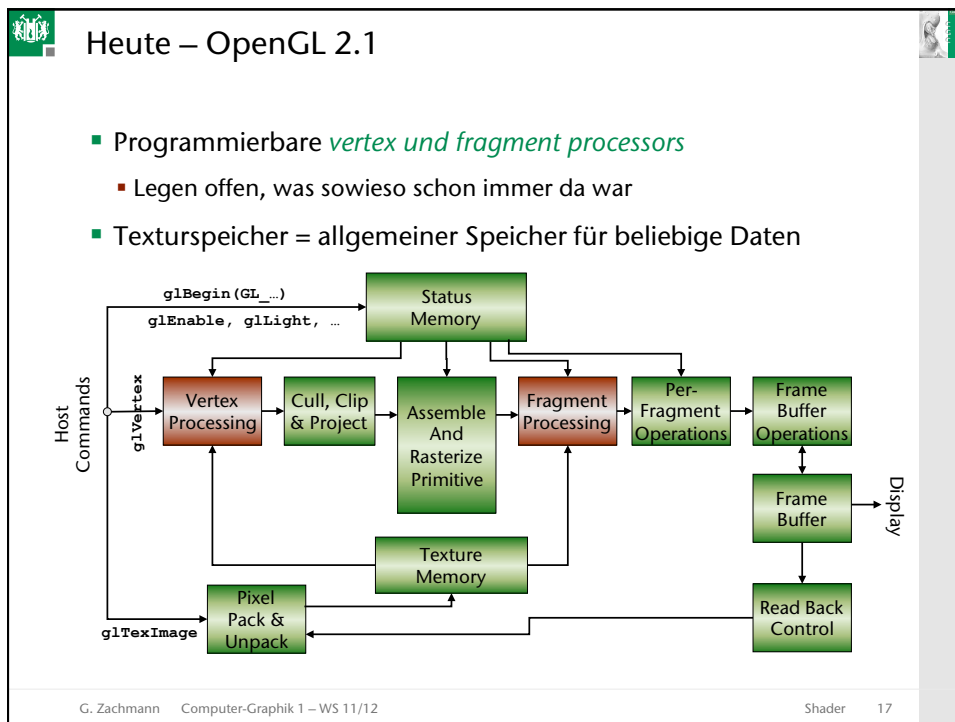
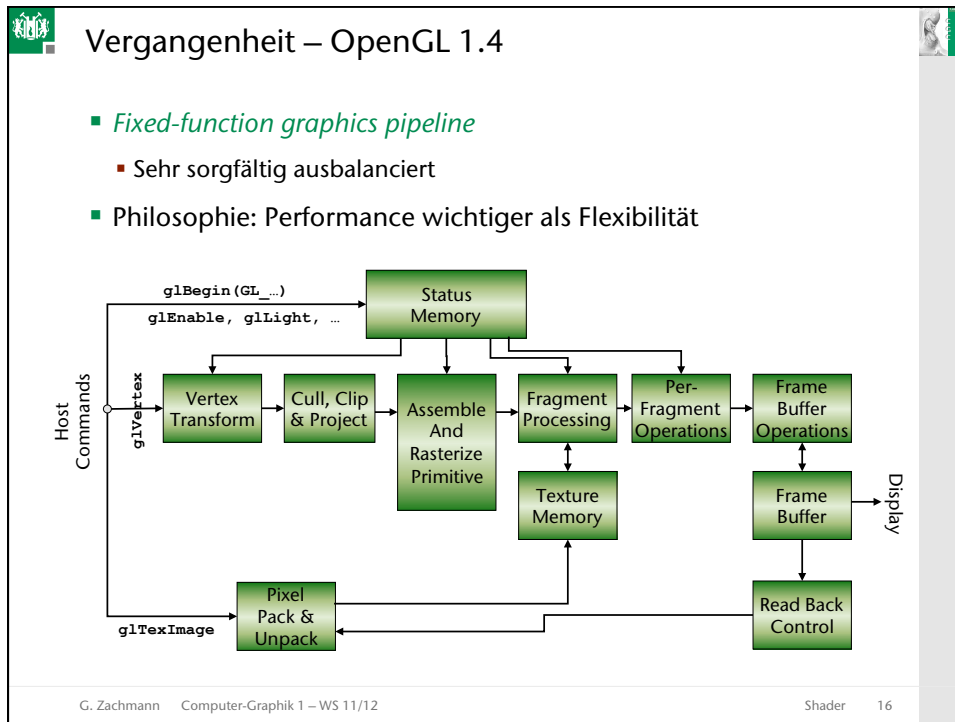
G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 11

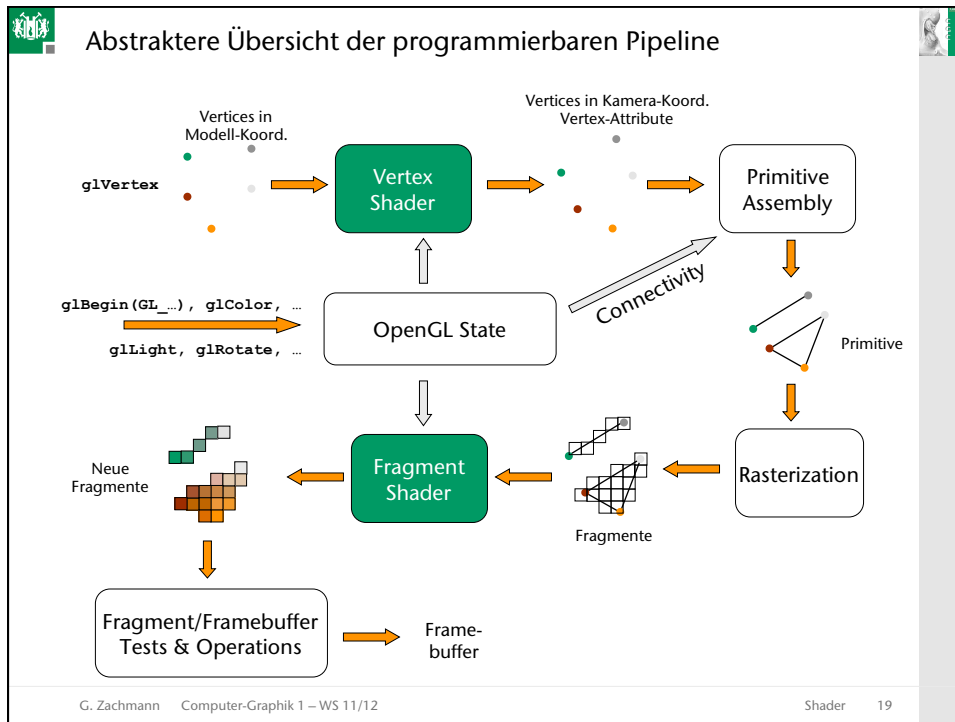


<http://ati.amd.com/developer/demos.html>

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 12







▪ 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

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 21

Inputs & Outputs eines Vertex-Prozessors

- Vertex "*shader*" bekommt eine Reihe von Parametern:
 - Vertex Parameter, OpenGL Zustand, selbst-definierte Attribute
- Resultat muß in vordefinierte Register geschrieben werden, die der Rasterizer dann ausliest und interpoliert

Zur Anzahl der I/O-Register s. "Shader Model 4.0", z.B. http://en.wikipedia.org/wiki/Shader_Model_4.0

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 22

Aufgaben des Vertex-Prozessors

- Beleuchtung und Vertex-Attribute pro Vertex berechnen
- Ein Vertex-Programm ersetzt folgende Funktionalität der fixed-function Pipeline:
 - Vertex- & Normalen-Transformation ins Kamera-Koord.system
 - Transformation mit Projektionsmatr. (perspektivische Division durch z)
 - Normalisierung
 - Per-Vertex Beleuchtungsberechnungen
 - Generierung und/oder Transformation von Texturkoordinaten
- Ein Vertex-Programm ersetzt **NICHT**:
 - Projektion nach 2D und Viewport mapping
 - Clipping
 - Backface Culling
 - Primitive assembly (Triangle setup, edge equations, etc.)

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 23

Inputs & Outputs eines Fragment-Prozessors

- *Fragment "shader"* bekommt eine Reihe von Parametern:
 - OpenGL-Zustand
 - Fragment-Parameter = alle Ausgaben des Vertex-Shaders, aber **interpoliert!**
- Resultat: neues Fragment (i.A. mit anderer Farbe als vorher)

User-Defined Uniform Variables
eyePosition, lightPosition, modelScaleFactor, epsilon, etc.

Standard Rasterizer Attributes
color (r, g, b, a), depth (z),
texture coordinates,
fragment coordinates

User-Defined Varying Attributes
Normals, modelCoord, density,
etc

Fragment Processor

Standard OpenGL variables
FragmentColor,
FragmentDepth

Standard OpenGL State
ModelViewMatrix, glLightSource[0..n],
glFogColor, glFrontMaterial, etc.

Texture Memory
Textures, Tables,
Temp Storage

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 24

Aufgaben des Fragment-Processors

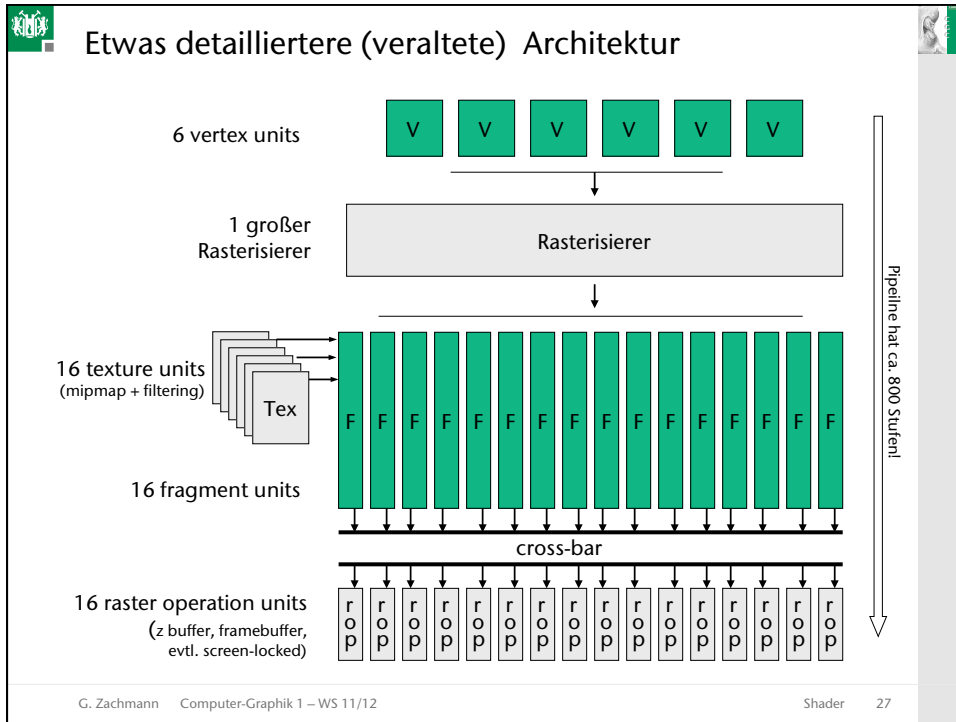
- Ein Fragment-Programm ersetzt folgende Funktionalität der *fixed-function Pipeline* :
 - Operationen auf interpolierten Werten
 - Textur-Zugriff und -Anwendung (z.B. modulate, decal)
 - Fog (color, depth)
 - u.v.m.
- Ein Fragment-Programm ersetzt NICHT :
 - Scan Conversion
 - Pixel packing und unpacking
 - Alle Tests, z.B. Z-Test, Alpha-Test, Stencil-Test, etc.
 - Schreiben in den Framebuffer inkl. Operationen zwischen Fragment und Framebuffer (z.B. Alpha-Blending, logische Operationen, etc.)
 - Schreiben in den Z-Buffer
 - u.v.m.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 25

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 1 – WS 11/12 Shader 26



Aktuelle Architektur (hier Fermi)

- Keine Unterscheidung mehr zwischen Vertex- und Fragment-Shader sondern programmierbare Shader genannt **Cores**
- Jeder Core hat eine FP- und Int-Unit
 - Mehrere Cores teilen sich eine SFU = *special function unit* (trig. Fkt.en, log, etc.)
- Cores werden zu einem sogenannten **Streaming Multiprocessor (SM)** zusammengefasst
- Eine GPU hat mehrere SM's

The diagram shows the internal structure of a Streaming Multiprocessor (SM):

- Instruction Cache** at the top.
- Warp Scheduler** and **Dispatch Unit** blocks.
- Register File (32,768 x 32-bit)** in the middle.
- A grid of **Core** units, each containing **LD/ST** and **SFU** components.
- Interconnect Network** below the core grid.
- 64 KB Shared Memory / L1 Cache**.
- Uniform Cache** and **Texture Cache** (containing 'Tex' blocks).
- PolyMorph Engine** at the bottom, including **Vertex Fetch**, **Tessellator**, **Viewport Transform**, **Attribute Setup**, and **Stream Output**.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 28

Wie sieht nun echter Shader-Code aus?

Assembly	Hochsprache
<pre> 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; </pre>	<pre> float spec = pow(max(0, dot(n,h)), phongExp); color cResult = Cd * (cAmbi + cDiff) + Cs * spec * cSpec; </pre>
<p>Einfacher Phong-Shader ausgedrückt in Assembly und GLSL</p>	

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 29

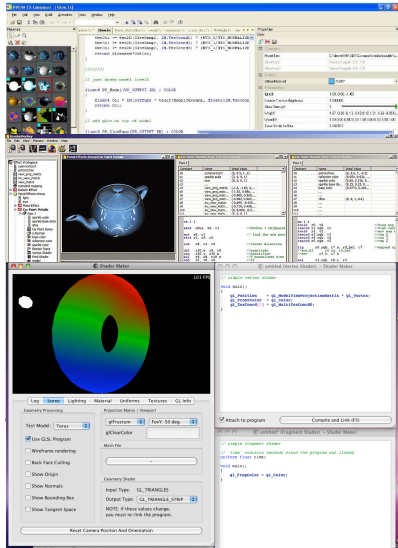
Explosion von GPU-Hochsprachen

- 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 1 – WS 11/12 Shader 30

GPU IDEs

- Ein nicht-triviales Problem ...
 - **Eigene** Testprogramme sind manchmal nicht vermeidbar
- Nvidia: **FX Composer**
 - Kann kein GLSL (?)
- ATI: **RenderMonkey**
- Beide kostenlos, beide nur unter Windows, beide für unsere Zwecke eigtl. schon zu komplex
- **Shader Maker** (Studienarbeit):
 - http://cg.in.tu-clausthal.de/publications.shtml#shader_maker



The image shows three screenshots of GPU IDEs. The top one is RenderMonkey, showing a 3D scene with a blue sphere and a red cube. The middle one is FX Composer, showing a 3D scene with a blue sphere and a red cube. The bottom one is Shader Maker, showing a 3D scene with a blue sphere and a red cube, and a shader editor window with GLSL code.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 31

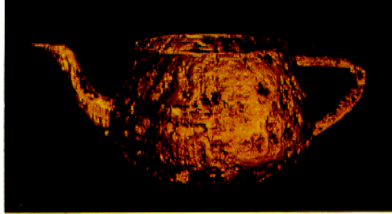
Debugging ...

- Es gibt keinen Debugger!
- Es gibt noch nicht einmal "printf-Debugging"!!
- Meine Tips:
 - Von einem funktionierenden Shader ausgehen und diesen in winzigen Schritten (einzelne Zeilen) modifizieren
 - Bei Aufgaben, wo mehrere Durchläufe gemacht werden müssen: nach jedem Durchlauf Textur / Framebuffer anzeigen

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 32

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 $w=.4, $d=.5, $r=.1, roughness=.25, dent=.4 )
{
  float turbulence;
  point Nf, Vf;
  float i, freq;

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

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

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

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

  /* Perform shading calculation */
  Di = 1 - smoothstep( 0.05, 0.05, turbulence );
  Ci = Di * Cs * ($r*ambient() + $r*specular(Nf,V,roughness));
}

```

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 33

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 1 – WS 11/12 Shader 34

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 1 – WS 11/12 Shader 35

Qualifier (Variablen-Arten)

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

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 36

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 1 – WS 11/12 Shader 37

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 1 – WS 11/12 Shader 38

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!

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

```

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

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 39

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 1 – WS 11/12 Shader 40

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 1 – WS 11/12 Shader 41

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, Pixel wird **nicht** 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 1 – WS 11/12 Shader 42

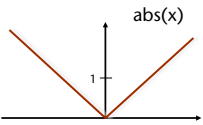


## Eingebaute Funktionen

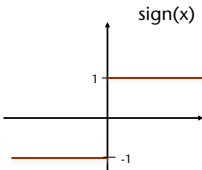
- Trigonometrie: `sin`, `asin`, `radians`, ...
- Exponentialfunktionen: `pow`, `exp`, `log`, `sqrt`, ...
- Sonstige: `abs`, `clamp`, `max`, `sign`, ...
- Alle o.g. Funktionen nehmen und liefern `float`, `vec2`, `vec3`, oder `vec4`, und arbeiten komponentenweise!
- Geometrische Funktionen: `cross(vec3,vec3)`, `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 1 – WS 11/12 Shader 43

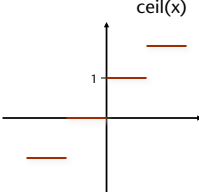
## Einige häufige Funktionen



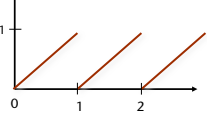
`abs(x)`



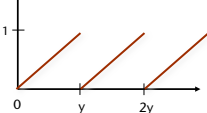
`sign(x)`



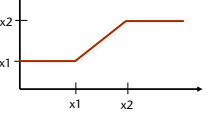
`ceil(x)`



`fract(x)`



`mod(x,y)`



`clamp(x, x1, x2)`

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

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 44

The slide contains three graphs and their corresponding code definitions:

- 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 passing through the points (0, y1) and (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 1 – WS 11/12 Shader 45

## 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 den 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 1 – WS 11/12 Shader 46

### 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
- Weitere Output-Variablen:
  - deren Werte werden dann vom Rasterizer interpoliert (über ein Primitiv)
  - `vec4 gl_FrontColor;`  
`vec4 gl_TexCoord[]; ...`

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 47

### Spezielle vordefinierte Variablen im Fragment-Shader

- Input: `gl_Color (vec4)`, `gl_TexCoord[]`
  - Diese werden vom Rasterizer belegt (durch 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 1 – WS 11/12 Shader 48

## Laden eines Shaders

- Shader-Programme werden – wie in C – separat kompiliert und dann zu einem Programm zusammengelinkt

Diagram illustrating the process of loading and linking shaders:

- Program:**
  - glCreateProgram
  - glAttachShader
  - glAttachShader
  - glLinkProgram
  - glUseProgram
- Vertex Shader:**
  - glCreateShader
  - glShaderSource
  - glCompileShader
- Fragment Shader:**
  - glCreateShader
  - glShaderSource
  - glCompileShader

The diagram shows that the Vertex Shader and Fragment Shader are compiled separately and then attached to the Program. The Program is then linked and used.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 49

## Im Detail

```

uint vert_sh_handle = glCreateShader(GL_VERTEX_SHADER);
const char * vert_sh_src = textFileRead("toon.vert");
glShaderSource(vert_sh_handle, 1, vert_sh_src, NULL);
free(vert_sh_src);
glCompileShader(vert_sh_handle);

// analog für das Fragment_Shader_Programm
...

uint progr_handle = glCreateProgramm();
glAttachShader(progr_handle, vert_sh_handle);
glAttachShader(progr_handle, frag_sh_handle);

glLinkProgramm(progr_handle);
glUseProgram(progr_handle);

```

Diagram illustrating the detailed process of loading and linking a shader:

- Shader:**
  - glCreateShader
  - glShaderSource
  - glCompileShader
- Program:**
  - glCreateProgramm
  - glAttachShader
  - glAttachShader
  - glLinkProgramm
  - glUseProgram

The diagram shows that the Shader is compiled and then attached to the Program. The Program is then linked and used.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 50

## 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 1 – WS 11/12 Shader 51

## 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 shows a solid blue silhouette of a teapot. Below the teapot, the text "lighthouse\_tutorial/hello\_gsl\*" is visible.

lighthouse\_tutorial/hello\_gsl\*

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 52

## Inspektion eines GLSL-Programms

- Über das Programm:
  - `glGetProgramiv()` : liefert verschiedene Infos über das aktuell aktive Shader-Programm, z.B. eine Liste aktiver "attribute"- oder "uniform"-Variablen
- Attribut-Variablen:
  - `glGetActiveAttrib()` : liefert Info über ein bestimmtes Attribut
  - `glGetAttribLocation()` : liefert einen Handle ein Attribut
- Uniform-Variablen:
  - `glGetActiveUniform()` : liefert Info zu einem Parameter
- Benötigt man vor allem zur Implementierung von sog. Shader-Editoren

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 53

## Setzen von "uniform"-Variablen

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

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

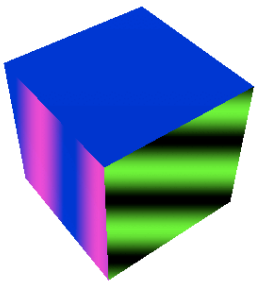
```
glUniform1f(var_handle, f)
```
  - Für Matrizen
 

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

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

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 54

### Beispiel für uniform-Variable



G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 55

### 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 transformieren
- Idiom:
 

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

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

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 56

## Beispiel für die Modifikation der Geometrie

- Wie man mit den Koordinaten (und sonstigen Attributen) eines Vertex im Vertex-Shader verfährt, ist völlig frei – Beispiel:

Flatten Shader

`lighthouse_tutorial/flatten.*`

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 57

## Zustandsvariablen

- Zeigen den aktuellen Zustand von OpenGL an
- Sind als "uniform"-Variablen implementiert
- Die am häufigsten benötigten Zustandsvariablen sind: die aktuellen Matrizen

```

uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat3 gl_NormalMatrix;
uniform mat4 gl_TextureMatrix[n];
uniform mat4 gl_*MatrixInverse;

```

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 58



- Das aktuelle Material:

```
struct gl_MaterialParameters
{
 vec4 emission;
 vec4 ambient;
 vec4 diffuse;
 vec4 specular;
 float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
```

- Aktuelle Lichtquellen(-parameter):

```
struct gl_LightSourceParameters
{
 vec4 ambient;
 vec4 diffuse;
 vec4 specular;
 vec4 position;
 vec4 halfVector;
 vec3 spotDirection;
 float spotExponent;
 float spotCutoff;
 float spotCosCutoff;
 float constantAttenuation;
 float linearAttenuation;
 float quadraticAttenuation;
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

- Und viele weitere (z.B. zu Texturen, Clipping Planes,...)

## Parameter-Übergabe von Vertex- zu Fragment-Shader

- Mittels sog. "**varying**"-Variablen:
 

```
varying vec3 myInterpolant;
```
- Achtung: dazwischen sitzt der Rasterizer und interpoliert!
  - Der Rasterizer interpoliert auch die "**varying**"-Variablen! (zusätzlich zu Position, Farbe, etc.; hence the name)

The diagram illustrates the rendering pipeline. It starts with 'Vertex Processing', which outputs vertices (represented by colored dots). These vertices are then processed by 'Assemble And Rasterize Primitive', which generates a rasterized primitive (represented by a grid of squares). Finally, 'Fragment Processing' is applied to the rasterized primitive. The variable 'myInterpolant' is shown at each stage, indicating its use in the vertex, rasterization, and fragment processing stages.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 61

## Beispiel für Verwendung von varying- und Zustands-Variablen

- Der "Toon-Shader":
  - Berechnet einen stark diskretisierten diffusen Farbanteil (typ. 3 Stufen)
- Der "Gooch-Shader":
  - Interpoliert zwischen 2 Farben, abhängig vom Winkel zwischen Normale und Lichtvektor
- Sind schon einfache Beispiele für "*non-photorealistic rendering*" (NPR)

The first screenshot, titled 'Toon Shading', shows a teapot rendered with a flat, discrete color palette, characteristic of toon shading. The second screenshot, titled 'Rendering', shows the same teapot rendered with a smooth, interpolated color gradient, characteristic of the Gooch shader.

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 62

## Attribute

- Vordefiniert:

```
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
attribute vec4 gl_MultiTexCoord[n];
attribute vec4 gl_SecondaryColor;
attribute float gl_FogCoord;
```

- Man kann selbst eigene Attribute definieren:

- Im Vertex-Shader: `attribute vec3 myAttrib;`

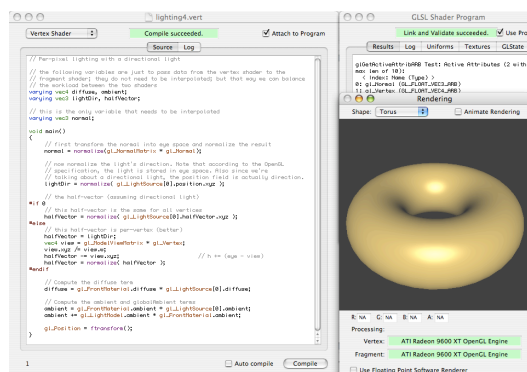
- Im C-Programm :

```
handle = glGetAttribLocation(prog_handle, "myAttrib");
. . .
glVertexAttrib3f(handle, v1, v2, v3);
```

## Beispiel: Per-Pixel Lighting

1. Diffuse lighting per-vertex
2. Mit ambientem Licht
3. Mit spekularem Lichtanteil
4. Per-Pixel Lighting

lighting[1-4].\*



## Achtung bei Subtraktion homogener Punkte

- Homogener Punkt  $\mathbf{v} = \text{vec4}(\mathbf{v}.xyz, \mathbf{v}.w)$ 
  - 3D-Äquivalent =  $\mathbf{v}.xyz/\mathbf{v}.w$
- Subtraktion zweier Punkte/Vektoren  $\mathbf{v}$  und  $\mathbf{e}$  :
  - Homogen:  $\mathbf{v} - \mathbf{e}$
  - Als 3D-Äquivalent:
 
$$\frac{\mathbf{v}.xyz}{\mathbf{v}.w} - \frac{\mathbf{e}.xyz}{\mathbf{e}.w} = \frac{\mathbf{v}.xyz \cdot \mathbf{e}.w - \mathbf{e}.xyz \cdot \mathbf{v}.w}{\mathbf{v}.w \cdot \mathbf{e}.w}$$
- Für die Normierung gilt:  $\left(\frac{\mathbf{v}}{a}\right)^0 = \mathbf{v}^0$
- Zusammen in GLSL :
 

```
normalize(v-e) ≡ normalize(v.xyz*e.w - e.xyz*v.w)
```
- Vorteil: klappt auch falls  $\mathbf{v}.w$  oder  $\mathbf{e}.w = 0$  !

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 65

## Der Z-Test in der Pipeline

- Wann findet der Z-Test statt?
 

Fragment  
Shader

→

Z-Test

oder

Z-Test


→

Fragment  
Shader
- Early-Z:
 

Z-Test

→

Fragment  
Shader

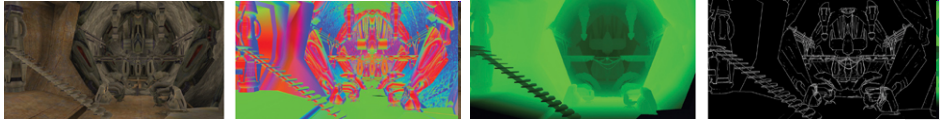


  - Spart teure Fragment-Shader-Programmausführungen
  - Reduziert Bandbreite von der GPU zum Framebuffer, und vom Texturspeicher zur GPU
  - Wird automatisch *deaktiviert*, falls Shader-Programm Z-Wert (`gl_FragDepth`) manipuliert!

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 66

## Deferred Shading / Deferred Lighting

- 2-Pass-Rendering (oder mehr Passes):
  1. Geometry-only: rendere Geometrie, ohne Lighting/Shading, speichere statt dessen alle für's Lighting notwendigen Attribute in einem "G-Buffer" (= Satz von user-defined Buffers für die notwendigen Daten)
 



Fragment Colors      Normals      Depth      Edge Weight
  2. Lighting-only:
    - Setze Lichtquellen
    - Rendere 1 großes Quad (um pro Pixel den Fragment-Shader 1x zu aktivieren)
    - Lese im Fragment-Shader den G-Buffer
    - Werte im Fragment-Shader das Lighting-Modell aus & schreibe in Color-Buffer

G. Zachmann    Computer-Graphik 1 – WS 11/12      Shader    67

- Fertige Szene:
 

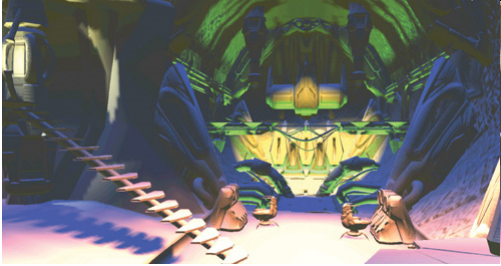


Image from Tabula Rasa
- Vorteile: vermeidet aufwendiges Shading von Fragmenten, die den Z-Test sowieso nicht überleben
- Nachteil: benötigt mehr Framebuffer-Speicher
- Frage: Was ist mit der Bandbreite?
- Literatur: GPU Gems 3, Kapitel 19 ( <http://developer.nvidia.com/object/gpu-gems-3.html> )

G. Zachmann    Computer-Graphik 1 – WS 11/12      Shader    68

Weiteres Beispiel, wo diese Technik angewendet wurde



*S.T.A.L.K.E.R.: Clear Skies*

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 69

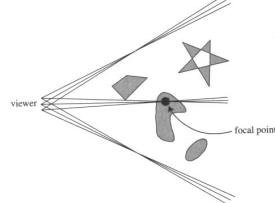
▪ Weitere Vorteile:


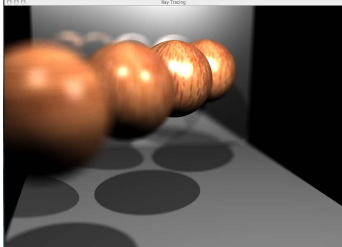
- Einige Rendering-Techniken benötigen sowieso einen ersten Z-Only-Pass (z.B. Shadow Volumes), also kann man gleich etwas mehr bei diesem Pass im Buffer speichern
- Die Kosten für Lighting sind unabhängig von der Objekt-Komplexität (= Anzahl Vertices)
- Anzahl Lichtquellen ist (potentiell) unendlich (speichere Lichtquellen in Textur)

G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 70

## Depth-of-Field (Tiefen(un)schärfe)


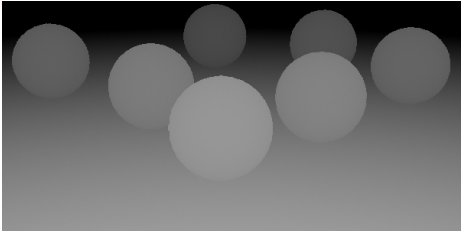
- Die alte (teure) Methode:
  - Rendere die Szene  $n$  Mal von leicht verschiedenen Viewpoints, je nach Größe der (virtuellen) Blende (*aperture*)
  - Rendere jedesmal in den **Accumulation-Buffer**
  - Teile am Ende Werte im Accumulation-Buffer durch  $n \rightarrow$  Mittelwert




G. Zachmann Computer-Graphik 1 – WS 11/12
Shader 71

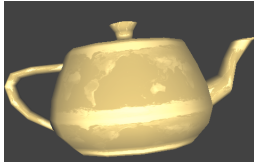
## Beispiel

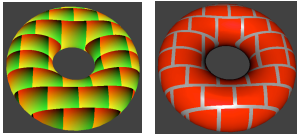


G. Zachmann Computer-Graphik 1 – WS 11/12
Shader 72


## Ausblick (in Computer-Graphik 2)



"Gloss-Textur"



Prozedurale Texturen



*Our Method*      *Ray Traced*

Lichtbrechung (1 bounce)

G. Zachmann    Computer-Graphik 1 – WS 11/12      Shader    73



G. Zachmann    Computer-Graphik 1 – WS 11/12      Shader    74





G. Zachmann Computer-Graphik 1 – WS 11/12 Shader 75