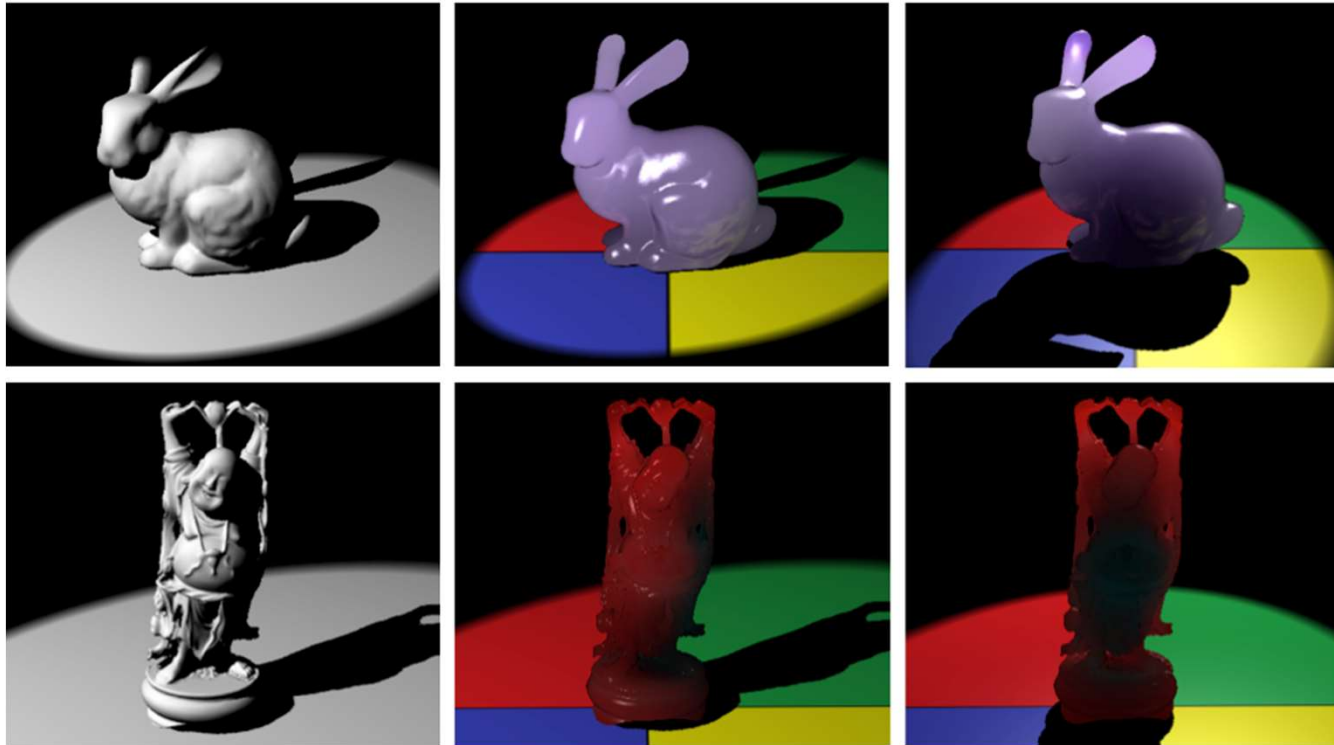
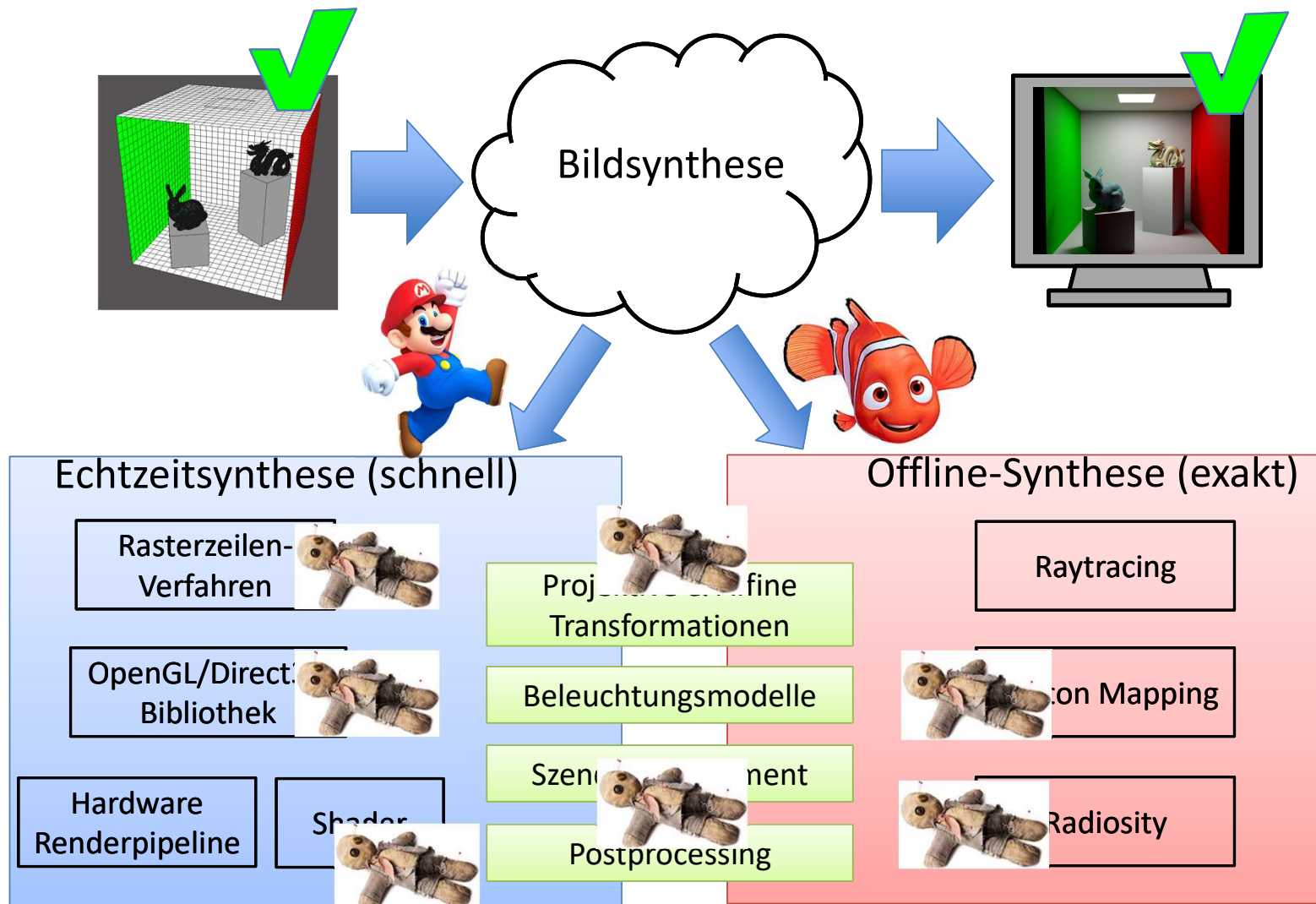


Computergrafik



Vorlesung

Philipp Lensing



Themen heute

- Einführung OpenGL
- Vertex Buffer Objects (GPU-Buffer)
- Vertex-Attribute
- Index Buffer Objects (GPU-Buffer)
- Normalisierter Bildraum
- Einfache GLSL-Shader
- Einfache Texturierung
- Praktikumsklassen

OpenGL/GLFW/GLUT

OpenGL: Open Graphics Library

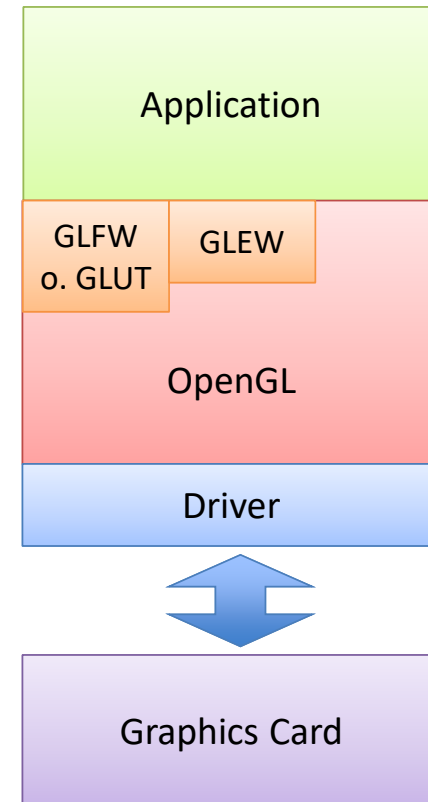
- Software-**Interface** (API) zur hardwarebeschleunigten Darstellung von 2D- und 3D-Grafikdaten.
- Der Schnittstellenstandard wird von Khronos Group entwickelt.
- Plattformunabhängig
- Low-Level-C-Schnittstelle
- Herstellerspezifische Erweiterungen als *Extensions* möglich
- Client/Server-Architektur

GLFW oder GLUT (OpenGL Utility Toolkit)

- Einfache Frameworks für plattformunabhängige OpenGL-Fensteranwendungen
- Kapselt den Simulation-Loop
- Unterstützt verschiedene Eingabesysteme
- U. v. m

GLEW: OpenGL Extension Wrangler

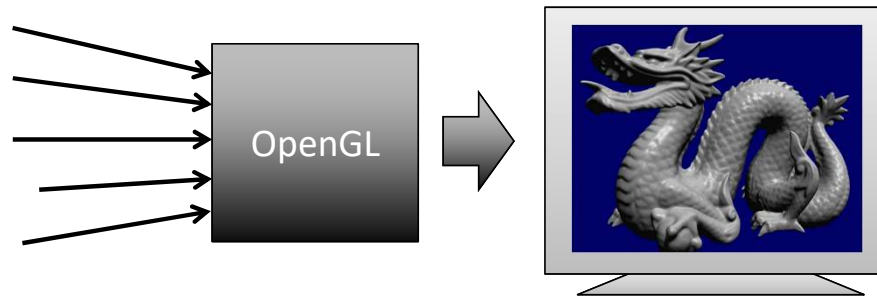
- Hilfsbibliothek für das OpenGL-Extension-Management



OpenGL

Gegenwärtig ist OpenGL für uns eine Blackbox: Wir übergeben

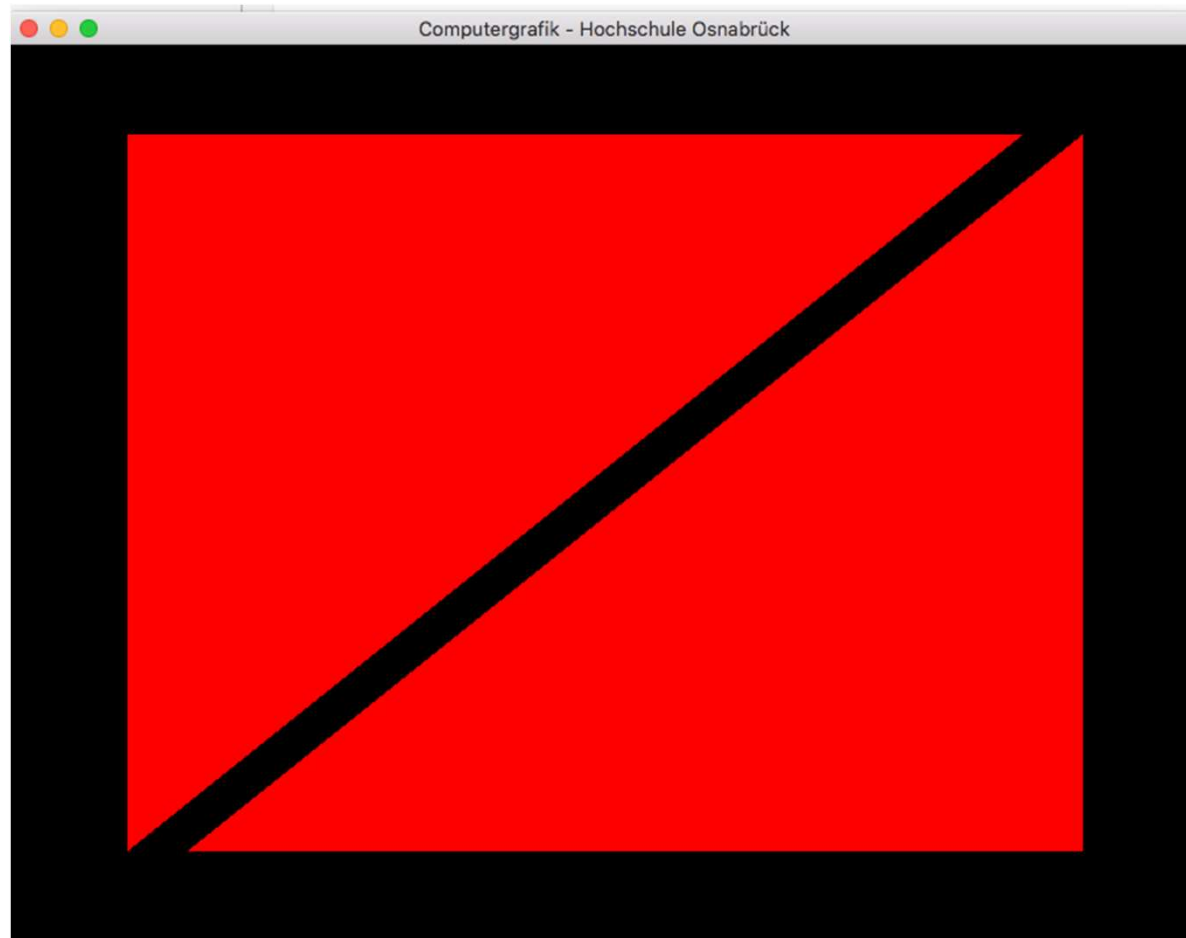
- Modelldaten,
- Texturen,
- Shader,
- Transformationen,
- etc.



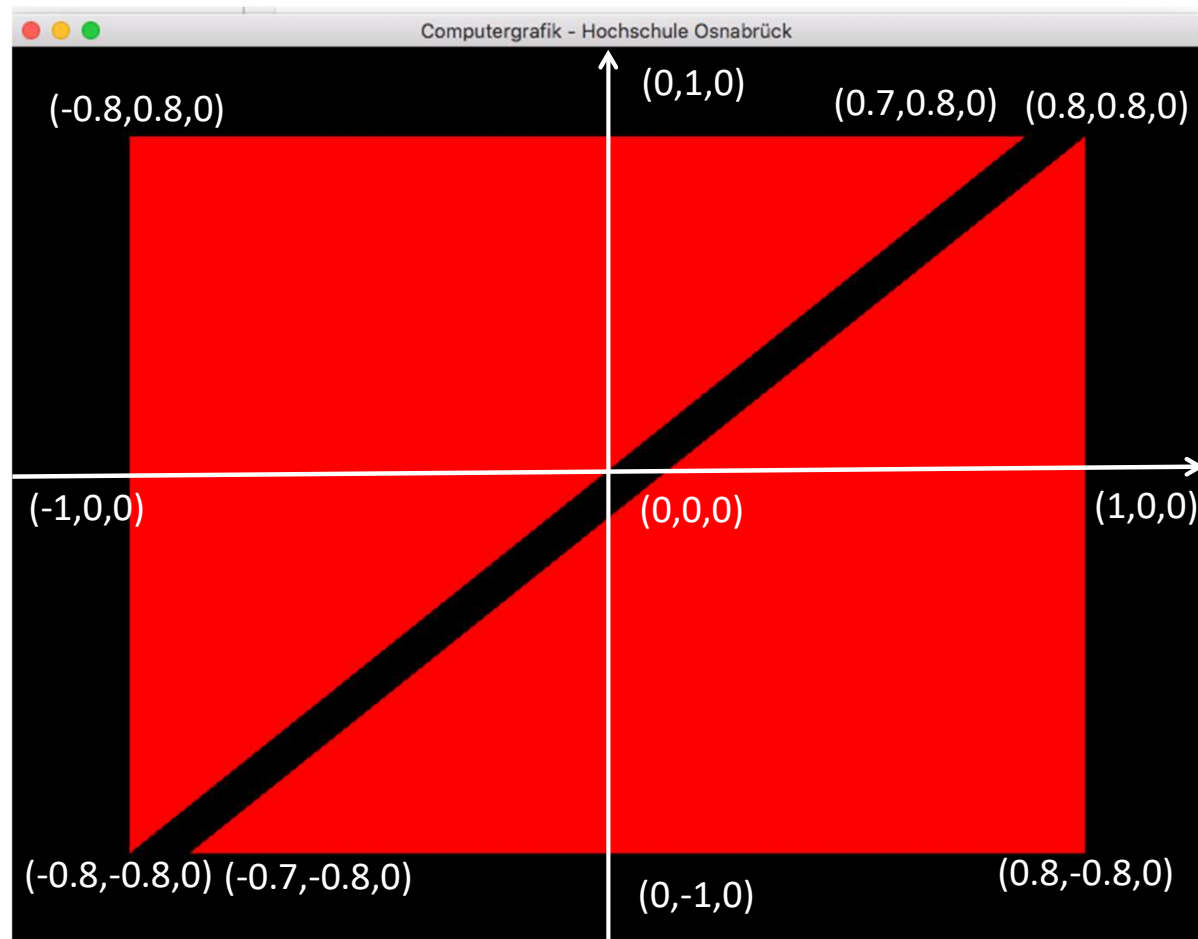
und erhalten das fertige Bild.

Das Bildsynthese-Verfahren, welches von OpenGL verwendet wird, ist uns noch unbekannt. Die Bildsynthese wird hier in mehrere Schritte zerlegt und wird als **Hardware-Rendering-Pipeline** implementiert (siehe letzte Vorlesung für grobe Übersicht).

OpenGL Beispiel 1: Zwei rote Dreiecke (Application1.cpp)



OpenGL Beispiel 1: Normalisierter Bildraum



Application-Klasse

Schnittstelle der Application-Klasse (Praktikum):

```
class Application
{
public:
    Application(GLFWwindow* Window); // Start d. Anwendung
    void start(); // Start d. Anwendung
    void update(); // pro Frame
    void draw(); // pro Frame
    void end(); // Ende d. Anwendung
    // ...
protected:
    GLFWwindow* pWindow;
};
```


OpenGL Beispiel 1 (Vertex Buffer Objects)

```
void Application::start()
{
    struct Vertex{ Vector Pos; }; Vertex-Struktur (Vertex hat hier nur Positionsattrib.)
    Vertex TwoTriBuffer[6] {
        Vertex{ Vector(-0.8, -0.8, 0) },
        Vertex{ Vector(-0.8, 0.8, 0) },
        Vertex{ Vector(0.7, 0.8, 0) },
        Vertex{ Vector(0.8, 0.8, 0) },
        Vertex{ Vector(0.8, -0.8, 0) },
        Vertex{ Vector(-0.7, -0.8, 0) }
    };

    glGenBuffers (1, &VB0); Erzeugt einen Identifier für ein Vertex Buffer Object (VBO).
    glBindBuffer (GL_ARRAY_BUFFER, VB0); Erzeugt das VBO und aktiviert es.
    glBufferData (GL_ARRAY_BUFFER, sizeof(TwoTriBuffer),
        TwoTriBuffer, GL_STATIC_DRAW); Kopiert die Daten ins VBO.

    glGenVertexArrays(1, &VA0); Erzeugt einen Identifier für ein Vertex Array Object (VAO)
    glBindVertexArray(VA0); Erzeugt das VAO und aktiviert es.
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
        sizeof(Vertex), BUFFER_OFFSET(0)); Beschreibt die Struktur
    der Vertices im VBO.
    glBindVertexArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0); Deaktiviert das VBO & VAO.

    ShaderProgram = createShader(); Erzeugt ein Shader-Programm
    (siehe nächste Folie).
}
```

OpenGL Beispiel 1 (Vertex & Fragment Shader)

```
GLuint Application::createShader()
{
    const char * VScode =
        "#version 400\n"
        "in vec4 VertexPos;"
        "void main() {"
        "    gl_Position = VertexPos;"
        "}";
    const char * FScode =
        "#version 400\n"
        "out vec4 FragColor;"
        "void main() {"
        "    FragColor = vec4(1,0,0,0);"
        "}";

    GLuint VS = glCreateShader(GL_VERTEX_SHADER);
    GLuint FS = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(VS, 1, &VScode, NULL);
    glShaderSource(FS, 1, &FScode, NULL);
    glCompileShader(VS);
    glCompileShader(FS);

    GLuint Shader = glCreateProgram();
    glAttachShader(Shader, VS);
    glAttachShader(Shader, FS);
    glLinkProgram(Shader);
    glDeleteShader(VS);
    glDeleteShader(FS);
    return Shader;
}
```

Vertex-Shader-Code: Der Shader übergibt nur die Position (es werden keine Transformationen durchgeführt).

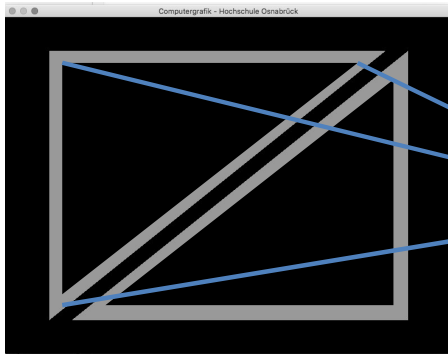
Fragment-Shader-Code: Der Shader färbt einen Ausgabe-Pixel rot ein.

Erzeugung der Shader-Objekte.

Der Shader-Code wird übergeben und anschließend kompiliert.

Die kompilierten Shader werden zu einem ShaderProgram gelinkt. Anschließend werden die einzelnen Shader gelöscht.

OpenGL Beispiel 1 (Vertex & Fragment Shader)



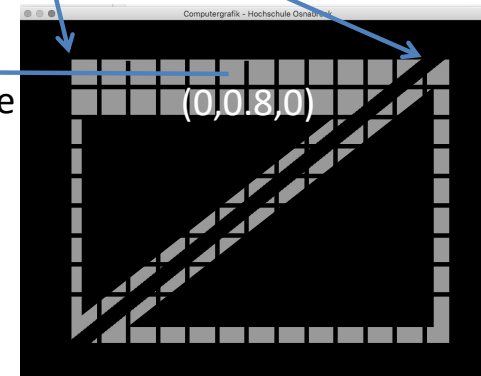
```
const char * VScode =  
"#version 400\\n"  
"in vec4 VertexPos;"  
"void main() {"  
"    gl_Position = VertexPos;"  
"}";
```

(-0.8,0.8,0)

(0.8,0.8,0)

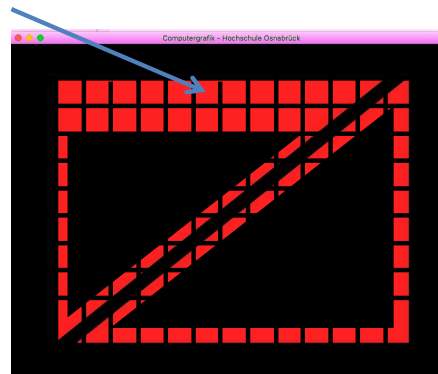
```
const char * FScode =  
"#version 400\\n"  
"out vec4 FragColor;"  
"void main() {"  
"    FragColor = vec4(1,0,0,0);"  
"}";
```

Interpolierte Attribute



Rasterung

Farbe des Fragments
festlegen.



OpenGL Beispiel 1 (Zeichnen von VBOs)

```
void Application::draw()
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindVertexArray(VAO);
    glUseProgram(ShaderProgram);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    GLenum Error = glGetError();
    assert(Error==0);
}
```

Der Farb- & Tiefen-Backbuffer wird mit
glClearColor(...) zurückgesetzt.

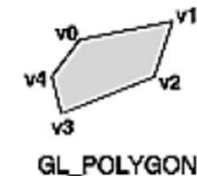
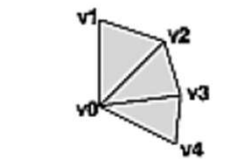
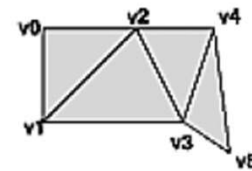
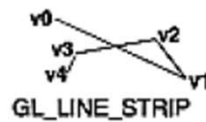
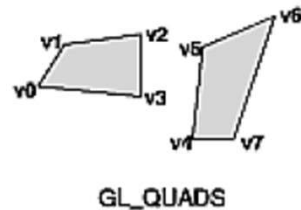
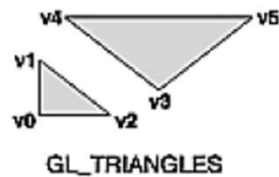
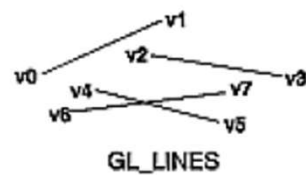
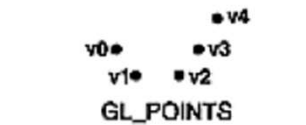
Vertex Array Object aktivieren

Shader-Programm aktivieren

Primitiven Zeichnen (hier Dreiecke)

Optional: OpenGL Fehler
abfragen.

OpenGL-Zeichenprimitiva



`glDrawArrays(<Primitivum-Typ>, <Offset>, <Anzahl>);`

`glDrawElements(<Primitivum-Typ>, <Anzahl>, <Index-Typ>, <OffsetPtr>);`

OpenGL/GLUT/GLFW-Syntax

- OpenGL-Funktionen beginnen immer mit „gl“.
 - Bsp: `glLoadIdentity()`
- GLUT-Funktionen beginnen mit „glut“.
- GLFW-Funktionen beginnen mit „glfw“
- OpenGL-Konstanten beginnen immer mit „GL_“
 - Bsp: `GL_PROJECTION`
- GLUT-Konstanten beginnen mit „GLUT_“
- Manche OpenGL-Funktionen besitzen unterschiedliche Suffixe um unterschiedliche Parametertypen zu erlauben, z. B.:
 - `glVertex3f(...)` → 3 Floating-Point Parameter
 - `glVertex4f(...)` → 4 Floating-Point Parameter
 - `glVertex3i(...)` → 3 Integer Parameter
 - etc.

OpenGL

- OpenGL ist ein (komplexer) Zustandsautomat (State-Machine).
- Zustände die über die API gesetzt wurden, bleiben solange erhalten, bis der Zustand überschrieben wird.

Beispiel:

```
glClearColor(0,0,0,0);  
glUseProgram( .. );  
glBindBuffer( .. );  
glEnable( .. );  
Etc.
```

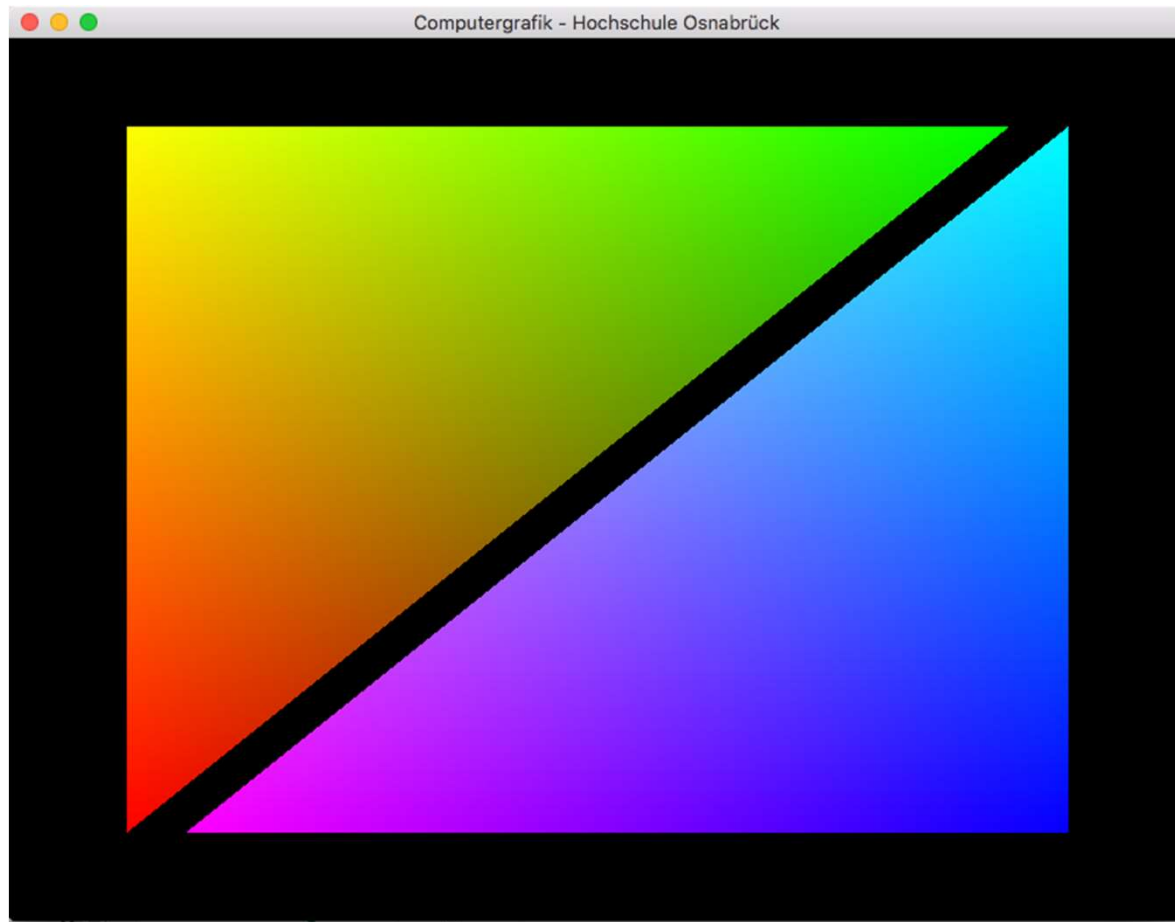
Setzt die Füllfarbe für den glClear(..)-Befehl fest. Solange glClearColor(..) nicht erneut gesetzt wird, bleibt der Zustand für die Laufzeit der Anwendung erhalten.

Das Shader-Programm bleibt so lange aktiv, bis ein neues Shader-Programm gesetzt wird (oder 0).

Gesetzte Buffer Objects bleiben aktiv.

Sämtliche Zustände die mit glEnable(..) gesetzt werden bleiben aktiv.

OpenGL Beispiel 2: Zwei bunte Dreiecke (Application2.cpp)



OpenGL Beispiel 2 (Vertex Buffer Objects)

```
void Application2::start()
{
    struct Vertex{ Vector Pos; Color C; };

    Vertex TwoTriBuffer[6] {
        Vertex{ Vector(-0.8, -0.8, 0), Color(1,0,0) },
        Vertex{ Vector(-0.8, 0.8, 0), Color(1,1,0) },
        Vertex{ Vector(0.7, 0.8, 0), Color(0,1,0) },
        Vertex{ Vector(0.8, 0.8,0), Color(0,1,1) },
        Vertex{ Vector(0.8,-0.8,0), Color(0,0,1) },
        Vertex{ Vector(-0.7,-0.8,0), Color(1,0,1) }
    };

    // glGenBuffers, glBindBuffers, etc.
    ...

    glGenVertexArrays(1, &VA0);
    glBindVertexArray(VA0);
    glEnableVertexAttribArray(0); Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(0));
    glEnableVertexAttribArray(1); Farbe
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(12));
    ...
}
```

Die Vertex-Struktur wird um einen Farbwert erweitert.

Der ergänzte Farbwert muss in der Bufferstruktur berücksichtigt werden, damit das Shader-Programm die Daten korrekt interpretieren kann.

OpenGL Beispiel 2 (Vertex & Fragment Shader)

```
GLuint Application2::createShader()  
{
```

```
    const char * VScode =  
        "#version 400\n"  
        "in vec4 VertexPos;"  
        "in vec3 vColor;"  
        "out vec3 fColor;"
```

Die Farbe wird als zusätzliches Ein- und
Ausgabe-Attribut vom Vertex Shader

```
    "void main() {"  
        "    gl_Position = VertexPos;"  
        "    fColor = vColor;"  
    "}";
```

Der Vertex Shader gibt die Farbe
lediglich weiter.

```
    const char * FScode =  
        "#version 400\n"  
        "in vec3 fColor;"  
        "out vec4 FragColor;"
```

Die Farbe ist Eingang für den
Fragment Shader.

```
    "void main() {"  
        "    FragColor = vec4(fColor,0);"  
    "}";
```

Die Farbe wird vom Fragment Shader
benutzt, um die Pixel einzufärben.

```
    ...  
    // create, compile & link shader  
    return Shader;
```

```
}
```

OpenGL Beispiel 2 (Vertex & Fragment Shader)

```
glGenVertexArrays(1, &VA0);  
glBindVertexArray(VA0);  
glEnableVertexAttribArray(0);  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(0));  
glEnableVertexAttribArray(1);  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(12));
```

Zuordnung zum Shader

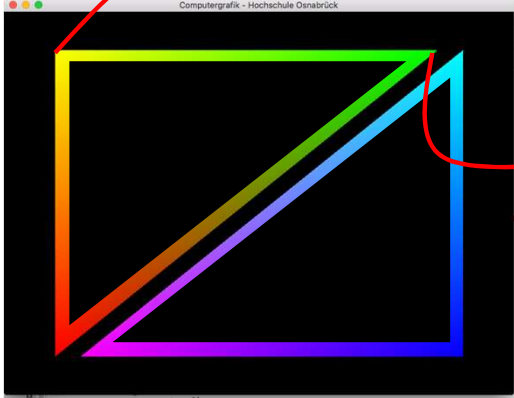
```
const char * VScode =  
"#version 400\\n"  
"in vec4 VertexPos;"  
"in vec3 vColor;"  
"out vec3 fColor;"  
"void main() {"  
"    gl_Position = VertexPos;"  
"    fColor = vColor;"  
"}";
```

Durch die Reihenfolge der Variablen-Deklarationen wird die „Location“ festgelegt. Die *Location* kann aber auch explizit im Shader-Programm festgelegt werden (empfohlen):

```
layout(location = 0) in vec4 VertexPos;  
layout(location = 1) in vec4 vColor;
```

OpenGL Beispiel 2 (Vertex & Fragment Shader)

$F(1,1,0)$

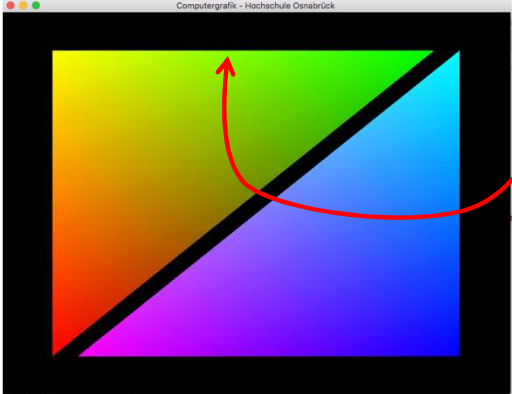


$F(0,1,0)$

```
const char * VScore =  
"#version 400\\n"  
"in vec4 VertexPos;"  
"in vec3 vColor;"  
"out vec3 fColor;"  
"void main() {"  
"    gl_Position = VertexPos;"  
"    fColor = vColor;"  
"}";
```

Interpolierte Attribute
(Rasterung)

$F(0.5,1,0)$



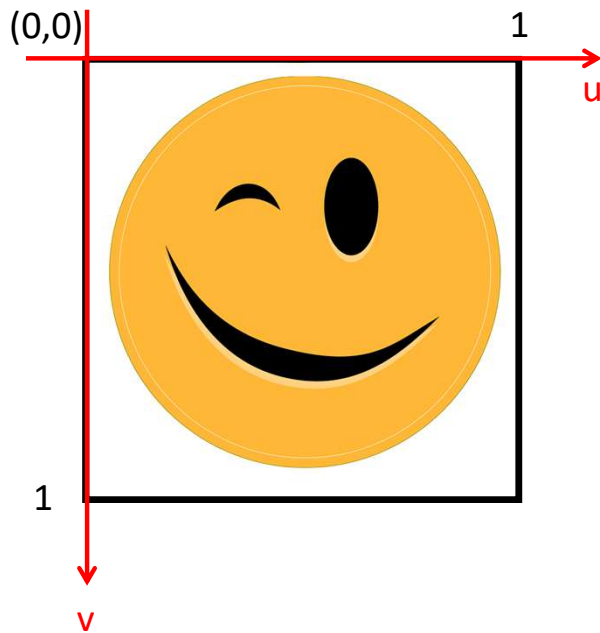
```
const char * FScore =  
"#version 400\\n"  
"in vec3 fColor;"  
"out vec4 FragColor;"  
"void main() {"  
"    FragColor = vec4(fColor,0);"  
"}";
```

OpenGL Beispiel 3: Zwei bunte, texturierte Dreiecke (Application3.cpp)

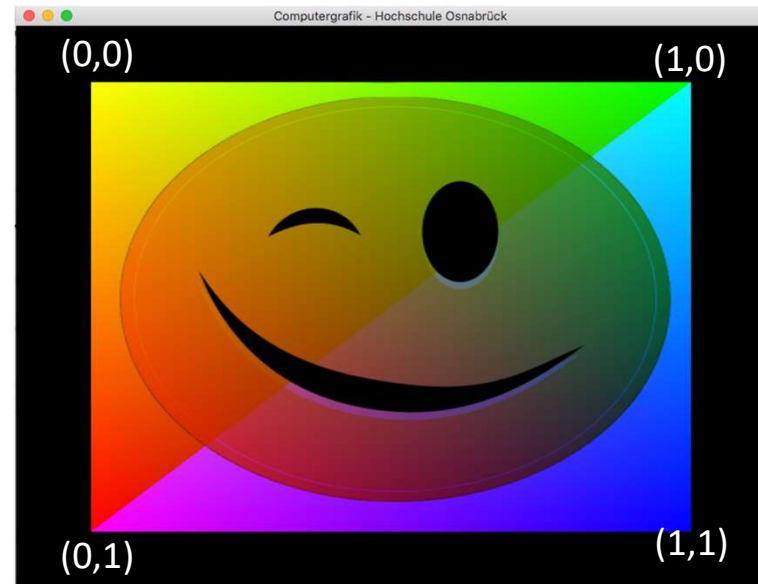


OpenGL Beispiel 3: Texturkoordinaten

Texturraum (hier smiley.png):



Bildraum:



Mit Hilfe von Texturen und Texturkoordinaten können 2D-Bilder auf 3D-Oberflächen „gezogen“ werden.

Hierauf werden wir später in der Veranstaltung noch genauer eingehen.

OpenGL Beispiel 3 (Vertex Buffer Objects)

```
void Application3::start()
{
    struct Vertex{ Vector Pos; Color C; float TexCoordU; float TexcoordV; };

    Vertex TwoTriBuffer[6] {
        Vertex{ Vector(-0.8, -0.8, 0), Color(1,0,0), 0, 1 },
        Vertex{ Vector(-0.8, 0.8, 0), Color(1,1,0), 0, 0 },
        Vertex{ Vector(0.8, 0.8, 0), Color(0,1,0), 1, 0 },
        Vertex{ Vector(0.8, 0.8,0), Color(0,1,1), 1, 0 },
        Vertex{ Vector(0.8,-0.8,0), Color(0,0,1), 1, 1 },
        Vertex{ Vector(-0.8,-0.8,0), Color(1,0,1), 0, 1 }
    };

    // glGenBuffers, glBindBuffers, etc.
    ...
    glGenVertexArrays(1, &VA0);
    glBindVertexArray(VA0);
    glEnableVertexAttribArray(0); Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(0));
    glEnableVertexAttribArray(1); Farbe
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(12));
    glEnableVertexAttribArray(2); Texturkoordinaten
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), BUFFER_OFFSET(24));
    // ...
    Tex = Texture::LoadShared("../assets/smiley.png");
}
```

Die Vertex-Struktur
wird um
Texturkoordinaten
erweitert (U,V).

Die Buffer-Struktur wird um die Texturkoordinaten erweitert.

Es wird eine Textur von der Festplatte geladen (*Texture* ist eine Hilfsklasse für das Praktikum und gehört nicht zur OpenGL-Schnittstelle).

OpenGL Beispiel 3 (Vertex & Fragment Shader)

```
GLuint Application3::createShader()  
{
```

```
    const char * VScore =
```

```
    "#version 400\n"
```

```
    "in vec4 VertexPos;"
```

```
    "in vec3 vColor;"
```

```
    "in vec2 vTexc;"
```

```
    "out vec3 fColor;"
```

```
    "out vec2 fTexc;"
```

```
    "void main() {"
```

```
    "    fTexc = vTexc;"
```

```
    "    gl_Position = VertexPos;"
```

```
    "    fColor = vColor;"
```

```
    "};"
```

```
    const char * FScore =
```

```
    "#version 400\n"
```

```
    "in vec3 fColor;"
```

```
    "in vec2 fTexc;"
```

```
    "out vec4 FragColor;"
```

```
    "uniform sampler2D Tex;"
```

```
    "void main() {"
```

```
    "    vec3 texcolor = texture(Tex, fTexc).rgb;"
```

```
    "    FragColor = vec4(fColor.xyz * texcolor, 0);"
```

```
    "};"
```

```
    // create, compile & link shader
```

```
    ...
```

```
    GLint TexLoc = glGetUniformLocation(Shader, "Tex");
```

```
    glUseProgram(Shader);
```

```
    if(TexLoc >= 0)
```

```
    {  
        glUniform1i(TexLoc, 0);
```

```
    }  
    glUseProgram(0);
```

```
    return Shader;
```

```
}
```

Die Texturkoordinaten werden als drittes Attribut vom Vertex Shader.

Der Vertex Shader leitet die Texturkoordinaten an den Fragment Shader weiter.

Die Texturkoordinaten im Eingang des Fragment Shaders.

Der Sampler für die Textur (per glUniform gesetzt).

Die Textur wird mit den Texturkoordinaten gesampled und das Ergebnis wird mit der Ausgabefarbe multipliziert.

Die Uniform-Variable mit der Textur (Tex) wird vom Host-Programm übergeben (es wird der Index der Textur-Stage übergeben).

OpenGL Beispiel 3 (Zeichnen)

```
void Application::draw()
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindVertexArray(VAO);
    glUseProgram(ShaderProgram);

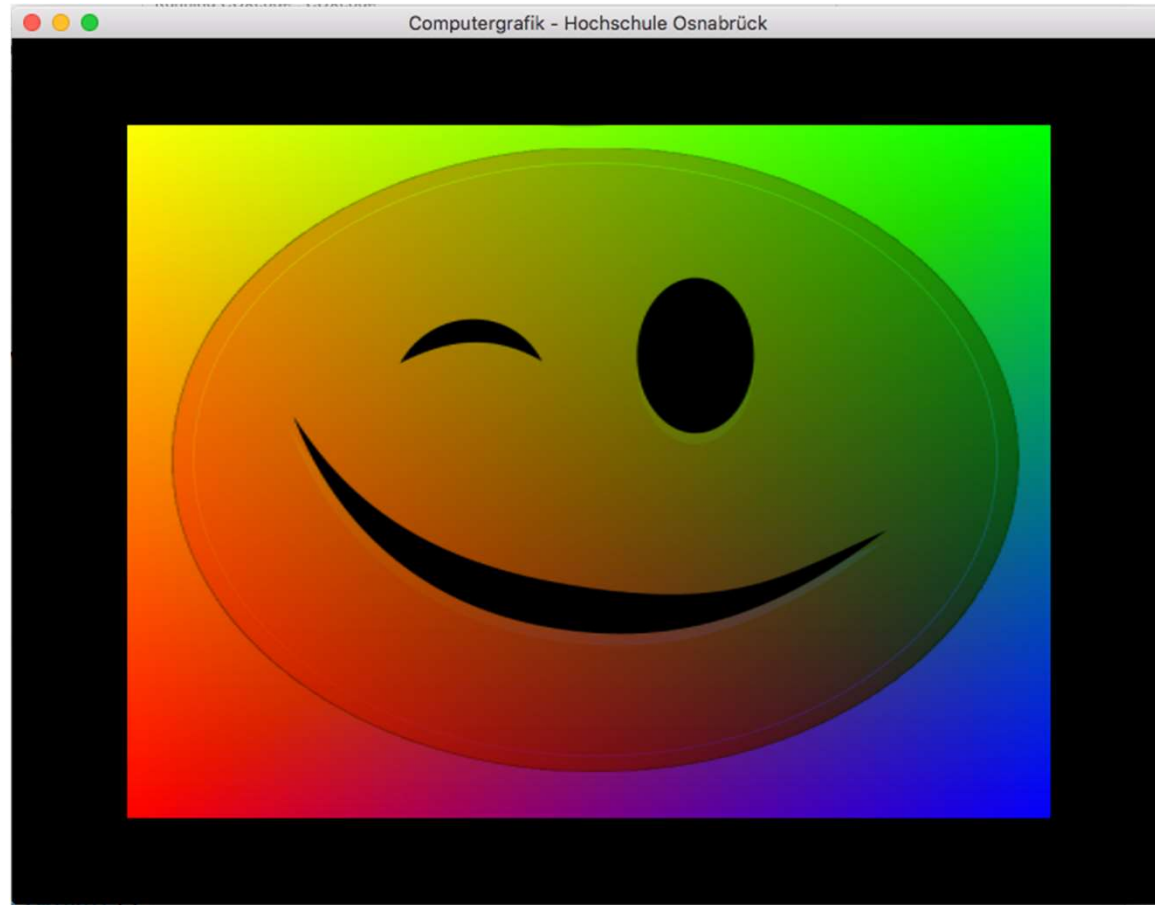
    Tex->activate(); ← Die geladene Textur wird in die Texture-Stage
                       0 gesetzt.

    glDrawArrays(GL_TRIANGLES, 0, 6);

    Tex->deactivate(); ← Die Textur wird wieder von der Texture-Stage
                       entfernt.

    // 3. check once per frame for opengl errors
    GLenum Error = glGetError();
    assert(Error==0);
}
```

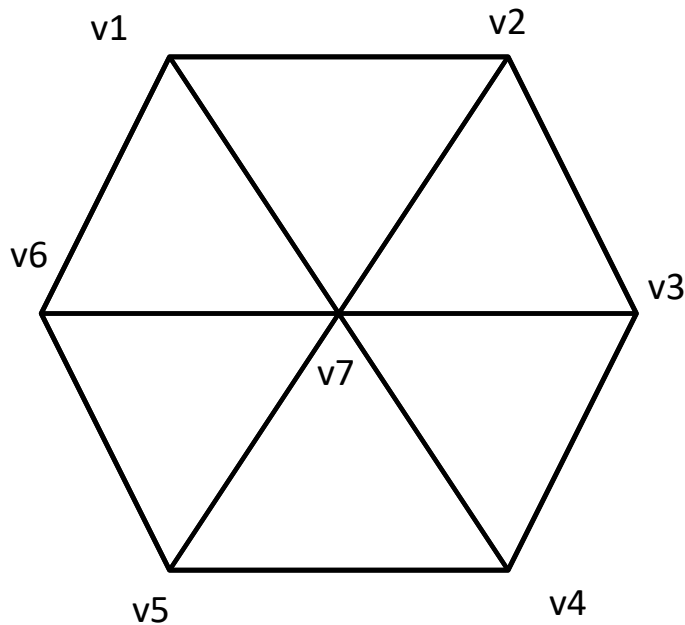
OpenGL Beispiel 4: Verwendung von **Index Buffer Objects** (Application4.cpp)



Index Buffer Objects

Index Buffer Objects werden genutzt, um Speicher zu sparen.

Anstatt Vertices mit gleichen Attributen mehrmals im VBO zu speichern, werden diese durch ein Index Buffer Object indiziert.



Nur VBO: `Vector v[18] = {
Vector(v1), Vector(v2), Vector(v7),
Vector(v2), Vector(v3), Vector(v7),
Vector(v3), Vector(v4), Vector(v7),
...
};`

VBO & IBO: `Vector v[7] = {
Vector(v1), Vector(v2), Vector(v3),
Vector(v4), Vector(v5), Vector(v6),
Vector(v7)
};
unsigned short i[18] = {
0, 1, 6, 1, 2, 6, 2, 3, 6,
3, 4, 6, 4, 5, 6, 5, 0, 6
};`

OpenGL Beispiel 4 (Vertex & Index Buffer Objects)

```
void Application4::start()
{
    struct Vertex{ Vector Pos; Color C; float TexCoordU; float TexcoordV; };

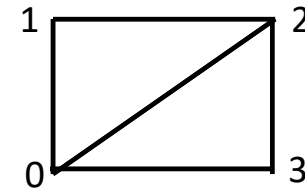
    Vertex TwoTriBuffer[4] {
        Vertex{ Vector(-0.8, -0.8, 0), Color(1,0,0), 0, 1 },
        Vertex{ Vector(-0.8, 0.8, 0), Color(1,1,0), 0, 0 },
        Vertex{ Vector(0.8, 0.8, 0), Color(0,1,0), 1, 0 },
        Vertex{ Vector(0.8,-0.8,0), Color(0,0,1), 1, 1 },
    };

    // initialize Vertex Buffer Object & Vertex Array Object
    ...

    unsigned short Indices[6] { 0, 1, 2, 2, 3, 0 };
    glGenBuffers(1, &IB0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IB0);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices, GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

    ...
}
```

Es werden nur 4 Vertices im VBO abgelegt.



Anlegen des Index Buffer Objects: Dieser speichert die Topologie des Objekts. Durch das IBO können gleiche Vertices mehrmals genutzt werden → Speicherersparnis.

OpenGL Beispiel 4 (Zeichnen mit IBOs)

Werden Index Buffer Objects verwendet, so muss die *glDrawElements(..)*-Funktion genutzt werden:

```
void Application4::draw()
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

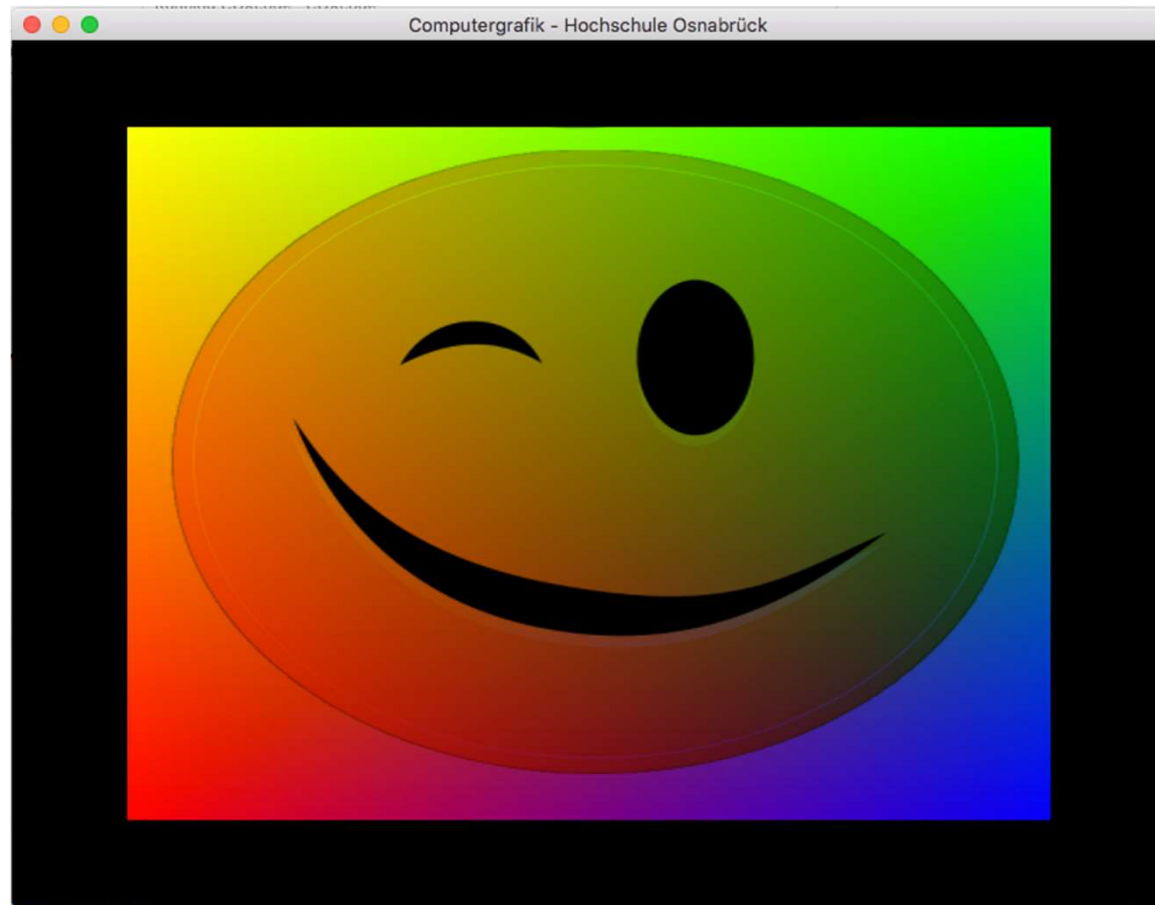
    glBindVertexArray(VAO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO); IBO aktivieren.
    glUseProgram(ShaderProgram);

    Tex->activate();
    // Zeichnet die Dreiecke mit Hilfe des IBOs.
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);

    Tex->deactivate();

    // 3. check once per frame for opengl errors
    GLenum Error = glGetError();
    assert(Error==0);
}
```

Beispiel 5: Verwendung der Hilfsklassen Vertex- und IndexBuffer des Praktikums (Application5.cpp)



OpenGL Beispiel 4 (Vertex- und IndexBuffer)

```
void Application5::start()
{
    VertexBuffer VB;
    VB.begin();
    VB.addColor(Color(1,0,0));
    VB.addTexCoord(0, 1);
    VB.addVertex(-0.8, -0.8, 0); // Vertex 0
    VB.addColor(Color(1,1,0));
    VB.addTexCoord(0, 0);
    VB.addVertex(-0.8, 0.8, 0); // Vertex 1
    VB.addColor(Color(0,1,0));
    VB.addTexCoord(1, 0);
    VB.addVertex(0.8, 0.8, 0); // Vertex 2
    VB.addColor(Color(0,0,1));
    VB.addTexCoord(1, 1);
    VB.addVertex(0.8, -0.8, 0); // Vertex 4
    VB.end();

    IndexBuffer IB;
    IB.begin();
    IB.addIndex(0); // Dreieck 0
    IB.addIndex(1);
    IB.addIndex(2);
    IB.addIndex(2); // Dreieck 1
    IB.addIndex(3);
    IB.addIndex(0);
    IB.end();
    ...
}
```

Vertex-Daten können zwischen begin() & end() definiert werden. Das letzte Attribut eines jeden Vertex muss die Position sein (addVertex(..)).

Index-Daten können zwischen begin() & end() definiert werden.

Die Klassen erzeugen alle nötigen OpenGL-Objekte selbständig (bei Aufruf von end()).

OpenGL Beispiel 4 (Zeichnen von Vertex- und IndexBuffern)

```
void Application5::draw()
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    VB.activate();
    IB.activate(); Die Vertex- und Indexbuffer wird aktiviert.
    glUseProgram(ShaderProgram);
    Tex->activate();
    glDrawElements(GL_TRIANGLES, IB.indexCount(), IB.indexFormat(), 0);
    IB.deactivate();
    VB.deactivate(); Der Vertex- und Indexbuffer wird deaktiviert.

    // 3. check once per frame for opengl errors
    GLenum Error = glGetError();
    assert(Error==0);
}
```


Vielen Dank für Ihre Aufmerksamkeit!