



# Einführung in GLSL - OpenGL Shading Language

Athanasios Karamalis

---

## Allgemein zur Shader-Programmierung

- Vor 2001 konnte nur die sogenannte „Fixed Functionality“ der Graphik API und Graphikkarte verwendet werden
- In 2001 kam die erste über Shader programmierbare Graphikkarte (GeForce3) auf dem Markt
- Shader-Programmierung erlaubt es die Fixed Functionality zu erweitern
- Die Graphik-Verarbeitungsschritte können teilweise selbst für die Graphikkarte programmiert werden
- Vorteile:
  - Mehr Flexibilität
  - Verbesserte Effekte wie für Schatten, Licht etc.
  - Neue Effekte wie Ocean Simulation, Interactive Flamen usw.
  - Neue Visualisierungs-Methoden wie Ray-Casting für Volumen
- Unter Nvidia.com gibt es gratis die Bücher-Serie GPU Gems 1,2,3 welche Unmengen an State-Of-The-Art Effekten beschreibt

# Graphics Processing Pipeline

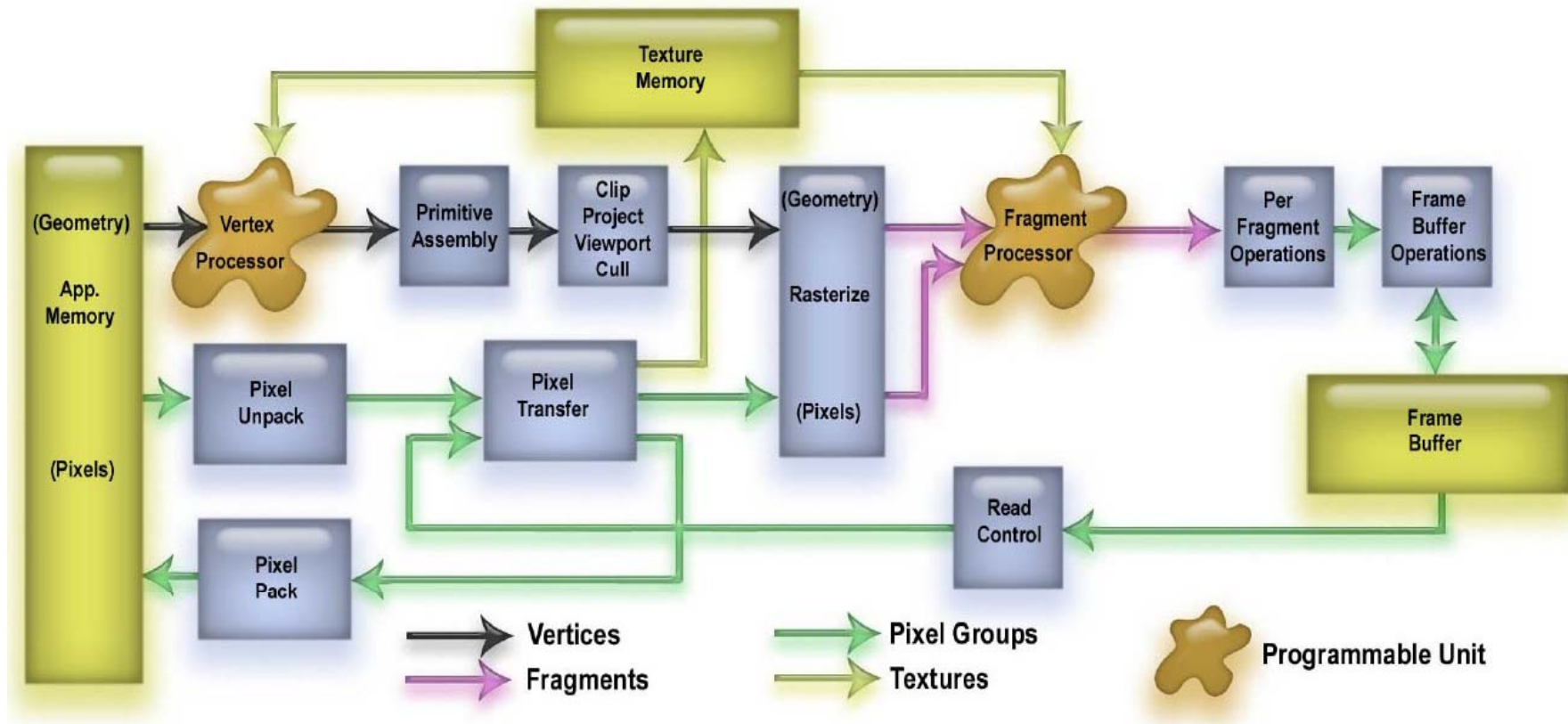


Image: 3DLabs



## GLSL-OpenGL Shading Language

- GLSL ist eine Sprache zum Shader Programmieren (Plattformunabhängig)
  - Es gibt auch Cg (Plattformunabhängig) von Nvidia und Microsoft
  - Und HLSL welche nur für Windows zur Verfügung steht
- Seit OpenGL 2.0 im Standard mit enthalten
  - Der Standard wird durch die Grafiktreiber implementiert
- GLSL ist eine C-basierte high-level Sprache
  - Es wird nicht der komplette C Standard unterstützt (z.B. keine Pointer)
  - Mathematische Funktionen wie Matrix und Vektor Operationen sind unterstützt
- Zwei verschiedene Shader Typen, Vertex und Fragment Shader
- Geometry Shader als Extension erhältlich, wird aber hier nicht bearbeitet

## Vertex Shader

- Input:
  - Vertex-Informationen, z.B. Position, Farbe, Normale
  - nur per-Vertex, ohne topologische Informationen
- Vertex shader:
  - Vertex-Transformationen, normalerweise modelview und projective Transformation

```
glBegin(GL_TRIANGLES);  
glColor3f(1.0, 0.0, 0.0);  
glVertex3f(0.0, 0.0, 0.0);  
glColor3f(0.0, 1.0, 0.0);  
glVertex3f(0.0, 1.0, 0.0);  
glVertex3f(0.0, 0.0, 1.0);  
glEnd;
```

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

- Beleuchtung per Vertex
  - Textur-Koordinaten
- Output: `gl_Position`
- Ein Vertex Shader muss die komplette Standardfunktionalität implementieren (z.B. Beleuchtung)

## Fragment (pixel) Shader

- Input:
  - Interpolierte Informationen, z.B. Farbe, Normale
- Fragment shader:
  - Beleuchtung
  - Textur
  - Nebel
- Output: `gl_FragColor`, Farbwert pro Pixel
- Ein Fragment shader muss die komplette Standardfunktionalität implementieren (z.B. Nebel)

## Beispiel: Vertex-Shader

### Default

```
void main() {  
    gl_Position = ftransform();  
}
```

### oder

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

### Änderung der z-Koordinate

```
void main() {  
    vec4 v = vec4(gl_Vertex);  
    v.z = sin(5.0*v.x)*0.5;  
    gl_Position = gl_ModelViewProjectionMatrix * v;  
}
```



## Kommunikation OpenGL -> Shader

- Kommunikation nur in eine Richtung
- Über Vertices: z.B. glVertex, glColor, glNormal
- Variablen, read-only für den Shader
  - *uniform*:
    - per primitive (kann nur außerhalb glBegin / glEnd geändert werden)
    - Kann von vertex und fragment shadern gelesen werden
    - Built-in uniforms: z.B. gl\_ModelViewMatrix
  - *attribute*:
    - per vertex
    - Kann nur von vertex shadern gelesen werden
- Variablen werden aus dem CPU-Program über glUniform bzw. glVertexAttrib gesetzt

## Kommunikation vertex shader -> fragment shader

- *Varying* Variablen
- müssen sowohl im vertex als auch im fragment shader definiert sein
- werden per-vertex definiert und automatisch per-fragment interpoliert
- Built-in varying: z.B. Farbe



## Beispiel 1: Vertex und Fragment-Shader

```
varying float intensity;
void main()
{
    vec4 vertexPos = gl_ModelViewMatrix * gl_Vertex;
    vec3 ld = vec3(gl_LightSource[0].position-vertexPos);
    vec3 lightDir = normalize(ld);
    intensity = dot(lightDir,gl_Normal);
    gl_Position = vertexPos;
}
```

### Vertex Shader

intensity wird per-vertex berechnet und dann interpoliert

```
varying float intensity;
void main()
{
    gl_FragColor = vec4(intensity, intensity, intensity, 1.0);
}
```

### Fragment Shader



## Beispiel 2: Vertex und Fragment-Shader

### Vertex Shader

```
varying vec3 Id, normal;
void main()
{
    vec4 vertexPos = gl_ModelViewMatrix * gl_Vertex;
    Id = vec3(gl_LightSource[0].position-vertexPos);
    normal = gl_Normal;
    gl_Position = vertexPos;
}
```

Id und normal werden per-vertex berechnet und per-fragment interpoliert. Intensity wird dann per-fragment berechnet

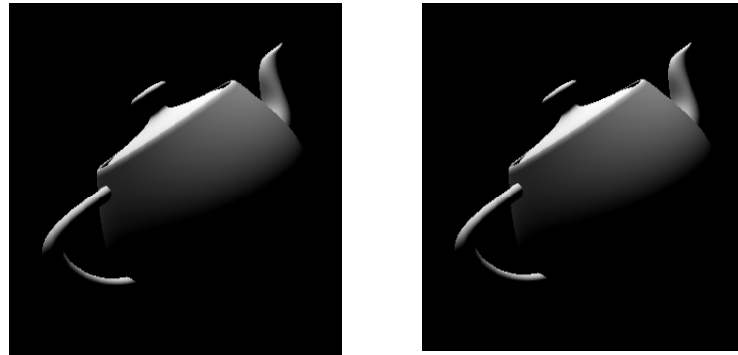
### Fragment Shader

```
varying vec3 Id, normal;
void main()
{
    vec3 lightDir = normalize(Id);
    float intensity = dot(lightDir, normal);
    gl_FragColor = vec4(intensity, intensity, intensity, 1.0);
}
```



# Unterschied

Hier kaum sichtbar



Aber: diffuse + ambient + specular point light



*Beispiel 1*



*Beispiel 2*

## Textur hinzufügen

```
varying vec3 Id, normal;
void main()
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    vec4 vertexPos = gl_ModelViewMatrix * gl_Vertex;
    Id = vec3(gl_LightSource[0].position-vertexPos);
    normal = gl_Normal;
    gl_Position = vertexPos;
}
```

### Vertex Shader

```
varying vec3 Id, normal;
uniform sampler2D tex;
void main()
{
    vec3 lightDir = normalize(Id);
    float intensity = dot(lightDir, normal);
    vec4 color = texture2D(tex,gl_TexCoord[0].st);
    gl_FragColor = color * intensity;
}
```

### Fragment Shader



## Interne Ausführung von Vertex und Fragment-Shader

- Vertex Shader wird pro Vertex ausgeführt
- Fragment Shader wird pro Fragment (Pixel) ausgeführt
- Ausführen heisst das der gegebene Shader (C-Programm) für jeden einzelnen Vertex, sprich Fragment, ausgeführt wird
- Heutige Graphikkarten verfügen hunderte von Prozessoren (GeForce 295 mit 480 Cores)
- Daten werden parallel verarbeitet
- Für bessere Performance kann **nicht** auf andere Vertex/Fragmente innerhalb eines Shader zugegriffen werden



## Typen

- float, vec2, vec3, vec4
- int, ivec2, ivec3, ivec4
- bool, bvec2, bvec3, bvec4
- mat2, mat3, mat4
- void
- sampler1D, sampler2D, sampler3D
- samplerCube
- sampler1DShadow, sampler2DShadow

## Arrays

- Arrays ohne Pointer Arithmetik
- Grösse muss eine Integer Konstante sein
- Type und Grösse müssen zusammen definiert werden

```
float m1[10];  
float m2[3]={1.0,2.0,3.0};  
float m3[10][20];  
  
m1[9] = 1.0f;  
m3[9][19] = 10.0f;  
mat4 m16;  
m16[3][3] = 0.5;
```

## Konstruktoeren

```
vec4 myColor = vec4(1.0, 1.0, 0.0, 0.0);  
mat4 myMat = mat4(myColor, myColor, myColor, myColor);
```

## Vektor- und Matrixzugriff

```
vec2 v2;  
vec3 v3;  
v2.x // returns a float  
v4.rgba // returns a vec4  
v4.stp // returns a vec3
```



## Built-In Vertex Shader Variablen

- Attribute Inputs (Read-Only)
  - Können nur vom Vertex Shader gelesen werden
  - `gl_Vertex`, `gl_Normal`, `gl_Color`, `gl_MultiTexCoord0...`
- Varying Outputs
  - Werden interpoliert dem Fragment Shader übergeben
  - `gl_FrontColor`, `gl_TexCoord[0]...`
- Special Outputs
  - Werden an die Graphics Processing Pipeline übergeben
  - `gl_Position`, `gl_PointSize`, `gl_ClipVertex`

## Built-In Fragment Shader Variablen

- Varying Inputs (Read-Only)
  - Interpoliert Werte vom Vertex Shader
  - `gl_Color`, `gl_TexCoord[0]`, ...
- Special Input (Read-Only)
  - Selten benutzte Eingaben
  - `gl_FragCoord` (pixel Koordinate), `gl_FrontFacing`
- Special Outputs
  - Werden an die Graphics Processing Pipeline übergeben
  - `gl_FragColor`, `gl_FragDepth`, `gl_FragData[0]`...



## Built-In Konstanten, Uniforms und Funktionen

- Können von Vertex wie auch von Fragment Shadern benutzt werden
- Konstanten
  - Sind Treiber und Graphikkarten spezifisch (Umsetzung des OpenGL Standards)
  - `gl_MaxLights`, `gl_MaxTextureCoords`
- Uniforms
  - Sind nur pro Primitiv (Point, Line, Polygon) erhältlich
  - `gl_ModelViewMatrix`, `gl_ModelViewProjectionMatrix`,..
- Funktionen
  - Verschiedene erhältlich wie z.B. `cos`, `sin`, `log`, `min`, `normalize`....

## Kann meine Grafikkarte GLSL?

```
...
glewInit();
if (glewIsSupported("GL_VERSION_2_0")) {
    std::cout << "OpenGL 2.0 supported\n";
} else if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader) {
    std::cout << "GLSL ARB-extension supported\n";
} else {
    std::cout << "GLSL not supported\n";
}
```

## Wie verwende ich vertex und fragment shader

- Shader müssen kompiliert und gelinkt werden
- Shader erzeugen  
`GLuint vShader = glCreateShaderObject(GL_VERTEX_SHADER);`  
`GLuint fShader = glCreateShaderObject(GL_FRAGMENT_SHADER);`
- Shader Source setzen  
`glShaderSource(vShader, 1, vsSource, 0);`  
`glShaderSource(fShader, 1, fsSource, 0);`
- Wobei vsSource, fsSource die char\* mit dem C Shader source Code
- Shader kompilieren  
`glCompileShader(vShader);`  
`glCompileShader(fShader);`



- GLSL Programm erzeugen  
**GLuint glslProg = glCreateProgramObject();**
- Shader attachen  
**glAttachObject(glslProg, vShader);**  
**glAttachObject(glslProg, fShader);**
- Programm linken  
**glLinkProgramObject(glslProg);**
- Programm aktivieren/deaktivieren  
**glUseProgramObject(glslProg); // aktiviere Programm**  
**glUseProgramObject(0); // Fixed Funtionality wird benutzt**

## Es kann auch nur ein Shader definiert werden

- Vertex und Fragment Shader müssen nicht zusammen definiert werden
- Es kann nur ein Shader implementiert und einem GLSL Program übergeben werden
- Der andere Shader wird dann automatisch die Fixed-Functionality Pipeline implementieren

## Wie übergebe ich Werte an einem Shader? (Übersicht)

- Wichtig! : Der Shader muss aktiviert werden bevor man Werte übergeben kann
- Uniform Variablen
  - Speicher Adresse (GPU) der Variable ermitteln durch  
`GLint glGetUniformLocation(GLuint program, const char *name);`
  - Wert setzen, z.B.  
`void glUniform1f(GLint location, GLfloat v0);`
  - Für Texturen  
`void glUniform1i(GLint location, GLint v0);`
- Attribute Variablen
  - Speicher Adresse (GPU) der Variable ermitteln durch  
`GLint glGetAttribLocation(GLuint program, char *name);`
  - Wert setzen, z.B.  
`void glVertexAttrib1f(GLint location, GLfloat v0);`
- Mehr dazu in der OpenGL Spezifikation oder Orange Book



## Debugging

- Schwierig, wird aber immer einfacher...
- Selbst Code mit Debug Info anreichern.
  - switch/if/#define etc ... im Code
  - Im Fragment Shader z.B. das Fragment rot färben wenn Ereignis X eintritt ...
- Fehlerlog beim kompilieren auswerten
- Externe Debugger, z.B. RenderMonkey, gDEBugger, glslDevil

## Dokumentation

- <http://www.opengl.org/documentation/glsl/> GLSL Spezifikation
- <http://www.khronos.org/files/opengl-quick-reference-card.pdf> Referenz Karte mit allen Befehlen von OpenGL und GLSL als Übersicht
- [http://mew.cx/glsl\\_quickref.pdf](http://mew.cx/glsl_quickref.pdf) Referenz Karte nur mit GLSL Befehlen
- <http://www.opengl.org/sdk/>
  - Links to Documentation
  - Links to Tutorials
- <http://www.lighthouse3d.com/opengl/glsl/> GLSL Tutorial, am Anfang **sehr** empfehlenswert
- OpenGL 2.0 Programming Manual (Red Book)
- OpenGL Shading Language (Orange Book)
- Vendor SDKs
  - [http://developer.nvidia.com/object/sdk\\_home.html](http://developer.nvidia.com/object/sdk_home.html)
  - <http://developer.amd.com/GPU/Pages/default.aspx>



## Zu den Aufgaben

- Ein kleines Framework (GLLib) wird zur Verfügung gestellt
  - Erstellen und benutzen von GLSL Programmen bereits implementiert
  - Funktionen zum Übergeben von Werten an Shader verfügbar
- Tips
  - Benutzen Sie die `gl_FragColor` um Fehler anzuzeigen
  - Haben Sie die jeweiligen Vektoren im Shader normalisiert?
  - Was ist das Ergebnis einer Matrix Vektor Multiplikation? Werden nur 3-Komponenten gebraucht?