Today's Agenda

Shaders fundamentals

Programming with shader-based OpenGL

Shaders

“Like a function call – data are passed in, processed, and passed back out”

-- Shreiner et al, OpenGL Programming Guide

GLSL is like a complete C program, but without

- recursion
- Pointer
- Dynamic allocation of memory

Vertex Shader

Basic task: Sending vertices positions to the rasterizer

Advanced tasks:

- Transformation
  - Projection
- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Processing color

A Simple Vertex Shader: triangles.vert (Shreiner et al)

```
#version 430 core
```

Specify it is an input to the shader

```
in vec4 vPosition;
```

Global variable, copied from the application to the shader

```
void main()

{

	gl_Position = vPosition;

}
```

A built-in variable, passing data to the rasterizer

A Simple Fragment Shader

```
#version 330 core


out vec4 fColor;


void main()
{
        fColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```

Example of Vertex Shader: Color Processing

```
// vertex shader
#version 150
in vec4 vPosition;
out vec4 color;
void main()
{
    color = vec4( 0.5 + vPosition.x, 0.5
+ vPosition.y, 0.5 + vPosition.z, 1.0 );
    gl_Position = vPosition;
}
```

```
// fragment shader
#version 150
in  vec4  color;
out vec4  fColor;
void main()
{
    fColor = color;
}
```

Declaring Variables

Allowed: Letters, numbers, “_”

Not allowed:

- Digits and “_” cannot appear as the first character
- Do not allow consecutive “_”

Data Types

Basic types: int, float, double, uint, bool

Fewer implicit conversion

int ~~f =~~ false

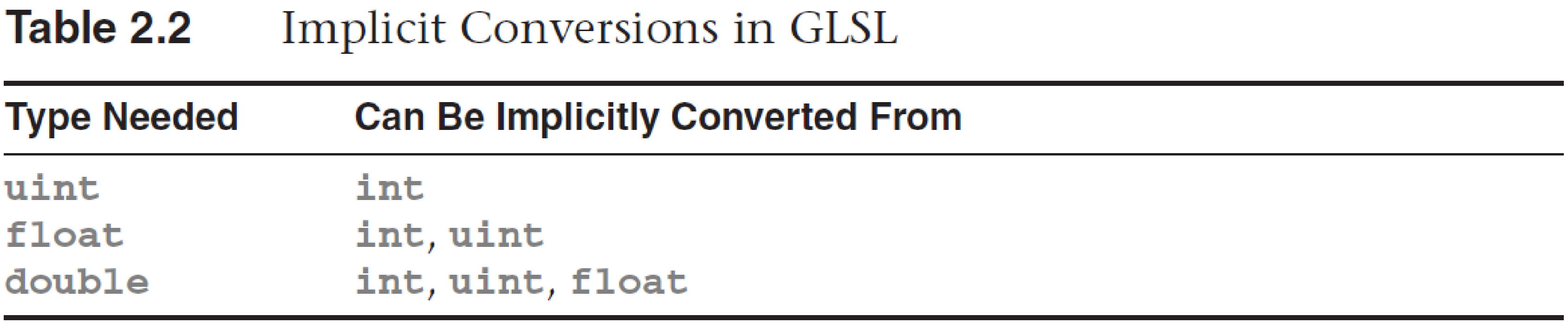


Table 2.2 Implicit Conversions in GLSL

Type Needed	Can Be Implicitly Converted From
`uint`	`int`
`float`	`int`, `uint`
`double`	`int`, `uint`, `float`

Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

Data Types

Vectors:

- float vec2, vec3, vec4
- Also int (ivec) and boolean (bvec)

Matrices: mat2, mat3, mat4, mat3x4

- Stored by columns

```
mat2 M=mat2 (1.0, 2.0,
             3.0, 4.0);
```

⟺

```
vec2 col1=vec2(1.0,2.0);
vec2 col2=vec2(3.0,4.0);
mat2 M=mat2(col1,col2);
```

Structures: grouping different types

Data Types

C++ style constructors

- Truncate a vector:
  - vec4 color =vec4(1.0, 2.0, 3.0, 1.0); vec3 rgb = vec3(color)
- Lengthen a vector:
  - vec3 white = vec3(1.0) → white = (1.0,1.0,1.0)
  - vec4 translucent = vec4 (white, 0.5)
- Initialize a matrix

$$\mathrm{M}=\mathrm{mat3}(4.0)=\begin{pmatrix} 4.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 4.0 \end{pmatrix}$$

Samplers: used to represent texture

Data Referencing

Standard referencing:

- float green = color[1];
- float m12 = mat[row=1][column=2]

Component access

- float green = color.g;
- float velocity_x = velocity.x

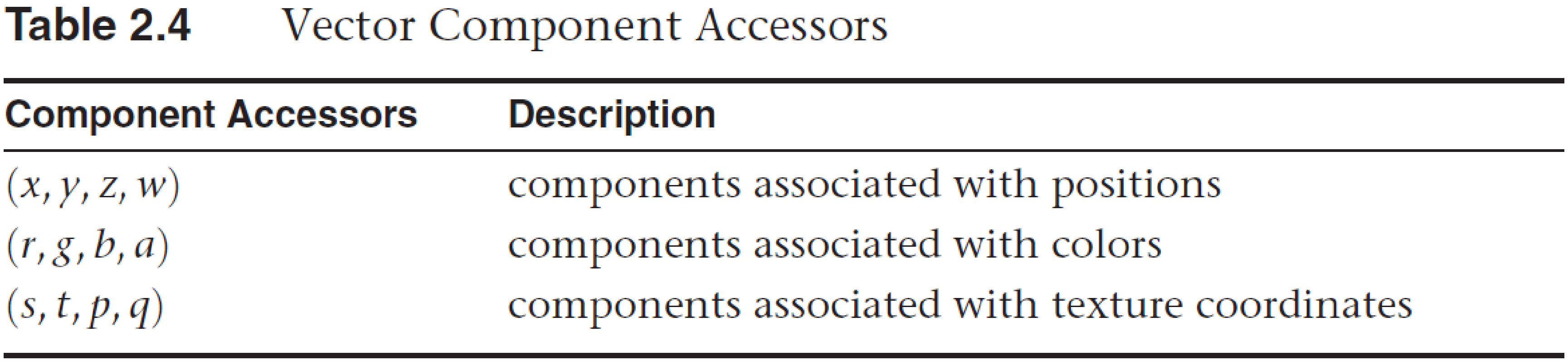


Table 2.4 Vector Component Accessors

Component Accessors	Description
(x, y, z, w)	components associated with positions
(r, g, b, a)	components associated with colors
(s, t, p, q)	components associated with texture coordinates

Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

Swizzling

Can refer to array elements by element using [] or selection (.) operator

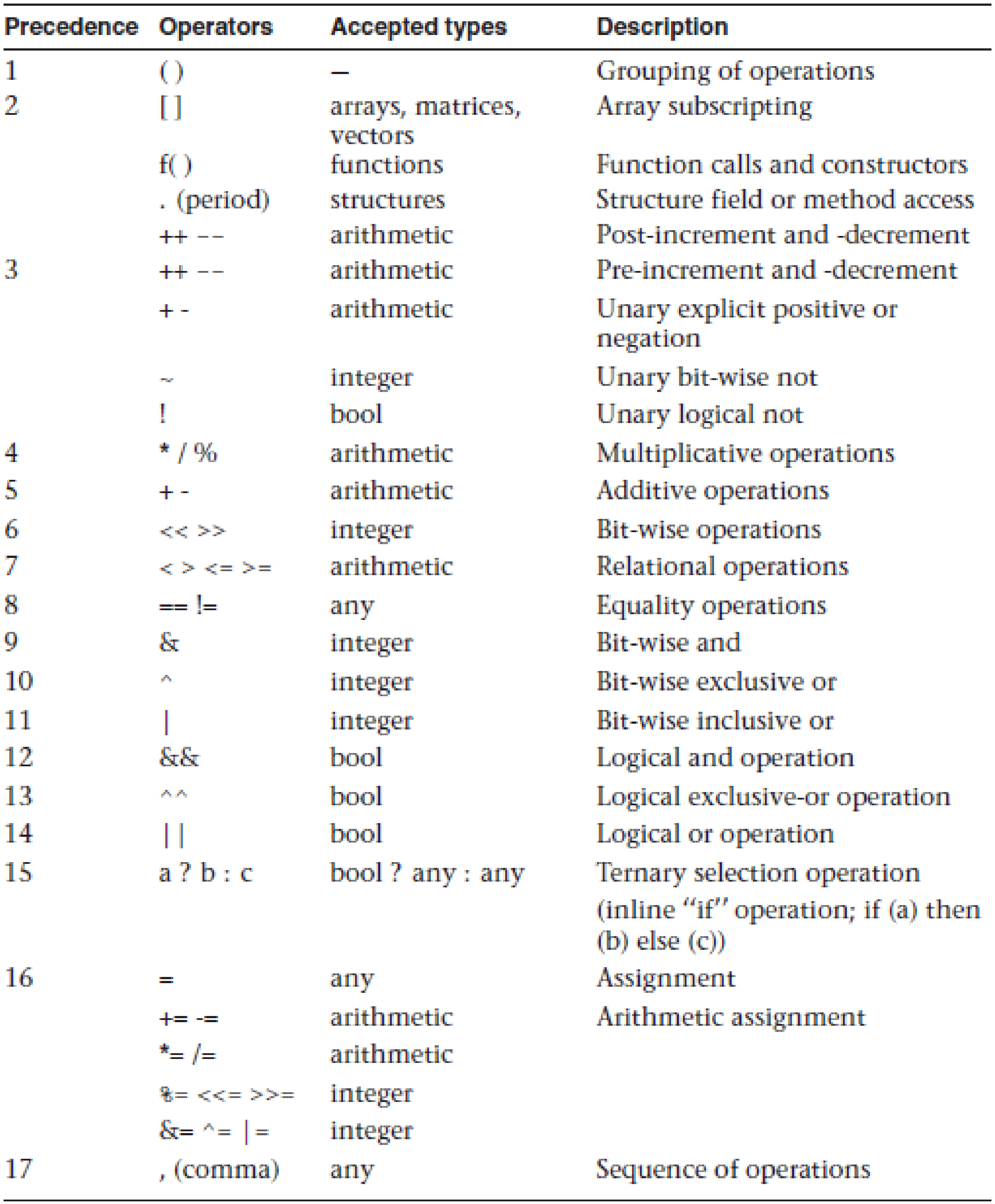
`a[2], a.b, a.z, a.p` are the same

```
vec3 green=color.ggg

vec4 revcolor=color.abgr
vec3 a; a.yz=color.gr

vec3 vector1=color.rrg
vec3 vector2=color.zrg
```

Operators



Precedence	Operators	Accepted types	Description
1	( )	—	Grouping of operations
2	[ ]	arrays, matrices, vectors	Array subscripting
	f( )	functions	Function calls and constructors
	. (period)	structures	Structure field or method access
	++ --	arithmetic	Post-increment and -decrement
3	++ --	arithmetic	Pre-increment and -decrement
	+ -	arithmetic	Unary explicit positive or negation
	~	integer	Unary bit-wise not
	!	bool	Unary logical not
4	* / %	arithmetic	Multiplicative operations
5	+ -	arithmetic	Additive operations
6	<< >>	integer	Bit-wise operations
7	< > <= >=	arithmetic	Relational operations
8	== !=	any	Equality operations
9	&	integer	Bit-wise and
10	^	integer	Bit-wise exclusive or
11	|	integer	Bit-wise inclusive or
12	&&	bool	Logical and operation
13	^^	bool	Logical exclusive-or operation
14	||	bool	Logical or operation
15	a ? b : c	bool ? any : any	Ternary selection operation (inline "if" operation; if (a) then (b) else (c))
16	=	any	Assignment
	+= -=	arithmetic	Arithmetic assignment
	*= /=	arithmetic	
	%= <<= >>=	integer	
	&= ^= |=	integer	
17	, (comma)	any	Sequence of operations

Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

Operators and Built-in Functions

Built-in functions

- Arithmetic, e.g., pow(), exp(), sqrt(), etc.
- Trigonometric, e.g., sin(), cos(), etc.
- Matrix functions, e.g., transpose(), inverse(), etc.
- Many more: normalize(), reflect(), length(), distance(), etc.

Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Example of Vertex Shader: Geometric Transformation

```
#version 330 core
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform mat4 ModelViewProjectionMatrix;
void main()
{
color = vColor;
gl_Position = ModelViewProjectionMatrix * vPosition;
}
```

Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

Type Qualifier

Define and modify the behavior of variables

- **Storage qualifiers: where the data come from**
  - const: read-only, must be initialized when declared
  - in: vertex attributes or from the previous stage
  - out: output from the shader
  - uniform: a global variable shared between all the shader stages
  - buffer: share buffer with application (r/w)

Copy in/out data

- **Layout qualifiers: the storage location**
- **Invariant/precise qualifiers: enforcing the reproducibility**

Uniform Qualifiers

A global variable

- Used to pass information to shader such as the bounding box and the transformation matrix of a primitive
- shared between all the shader stages
- Can be changed in the application and sent to shaders
- Cannot be changed in shader

How to set the value for uniform qualifiers?

GLSL compiler creates a table for all uniform variable when linking the program.

Step1: You need to get the index of the variable by

glGetUniformLocation()

The index is not changed unless relinking the program

Step2: set the value using

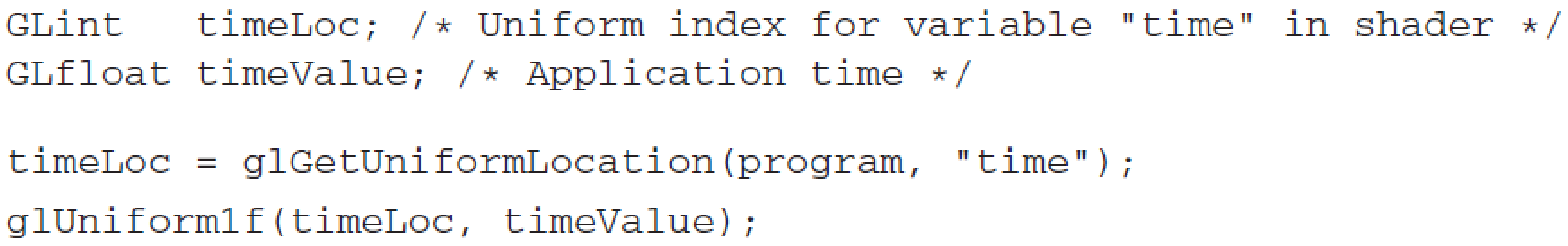
glUniform*() or glUniformMatrix*()

An Example of Uniform Qualifiers (Shreiner et al)

In the shader

uniform GLint time;

In the application



```
GLint   timeLoc; /* Uniform index for variable "time" in shader */
GLfloat timeValue; /* Application time */

timeLoc = glGetUniformLocation(program, "time");
glUniform1f(timeLoc, timeValue);
```

Example of Vertex Shader: Wave Motion Vertex Shader

```
in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h; // height scale
void main()
{
  vec4 t = vPosition;
  t.y = vPosition.y
     + h*sin(time + xs*vPosition.x)
     + h*sin(time + zs*vPosition.z);
  gl_Position = t;
}
```

Example of Vertex Shader: Particle System

```
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;
void main()
{
vec3 object_pos;
object_pos.x = vPosition.x + vel.x*t;
object_pos.y = vPosition.y + vel.y*t
       + g/(2.0*m)*t*t;
object_pos.z = vPosition.z + vel.z*t;
gl_Position =
  ModelViewProjectionMatrix*vec4(object_pos,1);
}
```

Double Buffering

Updating the value of a uniform variable opens the door to animating an application

- Execute glUniform in display callback
- Force a redraw through glutPostRedisplay()

Need to prevent a partially redrawn frame buffer from being displayed ⟹ **Double buffering**

Draw into back buffer

Display front buffer

Swap buffers after updating finished

Adding Double Buffering

Request a double buffer

- glutInitDisplayMode(GLUT_DOUBLE)

Swap buffers

```
void mydisplay()
{
        glClear(……);
        glDrawArrays();
        glutSwapBuffers();
}
```

Compiling Shaders

For each shader object,

Step1: create a shader object

```
GLuint glCreateShader(GLenum type);
```

Type: GL_VERTEX_SHADER and

GL_FRAGMENT_SHADER

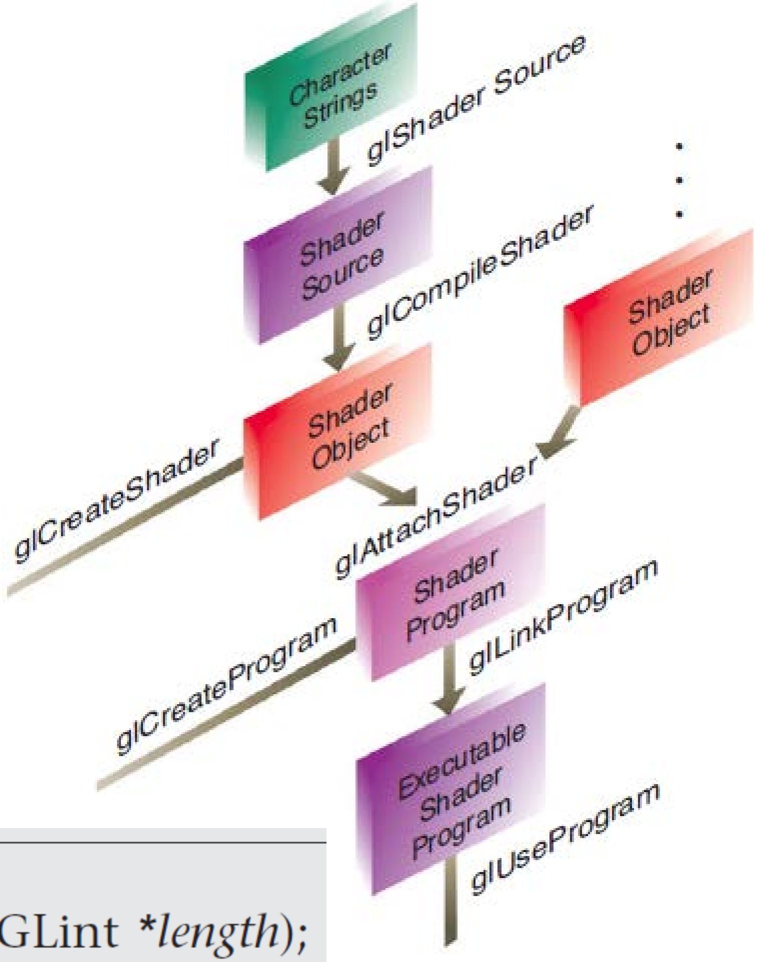
Step2: read the shader source

Step3: associate the shader source

with the shader object

```
void glShaderSource(GLuint shader, GLsizei count,
                    const GLchar **string, const GLint *length);
```

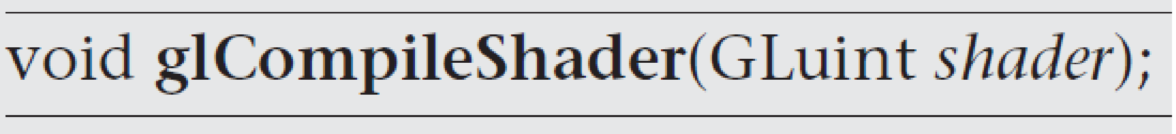
Shader source



Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

Compiling Shaders

Step4: compile a shader object



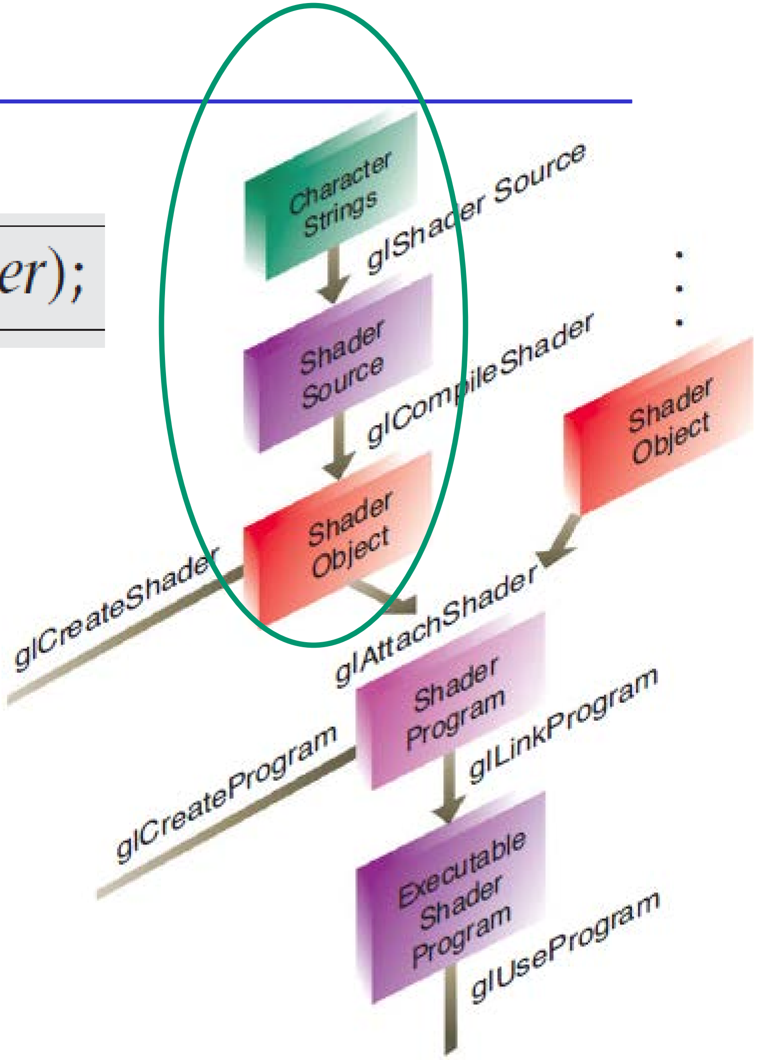
void **glCompileShader**(GLuint *shader*);

Step5: verify the shader compiled successfully

GLint compiled;

glGetShaderiv(shader,

GL_COMPILE_STATUS, &compiled);



Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

Linking Shaders

Step1: create a shader program

- Can contain multiple shaders

```
GLuint glCreateProgram(void);
```

Step2: attach the shader objects to program

```
void glAttachShader(GLuint program, GLuint shader);
```

Step3: link the shader program

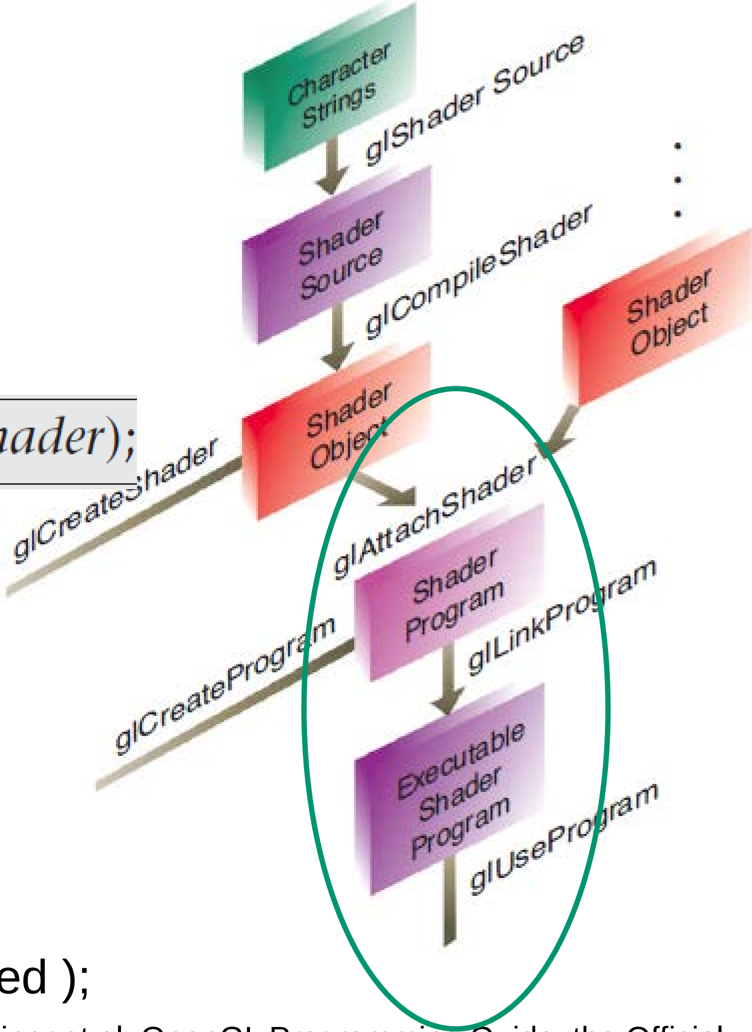
```
void glLinkProgram(GLuint program);
```

Step4: verify the link is successful

```
GLint linked;

glGetProgramiv( program, GL_LINK_STATUS, &linked );
```

Step5: use the shader program



Shreiner et al, OpenGL Programming Guide, the Official Guide to Learning OpenGL, Version 4.3, the 8th Ed,

init()

```
ShaderInfo  shaders[] = {
        { GL_VERTEX_SHADER, "triangles.vert" },
        { GL_FRAGMENT_SHADER, "triangles.frag" },
        { GL_NONE, NULL }
    };
```

Initialize the vertex and fragment shaders

```
    GLuint program = LoadShaders(shaders);
    glUseProgram(program);
```

Load, compile and link shaders

Location of shader attributes

```
    glVertexAttribPointer(vPosition, 2, GL_FLOAT,
                          GL_FALSE, 0, BUFFER_OFFSET(0));
    glEnableVertexAttribArray(vPosition);
}
```

Connect shader to a vertex-attribute array

An Example of Adding a Vertex Shader

```
GLuint myProgObj;
myProgObj = glCreateProgram();

GLuint vShader;
GLunit myVertexObj;
GLchar vShaderfile[] = “my_vertex_shader”;
GLchar* vSource = readShaderSource(vShaderFile);
glShaderSource(myVertexObj,1,&vertexShaderFile, NULL);
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
glCompileShader(myVertexObj);
glAttachObject(myProgObj, myVertexObj);

glLinkProgram(myProgObj);
glUseProgram(myProgObj);
```