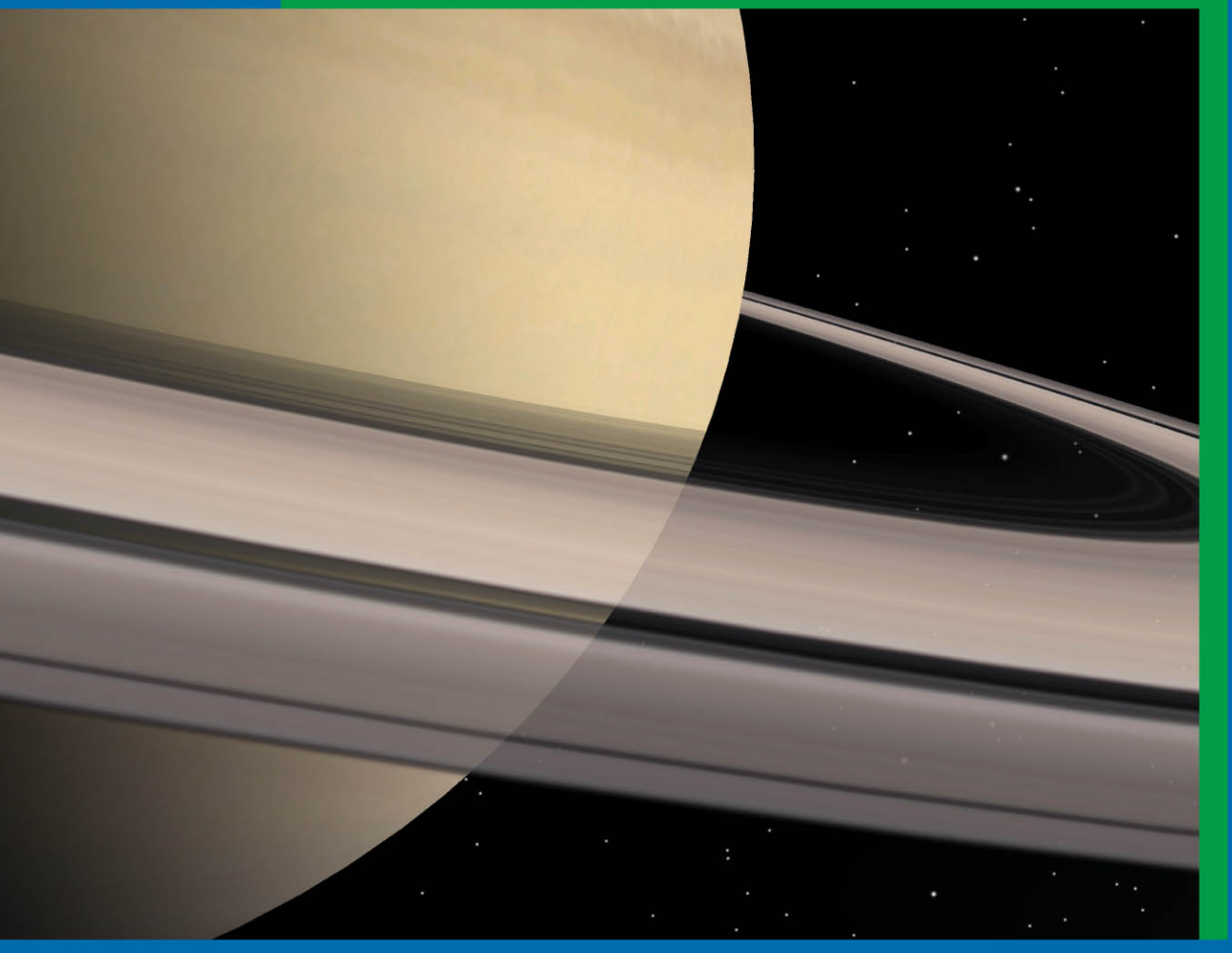
OpenGL® SuperBible

Seventh Edition

Comprehensive Tutorial and Reference



Graham Sellers ■ Richard S. Wright, Jr. ■ Nicholas Haemel

OpenGL®

SuperBible

Seventh Edition

As you can see, several errors and a warning have been generated and recorded in the shader's information log. For this particular compiler, the format of the error messages is `ERROR` or `WARNING` followed by the string index (remember, **`glShaderSource()`** allows you to attach multiple source strings to a single shader object), followed by the line number. Let's look at the errors one by one:

```
ERROR: 0:5: error(#12) Unexpected qualifier
```

Line 5 of our shader is this:

```
uniform scale;
```

It seems that we have forgotten the type of the `scale` uniform. We can fix that by giving `scale` a type (it's supposed to be `vec4`). The next three issues are on the same line:

```
ERROR: 0:10: error(#143) Undeclared identifier: scale
WARNING: 0:10: warning(#402) Implicit truncation of vector from
size: 4 to size: 3
ERROR: 0:10: error(#162) Wrong operand types: no operation "+" exists
that takes a left-hand operand of type "4-component vector of vec4" and
a right operand of type "uniform 3-component vector of vec3" (or there
is no acceptable conversion)
```

The first one says that `scale` is an undefined identifier—that is, the compiler doesn't know what `scale` is. This is because of that first error on line 5: We haven't actually defined `scale` yet. Next is a warning that we are attempting to truncate a vector from a four-component type to a three-component type. This might not be a serious issue, given that the compiler might be confused as a result of another error on the very same line. This warning is saying that there is no version of the + operator that can add a `vec3` and a `vec4`. It appears because, even once we've given `scale` its `vec4` type, `bias` has been declared as a `vec3` and therefore can't be added to a `vec4` variable. A potential fix is to change the type of `bias` to `vec4`.

If we apply our now known fixes to the shader (shown in Listing 6.1), we have

```
#version 450 core

layout (location = 0) out vec4 color;

uniform vec4 scale;
uniform vec4 bias;

void main(void)
{
    color = vec4(1.0, 0.5, 0.2, 1.0) * scale + bias;
}
```

Once we compile this updated shader, we should have success: Calling **glGetShaderiv()** with pname set to GL_COMPILE_STATUS should return GL_TRUE, and the new info log should either be empty or simply indicate success.

Getting Information from the Linker

Just as compilation may fail, so linking of programs may also fail or not go exactly the way you planned. While the compiler will produce an info log when you call **glCompileShader()**, when you call **glLinkProgram()**, the linker can also produce a log that you can query to figure out what happened. Also, a program object has several properties, including its link status, resource usage, and so on, that you can retrieve. In fact, a linked program has quite a bit more status than a compiled shader. You can retrieve all of this information by using **glGetProgramiv()**, whose prototype is

```
void glGetProgramiv(GLuint program,
                    GLenum  pname,
                    GLint * params);
```

Notice that **glGetProgramiv()** is very similar to **glGetShaderiv()**. The first parameter, program, is the name of the program object whose information you want to retrieve. The last parameter, params, is the address of a variable where you would like OpenGL to write that information. Just like **glGetShaderiv()**, **glGetProgramiv()** takes a parameter called pname that indicates what you would like to know about the program object. There are many more valid values for pname for program objects as well, including these:

- GL_DELETE_STATUS, like the same property of shaders, indicates whether **glDeleteProgram()** has been called for the program object.
- GL_LINK_STATUS, similarly to the GL_COMPILE_STATUS property of a shader, indicates the success of linking the program.
- GL_INFO_LOG_LENGTH returns the info log length for the program.
- GL_ATTACHED_SHADERS returns the number of shaders that are attached to the program.
- GL_ACTIVE_ATTRIBUTES returns the number of attributes that the vertex shader in the program actually[5] uses.

5. More precisely, that the compiler thinks the vertex shader uses.

- `GL_ACTIVE_UNIFORMS` returns the number of uniforms used by the program.
- `GL_ACTIVE_UNIFORM_BLOCKS` returns the number of uniform blocks used by the program.

You can tell whether a program has been successfully linked by calling **`glGetProgramiv()`** with `pname` set to `GL_LINK_STATUS`. If it returns `GL_TRUE` in `params`, then linking worked. You can also get the information log from a program just as you can from a shader. To do this, you can call **`glGetProgramInfoLog()`**, whose prototype is

```
void glGetProgramInfoLog(GLuint program,
                         GLsizei bufSize,
                         GLsizei * length,
                         GLchar * infoLog);
```

The parameters to **`glGetProgramInfoLog()`** work just the same as they do for **`glGetShaderInfoLog()`**, except that instead of `shader`, we have `program` (the name of the program object whose log you want to read). Now, consider the shader shown in Listing 6.2.

```
#version 450 core

layout (location = 0) out vec4 color;

vec3 myFunction();

void main(void)
{
    color = vec4(myFunction(), 1.0);
}
```

Listing 6.2: Fragment shader with external function declaration

Listing 6.2 includes a declaration of an external function. This works similarly to C programs, where the actual definition of the function is contained in a separate source file. OpenGL expects that the function body for `myFunction` will be defined in one of the fragment shaders attached to the program object (remember, you can attach multiple shaders of the same type to the same program object and have them link together). When you call **`glLinkProgram()`**, OpenGL will look in all the fragment shaders for a function called `myFunction`; if it doesn't find this function, it will generate a link error. Trying to link just this fragment shader into a program object generates the following error message:

```
Vertex shader(s) failed to link, fragment shader(s) failed to link.
ERROR: error(#401) Function: myFunction() is not implemented
```

To resolve this error, we can either include the body of `myFunction` in the shader of Listing 6.2, or attach a second fragment shader to the same program object that includes the function body.

Separate Programs

So far, all of the programs you have used have been considered *monolithic* program objects. That is, they contain a shader for each stage that is active. You have attached a vertex shader, a fragment shader, and possibly tessellation or geometry shaders to a single program object and then called `glLinkProgram()` to link the program object into a single representation of the entire pipeline. This type of linking might allow a compiler to perform inter-stage optimizations such as eliminating code in a vertex shader that contributes to an output that is never used by the subsequent fragment shader. However, this scheme comes at a potential cost of flexibility and possibly performance to the application. For every combination of vertex, fragment, and possibly other shaders, you need to have a unique program object, and linking all those programs doesn't come cheap.

For example, consider the case where you want to change only a fragment shader. With a monolithic program, you would need to link the same vertex shader to two or more different fragment shaders, creating a new program object for each combination. If you have multiple fragment shaders and multiple vertex shaders, you now need a program object for each combination of shaders. This problem gets worse as you add more shaders and shader stages to the mix. Eventually, you may end up with a combinatorial explosion of shader combinations that can quickly balloon into thousands of permutations, or more.

To alleviate this problem, OpenGL supports linking program objects in *separable* mode. A program linked this way can contain shaders for only a single stage in the pipeline or for just a few of the stages. Multiple program objects, each representing a section of the OpenGL pipeline, can then be attached to a *program pipeline object* and matched together at runtime rather than at link time. Shaders attached to a single program object can still benefit from inter-stage optimizations, but the program objects attached to a program pipeline object can be switched around at will with relatively little cost in performance.

To use a program object in separable mode, you need to tell OpenGL what you plan to do *before* you link it by calling **`glProgramParameteri()`** with `pname` set to `GL_PROGRAM_SEPARABLE` and `value` set to `GL_TRUE`. This tells

OpenGL not to eliminate any outputs from a shader that it thinks aren't being used. It will also arrange any internal data layout such that the last shader in the program object can communicate with the first shader in another program object with the same input layout. Next, you should create a program pipeline object with **glGenProgramPipelines()**, and then attach programs to it representing the sections of the pipeline you wish to use. To do so, call **glUseProgramStages()**, passing in the name of the program pipeline object, a bitfield indicating which stages to use, and the name of a program object that contains those stages.

Listing 6.3 shows an example in which we set up a program pipeline object with two programs, one containing only a vertex shader and one containing only a fragment shader.

```
// Create a vertex shader
GLuint vs = glCreateShader(GL_VERTEX_SHADER);

// Attach source and compile
glShaderSource(vs, 1, vs_source, NULL);
glCompileShader(vs);

// Create a program for our vertex stage and attach the vertex shader to it
GLuint vs_program = glCreateProgram();
glAttachShader(vs_program, vs);

// Important part - set the GL_PROGRAM_SEPARABLE flag to GL_TRUE *then* link
glProgramParameteri(vs_program, GL_PROGRAM_SEPARABLE, GL_TRUE);
glLinkProgram(vs_program);

// Now do the same with a fragment shader
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, fs_source, NULL);
glCompileShader(fs);
GLuint fs_program = glCreateProgram();
glAttachShader(fs_program, vs);
glProgramParameteri(fs_program, GL_PROGRAM_SEPARABLE, GL_TRUE);
glLinkProgram(fs_program);

// The program pipeline represents the collection of programs in use:
// Generate the name for it here.
GLuint program_pipeline;
glGenProgramPipelines(1, &program_pipeline);

// Now use the vertex shader from the first program and the fragment shader
// from the second program.
glUseProgramStages(program_pipeline, GL_VERTEX_SHADER_BIT, vs_program);
glUseProgramStages(program_pipeline, GL_FRAGMENT_SHADER_BIT, fs_program);
```

Listing 6.3: Configuring a separable program pipeline

Although this simple example includes only two program objects, each with just a single shader in it, it's possible to have more complex arrangements where more than two program objects are used, or where

one or more of the program objects contain more than one shader. For example, tessellation control and tessellation evaluation shaders are often tightly coupled, such that one does not make much sense without the other. Also, very often when tessellation is used, it is possible to use a pass-through vertex shader and do all of the real vertex shader work either in the tessellation control shader or in the tessellation evaluation shader. In those cases, it may make sense to couple a vertex shader and both tessellation shaders in one program object, but still use separable programs to be able to switch the fragment shader on the fly.

If you really do want to create a simple program object with exactly one shader object in it, you can take a shortcut and call

```
GLuint glCreateShaderProgramv(GLenum type,
                              GLsizei count,
                              const char ** strings);
```

The **`glCreateShaderProgramv()`** function takes the type of shader you want to compile (e.g., `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`), the number of source strings, and a pointer to array of strings (just like **`glShaderSource()`**), and compiles those strings into a new shader object. Then, it internally attaches that shader object to a new program object, sets its separable hint to `TRUE`, links it, deletes the shader object, and returns the program object to you. You can then use this program object in your program pipeline objects.

Once you have a program pipeline object with a bunch of shader stages compiled into program objects and attached to it, you can make it the current pipeline by calling **`glBindProgramPipeline()`**:

```
void glBindProgramPipeline(GLuint pipeline);
```

Here, `pipeline` is the name of the program pipeline object that you wish to use. Once the program pipeline object is bound, its programs will be used for rendering or compute operations.

Interface Matching

GLSL provides a specific set of rules for how the outputs from one shader stage are matched up with the corresponding inputs in the next stage. When you link a set of shaders together into a single program object, OpenGL's linker will tell you if you didn't match things up correctly. However, when you use separate program objects for each stage, the matching occurs when you switch program objects—and not lining things