

Computergrafik

**Vorlesung im Wintersemester 2014/15
Kapitel 7: OpenGL und Grafik-Hardware**

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

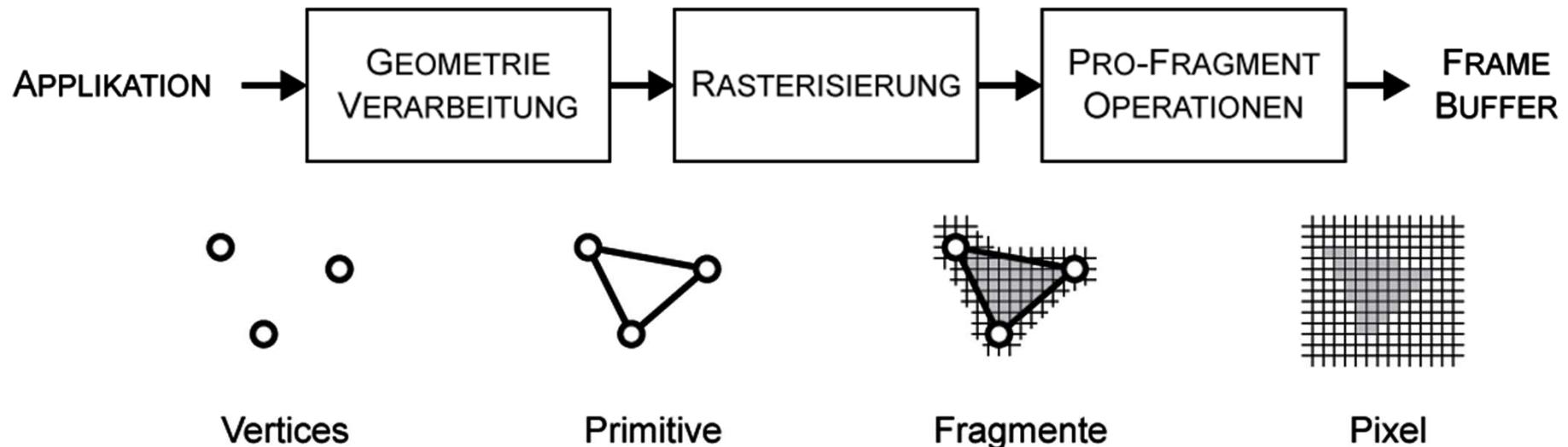


Inhalt

- ▶ Allgemeines und Historisches zu OpenGL
- ▶ kurze Einführung in GLUT und klassisches OpenGL
- ▶ modernes OpenGL
 - ▶ Shader Programmierung: OpenGL Shading Language
 - ▶ Vertex Arrays, Index Buffers, ...
 - ▶ Texturen
- ▶ Überblick zu speziellen Rendering-Techniken (Schatten, ...)

Interaktive Computergrafik

- ▶ Rasterisierung (in Verbindung mit Tiefenpuffer) ist effizient:
Dreiecke durchlaufen – nacheinander, aber unabhängig – alle dieselben
Verarbeitungsschritte
- ▶ Verarbeitung in der „Grafik-Pipeline“
- ▶ Konsequenzen:
i.d.R. nur direkte Beleuchtung (+ Schatten), Spezialeffekte und globale
Beleuchtungseffekte über mehrere Rendering-Durchgänge
- ▶ Grafik-Hardware verwenden über APIs wie OpenGL, Direct3D, ...



Was ist OpenGL?



3D Rendering API: OpenGL

- ▶ Plattform-, Hardware- und Programmiersprachen-unabhängige Programmierschnittstelle für Grafik-Hardware
- ▶ Fenstersystem-neutral (keine Events, Fenster, Menüs, ...)
- ▶ OpenGL implementiert die Grafik-Pipeline
 - ▶ geometrische Primitive: Punkte, Linien, Dreiecke, ...
 - ▶ Bild-Primitive: Images und Bitmaps
 - ▶ Texturen, Texturfilterung (Mip-Mapping etc.), z-Buffer
 - ▶ Stencil-Buffer, Accumulation-Buffer, Alpha-Blending
 - ▶ **klassisches OpenGL:** konfigurierbare Pipeline
 - ▶ Blinn-Phong-Beleuchtungsmodell und Gouraud-Shading
 - ▶ **modernes OpenGL:** programmierbare Stufen in der Pipeline
 - ▶ frei programmierbare Geometrieverarbeitung, Schattierungsberechnung und Tesselierung, ...



OpenGL – Weitere Merkmale



- ▶ Zustandsmaschine
 - ▶ die Verarbeitung in der Grafik-Pipeline wird konfiguriert
 - ▶ Beispielzustände: Beleuchtung, Material, Texturen, Shader, ...
- ▶ Client-Server Konzept
 - ▶ heute: Client und Server auf einem Rechner
 - ▶ „Client“ = Applikation, Daten im Hauptspeicher
 - ▶ „Server“ = Grafiktreiber- und Karte bzw. Graphics Processing Unit GPU
- ▶ Low-Level und Immediate Mode API:
 - ▶ keine höheren Modellierungs-/Animationskonzepte, kein Szenengraph
 - ▶ Immediate Mode: „ein OpenGL Befehl bewirkt sofortiges Rendern“
 - ▶ Gegenteil: „Retained Mode“-APIs verwalten Objekte mit Texturen etc.
- ▶ OpenGL Utility Library (GLU) bietet Modellierungsfunktionen an, z.B. Quadriken, NURBS
- ▶ Shading Language GLSL für Programmierung der Geometrie- und Fragmentverarbeitung direkt auf GPUs
- ▶ Erweiterungen („Extensions“) für Zugriff auf spezielle Fähigkeiten der GPUs

OpenGL – Historie und Entwicklung



- ▶ 1983 - 1992: SGI Graphics Library (GL)
 - ▶ nahezu proprietär (Portierung durch Dritte auf IBM, Sun,...)
 - ▶ 3D Rendering + Eingabeereignisse, Fenster, Menüs, Text, ...
 - ▶ Konkurrenz durch „Standards“: GKS-3D, PHIGS PLUS
- ▶ OpenGL www.opengl.org
 - ▶ definiert durch Architecture Review Board (ARB)/Khronos Group: SGI, Compaq, IBM, Intel, Microsoft, HP, Sun, Evans&Sutherland, NVIDIA, ATI, Apple, 3Dlabs, ...
 - ▶ 1992: OpenGL 1.0
 - ▶ 1992: OpenGL 1.2 (3D Texturen)
 - ▶ 2001: OpenGL 1.3 (Multi-Texturen, Cube Maps)
 - ▶ 2002: OpenGL 1.4 (Vertex Shader)
 - ▶ 2003: OpenGL 1.5 (Fragment Shader, Buffer Objects, GLSL EXT)
 - ▶ 2004: OpenGL 2.0 (GL Shading Language, MRT)
 - ▶ 2009: OpenGL 3.0 (Entfernen von „Altlasten“)
 - ▶ 2010: OpenGL 3.3 (OpenCL Einbindung)
 - ▶ 2010: OpenGL 4.1 (Tesselierung, binäre Shaderprogramme, 64-Bit FP)
 - ▶ 2011: OpenGL 4.2 (Packing, Atomic Counters, ...)
 - ▶ 2012: OpenGL 4.3 (read/write buffers, ...)
 - ▶ 2013: OpenGL 4.4 (bindless/sparse textures, ...)
 - ▶ 2014: OpenGL 4.5 (Direct State Access, ... für multi-threaded Anwendungen)
- ▶ OpenGL ES: OpenGL for mobile and embedded systems (seit 2004)

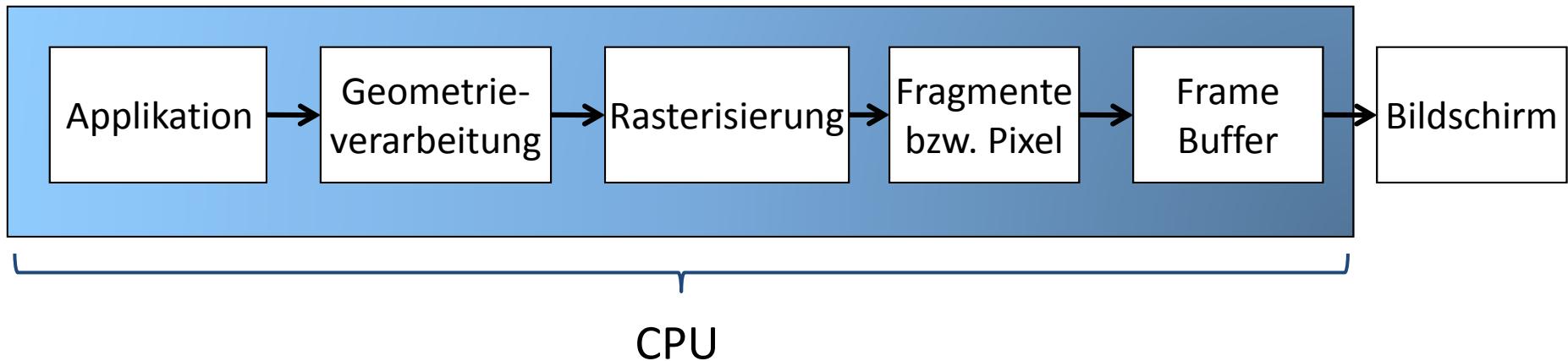
„Grafik Hardware“



- ▶ Sinclair ZX81 (~1982)
CPU generiert Video Signal



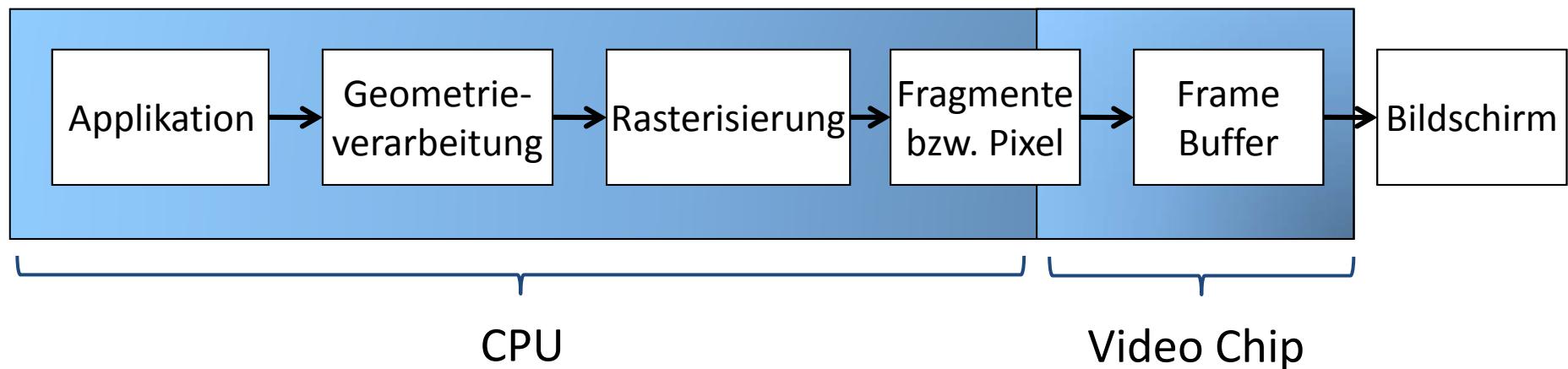
ZX81 Technical Specifications.
CPU: Z80A at 3.5 MHz
ROM: 8 Kbytes
RAM: (expandable) 1 (16) KByte
Display: TV
Text display: 32*15
Graphics display: 64 *44
Colours: black and white
Data Storage: Cassette Based
Power Supply: External 9v DC power pack



„Grafik Hardware“



- ▶ Commodore 64:
CPU schreibt in den Frame Buffer
Grafik Chip generiert Video Signal



„Grafik Hardware“



- ▶ Commodore 64:
CPU schreibt in den Frame Buffer
Grafik Chip generiert Video Signal
- ▶ GDC 2011: Classic Game Postmortem:
ELITE, David Braben

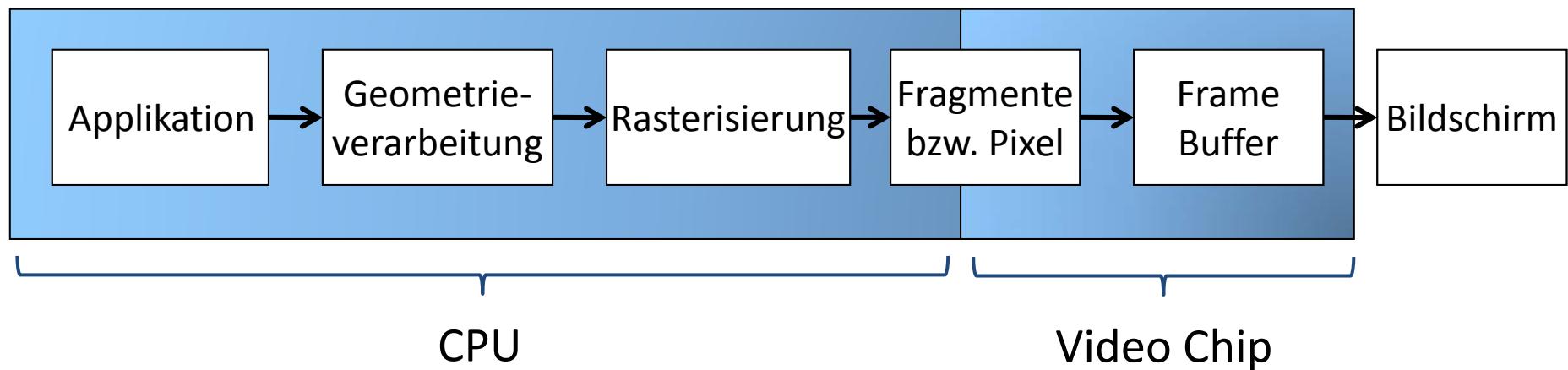
<http://www.gdcvault.com/play/1014628/Classic-Game-Postmortem>



„Grafik Hardware“



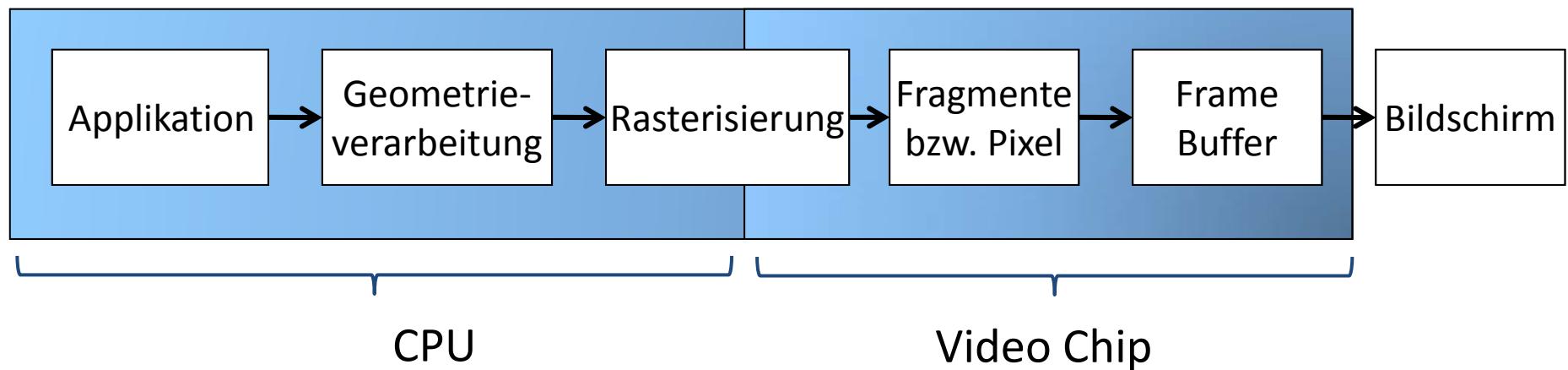
- ▶ Atari ST:
Hardware unterstützt einfache
2D Operationen für GUIs (Fenster, Menüs etc.)



Grafik Hardware



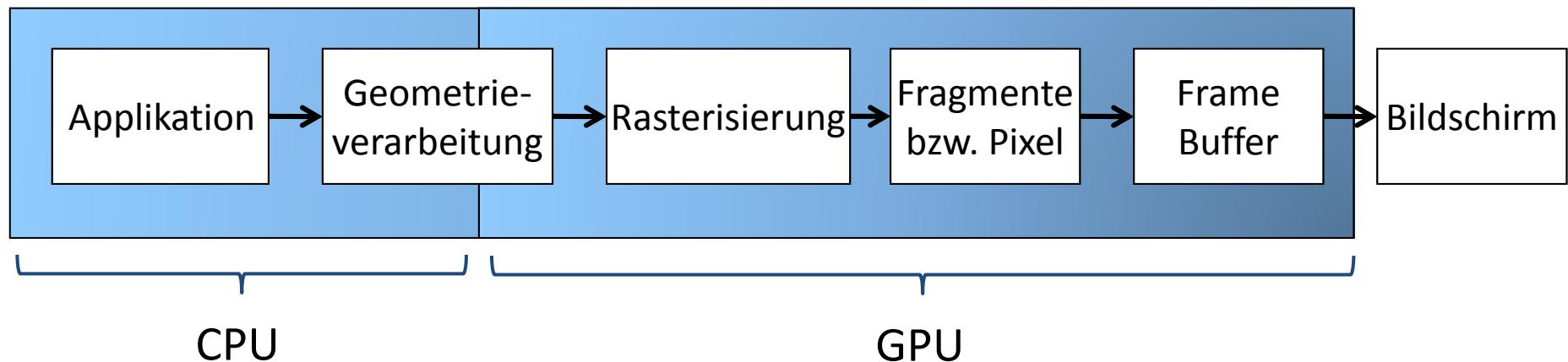
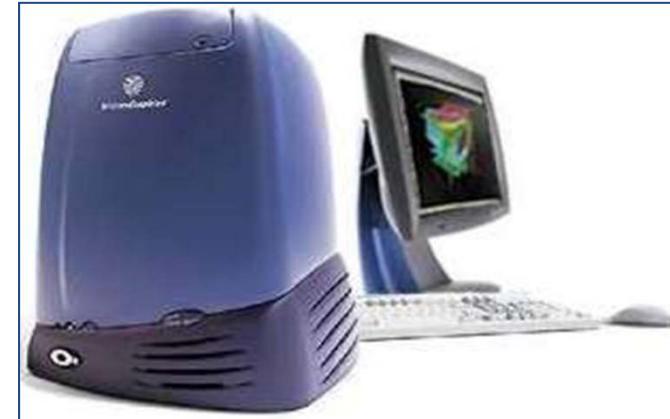
- ▶ SGI Indy:
Hardware rasterisiert Dreiecke
(Gouraud Shading, keine Texturen)



Grafik Hardware

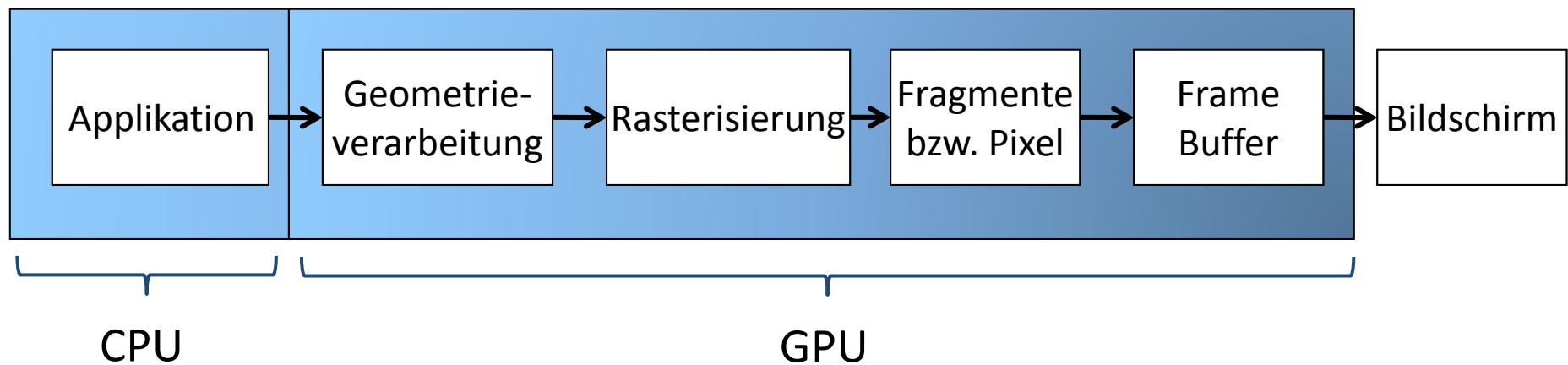


- ▶ SGI O₂:
Hardware für Rasterisierung
und Transformation
→ GPU (graphics processing unit)



Grafik Hardware

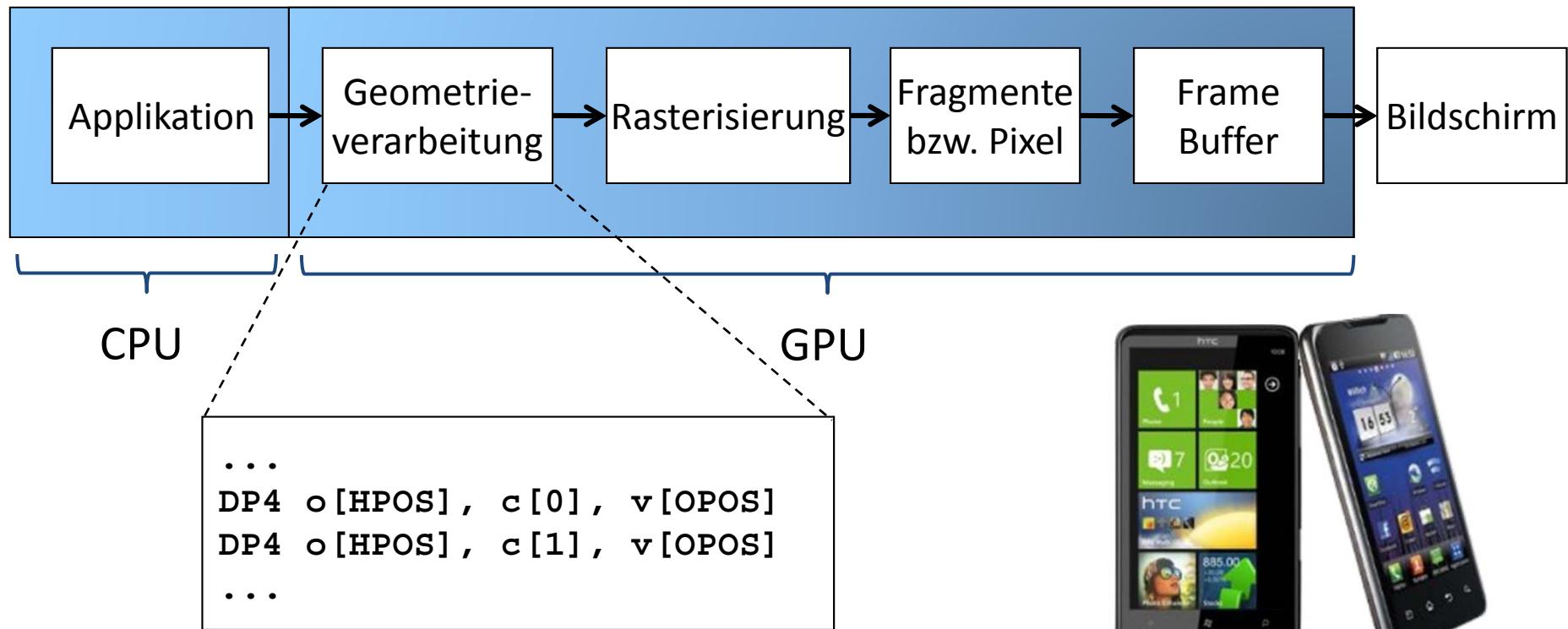
- ▶ SGI Onyx (1993), nVidia GeForce2, ...
GPU übernimmt die ganze Pipeline



Grafik Hardware



- ▶ programmierbare GPUs von AMD/ATI, NVIDIA, Intel, Imagination, ...
- ▶ heute: **große Teile der Pipeline frei programmierbar mit speziellen Hochsprachen**
 - ▶ Geometrieverarbeitung, Primitive Assembly, Fragmentverarbeitung, ...
 - ▶ **Programmierung der GPU „ohne“ die Grafik-Pipeline (OpenCL, CUDA)**



OpenGL – Vorgeschmack



- OpenGL stammt aus der Zeit vor C++

- OpenGL-Funktionen beginnen mit dem Präfix „gl“

`glFunction{1234}{bsifd...}{v}(T arg1, T arg2, ...);`

Bsp.: `glVertex3f(1.0f, 2.0f, 3.0f);`

`glDrawArrays(GL_TRIANGLES, 0, vertexCount);`

- OpenGL-Konstanten beginnen mit „GL_“

`GL_IRGENDEINE_KONSTANTE`

Bsp.: `GL_TRIANGLES`

- OpenGL-Typen beginnen mit „GL“

`GLtype`

Bsp.: `GLfloat`

OpenGL – Vorgeschmack

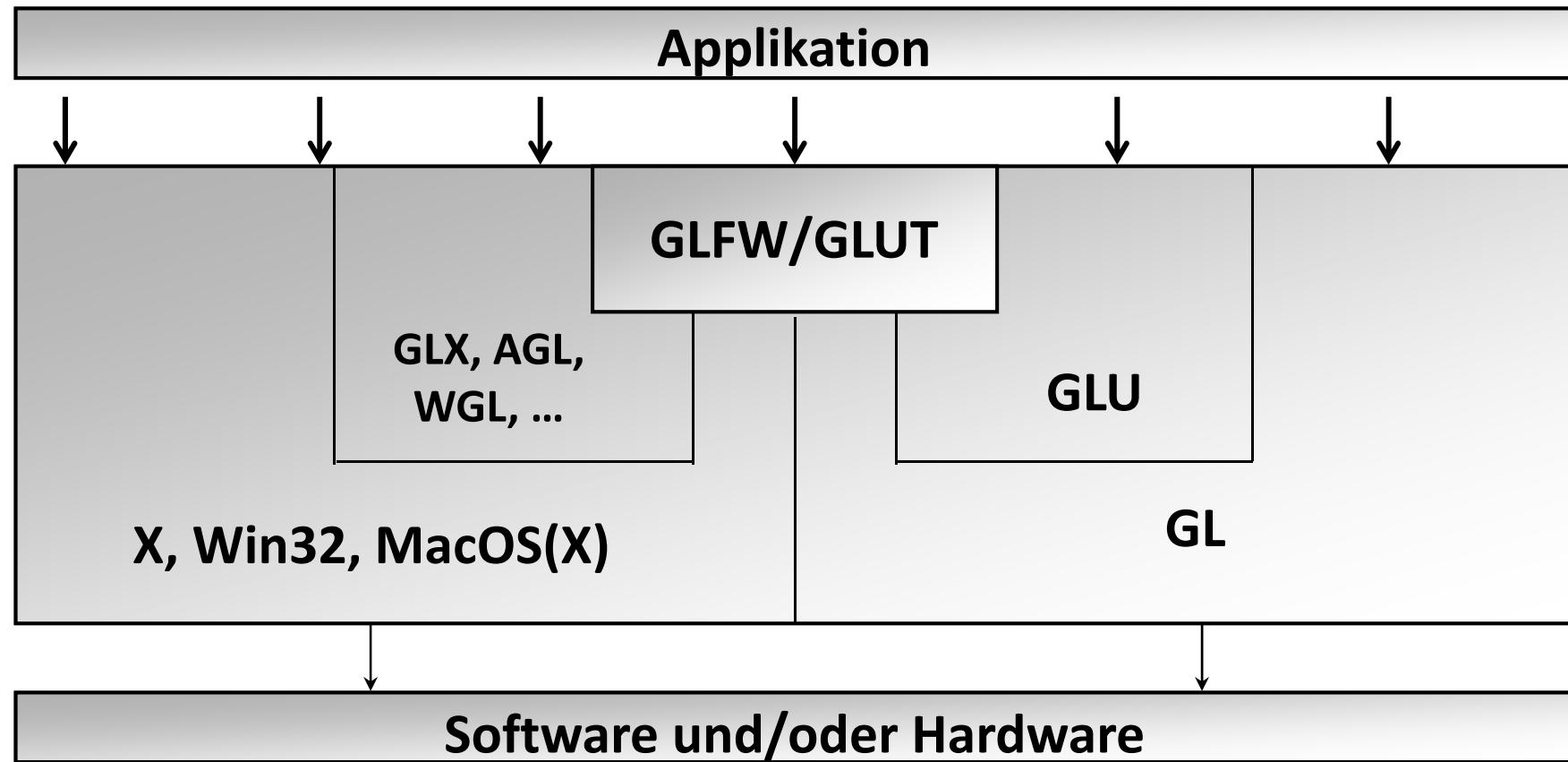


- ▶ OpenGL ist eine Zustandsmaschine
 - ▶ ein Zustand bleibt erhalten, bis er explizit geändert wird
- ▶ setzen von OpenGL-Zuständen, z.B. **GL_LIGHTING**
`glEnable(...);`
`glDisable(...);`
`gl*(...); // einige spezielle States`
- ▶ Auslesen von OpenGL-Zuständen
`glGet*(...); // spezielle Aufrufe`

OpenGL Bibliotheken

- ▶ 200+ Funktionen im klassischen OpenGL
z.B. `glClear(GL_COLOR_BUFFER_BIT)`
- ▶ GLU OpenGL Utility Library
z.B. `gluLookAt`, `gluSphere`, `gluNurbs`
- ▶ GLX Interface Lib zum X-Window-System
z.B. `glXChooseVisual`, `glXSwapBuffers`
- ▶ WGL Microsoft Windows – AGL MacOS
z.B. `wglCreateContext`
- ▶ GLUT (OpenGL Utility Toolkit), GLFW, Qt, SDL, ...
 - ▶ Plattform-neutrale Schnittstellen zum Fenstersystem
 - ▶ Handhabung von Tastatur-/Maus-/Joystick-Eingabe
 - ▶ keine offiziellen OpenGL-Bestandteile
- ▶ weitere Bibliotheken, wie z.B. OpenGL Mathematics Bibliothek („glm“)

APIs um OpenGL



OpenGL – in dieser Vorlesung

- ▶ zunächst nur das Allerwichtigste über klassisches OpenGL
 - ▶ OpenGL-Kommandos für Geometrie, Beleuchtung, ...
 - ▶ einige OpenGL Konzepte behandeln wir erst im 2. Teil
 - ▶ Literatur: **The OpenGL Programming Guide - The Redbook**
HTML Version und Beispielprogramme:
<http://www.openglprogramming.com/red/>
- ▶ dann: modernes OpenGL (3+) und „Core Profiles“, sowie OpenGL ES
 - ▶ programmierbare Geometrie- und Fragment-Verarbeitung („Shader“)
<http://www.opengl.org/registry/doc/glspec45.core.pdf>
<http://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf>
 - ▶ in den Übungsaufgaben: OpenGL 3.x
 - ▶ keines der klassischen Konzepte verliert seine Bedeutung
 - ▶ Vorteile: keine Altlasten, viel Kontrolle über die Pipeline, nur minimale Unterschiede zu Direct3D (10+)
 - ▶ Nachteile: „convenience features“ gehen verloren
- ▶ **Wichtig:** OpenGL lernt man am besten durch
Ausprobieren und Experimentieren
mit Beispielprogrammen

Disclaimer



- ▶ die Folien dieses Kapitels enthalten
 - ▶ viel Programm-Code und
 - ▶ noch viel mehr OpenGL Befehle
- ▶ (1) viele der zugrundeliegenden Konzepte (z.B. Transformationen, Matrix-Stack etc.) kennen wir schon, hier dienen die Folien als Verweis auf die entsprechenden OpenGL Kommandos
- ▶ (2) trotzdem werden wir einige Code-Beispiele besprechen, die die OpenGL-Programmierung und –Konzepte verdeutlichen
- ▶ (3) und wir lernen neue Konzepte kennen, wie z.B. Blending, Stencil-Buffering, Accumulation-Buffers, ...
- ▶ OpenGL lernen Sie **nur** durch **Ausprobieren und Experimentieren** mit Beispielprogrammen (kommen Sie trotzdem in die Vorlesung...)



Klassisches OpenGL

GLUT: Grundlagen



Struktur einer GLUT-Anwendung

- ▶ initialisiere, konfiguriere und öffne Fenster mit Render-Kontext
- ▶ initialisiere OpenGL-States (es gibt einen Default-Zustand)
- ▶ registriere Callback-Funktionen
 - ▶ Zeichnen: `glutDisplayFunc (display)`
 - ▶ Fenstergröße ändern: `glutReshapeFunc (resize)`
 - ▶ Animation/Simulation: `glutIdleFunc (idle)`
 - ▶ Benutzereingaben: `glutKeyboardFunc(keyboard)`
 - ▶ Mausereignisse: `glutMouseFunc (mouse)`
- ▶ warte in Ereignisschleife (event processing loop)
- ▶ Header-Dateien `#include <GL/gl.h>`
 `#include <GL/glu.h>`
 `#include <GL/glut.h>`
- ▶ weiterer sehr ähnlicher Wrapper: GLFW, andere Konzepte: Qt, SDL, ...

GLUT: Beispiel



```
void main( int argc, char** argv ) {  
  
    // initialisiere/öffne Fenster mit Render-Kontext  
    int mode = GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH;  
    glutInit( &argc, argv );  
    glutInitDisplayMode( mode );  
    glutInitWindowSize ( 1024, 768 );  
    glutCreateWindow( "Mein GLUT Programm" );  
  
    // Initialisiere OpenGL States, konfiguriere Pipeline  
    init();  
  
    // Callback Funktionen registrieren  
    glutDisplayFunc ( display );  
    glutReshapeFunc ( resize );  
    glutKeyboardFunc( keyboard );  
    glutMouseFunc     ( mouse );  
    glutIdleFunc     ( idle );  
    glutMainLoop();  
}
```

Double-Buffering



Bestandteile eines Framebuffers in OpenGL

- ▶ mehr als nur der Speicherbereich in dem die Pixeldaten gehalten werden
- ▶ i.d.R. versteht man unter dem Framebuffer mehrere Komponenten:
 - ▶ Color Buffer enthält die Pixeldaten, z.B. RGBA-Werte (**GLUT_RGBA**)
 - ▶ Z-Buffer für die Tiefenwerte (**GLUT_DEPTH**)
 - ▶ weitere Buffer, z.B. Stencil, Accumulation (später mehr)
- ▶ **Double Buffering (GLUT_DOUBLE)** bedeutet es gibt 2 Color Buffer
 - ▶ Front Buffer, der das gerade dargestellte Bild enthält und
 - ▶ Back Buffer, unsichtbarer Puffer in dem das nächste Bild erzeugt wird
- ▶ das Umschalten („Swapping“) erfolgt, wenn das Rendering abgeschlossen ist mit **glutSwapBuffers();**
- ▶ es genügt ein z-Buffer (für das dargestellte Bild benötigt man keinen)

GLUT: Beispiel *cont.*

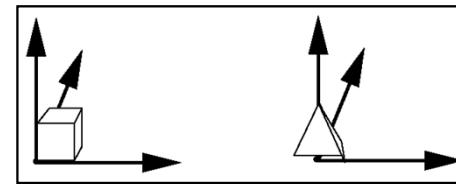
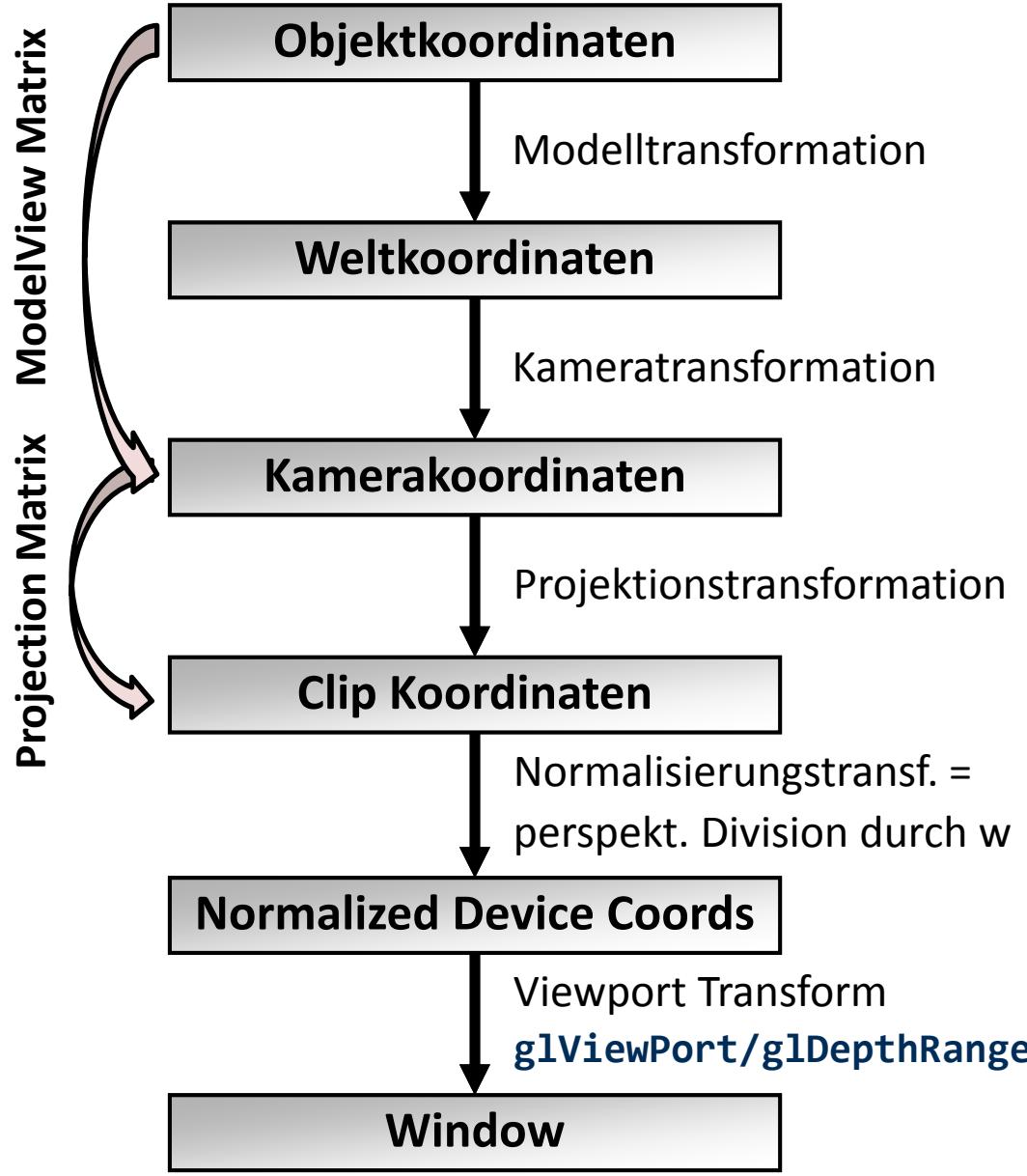


```
// Initialisiere OpenGL States
void init( void ) {
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
    glClearDepth( 1.0f );    // min. Tiefe 0.0, max. Tiefe 1.0

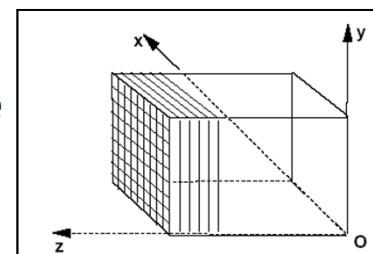
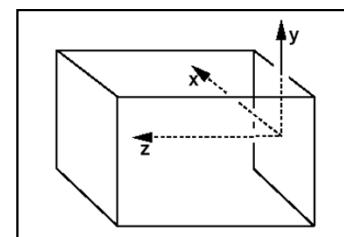
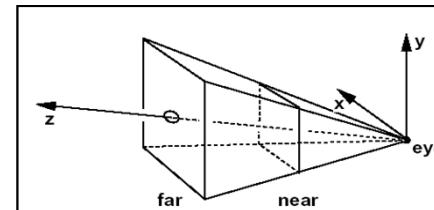
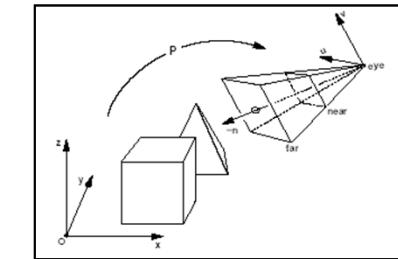
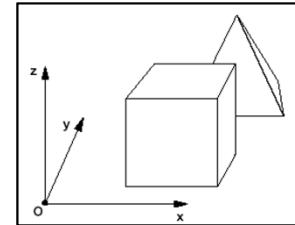
    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
    glEnable( GL_DEPTH_TEST );
}

// Rendering (=Zeichnen) eines Bildes
void display( void ) {
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glBegin( GL_TRIANGLES );
        glVertex3f( 0.0f, 0.0f, 0.0f );
        glVertex3f( 1.0f, 0.0f, 0.0f );
        glVertex3f( 0.0f, 1.0f, 0.0f );
    glEnd();
    glutSwapBuffers();
}
```

Koordinatensysteme



**glVertex()
glNormal()**



Mehr Infos:
[http://www.opengl.org/
resources/faq/technical/
viewing.htm](http://www.opengl.org/resources/faq/technical/viewing.htm)

[http://www.songho.ca/
opengl/gl_transform.html](http://www.songho.ca/opengl/gl_transform.html)

GLUT: Beispiel *cont.*



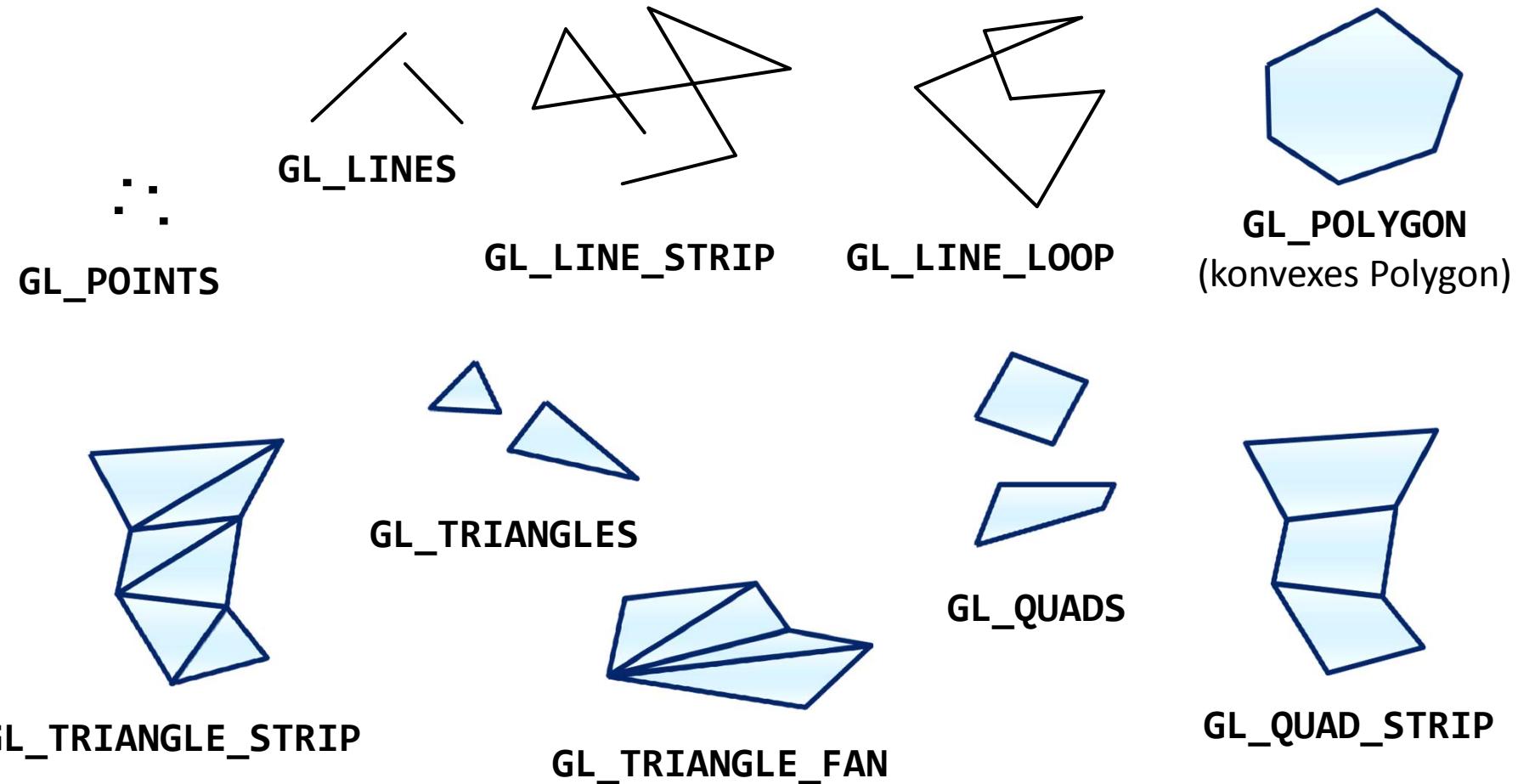
```
// Aufruf bei Änderung der Fenstergröße
// Aktualisierung der Kamera- und Projektionsmatrizen
void resize( int w, int h ) {
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0,           // Position
               0.0, 0.0, 0.0,           // Ziel
               0.0, 1.0, 0.0 );        // Up-Vektor
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat)w / h, 1.0, 100.0 );
    glViewport( 0, 0, (GLsizei)w, (GLsizei)h );
}

// Behandlung von Tastatureingaben
// key = Taste, (x, y) = Koordinaten des Mauszeigers
// Konstanten für Spezial-Tasten, z.B. GLUT_KEY_UP
void keyboard( char key, int x, int y ) {
    switch( key ) {
        case 'q': case 'Q': exit( EXIT_SUCCESS ); break;
        case 'r': case 'R': rotate = GL_TRUE; break;
    }
}
```

OpenGL: Geometrische Primitive



- ▶ Alle geometrischen Primitive werden durch **Vertizes in homogenen Koordinaten** definiert
 - ▶ wird die homogene Koordinate nicht angegeben, gilt $w = 1$



glVertex3fv(vtx)

Anzahl der
Komponenten

2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)

Datentyp

b - byte
ub - unsigned byte
s - short
us - unsigned short
i - int
ui - unsigned int
f - float
d - double

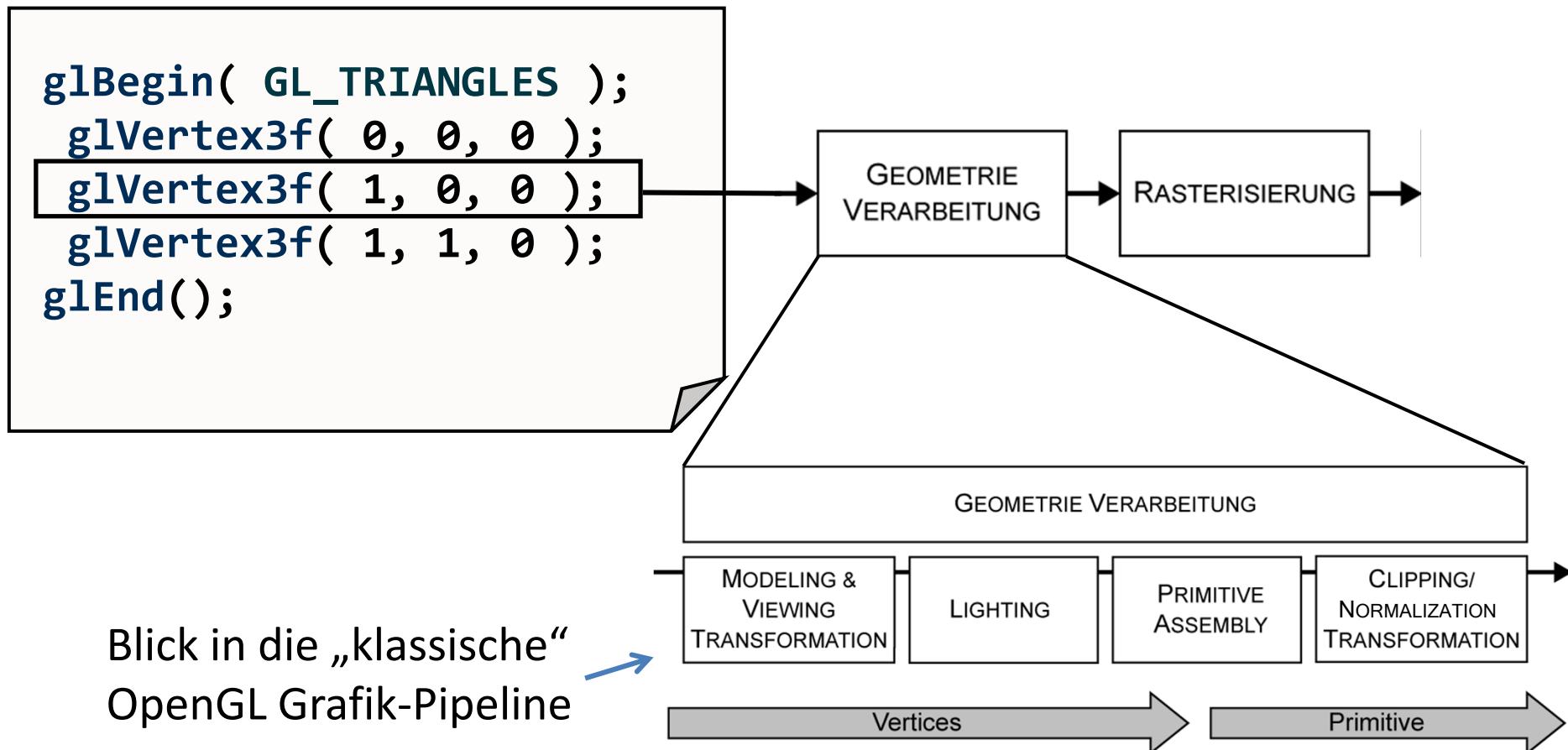
Vektor

“v” für die Skalarform
weglassen
glVertex2f(x, y)

- Schreibweise auf den Folien manchmal:
glVertex*(...) oder einfach nur **glVertex(...)**

Vertizes in der Grafik-Pipeline

- ▶ Primitive werden aus dem Strom von Vertices zusammengesetzt
- ▶ wie zusammengesetzt wird, wird mit **glBegin** angegeben
- ▶ im Anschluss an das Zusammensetzen (Primitive Assembly) findet
 - ohne eigenes Zutun - Clipping und Rasterisierung statt

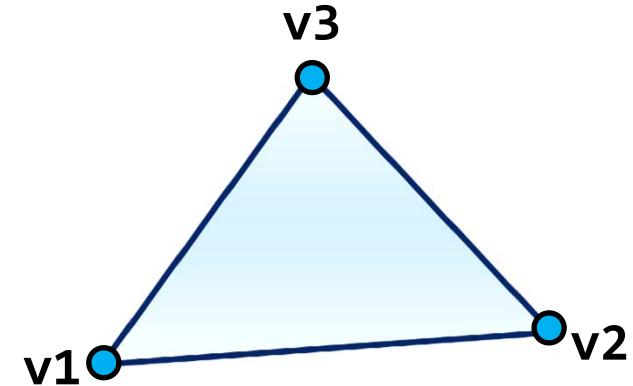


Geometrische Primitive

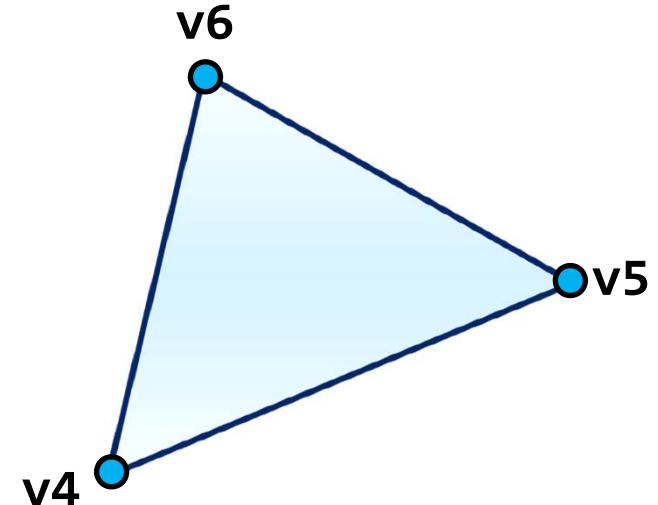
Isolierte/unabhängige Dreiecke

```
glBegin( GL_TRIANGLES );  
    glVertex3fv( v1 );  
    glVertex3fv( v2 );  
    glVertex3fv( v3 );  
    glVertex3fv( v4 );  
    glVertex3fv( v5 );  
    glVertex3fv( v6 );  
glEnd();
```

The code defines two triangles. The first triangle is formed by vertices v1, v2, and v3. The second triangle is formed by vertices v4, v5, and v6. A brace on the right side groups v1-v3 as 'Dreieck 1' and v4-v6 as 'Dreieck 2'.



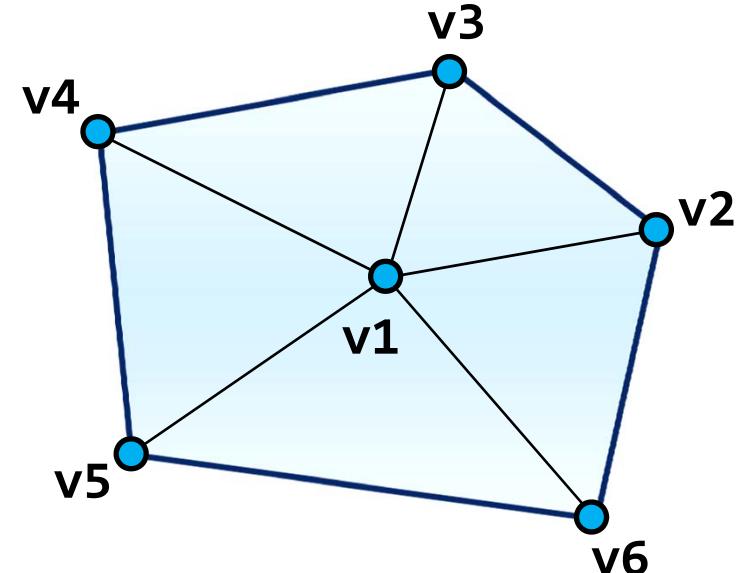
- Verhältnis Vertizes-zu-Dreiecken:
 - n Dreiecke
 - 3 Vertizes pro Dreieck ($3n$ insgesamt) in der Geometriestufe verarbeitet



Geometrische Primitive

Dreiecksfächer, Triangle Fans

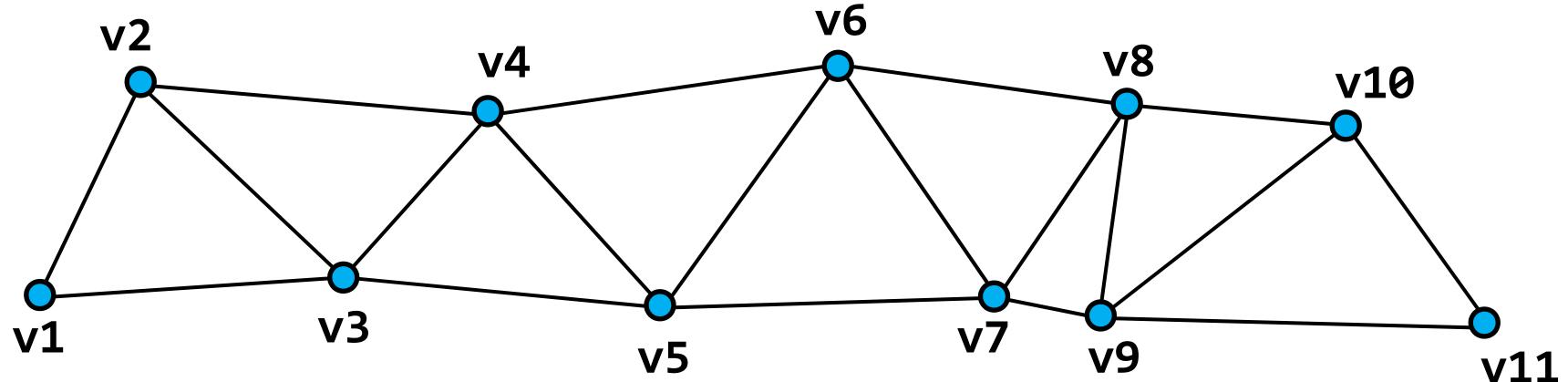
```
glBegin( GL_TRIANGLE_FAN );  
    glVertex3fv( v1 );  
    glVertex3fv( v2 );  
    glVertex3fv( v3 );  
    glVertex3fv( v4 );  
    glVertex3fv( v5 );  
    glVertex3fv( v6 );  
    glVertex3fv( v2 );  
glEnd();
```



- ▶ $n + 2$ Vertizes werden verarbeitet, um n Dreiecke zu zeichnen
- ▶ nur ca. ein Drittel der Arbeit für die Geometriestufe (für große n) im Vergleich zu **GL_TRIANGLES**
- ▶ allerdings ist n bei Dreiecksfächern typischerweise nicht sehr groß

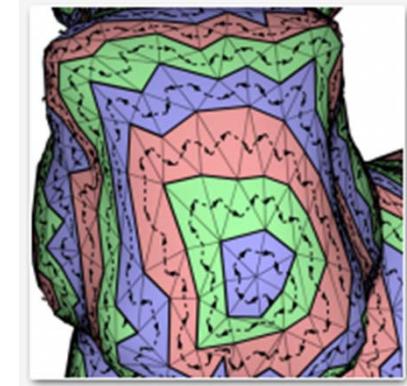
Geometrische Primitive

Dreiecksstreifen, Triangle Strips



```
glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v1 );
    glVertex3fv( v2 );
    glVertex3fv( v3 );
    glVertex3fv( v4 );
    glVertex3fv( v5 );
    ...
};
```

The diagram illustrates the vertex order for a GL_TRIANGLE_STRIP. It shows five vertices labeled v1 through v5. The first four vertices (v1, v2, v3, v4) are grouped by a brace and labeled 'Dreieck 1', representing the first triangle. The last two vertices (v5) are grouped by another brace and labeled 'Dreieck 2', representing the second triangle.



- $n + 2$ Vertizes werden verarbeitet, um n Dreiecke zu zeichnen
 - Dreiecksstreifen sind wichtig: es gibt Algorithmen, die Dreiecksnetze in möglichst wenige Streifen zerlegen

Primitive und Vertex-Attribute

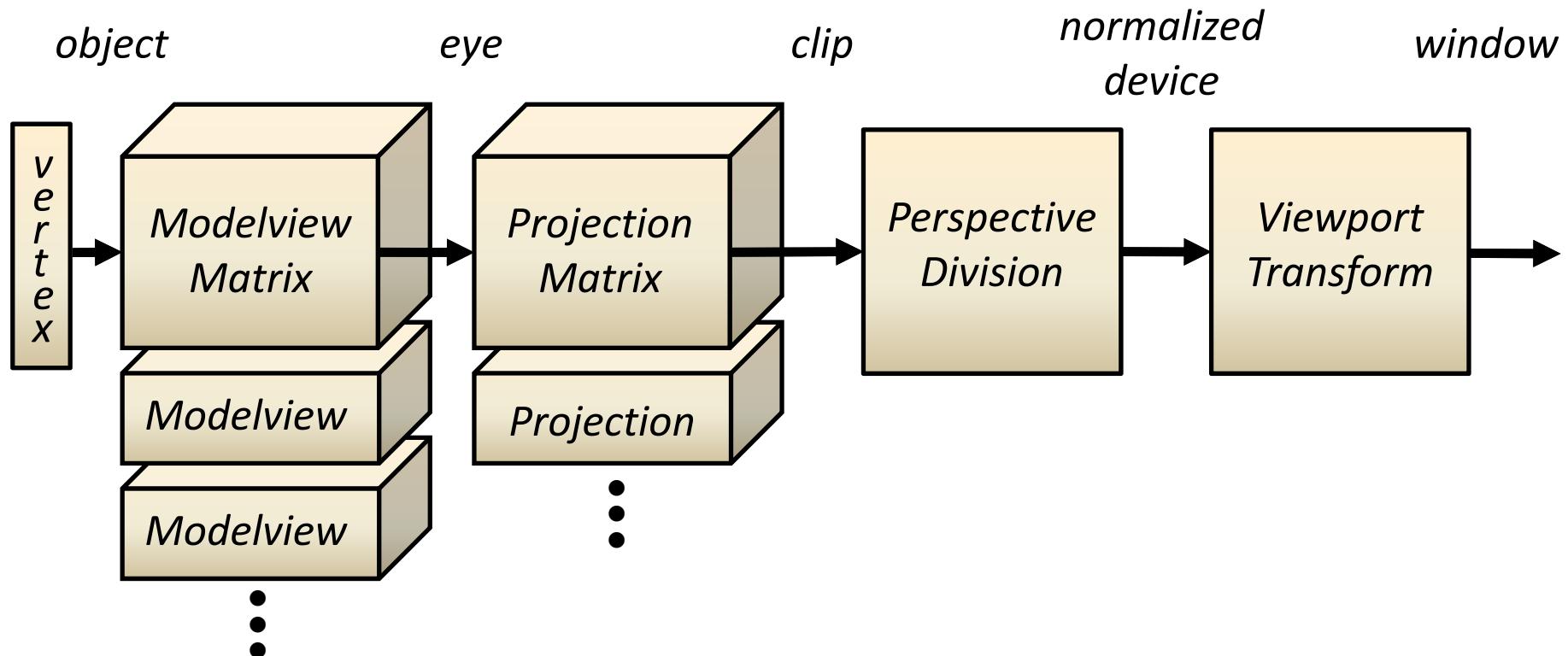


- ▶ Vertizes werden verbunden wie angegeben mit
`glBegin(primType); ...; glEnd();`
- ▶ Zeichnen von Dreiecken mit zusätzlichen Attributen
 - ▶ Vertex-Attribute in OpenGL:
`glColor*(); glNormal*(); glTexCoord*();`
 - ▶ es gelten jeweils die Attribute die vor `glVertex` gesetzt wurden 
 - ▶ globale Zustände müssen vor `glBegin` gesetzt werden
`glPointSize(size); glShadeModel(GL_SMOOTH);
 glEnable(GL_LIGHTING);`

```
GLfloat red, greed, blue;
GLfloat coords[3];
glBegin( primType );
for ( int i = 0; i < nVerts; ++i ) {
    glColor3f( red, green, blue );
    glVertex3fv( coords );
}
glEnd();
```

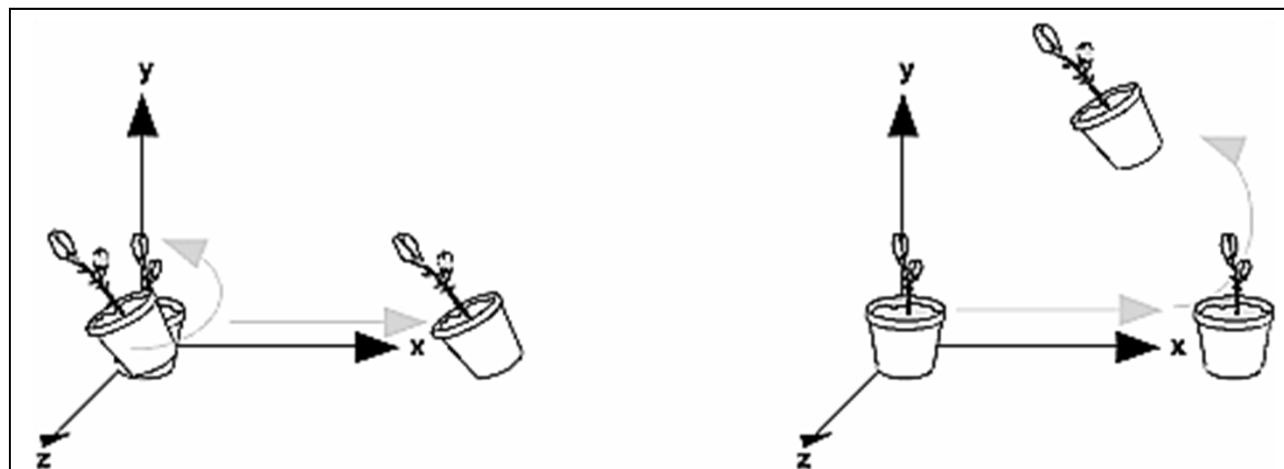
Transformationen: Pipeline

- ▶ es gibt Matrix Stacks für Modelview, Projektions- und Textur-Matrix
 - ▶ letztere dient zur automatischen Erzeugung von Texturkoordinaten, wofür heute nur noch selten klassische OpenGL-Funktionalität eingesetzt wird



Transformationsmatrizen

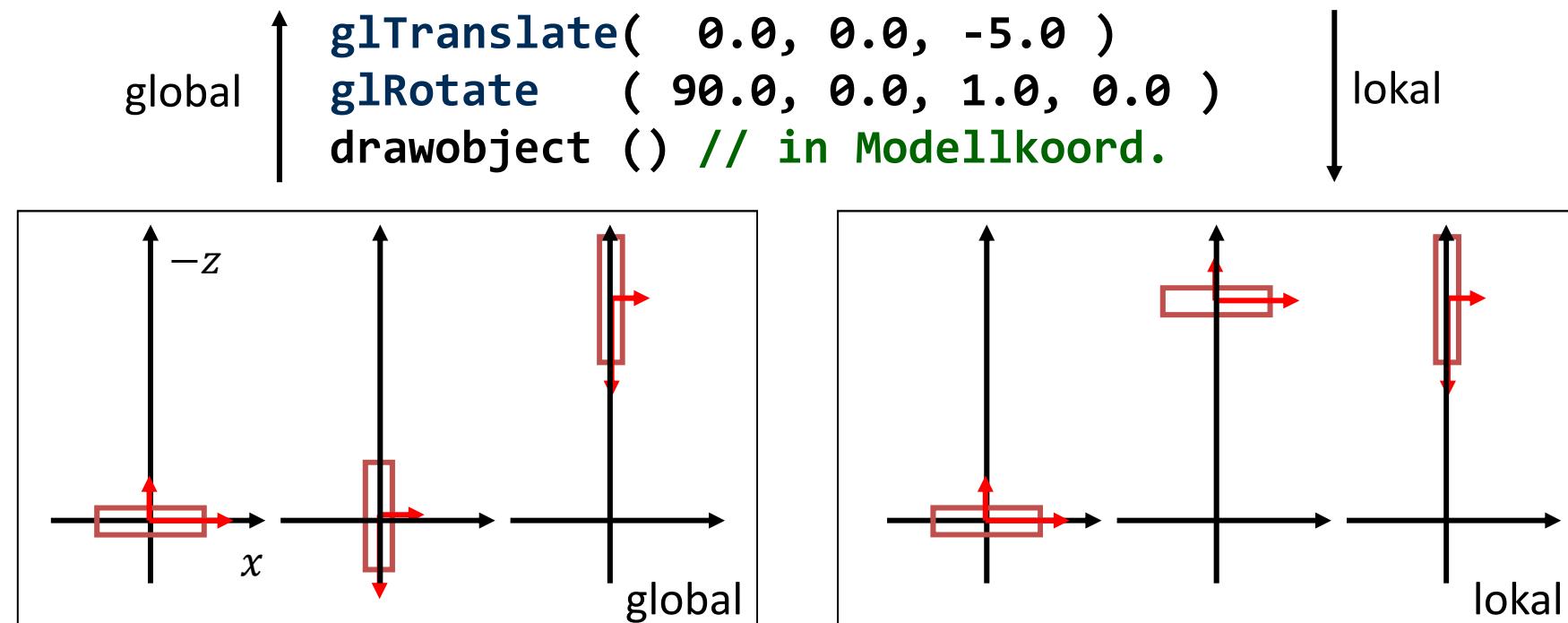
- ▶ wähle Matrix-Stack
glMatrixMode(GL_MODELVIEW oder GL_PROJECTION)
- ▶ Lade oder multipliziere Matrix
glLoadMatrix(), glMultMatrix{f|d}(const GLtype *m)
- ▶ Befehle zur Stack-Verwaltung
glLoadIdentity(), glPushMatrix(), glPopMatrix()
- ▶ spezielle Modelview-Transformationen
glRotate(), glTranslate(), glScale()



Zusammengesetzte Transformationen

- ▶ globale Sicht („wie transformieren wir das Objekt“)
 - ▶ platziere/bewege Objekte relativ zum Ursprung der Weltkoordinaten
 - ▶ Reihenfolge im Code rückwärts

- ▶ lokale Sicht („wie sieht das Ko-Sys aus, in dem das Objekt liegt“)
 - ▶ bewege lokale Koordinatensysteme der Objekte
 - ▶ Reihenfolge im Code vorwärts



Hierarchische Modellierung



Beispiel

- verwendet Push/Pop auf ModelView-Matrix-Stack

```
draw_body_wheel_and_bolts()
{
    draw_car_body();
    glPushMatrix();
    glTranslatef( 40, 0, 30 );
    // move to 1st wheel position
    draw_wheel_and_bolts();
    glPopMatrix();
    glPushMatrix();
    glTranslatef( 40, 0, -30 );
    // move to 2nd wheel position
    draw_wheel_and_bolts();
    glPopMatrix();
    ...
    // draw last two wheels
}
```

```
draw_wheel_and_bolts()
{
    draw_wheel();
    for( int i = 0; i < 5; i++ ) {
        glPushMatrix();
        glRotatef( 72.0 * i,
                  0.0, 0.0, 1.0);

        glTranslatef(3.0, 0.0, 0.0);
        draw_bolt();
        glPopMatrix();
    }
}
```

Transformationen: Projektion

- ▶ spezielle Funktionen für Projektionen

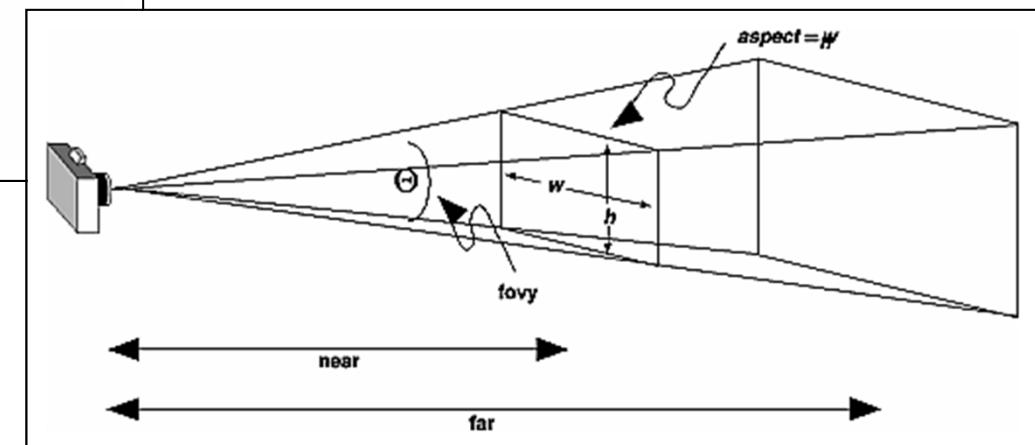
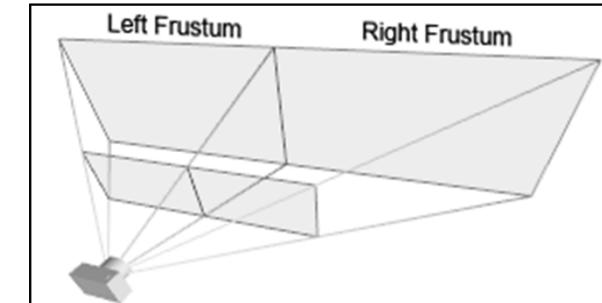
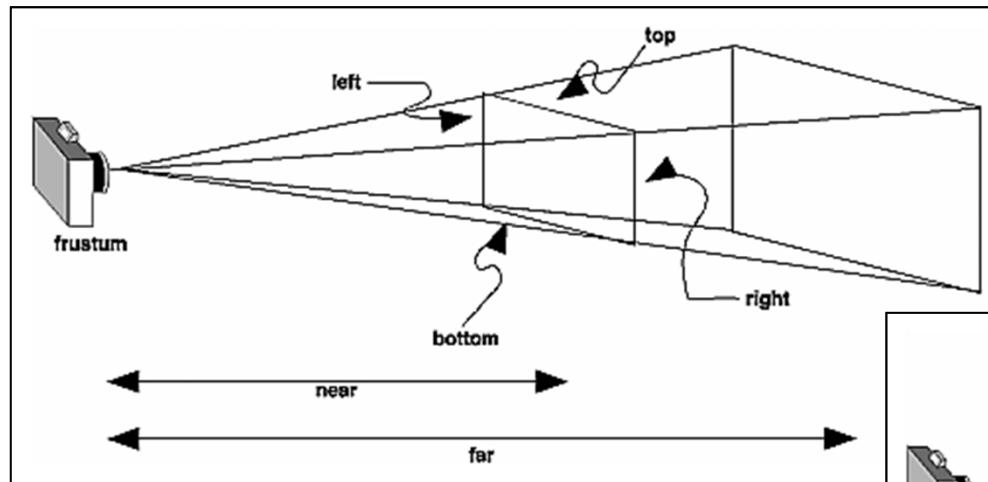
`gluPerspective(fovy, aspect, zNear, zFar)`

`glFrustum(left, right, bottom, top, zNear, zFar)`

`glOrtho (left, right, bottom, top, zNear, zFar)`

- ▶ verwendet mit `glMatrixMode(GL_PROJECTION)`

- ▶ `glFrustum` erlaubt nicht-symmetrische Sichtpyramiden

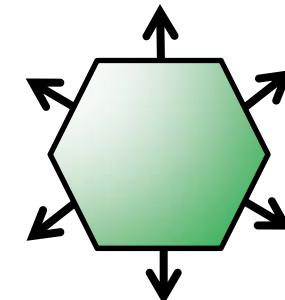


Dreiecke



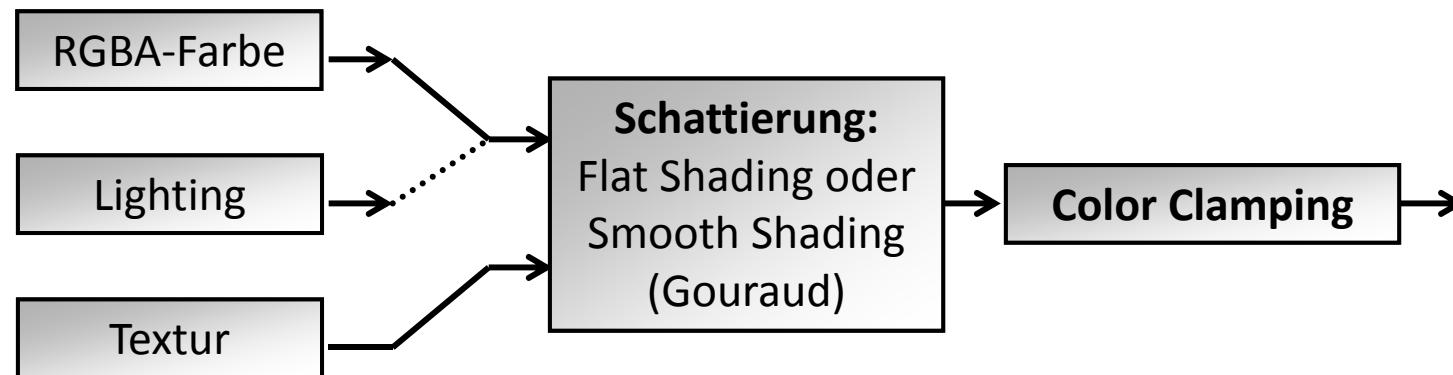
Rasterisierung, Vorder- und Rückseiten, Backface Culling

- ▶ bei Dreiecken und Polygonen wird Vorder- und Rückseite unterschieden
 - ▶ übliche Festlegung: man sieht ein Polygon von vorne, wenn die Vertizes am Bildschirm entgegen den Uhrzeigersinn laufen
 - ▶ man kann aber explizit festlegen, was als Vorderseite angesehen wird:
`glFrontFace(GL_CCW oder GL_CW);`
- ▶ **Backface Culling:** Flächen, auf deren Rückseite man blickt, kann man verwerfen lassen
`glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);`
- ▶ bei geschlossenen Dreiecksnetze sind die Rückseiten nicht sichtbar



Beleuchtung und Schattierung

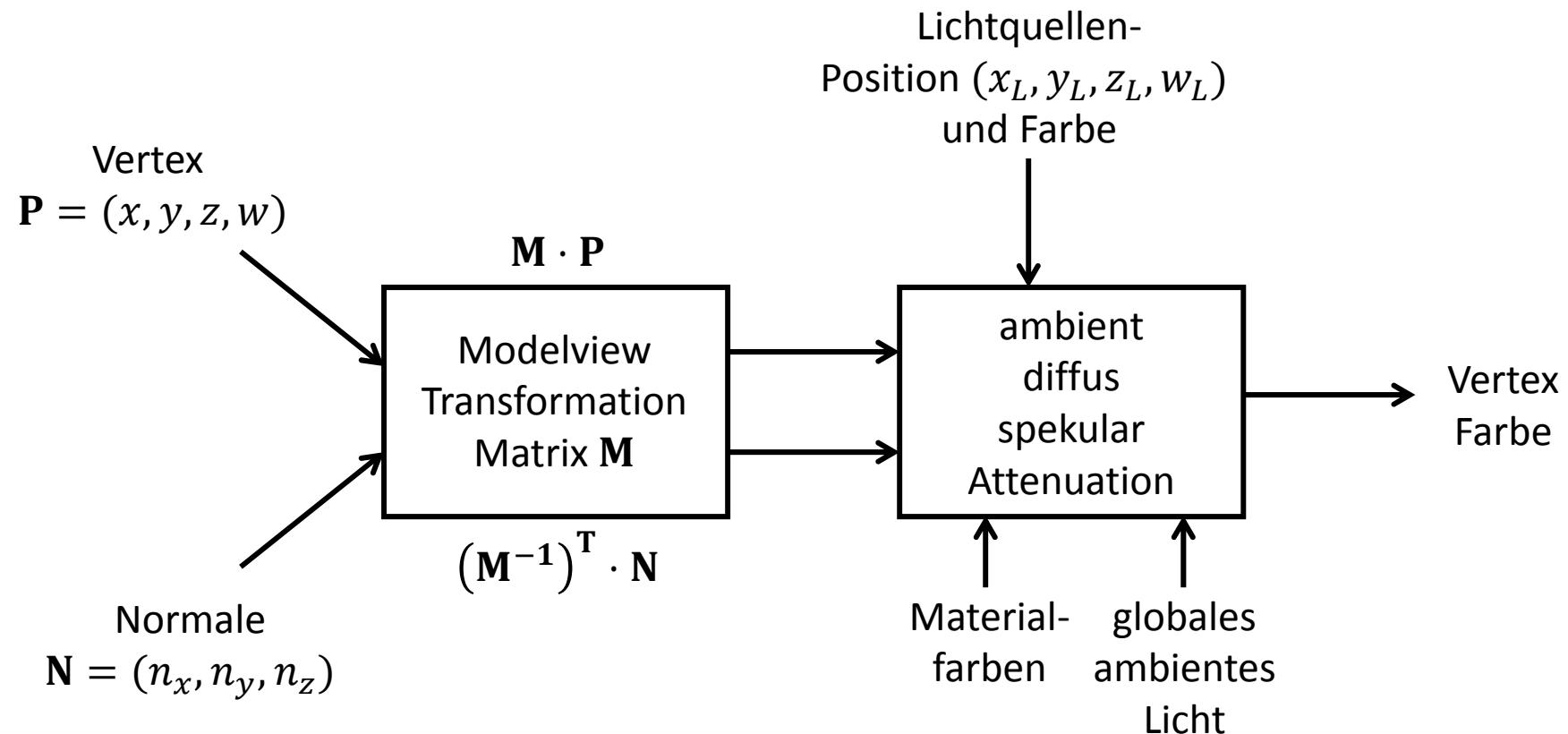
- ▶ zwei Arten die Farbe eines Vertex anzugeben
 - ▶ direkt als Vertex-Attribut: `glColor*`()
 - ▶ Definition eines Materials und OpenGL Beleuchtungsberechnung
 - ▶ (historisch: beide Optionen gekoppelt mit `glColorMaterial`)
- ▶ Einschalten der Beleuchtungsberechnung mit
 `glEnable(GL_LIGHTING)`



- ▶ Schattierung ohne Beleuchtung: nur Interpolation der Vertex-Farben (sogenanntes Gouraud Shading)

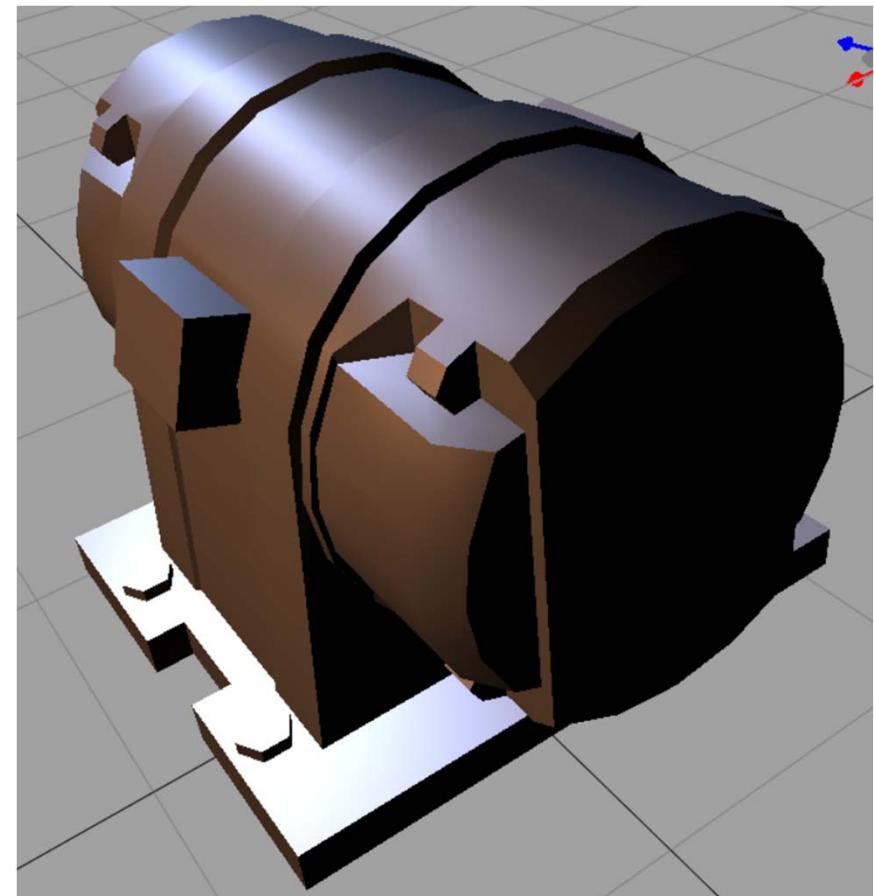
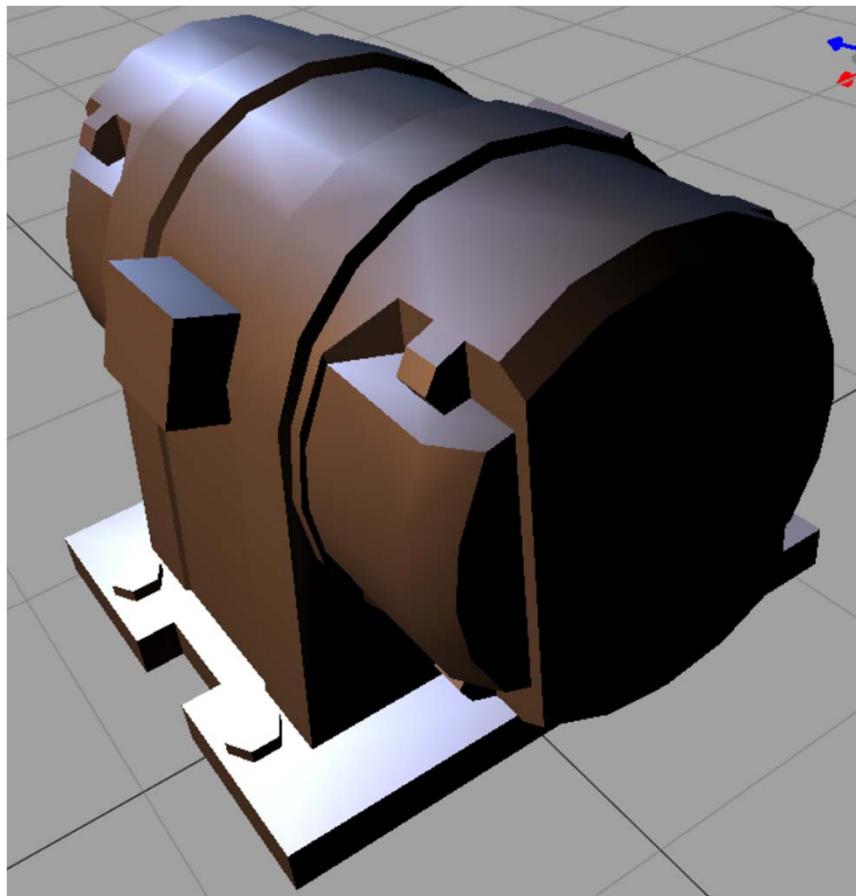
Beleuchtungsberechnung

- ▶ Beleuchtungsberechnung in **Kamerakoordinaten** nach Modelview-Transf.
- ▶ entweder **pro Vertex (Smooth Shading)** oder **pro Dreieck (Flat Shading)**
 - ▶ OpenGL unterstützt nativ keine Beleuchtungsberechnung pro Pixel, also kein Phong Shading, nur Gouraud Shading



Gouraud vs. Phong Shading

- ▶ Gouraud Shading (links, Machsche Effekte, verpasste Glanzlichter): Berechnung der Beleuchtung pro Vertex, Interpolation der Farbe
- ▶ Phong Shading (rechts): Interpolation der Normale, Berechnung der Beleuchtung pro Pixel/Fragment



Blinn-Phong Modell

Abwandlung des Phong Modells (betrifft nur den spekularen Teil)

- ▶ original Phong-Modell: $I_s = k_s \cdot I_L \cdot \cos^n \alpha = k_s \cdot I_L \cdot (\mathbf{R}_L \cdot \mathbf{V})^n$
 - ▶ spekularer Reflexionskoeffizient k_s und Phong-Exponent n
- ▶ Blinn-Phong verwendet keinen Reflexionsvektor \mathbf{R}_L , sondern den Halfway-Vektor $\mathbf{H} = \frac{\mathbf{V} + \mathbf{L}}{\|\mathbf{V} + \mathbf{L}\|}$ und einen Exponent n'
 - ▶ spekularer Anteil $I_s = k_s \cdot I_L \cdot \cos^n \alpha' = k_s \cdot I_L \cdot (\mathbf{N} \cdot \mathbf{H})^{n'}$

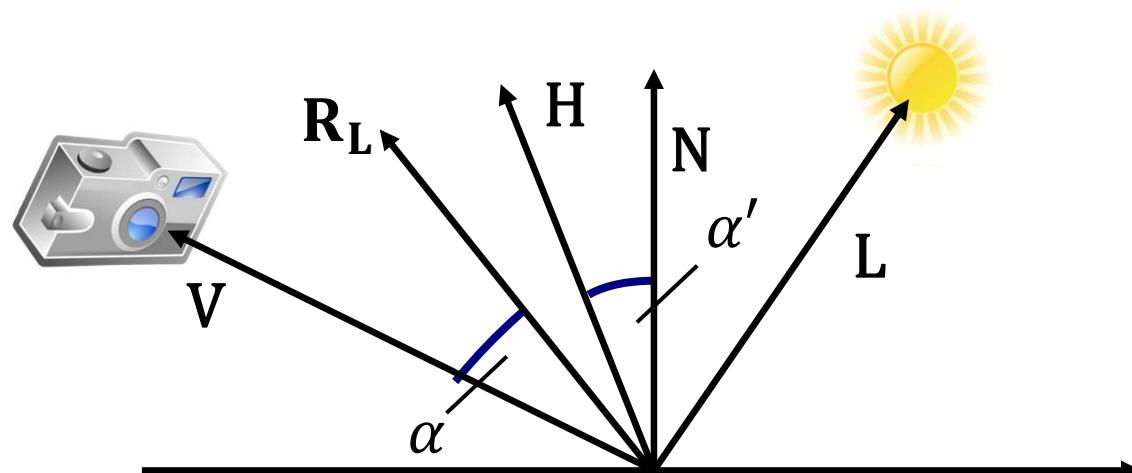


Bild: Andrea Weidlich

Blinn-Phong-Beleuchtung in OpenGL (erweitert)



- ▶ Beleuchtungsberechnung erfolgt pro Vertex
- ▶ Vertex-Farbe setzt sich zusammen aus:
 - ▶ Material-Emission an diesem Vertex +
 - ▶ globales ambientes Licht mult. mit der Materialeigenschaft am Vertex +
 - ▶ abgeschwächte ambiente, diffuse und spekulare Beiträge aller Lichtquellen
- ▶ nach der Beleuchtungsberechnung werden alle Werte auf [0; 1] begrenzt

Blinn-Phong-Beleuchtung in OpenGL (erweitert)



- ▶ Intensität I eines Vertex

$$I = O_{E_\lambda} + I_{a_\lambda} \cdot O_{a_\lambda}$$

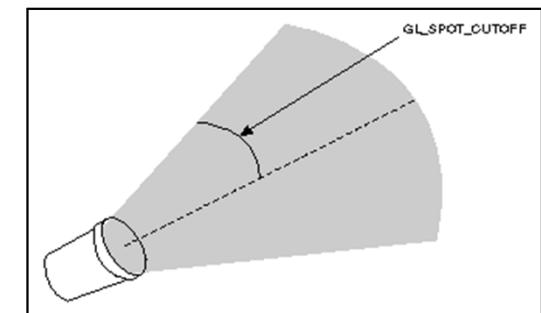
$$+ \sum_i f^i \cdot spot^i \left(I_{a_\lambda}^i \cdot O_{a_\lambda} + \max(\mathbf{N} \cdot \mathbf{L}, 0) \cdot I_{d_\lambda}^i \cdot O_{d_\lambda} + \max(\mathbf{N} \cdot \mathbf{H}, 0)^n \cdot I_{s_\lambda}^i \cdot O_{s_\lambda} \right)$$

- ▶ Emission des Materials O_{E_λ} , globales ambientes Licht I_{a_λ}
- ▶ **ambiente**, **diffuse** und **spekulare** Lichtintensität $I_{a_\lambda}, I_{d_\lambda}, I_{s_\lambda}$ (bis zu 8 Lichtquellen)
- ▶ ambiente, diffuse, spekulare Reflexion des Materials $O_{a_\lambda}, O_{d_\lambda}, O_{s_\lambda}$
- ▶ Abschwächung (Attenuation) für Vertex mit Abstand d

$$f^i = \frac{1}{k_c + k_l d + k_q d^2}$$

- ▶ Spot-Light (Lichtrichtung \mathbf{D} , Vektor zum Vertex \mathbf{V})

$$spot^i = \max(\mathbf{V} \cdot \mathbf{D}, 0)^{spot \ exp}$$



OpenGL-Beleuchtung



- ▶ nur damit Sie es gesehen haben!!!

- ▶ Licht einschalten

```
glEnable( GL_LIGHTING )  
glEnable( GL_LIGHTn )
```

- ▶ Lichtquellen-Eigenschaften

```
glLightf{v}( Light, property, value )
```



| |
|------------------|
| GL_AMBIENT |
| GL_DIFFUSE |
| GL_SPECULAR |
| GL_POSITION |
| GL_SPOT_* |
| GL_*_ATTENUATION |

- ▶ Material-Eigenschaften

```
glMaterialfv( face, property, value )
```



| |
|--------------|
| GL_AMBIENT |
| GL_DIFFUSE |
| GL_SPECULAR |
| GL_SHININESS |
| GL_EMISSION |

- ▶ Lichtquellen-Arten

- ▶ Punktlichtquelle (Point Light): strahlt in alle Richtungen

- ▶ Spotlight: Vorzugsrichtung

- ▶ im Unendlichen (Directional Light; gerichtet):

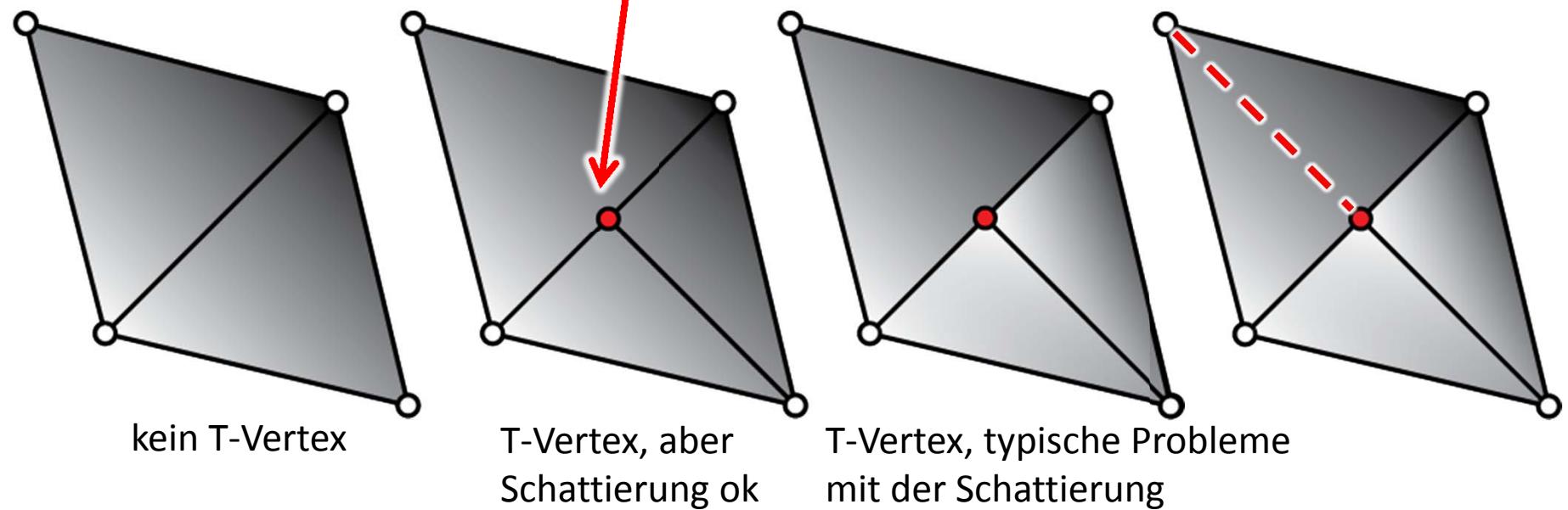
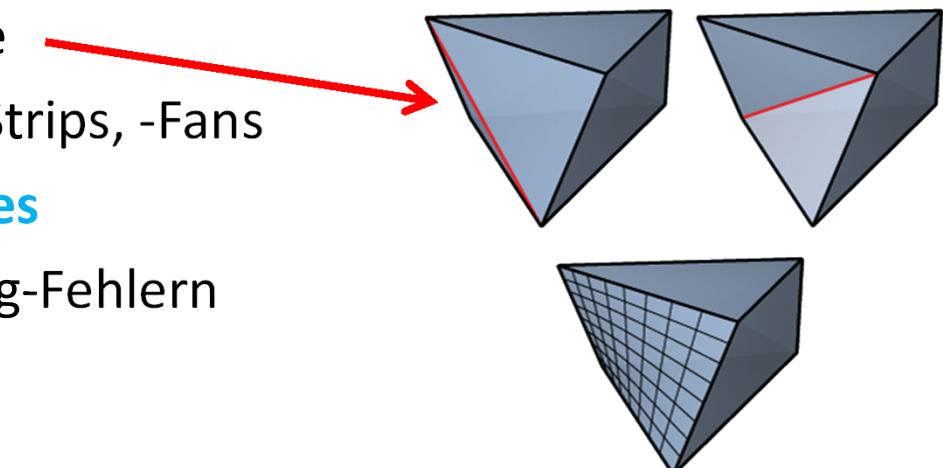
- $w = 0$ in Position (homogene Koordinaten)

- ▶ eventuell beide Seiten von Polygonen beleuchten

```
glEnable( GL_LIGHT_MODEL_TWO_SIDE )
```

Dreiecke und Polygone cont.

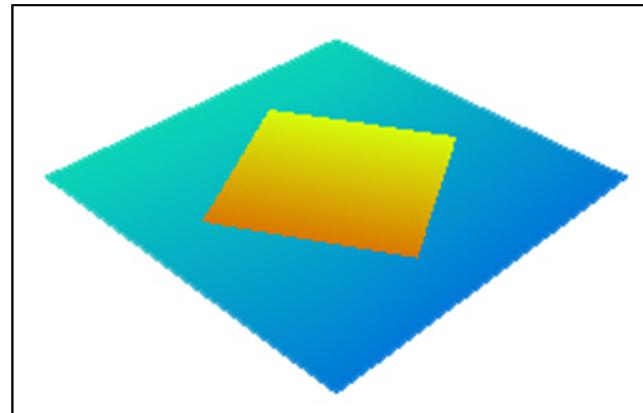
- ▶ Polygone werden vor der Rasterisierung in Dreiecke zerlegt
 - ▶ passiert automatisch bei **GL_POLYGON**, **GL_QUADS**, ...
 - ▶ vermeide nichtebene Polygone
 - ▶ verwende bevorzugt Triangle-Strips, -Fans
 - ▶ vermeide sogenannte **T-Vertizes**
 - ▶ führt zu Löchern und Shading-Fehlern
 - ▶ füge Kante  ein



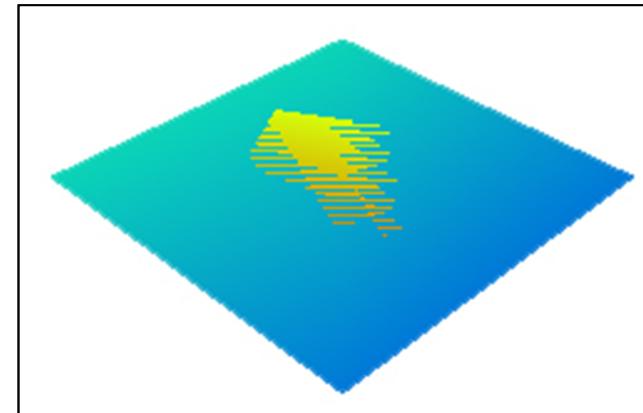
Dreiecke und Polygone *cont.*

Z-Fighting

- ▶ Polygone die sich in derselben Ebene überlagern bereiten Probleme
 - ▶ z.B. bei projizierten Flächen oder Details (surface decals)
 - ▶ Rundungsprobleme bei der Tiefe führen zu **Z-Fighting** (Flackern)
 - ▶ Tiefenwerte werden leicht um einen konstanten Offset **unit** und einen neigungsabhängigen Offset **factor** · Δz verschoben
(positive Wert verursacht Verschiebung nach hinten)
`glPolygonOffset(factor, unit)`
`glEnable(GL_POLYGON_OFFSET_FILL)`



mit Offset (Tiefentest „kleiner-gleich“)

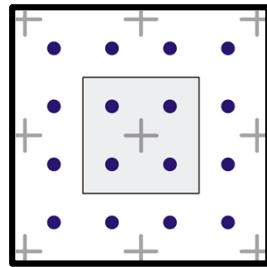


Z-Fighting trotz Tiefentest „kleiner-gleich“

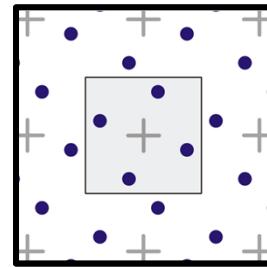
Rasterisierung und Antialiasing



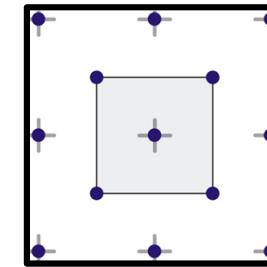
- ▶ im Prinzip verwendet man (starres) Supersampling
 - ▶ erhöhter Speicherbedarf für Framebuffer und Tiefenpuffer
 - ▶ fixes Abtastschema mit im Mittel möglichst wenigen Samples pro Pixel
 - ▶ anschließende Mittelung der „Subpixels“



einfaches Supersampling



Rotated Grid Supersampling



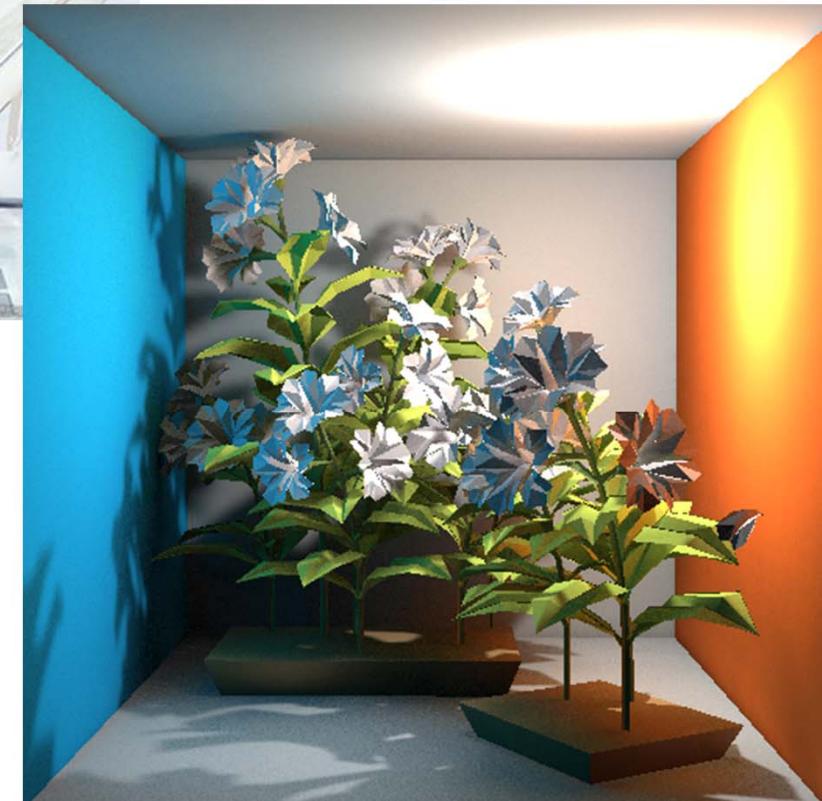
2x Quincunx (NVIDIA)
im Schnitt:

2 Auswertungen pro Pixel

Bilder: [http://de.wikipedia.org/wiki/Antialiasing_\(Computergrafik\)](http://de.wikipedia.org/wiki/Antialiasing_(Computergrafik))

- ▶ Coverage Sample Antialiasing
 - ▶ verwendet einige Samples (z.B. 4) wie bisher
 - ▶ weitere Samples (z.B. 12) nur zur Bestimmung der Sichtbarkeit
 - ▶ Zuordnung schattierte Samples \leftrightarrow Sichtbarkeits-Samples
 - ▶ Idee: Schattierungsberechnung ist teurer als Sichtbarkeit
- ▶ in OpenGL: festgelegt bei der Erzeugung des Framebuffers/Rendertargets (z.B. mit GLUT_MULTISAMPLE)

Auch das geht mit OpenGL...



Modernes OpenGL

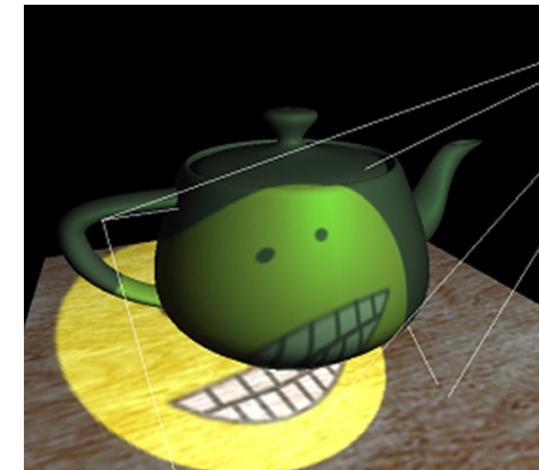
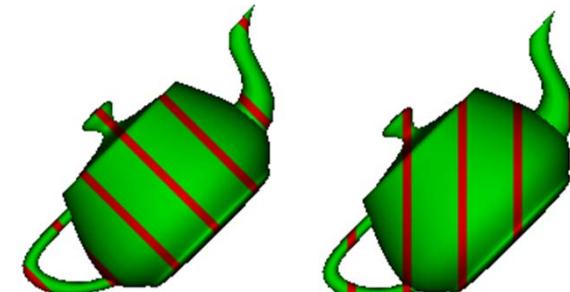
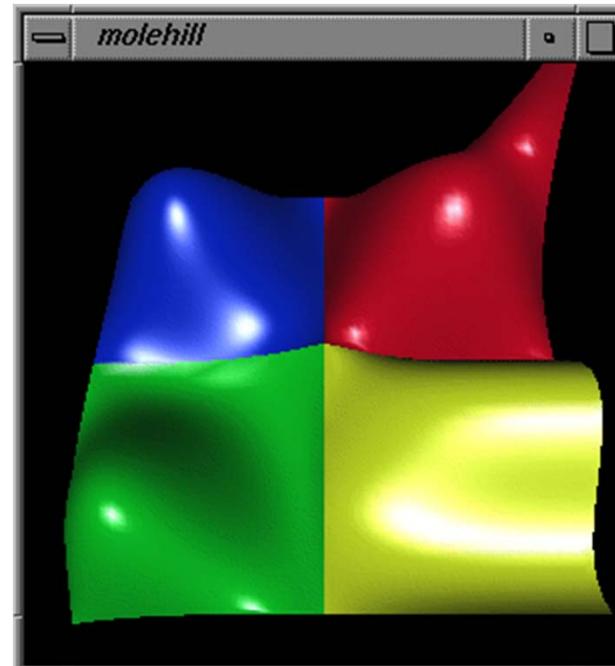
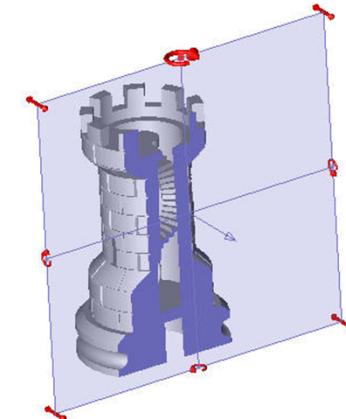


- ▶ die OpenGL-API hat sich mit der Hardware weiterentwickelt
 - ▶ extrem hohe Rechenleistung von GPUs (Teraflop-Bereich, >5 TFlop)
 - ▶ schnelle Anbindung an dedizierten GPU-Speicher (>250 GB/s)
 - ▶ vgl. geringe Busbandbreite zum Hauptspeicher (64 GB/s bei PCIe 4.0)
 - ▶ zu hoher Overhead mit klassischer OpenGL API (**glVertex**-Aufrufe, ...)
- ▶ wesentliche Änderungen mit dem Ziel einer schlanken, effizienten API
 - ▶ kein Immediate Mode (**glBegin**, **glVertex**, **glEnd**) mehr:
Daten werden in Puffern (im Speicher der Grafikhardware) abgelegt
 - ▶ keine Hilfsfunktionen (z.B. Matrix Stacks) mehr
 - ▶ Ziel: keine Fixed-Function Pipeline mehr
 - ▶ Fixed Function = Standardverarbeitung der Vertizes und Fragmente
 - ▶ dafür: frei programmierbar über sogenannte „Shader“
- ▶ Kompatibilitätsprofil: Fixed Function Pipeline existiert „nebenher“
- ▶ Core Profil: entrümpelt, alle alte Funktionalität muss selbst programmiert werden

Jede Menge „entrümpelte“ Features



- ▶ Clip-Ebenen
- ▶ Texturkoordinatengenerierung
- ▶ Polygon Modes
- ▶ Pixel Operations
- ▶ Evaluators
- ▶ ...
- ▶ siehe freiwilliger Appendix!



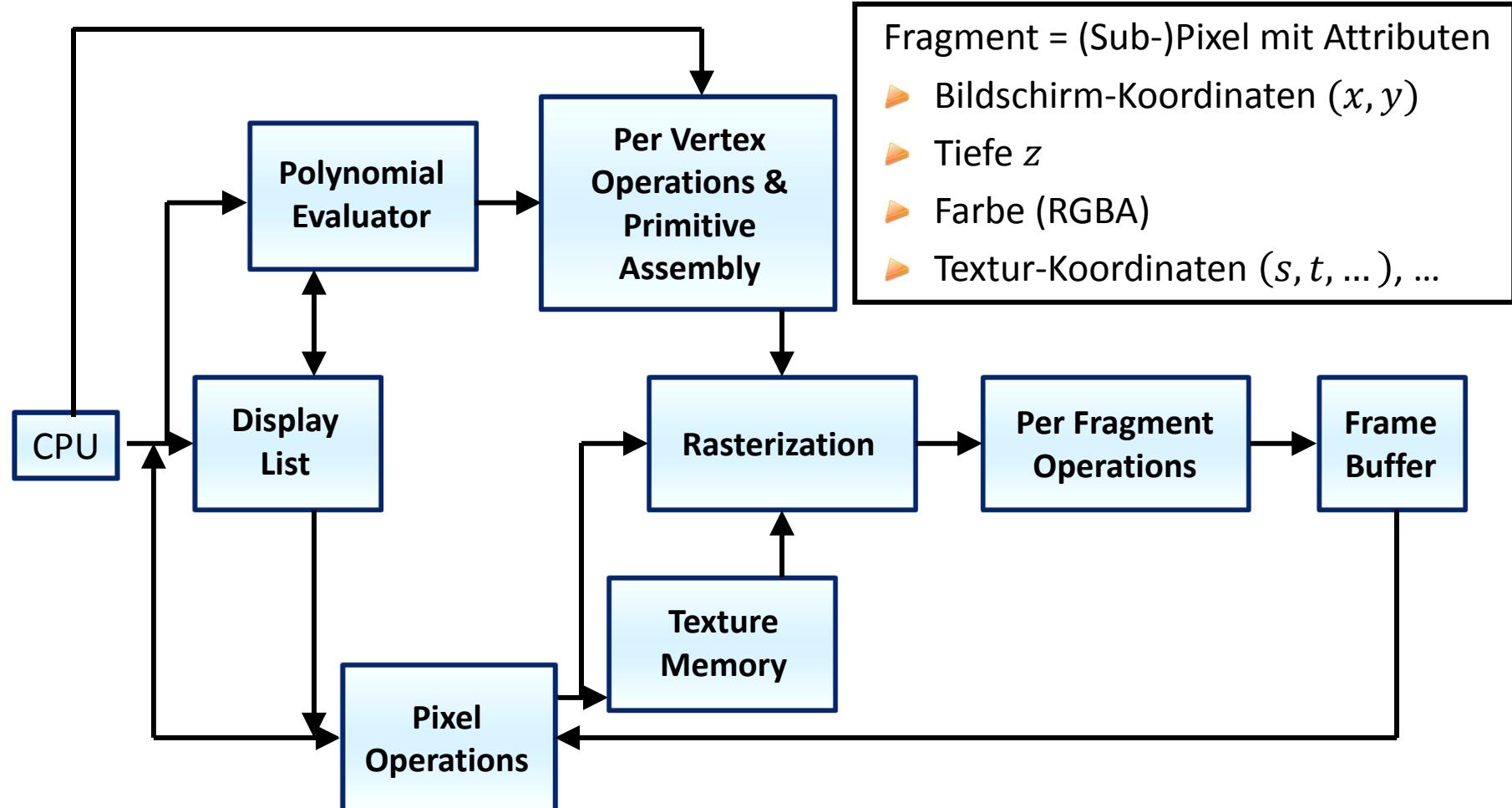


Modernes OpenGL

OpenGL 2.x/3.x/4.x, OpenGL ES 2.x/3.x

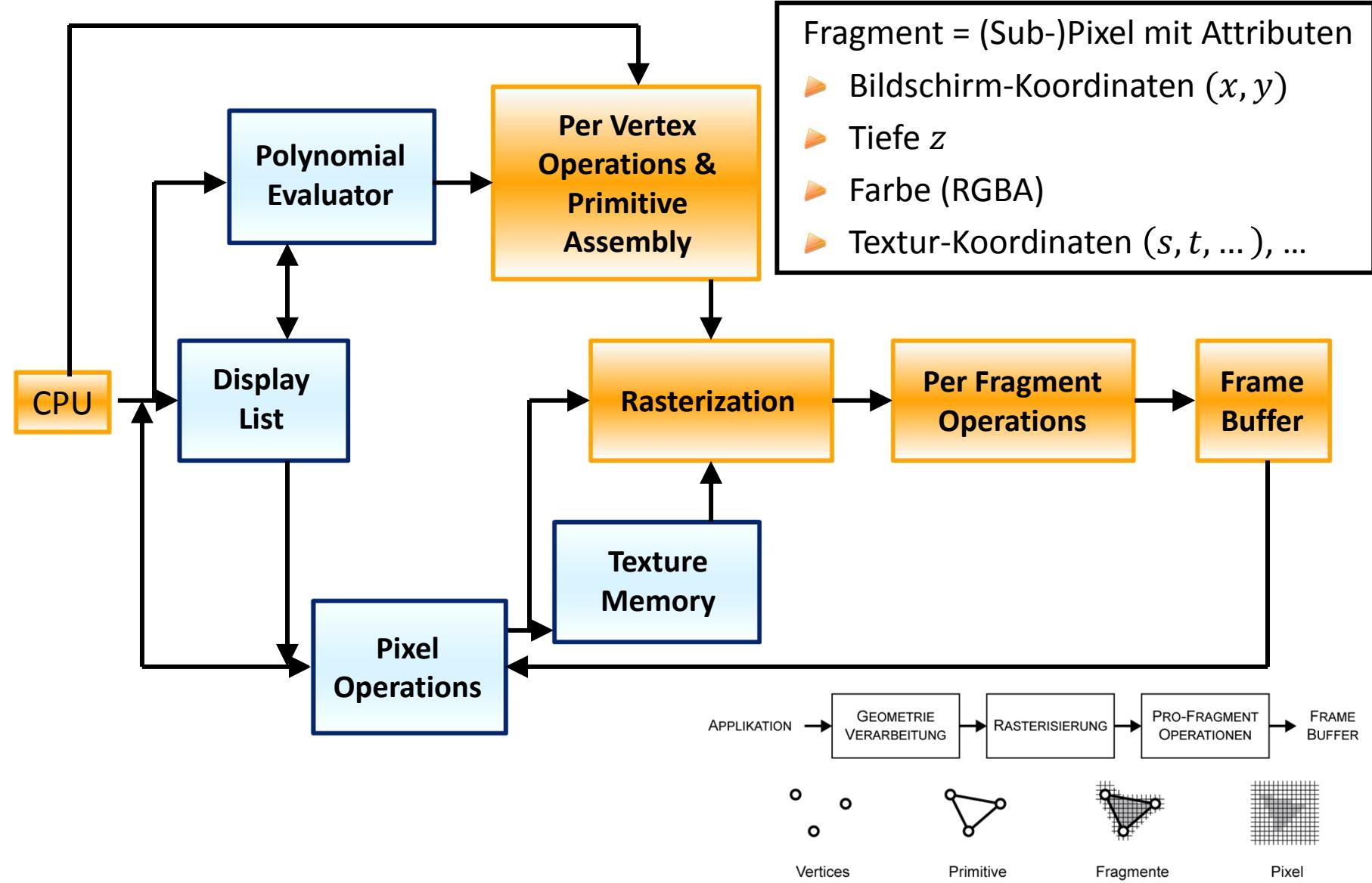
(entspricht der Funktionalität von DirectX 9, 10, 11)

Klassische OpenGL-Pipeline

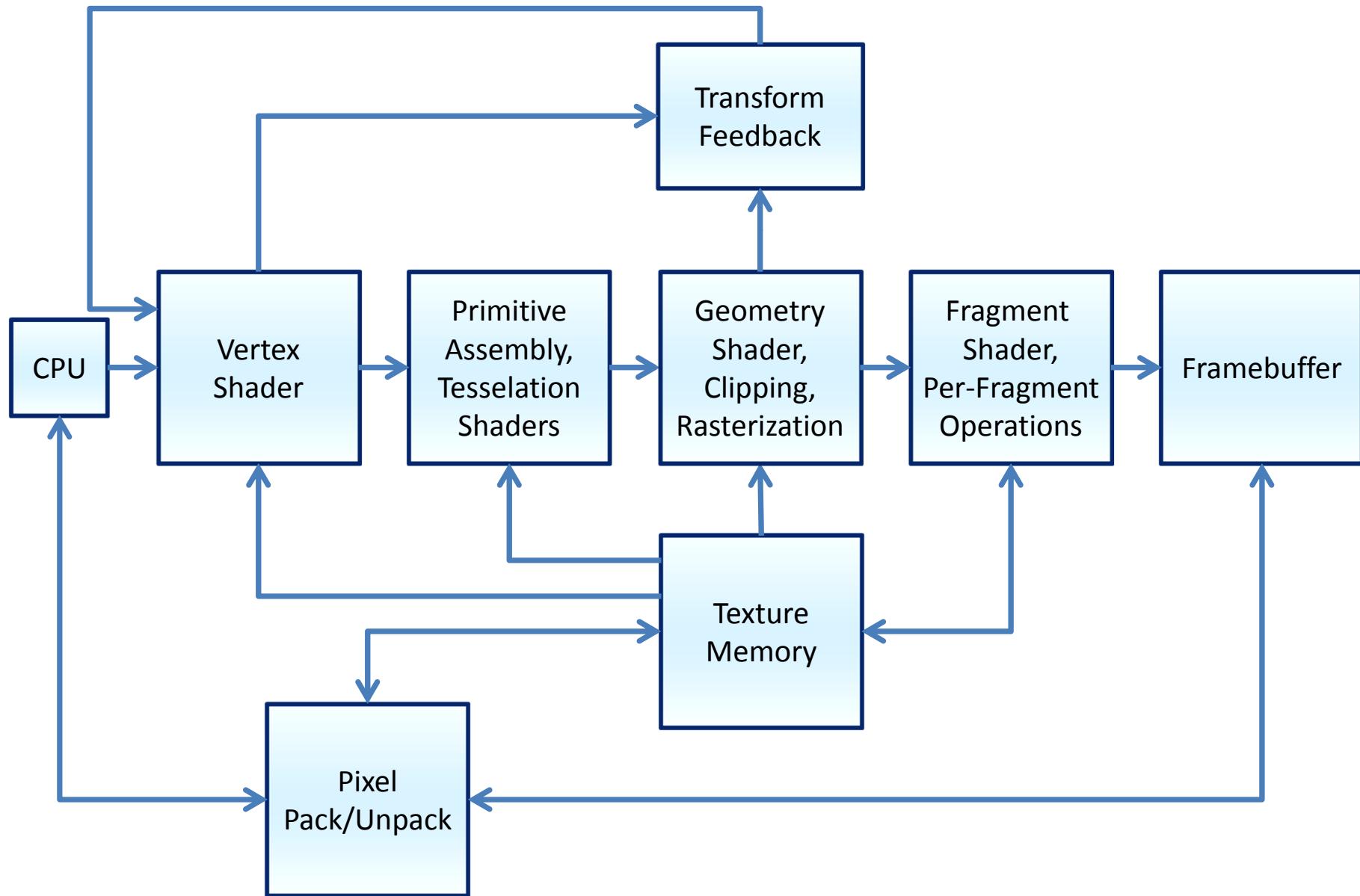


- ▶ Texturen und Pro-Fragment-Operationen kann OpenGL natürlich schon immer → besprechen wir aber später

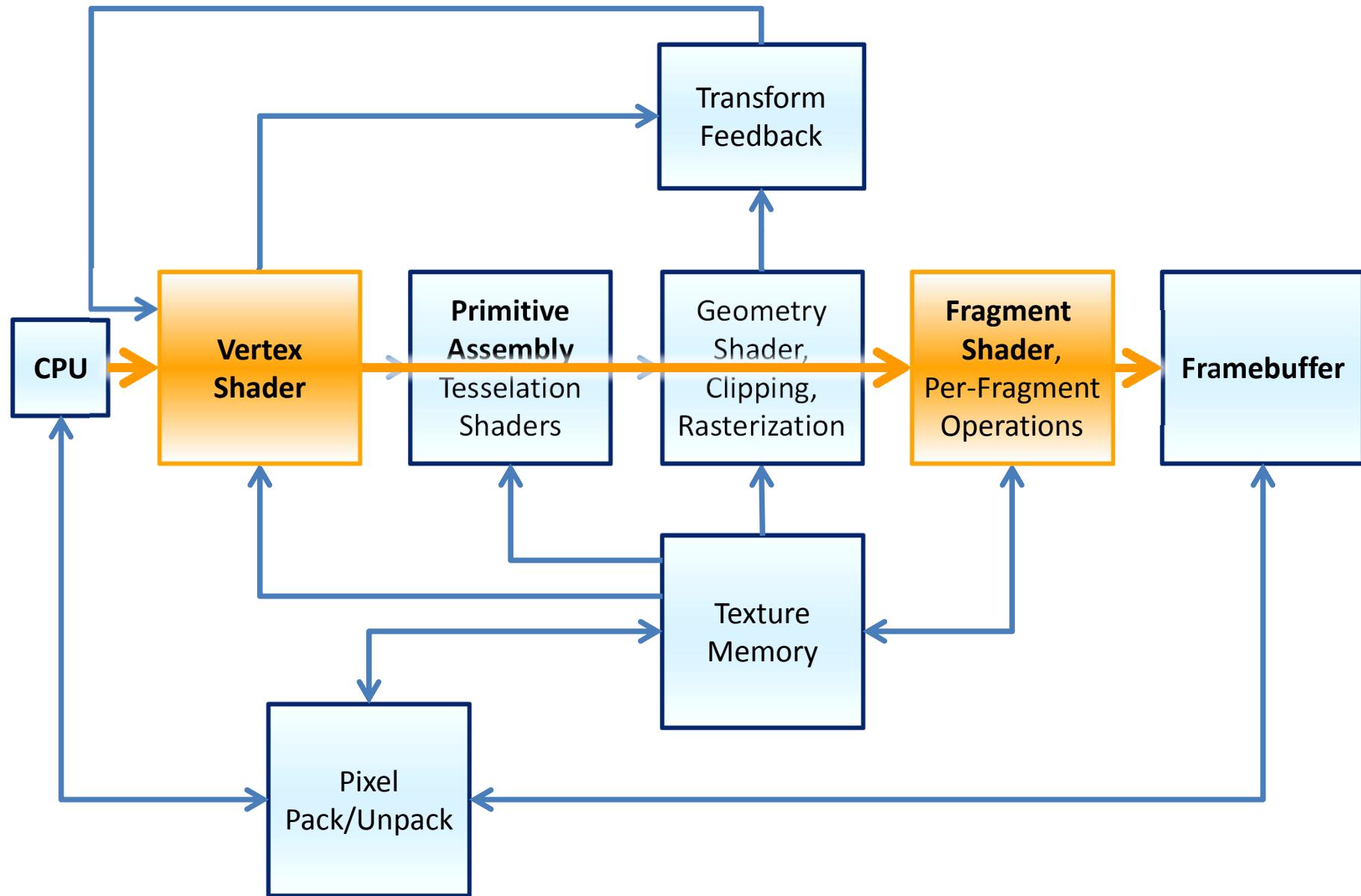
Klassische OpenGL-Pipeline



OpenGL(4.2+)-Pipeline



OpenGL(4.2+)-Pipeline



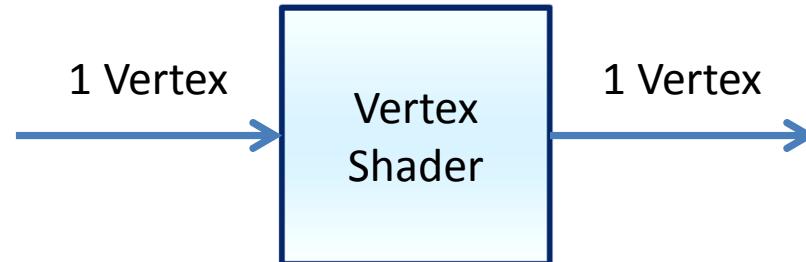
Geometrieverarbeitung: Vertex Shader



- ▶ Programmierung mit der C-ähnlichen OpenGL Shading Language (GLSL)
 - ▶ im [Kompatibilitätsprofil](#), d.h. wenn man OpenGL-Altlasten mitschleppt:
Zugriff auf OpenGL Zustände (Matrizen, Lichtquellen, ...)
- ▶ Transformation **einzelner** Vertizes und Verarbeitung deren Attribute
- ▶ Beispiel eines Vertex Shader / Vertex Program im Kompatibilitätsprofil:

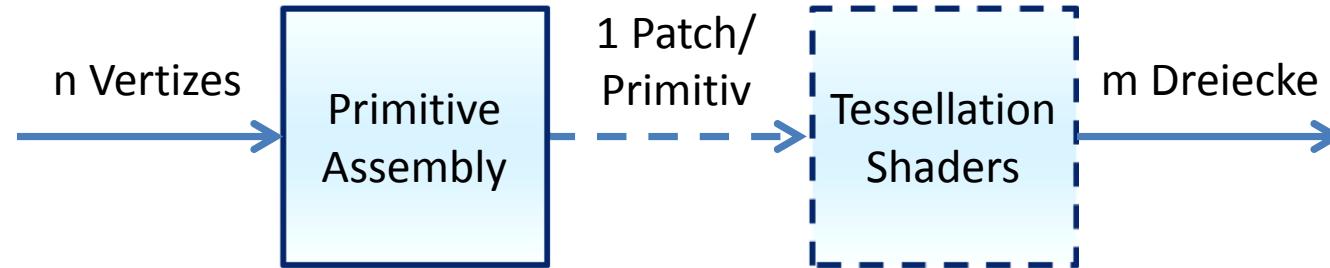
```
void main() {  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Geometrieverarbeitung: Vertex Shader



- ▶ Transformation **einzelner** Vertizes und Verarbeitung deren Attribute
 - ▶ keine Vertex-Erzeugung
 - ▶ keine Vertex-Löschung
 - ▶ keine Informationen über andere Vertizes
- ▶ Berechnung von Attributen, die für Fragmente interpoliert werden sollen
 - ▶ z.B. Beleuchtung per Vertex (Gouraud Shading, altmodisch) oder
 - ▶ Normalen für Phong Shading

Geometrieverarbeitung: Assembly + Tessellation



- ▶ **Primitive Assembly** setzt aus den Vertices die angeforderten Primitive zusammen und ist **nicht programmierbar**:
 - ▶ `GL_POINT`, `GL_LINE[*]`, `GL_TRIANGLE[*]` in ihren unterschiedlichen Modi
`*_FAN`, `*_LOOP`, `*_STRIP`
- ▶ **Tessellation** (siehe Interaktive Computergrafik im Sommersemester):
 - ▶ Tessellation Control Shader bestimmt die Unterteilung
 - ▶ Tessellator führt sie durch (nicht programmierbar)
 - ▶ Tessellation Evaluation Shader arbeitet auf der resultierenden Geometrie
 - ▶ spezieller Primitivtyp `GL_PATCH`

Beispiel Tessellation – Unigine Benchmark



Eingabegeometrie vor der Unterteilung durch die Tessellation-Einheit



Beispiel Tessellation – Unigine Benchmark



Displacement Mapping: Unterteilung und Verschiebung der neuen Vertizes

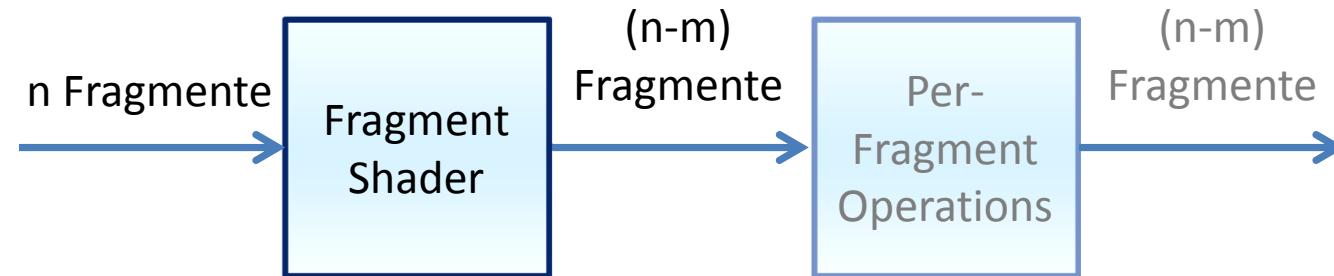


Geometrieverarbeitung: Geometry Shader etc.



- ▶ Geometry Shader (programmierbar, optional)
 - ▶ bearbeitet einzelne Primitive (Punkt, Linie, Dreieck)
 - ▶ kann Primitive vervielfachen, entfernen oder beliebig umwandeln (Punkte zu Dreiecke etc.)
- ▶ Clipping (am Sichtvolumen) ist nicht programmierbar
 - ▶ verwirft unsichtbare Punktprimitive
 - ▶ verändert Dreiecke durch Schnitt mit Sichtvolumen
- ▶ anschließend: perspektivische Division (nicht programmierbar)
- ▶ Rasterisierung generiert Fragmente für den Ausgabepuffer...

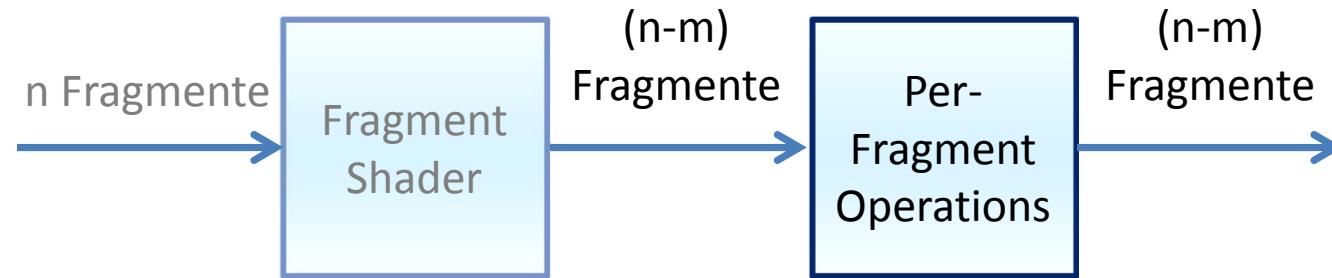
Fragmentverarbeitung



- ▶ ...für jedes dieser Fragmente wird ein **Fragment Shader/Program** ausgeführt
 - ▶ Berechnung von **Farbe**, **Transparenz**, und optional **Tiefe** pro Fragment
 - ▶ z.B. Phong Shading, also Beleuchtungsberechnung pro Pixel
 - ▶ Eingabe-Attribute werden innerhalb des Primitivs interpoliert
 - ▶ z.B. pro-Eckpunkt Normalen, die im Vertex Shader weitergegeben wurden
- ▶ Beispiel eines Fragment Programs im OpenGL 2.0 oder Kompatibilitätsprofil:

```
void main() {  
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 0.0 );  
}
```

Fragmentverarbeitung

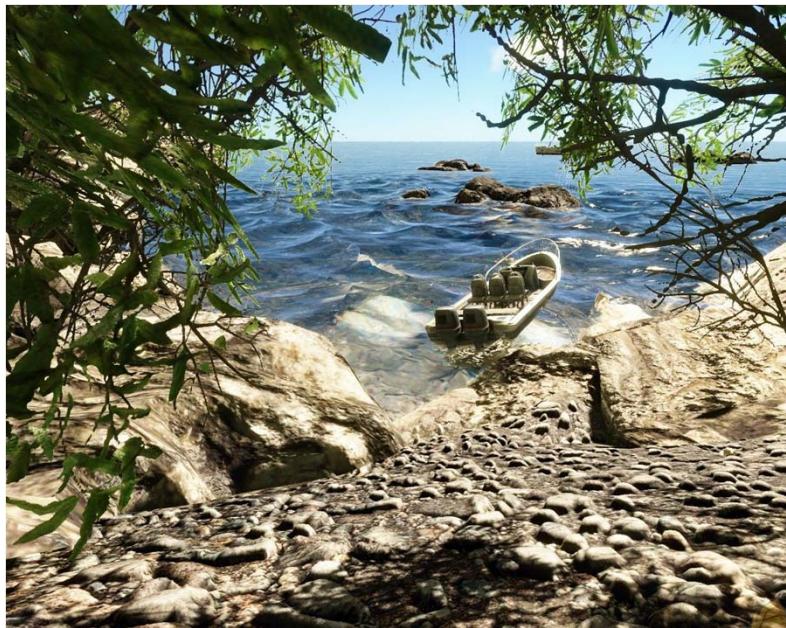


- ▶ Fragmentoperationen sind sehr wichtig für Rasterisierungsverfahren
 - ▶ Tiefentest
 - ▶ Blending: Kombination von Fragment- und Pixelfarbe im Framebuffer
 - ▶ die meisten dieser Fragmentoperationen gibt es ebenfalls im klassischen OpenGL
 - ▶ später mehr!

Möglichkeiten mit programmierbarer Grafik-HW



- ▶ beliebige Beleuchtungsberechnung und Materialien programmierbar
 - ▶ nicht mehr an `glMaterial` / `glLight` gebunden
 - ▶ im Kompatibilitätsprofil trotzdem Zugriff auf OpenGL States
 - ▶ prozedurale Texturierung direkt auf der Grafik-HW
 - ▶ komplexe Datenstrukturen für Texture Mapping, z.B. Octrees
- ▶ komplexe Geometrieverarbeitung und Animation
 - ▶ Character Animation, Skinning, ...



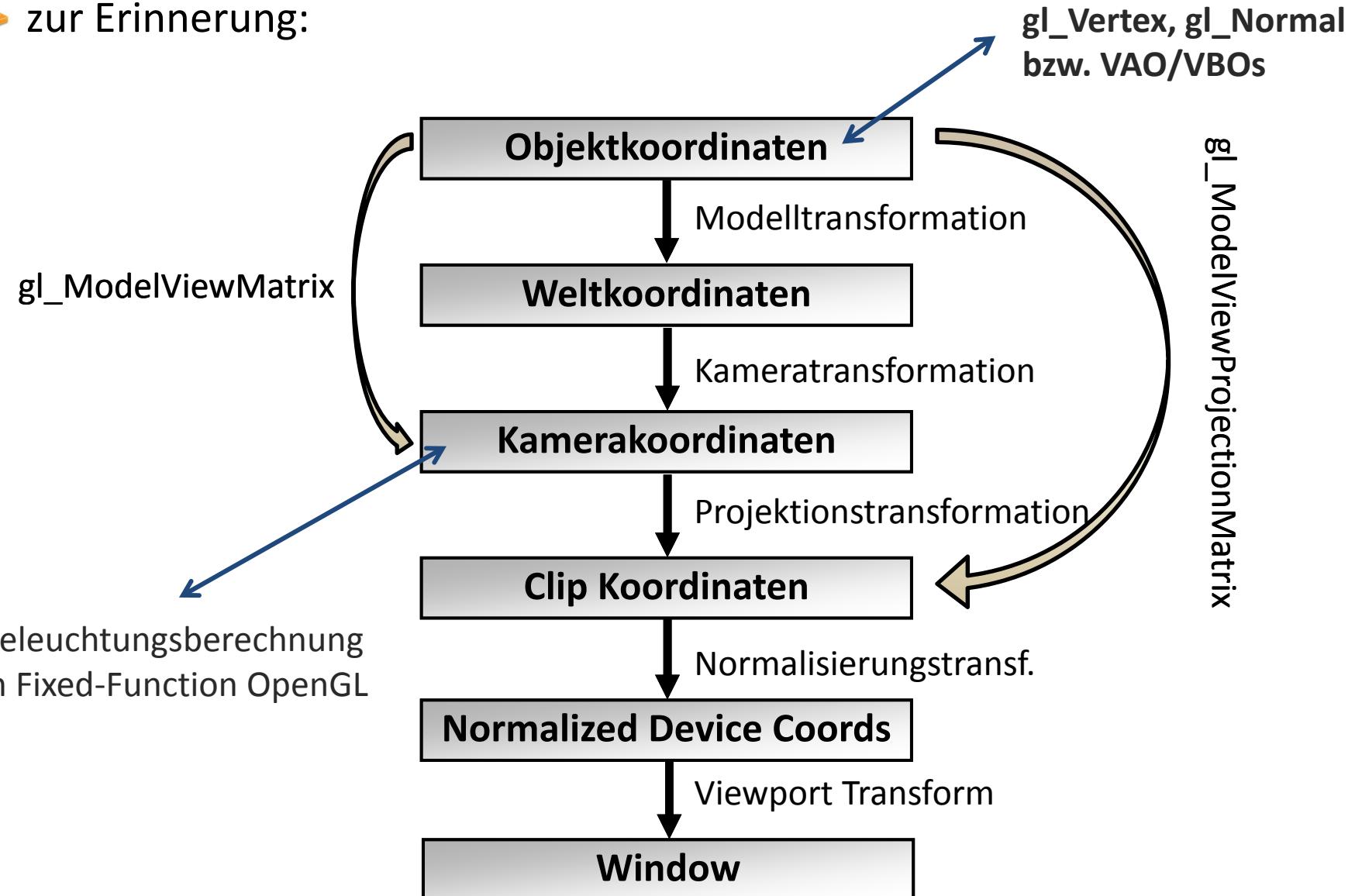
OpenGL Shading Language



- ▶ stellen Sie sich vor: „1 Prozessor pro Vertex, Primitiv oder Fragment“
- ▶ C-ähnliche Sprache
 - ▶ mit speziellen Datentypen und Befehlen
 - ▶ ...und Einschränkungen (z.B. keine Pointer)
- ▶ Basis-Datentypen:
 - ▶ **float, double, bool, int, half**
 - ▶ **float,double/int** wie in C, **bool** strikt nur **true/false**
 - ▶ reserviert: **fixed**
 - ▶ GLSL Precision Qualifiers
- ▶ Vektoren mit 2, 3 oder 4 Komponenten
 - ▶ **vec{2, 3, 4}, dvec{2, 3, 4}, bvec{2, 3, 4}, ivec{2, 3, 4}**
- ▶ Matrizen 2×2 , 3×3 oder 4×4 bestehend aus Floats oder Doubles
 - ▶ **[d]mat2, [d]mat3, [d]mat4**

Transformationspipeline

► zur Erinnerung:



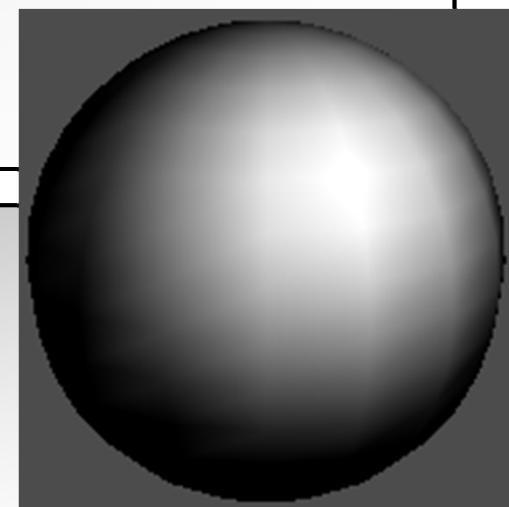
Mehr Infos:

<http://www.opengl.org/resources/faq/technical/viewing.htm>

GLSL 3.x/4.x Diffuse Beleuchtung



```
uniform mat4 matrixMVP, matrixMV, matrixNrml;  
uniform vec3 lightSourcePos;  
in vec4 in_position;  
in vec3 in_normal;  
out vec4 color;  
void main() {  
    gl_Position = matrixMVP * in_position;  
    // Beleuchtungsberechnung in Kamerakoordinaten  
    vec3 P = vec3( matrixMV * in_position );  
    vec3 N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );  
    vec3 L = normalize( lightSourcePos - P );  
    color = vec4( max( 0.0, dot( L, N ) ) );  
}
```



```
in vec4 color;  
out vec4 out_color;  
void main() {  
    out_color = color;  
}
```

GLSL 3.x/4.x „Per-Pixel“ Beleuchtung

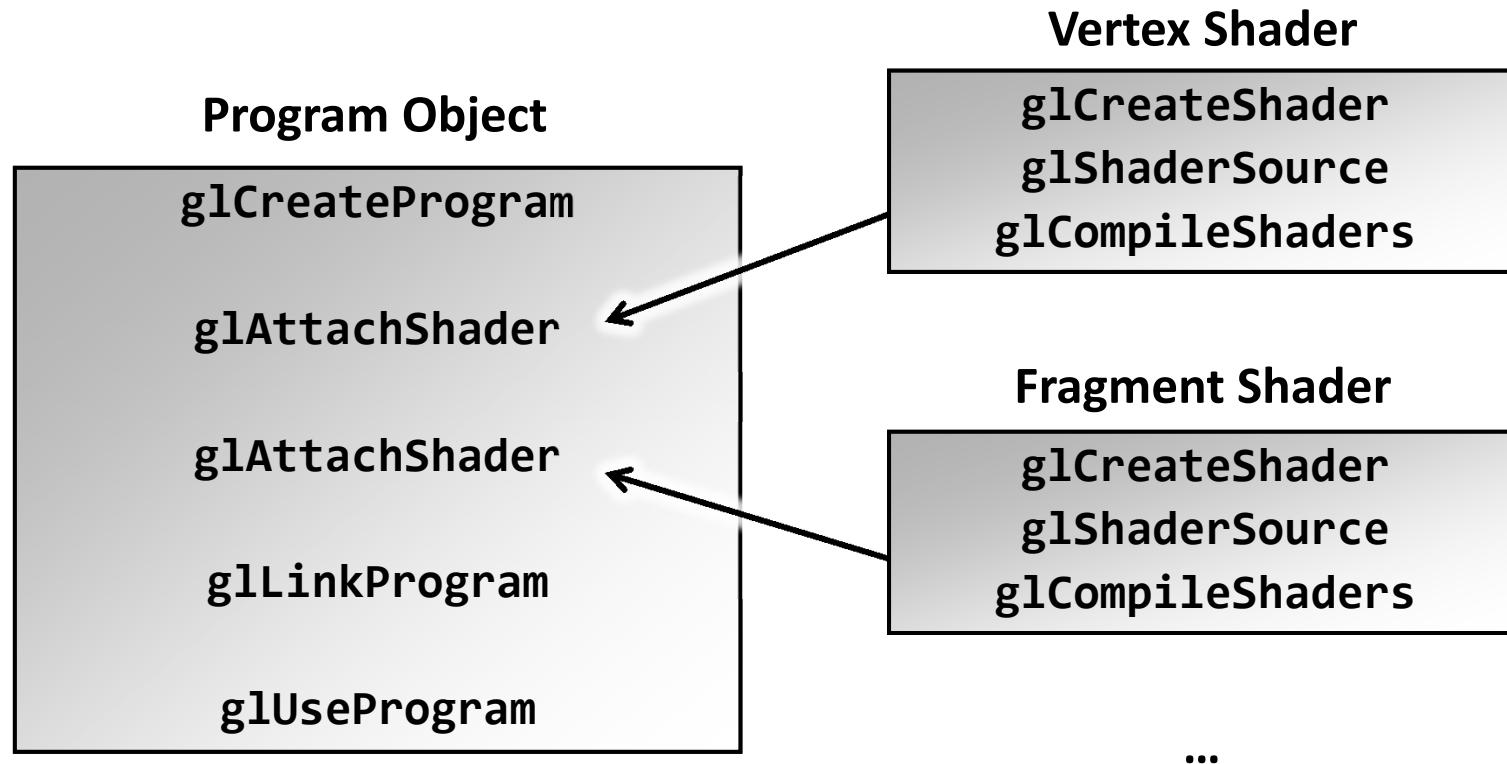
```
uniform mat4 matrixMVP, matrixMV, matrixNrml;  
uniform vec3 lightSourcePos;  
in vec4 in_position;  
in vec3 in_normal;  
out vec3 L, N;  
void main() {  
    gl_Position = matrixMVP * in_position;  
    vec3 P = vec3( matrixMV * in_position );  
    N = vec3( matrixNrml * vec4( in_normal, 0.0 ) );  
    L = lightSourcePos - P;  
}
```

```
in vec3 L, N;  
out vec4 out_color;  
void main() {  
    float kd = max( 0.0, dot( normalize(L),  
                             normalize(N) ) );  
    out_color = vec4( kd );  
}
```



OpenGL Shading Language: Nutzung

- ▶ GLSL steht über OpenGL-Aufrufe zur Verfügung
- ▶ Shader/Programs sind OpenGL-Objekte



- ▶ Erklärung der einzelnen Befehle im Bonus-Material und Beispiel im Framework (...und das müssen Sie natürlich nicht auswendig kennen)

OpenGL-Objekte



- ▶ Texturen, Geometrie, Shader etc. sind OpenGL-Objekte
 - ▶ keine Objekte im OOP-Sinn!
 - ▶ zustandsbehaftet: Texturobjekte speichern die Textur inkl. Mipmaps, aber auch Filter- und Adressierungsmodi
- ▶ Objekterzeugung, Initialisierung, Löschen
 - ▶ erzeuge ein **Handle** (in OpenGL auch oft **name** genannt)
`GLuint someHandle;`
`glGen{Textures|Buffers|...}(1, &someHandle);`
 - ▶ selektiere ein Objekt (binding)
`glBind*(<target>, someHandle);`
 - ▶ übergebe die Daten an OpenGL (solange sich die Daten nicht ändern, müssen sie nicht nochmal übertragen werden)
 - ▶ deselektiere ein Objekt (unbinding)
`glBind*(<target>, 0);`
 - ▶ Objekt löschen
`glDelete*(1, &someHandle);`

Program: Aktivieren



- ▶ Verwenden eines Program-Objects (Binding)

```
void glUseProgram( GLuint program );
```

- ▶ Zurück zur Fixed-Function-Pipeline (falls im OpenGL Profil unterstützt):

```
glUseProgram( 0 );
```

- ▶ Achtung: in modernem OpenGL ist die Semantik hier unklar, da es eigentlich keine Fixed-Function-Pipeline mehr gibt!

- ▶ im Nvidia-Treiber 280.26 bspw. funktioniert es trotzdem ;-)



GLSL Variablen



► Deklaration

```
float a, b;  
int c = 2;  
bool d = true;
```

► Achtung: Konstruktoren und Type Casts

```
float b = 2;           // (!) kein automatisches Type Casting  
float e = (float)2;    // (!) Konstruktor benötigt  
int a = 2;  
float c = float( a );  
vec3 f;  
vec3 g = vec3( 1.0, 2.0, 3.0 );
```

► Anmerkung:

- ▶ diese Schriftart ist GLSL-Code
- ▶ diese Schriftart ist CPU/OpenGL-Code

GLSL Variablen



► Initialisierung

```
vec2 a = vec2( 1.0, 2.0 );
vec2 b = vec2( 3.0, 4.0 );
vec4 c = vec4( a, b )           // c = vec4( 1.0, 2.0, 3.0, 4.0 );
float h = 3.0;
vec3 j = vec3( a, h );
```

► Matrizen

```
mat4 m = mat4( 1.0 )           // Diagonalmatrix mit 1.0
vec2 a = vec2( 1.0, 2.0 );
vec2 b = vec2( 3.0, 4.0 );
mat2 n = mat2( a, b );         // a und b werden Spaltenvektoren
mat2 k = mat2( 1.0, 0.0, 0.0, 1.0 ); // 2x2 Diagonalmatrix
```

GLSL: Vektoren und Swizzling



- Zugriff auf die Komponenten der Vektoren wie auf **structs** in C

```
vector.[xyzwrgbastpq]
```

- Koordinaten (xyzw), Farbkanäle (rgba) und Texturkoordinaten (stpq) sind synonym und können beliebig vermischt werden
- Mischen geht nicht mehr!!!

```
vec3 col; col.rgb == col.xgp == col.stp; // alles äquivalent
```

- Verdrehen und/oder Replizieren ist möglich

```
vec4 g, h; g.wxyz = h.yyx;
```

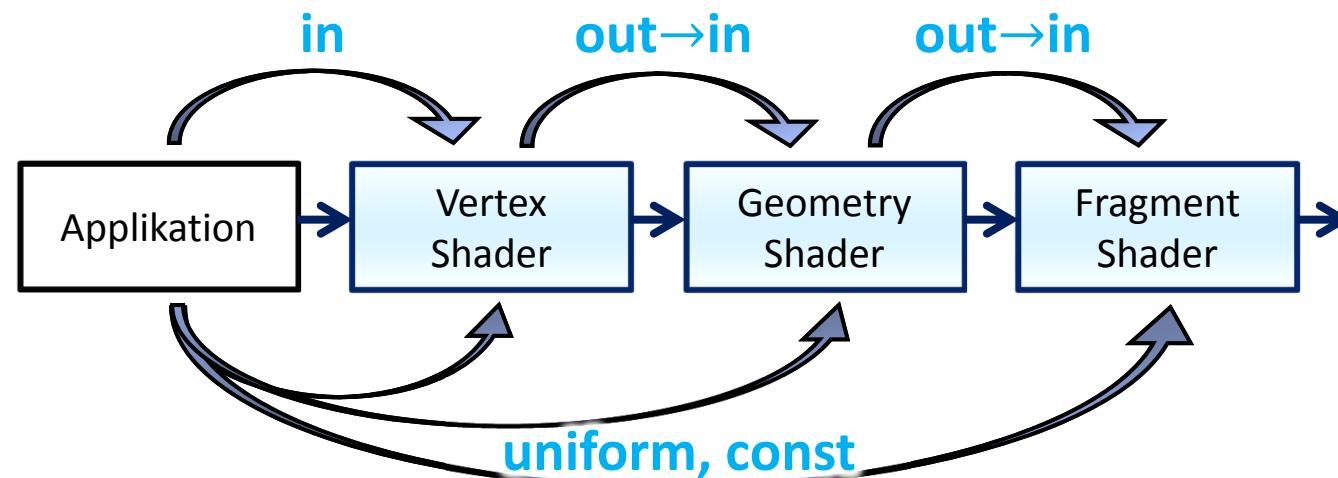
- aber Komponenten müssen zum Typ passen

```
vec2 g; vec4 f; f = g.xxxz; // so nicht!
```

GLSL Qualifiers (OpenGL 3.x, 4.x)

Definition des Verwendungszwecks von Variablen

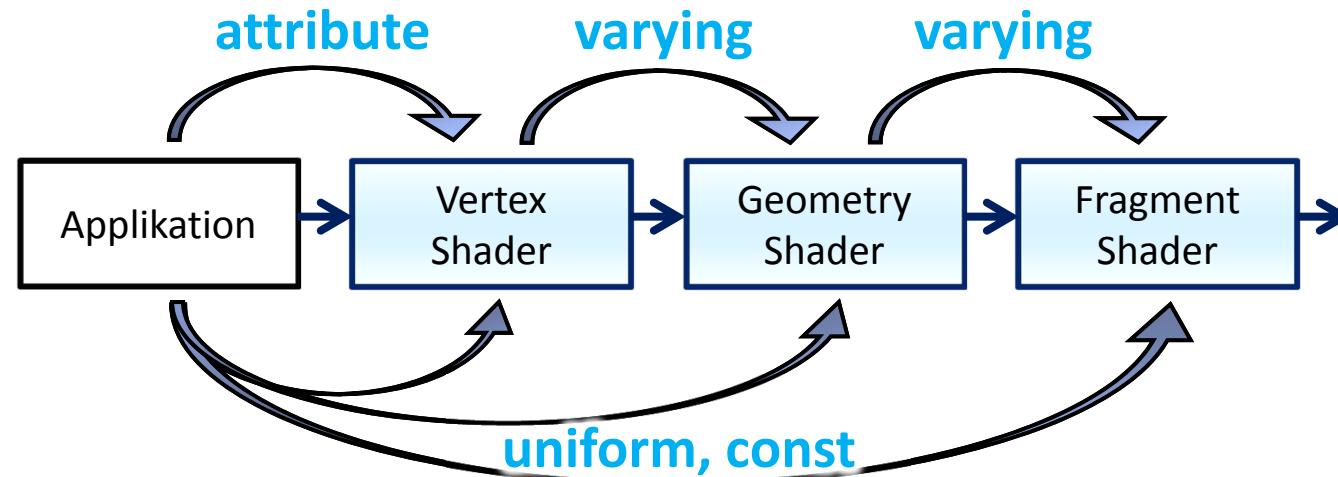
- ▶ **const**: Konstante, zur Compile-Zeit festgelegt
- ▶ **uniform**: read-only
 - ▶ globale Variablen, die pro Aufruf zum Zeichnen konstant bleiben
 - ▶ werden von der Applikation gesetzt (z.B. eine Transformationsmatrix, Position von Kamera und Lichtquelle, ...)
- ▶ **in**: Eingabe-Attribute des aktuellen Shaders
 - ▶ Attribute eines Vertex, z.B. Koordinate, Normale, Farbe etc.
 - ▶ oder Ausgabe eines vorherigen Shaders
- ▶ **out**: Ausgabe-Attribute des aktuellen Shaders



GLSL Qualifiers (OpenGL 2.x, OpenGL ES)

Definition des Verwendungszwecks von Variablen

- ▶ **const**: Konstante, zur Compile-Zeit festgelegt
- ▶ **uniform**: read-only
 - ▶ globale Variablen, die pro Aufruf zum Zeichnen konstant bleiben
 - ▶ werden von der Applikation gesetzt (z.B. eine Transformationsmatrix, Position von Kamera und Lichtquelle, ...)
- ▶ **attribute**: nur für Vertex Shader, read-only
 - ▶ Vertex-Attribute, z.B. Koordinate, Normale, Farbe etc.
- ▶ **varying**: weitergegebene/interpolierte Werte
 - ▶ schreiben in einem Shader, lesen im darauffolgenden Shader



GLSL 3.x/4.x Minimalbeispiel

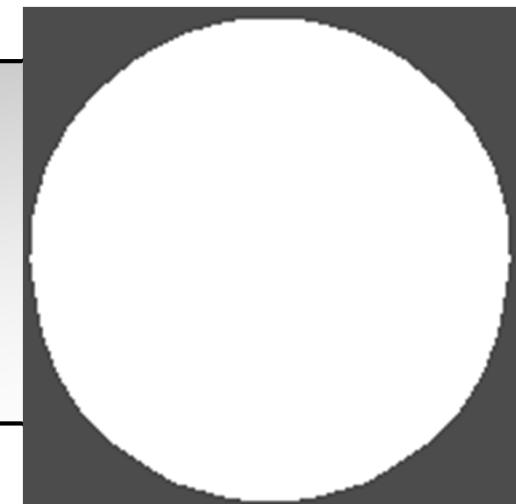


- Vertex Program: **gl_Position** ist eine vordefinierte Ausgabe und muss immer geschrieben werden

```
#version 150 core
uniform mat4 matrixMVP;
in vec4 in_position;
void main() {
    gl_Position = matrixMVP * in_position;
}
```

- Fragment Program:

```
#version 150 core
out vec4 out_color;
void main() {
    out_color = vec4( 1.0, 1.0, 1.0, 0.0 );
}
```



GLSL 3.x/4.x



- ▶ woher weiß OpenGL, dass die Vertex-Koordinaten in **in_position** stehen sollen?

```
#version 150 core  
uniform mat4 matrixMVP;  
in vec4 in_position;  
void main() {  
    gl_Position = matrixMVP * in_position;  
}
```

- ▶ diese Verbindung („binding“) muss explizit hergestellt werden
- ▶ wenn die Semantik der OpenGL FF-Pipeline verwendet werden soll (was wir aber im Folgenden vermeiden wollen – keine Altlasten):

```
#version 150 compatibility  
void main() {  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

GLSL 3.x/4.x Minimalbeispiel



- ▶ Vertex Program:

```
uniform mat4 matrixMVP;  
in vec4 in_position;  
void main() {  
    gl_Position = matrixMVP * in_position;  
}
```

- ▶ Fragment Program:

```
out vec4 out_color;  
void main() {  
    out_color = vec4( 1.0, 1.0, 1.0, 0.0 );  
}
```

- ▶ noch ein letztes Binding: es muss festgelegt werden, welche Fragment Shader Variable die Ausgabe-Farbe enthält
`glBindFragDataLocation(shader, 0, "out_color");`

GLSL Control Flow



- ▶ **if/else, for, while, do/while**
 - ▶ **continue:** Sprung zur nächsten Iteration
 - ▶ **break:** Beendigung der Schleife
 -  ▶ gehen Sie nicht davon aus, dass der Kontrollfluss so läuft, wie Sie es von CPUs gewohnt sind
 - ▶ Bsp. bei **if/else** werden u.U. beide Äste ausgeführt und am Ende das Ergebnis entsprechend der **if**-Klausel ausgewählt
- ▶ **discard**
 - ▶ nur im Fragment Shader
 - ▶ beendet den Shader, ohne ein Fragment zu produzieren

GLSL Funktionen



- ▶ wie C-Programme können Shader durch Funktionen strukturiert werden
 - ▶ obligatorisch ist die Funktion: **void main()**
- ▶ Definition der Parameter für eigene Funktionen
 - ▶ **in**: es handelt sich um einen Eingabe-Parameter
 - ▶ **out**: Rückgabewert (auch möglich: **return**, dann aber nur eine Variable)
 - ▶ **inout**: Eingabe und gleichzeitig Rückgabewert
 - ▶ bei keiner Angabe wird **in** angenommen

```
void computeLighting( in vec3 lightPos, in vec3 normal,
                      out float diffuse, out float specular ) {
    ...
}
```

- ▶ Überladen von Funktionen ist möglich
- ▶ Rekursion ist nicht spezifiziert, weil es keinen Stack auf GPUs gibt
 - ▶ wird Rekursion benötigt, muss ein selbstverwalteter Stack verwendet werden

GLSL Built-In Functions



- ▶ es gibt eine Auswahl von Funktionen in GLSL, die sich an graphischen Anwendungen orientieren
- ▶ die meisten arbeiten Komponentenweise (SIMD)
 - ▶ degrees, radians
 - ▶ [a]sin[h], [a]cos[h], [a]tan[h]
 - ▶ pow, exp[2], log[2], [inverse]sqrt
 - ▶ abs, sign, floor, trunc, round[Even], ceil, fract, mod[f], min, max
 - ▶ clamp(x , minV, maxV), mix(x, y, a) = lineare Interpolation „lerp“
 - ▶ step($x, edge$) = ($x < edge$) ? 0.0 : 1.0
 - ▶ smoothstep(a, b, x) = Hermite-Interpolation
 - ▶ length, distance, dot, cross, normalize, reflect, refract, isnan, isinf
 - ▶ Packing, Bit-Operationen, Matrixoperationen, ...

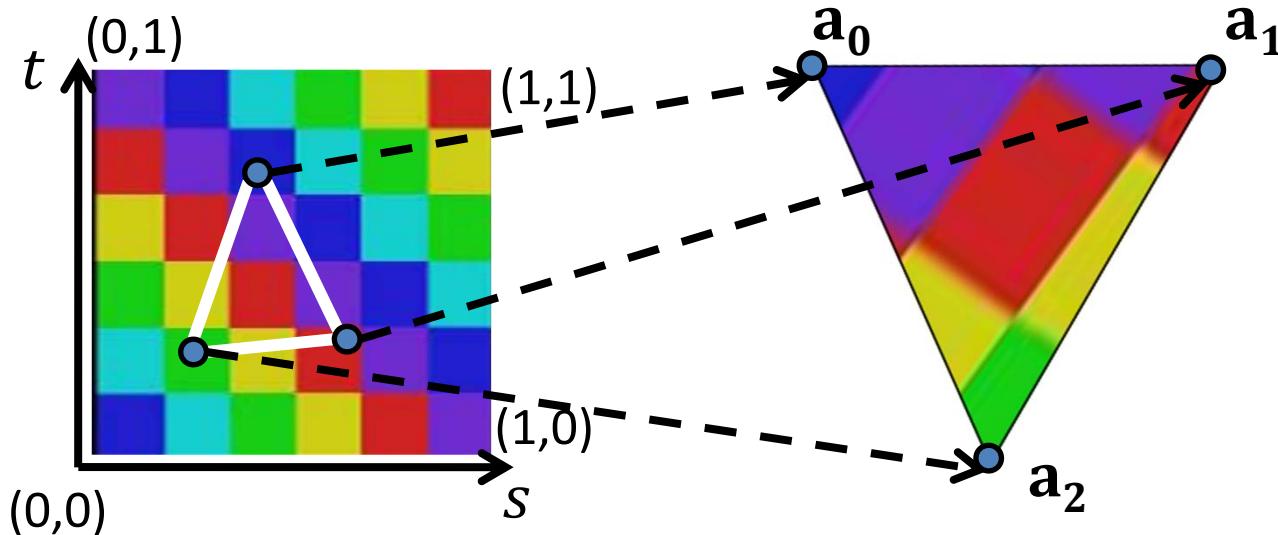
<http://www.khronos.org/files/opengl45-quick-reference-card.pdf>

Vorsicht: klare Kennzeichnung der core-Funktionalität fehlt (siehe Spec)



Texturkoordinaten

- jeder Vertex $\mathbf{a}_i = (x_i, y_i, z_i)$ hat eine Texturkoordinate (s_i, t_i)



- Immediate Mode: `glTexCoord{1|2|3|4}{fd...}[v](...)`
- in OpenGL 3.x/4.x einfach einen (weiteren) Puffer verwenden

```
glBufferData( GL_ARRAY_BUFFER, sizeof(GLfloat) * 2 * 3,  
              textureCoordinates, GL_STATIC_DRAW );
```

- Komponenten werden in OpenGL mit s, t, p, q bezeichnet
 - Umbenennung auf Grund des Konflikts mit r (rot)



Texturen in OpenGL



- ▶ Rasterbilder mit frei wählbarer...
 - ▶ **Größe:** bis `GL_MAX_TEXTURE_SIZE`; seit GeForce500: ≥ 16384
 - ▶ **Komponentenzahl:** 1 bis 4 (R, RG, RGB, RGBA, ...)
 - ▶ weitere Formate für Tiefe und Tiefe+Stencil etc.
 - ▶ **Layout:** 1D, 2D, 3D Texturen, Texture-Arrays, Cube-Maps etc.
 - ▶ **Datentyp:** 8, 10, 12 Bit Integer pro Komponente, 16/32 Bit Floating Point, komprimierte Texturen, und viele weitere – teils exotische – Varianten, z.B. `R11F_G11F_B10F`
- ▶ Texturen sind OpenGL-Objekte
 - ▶ ... und sie funktionieren auch so, aber es gibt dedizierte Funktionen (hier kein Unterschied zu klassischen OpenGL):

```
glGenTextures  ( GLsizei n, GLuint *textures );  
glBindTexture  ( GLenum target, GLuint texture );  
glDeleteTextures( GLsizei n, const GLuint *textures );
```

Texturen in OpenGL



- Übergeben der Texturdaten

```
glTexImage{1,2,3}D( target, level, int_format,  
width, height, border, format, type, *texels )
```

- ▶ *target* bezeichnet das Ziel
`GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D,`
`GL_TEXTURE_CUBE_MAP_POSITIVE_X, ...`
- ▶ mit *Level* kann angegeben werden, welche Mipmap-Stufe gerade angeben wird (*Level=0* ist höchste Auflösung)
- ▶ Rasterbild wird in ein internes Texturformat (*int_format*) konvertiert
 - ▶ `GL_RGBA`, `GL_RG`, `GL_RGBA16F` etc.
 - ▶ komprimierte Formate, z.B. `COMPRESSED_RGB_BPTC_SIGNED_FLOAT`
- ▶ *width*, *height*, *format*, *type* des Eingabebilds
- ⚠️ ▶ *border* muss immer *0* sein (Relikt aus sehr altem OpenGL)
- ▶ verwende Bild im Framebuffer als Textur: `glCopyTexImage*D(...)`

Texturen in OpenGL



- ▶ Texturen müssen zur Verwendung an Textureinheiten gebunden werden

```
glActiveTexture( GL_TEXTURE<i> );  
glBindTexture ( GL_TEXTURE_[123]D, texture );
```

- ▶ Textureinheit = Hardware zum Auslesen der Textur, Interpolieren etc.

- ▶ Anzahl der Textureinheiten variiert (GeForce 500: 160 Einheiten)
 - ▶ Abfrage: `glGet*` mit `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`

- ▶ Konstanten sind nur bis 31 definiert, darum z.B.:
`glActiveTexture(GL_TEXTURE0 + 64);`

- ▶ Texturkoordinaten der Vertizes kommen aus...

- ▶ wie gerade gesehen: aus eigenen Vertex-Attributen/Vertex Buffer oder über Berechnungen im Vertex- oder Fragment-Shader

- ▶ ab der GeForce 600 Serie: beliebig viele Texturen mit „Bindless Textures“

Texturfilterung und Adressierung in OpenGL



Steuerung über `glTexParameter{i|f}`

- ▶ automatische Erzeugung der Mipmaps:

```
glTexImage2D( target, ... );
glGenerateMipmap( target );
```

- ▶ Beispiele für Minification und Magnification:

```
// Nearest Neighbor Filtering bei Vergrößerung
glTexParameteri( GL_TEXTURE_2D,
                  GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST )
```

```
// Nearest Neighbor Filtering und Wahl der nahsten
// Mipmap (bei Verkleinerung natürlich)
glTexParameteri( GL_TEXTURE_2D,
                  GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST_MIPMAP_NEAREST )
```

```
// Tri-lineares Filtering
glTexParameteri( GL_TEXTURE_2D,
                  GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR )
```

Texture Wrapping



- ▶ Repeat/Wrapping: Fortsetzen einer Textur über $[0,1]^{[1,2,3]}$ hinaus
(Repeat nur bei quadratischen Texturen erlaubt!)
- ▶ Adressierung wird für jede Dimension separat gewählt
(Achtung: hier ist die 3. Dimension inkonsistenterweise **WRAP_R!**)
- ▶ Beispiele:

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_WRAP_S,  
                  GL_CLAMP_TO_EDGE )
```

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_WRAP_T,  
                  GL_REPEAT )
```

- ▶ weitere Modi:
`GL_MIRRORED_REPEAT, GL_CLAMP_TO_BORDER, GL_CLAMP_TO_EDGE`
- ▶ Achten Sie darauf, dass Sie auch Mipmaps generiert haben, wenn Sie sie verwenden → siehe http://www.opengl.org/wiki/Common_Mistakes

Texturen



- ▶ Auslesen von Texturen in einem Shader über „Sampler“ `sampler[1,2,3]D`
 - ▶ ein Sampler repräsentiert im Prinzip eine Textureinheit
 - ▶ damit verknüpft ist das Texturobjekt, Filtering-Modi etc.
- ▶ zur Verwendung müssen Textur und Sampler verbunden werden
 - ▶ Deklaration und Zugriff im Shader:

```
uniform sampler2D sam;  in vec2 texCoord;  
...  
vec4 color = texture( sam, texCoord );
```

- ▶ Setzen der Textureinheit im OpenGL-Programm:

```
GLint slot = glGetUniformLocation( shaderPrg, "sam" );  
glUniform1i( slot, 17 );  
// verbinden mit Textureinheit: glActiveTexture(GL_TEXTURE0+17)
```

- ▶ Randbemerkung: es gibt auch „Images“ `image[1,2,3]D`
 - ▶ normalerweise für Berechnungen/als Lookup-Tabelle
 - ▶ unterstützt Lese und Schreibzugriff, Atomics, Memory Barrier, ...

Texturfunktionen im Shader



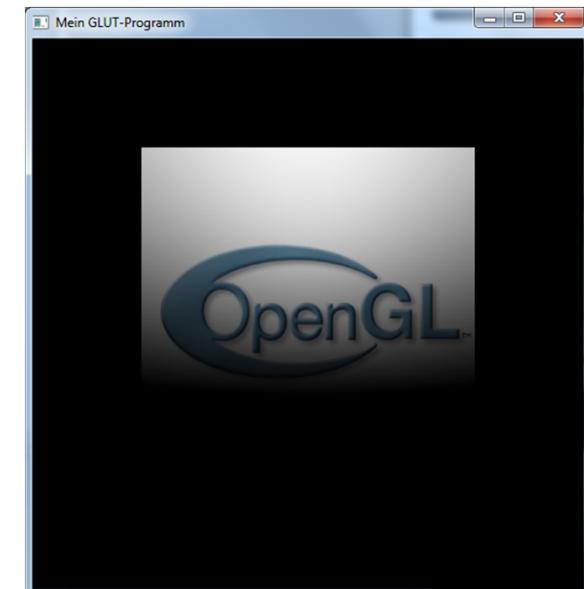
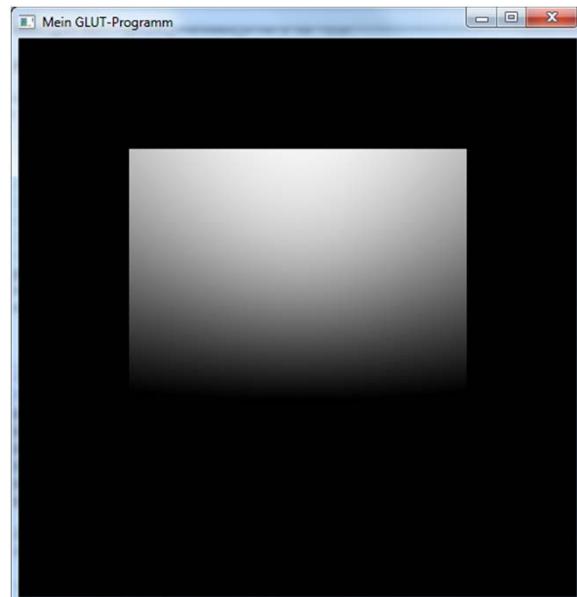
- ▶ **vec4 texture[Proj|Lod|Offset] (sampler, coord, ...)**
 - ▶ gibt den **interpolierten** Farbwert für Position **coord** zurück
 - ▶ **Proj** mit perspektivischer Division (geteilt durch **coord.w**)
 - ▶ **Lod** Auslesen einer explizit angegebenen Mipmap-Stufe
 - ▶ **Offset** mit zusätzlichem Offset (in Texels) auf Koordinaten
 - ▶ **Grad** mit explizit gegebenen Gradienten
 - ▶ ...und Kombinationen davon...
- ▶ **vec4 texelFetch[Offset] (sampler, coord, ...)**
 - ▶ Auslesen eines Texels mit Integerkoordinaten (ohne Interpolation)
- ▶ Hilfsfunktionen
 - ▶ **{int|ivec[2,3]} textureSize (sampler, lod)**
 - ▶ liefert einen Vektor mit den Dimensionen der Textur
 - ▶ **vec2 textureQueryLod (sampler, coord)**
 - ▶ Informationen zur Mipmap-Stufe, die verwendet werden würde

Texturierung im Fragment Shader: Beispiel



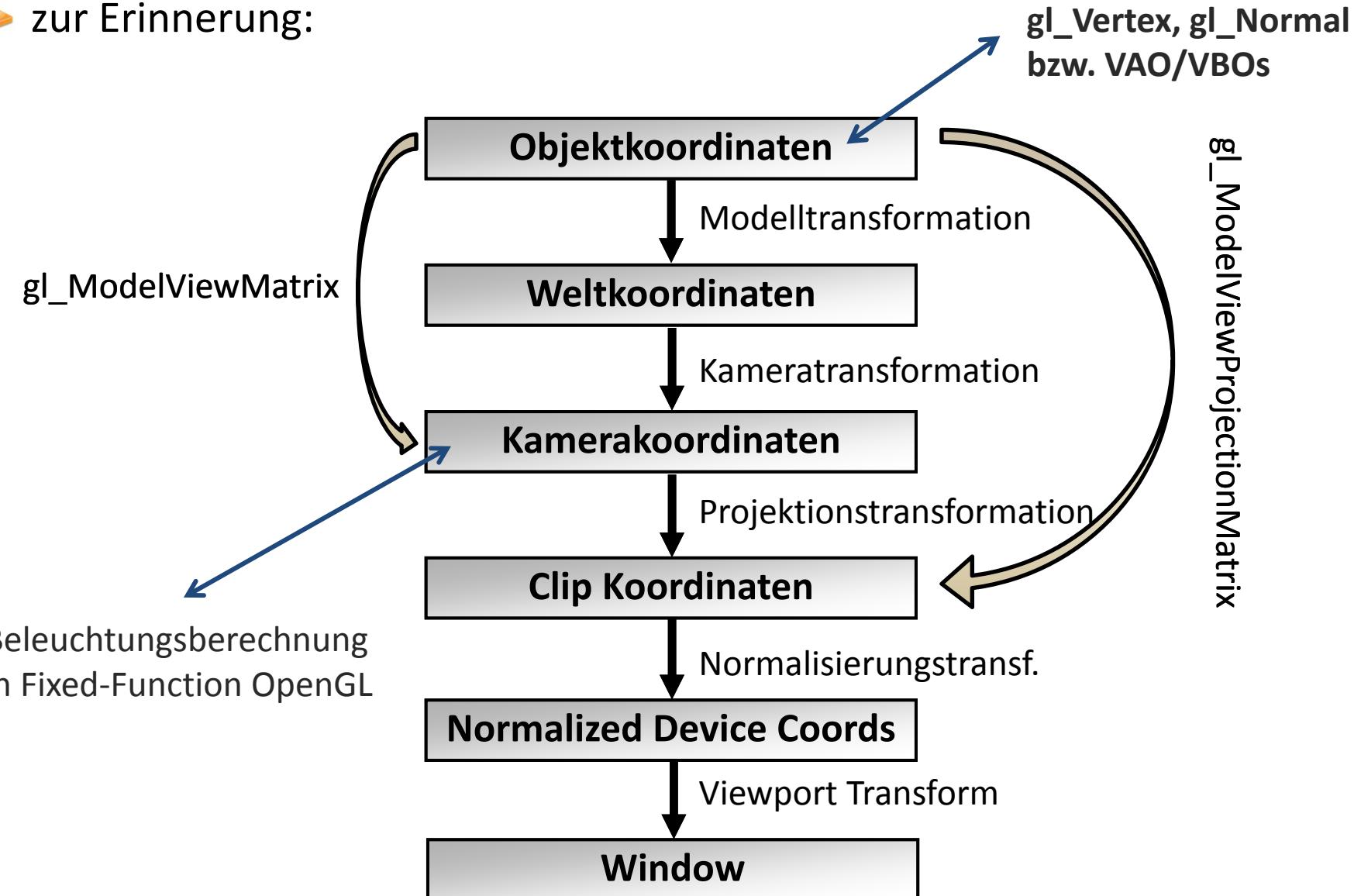
```
#version 410
uniform sampler2D tex;
in vec2 texCoord;

void main () {
    vec3 ogl = texture( tex, texCoord ).rgb;
    // [Beleuchtung aus Beispiel]
    out_frag_color = vec4( kd * ogl.rgb, 1.0 );
}
```



Nochmal: Transformationspipeline

► zur Erinnerung:



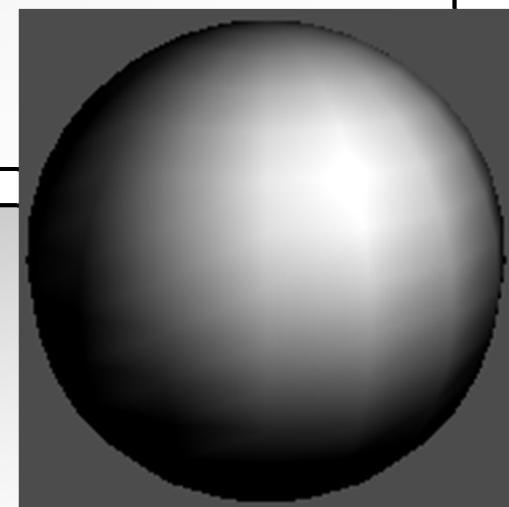
Mehr Infos:

<http://www.opengl.org/resources/faq/technical/viewing.htm>

Nochmal: GLSL 3.x/4.x Diffuse Beleuchtung



```
uniform mat4 matrixMVP, matrixMV, matrixNrml;  
uniform vec3 lightSourcePos;  
in vec4 in_position;  
in vec3 in_normal;  
out vec4 color;  
void main() {  
    gl_Position = matrixMVP * in_position;  
    // Beleuchtungsberechnung in Kamerakoordinaten  
    vec3 P = vec3( matrixMV * in_position );  
    vec3 N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );  
    vec3 L = normalize( lightSourcePos - P );  
    color = vec4( max( 0.0, dot( L, N ) ) );  
}
```



```
in vec4 color;  
out vec4 out_color;  
void main() {  
    out_color = color;  
}
```

Nochmal: GLSL 3.x/4.x „Per-Pixel“ Beleuchtung



```
uniform mat4 matrixMVP, matrixMV, matrixNrml;  
uniform vec3 lightSourcePos;  
in vec4 in_position;  
in vec3 in_normal;  
out vec3 L, N;  
void main() {  
    gl_Position = matrixMVP * in_position;  
    vec3 P = vec3( matrixMV * in_position );  
    N = vec3( matrixNrml * vec4( in_normal, 0.0 ) );  
    L = lightSourcePos - P;  
}
```

```
in vec3 L, N;  
out vec4 out_color;  
void main() {  
    float kd = max( 0.0, dot( normalize(L),  
                               normalize(N) ) );  
    out_color = vec4( kd );  
}
```



Wdh. GLSL Funktionen



- ▶ wie C-Programme können Shader durch Funktionen strukturiert werden
 - ▶ obligatorisch ist die Funktion: **void main()**
- ▶ Definition der Parameter für eigene Funktionen
 - ▶ **in**: es handelt sich um einen Eingabe-Parameter
 - ▶ **out**: Rückgabewert (auch möglich: **return**, dann aber nur eine Variable)
 - ▶ **inout**: Eingabe und gleichzeitig Rückgabewert
 - ▶ bei keiner Angabe wird **in** angenommen

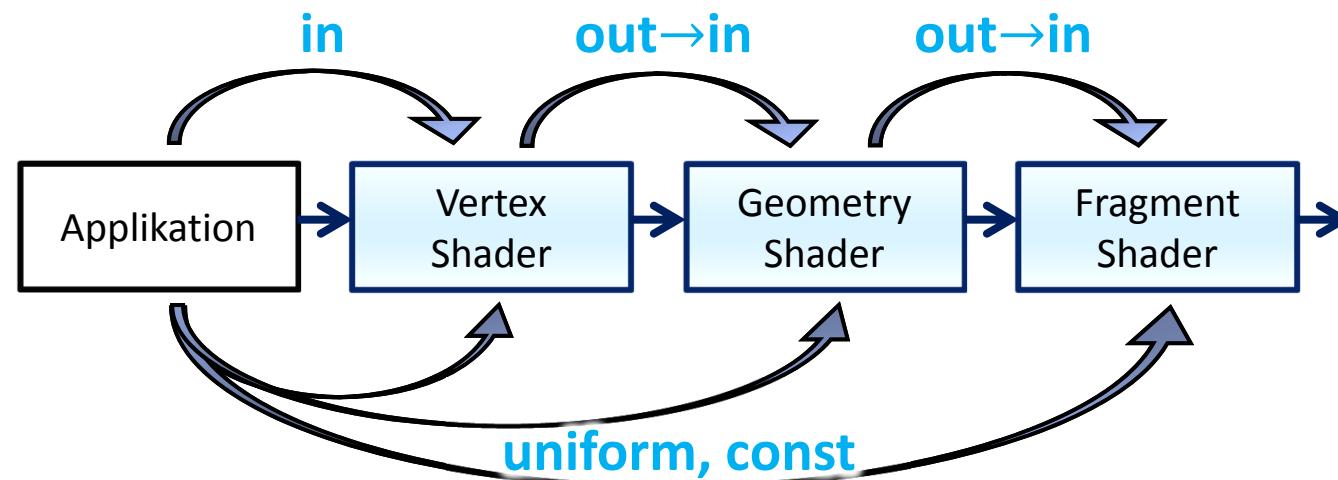
```
void computeLighting( in vec3 lightPos, in vec3 normal,
                      out float diffuse, out float specular ) {
    ...
}
```

- ▶ Überladen von Funktionen ist möglich
- ▶ Rekursion ist nicht spezifiziert, weil es keinen Stack auf GPUs gibt
 - ▶ wird Rekursion benötigt, muss ein selbstverwalteter Stack verwendet werden

Wdh. GLSL Qualifiers (OpenGL 3.x, 4.x)

Definition des Verwendungszwecks von Variablen

- ▶ **const**: Konstante, zur Compile-Zeit festgelegt
- ▶ **uniform**: read-only
 - ▶ globale Variablen, die pro Aufruf zum Zeichnen konstant bleiben
 - ▶ werden von der Applikation gesetzt (z.B. eine Transformationsmatrix, Position von Kamera und Lichtquelle, ...)
- ▶ **in**: Eingabe-Attribute des aktuellen Shaders
 - ▶ Attribute eines Vertex, z.B. Koordinate, Normale, Farbe etc.
 - ▶ oder Ausgabe eines vorherigen Shaders
- ▶ **out**: Ausgabe-Attribute des aktuellen Shaders



Nochmal: GLSL 3.x/4.x „Per-Pixel“ Beleuchtung



```
uniform mat4 matrixMVP, matrixMV, matrixNrml;  
uniform vec3 lightSourcePos;  
in vec4 in_position; ←  
in vec3 in_normal; ←  
out vec3 L, N;  
void main() {  
    gl_Position = matrixMVP * in_position;  
    vec3 P = vec3( matrixMV * in_position );  
    N = vec3( matrixNrml * vec4( in_normal, 0.0 ) );  
    L = lightSourcePos - P;  
}
```

```
in vec3 L, N;  
out vec4 out_color;  
void main() {  
    float kd = max( 0.0, dot( normalize(L),  
                               normalize(N) ) );  
    out_color = vec4( kd );  
}
```

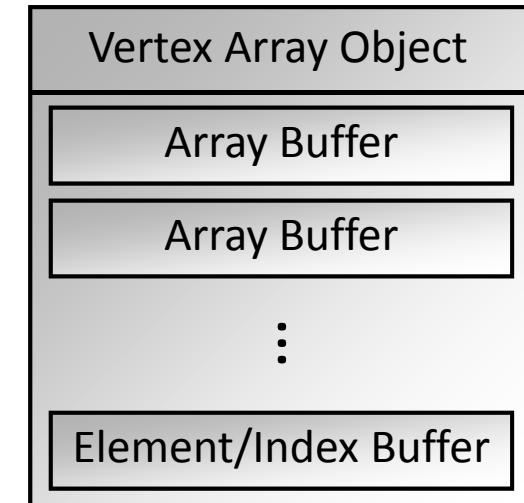


OpenGL Buffer Objects für Geometry



Überblick Vertex Buffer Objects, Vertex Array Objects

- ▶ wir betrachten als nächstes **Buffer Objects** die man benötigt um **Geometrie** zu speichern und zu zeichnen
- ▶ wir betrachten zwei Arten von Puffern
 - ▶ **Vertex Buffer (`GL_ARRAY_BUFFER`):**
Speichern von Vertizes und Vertex-Attributen
 - ▶ **Index Buffer (`GL_ELEMENT_ARRAY_BUFFER`):**
Speichern von Indizes,
„welche Vertizes bilden ein Dreieck“
- ▶ i.d.R. speichern wir Vertex-Positionen, -Farben, -Texturkoordinaten und optional eine Index-Liste
 - ▶ diese werden durch ein **Vertex Array Object** zusammengefasst
- ▶ **Anmerkung:** alle OpenGL Setups sind furchtbar... deswegen: einmal verstehen, einmal als C++-Klasse implementieren, nie mehr anschauen



GLUT: Beispiel – OpenGL 3.x/4.x



```
// (statische) Vertex-Daten
GLfloat vertices[] = {
    1.0f, 3.2f, 0.0f,
    ...
};

// (statische) Normalen
GLfloat normals[] = {
    1.0f, 0.0f, 0.0f,
    0.707f, 0.0f, 0.707f,
    ...
};

// "Handles" für Buffer-Objects
// vbo = Vertex Buffer
// nbo = Normal Buffer (1 Normale für jeden Vertex)
// vao = Vertex Array Object
GLuint vbo, nbo, vao;
```

GLUT: Beispiel – OpenGL 3.x/4.x



```
void initGeometry( void ) {
    glGenVertexArrays( 1, &vao ); // “Vertex Array Object”:
    glBindVertexArray( vao ); // gemeinsames Zustandsobjekt

    glGenBuffers( 1, &vbo ); // Vertex-Buffer
    glBindBuffer( GL_ARRAY_BUFFER, vbo );
    glBufferData( GL_ARRAY_BUFFER,
                  sizeof( GLfloat ) * NUM_VERT_COMPONENTS * NUM_VERTS,
                  vertices, GL_STATIC_DRAW );

    GLint attrLoc = glGetAttribLocation( shaderPrg, “in_position” );
    glEnableVertexAttribArray( attrLoc );
    glVertexAttribPointer( attrLoc, 3, GL_FLOAT, GL_FALSE, 0, NULL );

    glGenBuffers( 1, &nbo ); // Normal-Buffer
    glBindBuffer( GL_ARRAY_BUFFER, nbo );
    ...
    attrLoc = glGetAttribLocation( shaderPrg, “in_normal” ); ...

    glBindVertexArray( 0 ); // Zustandsobjekt abkoppeln
}
```

GLUT: Beispiel – OpenGL 3.x/4.x



```
void initGeometry( void ) {
    glGenVertexArrays( 1, &vao ); // "Vertex Array Object":
    glBindVertexArray( vao );      // gemeinsames Zustandsobjekt

    glGenBuffers( 1, &vbo );        // Vertex-Buffer
    glBindBuffer( GL_ARRAY_BUFFER, vbo );
    glBufferData( GL_ARRAY_BUFFER,
                  sizeof( GLfloat ) * NUM_VERT_COMPONENTS * NUM_VERTS,
                  vertices, GL_STATIC_DRAW );

    GLint attrLoc = glGetAttribLocation( shaderPrg, "in_position" );
    glEnableVertexAttribArray( attrLoc );
    glVertexAttribPointer( attrLoc, 3, GL_FLOAT, GL_FALSE, 0, NULL );

    glGenBuffers( 1, &nbo );        // Normal-Buffer
    glBindBuffer( GL_ARRAY_BUFFER, nbo );
    ...

    attrLoc = glGetAttribLocation( shaderPrg, "in_normal" ); ...

    glBindVertexArray( 0 );        // Zustandsobjekt abkoppeln
}
```

GLUT: Beispiel – OpenGL 3.x/4.x



```
void initGeometry( void ) {  
    glGenVertexArrays( 1, &vao ); // “Vertex Array Object”:  
    glBindVertexArray( vao ); // gemeinsames Zustandsobjekt
```

```
    glGenBuffers( 1, &vbo ); // Vertex-Buffer  
    glBindBuffer( GL_ARRAY_BUFFER, vbo );  
    glBufferData( GL_ARRAY_BUFFER,  
        sizeof( GLfloat ) * NUM_VERT_COMPONENTS * NUM_VERTS,  
        vertices, GL_STATIC_DRAW );
```

```
    GLint attrLoc = glGetUniformLocation( shaderPrg, “in_position” );  
    glEnableVertexAttribArray( attrLoc );  
    glVertexAttribPointer( attrLoc, 3, GL_FLOAT, GL_FALSE, 0, NULL );  
  
    glGenBuffers( 1, &nbo ); // Normal-Buffer  
    glBindBuffer( GL_ARRAY_BUFFER, nbo );  
    ...  
    attrLoc = glGetUniformLocation( shaderPrg, “in_normal” ); ...  
  
    glBindVertexArray( 0 ); // Zustandsobjekt abkoppeln  
}
```

GLUT: Beispiel – OpenGL 3.x/4.x



```
void initGeometry( void ) {
    glGenVertexArrays( 1, &vao ); // “Vertex Array Object”:
    glBindVertexArray( vao ); // gemeinsames Zustandsobjekt

    glGenBuffers( 1, &vbo ); // Vertex-Buffer
    glBindBuffer( GL_ARRAY_BUFFER, vbo );
    glBufferData( GL_ARRAY_BUFFER,
        sizeof( GLfloat ) * NUM_VERT_COMPONENTS * NUM_VERTS,
        vertices, GL_STATIC_DRAW );

    GLint attrLoc = glGetAttribLocation( shaderPrg, “in_position” );
    glEnableVertexAttribArray( attrLoc );
    glVertexAttribPointer( attrLoc, 3, GL_FLOAT, GL_FALSE, 0, NULL );

    glGenBuffers( 1, &nbo ); // Normal-Buffer
    glBindBuffer( GL_ARRAY_BUFFER, nbo );
    ...

    attrLoc = glGetAttribLocation( shaderPrg, “in_normal” ); ...

    glBindVertexArray( 0 ); // Zustandsobjekt abkoppeln
}
```

GLUT: Beispiel – OpenGL 3.x/4.x

```
void initGeometry( void ) {  
    glGenVertexArrays( 1, &vao ); // "Vertex Array Object":  
    glBindVertexArray( vao );  
  
    uniform mat4 ...;  
    glG...  
    glG...  
    glB...  
    glB...  
    glB...  
    void main() { ... }  
};  
// ger...  
// standsobjekt  
Attribut Location:  
Index eines  
„Eingaberegisters“  
* NUM_VERTS,  
    _DRAW );
```

```
GLint attrLoc = glGetAttribLocation( shaderPrg, "in_position" );  
	glEnableVertexAttribArray( attrLoc );  
	glVertexAttribPointer( attrLoc, 3, GL_FLOAT, GL_FALSE, 0, NULL );
```

```
glGenBuffers( 1, &nbo );           // Normal-Buffer  
	glBindBuffer( GL_ARRAY_BUFFER, nbo );  
...  
attrLoc = glGetAttribLocation( shaderPrg, "in_normal" ); ...  
  
	glBindVertexArray( 0 );          // Zustandsobjekt abkoppeln  
}
```

GLSL 3.x/4.x „Per-Pixel“ Beleuchtung

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos; ←
in  vec4 in_position; ←
in  vec3 in_normal; ←
out vec3 L, N;
void main() {
    gl_Position = matrixMVP * in_position;
    vec3 P = vec3( matrixMV * in_position );
    N = vec3( matrixNrml * vec4( in_normal, 0.0 ) );
    L = vec3( lightSourcePos ) - P;
}
```

```
in vec3 L, N;
out vec4 out_color;
void main() {
    float kd = max( 0.0, dot( normalize(L),
                                normalize(N) ) );
    out_color = vec4( kd );
}
```



GLUT: Beispiel – OpenGL 3.x/4.x



```
#include "glm/glm.hpp" // OpenGL Mathematics Bibliothek

glm::mat4x4 matM, matV, matP, matMV, matMVP, matNrml;
glm::vec3 camPosition, camTarget, camUp;

// Aufruf bei Änderung der Fenstergröße
void resize( int w, int h ) {
    matP =
        glm::perspective( 45.0f, (float)w / (float)h, 0.1f, 40.0f );
    matV = glm::lookAt( camPosition, camTarget, camUp );
    matM = ...;

    matMV = matV * matM;
    matMVP = matP * matV * matM;
    matNrml = glm::transpose( glm::inverse( matM ) );

    glViewport( 0, 0, (GLsizei)w, (GLsizei)h );
}
```

GLUT: Beispiel – OpenGL 3.x/4.x



```
// Rendering (=Zeichnen) eines Bildes
void display( void ) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram( shaderPrg );

    // Setzen einer Uniform-Variable
    GLuint matMVPLoc =
        glGetUniformLocation( shaderPrg, "matrixMVP" )
    glUniformMatrix4fv( matMVPLoc, 1, GL_FALSE, matMVP );
    ...

    // Zeichnen mit dem Vertex Array Object
    glBindVertexArray( vao );
    // DrawArrays und GL_TRIANGLES: je 3 Vertizes aus dem VBO
    // bilden ein Dreieck
    glDrawArrays( GL_TRIANGLES, 0, NUM_VERTICES );
    glutSwapBuffers();
}
```

OpenGL Vertex Buffers



Überblick OpenGL-Puffer für Vertex/Index-Buffer

- ▶ Erzeugen

```
glGenBuffers( 1, &vbo )
 glBindBuffer( target, vbo );
 glBufferData( target, size, data, usage )
```

- ▶ *target* ist **GL_ARRAY_BUFFER** für Vertex- und Attributdaten oder **GL_ELEMENT_ARRAY_BUFFER** für Indizes
- ▶ *size* ist Größe des benötigten Speichers in Bytes
- ▶ *usage* ist
 - ▶ übertragene Daten werden...
 - GL_STREAM_...** einmal verwenden, dann ersetzt
 - GL_STATIC_...** werden oft verwendet, ohne sie zu ändern
 - GL_DYNAMIC_...** werden hin und wieder ersetzt
 - ▶ ...und die Daten stammen von...
 - ..._DRAW** der Applikation und werden an OpenGL übergeben
 - ..._READ** werden von OpenGL erzeugt und in das VBO kopiert
 - ..._COPY** werden von OpenGL erzeugt und zur Appl. übertragen

Vielfalt der Vertex-Daten



```
glBufferData(GL_ARRAY_BUFFER,  
             NUM_VERTICES * sizeof(GLfloat) * NUM_VERT_COMPONENTS,  
             vertices, // Zeiger  
             GL_DYNAMIC_DRAW );
```

Datentyp

- GLbyte - byte
- GLubyte - unsigned byte
- GLshort - short
- GLushort - unsigned short
- GLint - int
- GLuint - unsigned int
- GLfloat - float
- GLdouble - double

Anzahl der Komponenten

- 2 - (x,y)
- 3 - (x,y,z)
- 4 - (x,y,z,w)

Alternative: Speicher-Mapping



Mapping von Buffer Objects

- ▶ es ist auch möglich Buffer Objects zunächst nur anzulegen
 - ▶ dabei wird deren Größe bereits festgelegt,
aber keine Daten übergeben

```
// Buffer-Initialisierung mit Größe  
glBufferData( GL_ARRAY_BUFFER,  
    sizeof(GLfloat) * NUM_VERT_COMPONENTS * NUM_VERTICES, NULL,  
    GL_DYNAMIC_DRAW );
```

- ▶ zur Laufzeit wird der Puffer in den Hauptspeicher „eingebendet“:

```
// Daten setzen/aktualisieren  
void *m = glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );  
// <Speicheroperationen...>  
glUnmapBuffer( GL_ARRAY_BUFFER );
```

Index Buffers und Shared Vertex Repräsentation



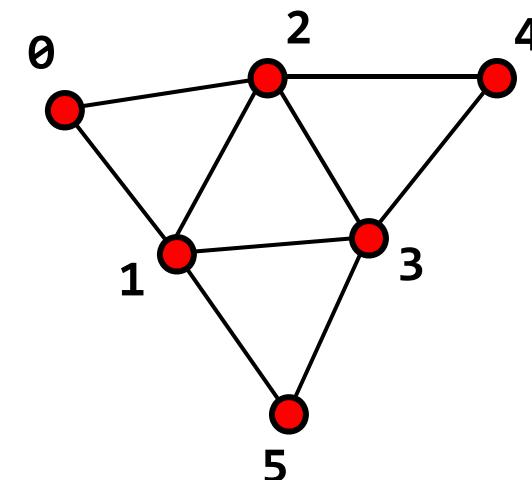
- ▶ **Indexed Face Set** oder **Shared Vertex** Darstellung des Dreiecksnetzes
 - ▶ verwende Array mit Vertizes (und weiteren Attributen)
 - ▶ verwende Index-Liste: welche Vertizes bilden zusammen ein Dreieck
 - ▶ Vorteile: weniger Daten (ein Vertex ist Eckpunkt mehrerer Dreiecke)

```
// Vertex-Daten
```

```
GLfloat vertices[] = {  
    1.0f, 3.2f, 0.0f, ...  
};
```

```
// Index-Daten
```

```
GLuint indices[NUM_INDICES] = {  
    0, 1, 2, 2, 1, 3, 3, 1, 5,...  
};
```



Index Buffers und Shared Vertex Repräsentation



```
// "Handles" für Buffer-Objects
GLuint vbo, ibo, vao;

void initGeometry( void ) {
    glGenVertexArrays( 1, &vao ); // gemeinsames Zustandsobjekt
    glBindVertexArray( vao );

    glGenBuffers( 1, &vbo );
    glBindBuffer( GL_ARRAY_BUFFER, vbo );
    ...

    glGenBuffers( 1, &ibo );      // Index-Buffer
    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, ibo );
    glBufferData( GL_ELEMENT_ARRAY_BUFFER,
        NUM_INDICES * sizeof( GLuint ),
        indices, GL_STATIC_DRAW );
    ...

    glBindVertexArray( 0 );      // Zustandsobjekt abkoppeln
}
```

Index Buffers und Shared Vertex Repräsentation



```
// Rendering (=Zeichnen) eines Bildes
void display(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram( shaderPrg );

    glBindVertexArray( vao );

    // DrawElements verwendet Indizes (DrawArrays tut das nicht!)
    glDrawElements( GL_TRIANGLES, NUM_INDICES,
                    GL_UNSIGNED_INT, NULL );

    glutSwapBuffers();
}
```

Vertex Cache

- ▶ GPUs speichern Ergebnisse der letzten verarbeiteten Vertizes
 - ▶ wird derselbe Vertex nochmal benötigt, wird auf den Vertex Cache zurückgegriffen
 - ▶ typische Vertex Cache Größe in etwa $N = 24$
(lässt sich für heutige GPUs nicht so explizit angeben)
- ▶ **Vertex Caching funktioniert nur mit indizierten Vertizes**
 - ▶ d.h. mit **glDrawElements**, aber nicht mit **glDrawArrays**
 - ▶ Identifikation der Vertizes über deren Indizes
- ▶ Beispiel:
 - ▶ Indizes: 0, 1, 2, 3, 2, 5, 1, 3, 4, 4, 2, 3, 6, 7, 8, ...
 - ▶ **Misses & Hits:** 0, 1, 2, 3, 2, 5, 1, 3, 4, 4, 2, 3, 6, 7, 8, ...
 - ▶ Anmerkung: es gibt Algorithmen, um die Reihenfolge der Dreiecke bzw. Vertizes für bestmögliches Caching zu optimieren

3D Modelle



- ▶ Export/Import der Daten über Binär- oder ASCII-Formate

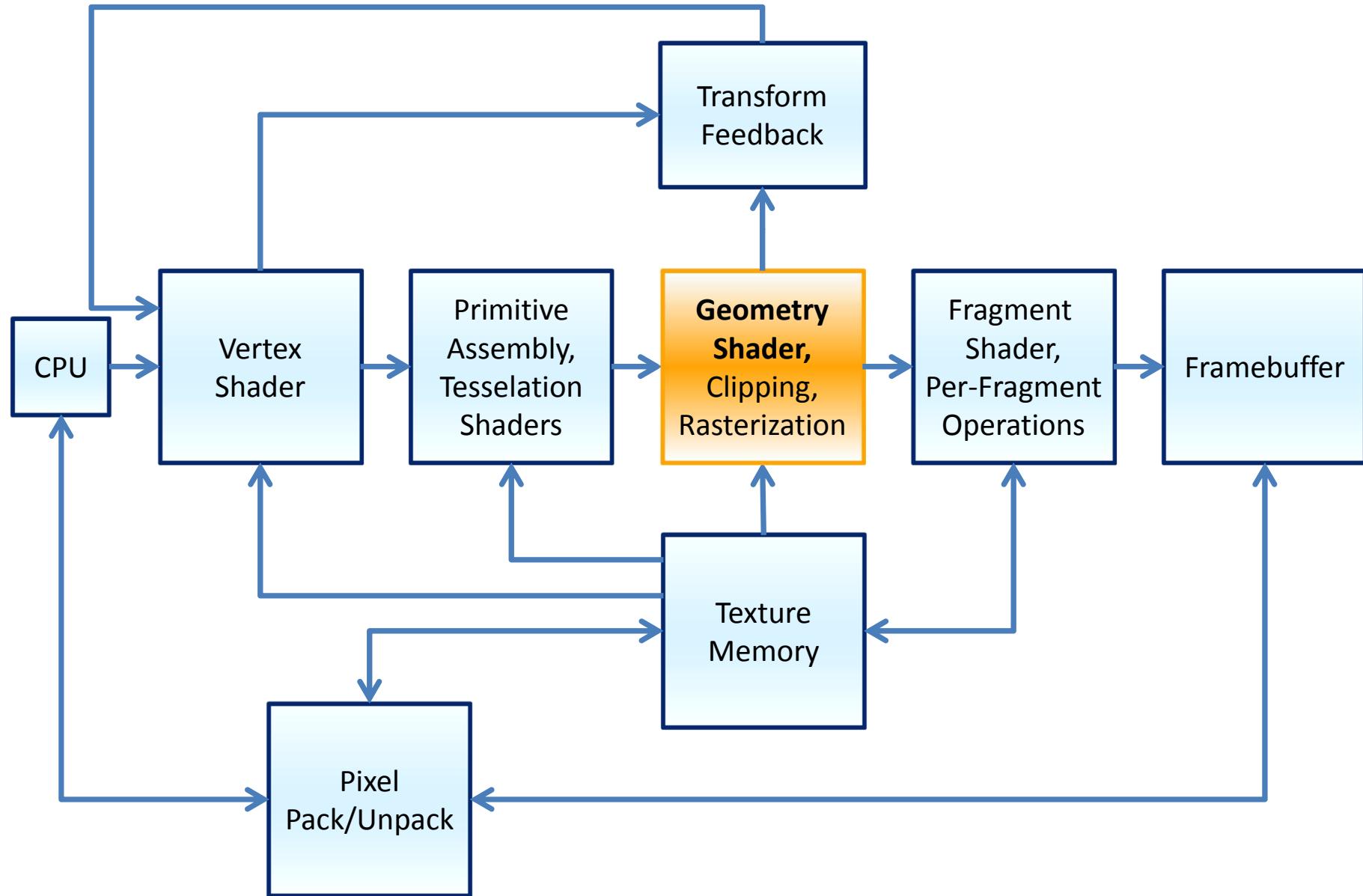
- ▶ Beispiel: Wavefront-OBJ Format (*.obj)

```
mtllib materialdefinition.mtl
v -1.0 0.0 -1.0      # Vertex-Position
vn 0.0 1.0 0.0        # Normale
vt 0.2 0.8            # Texturkoordinate
...
usemtl MeinMaterial
f 1 2 3                # Definition eines Dreiecks
f 1/1/2 2/2/1 3/2/1    # Definition eines Dreiecks mit...
                        # ...Normalen-/TexCoord-Indizes
...
```

- ▶ Material-Definition (*.mtl)

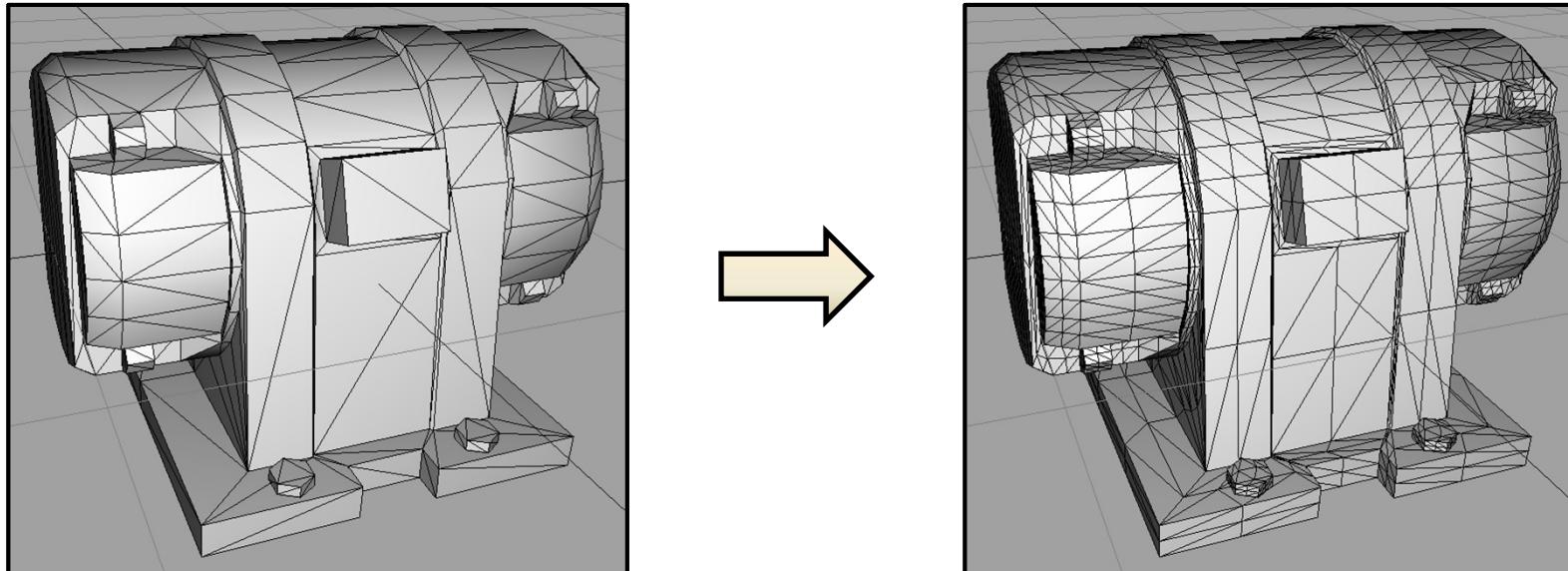
```
newmtl MeinMaterial
Kd 0.50 0.20 0.10
Ka 0.00 0.00 0.00
Ks 0.90 0.90 0.90
```

OpenGL(4.2+)-Pipeline



Geometry Shader

- ▶ Geometry Shader bearbeiten **Primitive** (Punkt, Linie, Dreieck) und können Primitive vervielfachen, entfernen oder umwandeln
- ▶ Ausführung „direkt“ nach dem Vertex Shader
- ▶ kann aber nicht beliebig viel Geometrie ausgeben: insgesamt `glGet*(MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS, &anzahl)` Komponenten (GeForce 500: 1024)
- ▶ Beispiel: Unterteilung von Dreiecken



Geometry Shader – Bsp. Shader reicht nur durch



```
// deklariere die Ein/Ausgabe eines Aufrufs: je 1 Dreieck(streifen)
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

// Eingabe der selbstdefinierten Vertex-Attribute (out vec3 vertexColorVS)
// hier: Array von Farbwerten, Ausgabe des Vertex Shader
in vec3 vertexColorVS[]; // Aufruf pro Primitiv, ergo ein Array

out vec3 vertexColorGS; // Ausgabe: 1 Attribut (Farbe) pro neuem Vertex

void main() {
    for ( int i = 0; i < gl_in.length(); i++ ) {
        gl_Position = gl_in[ i ].gl_Position;
        vertexColorGS = vertexColorVS[ i ];
        EmitVertex(); // Vertex ist fertig ⇒ ausgeben
    }
    EndPrimitive(); // Primitiv ist fertig: ausgeben
}
```

```
in vec3 vertexColorGS;      out vec4 out_color;
void main() { out_color = vec4( vertexColorGS.rgb, 1.0 ); }
```

Attribute in der modernen OpenGL-Pipeline



Geometry Shader

- ▶ Anzahl der Elemente pro Eingabeprimitiv: **gl_in.length()**
 - ▶ nicht in allen OpenGL-Versionen vorhanden, aber ohnehin durch den Eingabeprimitivtyp festgelegt: **layout(primitive type) in;**
 - ▶ Primitive: points, lines, lines_adjacency, triangles, triangles_adjacency
- ▶ es sind folgende Standardattribute definiert

```
gl_PerVertex {  
    vec4 gl_Position;  
    float gl_Points;           // für Punkt-Rendering  
    float gl_ClipDistance[];   // Abstand zu Clip-Ebenen  
} gl_in[];
```

- ▶ Zugriff in über **gl_in[].<attrib>** und **gl_out[].<attrib>**
- ▶ Konfiguration der Ausgabe **layout (triangle_strip, max_vertices = 3) out;**
 - ▶ Ausgabeprimitiv: points, line_strip, triangle_strip
 - ▶ Angabe von **max_vertices** ist optional, hilft bei der Optimierung

GLSL: Beleuchtung im Geometry Shader



```
// Vertex Shader
in vec4 in_position; in vec3 in_normal;
out vec3 vs_normal;
void main() {
    gl_Position = in_position; vs_normal = in_normal;
}
```

```
// Geometry Shader
in vec3 vs_normal[];
layout (triangle_strip, max_vertices = 3) out;

void main() {
    for ( int i = 0; i < gl_in.length(); i++ ) {
        gl_Position = matMVP * gl_in[i].gl_Position;
        ... // siehe Vertex Shader vorhin
        vertexColorGS = vec4( kd );
        EmitVertex();
    }
    EndPrimitive();
}
```

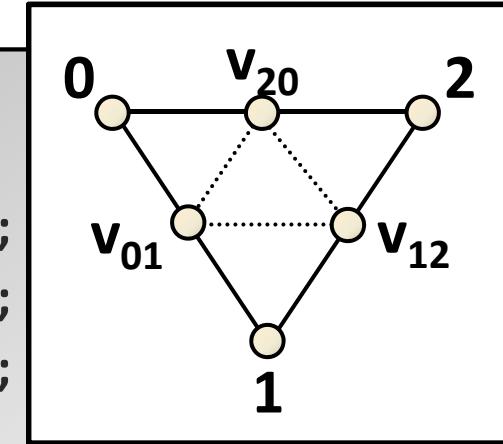
GLSL: Unterteilung im Geometry Shader



```
...
void main() {
    vec4 v01 = (gl_in[0].gl_Position + gl_in[1].gl_Position) * 0.5;
    vec4 v12 = (gl_in[1].gl_Position + gl_in[2].gl_Position) * 0.5;
    vec4 v20 = (gl_in[2].gl_Position + gl_in[0].gl_Position) * 0.5;

    gl_Position = matMVP * gl_in[0].gl_Position; EmitVertex();
    gl_Position = matMVP * v01; EmitVertex();
    gl_Position = matMVP * v20; EmitVertex();
    EndPrimitive();

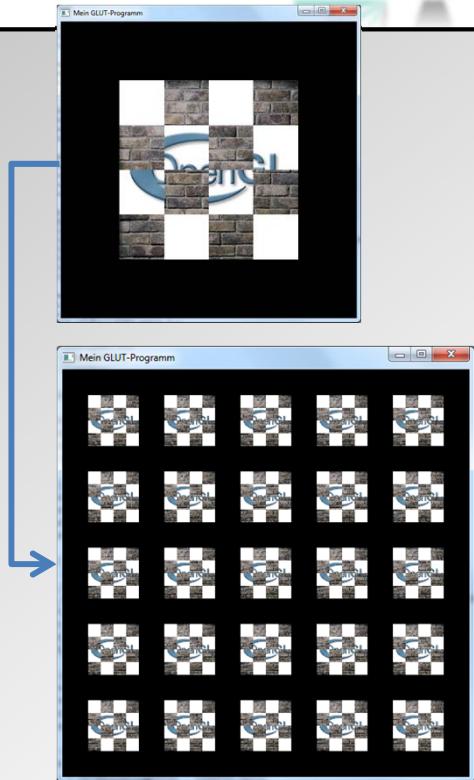
    gl_Position = matMVP * v01; EmitVertex();
    gl_Position = matMVP * v12; EmitVertex();
    gl_Position = matMVP * v20; EmitVertex();
    EndPrimitive();
    // etc... }
```



Geometry Shader: Instantiierung

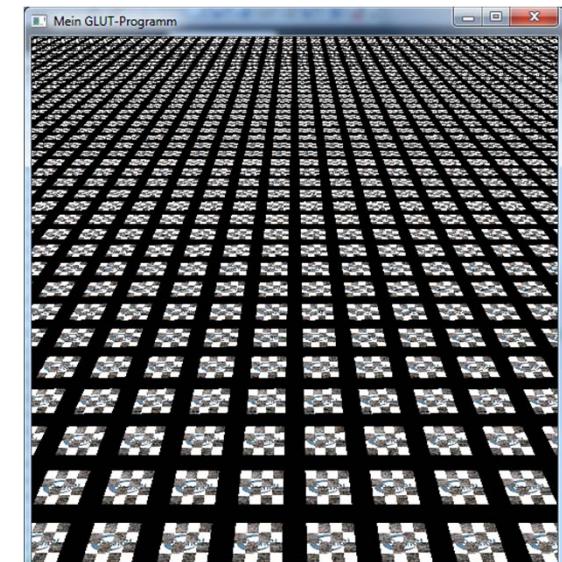
```
#version 410
layout( triangles, invocations = 25 ) in;
layout( triangle_strip, max_vertices = 3 ) out;
...
void main() {
    for ( int i = 0; i < gl_in.length(); i++ ) {
        vec4 p = gl_in[i].gl_Position;
        p.x += gl_InvocationID % 5;
        p.y += gl_InvocationID / 5;
        vec4 worldPos = matM * p;

        N = ( matNRML * vec4( N[i], 0.0 ) ).xyz;
        L = normalize( lightPos - worldPos.xyz );
        gl_Position = matP * matV * worldPos;
        EmitVertex();
    }
    EndPrimitive(); }
```



Instantiierung

- ▶ Die Zahl der Invocations in einem Geometry Shader ist begrenzt:
GL_MAX_GEOMETRY_SHADER_INVOCATIONS (GF500: 32)
- ▶ es ist auch möglich von Applikationsseite zu Instanziieren:
`glDrawArraysInstanced(GLenum mode, GLint first,
 GLsizei count, GLsizei primcount)`
`glDrawElementsInstanced(...)`
 - ▶ „wiederholt“ **primcount**-mal **glDrawArrays** (aber: kein API-Mehraufwand)
 - ▶ **primcount** darf beliebig groß sein
 - ▶ im **Vertex Shader** kann die Variable **gl_InstanceID** verwendet werden
(nicht verwechseln mit **gl_InvocationID**)
- ▶ meist werden die Instanzen auf Basis von
Daten aus einer Textur beeinflusst, z.B.
Position aus einer GPU-Partikelsimulation o.Ä.



Bemerkungen zu OpenGL Shader



- ▶ wie wir gesehen haben: ein Shader ist eine Art C-Programm Code für eine der programmierbaren Stufen der Grafik-Pipeline
 - ▶ Vertex-, Geometry- und Fragment-Shader: in dieser Vorlesung
 - ▶ Tessellation-Control und Tessellation-Evaluation Shader
 - ▶ Compute Shader (für GPGPU Berechnungen)
- ▶ jeder Shader-Typ ist für spezielle Aufgaben gedacht
 - ▶ trotzdem kann man einige Aufgaben an mehreren Stellen durchführen (man wählt am besten die dafür vorgesehenen Verarbeitungsstufe)
 - ▶ Operationen und Funktionen sind identisch
 - ▶ Semantik und Layout der Eingangs- und Ausgangsdaten variiert

Compute Shader (OpenGL 4.3+)



- ... ein Programm losgelöst von der Rendering-Pipeline

```
// Compute Shader
layout (local_size_x = 16, local_size_y = 16) in;

uniform image2D destTex;

void main() {
    ivec2 storePos = ivec2(gl_GlobalInvocationID.xy);

    vec4 color = ...;

    imageStore( destTex, storePos, color );
}
```

- Aufruf der Berechnung
`glDispatchCompute(nWorkGroupsX, nWorkGroupsY, nWorkGroupsZ);`

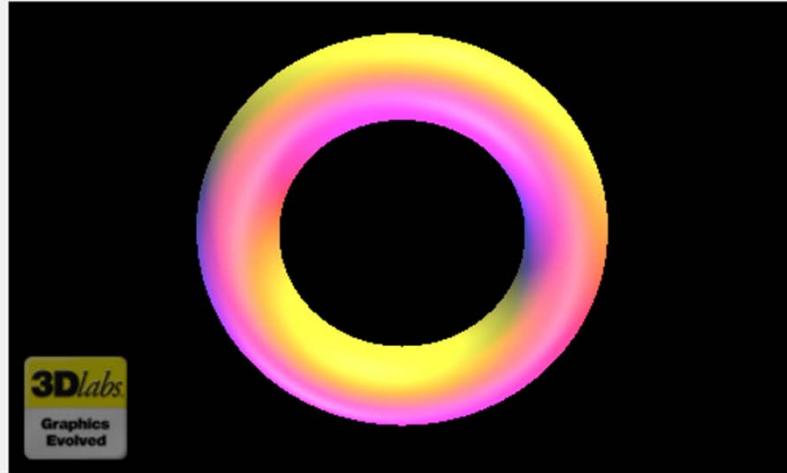
ShaderGen



3Dlabs GLSL ShaderGen

File View Model Help

Select GL Mode
 FIXED FUNCTIONALITY MODE EQUIVALENT SHADER MODE



Vertex Shader | Fragment Shader | InfoLog

1. GENERATE SHADERS | 2. COMPILE | 3. LINK | BUILD | CLEAR INFO LOG

LIGHT MATERIAL FOG | TEXTURE COORDINATE SET | TEXTURE ENVIRONMENT SET |

Lighting Parameters

glEnable/glDisable
 L0 L1 L2 L3 L4 L5 L6 L7 GL_LIGHTING GL_NORMALIZE GL_SEPARATE_SPECULAR_COLOR

Select Light
 L0 L1 L2 L3 L4 L5 L6 L7

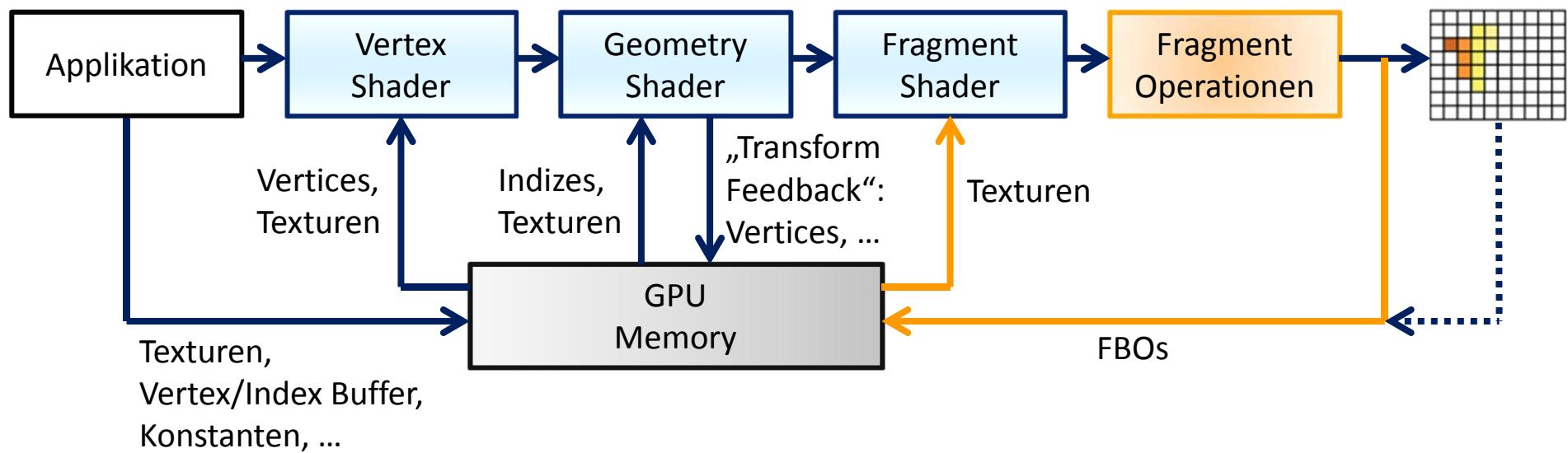
Selected Light Properties

| | | | | | | | |
|-------------|------------------|------------------|----------------|-----------------------|--------------|--------------------------|-----|
| GL_POSITION | 3.0.10.0.0.0.1.0 | GL_AMBIENT | [Color Swatch] | GL_SPOT_DIRECTION | 0.0,0.0,-1.0 | GL_CONSTANT_ATTENUATION | 1.0 |
| GL_SPECULAR | [Color Swatch] | GL_SPOT_EXPONENT | 0.0 | GL_LINEAR_ATTENUATION | 0.0 | GL_QUADRATIC_ATTENUATION | 0.0 |
| GL_DIFFUSE | [Color Swatch] | GL_SPOT_CUTOFF | 180.0 | | | | |

3Dlabs GLSL ShaderGen

Ausblick: Weitere OpenGL Puffer

- ▶ Pixel Buffer Objects (PBOs)
 - ▶ wie VBOs, aber für Pixeldaten/Bilder
 - ▶ effizienter Textur-Upload, asynchroner Transfer
- ▶ Uniform Buffer Objects (UBOs)
 - ▶ fassen Uniforms für GLSL Programme zusammen
 - ▶ → weniger OpenGL Aufrufe
- ▶ Frame Buffer Objects (FBOs)
 - ▶ erlaubt der Applikation „eigene“ Frame Buffer anzulegen

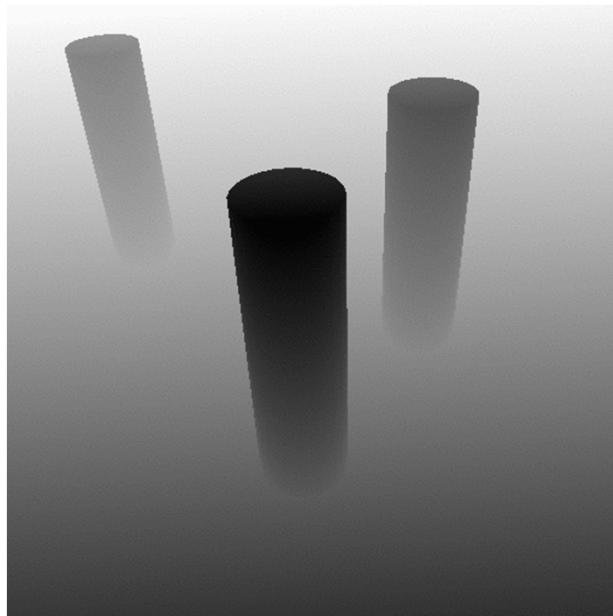


Shadow Mapping (Williams, SIGGRAPH'78)

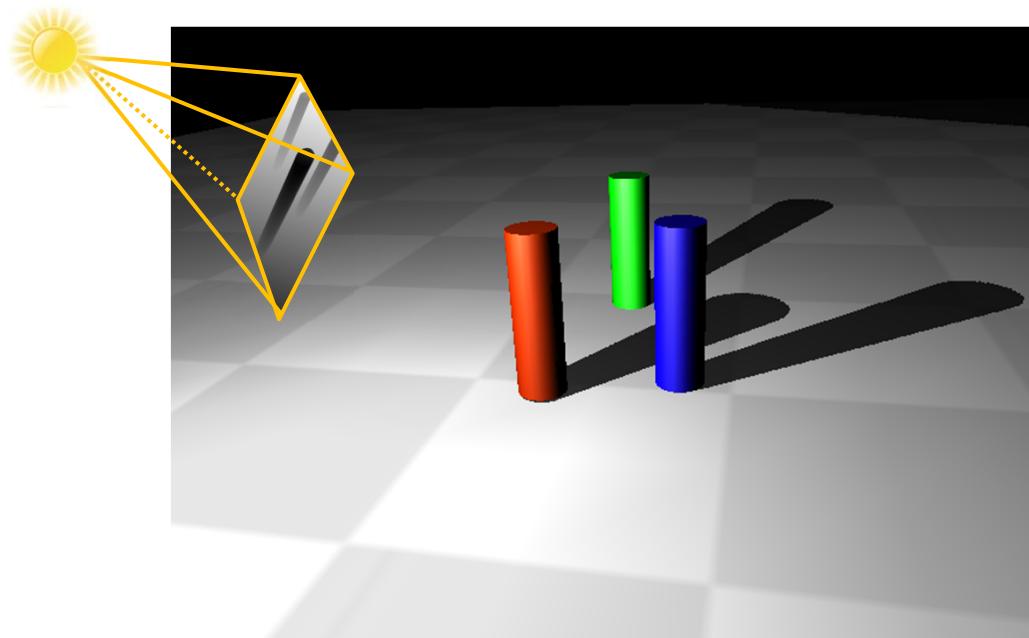


Shadow Mapping Grundidee

- ▶ eine Textur speichert – für viele Augstrahlen/Richtungen – wie weit die nahsten Oberflächen von der Lichtquelle entfernt sind
- ▶ wir betrachten hier nur Spotlights, d.h. wir ersetzen die Lichtquelle gedanklich durch eine perspektivische Kamera und berechnen/speichern den Tiefenpuffer des Bildes (links)



Shadow Map: Szene aus
der Sicht der Lichtquelle



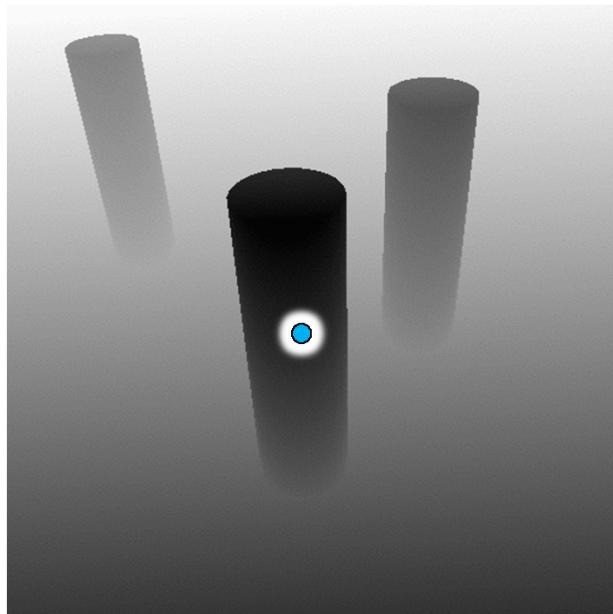
Szene aus der Sicht der Kamera

Shadow Mapping (Williams, SIGGRAPH'78)

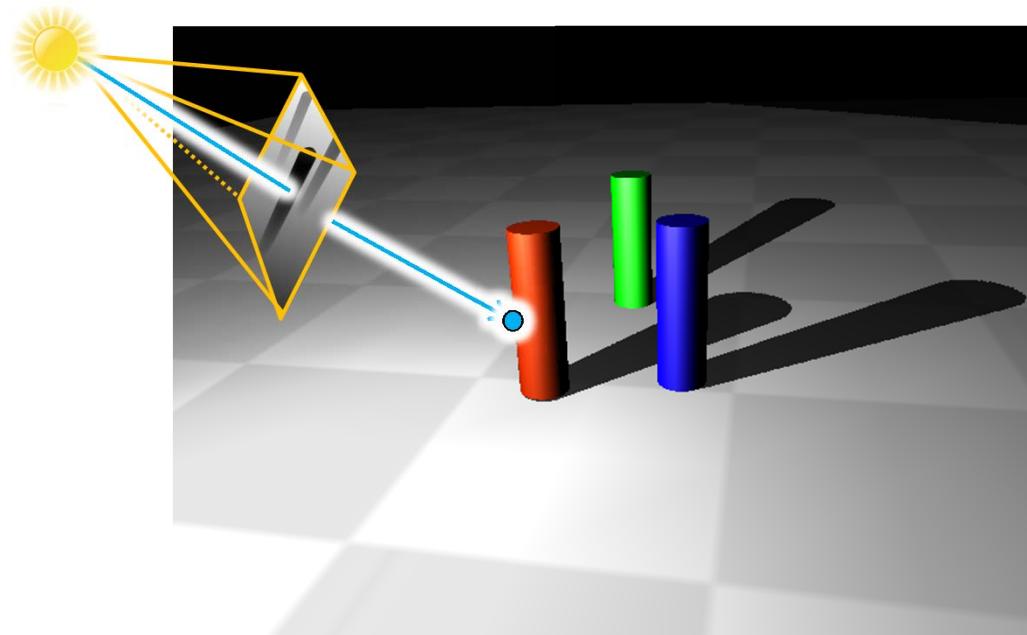


Shadow Mapping Grundidee

- ▶ eine Textur speichert – für viele Augstrahlen/Richtungen – wie weit die nahsten Oberflächen von der Lichtquelle entfernt sind
- ▶ beim Rendering des Kamerabildes (rechts):
 - ▶ jeder Punkt im Raum entspricht einer Richtung von der LQ
 - ▶ für diese Richtung lässt sich die Entfernung zur nahsten Oberfläche nachschlagen



Shadow Map: Szene aus
der Sicht der Lichtquelle



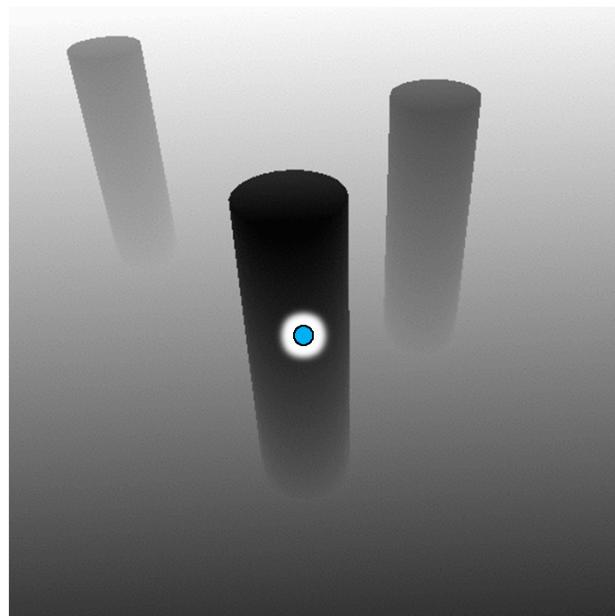
Szene aus der Sicht der Kamera

Shadow Mapping (Williams, SIGGRAPH'78)

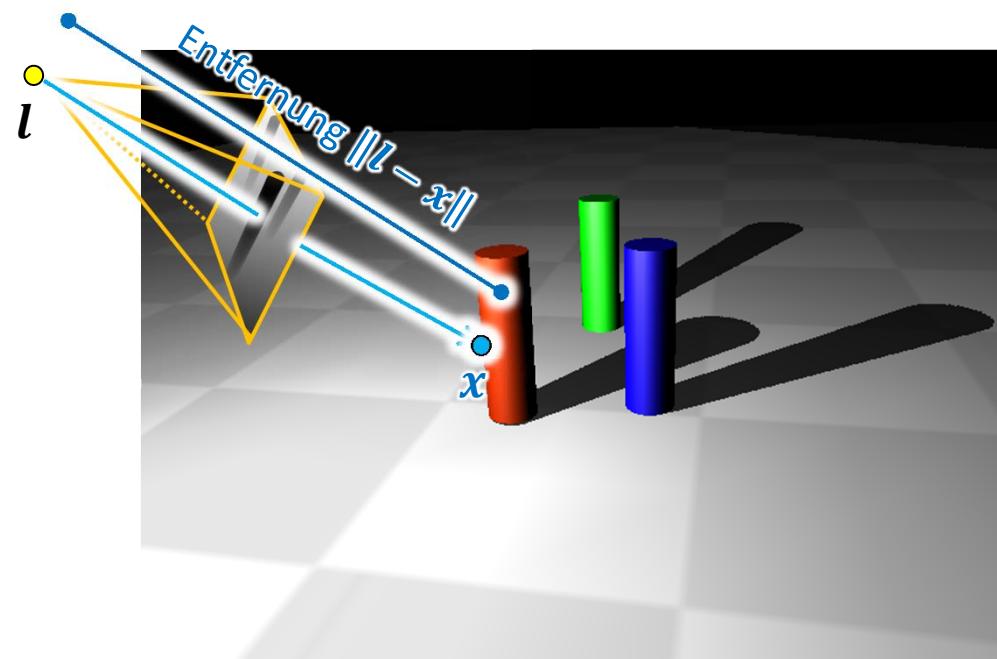


Shadow Mapping Grundidee

- ▶ der **blaue Punkt** befindet sich nicht im Schatten
- ▶ die in der Shadow Map gespeicherte Entfernung entspricht der tatsächlichen Entfernung



Shadow Map: Szene aus
der Sicht der Lichtquelle



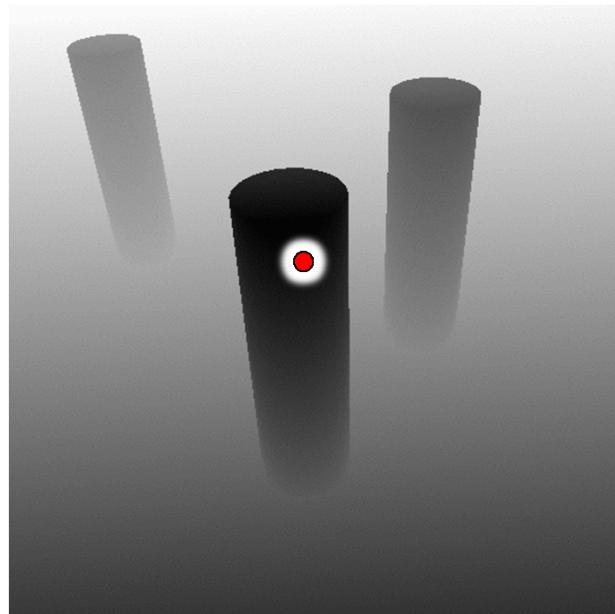
Szene aus der Sicht der Kamera

Shadow Mapping (Williams, SIGGRAPH'78)

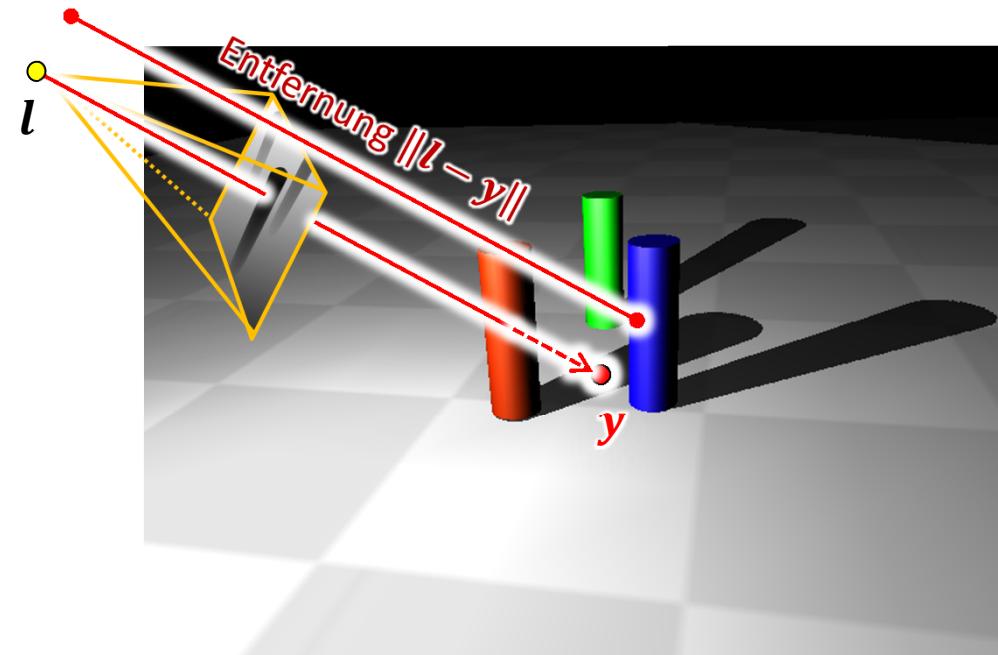


Shadow Mapping Grundidee

- ▶ der **rote Punkt** befindet sich im Schatten
- ▶ die in der Shadow Map gespeicherte Entfernung ist kleiner als seine tatsächliche Entfernung zur Lichtquelle



Shadow Map: Szene aus
der Sicht der Lichtquelle



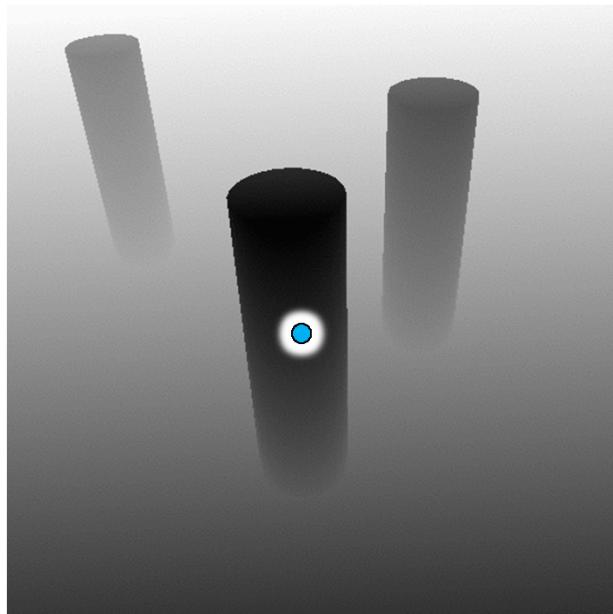
Szene aus der Sicht der Kamera

Shadow Mapping (Williams, SIGGRAPH'78)

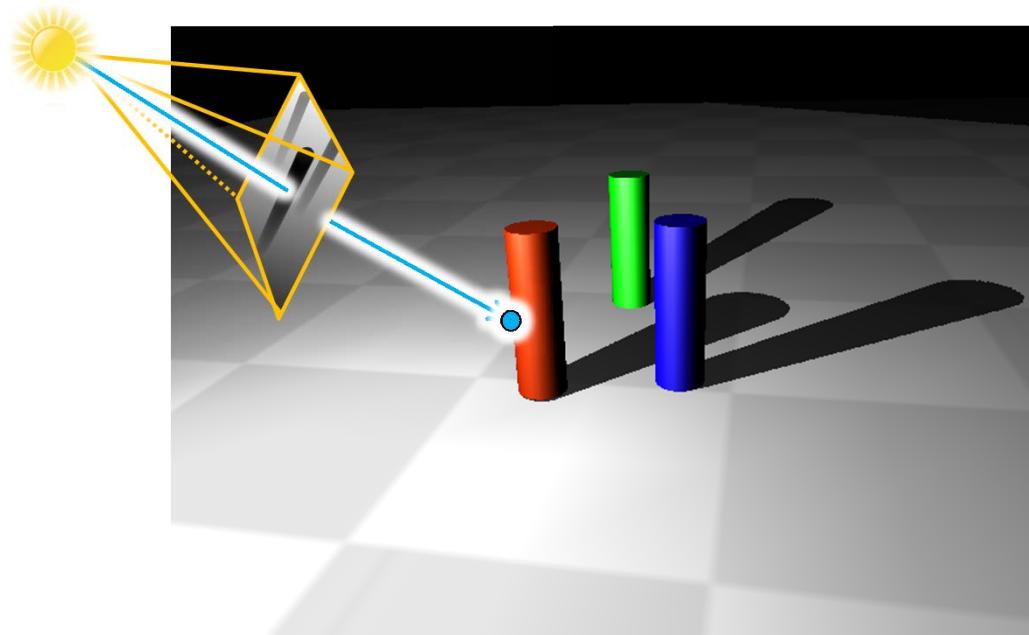


Shadow Mapping Grundidee

- ▶ erzeuge pro Lichtquelle eine Shadow Map → verwende sie während dem Rendering des Kamerabildes für den Schattentest
- ▶ es handelt sich um ein **Bildraum-Verfahren**: durch Rendering der Szene aus Sicht der LQ erhalten wir eine diskrete Abtastung der Flächen
- ▶ Anm. funktioniert eingeschränkt auch mit klassischem OpenGL

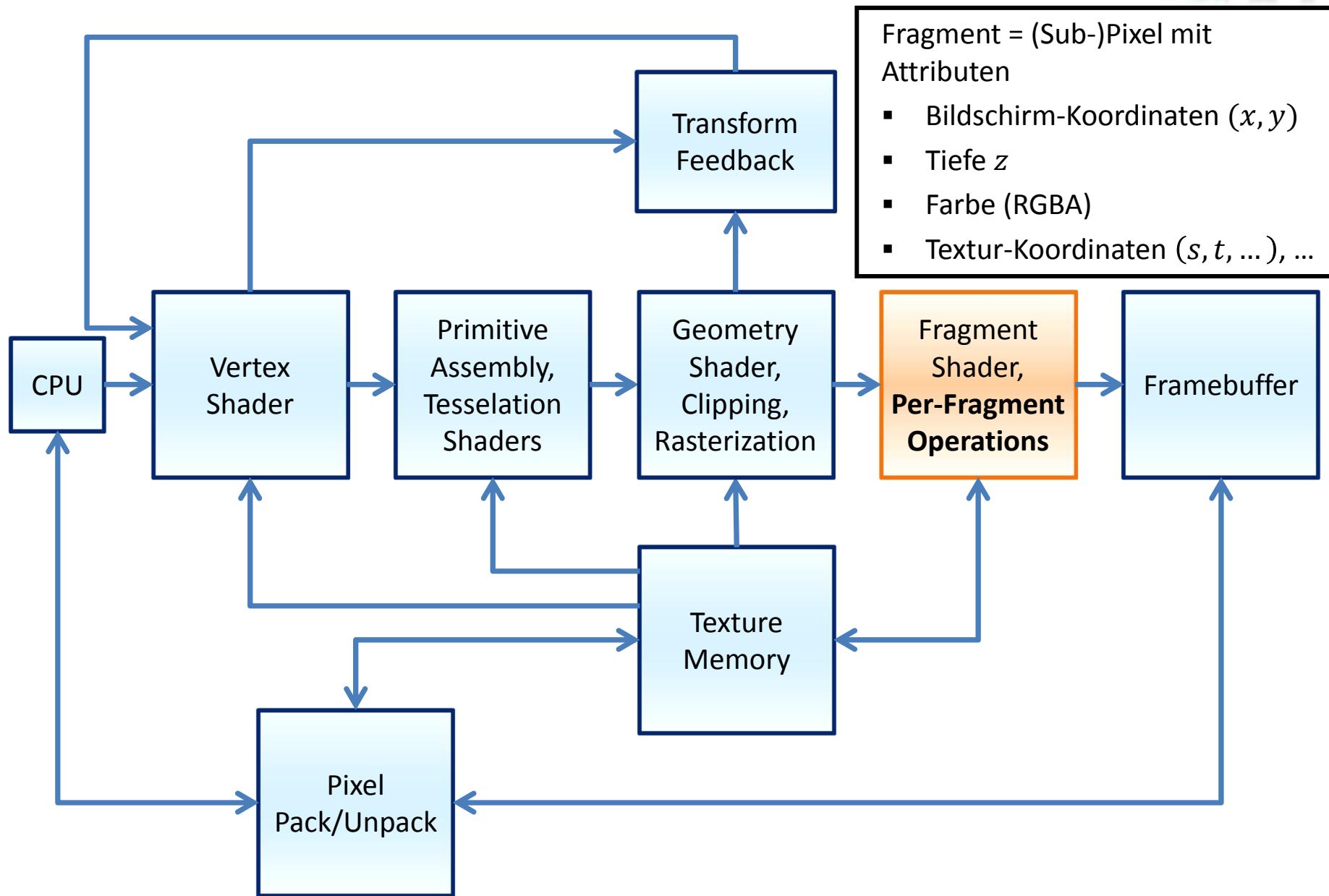


Shadow Map: Szene aus
der Sicht der Lichtquelle



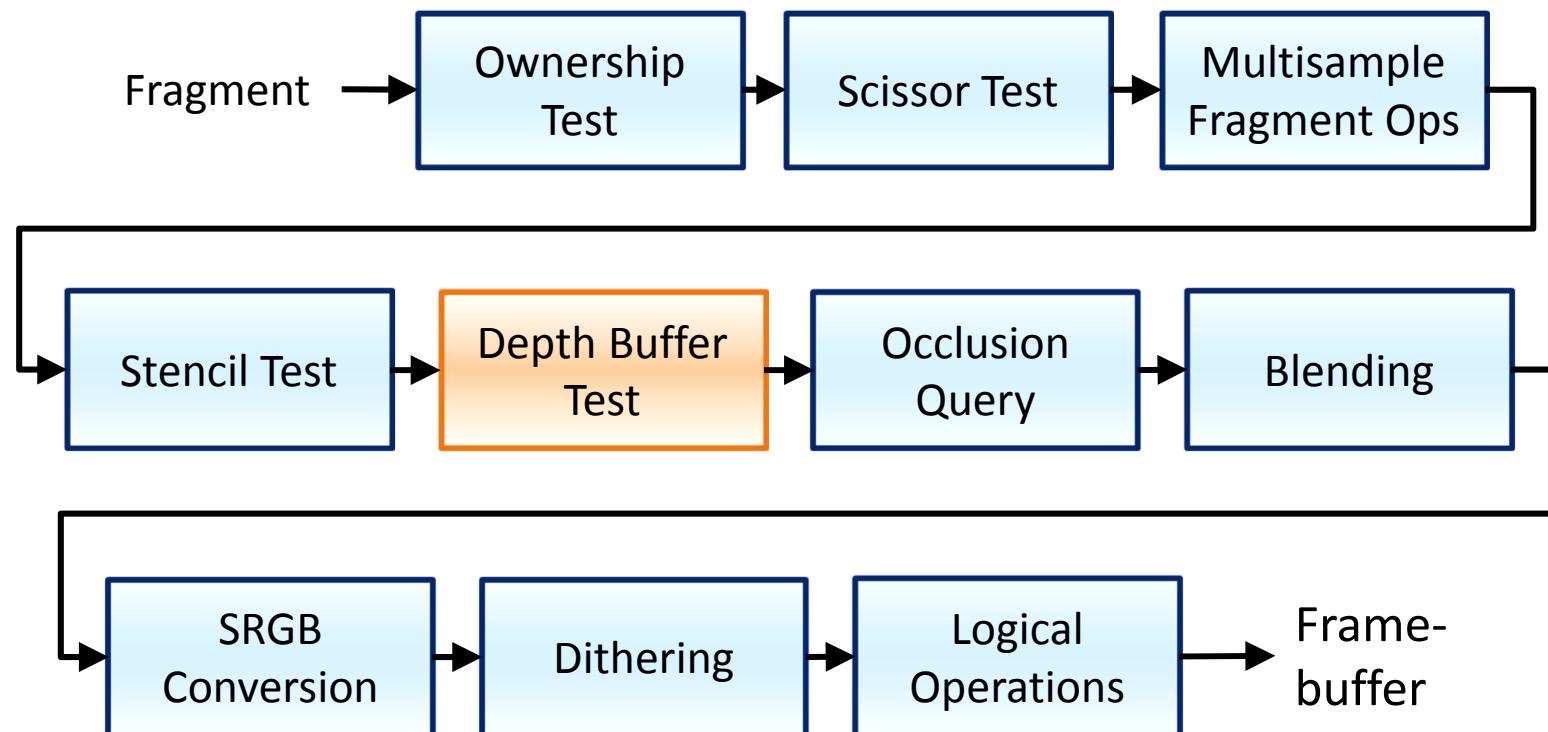
Szene aus der Sicht der Kamera

OpenGL-Pipeline



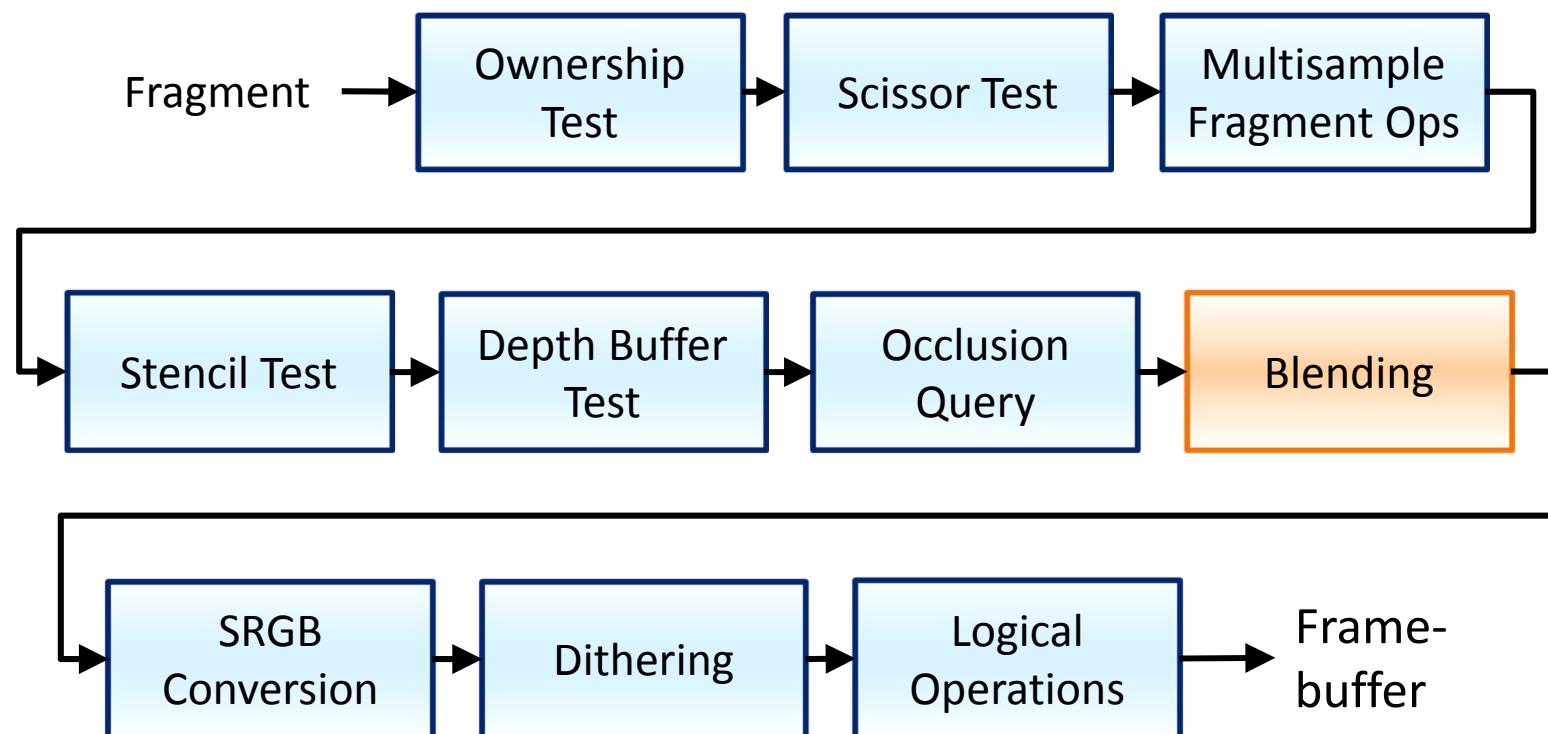
Fragment-Verarbeitung

- ▶ durch die Rasterisierung werden Fragmente erzeugt
- ▶ vor dem Schreiben in den Framebuffer müssen die Fragmente Tests und Framebuffer-Operationen durchlaufen
- ▶ einen kennen wir schon: den Tiefentest!



Fragment-Verarbeitung

- ▶ durch die Rasterisierung werden Fragmente erzeugt
- ▶ vor dem Schreiben in den Framebuffer müssen die Fragmente Tests und Framebuffer-Operationen durchlaufen
- ▶ **wichtig:** auf den Framebuffer kann in einem Shader nicht zugegriffen werden → die Framebuffer-Operationen können nur konfiguriert werden



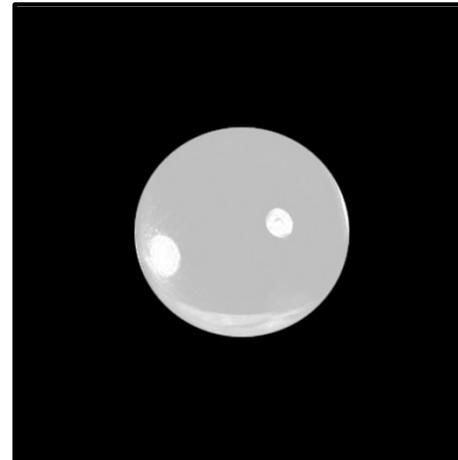
OpenGL Blending

- Kombination: Farbwerte im Framebuffer mit Farben von Fragmenten

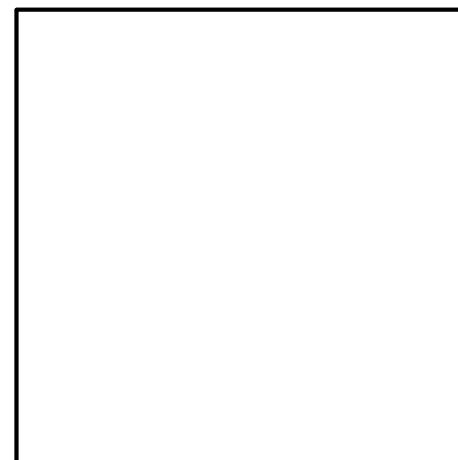
RGB der Fragmente



Alpha der Fragmente



RGB im Framebuffer



Alpha im Framebuffer

Blending Resultat


$$\text{FragmentRGB} * \text{FragmentAlpha} + \\ \text{FramebufferRGB} * (1-\text{FragmentAlpha})$$

Blending

- ▶ Blending bezeichnet die Farb-Kombination von
 - ▶ erzeugten Fragmenten (source) (R_S, G_S, B_S, A_S) und
 - ▶ Pixel im Framebuffer (destination) (R_D, G_D, B_D, A_D)
 - ▶ jede Komponente wird separat ausgerechnet, z.B.
 $R_S * \text{SourceFactor} \text{ Op } R_D * \text{DestFactor}$
- ▶ spezifiziere Blending-Faktoren für Source und Destination:
 (S_R, S_G, S_B, S_A) und (D_R, D_G, D_B, D_A)
 - ▶ oft sind die Faktoren abhängig von den Alpha-Werten (A_S und/oder A_D)
 - ▶ Beispiele:
 - $\text{GL_SRC_ALPHA} = (A_S, A_S, A_S, A_S)$
 - $\text{GL_ONE_MINUS_SRC_ALPHA} = (1-A_S, 1-A_S, 1-A_S, 1-A_S)$ oder auch
 - $\text{GL_ZERO} = (0, 0, 0, 0)$, $\text{GL_ONE} = (1, 1, 1, 1)$
 - ▶ Einschalten und Blending-Funktion


```
glEnable( GL_BLEND )
glBlendFunc( sfactor, dfactor )
```
 - ▶ sofern nichts anderes gewählt wird für **Op** eine Addition ausgeführt

$$(R_S S_R + R_D D_R, G_S S_G + G_D D_G, B_S S_B + B_D D_B, A_S S_A + A_D D_A) =$$

$$(R_S S_R, G_S S_G, B_S S_B, A_S S_A) + (R_D D_R, G_D D_G, B_D D_B, A_D D_A)$$

Blending *cont.*



- ▶ typische Anwendung für Blending:
Zeichnen eines semitransparenten Objekts über ein Opakes

```
glBlendFunc( GL_SRC_ALPHA,  
              GL_ONE_MINUS_SRC_ALPHA )
```
- ▶ spezielle Blending Funktionen, z.B.
`GL_CONSTANT_COLOR, GL_ONE_MINUS_CONST_ALPHA`
für konstante Farbe bzw. Transparenz
- ▶ die **Blend Equation** legt die Verknüpfungsoperation **Op** fest
`glBlendEquation(GL_FUNC_ADD)` (default)
 - ▶ Differenzbilder: `GL_FUNC_SUBTRACT`
 - ▶ Inverse Differenzbilder: `GL_FUNC_REVERSE_SUBTRACT`
 - ▶ Minimum/Maximum aller Werte: `GL_FUNC_MIN, GL_FUNC_MAX`
 - ▶ z.B. für Maximum Intensity Projection (MIP) in der Medizin oder morphologische Filter

OpenGL Blending



► <http://www.andersriggelsen.dk/glblendfunc.php>

Anders Riggelsen

Home Curriculum vitae Projects Clickteam Web-stuff Books I own Links
Binary File Schema glBlendFunc Tool Disk Usage Analyzer BaseEngine

Visual glBlendFunc + glBlendEquation Tool

Use this tool to quickly visualize how the different blending modes work in OpenGL.

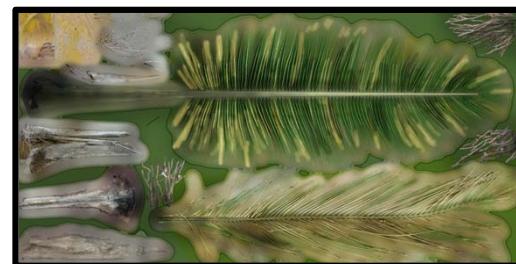
Historischer Einschub: Alpha-Test

- ▶ verwerfe Fragmente aufgrund ihres Alpha-Wertes

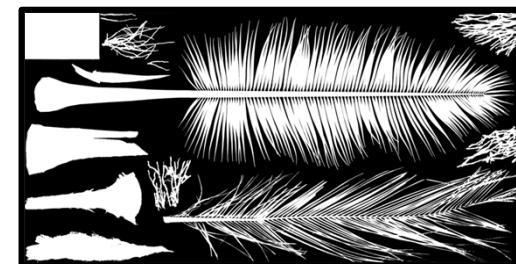
```
glAlphaFunc( func, value )
```

- ▶ Beispiel:

```
// nur Fragmente mit Alpha == 1.0 werden gezeichnet  
glAlphaFunc( GL_EQUAL, 1.0 )  
glEnable( GL_ALPHA_TEST )
```



24 Bit Farbinformation



8 Bit Alpha-Kanal
schwarz = transparent

Historischer Einschub: Alpha-Test (2)



- ▶ verwerfe Fragmente aufgrund ihres Alpha-Wertes

```
glAlphaFunc( func, value )
```

- ▶ Beispiel:

```
// nur Fragmente mit Alpha == 1.0 werden gezeichnet
glAlphaFunc( GL_EQUAL, 1.0 )
 glEnable( GL_ALPHA_TEST )
```

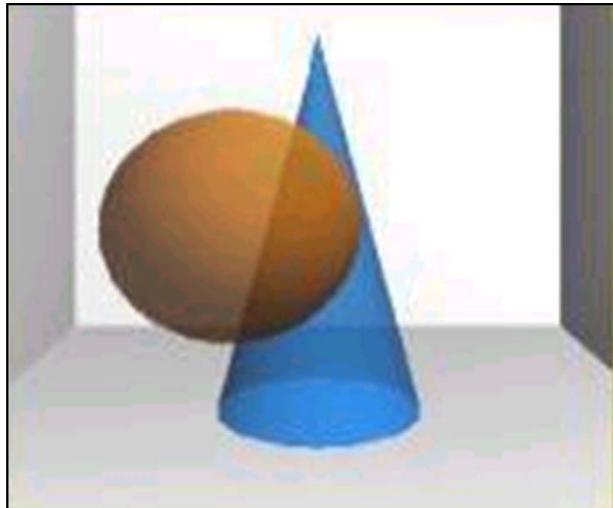
 Alpha-Tests gibt es im Core-Profil von OpenGL 3.x/4.x nicht mehr!

- ▶ kann aber einfach ersetzt werden:

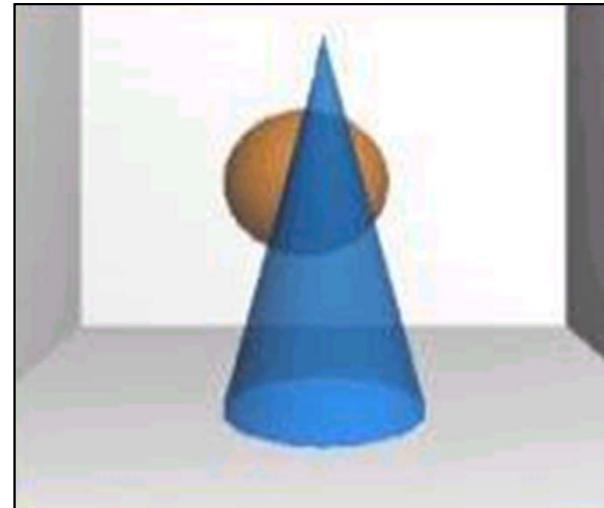
- ▶ im Fragment Programm ein bedingtes **discard** implementieren
- ▶ es muss zwar selbst implementiert werden, ist aber flexibler als ein konfigurierbarer Alpha-Test

Semitransparente Objekte

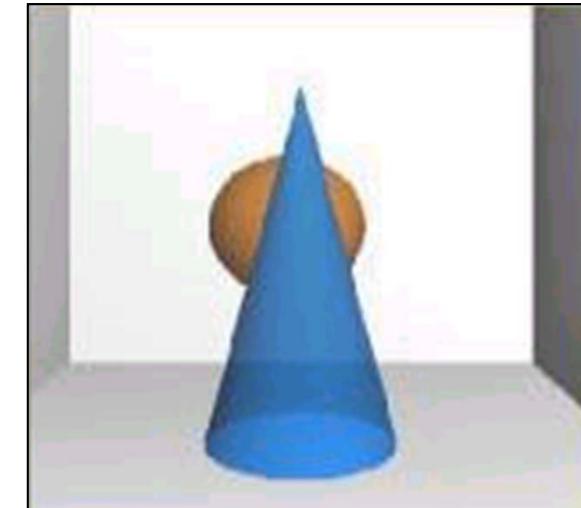
- die Darstellung semitransparenter Objekte erfordert Tiefensortierung, weil Blending i.A. nicht kommutativ ist
- Beispiel: braune Kugel und blauer Kegel (beide semitransparent)



Kugel vor dem Kegel,
Kegel zuerst gezeichnet



Kugel hinter dem Kegel,
Kugel zuerst gezeichnet



Kugel hinter dem Kegel,
Kegel zuerst gezeichnet:
Tiefentest verhindert
Blending

Transparente Objekte ohne Tiefensortierung



- im Bild überlappende transparente Polygone werden **ohne Sortierung nicht korrekt** dargestellt
- übliche „Notlösung“ bei der Rasterisierung: transparente Polygone verschwinden hinter opaken, aber ändern nicht den z-Buffer
(Renderstate **glDepthMask**):

```
glEnable( GL_DEPTH_TEST );  
  
glDisable ( GL_BLEND );  
glUseProgram( DrawOpaqueProgram );  
<drawscene> // nur Fragmente von opaken Objekte ausgeben  
  
glEnable ( GL_BLEND );  
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );  
glDepthMask( GL_FALSE );  
glUseProgram( DrawTransparentProgram );  
<drawscene> // blende sortierte transparente Objekte  
glDepthMask( GL_TRUE );
```

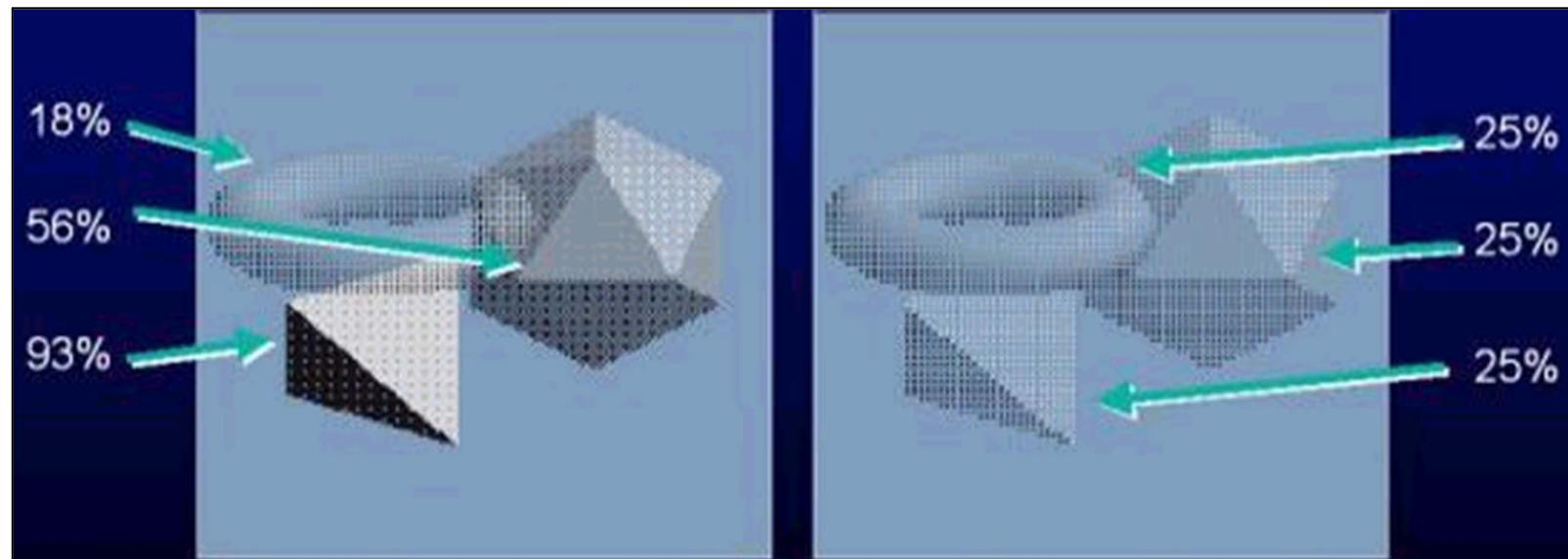
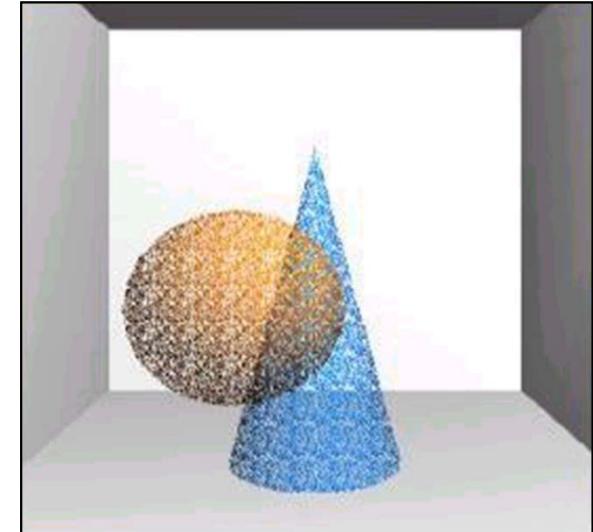
```
...  
if (inColor.a < 1.0) discard;  
...
```



Screendoor Transparency

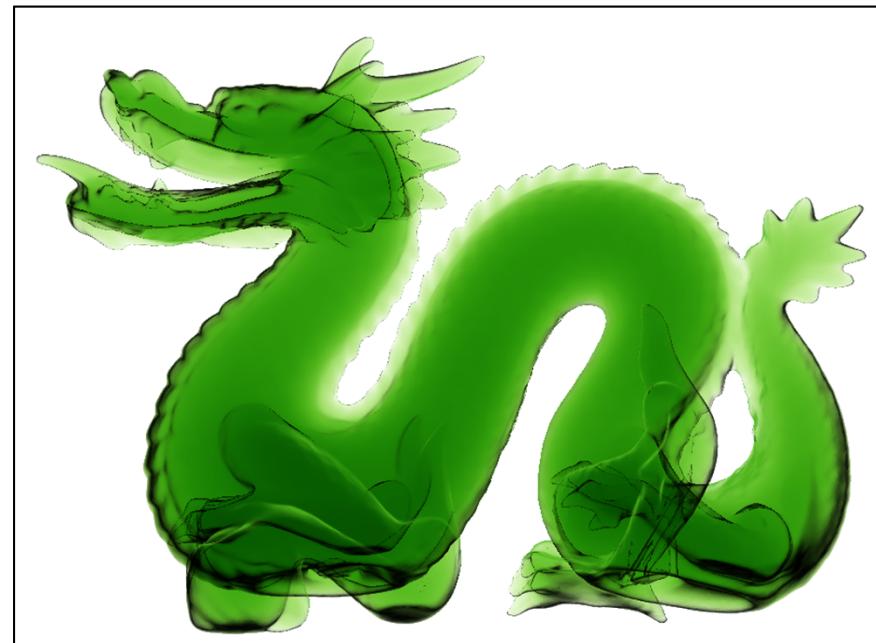
Semitransparenz ohne Blending

- ▶ setze Pixel beim Rasterisieren entsprechend eines Binärmusters mit Anteil der Einsen entsprechend der Opazität (Polygon Stippling)
- ▶ wichtig: verwende unterschiedliche Muster für unterschiedliche Objekte, sonst kein Transparenzeffekt (Bild rechts-unten)



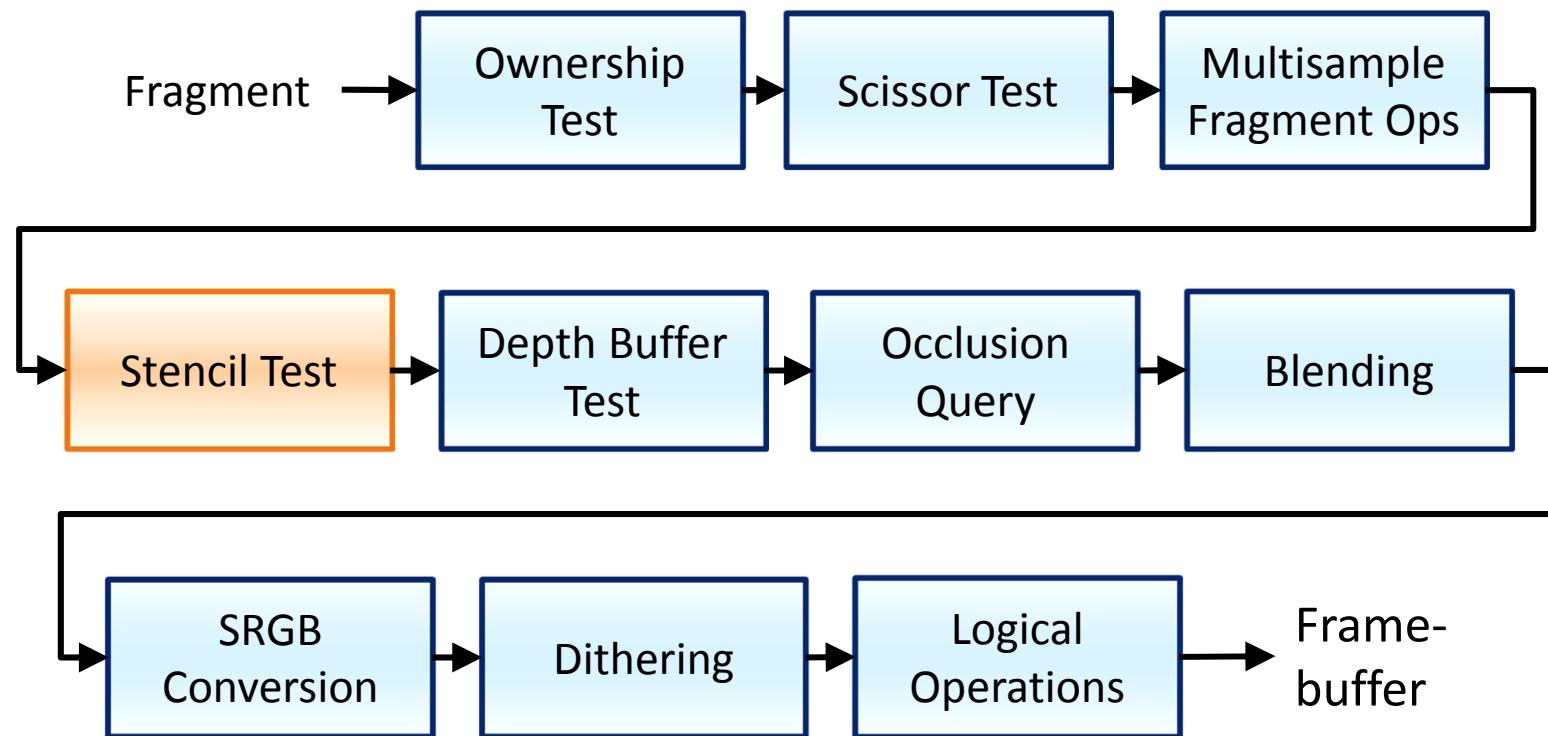
Transparenz ohne Sortierung

- ▶ es gibt auch kommutative Blending-Modi
 - ▶ Multiplikation der Source- mit der Destination-Farbe
`glBlendFunc(GL_DST_COLOR, GL_ZERO)`
 - ▶ Addition der beider Farbvektoren
`glBlendFunc(GL_ONE, GL_ONE)`
- ▶ Alpha-Blending kann nur eine grobe Approximation sein
 - ▶ das Aussehen hängt natürlich auch von der Dicke des Objekts ab



Fragment-Verarbeitung

- ▶ durch die Rasterisierung werden Fragmente erzeugt
- ▶ vor dem Schreiben in den Framebuffer müssen die Fragmente Tests und Framebuffer-Operationen durchlaufen



OpenGL-Framebuffer



- ▶ Framebuffer besteht aus
 - ▶ Farb- und Tiefenpuffer
 - ▶ kann aber auch einen **Stencil Buffer** beinhalten:
eine „Stanz-Maske“, die oft verwendet wird, um Zustände für jeden Pixel zu speichern (meist 8-Bit pro Pixel)
- ▶ Anfordern des gewünschten Framebuffer-Formats

```
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE |
                      GLUT_DEPTH | GLUT_STENCIL )
```
- ▶ Löschen (früher teure Operation, heute vor dem Zeichnen immer!)

```
glClearColor(0,0,0,1); glClearDepth(1.0);
glClear( GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT |
          GL_STENCIL_BUFFER_BIT )
```

Stencil-Buffer und Stencil-Test



- ▶ mithilfe des **Stencil-Tests** kann ebenfalls entscheiden werden, ob ein Fragment gezeichnet wird, z.B. Vergleich des Stencil-Werts im Buffer mit einem Referenzwert
- ▶ typische Anwendungen
 - ▶ Maskierung von Bildteilen
 - ▶ projektive Schatten und Schattenvolumen
 - ▶ Stencil-Routing



Color buffer without stencil test

| | | |
|------------------|-------|-----|
| 00000 | 11111 | 000 |
| 00000 | 11111 | 000 |
| 00000 | 11111 | 000 |
| 00000 | 11111 | 000 |
| 00000 | 11111 | 000 |
| 0000000000000000 | | |
| 0000000000000000 | | |

Stencil buffer



Color buffer with stencil test

Bild: <https://open.gl/depthstencils>

Stencil-Test und Tiefentest



Reihenfolge der Fragment-Tests (Alpha-Test nur in klassischen OpenGL)

```
if ( fragment.alpha alphafunc refalpha )
    if ( buffer.stencil stencilfunc refstencil ) {
        if ( fragment.depth depthfunc buffer.depth ) {
            // es kann mehrere Color-Buffers in einem
            // Backbuffer geben
            foreach colorbuffer
                // hier würde noch Blending stattfinden
                buffer.color = fragment.color;
                buffer.depth = fragment.depth;
                buffer.stencil = zpass();
    }
```

Stencil-Test und Tiefentest



Reihenfolge der Fragment-Tests (Alpha-Test nur in klassischen OpenGL)

```
if ( fragment.alpha alphafunc refalpha )
    if ( buffer.stencil stencilfunc refstencil ) {
        if ( fragment.depth depthfunc buffer.depth ) {
            // es kann mehrere Color-Buffers in einem
            // Backbuffer geben
            foreach colorbuffer
                // hier würde noch Blending stattfinden
                buffer.color = fragment.color;
                buffer.depth = fragment.depth;
                buffer.stencil = zpass();
        } else
            // Stencil&Alpha bestanden, Tiefentest nicht
            buffer.stencil = zfail();
    } else
        // Stencil Test nicht bestanden
        buffer.stencil = fail();
```

Stencil-Test *cont.*



Konfiguration des Stencil-Tests

► Stencil-Test festlegen

```
glStencilFunc( stencilfunc, refstencil, mask )
```

- vergleiche mit Referenzwert in maskierten Bits

```
GL_ALWAYS, GL_NEVER, GL_LESS, GL_EQUAL, ...
```

► Änderungen im Stencil-Buffer je nach dem, ob der Stencil-Test bestanden und der Tiefentest bestanden wurde oder nicht

```
glStencilOp( fail, zfail, zpass )
```

- mögliche Funktionen:

```
GL_KEEP, GL_ZERO, GL_INCR, GL_DECR, GL_INVERT, GL_REPLACE
```

► Beispiele

- zeichne Maske in den Stencil-Buffer

```
glStencilFunc( GL_ALWAYS, 0x1, 0x1 );
```

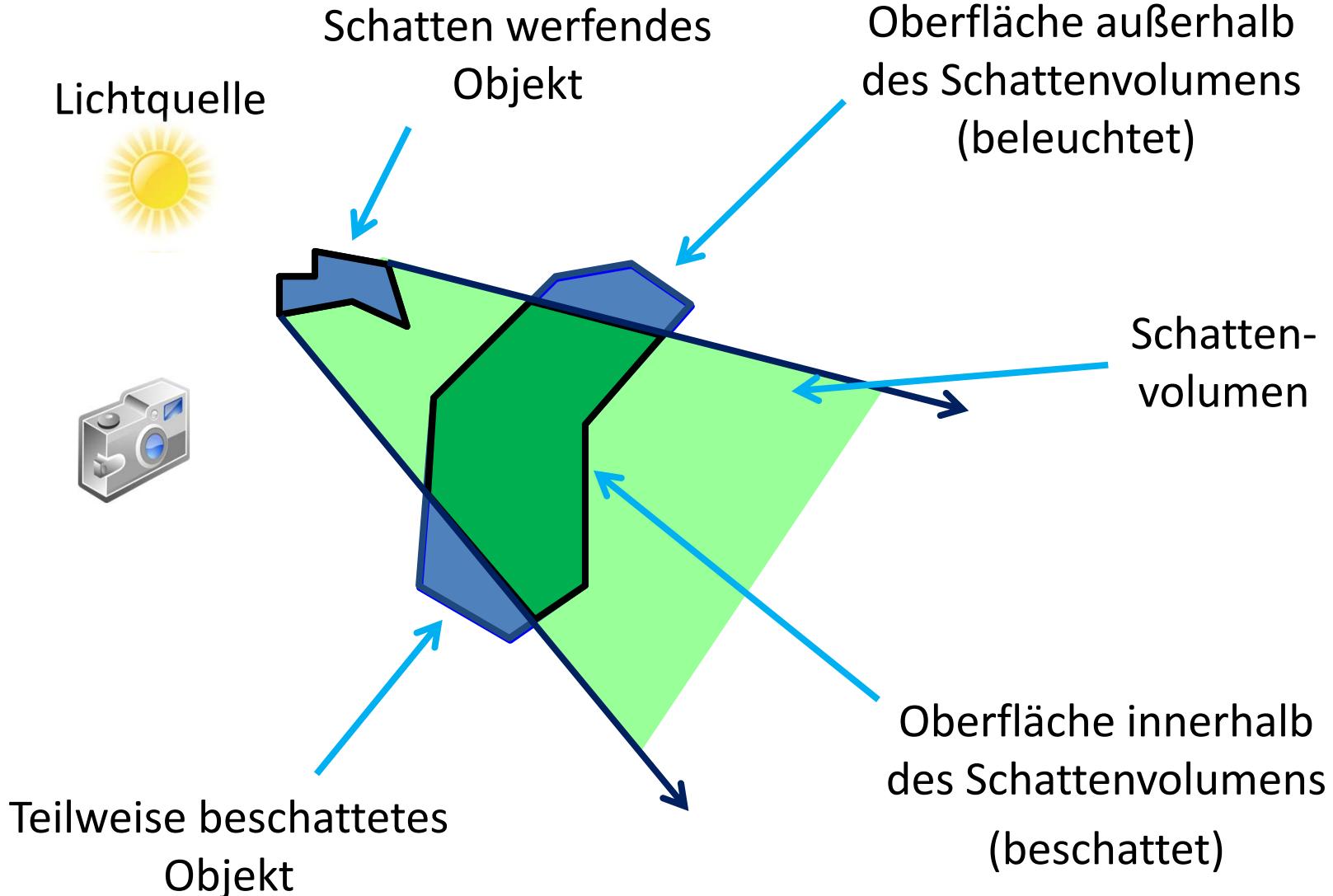
```
glStencilOp( GL_REPLACE, GL_REPLACE, GL_REPLACE );
```

- zeichne Objekte nur dort, wo Stencil ungleich 1

```
glStencilFunc( GL_NOTEQUAL, 0x1, 0x1 );
```

```
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
```

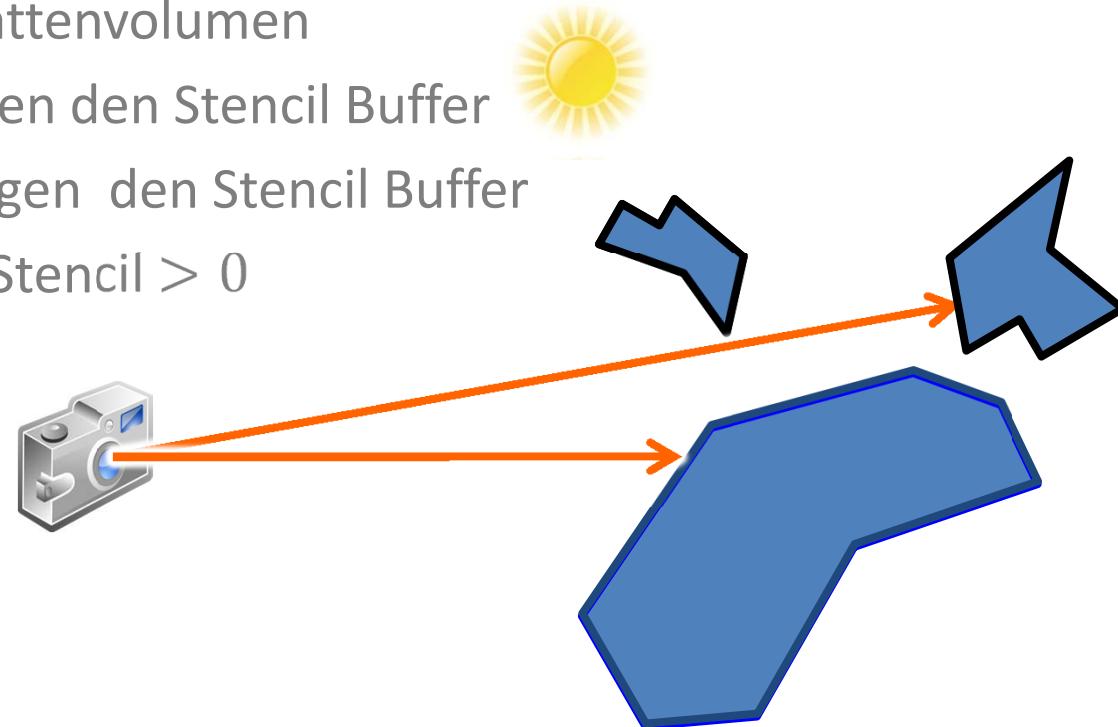
Exkurs: Schattenvolumen



Schattenvolumen: Schattentest

Stencil Buffer zum Zählen der Schnitte mit Schattenvolumen

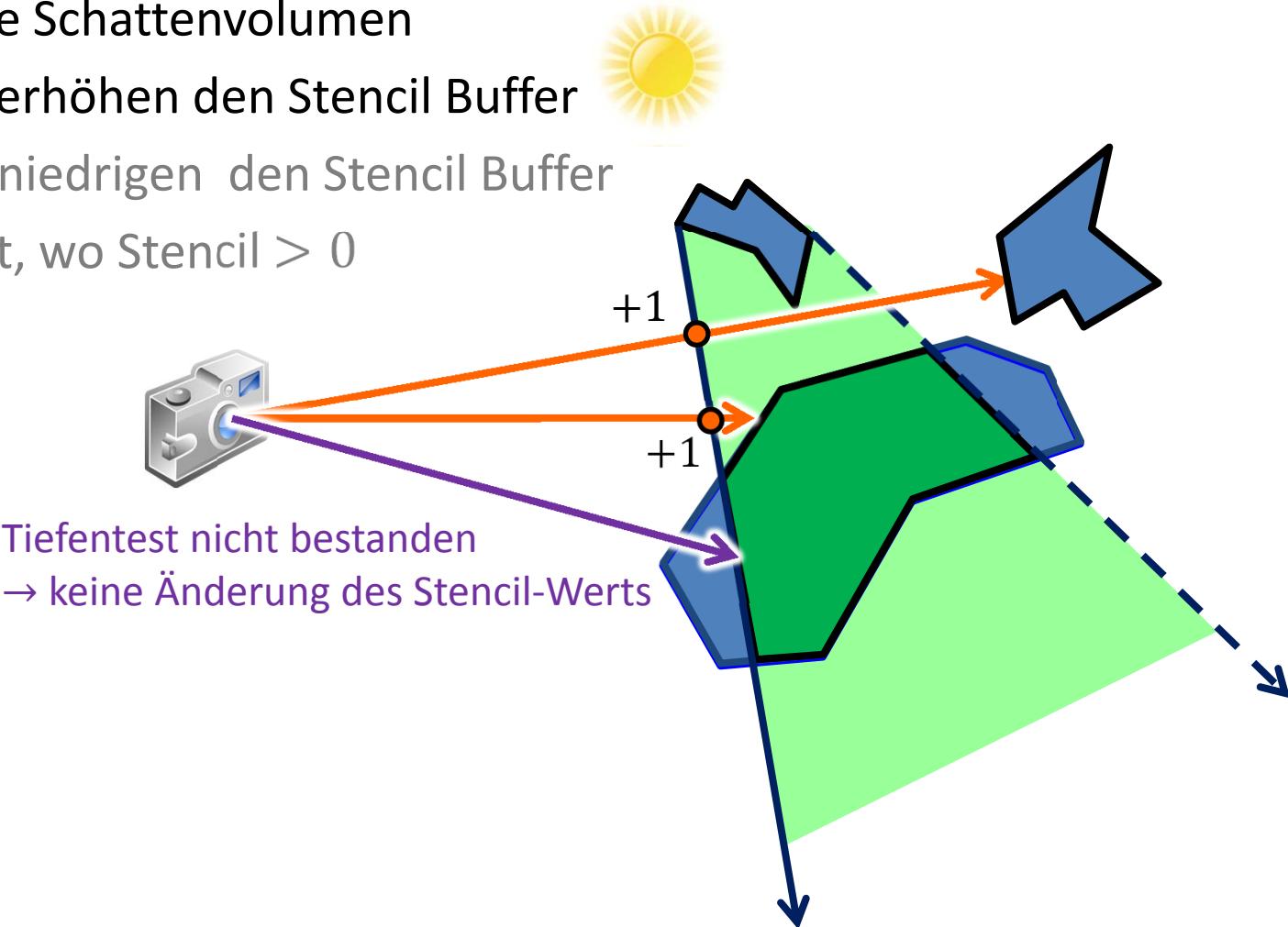
- ▶ zuerst werden die Objekte gezeichnet
- ▶ anschließend wird Schreiben in den Farb- und Tiefenpuffer deaktiviert
- ▶ zeichne dann die Schattenvolumen
 - ▶ Vorderseiten erhöhen den Stencil Buffer
 - ▶ Rückseiten erniedrigen den Stencil Buffer
- ▶ Schatten ist dort, wo $\text{Stencil} > 0$



Schattenvolumen: Schattentest

Stencil Buffer zum Zählen der Schnitte mit Schattenvolumen

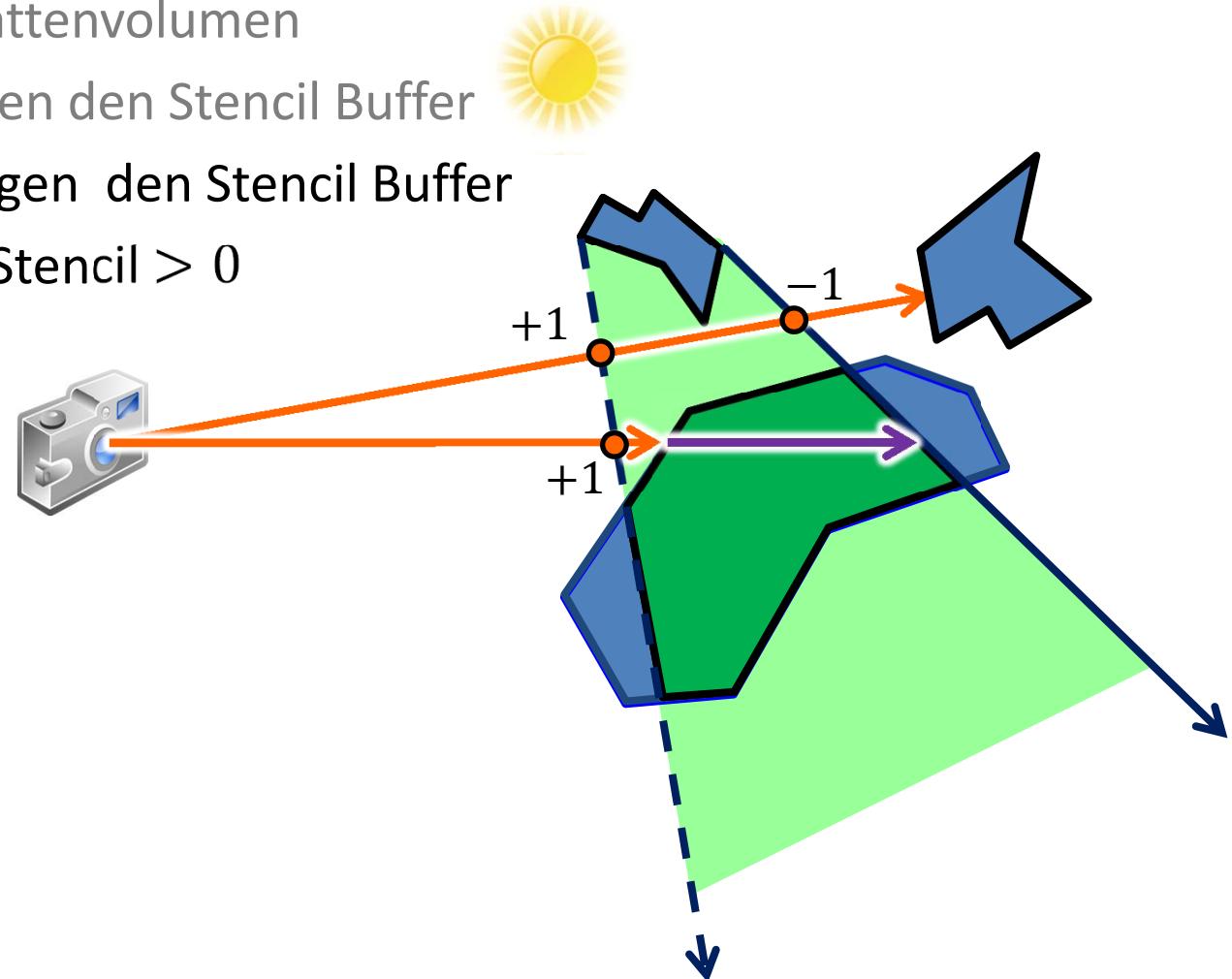
- ▶ zuerst werden die Objekte gezeichnet
- ▶ anschließend wird Schreiben in den Farb- und Tiefenpuffer deaktiviert
- ▶ zeichne dann die Schattenvolumen
 - ▶ Vorderseiten erhöhen den Stencil Buffer
 - ▶ Rückseiten erniedrigen den Stencil Buffer
- ▶ Schatten ist dort, wo $\text{Stencil} > 0$



Schattenvolumen: Schattentest

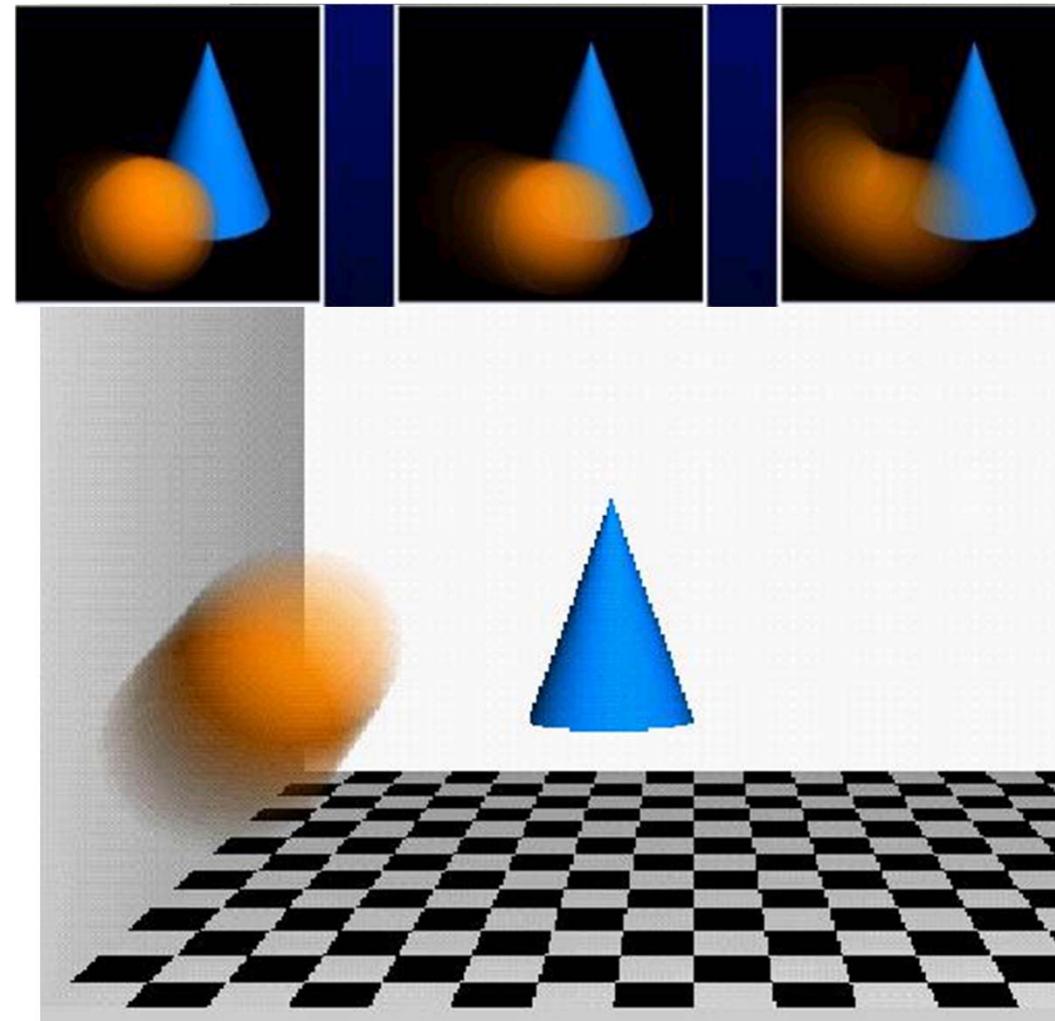
Stencil Buffer zum Zählen der Schnitte mit Schattenvolumen

- ▶ zuerst werden die Objekte gezeichnet
- ▶ anschließend wird Schreiben in den Farb- und Tiefenpuffer deaktiviert
- ▶ zeichne dann die Schattenvolumen
 - ▶ Vorderseiten erhöhen den Stencil Buffer
 - ▶ Rückseiten erniedrigen den Stencil Buffer
- ▶ Schatten ist dort, wo $\text{Stencil} > 0$



Accumulation-Buffer: Bewegungsunschärfe

- ▶ Zwischenspeicher zur Kombination mehrerer Rendering-Schritte
- ▶ zeichne mehrere Bilder zu verschiedenen Zeitpunkten



Stereo/Double Buffering



Stereoskopisches Rendering (Hardware-unterstützt)

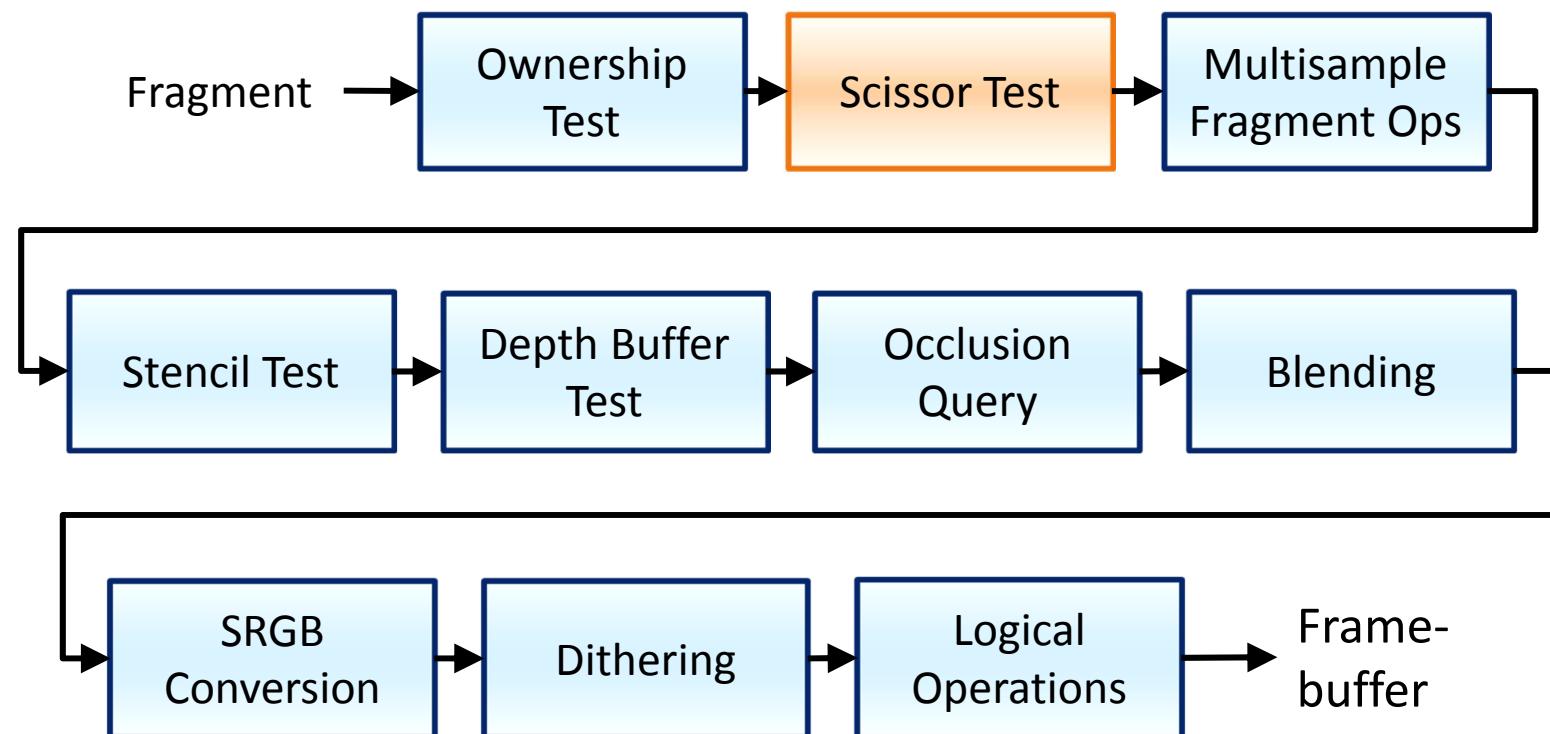
- ▶ erzeuge je ein Bild für jedes Auge
- ▶ wir benötigen dann einen **Quad Buffer** (2 Front und 2 Back Buffer)
`glutInitDisplayMode(GLUT_RGBA | GLUT_STEREO ...)`
- ▶ Wechsel zwischen den Front und Back Buffers erfolgt nach wie vor mit
`glutSwapBuffers()`
- ▶ Back Buffer für linkes bzw. rechtes Auge muss explizit gewählt werden:

```
while( GL_TRUE ) {  
    glDrawBuffer( GL_LEFT_BUFFER );  
    SetPerspective( <LeftEye> );  
    drawscene();  
  
    glDrawBuffer( GL_RIGHT_BUFFER );  
    SetPerspective( <RightEye> );  
    drawscene();  
  
    glutSwapBuffers();  
}
```

Fragment-Verarbeitung

Scissor-Test

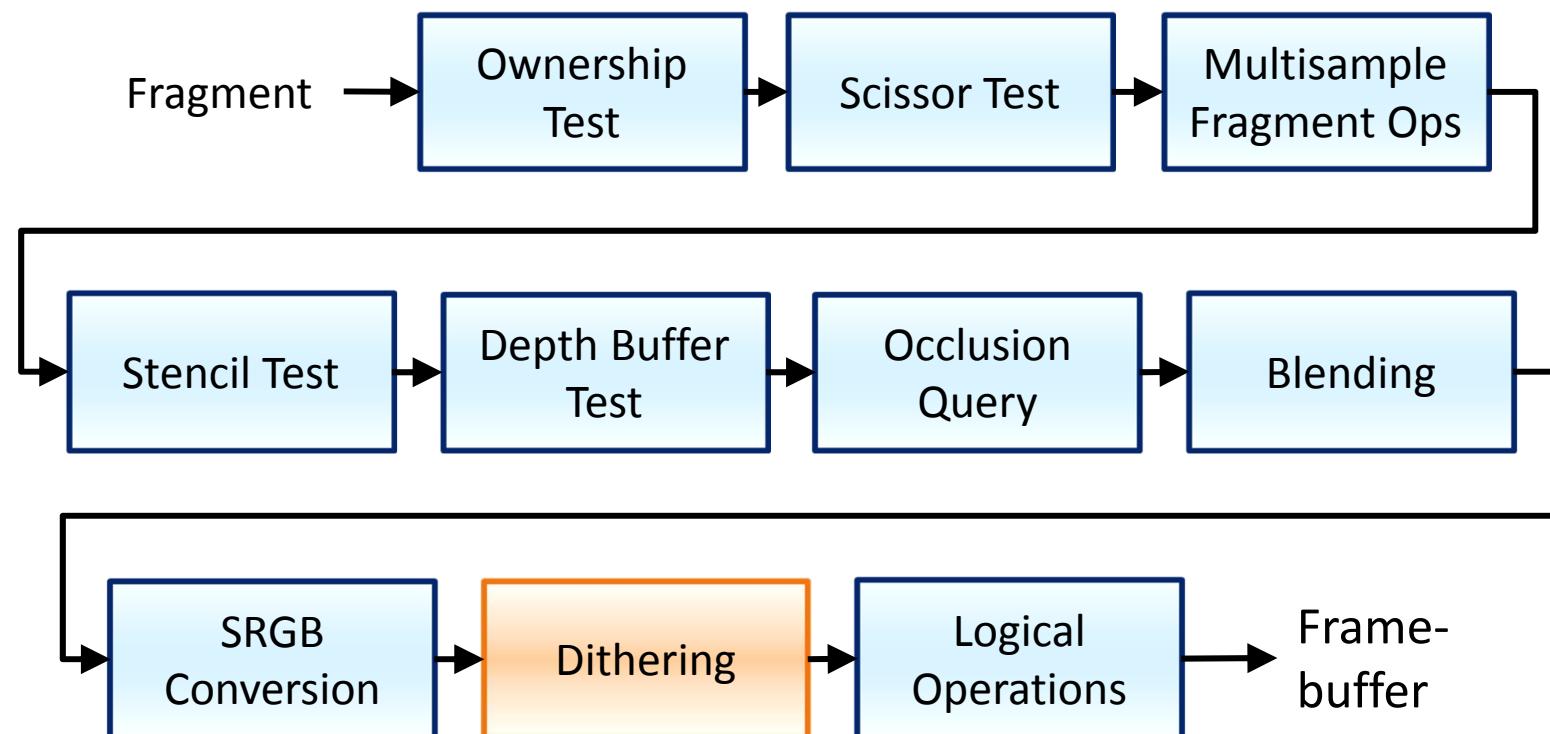
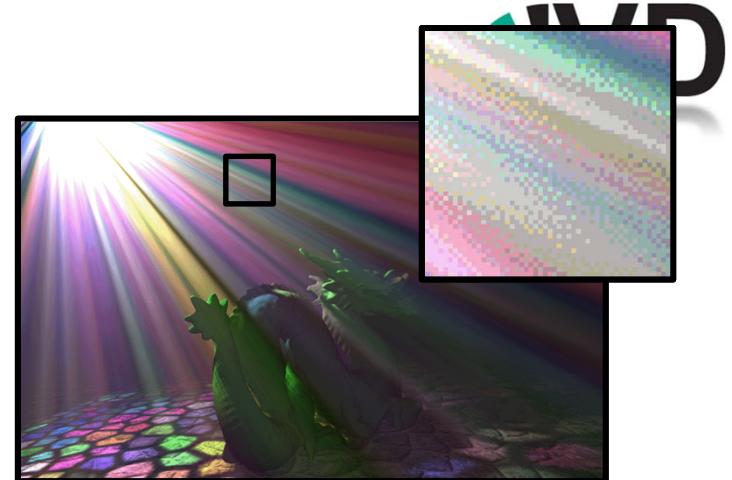
- ▶ dient lediglich dazu Fragmente außerhalb eines Rechtecks zu entfernen
`glScissor(x, y, width, height)`
- ▶ wird verwendet, um kleinere Bereiche des Fensters zu aktualisieren
 - ▶ z.B. Löschen mit `glClear()`
- ▶ vergleichbar mit einer rechteckigen Stencil-Maske



Fragment-Verarbeitung

Dithering

- ▶ Fehlerdiffusion, nur historisch für Hardware ohne TrueColor Darstellung



Fragment-Verarbeitung

Logical Ops

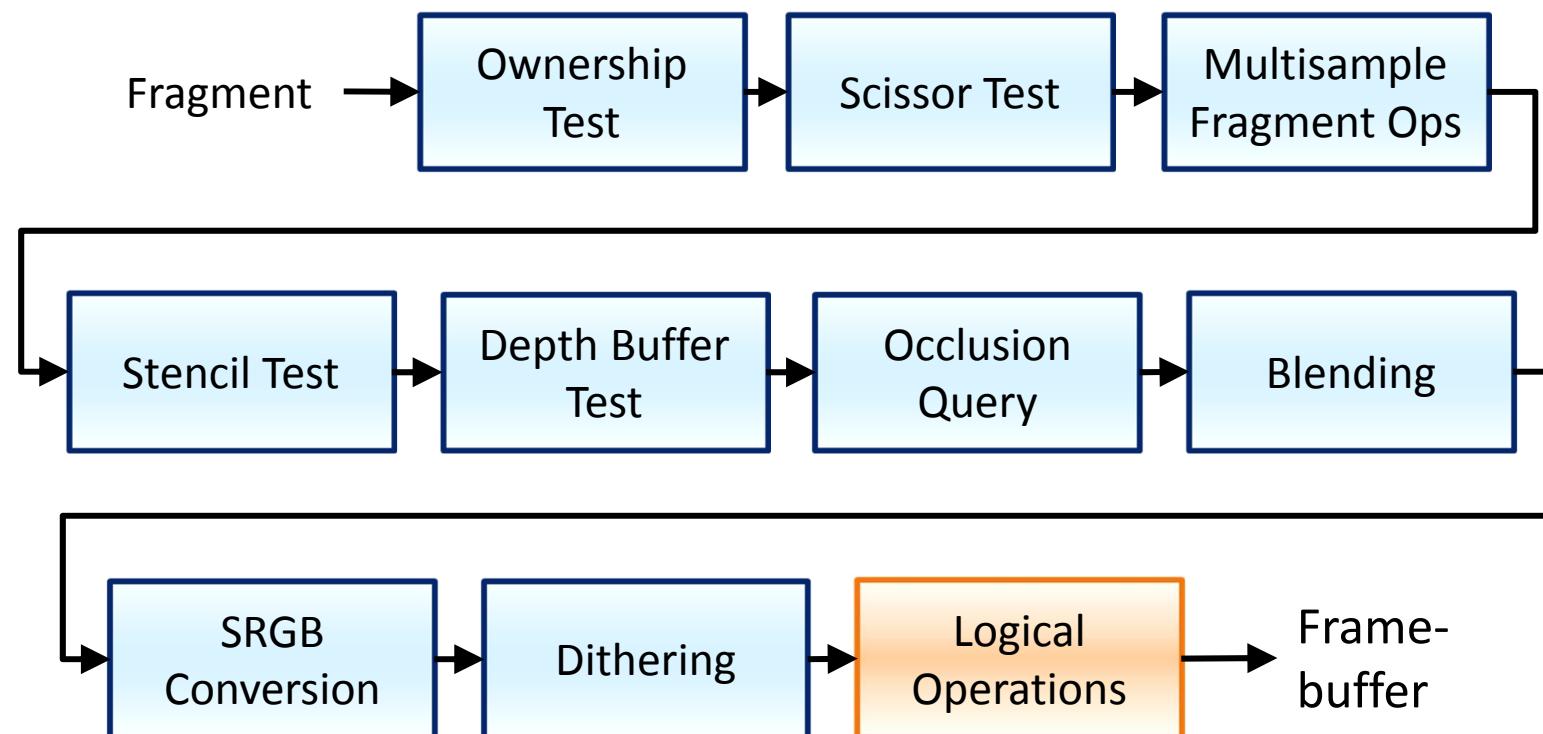
- ▶ bitweise Operationen der Fragment- und Framebuffer-Farbwerte

```
glEnable( GL_LOGIC_OP );  
glLogicOp( GL_XOR );
```

- ▶ 15 weitere Modi, z.B. **GL_AND**, **GL_OR**, **GL_NAND**, ...



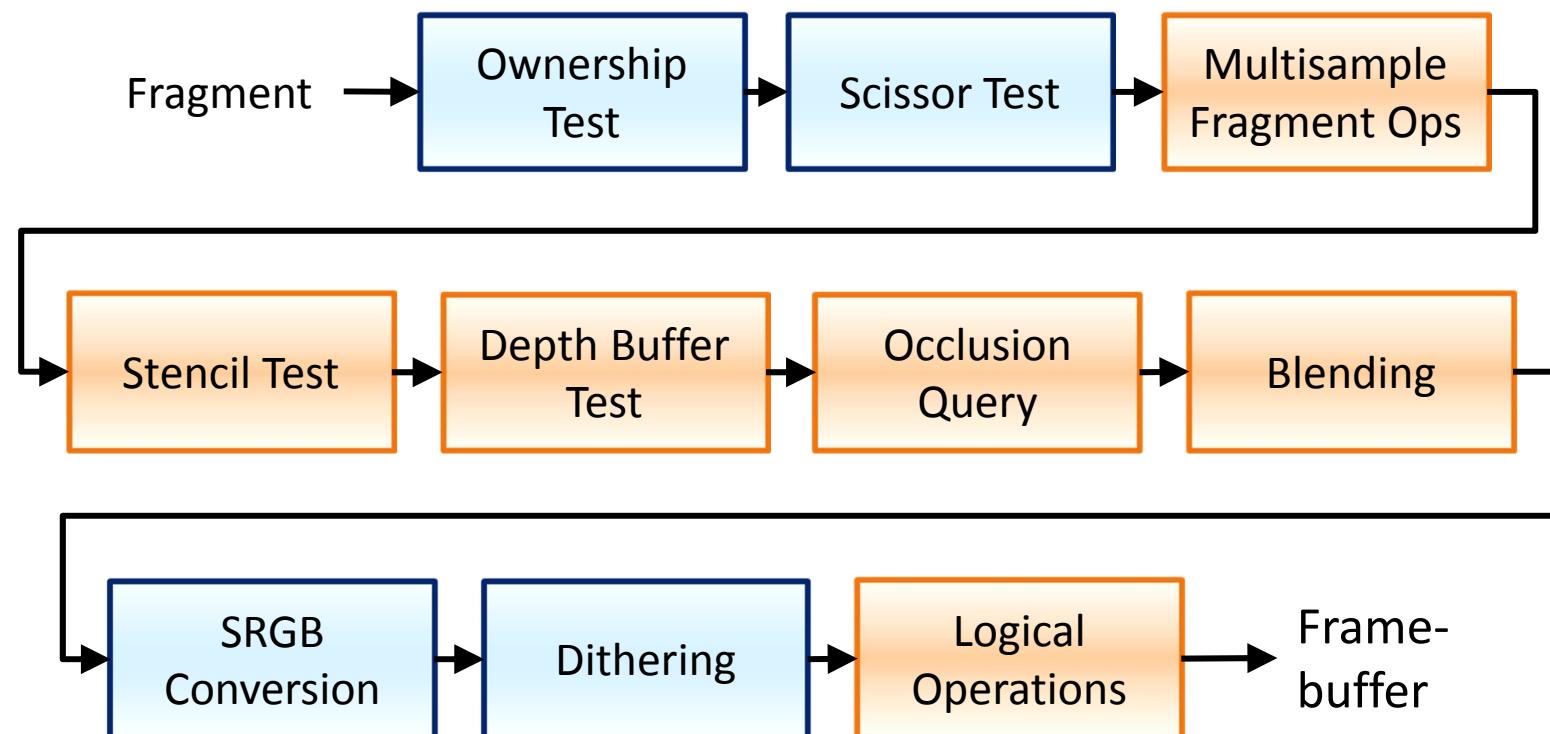
- ▶ verhindert Blending, auch wenn dieses angeschaltet ist



Ausblick

Echtzeit-Rendering Techniken

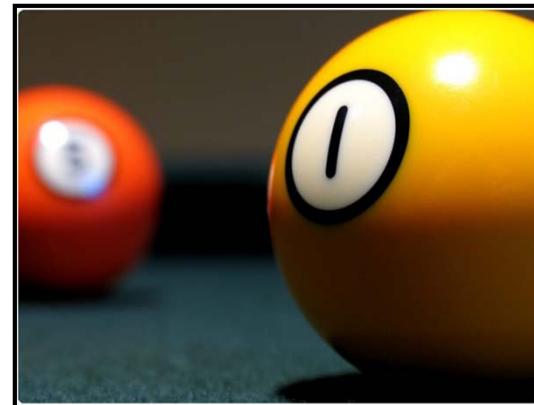
- ▶ ... machen intensiv Gebrauch von den speziellen Funktionen der Grafik-Hardware



Frame Buffer Objects (FBOs)



- ▶ bisher: beim Rendering werden Fragmente in den Farb-, Tiefen-, Stencil-Buffer des Framebuffers geschrieben
- ▶ FBOs erlauben es die Ausgabe in eigene „Off-Screen“ Buffer umzuleiten
 - ▶ Color Buffers: eine oder mehrere Texturen in die Farbwerte geschrieben werden
 - ▶ Renderbuffers: Tiefen- und Stencil-Buffer
- ▶ FBOs sind Grundlage für fast alle (modernen) Rendering-Techniken in der interaktiven Computergrafik



- ▶ Referenzen:

<http://www.gamedev.net/reference/programming/features/fbo1/>
<http://www.gamedev.net/reference/programming/features/fbo2/>

Weitere Ausblicke



Hardware Tesselierungs-Einheit

- ▶ erlaubt es (im Gegensatz zum Geometry Shader *effizient*) Dreiecke zu unterteilen und neue Vertizes zu verschieben (Displacement Mapping)
- ▶ hierzu gibt es 3 neue Stufen zwischen Vertex und Geometry Shader
 - ▶ Tessellation Control:
programmierbar, entscheidet wie oft ein Patch unterteilt wird
 - ▶ Tessellator:
konfigurierbar, erzeugt neue Primitive
 - ▶ Tessellation Evaluation:
programmierbar, kann Vertizes aus dem Tessellator modifizieren
- ▶ Info:
<http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/>

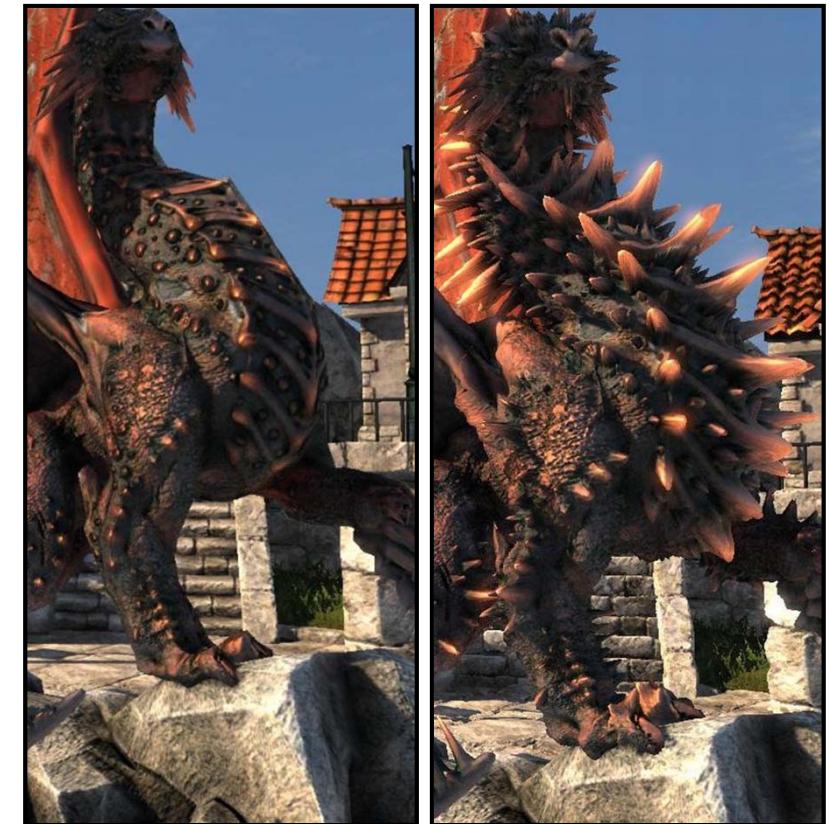


Bild: Unigine Heaven Benchmark