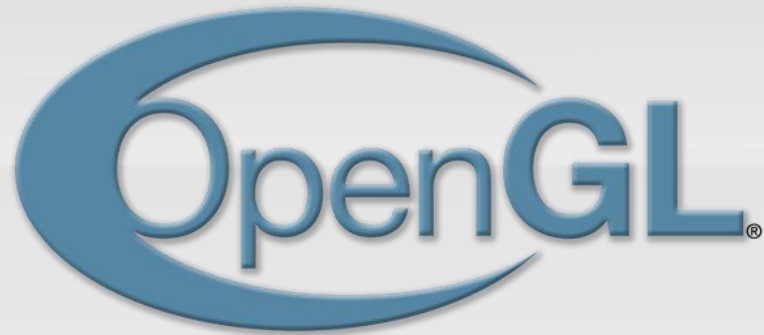




Eine Einführung

Computergrafik SS 2016

Timo Bourdon

Kapitel 21

Organisatorisches

Zum OpenGL-Übungsblatt

- OpenGL 3.0 oder höher notwendig (Shading Language 1.50 oder höher)
- CIP Pool (Nvidia GeForce 9300, GeForce GTS 450)
- Wenn es gut läuft:

```
+-- OpenGL info -----+
| AMD Radeon HD 7470M, ATI Technologies Inc., Driver: 8.15.10.2509 |
| OpenGL 3.2.11247 Core Profile Forward-Compatible Context - Shading Language 4.10 |
| Operating system: Windows 7 (6.1), amd64 |
| Java: Java HotSpot(TM) 64-Bit Server VM, runtime version: 1.7.0_51-b13 |
+-----+
Vertex shader was successfully compiled to run on hardware.
Fragment shader was successfully compiled to run on hardware.
Vertex shader(s) linked, fragment shader(s) linked.
```

```
org.lwjgl.LWJGLException: Could not create context (WGL ARB create context)
at org.lwjgl.opengl.WindowsContextImplementation.nCreate(Native Method)
at org.lwjgl.opengl.WindowsContextImplementation.create(WindowsContextImplementation.java:50)
at org.lwjgl.opengl.ContextGL.<init>(ContextGL.java:132)
at org.lwjgl.opengl.Display.create(Display.java:850)
at org.lwjgl.opengl.Display.create(Display.java:797)
at main.helloTriangles.init(helloTriangles.java:175)
at main.helloTriangles.main(helloTriangles.java:221)
```

Motivation

Was nach der Vorlesung nicht möglich sein wird...



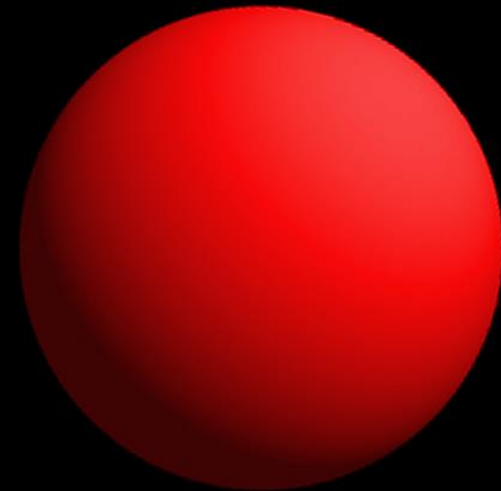
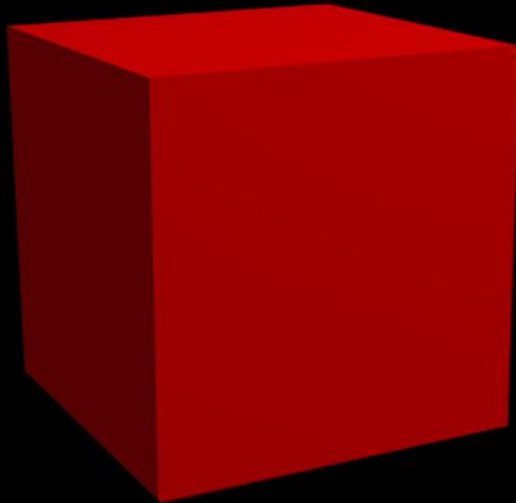
Motivation

... das auch nicht...



Motivation

...aber wir legen die Grundsteine dafür!



Einführung

OpenGL - Definition

API für Rastergrafik

- Prozedural
- Hardwarenah
- Plattform-, Betriebssystem- und sprachunabhängig

Spezifikationen variieren pro Version im Funktionsumfang

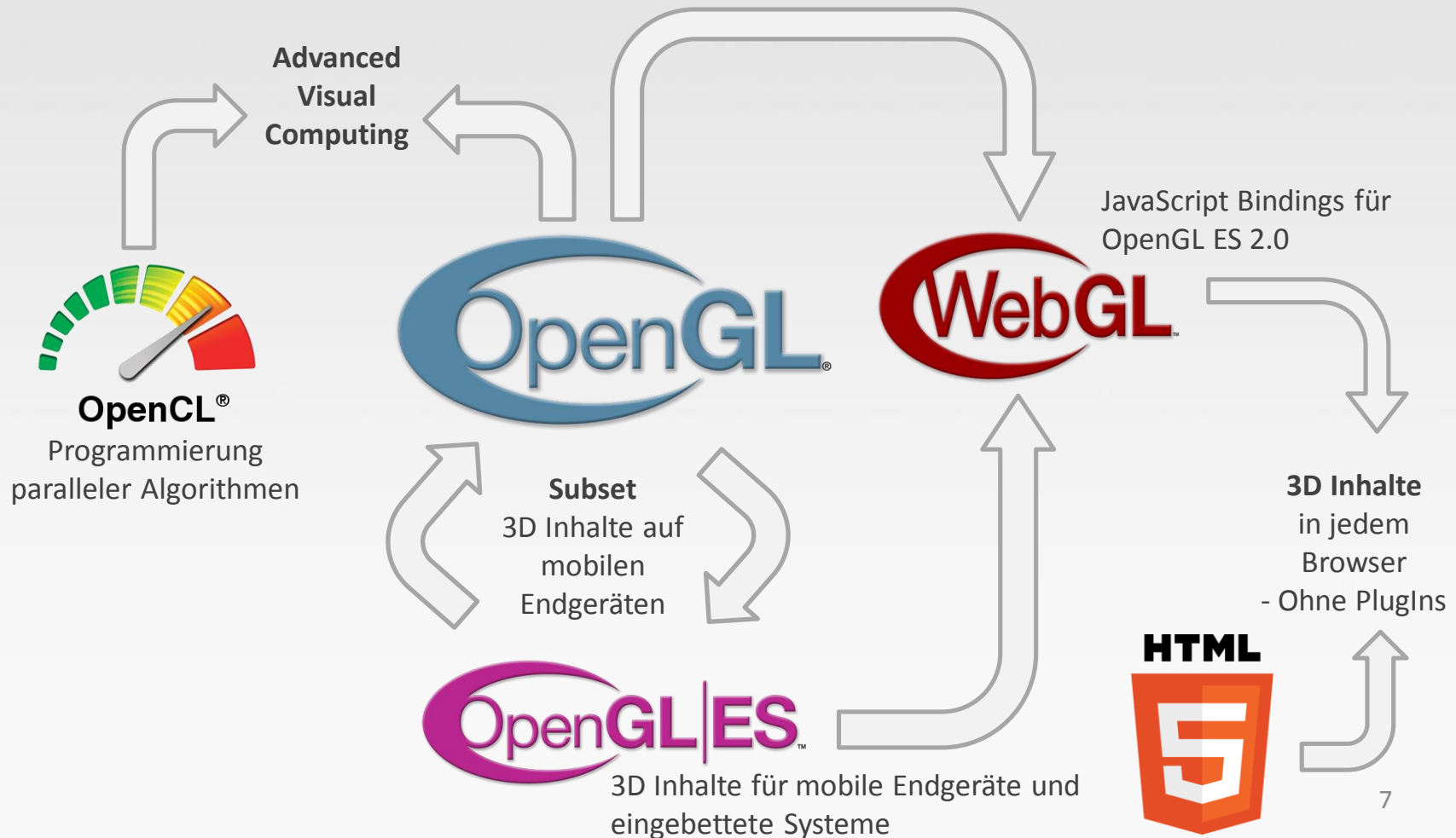
- OpenGL (Open Graphics Library)
- Die Anwendungen basieren hier auf OpenGL 3.0 Core Profile

Verwandte APIs



Einführung

API Interaktion



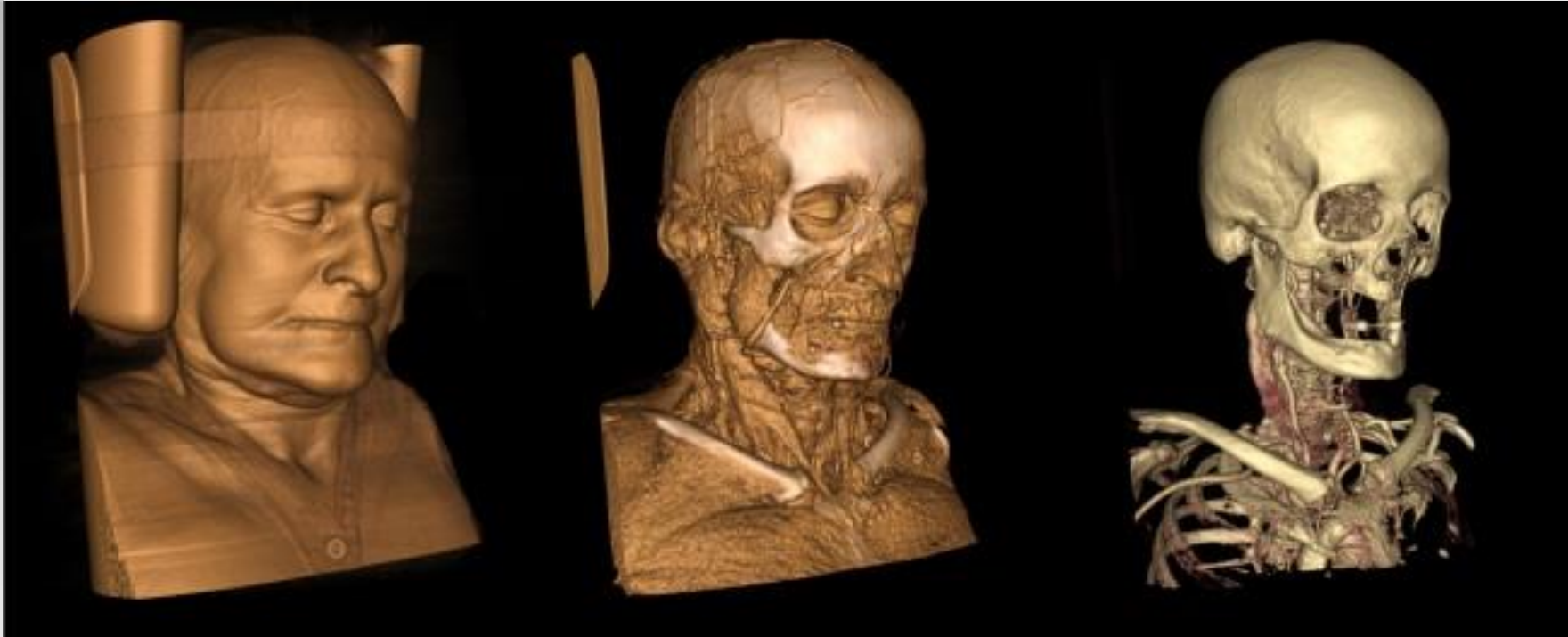
Einführung

Anwendungsbeispiele – Videospiele



Einführung

Anwendungsbeispiele - Medizin



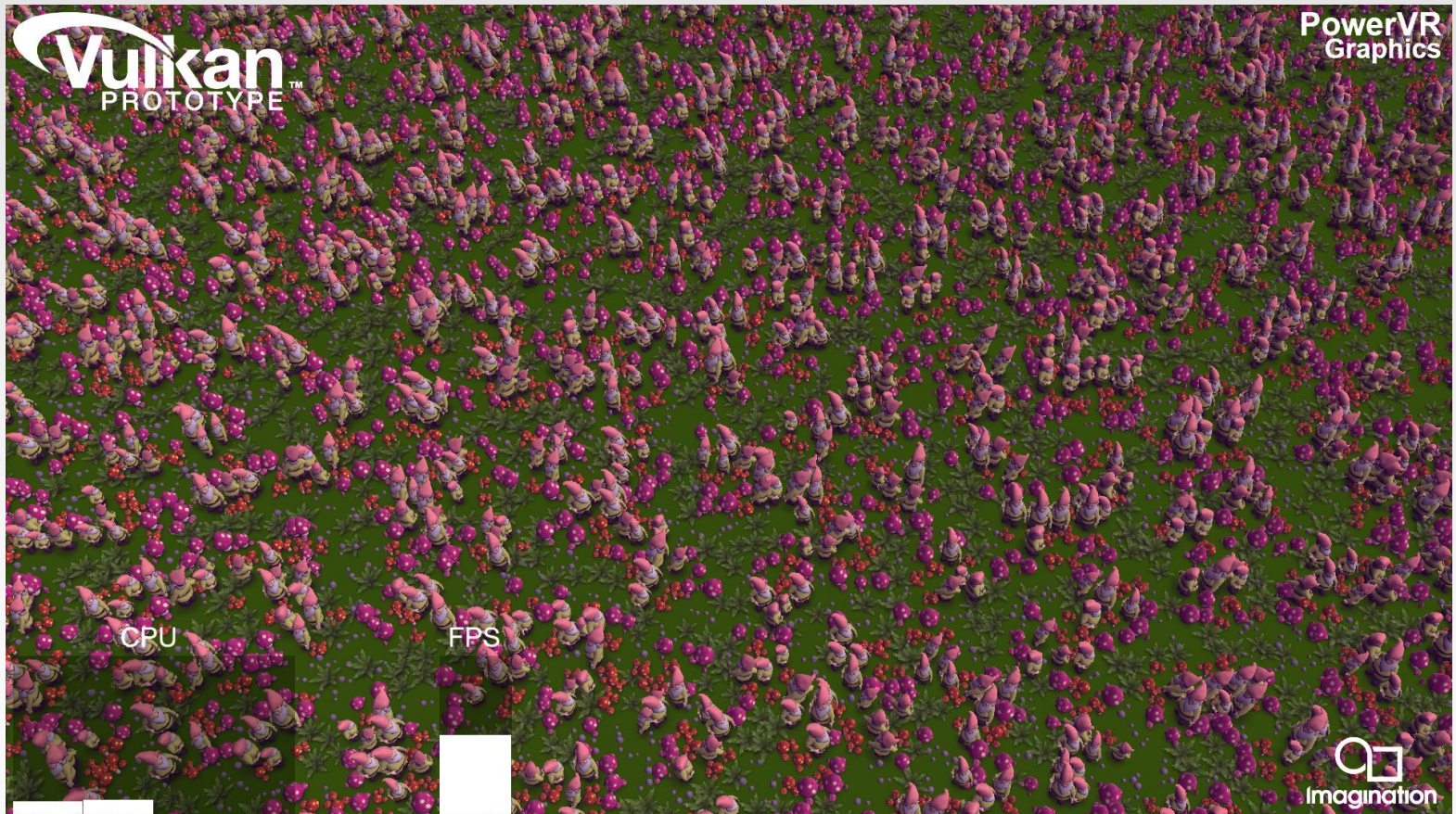
Einführung

Anwendungsbeispiele



Einführung

Anwendungsbeispiele



Einführung

Anwendungsbeispiele



Einführung

Entwicklungsgeschichte



Einführung

Entwicklungsgeschichte - Versionen

OpenGL 1.x (ab 1992)

- Feste, konfigurierbare Pipeline
- Spätere Versionen bringen weitere „Optionen“

OpenGL 3.1 (03-2009)

- Legacy Funktionen aus Kern entfernt
- aufwärtskompatibel

OpenGL 3.2 (09-2009)

- Geometry Shader

OpenGL 3.3 & 4.0 (03-2010)

- Tessellation (ab 4.0)
- Bessere Zusammenarbeit mit OpenCL

OpenGL 2.x (ab 2004)

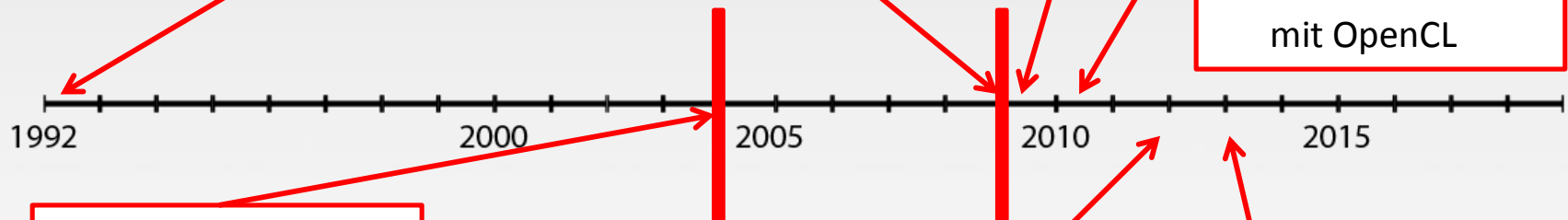
- Einführung programmierbarer Stages (VS / FS)
- Feste Pipeline bleibt parallel bestehen

OpenGL 4.3 (08-2012)

- Compute-Shaderprogramme
- Plattformübergreifende Texturkompression
- Stabilität

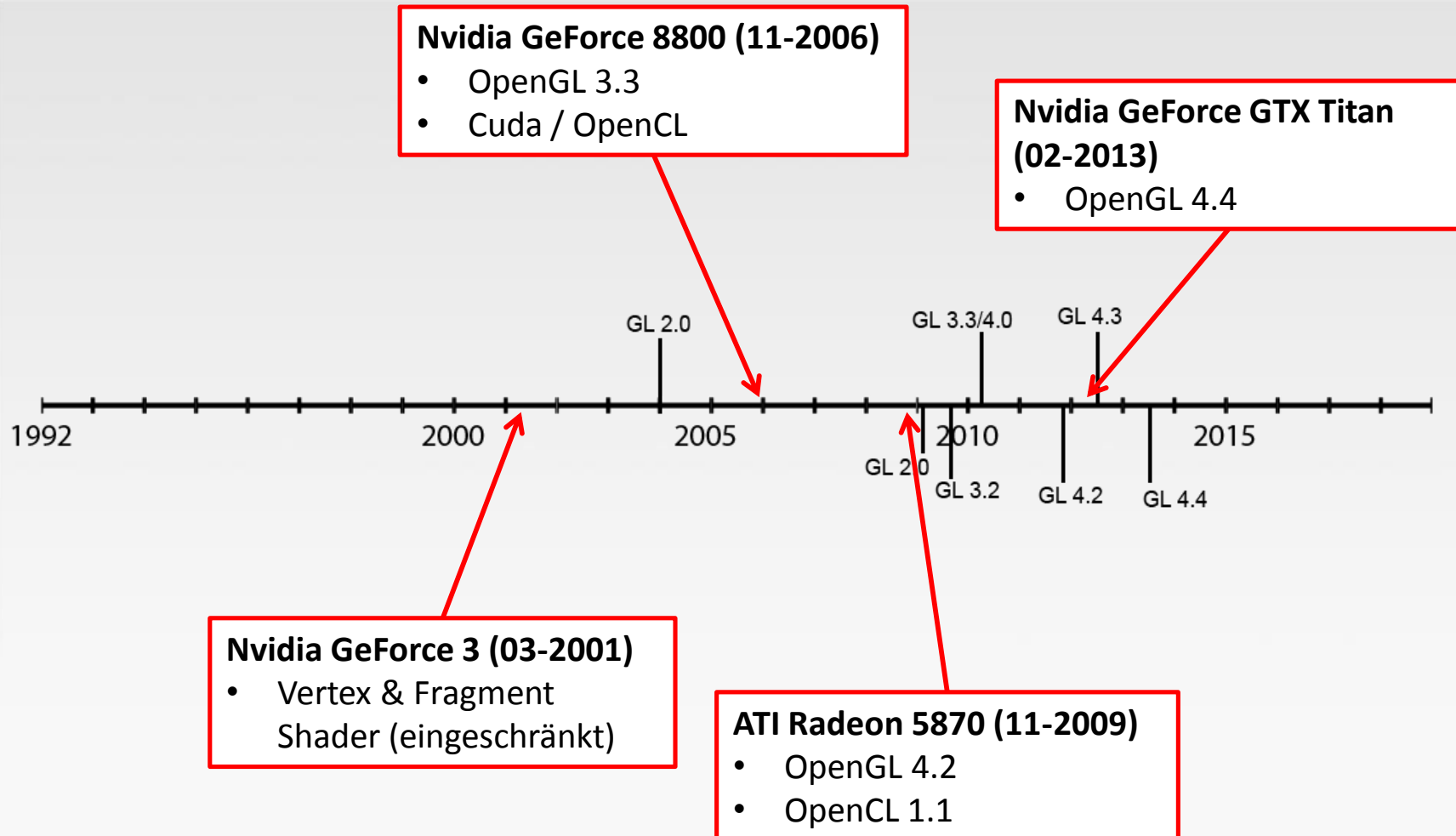
OpenGL 4.4 (07-2013)

- Mehrere OpenGL Objekte an einem Kontext
- Kontrolle der Buffer-Platzierung im Speicher



Einführung

Entwicklungsgeschichte - Hardware



Einführung

Zielsetzung

- **Hardwarenah aber unabhängig**
- **Verschiedene Anwendungsbereiche**
 - Wissenschaft, Visualisierung & Entwicklung (CAD)
 - Spiele-Entwicklung
 - Grafik im Web und auf Smartphones
- **Betriebssystem- & Plattformunabhängig**
 - Windows, Linux, Mac,...
 - PCs, Notebooks, Smartphones, Tablets,...
- **Aus vielen Programmiersprachen verwendbar**
 - C/C++, Java
 - PHP, JavaScript (speziell WebGL)

Einführung

Aufgaben von OpenGL

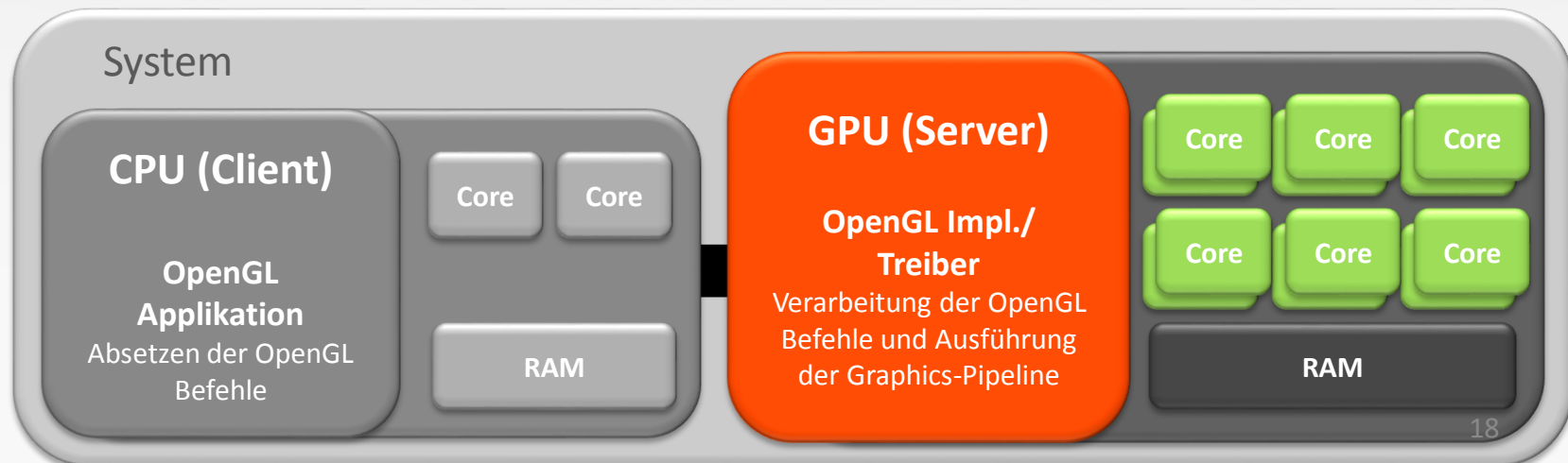
- Verwaltung der Graphics Pipeline
- Freie Stages werden vom Entwickler programmiert
- Feste Stages werden vom Entwickler konfiguriert
- Erzeugen, übersetzen, etc. der Shader-Programme
- Kontrolle des Datenflusses

Einführung

Client-Server & State Machine

- **Client-Server Modell**

- Client, die Java Applikation, setzt OpenGL Befehle ab
- Server, OpenGL Implementation der Grafikkarte + Treiber, führt diese aus
- Client und Server typischerweise in einem Rechner
 - Rendering beim Client vs. Remote Rendering
 - Trotzdem i.d.R. kein gemeinsamer Speicher



Einführung

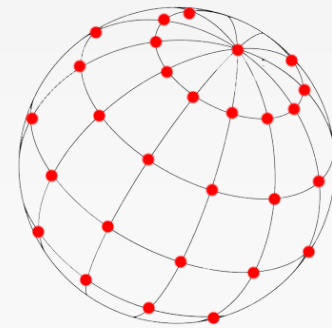
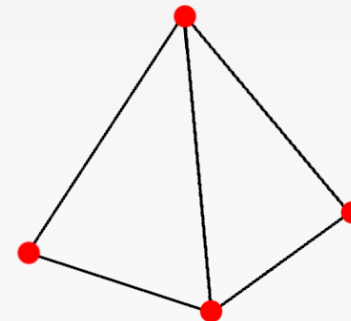
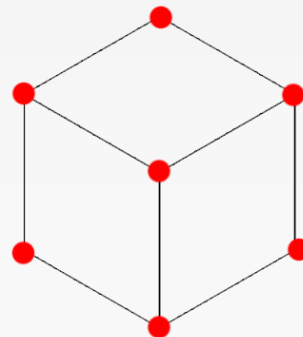
Client-Server & State Machine

- Zustandsmaschine
 - Minimierung der Client-Server Kommunikation
 - Gesetzte Zustände, etwa die Hintergrundfarbe, bleiben bis zum Widerruf gültig
 - State-Variablen sind u.a. an den Aufrufen `glEnable()` oder `glDisable()` zu erkennen

Technik

Vertex

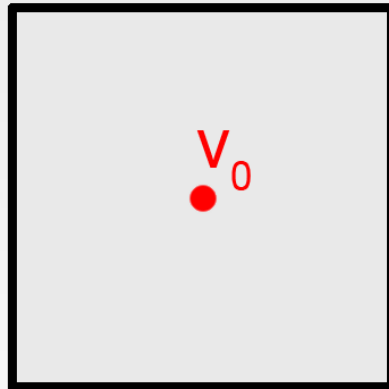
- **Allgemein: Mathematischer Punkt im Raum**
- **Speziell: „Eckpunkt“ einer geometrischen Figur**
- **Enthält oft weitere Eigenschaften an diesem Punkt**
 - Normale
 - Farbe
 - Texturkoordinaten
 - Geschwindigkeit
 - Beschleunigung
 - Materialdichte
 - etc.



Technik

Primitive

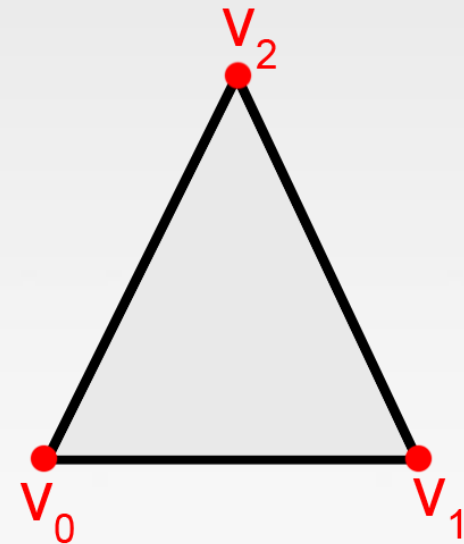
- Elementare grafische Grundform“
- Besteht aus 1 – 3 Vertices
- Topologische Information



Punkt



Linie

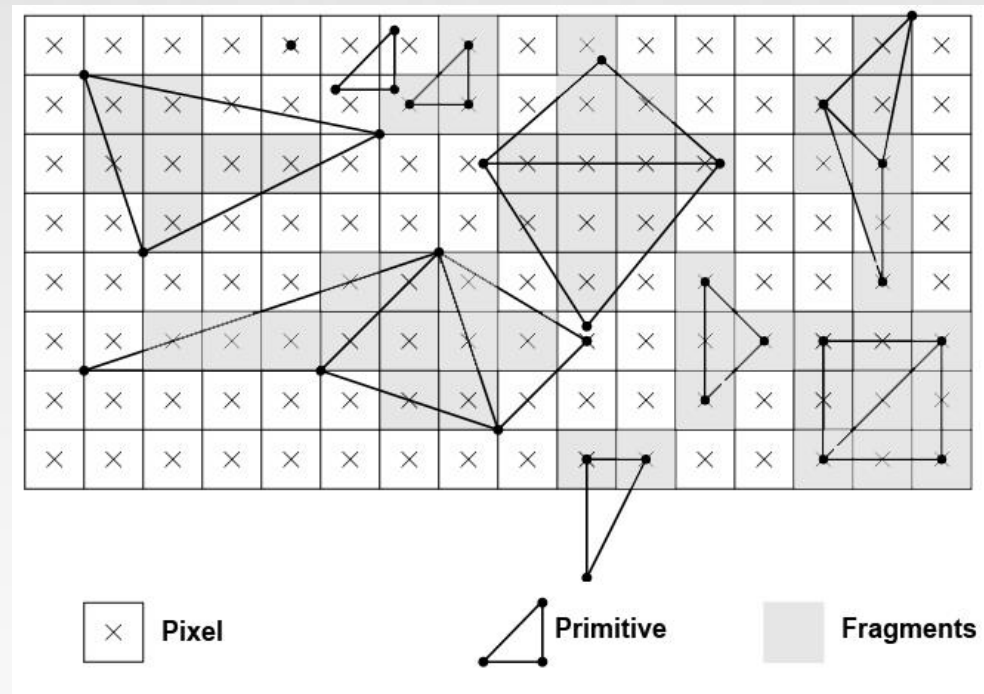


Dreieck

Technik

Fragment

- Vom Rasterizer aus Primitive für einen Pixel erzeugte Datenstruktur



- Enthält zunächst für diese Stelle interpolierte Daten der zugehörigen Vertices (Position, Normale, Tiefe)
- Pixelvorstufe**

Technik

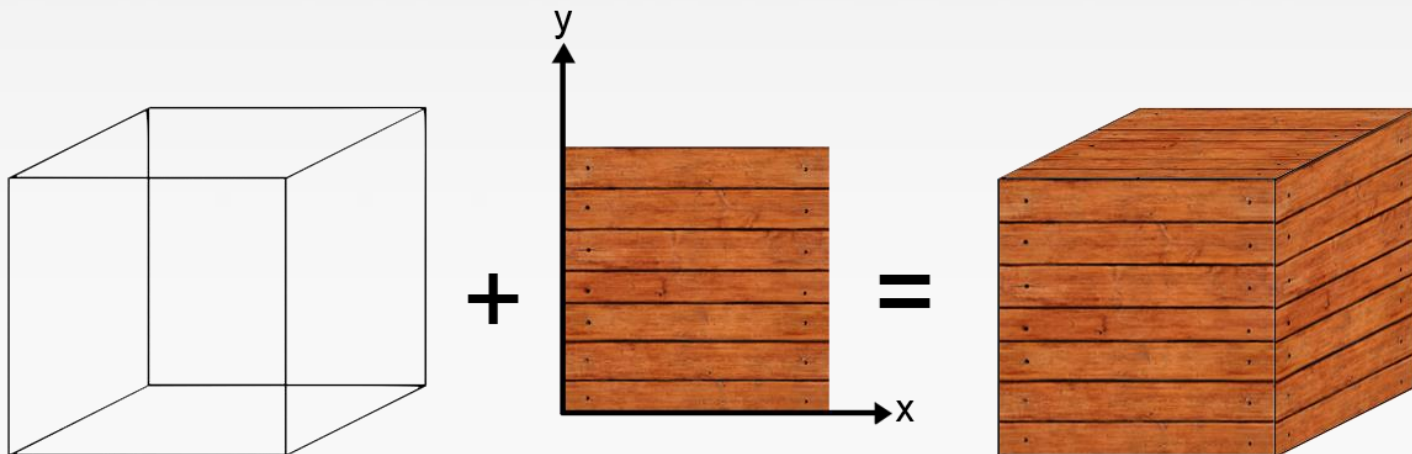
Uniform Data

- **In Shadern lesbare, globale Variablen, welche aus der Applikation heraus geschrieben wird**
- **Für alle Vertices und Fragments während eines Durchlaufs der Graphics-Pipeline konstant**
- **In der Regel sind dies:**
 - Matrizen (Modelling-Transformations, Projektionen, etc.)
 - Kameradaten
 - Lichtquellen (als Vektoren beschrieben)
 - Farben und Reflektion

Technik

Texture

- Ein- bis dreidimensionale Datenstruktur
- Texel (Texture-Element) ein- bis vierdimensional
- Enthält beliebige numerische Informationen
 - Farbe, Normale, Dichte, Geschwindigkeit, etc.
 - Informationsdichte oft höher als Geometrieauflösung



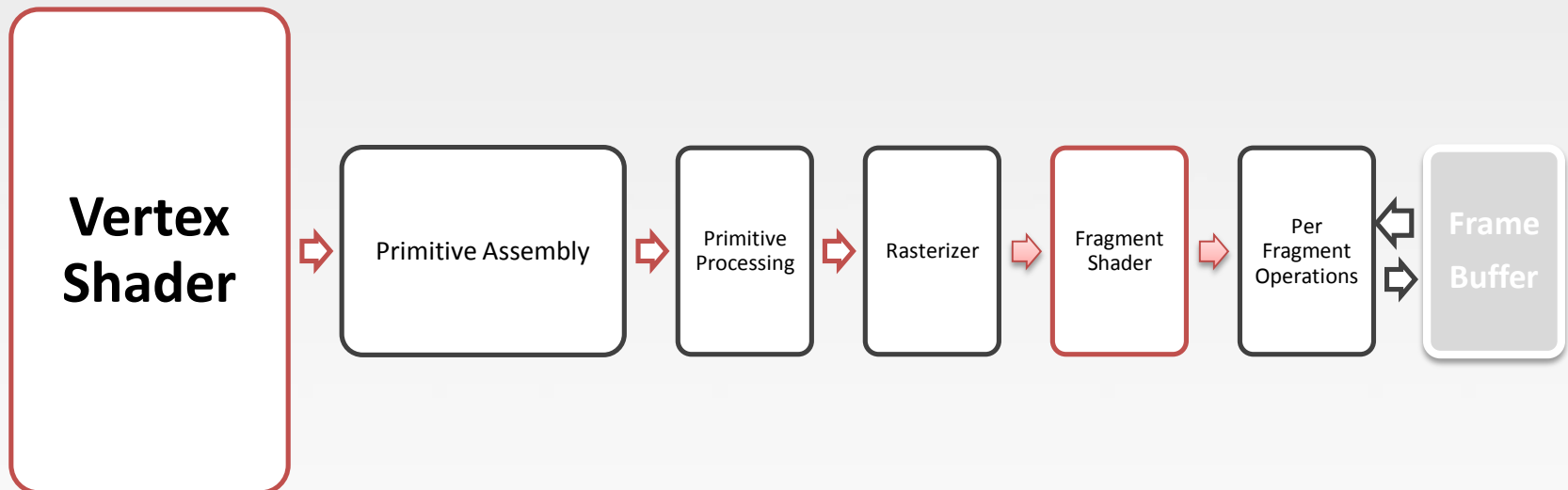
Technik

Shader

- Programm zur Beschreibung / Berechnung / Transformierung von Geometrie, Licht, Schatten, Oberflächeneigenschaften, etc.
- Shading Languages (GLSL, HLSL,...)
- Auf der Grafikkarte ausführbare Programme
- Prozedural
- Angelehnt an Hochsprache C mit Vektor- und Matrixdatentypen
- Hier: Beschränkung auf Vertex- und Fragment Shader

Die OpenGL-Graphics-Pipeline





Vertex Shader

Definition

- **Programm**
- **Wird unabhängig für jeden Vertex einer Geometrie ausgeführt**
- **Verarbeitung der Daten des Vertex (Vertex Attribute)**
 - Position im Koordinatensystem
 - Normale
 - Farbe
 - etc.

Vertex Shader

Beispiel 1

```
#version 330

float getAB(float a, float b){
    return a * b;
}

void main(){
    float a= 5.0;
    a = getAB(10.0, 1.5);
    vec3 vectorA = vec3(1.0, 0.0, 0.0);
    vec3 vectorB = vec3(1.0, 1.0, 0.0);
    vec3 kp = cross(vectorA, vectorB);
    float sp = dot(vectorA, vectorB);
    vec3 vecMul = vectorA * vectorB;
    vec3 vecAdd = vectorA + vectorB;

    for (int i = 0; i < 5; i++){
        if(a < 1000)
            a *= 2;
    }
}
```

Vertex Shader

Beispiel 2

```
#version 330

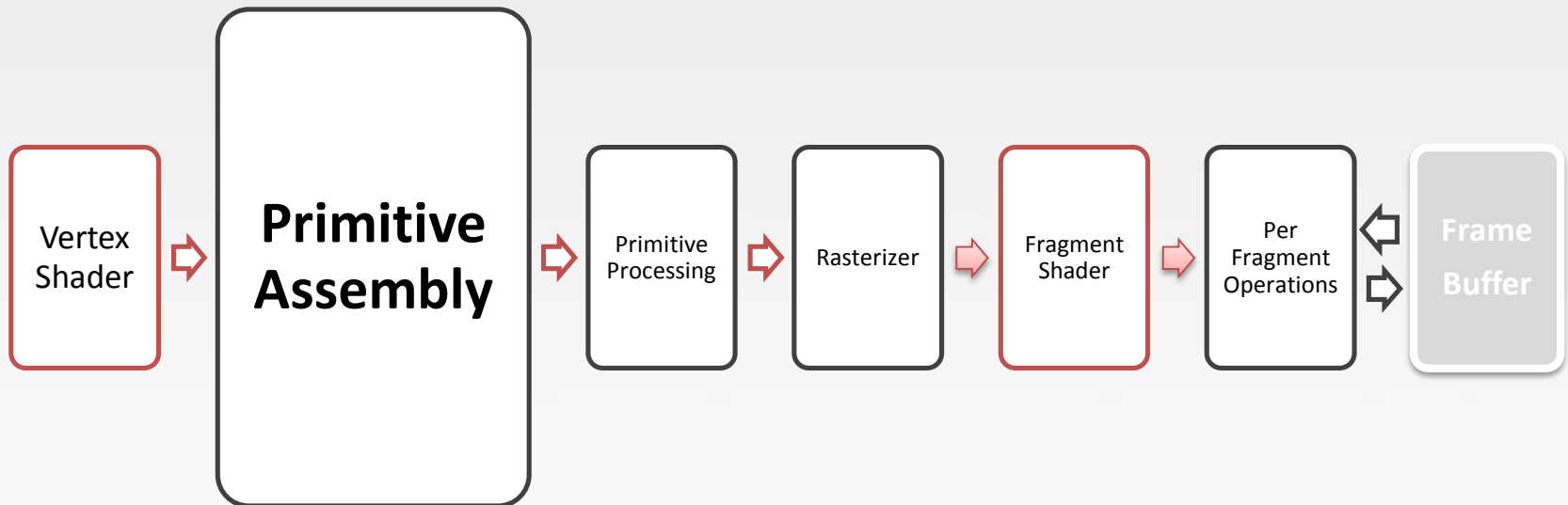
uniform mat4 translate;
uniform mat4 rotation;
uniform mat4 scale;

in vec3 vs_in_pos;

void main() {

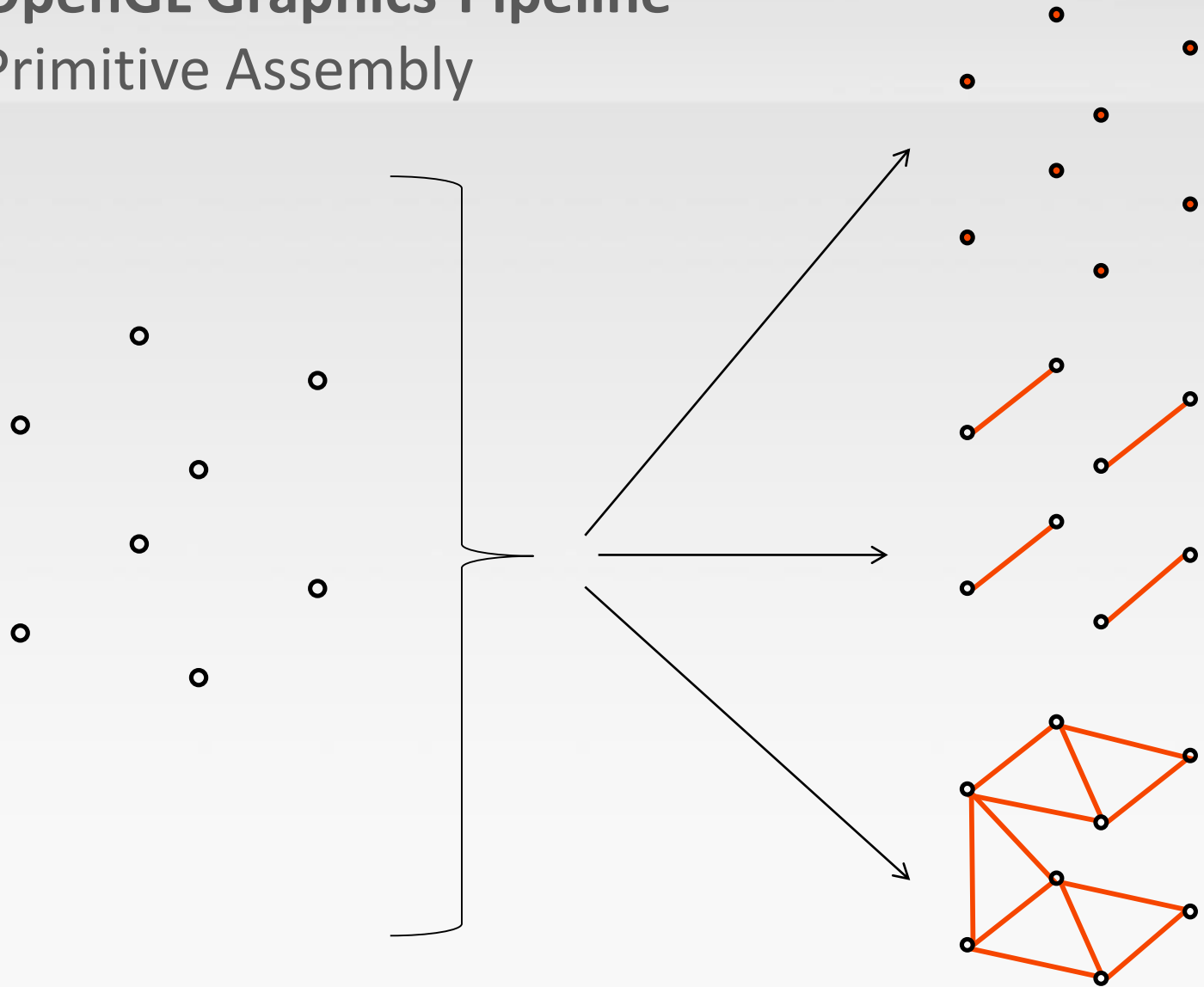
    gl_Position =
        translate * rotation * scale * vec4(vs_in_pos, 1);

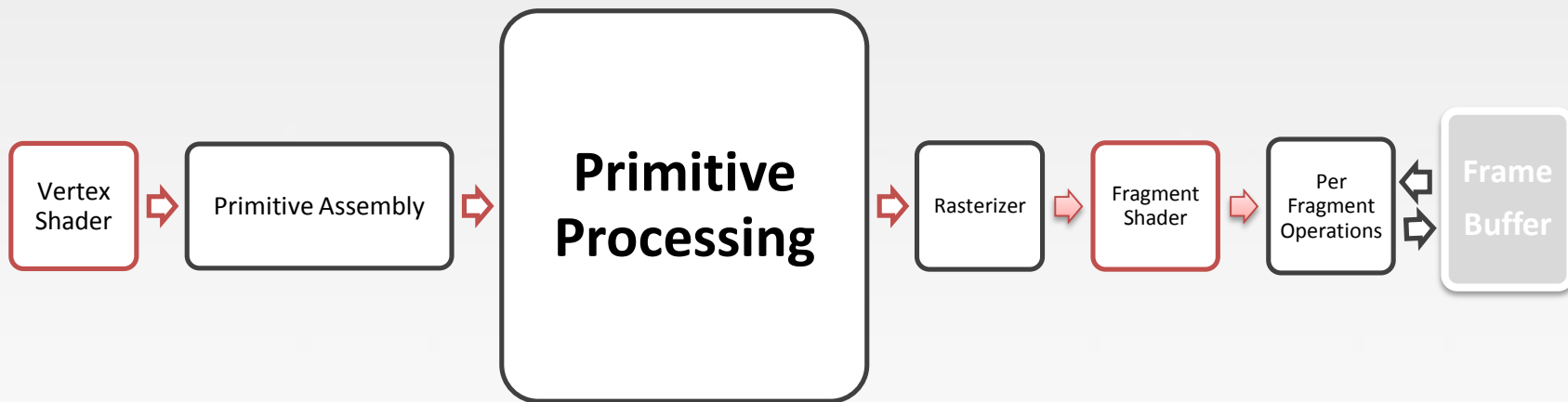
}
```



OpenGL Graphics-Pipeline

Primitive Assembly

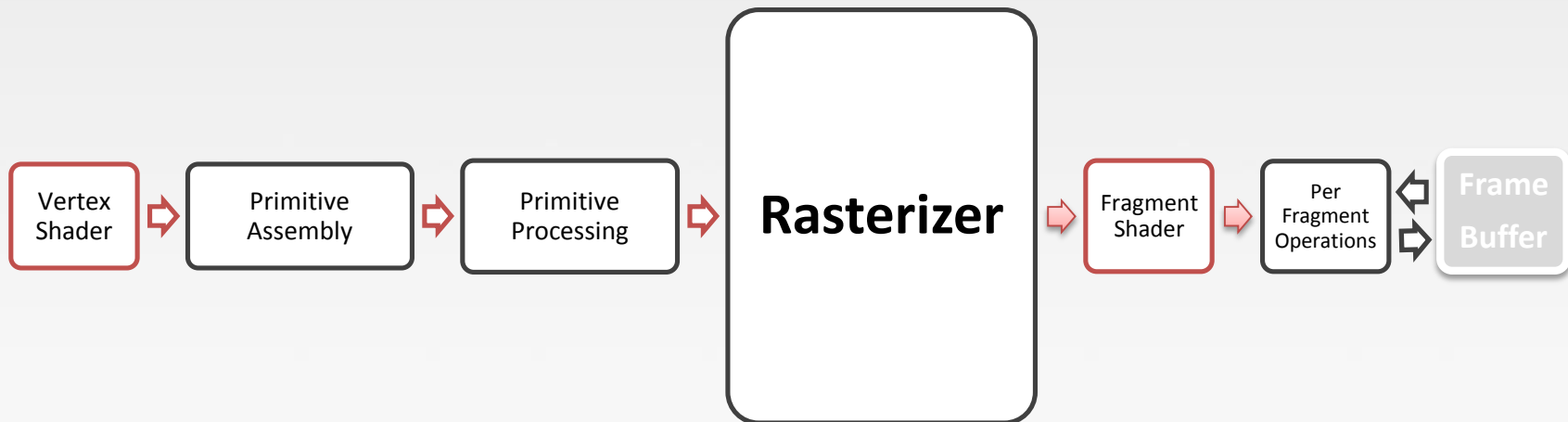




OpenGL Graphics-Pipeline

Primitive Processing

- Operationen, die Information über ganzes Primitive benötigen
 1. Clipping
 2. **Perspective Division**
 - Transformation des sichtbaren Teils in den Einheitswürfel
 3. **Viewport Transformation**
 - Mapping auf das Ausgabefenster
 4. Culling



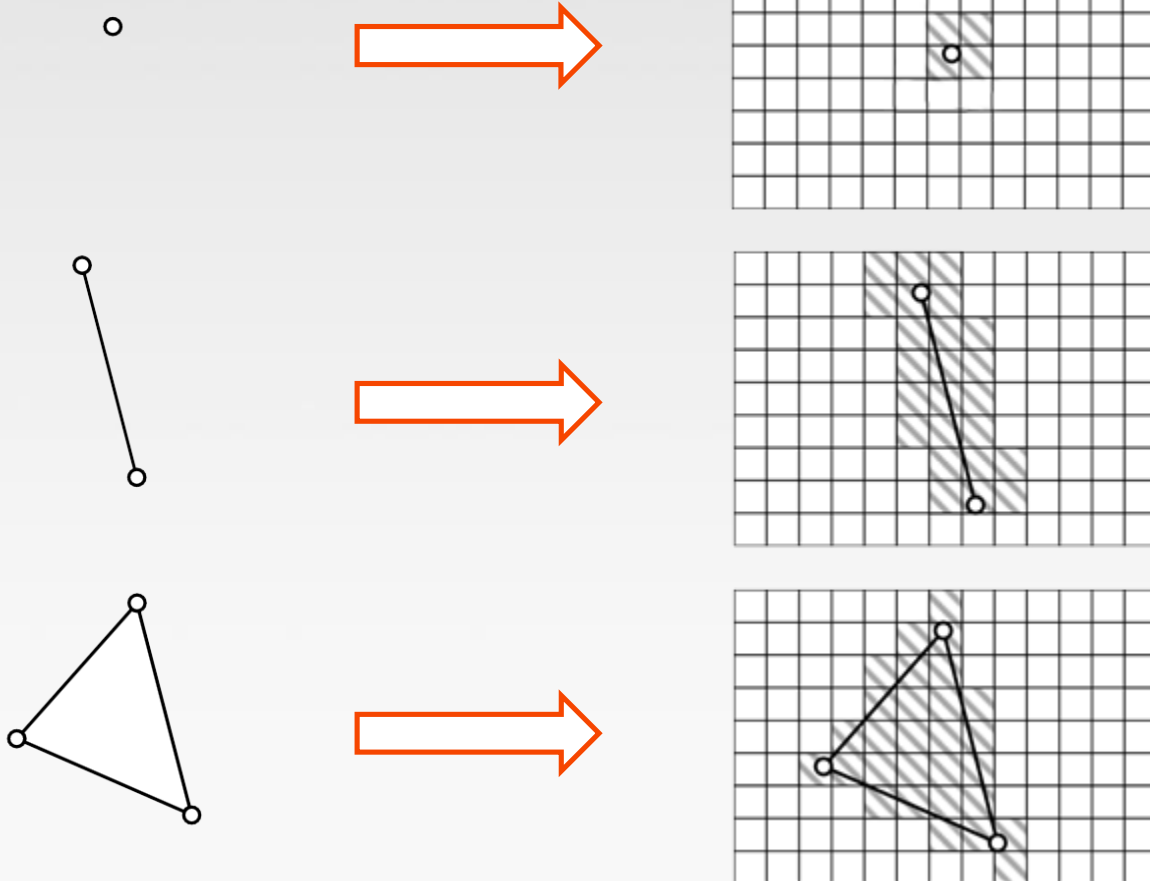
OpenGL Graphics-Pipeline

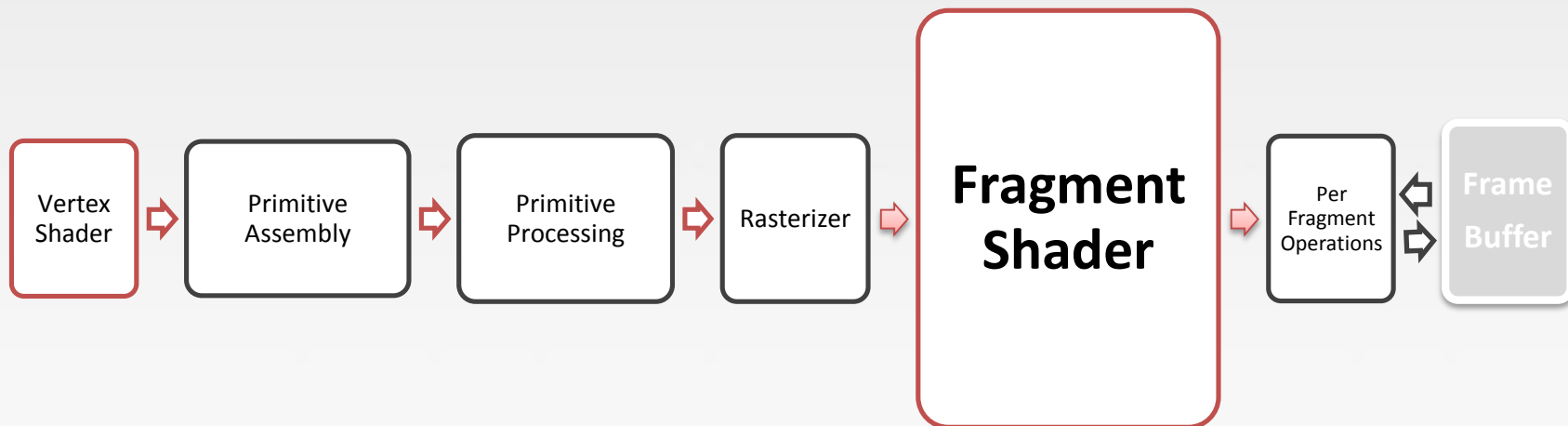
Rasterizer

- Überführung von Primitives in Fragments
- Für jedes von einem Primitive überlappte Pixel wird ein Fragment erzeugt
- **Fragment**
 - Korrespondierende 2D-Komponente eines Pixels
 - Enthält zusätzlich **Tiefeninformation**

OpenGL Graphics-Pipeline

Rasterizer





Fragment Shader

Definition

- **Programm**
- **Verarbeitet durch Rasterizer fabrizierte Fragments**
 - Festlegung der Farbe eines Fragments
 - Texturberechnungen
 - Auswertung eines Beleuchtungsmodells (Phong Shading)

Fragment Shader

Beispiel

```
#version 330 VS

uniform mat4 translate;
uniform mat4 rotation;
uniform mat4 scale;

in vec4 vs_in_pos;
in vec2 myTexCoords;

out vec2 texCoords;

void main() {

    gl_Position = . . .

    texCoords = myTexCoords;
}
```

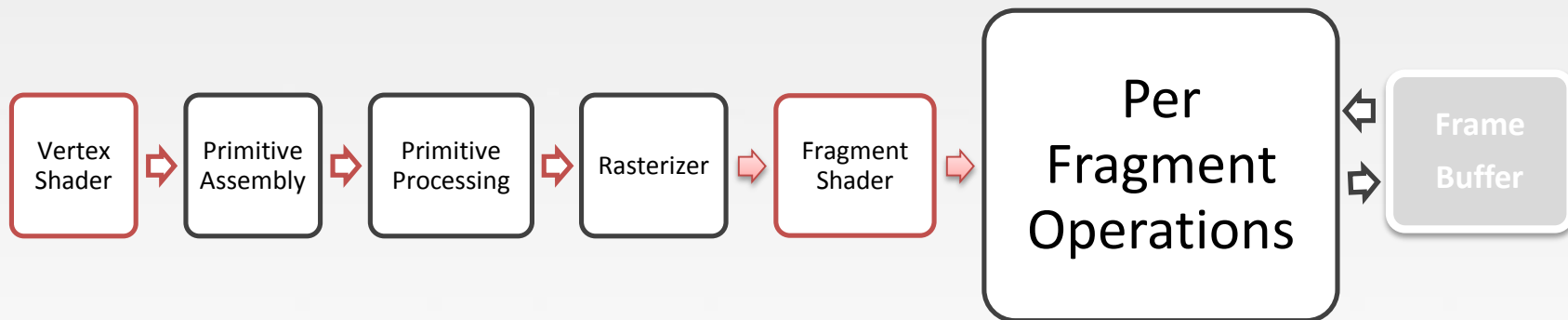
```
#version 330 FS

uniform sampler2D earthTex;

in vec2 texCoords;

void main() {

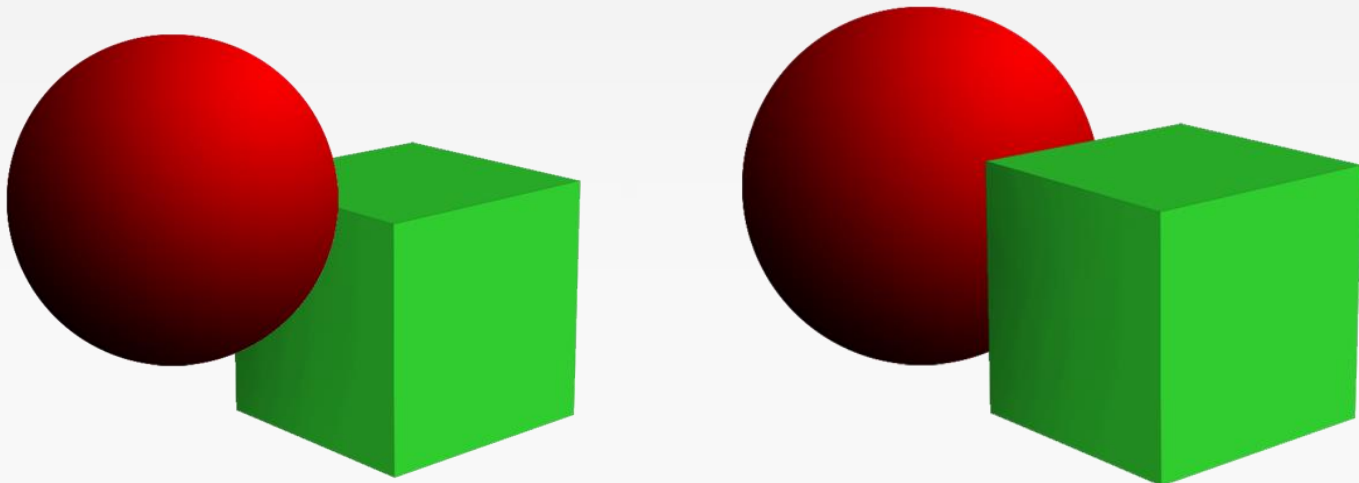
    fragColor = texture2D(earthTex, texCoords);
}
```



OpenGL Graphics-Pipeline

Per Fragment Operations

- **Regelung des Einflusses der Fragments auf das jeweilige korrespondierende Pixel**
 - Pixel können von mehreren Primitives überlagert werden
 - Viele Fragments für ein Pixel
- **Tiefentest**
 - Vorderstes Fragment rendern (ggf. im FB überschreiben)



OpenGL Graphics-Pipeline

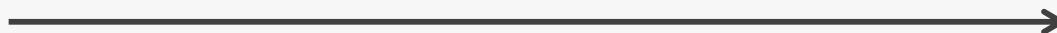
Per Fragment Operations

- Tiefentest

Fragments



Korrespondierendes
Pixel

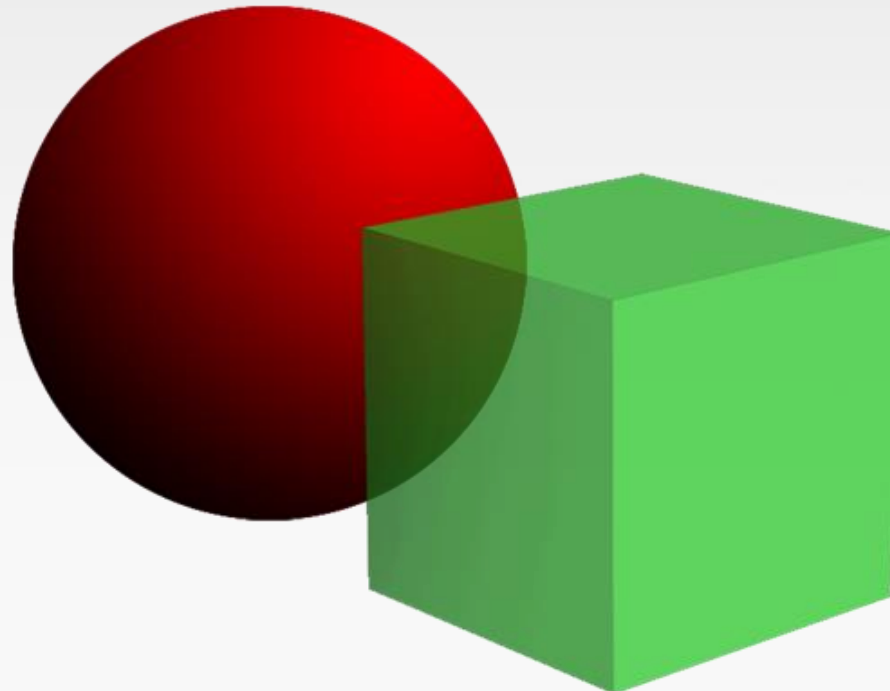


Zeit
Bedingung „>“

OpenGL Graphics-Pipeline

Per Fragment Operations

- Regelung des Einflusses der Fragments auf das jeweilige korrespondierende Pixel
 - Blending Funktion



OpenGL Graphics-Pipeline

Übersicht

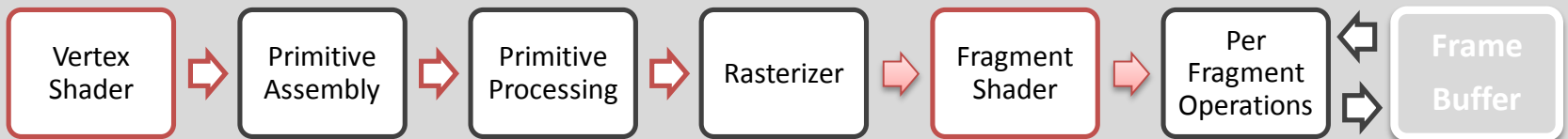
Client Memory (Java Applikation)



OpenGL Befehle



Server Memory (GPU)



Vertices



Fragments



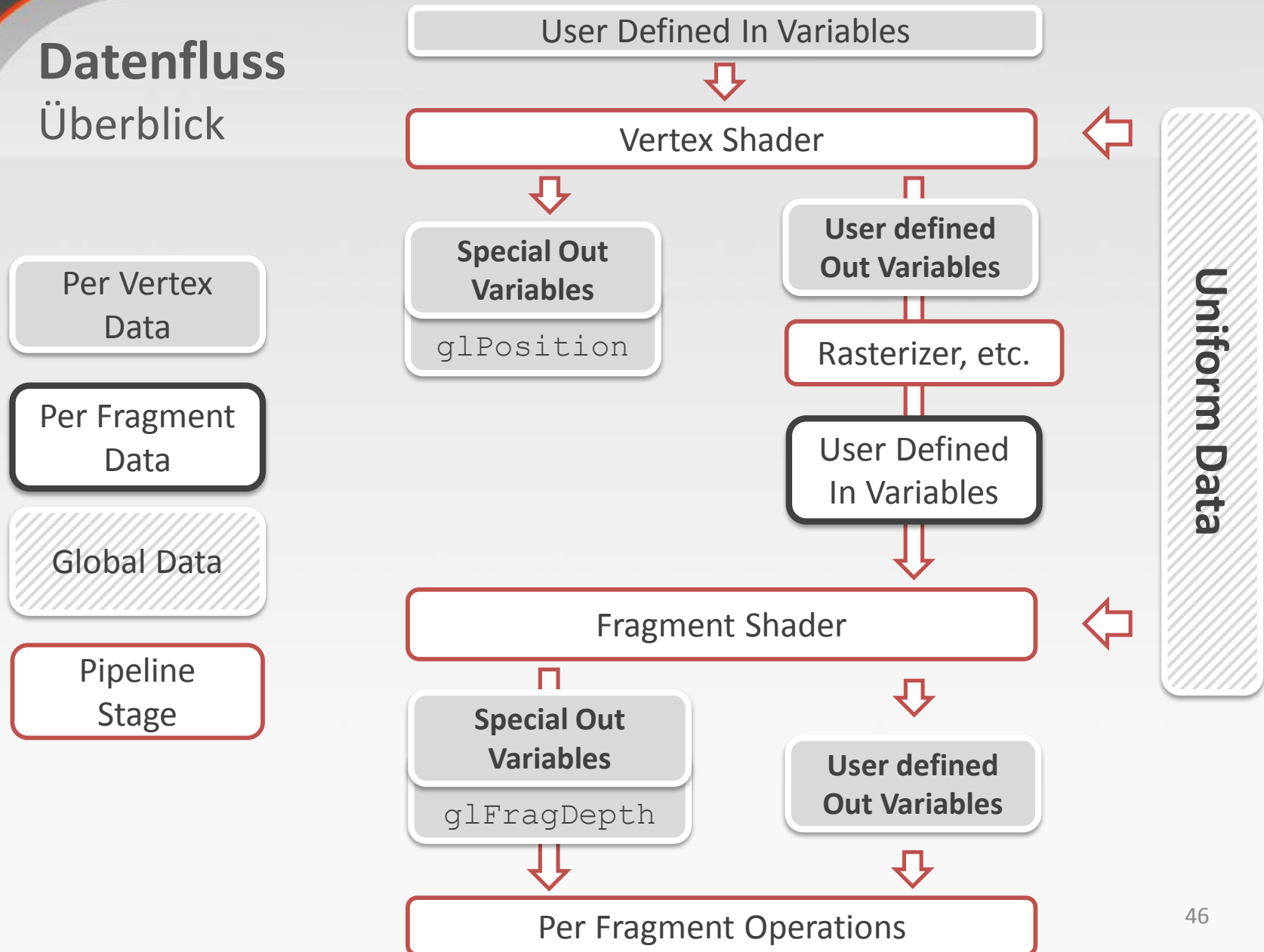
Pixel/Texture Daten

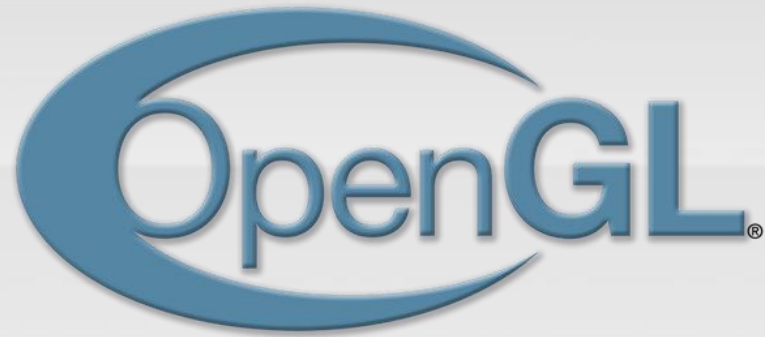
**Programmable
Stage**

**Fixed
Stage**

Memory

Datenfluss Überblick





Spracheigenschaften und Syntax

Spracheigenschaften und Syntax

Aufgaben der GL-Befehle

- **Konfigurieren der Graphics Pipeline**
- **Datenübergabe an den Server (GPU)**
- **Steuerung des Datenflusses**

Spracheigenschaften und Syntax

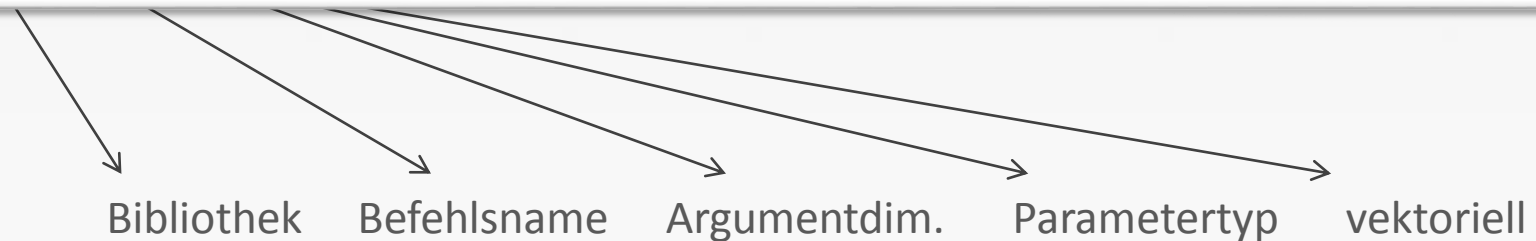
Aufbau der GL-Befehle

Einheitliche Form der Befehle

```
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)

```
glUniform3iv(int location, int v0, int v1, int v2);
```



Spracheigenschaften und Syntax

Aufbau der GL-Befehle

Spezialisierung durch Konstanten

```
glEnable (...);
```

- GL_DEPTH_TEST
- GL_CULL_FACE
- GL_PRIMITIVE_RESTART
- ...

OpenGL spezifiziert eigene Datentypen, zur einheitlichen Verwendung der Befehle aus verschiedenen Sprachen

- Entsprechen C-Datentypen (GLfloat = C-Datentyp float)
- Buffer-Objects statt Pointer (für Datentransfer auf GPU)

Spracheigenschaften und Syntax

Fehlerbehandlung

- Wichtig: OpenGL meldet selbstständig keine Fehler
- OpenGL besitzt ein Error-Flag
- Kann auf eine von 7 Fehlerkonstanten oder initial `GL_NO_ERROR` 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`

Spracheigenschaften und Syntax

Fehlerbehandlung

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

```
//Mögliche Fehlerkonstanten. Jeweilige Bedeutung kann in  
//der Dokumentation, z.B. GL Reference Pages, des  
//verursachenden GL-Befehls nachgeschlagen werden.
```

```
0          GL_NO_ERROR  
0x0500 GL_INVALID_ENUM  
0x0501 GL_INVALID_VALUE  
0x0502 GL_INVALID_OPERATION  
0x0503 GL_STACK_OVERFLOW  
0x0504 GL_STACK_UNDERFLOW  
0x0505 GL_OUT_OF_MEMORY  
0x0506 GL_INVALID_FRAMEBUFFER_OPERATION
```

Spracheigenschaften und Syntax

LWJGL

- Liefert zu jedem OpenGL Befehl einen Java Befehl gleichen Namens, der diesen ausführt
- **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);
```

Spracheigenschaften und Syntax

LWJGL

- Anbindung an das Fenstersystem
- Ermöglichen von Maus- und Keyboard Interaktion

- Code:

C: `glEnable(GL_DEPTH_TEST);`

JOGL: `gl.glEnable(GL11.GL_DEPTH_TEST);`

LWJGL: `glEnable(GL11.GL_DEPTH_TEST);`

Spracheigenschaften und Syntax

CG12 Wrapper

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

```
// 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){

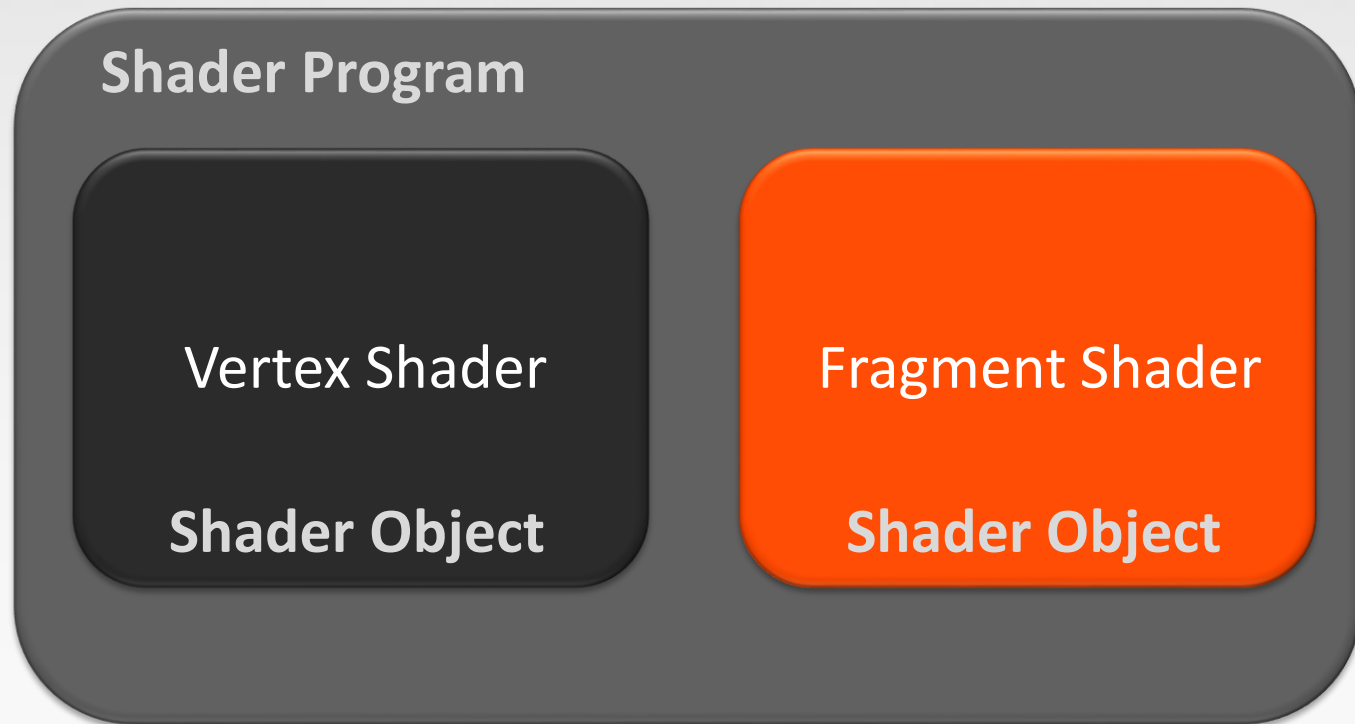
        // Der ruft den original Befehl auf
        glClearColor(...);

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

Spracheigenschaften und Syntax

Erzeugen eines Shader Programs

- Shader Program kapselt programmierbaren Teil der Graphics Pipeline



- Werden zur Laufzeit durch Grafikkartentreiber übersetzt

Spracheigenschaften und Syntax

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);      // Source-Code des Shaders
```

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

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

Spracheigenschaften und Syntax

Erzeugen eines Program Objects

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

```
// 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
```

Spracheigenschaften und Syntax

Erzeugen eines Program Objects

```
// Beispiel: Program Objects erzeugen, bestehend aus VS und FS
int myProgram = glCreateProgram()

// Erzeugen von VS und FS
int myVS = glCreateShader(GL_VERTEX_SHADER);
int myFS = glCreateShader(GL_FRAGMENT_SHADER);

// Shader Quellcode
glShaderSource(myVS, „version...“);
glShaderSource(myFS, „version...“);

// Shader Kompilierung
glCompileShader(myVS);
glCompileShader(myFS);

// Hinzufügen der Shader und Linken
glAttachShader(myProgram, myVS);
glAttachShader(myProgram, myFS);
glLinkProgram(myProgram)
```

Spracheigenschaften und Syntax

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: Verwendung des „program“ der letzten Folie
```

```
glUseProgram(myProgram);
```

Spracheigenschaften und Syntax

Uniform Variablen

- Im Shadern lesbare, globale Variablen
- Für alle Vertices und Fragments während eines Durchlaufs der Graphics-Pipeline konstant
- Schreibbar aus Applikation heraus, lesbar in allen Shadern

Spracheigenschaften und Syntax

Uniform Variablen im Shader

```
#version 330

uniform mat4 model;

in vec4 vs_in_pos;

void main() {

    gl_Position = model * vec4(vs_in_pos, 1);

}
```

Spracheigenschaften und Syntax

Anbinden von Uniform Variablen

```
// Liefert einen Index eindeutig für das gelinkte Programm-Objekt  
// welches die Shader enthält
```

```
int glGetUniformLocation(int program, String name);
```

```
// Anbinden eines 3-dim. Vektors an eine Uniform-Variable.
```

```
// Weitere Dimensionen in LWJGL möglich
```

```
glUniform3f(glGetUniformLocation(programID, „uLightDir“),  
            0.0f, 0.1f, 0.0f);
```

```
// Bsp: Anbinden der Model-Matrix an eine Uniform-Variable. Siehe
```

```
// Vertex Shader Folie 16.
```

```
Matrix4f modelMat = new Matrix4f();
```

```
int uniformModelMatrix;
```

```
...
```

```
uniformModelMatrix = glGetUniformLocation(programID, „model“);
```

```
matrix2uniform(modelMat, uniformModelMatrix);
```

Spracheigenschaften und Syntax

Uniform Variablen

```
Matrix4f matRotX = new Matrix4f();
Matrix4f modelMat = new Matrix4f();
private static int uniformModelMat;
...
public static void ... {

    // Update Rotations-Matrizen
    matRotX.rotate(float winkel, Vector3f X-Achse);

    // Update Model-Matrix um Rotation
    Matrix4f.mul(modelMat, matRot, modelMat);

    // Bekanntmachung der Uniform-Variable mit Shader
    uniformModelMat = glGetUniformLocation(programID, "model");

    // Wertzuweisung an Uniform-Variable
    matrix2uniform(modelMat, uniformModelMat);

}
```

Spracheigenschaften und Syntax

Vertex Daten anlegen und speichern

```
// Anlegen des Floatbuffers
FloatBuffer fb = BufferUtils.createFloatBuffer(int size);

// Anlegen des Float-Arrays für Vertices
float[] vertices = new float []{
    0.0f, 1.0f, 0.0f, ...
};
// Übergabe der Vertices an den Buffer
fb.put(vertices);

// Analog Normalen
FloatBuffer nb = BufferUtils.createFloatBuffer(int size);
float[] normals = new float []{
    ...
};
nb.put(normals);

// Anlegen des IntBuffers: später!
IntBuffer ib = BufferUtils.createIntBuffer(int indexSize);
```

Spracheigenschaften und Syntax

Transfer der Daten nach OpenGL

```
// Erzeugt (leeres) Buffer Object und liefert ID
int glGenBuffers();

// Binden des Buffers

// GL_ARRAY_BUFFER: Speicherung von VertexArray-Daten
// GL_ELEMENT_ARRAY: Speicherung von Indexdaten für Vertices

glBindBuffer(GL_ARRAY_BUFFER, int bufferObject_ID);
glBufferData(GL_ARRAY_BUFFER, bufferData, GL_STATIC_DRAW);
```

Spracheigenschaften und Syntax

Transfer der Daten nach OpenGL

// Beispiel

// Vertices

```
int vboId = glGenBuffers();  
glBindBuffer(GL_ARRAY_BUFFER, vboId);  
glBufferData(GL_ARRAY_BUFFER, fb, GL_STATIC_DRAW);
```

// Analog für Normalen

// Indices

```
int iboId = glGenBuffers();  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, ib, GL_STATIC_DRAW);
```

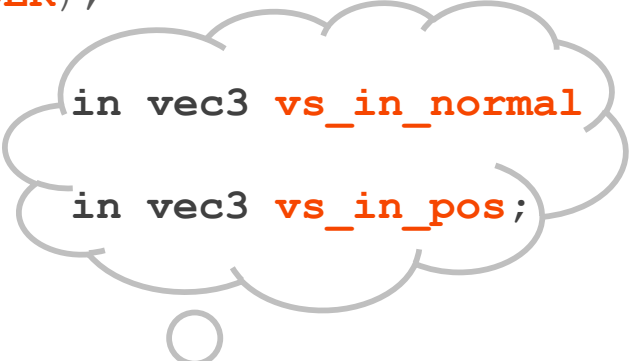
Spracheigenschaften und Syntax

Binden der Daten an Shader – in-Variablen

```
// Weißt einem Attribut des VS (=in-Variable) einen Index zu. Muss  
// vor dem Linken des PO passieren und wirkt danach. Index muss  
// für PO eindeutig sein. Attribut ansprechbar via Index  
void glBindAttribLocation(int program, int index, String name);
```

```
int myProgram = glCreateProgram()  
int myVS = glCreateShader(GL_VERTEX_SHADER);  
int myFS = glCreateShader(GL_FRAGMENT_SHADER);  
glShaderSource(myVS, „version..  
glShaderSource(myFS, „version..  
glCompileShader(myVS);  
glCompileShader(myFS);  
glAttachShader(myProgram, myVS);  
glAttachShader(myProgram, myFS);
```

```
Static final int ATTR_POS = 0, ATTR_NORMAL = 1; //Indices setzen  
glBindAttribLocation(myProgram, ATTR_POS, „vs_in_pos“);  
glBindAttribLocation(myProgram, ATTR_NORMAL, „vs_in_normal“);  
glLinkProgram(myProgram)
```



```
in vec3 vs_in_normal
```

```
in vec3 vs_in_pos;
```

Spracheigenschaften und Syntax

Aktivieren und Layout der Vertex Daten

```
// Aktiviert ein Vertex Attribute Array zum Attribute „index“
glEnableVertexAttribArray(int index);

// Beschreibt Layout der Daten des Attributs „index“ des gerade an
// GL_ARRAY_BUFFER (oder ELEMENT_ARRAY_BUFFER) angebundenen Buffer
// Objects.
glVertexAttribPointer(
    int index,          // Index des Attributs
    int size,           // Komponentenzahl des Attributs. {1,2,3,4}
    int type,           // Konstante für Datentyp
    int stride,         // Größe eines Arguments
    boolean normalized  // Normalisierte Werte
    long buffer_offset  //Offset des 1. Eintrags
);
```

Spracheigenschaften und Syntax

Aktivieren und Layout der Vertex Daten

// Beispiel für VertexBufferObjekt mit 3-dimensionalen Argumenten

// Bereits passiert:

```
glBindBuffer(GL_ARRAY_BUFFER, vboId);
```

Beschreibung der Vertex Daten des zuletzt gebundenen VertexBuffer

(Für die Normalen, Farben, und andere Vertex-Attribute analog)

```
glEnableVertexAttribArray(ATTR_POS);
```

```
glVertexAttribPointer(ATTR_POS,      // Index der Position
                        3,             // 3 Komponenten (x,y,z)
                        GL_FLOAT,      // Datentyp: Float
                        false,         // nicht nötig zu wissen
                        3*4,           // Eintrag-Größe (3 * 4 Byte)
                        0              // Offset 1. Eintrag
);
```

Spracheigenschaften und Syntax

// Beispiel

```
vboId = glGenBuffers();  
glBindBuffer(GL_ARRAY_BUFFER, vboId);  
glBufferData(GL_ARRAY_BUFFER, vBuffer, GL_STATIC_DRAW);  
  
nboId = glGenBuffers();  
glBindBuffer(GL_ARRAY_BUFFER, nboId);  
glBufferData(GL_ARRAY_BUFFER, nBuffer, GL_STATIC_DRAW);  
  
iboId = glGenBuffers();  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, iBuffer, GL_STATIC_DRAW);  
  
glBindBuffer(GL_ARRAY_BUFFER, vboId);  
glEnableVertexAttribArray(ATTR_POS);  
glVertexAttribPointer(ATTR_POS, 3, GL_FLOAT, false, 12, 0);  
  
glBindBuffer(GL_ARRAY_BUFFER, nboId);  
glEnableVertexAttribArray(ATTR_NORMAL);  
glVertexAttribPointer(ATTR_NORMAL, 3, GL_FLOAT, false, 12, 0);71
```

Spracheigenschaften und Syntax

Rendering

```
// Binden des Index-Buffers
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);

glDrawElements (
    int topology,           // GL_LINE, GL_TRIANGLES, etc.
    int indices_count,      // Anzahl Indices
    int type,               // Datentyp der Indices
    long indices_buffer_offset
);

// Beispiel

glDrawElements (GL_TRIANGLES,
                ib.capacity(),
                GL_UNSIGNED_INT,
                0
);
```

OpenGL Einschub: Topologien

Vertices und ihre Indizes

Verschiedene Indizierungstechniken für Folge von Vertices
bzw. Indices möglich

Gegeben:

- Vertex-Buffer, mit Daten jedes Vertices (Koordinaten,...)
- Index-Buffer, legt Reihenfolge fest und kann Index eines Vertices mehrmals enthalten. Hier 6 Indices

Vertex Buffer

V ₀	V ₀	V ₀	V ₁	V ₁	V ₁	V ₂	V ₂	V ₂	V ₃	V ₃	V ₃	V ₄	V ₄	V ₄	V ₅	V ₅	V ₅
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z

Index Buffer

0	1	2	3	4	5
---	---	---	---	---	---

OpenGL Einschub: Topologien

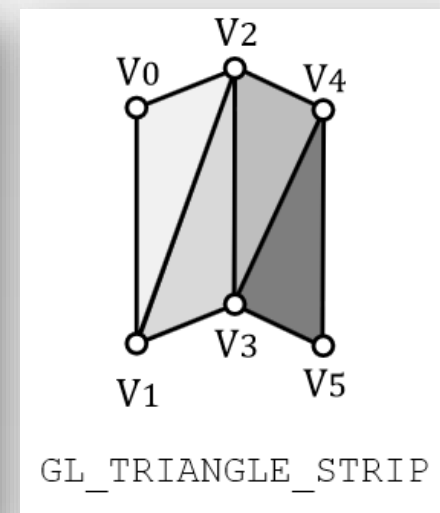
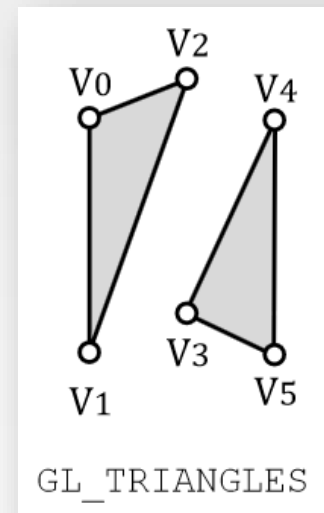
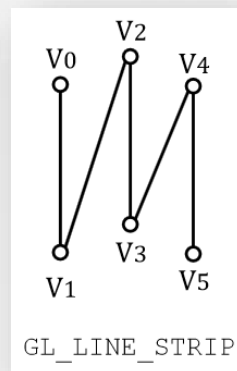
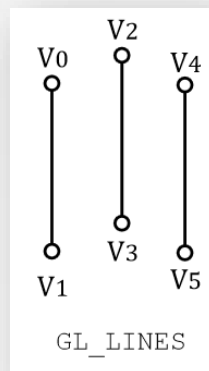
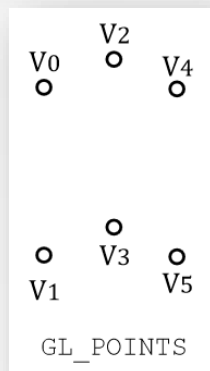
Vertices und ihre Indizes

Vertex Buffer

V ₀	V ₀	V ₀	V ₁	V ₁	V ₁	V ₂	V ₂	V ₂	V ₃	V ₃	V ₃	V ₄	V ₄	V ₄	V ₅	V ₅	V ₅
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z

Index Buffer

0	1	2	3	4	5
---	---	---	---	---	---



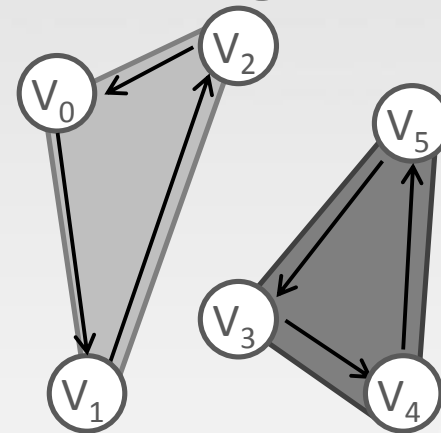
OpenGL Einschub: Topologien und Dreiecke

Vertices und ihre Indizes

Achtung: Bei Dreiecken Indexreihenfolge beachten

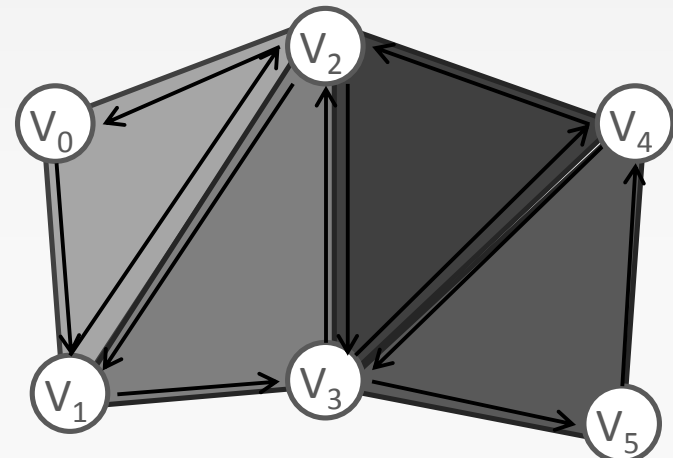
GL_TRIANGLES

- 1. Dreieck: V_0, V_1, V_2
- 2. Dreieck: V_3, V_4, V_5
- ...



GL_TRIANGLE_STRIP

- 1. Dreieck: V_0, V_1, V_2
- 2. Dreieck: V_2, V_1, V_3
- 3. Dreieck: V_2, V_3, V_4
- ...



OpenGL Einschub: Topologien und Meshes

GL_TRIANGLES

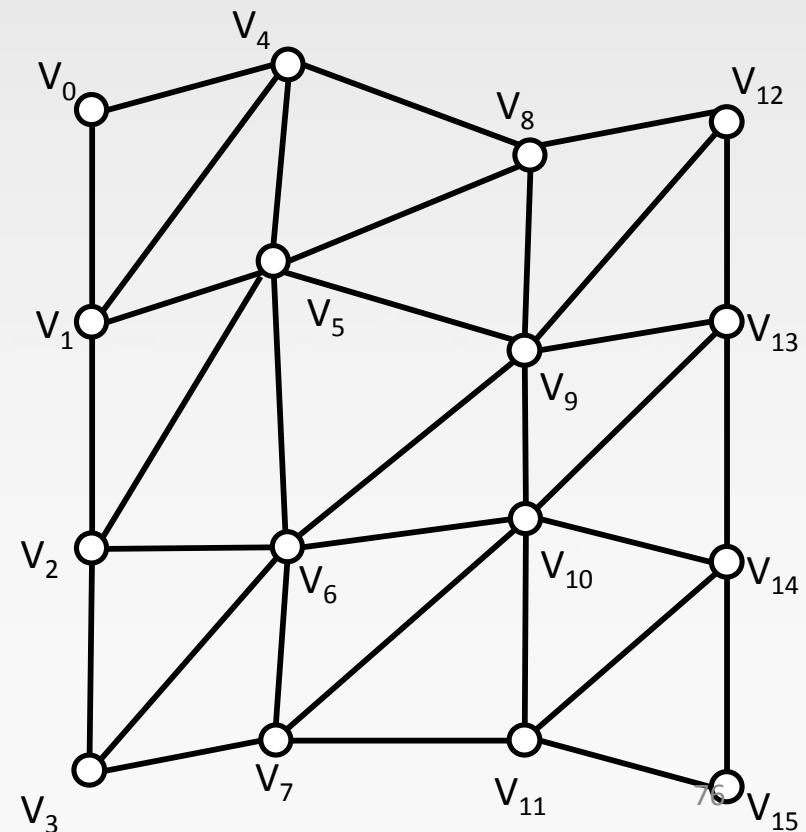
Vertex Buffer

V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}	V_{14}	V_{15}
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------

Index Buffer

0	1	4	4	1	5	8	4	5
8	5	9	12	8	9	12	9	13
1	2	5	5	2	6	9	5	6
9	6	10	13	9	10	13	10	14
2	3	6	6	3	7	10	6	7
10	7	11	14	10	11	14	11	15

54 Indices



OpenGL Einschub: Topologien und Meshes

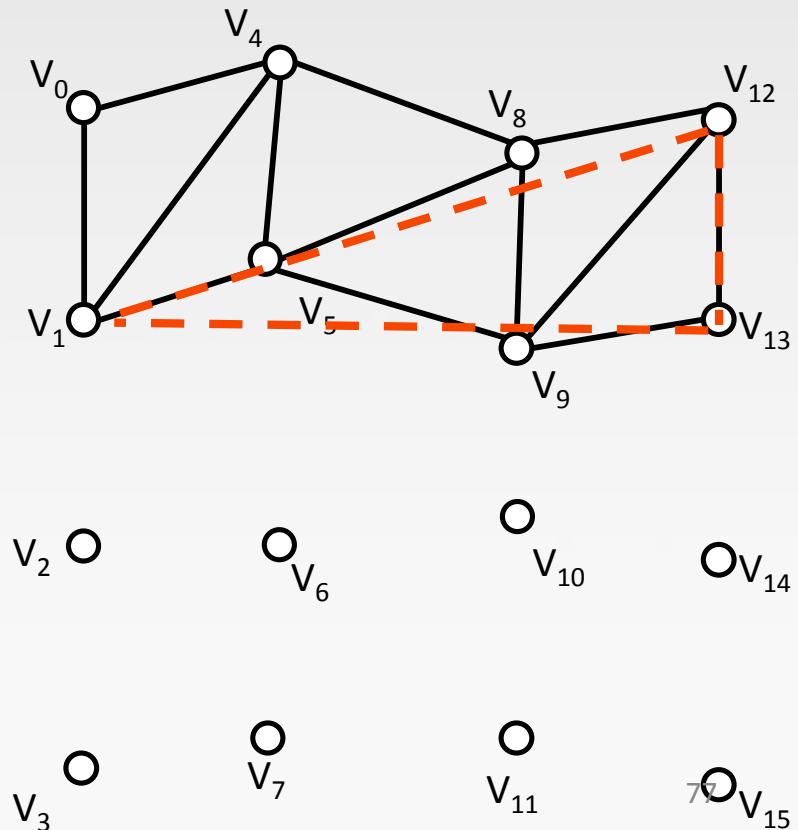
GL_TRIANGLE_STRIP

Vertex Buffer

V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}	V_{14}	V_{15}
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------

Index Buffer

0	1	4	5	8	9	12	13	1
2	5	6	9	10	14	...		



GL_TRIANGLE_STRIP + + Primitive Restart Index

V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}	V_{14}	V_{15}
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------

0	1	4	5	8	9	12	13	-1
1	2	5	6	9	10	13	14	-1
2	3	6	7	10	11	14	15	

27 Indices

GLSL Basics

Spracheigenschaften und Syntax

GLSL Programm-Grundaufbau

```
// 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

}
```

Spracheigenschaften und Syntax

GLSL 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 = 90000.0;   // Double
    ...

    // Type Conversion

    int exactValue = 3;
    float PI = float(exactValue); // Cast Integer -> Float
}
```

Spracheigenschaften und Syntax

GLSL 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] oder [rgba]

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)
```

Spracheigenschaften und Syntax

GLSL 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: 
$$\begin{pmatrix} 1.0 & 7.0 \\ 2.0 & 8.0 \\ 3.0 & 9.0 \end{pmatrix}$$

```

Spracheigenschaften und Syntax

GLSL Operatoren

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

```
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 oder Vektor * Matrix die  
// bekannte Matrizenmultiplikation
```

```
Mat2x3 x; Mat3x4 y;  
x * y; // Ergibt mat2x4  
y * x; // Nicht möglich, aus Dimensionsgründen
```

Spracheigenschaften und Syntax

GLSL Flusskontrolle

```
// 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-while 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;  
}
```

Spracheigenschaften und Syntax

GLSL Built-in Funktionen

```
// GLSL beinhaltet einige mathematische und vektorielle
// Funktionen. Beispiele:

float sin(float radians) // Trigonometrische Funktion(en)

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

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

float length(vec4 x) // Betrag eines Vektors

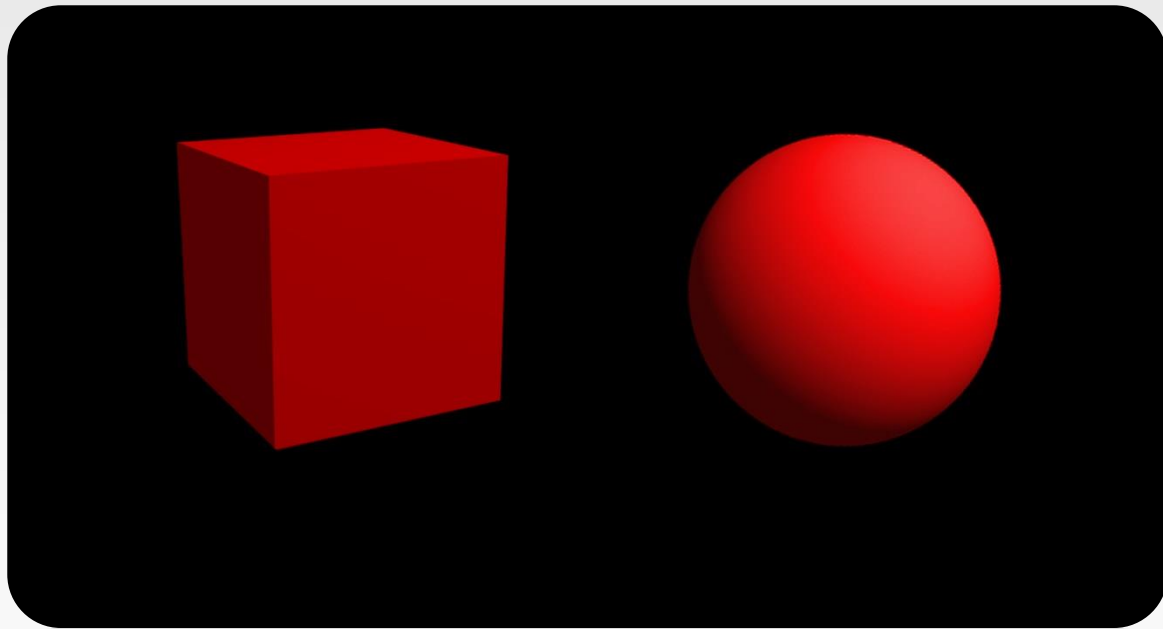
float dot(vec4 a, vec4 b) // Skalarprodukt aus a und b

vec4 normalize(vec4 x) // Liefert normalisierten Vektor

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

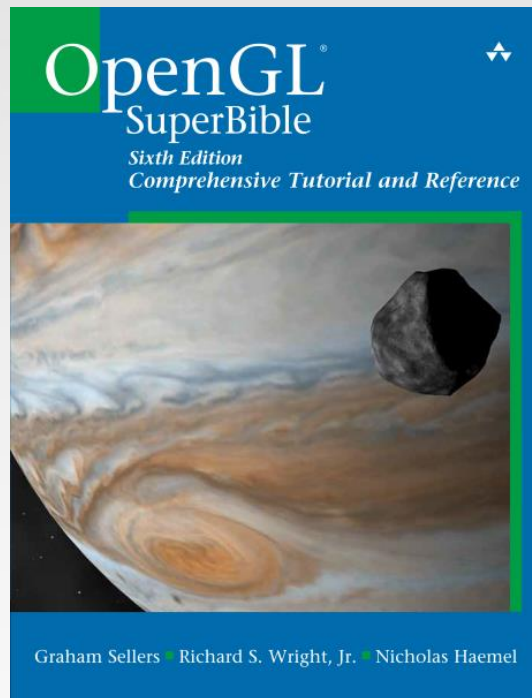
mat4 inverse(mat4 a) // Inverse Matrix, ab GLSL 1.50 / GL 3.2
```

Viel Erfolg!

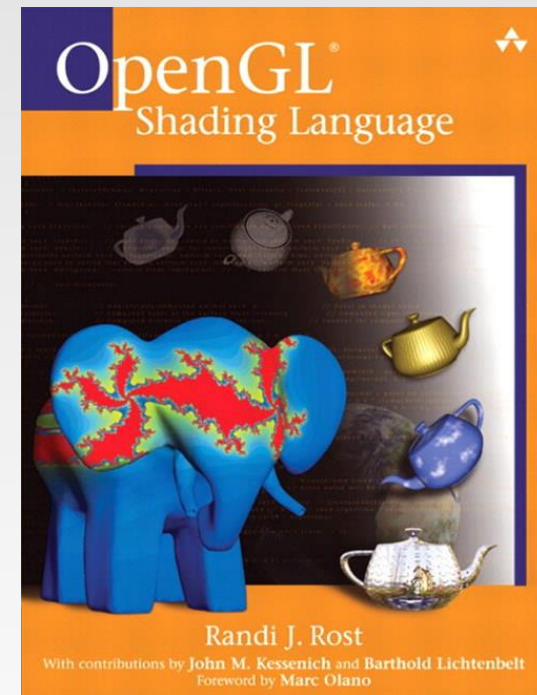


Einführung

Literatur zur OpenGL - Programmierung



Wright, Haemel, Sellers, Lipchak
OpenGL SuperBible Addison-Wesley
Juli 2013 (verfügbar unter Safari)



OpenGL Shading Language
Addison-Wesley 2009
(verfügbar unter Safari)

Einführung

Literatur zur OpenGL - Programmierung

Skript der Computergrafik-Vorlesung (2010, insb. 2012, 2014, 2016)

Tutorials von lwjgl.org etc.

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/>

OpenGL 4.3 Quick Reference Pages

<http://www.khronos.org/files/opengl43-quick-reference-card.pdf>