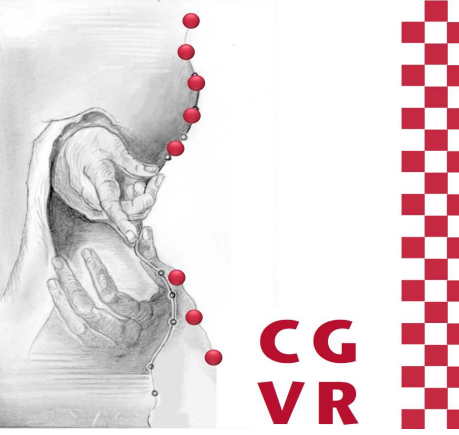




Computer-Graphik I

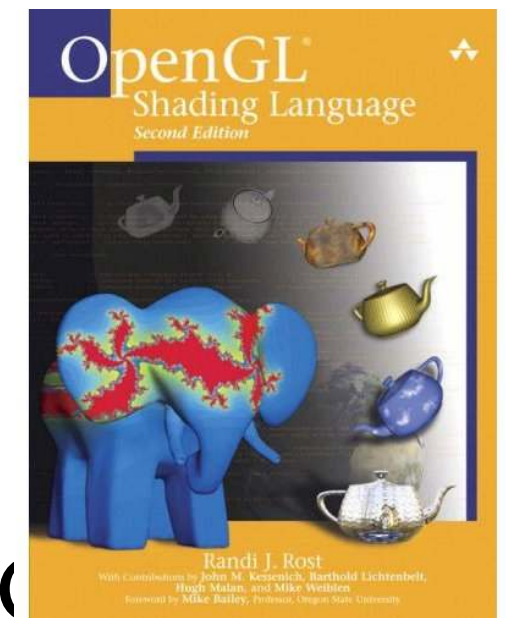
Shader-Programmierung



G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

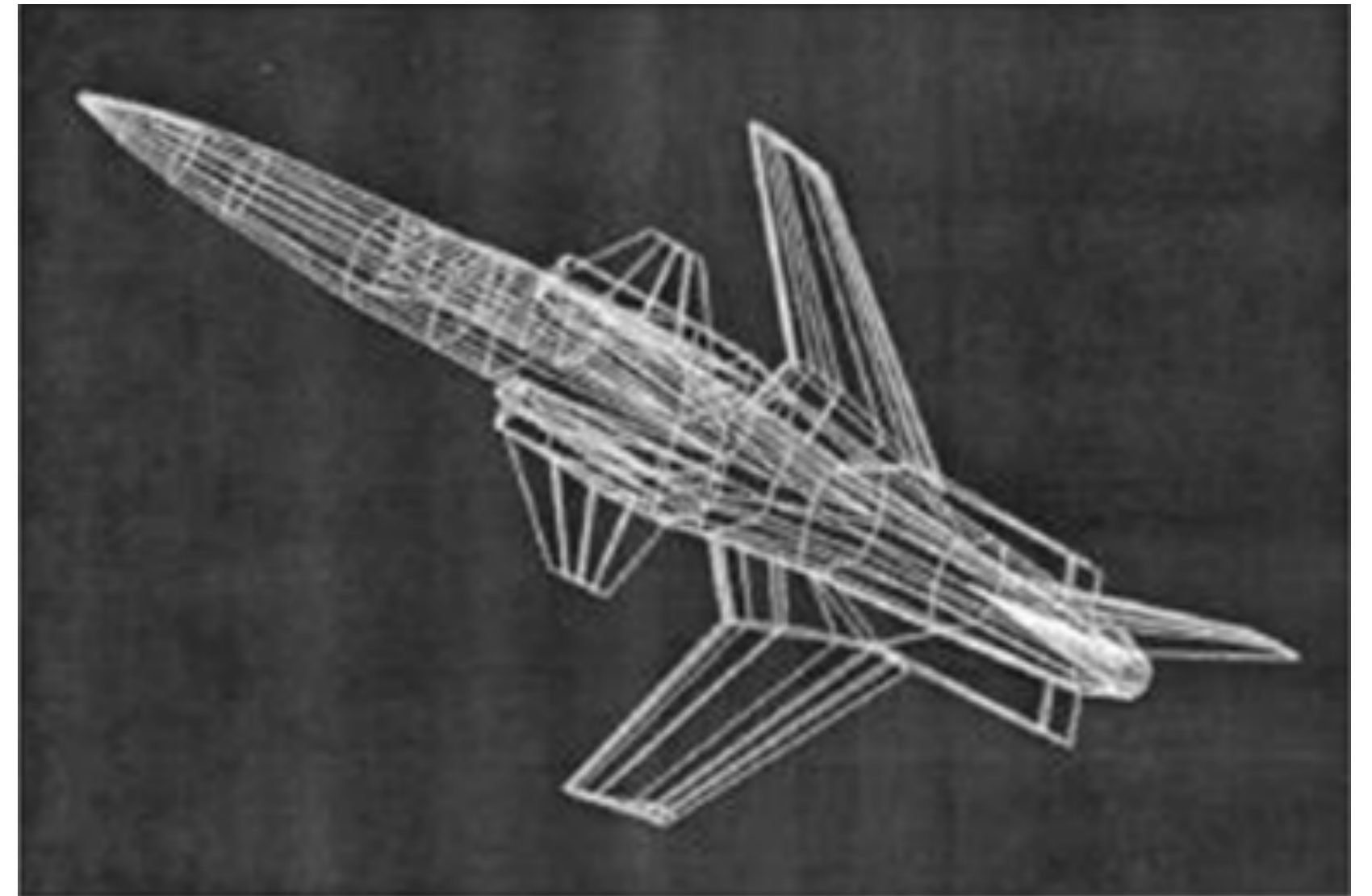


- Das "Orange Book" (veraltet)
- Das aktuelle "Red Book"
- Jacobo Rodríguez: *GLSL Essentials* (2013)
 - <https://www.packtpub.com/mapt/book/Hardware%20&%209781849698009>
- OGLdev: <http://ogldev.atspace.co.uk/index.html>
- Siehe die zahlreichen Links und Dokumente auf der VL-Homepage!



The Quest for Realism

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



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



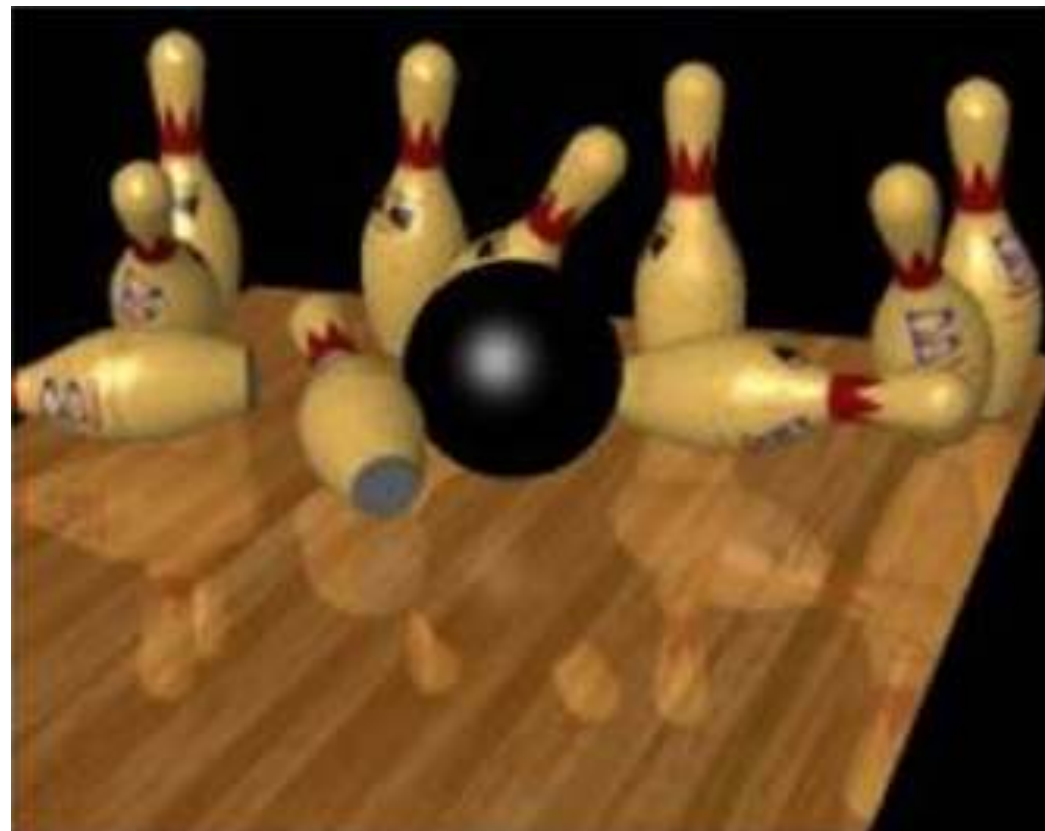
(Dogfight - SGI)

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



Performertown (SGI)

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



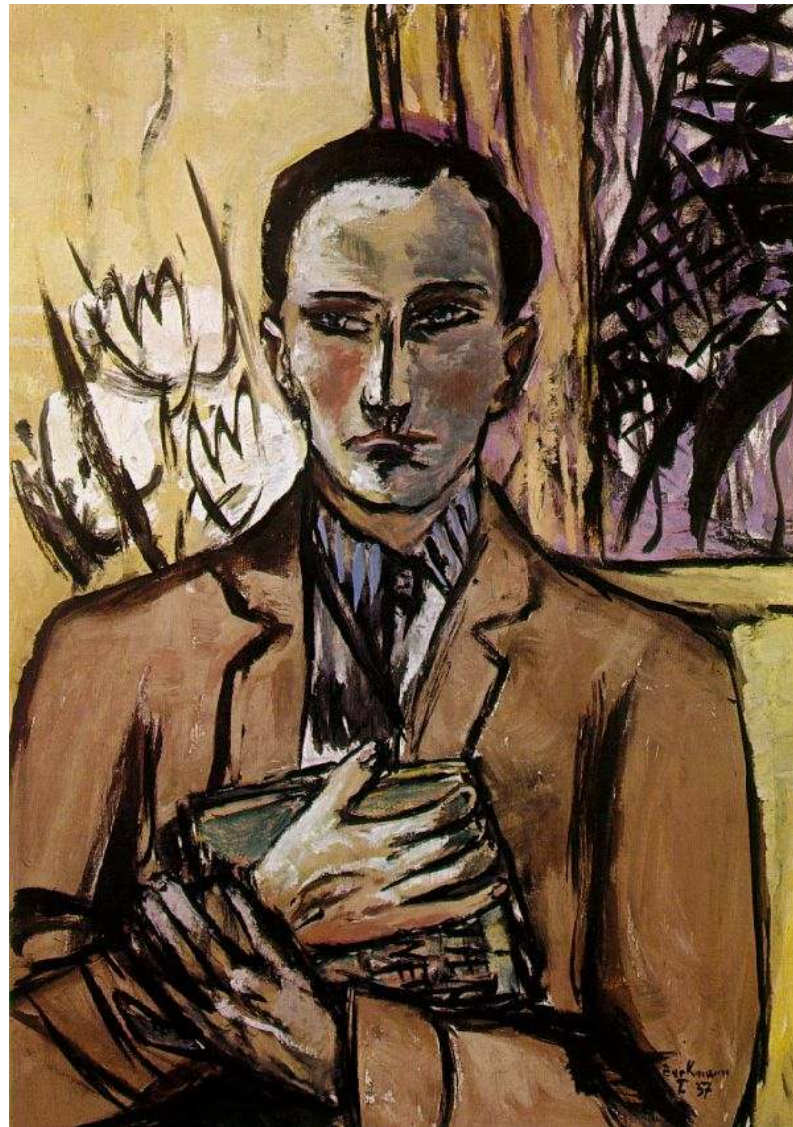
Final Fantasy

Beispiele für Effekte, die nur mit Shadern erreichbar sind

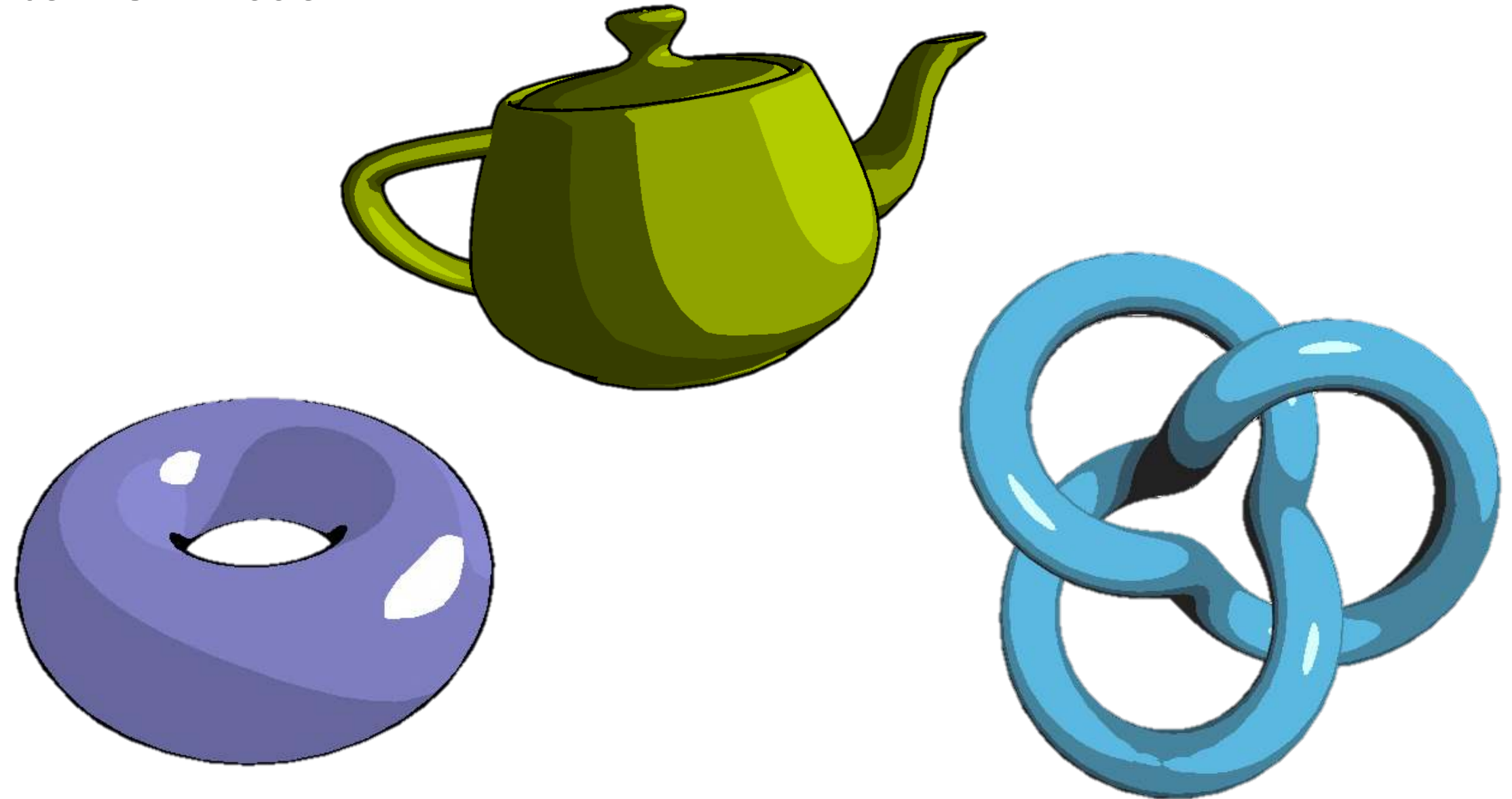
- Brushed Steel:
 - Prozedurale Textur
 - Anisotropes Lighting-Model
- Schmelzendes Eis:
 - Prozedurale, animierte Textur
 - Bump-mapped environment map



- Sog. **Toon Shading**:
 - Eher eine Form des Non-Photorealistic Rendering (NPR)
 - Oft kombiniert mit betonten Umrissen



Max Beckmann



- Vegetation & *Thin Film*

Translucent Backlighting



Simulation von Schillern durch Interferenz; kein einfaches Phong-Modell mehr



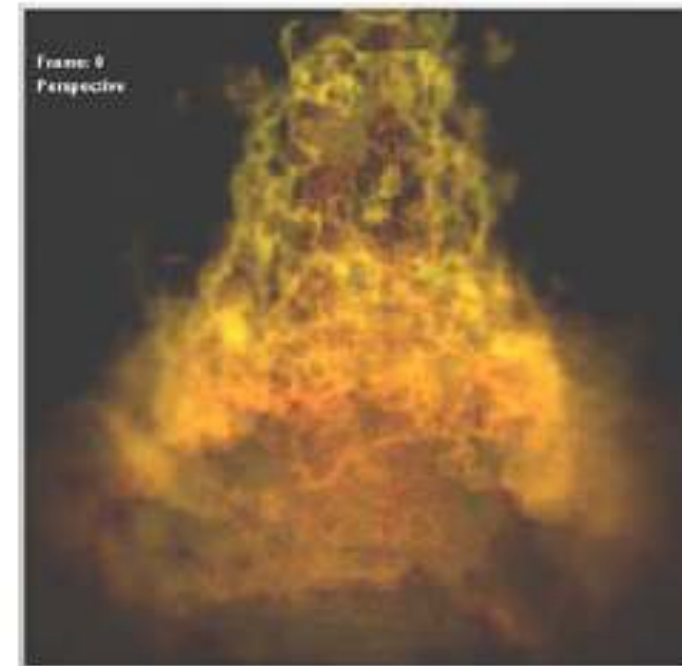
Other Advanced Shader Techniques



Subsurface Scattering



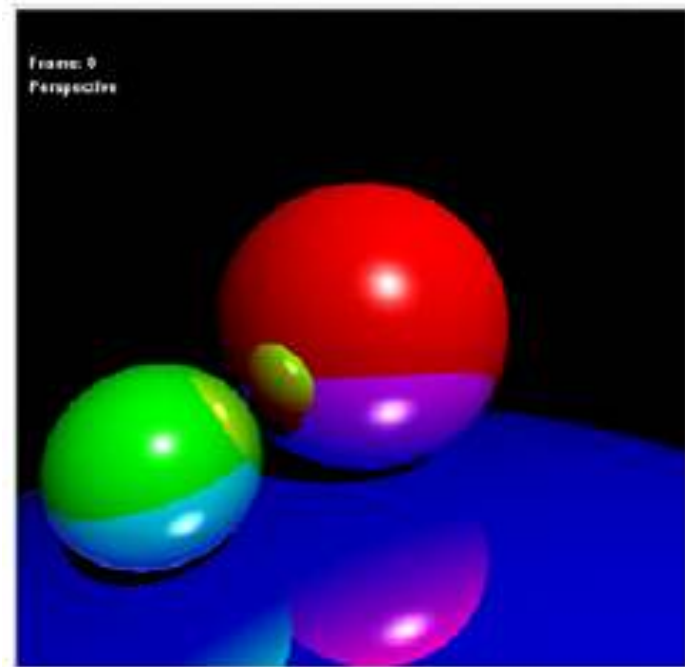
NPR Renders



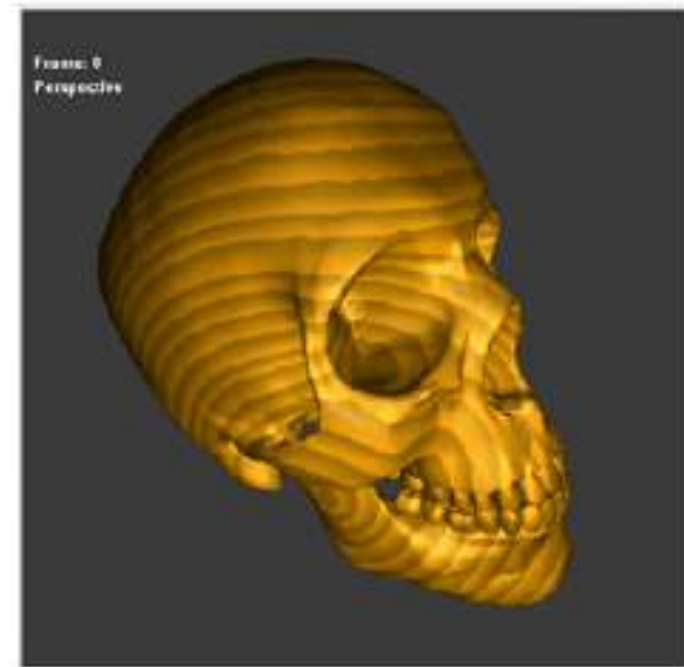
Fire Effects



Refraction



Ray Tracing



Solid Textures

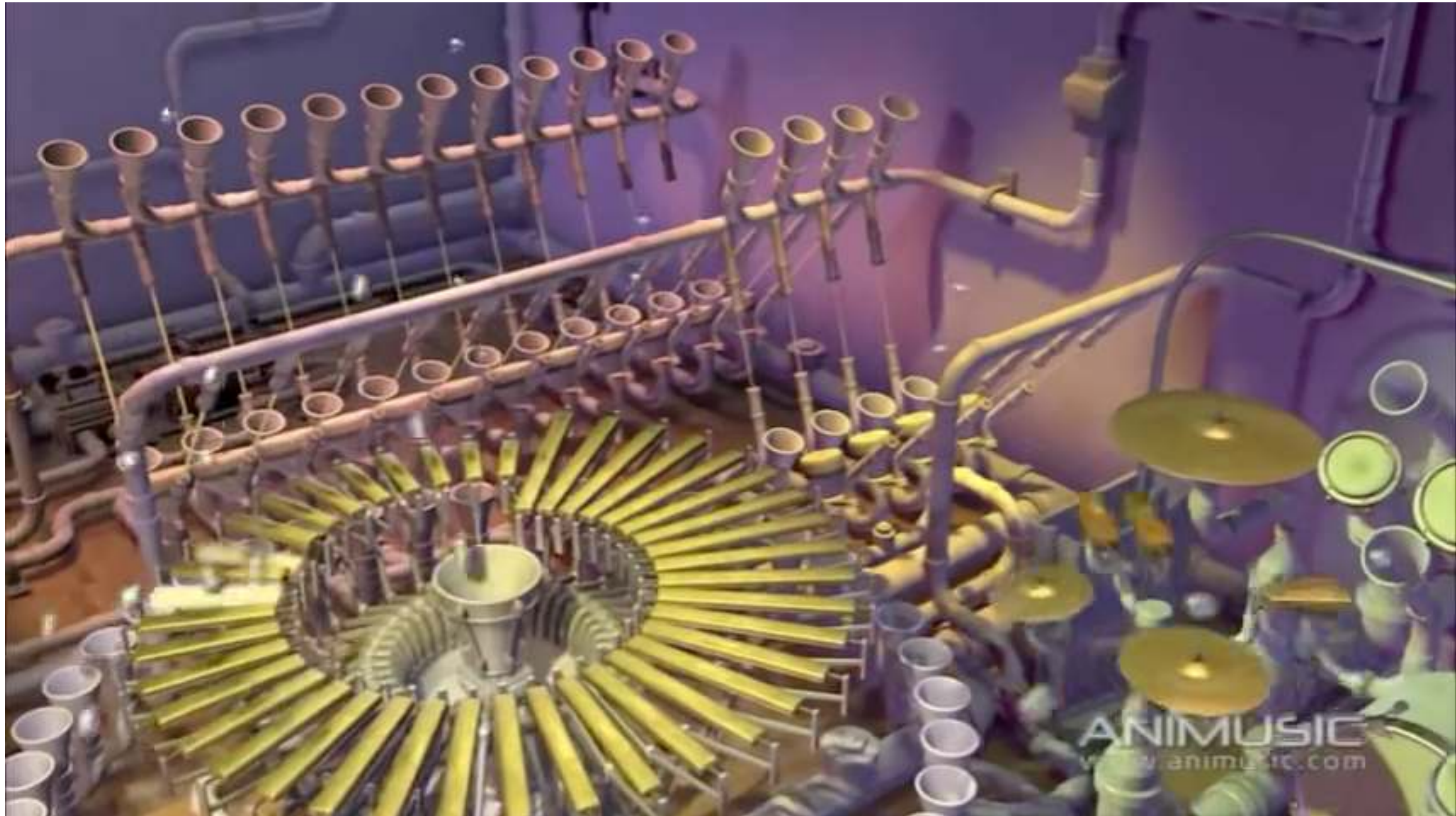


Ambient Occlusion



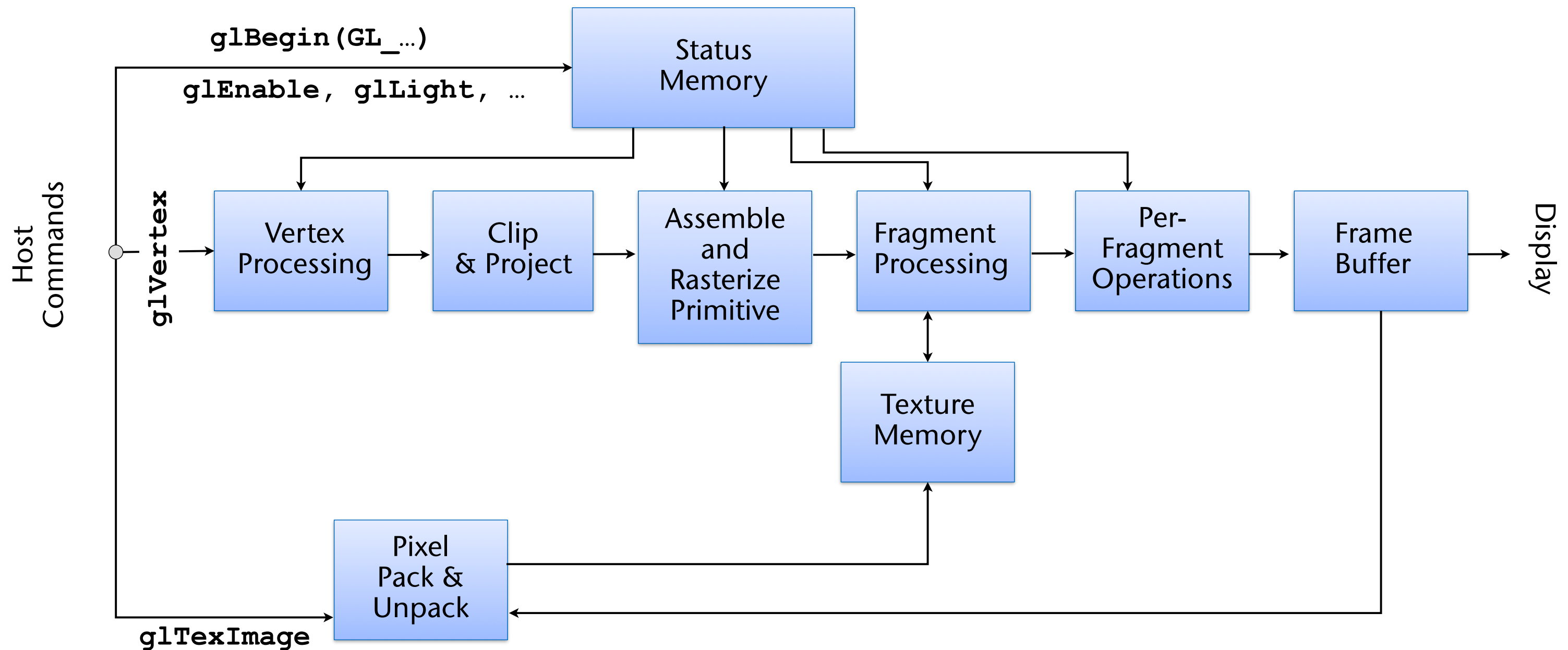
Cloth Simulation

Demo: Animusic's Pipe Dream

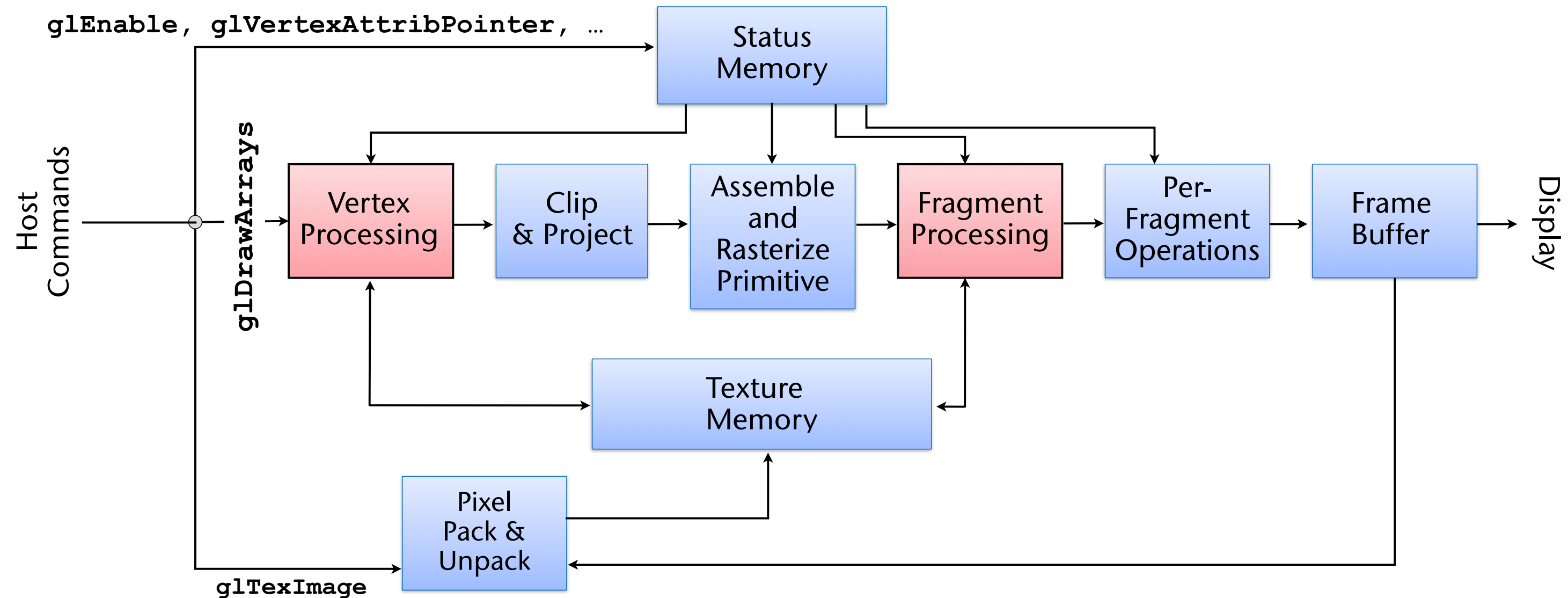


Erinnerung: die Graphik-Pipeline früher und heute

- *Fixed-function graphics pipeline*
- Philosophie: Performance ist wichtiger als Flexibilität (sorgfältig ausbalanciert)



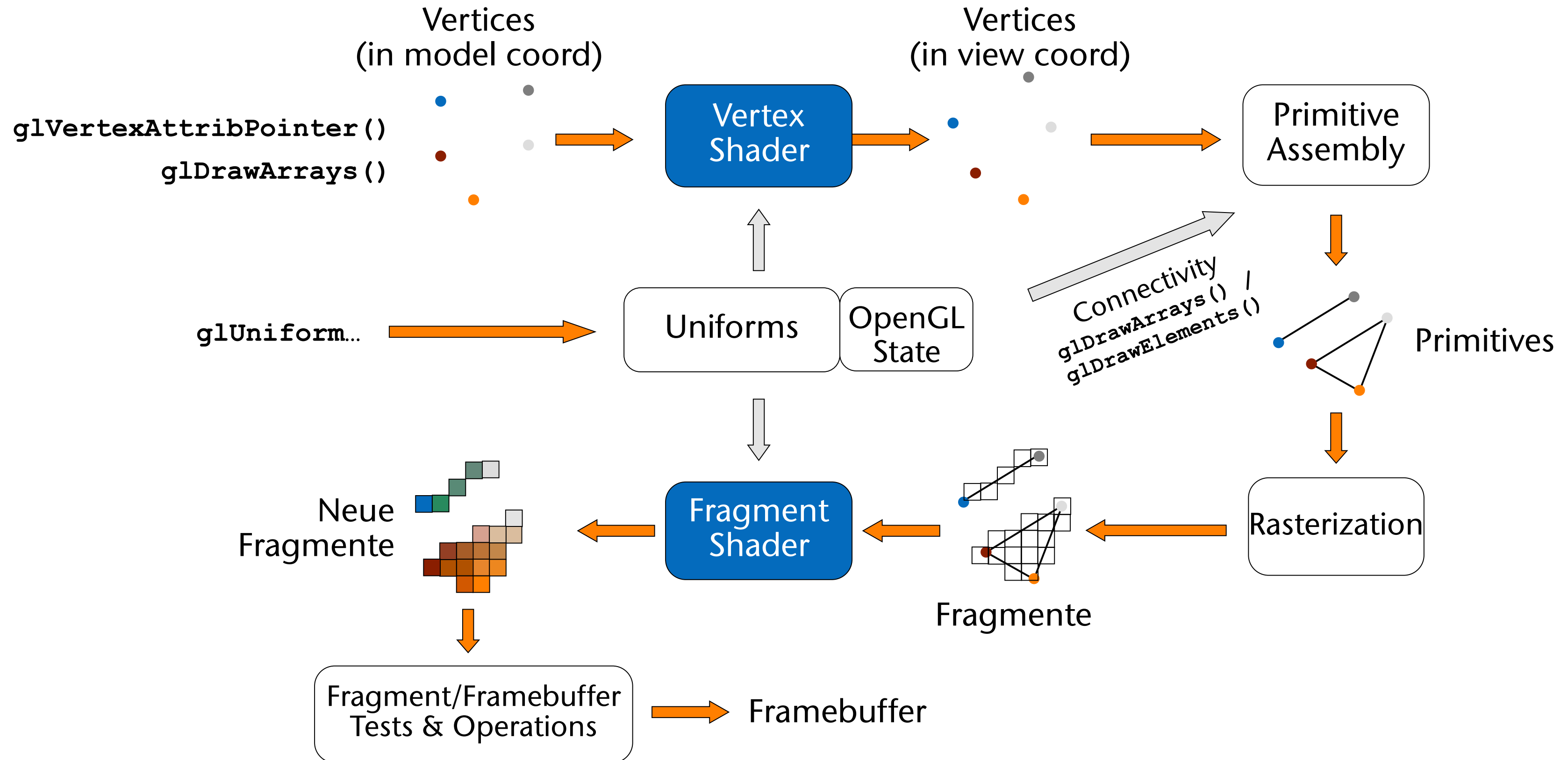
- Heute: programmierbare *vertex und fragment processors*
- Texturspeicher = allgemeiner Speicher für beliebige Daten
- Balancierung der Pipeline ist jetzt Programmierer's Aufgabe



Kommunikation mit OpenGL bzw. der Applikation

- Es gibt *keine vordefinierten* Normalen, Farben, Koordinaten mehr — nur noch **Vertex-Attribute**
- Es gibt *keine* vordefinierten Lichtquellen-Attribute mehr
- Zur Erinnerung: man kann Variablen deklarieren, die von außen (= OpenGL-Programm) gesetzt werden können:
 - Sog. **uniform**–Variablen: können sowohl von Vertex- als auch Fragment-Shader gelesen werden (aber nicht geschrieben)
 - Sog. **attribute**–Variablen: werden entlang der Pipeline durchgereicht (App. → OpenGL → Vertex-Shader → Fragment-Shader → Framebuffer)
- Im Textur-Speicher können beliebige Daten an Shader übergeben werden
 - Interpretation bleibt Shader überlassen

Abstraktere Übersicht der programmierbaren Pipeline



Hilfsvorstellung zum Execution Model einer GPU

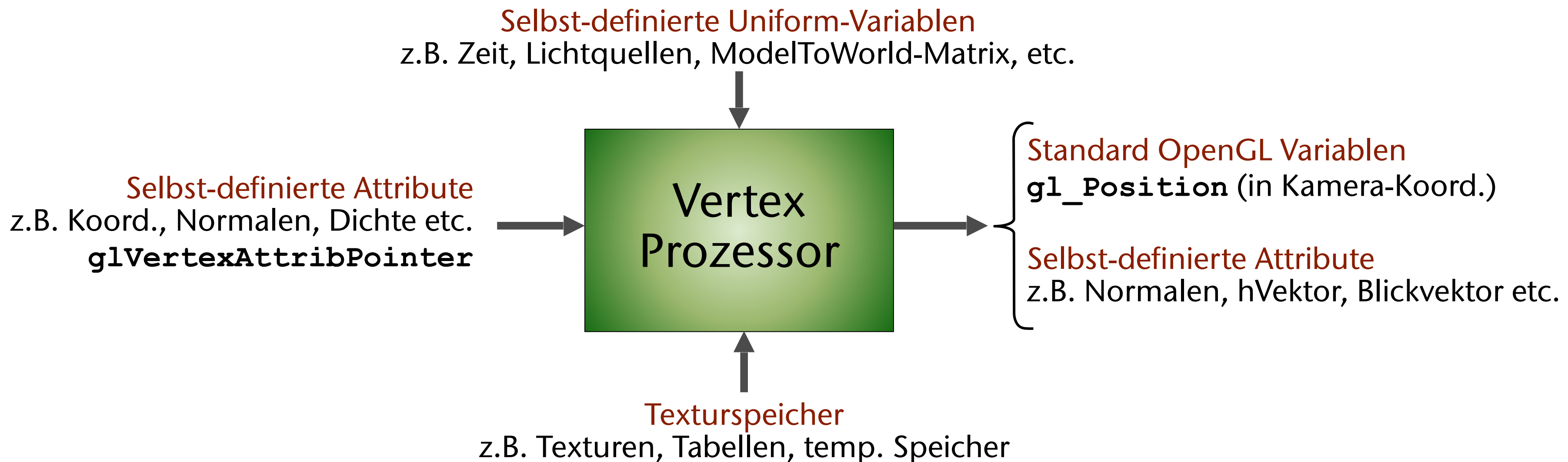
```
foreach tri in triangles:
    // run the vertex program on each vertex
    v1 = process_vertex( tri.vertex[0] )
    v2 = process_vertex( tri.vertex[1] )
    v3 = process_vertex( tri.vertex[2] )
    // assemble the vertices into a triangle
    assembledtriangle = setup_triangle( v1, v2, v3 )
    // 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

- Erinnerung: 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 (im Framebuffer)
 - Ein Fragment ist ein "Struct", das durch die Pipeline "wandert" und am Ende in
ein Pixel gespeichert wird

Input & Output eines Vertex-Shaders

- **Vertex Shader** (= Programm) bekommt eine Reihe von Parametern:
 - Selbst-definierte Attribute, einen Satz pro Vertex aus dem VAO
- Resultat muß in **out**-Variablen geschrieben werden, die der Rasterizer dann ausliest und interpoliert (nur eine vordefinierte)

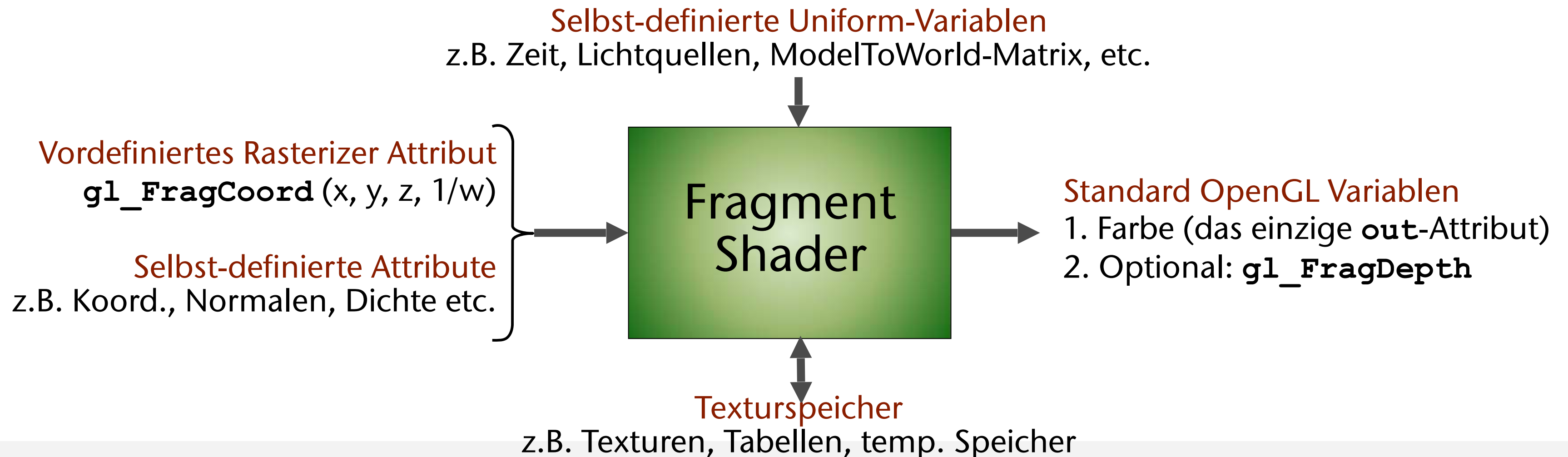


Aufgaben des Vertex-Shaders

- Beleuchtung und Vertex-Attribute pro Vertex berechnen
- Ein Vertex-Programm ersetzt folgende Funktionalität der alten fixed-function Pipeline:
 - Vertex- & Normalen-Transformation ins Kamera-Koord.-System
 - Transformation mit Projektionsmatrix (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.)

Input & Output eines Fragment-Shaders

- **Fragment shader** bekommt eine Reihe von Parametern:
 - Selbst-definierte Attribute, Uniforms
 - Fragment-Attribute = alle Ausgaben des Vertex-Shaders, aber **interpoliert!**
 - Ausnahme: Qualifier **flat**, z.B. (**flat out vec3 normal**) → keine Interpolation
- Resultat: neues Fragment (i.A. mit anderer Farbe als vorher)



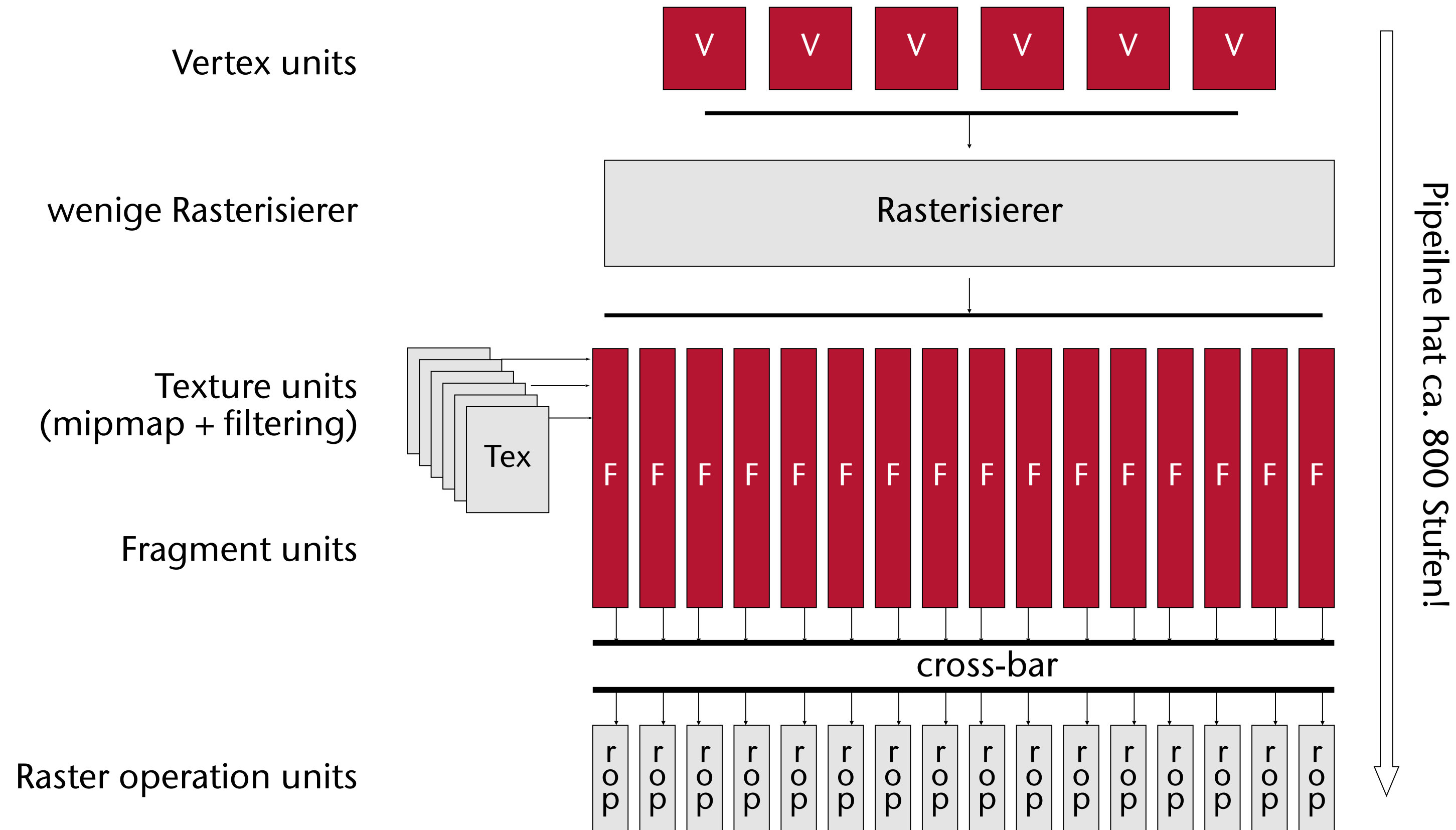
Aufgaben des Fragment-Shaders

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

Was ein Shader **nicht** kann

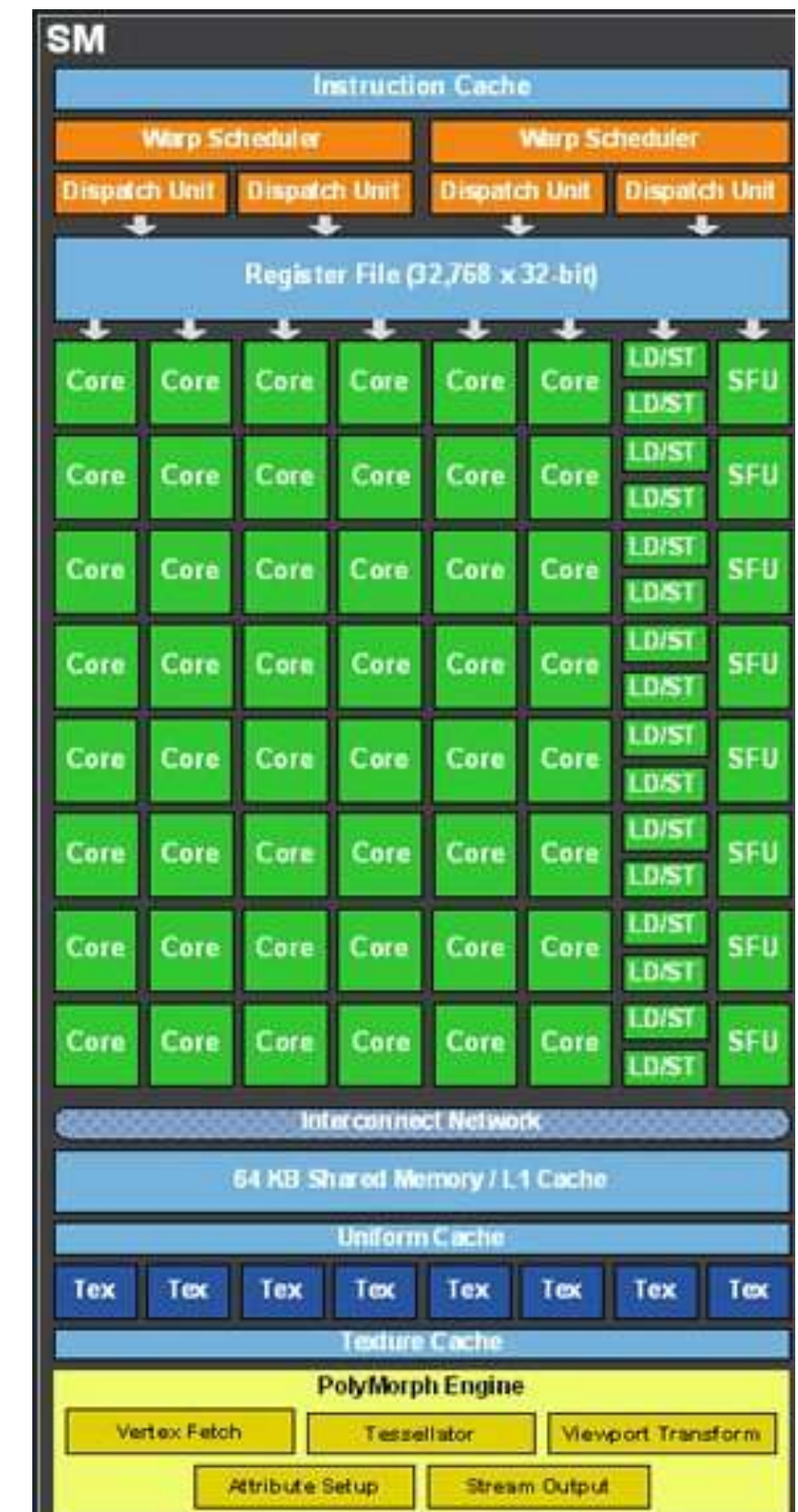
- Ein Vertex-Shader hat *keinen* Zugriff auf Connectivity-Info und Framebuffer
- Ein Fragment-Shader
 - hat *keinen* Zugriff auf danebenliegende Fragmente
 - hat *keinen* Lese-Zugriff auf den Framebuffer
 - kann *nicht* die Pixel-Koordinaten wechseln (aber kann auf sie zugreifen)

Etwas detailliertere Architektur



Streaming Multiprocessors

- Keine Unterscheidung mehr zwischen Vertex- und Fragment-Prozessoren, sondern einheitliche 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
 - Dekodiert Instruktionen
 - Enthält Caches
- Eine GPU hat mehrere SM's



Shader-Assembler-Code versus Shader-Highlevel-Code

Assembly

```

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

```

Hochsprache

```

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

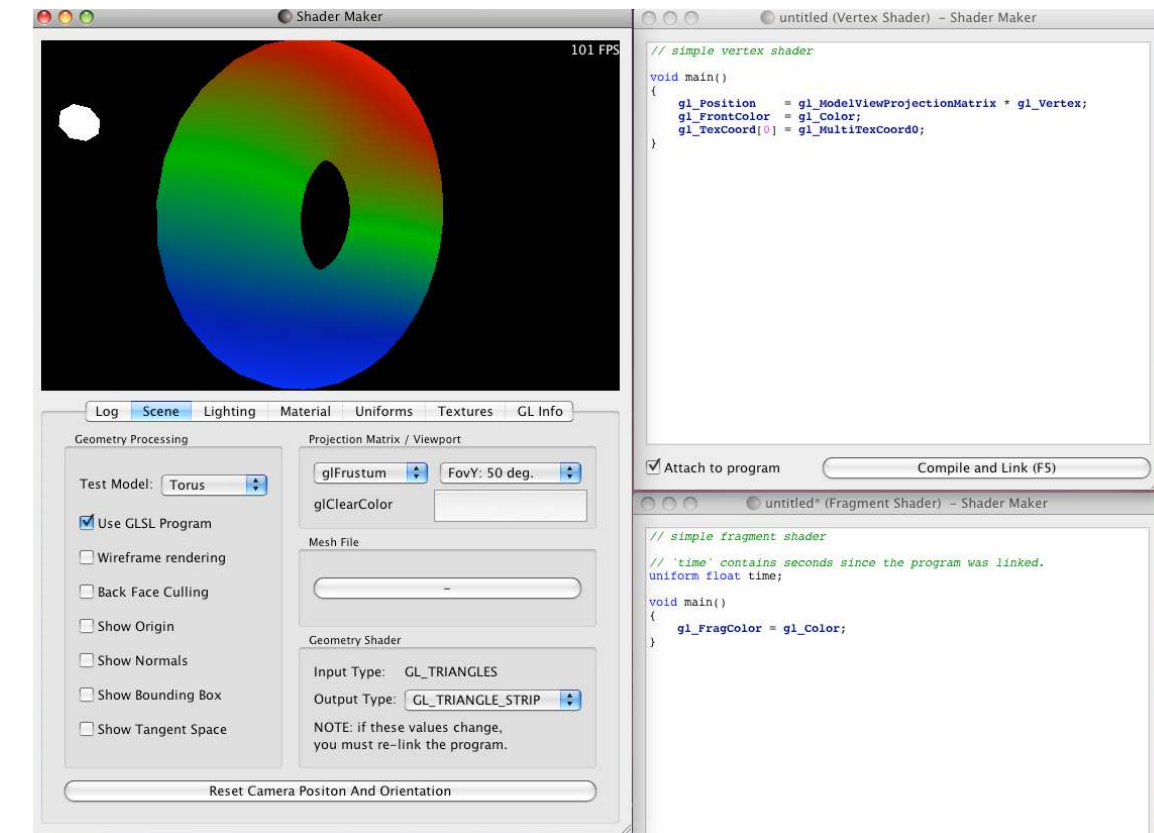
```

GPU-Hochsprachen

- GLSL ("*glslang*"; OpenGL Shading Language)
- HLSL (Microsoft)
- RenderMan (Pixar)
- MetaSL (mental ray)
- ...
-

Shader-Editoren (Shader-IDE's)

- Shader Maker (Neuaufgabe):
 - Cross-platform
 - OpenGL 4+
 - cgvr.informatik.uni-bremen.de/research/shader_maker/
- SHADERed: mit Debugger
 - shadered.org bzw. github.com/dfranx/SHADERed
 - Needs mouse with 3 buttons, unusual GUI
- ShaderFrog: runs in the Browser
 - Old GLSL syntax ("varying") only
- ShaderLabFramework (by Imperial College, London):
 - Offers 2-pass rendering
 - Binaries at <http://wp.doc.ic.ac.uk/bkainz/teaching/co317-computer-graphics/>
 - Source on Github: <https://github.com/bkainz/ShaderLabFramework>
- Die meisten Modeller (3dsMax, Blender, etc.)
 - Manche nur HLSL, manche nur eine Meta-Sprache
- KodeLife (Mac, Linux, Windows): live editing; keine Doku
- Im Browser (meist nur Fragment-Shader):
 - BKcore's Shdr: <http://shdr.bkcore.com> (no interactive uniforms)
 - WebGL playground: <http://webglplayground.net> (inkl. Javascript host side progr.)

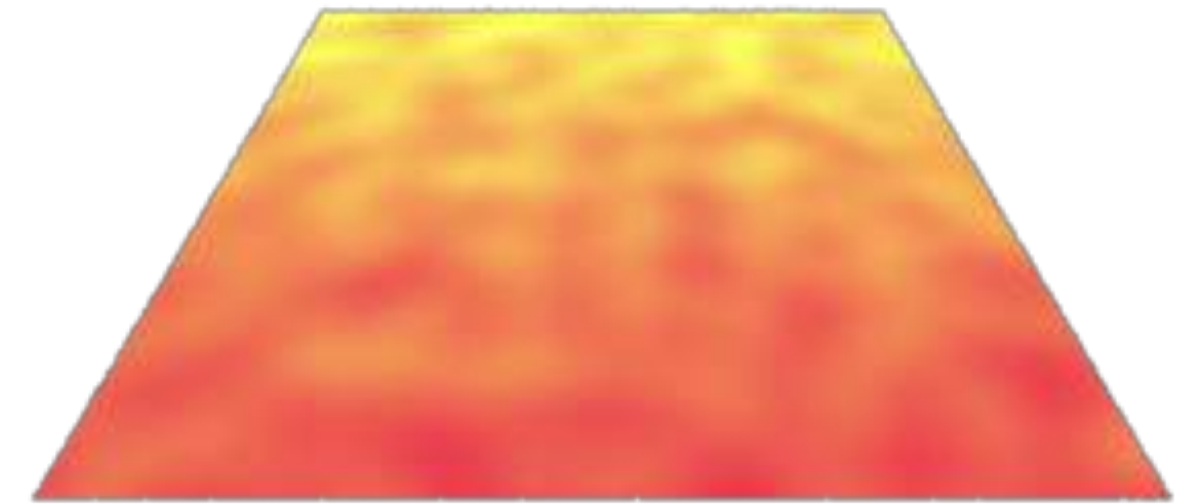


Debugging ...

- Nsight: Plugin für Visual Studio; und als eigene Eclipse-Version unter Linux (in CUDA enthalten)
- "Printf-Debugging" geht nicht
- Meine Tips:
 - Von einem funktionierenden Shader ausgehen und diesen in winzigen Schritten (einzelne Zeilen) modifizieren bzw. weiterentwickeln
 - Bei Aufgaben, wo mehrere Render-Passes gemacht werden müssen: nach jedem Pass die Textur / den Framebuffer anzeigen

Historischer Exkurs: RenderMan

- Entwickelt von Pixar in 1988
- Ist heute ein Industriestandard
- Eng an das Ray-Tracing-Paradigma angelehnt
- Mehrere Shader-Arten, je eine für Lichtquellen, Oberflächen, Volumen, Displacements, ...



```
surface
noise_test1(float      Kfb = 1,
             amp = 0,    /* amplitude of the noise */
             freq = 4;   /* frequency of the noise */
             color      top = 1,
             lower = 0)
{
    // Noise values range from 0 to 1.
    float    ns = noise(s * freq, t * freq);

    // Offset the true value of 't'. The 'amp' parameter will allow
    // the artist to strengthen or weaken the visual effect.
    float     tt = t + ns * amp;

    color     surfcolor = mix(top, lower, tt);
    Oi = Os;
    Ci = Oi * Cs * surfcolor * Kfb;
}
```

Einführung in GLSL

- Fester Bestandteil seit 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

Datentypen: die üblichen Verdächtigen, plus domänenspezifische

- `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-Deklarationen wie in C
- **Es gibt keine Pointer!**

Qualifier (Variablen-Arten)

- **const**
- **uniform:**
 - globale Variable, im Vertex- und Fragment-Shader, gleicher Wert in beiden Shadern, konstant während eines gesamten Primitives
- **in / out / inout**
 - Zur Deklaration von Vertex-Attributen
 - Können von Host-Seite nur für Vertex-Shader per **glVertexAttribPointer()** zugewiesen werden.
 - Werden vom Rasterizer interpoliert, ..
 - und vom Fragment-Shader gelesen (pro Fragment)

Operatoren

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

Skalar/Vektor Constructors

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

```
vec3 red = vec3(1.0, 0.0, 0.0);
vec3 green = vec3(0.0, 1.0, 0.0);
vec3 yellow = red + green;
vec3 white = vec3(1.0);           // all components = 1.0
vec4 v4 = vec4(red, 1.0);         // # components must match
vec2 v2 = vec2( v4 );             // throws away 2 components

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

Matrix Constructors

```

vec4 v4;
mat4 m4;
m4 = mat4( 1.0, 2.0, 3.0, 4.0, // COLUMN MAJOR
           5.0, 6.0, 7.0, 8.0, // memory layout!
           9.0, 10., 11., 12.,
           13., 14., 15., 16.);
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.);

mat4( c1, c2, c3, c4 ) // wird spaltenweise eingetragen

mat4( 1.0 ) // = identity matrix
mat3( m4 ) // upper 3x3
vec4( m4 ) // 1st column
float( m4 ) // upper left
    
```

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

Zugriff auf Komponenten von Vektoren und Matrizen

- 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
```

Flow-Control: 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
 - Beispiel:

```
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 ); }
```

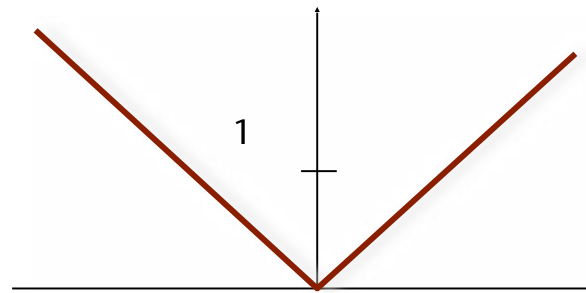
Echten Code so
bitte nicht
formatieren!
Dies ist hier nur
dem (fehlenden)
Platz geschuldet!

Eingebaute Funktionen

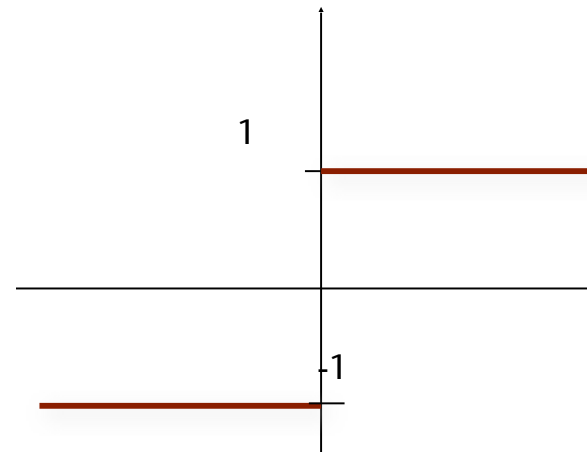
- Trigonometrie: `sin`, `asin`, `radians`, ...
- Exponentialfunktionen: `pow`, `exp`, `log`, `sqrt`, ...
- Sonstige: `abs`, `clamp`, `max`, `sign`, ...
- Alle o.g. Funktionen nehmen und liefern `float`, `vec2`, `vec3`, oder `vec4`, und arbeiten komponentenweise!
- Geometrische Funktionen: `cross(vec3,vec3)`,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

Einige häufige Funktionen

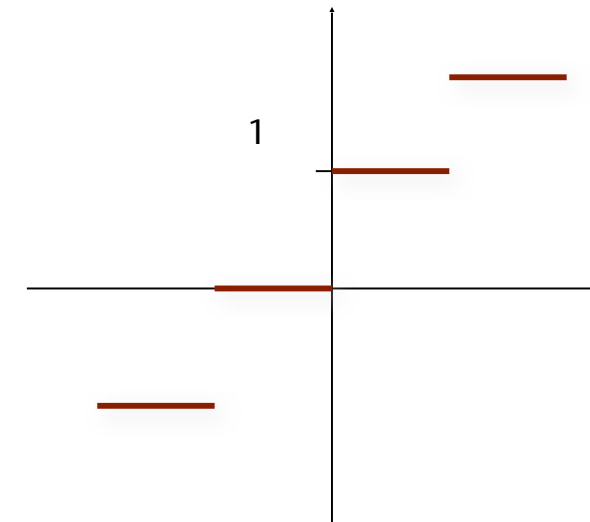
$\text{abs}(x)$



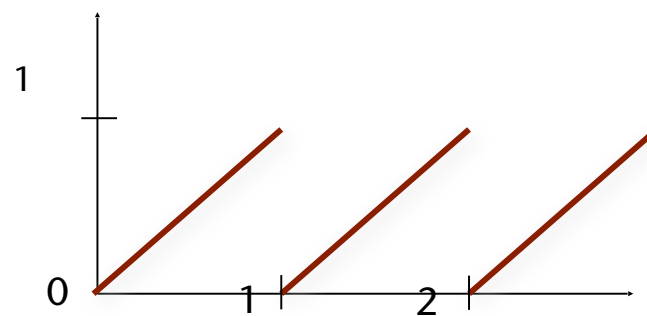
$\text{sign}(x)$



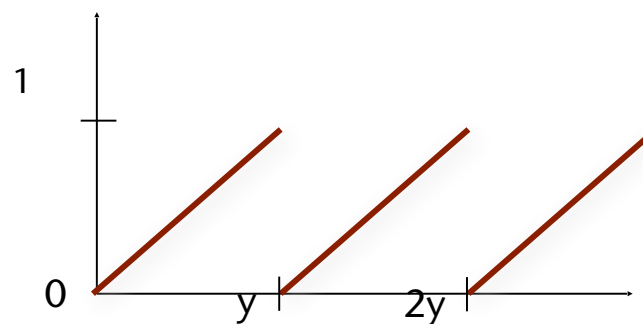
$\text{ceil}(x)$



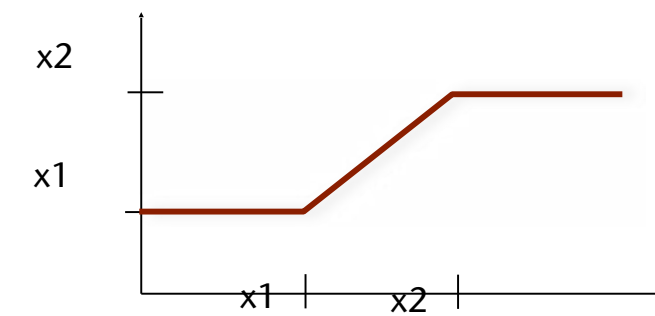
$\text{fract}(x)$



$\text{mod}(x, y)$

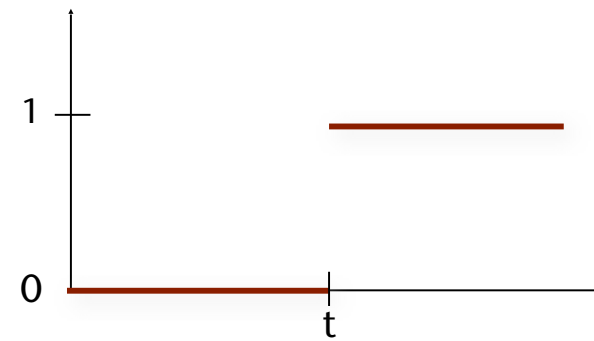


$\text{clamp}(x, x1, x2)$



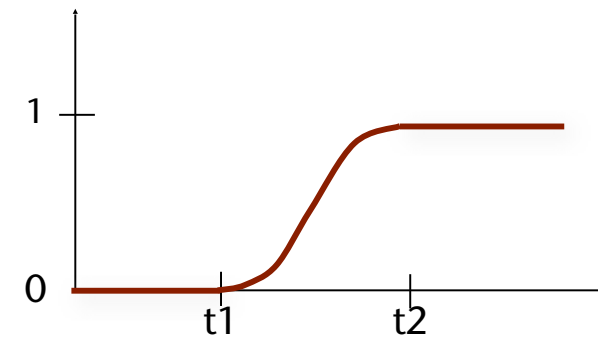
Zur Erinnerung: alle Funktionen arbeiten (komponentenweise) auf **float** ... **vec4** !

step(t, x)



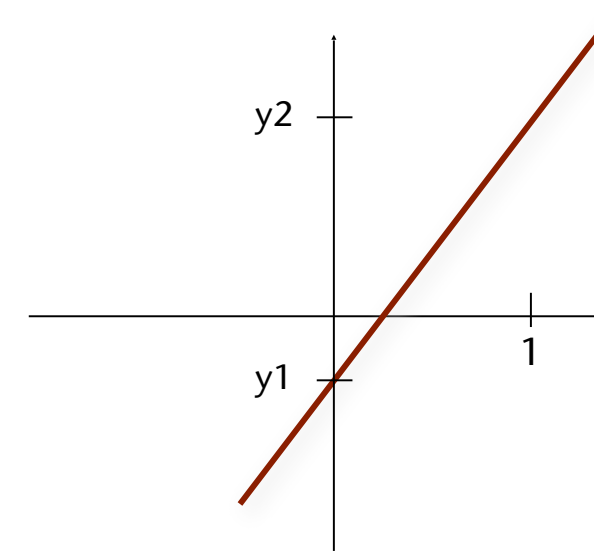
```
step(t, x) :=  
x <= t ? 0.0 : 1.0
```

smoothstep(t1, t2, x)



```
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)

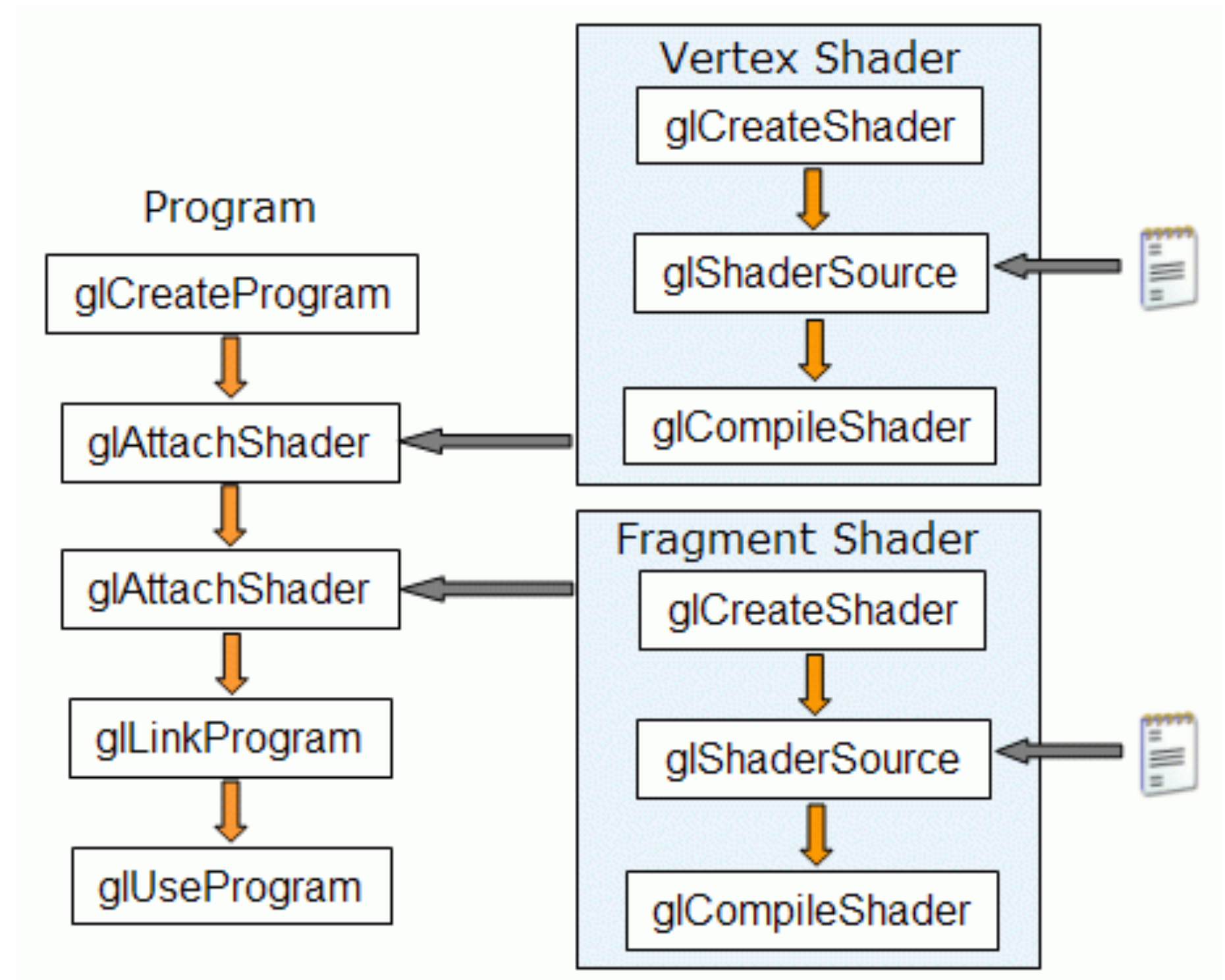


```
mix(y1, y2, t) :=  
y1 * (1.0 - t) + y2 * t
```

Laden eines Shaders

FYI (nicht Klausur-Relevant)

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



Im Detail

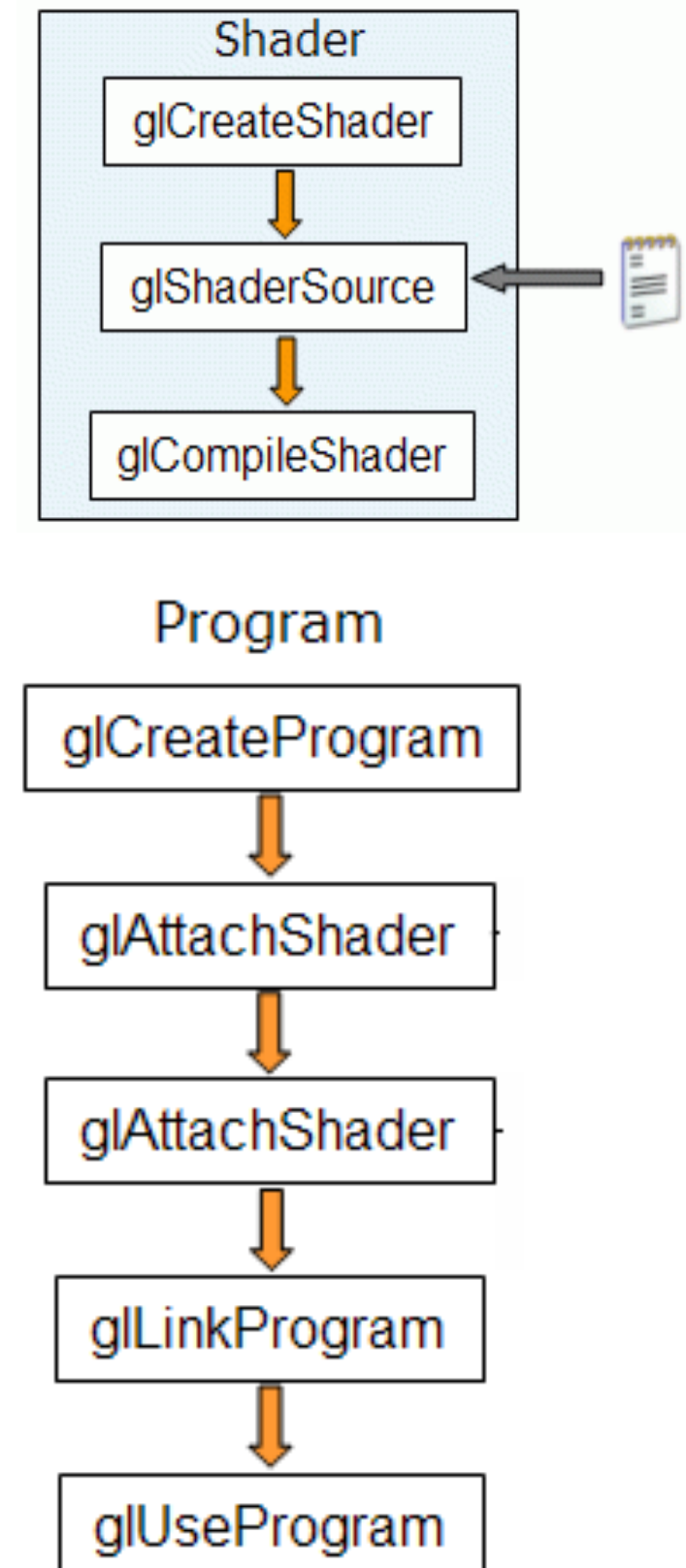
FYI (nicht Klausur-Relevant)

```
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
...

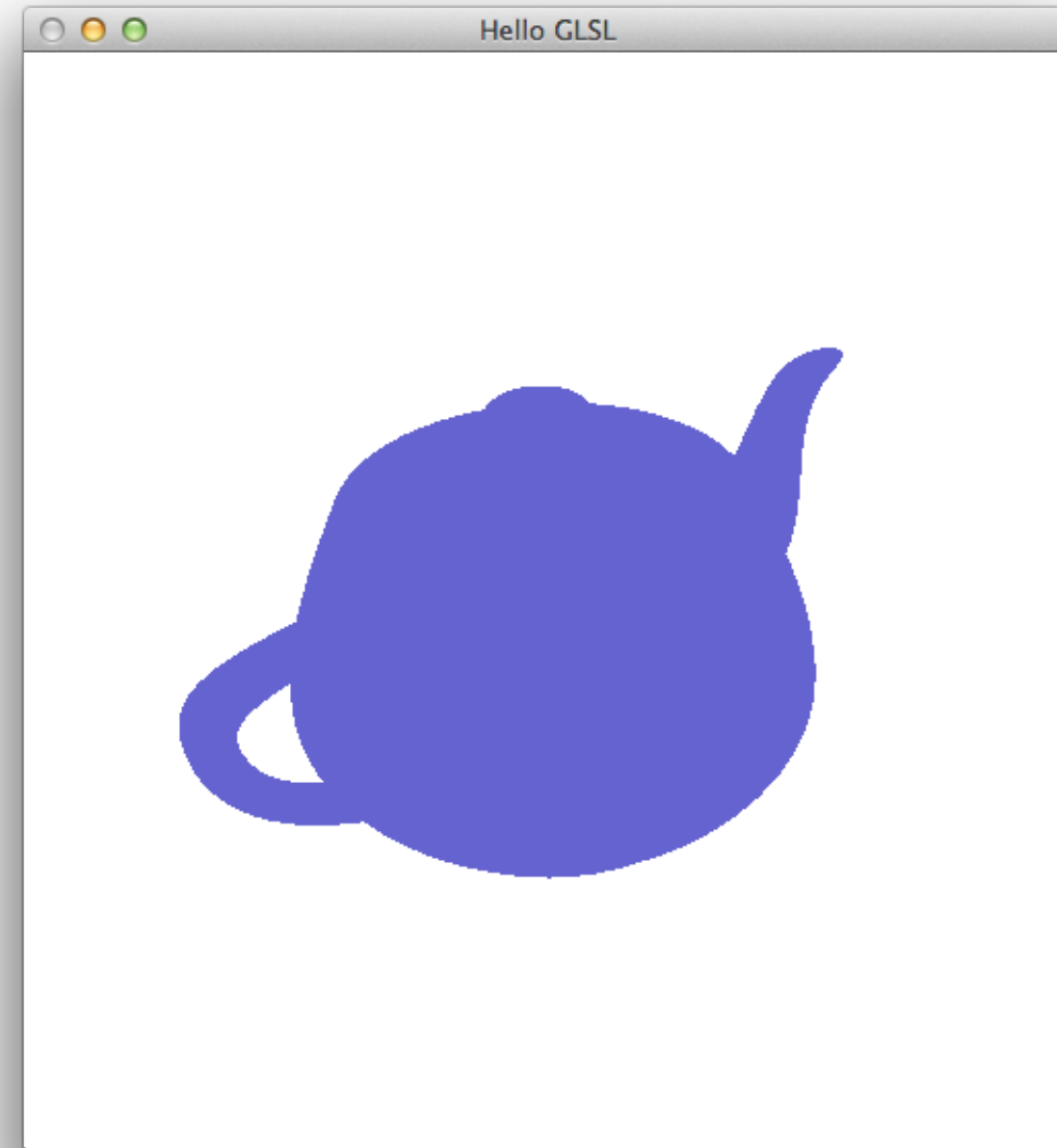
uint progr_handle = glCreateProgram();
glAttachShader( progr_handle, vert_sh_handle );
glAttachShader( progr_handle, frag_sh_handle );

glLinkProgram( progr_handle );
glUseProgram( progr_handle );
```



- 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*
 - *Nur im Compatibility Mode*
- Man kann einen Shader zu mehreren verschiedenen Programmen attachen

Beispiel: Hello_GLSL



hello_glsl*

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 für ein Attribut
- Uniform-Variablen:
 - **glGetActiveUniform()** : liefert Info zu einem Parameter
- Benötigt man vor allem zur Implementierung von sog. Shader-Editoren

Setzen von *uniform*-Variablen (auf Host-Seite)

- Handle auf Variable besorgen:

```
uint var_handle = glGetUniformLocation( progr_handle, "uniform_name" )
```

- Setzen einer uniform-Variable:

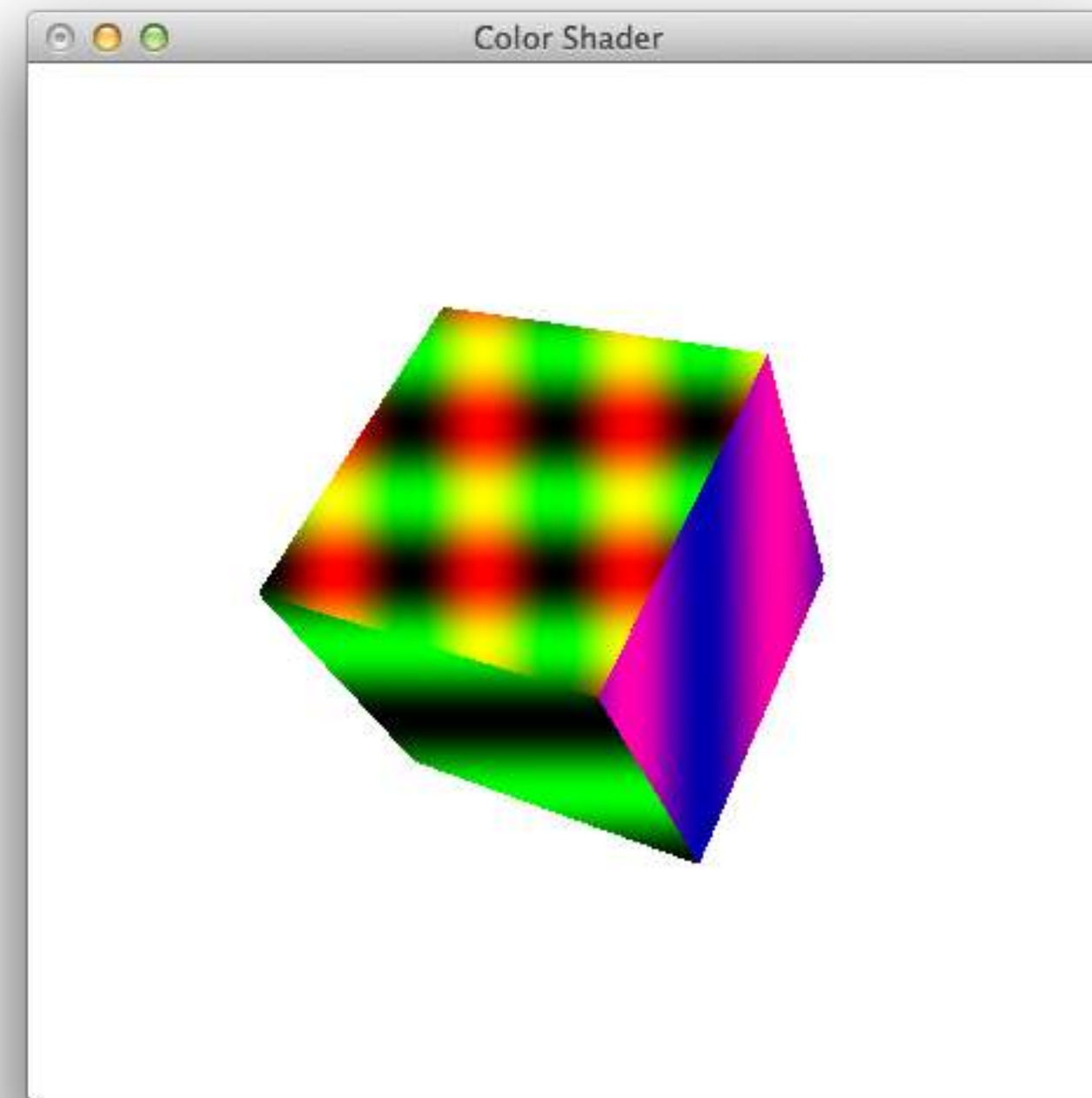
- Für Float:

```
glUniform1f( var_handle, f )
```

- Für Matrizen:

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

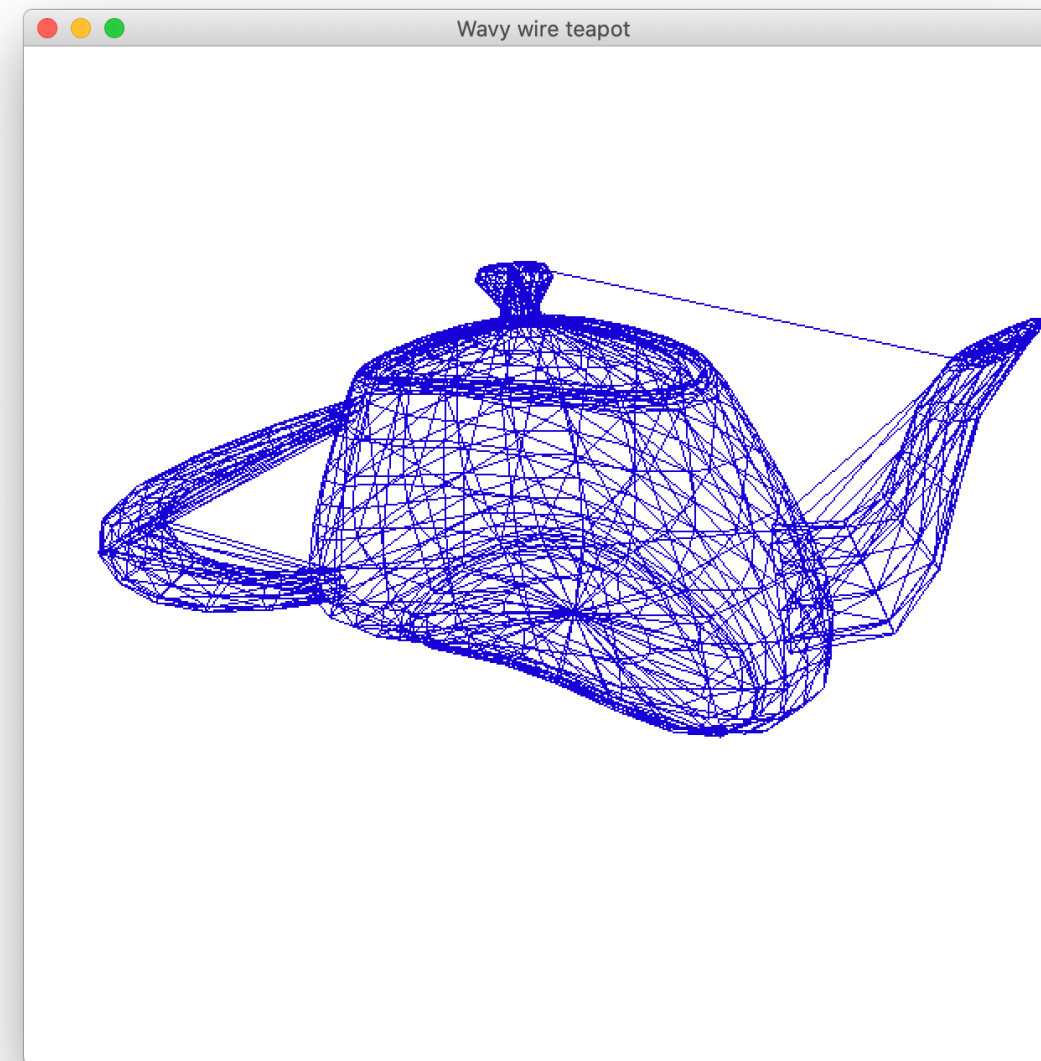
Beispiel für *uniform*-Variable



```
color. {vert, frag}
```

Beispiel für die Modifikation von Geometrie im Shader

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



Wavy wire teapot

Attribute als Input zum Vertex-Shader

- Man **muss** eigene Attribute definieren (mindestens eines):
 - Im Vertex-Shader: z.B. `in vec3 vertexPos;`
 - Im C-Programm :

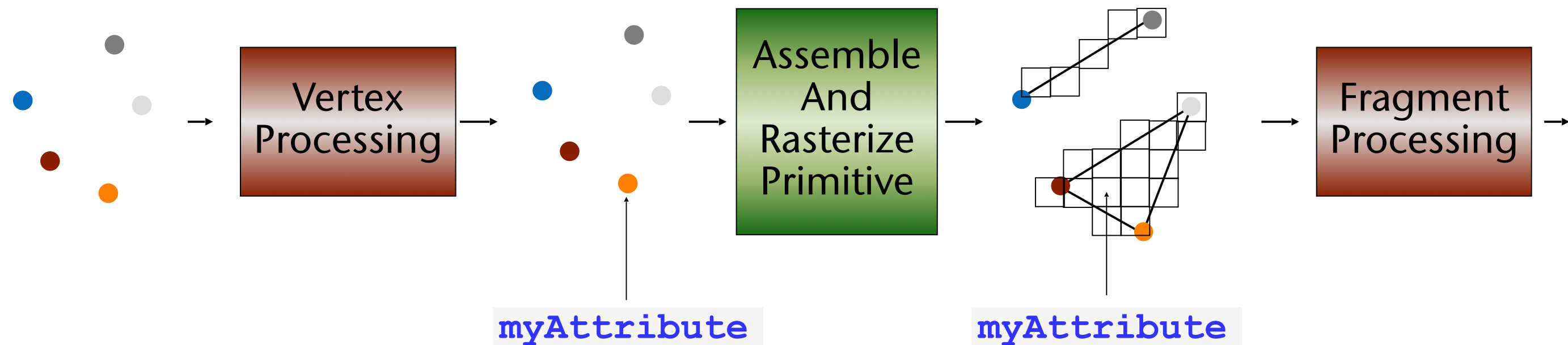
```
glBindVertexArray( m_vao );           // activate VAO
glBindBuffer( GL_ARRAY_BUFFER, m_data );
glBufferData( GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW );
handle = glGetAttribLocation( prog_handle, "vertexPos" );
glEnableVertexAttribArray( handle );
glVertexAttribPointer( handle,         // handle ID in shader
                      3,               // size (here: 3 floats)
                      GL_FLOAT,        // type: float
                      GL_FALSE,        // normalized?
                      0,               // stride
                      (void*)0 );      // offset in array
```

Parameter-Übergabe von Vertex- zu Fragment-Shader

- Mittels Attributen:

```
out vec3 myAttribute; // in vertex shader
...
in vec3 myAttribute; // in fragment shader
```

- Achtung, dazwischen sitzt der Rasterizer und interpoliert!
 - I.A. sinnvoll, z.B. für Position, Farbe, Normale, etc.
 - Mittels Qualifier **flat** vermeidbar



Beispiele für die Verwendung von Attribut-Variablen

- Der "Toon-Shader"
 - Setze die Helligkeit nur abhängig von

$$a = \mathbf{l} \cdot \mathbf{n}$$

mit $\mathbf{l}$ = Lichtvektor, $\mathbf{n}$ = Normale

- Diskretisiere die Helligkeit in (z.B.) 3 Stufen:

$$C_{\text{out}} = C_{\text{in}} \cdot \begin{cases} a_1, & a \geq a_1 \\ a_2, & a \geq a_2 \\ a_3, & a \geq a_3 \\ a_4, & \text{else} \end{cases}$$

wobei $a_i \in [0,1]$ die Levels sind, C_{in} die Farbe der Oberfläche ist

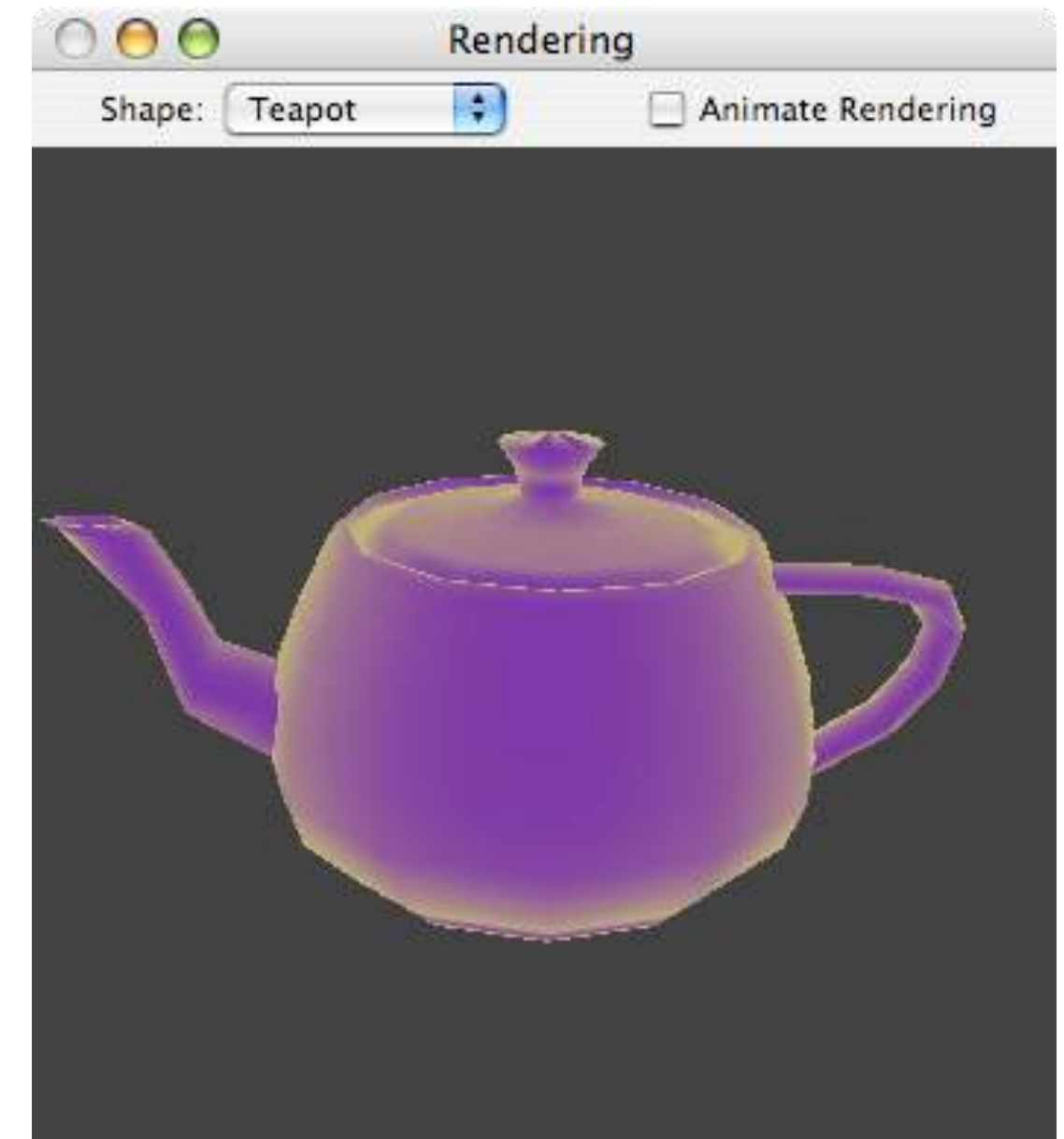


- Der "Gooch-Shader"
 - Interpoliert zwischen 2 Farben, abhängig vom Winkel zwischen Normale und Lichtvektor:

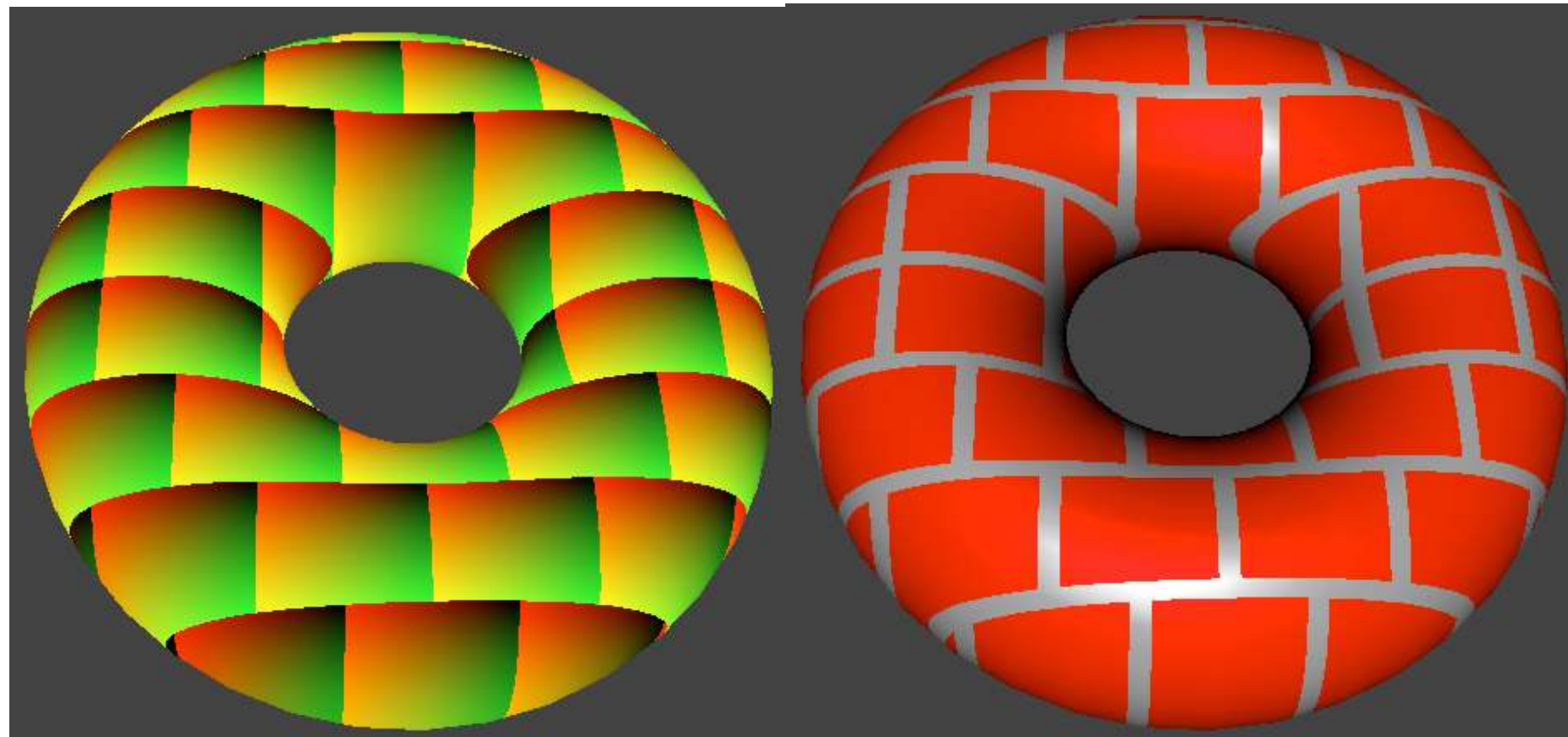
$$t = \max(0, \mathbf{n} \cdot \mathbf{l}) \quad t_{\text{alt}} = \frac{1 + \mathbf{n} \cdot \mathbf{l}}{2}$$

$$C = t C_{\text{warm}} + (1 - t) C_{\text{cool}}$$

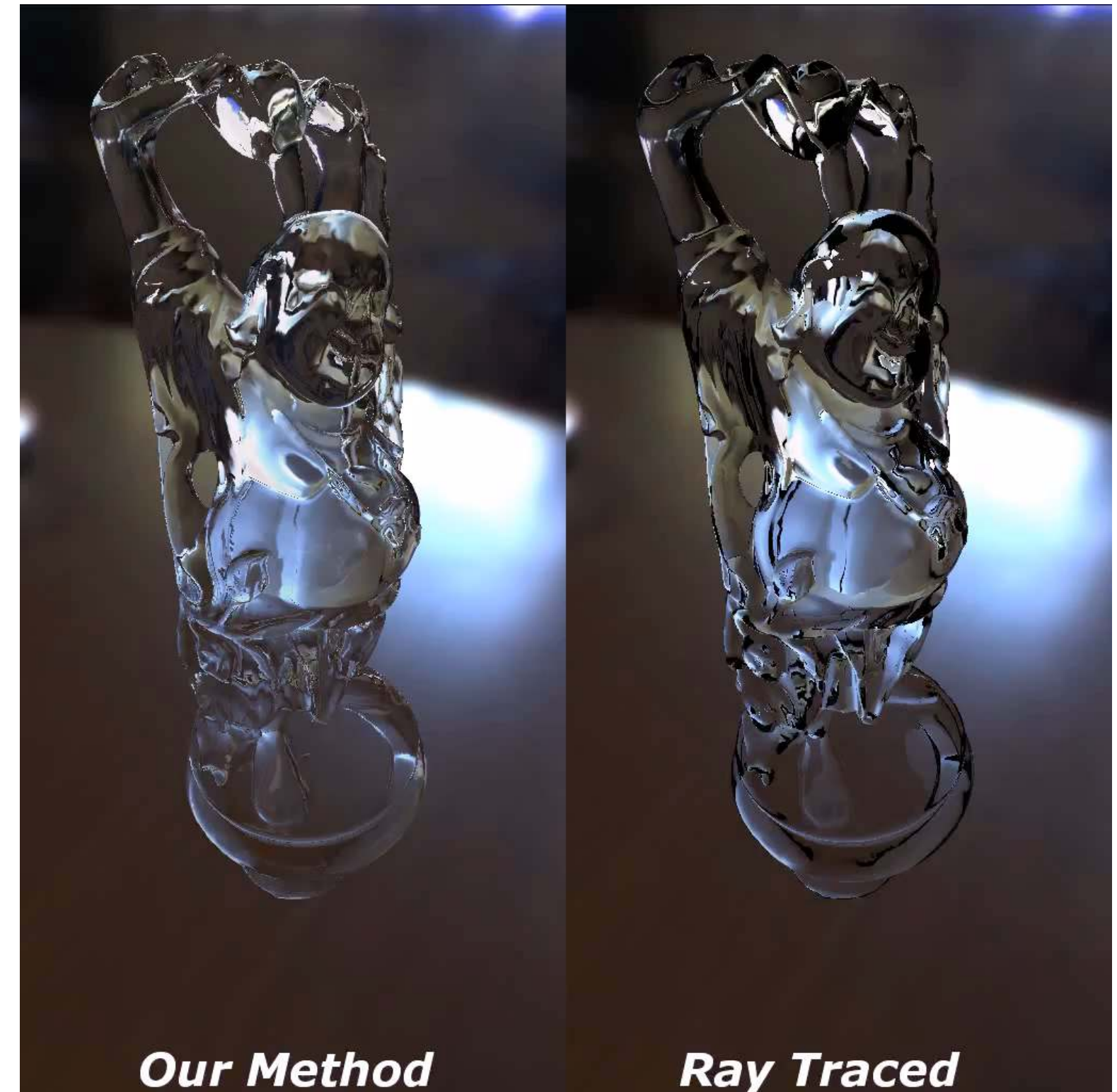
- Dies sind schon einfache Beispiele für sogenanntes "*non-photorealistic rendering*" (NPR)



Ausblick (in Advanced Computer Graphics)



Prozedurale Texturen



Our Method

Ray Traced

Lichtbrechung (1 bounce)