Week 5 : GLSL Shaders

Topics: Shader syntax, passing textures into shaders, per-pixel lighting, vertex displacement, multi-texturing

Syntax

GLSL looks like a simplified version of C.

The main addition to GLSL is vector, matrix, and texture-sampler types, and associated built-in functions and operations to those types.

floats: float, vec2, vec3, vec4, mat2, mat3, mat4

ints: int, ivec2, ivec3, ivec4

booleans: bool, bvec2, bvec3, bvec4

textures: sampler1D, sampler2D, sampler3D, sampler1DShadow, sampler2DShadow

Syntax

```
vec3 pos = vec3(0.0, 0.0, -0.5);        = [0.0. 0.0, -0.5]

vec4 v = vec4(0.5);                     = [0.5, 0.5, 0.5, 0.5]

mat2 m = mat2(1.0, 2.0, 3.0, 4.0);      = [1.0 3.0
                                           2.0 4.0]

mat4 identity = mat4(1.0);              = [1.0 0.0 0.0 0.0
                                           0.0 1.0 0.0 0.0
                                           0.0 0.0 1.0 0.0
                                           0.0 0.0 0.0 1.0]
```

Indexing

You can index into a vector or matrix type using array notation:

```
vec3 v = vec3(1.0, 2.0, 3.0);          = [1.0, 2.0, 3.0]
float val = v[2];                      = [3.0]

mat2 m = mat2(1.0, 2.0, 3.0, 4.0)      = [1.0 3.0
                                          2.0 4.0]
vec2 v2 = m[1] //gets the 2nd column   = [3.0 4.0]
```

Swizzling

The vector types can be “swizzled” using any of these conventions

x,y,z,w (usually indicates position)
r,g,b,a (usually indicates color)
s,t,p,q (usually indicates texture coordinates)

to refer to the 1st, 2nd, 3rd, or 4th component of the vector.

For instance:

```
vec4 vecA = vec4(1.0, 2.0, 3.0, 4.0);     = [1.0, 2.0, 3.0, 4.0]
vec2 vecB = vecA.xz;                      = [1.0, 3.0];
vec3 vecC = vecB.grg;                     = [3.0, 1.0, 3.0];
```

Operations

Operations are the same as in C, except for a few special cases for vector and matrix types.

In general, any operation to a vector or matrix is component-wise...

Except for multiplication between a vector and a matrix, or between two matrices:

```
vec4 v, u; mat4 m;

vec4 v1 = v * u;  //v1 = [v.x * u.x; v.y * u.y; v.z * u.z; v.w * u.w]
vec4 v2 = v * m; //vector first treats v like a row-vector when multiplying
vec4 v3 = m * v; //matrix first treats v like a column-vector when multiplying
mat4 m1 = m * m; //performs a matrix multiplication
```

Qualifiers

There are also special global “qualifiers” that need to be attached to the variables when communicating between OpenGL and a GLSL program or between a vertex shader and a fragment shader.

“uniform” = read-only global variable accessible to the entire primitive during the binding of the program.

“attribute” = vertex specific variable accessible to each vertex in the primitive within the vertex shader

“varying” = write-only variable created in the vertex shader and interpolated across each vertex, read-only accessible within the fragment shader

Built-in functions

GLSL has a number of built-in functions that operate on the vector and matrix data types. Most of these functions are hardware accelerated.

Refer to the reference page linked to on the syllabus, but they include:

common math functions, trigonometry, exponential, geometric, derivatives (for fragments), lighting calculations (reflection, refraction), matrix functions, texture access functions (for Sampler data types), noise functions (although noise functions are NOT hardware accelerated on my graphics card!)

Built-in variables

There are also a number of built-in variables available to the shaders.

Vertex attributes (available to the vertex shaders):

gl_Color, gl_Normal, gl_Vertex, gl_MutliTexCoord0 → 16 (or however many are textures are available for simultaneous processing on your card)

Uniform variables (available to both vertex and fragment shaders):

modelview and projection info:

gl_ModelViewMatrix, gl_ProjectionMatrix, gl_ModelViewProjectionMatrix, gl_NormalMatrix, etc

light and material info:

gl_ModelViewMatrixInverse, gl_FrontMaterial, gl_BackMaterial, gl_LightSource[]

Varying variables (automatically passed to the fragment shader):

gl_Color, gl_TexCoord[]

Vertex shader

the information passed from the vertex shader to the fragment shader is automatically interpolated across the geometry.

In the fixed-functionality pipeline, this happens automatically with the position, the color, the normals, and the texture coordinates.

In the vertex shader you can create your own interpolated variables by defining and writing to a special type of data called “varying”

At the very least, you must specify a built-in varying variable called “gl_Position” which indicates how the geometry is projected onto the 2D screen.

The vertex shader has access to the modelview matrix and the projection matrix, or you can call ftransform() to automatically mimic the fixed-functionality pipeline.

Vertex shaders are used for particle effects, deformations, etc

Fragment shader

The fragment shader ultimately defines the color of every pixel of the geometry.

You use a combination of global ("uniform") and interpolated ("varying") data to determine how to color the pixels.

Fragment shaders commonly are used for lighting models, image processing techniques, blurring, glow filters, light scattering effects, etc.

Fragment shaders run for every pixel, whereas vertex shaders run only once per vertex. So if possible, put the necessary logic in the vertex shader. In some cases, the automatic interpolation of values may not be sufficient, and some or all logic may have to be placed in the fragment shader.

Installing a shader program

1. Create a shader program
```
programID = glCreateProgram();
```

2. Create the vertex and fragment shaders
```
vertexID = glCreateShader(GL_VERTEX_SHADER);
fragmentID = glCreateShader(GL_FRAGMENT_SHADER);
```

3. Load source code from a file
```
glShaderSource(vertexID, numLines, vertexSource, lineLengths, 0);
glShaderSource(fragmentID, numLines, fragmentCode, lineLengths, 0);
```

4. Compile the shader
```
glCompileShader(vertexID);
glCompileShader(fragmentID);
```

5. Attach the shaders to the program
```
glAttachShader(programID, vertexID);
glAttachShader(programID, fragmentID);
```

6. Link the program to the OpenGL context
```
glLinkProgram(programID);
```

Using a shader program

1. Bind the program within your display loop to bypass fixed-functionality

```
glUseProgram(programID);
```

2. Pass “uniform” data into the shaders

```
uniformID = glGetUniformLocation(programID, variableName);
glUniform...(uniformID, values...);
```

3. Pass texture data into the shaders

```
glEnable(GL_TEXTURE_2D);
glActiveTexture(GL_TEXTURE#);
glBindTexture(GL_TEXTURE_2D, textureID);
uniformID = glGetUniformLocation(programID, textureVariableName);
glUniform1i(textureVariableName, #);
```

3. Draw geometry (passing in “attribute” data if necessary)

```
attributeID = glGetAttribLocation(programID, variableName);
glBegin(GL_POINTS);
  glVertexAttrib...(attributeID, values...);
  glVertex3f(x, y, z);
glEnd();
```

4. Unbind the program to return to fixed-functionality pipeline

```
glUseProgram(0);
```

OpenGL→GLSL data types

Uniform data = Data that is global to the entire shader program

e.g. textures, timestamp, counters, blending value, states, filter kernels

Vertex Attribute data = Data that is specific to a particular vertex

e.g. position offsets, color offsets, texture coordinates

glColor, glNormal, glVertex, glTexCoord are all automatically available

GLSL data types

Uniform data = global data passed in from OpenGL, read-only and available in the vertex shaders and the fragment shaders

Attribute data = vertex specific data passed in from OpenGL, read-only, available only in the vertex shader

Varying data = data interpolated between vertices, writable in the vertex shader and read-only in fragment shader

Simple example

This example mimcs fixed-functionality with no lighting and no texturing.

simple.vert //passes the position to the fragment shader

```
void main()
{
    gl_Position = ftransform(); //gl_Position must ALWAYS be set
}
```

simple.frag //interpolates the colors of the vertices

```
void main()
{
    gl_FragColor = gl_Color; //gl_FragColor must ALWAYS be set
}
```

Demos

Basic

Per-Vertex lighting w/material properties

comparison of fixed functionality to Per-Vertex

Per-Pixel version

Vertex Displacement

Multi-texturing

--next class: bump mapping, shadows, environment mapping, light scattering
--class after that: research into your own shaders