

Computergrafik

Universität Osnabrück, Henning Wenke, 2012-05-08

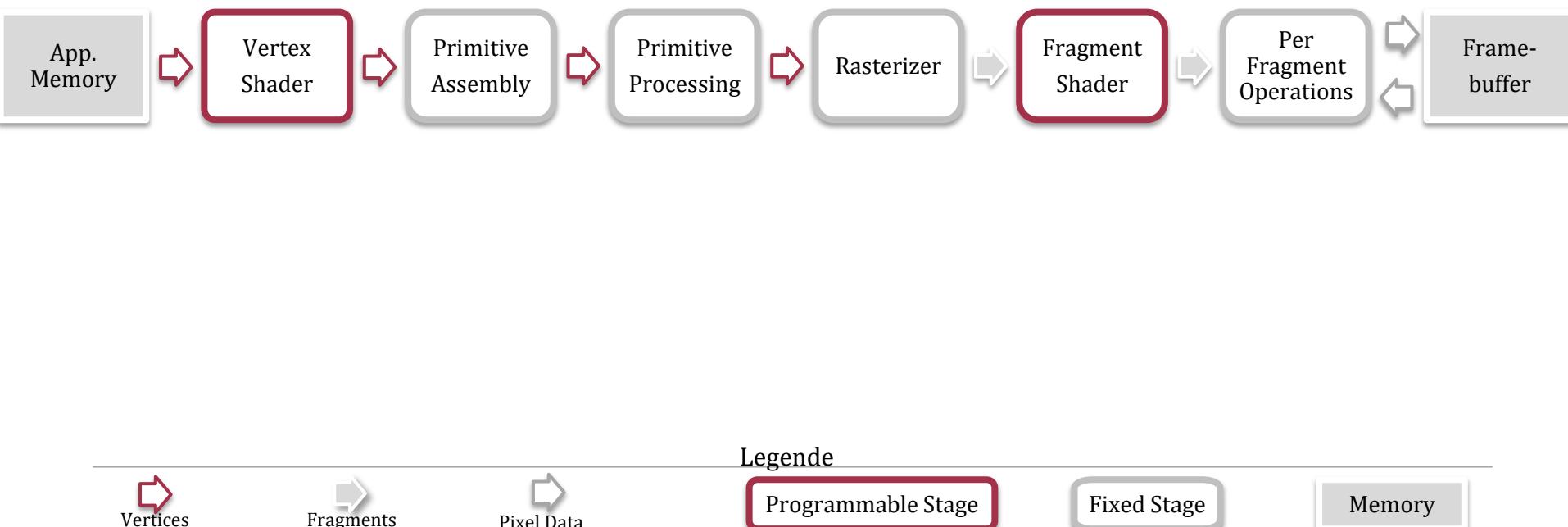
Noch Kapitel IV: OpenGL

Zielsetzungen

- Hardwarenah aber unabhängig
- Verschiedene Anwendungsbereiche
 - Wissenschaft, Visualisierung & Entwicklung (CAD)
 - Spiele, Computergrafik
 - Grafik im Web und auf Smartphones
- Betriebssystem- & Plattformunabhängig
 - Windows, Linux, Mac, ...
 - Pc, Smartphone, Web, Workstation, ...
- Aus vielen Programmiersprachen verwendbar
 - Fortran, C/C++, Java
 - Ruby, Php, JavaScript
 - Deshalb z.B. Kein überladen der Funktionsargumente

Graphics Processing Pipeline

- Folge von Operationen, die mathematisch beschriebene Szene in ein Bild umzusetzen
- Vereinfachte Version aus OpenGL 3.1:



Shader

- In einer Shadersprache (GLSL, HLSL, Cg, ...) geschriebenes Programm, welches auf einer GPU ausgeführt werden kann
- Prozedural
- Heute oft an C angelehnte Hochsprachen mit speziellen Vektor und Matrix Datentypen und Operatoren
- Repräsentiert einen programmierbaren Teil der Graphics Pipeline
- Wird nach den jeweiligen Daten benannt:
 - Vertex Shader (VS) (zwingend)
 - Tesselation Control Shader (optional, ab GL 4.0)
 - Tesselation Evaluation Shader (optional, ab GL 4.0)
 - Geometry Shader (optional, ab GL 3.2)
 - Fragment Shader (FS) („optional“)

Aufgaben von OpenGL

- Verwaltet Graphics Pipeline
- Algorithmen der programmierbaren Abschnitte sind selbst zu implementieren
- Feste Stages können ausschließlich konfiguriert werden
- Erzeugen, übersetzen, etc. der Shader Programme
- Kontrollieren des Datenflusses
- Keinerlei High Level Funktionalität

State Machine & Client Server

➤ Client-Server Modell

- Client, z.B. unsere Java Applikation, setzt OpenGL Befehle ab
- Server, d.h. die OpenGL Implementation, führt diese aus
- Server Beispiel: Grafikkarte + Treiber
- Client und Server typischerweise in einem Rechner
- Trotzdem i.d.R. kein gemeinsamer Speicher

➤ Zustandsmaschine

- Einmal gesetzte Zustände, etwa die Hintergrundfarbe, bleiben bis zum Widerruf gültig
- Minimiert Client-Server Kommunikation

4.1

OpenGL Programmierung: Grundlagen

GL-Befehle I

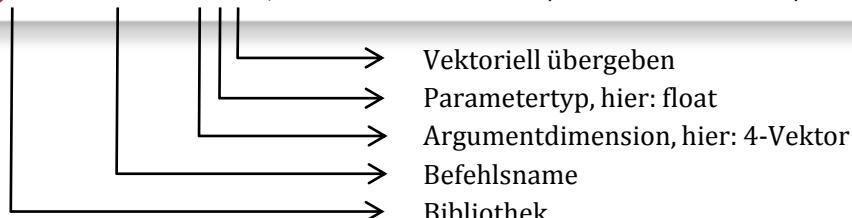
- Form der Befehle folgt folgendem Schema:

```
gl<Root Command>[arg count] [arg type] [v]
```

- Beginnen immer mit **gl**
- Gefolgt vom Befehlsnamen
- Anzahl und Art der Argumente (optional)
- Endung **v** weist auf Pointer oder Arrays als Argumente hin (optional)

- Beispiele

```
glUniform4fv(int location, int count, float* value);
```



```
glUniform3i(int location, int v0, int v1, int v2);
```

GL-Befehle II

➤ Weitere Spezialisierung durch Konstanten

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

cap kann sein:

`GL_DEPTH_TEST`

`GL_CULL_FACE`

`GL_PRIMITIVE_RESTART`

...

➤ OpenGL spezifiziert eigene Datentypen, zur einheitlichen Verwendung der Befehle

- Entsprechen C-Datentypen, so ist `GLfloat` identisch mit dem C-Datentyp `float`
- Wir verwenden ausschließlich die C Datentypen
- Buffer Objects statt Pointer
- Keine unsigned Varianten

GL-Befehle III

- OpenGL-Befehle beziehen sich auf:
 - Übergebene Daten
 - GL-State
 - GL-Objekte, (auch) über Ids
- Viele OpenGL-Befehle werden nicht sofort ausgeführt
- Ermöglicht parallele Verarbeitung möglichst großer Datenmengen
- Passiert spätesten bei Anzeige des Bildes
- Auch manuell möglich, z.B. mit `glFinish();`

Fehler I

- Wichtig: OpenGL meldet selbständig keine Fehler
- OpenGL besitzt ein Error-Flag
- Kann auf eine von 5 Fehlerkonstanten oder `GL_NO_ERROR` (initial) gesetzt werden
- Meldet eine GL Funktion einen Fehler, wird entsprechende Konstante gesetzt
- Neuer Fehler: Alte Konstante wird überschrieben
- Konstante muss manuell abgefragt werden
- Anschließend gilt wieder `GL_NO_ERROR`

Fehler II

```
// Liefert Fehlerkonstante. Setzt danach Error-Flag auf 0  
int glGetError();
```

```
// Mögliche Fehlerkonstanten. Jeweilige Bedeutung kann in  
// Dokumentation, z.B. GL Reference Pages, des verursachenden  
// GL-Befehls nachgeschlagen werden.  
GL_NO_ERROR  
GL_INVALID_ENUM  
GL_INVALID_VALUE  
GL_INVALID_OPERATION  
GL_INVALID_FRAMEBUFFER_OPERATION  
GL_OUT_OF_MEMORY
```

4.2

LWJGL & Unser Wrapper

LWJGL

- Liefert zu jedem OpenGL Befehl einen Java Befehl gleichen Namens, der diesen ausführt
- Klassen GL11 bis GL42 enthalten die in der jeweiligen Version eingeführten Funktionen und Konstanten
- Beispiel: Wrapper um den Befehl `glClearColor` der Klasse GL11 (vereinfacht)

```
// in Java formulierter LWJGL-Befehl
void org.lwjgl.opengl.GL11.glClearColor(float red, float green,
                                         float blue, float alpha) {

    // ...ruft den Systemnahen Original OpenGL-Befehl auf
    void glClearColor(GLfloat red, GLfloat green,
                     GLfloat blue, GLfloat alpha);
}
```

LWJGL vs. „C-OpenGL“

- Verwendet von `java.nio.Buffer` abgeleitete Klassen, etwa `FloatBuffer`, anstelle der C-Pointer
- Strings statt `char*`
- Argumente weichen manchmal ab, Beispiel:

```
//Grob: Befehl zur Übergabe von Daten an GL
void glBufferData(..., GLsizeiptr size, // Größe in Bytes
                  const GLvoid* data, // Pointer auf Daten
                  ...);
```

```
// Entspricht dem LWJGL-Befehl
void GL15.glBufferData(..., FloatBuffer data, // FloatBuffer mit Daten
                      ...);
// Hinweis: size entspricht data.capacity() und muss daher nicht
// gesondert übergeben werden
```

Probleme mit LWJGL

- Nicht erkennbar, ob eine Funktion / Konstante aus OpenGL (Core) entfernt ist oder nicht
- In welcher Klasse ist der Befehl denn nun?
- Umständliche Fehlersuche
- Sehr viele Funktionen (knapp 1000)

Unser Wrapper

- Sammelt benötigte/erlaubte GL-Funktionen (ca. 50) und Konstanten in einer Klasse „GL“
- Führt (optional) Fehlerkontrolle aus
- Befehlsname und Parameter mit LWJGL identisch

```
// Unser Wrapper...

void glClearColor(float red, float green, float blue, float alpha) {

    // ...führt erst den entsprechenden LWJGL Befehl aus
    void GL11.glClearColor(float red, float green, float blue,
                          float alpha) {
        // Der ruft den original Befehl auf
        glClearColor(...);

    }

    if(checkForErrors) // Anschließend wird, wenn gewünscht,
        checkError(); // eine Fehlerkontrolle durchgeführt
}
```

4.3

Erzeugen von Shader- und Program Object

Erzeugen eines Shader Programs

- Shader Program kapselt programmierbaren Teil der Graphics Pipeline
- Enthält Shader Objects, welche die jeweiligen Shader repräsentieren. Mindestens: Vertex Shader
- Werden zur Laufzeit durch Grafikkartentreiber übersetzt

Erzeugen eines Shader Objects

```
// Erzeugt ein leeres Shader Object. Liefert dessen id.  
// shaderType: Konstante zur Angabe der Shaderart  
//           GL_VERTEX_SHADER, GL_FRAGMENT_SHADER, ...  
int glCreateShader(int shaderType);
```

```
// Hinzufügen des Sourcecodes zum bisher leeren Shader Object  
glShaderSource(  
    int shader,      // id des Shader Objects  
    String string, // Sourcecode des Shaders  
) ;
```

```
// Übersetzen des Shader Objects mit der id shader  
glCompileShader(int shader);
```

```
// Beispiel: Erzeugen eines Vertex Shaders "ourVS"  
int ourVS = glCreateShader(GL_VERTEX_SHADER);  
glShaderSource(ourVS, „version 330 core ...”);  
glCompileShader(ourVS);
```

Erzeugen eines Program Objects

```
int glCreateProgram(); // Erzeugt (leeres) Program Object und liefert dessen id.
```

```
// Hinzufügen eines Shader Objects zu einem Program Object
glAttachShader(
    int program,      // id des Program Objects
    int shader        // id des Shader Objects
);
```

```
// Verbinden der einzelnen Module zu ausführbarem Programm
glLinkProgram(int program); // id des Program Objects
```

```
// Beispiel: Erzeugen PO, bestehend aus VS und FS
int ourProgram = glCreateProgram()
// Erzeugen von VS und FS
int ourVS = glCreateShader(GL_VERTEX_SHADER);
int ourFS = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(ourVS, "version..."); glShaderSource(ourFS, "version...");
glCompileShader(ourVS); glCompileShader(ourFS);
// Hinzufügen der Shader und Linken.
glAttachShader(ourProgram, ourVS);
glAttachShader(ourProgram, ourFS);
glLinkProgram(ourProgram);
```

Verwenden des Program Objects

```
// Setzt das Program Object "program" als Teil des Rendering State.  
// Anschließend besteht der programmierbare Teil der Pipeline aus den  
// mit "program" assoziierten Shadern  
glUseProgram(int program);
```

```
// Beispiel: Verwende das PO der letzten Folie  
glUseProgram(ourProgram);
```

4.4

GLSL Basics

Basics

```
// Gibt die GLSL-Version an, hier 3.3 (gehört zu OpenGL 3.3)
// sowie die Beschränkung auf das Core Profile
#version 330 core

in <varType><varName>      // Hereinkommende Daten
out <varType><varName>     // Zu schreibende Daten

// Ausführung des Shaders beginnt hier
void main() {

    // Führe Berechnungen v.A. basierend auf "in" Variablen durch
    // und schreibe Ergebnisse mit "out" Variablen raus

    // Andere Berechnungen sind lokal und haben keine Auswirkungen
}
```

Skalare Datentypen

```
#version 330 core
void main(void){

    // Deklaration und Initialisierung einiger skalarer Typen
    bool done = false;                      // Boolean
    int price = 90;                         // Integer
    uint possiblyHigherPrice = 90000;        // Unsigned Integer
    float cost = 90.0;                       // Float (Standard)
    double moreCost = 90000.0LF;             // Double (Genauer, oft unnötig)

    ...

    // Type Conversion
    int exactValue = 3;
    float PI = float(exactValue); // Cast Integer -> Float

}
```

Vektorielle Datentypen

```
// Deklaration und Initialisierung einiger vektorieller Typen
vec3 a = vec3(0.9, 0.0, 0.0);           // 3-float Vektor
vec4 b = vec4(0.0, 0.0, 0.0, 9.0);     // 4-float Vektor
ivec3 c = ivec3(0, 9, 0);             // 3-int Vektor
vec4 d = vec4(1.0);                  // 4-float Vektor, alle Komponenten 1
vec4 e = vec4(b);                   // initialisiere e mit Werten von b
...
// Zugriff auf Komponenten mit [xyzw], [rgba] oder [stpq]
e.y          // Liefert zweite Komponente von e
e.g          // Liefert ebenfalls zweite Komponente von e
e.b = 2.0;    // Setzt dritte Komponente von e auf 2.0
e.yz         // Liefert vec2(e.y, e.z)
e.zyx        // Liefert vec3(e.z, e.y, e.x)
```

Matrix Datentypen

```
// Deklaration und Initialisierung einiger Matrix Typen
// Setzt eine 2x3 Matrix in Column-Major-Order
mat2x3 aMatrix = mat2x3(1.0, 2.0, 3.0, 4.0, 5.0, 6.0);

// mat2x3 aMatrix hat also die Form: 
$$\begin{pmatrix} 1.0 & 4.0 \\ 2.0 & 5.0 \\ 3.0 & 6.0 \end{pmatrix}$$

// weitere float Matrizen:
mat2 (auch: mat2x2), mat3x2, mat3, mat3x4, mat4x3, mat4

// Zugriff auf Komponenten
aMatrix[0]           // Liefert Spalte 0 der Matrix: vec3(1.0, 2.0, 3.0)
aMatrix[1][2]         // Liefert Element[1][2] der Matrix: 6.0
aMatrix[1] = vec3(7.0, 8.0, 9.0);      // Setzt Spalte 1 der Matrix

// aMatrix hat dann die Form: 
$$\begin{pmatrix} 1.0 & 7.0 \\ 2.0 & 8.0 \\ 3.0 & 9.0 \end{pmatrix}$$

```

Operatoren

```
// Typische Operatoren wie +, -, *, /, ++, %, ==, <, &, |, ! verhalten
// sich bei Skalaren wie von Java gewohnt.

// Anwendung bei Vektoren / Matrizen gleicher Größe komponentenweise
vec3 a = vec3(1.0, 0.0, 0.0), b = vec3(0.0, 1.0, 0.0), c;
float s = 0.5;
c = a + b; // ergibt c = vec3(a.x+b.x, a.y+b.y, a.z+b.z);
c = a * b; // ergibt c = vec3(a.x*b.x, a.y*b.y, a.z*b.z);
c = a * s; // ergibt c = vec3(a.x * s, a.y * s, a.z * s);
c = a + s; // ergibt c = vec3(a.x + s, a.y + s, a.z + s);

// Ausnahme: * ist bei 2 Matrizen Vektor * Matrix die bekannte
// Matrizenmultiplikation
Mat2x3 x; Mat3x4 y;
x * y; // Ergibt mat2x4
y * x; // Nicht möglich
```

Flow Control

```
// if, if-else und switch, wie aus Java bekannt,  
// aber oft für GPUs aufwendig  
if(condition)  
    do something;  
else  
    do something else;  
  
// Ebenso for, while und do Schleifen. Es dürfen keine Variablen im  
// Körper deklariert werden.  
for(int i=0; i<10; i++){  
    do something;  
}  
  
// Funktionen ähneln Java Methoden, erlauben aber keine Rekursion.  
float product(float a, float b){  
    return a * b;  
}
```

Build-In Functions

```
// GLSL beinhaltet einige mathematische Funktionen, die effizient
// implementiert sein sollten. I.d.R. auch vektoriell. Beispiele:
float sin(float radians)          // Trigonometrische Funktion(en)

vec4 cos(vec4 radians)           // Berechnet komponentenweise den Kosinus

float pow(float x, float y) // berechnet x^y

float sqrt(float x)            // Berechnet Wurzel aus x

float length(vec4 x)          // Betrag eines Vektors

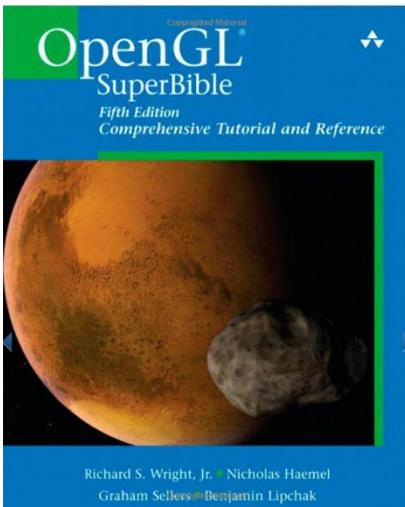
float dot(vec4 a, vec4 b)      // Skalarprodukt aus a und b

vec4 normalize(vec4 x)         // Liefert normalisierten Vektor

vec3 cross(vec3 a, vec3 b)     // Kreuzprodukt a x b (nur für vec3)

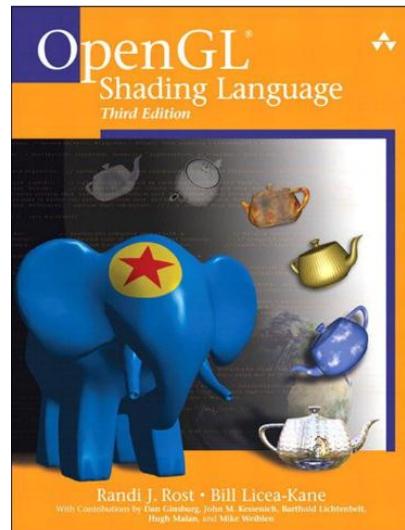
mat4 inverse(mat4 a)           // Inverse Matrix, ab GLSL 1.50 / GL 3.2
```

Literatur OpenGL Programmierung



Wright, Haemel, Sellers,
Lipchak
OpenGL SuperBible
Addison-Wesley 2010
(verfügbar unter Safari)

[Safari Link](#)



Rost, ...
**OpenGL Shading
Language**
Addison-Wesley
2009
(verfügbar unter
Safari)

Links OpenGL Programmierung

- Tutorials, z.B. auf lwjgl.org, meist veraltet
- OpenGL 4.2 Specification
 - <http://www.opengl.org/registry/doc/glspec42.core.20110808.pdf>
- OpenGL Shading Language (GLSL) 4.20 Specification
 - <http://www.opengl.org/registry/doc/GLSLangSpec.4.20.6.clean.pdf>
- OpenGL 3.3 Reference Pages
 - <http://www.opengl.org/sdk/docs/man3/>
- GLSL 4.2 Reference Pages
 - <http://www.opengl.org/sdk/docs/manglsl/>
- OpenGL 4.2 Reference Pages
 - <http://www.opengl.org/sdk/docs/man4/>
- Quick Reference Card (blaue Befehle ignorieren!)
 - <http://www.khronos.org/files/opengl42-quick-reference-card.pdf>