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;
```

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

Swizzling:

```cpp
vec3 a, b;
```

```cpp
a.xy = b.yx;
```
• **in, out**
  
  ▪ Copy vertex attributes and other variable to/from shaders

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

• **Uniform: variable from application**

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

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

- Built in
  - Arithmetic: \(\sqrt{}, \text{power}, \text{abs}\)
  - Trigonometric: \(\sin, \text{asin}\)
  - Graphical: \(\text{length}, \text{reflect}\)

- 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;
}
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

These steps need to be repeated for each type of shader in the shader program.

- Create Program
  - `glCreateProgram()

- Create Shader
  - `glCreateShader()`

- Load Shader Source
  - `glShaderSource()`

- Compile Shader
  - `glCompileShader()`

- Attach Shader to Program
  - `glAttachShader()`

- Link Program
  - `glLinkProgram()`

- Use Program
  - `glUseProgram()`
A Simpler Way

- 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
Determining Locations After Linking

Assumes you already know the variables’ name

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

```c
GLint idx =
    glGetUniformLocation( program, "name" );
```
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 );
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 GLUT Callbacks

```c
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;
    }
}
```
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