Chapter 4. OpenGL Shading Language

Marc Olano

A Brief OpenGL Shading Tutorial

Marc Olano
University of Maryland, Baltimore County

1 Overview

There are two main pieces to using the OpenGL Shading language in an application, the language itself, and the portions of the OpenGL API that control that language. The language is similar in many respects to the other real-time shading languages presented here, Cg and HLSL, though we will highlight some of the similarities and differences. It is in the API that the difference is most evident.

2 The OpenGL API

As of OpenGL 2.0, the shading language features are a required part of the core API. If you have OpenGL 2.0, you need not worry about whether your particular card and driver support the shading language — they must. In OpenGL 1.5, the shading language features were an ARB-approved optional extension. The OpenGL ARB is the standards body that determines what is *officially* part of OpenGL, as compared to vendor-defined extensions that don't need ARB approval. As an ARB-approved extension, you know that anyone who does support it will support the same interface and language [Lichtenbelt and Rost 2004]. You can check if an OpenGL 1.5 driver supports the shading language by checking for the `GL_ARB_shader_objects` extension string in the `glGetString()` results.

We will present the OpenGL 2.0 versions of the calls. In most cases, the `ARB_shader_object` versions just add an ARB extension, so functions like `glCreateShaderObject()` become `glCreateShaderObjectARB()` and symbols like `GL_VERTEX_SHADER` become `GL_VERTEX_SHADER_ARB`. The definitive source for the OpenGL 2.0 API is the specification [Segal and Akeley 2004]. The OpenGL specification used to be a rather difficult document to use for all but the most determined, but as a searchable PDF, it is actually not too difficult to use it directly as a reference.

The main difference between OpenGL and the other languages is that the OpenGL shading language and compiler are built directly into the OpenGL API and provided by the graphics card vendor as part of their driver. There is no intermediate low-level code to get from the compiler and hand to the API. This has some potential hidden advantages as well. The instruction sets actually executed by the graphics hardware already different from the standard instructions sets. For example, even if the hardware can execute up to four independent arithmetic operations in a single 'instruction', the low-level instruction sets only support executing

the same operation on the same data. A final optimization on the low-level code can find and fix these, but the high-level language compiler is left to try to match a pattern that the low-level optimizer will recognize and fix. There's some chance this will work if both come from your hardware vendor, but it's easy to get non-optimal code. By hiding any low-level translation, the shading compiler built into OpenGL is free to target whatever the hardware really does, and the hardware vendors are more free to change it later to improve their hardware without worrying about matching that hardware to the language. Note that Direct3D is moving to this model as well [Blythe 2004].

2.1 The Interface

The OpenGL shading interface has two important concepts, *shader objects* and *program objects*. Shader objects hold a single high-level (vertex or fragment) shader, while program objects contain the collection of shader objects used to render an object. Shader types for other future kinds of shaders have not yet been defined (e.g. for geometry shaders [Blythe 2004]), but would fit cleanly into this interface.

To create, load from a single C-style string, and compile a vertex and fragment shader object, you could use code like this:

```
vert = glCreateShaderObject(GL_VERTEX_SHADER);
glShaderSource(vert, 1,(const GLcharARB**)&vString,NULL);
glCompileShader(vert);

frag = glCreateShaderObject(GL_FRAGMENT_SHADER);
glShaderSource(frag, 1,(const GLcharARB**)&fString,NULL);
glCompileShader(frag);
```

These two can be joined into a single program object like this:

```
prog = glCreateProgramObject();
glAttachObject(prog,vert);
glAttachObject(prog,frag);
glLinkProgramObject(prog);
```

Whenever you want to use this set of shaders on an object, you just tell OpenGL that you want to start using it.

```
glUseProgramObject(prog);
```

2.2 Alternatives for Loading

The interface for loading shader source is quite flexible, allowing many variations for how your program handles its shaders. The actual parameters are

```
glShaderSource(object,segments,stringArray,lengthArray)
```

The arrays (containing `segments` entries; one in the example above) allow shaders to be stored as a set of lines or code segments, which need not be null-terminated. If any length is -1, OpenGL assumes that segment is a null-terminated string and computes the length. If the pointer to the length array is NULL, OpenGL assumes every segment in the array is null-terminated.

Having an array of lengths means the segments need **not** be null-terminated if that isn't convenient. In particular, if you have the ability to map a file directly to memory, you could use code similar to this:

```
fd = open(vsname.c_str(), O_RDONLY, 0);
fstat(fd,&sb);
sh = (char*)mmap(0,sb.st_size,PROT_READ,MAP_FILE,fd,0);
glShaderSource(vert, 1, (const GLcharARB**)&sh,
               (const GLint*)&sb.st_size);
```

This code relies on `fstat()`, which reports information on a file, including its size, and `mmap()`, which maps a file directly into memory, potentially more efficiently than reading it in.

2.3 Compile Errors

The above code may be fine for production use, once the shaders are known to be error-free, but no one is capable of writing error-free code every time. After the `glCompileShader()`, you can find out if the shader compiled successfully with

```
glGetShaderiv(vert, GL_COMPILE_STATUS, &result);
```

`result` will be true (non-zero) if the shader compiled successfully. To find out what is wrong with a shader that didn't compile, check the info log:

```
glGetShaderInfoLog(vert, bufsize, NULL, buffer);
```

Similarly, `glGetProgramiv()` and `glGetProgramInfoLog()` can tell you about the success (or failure) of the progam linking step.

2.4 Shading Parameters

Of course, once you can load and use a shader, you still need a means to control it. Since the OpenGL shading language is built into OpenGL, every vertex shader has access to all of the usual OpenGL vertex attributes (position, color, normal, etc.), and all shaders have access to the built-in OpenGL state (light positions, matrices, etc.). Shaders access all of these using special pre-defined names in the shading language. However, any shader can also define additional attributes and state that it would like to use. You can find out the *index* for a per-vertex attribute with

```
index = glGetAttribLocation(prog, "vertexAttribute");
```

and set a value using one of the `glVertexAttrib*()` functions. For example

```
glVertexAttrib3f(index, 0,.5,1);
```

All vertex attributes, whether built-in ones like `glColor` or user-defined must be set before the corresponding `glVertex` call.

A similar set of functions exist for any user-defined *uniform* state. The uniform state affects the current program, so switch programs before you start setting the state.

```
index = glGetUniformLocation(prog, "uniformVariable");
glUniform4f(index, .2,.4,.6,1);
```

The string name used to identify a variable can be more than the variable name alone. It can include array indexing and structure dereferencing operations to allow you to set a single element. For example,

```
index = glGetUniformLocation(prog,
    "structArray[2].element");
glUniformMatrix4fv(index, 1,0, matrix);
```

3 Shading Language

Many excellent references exist for the OpenGL Shading Language exist, so this document will not attempt to exhaustively list every feature. For more details, refer to one of the other sources [Kessenich et al. 2004; Rost 2004]. Many of the features of the OpenGL shading language, are similar if not identical to the other shading language options. All have you write vertex and fragment/pixel shaders (as opposed to the RenderMan model of displacement, surface, light, volume and imager). All inherit many syntactic features from C, including `if`, `while`, `for`, the

use of "{", "}", and ";", and the existence of structs and arrays for grouping data. They also share certain features of all shading languages (even non-real-time languages like RenderMan), in having small vectors and matrices as built-in types, and a common set of math and shading functions. Like other real-time languages, screen-space derivatives are available (through the `dFdx()` and `dFdy()` functions), a struct-like notation is used for swizzling vector components and writing to only specific vector components. For example

```
vec2.xzw = vec1.yyw;
```

assigns `vec1`'s y value to `vec2`'s x and z component and `vec1`'s w value to `vec2`'s w component. `vec2`'s z component keeps whatever value it had before the assignment.

3.1 Notable Differences

Probably the first difference you'd notice between the OpenGL Shading Language and either Cg or HLSL is the important part that *virtualization* plays in the OpenGL language philosophy. If all of the programs running on your CPU exceed the available physical memory, the operating system can make it seem as if you have a much larger pool of *virtual memory* by swapping some stuff off to disk. It's not as fast switching between applications as if you had a larger pool of physical memory, but in most instances it's **much** better than crashing or not letting you switch back and forth between two applications.

Similarly, the OpenGL shading language defines some minimum features (and some features like instruction count for which there are no defined limits). A working OpenGL Shading Language implementation is required to make it seem as if you're running on that ideal hardware. It may switch to running multiple passes, it may run some things in software, but they will always run. This means one code base may run in a two passes on one machine, in three on another, or in one on a future machine that didn't even exist when you wrote the shader. Splitting shaders into multiple passes is hard to do by hand, especially when you are writing high-level code, so it makes much more sense to let the shading compiler use one of the multi-passing compilation techniques [Foley et al. 2004; Riffel et al. 2004] than for you to try to do it twelve different ways by hand.

The second notable difference is that in the OpenGL Shading Language, vertex to fragment communication is determined by the vertex shader writing to a varying variable and the fragment shader using it. The compiler chooses the *interpolator* to use. From the language standpoint, it's just data.

There are other minor differences in the names of some of the data types (OpenGL's `vec4` vs. Cg or HLSL's `float4`). Those are usually pretty obvious and easy to translate from one to the other. One that may catch more users is that in OpenGL, `matrix*vector` is a matrix/vector product and `matrix*matrix` is a linear algebraic matrix product, as compared to Cg where you use the `mul()` function and `matrix*matrix` gives a component-wise multiply (there's a function for that in OpenGL). They're the same operations, just with syntax that differs in a way that may surprise the unsuspecting.

4 Example

This example shows a simple single-light diffuse shading computed per vertex and applied to a 3D noise fragment shader

Vertex shader:

```
// noise input to fragment shader
varying vec3 Nin;

void main()
{
    // transform vertices into projection space
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;

    // vertex in view space for lighting
    vec4 viewPos = gl_ModelViewMatrix*gl_Vertex;

    // normalized normal, also in view space
    vec3 nn = normalize(gl_NormalMatrix*gl_Normal);

    // handle directional and point lights together
    vec3 nl =
        normalize(gl_LightSource[0].position.xyz*viewPos.w
            - viewPos.xyz*gl_LightSource[0].position.w);

    // add ambient and diffuse lighting
    gl_FrontColor =
        gl_FrontMaterial.ambient*gl_LightSource[0].ambient
       +gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse
            * max(0,dot(nn,nl));

    // compute scaled noise input
    Nin = gl_Vertex.xyz/10;
}
```

Fragment Shader:

```
// vertex to fragment communication for noise shaders
varying vec3 Nin;

// 2D noise texture
uniform sampler2D ntex;

// modulus for random hash
const float modulus = 61;

void
main()
{
    // integer and fractional components of input
    float fracArg = fract(modulus*Nin.z);
    float intArg = floor(modulus*Nin.z);

    // hash z & z+1 to get offsets for noise slices
    vec2 hash = mod(intArg,modulus);
    hash.y = hash.y+1;
    hash = mod(hash*hash,modulus);
    hash = hash/modulus;

    // look up noise and blend slices
    vec2 g0, g1;
    g0 = texture2D(ntex, vec2(Nin.x,Nin.y+hash.x)).ra*2-1;
    g1 = texture2D(ntex, vec2(Nin.x,Nin.y+hash.y)).ra*2-1;
    float noise = mix( g0.x+g0.y*fracArg,
                       g1.x+g1.y*(fracArg-1),
                       smoothstep(0,1,fracArg));

    // combine with lighting
    gl_FragColor = (noise*.5+.5)*gl_Color;
}
```

The results of this vertex and fragment shader is shown in Figure 1.

a b c

Figure 1. a) red component of texture. b) alpha component of texture. c) Teapot rendered with resulting appearance

References

[Blythe 2004] David Blythe, "Windows Graphics Foundation", WinHEC 2004 presentation, May 2004.

[Foley et al. 2004] Tim Foley, Mike Houston and Pat Hanrahan, "Efficient Partitioning of Fragment Shaders for Multiple-Output Hardware", Proceedings of Eurographics/SIGGRAPH Graphics Hardware 2004, July 2004.

[Kessenich et al. 2004] John Kessenich, Dave Baldwin and Randi Rost, *The OpenGL Shading Language*, Language Version 1.10, OpenGL ARB, April 2004.

[Lichtenbelt and Rost 2004] Barthold Lichtenbelt and Randi Rost, "ARB_shader_objects", OpenGL extension document, OpenGL ARB, June 2004.

[Riffel et al. 2004] Andrew T. Riffel, Aaron E. Lefohn, Kiril Vidimce, Mark Leone and John D. Owens, "Mio: Fast Multipass Partitioning via Priority-Based Instruction Scheduling", Proceedings of Eurographics/SIGGRAPH Graphics Hardware 2004, July 2004.

[Rost 2004] Randi Rost, *OpenGL Shading Language*, Addison Wesley, 2004.

[Segal and Akeley 2004] Mark Segal and Kurt Akeley, *The OpenGL® Graphics System: A Specification*, Version 2.0, Editors Jon Leech and Pat Brown, OpenGL ARB, October 2004.