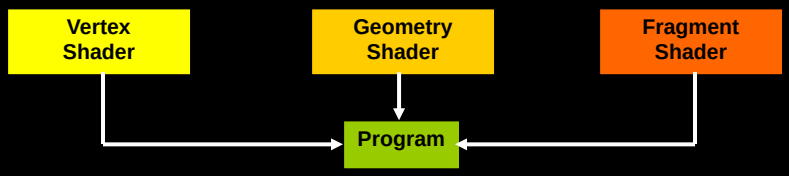
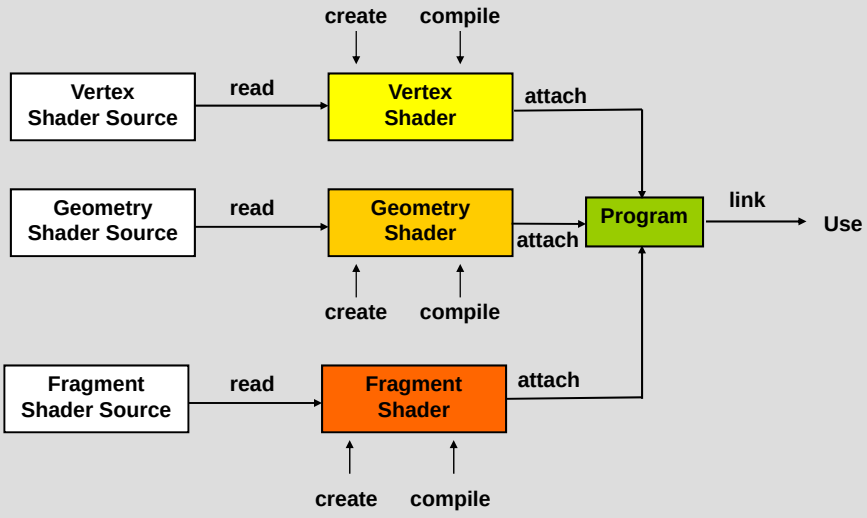
The GLSL API

Mike Bailey

Oregon State University



The GLSL Shader-creation Process



mjb – May 19, 2009

Initializing the GL Extension Wrangler (GLEW)

```
#include "glew.h“

. . .

GLenum err = glewInit();
if( err != GLEW_OK )
{
        fprintf( stderr, "glewInit Error\n" );
        exit( 1 );
}

fprintf( stderr, "GLEW initialized OK\n" );
fprintf( stderr, "Status: Using GLEW %s\n", glewGetString(GLEW_VERSION) );
```

http://glew.sourceforge.net

mjb – May 19, 2009

Reading a Vertex, Geometry, or Fragment Shader source file into a character array

```
#include <stdio.h>

FILE *fp = fopen( filename, “r” );
if( fp == NULL )  { . . . }

fseek( fp, 0, SEEK_END );
int numBytes = ftell( fp );     // length of file

GLchar * buffer = new GLchar [numBytes+1];

rewind( fp );          // same as:  “fseek( in, 0, SEEK_SET )”

fread( buffer, 1, numBytes, fp );
fclose( fp );
buffer[numBytes] = ‘\0‘;     // the entire file is now in a byte string
```

mjb – May 19, 2009

Creating and Compiling a Vertex Shader from that character buffer (Geometry and Fragment files work the same way)

Only part of this process specific to the type of shader it is

```
int status;
int logLength;

GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );

glShaderSource( vertShader, 1, (const GLchar **)&buffer, NULL );
delete [ ] buffer;
glCompileShader( vertShader );
CheckGlErrors( "Vertex Shader 1" );

glGetShaderiv( vertShader, GL_COMPILE_STATUS, &status );
if( status == GL_FALSE )
{
        fprintf( stderr, "Vertex shader compilation failed.\n" );
        glGetShaderiv( vertShader, GL_INFO_LOG_LENGTH, &logLength );
        GLchar *log = new GLchar [logLength];
        glGetShaderInfoLog( vertShader, logLength, NULL, log );
        fprintf( stderr, "\n%s\n", log );
        delete [ ] log;
        exit( 1 );
}
CheckGlErrors( "Vertex Shader 2" );
```

2009

How does that array of strings thing work?

```
GLchar *ArrayOfStrings[3];
ArrayOfStrings[0] = "#define SMOOTH_SHADING";
ArrayofStrings[1] = " . . . a commonly-used procedure . . . ";
ArrayofStrings[2] = " . . . the real vertex shader code . .. ";
glShaderSource( vertShader, 3, ArrayofStrings, NULL );
```

These are two ways to provide a *single* character buffer:

```
GLchar *buffer[1];
buffer[0] = " . . . the entire shader code . . . ";
glShaderSource( vertShader, 1, buffer, NULL );
```

```
GLchar *buffer = " . . . the entire shader code . . . ";
glShaderSource( vertShader, 1, (const GLchar **)&buffer, NULL );
```

mjb – May 19, 2009

Why use an array of strings as the shader input, instead of just a single string?

- You can use the same shader source and insert the appropriate #defines at the beginning
- You can insert a common header file (≈ a .h file)
- You can simulate a #include to re-use common pieces of code

If-tests versus preprocessing

```
if( Mode == SmoothShading )
{ . . . }
else if( Mode == PhongShading )
{ . . . }
```

```
#ifdef SMOOTH_SHADING
{ . . . }
#endif

#ifdef PHONG_SHADING
{ . . . }
#endif
```

mjb – May 19, 2009

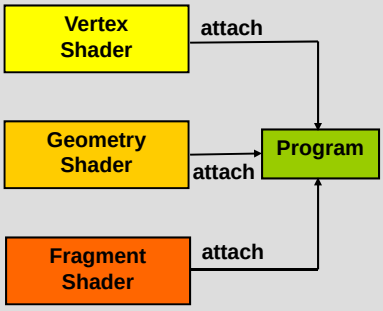
Creating the Program and Attaching the Shaders to It

```
GLuint program = glCreateProgram( );

glAttachShader( program, vertShader );

glAttachShader( program, fragShader );

glAttachShader( program, geomShader );
```



mjb – May 19, 2009

Linking the Program and Checking its Validity

```
glLinkProgram( program );
CheckGlErrors( "Shader Program 1" );
glGetProgramiv( program, GL_LINK_STATUS, &status );
if( status == GL_FALSE )
{
        fprintf( stderr, “Link failed.\n” );
        glGetProgramiv( program, GL_INFO_LOG_LENGTH, &logLength );
        log = new GLchar [logLength];
        glGetProgramInfoLog( program, logLength, NULL, log );
        fprintf( stderr, “\n%s\n”, log );
        delete [ ] log;
        exit( 1 );
}
CheckGlErrors( "Shader Program 2" );

glValidateProgram( program );
glGetProgramiv( program, GL_VALIDATE_STATUS, &status );
fprintf( stderr, “Program is %s.\n”, status == GL_TRUE ? “valid” : “invalid” );
```

mjb – May 19, 2009

Making the Program Active

```
glUseProgram(  program  );
```

This is now an “attribute”, i.e., this shader combination is in effect until you change it

Making the Program Inactive (use the fixed function pipeline instead)

```
glUseProgram(  0  );
```

mjb – May 19, 2009

Passing in Uniform Variables

```
float lightLoc[3] = { 0., 100., 0. };

GLint location = glGetUniformLocation( program, “LightLocation” );

if( location < 0 )
        fprintf( stderr, “Cannot find Uniform variable ‘LightLocation’\n” );
else
        glUniform3fv( location, 3, lightLoc );
```


Passing in Attribute Variables

```
GLint location = glGetAttribLocation( program, “abArray” );

if( location < 0 )
 {
      fprintf( stderr, “Cannot find Attribute variable ‘abArray’\n” );
 }
else
{
      glBegin( GL_TRIANGLES );
            glVertexAttrib2f( location, a0, b0 );
            glVertex3f( x0, y0, z0 );
            glVertexAttrib2f( location, a1, b1 );
            glVertex3f( x1, y1, z1 );
            glVertexAttrib2f( location, a2, b2 );
            glVertex3f( x2, y2, z2 );
      glEnd();
}
```

Checking for Errors

```
void
CheckGlErrors( const char* caller )
{
	unsigned int glerr = glGetError();
	if( glerr == GL_NO_ERROR )
		return;
	fprintf( stderr, "GL Error discovered from caller '%s': ", caller );
	switch( glerr )
	{
		case GL_INVALID_ENUM:
			fprintf( stderr, "Invalid enum.\n" );
			break;
		case GL_INVALID_VALUE:
			fprintf( stderr, "Invalid value.\n" );
			break;
		case GL_INVALID_OPERATION:
			fprintf( stderr, "Invalid Operation.\n" );
			break;
		case GL_STACK_OVERFLOW:
			fprintf( stderr, "Stack overflow.\n" );
			break;
		case GL_STACK_UNDERFLOW:
			fprintf(stderr, "Stack underflow.\n" );
			break;
		case GL_OUT_OF_MEMORY:
			fprintf( stderr, "Out of memory.\n" );
			break;
		default:
			fprintf( stderr, "Unknown OpenGL error: %d (0x%0x)\n", glerr, glerr );
	}
}
```

This is not a bad idea to do all through your OpenGL programs, even without shaders!

2009

Writing a C++ Class to Handle Everything is Fairly Straightforward

Setup:

```
int Polar;
float K;
GLSLProgram *Hyper = new GLSLProgram("hyper.vert", "hyper.geom", "hyper.frag" );
```

This loads, compiles, and links the shader.
It prints error messages and returns NULL if something failed.

Using the GPU program during display:

```
Hyper->Use( );
Hyper->SetVariable( "Polar", Polar );
Hyper->SetVariable( "K", K );
```

Reverting to the fixed-function pipeline during display:

```
Hyper->Use( 0 );
```

mjb – May 19, 2009