Shaders and GLSL

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations
- The vector and matrix classes of GLSL are first-class types, with arithmetic and logical operations well defined. This helps simplify your code, and prevent errors.
- Note in the above example, overloading ensures that both a*m and m*a are defined although they will not in general produce the same result.

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
v[1], v.y, v.g, v.t
```
 - all refer to the same element

- Component swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

- *Swizzles* allow components within a vector to be accessed by name. For example, the first element in a vector – element 0 – can also be referenced by the names “x”, “s”, and “r”. Why all the names – to clarify their usage. If you’re working with a color, for example, it may be clearer in the code to use “r” to represent the red channel, as compared to “x”, which make more sense as the x-positional coordinate

Qualifiers

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- `in` qualifiers that indicate the shader variable will receive data flowing into the shader, either from the application, or the previous shader stage.
- `out` qualifier which tag a variable as data output where data will flow to the next shader stage, or to the framebuffer

`uniform`

  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

- `uniform` qualifiers for accessing data that doesn't change across a draw operation

Functions

- GLSL also provides a rich library of functions supporting common operations. While pretty much every vector- and matrix-related function available you can think of, along with the most common mathematical functions are built into GLSL,
- there's no support for operations like reading files or printing values.
- Shaders are really data-flow engines with data coming in, being processed, and sent on for further processing.
- Built in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- User defined

Built-in Variables

- vertex data, which can be processed by up to four shader stages in OpenGL, are all ended by setting a positional value into the built-in variable, `gl_Position`.
- `gl_Position`
  - (required) output position from vertex shader
- `gl_FragCoord` is a read-only variable, while gl_FragDepth is a read-write variable.
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Simple Vertex Shader for Cube Example

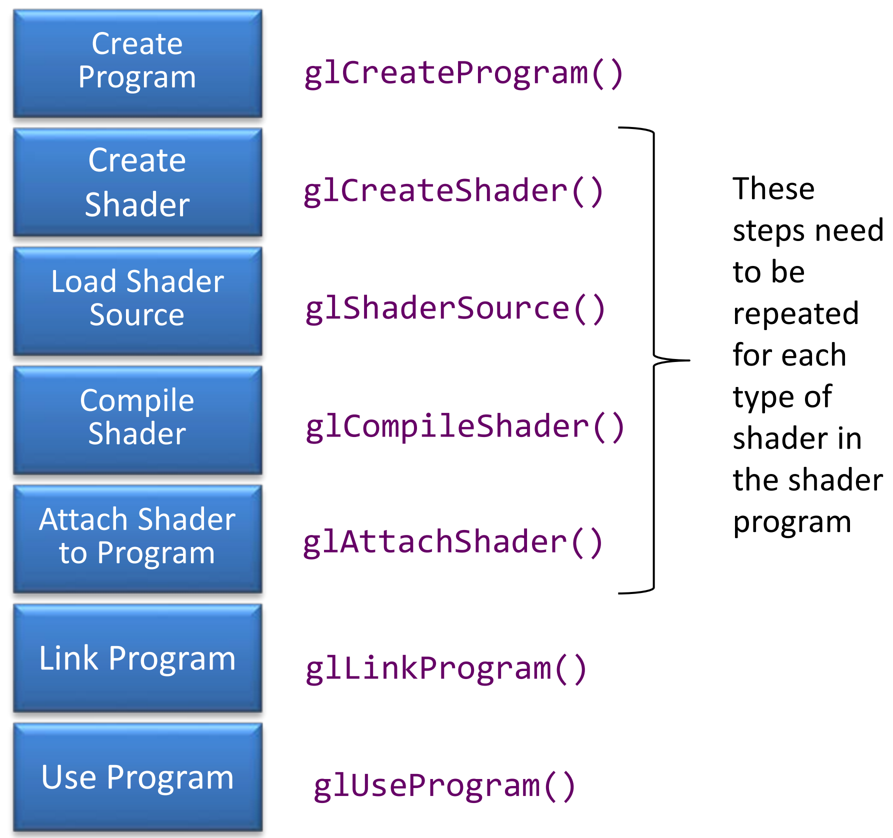
```
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

The Simplest Fragment Shader

```
#version 430

in vec4 color;

out vec4 fColor; // fragment’s final color

void main()
{
    fColor = color;
}
```

Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



Getting Your Shaders into OpenGL

- Shaders need to be compiled in order to be used in your program. As compared to C programs, the compiler and linker are implemented in the OpenGL driver, and accessible through function calls from within your program.
- The diagram illustrates the steps required to compile and link each type of shader into your shader program. A program can contain either a vertex shader (which replaces the fixed-function vertex processing), a fragment shader (which replaces the fragment coloring stages), or both.
- If a shader isn't present for a particular stage, the fixed-function part of the pipeline is used in its place.
- Just a with regular programs, a syntax error from the compilation stage, or a missing symbol from the linker stage could prevent the successful generation of an executable program.
- There are routines for verifying the results of the compilation and link stages of the compilation process, but are not shown here. Instead, we've provided a routine that makes this process much simpler, as demonstrated on the next slide.

A Simpler Way

- We’ve created a routine for this course to make it easier to load your shaders
- `InitShaders()` accepts two parameters, each a filename to be loaded as source for the vertex and fragment shader stages, respectively.
- The value returned from `InitShaders()` will be a valid GLSL program id that you can pass into `glUseProgram()`.

```
GLuint InitShaders( const char* vFile,
                    const char* fFile );
```

- `InitShaders` takes two filenames
  - `vFile` path to the vertex shader file
  - `fFile` for the fragment shader file
- Fails if shaders don’t compile, or program doesn’t link

Associating Shader Variables and Data

- Need to associate a shader variable with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
  - specify association before program linkage
  - query association after program linkage

Initializing Uniform Variable Values

- Uniform Variables

```
glUniform4f( index, x, y, z, w );

GLboolean  transpose = GL_TRUE;

// Since we're C programmers
GLfloat  mat[3][4][4] = { … };
glUniformMatrix4fv( index, 3, transpose,
mat );
```

Finishing the Cube Program

```
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE |GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Color Cube" );

    glewInit();
    init();

    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );
    glutMainLoop();

    return 0;
}
```

Cube Program's GLUT Callbacks

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
        case 033: case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

Vertex Shader Examples

- A vertex shader is initiated by each vertex output by `glDrawArrays()`
- A vertex shader must output a position in clip coordinates to the rasterizer
- Basic uses of vertex shaders
  - Transformations
  - Lighting
  - Moving vertex positions