Computergrafik

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

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

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: „ein OpenGL-Befehl bewirkt sofortiges Rendern“
  - Gegenteil: „Retained Mode“-APIs verwalten Objekte mit Texturen etc.

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)
- 2016: Vulkan („Next Generation OpenGL“)
- 2017: OpenGL 4.6 (SPIR-V, effizienteres Batch-Rendering, …)

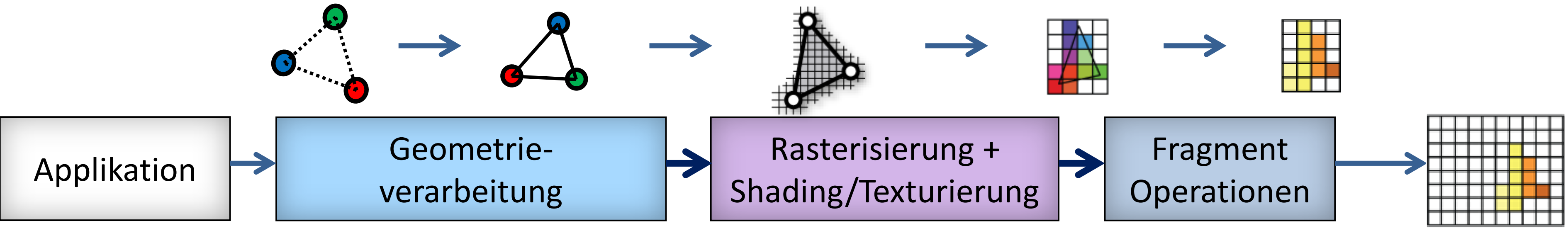
Features

OpenGL API

Vulkan

OpenGL – Zustandsmaschine

- die Verarbeitung in der Grafik-Pipeline wird konfiguriert, z.B. Zustände: Shading an/aus & Lichtquellen, Material, Texturen, Shader, …
- klassisches OpenGL: rein-konfigurierbare Pipeline
  - z.B. nur Blinn-Phong-Beleuchtungsmodell und Gouraud-Shading (Interpolation von Vertex-Farben)



- 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 – Zustandsmaschine

- modernes OpenGL: frei programmierbare Stufen in der Pipeline, u.a. Geometrieverarbeitung, Schattierungsberechnung, Tesselierung mittels der OpenGL Shading Language (GLSL)
  - es gibt zudem Puffer auf die lesend und schreibend zugegriffen werden kann (nicht in der Abbildung)
- Teile der Verarbeitung sind nach wie vor nur konfigurierbar, z.B. Fragment Operationen

Eingabe, App

Vertex Shader

Tessel-lation

Geometry Shader

Vertex Post-Processing und Primitive Assembly

Rasteri-zation

Fragment Shader

Fragment Operationen

Vertices, Texturen, ...

„Transform Feedback“: Vertices, ...

Texturen

FBOs

GPU Memory

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();
```

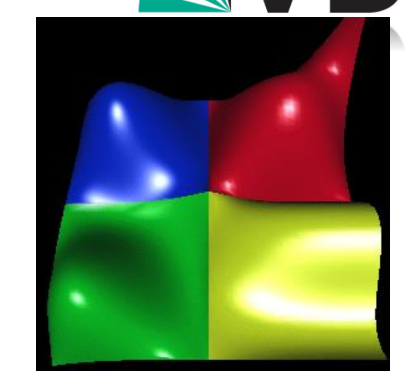
OpenGL ≤ 2.0: Low-level, immediate mode API

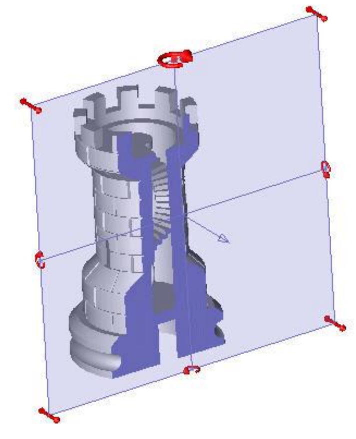
```
// Framebuffer löschen
glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
...

// 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();
```

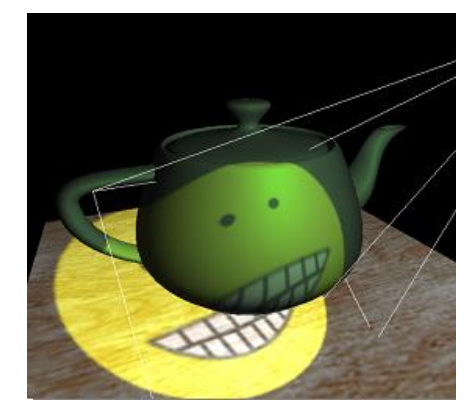
durchgestrichene Befehle sind in modernem OpenGL (Core Profile) nicht mehr vorhanden

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 );
```



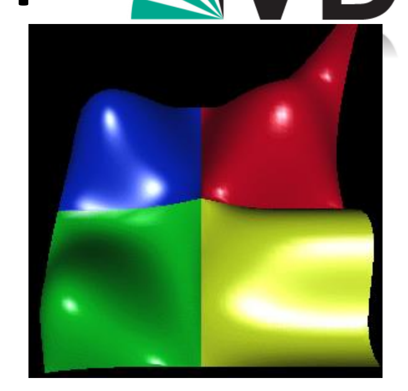


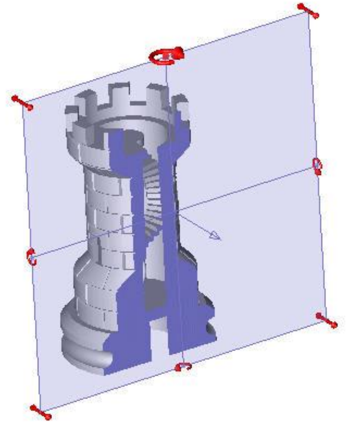


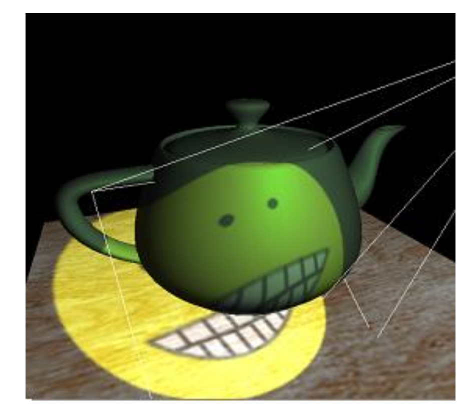
OpenGL ≤ 2.0: Low-level, immediate mode API

IVD

```
// 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 );
```







Inhalt des Vorlesungsteils

Wichtige Aspekte programmierbaren/modernen Renderings

- Vertex und Fragment Shader
  - globale Variablen (Transformationsmatrix, Lichtquellen, Texturen, …)
  - Ein- und Ausgabevariablen (z.B. Vertex-Attribute)
- Geometriedaten
  - … im Speicher der GPU
  - Optimierung, Performanz
  - Instanziierung, Geometry Shader und Tesselierung
- weitere Konzepte in der Vorlesung
  - Fragment Operationen (auch in klassischem OpenGL)
  - Compute Shader, Shader Storage Buffer Objects
  - Mesh Shader, Raytracing (Vulkan)
  - Ausblicke, Rendering-Techniken

Inhalt des Vorlesungsteils

Wichtige Aspekte programmierbaren/modernen Renderings

- Vertex und Fragment Shader
  - globale Variablen (Transformationsmatrix, Lichtquellen, Texturen, …)
  - Ein- und Ausgabevariablen (z.B. Vertex-Attribute)
- Geometriedaten
  - … im Speicher der GPU
  - Optimierung, Performanz
  - Instanziierung, Geometry Shader und Tesselierung
- weitere Konzepte in der Vorlesung
  - Fragment Operationen (auch in klassischem OpenGL)
  - Compute Shader, Shader Storage Buffer Objects
  - Mesh Shader, Raytracing (Vulkan)
  - Ausblicke, Rendering-Techniken

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 ) ) );
}
```

N

L

P

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

Zur Erinnerung: Koordinatensysteme in der CG

matrixMVP

matrixMV

Objektkoordinaten

Modelltrans-
formation

Weltkoordinaten

Kameratrans-
formation

Kamerakoordinaten

Projektions-
transformation

Clip Koordinaten

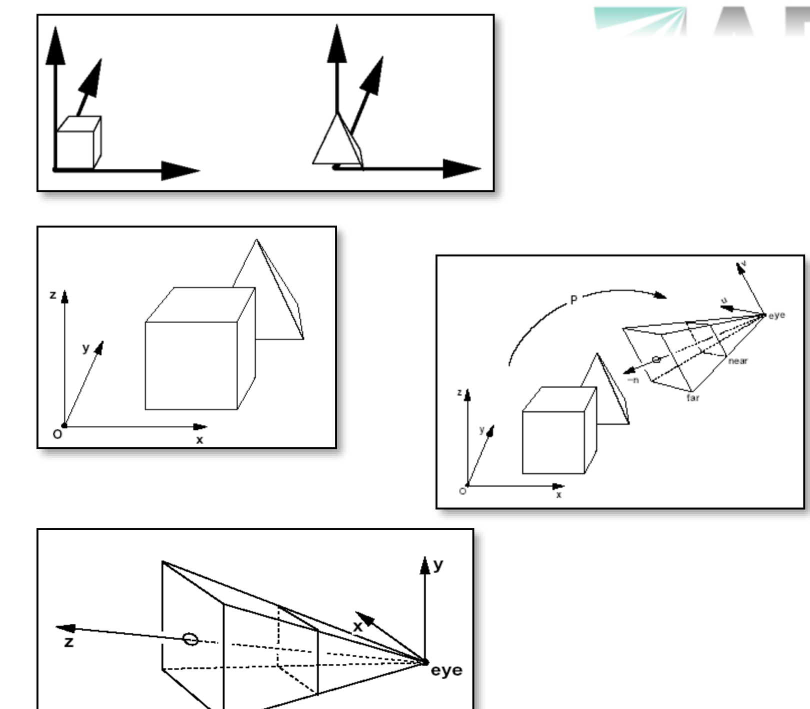
Normalisierungstransf. =
perspekt. Division durch w

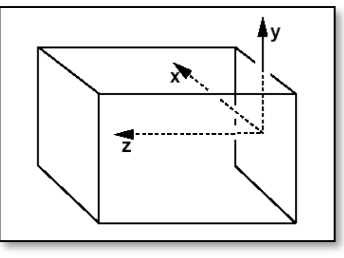
normalisierte Gerätekoord.

Normalized Device Coords

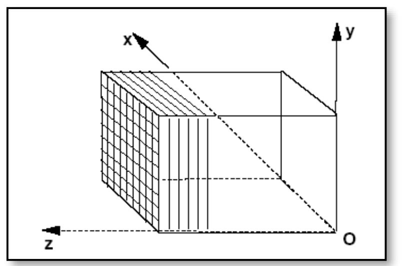
Viewport Transformation

Window/Screen Coords





Canonical
Viewing Volume



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 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 );
}
```

Transformationen: 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/Kameraparameter
void resize( int w, int h ) {
  matP = glm::perspective( 45.f, (float)w/(float)h, 1.f, 40.f );
  matV = glm::lookAt( camPosition, camTarget, camUp );
  matM = ...;

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

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

Transformationen: Beispiel OpenGL 3.x/4.x

```
// Rendering 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
  ...

  glutSwapBuffers();
}
```

Framebuffers in OpenGL:
Front und Back Color Buffer (meist RGBA),
Z-Buffer, Stencil-Buffer, Accumulation Buffer, …

Transformationen: Beispiel OpenGL 3.x/4.x

Explizite Zuweisung von location

```
layout(location = 2) uniform mat4 matrixMVP;
...
```

```
void display( void ) {
  glUseProgram( shaderPrg );
  // Setzen einer Uniform-Variable
  glUniformMatrix4fv( 2, 1, GL_FALSE, matMVP );
  ...
  // Zeichnen
  ...
  glutSwapBuffers();
}
```

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 );
    L = lightSourcePos – P;
    N = vec3( matrixNrml * vec4( in_normal, 0.0 ) );
}
```

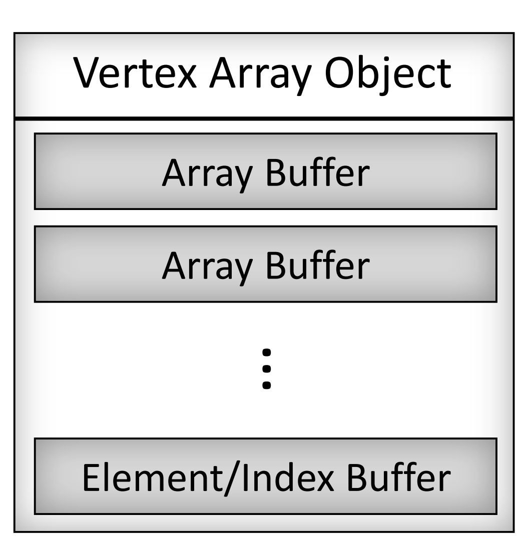
```
in vec3 L, N;
out vec4 out_color;
void main() {
    float kd = max( 0.0, dot( normalize(L),
                              normalize(N) ) );
    out_color = vec4( kd );
}
```

OpenGL Buffer Objects für Geometrie

Überblick Vertex Array Objects, Vertex und Element/Index Buffer

- Buffer Objects werden benötigt, um Geometrie im Speicher der Grafikkarte abzulegen und sie anschließend zu zeichnen

- wir betrachten hier zwei Arten von Puffern
  - Vertex Buffer (`GL_ARRAY_BUFFER`):
    Speichern von Vertices und Vertex-Attributen
  - Index Buffer (`GL_ELEMENT_ARRAY_BUFFER`):
    Speichern von Indizes,
    „welche Vertices bilden ein Dreieck“



- Vertex und (optional) Index Buffer werden durch ein Vertex Array Object zusammengefasst
  - Vertex Buffer können auch Vertices und Attribute zusammen speichern (sog. „Interleaved Arrays“, siehe Parameter ***stride*** bei `glVertexAttribPointer`)

- **Anmerkung**: alle OpenGL Setups sind furchtbar… deswegen: einmal verstehen, einmal (als C++-Klasse) implementieren, nie mehr anschauen

OpenGL-Objekte allgemein

- Texturen, Geometrie, Shader etc. sind OpenGL-Objekte (nicht im OOP-Sinn!)
  - zustandsbehaftet: Texturobjekte speichern bspw. die Textur inkl. Mipmaps, aber auch Filter- und Adressierungsmodi
- Befehle für Shader-Objekte / Texturen:
  `glCreateShader`, `glCreateProgram`, `glUseProgram` etc., `glTexImage2D`, ...
- Ablauf für andere Objekte zur Erzeugung, Initialisierung, Löschen
  - erzeuge ein **Handle** (in OpenGL auch **name** genannt)
    ```
    GLuint myHandle;
    glGen{Textures|Buffers|...}
                    ( 1, &myHandle );
    ```
  - selektiere ein Objekt (binding)
    ```
    glBind*( <target>, myHandle );
    ```
  - übergebe die Daten an OpenGL (erneute Übertragung nur, wenn sich Daten ändern)
    ```
    glBufferData( <target>, ... );
    ```
  - deselektiere ein Objekt (unbinding)
    ```
    glBind*( <target>, 0 );
    ```
  - Objekt löschen
    ```
    glDelete*( 1, &myHandle );
    ```

Beispiel: Vertex Buffer – OpenGL 3.x/4.x

```
// (statische) Vertex-Daten
GLfloat vertices[NUM_VERTICES * 3] = {
  1.0f, 3.2f, 0.0f,
  ...
};

// (statische) Normalen
GLfloat normals[NUM_VERTICES * 3] = {
  1.0f, 0.0f, 0.0f,
  0.707f, 0.0f, 0.707f,
  ...
};

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

Beispiel: Vertex Buffer – 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_VERTICES * 3,
      vertices, GL_STATIC_DRAW );

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

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

Beispiel: Vertex Buffer – 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_VERTICES * 3,
       vertices, GL_STATIC_DRAW );

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

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

Beispiel: Vertex Buffer – 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_VERTICES * 3,
      vertices, GL_STATIC_DRAW );

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

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

Beispiel: Vertex Buffer – 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_VER
       vertices, GL_STATIC_DRAW );

  GLint attrLoc = glGetAttribLoc
                                                               );
  glEnableVertexAttribArray( att
  glVertexAttribPointer( attrLoc,

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

GL_ARRAY_BUFFER ist eines von 14 sogenannten „Targets“ (Verwendungszwecken) von Puffern. Man muss also mitteilen, für was der Puffer verwendet wird.

Beispiel: Vertex Buffer – OpenGL 3.x/4.x

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

  glG...                          ...;
  glB...                          ...BUFF...
  glB...                          ...BUFF...
                                  ...* NUM_...
                                  ...C_DRAW );
```

```
uniform mat4 ...;
in  vec4 in_position;
in  vec3 in_normal;
out vec3 L, N;
void main() { ... }
```

Attribute Location: Index eines „Eingaberegisters“

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

Verbinde Eingaberegister mit aktuell gebundenem Puffer

```
  glGenBuffers( 1, &nbo );
  glBindBuffer( GL_ARRAY_BUFFER, n...
  ...
  attrLoc = glGetAttribLocation(...                    ); ...
  ...
  glBindVertexArray( 0 );       // ...                ...bkoppeln
}
```

GLUT: Beispiel – OpenGL 3.x/4.x

```
// Rendering 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 Vertices aus dem VBO bilden ein Dreieck
  glDrawArrays( GL_TRIANGLES, 0, NUM_VERTICES );
  glutSwapBuffers();
}
```

Beispiel: Vertex Buffer – OpenGL 3.x/4.x

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

  glG                                              ffer
  glB
  glBu
```

```
uniform mat4 ...;
layout(location = 0) in vec4 in_position;
layout(location = 1) in vec3 in_normal;
out vec3 L, N;
void main() { ... }
```

```
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE,0,0);
```

```
  glGenBuffers( 1, &nbo );
  glBindBuffer( GL_ARRAY_BUFFER, n
  ...
  attrLoc = glGetAttribLocation(                              ); ...
  ...
  glBindVertexArray( 0 );         // Z                 abkoppeln
}
```

alternative Möglichkeit
für das Binding
(ein Buffer, > 1 Shader
→ gemeinsame Locations)

OpenGL Buffer Objects

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 den Puffer kopiert
      `..._COPY` werden von OpenGL erzeugt und zur Appl. übertragen

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
glBuffer{Sub}Data( GL_ARRAY_BUFFER, {offset,}
  sizeof(GLfloat) * NUM_VERT_COMPONENTS * NUM_VERTICES,
  NULL, GL_DYNAMIC_DRAW );
```

- Änderung mit `glBufferData` bzw. `glBufferSubData` oder durch Einblenden in den Hauptspeicher:

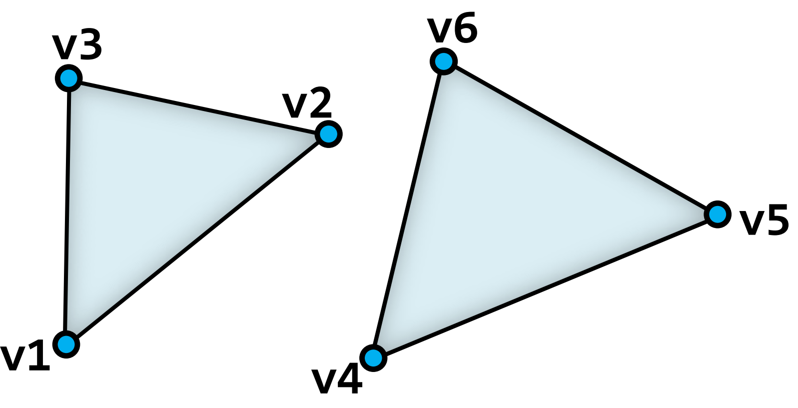
```
// Daten setzen/aktualisieren
void *m = glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
// <Speicheroperationen...>
glUnmapBuffer( GL_ARRAY_BUFFER );
```

  - analog `glMapBufferRange` für das Einblenden von Teilen eines Puffers
  - warum ist `GL_WRITE_ONLY` i.d.R. viel effizienter als `GL_READ_WRITE`?

Wdh. Geometrische Primitive (analog mit glDrawArrays!)

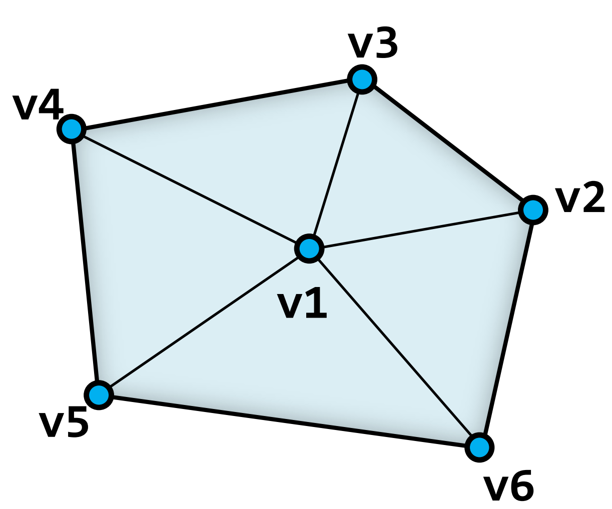
Isolierte/unabhängige Dreiecke: $3n$ Vertices für n Dreiecke

```
glBegin( GL_TRIANGLES );
 glVertex3fv( v1 );   } Dreieck 1
 glVertex3fv( v2 );
 glVertex3fv( v3 );
 glVertex3fv( v4 );   } Dreieck 2
 glVertex3fv( v5 );
 glVertex3fv( v6 );
glEnd();
```



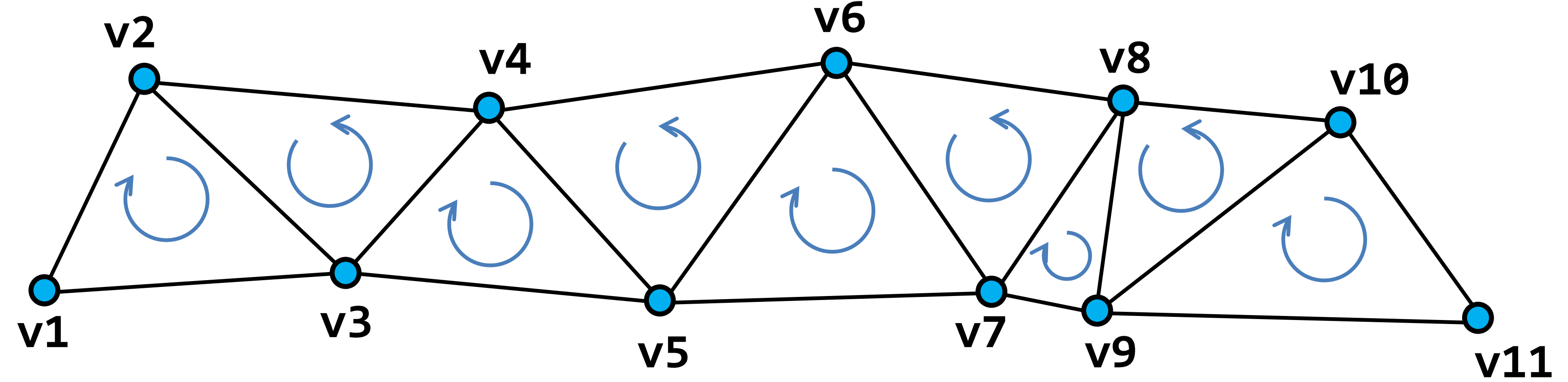
Dreiecksfächer, Triangle Fans: $n + 2$ Vertices für n Dreiecke (n typ. klein)

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



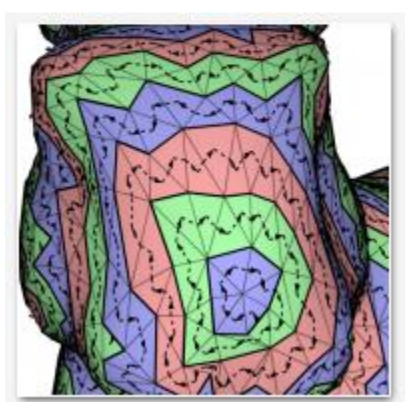
Wdh. Geometrische Primitive (analog mit glDrawArrays!)

Dreiecksstreifen, Triangle Strips




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

Dreieck 1
Dreieck 2
Dreieck 3



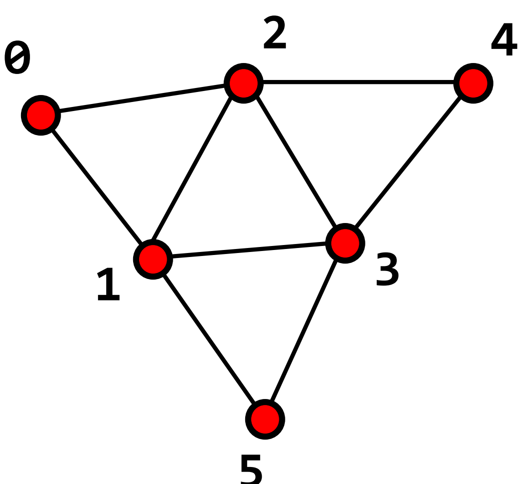
- $n + 2$ Vertices werden verarbeitet, um n Dreiecke zu zeichnen
- Orientierung bei Backface Culling?
- Dreiecksstreifen sind wichtig: es gibt Algorithmen, die Dreiecksnetze in möglichst wenige Streifen zerlegen

Element Array Buffers und Shared Vertex

Indexed Face Set oder Shared Vertex Darstellung des Dreiecksnetzes

- verwende Array(s) mit Vertices (und weiteren Attributen)
- verwende Index-Liste: welche Vertices bilden zusammen ein Dreieck
- Vorteile: weniger Daten (ein Vertex ist Eckpunkt mehrerer Dreiecke)
- funktioniert mit **GL_TRIANGLES**, **GL_TRIANGLE_STRIPS**, …

```
// Vertex-Daten
GLfloat vertices[NUM_VERTICES * 3] = {
  1.0f, 3.2f, 0.0f, ...
};

// Index-Daten für GL_TRIANGLES
// (Glushort würde Bandbreite sparen)
GLuint indices[NUM_INDICES] = {
  0, 1, 2, 2, 1, 3, 3, 1, 5, 2, 3, 4
};
```



Element Array Buffers und Shared Vertex

```
// “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
}
```

Element Array Buffers und Shared Vertex

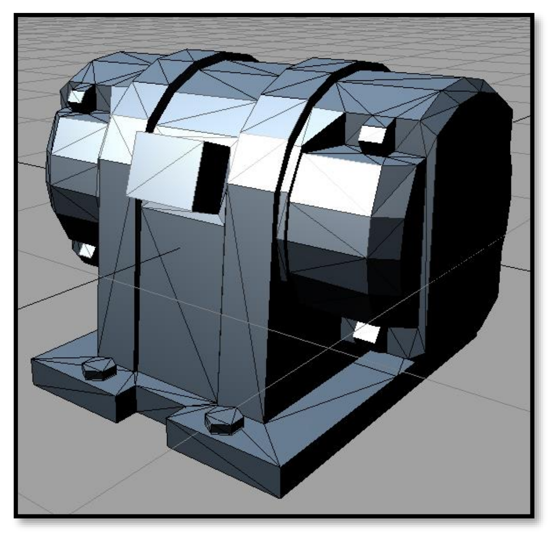
```
// Rendering eines Bildes
void display(void) {
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

  glUseProgram( shaderPrg );

  glBindVertexArray( vao );

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

  glutSwapBuffers();
}
```

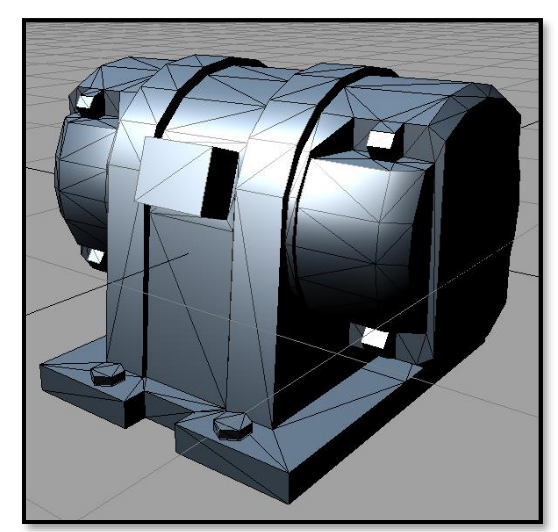
Ein Puffer mit mehreren Triangle Strips oder Fans?

- `glPrimitiveRestartIndex` legt einen Index fest, dessen Auftreten das Ende eines Dreiecksstreifens markiert
- alternativ bei Strips: degenerierte Dreiecke einfügen, z.B. mit
  `GLuint indices[] = { … , 3, 4, 5, 5, 6, 6, 7, 8, … }`

Schattierung und Shared Vertex Darstellung

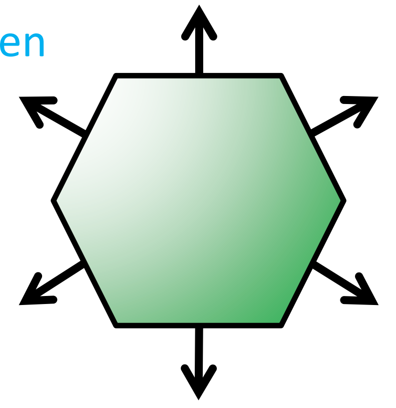
Normalen für Shading



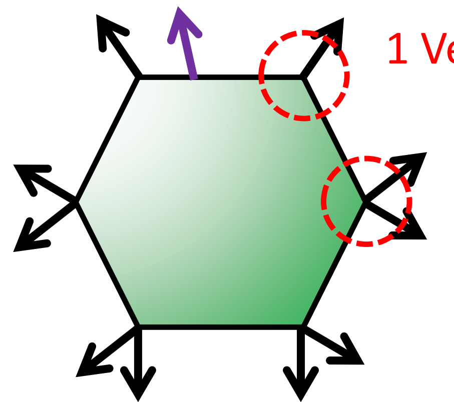


Flat Shading benötigt Vertices mit unterschiedl. Normalen

- `glProvokingVertex`
- in Vertex/Geometry Shader: **flat** Qualifier



Flat Shading (Dreiecksnormale)



Phong Shading (interpolierte Normale)

Element Array Buffers und Shared Vertex

Vertex Cache

- GPUs speichern Ergebnisse der letzten verarbeiteten Vertices: wird derselbe Vertex nochmal benötigt, wird auf den Cache zugegriffen
  - typische Vertex Cache-Größe in etwa 20 bis 32 (lässt sich für heutige GPUs nicht mehr so explizit angeben)
- **Vertex Caching funktioniert nur mit indizierten Vertices**
  - d.h. mit **`glDrawElements`**, aber nicht mit **`glDrawArrays`**
  - Identifikation der Vertices über deren Indizes
- Beispiel:
  - Indizes: 0, 1, 2, 3, 2, 3, 5, 1, 3, 4, 4, 2, 3, 6, 7, 8, ...
  - Misses & Hits: 0, 1, 2, 3, 2, 3, 5, 1, 3, 4, 4, 2, 3, 6, 7, 8, ...
- es gibt diverse Algorithmen, um die Reihenfolge der Dreiecke bzw. Vertices für bestmögliches Caching zu optimieren, siehe https://www.khronos.org/opengl/wiki/Post_Transform_Cache

3D Modelle verwenden oft Shared Vertex Darst.

- Export/Import der Daten über diverse 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
```

Datenstrukturen für Dreiecks- und Polygonnetze

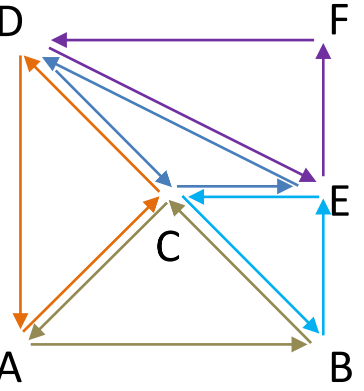
Shared Vertex-Datenstruktur

- Liste von Vertices { A, B, C, D, E, F }
- Indizes (topographische Information):
  { (A, B, C), (A, C, D), (C, A, B), (C, E, D), (E, F, D) }
- Frage z.B. nach Nachbardreiecken von (C, E, D) ist eine aufwändige Operation

D F
E
C
A B

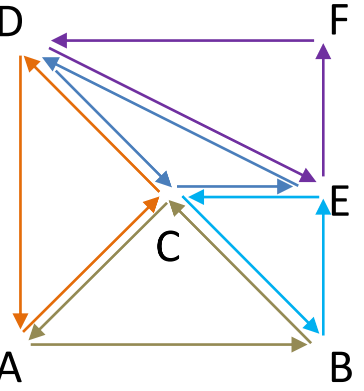
Half Edge-Datenstruktur

- Liste mit Position und ausgehenden Kanten für jeden Vertex
- Half Edges mit topologischer Information:
  - Geschwisterkante
  - End-Vertex
  - anliegendes Polygon (farblich dargestellt)
  - nächste Kante entlang des Kantenzugs
- pro Fläche: speichere Verweis auf <u>*eine*</u> Kante



Datenstrukturen für Dreiecks- und Polygonnetze

Half Edge-Datenstruktur



- Position und ausgehende Kanten für jeden Vertex
- Half Edge-Liste mit
  - Geschwisterkante
  - End-Vertex
  - anliegendes Polygon
  - nächste Kante entlang des Kantenzugs
- pro Fläche: speichere Verweis auf eine Kante

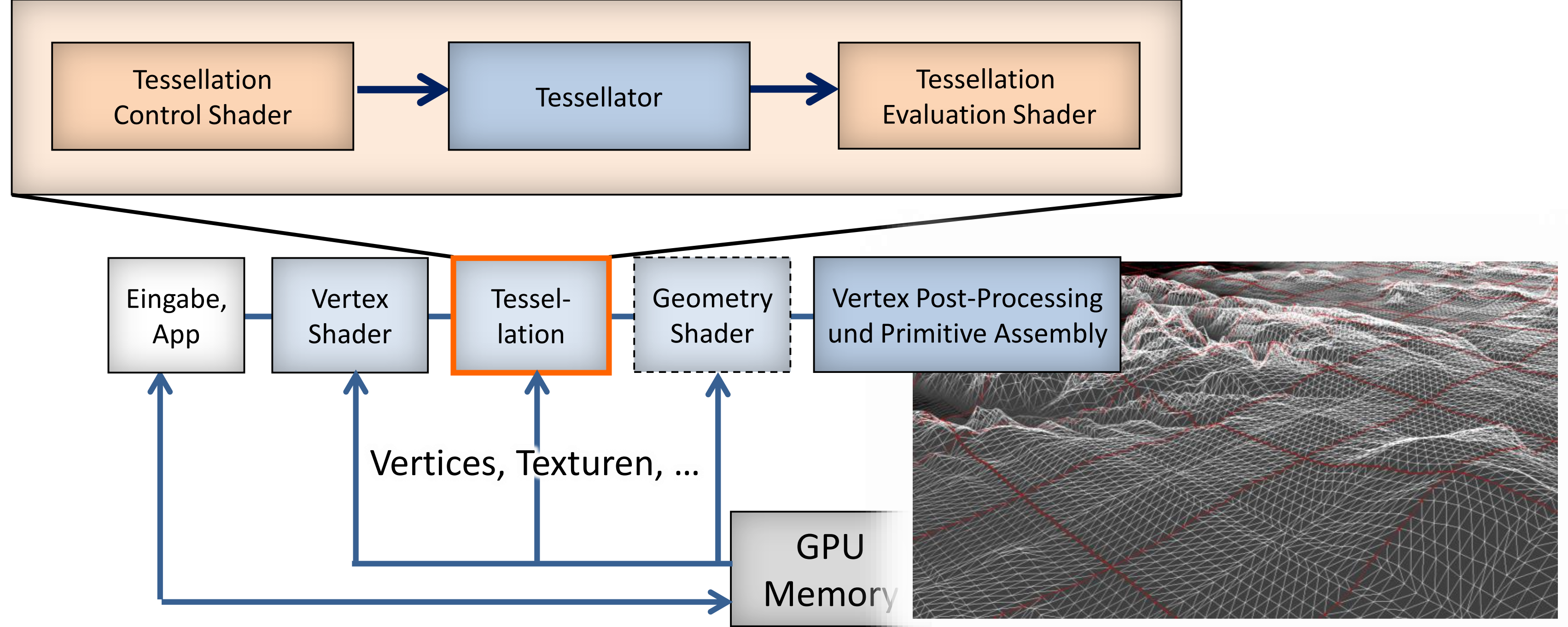
- benachbarte Dreiecke?
  - laufe entlang der Kanten den Polygons, jede Kante speichert Verweis auf Geschwisterkante und dieses das dazugehörige Polygon
- weitere effiziente Operationen: überprüfe Orientierung benachbarter Polygone, Selbstüberschneidung, Überlapp

Mehr Beispiele: https://jerryyin.info/geometry-processing-algorithms/half-edge/

Tesselierung in OpenGL 4.x

Bilder: learnopengl.com/GuestArticles/2021/Tessellation/Tessellation

- **Tessellation** (= Unterteilung von Primitiven) wird ab OpenGL 4.0/D3D 11 unterstützt und kann z.B. für **parametrische Flächen** oder **Displacement Mapping** genutzt werden
- in der Tessellation-Stufe gibt es 2 programmierbare Shader-Einheiten
  - Control Shader: bestimmt wie fein ein Eingabepatch unterteilt wird
  - Tessellator: nicht programmierbar, nimmt eigentliche Unterteilung vor
  - Evaluation Shader: berechnet Vertices der erzeugten Geometrie



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

```
layout (vertices=3) out;        // Anzahl Kontrollpunkte des Patch, Bedingung >0!
in vec2 texCoordVS[];           // Vertex Attribut
out vec2 texCoord[];

void main() {
  gl_out[ gl_InvocationID ].gl_Position =
     gl_in[ gl_InvocationID ].gl_Position;
  texCoord[ gl_InvocationID ] =
     texCoordVS[ gl_InvocationID ];

  // erste Invocation bestimmt
  // Unterteilung
  if ( gl_InvocationID == 0 ) {
   gl_TessLevelOuter[0] = …;
   gl_TessLevelOuter[1] = …;
   gl_TessLevelOuter[2] = …;
   gl_TessLevelInner[0] = …;
} }
```

gl_TessLevelInner
1 2 3 4
1
2
3
4
gl_TessLevelOuter

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

```
layout(triangles, equal_spacing, cw) in;
uniform mat4 matMVP;

void main() {
  vec3 p;
  p   = gl_TessCoord.x * gl_in[0].gl_Position;
  p += gl_TessCoord.y * gl_in[1].gl_Position;
  p += gl_TessCoord.z * gl_in[2].gl_Position;
  gl_Position = matMVP * vec4( p, 1.0 );
}
```

equal_spacing — Subdivide 1.00

fractional_even_spacing — Subdivide 2.00

fractional_odd_spacing — Subdivide 1.00

OpenGL (4.2+)-Pipeline

Eingabe, App → Vertex Shader → Tessel-lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri-zation → Fragment Shader → Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

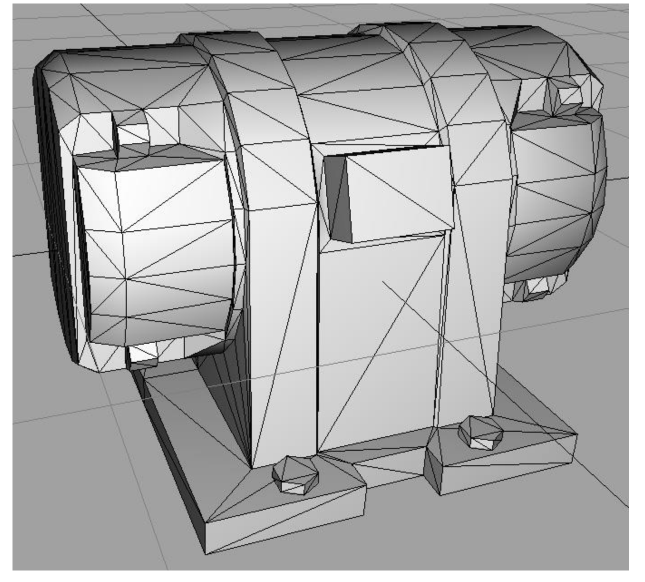
FBOs

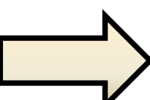
GPU Memory

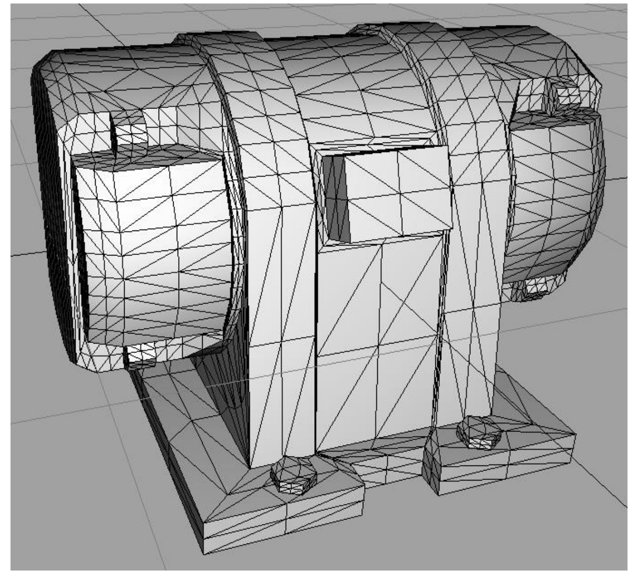
- programmierbare Stufe für Bearbeitung von Primitiven (optional)
  - Eingabe: **ein** Primitiv (Dreieck, Linie, Punkt)
  - Ausgabe: ein oder mehrere Primitive
  - Beispiel: Instanziierung,
    Dreieck → Kantenzug, Schattenvolumen, …

Geometry Shader

- Geometry Shader bearbeiten **Primitive** (Punkt, Linie, Dreieck) und können Primitive vervielfachen, entfernen oder umwandeln
- Ausführung nach dem Vertex und Tessellation Shader (erst dann sind die Primitive zusammengesetzt)
- kann aber nicht beliebig viel Geometrie ausgeben: insgesamt `glGet*( MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS, &anzahl )` Komponenten (seit 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 Vertex-Attribute (im VS: out vec3 vertexColorVS)
// ergibt hier ein Array von Farbwerten, da GS-Aufruf pro Primitiv erfolgt
in vec3 vertexColorVS[];

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
}
```

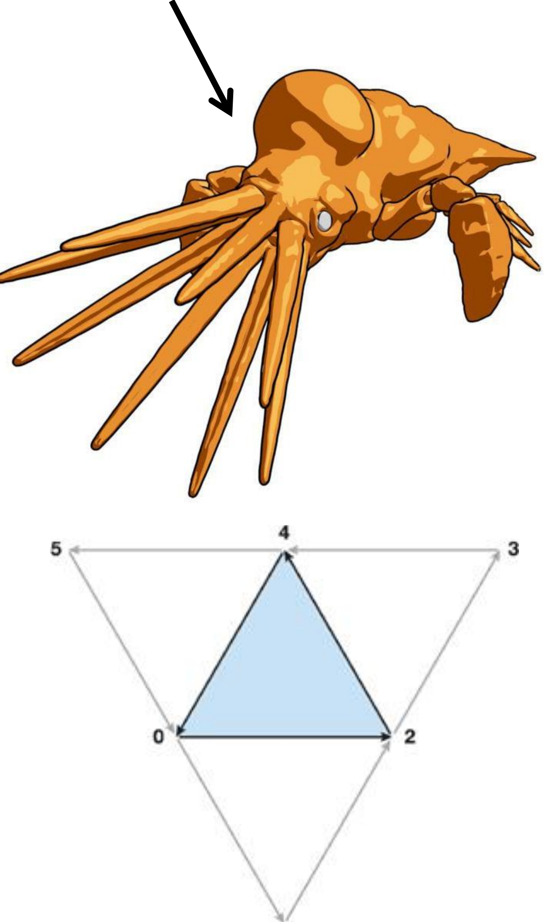
```
in vec3 vertexColorGS;     out vec4 out_color;
void main() { out_color = vec4( vertexColorGS.rgb, 1.0 ); }
```

Geometry Shader

Bild: https://www.sidefx.com/docs/houdini/nodes/shop/tooncolorshader.html

- Eingabeprim.: **points**, **lines**, **lines_adjacency**, **triangles**, **triangles_adjacency**
- Anzahl der Elemente pro Eingabeprimitiv: **gl_in.length()** (nicht in allen OpenGL-Versionen vorhanden, aber ohnehin durch den Eingabeprimitivtyp festgelegt: **layout(primitive_type) in;**)
- Zugriff auf Standardattribute über **gl_in[].<attrib>**, definiert sind:

```
gl_PerVertex {
        vec4 gl_Position;
        float gl_Pointsize;  // für Punkt-Rendering
         // Abstand zu User-Clip-Ebenen
         // ... im VS berechnet (optional)
        float gl_ClipDistance[];
} gl_in[];
```



Ausgabe

- Konfiguration der Ausgabe **layout (triangle_strip, max_vertices = 3) out;**
  - Ausgabeprimitive: **points**, **line_strip**, **triangle_strip**
  - Angabe von **max_vertices** hilft bei der Optimierung

Bsp.: Beleuchtung im GS statt VS

```
// Vertex Shader: reicht Attribute einfach weiter
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;
        … // Beleuchtungsberechnung wie im Vertex Shader vorhin
        vertexColorGS = vec4( kd );
        EmitVertex();
    }
    EndPrimitive();
}
```

Bsp.: Unterteilung von Dreiecken im GS

```
...
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... }
```

0 v_{20} 2

v_{01} v_{12}

1

Bsp.: Instantiierung im Geometry Shader

```
#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++ ) {
        // gl_in[i].gl_Position in Modellkoordinaten
        vec4 p = gl_in[i].gl_Position;
        p.x += gl_InvocationID % 5;
        p.y += gl_InvocationID / 5;
        gl_Position = matMVP * p;
        ...
        EmitVertex();
    }
    EndPrimitive();
}
```

Instantiierung vor dem Geometry Shader

- Die Zahl der Aufrufe eines Geometry Shader pro Primitiv ist begrenzt: `GL_MAX_GEOMETRY_SHADER_INVOCATIONS` (seit GF500: 32)
- es ist aber auch möglich von Applikationsseite zu Instanziieren:

```
glDrawArraysInstanced( GLenum mode, GLint first,
                       GLsizei count, GLsizei primcount )
glDrawElementsInstanced( ... )
```

  - „wiederholt" `primcount`-mal `glDraw*` ohne API-Mehraufwand
  - `primcount` darf „beliebig" groß sein
  - im Vertex Shader kann die Variable **gl_InstanceID** verwendet werden (nicht verwechseln mit **gl_InvocationID**)
- oft werden Instanzen auf Basis von Daten aus einer Lookup-Tabelle oder Textur beeinflusst (z.B. Position aus einer GPU-Partikelsimulation o.Ä.) oder mittels Daten aus einem weiteren Attribute Buffer (z.B. Matrizen)

Beispiel für Instancing: Rendering von Vegetation

- https://www.youtube.com/watch?v=AdabPy6xQ8A

Bild: Unreal Engine 4 Architectural Visualization

OpenGL (4.2+)-Pipeline

Eingabe, App → Vertex Shader → Tessel-lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri-zation → Fragment Shader → Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

FBOs

GPU Memory

Transform Feedback

- Ausgabe der Primitive nach Geometrieverarb. in den Speicher der GPU
- mögliche Anwendungen
  - mehrfache Verwendung aufwändiger Berechnungen, z.B. Animationen wie Vertex Skinning (auch Download der Daten zur CPU möglich)
  - Level-of-Detail: aktualisiere Geometrie nur wenn nötig
  - Partikelsysteme

Weitere ausgewählte OpenGL-Funktionalität

Vielfältige Varianten beim Rendering vom Geometrie

- Instancing, Base Vertex, `glDrawRangeElements`, Conditional, Indirect
  - https://www.khronos.org/opengl/wiki/Vertex_Rendering#Instancing
- Command Lists (NV-Extension), Command Buffers (Vulkan)
  - dafür und weitere Optimierungen siehe z.B.
    https://on-demand.gputechconf.com/gtc/2015/presentation/S5135-Christoph-Kubisch-Pierre-Boudier.pdf

Beispiel: Indirect Rendering

- `glDrawArrays`-Aufrufe deren Parameter in einem Puffer vom Target `GL_DRAW_INDIRECT_BUFFER` gespeichert sind

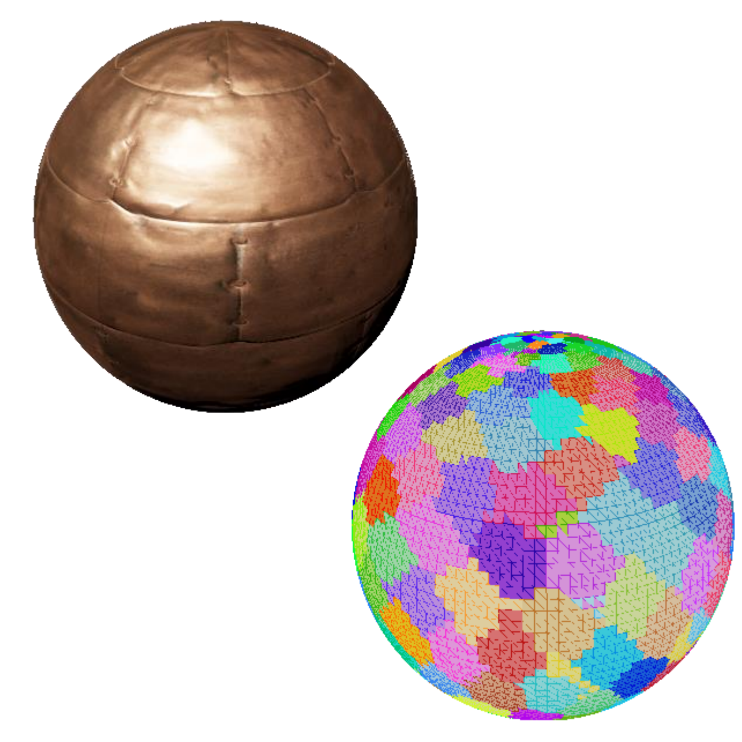
```
typedef struct {
  GLuint  count, instanceCount, first, baseInstance;
} DrawArraysIndirectCommand;

glDrawArraysIndirect( GL_TRIANGLES, &indirect );
```

Beispiel: Assassin’s Creed Unity

Komplexe Geometrie, viel Instanziierung

Beispiel: Assassin’s Creed Unity

- große Teile der Geometrie bestehen aus Teilmodulen so, dass insgesamt >50000 Draw Calls (>15000 mit Instanziierung) notwendig wären
- `glDrawInstancedIndirect`: Puffer mit zu zeichnenden Instanzen wird auf der GPU selbst gefüllt
- Vertex Daten werden in Shadern aus gemeinsamen Puffern gelesen
- spezielle Culling-Verfahren auf der GPU → InCG-Vorlesung (Bezüge zu klassischen Clipping-Verfahren, Outcodes, …)



Moderne Grafik-Pipeline (OpenGL, Vulkan, DX12)

Eingabe, App → Vertex Shader → Tessel-lation → Geometry Shader → Vertex Post-Processing und Primitive Assembly → Rasteri-zation → Fragment Shader → Fragment Operationen

GPU Memory

- Texturen, Vertex/Index Buffer, Konstanten, …
- Vertices, Texturen, …
- „Transform Feedback“: Vertices, …
- Texturen
- FBOs

- programmierbare Stufe
  - Eingabe: Fragmente mit 2D-Position, interpolierten Attributen, …
  - Ausgabe: Farbe(n) und Opazität(en), optional Tiefe, …

ShaderGen

- https://github.com/mojocorp/ShaderGen

3Dlabs GLSL ShaderGen
File View Model Help
Select GL Mode
FIXED FUNCTIONALITY MODE EQUIVALENT SHADER MODE
3Dlabs Graphics Evolved
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
GL_SPECULAR
GL_DIFFUSE
GL_SPOT_DIRECTION 0.0,0.0,-1.0
GL_SPOT_EXPONENT 0.0
GL_SPOT_CUTOFF 180.0
GL_CONSTANT_ATTENUATION 1.0
GL_LINEAR_ATTENUATION 0.0
GL_QUADRATIC_ATTENUATION 0.0
3Dlabs GLSL ShaderGen

Bemerkungen zu OpenGL Shader

- Shader für programmierbare Stufen der Grafik-Pipeline
  - Vertex-, Geometry- und Fragment-Shader: in dieser Vorlesung
  - Tessellation-Control und Tessellation-Evaluation Shader
  - Compute Shader (ursprünglich nur für GPGPU Berechnungen)
  - GLSL-Operationen und -Funktionen sind größtenteils identisch
  - Semantik und Layout der Eingangs- und Ausgangsdaten kann variieren
- 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
  - oder verschiebt Berechnungen, um Flaschenhälse zu vermeiden, z.B.
    - (Teile der) Beleuchtungsberechnung zw. VS, GS und FS verschieben
    - Berechnung im FS zu teuer → feiner Tesselieren und pro Vertex
    - komplexe Animation im VS/GS über Transform Feedback, oder per Compute Shader einen Vertex Buffer schreiben

Inhalt des Vorlesungsteils

Wichtige Aspekte programmierbaren/modernen Renderings

- Vertex und Fragment Shader
  - globale Variablen (Transformationsmatrix, Lichtquellen, Texturen, …)
  - Ein- und Ausgabevariablen (z.B. Vertex-Attribute)
- Geometriedaten
  - … im Speicher der GPU
  - Optimierung, Performanz
  - Instanziierung, Geometry Shader und Tesselierung
- weitere Konzepte in der Vorlesung
  - Fragment Operationen (auch in klassischem OpenGL)
  - Compute Shader, Shader Storage Buffer Objects
  - Mesh Shader, Raytracing (Vulkan)
  - Ausblicke, Rendering-Techniken

OpenGL (4.2+)-Pipeline

Eingabe, App
Vertex Shader
Tessel-lation
Geometry Shader
Vertex Post-Processing und Primitive Assembly
Rasteri-zation
Fragment Shader
Fragment Operationen

Vertices, Texturen, …

„Transform Feedback“: Vertices, …

Texturen

GPU Memory

FBOs

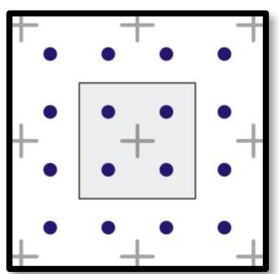
Fragment Operationen

- nicht-programmierbare Stufe („fixed-function“)
  - Tiefentest mit Z-Buffer
  - Maskierung und Blending
  - Schreiben in den Frame Buffer

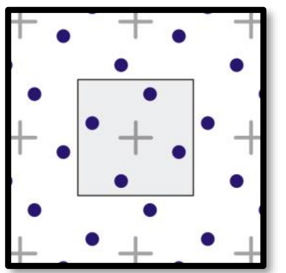
Rasterisierung und Antialiasing

Supersampling (Full Sample Antialiasing, FSAA)

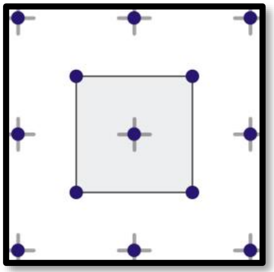
- Taxonomie: Pipeline erzeugt **Fragmente**, die später die Pixelfarbe bestimmen
- Supersampling: erhöhter Speicherbedarf für Framebuffer und Tiefenpuffer
  - fixes Schema mit mehreren Samples pro Pixel, anschließende Mittelung



einfaches Supersampling



Rotated Grid Supersampling



2x Quincunx (NVIDIA)

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

Multisample Antialiasing (MSAA) / Coverage Sampling AA (CSAA)

- Idee: Schattierungsberechnung ist teurer als Sichtbarkeit
- MSAA: nur Supersampling der Tiefe (Sichtbarkeit), nur 1 Farbe pro Pixel und Δ
- CSAA: speichert pro Pixel nur wenige Farb-Samples (z.B. 4) und nimmt Zuordnung schattierte Samples ↔ Sichtbarkeits-Samples (z.B. 16) vor
- in OpenGL: festgelegt bei der Erzeugung des Framebuffers/Rendertargets (z.B. mit `GLUT_MULTISAMPLE`), besser mit Render Targets (später)

- heutzutage: TXAA = temporales Anti-Aliasing (programmiert mit Shader)

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
- Alpha-Test: nur in klassischem OpenGL

Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops → Alpha Test → Stencil Test → Depth Buffer Test → Occlusion Query → Blending → SRGB Conversion → Dithering → Logical Operations → Frame-buffer

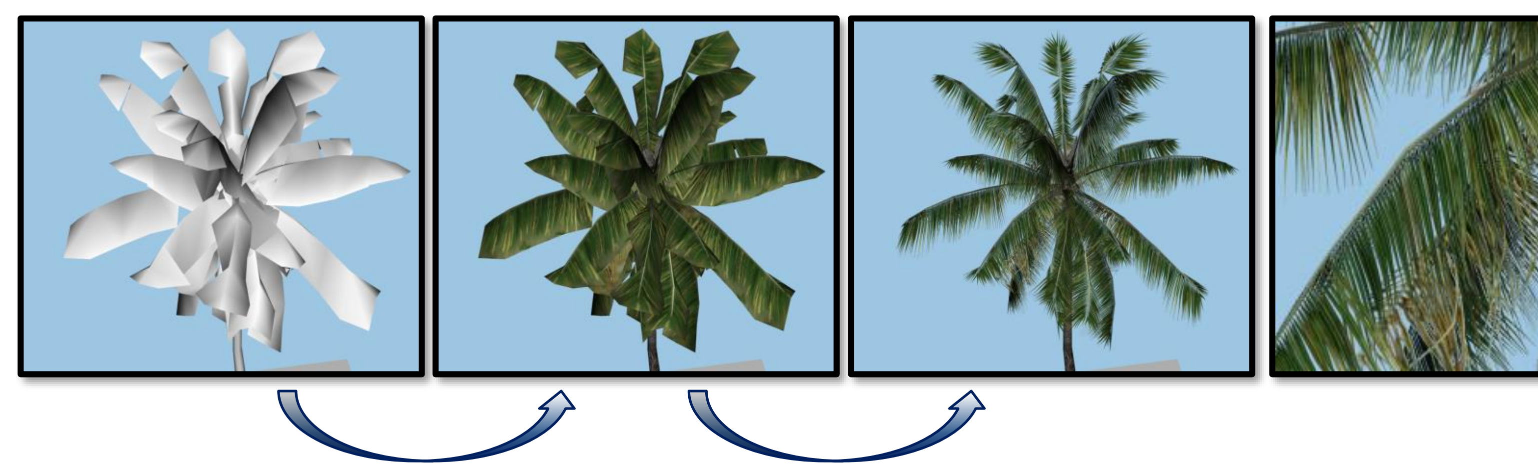
Klassisches OpenGL: Alpha-Test

- verwerfe Fragmente aufgrund ihres Alpha-Wertes

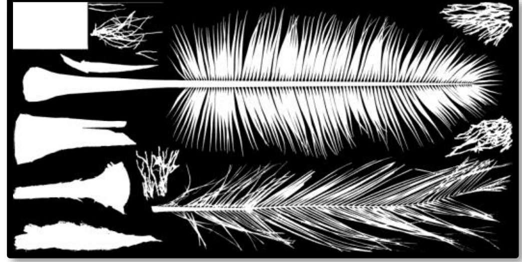
```
glAlphaFunc( func, value )
```

- 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

Klassisches OpenGL: 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 Shader ein bedingtes **discard** implementieren
  - es muss zwar selbst implementiert werden, ist aber flexibler als ein konfigurierbarer Alpha-Test

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

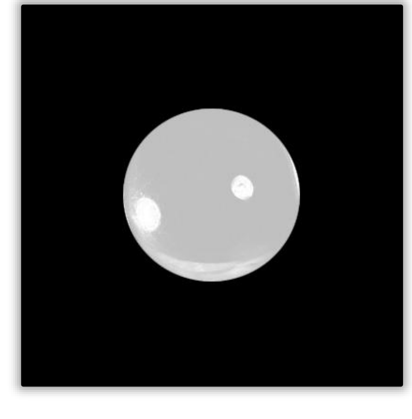
Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops → Stencil Test → Depth Buffer Test → Occlusion Query → Blending → SRGB Conversion → Dithering → Logical Operations → Frame-buffer

OpenGL Blending

- Kombination von Farbwerten des Fragments („Source“) mit Farben von Fragmenten im Frame-Buffer („Destination“) anhand Gewichten, die aus den RGBA-Werten erzeugt werden oder als konstant festgelegt werden

RGB der Fragmente

Alpha der Fragmente



RGB im Framebuffer

Alpha im Framebuffer
(in diesem Beispiel egal)

Blending Resultat

FragmentRGB * FragmentAlpha +
FramebufferRGB * (1-FragmentAlpha)

OpenGL Blending

- Kombination von Farbwerten des Fragments („Source“) mit Farben von Fragmenten im Frame-Buffer („Destination“) anhand Gewichten, die aus den RGBA-Werten erzeugt werden oder als konstant festgelegt werden

Source-RGB (R_S, G_S, B_S, A_S) Source-Alpha

* SourceFactor z.B. (A_S)

Blending Resultat

Op z.B. +

* DestFactor z.B. $(1-A_S)$

Dest-RGB (R_D, G_D, B_D, A_D) Dest-Alpha

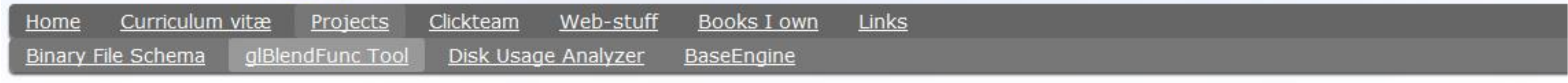
OpenGL Blending

- Einschalten und Blending-Funktion
  ```
  glEnable( GL_BLEND )
  glBlendFunc( sfactor, dfactor )
  ```
  - typische Anwendung: Zeichnen semitransparenter Objekte („Alpha Blending“)
    ```
    glBlendFunc( GL_SRC_ALPHA,
                 GL_ONE_MINUS_SRC_ALPHA )
    ```
  - Beispiele: `GL_SRC_ALPHA` = (A_S, A_S, A_S, A_S),
    `GL_ONE_MINUS_SRC_ALPHA` = $(1-A_S, 1-A_S, 1-A_S, 1-A_S)$,
    `GL_ZERO` = (0, 0, 0, 0), `GL_ONE` = (1, 1, 1, 1), …
  - spezielle Blending Funktionen, z.B.
    `GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONST_ALPHA`
    für konstante Farbe bzw. Transparenz (`glBlendColor`)
- 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 in der Visualisierung oder morphologische Filter)

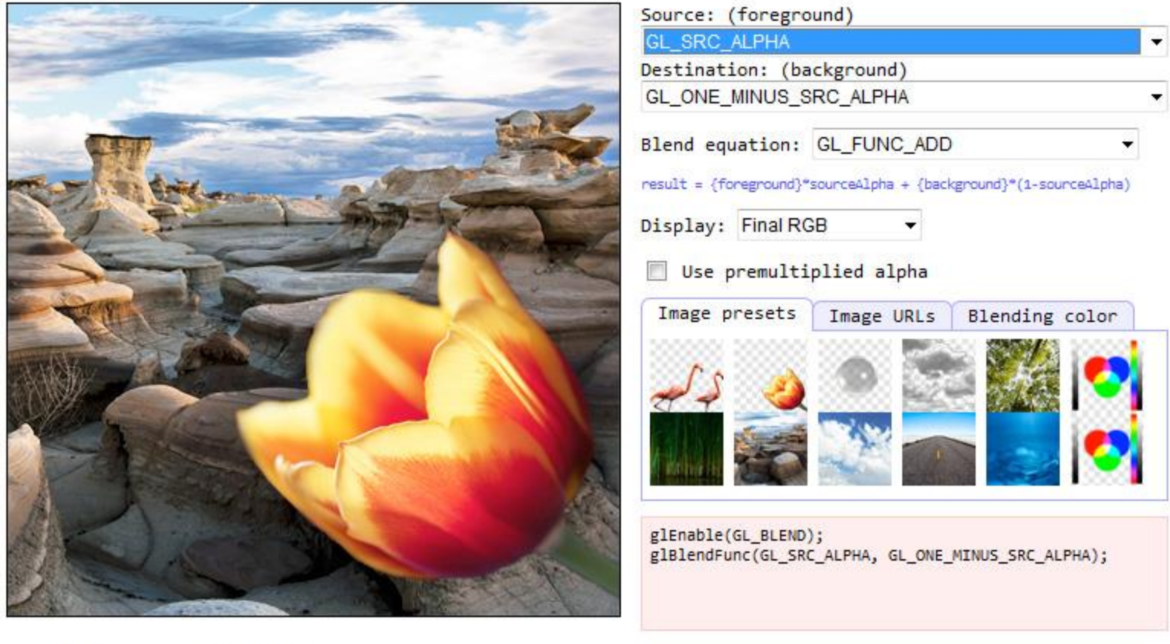
OpenGL Blending

- http://www.andersriggelsen.dk/glblendfunc.php

Anders Riggelsen



Visual glBlendFunc + glBlendEquation Tool

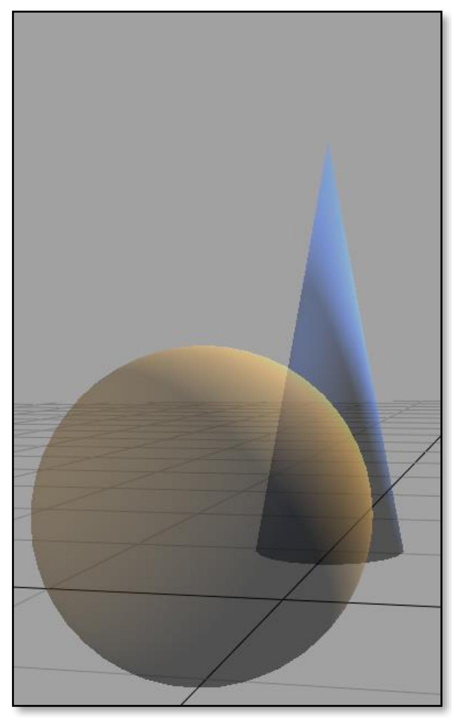


Use this tool to quickly visualize how the different blending modes work in OpenGL.

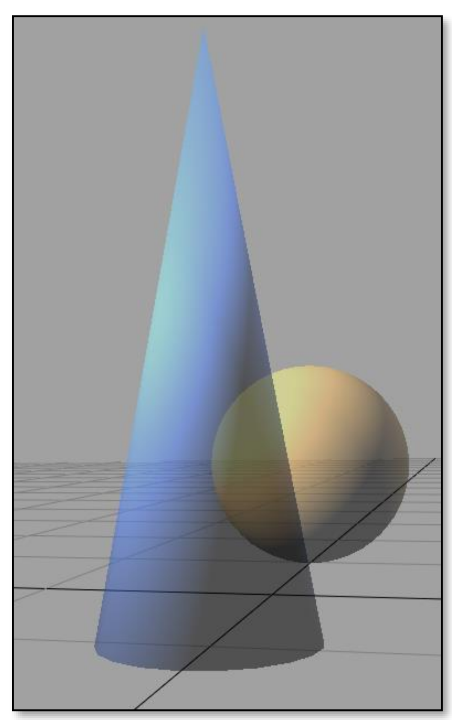
OpenGL Blending

Semitransparente Objekte

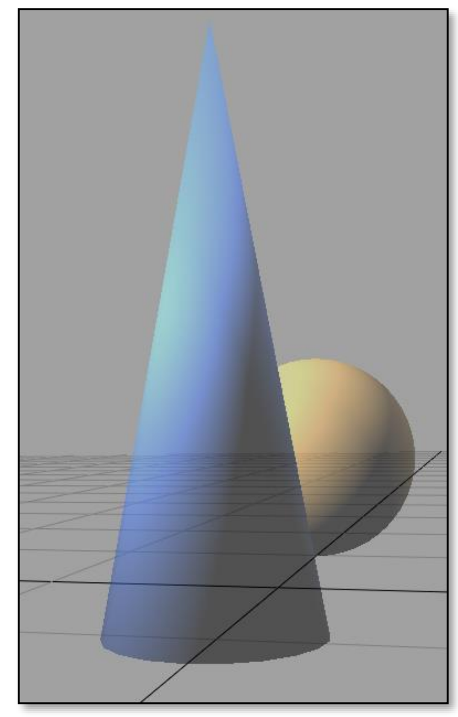
- Darstellung erfordert Tiefensortierung: Blending ist i.A. nicht kommutativ, z.B. **mix(mix(** c_0, c_1, α_1**),** c_2, α_2 **)** $\neq$ **mix(mix(** c_0, c_2, α_2**),** c_1, α_1 **)**
- weiteres Problem ist der Tiefentest bei semitransparenten Flächen
- Beispiel: braune Kugel und blauer Kegel (beide semitransparent)



Kugel vor dem Kegel, Kegel zuerst gezeichnet



Kugel hinter dem Kegel, Kugel zuerst gezeichnet



Kugel hinter Kegel, letzterer zuerst gezeichnet: Tiefentest verhindert Blending

Semitransparente Objekte ohne Tiefensortierung

- im Bild überlappende semitransparente Flächen werden **ohne Sortierung nicht korrekt** dargestellt

- eine „Notlösung" bei der Rasterisierung: transparente Polygone verschwinden hinter opaken, aber ändern nicht den z-Buffer (Renderstate **glDepthMask**):

```
...
// verwerfe nicht-opake Fragmente
if (color.a < 1.0) discard;
...
```

```
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 );
```

Alpha Blending vs. Alpha Test

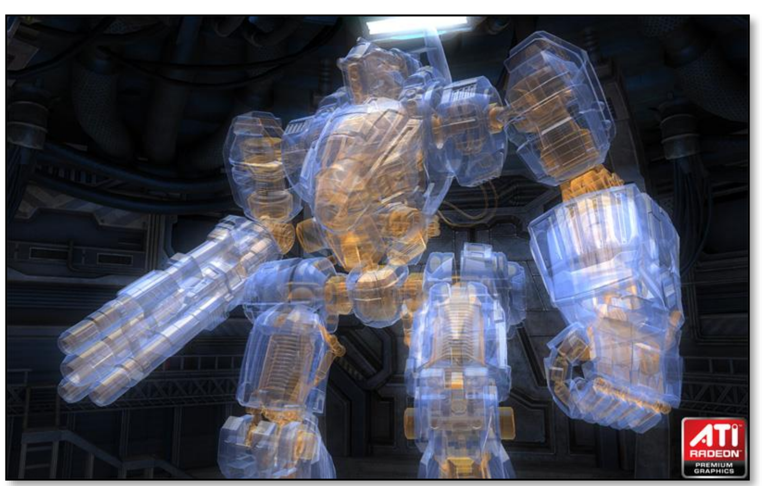
- Beispiel: Alpha Blending ohne Alpha Test (bzw. **discard)** würde hier zu Fehlern führen – warum?

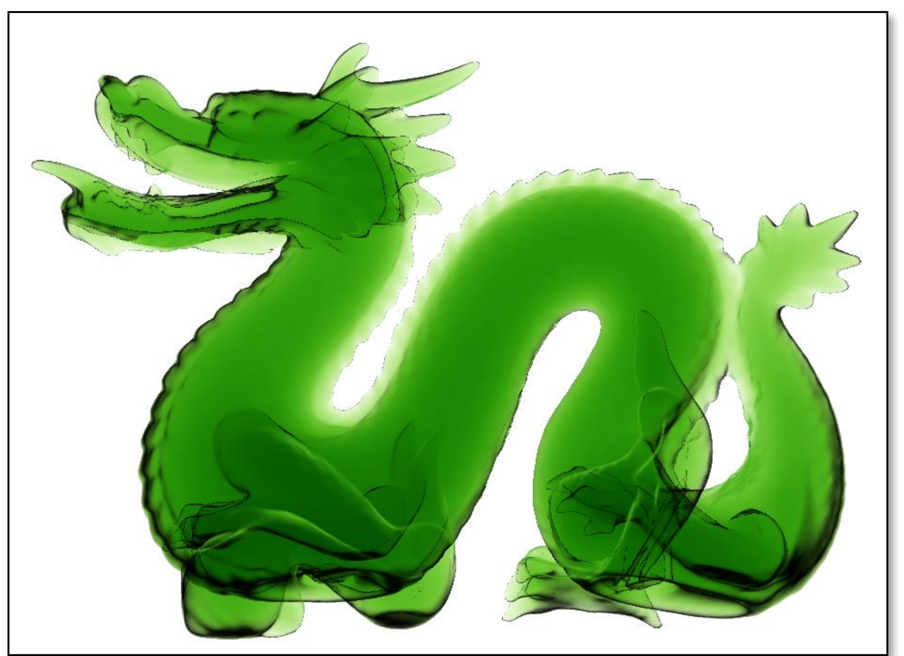
24 Bit Farbinformation

8 Bit Alpha-Kanal
schwarz = transparent

Transparenz ohne Sortierung

- es gibt auch kommutative Blending-Modi (funktionieren ohne Sortierung)
  - Multiplikation der Source- und Destination-Farbe
    `glBlendFunc( GL_DST_COLOR, GL_ZERO )`
  - Addition der beider RGB(A)-Vektoren
    `glBlendFunc( GL_ONE, GL_ONE )`
- Alpha-Blending kann nur eine grobe Approximation sein
  - Aussehen/Transluzenz hängt auch von der Dicke des Objekts ab
  - siehe „Order-Independent Transparency"-Techniken





Fragment-Verarbeitung

- Rasterisierung erzeugt Fragmente, vor dem Schreiben in den Framebuffer müssen sieTests und Framebuffer-Operationen durchlaufen
- Framebuffer besitzt zus. zu Farb- und Tiefenpuffer optional einen **Stencil Buffer** („Stanz-Maske“, „Schablone“), um Werte/Zustände für jeden Pixel zu speichern (8-Bit pro Pixel) und Fragmente ggf. zu verwerfen, zählen, …

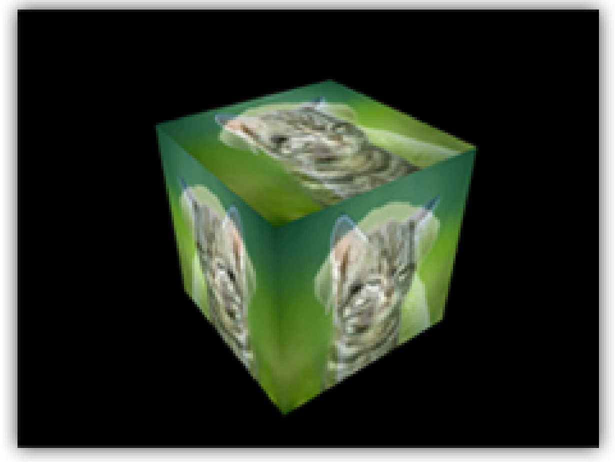
Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops →

Stencil Test → Depth Buffer Test → Occlusion Query → Blending →

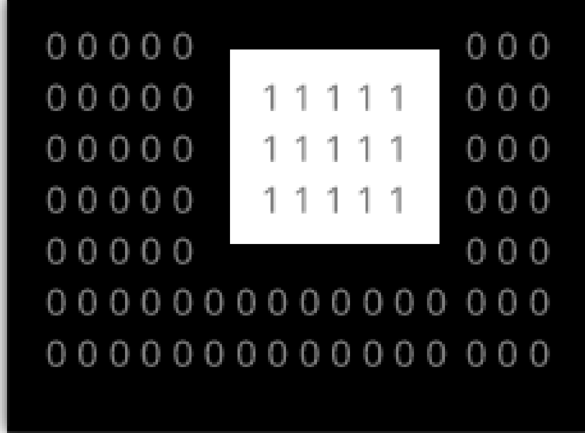
SRGB Conversion → Dithering → Logical Operations → Frame-buffer

OpenGL-Framebuffer

- mit einem **Stencil Test** kann dann entscheiden werden, ob ein Fragment gezeichnet wird, z.B. durch Vergleich des Stencil-Werts des Pixels/Fragments im Stencil Buffer mit einem Referenzwert
- typische Anwendungen
  - Maskierung von Bildteilen (siehe Beispiel unten)
  - projektive Schatten und Schattenvolumen
  - Stencil-Routing



Color buffer without stencil test



Stencil buffer

Color buffer with stencil test

Bild: https://open.gl/depthstencils

Stencil Test und Tiefentest

Reihenfolge der Fragment-Tests (Alpha Test nur in klassischem 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[i].color = fragment.color[i];
      buffer.depth = fragment.depth;
      buffer.stencil = zpass();
    }
```

- Alpha-Test ist der einzige Test, der nicht vom Framebuffer-Inhalt abhängt und kann daher per **discard** in einem Shader durchgeführt werden

Stencil Test und Tiefentest

Reihenfolge der Fragment-Tests (Alpha Test nur in klassischem 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[i].color = fragment.color[i];
      buffer.depth = fragment.depth;
      buffer.stencil = zpass();
    } else
      // Stencil Test bestanden, Tiefentest nicht
      buffer.stencil = zfail();
  } else
    // Stencil Test nicht bestanden
    buffer.stencil = fail();
```

Stencil Test und Operationen

Konfiguration des Stencil Tests

- Stencil-Test festlegen
  `glStencilFunc( stencilfunc, refstencil, bitmask )`
  - vergleiche mit Referenzwert *refstencil* 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 mit Wert `0xCD` in den Stencil-Buffer

```
glStencilFunc( GL_ALWAYS, 0xCD, 0xFF );
glStencilOp( GL_REPLACE, GL_REPLACE, GL_REPLACE );
```

  - zeichne Objekte nur dort, wo unterstes Bit Stencil nicht gesetzt

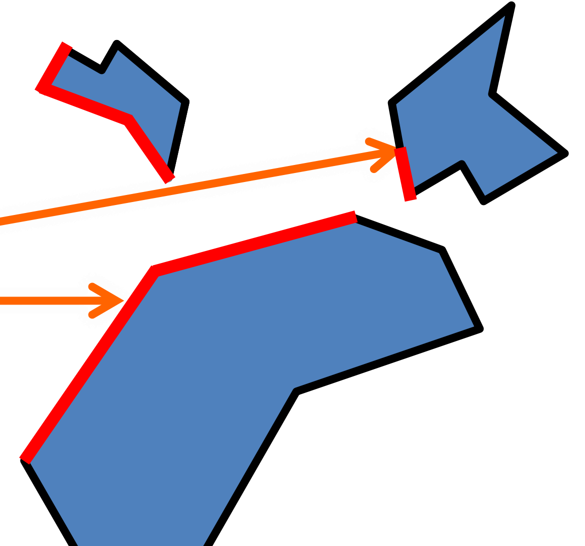
```
glStencilFunc( GL_NOT_EQUAL, 0x1, 0x1 );
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
```

Schattenvolumen (Crow, SIGGRAPH '77)

Schattenvolumen (Crow, SIGGRAPH '77)

Stencil Buffer zum Zählen der Schnitte mit Schattenvolumen

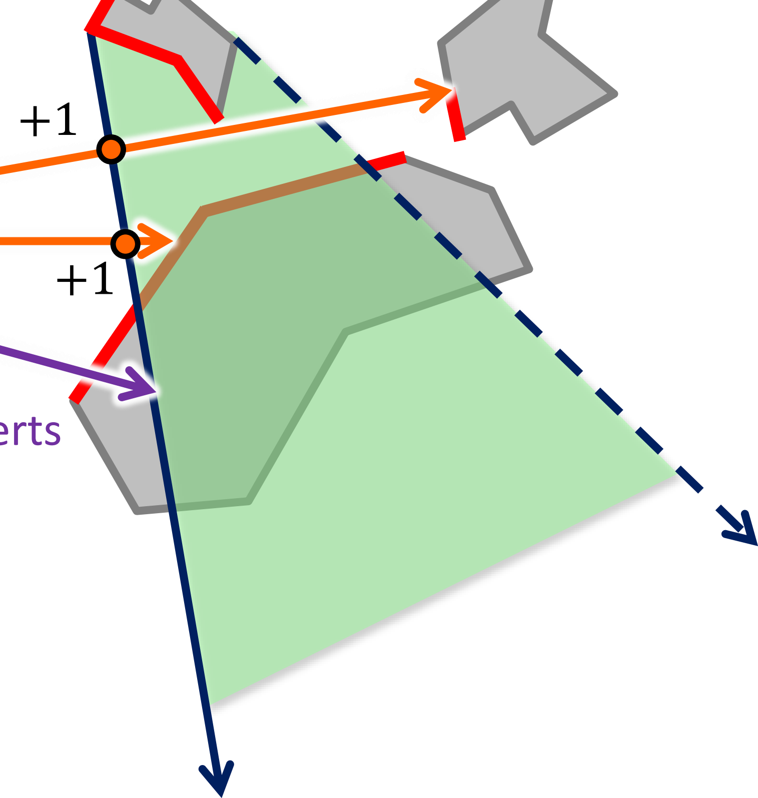
- zuerst werden die Objekte gezeichnet (sichtbare Fläche im Z-Buffer)
- anschließend wird Schreiben in den Farb- und Tiefenpuffer deaktiviert
- zeichne dann die Schattenvolumen
  - Vorderseiten erhöhen den Stencil Buffer
  - Rückseiten erniedrigen den Stencil Buffer
  - Unterscheidung mittels Backface Culling
- Schatten ist dort, wo Stencil > 0



Schattenvolumen (Crow, SIGGRAPH '77)

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
  - Unterscheidung mittels Backface Culling



Tiefentest nicht bestanden
→ keine Änderung des Stencil-Werts

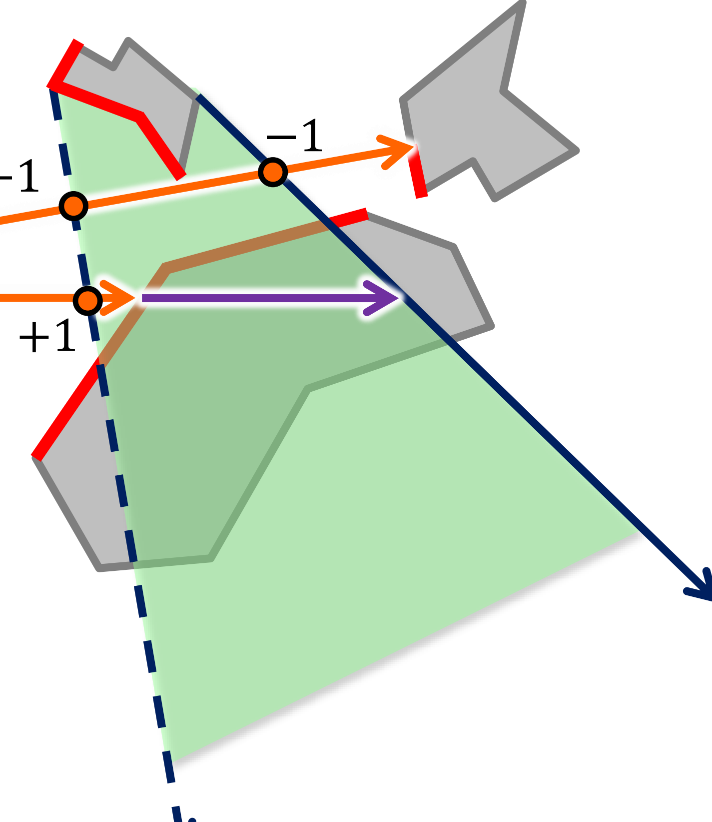
Backface Culling: Flächen kann man abhängig davon, ob man auf Vorder- oder Rückseite blickt verwerfen lassen

```
glEnable( GL_CULL_FACE );
glCullFace( GL_BACK );
```

Schattenvolumen (Crow, SIGGRAPH '77)

Stencil Buffer zum Zählen der Schnitte mit Schattenvolumen

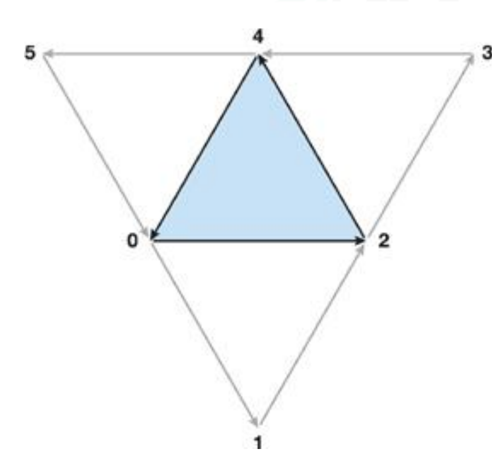
- zuerst werden die Objekte gezeichnet
- anschließend wird Schreiben in den Farb- und Tiefenpuffer deaktiviert
- zeichne dann die Schattenvolumen
  - Vorderseiten erhöhen den Stencil Buffer
  - Rückseiten erniedrigen den Stencil Buffer
  - Unterscheidung mittels Backface Culling
- Schatten ist dort, wo Stencil > 0

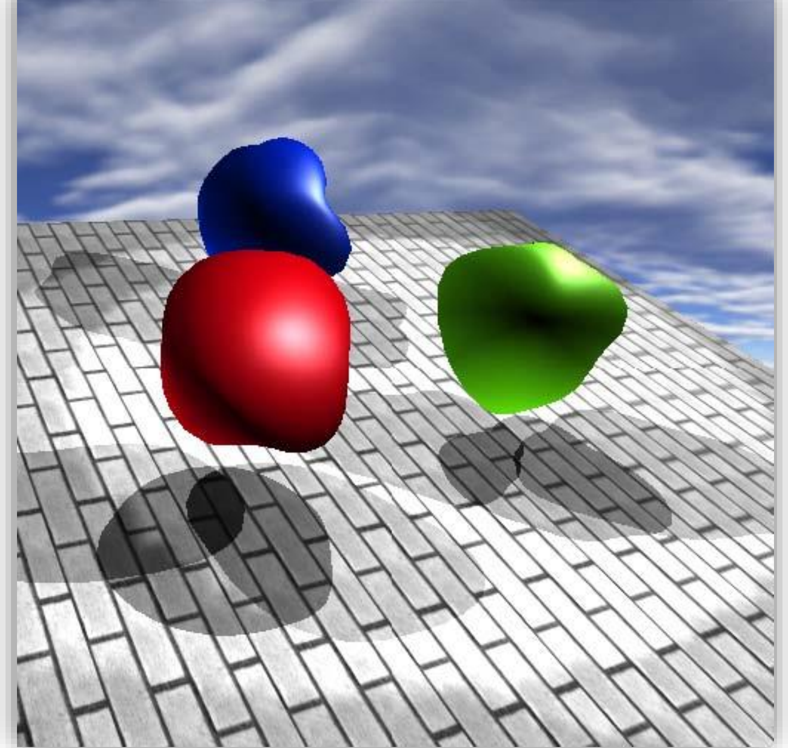


Schattenvolumen eines Dreiecksnetzes

Schritte

- bestimme die Schattenvolumen für jedes Objekt und jede Lichtquelle, z.B. mit Geometry Shader und `GL_TRIANGLES_ADJACENCY`
- stelle fest, welche Oberflächenpunkte in einem Schattenvolumen sind





Bilder: Stefan Brabec

Fragment-Verarbeitung

Scissor-Test (historisch)

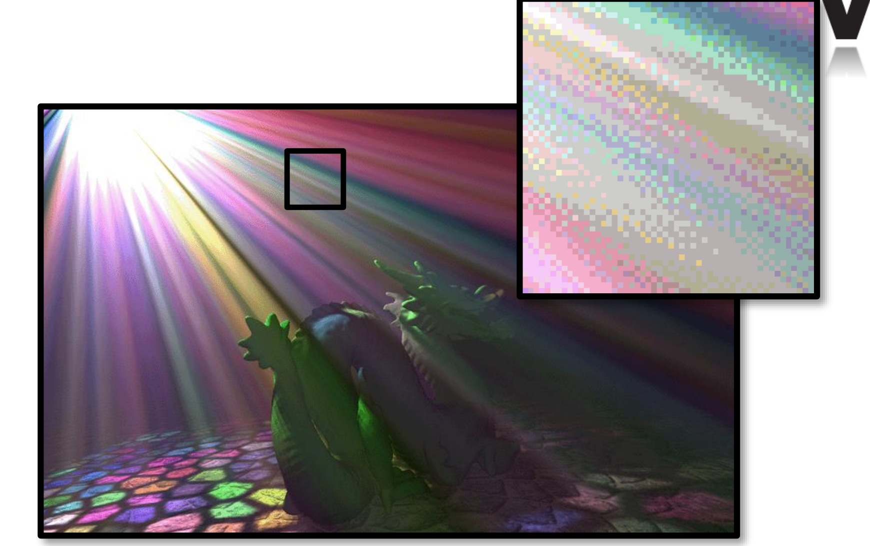
- dient lediglich dazu Fragmente außerhalb eines Rechtecks zu entfernen
  `glScissor( x, y, width, height )`
- wird verwendet, um kleinere Bereiche des Fensters zu aktualisieren (Löschen mit `glClear()`)
- vergleichbar mit einer rechteckigen Stencil-Maske

Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops → Stencil Test → Depth Buffer Test → Occlusion Query → Blending → SRGB Conversion → Dithering → Logical Operations → Frame-buffer

Fragment-Verarbeitung

Dithering (historisch)

- Fehlerdiffusion, nur historisch für Hardware ohne „TrueColor“-Darstellung



Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops → Stencil Test → Depth Buffer Test → Occlusion Query → Blending → SRGB Conversion → **Dithering** → Logical Operations → Framebuffer

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

Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops → Stencil Test → Depth Buffer Test → Occlusion Query → Blending → SRGB Conversion → Dithering → Logical Operations → Frame-buffer

Fragment-Verarbeitung

Occlusion Query

- Zählen der Fragmente die Stencil und Tiefentest bestehen, diese müssen nicht zwangsläufig Framebuffer verändern, siehe `gl{Depth|Color}Mask`
- Anwendung Occlusion Culling: besteht kein Fragment eines Hüllkörpers den Tiefentest, so ist auch das dazugehörige Objekt verdeckt und muss nicht gezeichnet werden

Ausblick

Echtzeit-Rendering Techniken

- … machen intensiv Gebrauch von den speziellen Funktionen der Grafik-Hardware

Fragment → Ownership Test → Scissor Test → Multisample Fragment Ops → Stencil Test → Depth Buffer Test → Occlusion Query → Blending → SRGB Conversion → Dithering → Logical Operations → Frame-buffer

Inhalt des Vorlesungsteils

Wichtige Aspekte programmierbaren/modernen Renderings

- Vertex und Fragment Shader
  - globale Variablen (Transformationsmatrix, Lichtquellen, Texturen, …)
  - Ein- und Ausgabevariablen (z.B. Vertex-Attribute)
- Geometriedaten
  - … im Speicher der GPU
  - Optimierung, Performanz
  - Instanziierung, Geometry Shader und Tesselierung
- weitere Konzepte in der Vorlesung
  - Fragment Operationen (auch in klassischem OpenGL)
  - Compute Shader, Shader Storage Buffer Objects
  - Mesh Shader, Raytracing (Vulkan)
  - Ausblicke, Rendering-Techniken

Weitere ausgewählte OpenGL-Funktionalität

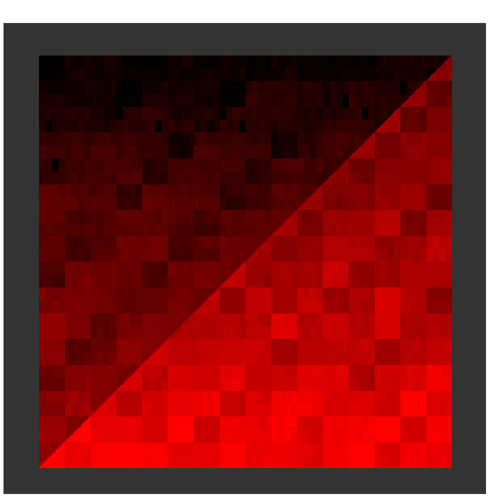
Atomare Zähler in „Atomic Counter Buffers“

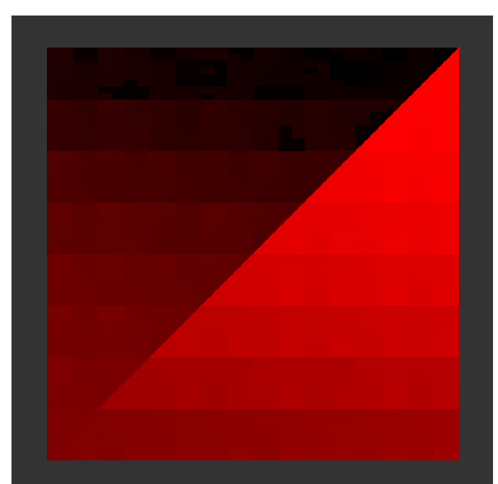
- atomare Zähler ohne explizite Synchronisation in **GL_ATOMIC_COUNTER_BUFFER** [1] (Vorsicht: langsam)
  - unsigned int, Inkrementieren/Dekrementieren

```
uniform atomic_uint counter; …
atomicCounterIncrement( counter );
```

- Bsp. Rasterisierungsabfolge darstellen mittels Atomic Counters (keine ernsthafte Anwendung, aber interessante Einblicke in HW)

GeForce GTX 780





Radeon HD 6970

[1] http://www.opengl.org/wiki/Atomic_Counter

Bilder: https://www.geeks3d.com/20131031/opengl-4-2-atomic-counters-rasterization-pattern-helper-for-rendering-optimization-windows-linux/a

Weitere ausgewählte OpenGL-Funktionalität

Shader Storage Buffer Object

- Speicher mit wahlfreien und atomaren Zugriffen für GLSL-Programme
- Beispiel:

```
int *data;
GLuint ssbo;
glGenBuffers( 1, &ssbo );
glBindBuffer( GL_SHADER_STORAGE_BUFFER, ssbo );
glBufferData( GL_SHADER_STORAGE_BUFFER, size, data, usage );
glBindBufferBase( GL_SHADER_STORAGE_BUFFER, index, ssbo );
glBindBuffer( GL_SHADER_STORAGE_BUFFER, 0 );
```

- Zugriff im Shader

```
struct mystruct {
      int value;
      float  data[ 7 ];  };

layout(std430, binding = 3) buffer bmystruct {
   mystruct payload[]; };
…
mystruct d = payload[ i ];
```

Weitere ausgewählte OpenGL-Funktionalität

Shader Storage Buffer Object

- Speicher mit wahlfreien und atomaren Zugriffen für GLSL-Programme
- Änderungen mit atomaren Zugriffen (nur Integers)

  ***u*int atomic{Add|Min|Max}(inout *u*int mem, *u*int data)**
  ***u*int atomic{And|Or|Xor} (inout *u*int mem, *u*int data)**
  ***u*int atomicExchange(inout *u*int mem, *u*int data)**
  ***u*int atomicCompSwap(inout *u*int mem, *u*int compare, *u*int data)**

- Memory Qualifier (z.B. **readonly**, …)
- ähnlich: **Image load/store** für beliebiges Schreiben und Lesen von Texturen (atomar möglich für 32-Bit Integer Images)

Compute Shader: Single Program, Multiple Data

- Ausführung eines GLSL-Programms losgelöst von der Rendering-Pipeline
  - keine festdefinierten Ein-/Ausgaben bzw. Anzahl der Aufrufe (wie bspw. 1x pro Vertex bei einem Vertex Shader)
  - innerhalb eines OpenGL-Kontext ausführbar, d.h. auf denselben Ressourcen, die in der Rendering-Pipeline genutzt werden (bei CUDA oder OpenCL nicht der Fall)
- Ein- und Ausgaben sind Storage Buffer oder Texturen
- Ausführung

```
void glDispatchCompute( GLuint num_groups_x,
                        GLuint num_groups_y,
                        GLuint num_groups_z );
```

  - Work Group ist die kleinste Anzahl von Compute-Operationen
  - Anzahl WGs für 1D-, 2D- oder 3D-Operationen beschränkt in jeder Dimension (z.B. Liste von Partikeln (1D), Bilder (2D), Volumen (3D))
  - WGs selbst haben unterschiedliche lokale Größen

Beispiel: Compute Shader

- Operation auf einem Bild/Textur in 16^2 2D-Workgroups

```
layout (local_size_x = 16, local_size_y = 16) in; // Workgroup-Größe

uniform image2D destTex;            // angelegt wie Texturen
shared float foo[256];              // Shared Memory zw. Threads

void main() {
    foo[ gl_LocalInvocationIndex ] = 0.0;
    // erzwinge foo-Änderungen beendet + sichtbar für andere Invocations
    memoryBarrierShared();
    // warte bis jeder Thread an dieser Stelle angekommen ist
    barrier();

    ivec2 storePos = ivec2( gl_GlobalInvocationID.xy );
    vec4 color = …;
    imageStore( destTex, storePos, color );
}
```

- weitere eingebaute Variablen: **gl_NumWorkGroups, gl_WorkGroupID, gl_LocalInvocationID**

Beispiel: Compute Shader

- Operation auf einem Bild/Textur in 16^2 2D-Workgroups

```
layout (local_size_x = 16, local_size_y = 16) in; // Workgroup-Größe

uniform image2D destTex;            // angeleg vi
shared float foo[256];              // Shared M

void main() {
    foo[ gl_LocalInvocationIndex ] = 0.0;
    // erzwinge foo-Änderungen beendet + sichtbar für andere Invocations
    memoryBarrierShared();
    // warte bis jeder Thread an dieser Stelle angekommen ist
    barrier();

    ivec2 storePos = ivec2( gl_GlobalInvocationID.xy );
    vec4 color = ...;
    imageStore( destTex, storePos, color );
}
```

Beschränkt in jeder Dimension *und* insgesamt

- weitere eingebaute Variablen: **gl_NumWorkGroups, gl_WorkGroupID, gl_LocalInvocationID**

Mesh Shader

- komplexe Dreiecksnetze werden in Form von Meshlets (kleinere Teile davon) mittels Compute Shader bearbeitet – und direkt rasterisiert

MESHLETS

TRADITIONAL PIPELINE

VERTEX ATTRIBUTE FETCH → VERTEX SHADER → TESS. CONTROL SHADER → TESSELLATION → TESS. EVALUATION SHADER → GEOMETRY SHADER → RASTER → PIXEL SHADER

Pipelined memory, keeping interstage data on chip

TASK/MESH PIPELINE

TASK SHADER → MESH GENERATION → MESH SHADER → RASTER → PIXEL SHADER

Optional Expansion

Pipelined memory

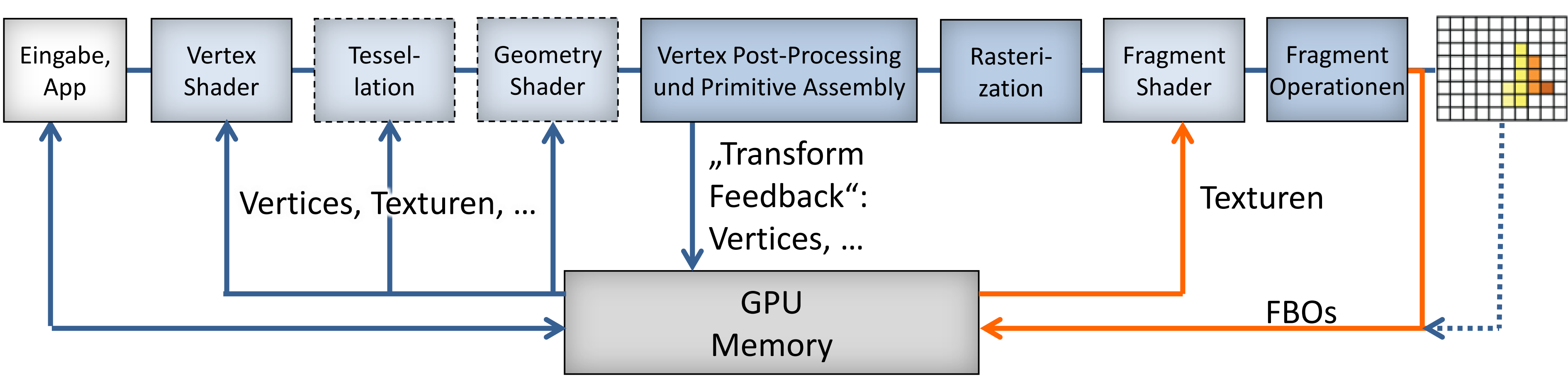
- Vorteil z.B.: Sichtbarkeit von Meshlets kann bestimmt werden und nur wenn notwendig werden Vertex Attribute aus dem Speicher gelesen
- **Task Shader** gibt Mesh Shader Workgroups aus
- **Mesh shader** generiert direkt Primitive

Beispiel: Mesh Shader

```
layout(local_size_x = 1) in;
layout(triangles, max_vertices = 3, max_primitives = 1) out;

layout (location = 0) out PerVertexData {
  vec4 color;
} v_out[];  // [max_vertices]

const vec3 vertices[3] = { … }, colors[3] = { … };

void main() {
  gl_MeshVerticesEXT[0].gl_Position = vec4( vertices[0], 1.0 );
  gl_MeshVerticesEXT[1].gl_Position = vec4( vertices[1], 1.0 );
  gl_MeshVerticesEXT[2].gl_Position = vec4( vertices[2], 1.0 );

  v_out[0].color = vec4(colors[0], 1.0);          // Vertex-Farben
  v_out[1].color = …; v_out[2].color = …;

  gl_PrimitiveIndicesEXT[0] = 0;                  // Vertex-Indizes
  gl_PrimitiveIndicesEXT[1] = 1;  gl_PrimitiveIndicesEXT[2] = 2;

  gl_PrimitiveCountEXT = 1;                       // Anzahl Dreiecke
}
```

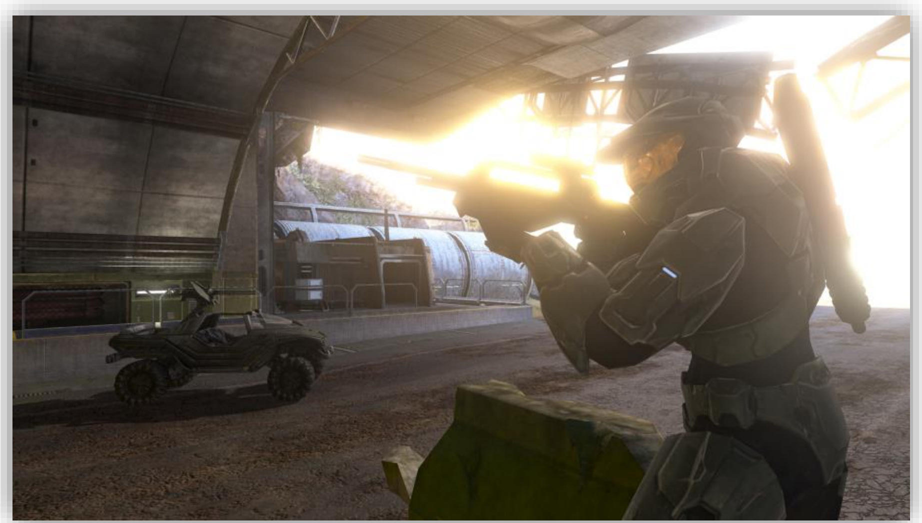
Ausblick: Weitere OpenGL Puffer

- Accumulation Buffer: Summe über Bilder im Framebuffer
- Pixel Buffer Objects für effizienten Textur-Upload, asynchroner Transfer
  - Funktionsweise ähnlich Array Buffer, aber für Pixeldaten/Bilder
- Uniform Buffer Objects fassen Uniforms für GLSL-Programme zusammen
  - → weniger OpenGL-Aufrufe
- Framebuffer Objects (FBOs)
  - erlaubt der Applikation „eigene“ Frame Buffer anzulegen



Ausblick: Framebuffer Objects (FBOs)

- bisher: beim Rendering werden Fragmente direkt in den Farb-, Tiefen-, Stencil-Buffer des *OpenGL-Framebuffers* geschrieben
- FBOs erlauben es die Ausgabe in eigene „Off-Screen Buffer" umzuleiten
  - Color Buffers: eine oder mehrere Texturen um Farbwerte zu schreiben
  - Renderbuffers: Tiefen- und Stencil-Buffer
- FBOs notwendig für moderne Rendering-Techniken der interaktiven CG, z.B. Blooming, oder Nachbildung von OpenGL-Accumulation Buffer

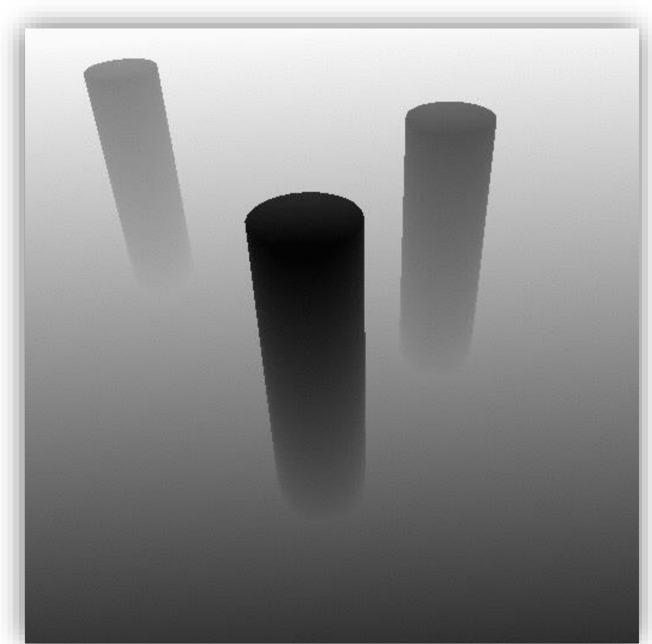


Links: https://www.khronos.org/opengl/wiki/Framebuffer_Object
http://www.songho.ca/opengl/gl_fbo.html

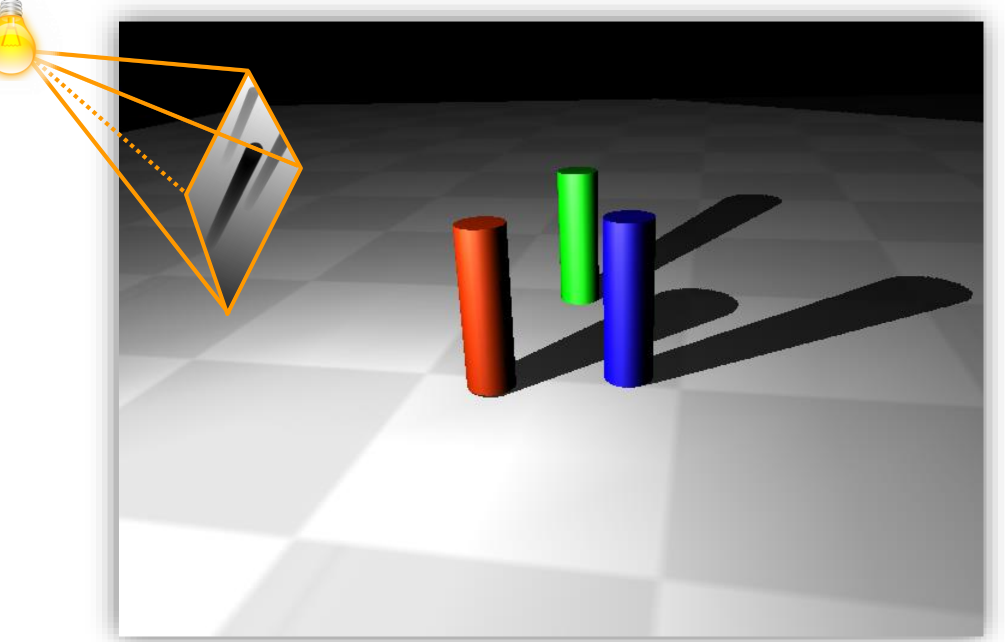
Shadow Mapping (Williams, SIGGRAPH '78)

Shadow Mapping Grundidee

- eine Textur („Shadow Map“, in einem FBO) wird in jedem Frame erzeugt und speichert – für viele Augstrahlen/Richtungen – wie weit die nahsten Oberflächen von der Lichtquelle entfernt sind
- hier am Beispiel von Spotlights, d.h. wir ersetzen die Lichtquelle gedanklich durch eine perspektivische Kamera und rasterisieren und speichern den Tiefenpuffer des Bildes (links)



Shadow Map: Szene aus der Sicht der Lichtquelle

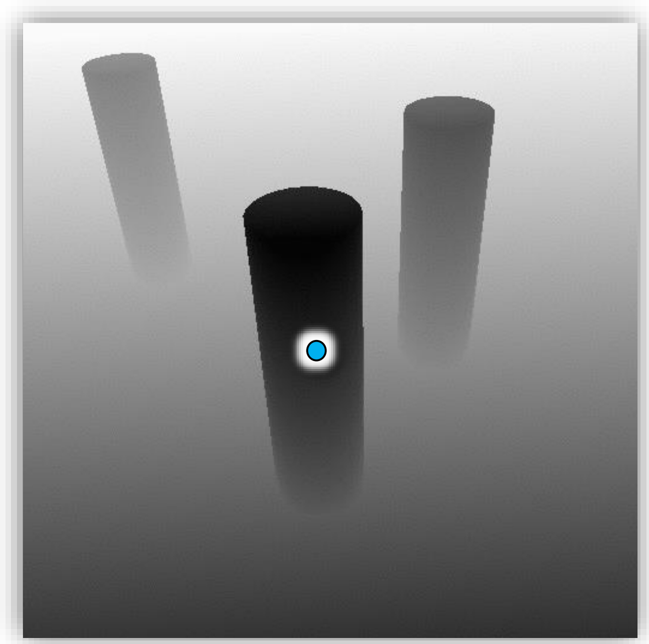


Szene aus der Sicht der Kamera

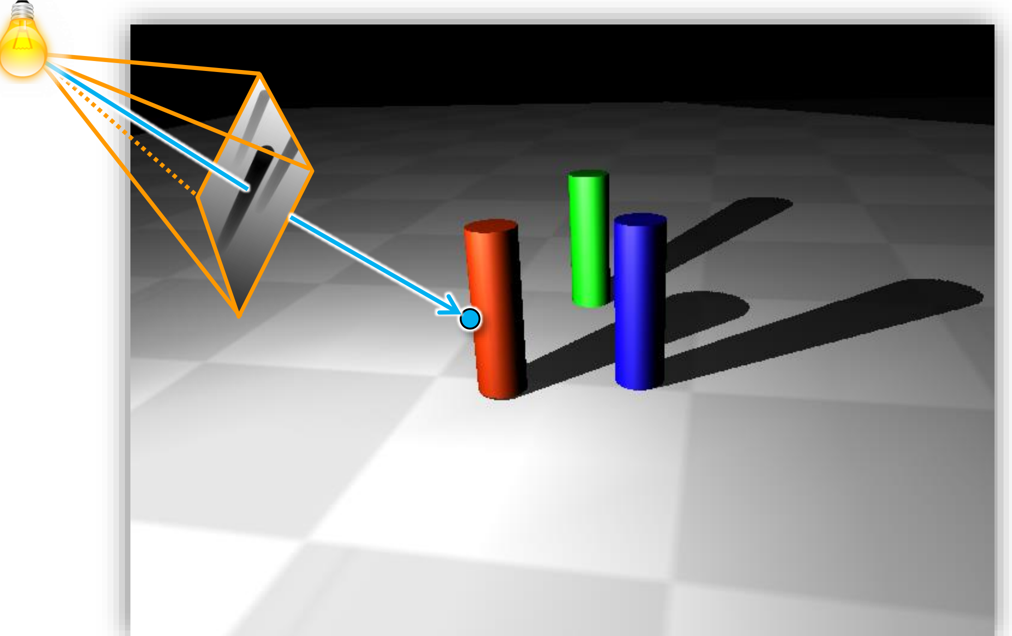
Shadow Mapping (Williams, SIGGRAPH '78)

Shadow Mapping Grundidee

- Shadow Map 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 (Transformation in Lichtkoordinatensysteme)



Shadow Map: Szene aus der Sicht der Lichtquelle

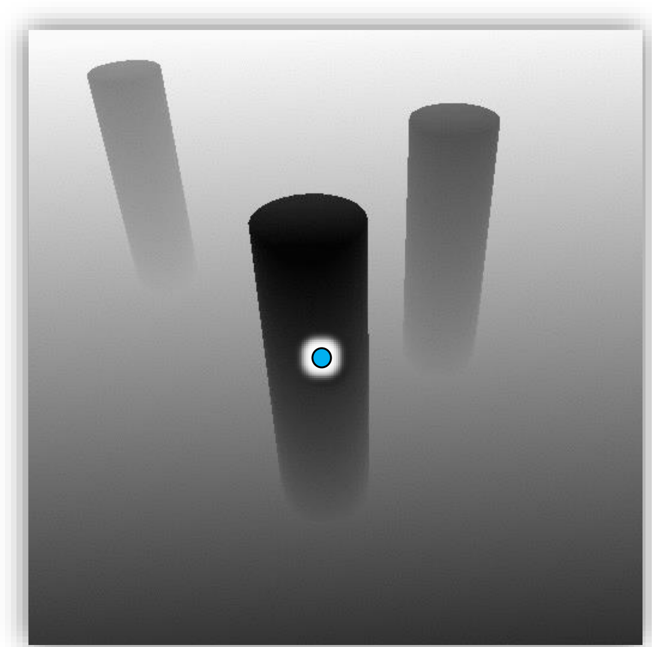


Szene aus der Sicht der Kamera

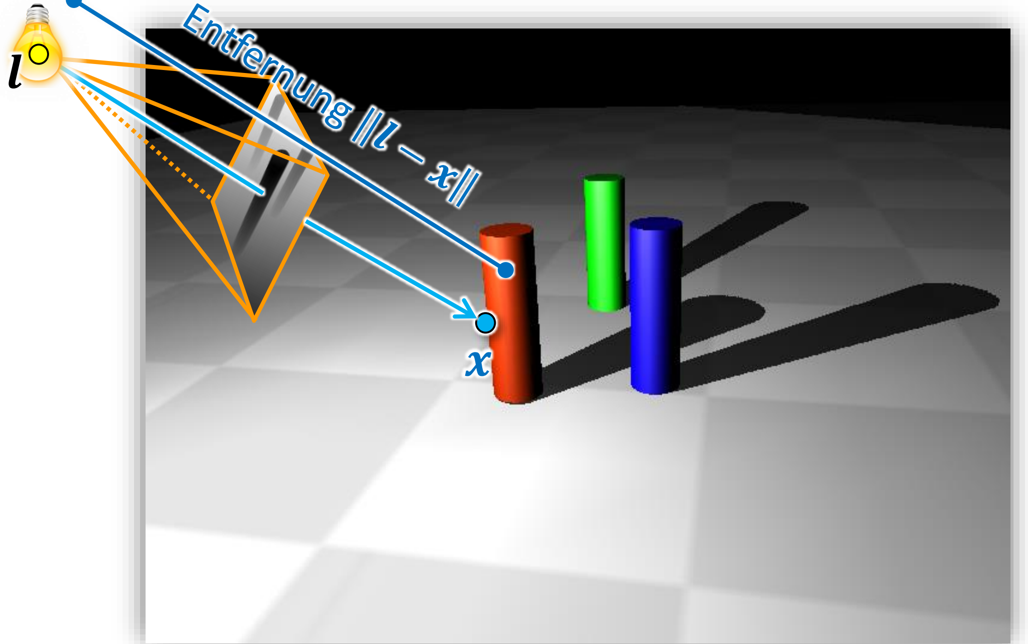
Shadow Mapping (Williams, SIGGRAPH '78)

Shadow Mapping Grundidee

- der blaue Punkt befindet sich nicht im Schatten:
  die in der Shadow Map gespeicherte Entfernung
  ist *gleich* der tatsächlichen Entfernung zur Lichtquelle



Shadow Map: Szene aus der Sicht der Lichtquelle

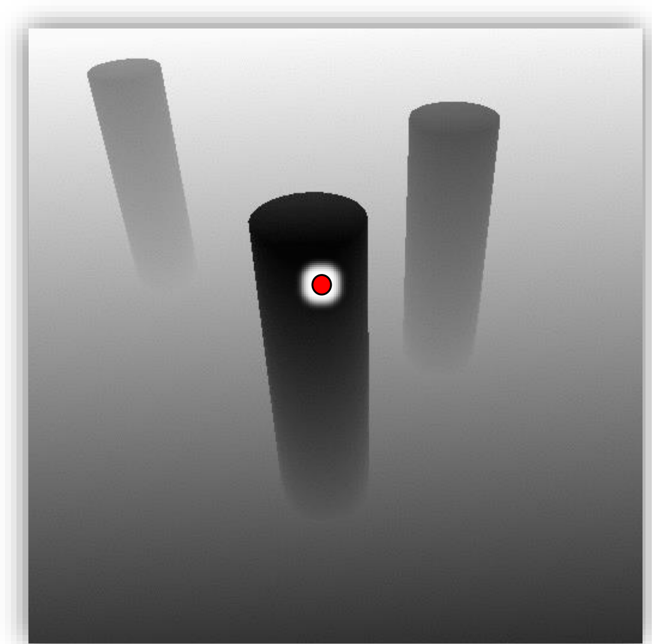


Szene aus der Sicht der Kamera

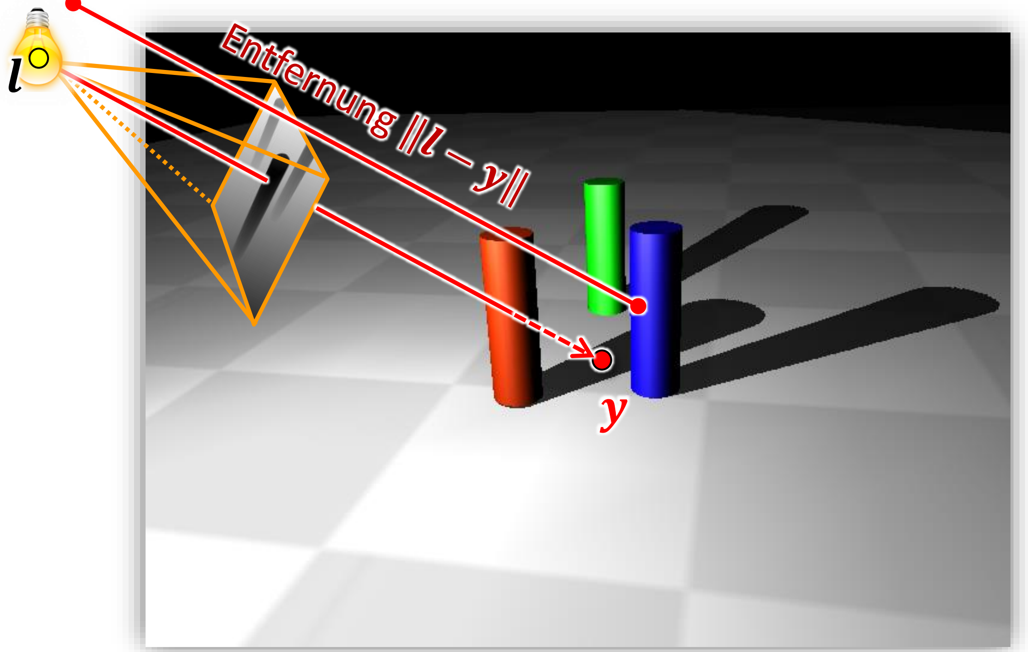
Shadow Mapping (Williams, SIGGRAPH '78)

Shadow Mapping Grundidee

- 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

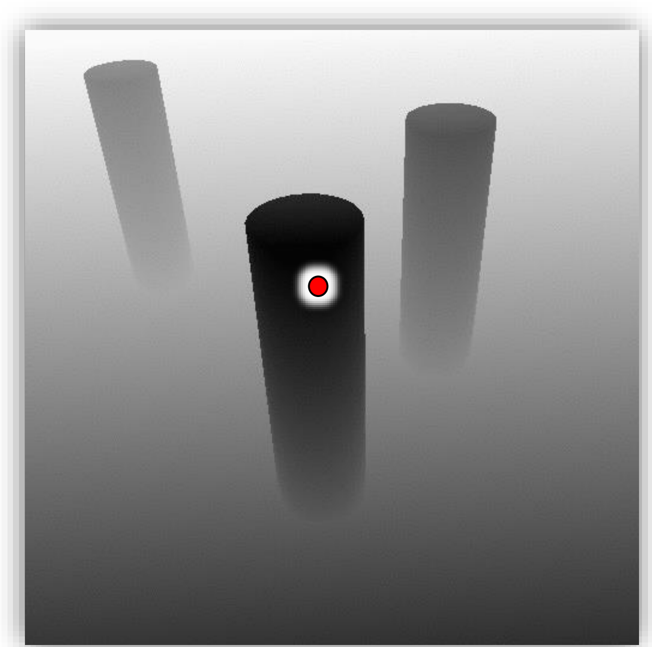


Szene aus der Sicht der Kamera

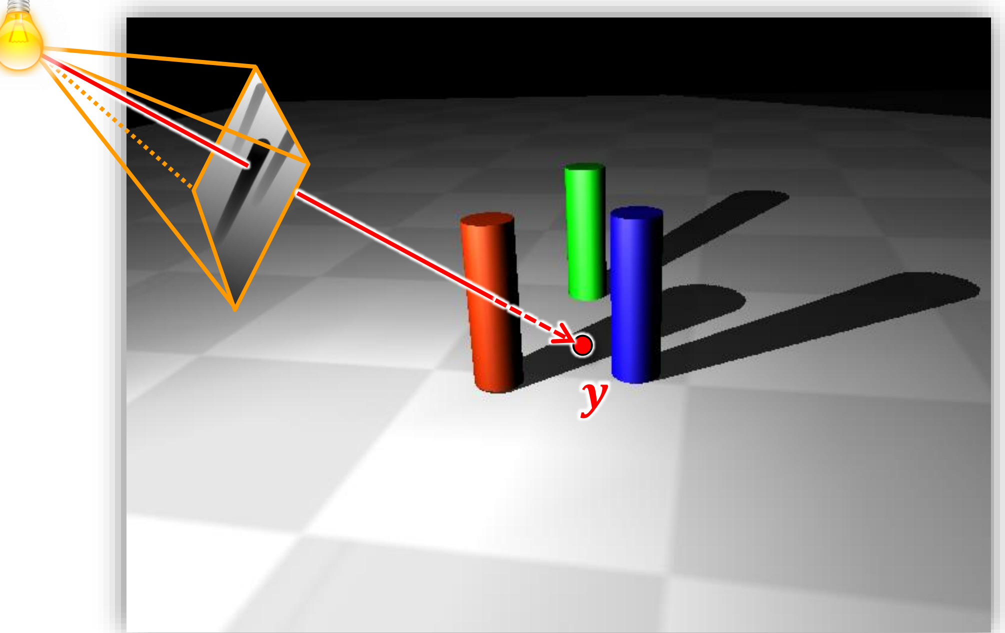
Shadow Mapping (Williams, SIGGRAPH '78)

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



Shadow Map: Szene aus der Sicht der Lichtquelle



Szene aus der Sicht der Kamera

Teures Shading? Viele Lichtquellen? Post-Processing?

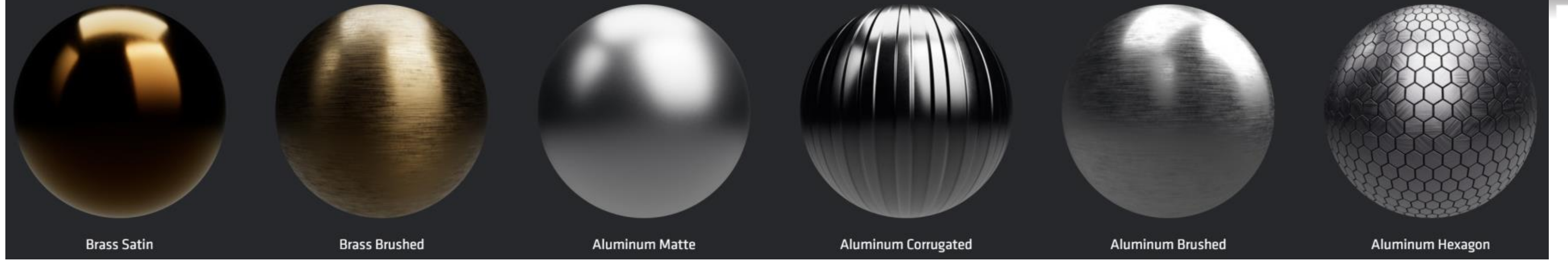


Bild: AMD GPUOpen MaterialX Library https://matlib.gpuopen.com

Bild: https://www.linkedin.com/pulse/comprehensive-explanation-deferred-rendering-kylin-jiangwei/

Bild: Lumen: Real-time Global Illumination in Unreal Engine 5, Wright et al.

Ausblick: Deferred Shading

- Geometriephase: Rendering der Szenengeometrie

Eingabe, App
Vertex Shader
Tessel-lation
Geometry Shader
Vertex Post-Processing und Primitive Assembly
Rasteri-zation
Fragment Shader
Fragment Operationen

Vertices, Texturen, …

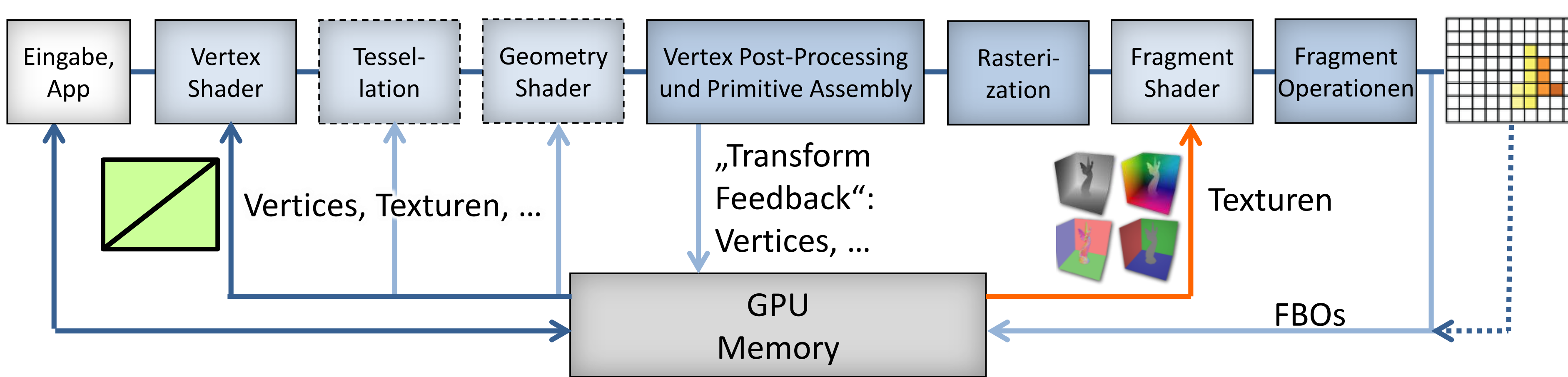
„Transform Feedback“: Vertices, …

Texturen

GPU Memory

FBOs

- Beleuchtungsphase: Rendering eines Rechtecks (u.U. mehrfach)



Vulkan

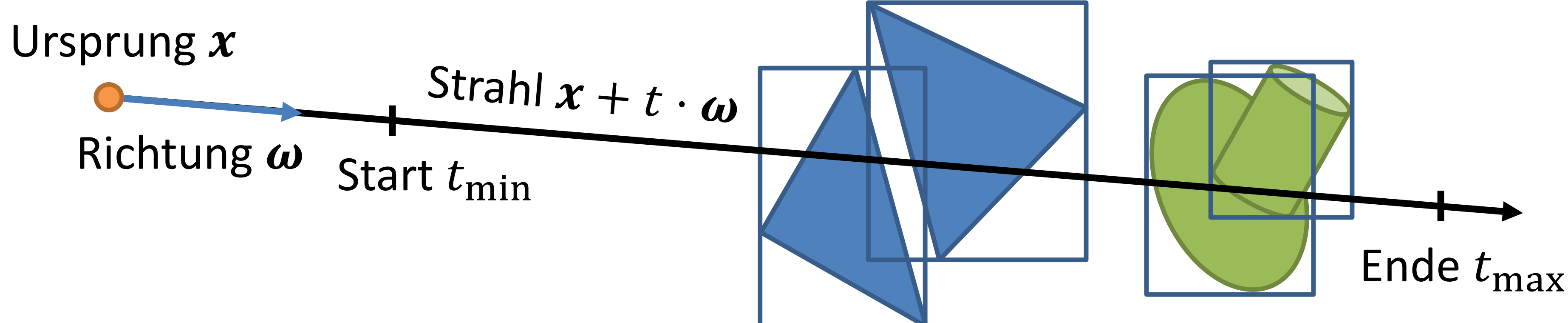
- **„lower-level OpenGL“**: näher an der Hardware, mehr Kontrolle über …
  - asynchrone Verarbeitung (z.B. Command-Buffers über mehrere Threads, Graphics/Compute Pipelines, ersetzt auch OpenCL)
  - Ressourcen: Speicherverwaltung ist Aufgabe der Applikation
  - **signifikant höhere Komplexität bei Setup und Rendering**
- Einsatz sinnvoll, wenn CPU-Last (insb. im OpenGL-Treiber) die Performanz begrenzt oder Multi-Threading diese steigert, sowie bei „Stottern“ und Problemen mit Reaktionsschnelligkeit
- i.W. keine zusätzlichen GPU-Features (Ausnahme: Raytracing), Koexistenz mit OpenGL
- **Standard Portable Intermediate Representation-V (SPIR-V)**
  - Zwischensprache für Shader (Compiler-Ziel ausgehend von unterschiedlichen Sprachen), auch verfügbar in OpenGL 4.6
  - kein High-Level Language Compiler im Treiber mehr nötig
  - Referenz-Compiler GLSL → SPIR-V
    https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/

Raytracing API

Details: https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/

Überblick (verfügbar in Vulkan, Direct3D 12 und Metal)

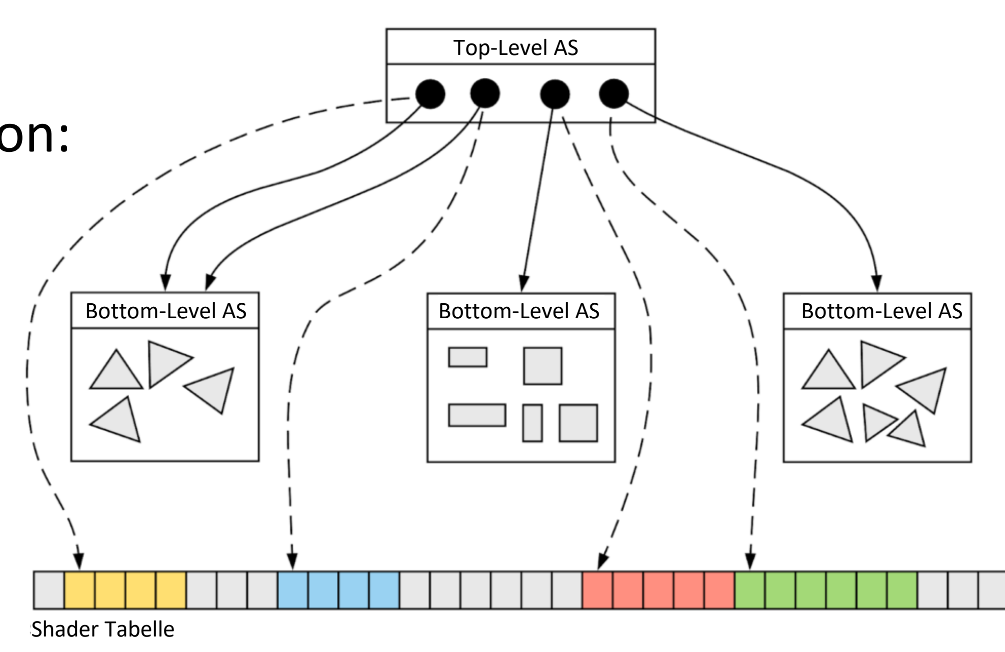
- Raytracing-Kerne der GPUs erinnern an Textureinheiten:
  - man bindet Beschleunigungsstrukturen statt Texturen und greift mit Strahlen darauf zu (statt mit Texturkoordinaten)
  - wahlweise von speziellen Raytracing-Shadern aus (EXT_ray_tracing) oder von beliebigen Shadern (EXT_ray_query)
- beschleunigt Durchführung von Schnitttests
  - Primitive können Dreiecksmeshes sein (wie gewohnt mit Vertexbuffer und optional Indexbuffer)
  - oder beliebige andere Primitive: definiere Bounding Box und bestimme Schnitt im sog. Intersection Shader



Raytracing API

Beschleunigungsstrukturen (Acceleration Structure, AS)

- API spezifiziert nicht den Typ Hüllkörperhierarchie/kD Baum
- zweistufige Hierarchie von Beschleunigungsstrukturen:
  - Bottom-Level AS enthält Dreiecke oder beliebige Primitive
  - Top-Level AS enthält Transformationen, Zeiger auf Bottom-Level AS und Shader (Intersection, Any Hit, Closest Hit)
  - Daten liegen im Speicher der GPU, Instancing möglich
- können inkrementell aktualisiert werden
- Hinweis-Flags bei der Konstruktion:
  - Intersektionstests oder Konstruktion schnell
  - geringerer Speicherbedarf



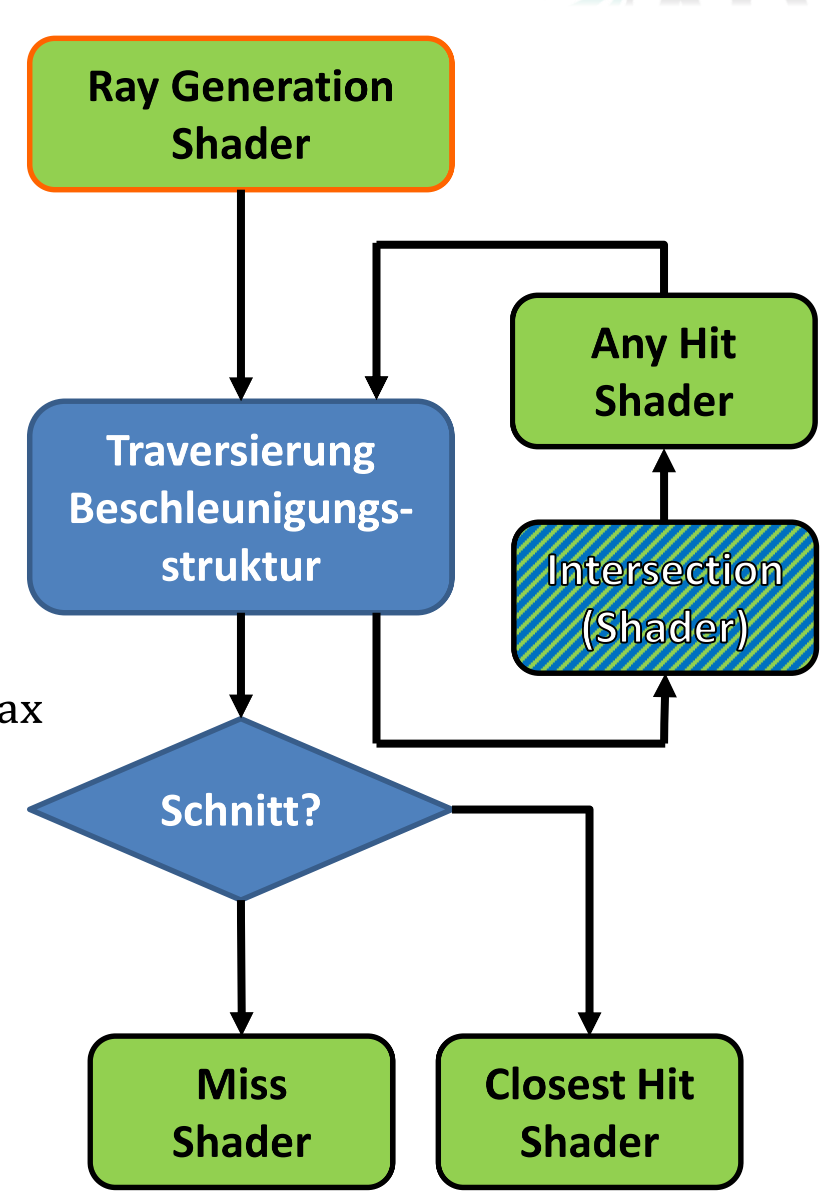
Raytracing API (EXT_ray_tracing)

Ray Generation Shader

- wird von CPU aus aufgerufen
- viele Threads in einem Gitter, z.B. einer pro Pixel (vgl. Compute Shader)
- jeder Thread kann beliebig oft `traceRayEXT()` aufrufen

Parameter von `traceRayEXT()`

- Strahl, d.h. Ursprung, Richtung, $t_{\mathrm{min}}, t_{\mathrm{max}}$
- Top-Level Beschleunigungsstruktur
- Miss Shader
- diverse Flags
- Payload-Struktur als Ein- und Ausgabe (z.B. finale Pixelfarbe, die vom Miss oder Closest Hit Shader gesetzt wird)



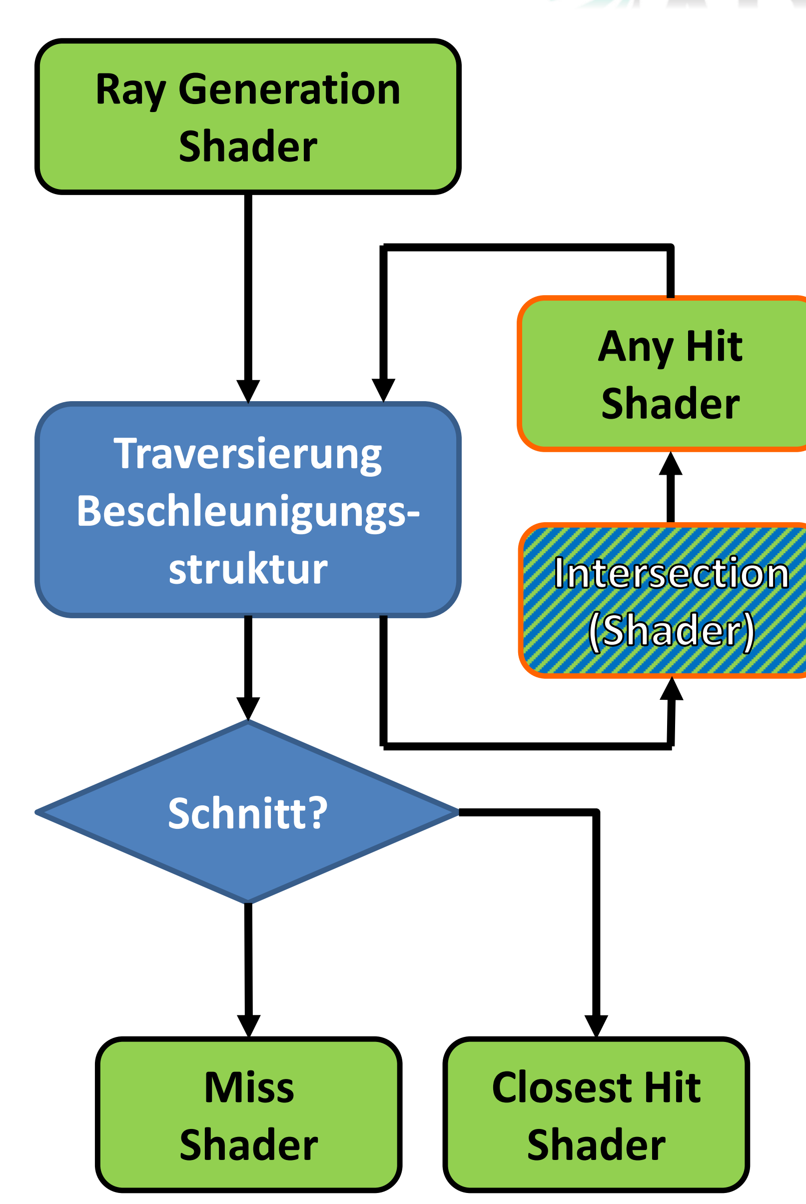
Raytracing API (EXT_ray_tracing)

Intersection Shader

- unnötig für Dreiecke
- wird für jeden Schnitt mit einer Boundingbox aufgerufen
- bestimmt ob ein Schnitt vorhanden ist
- aktualisiert ggf. $t_{\max}$

Any Hit Shader

- wird für jeden Schnitt aufgerufen
- kann Schnitt ignorieren (z.B. wegen vollkommen transparenter Textur)
- kann Suche nach weiteren Schnitten sofort beenden (z.B. für Schatten)
- kann Payload verändern



Raytracing API (EXT_ray_tracing)

Miss Shader

- aufgerufen, wenn es keinen Schnitt gab (z.B. der Strahl verlässt die Szene)
- dann z.B. Lookup in Environment Map
- kann rekursiv `traceRayEXT()` aufrufen
- kann Payload verändern

Closest Hit Shader

- wird für den Schnitt mit minimalem t aufgerufen
- typ. Verwendung: Shading auswerten
- kann rekursiv `traceRayEXT()` aufrufen (z.B. für Path Tracing)
- kann Payload verändern