March 4, 2025, Brno
Author: David Procházka

Shaders

Graphic Application Development

MENDELU
Faculty
of Business
and Economics

Table of content

1. Hardware
2. Integration in OpenGL
3. GLSL
4. Summary

Vertex shader

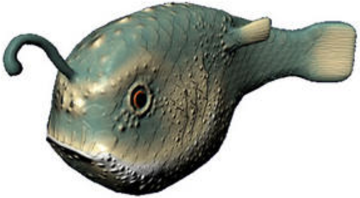
- Application that runs on shader core and processes a single vertex.
- Input values for the vertex shader are: position, color, normal etc.
- Its purpose is:
  - geometric transformations using modelview and projection matrix,
  - normals transformations and normalization,
  - transformation of texture coordinates,
  - per vertex lighting computation,
  - …
- Each vertex is processed separately, hence, it does not know its neighbors.
- Direct output from the shader is `gl_Position` value.

Pixel/Fragment shader

- Application that processes a single pixel of a given primitive.
- Its purpose is:
  - calculation of color,
  - calculation of texture from texture coordinates,
  - fog calculation,
  - per pixel lighting
  - …
- The input of the shader are values from previous step of the pipeline: interpolated color, normal, texture coordinates etc..
- Similarly to the vertex shader, it does not know any other pixel.
- The result of the shader is color stored in `gl_FragColor`, or into a general output variable (new versions).

Geometry shader

- Allow to create new vertices for complex shapes such as fur, grass etc.
- Supported from Direct3D 10 and OpenGL 3.2.
- Executed after vertex shader.



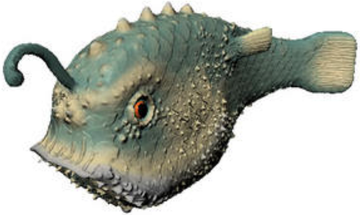
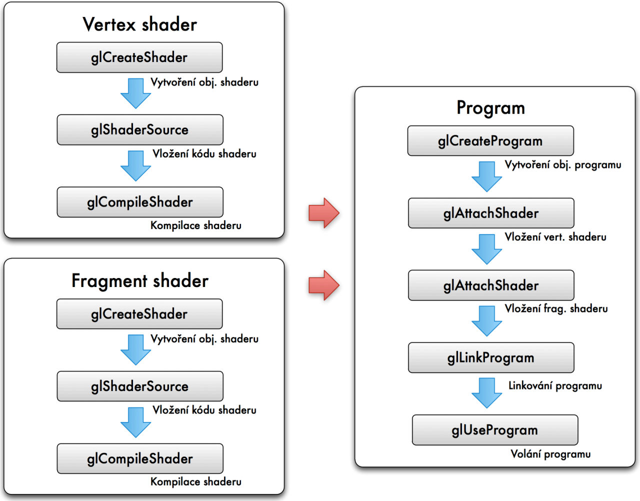


Table of content

1. Hardware
2. Integration in OpenGL
3. GLSL
4. Summary

Shader integration



Vertex shader

glCreateShader

Vytvoření obj. shaderu

glShaderSource

Vložení kódu shaderu

glCompileShader

Kompilace shaderu

Fragment shader

glCreateShader

Vytvoření obj. shaderu

glShaderSource

Vložení kódu shaderu

glCompileShader

Kompilace shaderu

Program

glCreateProgram

Vytvoření obj. programu

glAttachShader

Vložení vert. shaderu

glAttachShader

Vložení frag. shaderu

glLinkProgram

Linkování programu

glUseProgram

Volání programu

Example of a shader integration

```
Gluint v = glCreateShader(GL_VERTEX_SHADER);
Gluint f = glCreateShader(GL_FRAGMENT_SHADER);

char* vs = " // here is some vertex shader // ";
char* fs = " // here is some fragment shader // ";

glShaderSource(v, 1, vs, NULL);
glShaderSource(f, 1, fs, NULL);
```

Example of a shader integration

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

GLuint p = glCreateProgram();

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

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

There is a huge number of classes that streamlines this process!

Table of content

1. Hardware
2. Integration in OpenGL
3. GLSL
4. Summary

GLSL: How the shader looks like?

Each API has its own shader language. The OpenGL uses GLSL (GL Shader Language). It’s C-like program.

```
in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;

void main(void)
{
  gl_Position = vec4(a_Vertex, 1.0)
  color = vec4(a_Color, 1.0);
}
```

Data types

- Scalars: `float`, `(u)int`, `bool`.
- Vectors
  - `vec2`, `vec3`, `vec4` – vector 2, 3 and 4 floats,
  - `ivec2`, `ivec3`, `ivec4` – vector 2, 3 and 4 ints,
  - `uvec2`, `uvec3`, `uvec4` – vector 2, 3 and 4 un. ints,
  - `bvec2`, `bvec3`, `bvec4` – vector 2, 3 and 4 bools.
- Matrices
  - `mat2`, `mat3`, `mat4` – square matrix 2×2, 3×3 and 4×4.
  - `mat2x2`, `mat2x3`, `mat2x4`, `mat3x2` ...
- Textures (there are i..., u... versions)
  - `sampler1D`, `sampler2D`, `sampler3D` – for 1/2/3D textures,
  - `samplerCube` – for cube maps,
  - `sampler1DShadow` – for shadow maps (also in 2D),
  - `sampler1DArray` – for arrays of textures (also 2D, shadow...).

Accessing the vector components

- To access the vector values, we use standard component names
  - colors: r, g, b, a
  - coordinates: x, y, z, w
  - textures: s, t, p, q
- We can also use methods that returns some subset of values:
  - `vec3 colorWithoutAlpha = someColor.rgb;`
  - Let us have `vec4 coords`. W can call: `coords.x`, `coords.xyz`, `coords.xz`, etc.
    We cannot call: `coords.xyza`, etc.
- It can be used also to assign values:
  - `color.rgb = {1.0, 1.0, 0.0};`
- Constructors of the vectors are mighty!
  - `vec4 semitransparent = vec4s(someColor, 0.5)`, where `someColor` is `vec3`.

Variable qualifier

Qualifier	Meaning
nothing	local variable
`const`	constant

`in`
`(attribute)`
	variable from the previous step of the pipeline
	(e.g. from a program to vertex shader)

`out`
`(varying)`
	variable send to the next step
	(e.g.from vertex to fragment shader)

`uniform`	variable with same value for all vertexes/pixels
`centroid in`	same as in, but with cen. interp.
`centroid out`	same as out, but with cen. interp.

Function parameter qualifier

Qualifier	Meaning
nothing/`in`	input parameter
`out`	output parametr
`inout`	input/output parameter

Embedded functions

- Angle conversions: `radians`, `degrees`,
- Gon. func: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`,
- Power, etc.: `pow`, `exp`, `log`, `sqrt`,
- Rounding: `abs`, `floor`, `ceil`,
- Dividing: `mod`,
- Comparation and distances: `min`, `max`, `length`, `distance`,
- Vector operations: `dot`, `cross`, `normalize`,
- Texture application: `texture`.

Vertex Shader (GLSL 1.2)

Simple C-like program

```glsl
#version 120
attribute vec3 a_Vertex; // input from the previous step
attribute vec3 a_Color;
varying vec4 color;       // output for the next step

void main(void)
{
  gl_Position = vec4(a_Vertex, 1.0)
  color = vec4(a_Color, 1.0);
}
```

Vertex Shader (GLSL 1.3)

Different naming of input/output

```
#version 130

in vec3 a_Vertex;
in vec3 a_Color;
out vec4 color;

void main(void)
{
  gl_Position = vec4(a_Vertex, 1.0)
  color = vec4(a_Color, 1.0);
}
```

Fragment shader (GLSL 1.2)

```glsl
#version 120

// interpolated color from the vertex
attribute vec4 color;

void main(void) {
  // output var. up to GLSL 1.2
  gl_FragColor = color;
}
```

Fragment shader (GLSL 1.3)

```
#version 130

in vec4 color;
out vec4 outColor;

void main(void) {
  // gl_FragColor is obsolete
  outColor = color;
}
```

Sending variables to a shader

The variables can be uniform (same for multiple vertices) or an attribute of a vertex.

Sending of uniform data (will be used later)

```
// 1 == number of stored variables.
glUniformMatrix4fv(location, 1, transpose, matrix);
```

Registration of a variable for sending variables for a particular vertex (will be done after a shader is created)

```
GLuint coordId; // index/id used for sending into a shader
glBindAttribLocation(programID, coordId, coordNameInShader);
```

Can be done using different methods in libraries.

Attribute initialization

```
glEnableVertexAttribArray(coordId); //enable vertex attr.
glEnableVertexAttribArray(colorId); //enable color attr.

someObject->render(); //

glDisableVertexAttribArray(coordId); //disable vertex attr.
glDisableVertexAttribArray(colorId); //disable color attr.
```

This code replaces previously used:

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
...
```

Reading data from a vertex buffer object

Vertex attribute pointer

```
void glVertexAttribPointer(GLuint index, GLint size,
GLenum type, GLboolean normalized, GLsizei stride,
const GLvoid *pointer),
```

- index – number that identifies the attrib.,
- size – amount of data,
- type – type of data,
- normalized – is it normalized?,
- stride – distance between values,
- pointer – pointer on an array.

It replaces `glVertexPointer()` and `glColorPointer()`.

Reading data from a vertex buffer object

```
// Bind the vertex buffer
glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
// Load data into shader
glVertexAttribPointer(coordId, 3, GL_FLOAT,
                      GL_FALSE, 0, vertices);

// Bind the color buffer
glBindBuffer(GL_ARRAY_BUFFER, m_colorBuffer);
// Load data into shader
glVertexAttribPointer(colorId, 3, GL_FLOAT,
                      GL_FALSE, 0, colors);
```

Table of content

1 Hardware

2 Integration in OpenGL

3 GLSL

4 Summary

Takeaway

- What is shader and how we can pass data to a shader.
- What is GLSL and basic structure of a vertex and fragment shader.
- How to access vector components in GLSL.