

# Computergrafik SS 2010

Henning Wenke

Kapitel 21:

OpenGL 3.1

(Fortsetzung)

# OpenGL

---

## ➤ Aufgaben der GL-Befehle

- Konfigurieren der Graphics Pipeline
- Datenübergabe an Server
- Steuern des Datenflusses
- ...

## ➤ Java OpenGL Bindings (Jogl)

- Anbindung an Fenstersystem
- Zugriff auf native GL-Funktionen
- Code:

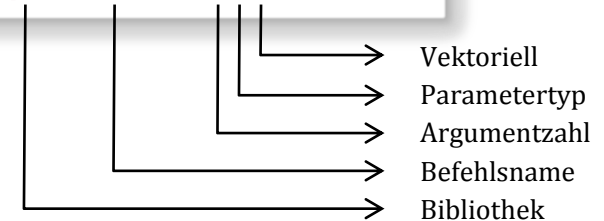
```
C-GL:      glEnable(GL_DEPTH_TEST);  
Jogl :     gl.glEnable(GL3.GL_DEPTH_TEST);
```

# Spracheigenschaften und Syntax

- Zustandsmaschine
- Eigene Datentypen
- Funktionskonvention

```
glClearColor(0, 0, 0, 0);
```

```
glUniform4fv(...);
```



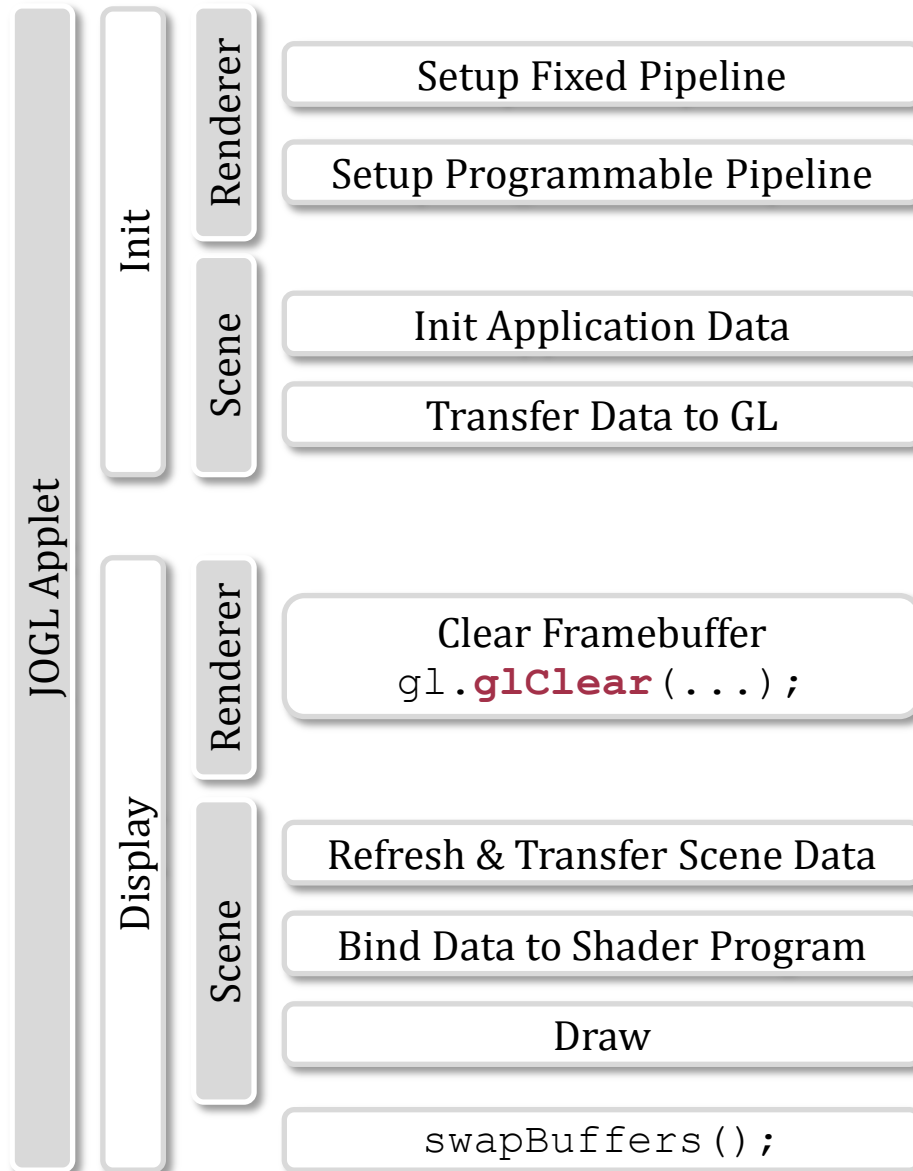
- Spezialisierung durch Konstanten

```
glEnable(int cap);
```

```
GL_DEPTH_TEST  
GL_CULL_FACE  
GL_PRIMITIVE_RESTART  
...
```

- Befehle beziehen sich auf:
  - GL-State
  - Übergebene Daten
  - GL-Objekte, (auch) über Ids

# OpenGL Programm Beispiel



# Setup Fixed Pipeline

---

```
// Hintergrundfarbe setzen
gl.glClearColor(0, 0, 0, 0);

// Z-Buffer aktivieren
gl.glEnable(GL3.GL_DEPTH_TEST);

// Transparenz deaktivieren
gl.glDisable(GL3.GL_BLEND);

// Culling der Flächen aktivieren
gl.glEnable(GL3.GL_CULL_FACE);

// Positiven Drehsinn festlegen
gl.glFrontFace(GL3.GL_CCW);

// Abgewandte Flächen nicht zeichnen
gl.glCullFace(GL3.GL_BACK);

// Informationen über Ausgabegerät
glViewport(x, y, width, height);
```

# Setup Programmable Pipeline

---

1. Erzeuge Shader Object (e)

2. Erzeuge Program Object

3. Aktiviere Program Object

```
// Erzeugt ein leeres (Vertex) Shader Object.  
int type = GL3.GL_VERTEX_SHADER  
int vertexShader = gl.glCreateShader(type);  
  
// Hinzufügen des Codes zum Shader Object  
int count = 1;  
IntBuffer length = null;  
String[] string = {"Sourcecode von gestern..."};  
gl.glShaderSource(vertexShader, count, string, length);  
  
// Übersetzen des Shaders  
gl.glCompileShader(vertexShader);
```

# Setup Programmable Pipeline (2)

---

1. Erzeuge Shader Object (e)

2. Erzeuge Program Object

3. Aktiviere Program Object

```
// Erzeugt (leeres) Program Object
int program = gl.glCreateProgram();

//Hinzufügen der Shader Objects
int shader = vertexShader;
gl.glAttachShader(program, shader);
shader = fragmentShader;
gl.glAttachShader(program, shader);

// Linken des Program Objects
gl.glLinkProgram(program);
```

```
// Aktiviere Program Object als Teil der Pipeline
gl.glUseProgram(program);
```

# Init Application Data

---

- Modelle
  - Vertex Daten
  - Primitive / Indizierung
  - Textur Daten
- Transformationsmatrizen
- Lichtquellen
- Shader Code



# Transfer Data to OpenGL

## Beispiel: Per Vertex Daten

```
// Erzeuge (leere) Buffer Objects
int n = 3;
java.nio.IntBuffer buffers = bufferNames;
gl.glGenBuffers(n, buffers);

// Initialisiere und aktiviere Buffer Object für
// Vertex Daten
int coordBuffer = bufferNames.get(0);
int buffer = coordBuffer;
int target = GL3.GL_ARRAY_BUFFER;
gl.glBindBuffer(target, buffer);

// Übergib Daten
int size = vertexCoords.capacity() * 4;
int usage = GL3.GL_STATIC_DRAW;
Buffer data = vertexCoords;
target = GL3.GL_ARRAY_BUFFER;
gl.glBufferData(target, size, data, usage);
```

# Bind Data to Shader

```
in vec4 vPosition;  
...
```

```
// Aktivieren eines Buffer Objects, etwa  
gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
```

```
// Abfragen der Adresse der In Variable  
int program = programObject;  
String name = "vPosition";  
int location = gl.glGetAttribLocation(program, name);  
  
// Buffer für Verwendung mit dieser Variable aktivieren  
int index = location;  
gl.glEnableVertexAttribArray(index);  
  
// Spezifikation des Datenformats  
int size = 3;  
int type = GL3.GL_FLOAT;  
boolean normalized = false;  
int stride = 0;  
int pointer = 0;  
gl.glVertexAttribPointer(location, size, type, normalized, stride, pointer);
```

# Draw

---

```
// Anzahl der zu rendernden Elemente, hier: alle
int count = sphere.indexCnt;

// Verweis auf das erste Element
int pointer = 0;

// Art des Primitives
int mode = GL3.GL_TRIANGLE_STRIP;

// Datentyp
int type = GL3.GL_UNSIGNED_INT;

// Auslösen des Renderns
gl.glDrawElements(mode, count, type, pointer);
```

# Weitere Beispiele

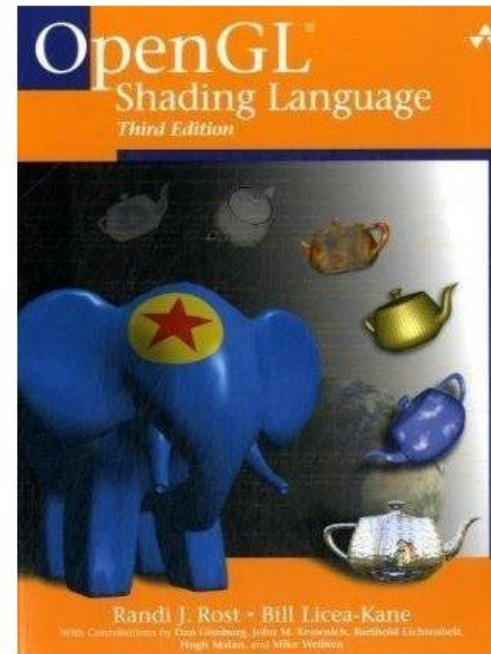
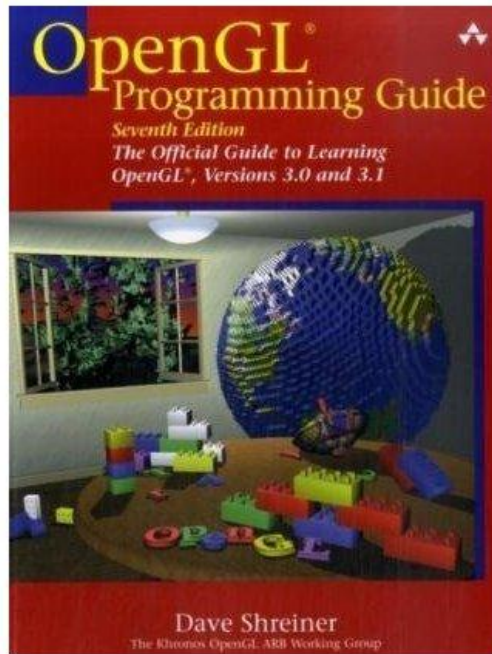
```
// Procedural Texturing Fragment Shader
#version 140
const float maxIterations = 100.0;
const vec3 innerColor = vec3(1.0, 0.0, 0.0);
const vec3 outerColor1 = vec3(0.0, 1.0, 0.0);
const vec3 outerColor2 = vec3(0.0, 0.0, 1.0);

// Berechnet nur in Abhängigkeit von den Texturkoordinaten...
in vec2 texCoords;

// ...die Farbe des Fragments
out vec4 myFragColor;

void main() {
    float real = texCoords.x; float imag = texCoords.y;
    float cReal = real; float cImag = imag;
    float r2 = 0.0;
    float iter;
    for(iter = 0.0; iter < maxIterations && r2 < 4.0; iter++) {
        float tempreal = real;
        real = (tempreal * tempreal) - ( imag * imag) + cReal;
        imag = 2.0 * tempreal * imag + cImag;
        r2 = (real * real) + (imag * imag);
    }
    vec3 color;
    if (r2 < 4.0)
        color = innerColor;
    else // mix ist eine GLSL-Funktion für lineare Interpolation
        color = mix(outerColor1, outerColor2, fract(iter * 0.05));
    myFragColor = vec4(color, 1.0);
}
```

# Literatur



Weitere Infos:

Henning Wenke ([hewenke@uos.de](mailto:hewenke@uos.de))

Raum: 31/318a