

# Computergraphik II

## 1. Einführung

Timon Zietlow

Graphische Datenverarbeitung und Visualisierung  
Fakultät für Informatik



TECHNISCHE UNIVERSITÄT  
CHEMNITZ

Sommersemester 2019

# Inhalt für heute

- 1 Organisatorisches
- 2 Ausblick Computergraphik 2
- 3 Recap Computergraphik 1
- 4 Einführung in modernes OpenGL

- Konzepte
  - Renderpipeline
  - GL-Objekte
  - Beispiel
- |          |          |
|----------|----------|
| Konzepte | Konzepte |
| 渲染管线     | GL对象     |
| 例子       |          |

组织

Outlook计算机图形2

回顾一下计算机图形1

现代OpenGL简介

# Organisation der Computergraphik II

- Vorlesung: wöchentlich, Do 15:30, Raum 1/205 (neu: A10.205)
- Übung
  - wöchentlich, Mi 11:30 **Raum 1/B101 (neu: A11.201)**
- Prüfungsvorleistungen (PVLs)
  - Mehrere im Verlauf des Semesters gestellte Programmieraufgaben
  - von jedem Studenten **selbstständig** anzufertigen

Bitte in OPAL-Kurs: „[571110 Computergraphik II SS 2019](#)“ einschreiben!

- Übungsmaterialien, PVL-Aufgaben und -Abgaben, Projektupload
- <https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/20055326726/CourseNode/94396602780253>

# Prüfung

- **eine** Prüfungsleistung bestehend aus
  - 90-minütiger Klausur
- Voraussetzung zur Zulassung:  
50% der PVLs bestanden

**Beachten Sie Ihre Studienordnung!**

- **freiwilliges** Programmierprojekt
  - Implementierung eines oder mehrerer Verfahren (nach Rücksprache),  
die **nicht** Teil der Übung sind
  - bis zu 15% der Klausurpunkte als Bonus

# Voraussetzungen

- Computergraphik I

- mathematische Grundlagen

- lineare Algebra, Matrix-Kalkül
  - geometrisches Grundverständnis

hilfreiche Module:

- Mathematik für Informatiker
  - Grundlagen der Computergeometrie

- Programmierkenntnisse

- Übungsframework: C++
  - Umgang mit Werkzeugen zur Software-Entwicklung
    - Framework verwendet MS Visual Studio (Windows)
    - alternativ: Linux-Support (gcc + Makefile)

hilfreiche Module:

- Algorithmen und Programmierung, Datenstrukturen
  - Grundlagen der Informatik (Dr. Müller)

# Lehrveranstaltungen an der Professur GDV

- Praxisorientierte Einführung in die Computergraphik
- Computergraphik I
- Computer-Aided Geometric Design
- Digitale Objektrekonstruktion
- Computergraphik II
- Virtuelle Realität
- Solid Modeling
- Hauptseminar Computergraphik

Praktika und Abschlussarbeiten  
(individuell mit jeweiligem Betreuer)

# Lernziele

## ■ Verfahren

- "Wie gehen Schatten?"
- "Animationen?"
- "Wie bekomme ich mehr Details in die Scene?"
- "Realistischere Beleuchtung?"
- "Mehr Lichtquellen?"
- "Alles in Echtzeit?"
- Umgang mit modernen Rendering APIs

阴影怎么样？“  
动画？”  
如何在场景中获得更多细节？“  
更真实的照明？”  
更多的光源？“  
一切都是实时的？”  
处理现代渲染API

# Lernziele

- Verfahren
- Informationstechnisches Allgemeinwissen
  - SIMD, parallele Programmierung
  - Datenstrukturen für parallele Algorithmen
  - Datenformate
  - Caching

# Lernziele

- Verfahren
- Informationstechnisches Allgemeinwissen
- Schaffung von Grundlagen zur eigenen Weiterbildung!
  - Alle Verfahren waren oder sind State of the Art
  - Grundlagen für die aktuelle Forschung

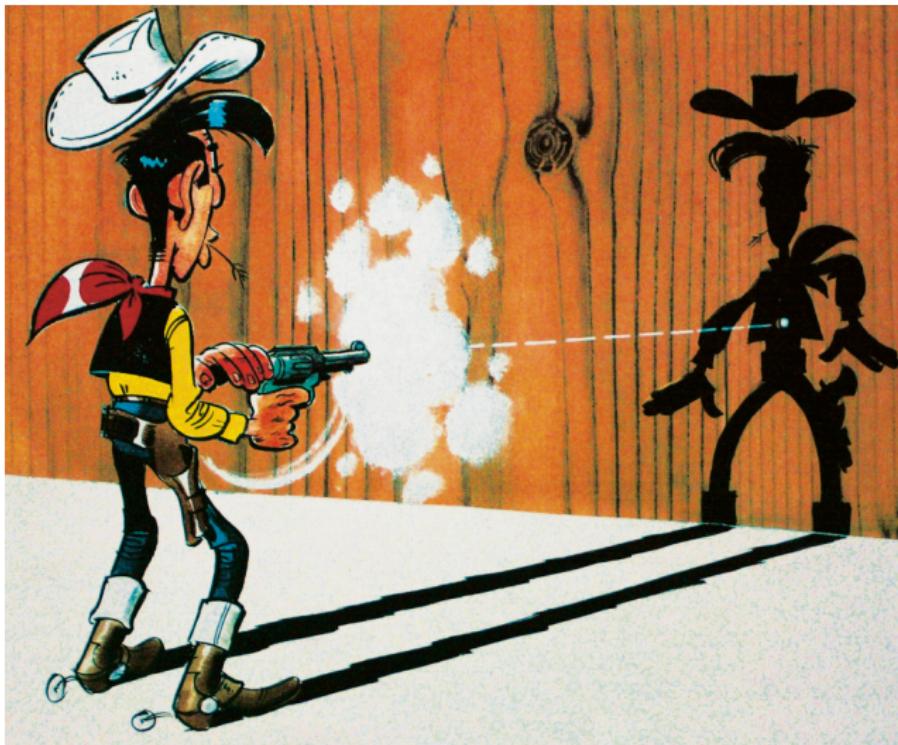
# Texturen



# Postprocessing 后期处理

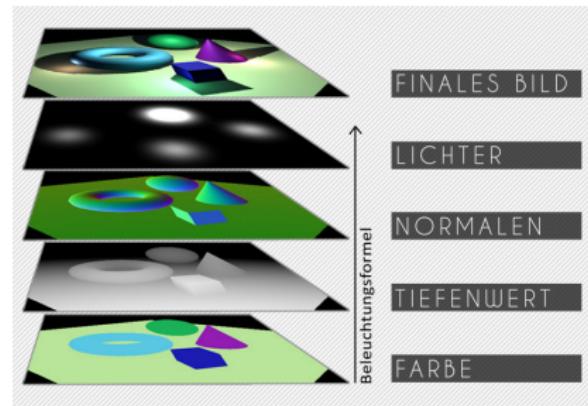
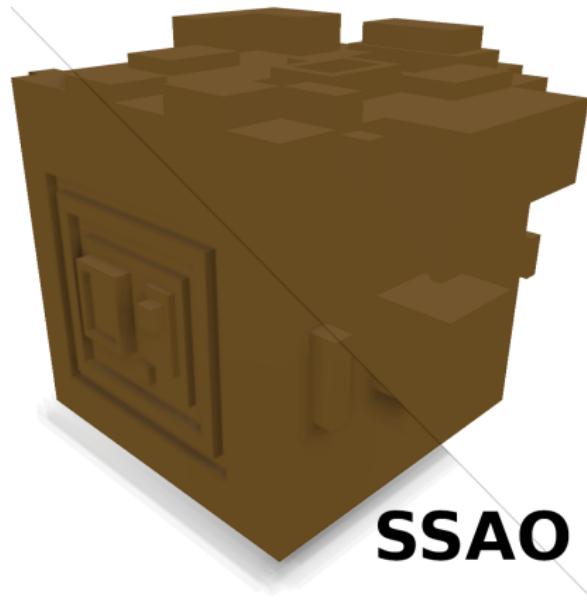


# Schatten



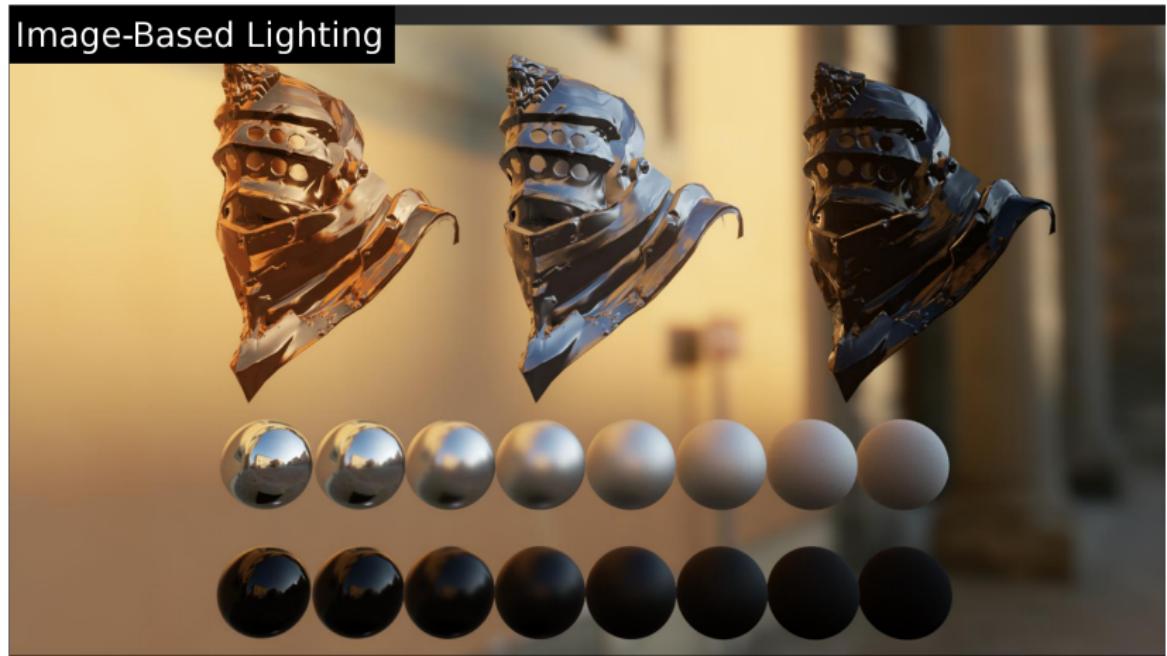
Quelle: [Lucky Luke \(EHAPA VERLAG\)](#) Bild von [tageswoche.ch](#)

# Screenspace und Screenspace-Verfahren



Quelle: [wikimedial.org](https://wikimedial.org) by McMalloc (CC BY-SA 3.0)

# Physically Based Rendering & Image Based Lighting

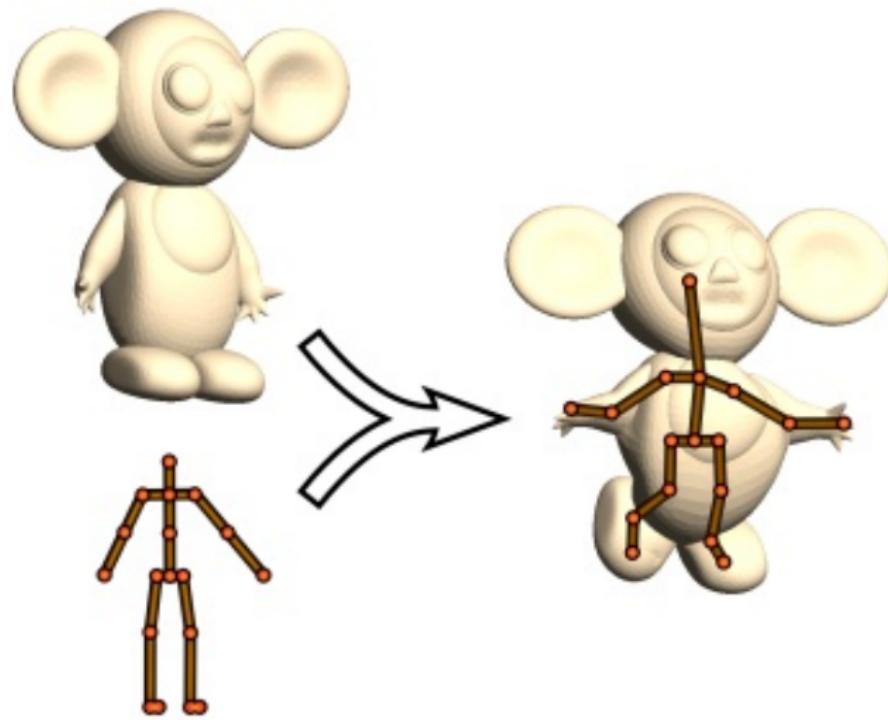


Quelle: [Real Shading in Unreal Engine 4 by Brian Karis](#)

# Reflexionen

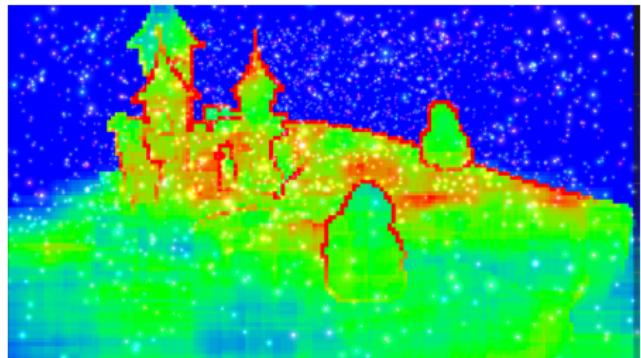


# Animationen



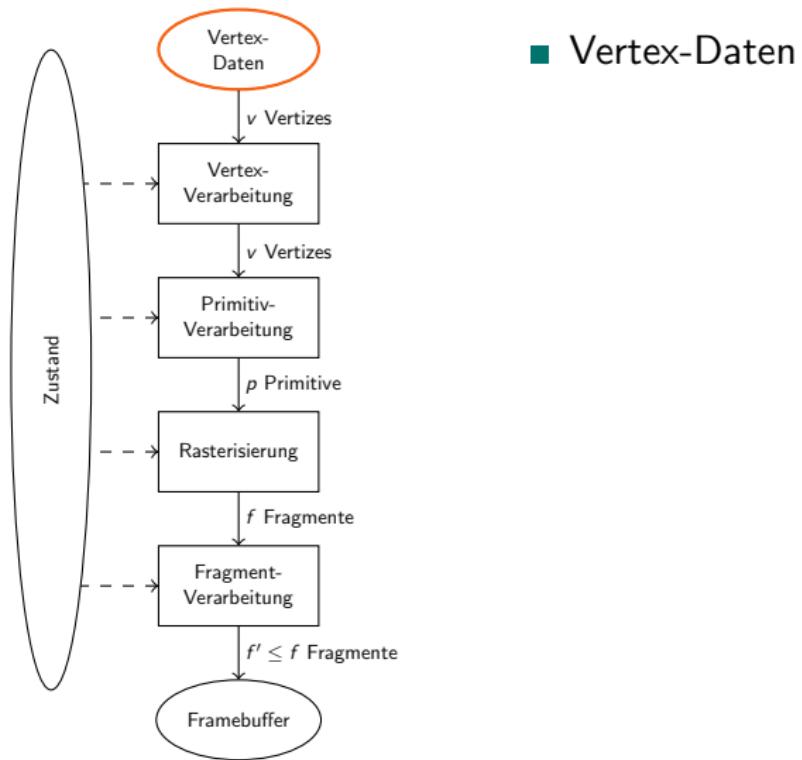
Quelle: Automatic Rigging and Animation of 3D Characters by Ilya Baran & Jovan Popović

# Erweiterungen und Entwicklungen in der Renderingpipeline

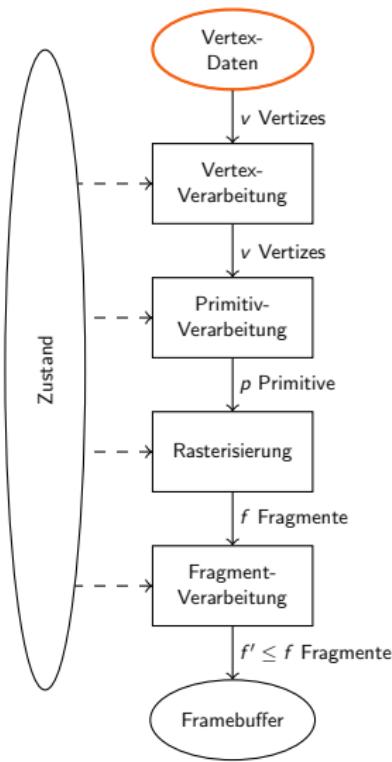


Quelle: Forward+: Bringing Deferred Lighting to the Next Level by Takahiro Harada, Jay McKee, and Jason C. Yang

# Konzeptioneller Aufbau einer Rendering-Pipeline



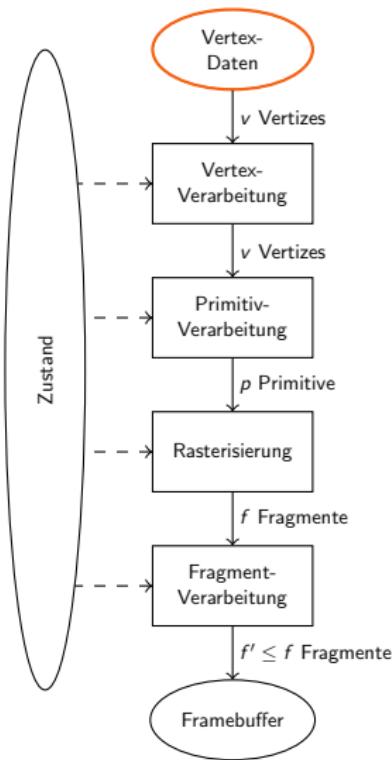
# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Vertex-Daten

- (definieren die Geometrie der Objekte, die dargestellt werden sollen.)

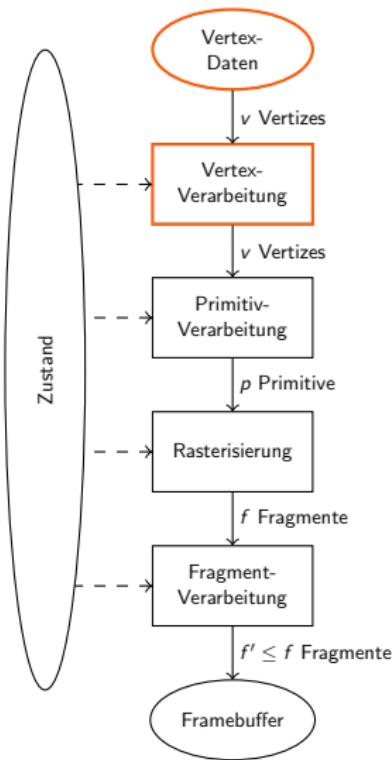
# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Vertex-Daten

- (definieren die Geometrie der Objekte, die dargestellt werden sollen.)
- Vertex: Satz von Attributen 属性集

# Konzeptioneller Aufbau einer Rendering-Pipeline



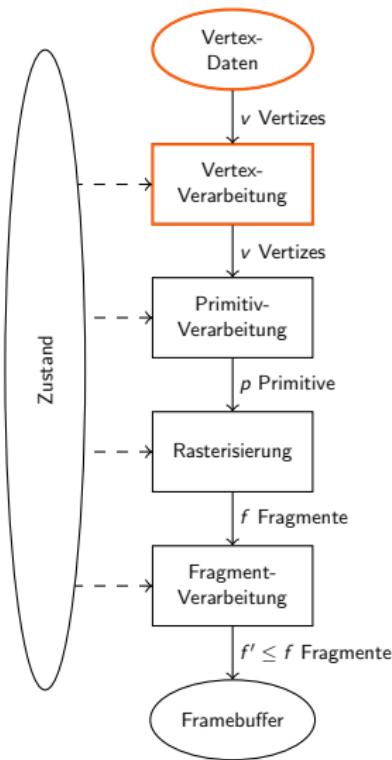
## ■ Vertex-Daten

- (definieren die Geometrie der Objekte, die dargestellt werden sollen.)
- Vertex: Satz von Attributen

## ■ Vertex-Verarbeitung

- Operationen auf individuellen Vertizes  
对各个顶点的操作

# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Vertex-Daten

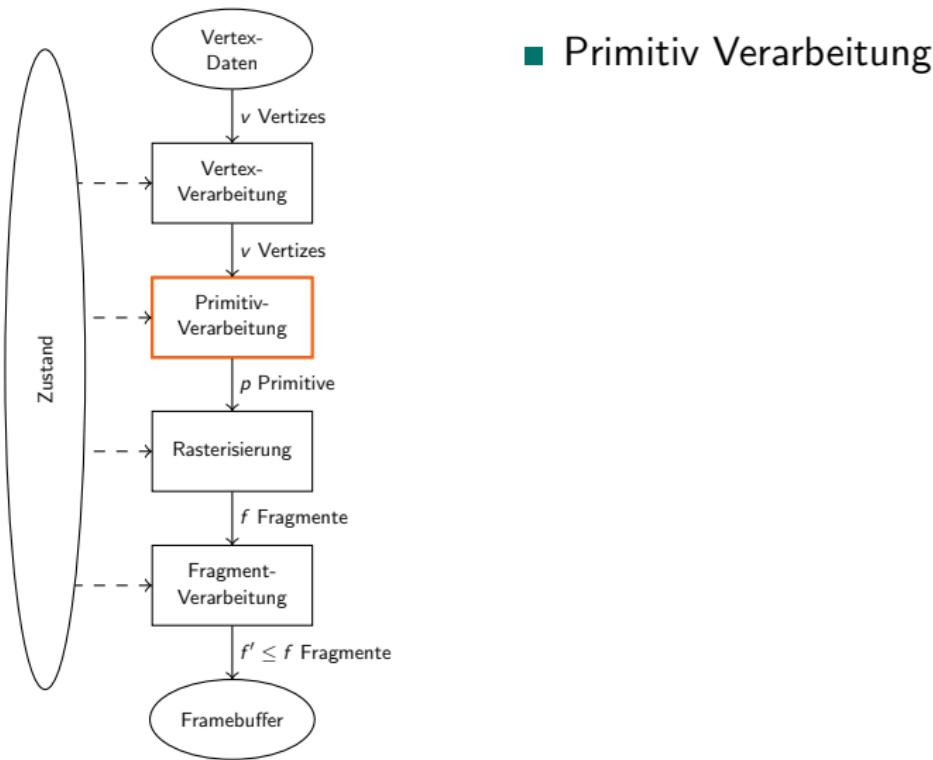
- (definieren die Geometrie der Objekte, die dargestellt werden sollen.)

- Vertex: Satz von Attributen

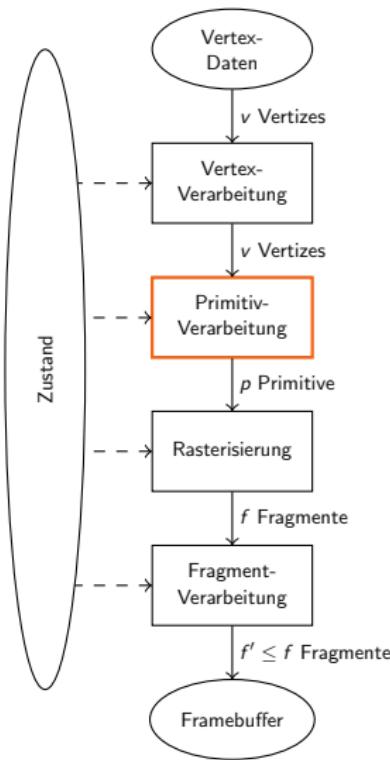
## ■ Vertex-Verarbeitung

- Operationen auf individuellen Vertizes
- **Object Space → Clip Space**

# Konzeptioneller Aufbau einer Rendering-Pipeline

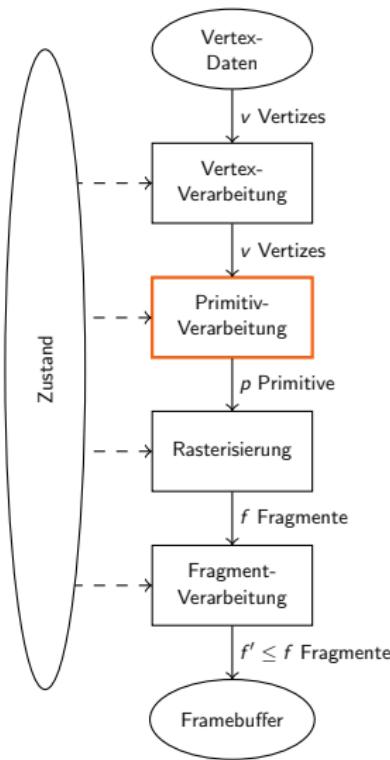


# Konzeptioneller Aufbau einer Rendering-Pipeline



- Primitiv Verarbeitung
  - Cipping

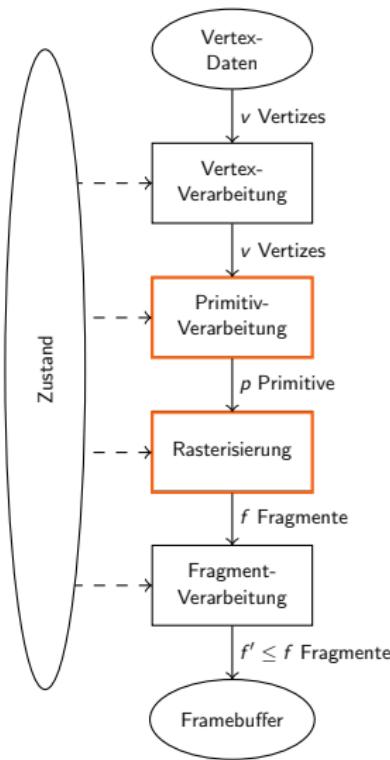
# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Primitiv Verarbeitung

- Culling
- Backface Culling

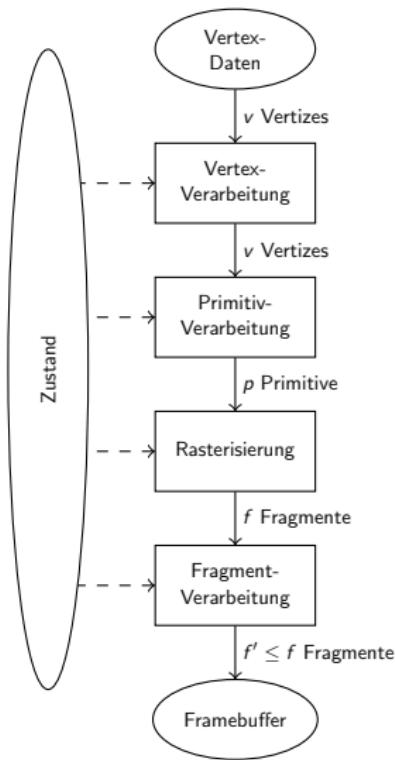
# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Primitiv Verarbeitung

- Culling
- Backface Culling
- **Clip Space → Window Space**

# Konzeptioneller Aufbau einer Rendering-Pipeline

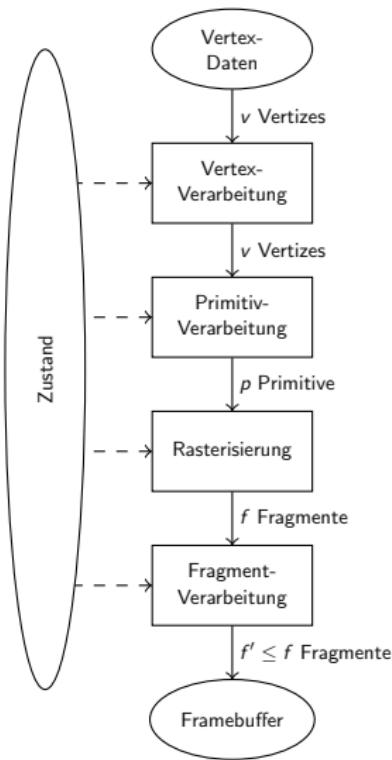


## ■ Primitiv Verarbeitung

- Culling
- Backface Culling
- **Clip Space → Window Space**

## ■ Rasterisierung

# Konzeptioneller Aufbau einer Rendering-Pipeline



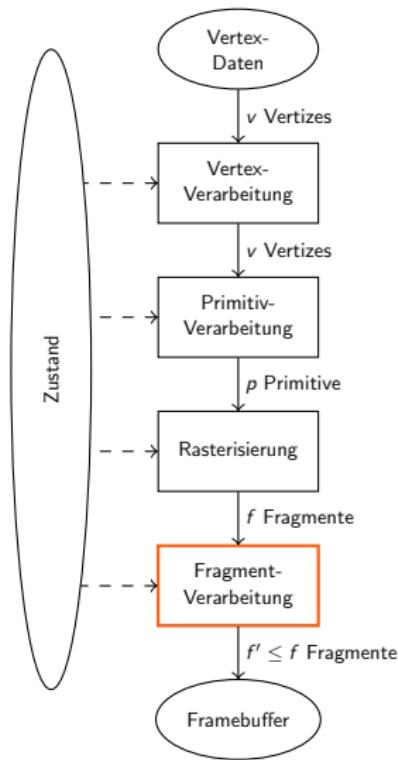
## ■ Primitiv Verarbeitung

- Culling
- Backface Culling
- **Clip Space → Window Space**

## ■ Rasterisierung

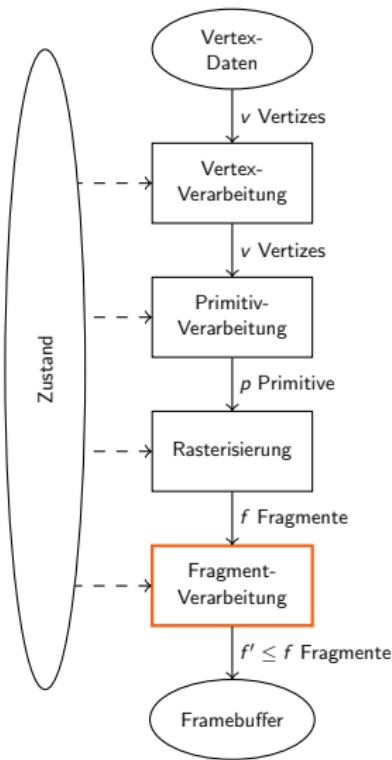
- interpolation der *varyings*  
变换的插值

# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Fragment Verarbeitung

# Konzeptioneller Aufbau einer Rendering-Pipeline

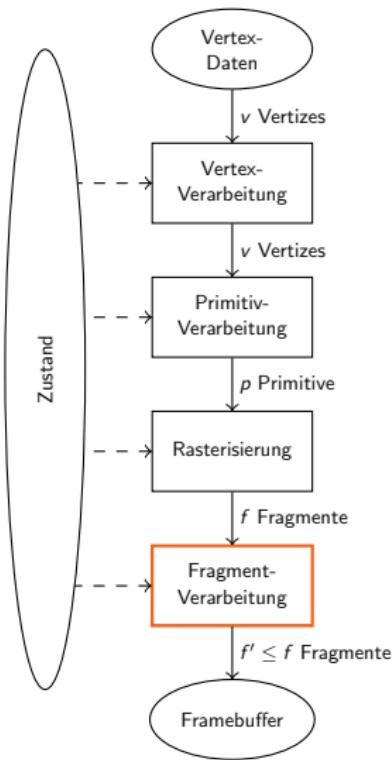


## ■ Fragment Verarbeitung

- Early Depth-Test

- Late Depth-Test

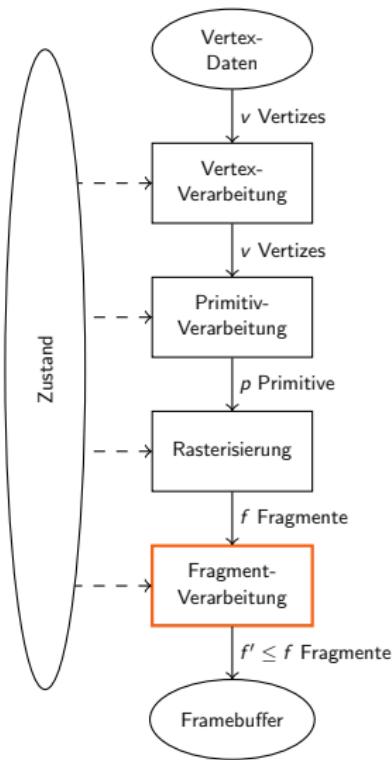
# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Fragment Verarbeitung

- Early Depth-Test
- Fragment Shader: Berechnung der Farbe des Fragments
- Late Depth-Test

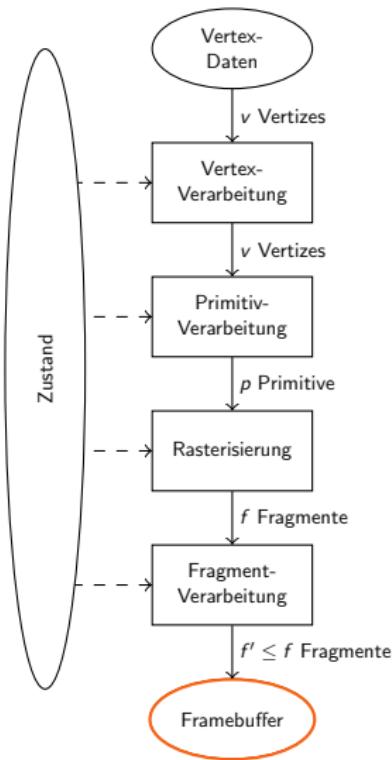
# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Fragment Verarbeitung

- Early Depth-Test
- Fragment Shader: Berechnung der Farbe des Fragments
- Late Depth-Test
- Blending: Mischen von Fragment- & Framebufferfarbe

# Konzeptioneller Aufbau einer Rendering-Pipeline



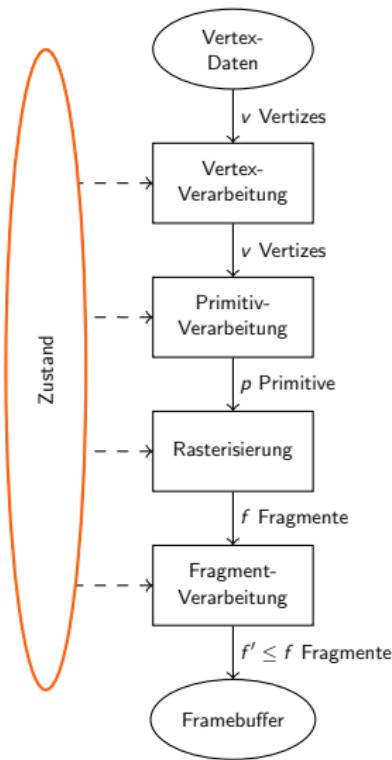
## ■ Fragment Verarbeitung

- Early Depth-Test
- Fragment Shader: Berechnung der Farbe des Fragments
- Late Depth-Test
- Blending: Mischen von Fragment- & Framebufferfarbe

## ■ Framebuffer

- Hält u.A. Farb- und Tiefen-Buffer

# Konzeptioneller Aufbau einer Rendering-Pipeline



## ■ Fragment Verarbeitung

- Early Depth-Test
- Fragment Shader: Berechnung der Farbe des Fragments
- Late Depth-Test
- Blending: Mischen von Fragment- & Framebufferfarbe

## ■ Framebuffer

- Hält u.A. Farb- und Tiefen-Buffer

## ■ Zustand

- Uniforms, Texturen, Sampler, Buffer, Einstellungen, ...

# Homogene Koordinaten

**Ziel:** affine Abb. ohne separaten Translationsvektor darstellen

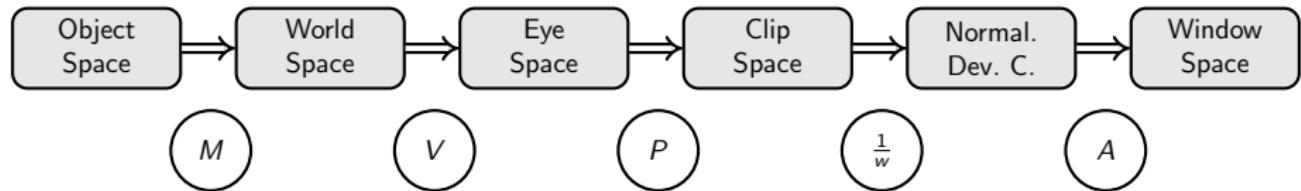
- Verwendung einer zusätzlichen Dimension

$$P = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} \quad \rightarrow \quad P_h = \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

- affine Abbildungen im 3D als  $4 \times 4$ -Matrizen darstellbar

$$F(P) = MP + \mathbf{d} \quad \rightarrow \quad F_h(P_h) = \begin{pmatrix} M & \mathbf{d} \\ 0 & 1 \end{pmatrix} P_h$$

# Überblick über die Koordinatentransformation



$M$  = Modelmatrix

$V$  = Viewmatrix

$P$  = Projektionsmatrix

$A$  = Viewport-Transformation-Matrix

# Koordinatentransformation

Hinrichtung:

$$\mathbf{p}^{\text{WS}} = \mathbf{M} \cdot \mathbf{p}^{\text{OS}}$$

$$\mathbf{p}^{\text{ES}} = \mathbf{V} \cdot \mathbf{p}^{\text{WS}}$$

$$\mathbf{p}^{\text{CS}} = \mathbf{P} \cdot \mathbf{p}^{\text{ES}} = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \mathbf{p}^{\text{OS}}$$

$$\mathbf{p}^{\text{NDS}} = \mathbf{p}^{\text{CS}} \cdot \frac{1}{\mathbf{p}_w^{\text{CS}}}$$

$$\mathbf{p}^{\text{SS}} = \mathbf{A} \cdot \mathbf{p}^{\text{NDS}}$$

Rückrichtung:

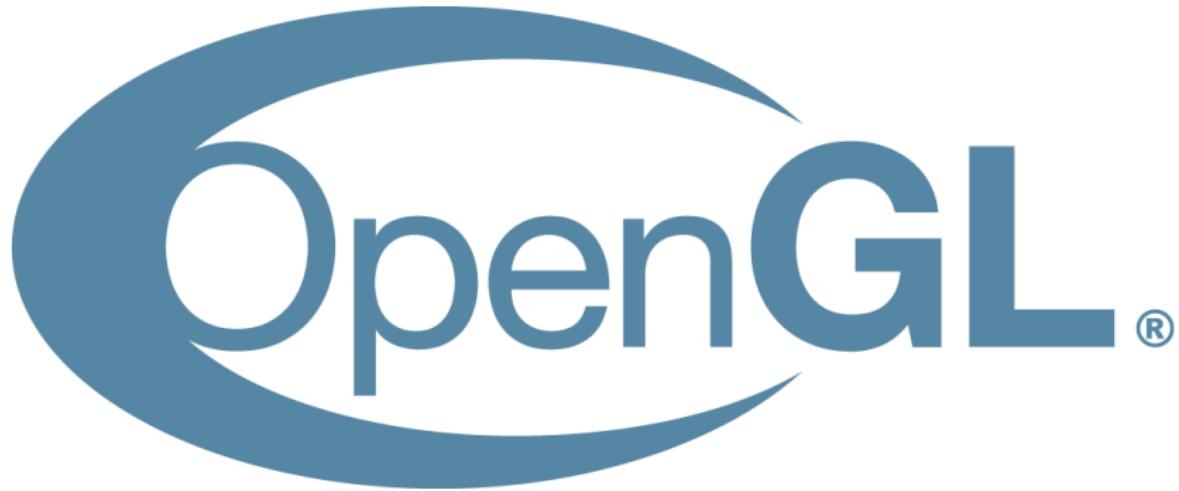
$$\mathbf{p}^{\text{NDS}} = \mathbf{A}^{-1} \cdot \mathbf{p}^{\text{SS}}$$

$$\mathbf{p}'^{\text{ES}} = \mathbf{P}^{-1} \cdot \mathbf{p}^{\text{NDS}}$$

$$\mathbf{p}^{\text{ES}} = \mathbf{p}'^{\text{ES}} \cdot \frac{1}{\mathbf{p}'_w}$$

$$\mathbf{p}^{\text{WS}} = \mathbf{V}^{-1} \cdot \mathbf{p}^{\text{ES}}$$

$$\mathbf{p}^{\text{OS}} = \mathbf{M}^{-1} \cdot \mathbf{p}^{\text{WS}}$$





- API zur Erzeugung von Echtzeit 3D-Visualisierungen.
  - Unabhängig von spezialisierter Hardware (GPUs)!
  - 1992 von SGI vorgestellt, seit 2006 von der Khronos Group verwaltet

# Khronos Group?

PROMOTER MEMBERS



Over 130 members worldwide  
Any company is welcome to join



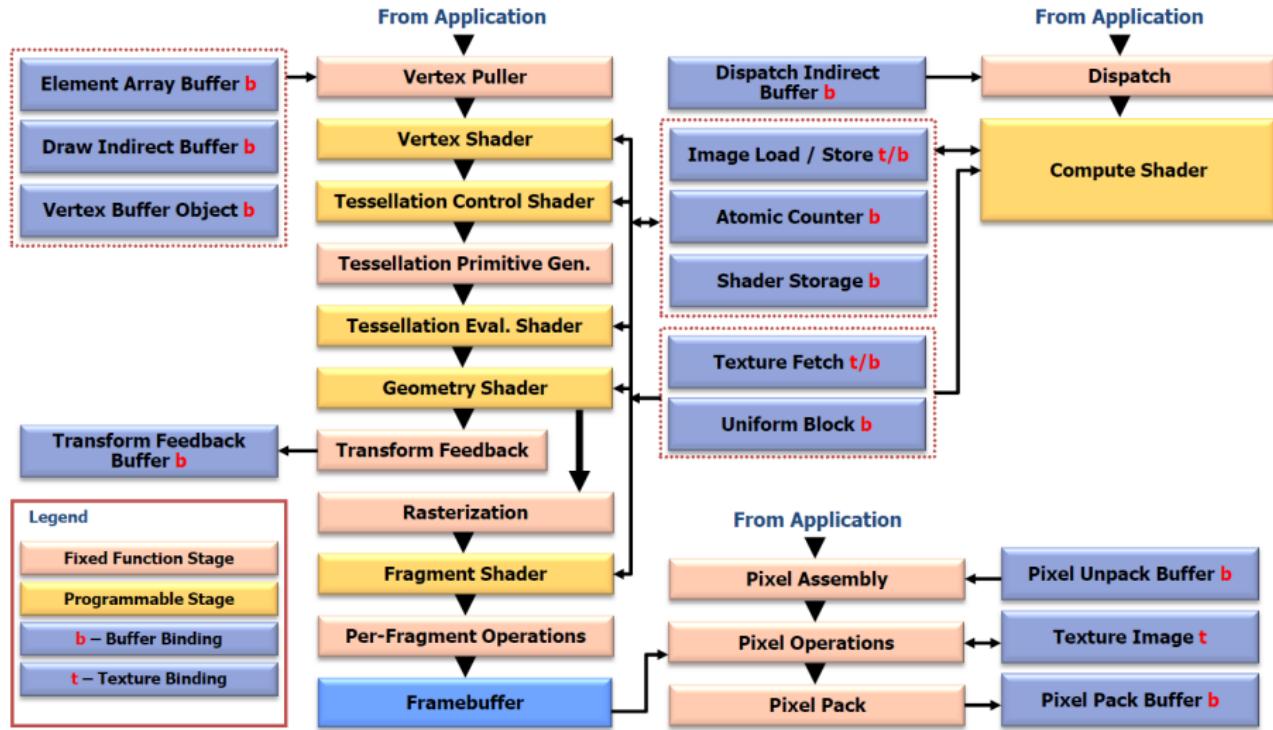
Quelle: [redgamingtech.com](http://redgamingtech.com)



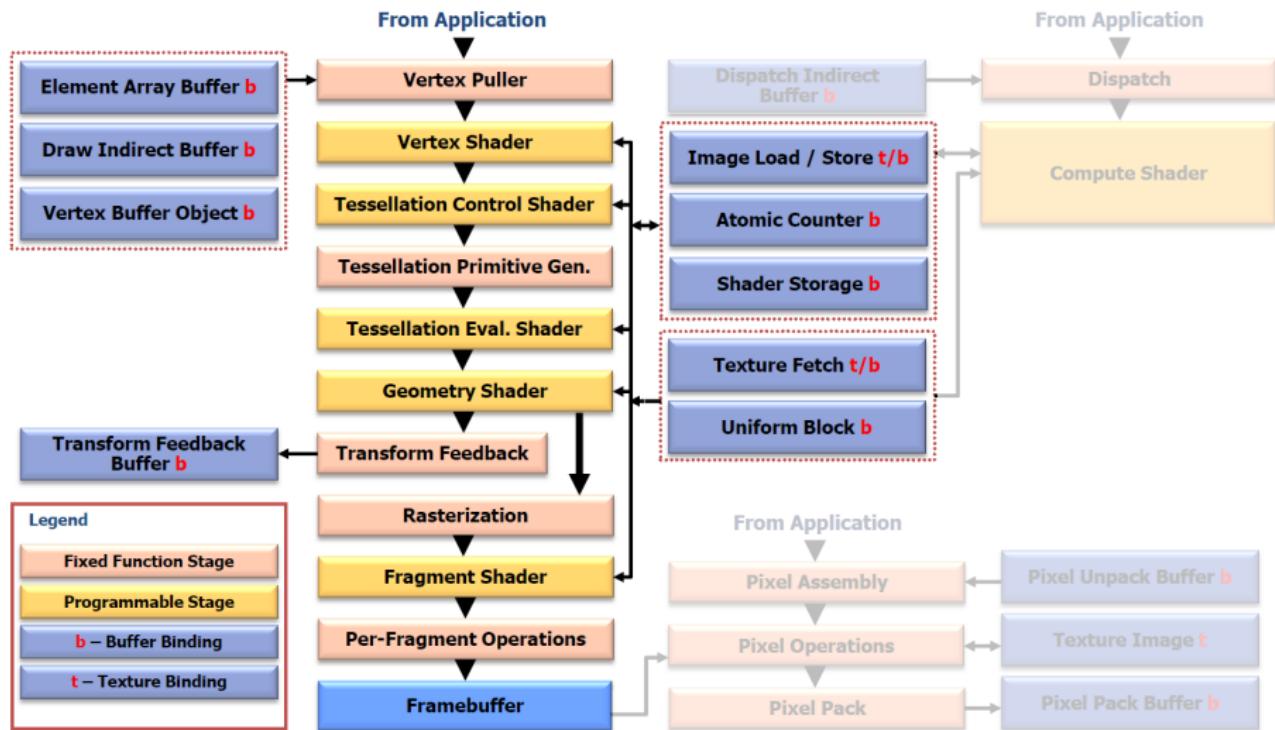
- API zur Erstellung von Echtzeit 3D-Visualisierungen.
  - Unabhängig von spezialisierter Hardware (GPUs)!
  - 1992 von SGI vorgestellt, seit 2006 von der Khronos Group verwaltet
  - Neueste Version 4.6 erschien 2017
- Inhalt:
  - Modelliert eine Rendering-Pipeline
  - Definiert eine Menge an Funktionen und deren Effekt auf die Pipeline
  - Definiert das Verhalten der Pipeline



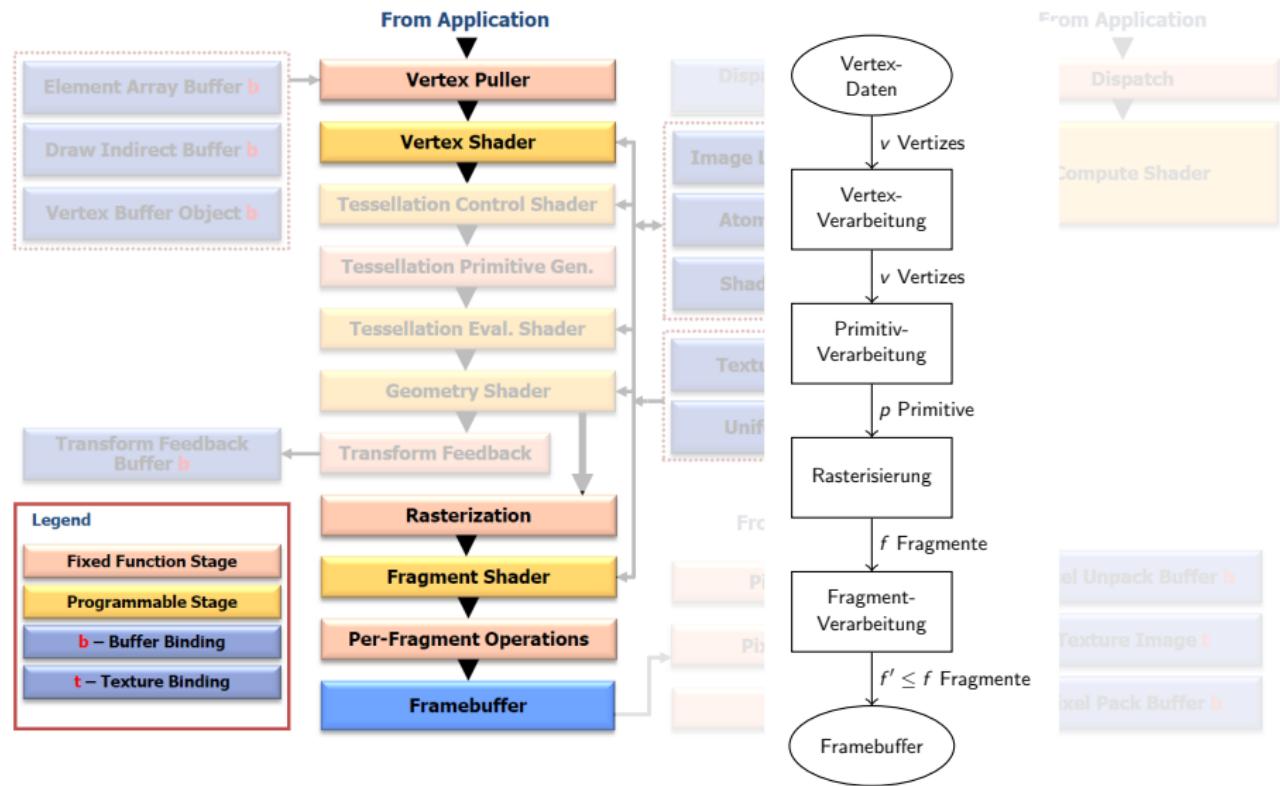
# GL-Pipeline



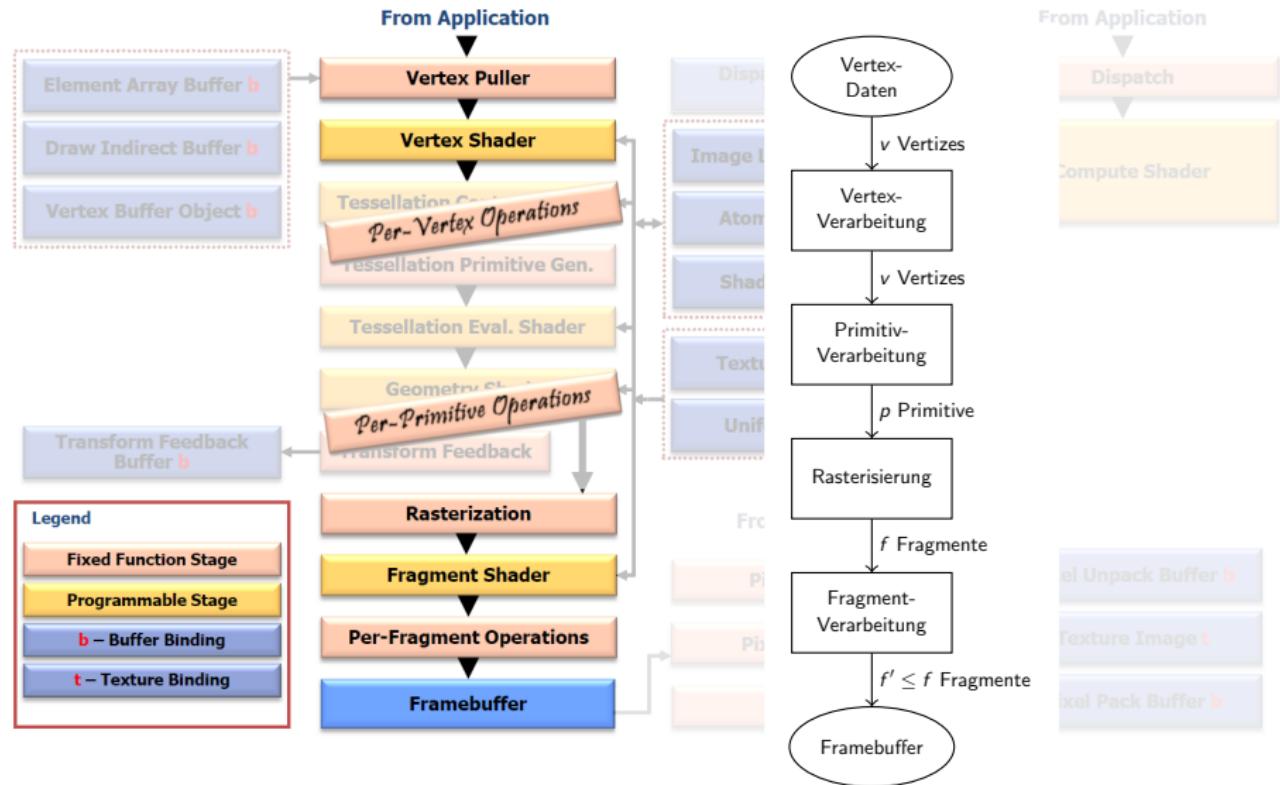
# GL-Pipeline



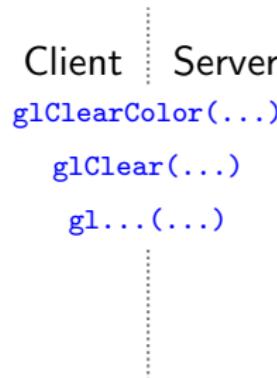
# GL-Pipeline



# GL-Pipeline



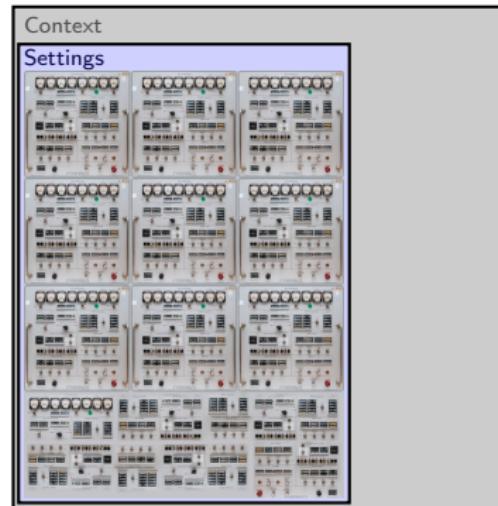
# OpenGL-Konzepte



- Trennung in Client und Server                           客户端和服务器中的分离
  - Server ist die eigentliche GL-Implementation   服务器是实际的GL实现
  - Client ist das Programm welches eine GL-Implementation nutzt   客户端是使用GL实现的程序
- GL-Funktionen bilden Schnittstelle zur Kommunikation mit dem Server   GL功能构成与服务器通信的接口
- Client und Server können asynchron arbeiten   客户端和服务器可以异步工作

# OpenGL-Context

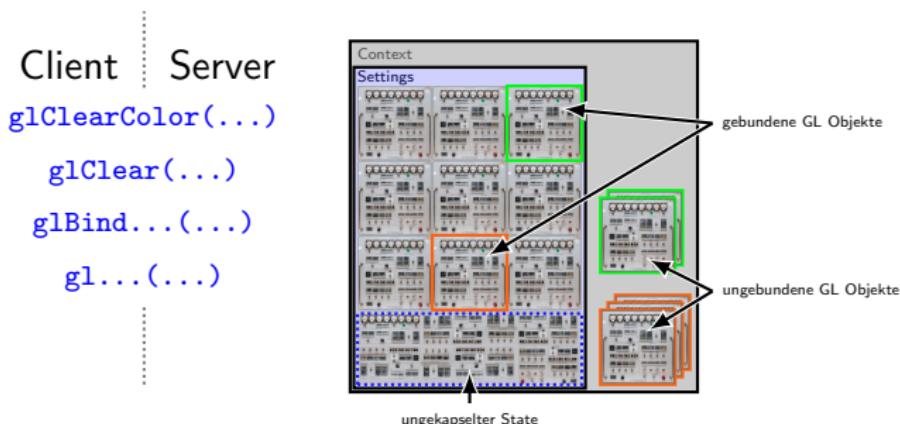
Client | Server  
`glClearColor(...)`  
`glClear(...)`  
`gl....(...)`



- GL-Context fasst den Gesamtzustand einer GL-Pipeline zusammen  
→ ein Server kann prinzipiell mehrere Kontexte verwalten!

GL-Context总结了GL管道的整体状态  
服务器原则上可以管理多个上下文！

# OpenGL-Objects



有些状态被分组为所谓的对象，可以设置（绑定）'en bloc'

- Manche States werden zu sog. Objekten zusammengefasst und können so 'en bloc' gesetzt (gebunden) werden

- mit Objekten kann nur interagiert werden, wenn diese gebunden sind<sup>1</sup> (Bind to Modify) 对象只能在绑定时进行交互 (绑定到修改)
- Buffer, Shader, Vertex Arrays, Texturen, etc.
- Objects werden mit sog. **Namen** (einem **GLuint**) identifiziert
- notwendig zum Rendern: Shaderprogramm und Vertex Array Object

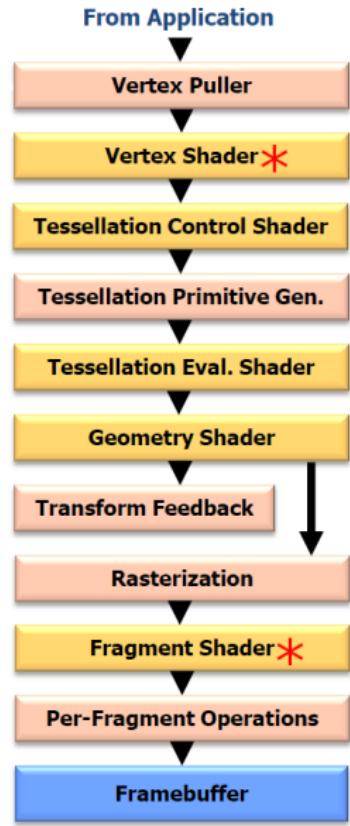
<sup>1</sup>Ausgenommen Teile der Shader-States und seit GL4.5 Direct State Access

# Shader-Programs

- Repräsentiert einen 'Satz' an Shadern  
(mindestens Vertex- und Fragment-Shader<sup>a</sup> \*)
- Kapseln, unter anderem, Uniforms, Uniform- und Attribute Locations
- Shader werden in GLSL geschrieben
- Die GL compiliert Shader
- Mehrere Shader werden zu einem Programm gelinkt

---

<sup>a</sup>Ausgenommen Computeshader



# Shader - Beispiel

Vertex Shader:

```
// Attributes  
in vec4 pos;  
// Uniforms  
uniform mat4 MVP;  
// Varying output  
out vec4 vclr;  
  
void main()  
{  
    gl_Position = MVP * pos;  
    vclr = vec4(0.42, 0.54, 0.15, 1);  
}
```

Fragment Shader:

```
// Varying input  
in vec4 vclr;  
// FS output  
out vec4 fclr;  
  
void main()  
{  
    fclr = vclr;  
}
```

# Locations

- Uniforms und Attribute können beliebige Namen haben
- Zuweisung von **Locations** zur eindeutigen Identifikation
- Zuweisung kann im Shader erfolgen, oder vom Client gesetzt bzw. abgefragt werden<sup>2</sup>

---

<sup>2</sup>`glGetUniformLocation`, `glBindAttribLocation`

# Shader - Beispiel

Vertex Shader:

```
// Attributes
layout(location = 0) in vec4 pos;
// Uniforms
layout(location = 0) uniform mat4 MVP;
// Varying output
out vec4 vclr;

void main()
{
    gl_Position = MVP * pos;
    vclr = vec4(0.42, 0.54, 0.15, 1);
}
```

# Erstellen eines Program Objects

Die Strings `char* vsc,fsc` enthalten den Code für Vertex- und Fragment-Shader.

```
GLuint vs,fs;  
// Create shader objects  
vs=glCreateShader(GL_VERTEX_SHADER);  
fs=glCreateShader(GL_FRAGMENT_SHADER);  
// Add the source code to the objects  
glShaderSource(vs, 1, (const GLchar**)&vsc, NULL);  
glShaderSource(fs, 1, (const GLchar**)&fsc, NULL);  
// Compile shaders  
glCompileShader(vs);  
glCompileShader(fs);
```

# Erstellen eines Program Objects

```
// Create Program
GLuint prg = glCreateProgram();
// Attach the shaders
glAttachShader(prg, vs);
glAttachShader(prg, fs);
// Link the program
glLinkProgram(prg);
// Use (bind) the program
glUseProgram(prg);
```

# Buffer Objects

- Repräsentiert 'ein Stück Speicher' auf Serverseite
- Kann verwendet werden um verschiedene Daten zu speichern
  - Vertex-Attribute
  - Uniform-Daten
  - Vertex-Indices
  - Bild-Daten ( $\neq$  Texturen)
- Entsprechend gibt es verschiedene 'buffer binding points'

# Buffer - Beispiel

In dem Array `GLfloat data[9]`; sind die 3D-Positionen für 3 Vertices gespeichert. Wir wollen diese auf Serverseite in einem Buffer speichern.

`GLuint bn;` 在数组GLfloat数据[9]; 3D位置存储3个版本。 我们希望将它们保存在服务器端的缓冲区中。

```
// Generate a buffer name
glGenBuffers(1,&bn);
// Bind the buffer name
glBindBuffer(GL_ARRAY_BUFFER, // binding point
             bn);           // buffer name
// Create buffer (reserve memory) and copy data
glBufferData(GL_ARRAY_BUFFER,      // binding point
              9*sizeof(GLfloat)), // size (bytes)
              data,            // data
              GL_STATIC_DRAW); // usage hint
```

# GL Vertex Array Objects (VAO)

- Kapselt die Zuordnung von Buffern zu Attributen (Vertex Attrib. Pointers)
  - In welchem Buffer liegen die Daten für welche Attribute Location?
  - Wie ist deren Format?
  - Welche Attribute sind aktiv?
  - In welchem Buffer liegen die Index-Daten (indexed rendering)?

# VAO - Beispiel

In dem Buffer `GLuint buff;` sind die 3D-Positionen (location 0) der drei Vertices gespeichert. 在缓冲区`GLuint buff;`存储三个顶点的3D位置（位置0）。

```
GLuint vao;
// Generate & bind the VAO
 glGenVertexArrays(1,&vao);
 glBindVertexArray(vao);
// Bind the buffer to GL_ARRAY_BUFFER
 glBindBuffer(GL_ARRAY_BUFFER,buff);
 glVertexAttribPointer(
    0, // location 0 (position)
    3, // number of elements (3->vec3)
    GL_FLOAT, // type of each element
    GL_FALSE, // normalize (only on int types)
    sizeof(float)*3, // stride
    (const void*) 0); // offset
 glEnableVertexAttribArray(0);
```

# Draw Call

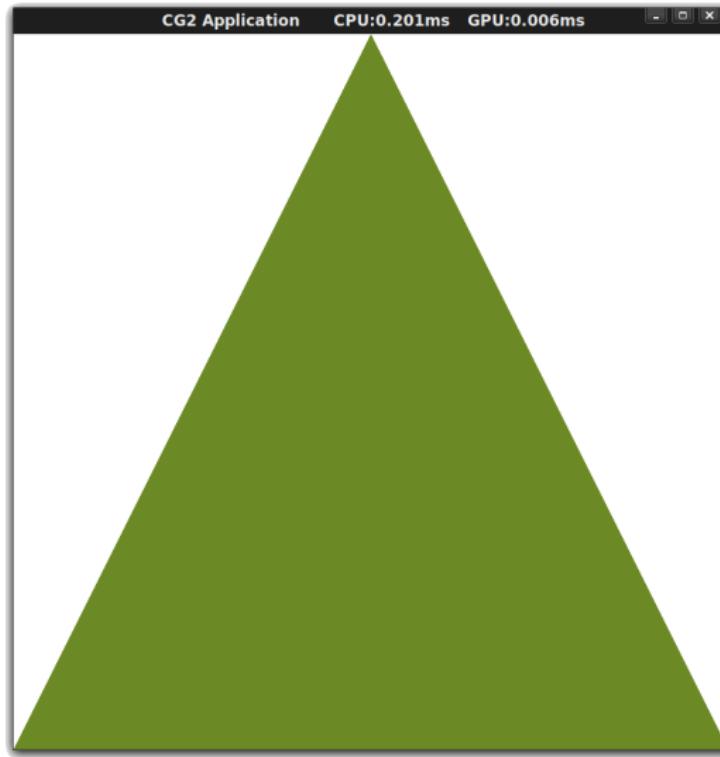
- vao ist das konfigurierte VAO vao是配置的VAO
- float** mat[16] ist  $P \cdot V \cdot M$

```
glUseProgram(prg);
glUniformMatrix4fv(
    0,           // uniform location 0 (MVP)
    1,           // number of matrices
    GL_FALSE,   // transpose?
    mat);       // pointer to the data

glBindVertexArray(vao);

glDrawArrays(
    GL_TRIANGLES, // primitive
    0,           // first vertex
    3);          // number of vertices
```

# Ergebnis



That's all Folks!