OpenGL® Shading Language

Third Edition

Randi J. Rost • Bill Licea-Kane

With Contributions by Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen

Praise for *OpenGL® Shading Language*

"As the 'Red Book' is known to be the gold standard for OpenGL, the 'Orange Book' is considered to be the gold standard for the OpenGL Shading Language. With Randi's extensive knowledge of OpenGL and GLSL, you can be assured you will be learning from a graphics industry veteran. Within the pages of the second edition you can find topics from beginning shader development to advanced topics such as the spherical harmonic lighting model and more."

—David Tommeraasen
CEO/Programmer
Plasma Software

"This will be the definitive guide for OpenGL shaders; no other book goes into this detail. Rost has done an excellent job at setting the stage for shader development, what the purpose is, how to do it, and how it all fits together. The book includes great examples and details, as well as good additional coverage of 2.0 changes!"

—Jeffery Galinovsky
Director of Emerging Market
Platform Development
Intel Corporation

"The coverage in this new edition of the book is pitched just right to help many new shader-writers get started, but with enough deep information for the 'old hands.'"

—Marc Olano
Assistant Professor
University of Maryland

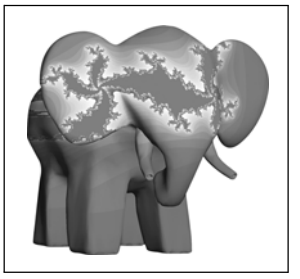
"This is a really great book on GLSL—well written and organized, very accessible, and with good real-world examples and sample code. The topics flow naturally and easily, explanatory code fragments are inserted in very logical places to illustrate concepts, and, all in all, this book makes an excellent tutorial as well as a reference."

—John Carey
Chief Technology Officer
C.O.R.E. Feature Animation

This page intentionally left blank

Chapter 7

OpenGL Shading Language API



In support of the OpenGL Shading Language, more than 40 entry points were added to OpenGL starting in version 2.0. This set of API calls is referred to throughout this book as the OPENGL SHADING LANGUAGE API. In this chapter, we look at the OpenGL entry points that have been added to create, load, compile, and link shaders, as well as the entry points that have been added for passing generic vertex attributes and uniform variables to shaders. Reference pages for all of the OpenGL Shading Language API entry points are found in Appendix B.

At the end of this chapter, we discuss the application code that is needed to create and use the brick shader presented in Chapter 6. If you just can't wait, go ahead and sneak a peek at Section 7.13, and then come back here to learn the details of the API.

Here is an overview of creating and using OpenGL shaders:

1. Create one or more (empty) shader objects with **glCreateShader**.
2. Provide source code for these shaders with **glShaderSource**.
3. Compile each of the shaders with **glCompileShader**.
4. Create a program object with **glCreateProgram**.
5. Attach all the shader objects to the program object with **glAttachShader**.
6. Link the program object with **glLinkProgram**.
7. Install the executable program as part of OpenGL's current state with **glUseProgram**.

8. If the shaders uses the default uniform block, query the locations of uniform variables with **glGetUniformLocation** and then set their values with **glUniform**.
9. If the shaders use named uniform blocks, query the uniform block information with **glGetUniformBlockIndex** and **glGetActiveUniformBlockiv**. Query the offsets of the uniform variables within the uniform block with **glGetActiveUniformsiv**. The uniform blocks are associated with buffer object binding points with **glUniformBlockBinding**, and buffer objects are bound to those binding points with **glBindBufferRange**.
10. If the vertex shader uses user-defined in variables, the application must provide values for them, using OpenGL API calls that place attribute values in generic, numbered vertex attribute locations. Before such attribute data is passed to the shader, the index of the generic vertex attribute should be associated with an in variable in a vertex shader in one of two ways. Applications can create this association explicitly by calling **glBindAttribLocation** before linking. Alternatively, if no explicit association is made, OpenGL makes these associations automatically during linking. An application can query the assignment that was made with **glGetAttribLocation**. Thereafter, generic vertex attributes can be passed to a vertex shader with **glVertexAttrib** or with **glVertexAttribPointer** and **glEnableVertexArrayPointer** in conjunction with standard OpenGL commands to draw vertex arrays.
11. If the fragment shader uses user-defined out variables, the application may create an association between a user-defined out variable and a fragment data index with **glBindFragDataLocation** before linking. Alternatively, if no explicit association is made, OpenGL makes these associations automatically during linking. An application can query the assignment that was made with **glGetFragDataLocation**.
12. An application may optionally record vertex shader out variables to one or more transform feedback buffers. Before vertex shader out variables are recorded, the vertex shader out variables are associated with feedback buffer binding points with **glTransformFeedbackVarying**, and buffer objects are bound to those binding points with **glBindBufferRange**.

7.1 Obtaining Version Information

With the addition of the OpenGL Shading Language, the OpenGL version number was changed from 1.5 to 2.0. The number before the period is

referred to as the MAJOR VERSION NUMBER, and the number after the period is referred to as the MINOR VERSION NUMBER. This did not reflect a change in compatibility, as is often the case when a product's major version number is changed. Instead, the OpenGL ARB believed that inclusion of a high-level shading language was a major addition to OpenGL. To call attention to this important capability, the committee decided that a change to OpenGL's major version number was warranted.

This caused some incompatibility with applications that were written assuming that OpenGL would never have a major version number greater than 1. The OpenGL Shading Language also has a version number since it is expected that it too will have additional features added over time. Both of these values can be queried with the OpenGL function **glGetString**.

To write applications that will work properly in a variety of OpenGL environments and that will stand the test of time, be sure to properly query and parse the OpenGL and OpenGL Shading Language version strings. Both strings are defined as

<version number><space><vendor-specific information>

The version number is defined to be either

majorVersionNumber.minorVersionNumber

or

majorVersionNumber.minorVersionNumber.releaseNumber

where each component contains one or more digits. The vendor specification information and the release number are optional and might not appear in the version string. The version number is not a floating-point number, but a series of integers separated by periods.

To determine the OpenGL version number, call **glGetString** with the symbolic constant GL_VERSION. To determine the OpenGL Shading Language version, call **glGetString** with the symbolic constant GL_SHADING_LANGUAGE_VERSION. The shading language version that was approved at the time OpenGL 2.0 was approved was 1.10.

Listing 7.1 contains code for C functions that query and parse the OpenGL and OpenGL Shading Language version strings. Both functions assume that a valid OpenGL context already exists, and both return 0 for the major and minor number if an error is encountered. Values returned by these functions can be tested to see if the underlying implementation provides the necessary support.

Listing 7.1 C functions to obtain OpenGL and OpenGL Shading Language version information

```
void getGlVersion(int *major, int *minor)
{
    const char *verstr = (const char *) glGetString(GL_VERSION);
    if ((verstr == NULL) || (sscanf(verstr,"%d.%d", major, minor) != 2))
    {
        *major = *minor = 0;
        fprintf(stderr, "Invalid GL_VERSION format!!!\n");
    }
}

void getGlslVersion(int *major, int *minor)
{
    int gl_major, gl_minor;
    getGlVersion(&gl_major, &gl_minor);

    *major = *minor = 0;
    if(gl_major == 1)
    {
        /* GL v1.x can provide GLSL v1.00 only as an extension */
        const char *extstr = (const char *) glGetString(GL_EXTENSIONS);
        if ((extstr != NULL) &&
            (strstr(extstr, "GL_ARB_shading_language_100") != NULL))
        {
            *major = 1;
            *minor = 0;
        }
    }
    else if (gl_major >= 2)
    {
        /* GL v2.0 and greater must parse the version string */
        const char *verstr =
            (const char *) glGetString(GL_SHADING_LANGUAGE_VERSION);

        if((verstr == NULL) ||
            (sscanf(verstr, "%d.%d", major, minor) != 2))
        {
            *major = *minor = 0;
            fprintf(stderr,
                "Invalid GL_SHADING_LANGUAGE_VERSION format!!!\n");
        }
    }
}
```

7.2 Creating Shader Objects

The design of the OpenGL Shading Language API mimics the process of developing a C or C++ application. The first step is to create the source code. The source code must then be compiled, the various compiled modules must be linked, and finally the resulting code can be executed by the target processor.

To support the concept of a high-level shading language within OpenGL, the design must provide storage for source code, compiled code, and executable code. The solution to this problem is to define two new OpenGL-managed data structures, or objects. These objects provide the necessary storage, and operations on these objects have been defined to provide functionality for specifying source code and then compiling, linking, and executing the resulting code. When one of these objects is created, OpenGL returns a unique identifier for it. This identifier can be used to manipulate the object and to set or query the parameters of the object.

The first step toward utilizing programmable graphics hardware is to create a shader object. This creates an OpenGL-managed data structure that can store the shader's source code. The command to create a shader is

---

GLuint **glCreateShader**(GLenum *shaderType*)

Creates an empty shader object and returns a nonzero value by which it can be referenced. A shader object maintains the source code strings that define a shader. *shaderType* specifies the type of shader to be created. Two types of shaders are supported. A shader of type GL_VERTEX_SHADER is a shader that runs on the programmable vertex processor; it replaces the fixed functionality vertex processing in OpenGL. A shader of type GL_FRAGMENT_SHADER is a shader that runs on the programmable fragment processor; it replaces the fixed functionality fragment processing in OpenGL.

When created, a shader object's GL_SHADER_TYPE parameter is set to either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER, depending on the value of *shaderType*.