

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:

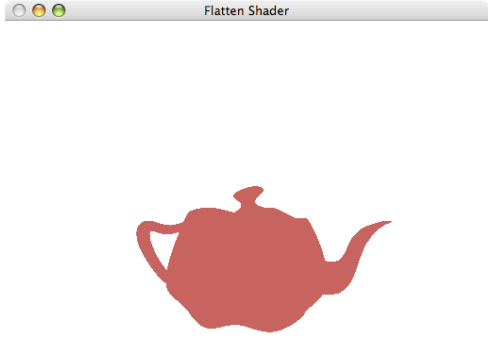

```
gl_Position = ftransform();
```
- Identisch:


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

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 55

Beispiel für die Modifikation der Geometrie

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



The screenshot shows a window titled "Flatten Shader" with a red teapot silhouette. The teapot is rendered as a flat, red shape, demonstrating the effect of a custom vertex shader that flattens the geometry.

`lighthouse_tutorial/flatten.*`

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 56

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;
```

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 57

Das aktuelle Material:

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

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 58

- 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

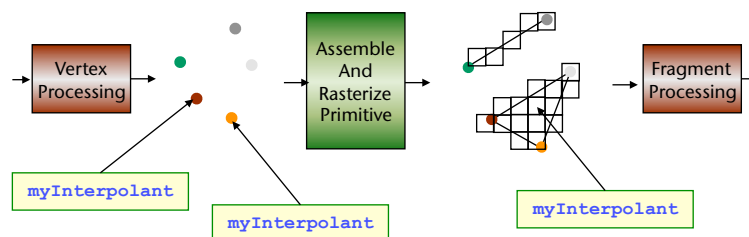
- Mittels sog. "varying"-Variablen:

```

varying vec3 myInterpolant;


```

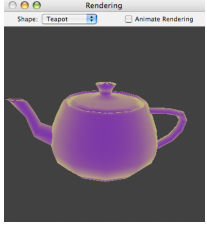
- Achtung: dazwischen sitzt der Rasterizer und interpoliert!



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)





G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 61

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 );
```

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 62

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 a GLSL Shader Program editor with two windows. The left window, titled 'lighting.vert', contains a vertex shader with comments explaining the lighting model: 'Per-pixel lighting with a directional light', 'The following variables are just to pass data from the vertex shader to the fragment shader', and 'Compute the diffuse term'. The right window, titled 'GLSL Shader Program', shows the program's status as 'Link and Validate succeeded' and a 3D render of a yellow torus with per-pixel lighting effects.

lighting[1-4].*

G. Zachmann Computer-Graphik 2 - SS 08
Shader und GPGPU 63

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:

$$\left(\frac{\mathbf{v}}{a}\right)^0 = \frac{\mathbf{v}}{\|\frac{\mathbf{v}}{a}\|} = \mathbf{v}^0$$
- Zusammen in GLSL :


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

G. Zachmann Computer-Graphik 2 - SS 08
Shader und GPGPU 64

Zugriff auf Texturen im Shader

- Deklareiere 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 );
```

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 65


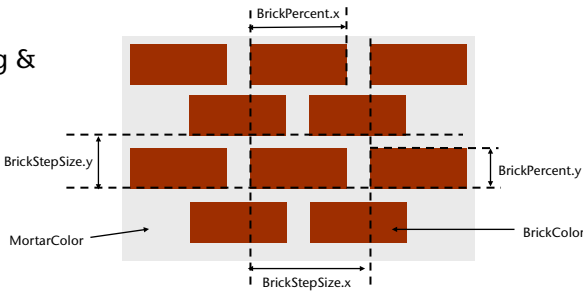
Beispiel: eine einfache "Gloss-Textur"



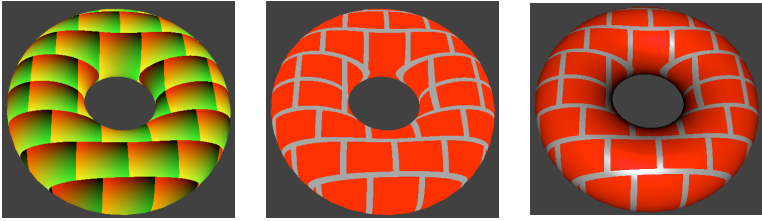
`vorlesung_demos/gloss.{frag,vert}`

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 66

Eine einfache prozedurale Textur

- Ziel:
 Ziegelstein-Textur
 
- Vereinfachung & Parameter:
 

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 67

- 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:
 

G. Zachmann Computer-Graphik 2 - SS 08 Shader und GPGPU 68