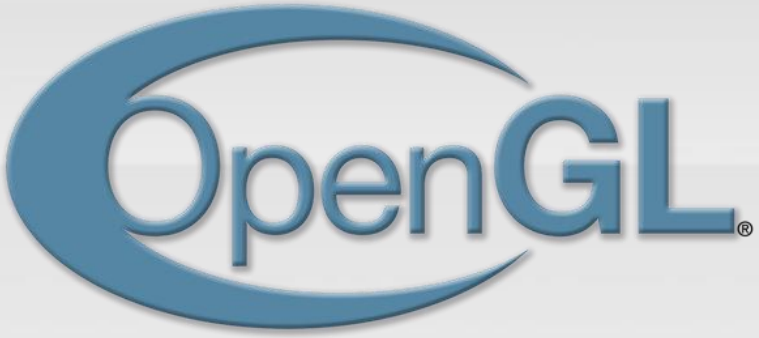


Eine Einführung

Computergrafik SS14

Timo Bourdon

Organisatorisches

Übung am Freitag den 11. Juli entfällt!

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.
```

- Wenn es nicht so gut läuft:

```
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

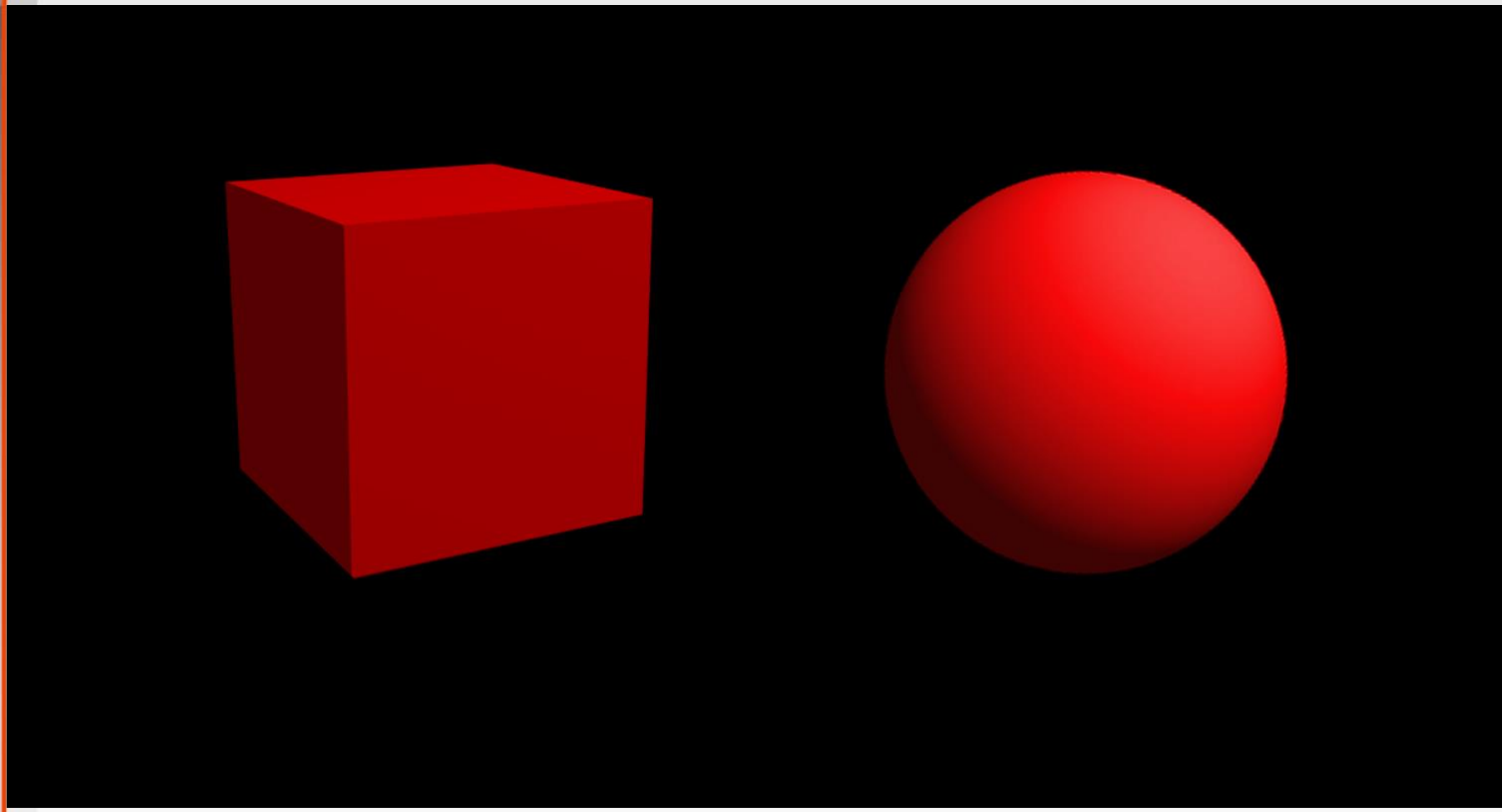
Was nach der Vorlesung nicht möglich sein wird…

Motivation

… das auch nicht…

Motivation

…aber wir legen die Grundsteine dafür!



Einführung

OpenGL - Definition

API für Rastergrafik

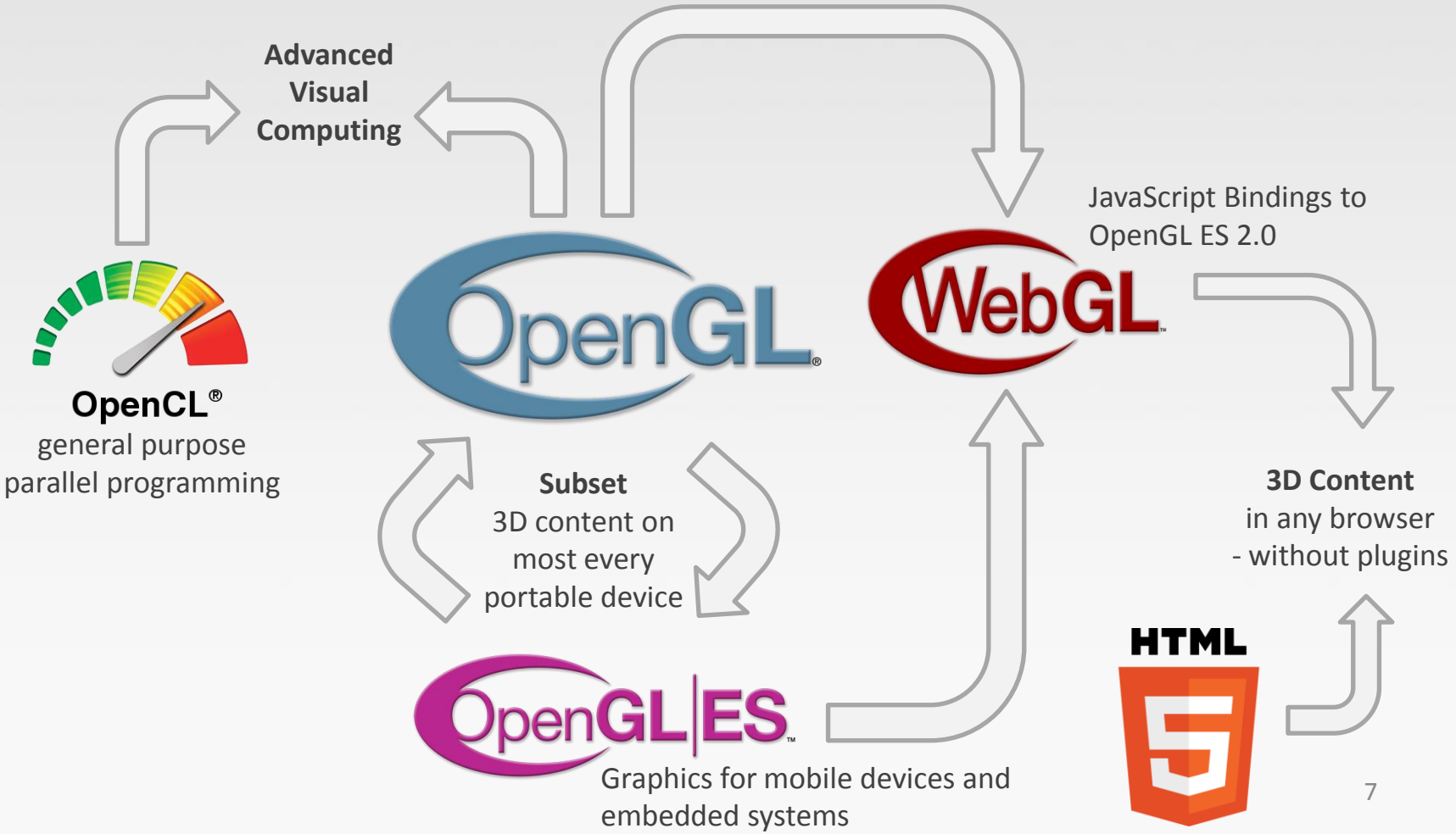
- Prozedural
- Hardwarenah
- Plattform-, Betriebssystem- und sprachunabhängig

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

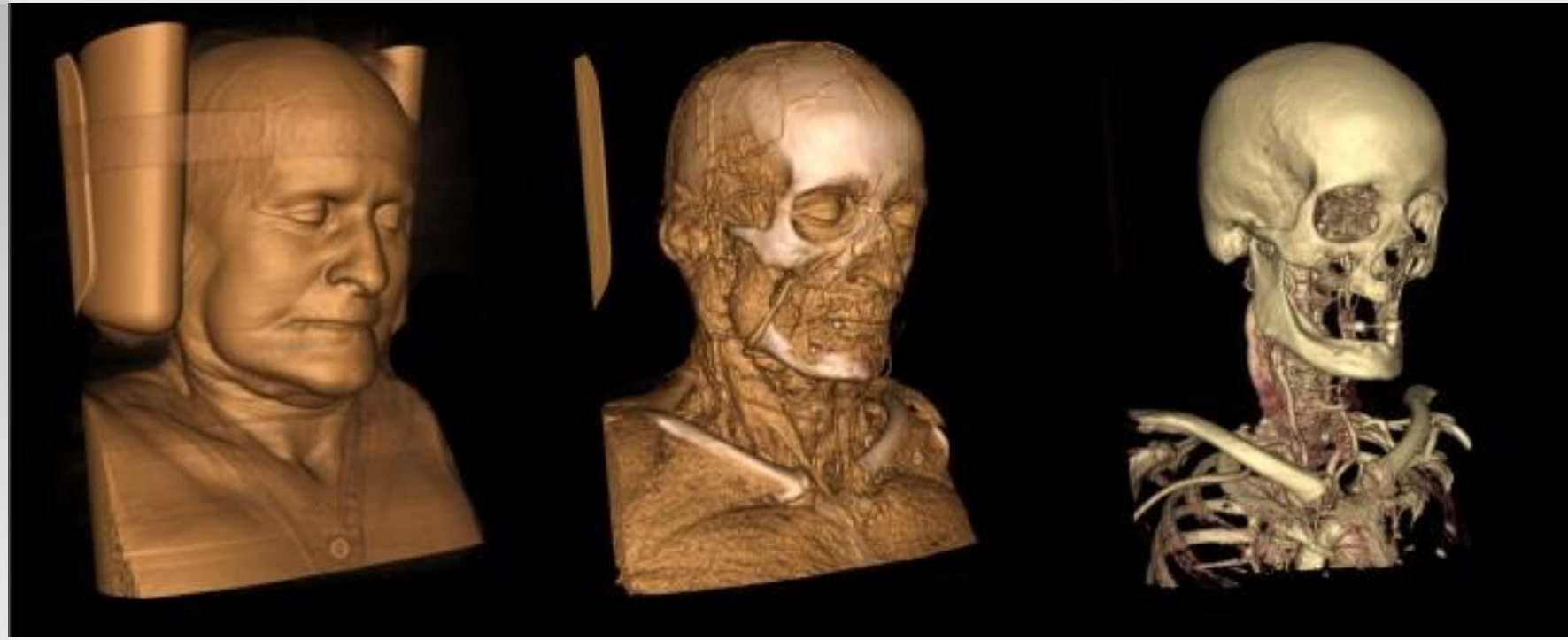


Einführung

Anwendungsbeispiele – Videospiele

Einführung

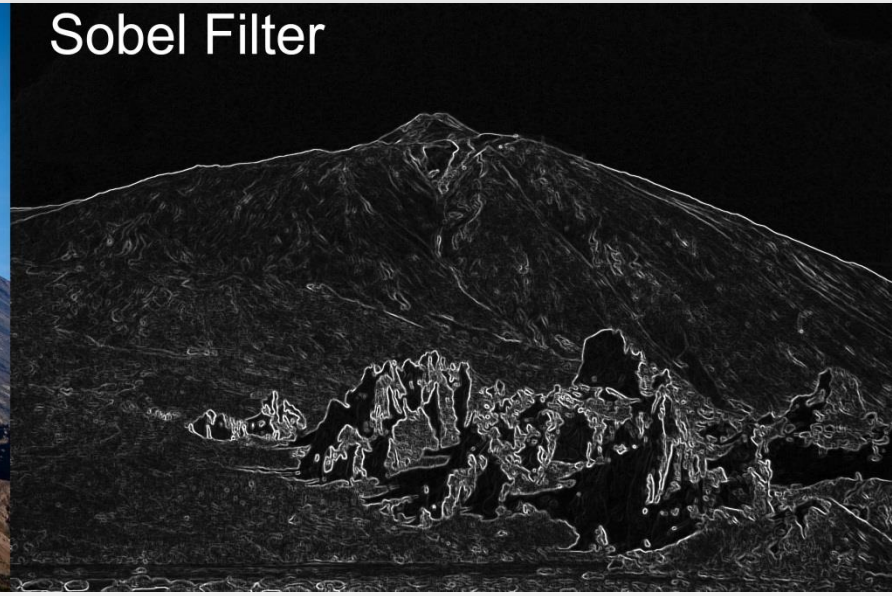
Anwendungsbeispiele - Medizin



Einführung
Anwendungsbeispiele

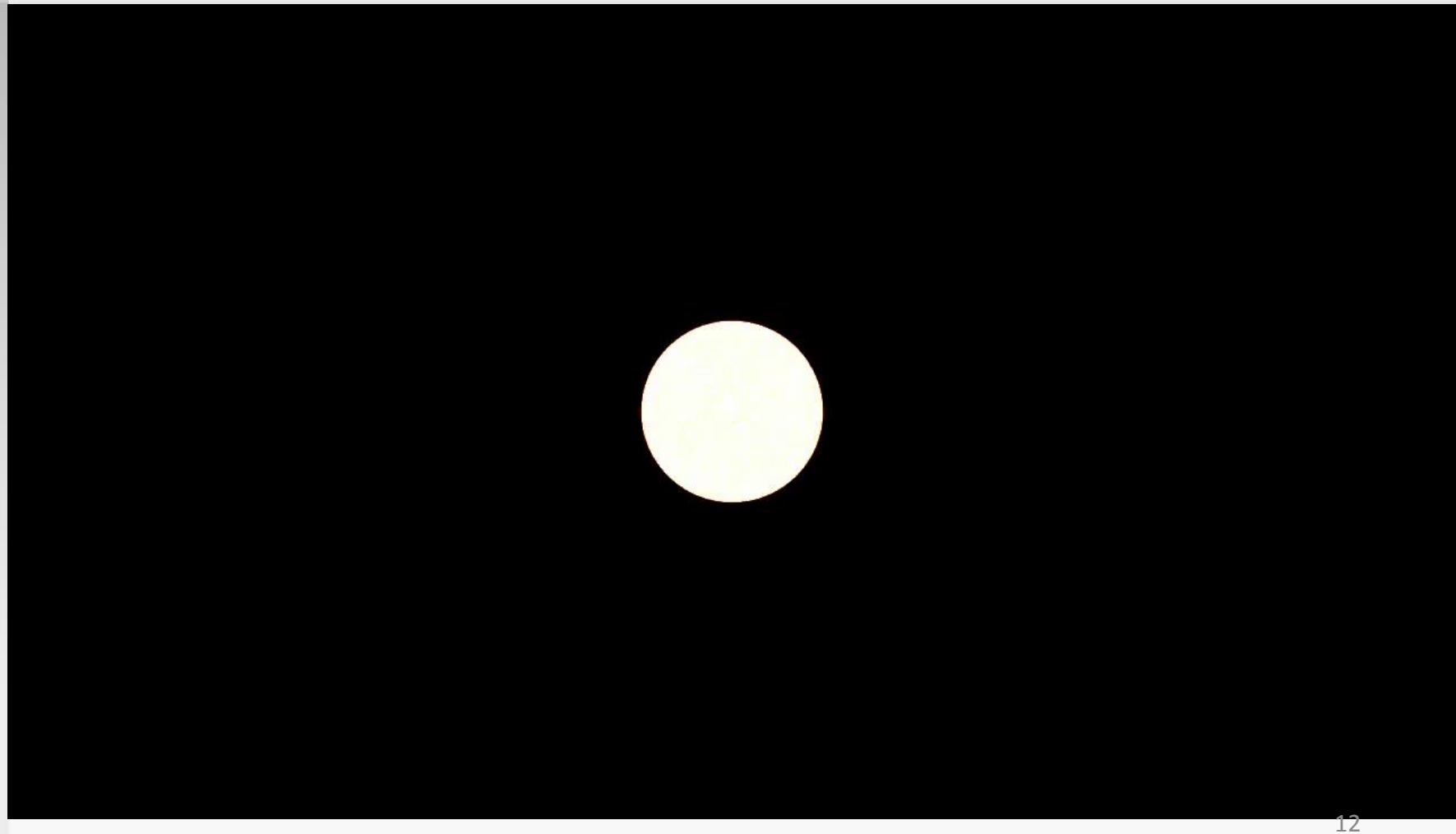
Einführung

Anwendungsbeispiele



Einführung

Anwendungsbeispiele



Einführung

Entwicklungsgeschichte

1992

- initiiert durch SGI
- Betreut durch das OpenGL ARB (Architecture Review Board) u.a.

u.v.m.

2006

- Übernahme durch die Khronos Group und Weiterführung

Einführung

Entwicklungsgeschichte

Einführung

Entwicklungsgeschichte - Versionen

OpenGL 1.x (ab 1992)
- Feste, konfigurierbare Pipeline
- Spätere Versionen bringen weitere „Optionen“

OpenGL 2.x (ab 2004)
- Einführung programmierbarer Stages (VS / FS)
- Feste Pipeline bleibt parallel bestehen

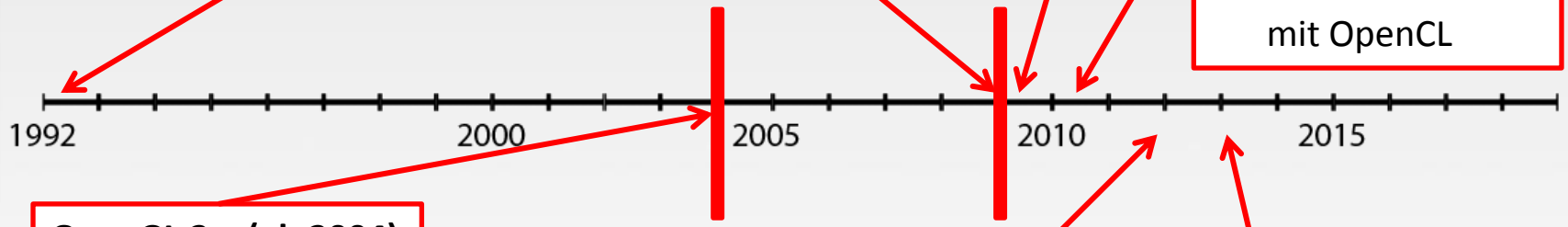
OpenGL 3.1 (03-2009)
- Legacy Funktionen aus Kern entfernt
- aufwärtskompatibel

OpenGL 3.2 (09-2009)
- Geometry Shader

OpenGL 3.3 & 4.0 (03-2010)
- Tesselation (nur 4.0)
- Bessere Zusammenarbeit mit OpenCL

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

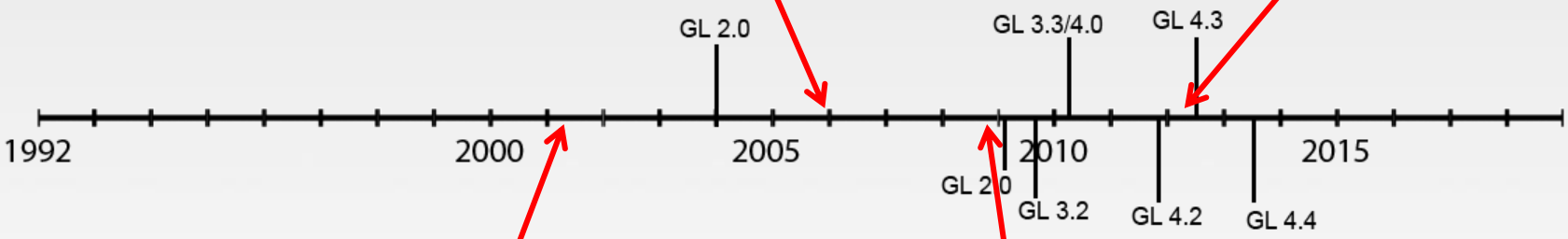


Einführung

Entwicklungsgeschichte - Hardware

Nvidia GeForce 8800 (11-2006)
- OpenGL 3.3
- Cuda / OpenCL

Nvidia GeForce GTX Titan (02-2013)
- OpenGL 4.4

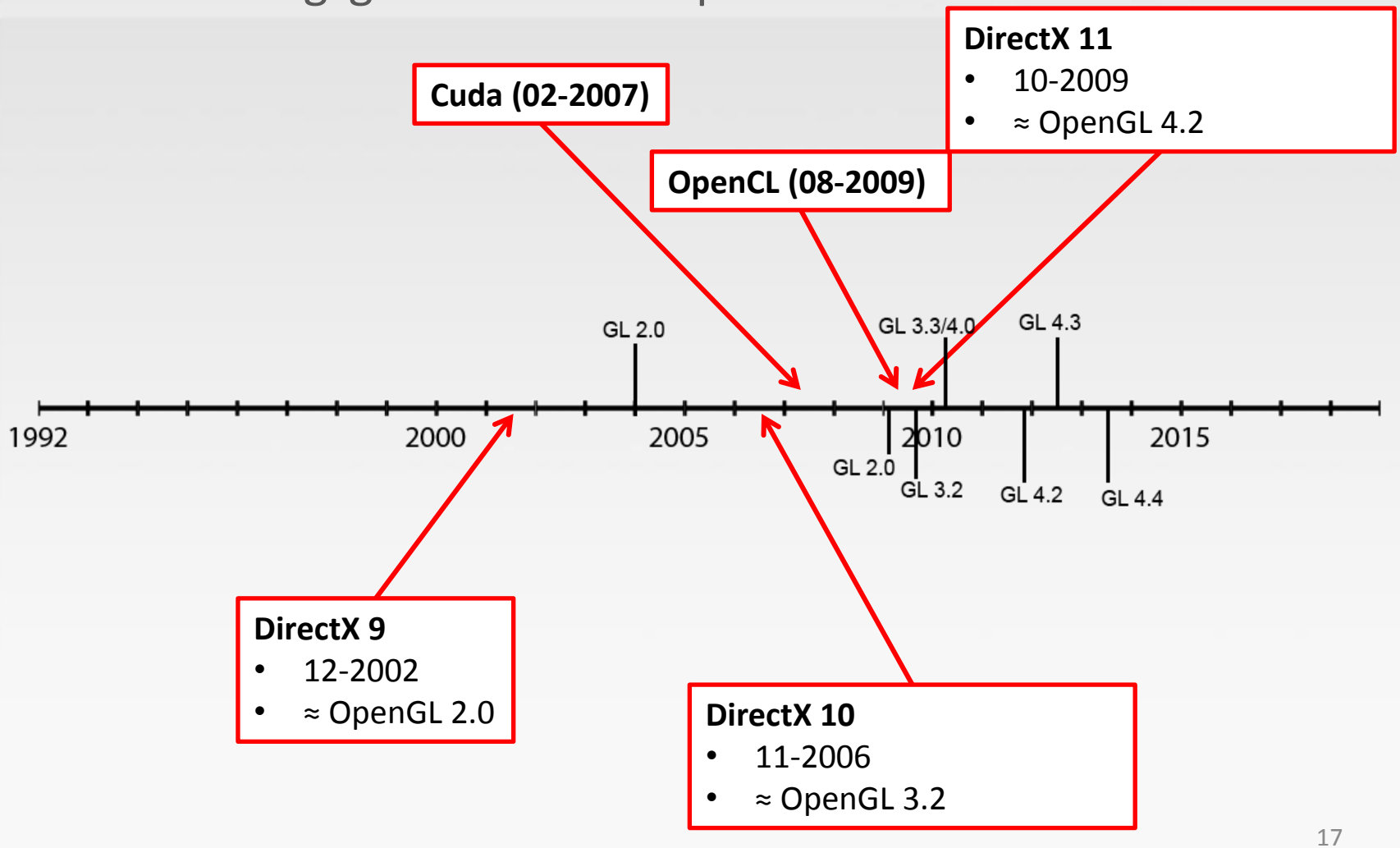


Nvidia GeForce 3 (03-2001)
- Vertex & Fragment Shader (eingeschränkt)

ATI Radeon 5870 (11-2009)
- OpenGL 4.2
- OpenCL 1.1

Einführung

Entwicklungsgeschichte – Proprietäre APIs



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**

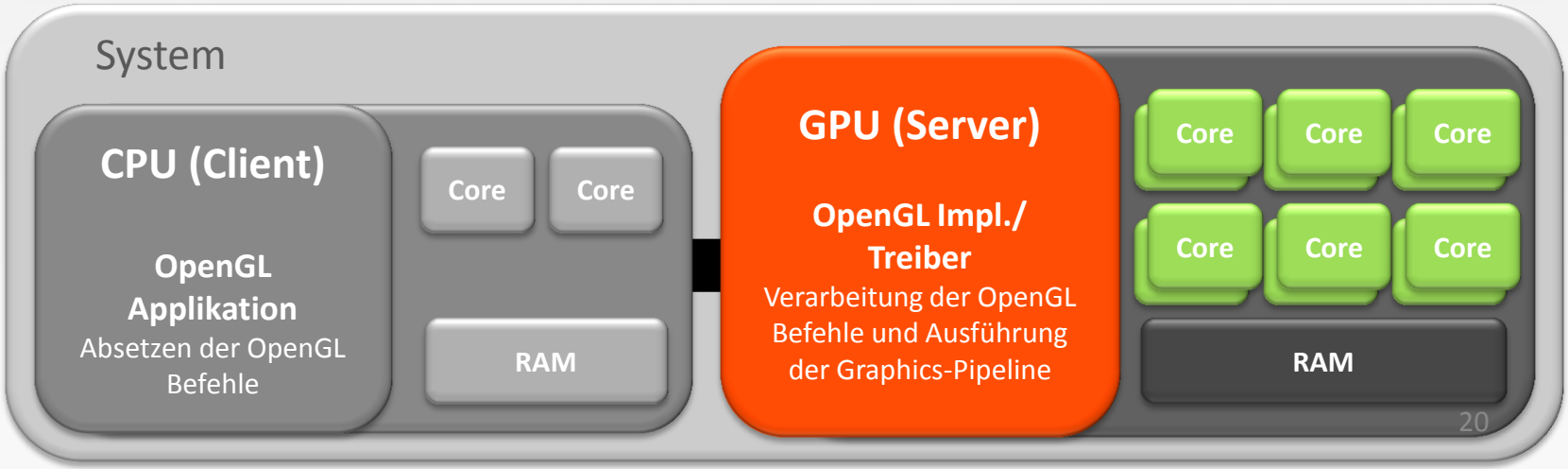
- **Erzeugen, übersetzen, etc. der Shader-Programme**

- **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

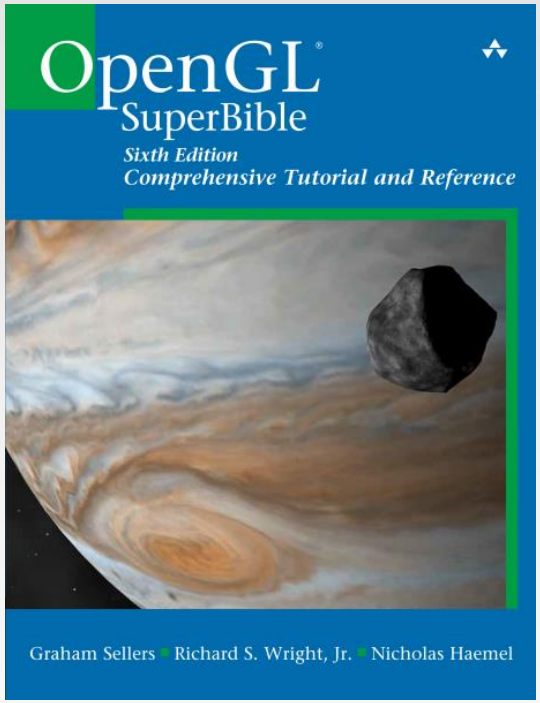


Einführung
Client-Server & State Machine

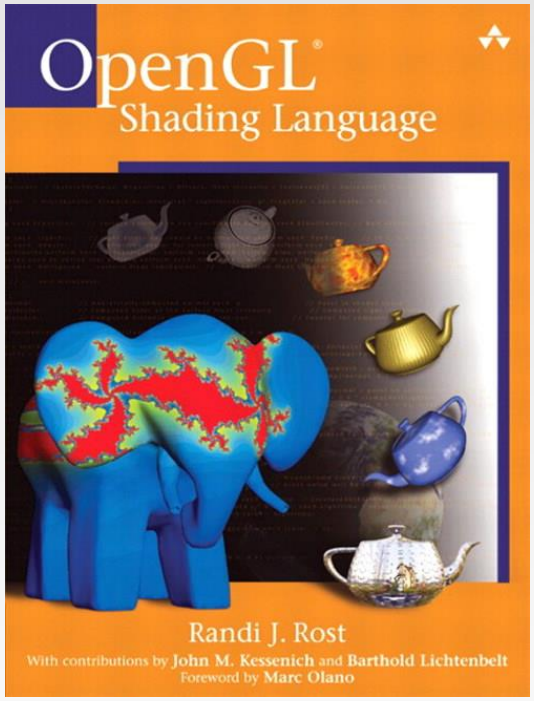
- **Zustandsmaschine**
  - Minimierung der Client-Server Kommunikation
  - Gesetzte Zustände, etwa die Hintergrundfarbe, bleiben bis zum Widerruf gültig
  - State-Variablen sind u.a. an den Aufrufen `glEnable()` oder `glDisable()` zu erkennen

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 und 2014)

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/

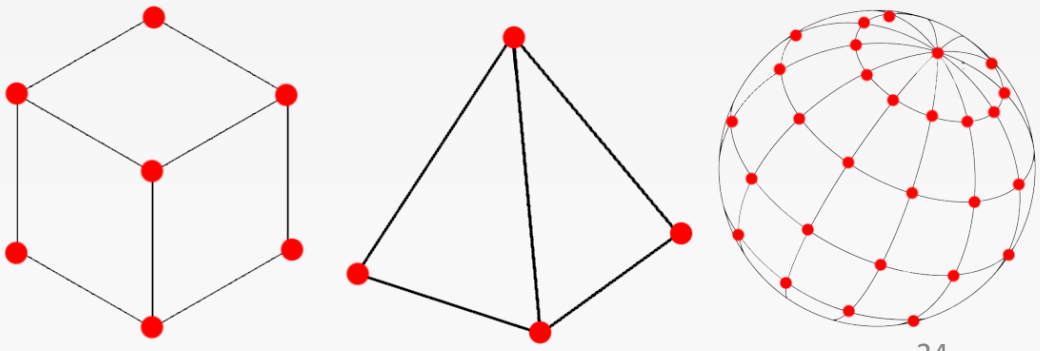
OpenGL 4.2 Reference Pages
http://www.opengl.org/sdk/docs/man4/
OpenGL 4.3 Quick Reference Pages
http://www.khronos.org/files/opengl43-quick-reference-card.pdf

Technik

Vertex

- **Allgemein: Mathematischer Punkt im Raum**
- **Speziell: „Eckpunkt“ einer geometrischen Figur**

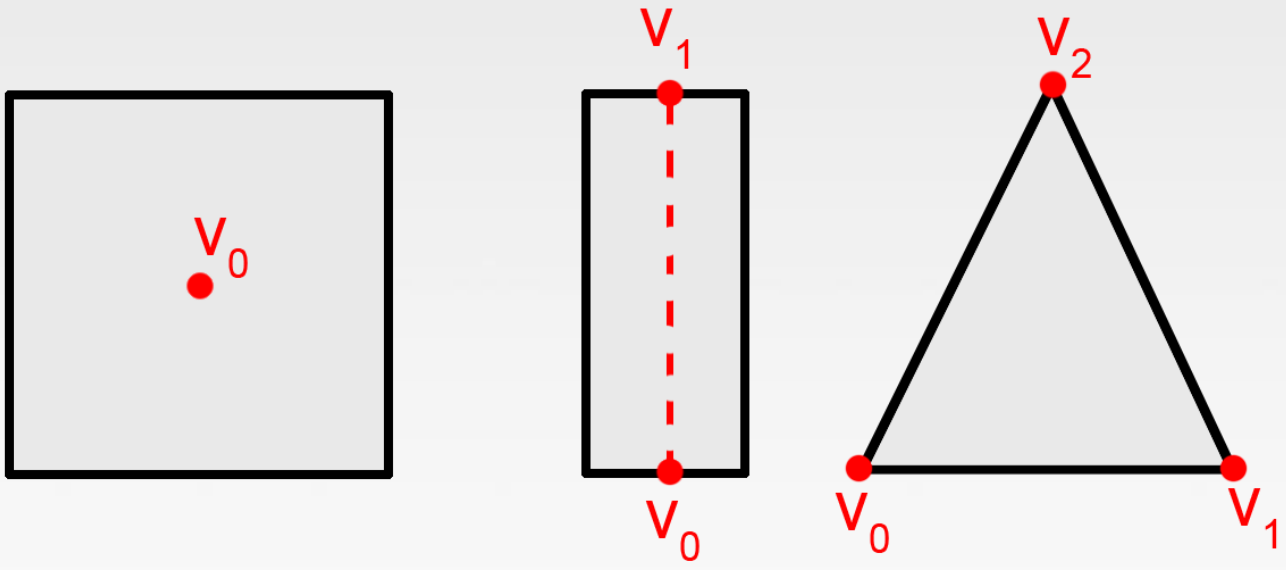
- **Enthält oft weitere Eigenschaften an diesem Punkt**
  - Normale
  - Farbe
  - Texturkoordinaten
  - Geschwindigkeit
  - Beschleunigung
  - Materialdichte
  - etc.



Technik

Primitive

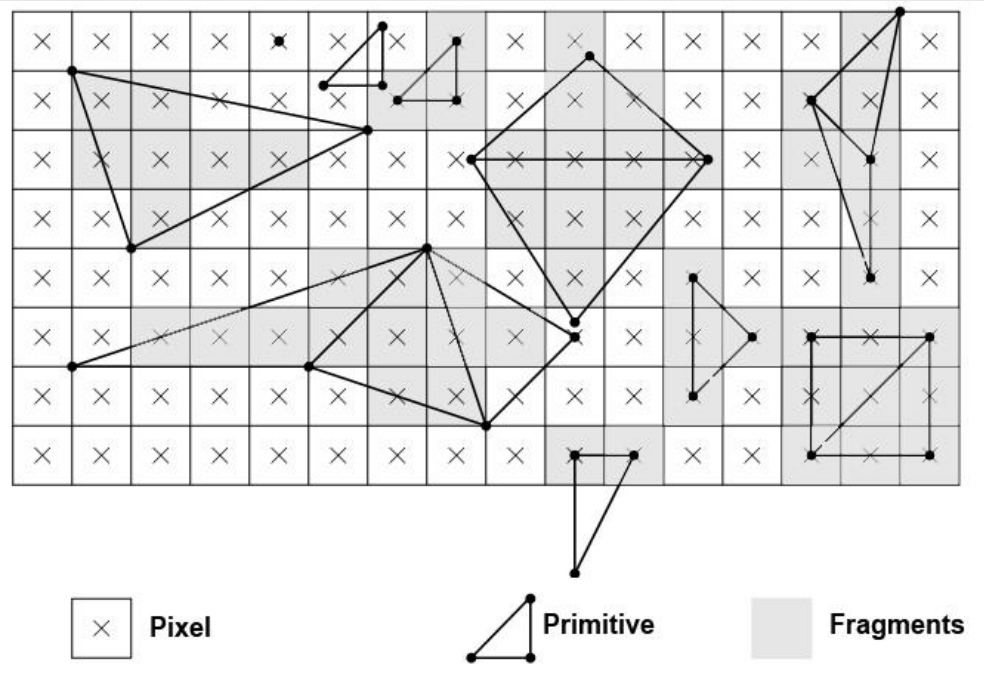
- Elementare grafische Grundform“
- Besteht aus 1 – 3 Vertices
- Topologische Information



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

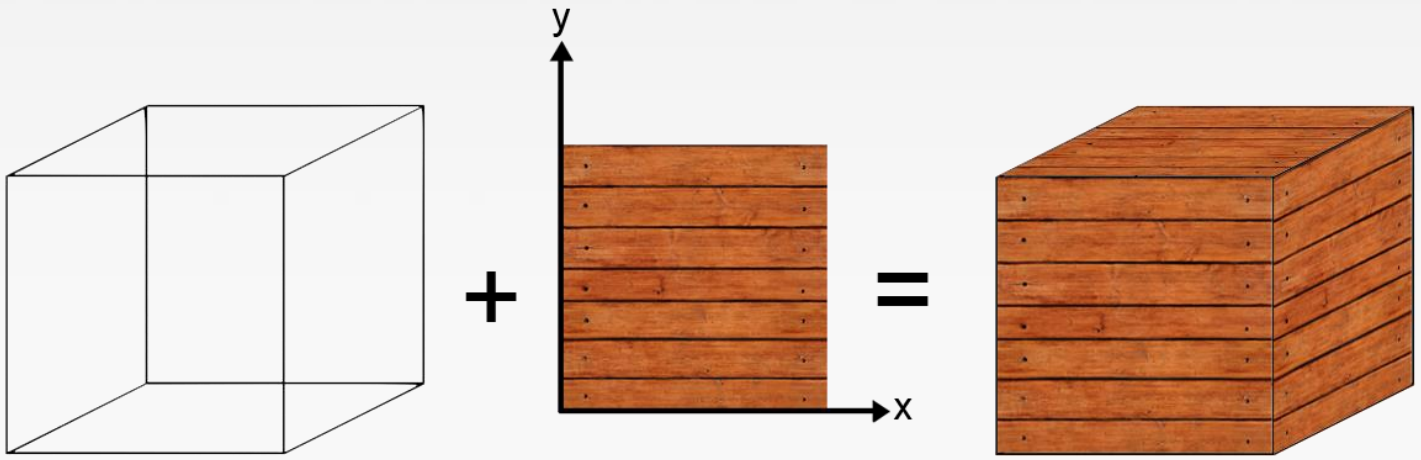
Uniform Data

- **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.)
  - 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 - damals

Shader (Cook, *Shade Trees*, 1984)

- Programm zur Beschreibung / Berechnung von Oberflächeneigenschaften
- Renderman Shading Language

Technik

Shader - heute

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 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;
        }
}
```

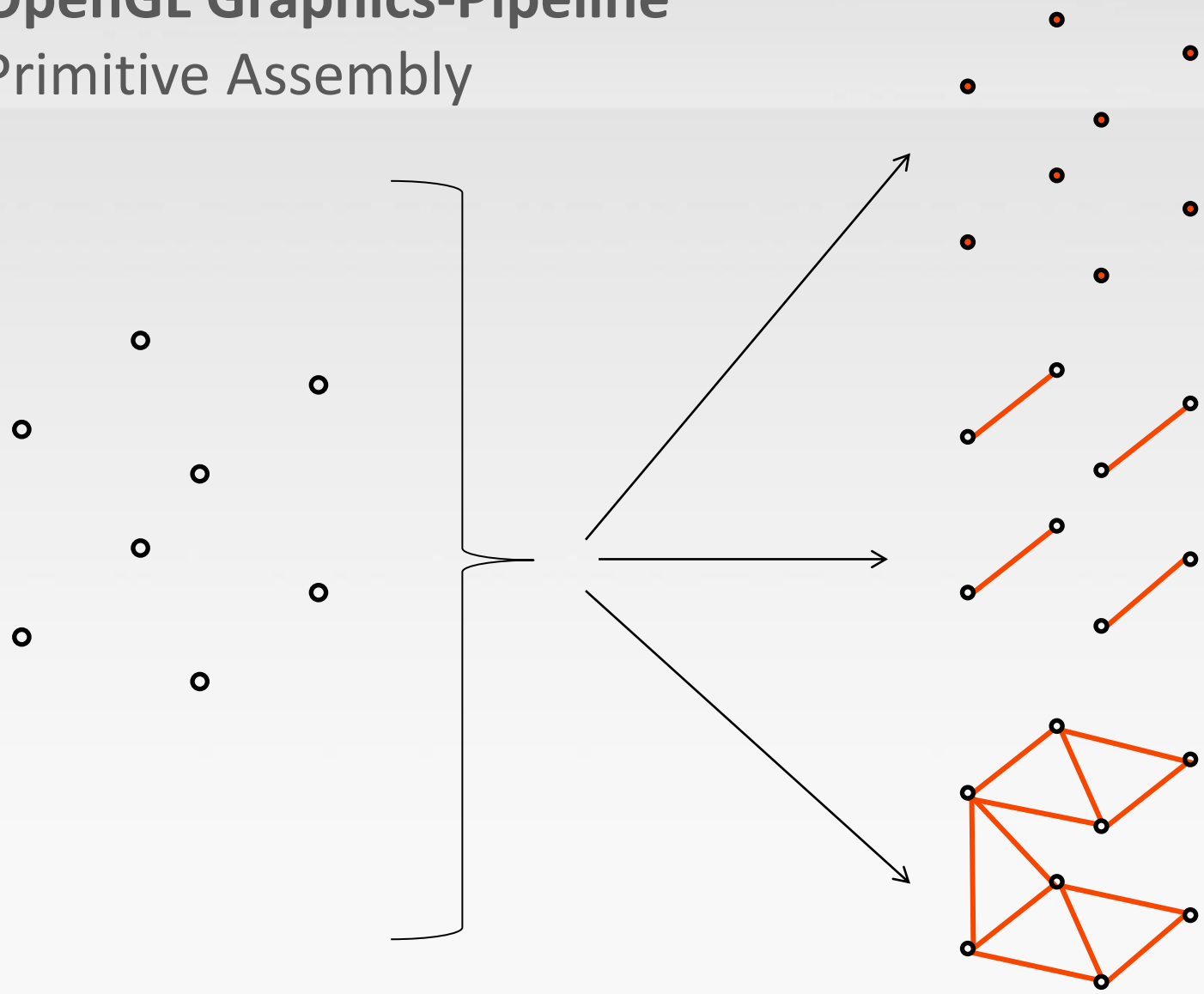
Vertex Shader

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
3. Viewport Transformation
4. Culling

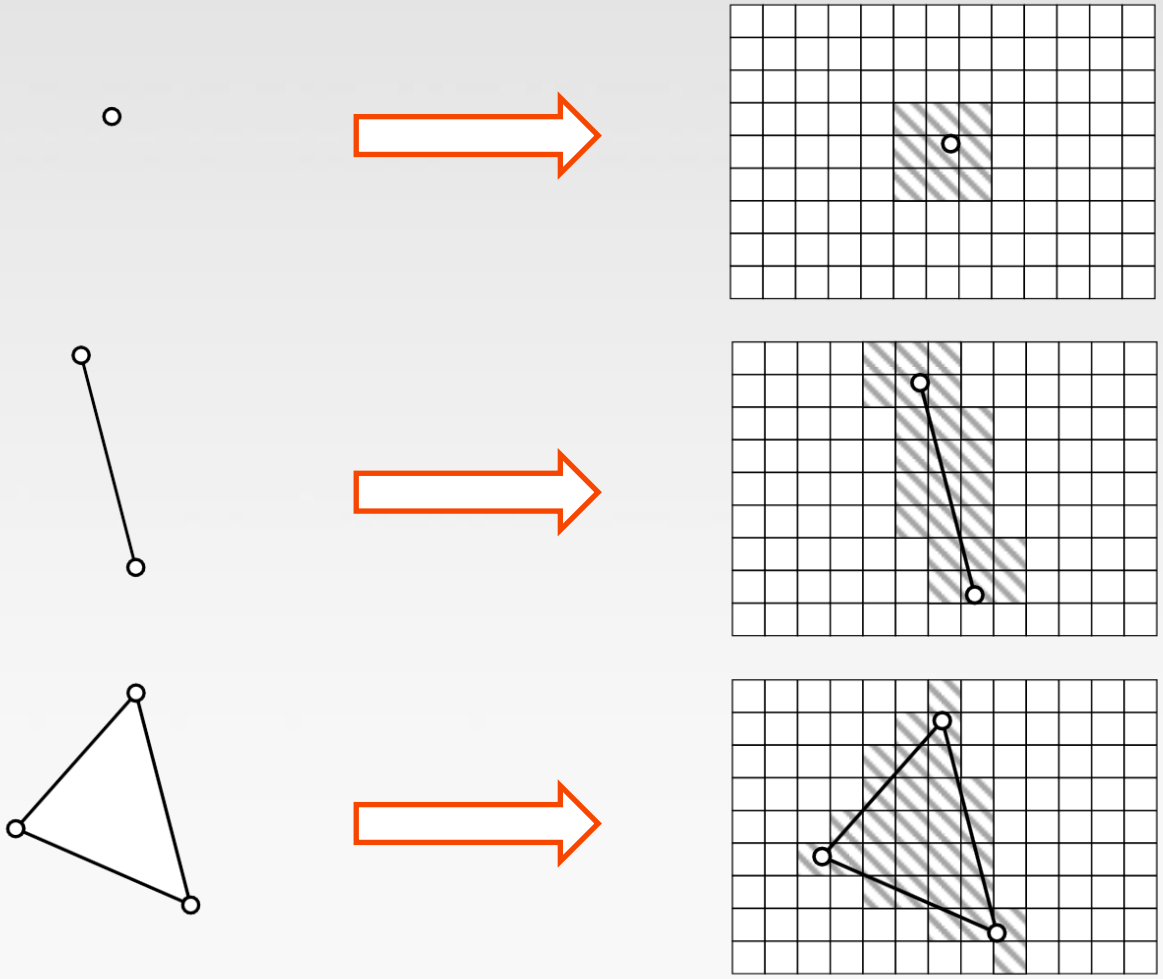
OpenGL Graphics-Pipeline

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

VS

```
#version 330

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;
}
```

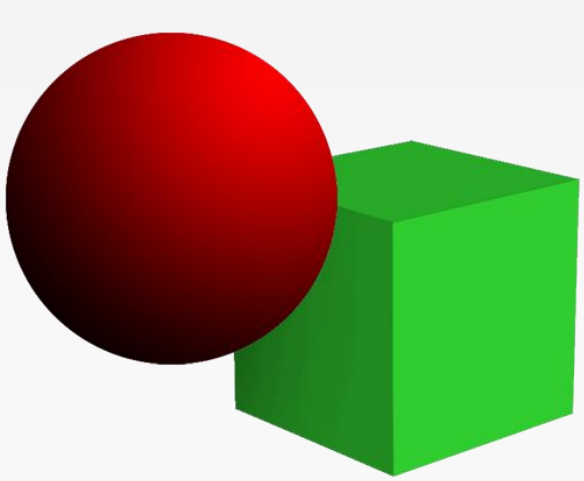
FS

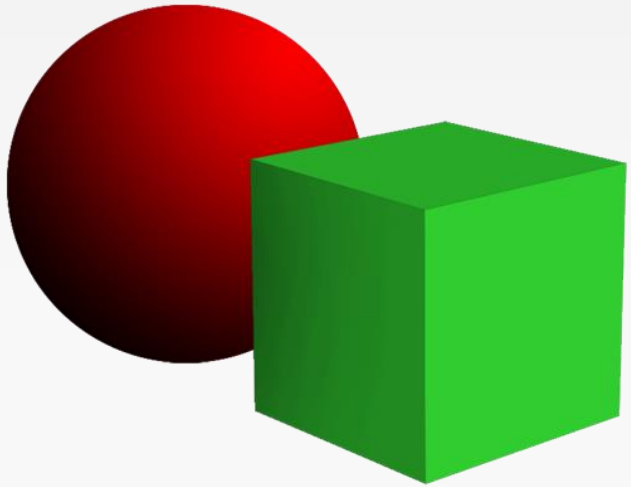
```
#version 330

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)

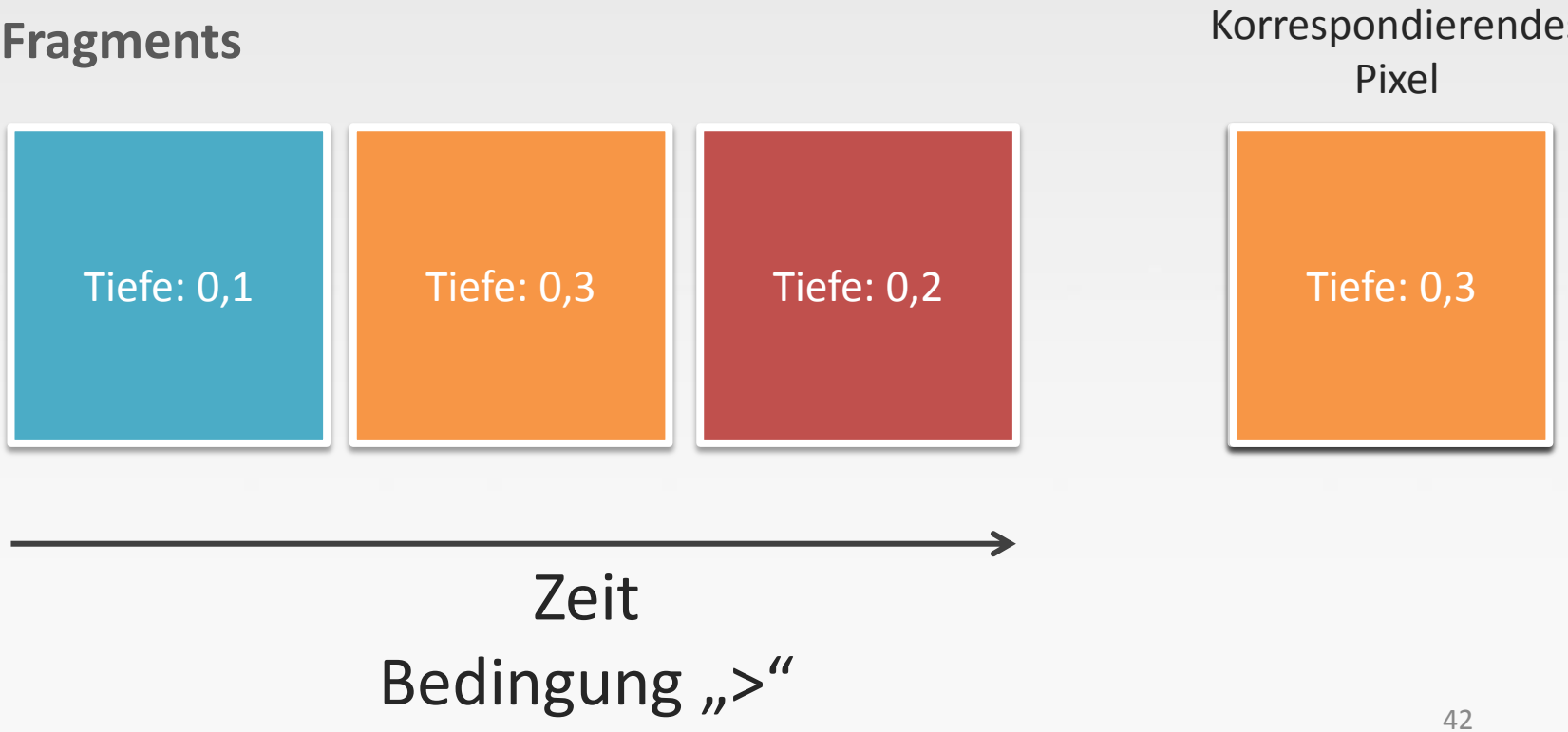




OpenGL Graphics-Pipeline

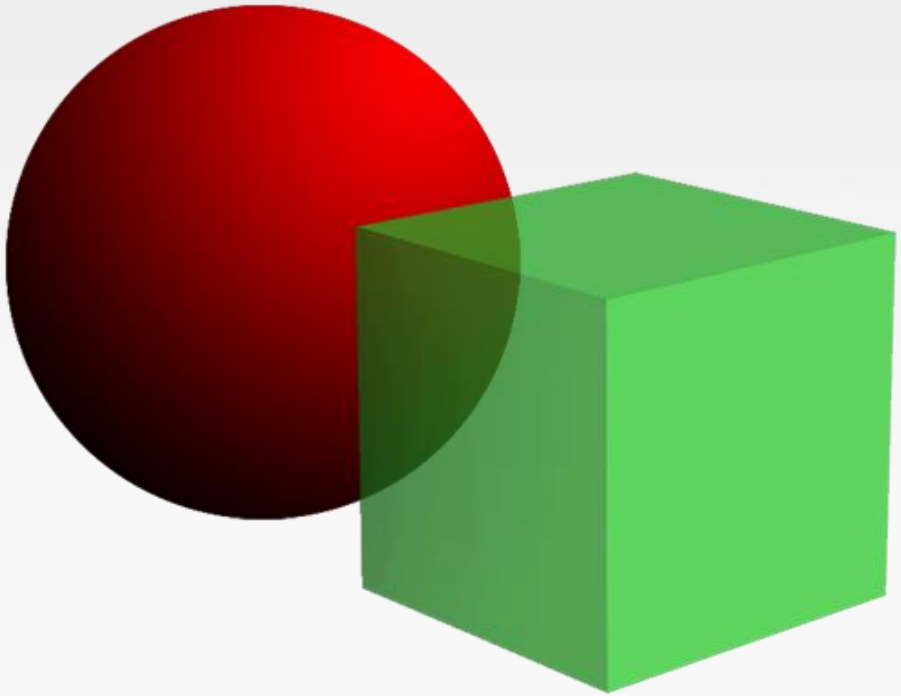
Per Fragment Operations

- **Tiefentest**



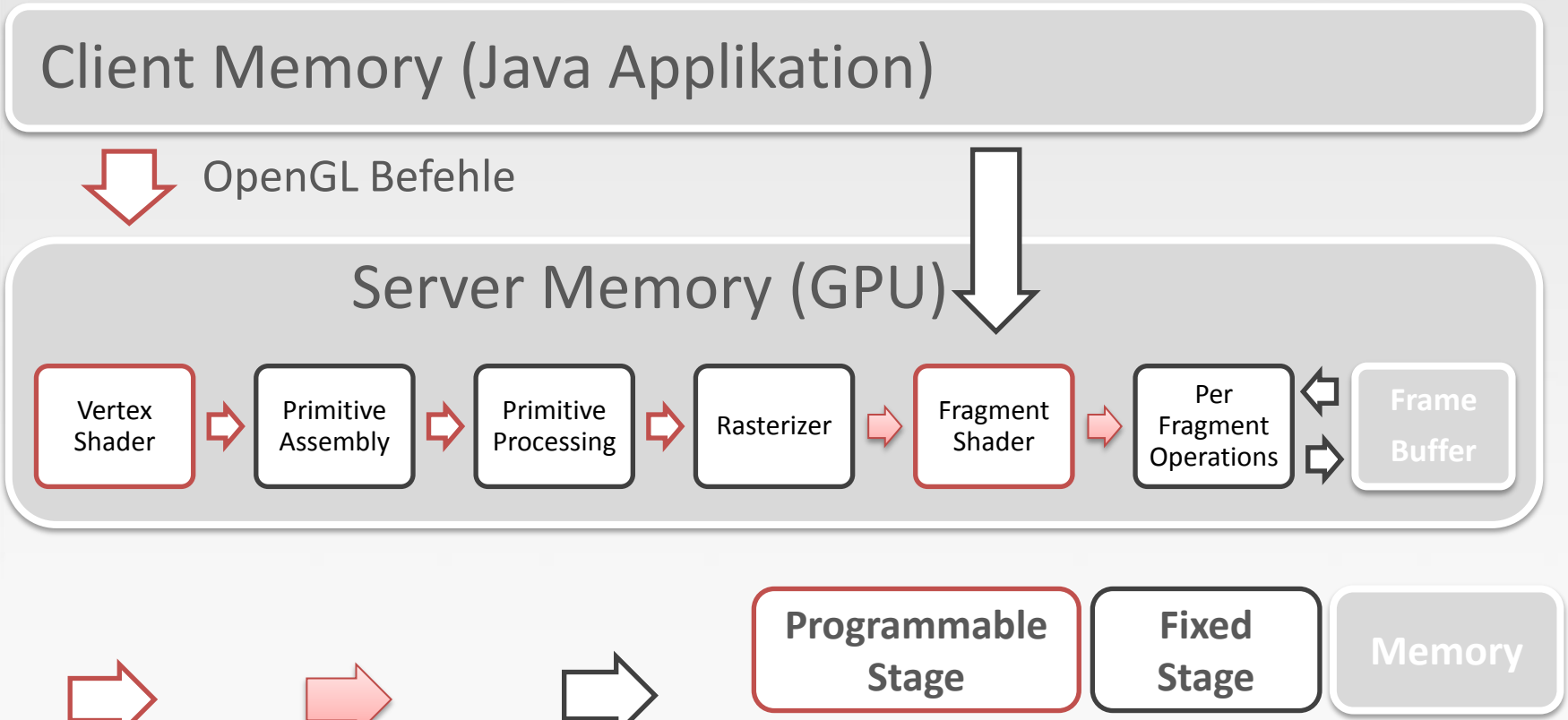
OpenGL Graphics-Pipeline
Per Fragment Operations

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

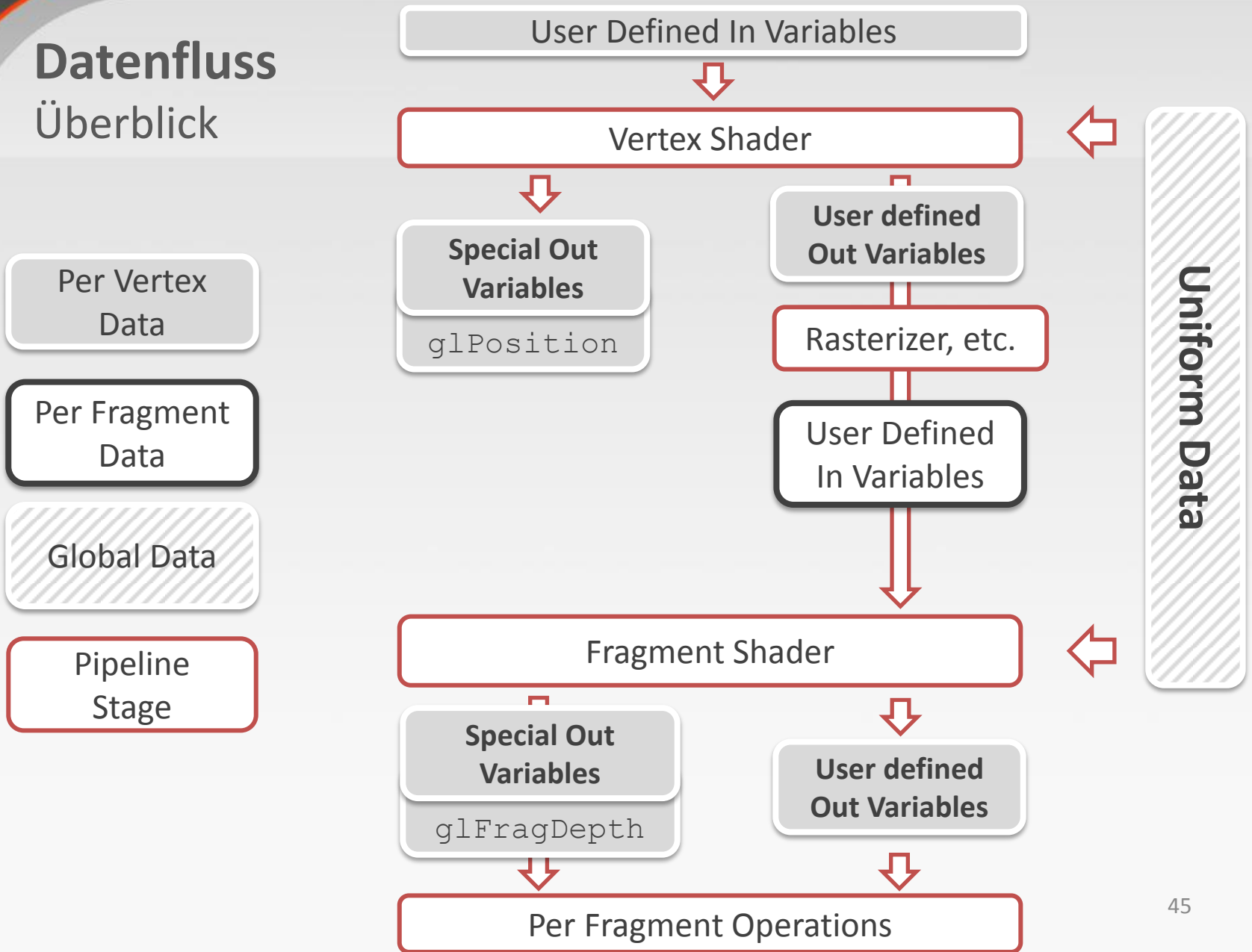


OpenGL Graphics-Pipeline

Übersicht



Datenfluss
Überblick



Ein kurzer Überblick

- Low-Level API
- **betriebssystemunabhängige** Entwicklung von **3D-Grafiken**
- **Hardwarebeschleunigung**
- Entwicklung durch **Khronos Group** sowie **Mozilla** als lizenzfreier Webstandard

- WebGL basiert auf **OpenGL ES 2.0**
- **Native Unterstützung** der gängigsten Browser

seit Version 11

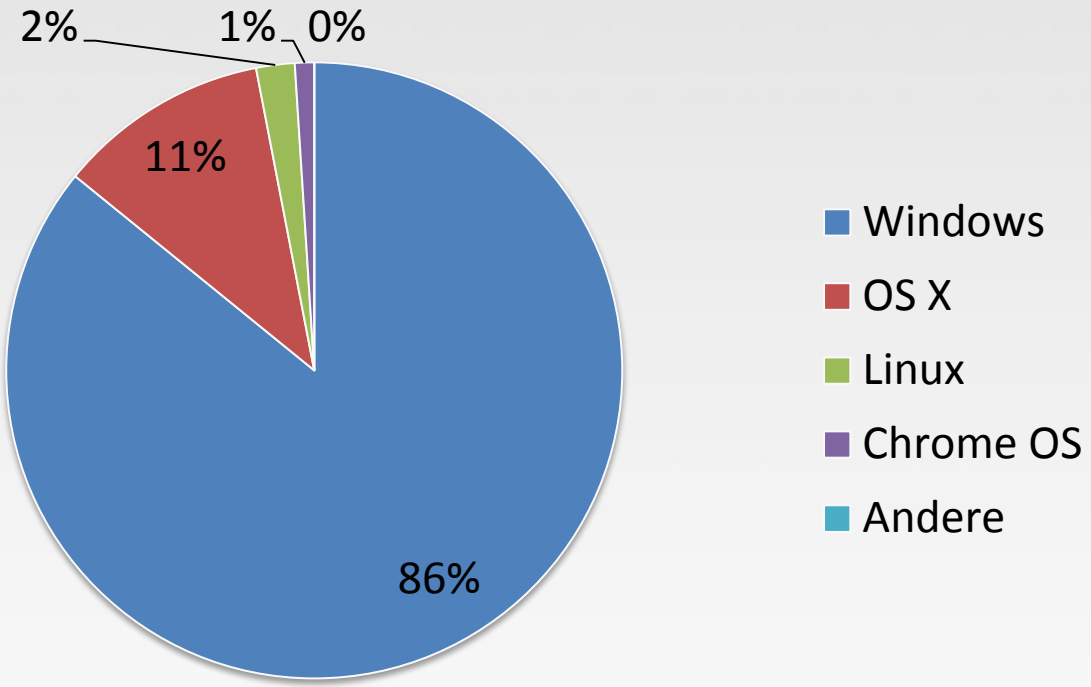
- Darstellung über HTML5 **Canvas-Element**

- Nativ im Browser ⇒ **keine PlugIns** nötig
- **Keine Cookies**
- Programmierung mit **JavaScript** und **HTML5**
- **Debugging** eingeschränkt (Skripte + Firebug, etc.)

WebGL

Nutzungs-Statistiken

Nutzung im **Desktop-Bereich**

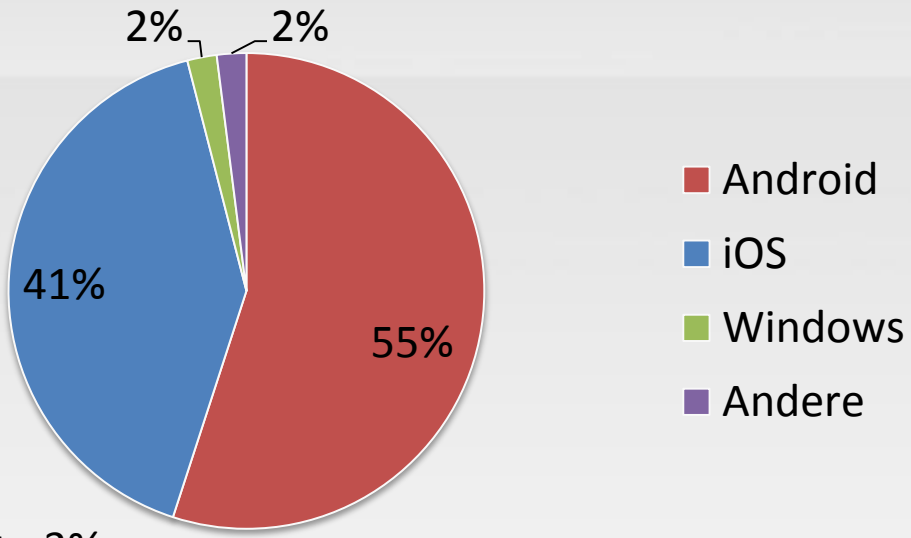


Quelle: http://webglstats.com/ (Stand: 01.07.2014)

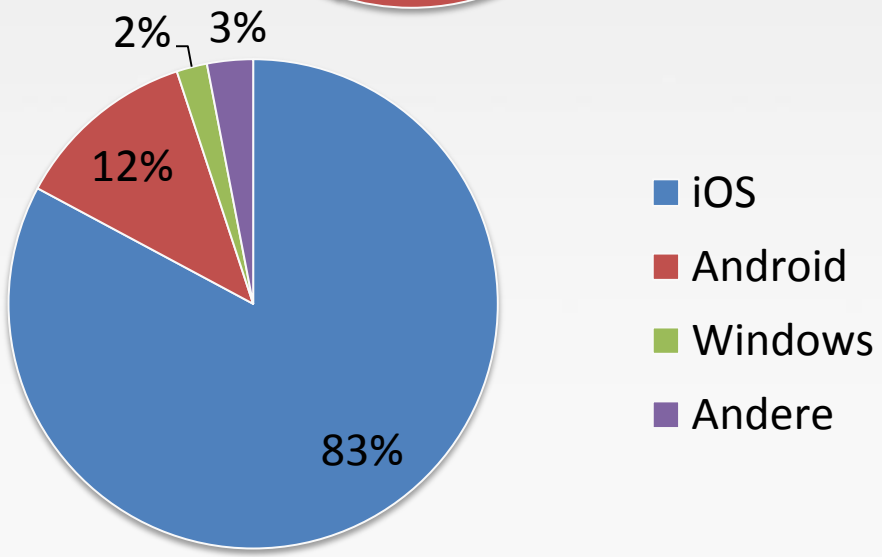
WebGL

Nutzungs-Statistiken

Nutzung im **Smartphone-Bereich**



Nutzung im **Tablet-Bereich**



Quelle: http://webglstats.com/ (Stand: 01.07.2014)

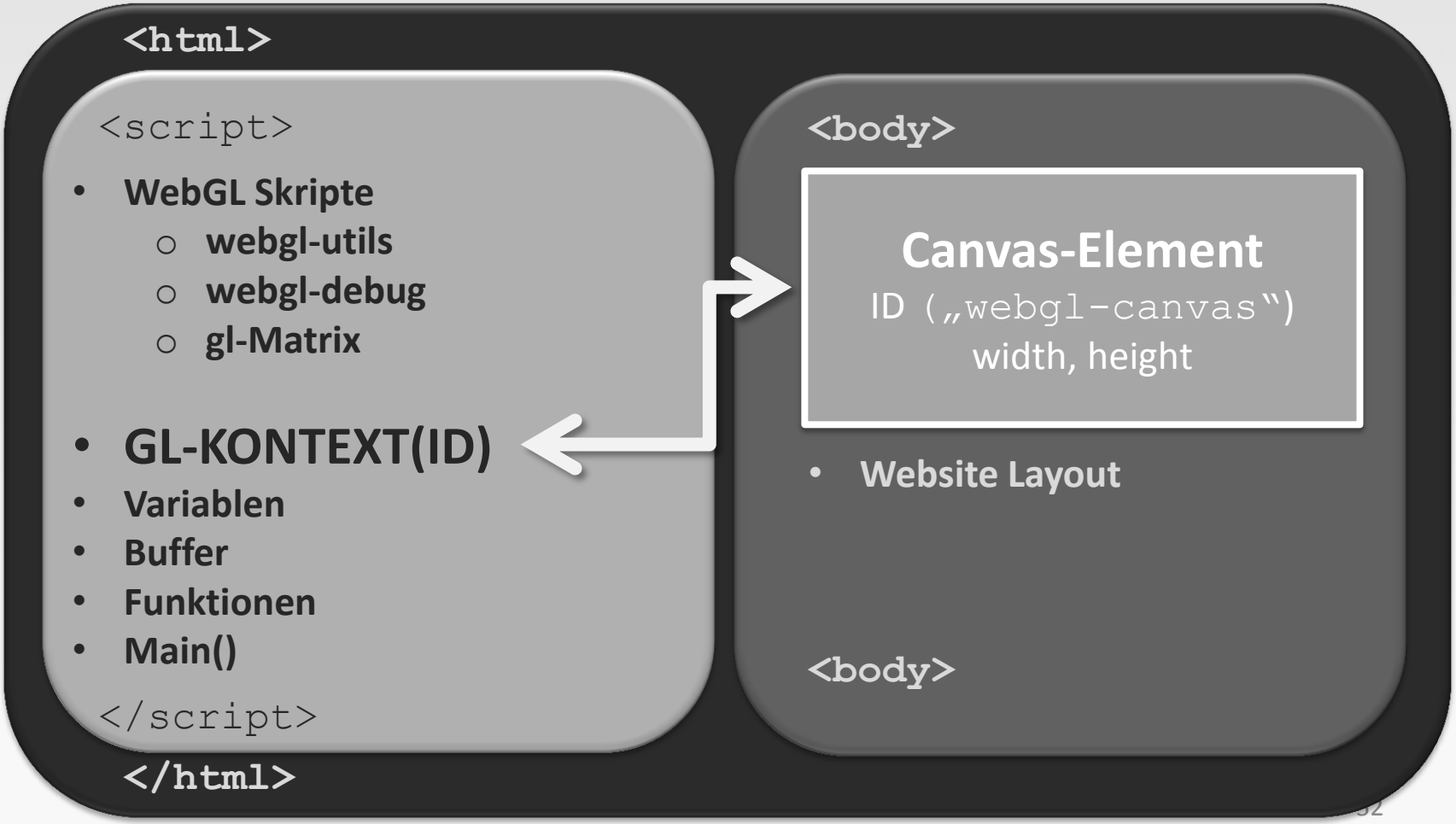
WebGL

Vergleich mit OpenGL

OpenGL	WebGL
Programmierung	
C und C++ (Java via LWJGL)	JavaScript, HTML5 und OpenGL ES 2.0
Daten	
• Vertices • Indices	• Vertices • Indices
Funktionsumfang	
• **OpenGL 4.3 (Aug. 2012)** • Vertex Shader • Fragment Shader • Geometry Shader • Tesselation Shader • Compute Shader + **OpenCL**	• **≈ OpenGL 2.0 (Sep. 2004)** • Vertex Shader • Fragment Shader + **WebCL**
Browserunterstützung	
• JavaApplet	

Graphics Pipeline

WebGL
Programmaufbau



WebGL

Hello Racer

WebGL

Autokonfigurator