The OpenGL Shading Language (GLSL)

- Or, see what you can do now that you understand what's going on behind the scenes
- Everything we've learned so far about how OpenGL does computer graphics is known as its *fixed-function* mode — it's how it works "out of the box," with no further intervention from you
- But, we have learned by now that, if you know what you're doing, you can do things differently, whether to add new effects or change an existing one

GLSL Big Picture

- GLSL is standard with OpenGL 2.0 or greater; it is an extension in prior versions
- GLSL programs replace the standard OpenGL graphics pipeline with your own: all state variables and vertex information are made available to you, and you determine the output produced by these values
- Using GLSL typically involves: (1) designing your own shading algorithm, (2) implementing the algorithm in GLSL, and (3) telling an OpenGL program to use your shader(s) instead of the default fixed functionality

Vertex and Fragment Shaders

- There are two types of shaders, corresponding to the two phases into which you can inject your own functionality: *vertex* and *fragment*
- A *vertex shader* takes data such as the current vertex, normal, and color, and produces a final position, front and back colors, plus additional user-defined values
- A *fragment shader* takes pixel coordinates, color, user-defined values, among others, and produces a final fragment color, depth, or other data

Hooking Up GLSL

GLSL is a programming language, and so using it with OpenGL is not unlike programming in general:

- Write the source code
- Pass the source code to OpenGL for compilation (catching errors if any)
- Link compiled *shaders* into an overall *program* (also catching possible errors)
- Pass values or attributes to the program using the designated API as needed

Language Highlights

As a language, GLSL is syntactically similar to C and Java, though of course includes features that specifically address computer graphics algorithms:

- Vector and matrix types and operations (*vec2*, *vec3*, *vec4*, *mat2*, *mat3*, *mat4*; *dot()*, *cross()*, *normalize()*, and vector/matrix overloaded +, *, etc.)
- Vector/matrix access includes array-style (e.g., *v[0]*), structure-style (e.g., *v.x* or *c.r*), and an interesting operation called *swizzling*, which concatenates attributes (e.g., *luminance* = *color.rrr*; *diag* = *v.xxx*)

- Variables may have *type modifiers*, three of which are specific to how the graphics pipeline works:
  - *const* resembles similar constructs in other languages
  - *attribute* indicates a value that may be attached on a per-vertex basis (in case color, material, normal, among others, are not enough)
  - *uniform* indicates a value that is passed by the calling OpenGL program that will not change within the shader
  - *varying* values are calculated by the vertex shader, then passed into the fragment shader
- A family of *sampler* data types enables access to texture maps from within a shader
- Identifiers starting with *gl_* are reserved — assorted state machine values are available through these names (*gl_Vertex*, *gl_Color*, *gl_FrontMaterial*, and many more)
- Wide variety of built-in functions, including specialized ones like *reflect()*, *refract()*, and *noise()*