Shaders and GLSL
OpenGL 3.1 Pipeline

- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders

- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects
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
- Operators overloaded for matrix and vector operations

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

b = a*m;
c = m*a;
```
Components and Swizzling

For vectors can use [ ], xyzw, rgba or stpq

Example:

```cpp
vec3 v;

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

Swizzling:

```cpp
vec3 a, b;

a.x = b.y;
```
Qualifiers

- **in**, **out**
  - Copy vertex attributes and other variable to/from shaders

```glsl
in vec2 tex_coord;
out vec4 color;
```

- **Uniform**: variable from application

```glsl
uniform float time;
uniform vec4 rotation;
```
Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for
Functions

- **Built in**
  - Arithmetic: $\sqrt{}$, power, abs
  - Trigonometric: $\sin$, $\arcsin$
  - Graphical: length, reflect

- **User defined**
Built-in Variables

- `gl_Position`: output position from vertex shader
- `gl_FragColor`: output color from fragment shader
  - Only for ES, WebGL and older versions of GLSL
  - Present version use an out variable
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
The Simplest Fragment Shader

```cpp
in vec4 color;
out vec4 FragColor;

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

### OpenGL Steps

1. **Create Program**
   - `glCreateProgram()`

2. **Create Shader**
   - `glCreateShader()`

3. **Load Shader Source**
   - `glShaderSource()`

4. **Compile Shader**
   - `glCompileShader()`

5. **Attach Shader to Program**
   - `glAttachShader()`

6. **Link Program**
   - `glLinkProgram()`

7. **Use Program**
   - `glUseProgram()`

These steps need to be repeated for each type of shader in the shader program.
We’ve created a routine for this course to make it easier to load your shaders:

- available at course website

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

- `InitShaders` takes two filenames:
  - `vFile` for the vertex shader
  - `fFile` for the fragment shader

- 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
Assumes you already know the variables’ name

```c
GLint idx =
    glGetAttribLocation( program, "name" );
```

```c
GLint idx =
    glGetUniformLocation( program, "name" );
```
Uniform Variables

```c
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

```c
int main( int argc, char **argv ) {
    glutInit( &argc, argv );

    if (!glfwInit())
        error = -1;

    m_window = glfwCreateWindow(640, 480, title.c_str(), NULL, NULL);
    glfwMakeContextCurrent(m_window);
    GLenum res = glewInit();
}
```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods){
    switch (key) {
    case GLFW_KEY_ESCAPE:
        if(action == GLFW_PRESS)
            glfwSetWindowShouldClose(window,GLFW_TRUE);
        break;
    default:
        break;
    }
}

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