GLSL: The GL Shading Language (GLSL)

- Prior incarnations of OpenGL included a fixed-function pipeline— it worked in a particular way "out of the box," with no further intervention from you
- The industry has evolved to the point where fixed-function is now more overhead than convenience—graphics programs and platforms vary greatly and user expectations have increased such that total customizability has been mandated as of OpenGL 4.0

GLSL Big Picture

(1) Determine how your "world" vertices get transformed into NDC (vertex shader)—this includes any supplementary information/variables needed to perform this calculation

(2) Determine how your polygons' final colors are computed (fragment shader)—similar to above, you need to know necessary variables, parameters, etc.

(3) Connect your shaders to the main program: this involves both their GLSL code and variables/attributes

Vertex and Fragment Shaders

- The two types of shaders correspond 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, per-vertex 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

GLSL is syntactically similar to C, with 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`)

- Some variables and types are specialized for graphics:
  - `const` resembles similar constructs in other languages
  - `attribute` indicates a value that may be attached on a per-vertex basis (e.g., color, normal, etc.)
  - `uniform` indicates a value that is passed by the calling program that will not change within the shader
  - `varying` values are calculated by the vertex shader, then passed into the fragment shader
  - `sampler` data types enable access to texture maps
- Identifiers starting with `gl_` are reserved—assorted values are available through these names, either as input or output (`gl_Position`, `gl_FragColor`, etc.)
- Built-in functions like `reflect()`, `refract()`, and `noise()`