

Interaktive Computergrafik



Prof. Dr. Frank Steinicke
Human-Computer Interaction
Department of Computer Science
University of Hamburg



Interaktive Computergrafik

Übung - Woche 5



Interaktive Computergrafik

Übung - Woche 5

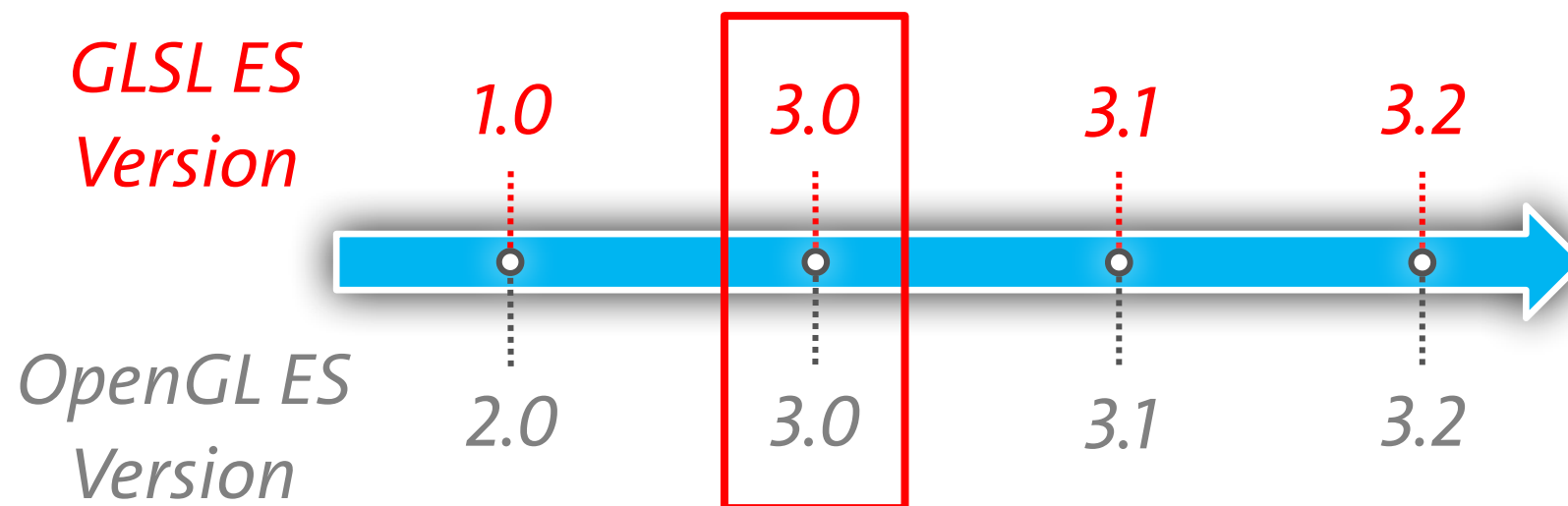
Die OpenGL Shading Language

GLSL

- Hochsprache, mit der GPUs programmiert werden können
- orientiert an **C/C++**
 - reduziert um einige Elemente des C++-Standards (z.B. Pointer)
 - erweitert um Vektor-/Matrixtypen und entsprechende Operationen

GLSL ES

- Abgeleitete Version von GLSL für eingebettete Systeme
→ eingeschränkt bzgl. Datentypen, Built-In-Variablen und -Funktionen
- WebGL 2.0 nutzt **GLSL ES 3.0!** (nicht aktuelle Version)



Programmaufbau

```
in vec4 color;  
const float PI = 3.14;
```

*globale Variablen,
Konstanten und
Attribute*

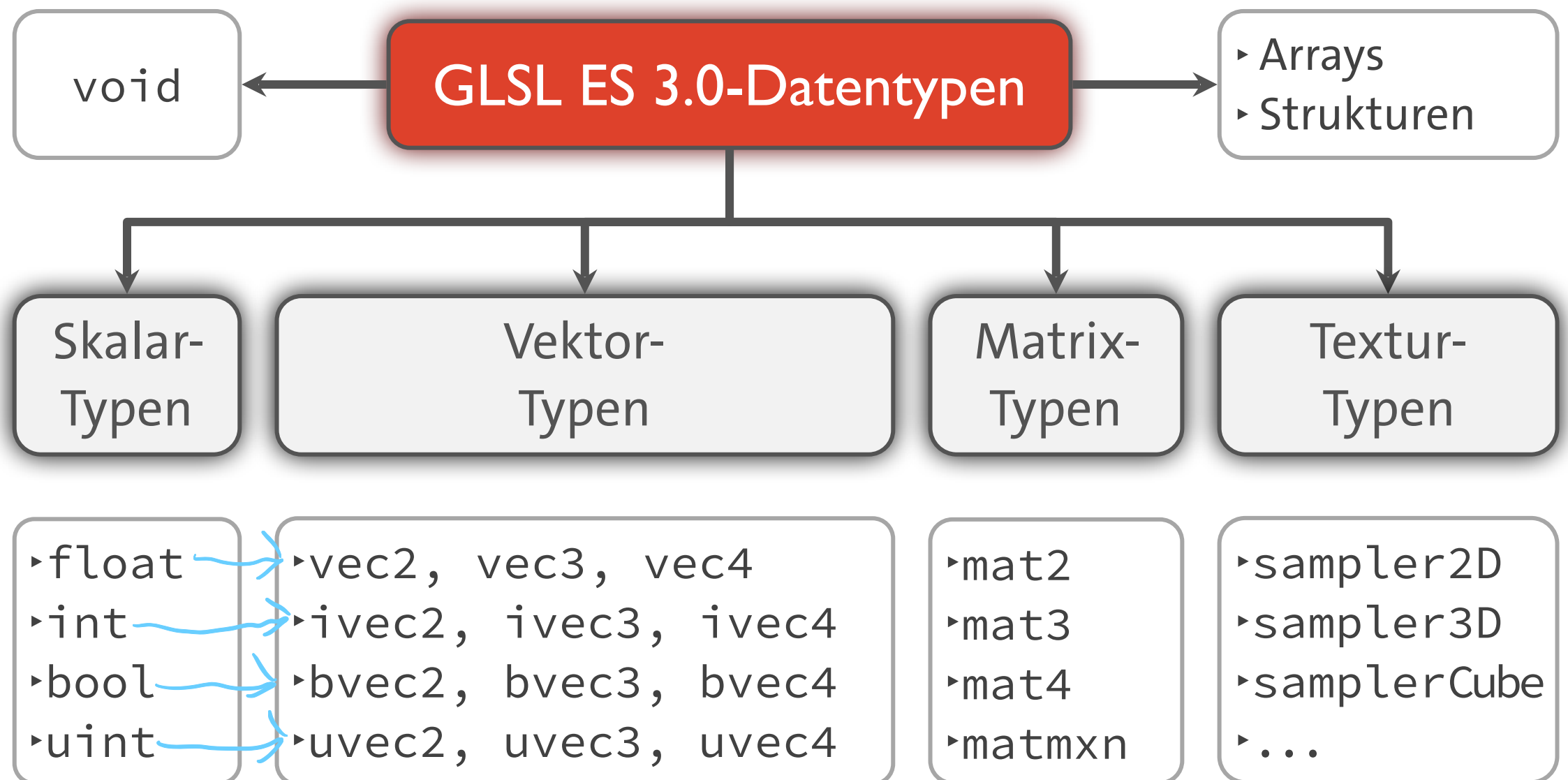
```
...
```

*[benutzerdefinierte
Funktionen]*

```
void main() {  
    ...  
}
```

main-Funktion

Datentypen



Vektoren

- Konstruktor

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
```

x y z w
r g b a
s t p q

- Zugriff

```
v[1], v.y, v.g, v.t    // 2.0  
v.xy                  // vec2(1.0, 2.0)  
v.rgr                 // vec3(1.0, 2.0, 1.0)
```


Matrizen

- Konstruktor

```
mat3 m = mat3 (  
    1.0, 2.0, 3.0, // 1. Spalte!  
    4.0, 5.0, 6.0, // 2. Spalte!  
    7.0, 8.0, 9.0  // 3. Spalte!  
);
```

- Zugriff

```
m[1]      // vec3(4.0, 5.0, 6.0)  
m[1][0], m[1].x  // 4.0
```

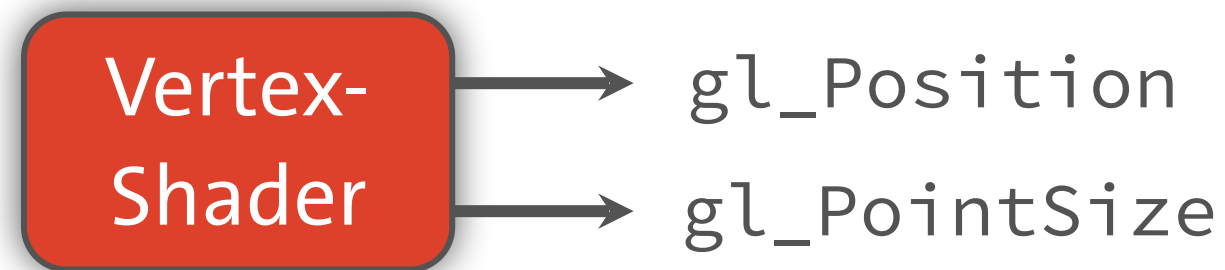
Hinweis

- Datentyp von Vertex-Position in Vertex-Shader immer `vec4` (auch bei Angabe von 2D-Koordinaten) → Dimension der Transformationsmatrix muss passen
→ 3x3-Matrix auf 4x4-Matrix erweitern:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Built-In-Variablen

- vom Shadertyp abhängige Menge vordefinierter Variablen



- **vec4** `gl_Position`:
Position des Vertices in homogenen Koordinaten
(sollte immer geschrieben werden)
- **float** `gl_PointSize`:
Größe der mit `GL_POINTS` gerenderten Punkte

Built-In-Funktionen

- **Trigonometrisch**, z.B.

`sin, cos, tan`

- **Exponential**, z.B.

`pow, sqrt, exp, log`

- **Geometrisch**, z.B.

`length, distance`

- **Sonstige**, z.B.

`floor, ceil, min, max`

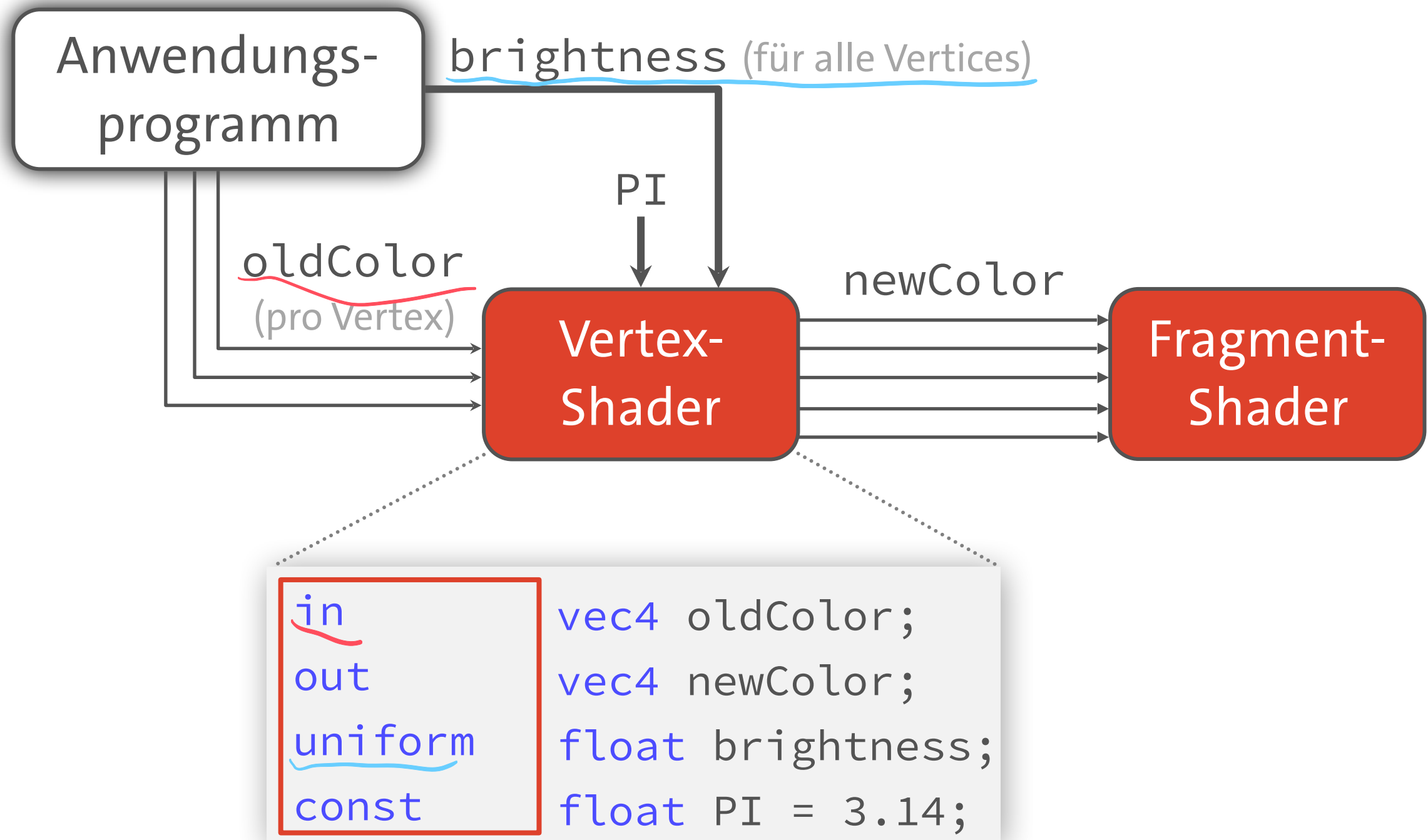
Typenqualifizierer

→ Beschreibung der Nutzung einer Variable

<code><none></code>	lokale Variable
<code>const</code>	Konstante
<code>in</code>	vertexspezifische Shader Input Variable ▶ in Vertex-Shader: aus Anwendungsprogramm(<i>javascript</i>) ▶ in Fragment-Shader: aus Vertex-Shader
<code>out</code>	vertexspezifische Shader Output Variable ▶ in Vertex-Shader: an Fragment-Shader ▶ in Fragment-Shader: an Framebuffer
<code>uniform</code>	Variable für Daten, die während eines Draw-Aufrufs über alle Vertices/Fragmente konstant bleiben, von Anwendungsprogramm an Shader

Typenqualifizierer

Beispiel: Vertex Shader



Wertübergabe



1. Speicheradresse ermitteln:

```
const loc = gl.getUniformLocation(  
    program, name);
```

2. Wert setzen:

```
gl.uniform1f(loc, value);
```

float

Wertübergabe



3. Wert lesen

```
uniform float name;
```


Wertübergabe

Beispiel: Matrix

- Übergabe von Transformationsmatrizen an Vertex-Shader über Uniform-Variable
- Schritt 1: Speicherort der Uniform-Variable bestimmen

```
const matrixLoc =  
    gl.getUniformLocation(  
        program, "transformMatrix");
```

Wertübergabe

Beispiel: Matrix

- Schritt 2: Wert der Variablen setzen

```
gl.uniformMatrix4fv(  
    matrixLoc,  
    false,  
    transformMatrix  
);
```

Speicherort der Uniform-Variable

Soll Matrix transponiert werden?

Neuer Wert der Matrix

Vorsicht: 2. Parameter ist Überbleibsel aus OpenGL
→ muss immer auf false gesetzt werden!

Wertübergabe

Beispiel: Matrix

- Schritt 3 (in Shader): Wert lesen

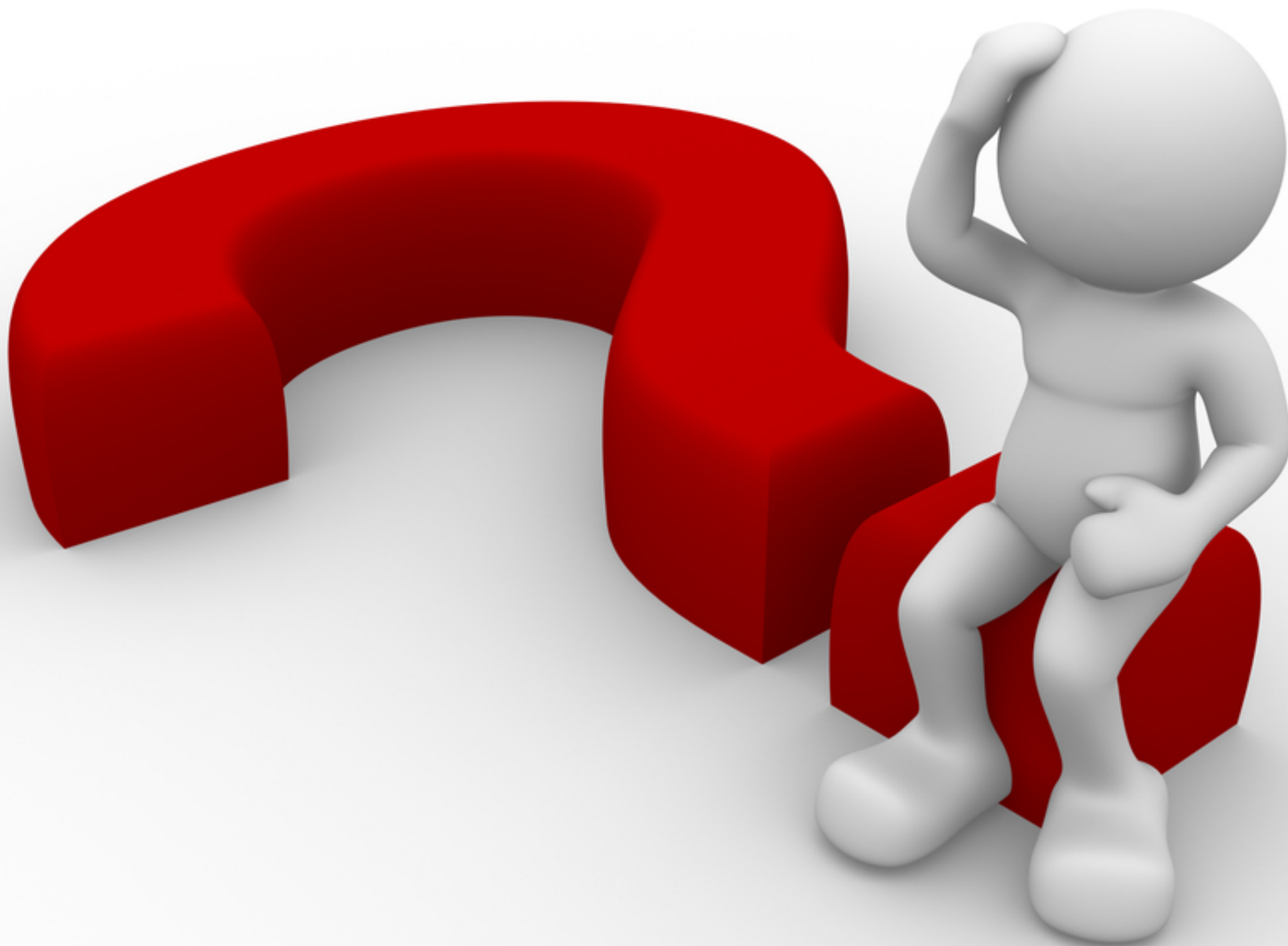
```
uniform mat4 transformMatrix;
```

- Matrizentyp muss zu Funktionsaufruf in Anwendungsprogramm passen, z.B.

`gl.uniformMatrix3fv` → `mat3`

Beispielprogramm

```
in vec4 vPosition;  
in vec4 vColor;  
out vec4 fColor;  
  
void main() {  
    fColor = vColor;  
    gl_Position = vPosition;  
}
```



JavaScript Matrizen

- Komfortable Nutzung von Matrizen im Shader durch GLSL-Datentyp `mat4`
- Aber: Transformationsmatrizen sollten aus Performancegründen von JavaScript-Anwendungsprogramm gesetzt werden
- Wie können Matrizen auch in JavaScript komfortabel genutzt werden?

Library *glmMatrix*

- Sammlung von JavaScript-Funktionen für Matrix- und Vektoroperationen
- Darf in Übungsaufgaben benutzt werden!
- Source + Dokumentation: glmatrix.net

Model Matrix

```
mat4.translate(out,in,translationVector);  
mat4.rotate(out,in,angle,axis);  
mat4.scale(out,in,scaleVector);
```

- Translationsdistanzen
- Rotationswinkel
- Skalierungsfaktoren

Model Matrix

Beispiel

- Einheitsmatrix erstellen:

```
let modelMatrix = mat4.create();
```

- Objekt verschieben:

```
mat4.translate(modelMatrix, modelMatrix,  
               [2, 2, 2]);
```

- Objekt um y-Achse rotieren:

```
mat4.rotate(modelMatrix, modelMatrix,  
            degToRad(90), [0, 1, 0]);
```

