matfyz

Computer Graphics Charles University

GLSL shading language

© 2016 Josef Pelikán

CGG MFF UK Praha

pepca@cgg.mff.cuni.cz

http://cgg.mff.cuni.cz/~pepca/

Content

- architecture, environment
- shader workflow
  - shaders, programs, linking
  - uniforms
- GLSL language
  - variables
  - data types
  - functions
  - recommendations

Programmable pipeline scheme

Texture Memory

Geom.

App Memory

Vertex Processor

Primitive Assembly

Clip Project Viewport Cull

Geom.

Rasterize

Pixels

Fragment Processor

Per-Fragment Operations

Vertices

Pixel Unpack

Pixel Transfer

Fragments

Frame Buffer Operations

Textures

Pixel Pack

Read Control

Frame Buffer

Pixel Groups

Environment

GLSL source code

Application

GLSL shader

GLSL program

GLSL compiler

3D API: OpenGL

GPU driver

GPU hardware

GLSL workflow

- **GLSL source:** `glShaderSource()`
  - text file
  - string in application memory
- **GLSL shader:** `glCreateShader()`
  - compiled from source: `glCompileShader()`
- **GLSL program:** `glCreateProgram()`
  - two or more shaders attached: `glAttachShader()`
  - link program: `glLinkProgram()`
  - use program: `glUseProgram()`

GLSL objects – shaders

- **Shader**
  - piece of code executable on GPU hardware in one specific context (vertex processor, fragment processor, geometry processor, ..)
  - lives in the OpenGL world (application has a handle to it)
  - compiled from one source code: `glCompileShader()`
  - single entry point: `void main ()`
  - **input:**
    - vertex attributes / input variables (“**varying**”)
    - “**uniforms**” (constant during processing of one batch)
  - **output:**
    - output variables (some of them are mandatory)

© Josef Pelikán, http://cgg.mff.cuni.cz/~pepca

Vertex shader source example

```
#version 130
in vec4 position;
in vec3 normal;
in vec2 texCoords;
in vec3 color;
out vec2 varTexCoords;
out vec3 varNormal;
out vec3 varWorld;
out vec3 varColor;
flat out vec3 flatColor;
uniform mat4 matrixModelView;
uniform mat4 matrixProjection;

void main ()
{
  gl_Position  = matrixProjection * matrixModelView * position;
  // propagated quantities:
  varTexCoords = texCoords;
  varNormal    = normal;
  varWorld     = position.xyz;
  varColor     = flatColor = color;
}
```

input – vertex attributes

optional output variables

uniforms (constants)

mandatory output

GLSL objects – programs

- **Program**
  - set of shaders able to work together on GPU hardware … **complete working pipeline**
  - **vertex shader** and **fragment shader** are mandatory
    - other shaders are optional
  - attaching an existing shader: `glAttachShader()`
    - shaders can be **shared** among different programs
  - linking a program: `glLinkProgram()`
    - check interconnecting variables ("plumbing")
  - if everything went well, a program can be used: `glUseProgram()`

Shader setup example I

- create and compile shader:

```
// shaderType – one of:
//   GL_VERTEX_SHADER
//   GL_FRAGMENT_SHADER
//   GL_TESS_CONTROL_SHADER
//   GL_TESS_EVALUATION_SHADER
//   GL_GEOMETRY_SHADER
//   GL_COMPUTE_SHADER

GLuint shader = glCreateShader( shaderType );
const char* src = “void main() { … }”;
glShaderSource( shader, 1, src, NULL );
glCompileShader( shader );
```

Shader setup example II

- error-check:

```
GLint compiled;
glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
if ( !compiled )
{
  GLint len;
  glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &len );
  std::string msgs( ' ', len );
  glGetShaderInfoLog( shader, len, &len, &msgs[0] );
  std::cerr << msgs << std::endl;
  throw shader_compile_error;
}
```

Shader setup example III

- create and link program:

```
// create an empty program:
GLuint program = glCreateProgram();

// attach required shaders to it:
glAttachShader( program, vertexShader );
glAttachShader( program, fragmentShader );

// link the program:
glLinkProgram( program );
```

Shader setup example IV

- error-check after linking:

```
GLint linked;
glGetProgramiv( program, GL_LINK_STATUS, &linked );
if ( !linked )
{
  GLint len;
  glGetProgramiv( program, GL_INFO_LOG_LENGTH, &len );
  std::string msgs( ' ', len );
  glGetProgramInfoLog( program, len, &len, &msgs[0] );
  std::cerr << msgs << std::endl;
  throw shader_link_error;
}
```

Shader linking (“plumbing”)

Application

Uniforms

Input
(attributes,
“varying”)

Shader1

Output
(“varying”)

Sha

Data compatibility !

Shader linking

- **output varying data** must be compatible with **input varying data** of the consequent shader!
  - otherwise program-linking stage will not succeed

vertex shader

```
out vec4 color;
out vec2 texCoord;
```

→ →

fragment shader

```
in vec4 color;
in vec2 texCoord;
```

Uniforms

- **Constant value** during the whole **draw-batch**
  - Microsoft calls them just "**constants**"
  - GLSL keyword: `uniform`
- **Setting value** to an uniform
  1. obtain handle: `glGetUniformLocation()`
  2. set value: `glUniform*()`, `glUniformMatrix()`, `glProgramuniform*()`, ..
  - Uniform Buffer Object

Builtin GLSL variables

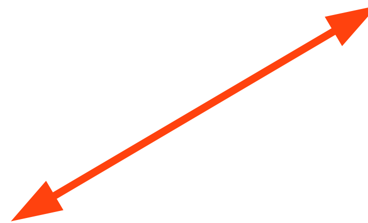
- prior to OpenGL 3.1 there were many of them !
  - .. emulation of FFP mostly ..
  - notation: `gl_*`
- now they are mostly **deprecated**
  - some of them survived: `gl_Position`
    - implicit output of all vertex-processing shaders
    - can be indexed (as an array)
  - `gl_FragCoord`, `gl_PointCoord`, `gl_FragDepth`, ..
  - indexed rendering, instanced rendering, ..
  - geometry-shading, tessellation-shading, ..

© Josef Pelikán, http://cgg.mff.cuni.cz/~pepca

Variable layouts

- defines exact position in vertex-attribute set
  - precedence over variable identifier !

```
// trivial vertex shader
#version 400 core

layout(location = 0)
  in vec4 vPos;

layout(location = 1)
  in vec3 vColor;

layout(location = 0)
  out vec3 color;

uniform mat4 MVP;

void main ()
{
  color = vColor;
  gl_Position = MVP * vPos;
}
```



```
// trivial fragment shader
#version 400 core

layout(location = 0)
  in vec3 col;

layout(location = 0)
  out vec4 fragColor;


void main ()
{
  fragColor = col;
}
```

GLSL language – data types I

- **non-vector types**
  - bool, int, uint
  - IEEE-754: **float**, double
- **vectors**
  - 2-, 3-, 4-component vectors
    - bvecN, ivecN, uvecN, **vecN**, dvecN
    - SIMD operations!
  - **swizzling**: record-like dereferencing of components

```
vec4 vecA;                 vec2 vecB = vecA.xy;
vecA.x + vecA.y;           vec4 vecC = vecA.yxzw;
vec3 vecD = vecB.xyy;      vecC.yzw = vecA.xxx;
```

GLSL language – data types II

- **matrices**
  - **matNxM**, dmatNxM
  - **matN**, dmatN (shorthand for [d]matNxN)
- **arrays**
  - type[]
    - like in C
    - no pointers!
  - type[][]
- **structs**
  - struct Name { … … } [variableName];

```
uniform sampler im[10];
uniform int n;

void main ()
{
  vec4 acc = vec4(0.0);
  for ( int i = 0; i < n; i++ )
    acc += texture( im[i], .. );
}
```

GLSL language – data types III

- **samplers**
  - sampler instance ~ one texture attached to a TU
  - only as uniforms or function parameters
  - special access modes ('g' = '' | 'i' | 'u', 'N' = 1 | 2 | 3)

`gsamplerND`	GL_TEXTURE_ND
`gsamplerCube`	GL_TEXTURE_CUBE_MAP
`gsampler2DRect`	GL_TECTURE_RECTANGLE
`gsamplerNDArray`	GL_TEXTURE_ND_ARRAY
`gsamplerBuffer`	GL_TEXTURE_BUFFER
`samplerNDShadow`	GL_TEXTURE_ND
`samplerCubeShadow`	GL_TEXTURE_CUBE_MAP
`sampler2DRectShadow`	GL_TEXTURE_RECTANGLE

...

GLSL language – functions I

- **general arithmetics & conversions**
  - abs, sign, dot, mod, modf, exp, exp2, log, log2, pow
  - frexp, ldexp, pack*, unpack*, sqrt, inversesqrt
  - step, mix, smoothstep, fma [a*b+c]
  - min, max, clamp, ceil, floor, fract, round, roundEven
  - degrees, radians
- **geometric**
  - cross, length, normalize, reflect, refract
- **goniometric & cyclometric**
  - sin, cos, tan, sinh, cosh, tanh
  - asin, acos, atan, asinh, acosh, atand

GLSL language – functions II

- **matrix**
  - transpose, inverse, outerProduct
- **comparison & logical**
  - equal, notEqual, isinf, isnan
  - not, all, any
- **texture**
  - texture, textureProj, textureGather, textureGrad, textureLod, ..
- **special & differential !**
  - noise, noiseX
  - dFdx, dFdy, … , fwidth

© Josef Pelikán, http://cgg.mff.cuni.cz/~pepca

GLSL language – control flow

- standard set of **C++ statements**
  - if-else, switch-case, loops, break, continue, return, ..
  - no “goto” statement
  - preprocessor: **#define, ##**, **#ifdef**, **#else**, **#pragma**, ..
- **function declaration**
  - no recursion!
  - parameters: **in** (default), **out**, **inout**
- **“discard”** command (fragment shader only)
  - similar to return but discards **all output generated by the fragment so far**
  - i.e. no buffer data will be modified by this fragment

GLSL recommendations

- declare your version in `#version xyy` directive
- use **swizzling** where possible
- use **MAD** (**M**ultiply then **AD**d):

```
0.5 * (1.0 + val) → 0.5 + 0.5 * val
```

```
const vec2 cList = vec2(1.0,0.0);
color = myCol.xyzw * cList.xxxy + cList.yyyx;
```

- use **builtin functions**
  - if there are not “single-cycle” today, they might be in the future
  - mix(), dot(), step(), smoothstep(), ..

GLSL misc stuff, not covered here..

- GLSL subroutines
- Tessellation & Geometry shaders
- Uniform Buffer Object
- Compute shaders
- Shader Storage Buffer Object
- Interface Block

Sources

- Tomas Akenine-Möller, Eric Haines: ***Real-time rendering, 2nd edition***, A K Peters, 2002, ISBN: 1568811829
- OpenGL ARB: ***OpenGL Programming Guide, 4th edition***, Addison-Wesley, 2004, ISBN: 0321173481
- Randi J. Rost: ***OpenGL Shading Language***, Addison-Wesley, 2004, ISBN: 0321197895
- **https://www.opengl.org/documentation/glsl/**
- **http://developer.nvidia.com/**