h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Visual Computing

Grafische Objekte und deren Programmierung

Hochschule Darmstadt

Prof. Dr. Elke Hergenröther
Björn Frömmer
Prof. Dr. Benjamin Meyer

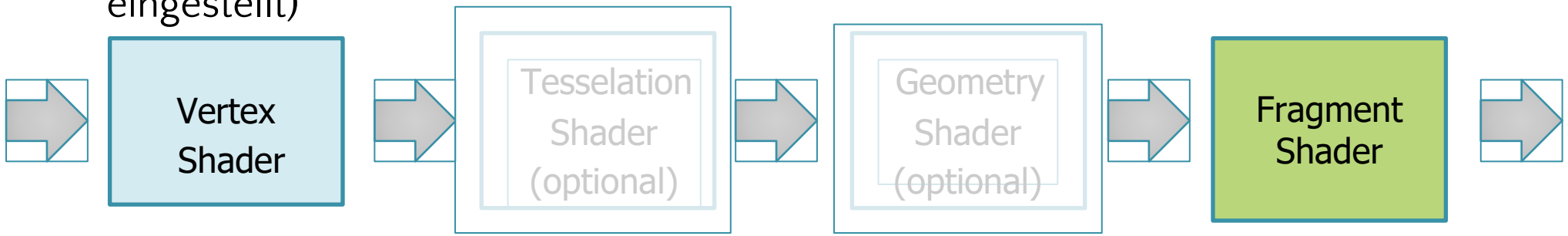
KAPITEL 6

Shaderprogrammierung

Shaderpipeline

- Die hochgeladenen Daten können vor dem Rendern noch manipuliert werden.
- Die gewünschten Berechnungen werden in sog. **Shadern** implementiert, kleinen Programmfragmenten, die direkt auf der GPU ausgeführt werden können.

- Computersprache der Shaderprogramme: GLSL (Open**GL S**hading **L**anguage)
  - Alternativen: HLSL für reine Windows/DirectX-Entwicklung, Cg (von NVidia, 2012 eingestellt)

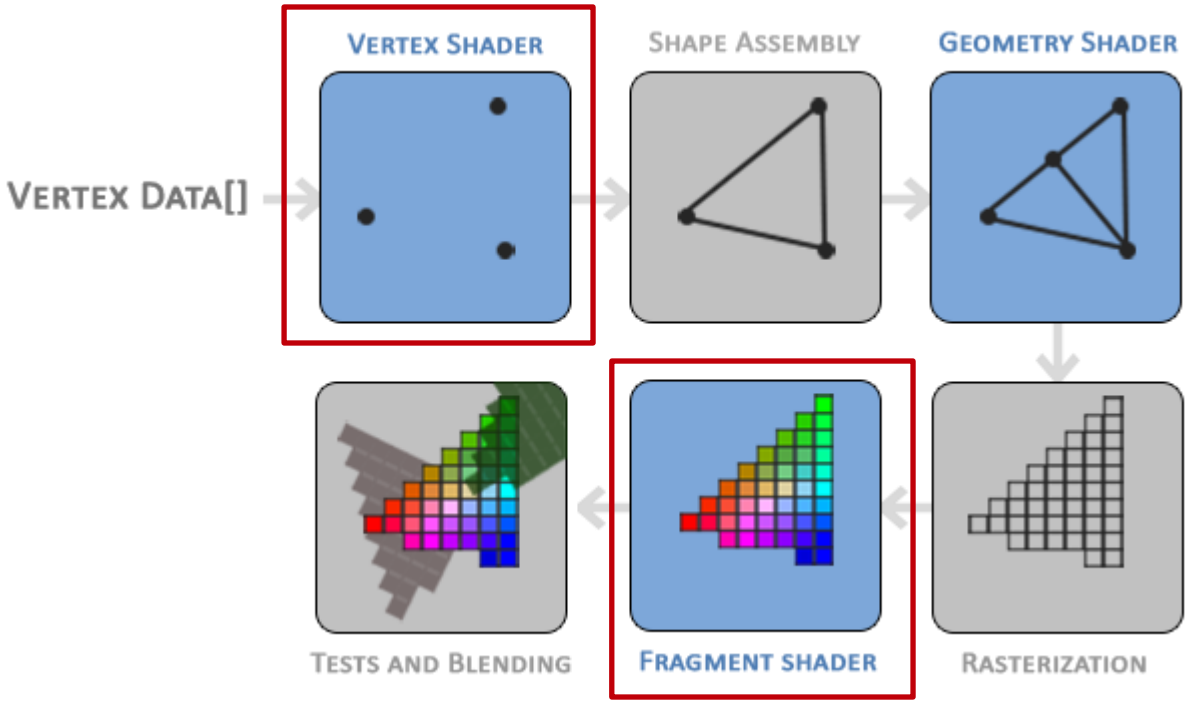


- Jeder Shader hat hierbei eine genau definierte Aufgabe:
  - **Vertex Shader:** Berechnet die **finale Position** jedes Vertex im Ausgabebild
  - **Fragment Shader:** Berechnet die **Farbe jedes Pixels** im Ausgabebild
    - (Tesselation Shader: Zerlegung der Objekte in feinere Dreiecksnetze)
    - (Geometry Shader: Veränderung der Geometrie-Daten zur Laufzeit)

Tradeoff Speicherplatz versus Performance

Shaderpipeline

Vertex Shader: Berechnet die **finale Position** jedes Vertex im Ausgabebild

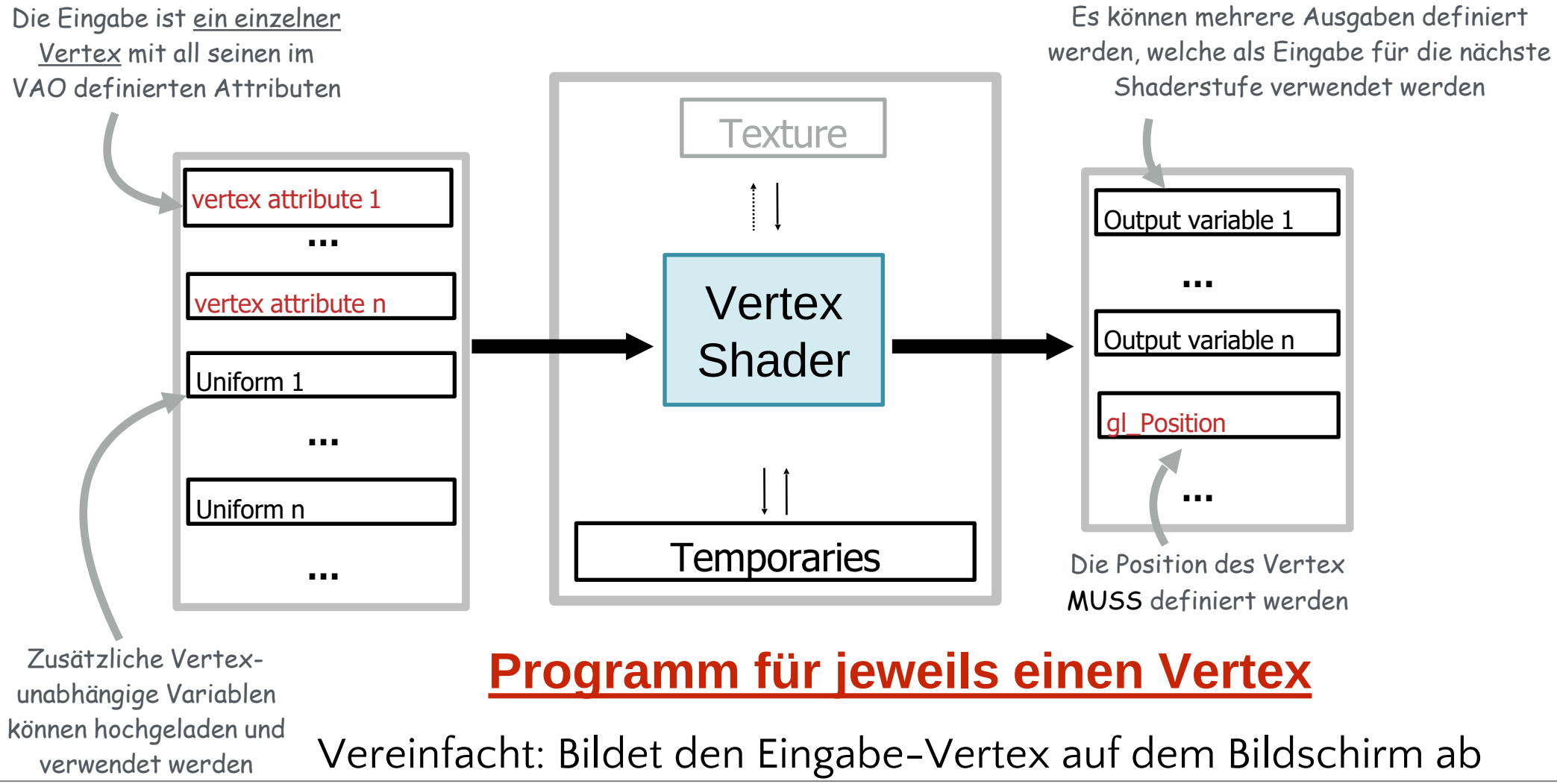


Der Geometrieshader ist ein optionaler Shader. Er kann einzelne Primitive, wie Dreiecke, transformieren und bspw. zusätzliche Vertices hinzufügen.

Fragment Shader: Berechnet die **Farbe jedes Pixels** im Ausgabebild

Aus: https://learnopengl.com/Getting-started/Hello-Triangle

Vertex Shader Stage

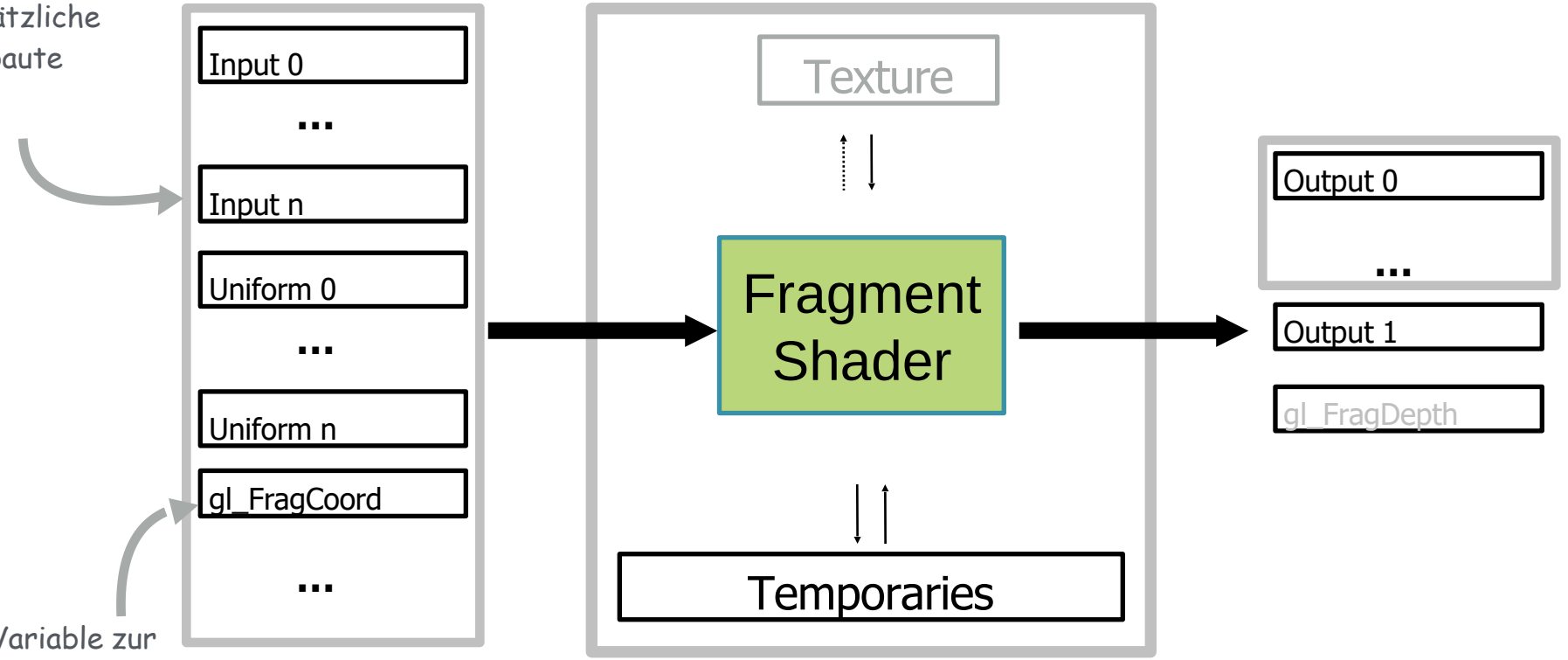


Programm für jeweils einen Vertex

Vereinfacht: Bildet den Eingabe-Vertex auf dem Bildschirm ab

Fragment Shader Stage

Erhält als Eingabe die Ausgabe der vorherigen Shaderstufe sowie optionale zusätzliche Daten und eingebaute Variablen



Eingebaute Variable zur Abfrage des aktuellen Fragments (Pixel)

Programm für jeweils ein Fragment!

Vereinfacht: Definiert die Ausgabefarbe für jedes Pixel

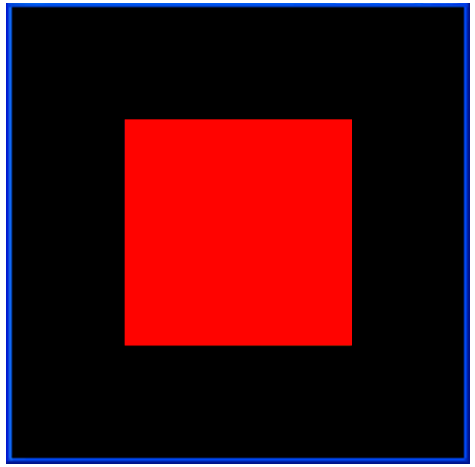
Ein einfaches Beispiel

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4 color;

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



Output

Ein einfaches Beispiel

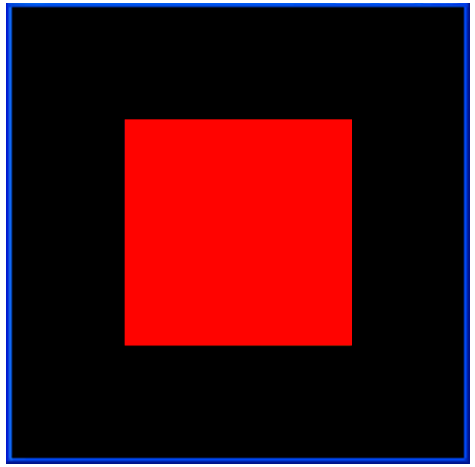
Definition der verwendeten OpenGL/GLSL-Version

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4 color;

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



Output

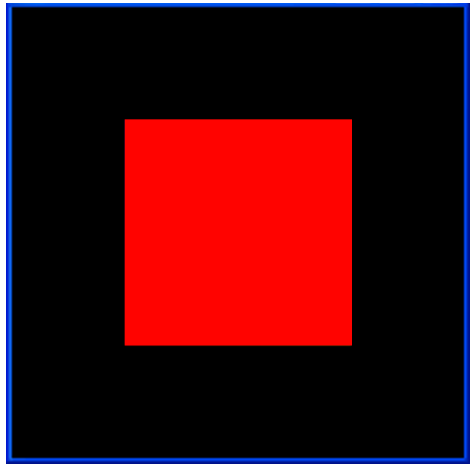
Ein einfaches Beispiel

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4 color;

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



Output

Allgemeine Form:

```
layout(location = index) in type name;
```

Der Index verweist auf das jeweilige VAO-Attribut aus

```
glVertexAttribPointer(GLuint index, ...)
```

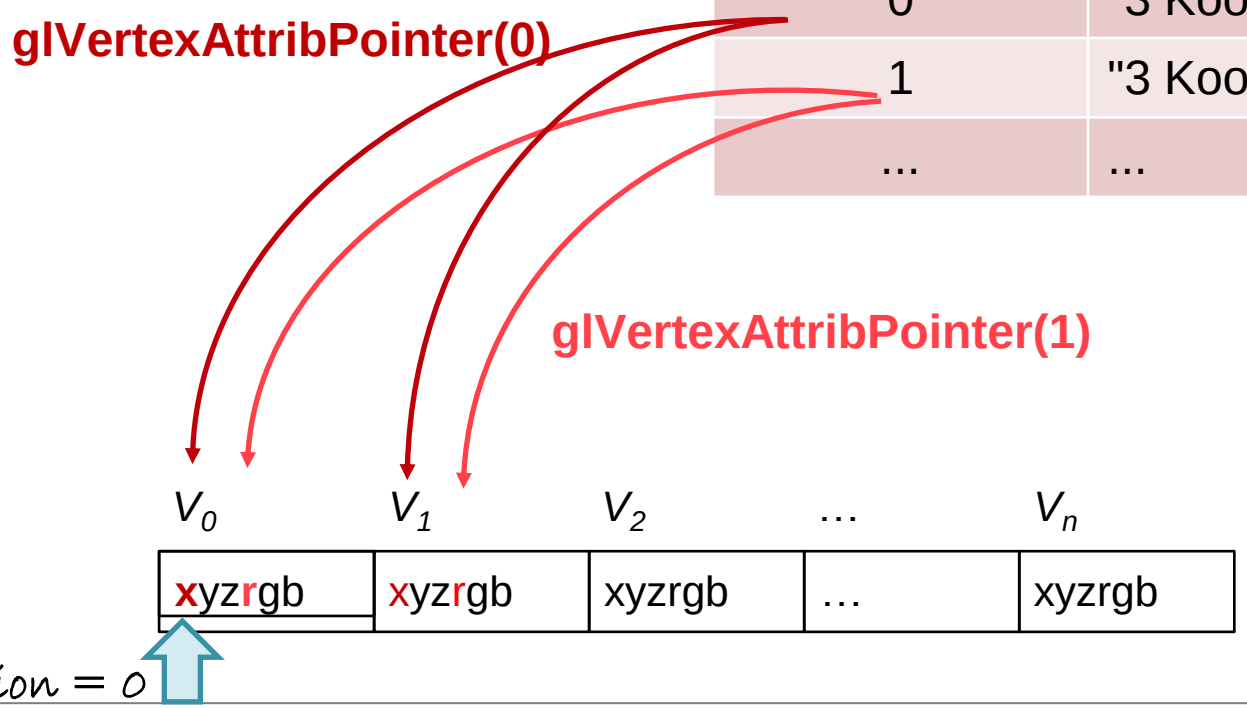
→ Der Programmierer muss wissen, was das Attribut enthält

Vertex Buffer Object (VBO), Vertex Array Object (VAO) und VertexAttribPointer

Vertex Array Object (VAO)

Aufgabe: Kennt die Struktur in der die Daten im VBO vorliegen. Die „VertexAttribPointer“ sind die Objekte in diesem Array, die die Ablagestruktur der Daten speichern/kennen.

Attribut-Index	Beschreibung
0	"3 Koordinaten vom Typ GL_FLOAT pro Vertex"
1	"3 Koordinaten vom Typ GL_FLOAT pro Vertex"
...	...



Vertex Buffer Object (VBO)

Aufgabe: Sammelt die Daten und lädt die Daten von der CPU auf die GPU, von wo sie dann bspw. visualisiert werden.

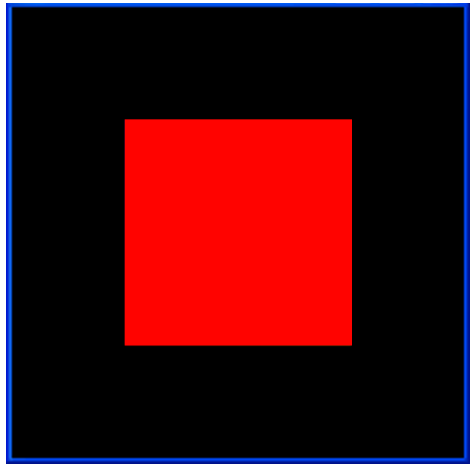
Ein einfaches Beispiel

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4 color;

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



Output

Datentypen

- Floatingpoint: `float, vec2, vec3, vec4`
- Integer: `int, ivec2, ivec3, ivec4`
- Boolean: `bool, bvec2, bvec3, bvec4`
- Matrix: `mat2 (2x2), mat3 (3x3), mat4 (4x4)`

- Zugriff auf verschiedene Elemente des Vektors/der Matrix:
  .xyzw, .rgba, .stqp, [i], [i][j] (für Matrizen)

```
vec2 v2; vec3 v3; vec4 v4;
v2.x                    // ergibt einen float
v2.z                    // Fehler: undefined for type
v4.rgba                 // ergibt einen vec4
v4.xy                   // ergibt einen vec2
v4.xgp                  // Fehler: nicht übereinstimmende Komponenten
```

Datentypen – Swizzling and Smearing

R-Values (lesend):

```
v4.wzyx      // korrekt, ergibt einen vec4(w,z,y,x)
v4.xxx       // korrekt, ergibt vec3(x,x,x)
v4.yyxx      // korrekt, verdoppelt x und y, ergibt einen vec4(y,y,x,x)
v2.yyzz      // Fehler: Zu viele Komponenten für Typ
```

L-Values (schreibend):

```
vec4 v4 = vec4( 1.0, 2.0, 3.0, 4.0);
v4.xw = vec2( 5.0, 6.0);                // 5.0, 2.0, 3.0, 6.0)
v4.wx = vec2( 7.0, 8.0);                // (8.0, 2.0, 3.0, 7.0)
v4.xx = vec2( 9.0,10.0);                // Fehler: x doppelt verwendet
v4.yz = 11.0;                           // Fehler: Type mismatch
v4.yz = vec2( 12.0 );                   // (8.0, 12.0, 12.0, 7.0)
```

Vordefinierte Funktionen

Verwenden Sie nach Möglichkeit eingebaute Funktionen anstelle Ihrer eigenen!

- Winkel & Trigonometrie:
  - `radians, degrees, sin, cos, tan, asin, acos, ...`
- Exponentialfunktionen:
  - `pow, exp, log, sqrt, ...`
- Allgemeine Funktionen:
  - `abs, ceil` (rundet den übergebenen Fließkommawert auf die nächst höhere natürliche Zahl), `clamp` (einen Wert auf einen Wert zwischen zwei weiteren Werten beschränken), `min, max, ...`
- Geometrische Funktionen:
  - `cross, dot, length, normalize, reflect` (Berechnung der Reflexionsrichtung für einen einfallenden (Licht-)Vektor), `...`

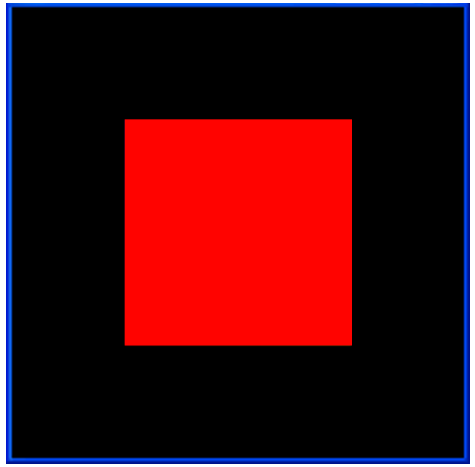
Ein einfaches Beispiel

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

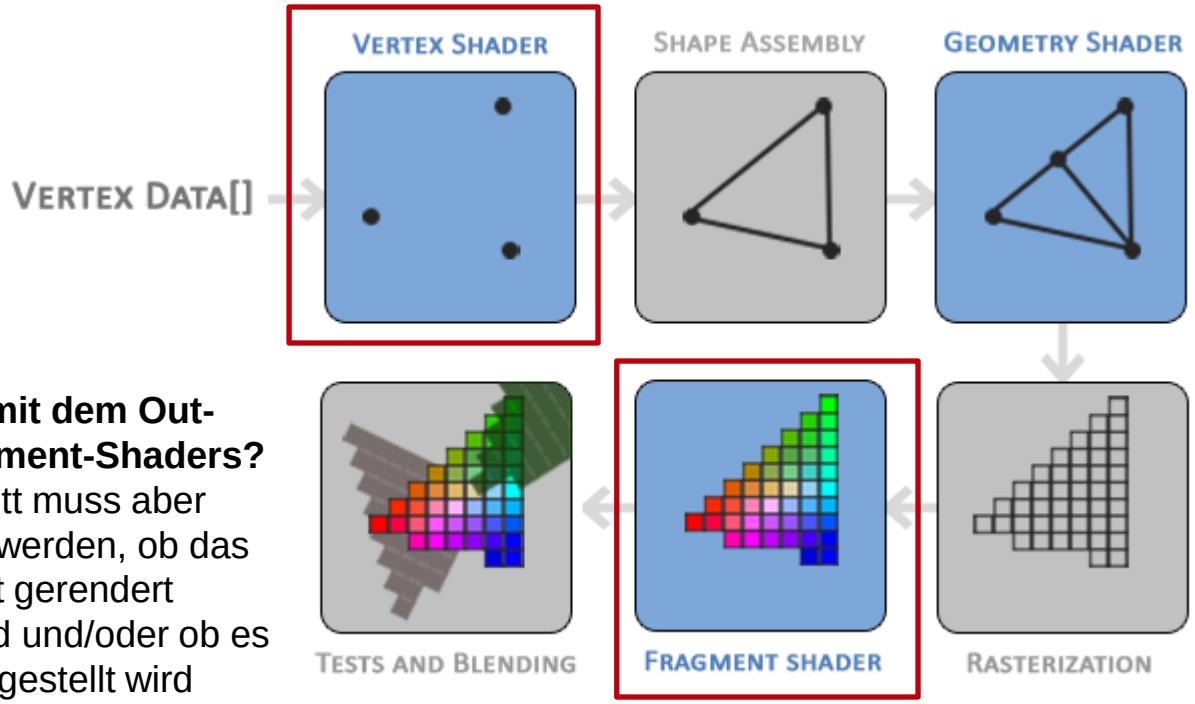
```
#version 330
out vec4 color;

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



Output

Shaderpipeline

Vertex Shader: Berechnet die **finale Position** jedes Vertex im Ausgabebild



Fragment Shader: Berechnet die **Farbe jedes Pixels** im Ausgabebild

Der Geometrieshader ist ein optionaler Shader. Er kann einzelne Primitive, wie Dreiecke, transformieren und bspw. zusätzliche Vertices hinzufügen.

Was passiert mit dem Out-Wert des Fragment-Shaders?
Im letzten Schritt muss aber noch überprüft werden, ob das Pixel überhaupt gerendert (Tiefentest) wird und/oder ob es transparent dargestellt wird (Blending).

Aus: https://learnopengl.com/Getting-started/Hello-Triangle

Speichertypen

- Erforderlich für die Kommunikation zwischen Shadern und Anwendung
- **in**
  - Verknüpfung mit dem Shader aus der vorherigen Stufe
  - Eingabe pro Vertex in den Vertex-Shader von OpenGL oder der Anwendung (READ-ONLY)
    oder: Eingabe pro Fragment für Fragment-Shader (READ-ONLY)
- **out**
  - Verknüpfung aus dem Shader zur nächsten Stufe
  - Weitergabe von Vertex (READ/WRITE) an Fragment-Shader zur endgültigen Ausgabe, interpoliert
- **uniform**
  - Eingabe in ein beliebiges Shader-Programm von OpenGL oder einer Anwendung (READ-ONLY)
  - konstant während des Rendervorgangs

Uniforms

- Uniforms sind vom Benutzer bereitgestellte Variablen von der Anwendung zu den Shadern

Adressieren von Variablen der Shader per Variablenname!

- **Upload von uniforms**
  - Speicherplatz der Variablen im Shader-Programm finden
    - `GLint glGetUniformLocation(GLuint programID, const GLchar *name);`
  - uniform hochladen
    - `void glUniform{1,2,3,4}{i,f}(GLint location,  GLfloat v0[, v1, v2, v3]);`
    - `void glUniformMatrix{234}fv(GLint location, GLsize count, GLboolean transpose, const Glfloat *value);`
    - `count` gibt die Anzahl der Matrizen an, die geändert werden sollen. 1, wenn es sich bei der Zielvariable nicht um ein Matrizenfeld handelt, und 1 oder mehr, wenn es sich um ein Array von Matrizen handelt.

In unserem Framework: (bei Matrizen)

```
ShaderProgram::setUniform(string variable, TYPE value, [bool transpose]);
```

Ein einfaches Beispiel

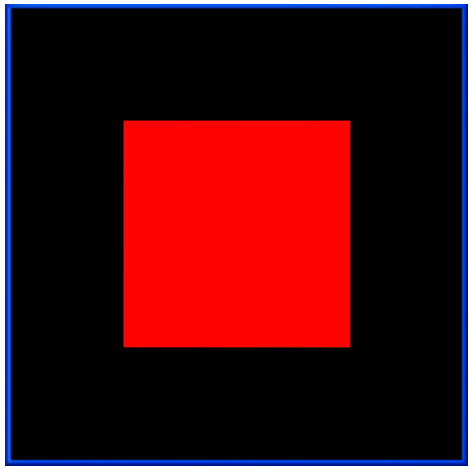
Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

out vec4 passOn;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
    passOn = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Gleicher Variablenname und -typ!

Fragment Shader

```
#version 330
out vec4 color;

in vec4 passOn;

void main()
{
    color = passOn;
}
```



Output

Ein einfaches Beispiel

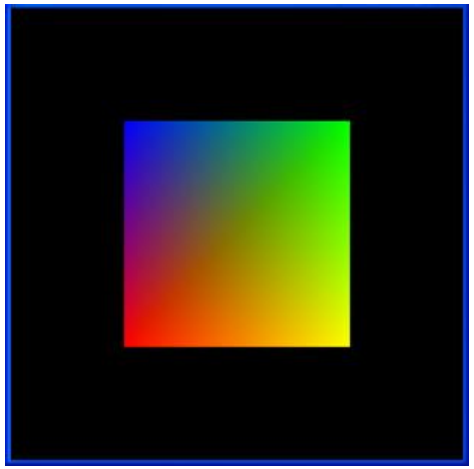
Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;
layout(location = 1) in vec3 color;

out vec4 passOn;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
    passOn = vec4(color, 1.0);
}
```

automatische Interpolation zwischen den Vertexdaten!

Fragment Shader

```
#version 330
out vec4 color;


in vec4 passOn;

void main()
{
    color = passOn;
}
```



Output

Ein einfaches Beispiel

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;
layout(location = 1) in vec3 color;

out vec4 passOn;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
    passOn = vec4(color, 1.0);
}
```

Multiplikation der alten Vertexposition mit der akkumulierten Transformationsmatrix

Jeder Vertex-Shader MUSS die (eingebaute) Variable gl_Position mit der homogenen Vertexposition füllen!

Fragment Shader

```
#version 330
out vec4 color;

in vec4 passOn;

void main()
{
    color = passOn;
}
```

Definition der Output-Variablen

Jeder Fragment-Shader MUSS mind. eine Output-variable definieren und diese mit der finalen Farbe des Fragments füllen!