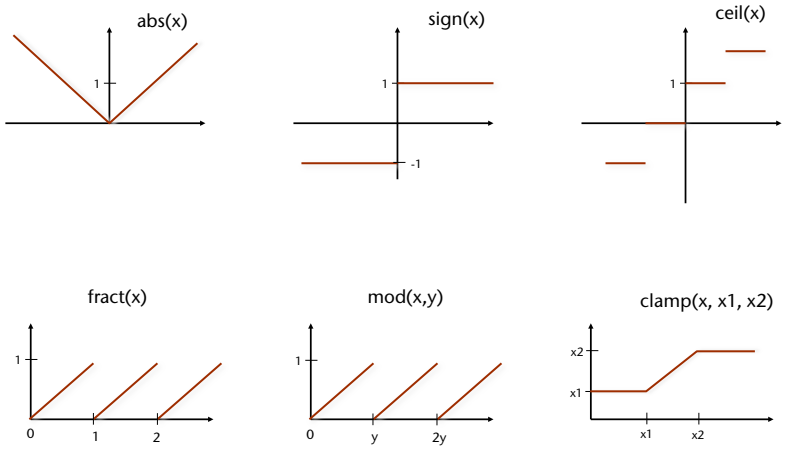
Statements und Funktionen

- Flow Control wie in C:
  - `if ( bool expression ) { ... } else { ... }`
  - `for ( initialization; bool expression; loop expr ) { ... }`
  - `while ( bool expression ) { ... }`
  - `do { ... } while ( bool expression )`
  - `continue, break`
  - `discard:` nur im Fragment-Shader, wie `exit()` in C, kein Pixel wird gesetzt
- Funktionen:
  - `void main():` muß 1x im Vertex- und 1x im Fragment-Shader vorkommen
  - `in` = input parameter, `out` = output parameter, `inout` = beides
  - ```
    vec4 func( in float intensity ) {
                vec4 color;
                if ( intensity > 0.5 ) color = vec4(1,1,1,1);
                else                   color = vec4(0,0,0,0);
                return( color ); }
    ```

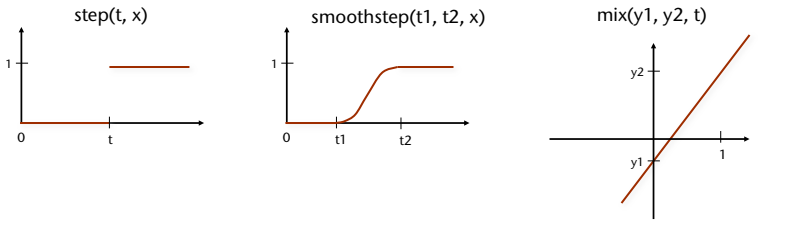

Eingebaute Funktionen

- 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

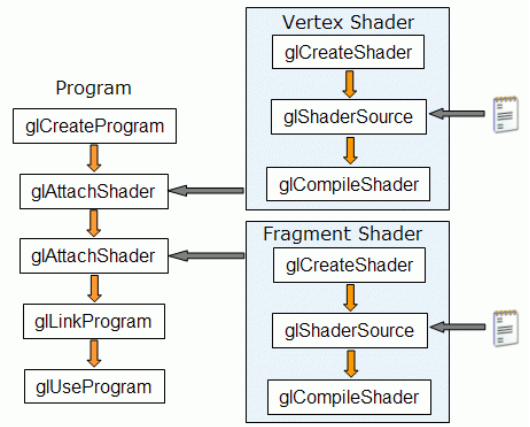
- Output: **`gl_Position = vec4`** …
  - Diese Variable **muss** vom Shader geschrieben werden!
- Input (*attributes*): **`gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord0,`** …
  - Alle sind **`vec4`**
  - Werden gesetzt durch den entsprechenden **`gl`**-Befehl (**`glNormal, glColor, glTexCoord`**; 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 (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



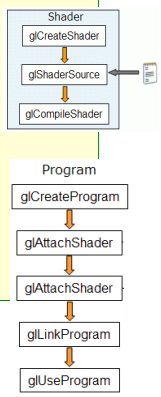
Im Detail

```
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 );

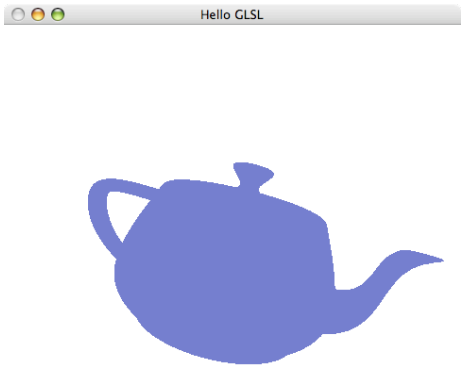
glLinkProgramm( progr_handle );
glUse Programm( progr_handle );
```



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 der Parameter eines GLSL-Programms

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

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:

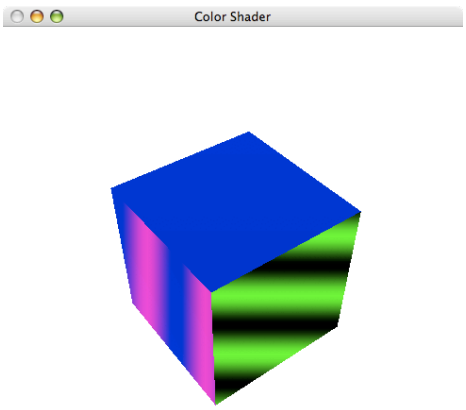
```
glUniform1f( var_handle, f )
```

  - 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 abbilden
- Idiom:

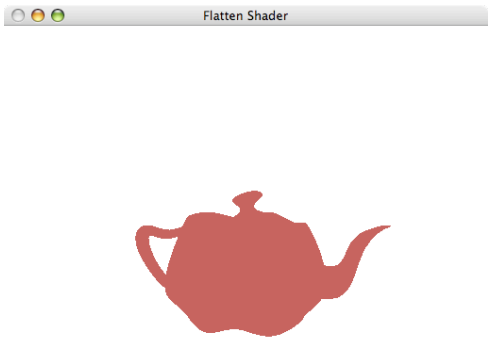
```
gl_Position = ftransform();
```

- 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:



`lighthouse_tutorial/flatten.*`

Zustandsvariablen

- Zeigen den aktuellen Zustand von OpenGL an
- Sind als "uniform"–Variablen implementiert
- 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];
```

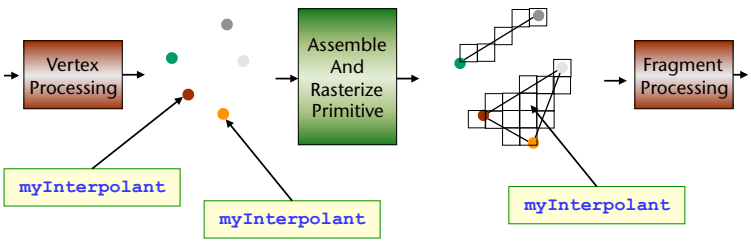
- Und viele weitere (z.B. zu Texturen, Clipping Planes,...)

Parameter-Übergabe von Vertex- zu Fragment-Shader

- Mittels sog. "varying"-Variablen:

```
varying vec3 myInterpolant;
```

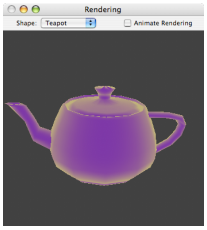
- Achtung: dazwischen sitzt der Rasterizer und interpoliert!
  - Der Rasterizer interpoliert auch die "varying"-Variablen! (hence the name)



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 Attribute definieren:
  - Im Vertex-Shader:

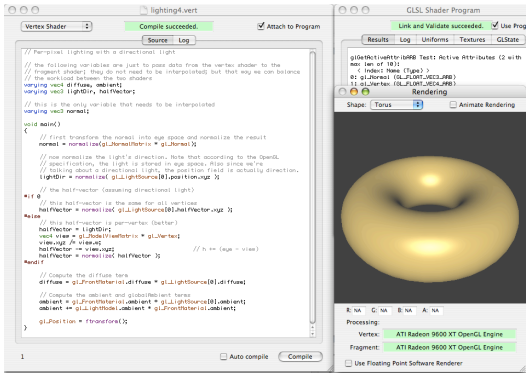
```
attribute vec3 myAttrib;
```

  - 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].*`