Computergrafik

Vorlesung im Wintersemester 2024/25
Kapitel 7: OpenGL und Grafik-Hardware
(Folienteil zum Selbststudium)

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

Wegweiser durch diesen Foliensatz

- dieser Foliensatz enthält Teile, die Sie sich bitte zuhause ansehen
- Folien für das Selbststudium sind mit dem Symbol  gekennzeichnet
  - es geht zunächst um den Teil bis Folie 75 (in diesem PDF sind nur diese Folien, daher sind auch die meisten gekennzeichnet)
  - die Informationsdichte ist im Vergleich zu anderen Folien geringer, es klingt also nach mehr, als es tatsächlich ist

Ziel des Selbststudiums / Anmerkungen

- die prinzipielle Grafik-Pipeline haben Sie bereits kennengelernt
- Sie sollen sich einen Eindruck verschaffen, wie diese in klassischem OpenGL umgesetzt war (konfigurierbare Verarbeitung, Shading etc.)
  - viele Aspekte sind nach wie vor gültig, z.B. Framebuffer, Primitivtypen
  - noch mehr Funktionalität dieser sog. Fixed-Function Pipeline ist auf den Folien nur angedeutet und wir besprechen sie nicht im Detail (siehe z.B. Folie 43-46)

Wegweiser durch diesen Foliensatz

Ziel des Selbststudiums / Anmerkungen (cont.)

- wichtig für die Praxisteile und Übungsaufgaben ist modernes OpenGL mit der programmierbaren Grafik-Pipeline
  - die einzelnen Stufen sind auf Folien 49-63 beschrieben (bitte kurz überfliegen, werde ich in der Vorlesung nochmal ansprechen)
  - die OpenGL Shading Language (GLSL), mit der die sog. Shader (also die Stufen der Grafik-Pipeline) programmiert werden, wird auf Folien 64-75 eingeführt

- schauen Sie sich bitte die o.g. Teile an
  - die wichtigsten Aspekte/Konzepte werde ich in der Vorlesung nochmals ansprechen, aber z.B. nicht ins Detail gehen bei GLSL

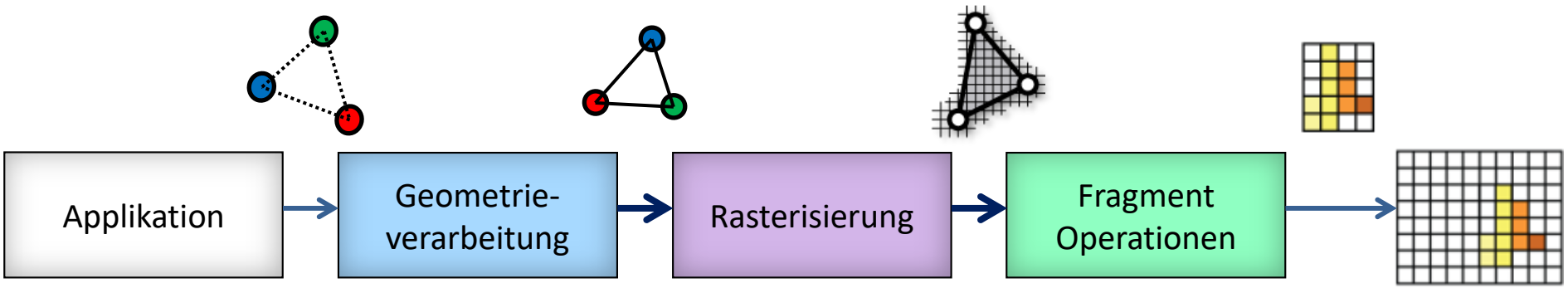
Inhalt

- Allgemeines und Historisches zu OpenGL
- klassischem OpenGL (in der Übung/Selbstvorbereitung): `glIrgendwas`
- modernes OpenGL
  - Shader Programmierung: OpenGL Shading Language
  - Vertex Arrays, Index Buffers, … Texturen
  - Konzepte gelten ebenso für andere moderne APIs (Vulkan, Direct3D, …)
- Überblick zu speziellen Rendering-Techniken (Schatten, …)

Grafik-Pipeline

Verarbeitung in der Grafik-Pipeline

- Rasterisierung (in Verbindung mit Tiefenpuffer) ist effizient: Dreiecke durchlaufen – nacheinander, aber unabhängig – alle dieselben Verarbeitungsschritte
- Konsequenzen der Pipeline-Architektur: naiv nur direkte Beleuchtung
  - Schatten, Spezialeffekte und globale Beleuchtungseffekte müssen über mehrere Rendering-Durchgänge realisiert werden
- Grafik-Hardware verwenden wir über APIs wie OpenGL, Direct3D, …



Was ist OpenGL?

3D Rendering API: OpenGL

- Plattform-, Hardware- und Programmiersprachen-unabhängig
- kein Handling von Fenstern, Events, Menüs, ...
- OpenGL implementiert nur die Grafik-Pipeline u.a. mit
  - geometrischen Primitive: Punkte, Linien, Dreiecke, ...
  - Texturen, Texturfilterung (Mip-Mapping etc.), Z-Buffer
  - Stencil-Buffer, Accumulation-Buffer, Alpha-Blending
  - u.v.m.

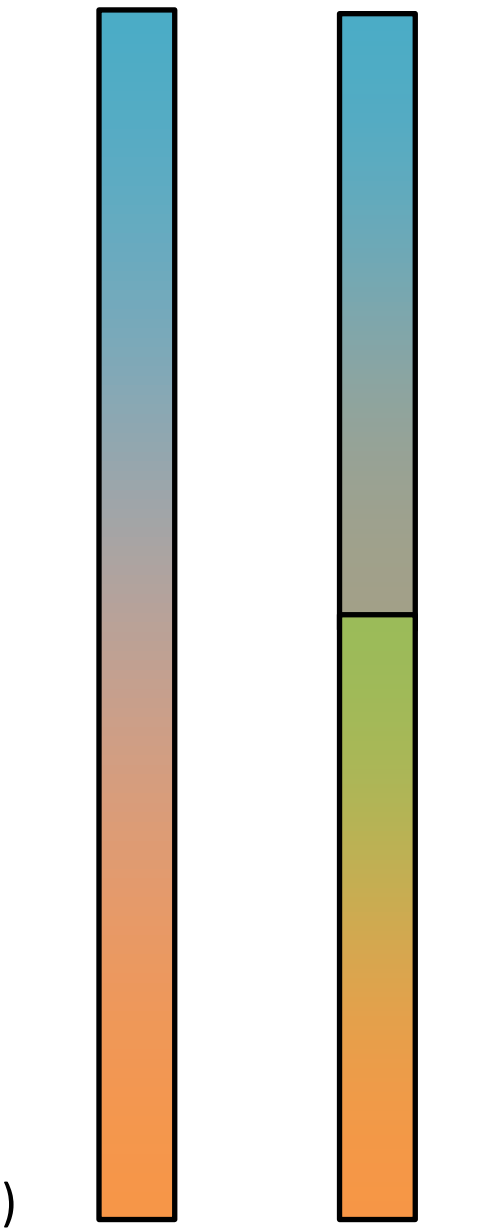
- Low-Level und Immediate Mode API:
  - keine höheren Modellierungs-/Animationskonzepte, kein Szenengraph
  - Immediate Mode, bedeutet in der Vorstellung:
    „ein OpenGL-Befehl bewirkt sofortiges Rendern“
  - Gegenteil: „Retained Mode“-APIs verwalten Objekte mit Texturen etc.

OpenGL – Weitere Merkmale

- Zustandsmaschine
  - die Verarbeitung in der Grafik-Pipeline wird konfiguriert
  - Beispielzustände: Beleuchtung an/aus & Lichtquellen, Material, Texturen, Shader, …
  - klassisches OpenGL: rein-konfigurierbare Pipeline
    - z.B. nur Blinn-Phong-Beleuchtungsmodell und Gouraud-Shading (Interpolation von Vertex-Farben)
  - modernes OpenGL: frei programmierbare Stufen in der Pipeline, u.a. Geometrieverarbeitung, Schattierungsberechnung, Tessellierung mittels der OpenGL Shading Language (GLSL)
- klassisch ein Client-Server Konzept
  - heute: Client und Server auf einem Rechner
  - „Client“ = Applikation, Daten im Hauptspeicher
  - „Server“ = Grafiktreiber- und Karte, Graphics Processing Unit (GPU)
- Erweiterungen (**Extensions**) für Zugriff auf spezielle Fähigkeiten der GPUs

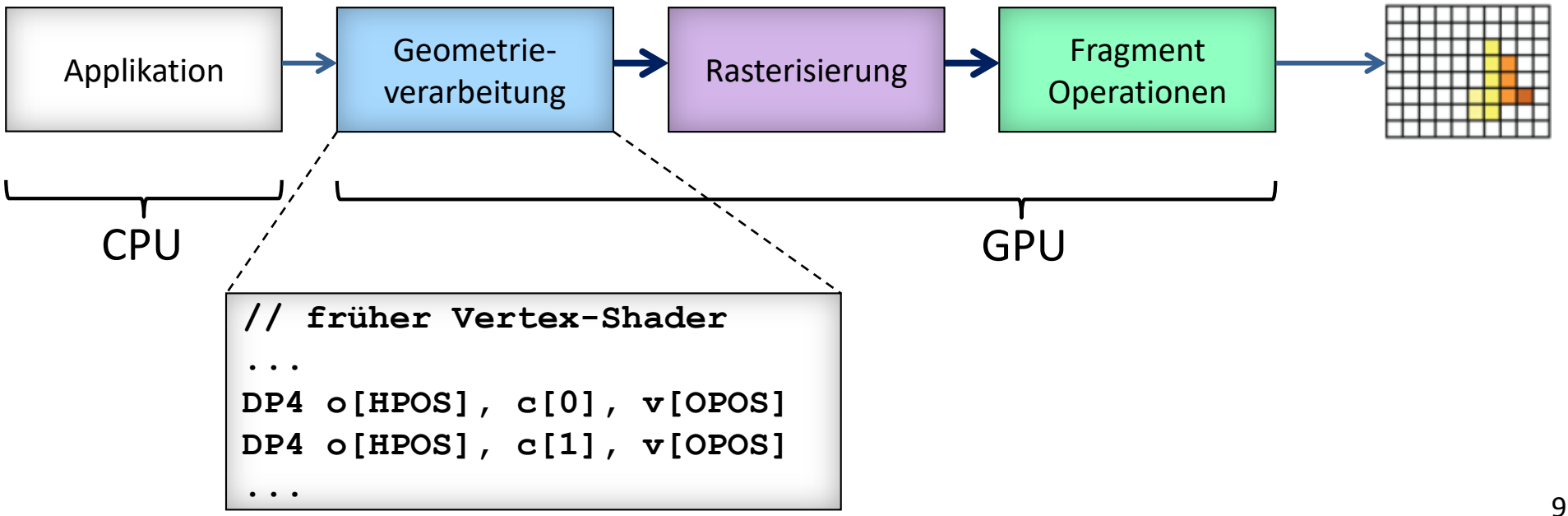
OpenGL – Historie und Entwicklung

- 1983 - 1992: SGI Graphics Library (GL), nahezu proprietär, Konkurrenz durch „Standards“: GKS-3D, PHIGS PLUS
- 1992: OpenGL 1.0
- 1992: OpenGL 1.2 (3D Texturen)
- 2001: OpenGL 1.3 (Multi-Texturen, Cube Maps)
- 2002: OpenGL 1.4 (Vertex Shader, GLSL)
- 2003: OpenGL 1.5 (Fragment Shader, Buffer Objects)
- 2004: OpenGL 2.0 (GL Shading Language, MRT), OpenGL ES: OpenGL for mobile and embedded systems
- 2009: OpenGL 3.0 (Entfernen von „Altlasten“)
- 2010: OpenGL 3.3 (OpenCL Einbindung)
- 2010: OpenGL 4.1 (Tesselierung, binäre Shader, 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, Multi-threading)
- 2017: OpenGL 4.6 (SPIR-V, effizienteres Batch-Rendering, …)



Grafik-Hardware

- heute: **große Teile der Pipeline frei programmierbar**
  (GPUs von vielen Herstellern: AMD/ATI, NVIDIA, Intel, Imagination, …)
  - früher Assembler-artig, heute Hochsprachen (C-ähnlich)
  - Geometrie-, Primitiv-, Fragmentverarbeitung, …
- Programmierung der GPU ohne die Grafik-Pipeline
  (OpenCL, CUDA, Compute Shader, …)
- letzte große Evolutionsstufe: native Unterstützung von Raytracing



OpenGL – in dieser Vorlesung

- das Wichtigste über klassisches OpenGL
  - → **zuhause vorbereiten, keine Details in der Vorlesung!**
  - OpenGL-Kommandos für Geometrie, Beleuchtung, …
  - einige (auch klassische) OpenGL-Konzepte behandeln wir in der Vorlesung
  - Literatur: **The OpenGL Programming Guide - The Redbook**
    HTML Version und Beispielprogramme:
    http://www.glprogramming.com/red/

- dann: modernes OpenGL (3+) und „Core Profiles“, sowie OpenGL ES
  - programmierbare Geometrie- und Fragment-Verarbeitung („Shader“)
    http://www.opengl.org/registry/doc/glspec46.core.pdf
    http://www.opengl.org/registry/doc/GLSLangSpec.4.60.pdf
  - → wir lernen Shading Language anhand von Beispiel kennen,
    weitere Folien bitte ebenfalls ansehen (bzw. in der Übung verwenden)
  - Vorteile: keine Altlasten, viel Kontrolle über die Pipeline,
    keine konzeptionellen Unterschiede zu anderen modernen APIs
  - Nachteile: „convenience features“ gehen verloren
  - in den Übungsaufgaben: OpenGL 3.x
  - keines der klassischen Konzepte verliert seine Bedeutung

Disclaimer

- die Folien dieses Kapitels enthalten
  viel Programm-Code und einige OpenGL-Befehle

- (1) viele der zugrundeliegenden Konzepte (z.B. Transformationen, Matrix-Stack etc.) kennen Sie schon, hier dienen die Folien als Verweis auf die entsprechenden OpenGL-Kommandos

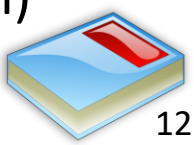
- (2) trotzdem werden wir einige Code-Beispiele besprechen, die die OpenGL-Programmierung und -Konzepte verdeutlichen

- (3) und wir lernen neue Konzepte kennen, wie z.B. Blending, Stencil-Buffering, Accumulation-Buffers, Shader …

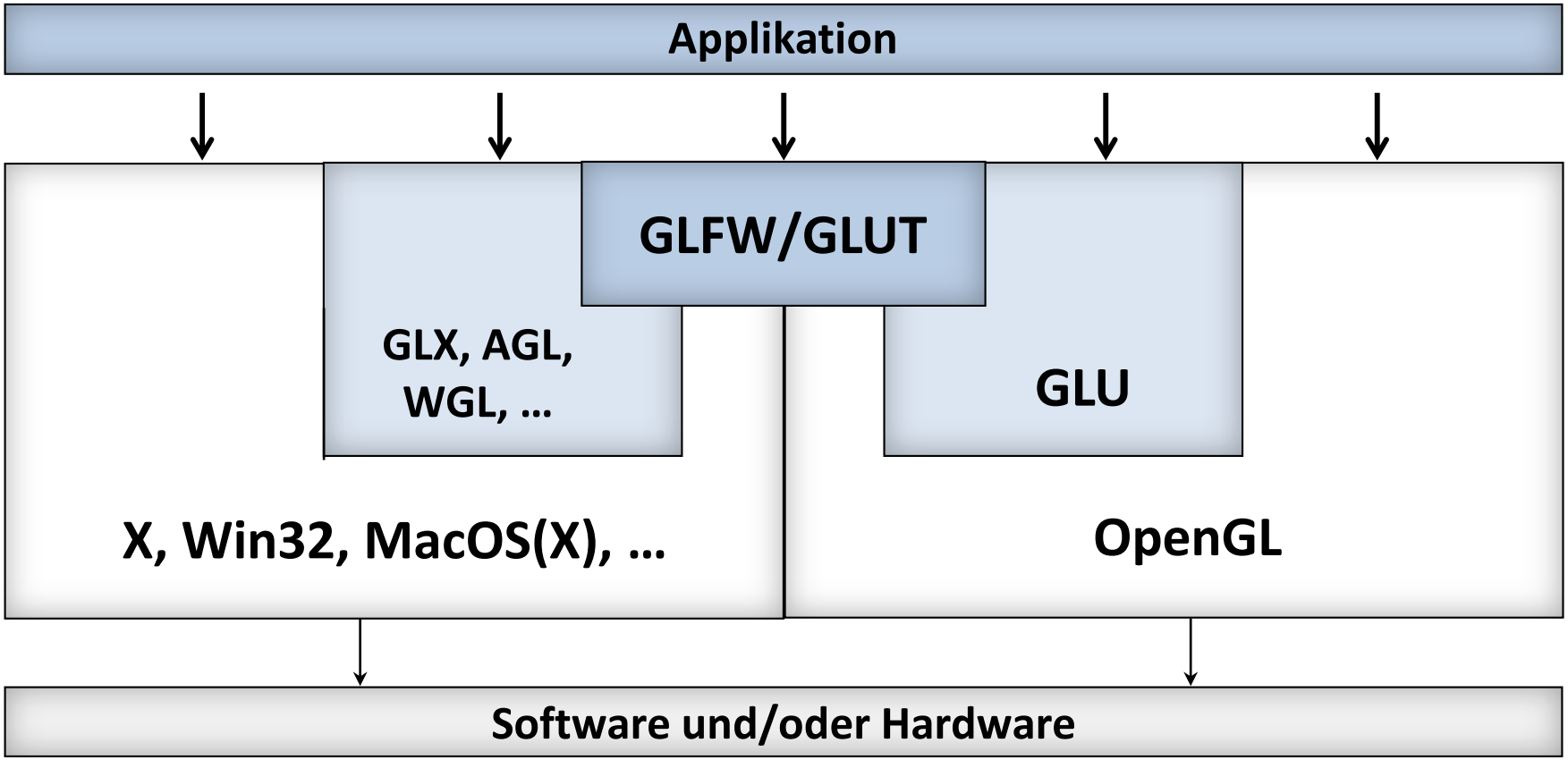
- OpenGL lernen Sie **nur** durch **Ausprobieren und Experimentieren** mit Beispielprogrammen (hören Sie trotzdem in die Vorlesung…)

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) oder OpenGL Extension Wrangler (GLEW)



APIs rund um OpenGL



Klassisches OpenGL

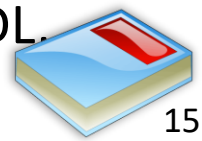
GLUT: Grundlagen

Struktur einer freeGLUT/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

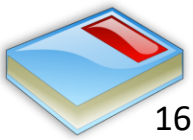
```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
```

- freeGlut: http://freeglut.sourceforge.net
- weiterer sehr ähnlicher Wrapper: GLFW – andere Konzepte: Qt, SDL



GLUT: Beispiel

```
void main( int argc, char** argv ) {

  glutInit( &argc, argv );

  // initialisiere/öffne Fenster mit Render-Kontext
  int mode = GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH;
  glutInitDisplayMode( mode );
  glutInitWindowSize ( 1920, 1080 );
  glutCreateWindow( “Mein GLUT Programm” );

  // Initialisiere OpenGL States, konfiguriere Pipeline
  init();

  // Callback Funktionen registrieren
  glutDisplayFunc ( display  );
  glutReshapeFunc ( resize   );
  glutKeyboardFunc( keyboard );
  glutMouseFunc   ( mouse    );
  glutIdleFunc    ( idle     );
  glutMainLoop();
}
```



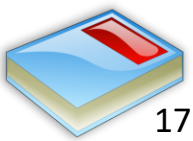
Double-Buffering

Bestandteile eines Framebuffers in OpenGL

- 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)

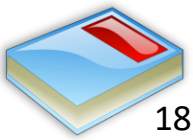
- **Double Buffering** (**GLUT_DOUBLE**) bedeutet es gibt zwei Color Buffer
  - Front Buffer, der das gerade dargestellte Bild enthält und
  - Back Buffer, unsichtbarer Puffer in dem das nächste Bild erzeugt wird
  - es genügt ein Z-Buffer (für das dargestellte Bild benötigt man keinen)

- das Umschalten („Swapping“) erfolgt, wenn das Rendering abgeschlossen ist mit `glutSwapBuffers();`
  (mit oder ohne VSYNC)

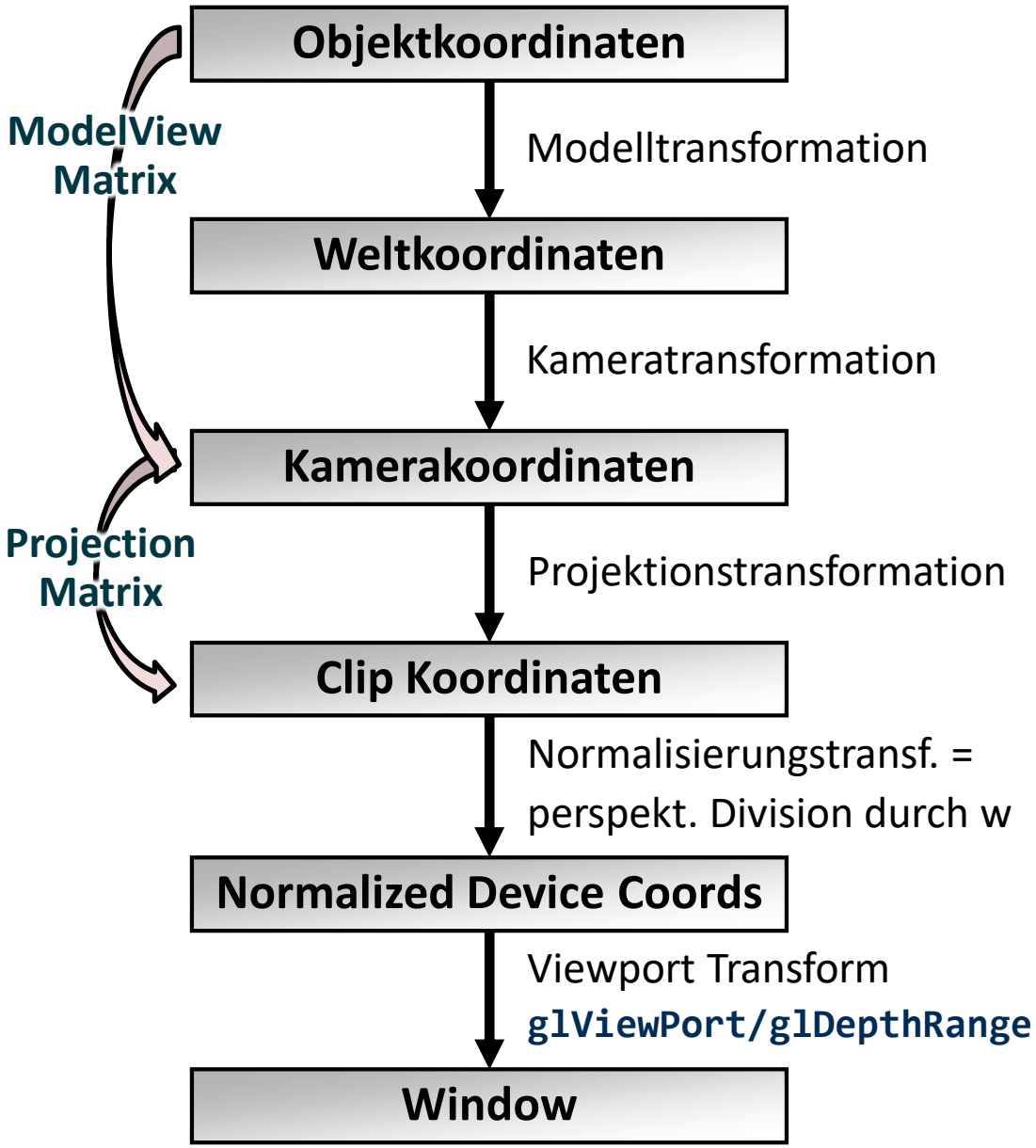


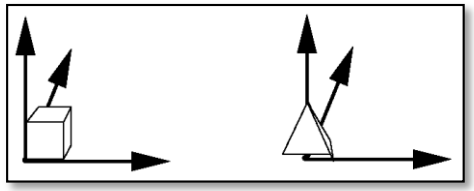
GLUT: Beispiel *cont.*

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

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

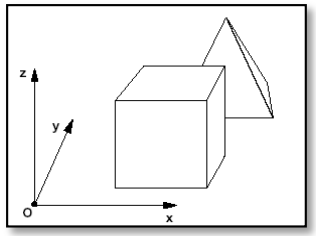


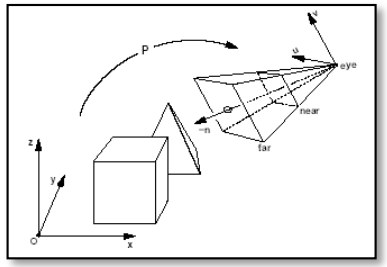
Koordinatensysteme

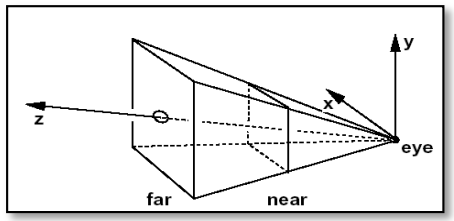


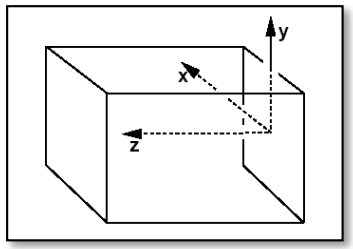


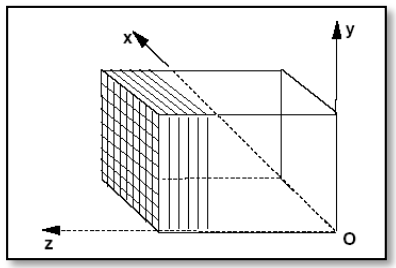
```
glVertex()
glNormal()
```











Mehr Infos:
http://www.opengl.org/resources/faq/technical/viewing.htm

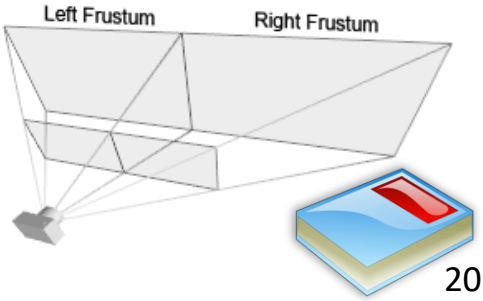
http://www.songho.ca/opengl/gl_transform.html

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 );
}
```

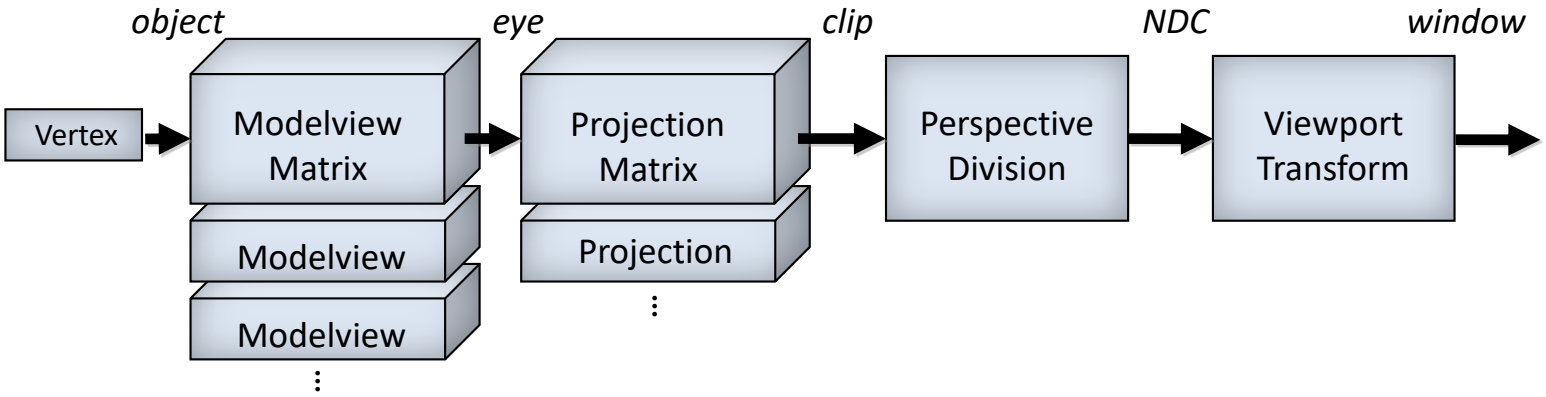
- Projektionstransformationen in OpenGL
  (**glFrustum** erlaubt nicht-symmetrische Sichtpyramiden)

```
gluPerspective( fovy, aspect, zNear, zFar )
glFrustum( left, right, bottom, top, zNear, zFar )
glOrtho  ( left, right, bottom, top, zNear, zFar )
```



Transformationen: Pipeline

- es gibt Matrix Stacks für Modelview, Projektions- und Textur-Matrix (letztere zur automatischen Erzeugung von Texturkoordinaten, heute praktisch nur in Shader umgesetzt)



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

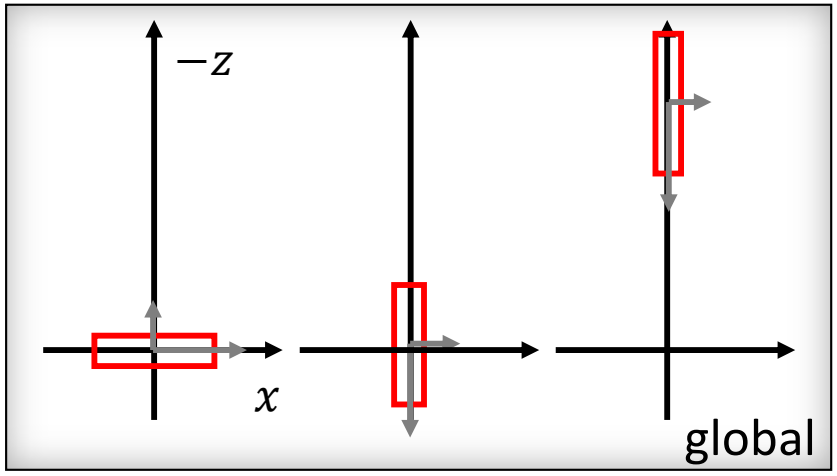
Zusammengesetzte Transformationen

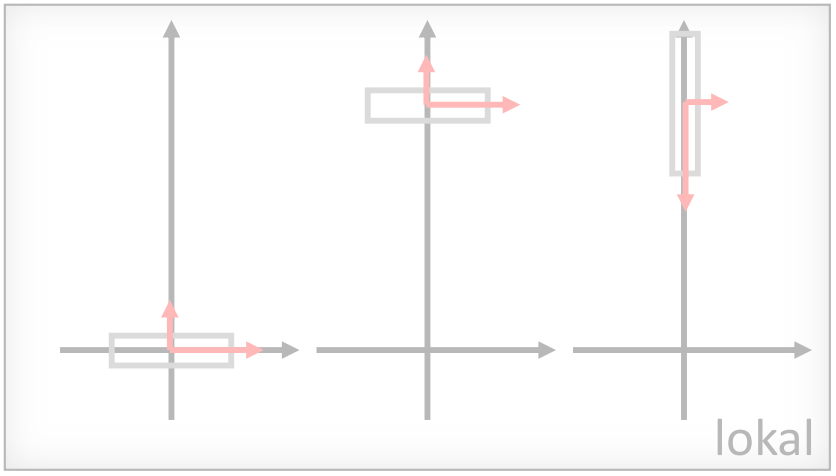
- globale Sicht („wie transformieren wir das Objekt im KoSys“)
  - platziere/bewege Objekte relativ zum Ursprung der Weltkoordinaten
  - Reihenfolge im Code rückwärts
- lokale Sicht („wie verändert sich das KoSys, in dem das Objekt liegt“)
  - bewege lokale Koordinatensysteme der Objekte
  - Reihenfolge im Code vorwärts

global ↑

```
glTranslate(  0.0, 0.0, -5.0 )
glRotate    ( 90.0, 0.0, 1.0, 0.0 )
drawobject () // in Modellkoord.
```

↓ lokal





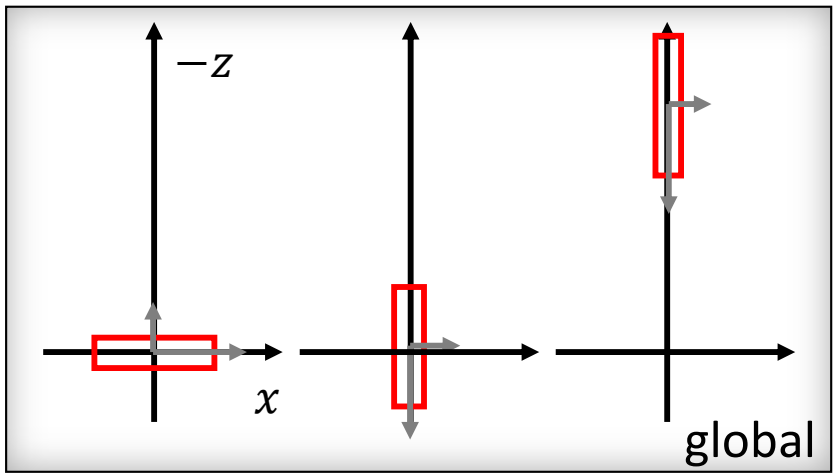
Zusammengesetzte Transformationen

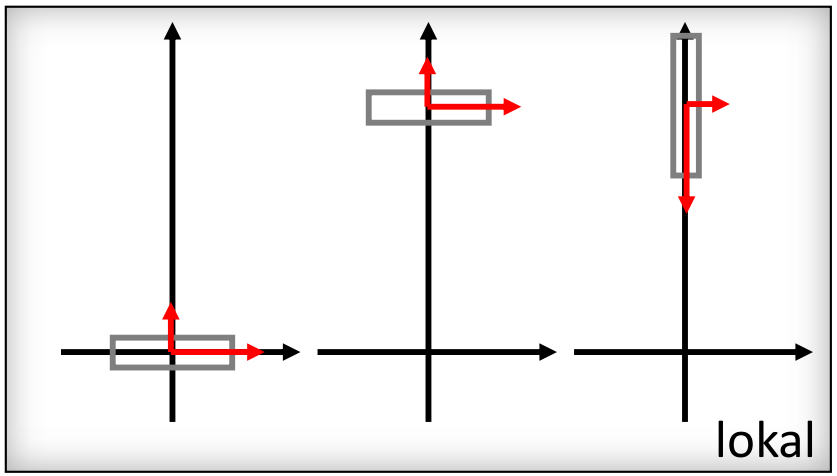
- globale Sicht („wie transformieren wir das Objekt im KoSys“)
  - platziere/bewege Objekte relativ zum Ursprung der Weltkoordinaten
  - Reihenfolge im Code rückwärts
- lokale Sicht („wie verändert sich das KoSys, in dem das Objekt liegt“)
  - bewege lokale Koordinatensysteme der Objekte
  - Reihenfolge im Code vorwärts

global ↑

```
glTranslate(  0.0, 0.0, -5.0 )
glRotate    ( 90.0, 0.0, 1.0, 0.0 )
drawobject () // in Modellkoord.
```

↓ lokal





Hierarchische Modellierung

Beispiel

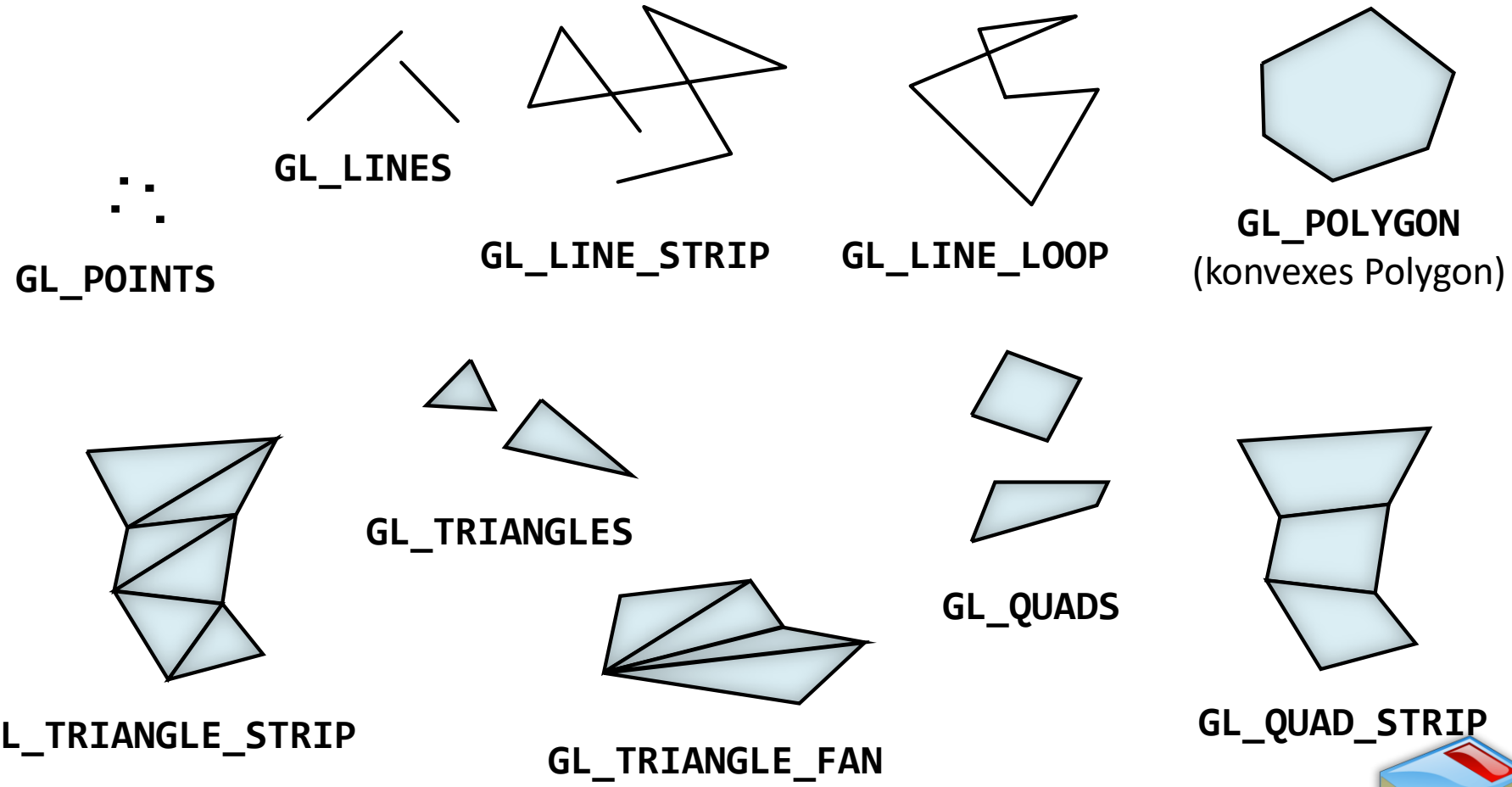
- verwende Push/Pop auf ModelView-Matrix-Stack

```
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
}
```

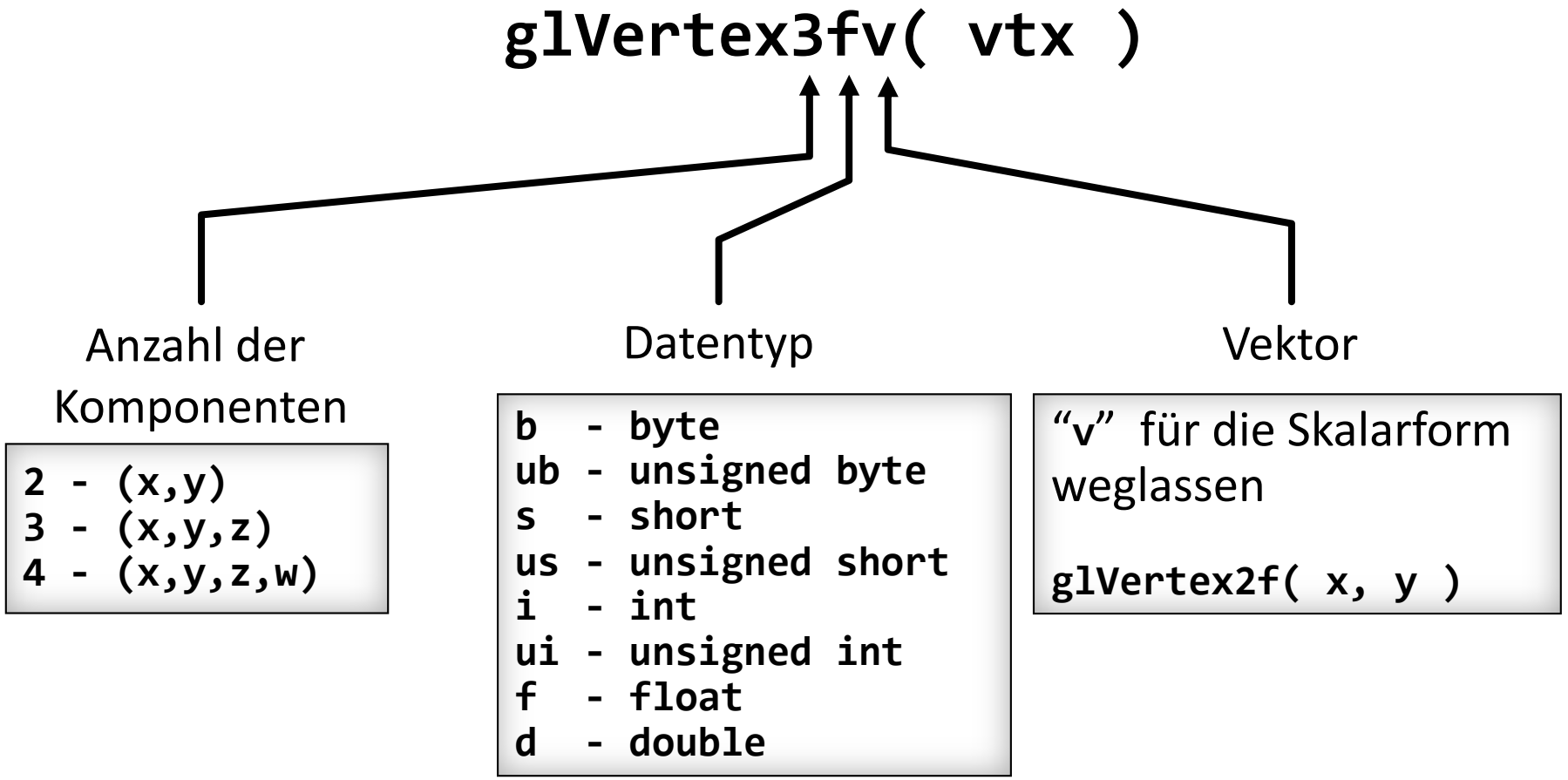
```
draw_wheel_and_bolts()
{
 draw_wheel();
 for( int i = 0; i < 5; i++) {
  glPushMatrix();
  glRotatef( 72.0 * i,
             0.0, 0.0, 1.0);

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

OpenGL: Geometrische Primitive

- geometrische Primitive werden durch **Vertices in homogenen Koordinaten** definiert
- wird die homogene Koordinate nicht angegeben, gilt $w = 1$



glVertex-Vielfalt

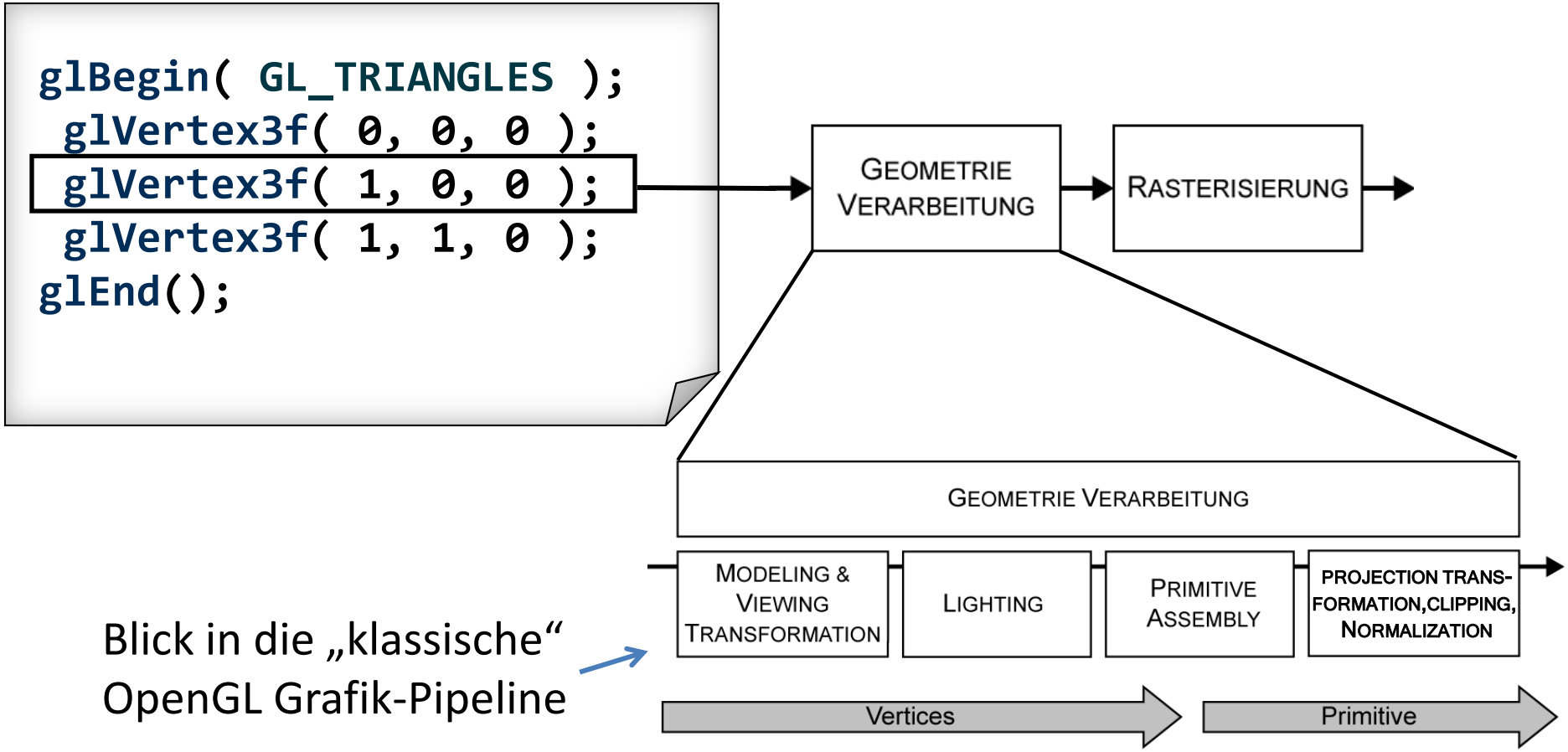


- Schreibweise auf den Folien manchmal:
  `glVertex*(...)` oder einfach nur **`glVertex(...)`**

Vertices 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

```
glBegin( GL_TRIANGLES );
 glVertex3f( 0, 0, 0 );
 glVertex3f( 1, 0, 0 );
 glVertex3f( 1, 1, 0 );
glEnd();
```



Blick in die „klassische“ OpenGL Grafik-Pipeline

GLUT: Beispiel *cont.*

```
// 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();
}
```

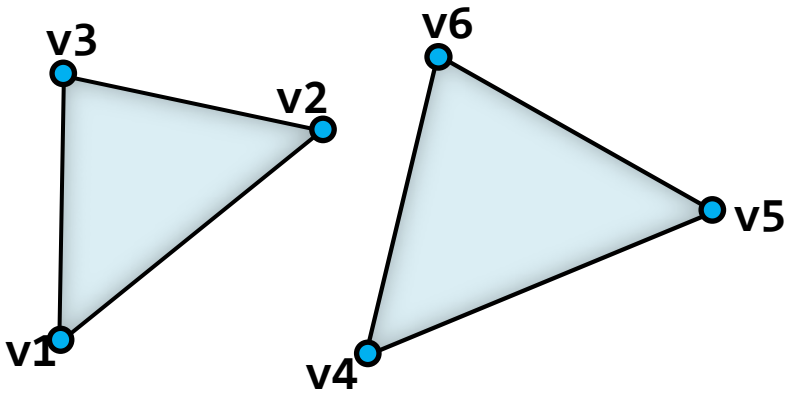
Geometrische Primitive

Isolierte/unabhängige Dreiecke: $3n$ Vertices für n Dreiecke

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

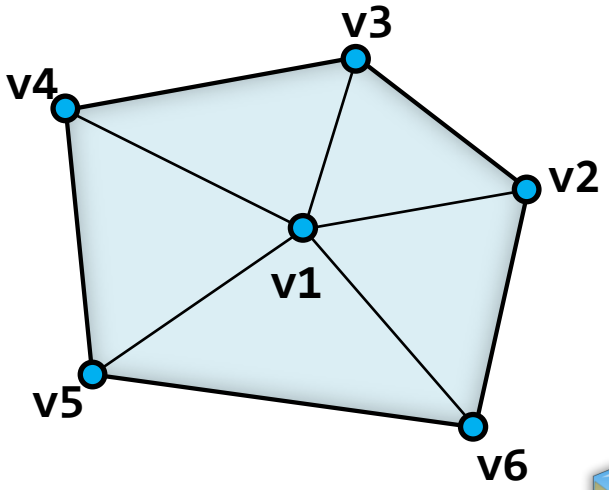
Dreieck 1

Dreieck 2



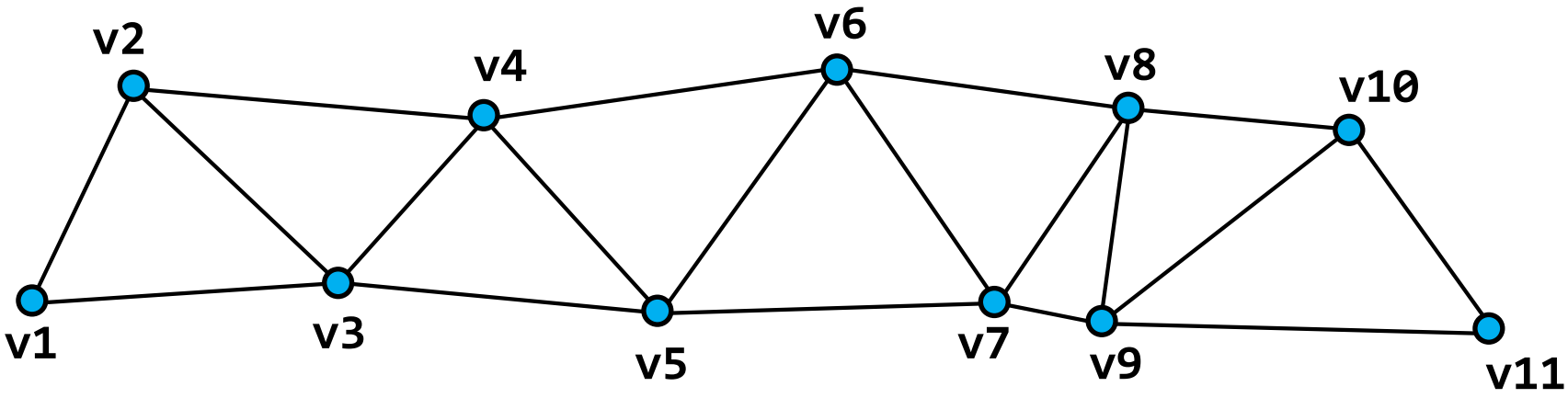
Dreiecksfächer, Triangle Fans: $n + 2$ Vertices für n Dreiecke (n typ. klein)

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



Geometrische Primitive

Dreiecksstreifen, Triangle Strips




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

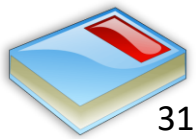
Dreieck 1
Dreieck 2
Dreieck 2

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

Primitive und Vertex-Attribute

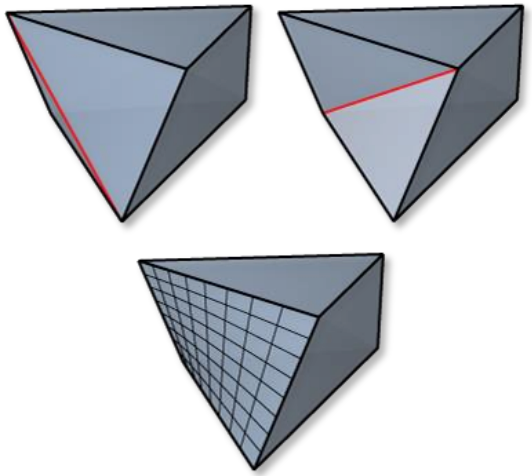
- Vertices werden verbunden wie angegeben mit
  `glBegin( primType ); ...; glEnd();`
- Zeichnen von Dreiecken mit zusätzlichen Attributen
  - Vertex-Attribute in OpenGL: **`glColor*`**, **`glNormal*`**, **`glTexCoord*`**
  - es gelten jeweils die Attribute die vor **`glVertex`** gesetzt wurden
  - globale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

```
GLfloat red, greed, blue;
GLfloat coords[ 3 * nVerts];

glBegin( primType );

for ( int i = 0; i < nVerts; ++i )
{
  glColor3f( red, green, blue );
  glVertex3fv( &coords[ 3 * i ] );
}

glEnd();
```



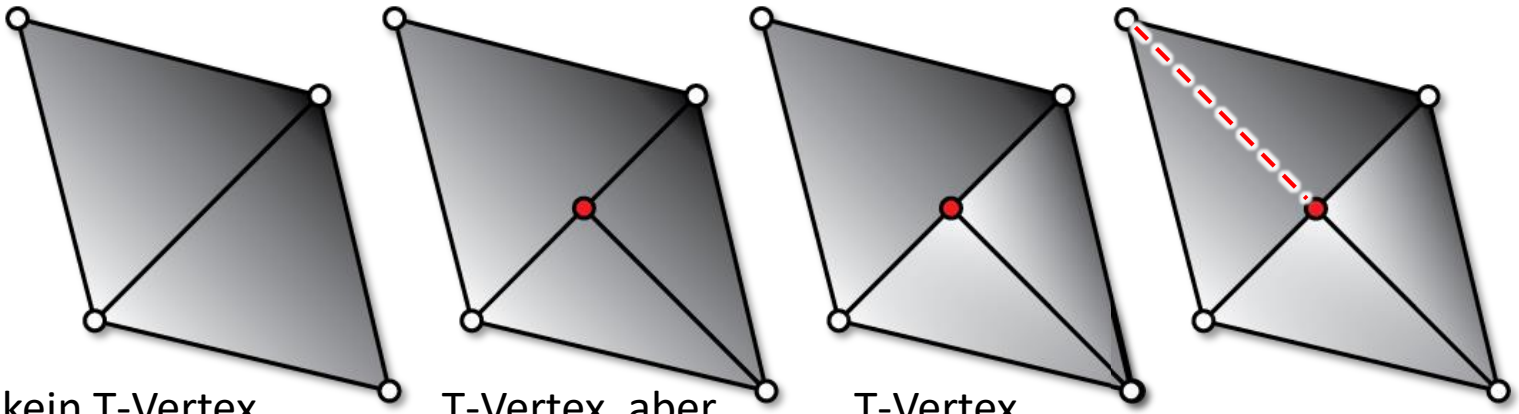
Dreiecke und Polygone *cont.*

Weitere Aspekte bei der Geometrie



- **GL_POLYGON**, **GL_QUADS**, … werden automatisch trianguliert, vermeide nichtebene Polygone

- vermeide sogenannte **T-Vertices**
  - führt zu Löchern und Shading-Fehlern
  - füge Kante - - - - ein



kein T-Vertex

T-Vertex, aber Schattierung ok

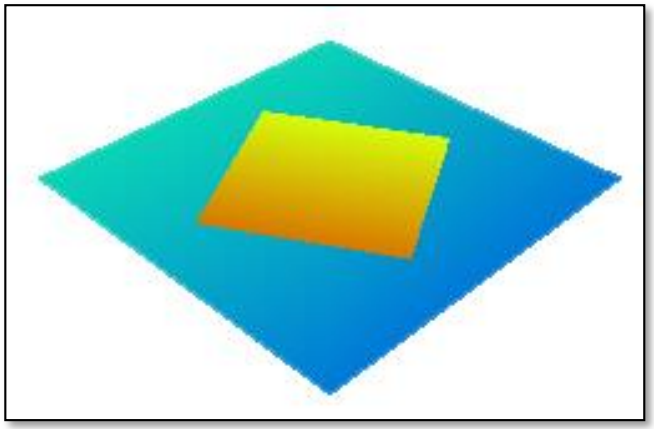
T-Vertex, typische Probleme mit der Schattierung

Dreiecke und Polygone *cont.*

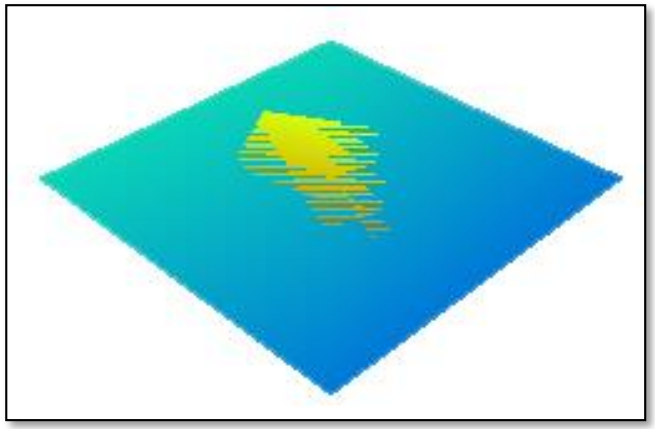
Z-Fighting

- 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 $\mathit{factor} \cdot \Delta z$ 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“

Backface Culling

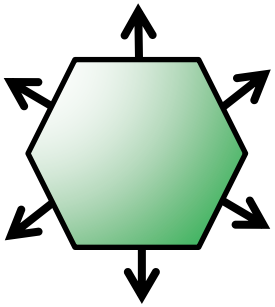
Rasterisierung, Vorder- und Rückseiten, Backface Culling

- bei Dreiecken und Polygonen wird Vorder- und Rückseite unterschieden
  - übliche Festlegung: man sieht ein Polygon von vorne, wenn die Vertices 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 );
```

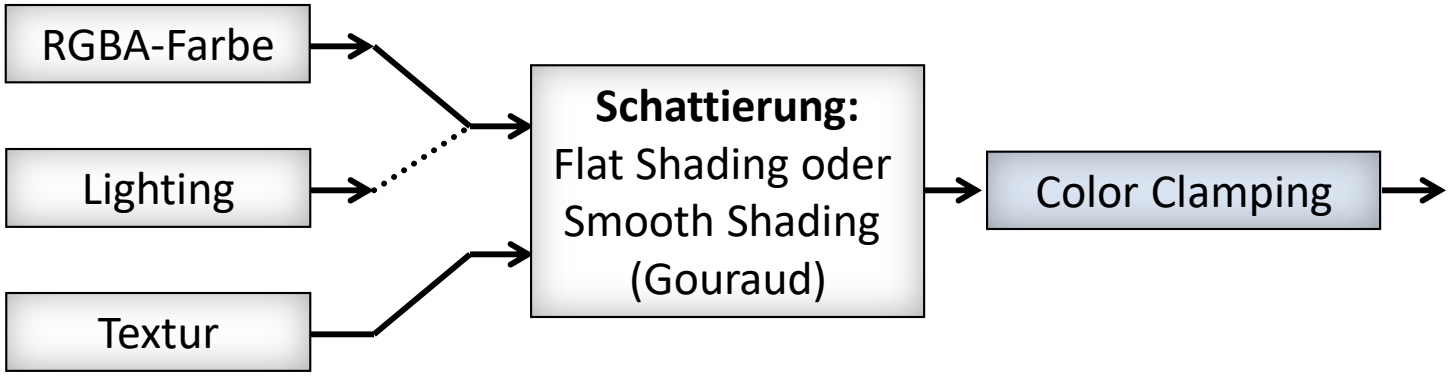
  - bei geschlossenen Dreiecksnetzen sind die Rückseiten nicht sichtbar
  - man spart Rasterisierungsaufwand, aber keine Transformationen



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`)

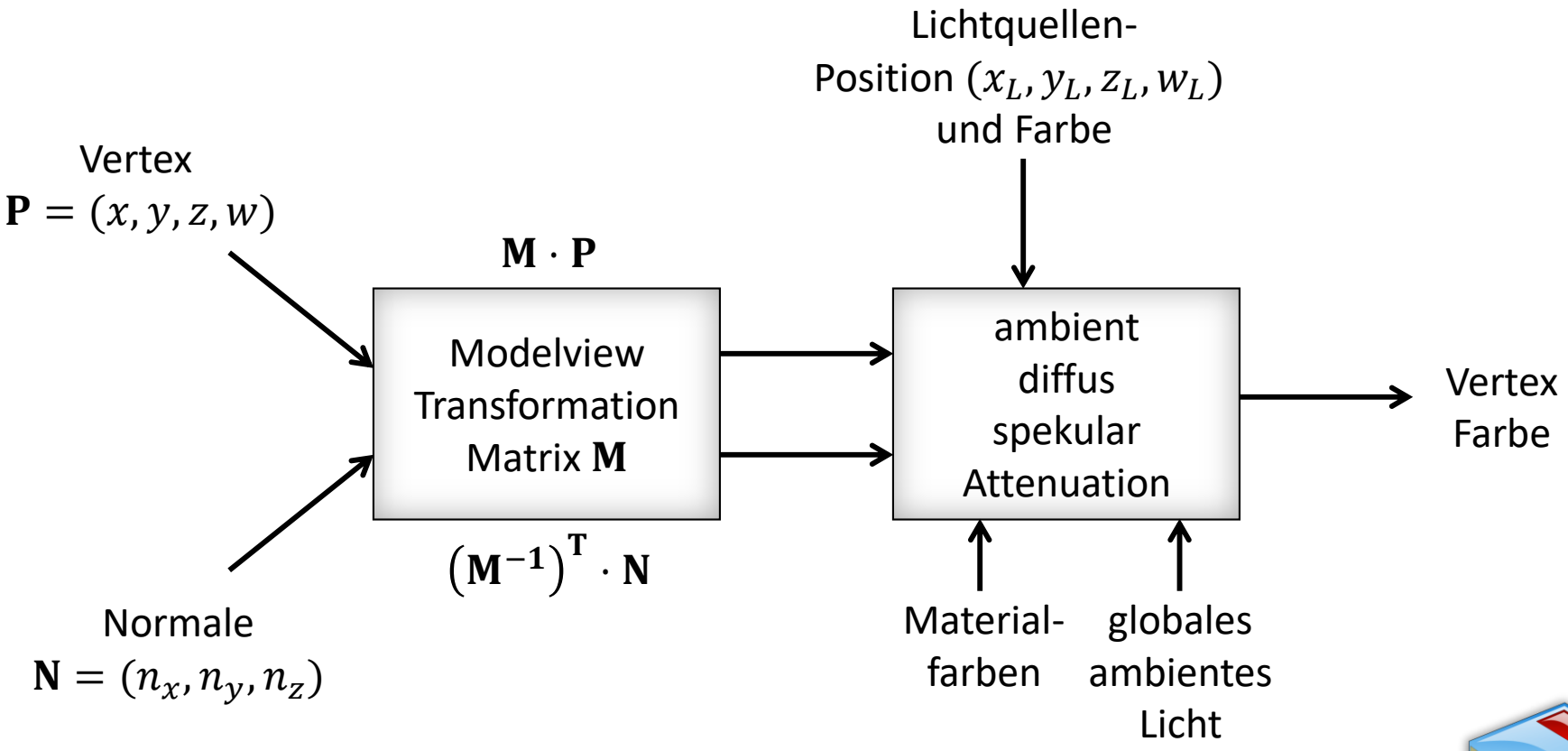
- Einschalten der Beleuchtungsberechnung mit
  `glEnable( GL_LIGHTING )`



- Kombination von Schattierung und Textur über Texture Environments:
  `glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE)`

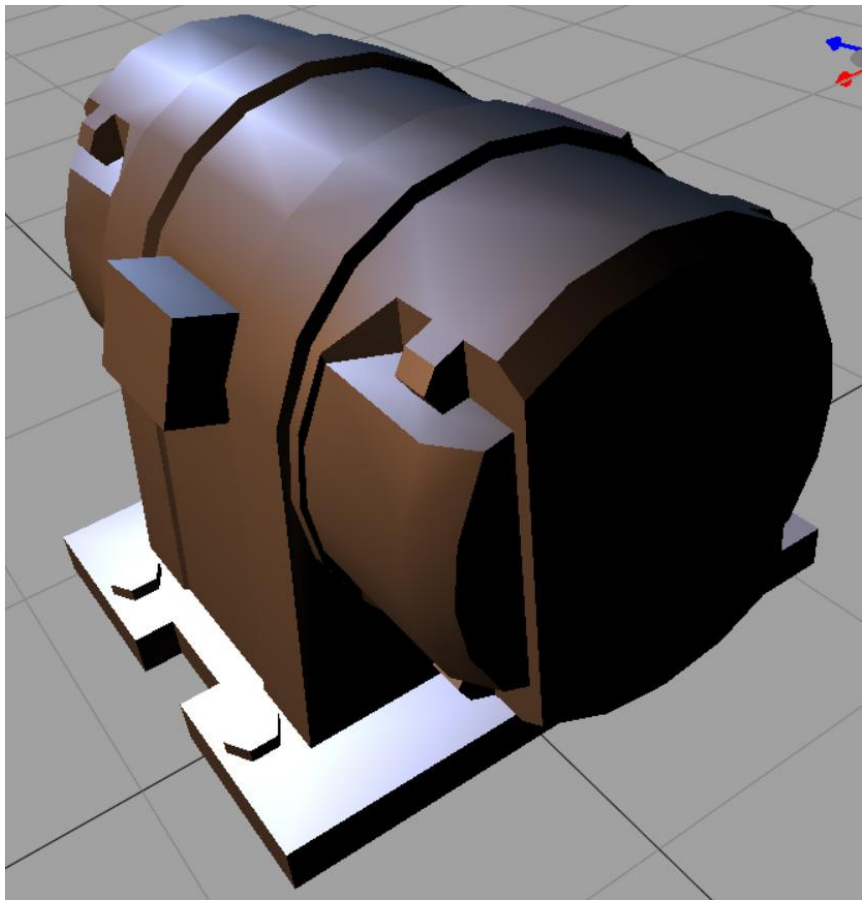
Beleuchtungsberechnung

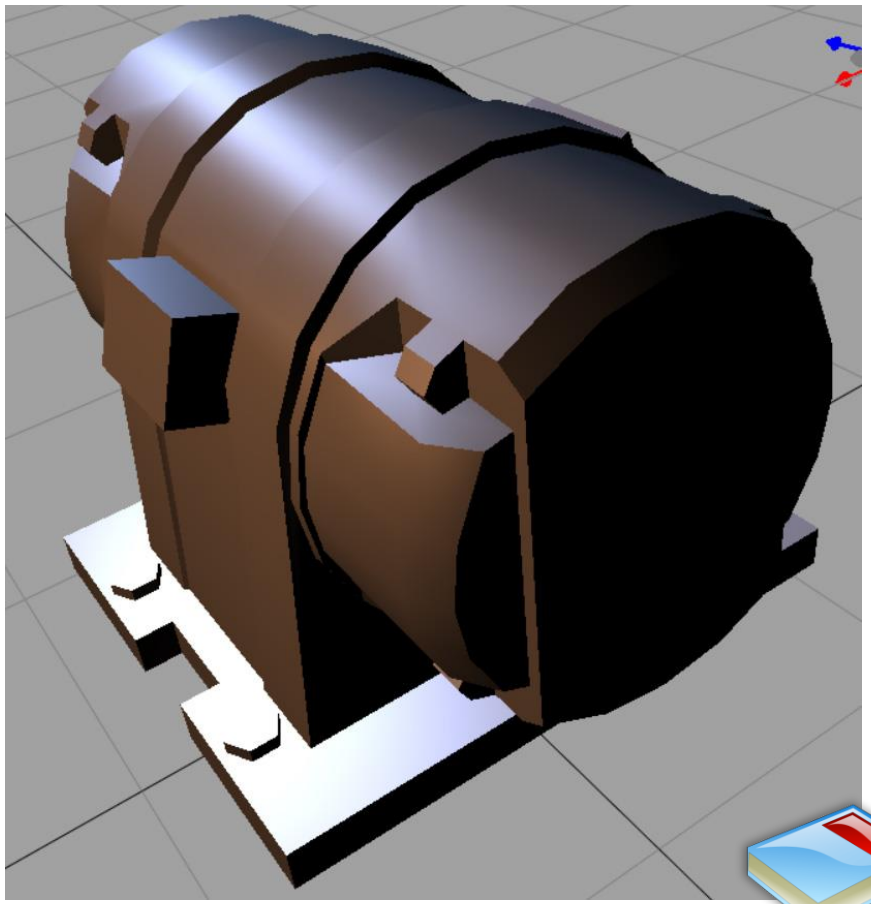
- Beleuchtungsberechnung in **Kamerakoordinaten** nach der Modelview-Transformation entweder pro Vertex (für sog. Smooth Shading) oder pro Dreieck (Flat Shading)
- OpenGL unterstützt nativ keine Beleuchtungsberechnung pro Pixel, also kein Phong Shading, nur Gouraud Shading



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

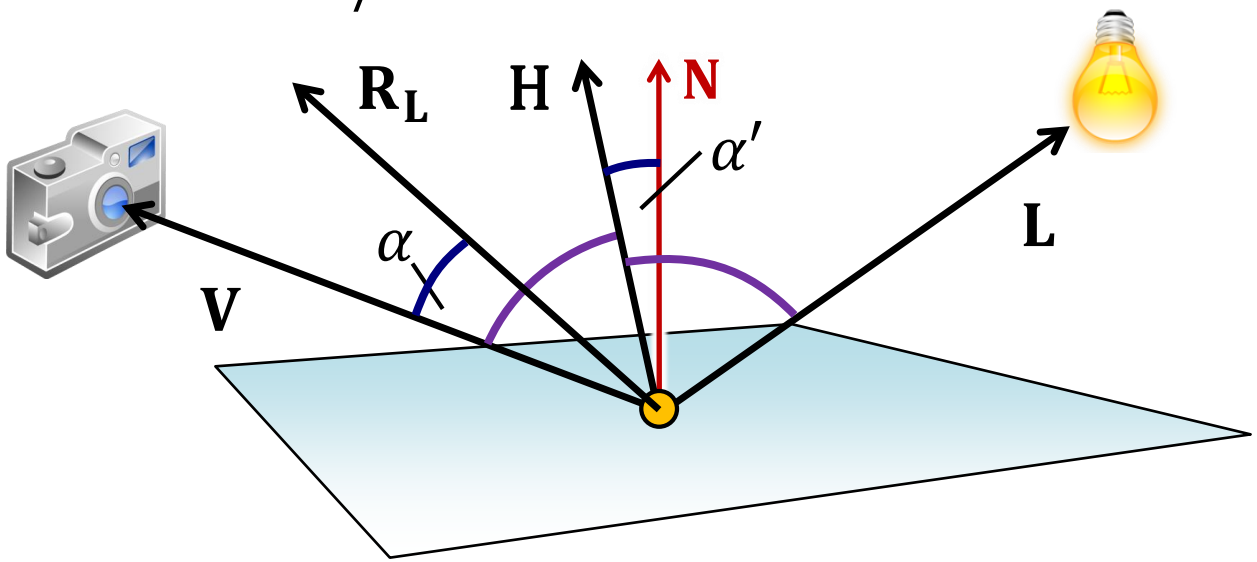


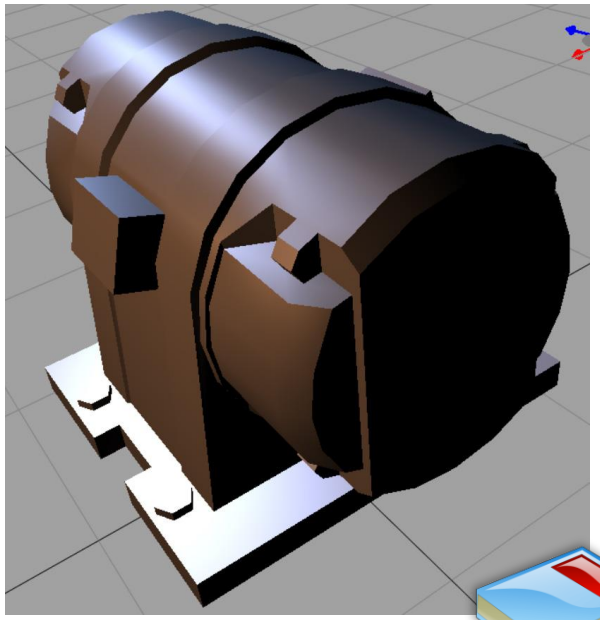


OpenGL verwendet das Blinn-Phong-Modell

Abwandlung des Phong-Modells (betrifft nur den spekularen Teil)

- original Phong-Modell: $I_s = k_s \cdot I_\mathrm{L} \cdot \cos^n \alpha = k_s \cdot I_\mathrm{L} \cdot (\mathbf{R_L} \cdot \mathbf{V})^n$ mit spekularem Reflexionskoeffizient k_s und Phong-Exponent n
- Blinn-Phong-Modell verwendet keinen Reflexionsvektor $\mathbf{R_L}$, sondern den Halfway-Vektor $\mathbf{H} = \frac{\mathbf{V+L}}{\|\mathbf{V+L}\|}$ und einen Exponent n'
  - spekularer Anteil $I_s = k_s \cdot I_\mathrm{L} \cdot \cos^n \alpha' = k_s \cdot I_\mathrm{L} \cdot (\mathbf{N} \cdot \mathbf{H})^{n'}$
  - $\mathbf{H} = const$ bei ∞-weit entferntem Betrachter/orthographischer Kamera und direktionaler LQ
  - $\alpha' \approx \alpha/2 \Rightarrow$ wähle $n' \approx 4n$





Blinn-Phong-Beleuchtung in OpenGL (erweitert)

- Intensität I eines Vertex
  - ambiente, diffuse und spekulare Lichtintensität I_{a_λ}, I_{d_λ}, I_{s_λ} (bis zu 8 Lichtquellen)
  - ambiente, diffuse, spekulare Reflexion des Materials O_{a_λ}, O_{d_λ}, O_{s_λ}
  - Emission des Materials O_{E_λ}, globales ambientes Licht I_{a_λ}

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

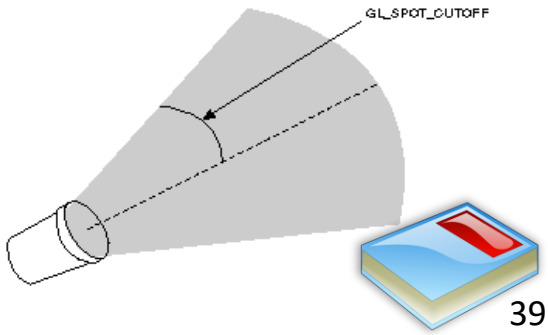
$$+ \sum_i f^i \cdot spot^i \left(I^i_{a_\lambda} \cdot O_{a_\lambda} + \max(\mathbf{N} \cdot \mathbf{L}, 0) \cdot I^i_{d_\lambda} \cdot O_{d_\lambda} + \max(\mathbf{N} \cdot \mathbf{H}, 0)^n \cdot I^i_{s_\lambda} \cdot O_{s_\lambda} \right)$$

- Abschwächung (Attenuation) für Vertex mit Abstand d

$$f^i = \left(k_c + k_l d + k_q d^2 \right)^{-1}$$

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

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



OpenGL-Beleuchtung

- Licht einschalten
  `glEnable( GL_LIGHTING )`
  `glEnable( GL_LIGHTn )`

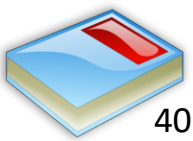
- Lichtquellen-Eigenschaften
  `glLightf{v}( light, property, value )`

- Material-Eigenschaften
  `glMaterialfv( face, property, value )`

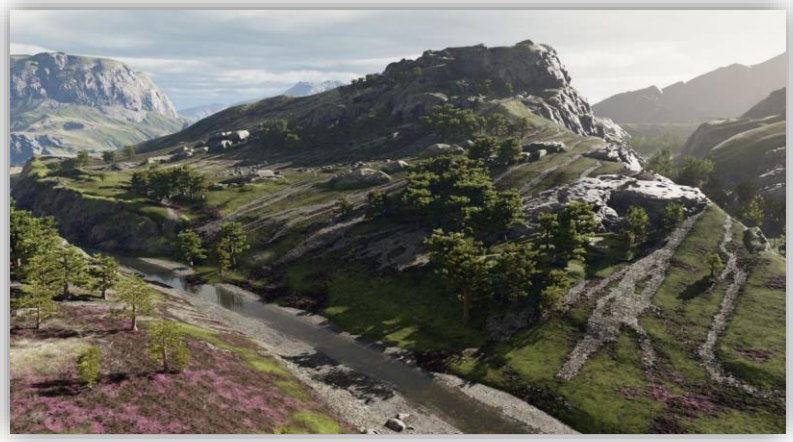
- Lichtquellen-Arten
  - Punktlichtquelle (Point Light): strahlt in alle Richtungen
  - Spotlight: Vorzugsrichtung
  - im Unendlichen (Directional Light; gerichtet):
    $w = 0$ in Position (homogene Koordinaten)
- eventuell beide Seiten von Polygonen beleuchten
  `glEnable( GL_LIGHT_MODEL_TWO_SIDE )`

GL_AMBIENT
GL_DIFFUSE
GL_SPECULAR
GL_POSITION
GL_SPOT_*
GL_*_ATTENUATION

GL_AMBIENT
GL_DIFFUSE
GL_SPECULAR
GL_SHININESS
GL_EMISSION



Auch das geht mit OpenGL…



Modernes OpenGL – Performance

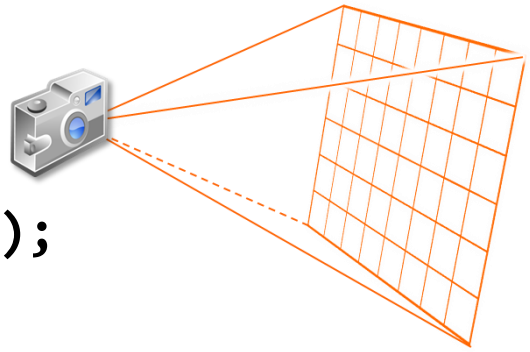
- die OpenGL-API hat sich mit der Hardware weiterentwickelt
  - extrem hohe Rechenleistung von GPUs (viele TFLOPS single precision)
  - schnelle Anbindung an dedizierten GPU-Speicher (>900 GB/s)
  - vgl. geringe Busbandbreite zum Hauptspeicher (64 GB/s bei PCIe 4.0)
  - zu hoher Overhead mit klassischer OpenGL API (`glVertex`-Aufrufe, …)

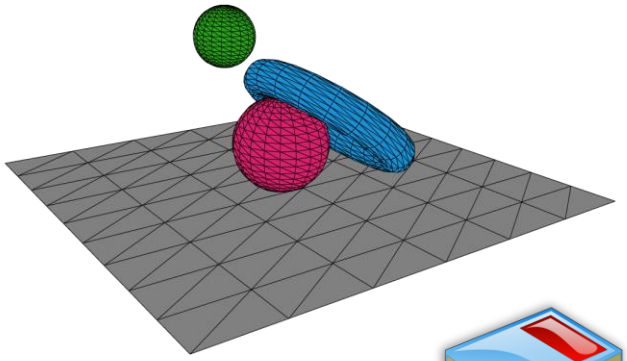
- wesentliche Änderungen mit dem Ziel einer schlanken, effizienten API
  - kein klassischer Immediate Mode (`glBegin`, `glVertex`, `glEnd`) mehr: Daten werden in Puffern (im Speicher der Grafikhardware) abgelegt
  - keine Hilfsfunktionen (z.B. Matrix Stacks) mehr
  - Ziel: keine **Fixed Function-Verarbeitung von Vertices/Fragmenten**
    - nur Kernaufgaben (z.B. Rasterisierung) bleiben Fixed Function
    - dafür: frei programmierbar über sogenannte „Shader“

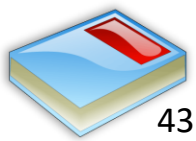
- **Kompatibilitätsprofil**: Fixed Function Pipeline existiert nebenher
- **Core Profile**: entrümpelt, alle alte Funktionalität muss selbst programmiert werden

OpenGL ≤ 2.0: Low-level, immediate mode API

```
// Framebuffer löschen
glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
...

// Pipeline konfigurieren
glEnable( GL_LIGHTING );
glEnable( GL_DEPTH_TEST );
...

// Transformationen
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 65., (float)w/h, 1., 100.);
...

// Rendering
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();
```



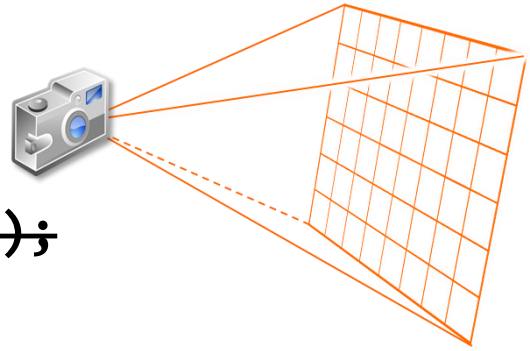


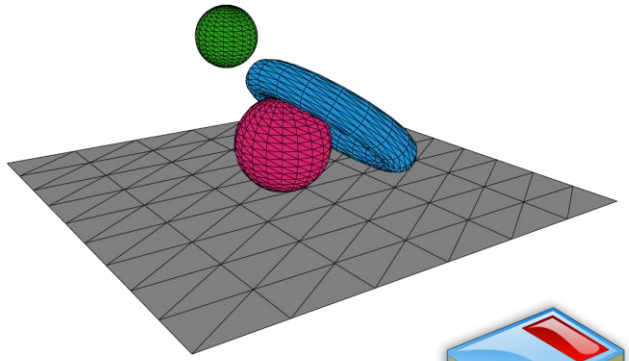


OpenGL ≤ 2.0: Low-level, immediate mode API

```
// Framebuffer löschen
glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
...

// Pipeline konfigurieren
glEnable( GL_LIGHTING );
glEnable( GL_DEPTH_TEST );
...

// Transformationen
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 65., (float)w/h, 1., 100.);
...

// Rendering
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();
```

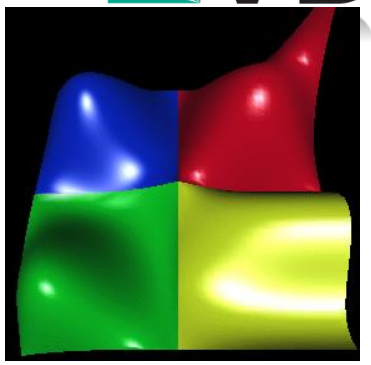
durchgestrichene Befehle sind in modernem OpenGL (Core Profile) nicht mehr vorhanden

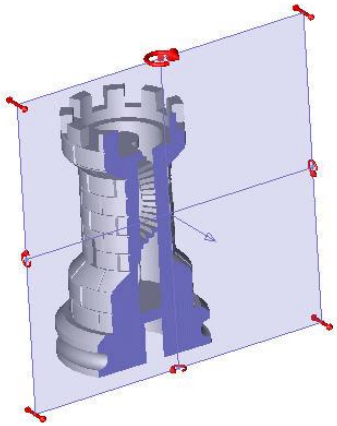


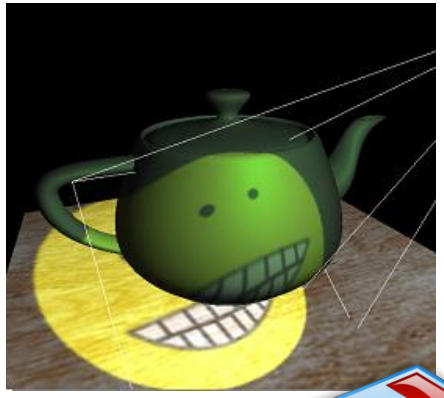


OpenGL ≤ 2.0: Low-level, immediate mode API

```
// OpenGL Evaluators: Beziér-Patches
GLfloat grid[2][2][3] = { ... };

glEnable( GL_MAP2_VERTEX_3 );
glMap2f( GL_MAP2_VERTEX_3, 0., 1.,
         3, 2, 0., 1., 6, 2, grid );
glMapGrid2f( 5, 0.0, 1.0, 6, 0.0, 1.0);
glEvalMesh2( GL_FILL, 0, 5, 0, 6 );
...

// Clip-Planes (Cut-away views)
Gldouble eqn[4] = {};
glClipPlane( GL_CLIP_PLANE0, eqn );
glEnable( GL_CLIP_PLANE0 );
...

// Texturkoordinatengenerierung
glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE,
            GL_OBJECT_LINEAR);
glTexGendv( GL_S, GL_OBJECT_PLANE, eqn );
glEnable( GL_TEXTURE_GEN_S );
```

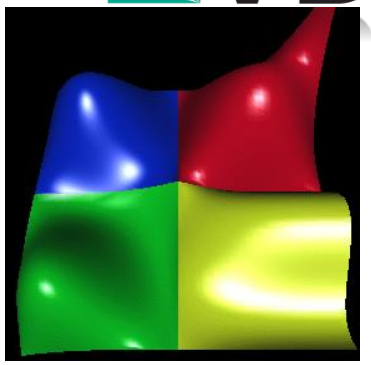


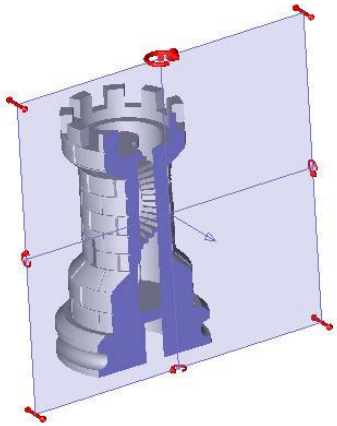


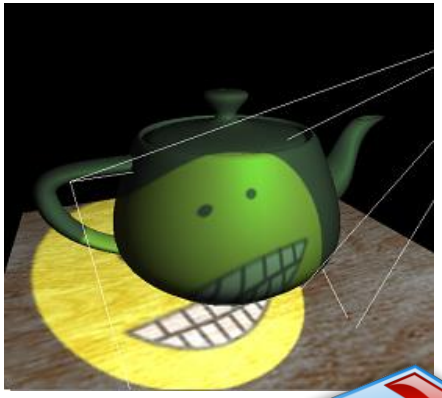


OpenGL ≤ 2.0: Low-level, immediate mode API

```
// OpenGL Evaluators: Beziér-Patches
GLfloat grid[2][2][3] = { ... };

glEnable( GL_MAP2_VERTEX_3 );
glMap2f( GL_MAP2_VERTEX_3, 0., 1.,
         3, 2, 0., 1., 6, 2, grid );
glMapGrid2f( 5, 0.0, 1.0, 6, 0.0, 1.0);
glEvalMesh2( GL_FILL, 0, 5, 0, 6 );
...

// Clip-Planes (Cut-away views)
Gldouble eqn[4] = {};
glClipPlane( GL_CLIP_PLANE0, eqn );
glEnable( GL_CLIP_PLANE0 );
...

// Texturkoordinatengenerierung
glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE,
            GL_OBJECT_LINEAR);
glTexGendv( GL_S, GL_OBJECT_PLANE, eqn );
glEnable( GL_TEXTURE_GEN_S );
```

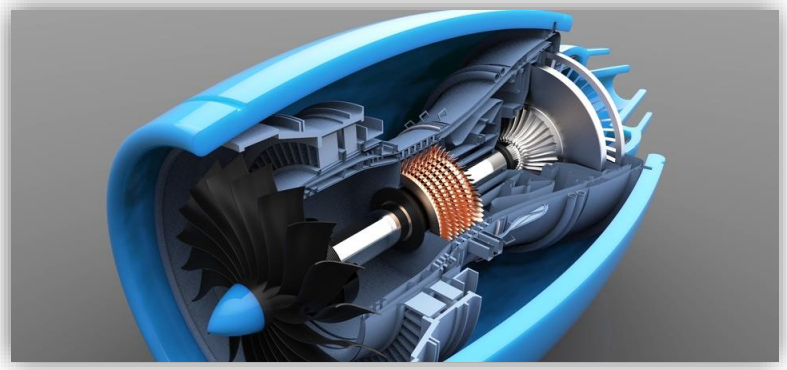


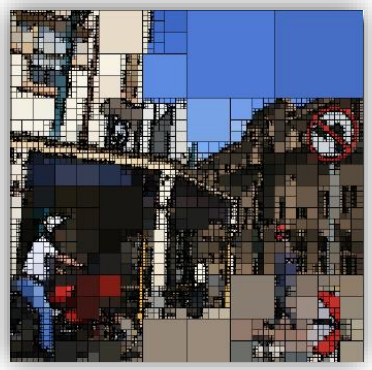


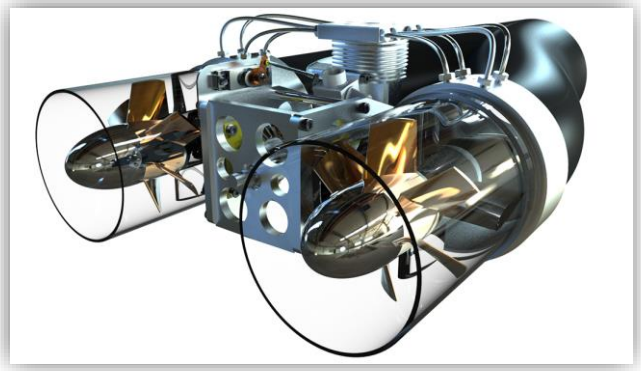


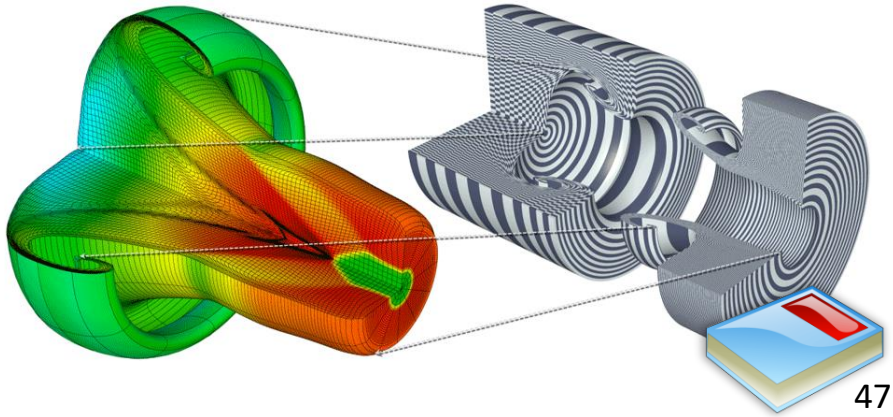
Modernes OpenGL – Funktionalität

- beliebige Beleuchtungsberechnung und Materialien programmierbar
  - nicht mehr an **`glMaterial`** / **`glLight`** gebunden
  - prozedurale Texturierung direkt auf der Grafik-HW
  - komplexe Datenstrukturen für Texture Mapping, z.B. Octrees
- komplexe Geometrieverarbeitung und Animation (Skinning, …)







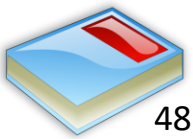


Modernes OpenGL

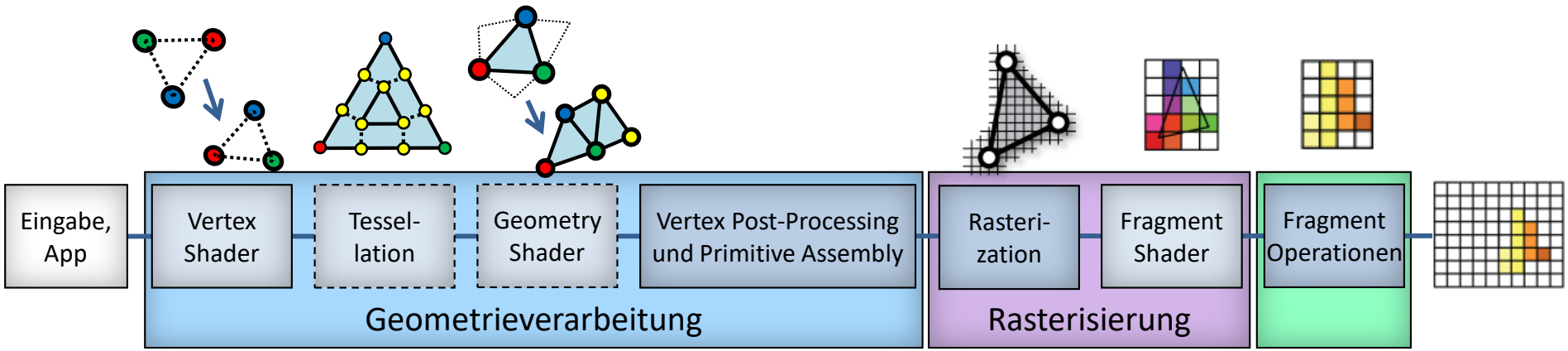
OpenGL 2.x/3.x/4.x, OpenGL ES 2.x/3.x

(entspricht der Funktionalität von DirectX 9, 10, 11, 12, …,

die Konzepte sind ebenso relevant für Vulkan etc.)

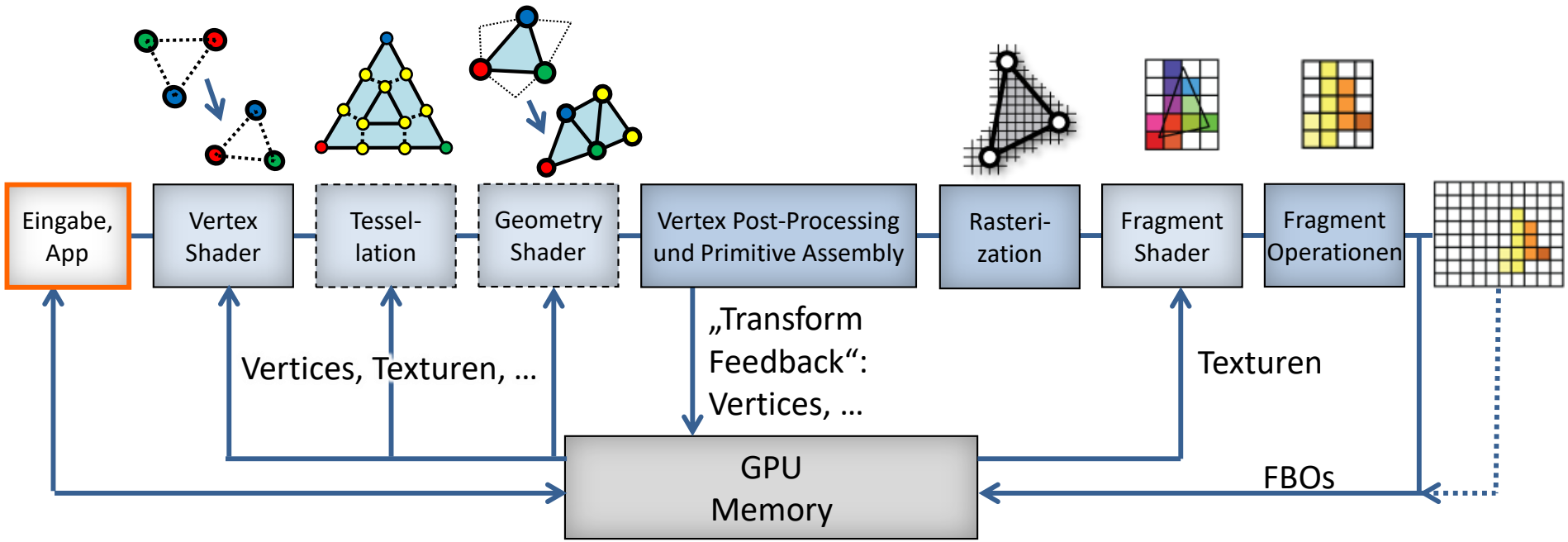


Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



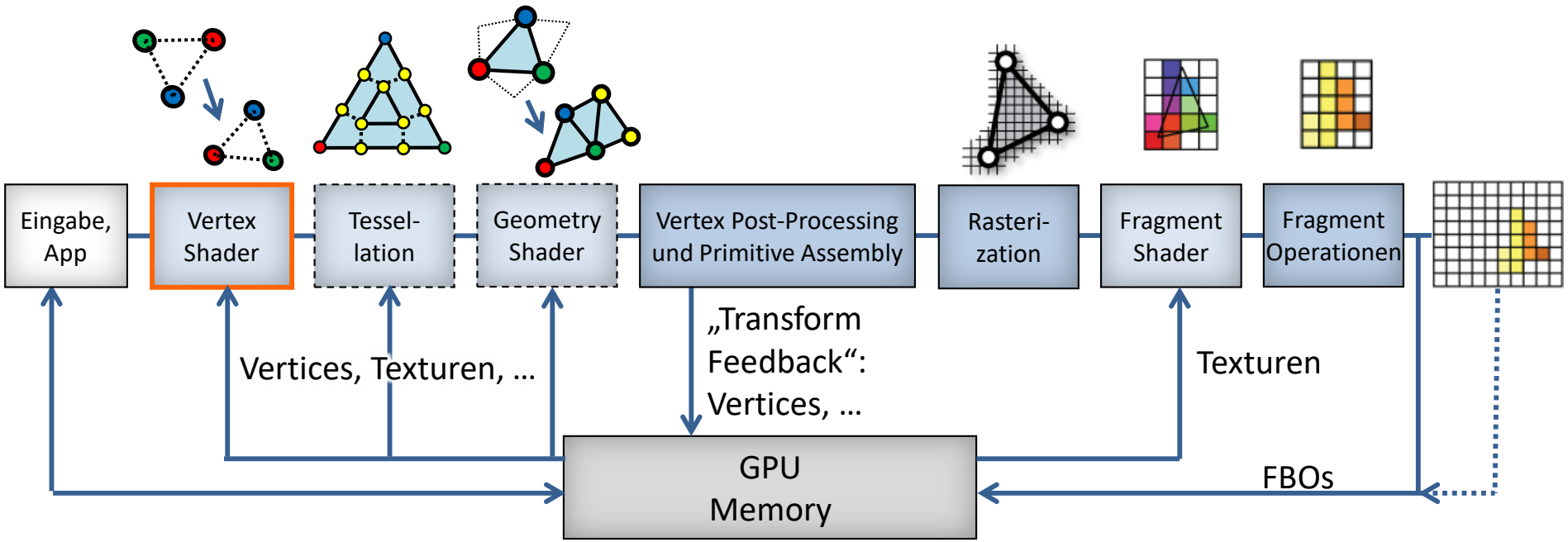
- Grafikprozessor (GPU):
  massiv-paralleler Prozessor, spezielle Hardware-Einheiten für Grafik
- teils programmierbar: Geometrieverarbeitung und Schattierung
- teils „fixed-function“: Rasterisierung selbst, Tiefentest, …
- Pipeline-Verarbeitung:
  - Daten werden von einer Stufe zur nächsten weitergegeben
  - gut geeignet für hohen Durchsatz: viel Geometrie, viele Pixel

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



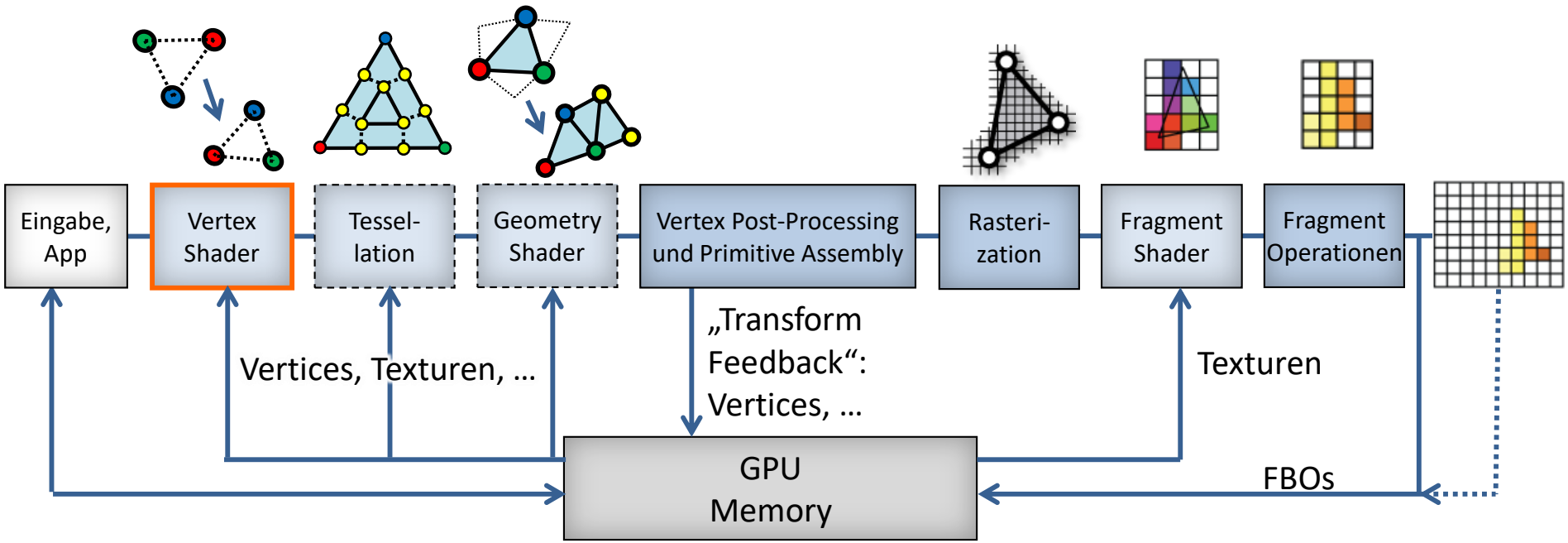
- Eingabedaten für die Grafik-Pipeline
  - Strom von Vertices und ggf. Indizes für Dreiecksnetze
  - Daten werden im Speicher der GPU abgelegt (sog. Buffer Objects)

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- Transformation **einzelner** Vertices und Verarbeitung deren Attribute
  - keine Vertex-Erzeugung oder -Löschung
  - Eingabe des Shaders ist:
    - ein Vertex und seine Attribute
    - „globale“ Zustände bzw. Konstanten (z.B. Transformationsmatrizen)
- Berechnung von **Attributen, die für Fragmente interpoliert werden**
  - z.B. Beleuchtung per Vertex (Gouraud Shading) oder Normalen für Phong Shading

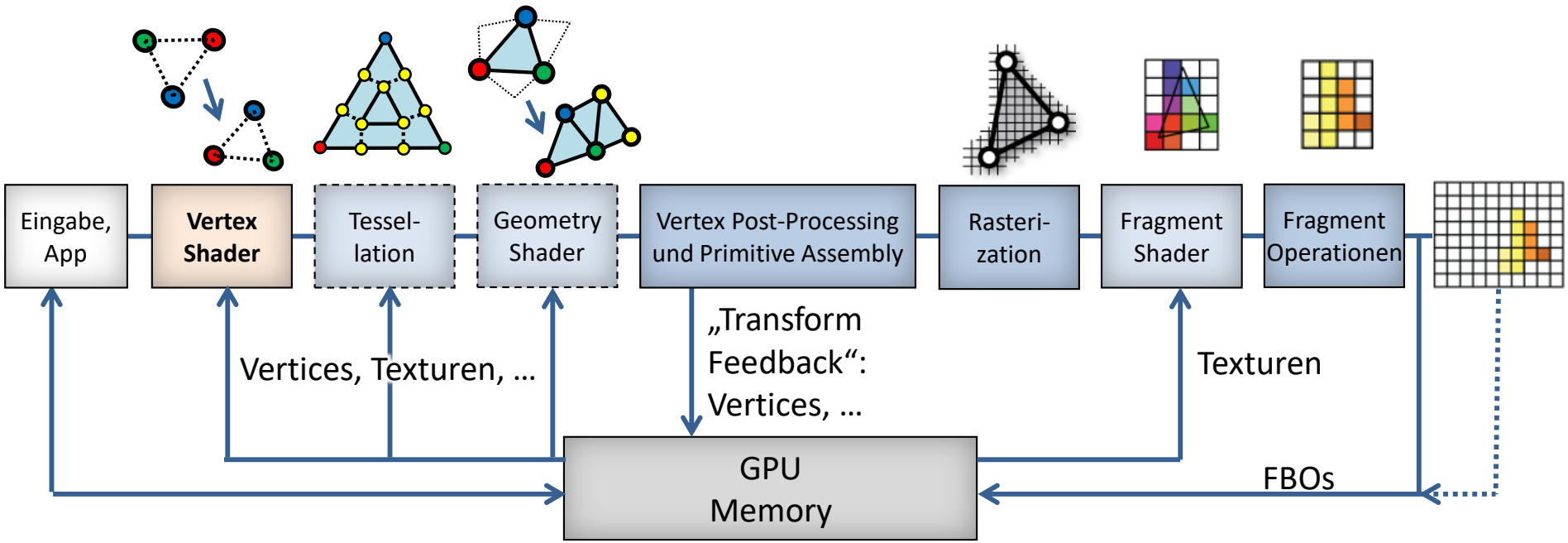
Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- Shader-Programmierung mit der OpenGL Shading Language (GLSL)
  - im Kompatibilitätsprofil, d.h. wenn man OpenGL-Altlasten mitschleppt: Zugriff auf OpenGL Zustände (Matrizen, Lichtquellen, Materialien, …)
- Beispiel eines Vertex Shader / Vertex Program im Kompatibilitätsprofil:

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

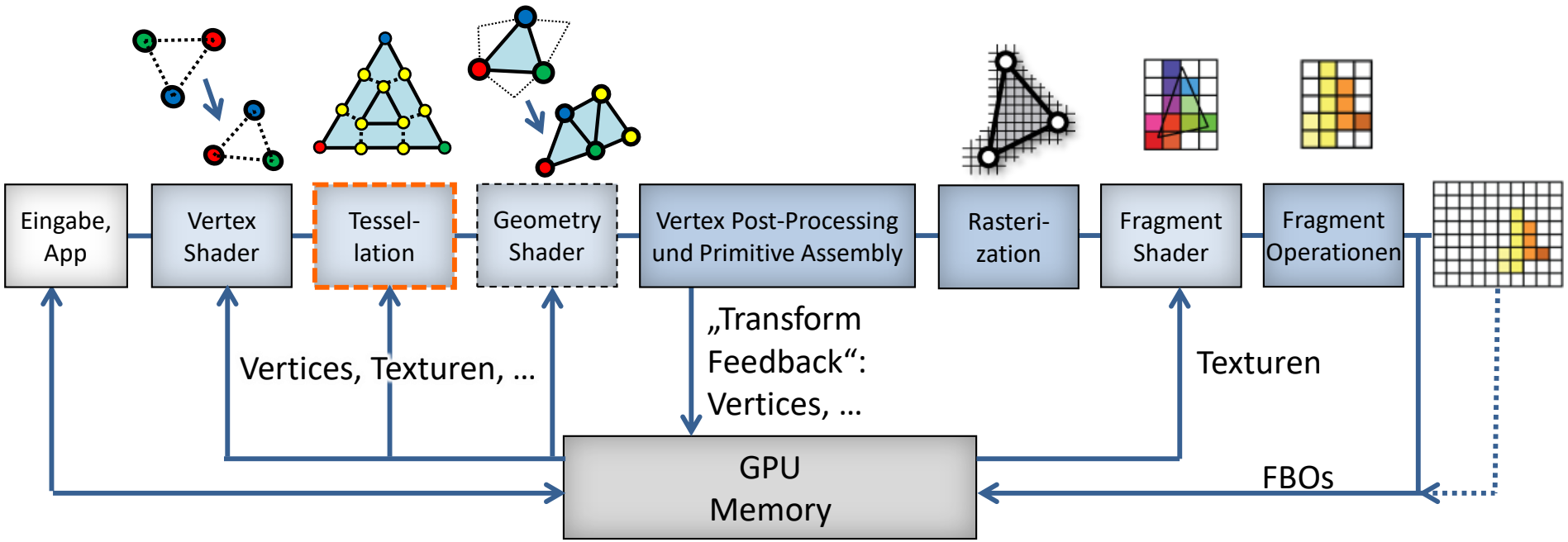
Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- Beispiel: Berechnung des Vektors L zur Lichtquelle und transf. Normale N

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;     // über „uniform“ und „in“ Variablen
uniform vec3 lightSourcePos;                      // sprechen wir in der Vorlesung
in  vec4 in_position, in_normal;
out vec3 L, N; // Ausgabe des Vertex Shader
void main() {
      gl_Position = matrixMVP * in_position;
      L = lightSourcePos – vec3( matrixMV * in_position );
      N = vec3( matrixNrml * in_normal ); }
```

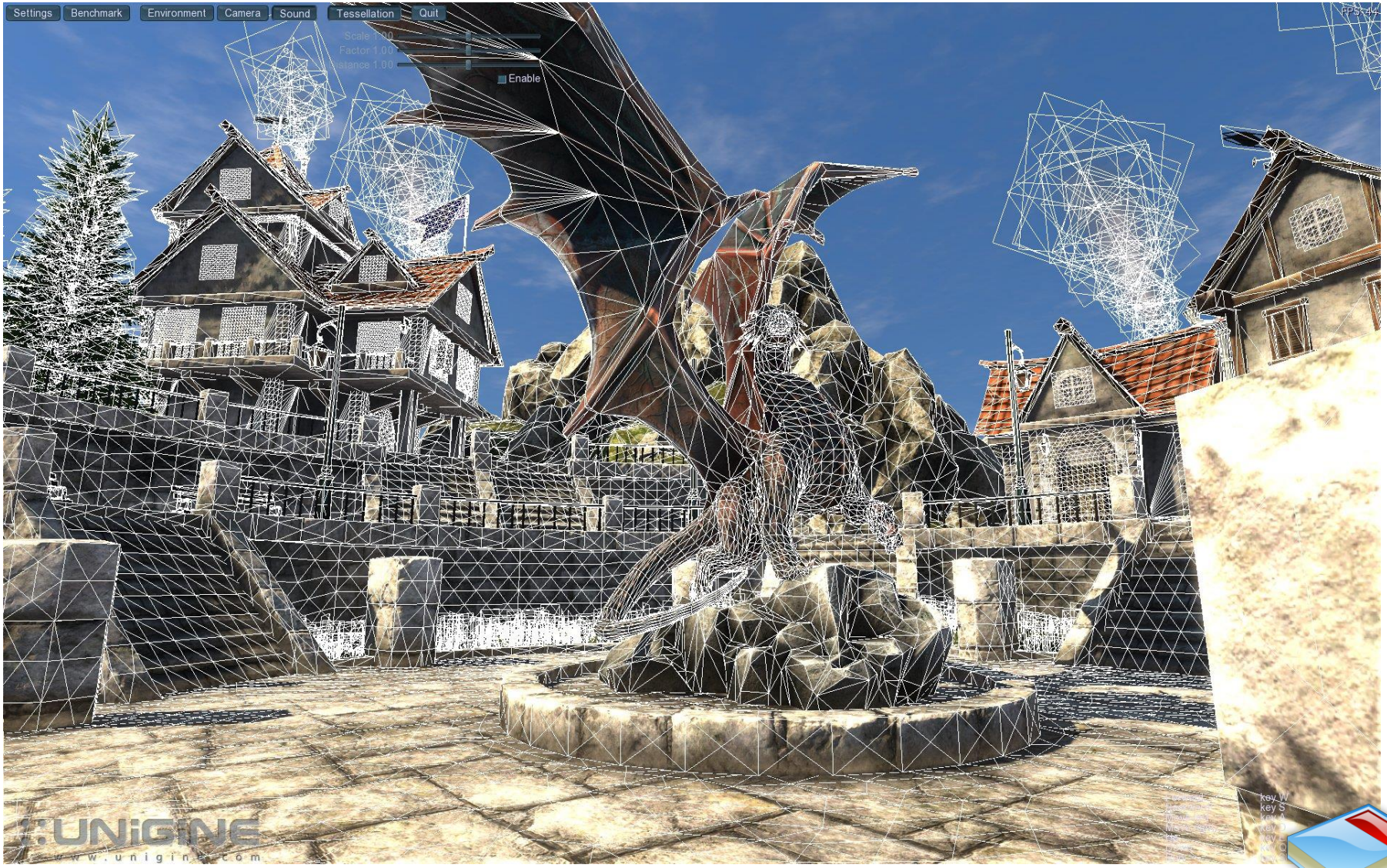
Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- programmierbare Stufe für Unterteilung von Primitiven (optional)
  - Tessellation Control Shader: bestimmt notwendige Unterteilung
  - eigentliche Unterteilung ist nicht programmierbar
  - Tessellation Evaluation Shader: Berechnung pro generiertem Vertex
  - spezieller Eingabe-Primitivtyp **GL_PATCH**

Beispiel Tessellation – Unigine Benchmark

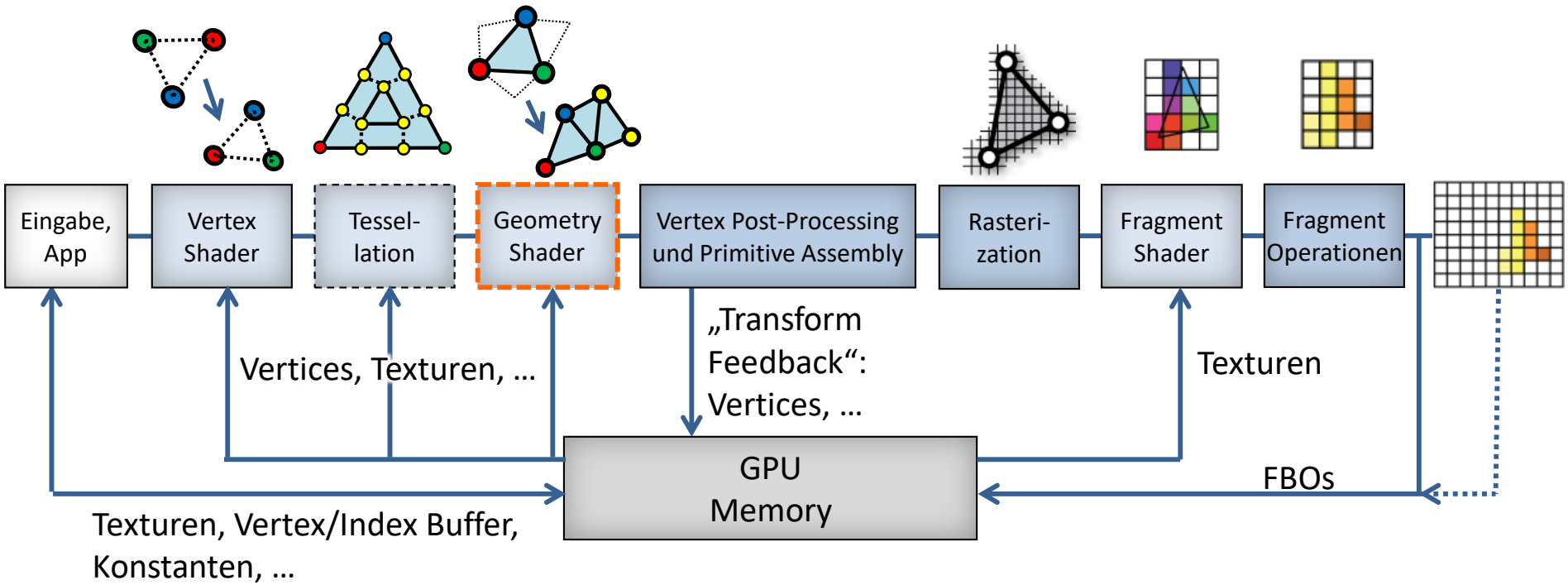
Eingabegeometrie vor der Unterteilung durch die Tessellation-Einheit



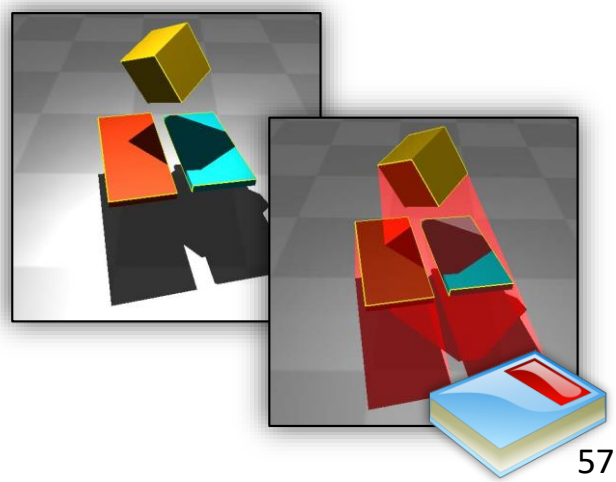
Beispiel Tessellation – Unigine Benchmark

Displacement Mapping: Unterteilung und Verschiebung der neuen Vertices

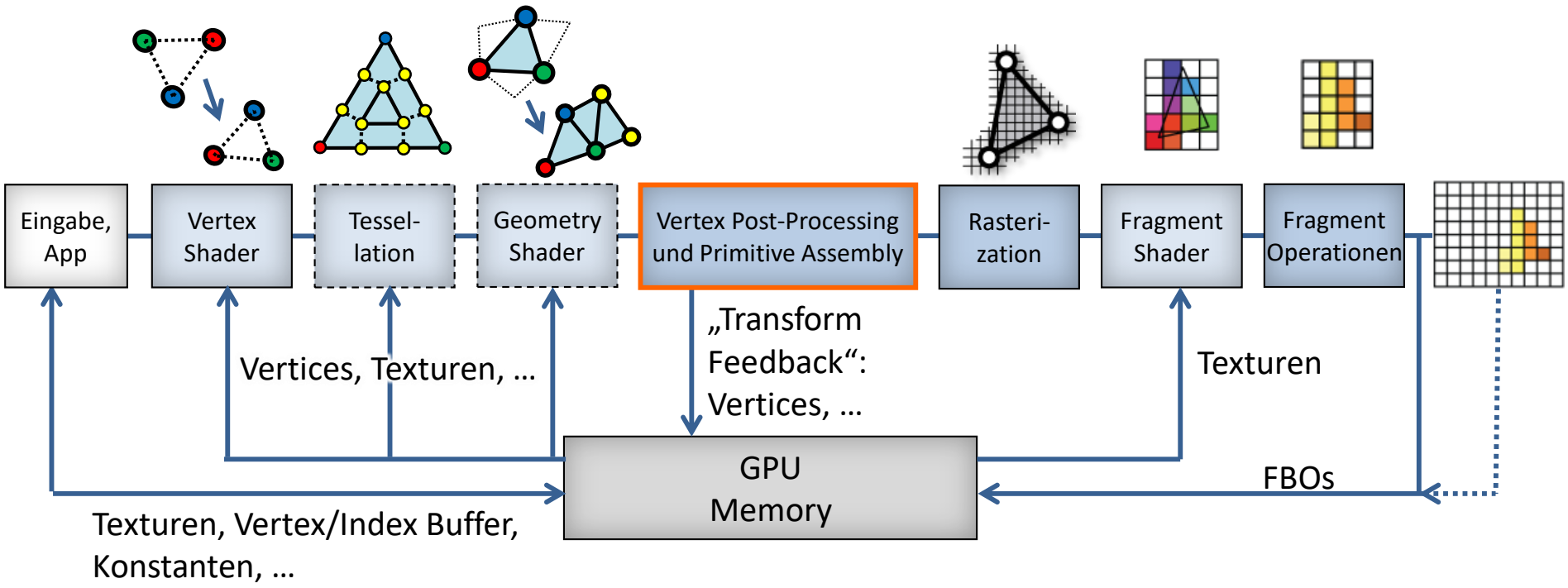
Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- programmierbare Stufe für Bearbeitung von Primitiven (optional)
  - Eingabe: **ein** Primitiv (Dreieck, Linie, Punkt)
  - Ausgabe: kann Primitive vervielfachen, entfernen oder beliebig umwandeln (Dreiecke zu Linien etc.)
  - kann auf Nachbarprimitive zugreifen, wenn die Applikation die Information zur Verfügung stellt
  - Beispiel: Instanziierung,
    Dreieck → Kantenzug, Schattenvolumen, …

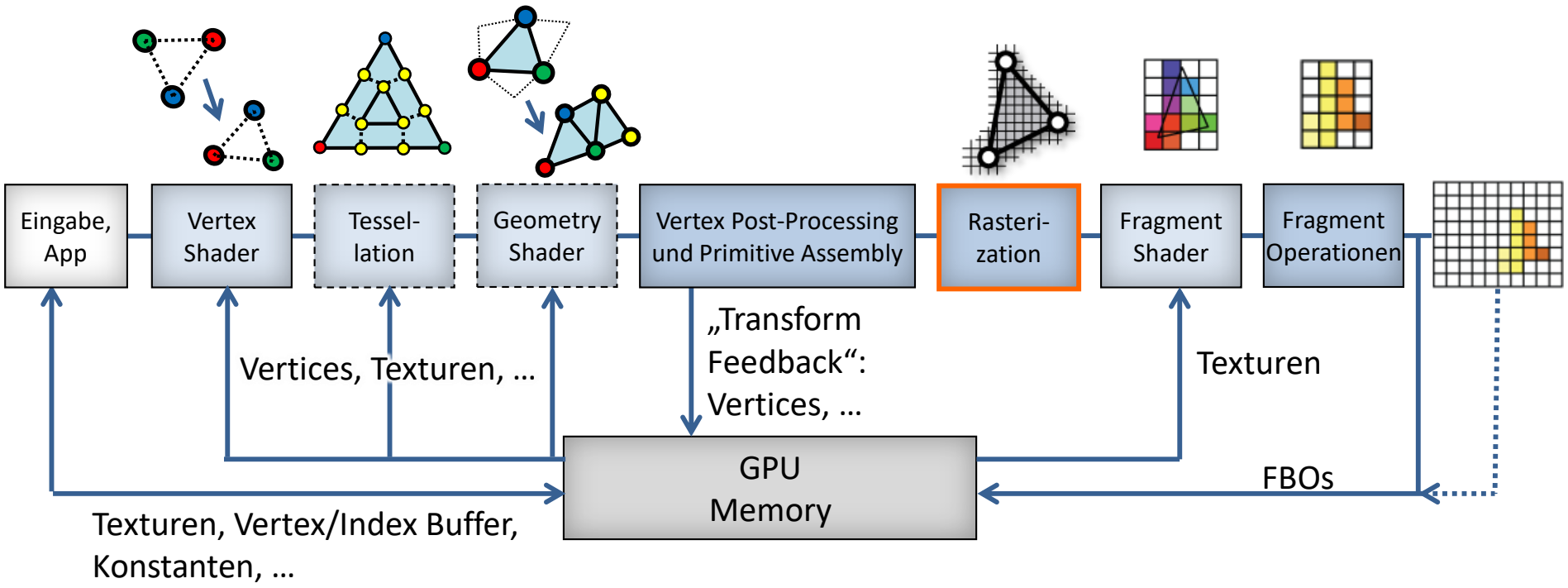


Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



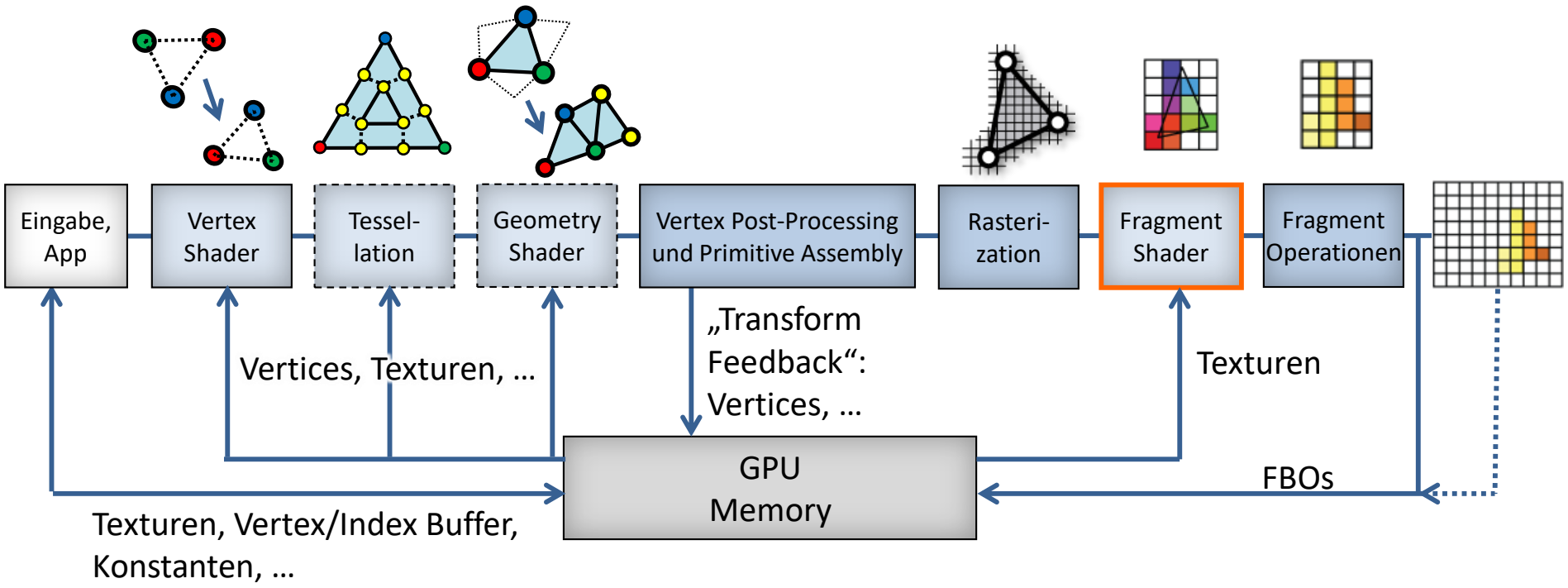
- nicht-programmierbare Stufe („fixed-function“)
  - Transform Feedback: Ausgabe transformierter Vertices, kein Rendering
  - **Primitive Assembly** setzt aus den Vertices die angeforderten Primitive zusammen und ist **nicht programmierbar**:
    `GL_POINT, GL_LINE{_STRIP|_LOOP}, GL_TRIANGLE{_FAN|_STRIP}`
  - Clipping (verwirft unsichtbare Punktprimitive, verändert Linien/Dreiecke durch Schnitt mit Sichtvolumen) und perspektivische Division
  - Backface Culling

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



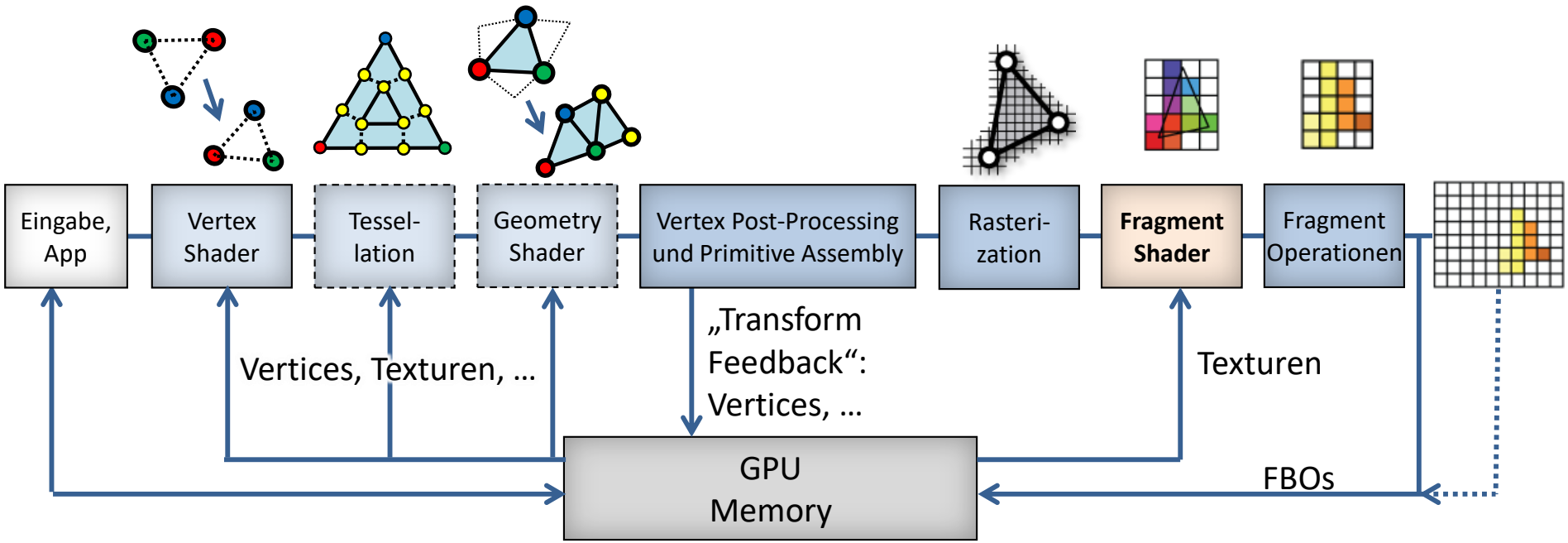
- Rasterisierung: nicht-programmierbare Stufe („fixed-function“)
  - Eingabe: Primitive
  - Eingabe: Fragmente mit 2D-Position, interpolierten Attributen, …

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- programmierbare Stufe
  - Eingabe: Fragmente mit 2D-Position, interpolierten Attributen, …
  - Ausgabe: Farbe, Opazität, optional Tiefe, …

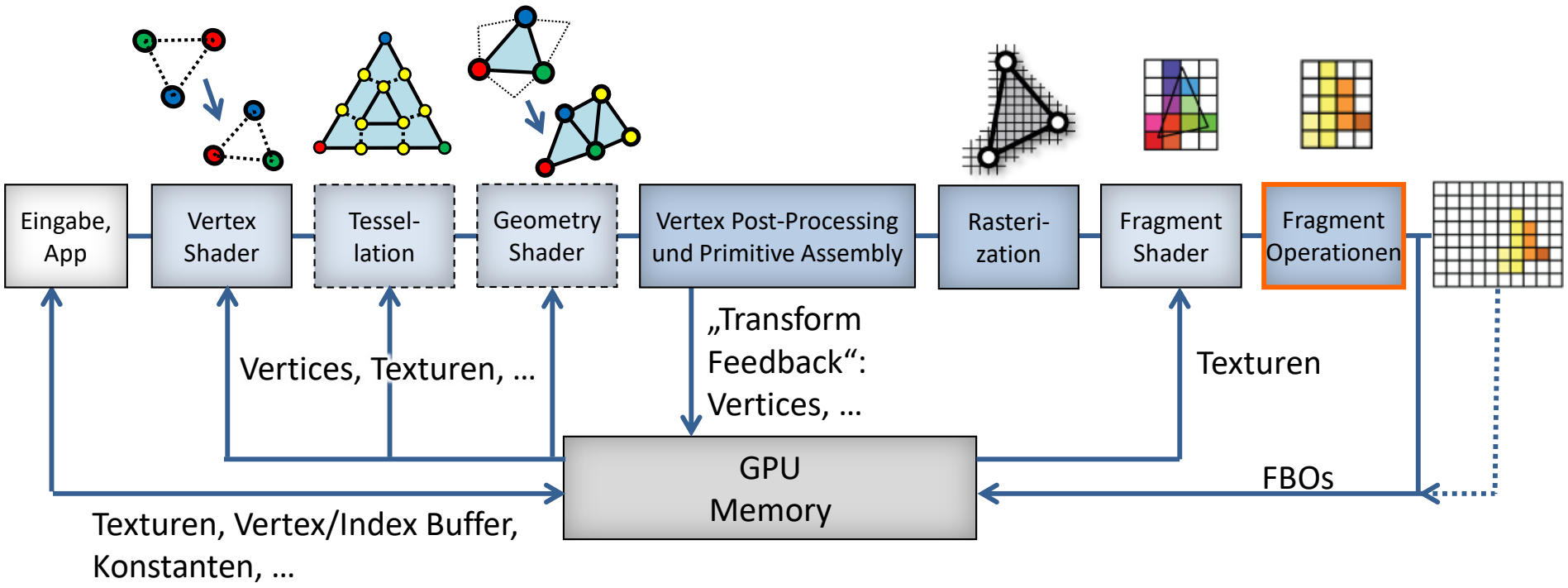
Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)



- Beispiel: Skalarprodukt für diffuse Beleuchtung aus interpolierter Richtung zur Lichtquelle L und Normale N

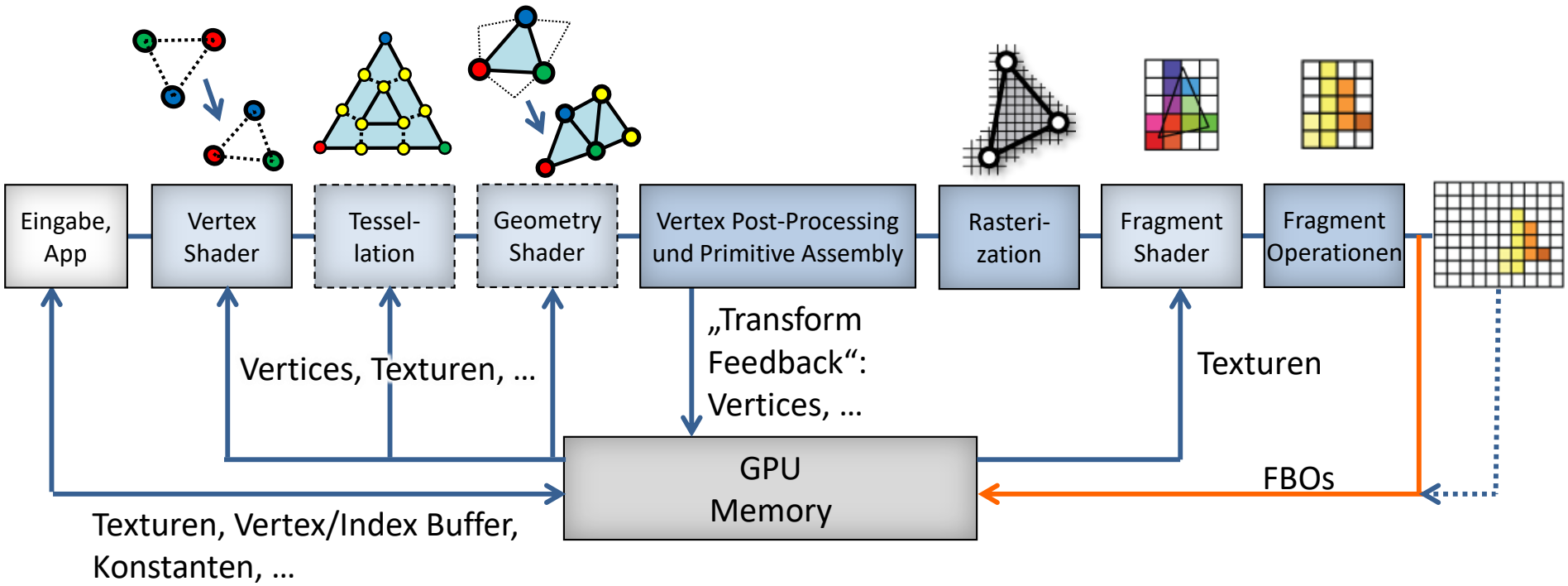
```
in vec3 L, N;                    // Eingabe aus dem Vertex Shader
out vec4 out_color;              // Ausgabe des Fragment Shader
void main() {
    float kd = max( 0.0, dot( normalize(L),
                              normalize(N) ) );
    out_color = vec4( kd );
}
```

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

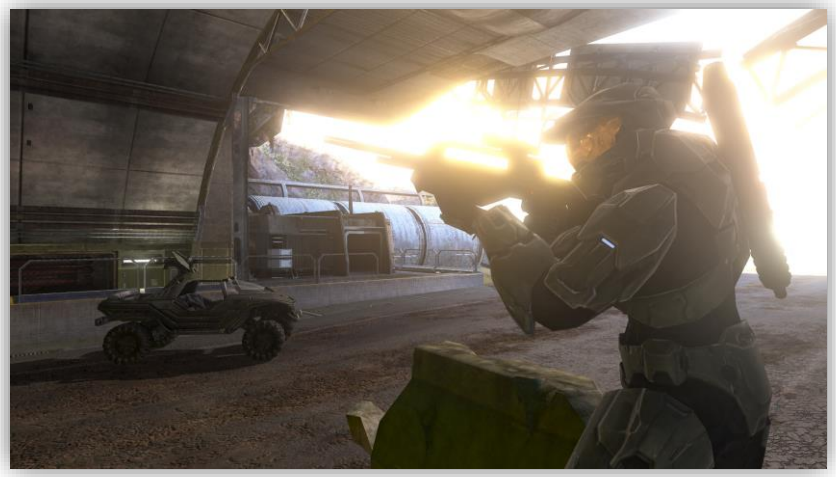


- nicht-programmierbare Stufe („fixed-function“)
  - Tiefentest mit Z-Buffer
  - Maskierung und Blending (= Kombination von Fragment- und Pixelfarbe im Framebuffer)
  - Schreiben in den Frame Buffer
  - Fragmentoperationen gibt es ebenfalls im klassischen OpenGL (wir besprechen sie später in diesem Kapitel)

Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

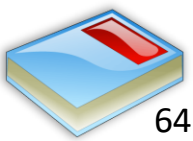


- FBOs: Resultat als Eingabe eines späteren Durchgangs
  - Post-Processing Effekte
  - Order-Independent Transparency
  - Deferred Shading Techniken
- jeder Shader kann lesend und schreibend auf Puffer zugreifen



OpenGL Shading Language (GLSL, GLSlang)

- 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** (1 Sign-Bit, 5 Bit Exponent, 10 Mantisse)
  - **float**/**double**/**int** wie in C, **bool** strikt nur **true**/**false**
  - **fixed** (Fixed Point-Zahlen insb. für OpenGL ES, 16.16 integer)
  - GLSL Precision Qualifiers (**lowp**, **mediump**, **highp**), für OpenGL ES mit Genauigkeit die von der jeweiligen Implementation abhängt
- 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**



GLSL: Variablen

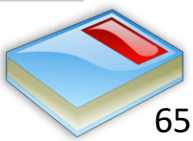
- Anmerkung:
  - **`diese Schriftart`** ist CPU/OpenGL-Code
  - **diese Schriftart** ist GLSL-Code

- 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 );
```



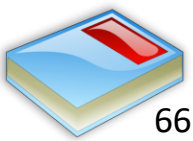
GLSL: Variablen

- 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

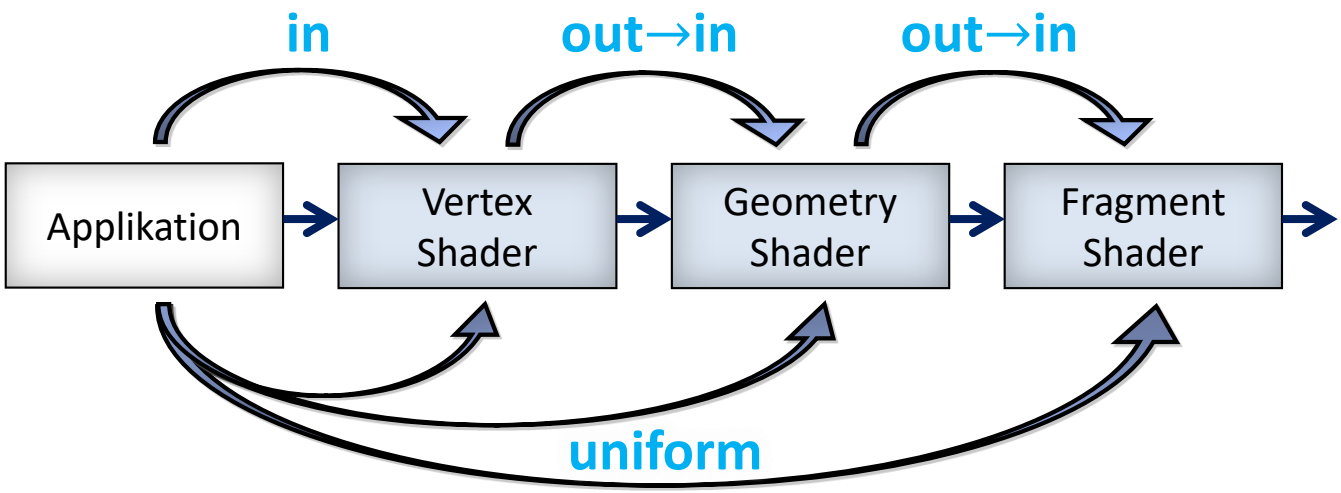
```
mat4 m = mat4( 1.0 )                    // Diagonalmatrix mit 1.0
vec2 a = vec2( 1.0, 2.0 );
vec2 b = vec2 (3.0, 4.0 );
mat2 n = mat2( a, b );                  // a und b werden Spaltenvektoren
mat2 k = mat2( 1.0, 0.0, 0.0, 1.0 );    // 2x2 Diagonalmatrix (Parameter
                                        // geben zuerst die Elemente der 1.
                                        // Spalte, dann der 2. an
```



GLSL Qualifiers (OpenGL 3.x/4.x, OpenGL ES 3.0)

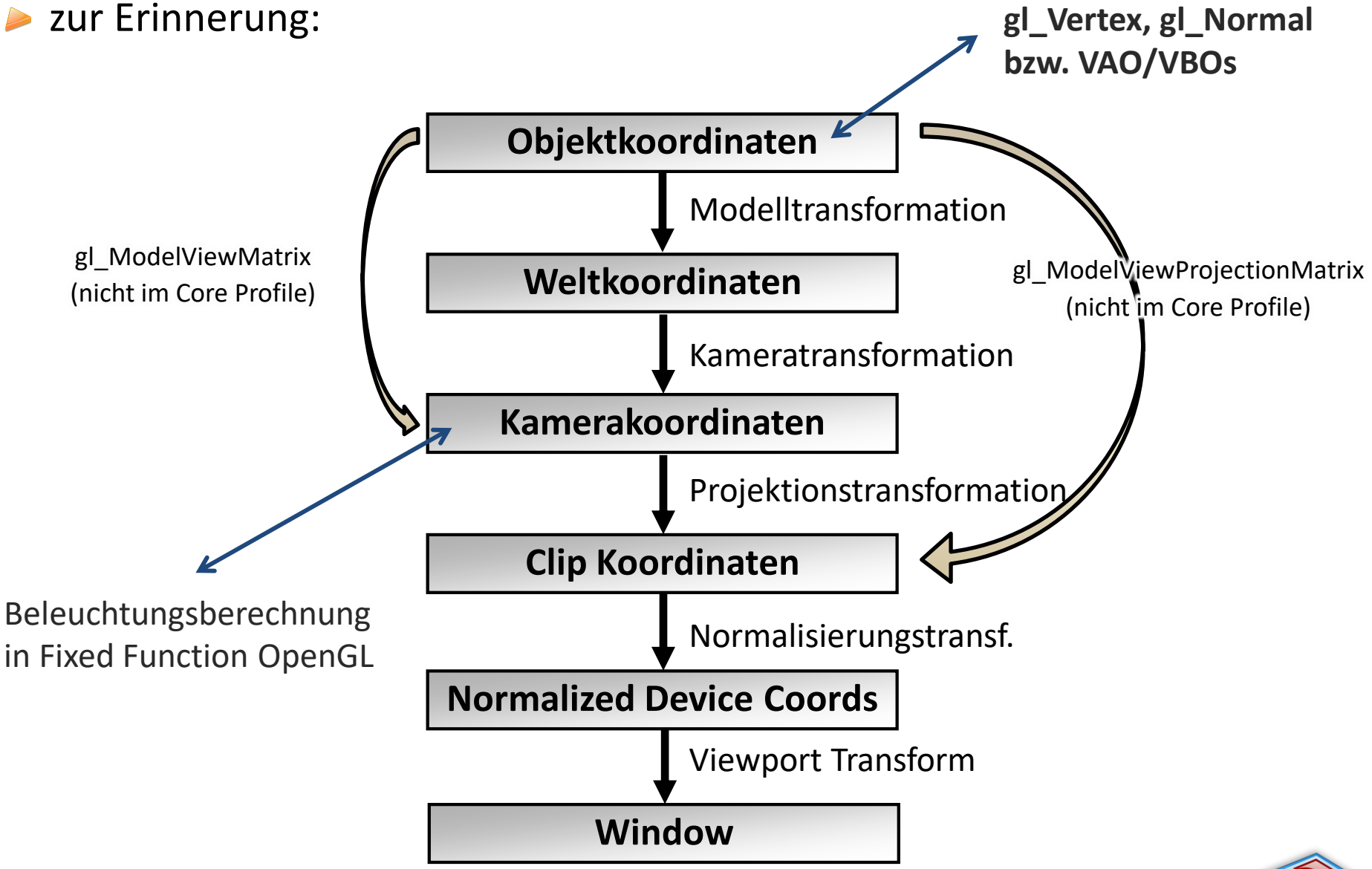
Definition des Verwendungszwecks von Variablen

- **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 (interp. im Fragment Shader)
- **out**: Ausgabe-Attribute des aktuellen Shaders



Transformationspipeline

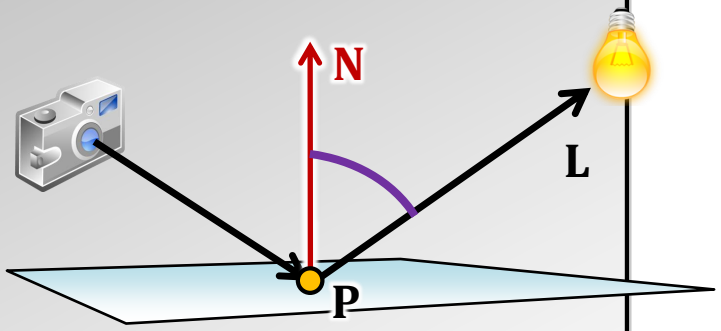
- zur Erinnerung:



Mehr Infos:
http://www.opengl.org/resources/faq/technical/viewing.htm

GLSL 3.x/4.x Diffuse Beleuchtung, Gouraud Shading

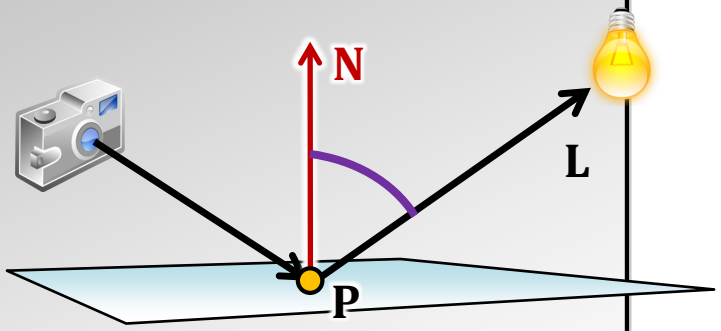
```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;
in  vec4 in_position;
in  vec3 in_normal;
out vec4 color; // Ausgabe des Vertex Shader
void main() {
    gl_Position = matrixMVP * in_position;
    // Beleuchtungsberechnung in Kamerakoordinaten
    vec3 P = vec3( matrixMV * in_position ); // Annahme: w = 1
    vec3 N = normalize( vec3( matrixNrml * vec4( in_normal, 0.0 ) ) );
    vec3 L = normalize( lightSourcePos – P );
    color = vec4( max( 0.0, dot( L, N ) ) );
}
```



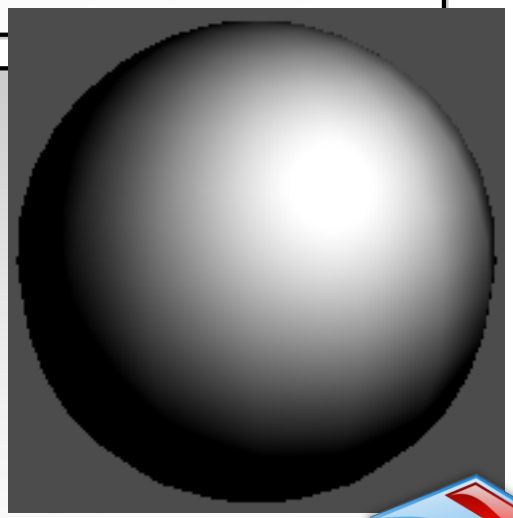

```
in vec4 color; // Wert aus Vertex Shader interpoliert
out vec4 out_color;
void main() {
    out_color = color;
}
```

GLSL 3.x/4.x „Per-Pixel“ Beleuchtung, Phong Shading

```
uniform mat4 matrixMVP, matrixMV, matrixNrml;
uniform vec3 lightSourcePos;
in  vec4 in_position;
in  vec3 in_normal;
out vec3 L, N; // Ausgabe des Vertex Shader
void main() {
    gl_Position = matrixMVP * in_position;
    vec3 P = vec3( matrixMV * in_position ); // Annahme: w = 1
    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 );
}
```



GLSL: Vektoren und Swizzling

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

```
vector.[xyzw rgba stpq]
```

- Koordinaten (xyzw), Farbkanäle (rgba) und Texturkoordinaten (stpq) sind synonym und können beliebig vermischt werden

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

- Verdrehen, Weglassen und/oder Replizieren ist möglich

```
vec4 g, h;   g.wxyz = h.yyxx;
vec2 a = g.xz;
```

- aber Komponenten müssen zum Typ passen

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

- Beispiel: was tut das wohl?

```
vec3 c = v0.yzx * v1.zxy - v0.zxy * v1.yzx;
```

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 große Auswahl von Funktionen in GLSL, die sich an grafischen Anwendungen orientieren
- viele davon arbeiten komponentenweise (SIMD)
- Beispiele:
  - 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/opengl46-quick-reference-card.pdf

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