



Die spezielle Funktion **ftransform**



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

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

- Identisch:

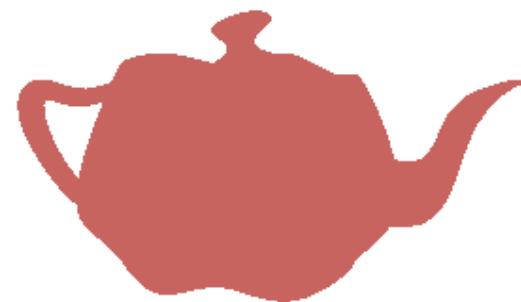
```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```



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,...)



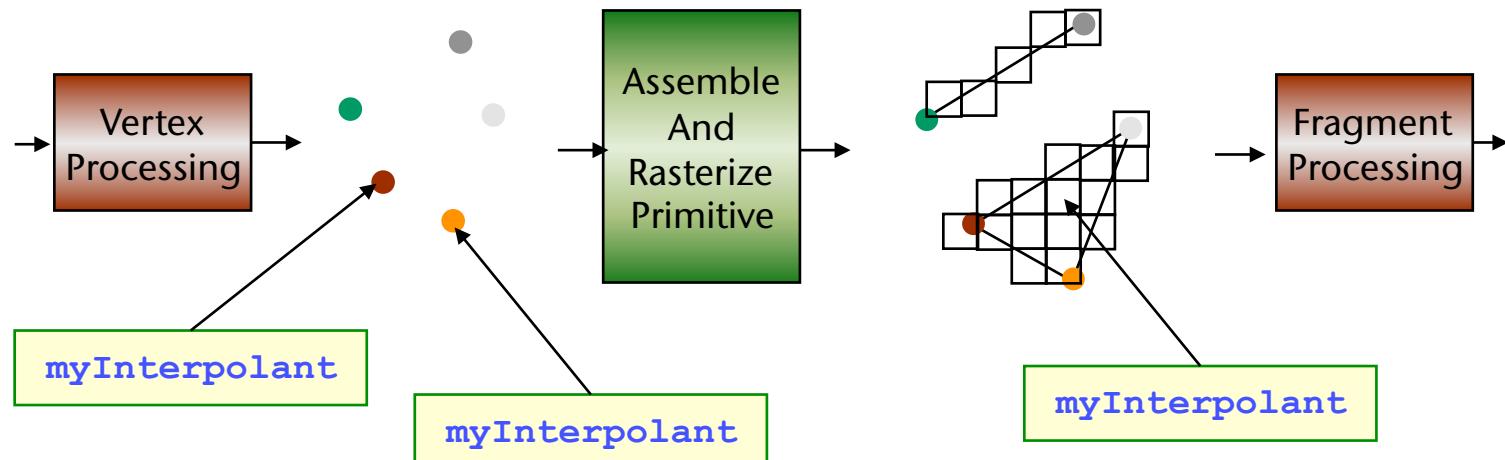
Parameter-Übergabe von Vertex- zu Fragment-Shader



- Mittels sog. "varying"-Variablen:

```
varying vec3 myInterpolant;
```

- Achtung: dazwischen sitzt der Rasterizer und interpoliert!





Beispiel für Verwendung von varying- und Zustands-Variablen



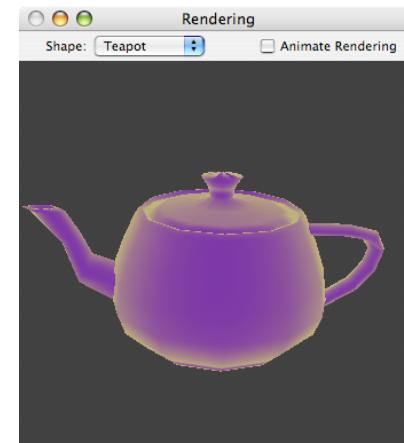
- Der "Toon-Shader":

- 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 = glGetUniformLocation( 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].*

The screenshot shows the GLSL Editor interface with two tabs open: 'lighting4.vert' (Vertex Shader) and 'lighting4.frag' (Fragment Shader).
The Vertex Shader code is as follows:

```
// Per-pixel lighting with a directional light
// the following variables are just to pass data from the vertex shader to the
// fragment shader; they do not need to be interpolated; but that way we can balance
// the workload between the two shaders
varying vec3 diffuse, ambient;
varying vec3 lightDir, halfVector;

// this is the only variable that needs to be interpolated
varying vec3 normal;

void main()
{
    // first transform the normal into eye space and normalize the result
    normal = normalize(gl_NormalMatrix * gl_Normal);

    // now normalize the light's direction. Note that according to the OpenGL
    // specification, the light is stored in eye space. Also since we're
    // talking about a directional light, the position field is actually direction.
    lightDir = normalize( gl_LightSource[0].position.xyz );

    // the half-vector (assuming directional light)
#ifndef
    // this half-vector is the same for all vertices
    halfVector = normalize( gl_LightSource[0].halfVector.xyz );
#else
    // this half-vector is per-vertex (better)
    halfVector = lightDir;
    vec4 view = gl_ModelViewMatrix * gl_Vertex;
    view.xyz /= view.w;
    halfVector -= view.xyz;
    halfVector = normalize( halfVector );
#endif

    // Compute the diffuse term
    diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;

    // Compute the ambient and globalAmbient terms
    ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
    ambient += gl_LightModel.ambient * gl_FrontMaterial.ambient;

    gl_Position = ftransform();
}
```

The Fragment Shader code is as follows:

```
glGetActiveAttribARB Test: Active Attributes (2 with
max len of 10):
< Index: Name (Type) >
0: gl_Normal (GL_FLOAT_VEC3_ARB)
1: gl_Vertex (GL_FLOAT_VEC4_ARB)
```

The rendering window shows a torus with a bright yellow glow at the center, indicating the light source.



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:

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

- 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)
```



Zugriff auf Texturen im Shader

- Deklariere Textur im Shader (Vertex oder Fragment):

```
uniform sampler2D myTex;
```

- Lade und binde Textur im C-Programm wie gehabt:

```
glBindTexture( GL_TEXTURE_2D, myTexture );
glTexImage2D(...);
```

- Verbinde beide:

```
uint mytex = glGetUniformLocation( prog, "myTex" );
 glUniform1i( mytex, 0 ); // 0 = texture unit, not ID
```

- Zugriff im Fragment-Shader:

```
vec4 c = texture2D( myTex, gl_TexCoord[0].xy );
```



Beispiel: eine einfache "Gloss-Textur"



`vorlesung_demos/gloss.{frag,vert}`



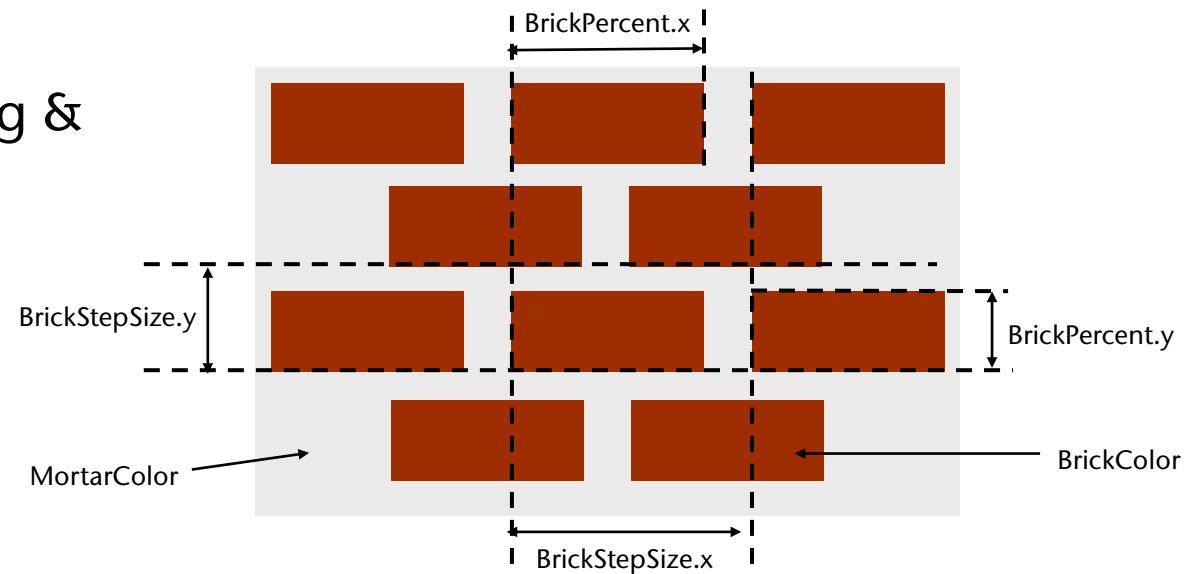
Eine einfache prozedurale Textur



- Ziel:
Ziegelstein-Textur



- Vereinfachung & Parameter:



- Generelle Funktionweise:
 - Vertex-Shader: normale Beleuchtungsrechnung
 - Fragment-Shader:
 - bestimme pro Fragment anhand der xy-Koordinaten des zugehörigen Punktes im Objektraum, ob der Punkt im Ziegel oder im Mörtel liegt
 - danach, entsprechende Farbe mit Beleuchtung multiplizieren
- Beispiele:

