LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Dipl.Inf. Otmar Hilliges

Programmierpraktikum 3D Computer Grafik

GLSL

Agenda

Rendering Pipeline

Prozessoren:

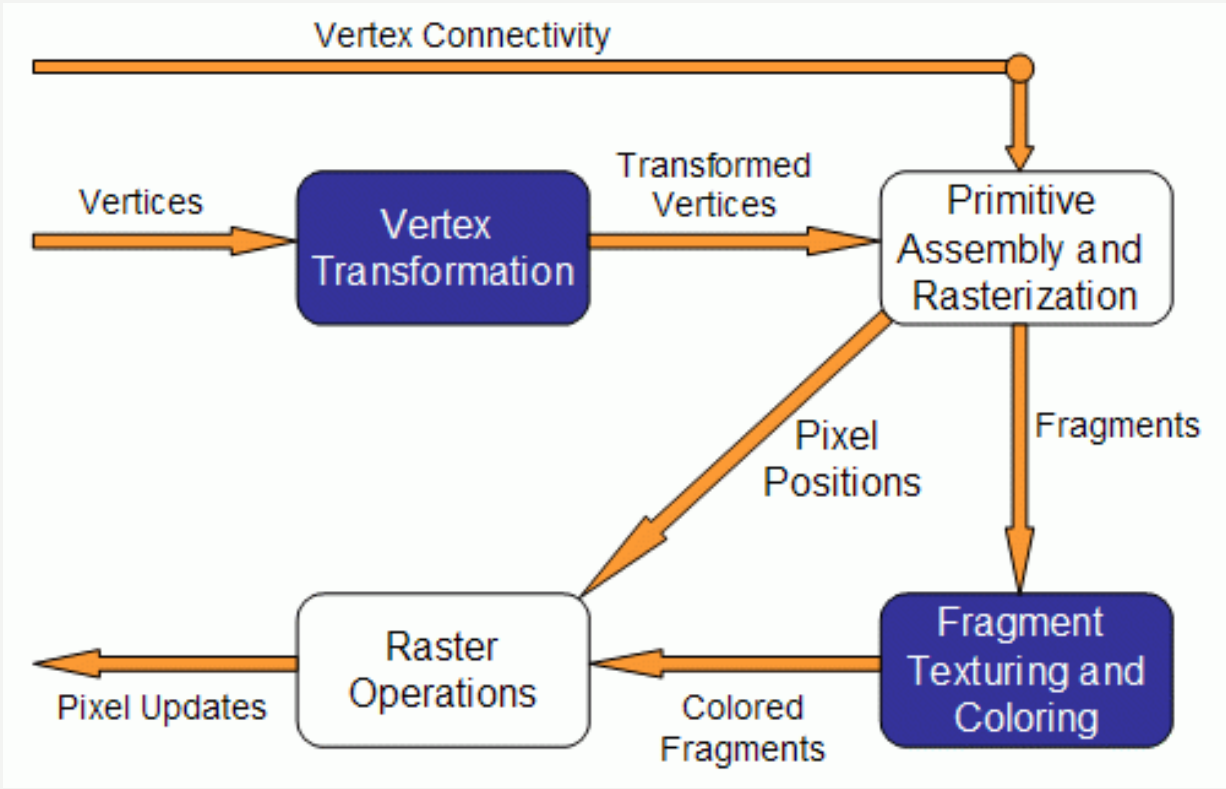
- Vertex Prozessor
- Fragment Prozessor

OpenGL Setup für GLSL

Shader-Grundlagen

Beispiele

Die Rendering-Pipeline



Vertex Transformation:

- Ein Vertex ist eine Menge von Attributen (Koordinaten, Normale, Farbe, …)
- Operationen, die durchgeführt werden:
  - Transformation der Vertexposition
  - Lichtberechnungen pro Vertex
  - Generierung und Transformation der Texturkoordinaten

Primitive Assembly and Rasterization:

- Zusätzlich zur Vertextransformation werden hier die Verbindungen der einzelnen Vertices als Eingabe verwendet
- Verantwortlich für Clipping, Backface Culling
- Ausgabe:
  - Position der Fragmente im Buffer
  - Interpolierte Werte für jedes Fragment der vorher berechneten Attribute

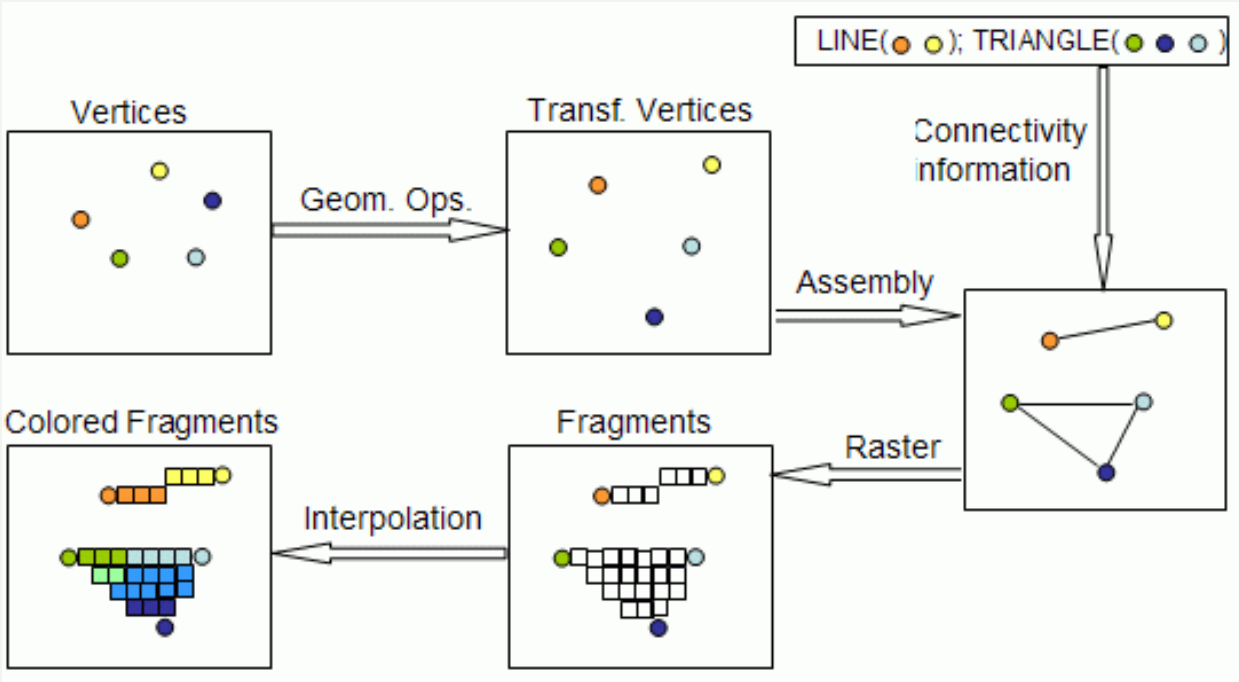
Fragment Texturing and Coloring:

- Farb- und/oder Texturwerte wurden vorher schon interpoliert
- Nebel wird in dieser Stufe hinzugefügt
- Ausgabe:
  - Farb- und Tiefenwert für jedes Fragment

Raster Operations:

- Führt Tests durch, z.B.
  - Alpha
  - Stencil
  - Tiefe
- Nur in dieser Stufe kann auf den Framebuffer zugegriffen werden

Die Rendering-Pipeline



Vertexprozessor:

- Verarbeitet Vertex-Shader
- Hat keinen Zugriff auf den Framebuffer
- Vertex-Shader können verwendet werden, um Vertexdaten zu verändern:
  - Positions- und Normalentransformation
  - Generierung und Transformation der Texturkoordinaten
  - Beleuchtung pro Vertex oder Wertberechnung für Beleuchtung pro Pixel
  - Farbberechnung

Fragmentprozessor:

- Verarbeitet Fragment-Shader
- Fragment-Shader können verwendet werden, um Fragmentdaten zu verändern:
  - Berechnung von Farben und Texturkoordinaten pro Pixel
  - Texturanbringung
  - Nebelberechnung
  - Normalenberechnung falls eine Beleuchtung pro Pixel gewünscht ist

OpenGL Setup für GLSL

Shader sind eine Art C Modul

Müssen separat kompiliert werden

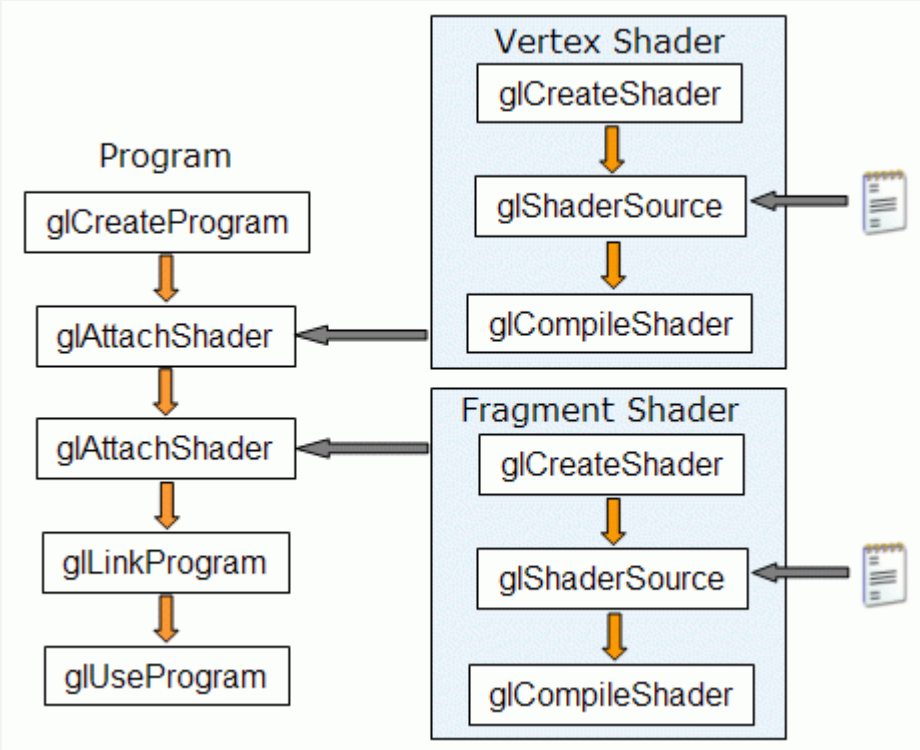
Wird später (wie bei C) in ein Programm gelinkt

Zwei Möglichkekiten in OpenGL:

- OpenGL 2.0
- OpenGL Extensions

Vereinfachung: GLEW (OpenGL Extension Wrangler)

OpenGL Setup für GLSL



Schritte, um einen Shader zu erstellen (1):

- Erzeugen eines Objekts (Shader-Container):

```
/* OpenGL 2.0 syntax */
GLuint glCreateShader(GLenum shaderType);
//shaderType: GL_VERTEX_SHADER oder GL_FRAGMENT_SHADER

/* ARB extensions syntax */
GLhandleARB glCreateShaderObjectARB(GLenum shaderType);
//shaderType: GL_VERTEX_SHADER_ARB oder
//                    GL_FRAGMENT_SHADER_ARB
```

Schritte, um einen Shader zu erstellen (2):

- Sourcecode laden:

```
void glShaderSource(GLuint shader, int numOfStrings,
                    const char** strings, int* lenOfStrings);

void glShaderSourceARB(GLhandleARB shader, int
                               numOfStrings, const char** strings,
                               int* lenOfStrings);

// shader: Handle des Shader-Objekt
// numOfStrings: Anzahl der Strings
// strings: Array mit Strings
// lenOfStrings: Array mit den Längen jedes Strings
// (NULL = dass die Strings NULL-terminiert sind)
```

Schritte, um einen Shader zu erstellen (3):

- Shader kompilieren:

```
/* OpenGL 2.0 syntax */
void glCompileShader(GLuint program);
// program: Handler zum Programm

/* ARB extensions syntax */
void glCompileShaderARB(GLhandleARB program);
// program: Handler zum Programm
```

Schritte für ein Programm (1):

- Erzeugen eines Objekts (program container):

```
/* OpenGL 2.0 syntax */
GLuint glCreateProgram(void);

/* ARB extensions syntax */
GLhandleARB glCreateProgramObjectARB(void);
```

Schritte für ein Programm (2):

- Shader hinzufügen:

```
/* OpenGL 2.0 syntax */
void glAttachShader(GLuint program, GLuint shader);

/* ARB extensions syntax */
void glAttachObjectARB(GLhandleARB program,
                                          GLhandleARB shader);

// program: Handler zum Program
// shader: Handler für den Shader, der hinzugefügt
//         werden soll
```

Schritte für ein Programm (3):

- Programm linken und verwenden:

```
/* OpenGL 2.0 syntax */
void glLinkProgram(GLuint program);
void glUseProgram(GLuint prog);

/* ARB extensions syntax */
void glLinkProgramARB(GLhandleARB program);
void glUseProgramObjectARB(GLhandleARB prog);

// program: Handler zum Program
// shader: Handler für das Programm oder 0, falls feste
//         Funktionalität benötigt wird
```

Beispiel (OpenGL 2.0):

```
…
char *vs, *fs;
v = glCreateShader(GL_VERTEX_SHADER);
f = glCreateShader(GL_FRAGMENT_SHADER);

vs = textFileRead(“toon.vert”);
fs = textFileRead(“toon.frag”);

const char *vv = vs;
const char *ff = fs;
glShaderSource(v, 1, &vv, NULL);
glShaderSource(f, 1, &ff, NULL);

free(vs); free(fs);
```

Beispiel (OpenGL 2.0):

```
…
glCompileShader(v);
glCompileShader(f);

p = glCreateProgram();

glAttachShader(p, v);
glAttachShader(p, f);

glLinkProgram(p);
glUseProgram(p);
…
```

Clean-Up-Verfahren:

```
/* OpenGL 2.0 syntax */
void glDetachShader(Gluint program, GLuint shader);
void glDeleteShader(GLuint id);
void glDeleteProgram(GLuint id);

/* ARB extensions syntax */
void glDetachObjectARB(GLhandleARB program,
                                          GLhandleARB shader);
void glDeleteObjectARB(GLhandleARB id);
```

Datentypen in GLSL:

`float, bool, int`

Vektoren in GLSL:

`vec{2, 3, 4}: Vektor mit 2, 3 oder 4 float-Werten`
`bvec{2, 3, 4}: Vektor mit boolschen Werten`
`ivec{2, 3, 4}: Vektor mit int-Werten`

Matrizen in GLSL:

`mat2, mat3, mat4: 2x2, 3x3 oder 4x4 Matrix`

Texturzugriff in GLSL:

`sampler{1D, 2D, 3D}: 1D-, 2D- oder 3D-Textur`
`samplerCube: für Cubemap-Texturen`
`sampler{1D, 2D}Shadow: für Shadowmaps`

- Strukturen und Arrays sind erlaubt und werden verwendet wie in C
- Automatisches Typecasting funktioniert nicht, d.h. `float a=2` gibt einen Fehler
- Qualifikatoren für Variablen:

  `const, attribute, uniform, varying`

- `if/else` und alle drei Schleifentypen existieren wie auch in C

Funktionen:

- Wie in C sind Shader in Funktionen unterteilt
- Sind aufgebaut wie in C, d.h. Rückgabewert möglich (darf aber kein Array sein)
- Jeder Shadertyp muss eine Funktion `void main()` enthalten

Varying bei Variablen:

- Meist müssen interpolierte Daten im Fragment-Shader verwendet werden
- Daher:
  - Deklaration einer Variablen in beiden Shadern
  - Variable muss dafür mit `varying` gesetzt werden
- Varying-Variablen werden im Vertex-Shader geschrieben, im Fragment-Shader gelesen

Vertex Shader Beispiel

```
//access the ModelViewProjectionMatrix
uniform mat4 gl_ModelViewProjectionMatrix;
//uniform mat4 gl_ModelViewMatrix;
//uniform mat4 gl_ProjectionMatrix;

//access to incoming Vertices
attribute vec4 gl_Vertex;

//Imitate standard procedure:
void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * glVertex;
}

//Beware: No lightning computation!
```

Fragment Shader Beispiel

```
//write fragment color
void main()
{
    gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Flatten Shader

```
void main(void)
{
    vec4 v = vec4(gl_Vertex);
    v.z = 0.0;
    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Beispiele und Tutorials

OpenGL Shading Language @ Lighthouse 3D:
http://www.lighthouse3d.com/opengl/glsl/index.php?shaders

Tutorial GLSL – DGL Wiki:
http://wiki.delphigl.com/index.php/Tutorial_glsl