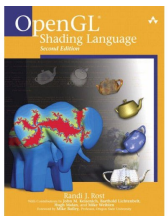
Computer-Graphik I
Shader-Programmierung

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

Literatur

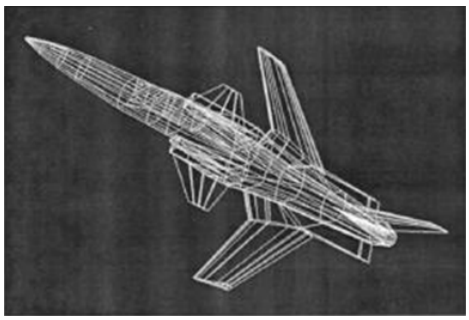
- Das "Orange Book":
  - Randi J. Rost, et al.:
    "OpenGL Shading Language",
    2nd edition, Addison Wesley.



- Auf der Homepage der Vorlesung:
  - Das Tutorial von Lighthouse3D
  - Mark Olano's *"Brief OpenGL Shading Tutorial"*
  - Der "GLSL Quick Reference Guide"
  - ...

The Quest for Realism

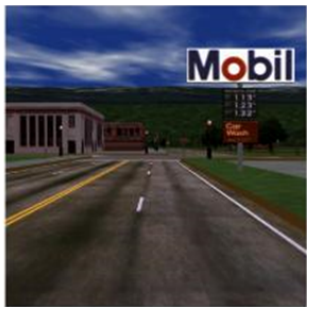
- Erste Generation – Wireframe
  - Vertex-Oper.: Transformation, Clipping und Projektion
  - Rasterization: Color Interpolation (Punkte, Linien)
  - Fragment-Op.: Overwrite
  - Zeitraum: bis 1987



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

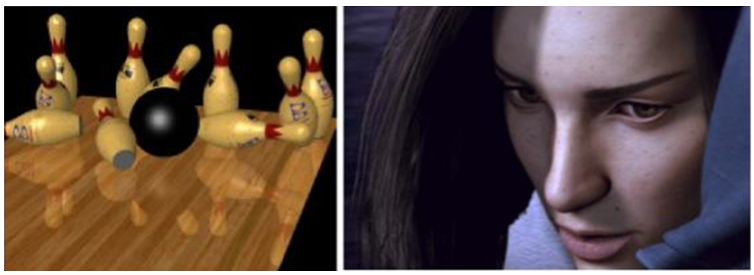
(Dogfight - SGI)

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



Performertown (SGI)

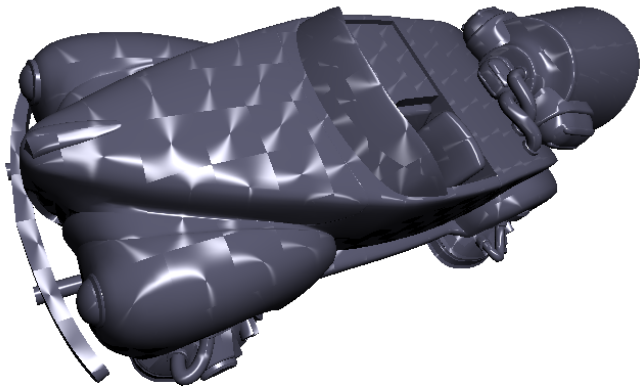
- Vierte Generation – Programmierbarkeit
  - Vertex-Oper.: eigenes Programm
  - Rasterization: Interpolation der (beliebigen) Ausgaben des Vertex-Programms
  - Fragment: eigenes Programm
  - Zeitraum-Oper.: ab 2000



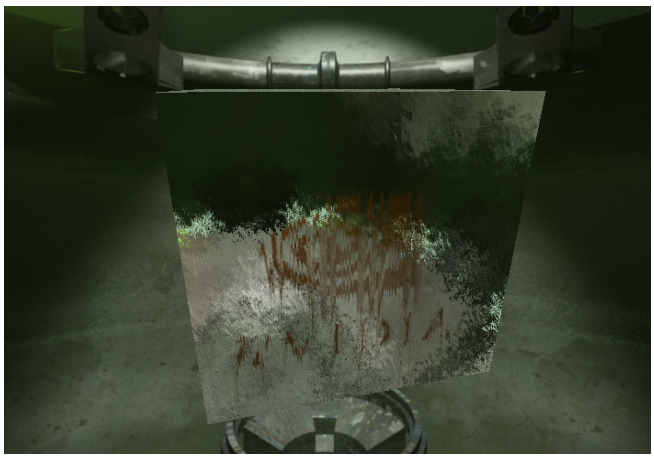
Final Fantasy

Beispiele

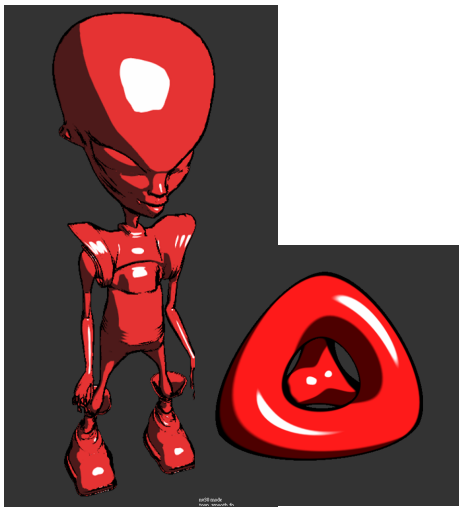
- Brushed Steel:
  - Prozedurale Textur
  - Anisotrope Beleuchtung



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

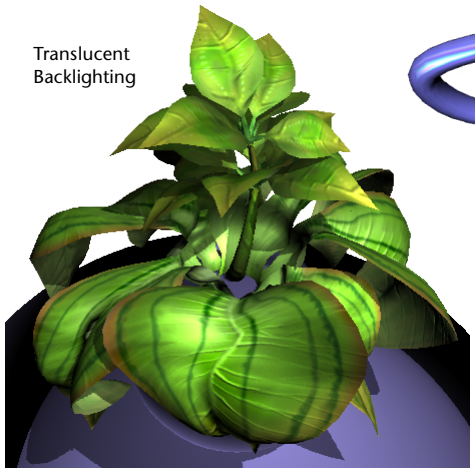


- Sog. „Toon Shading“
  - Ohne Texturen
  - Mit Anti-Aliasing
  - Gute Silhouetten ohne zu starker Verdunkelung



- Vegetation & *Thin Film*

Translucent Backlighting



Beispiel von selbstgemachter Beleuchtungsrechnung; hier: Simulation von Schillern

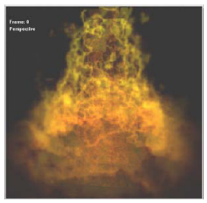
Animusic's Pipe Dream

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

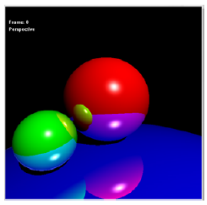
http://ati.amd.com/developer/demos.html

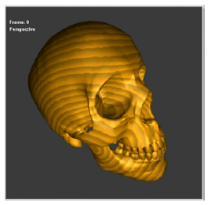

Subsurface Scattering


NPR Renders


Fire Effects


Refraction


Ray Tracing

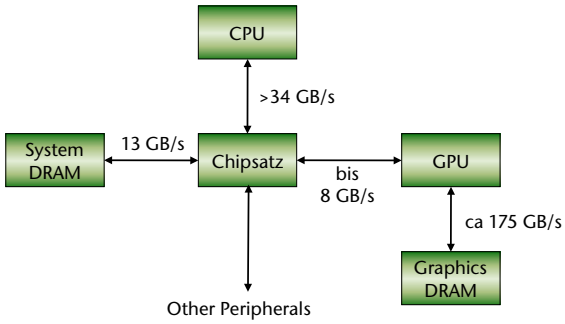

Solid Textures


Ambient Occlusion

Cloth Simulation

Einbettung der Graphik-Hardware in die System-Architektur

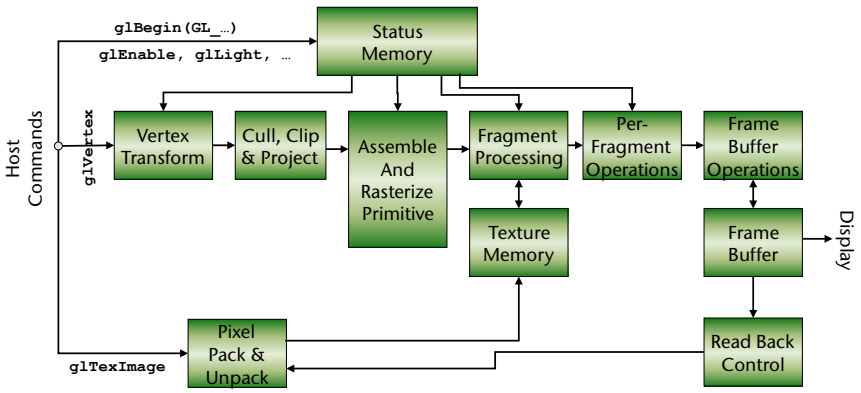
- Dedicated bus zwischen “host” CPU und GPU (*graphics processing unit* = Graphikkarte) (AGP, PCIExpress)
- Separater Speicher für die GPU (framebuffer, textures, etc.)
- GPU hat DMA zum Haupspeicher



Stand 2010

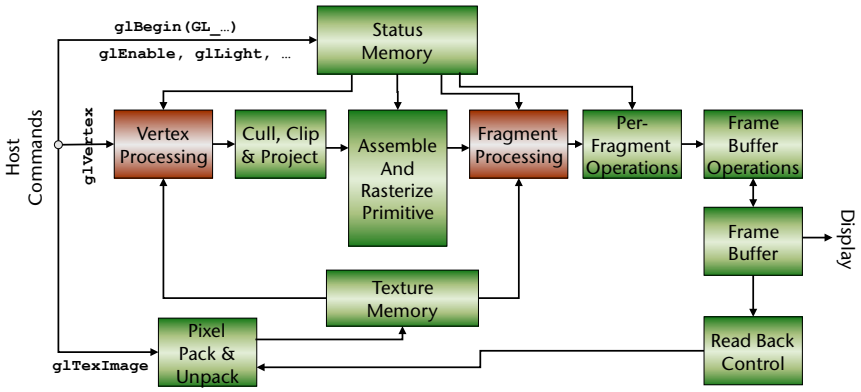
Vergangenheit – OpenGL 1.4

- *Fixed-function graphics pipeline*
  - Sehr sorgfältig ausbalanciert
- Philosophie: Performance wichtiger als Flexibilität

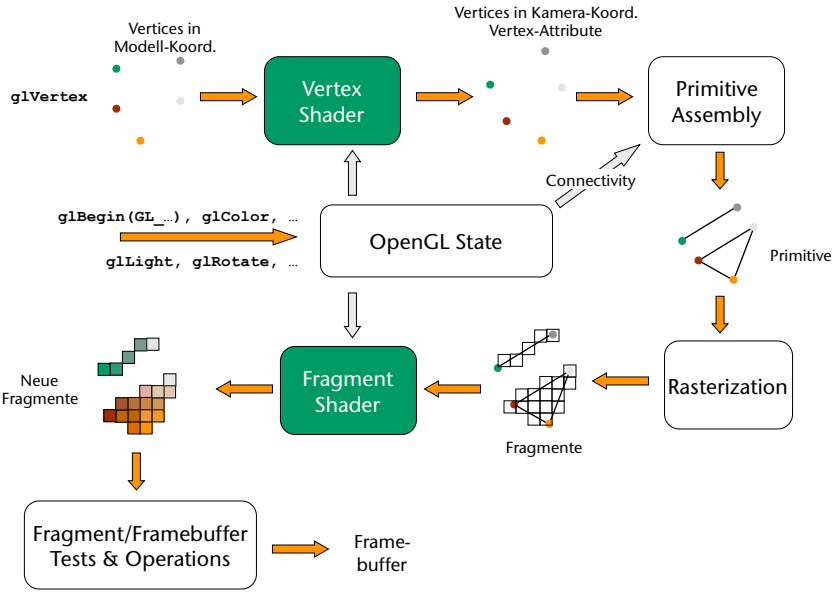


Heute – OpenGL 2.1

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



Abstraktere Übersicht der programmierbaren Pipeline



- 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 );

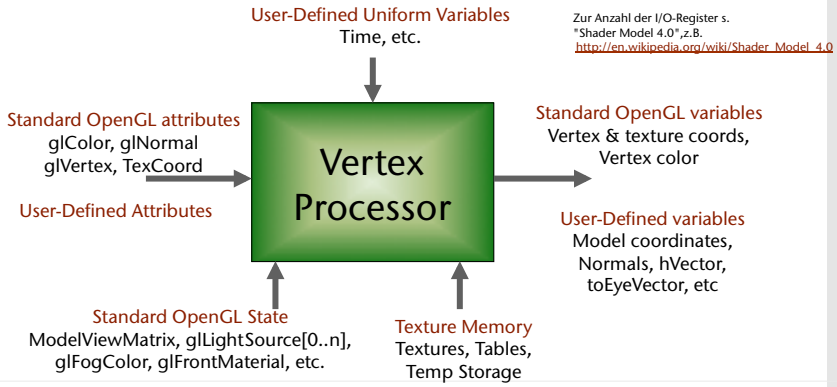
    // run the fragment program on each fragment
    foreach frag in fragments {
        framebuffer[frag.position] = process_fragment( frag );
    }
}
...
```

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

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

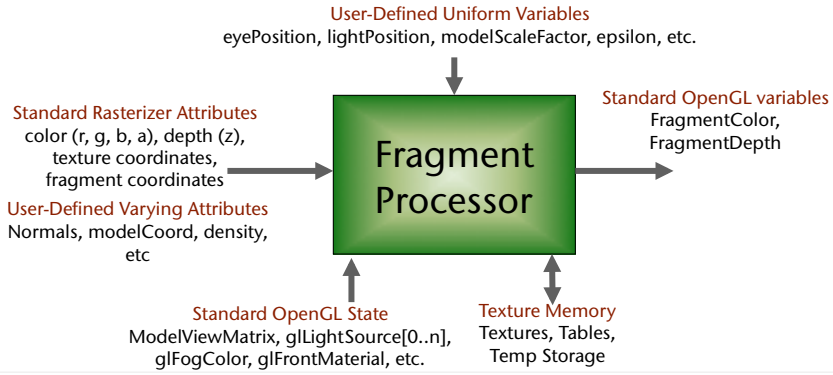


Aufgaben des Vertex-Prozessors

- 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.)

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)



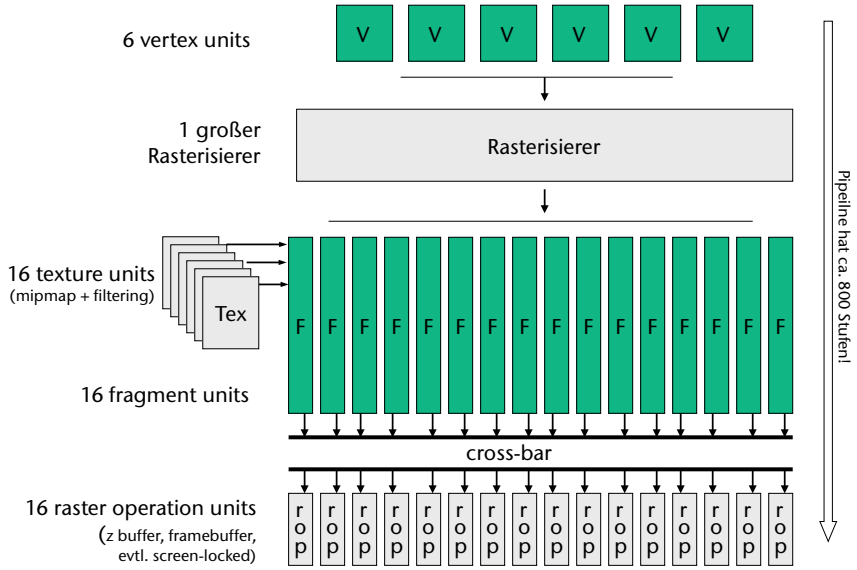
Aufgaben des Fragment-Processors

- 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.

Was ein Shader nicht kann

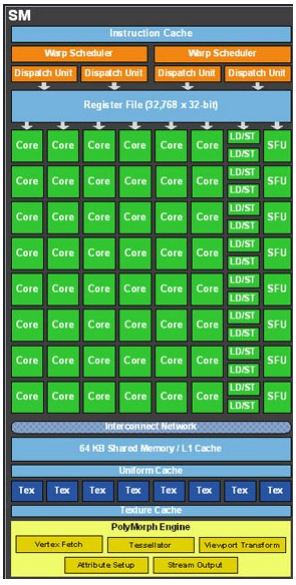
- Ein Vertex-Shader hat keinen Zugriff auf Connectivity-Info und Framebuffer
- Ein Fragment-Shader
  - hat keinen Zugriff auf danebenliegende Fragmente
  - hat keinen Zugriff auf den Framebuffer
  - kann nicht die Pixel-Koordinaten wechseln (aber kann auf sie zugreifen)

Etwas detailliertere (veraltete) Architektur



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



Wie sieht nun echter Shader-Code aus?

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 = Cd * (cAmbi + cDiff) +
                Cs * spec * cSpec;
```

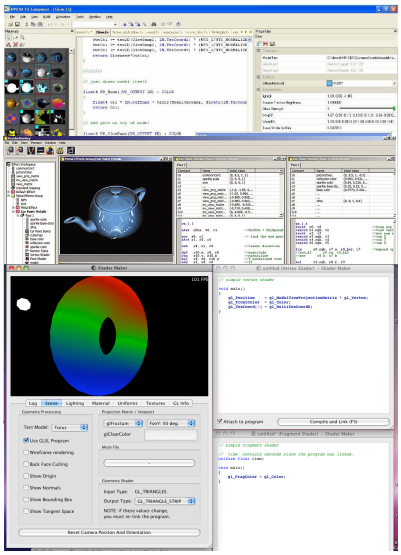
Einfacher Phong-Shader ausgedrückt in Assembly und GLSL

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, ...

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

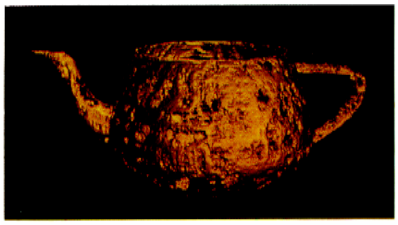


Debugging ...

- 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

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

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

    /* Sum 6 "octaves" 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;
    turbulence *= dent;

    /* Displace surface and compute normal */
    P -= turbulence * normalize(N);
    Nf = faceforward( normalize(calculatenormal(P)), I );
    V = normalize(-I);

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


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

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!

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)

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

```
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);
```

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.)      // COLUM 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
```

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

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
```

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)
```

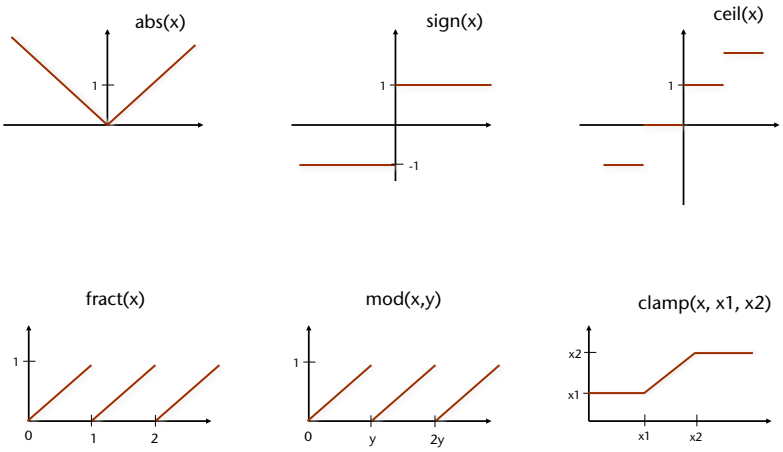

Statements und Funktionen

- 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 ); }
    ```
    ```

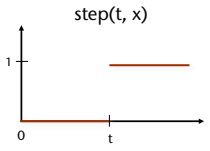
Eingebaute Funktionen

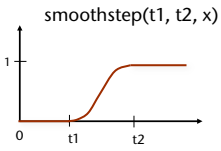
- Trigonometrie: **`sin, asin, radians,`** ...
- Exponentialfunktionen : **`pow, exp, log, sqrt,`** ...
- Sonstige: **`abs, clamp, max, sign,`** ...
- Alle o.g. Funktionen nehmen und liefern **`float, vec2, vec3,`** oder **`vec4`**, und arbeiten komponentenweise!
- Geometrische Funktionen: **`cross(vec3,vec3),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

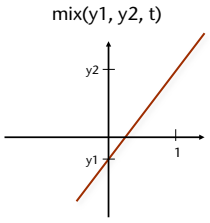
Einige häufige Funktionen



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








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

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

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


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 Fragement-Shader gelesen werden
  - Sog. "attribute"–Variablen nur vom Vertex-Shader
- Mittels Texturen können Daten an Shader übergeben werden
  - Interpretation bleibt Shader überlassen

Spezielle vordefinierte Variablen im Vertex-Shader

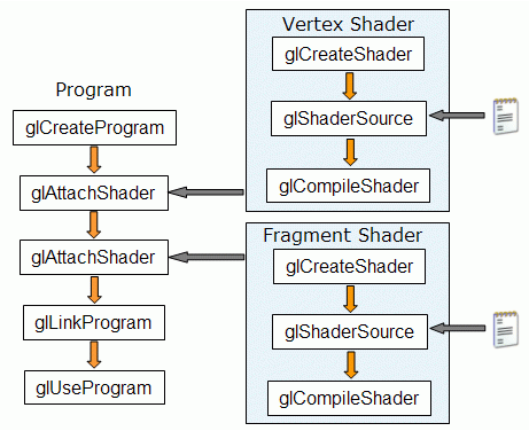
- Output: **`gl_Position = vec4`** …
  - Diese Variable **muss** vom Shader geschrieben werden!
- Input (*attributes*): **`gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord0,`** …
  - Alle sind **`vec4`**
  - Werden gesetzt durch den entsprechenden **`gl`**-Befehl (**`glNormal, glColor, glTexCoord`**;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

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`**, …

Laden eines Shaders

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



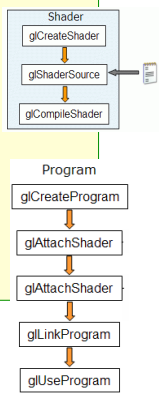
Im Detail

```
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 );
glUse Programm( progr_handle );
```



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

Beispiel: Hello_GLSL

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

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:

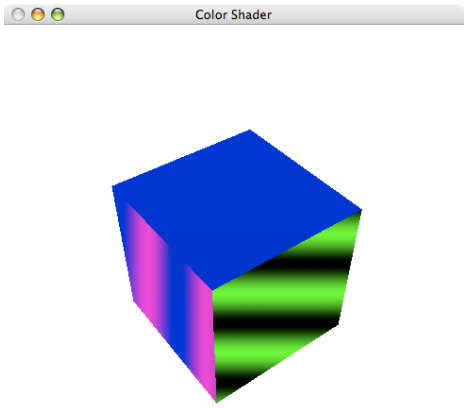
```
glUniform1f( var_handle, f )
```

  - Für Matrizen

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

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

Beispiel für uniform-Variable



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:

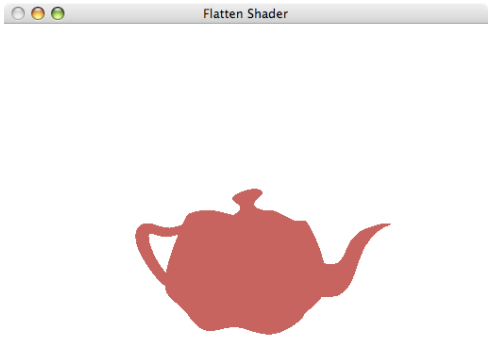
```
gl_Position = ftransform();
```

- Identisch dazu ist:

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

Beispiel für die Modifikation der Geometrie

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



lighthouse_tutorial/flatten.*

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;
```

- 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];
```

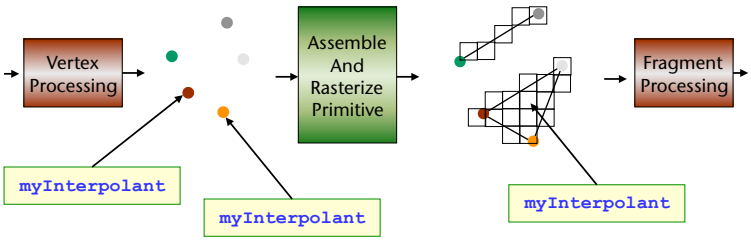
- Und viele weitere (z.B. zu Texturen, Clipping Planes,...)

Parameter-Übergabe von Vertex- zu Fragment-Shader

- Mittels sog. "varying"-Variablen:

```
varying vec3 myInterpolant;
```

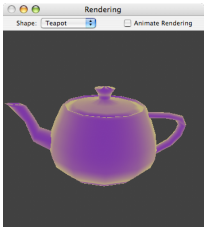
- Achtung: dazwischen sitzt der Rasterizer und interpoliert!
  - Der Rasterizer interpoliert auch die "varying"-Variablen! (zusätzlich zu Position, Farbe, etc.; hence the name)



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)

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:

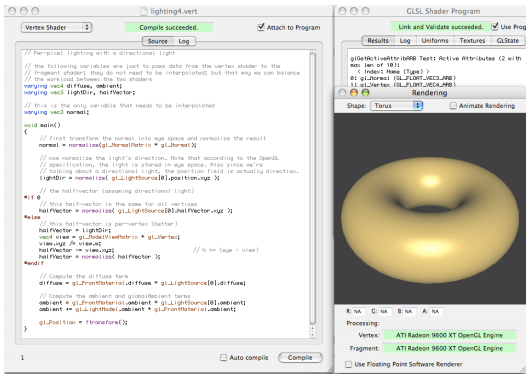
```
attribute vec3 myAttrib;
```

  - 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 `v = vec4(v.xyz, v.w)`
  - 3D-Äquivalent = `v.xyz/v.w`
- Subtraktion zweier Punkte/Vektoren **v** und **e** :
  - Homogen: $\mathbf{v} - \mathbf{e}$
  - Als 3D-Äquivalent:

$$\frac{\mathbf{v}.xyz}{\mathbf{v}.w} - \frac{\mathbf{e}.xyz}{\mathbf{e}.w} = \frac{\mathbf{v}.xyz \cdot \mathbf{e}.w - \mathbf{e}.xyz \cdot \mathbf{v}.w}{\mathbf{v}.w \cdot \mathbf{e}.w}$$

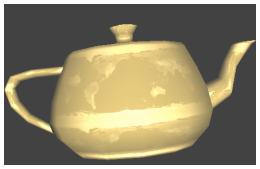
- Normalisierung:

$$\left(\frac{\mathbf{v}}{a}\right)^0 = \frac{\frac{\mathbf{v}}{a}}{\left\|\frac{\mathbf{v}}{a}\right\|} = \mathbf{v}^0$$

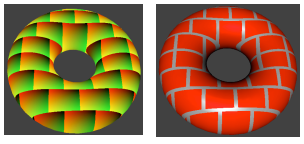
- Zusammen in GLSL :

```
normalize(v-e) = normalize(v.xyz*e.w - e.xyz*v.w)
```

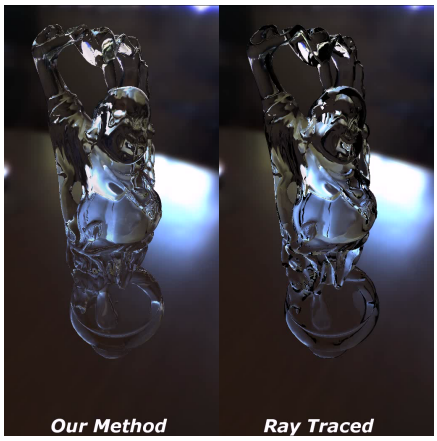

Ausblick (in Computer Graphik 2)



"Gloss-Textur"

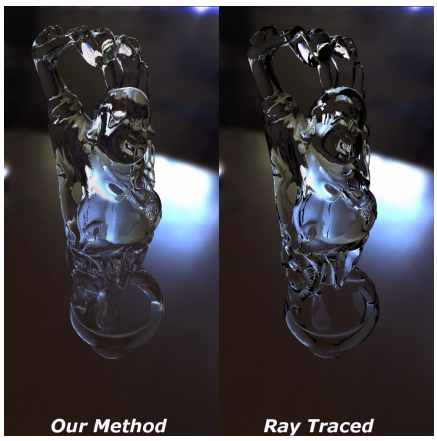


Prozedurale Texturen



Lichtbrechung (1 bounce)

Beispiele



Mit innerer Reflexion