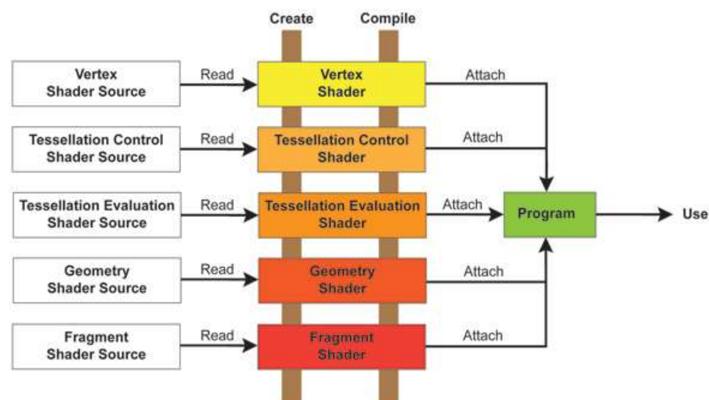
The GLSL API

Oregon State University

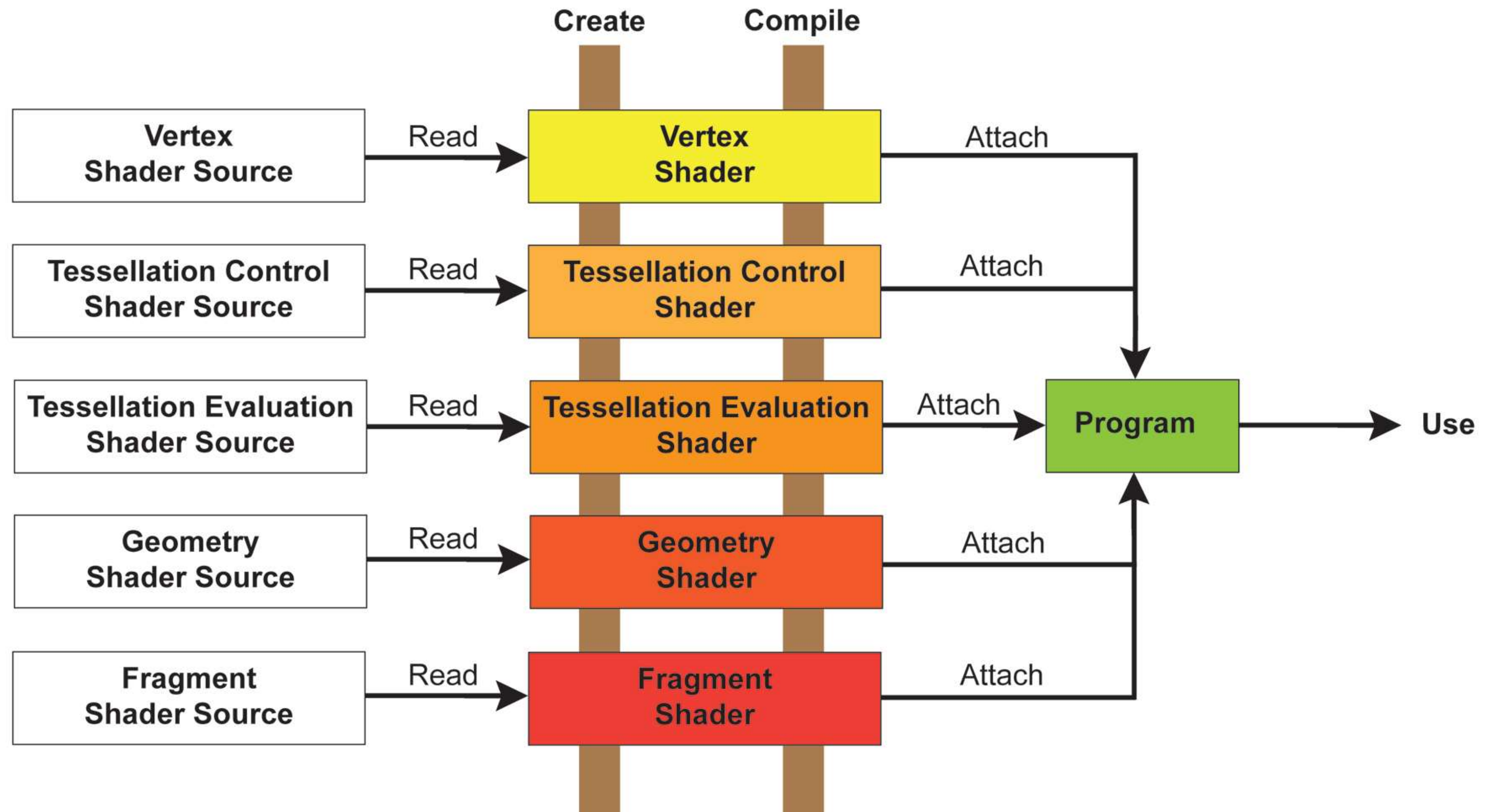
Mike Bailey

mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License



The GLSL Shader-Creation Process



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) );
```

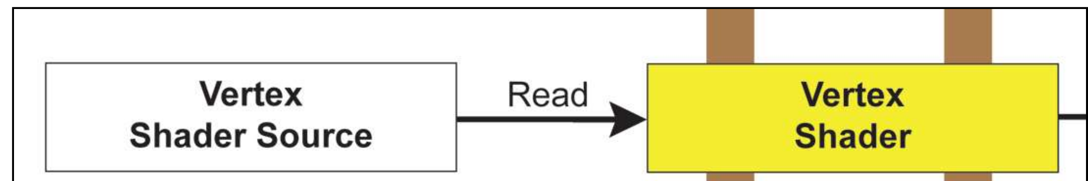
Do this *immediately* after opening the window

GLEW cannot be initialized until a graphics window is open. Like OpenGL itself, GLEW’s calls will not work unless it can see a graphics context (i.e., a graphics state).

http://glew.sourceforge.net

Reading a 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
```

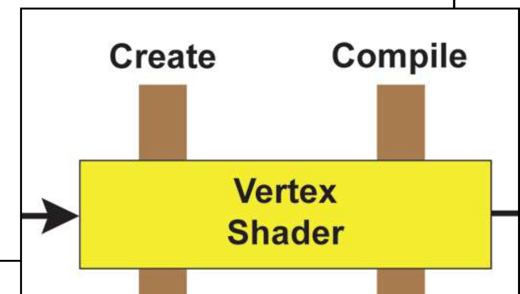


Creating and Compiling a Vertex Shader from that character buffer (Geometry and Fragment files work the same way)

This is the only part of this process that is 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" );
```

An array of strings



Creating Different Shader Types

```
GLuint shader = glCreateShader( GL_VERTEX_SHADER );

GLuint shader = glCreateShader( GL_GEOMETRY_SHADER );

GLuint shader = glCreateShader( GL_TESS_CONTROL_SHADER );

GLuint shader = glCreateShader( GL_TESS_EVALUATION_SHADER );

GLuint shader = glCreateShader( GL_FRAGMENT_SHADER );

GLuint shader = glCreateShader( GL_COMPUTE_SHADER );
```

Other than this, the rest of the create, compile, link process is the same for each shader type.

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* 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 );
```

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

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

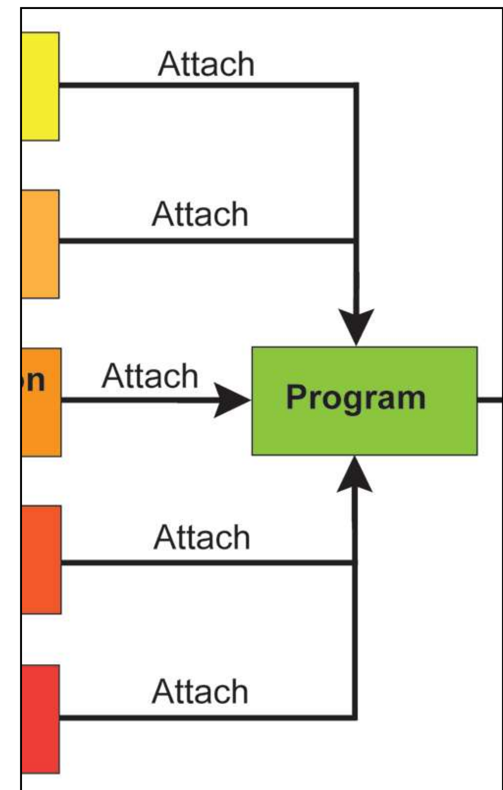
if-tests vs. preprocessing

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

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


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

Creating the Program and Attaching the Shaders to It


```
GLuint program = glCreateProgram( );

glAttachShader( program, vertShader );

glAttachShader( program, fragShader );

glAttachShader( program, geomShader );
```



Linking the Program and Checking its Validity

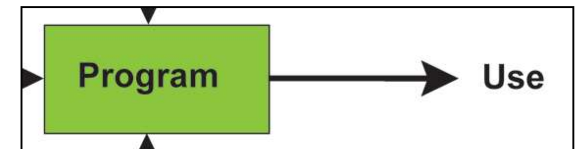

```
glLinkProgram( program );
CheckGlErrors( "Shader Program 1" );
glGetProgramiv( program, GL_LINK_STATUS, &status );
if( status == GL_FALSE )
{
        fprintf( stderr, “Link failed.\n” );
        glGetProgramiv( program, GL_INFO_LOG_LENGTH, &logLength );
        log = new GLchar [logLength];
        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_FALSE ? “invalid” : “valid” );
```

Making the Program Active

```
glUseProgram(  program  );
```

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

```
glUseProgram(  0  );
```



Using Multiple Shader Programs

```
glUseProgram(  program0  );
        <draw some stuff>

glUseProgram(  program1  );
    <draw some more stuff>

glUseProgram(  program2  );
    <draw some more stuff>

glUseProgram(  program3  );
    <draw some more stuff>

glUseProgram(  program4  );
    <draw some more stuff>

glUseProgram(  program5  );
    <draw some more stuff>
```

A specified shader program is an “attribute” – it stays in effect until you change it

Passing in Uniform Variables

You first need to find the variable's location in the shader program's symbol table.

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

GLint location = glGetUniformLocation( program, "uLightLocation" );

if( location < 0 )
		fprintf( stderr, "Cannot find Uniform variable 'uLightLocation'\n" );
else
		glUniform3fv( location, 1, lightLoc );
```

Then you need to fill it.

Passing in Attribute Variables

You first need to find the variable's location in the shader program's symbol table.

```
GLint location = glGetAttribLocation( program, "aArray" );

if( location < 0 )
 {
	fprintf( stderr, "Cannot find Attribute variable 'aArray'\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( );
}
```

Then you need to fill it per-vertex.

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 );
	}
}
```

Oregon State University Computer Graphics

It’s not a bad idea to do this in all your OpenGL programs, even without shaders!

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

Setup in InitGraphics( ):

```
GLSLProgram *Hyper = new GLSLProgram( );
bool valid = Hyper->Create( "hyper.vert", "hyper.geom", "hyper.frag" );
if( ! valid ) { . . . }
```

This loads, compiles, and links the shader. It prints error messages if something went wrong.

Using the GPU program in Display( ):

```
int Polar = ???;
float K = ???;

Hyper->Use( );
Hyper->SetUniformVariable( “uPolar", Polar );
Hyper->SetUniformVariable( “uK", K );
glBegin( GL_TRIANGLES );
            Hyper->SetAttributeVariable( “aTemperature”, T0 );
            glVertex3f( x0, y0, z0 );
            Hyper->SetAttributeVariable( “aTemperature”, T1 );
            glVertex3f( x1, y1, z1 );
            Hyper->SetAttributeVariable( “aTemperature”, T2 );
            glVertex3f( x2, y2, z2 );
glEnd( );
```

Reverting to the fixed-function pipeline in Display( ):

```
Hyper->UnUse(  );      // Hyper->Use( 0 ) also works
```

SPIR-V

SPIR-V is a file format that can be used to hold shader code that has been pre-compiled, but has not yet been turned into machine code. It was created as a way for software developers to pre-compile their code and then allow the vendor-specific driver to produce the final binary representation. There are four major advantages in doing things this way:

1. A software developer can more easily wring compiler errors from the code by having an external compiler that can be run independently from the application.

2. Vendors can still apply their optimization-magic in their device-specific drivers.

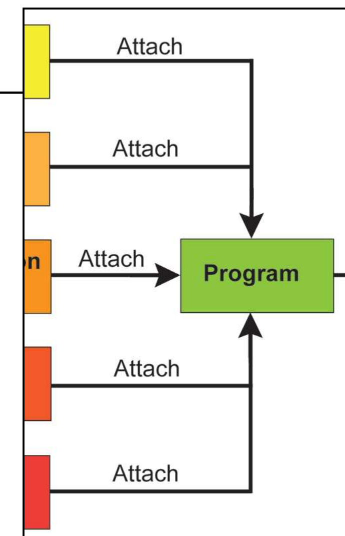
3. SPIR-V files can be read at the start of a program and be turned into machine code faster than the original GLSL files could have been turned into machine code.

4. Software developers can distribute their code without having to reveal the shaders' source code.

Reading SPIR-V-compiled Shaders

The new **glShaderBinary( )** call replaces both **glCreateShader( )** and **glCompilerShader( )**

```
FILE *fp = fopen( filename, "r" );
if( fp == NULL )  { . . . }
fseek( fp, 0, SEEK_END );
int numBytes = ftell( fp );     // length of file – guaranteed to be a multiple of 4
GLchar * buffer = new GLchar [numBytes];
rewind( fp );                   // same as:  "fseek( in, 0, SEEK_SET )"
fread( buffer, 1, numBytes, fp );
fclose( fp );

GLuint shaders[2]
glShaderBinary( 2, shaders, GL_SHADER_BINARY_FORMAT_SPIR_V, buffer, numbytes );
GLuint program = glCreateProgram( );
glAttachShader( program, shaders[0] );
glAttachShader( program, shaders[1] );
```



SPIR-V:
Standard Portable Intermediate Representation for Vulkan

glslangValidator shaderFile -G [-H] [-I<dir>] [-S <stage>] -o shaderBinaryFile.spv

Shaderfile extensions:

.vert	**Vertex**
.tesc	**Tessellation Control**
.tese	**Tessellation Evaluation**
.geom	**Geometry**
.frag	**Fragment**
.comp	**Compute**

(Can be overridden by the –S option)

-V	Compile for Vulkan
-G	Compile for OpenGL
-I	Directory(ies) to look in for #includes
-S	Specify stage rather than get it from shaderfile extension
-c	Print out the maximum sizes of various properties

Windows: glslangValidator.exe
Linux: setenv LD_LIBRARY_PATH /usr/local/common/gcc-6.3.0/lib64/

How do you know if SPIR-V compiled successfully?

Same as C/C++ -- the compiler gives you no nasty messages.

Also, if you care, legal .spv files have a magic number of **0x07230203**

So, if you do an **od –x** on the .spv file, the magic number looks like this:

0203 0723 . . .