Comp 410/510

Computer Graphics
Spring 2023

Programming with OpenGL
Part 3: Shaders

Objectives

- Basic shaders
  - Vertex shader
  - Fragment shader
- Programming shaders with GLSL
- Finish with the first program

main.cpp

```
void init(void)
{
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );

    GLfloat  vertices[NumVertices][3] = {{-0.5, -0.5, 0.0},{-0.5, 0.5, 0.0},{0.5, 0.5, 0.0},
                                         {0.5, -0.5, 0.0},{-0.5, -0.5, 0.0},{0.5, 0.5, 0.0}};
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Load shaders and use the resulting shader program
    GLuint program = InitShader("vshader.glsl"," "fshader.glsl");
    glUseProgram( program );
    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}
int main(int argc, char** argv)
{
   initWindowAPI();
   GLFWwindow* window = glfwCreateWindow(500, 500, "Simple", N
   glfwMakeContextCurrent(window);
   init();
   while (!glfwWindowShouldClose(window)){
        display();
        glfwSwapBuffers(window);
        glfwPollEvents();
   }}
```

vshader_simple.glsl

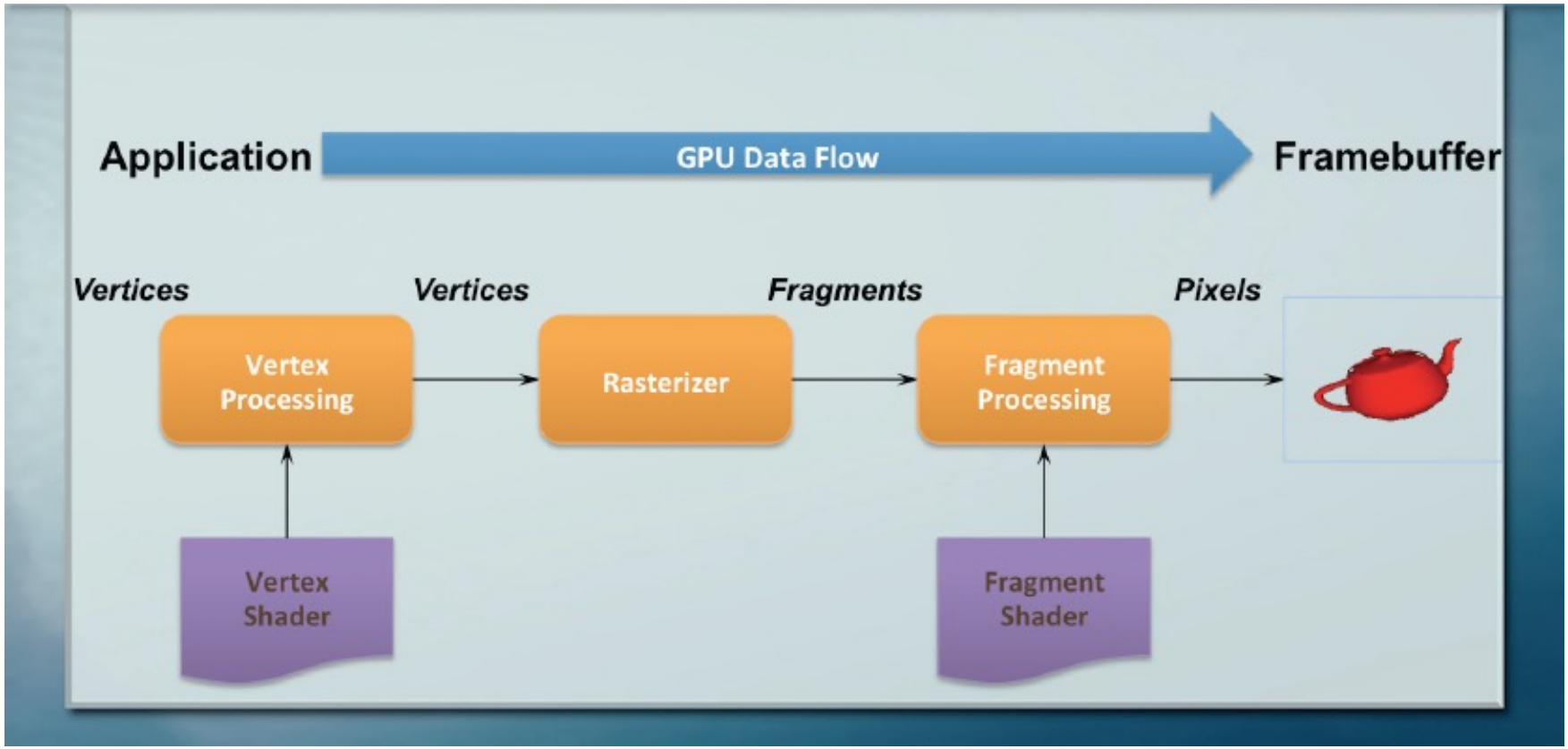
```
in vec4 vPosition;

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

fshader_simple.glsl

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

Graphics Pipeline (Simplified)



Vertex Shader Applications

- Moving vertices
  - Rotation, translation, scale, etc
  - Morphing
  - Wave motion
  - Fractals

- Lighting (per-vertex)
  - More realistic models
  - Cartoon-like shading

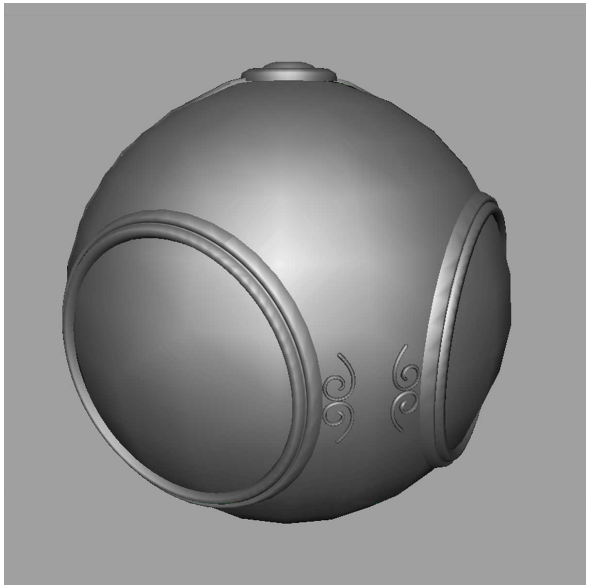
Fragment Shader Applications

per-vertex lighting

per-fragment lighting

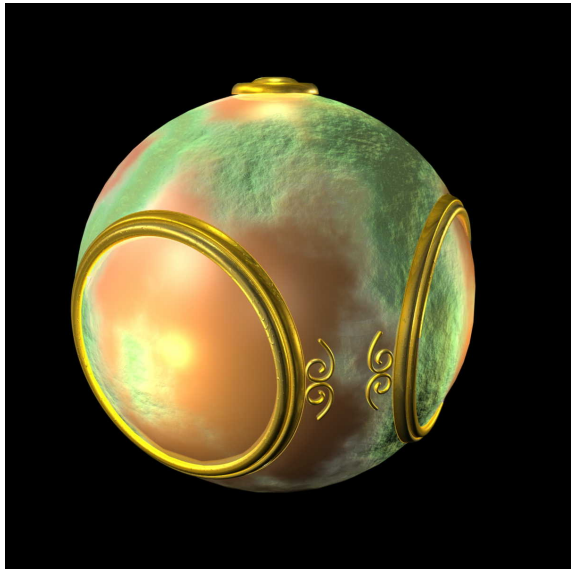
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



bump mapping

Coding Shaders

- First shaders were programmed in an assembly-like manner
- OpenGL extensions were then added for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders (by Nvidia)
  - Worked with both OpenGL and DirectX
  - Interface to OpenGL was complex
  - Deprecated in 2012
- Now, GL Shading Language (GLSL) for OpenGL
- HLSL for DirectX

GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High-level C-like language
- Extra data types
  - Matrices
  - Vectors
  - Samplers
- As of OpenGL 3.1, application **must** provide shaders with GLSL

Simple Vertex Shader

input from application

```
in vec4 vPosition;

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

Remark: `attribute` qualifier is deprecated

Simple Vertex Shader

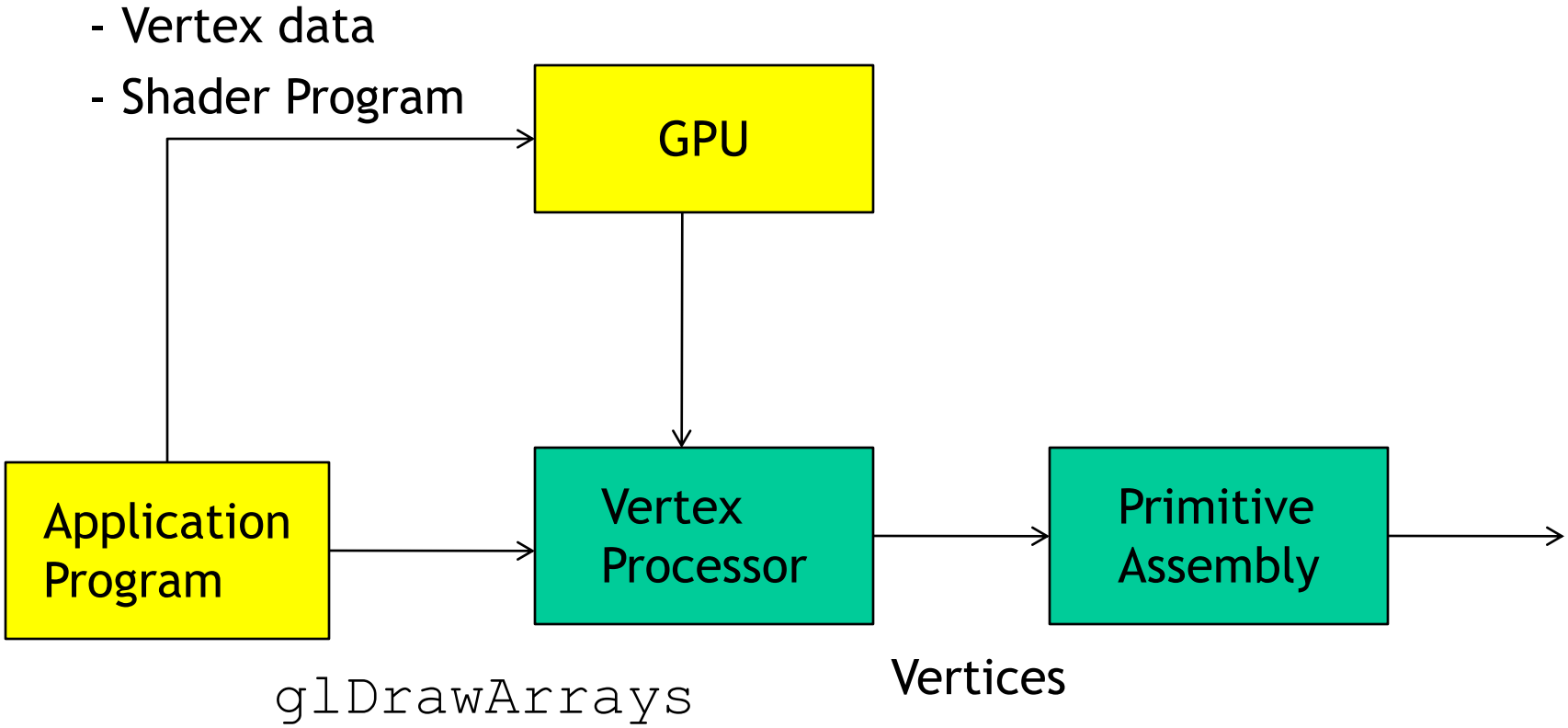
```
in vec4 vPosition;

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

must link to variable in application

built-in state variable

Execution Model



Simple Fragment Shader

output from shader

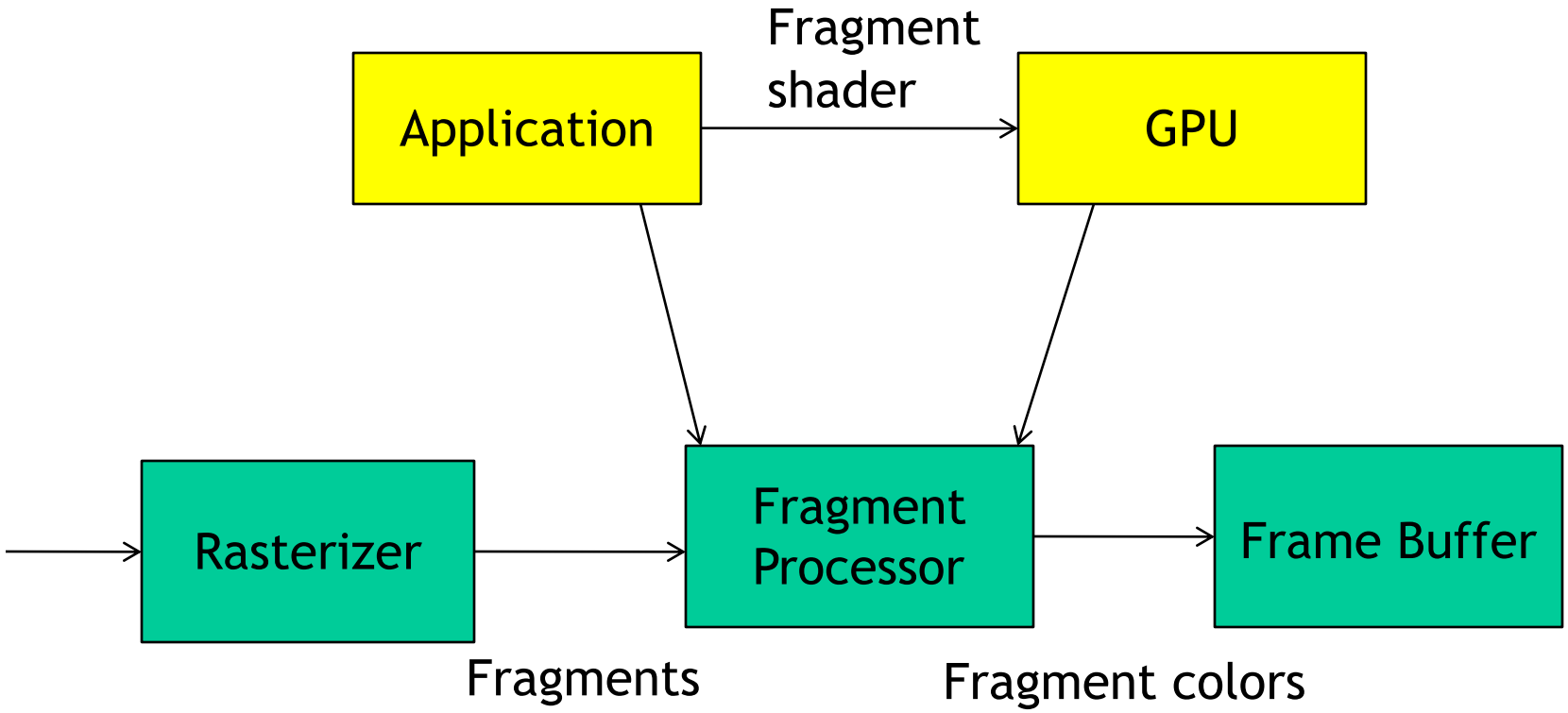
```
out vec4 fragcolor;
void main(void)
{
    fragcolor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Simple Fragment Shader (old)

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

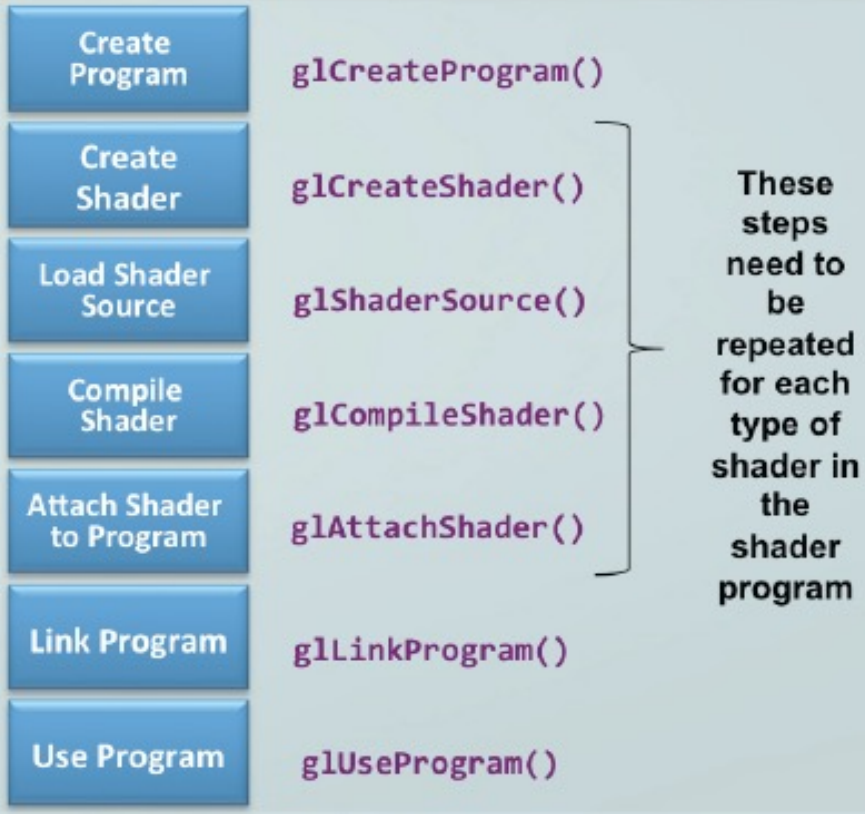
was built-in variable;
removed as of OpenGL 3.1

Execution Model



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



All at the runtime! (see `init` function)

main.cpp

```
void init(void)
{
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );
    GLfloat  vertices[NumVertices][3] = {{-0.5, -0.5, 0.0},{-0.5, 0.5, 0.0},{0.5, 0.5, 0.0},
                                         {0.5, -0.5, 0.0},{-0.5, -0.5, 0.0},{0.5, 0.5, 0.0}};
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Load shaders and use the resulting shader program
    GLuint program = InitShader("vshader.glsl"," "fshader.glsl");
    glUseProgram( program );

    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}
int main(int argc, char** argv)
{
   initWindowAPI();
   GLFWwindow* window = glfwCreateWindow(500, 500, "Simple", NULL, NULL);
   glfwMakeContextCurrent(window);
   init();
   while (!glfwWindowShouldClose(window)){
        display();
        glfwSwapBuffers(window);
        glfwPollEvents();
   }}
```

vshader_simple.glsl

```
in vec4 vPosition;

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

fshader_simple.glsl

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

Linking Shaders with Application

- Read shaders `glShaderSource`
- Compile shaders `glCompileShader`
- Create a program object containing shaders `glAttachShader`
- Link everything together `glLinkProgram`

Adding a Vertex Shader

```
GLuint myVShaderObj;
GLchar vShaderfile[] = “my_vertex_shader”;
GLchar* vSource =
        readShaderSource(vShaderFile);
myVShaderObj =
          glCreateShader(GL_VERTEX_SHADER);
glShaderSource(myVShaderObj, 1, &vSource, NULL);
glCompileShader(myVShaderObj);
glAttachShader(myProgObj, myVShaderObj);
```

see InitShader.cpp provided by the textbook code

```
GLuint InitShader(const char* vShaderFile, const char* fShaderFile)
    {
        struct Shader {
            const char*  filename;
            ...
        };

        GLuint program = glCreateProgram();

        for ( int i = 0; i < 2; ++i ) {
            Shader& s = shaders[i];
            s.source = readShaderSource( s.filename );
            ...
            }

            GLuint shader = glCreateShader( s.type );
            glShaderSource( shader, 1, (const GLchar**) &s.source, NULL );
            glCompileShader( shader );

            GLint  compiled;
            glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
            …
            glAttachShader( program, shader );
        }

        /* link  and error check */
        glLinkProgram(program);
        …
        /* use program object */
        glUseProgram(program);

        return program;
    }
```

InitShader.cpp

Reading a Shader

- Shaders are added to the program object and compiled in the runtime
- Usual method of passing a shader is as a null-terminated string using the function `glShaderSource`
- If the shader is in a file, we can write a reader to convert the file to a string (see `readShaderSource`)

```
const char* vertexShaderSource = "#version 410 \n in vec4
vPosition;\nvoid main()\n{\ngl_Position = vPosition;\n}\0";
```

InitShader.cpp

```
GLuint InitShader(const char* vShaderFile, const char* fShaderFile)
    {
        struct Shader {
            const char*  filename;
            ...
        };

        GLuint program = glCreateProgram();

        for ( int i = 0; i < 2; ++i ) {
            Shader& s = shaders[i];
            s.source = readShaderSource( s.filename );
            ...
            }

            GLuint shader = glCreateShader( s.type );
            glShaderSource( shader, 1, (const GLchar**) &s.source, NULL );
            glCompileShader( shader );

            GLint  compiled;
            glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
            …
            glAttachShader( program, shader );
        }

        /* link  and error check */
        glLinkProgram(program);
        …
        /* use program object */
        glUseProgram(program);

        return program;
    }
```

Shader Reader

```
static char*
readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");
    if ( fp == NULL ) { return NULL; }
    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);
    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);

    buf[size] = '\0';
    fclose(fp);

    return buf;
}
```

Program Object

- A container for shaders
  - can contain multiple shaders
  - managed by GLSL related functions

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

 /* define shader objects here */

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

Linking Shaders

Have to link variables in the application with variables in shaders

- Vertex attributes
- Uniform variables

main.cpp

```
void init(void)
{
    GLuint vao;
    glGenVertexArrays( 1, &vao );
    glBindVertexArray( vao );
    GLfloat  vertices[NumVertices][3] = {{-0.5, -0.5, 0.0},{-0.5, 0.5, 0.0},{0.5, 0.5, 0.0},
                                         {0.5, -0.5, 0.0},{-0.5, -0.5, 0.0},{0.5, 0.5, 0.0}};
    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Load shaders and use the resulting shader program
    GLuint program = InitShader("vshader.glsl"," "fshader.glsl");

    GLuint loc = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( loc );
    glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
    glUseProgram( program );
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();
}
int main(int argc, char** argv)
{
   initWindowAPI();
   GLFWwindow* window = glfwCreateWindow(500, 500, "Simple", NULL, NULL);
   glfwMakeContextCurrent(window);
   init();
   while (!glfwWindowShouldClose(window)){
        display();
        glfwSwapBuffers(window);
        glfwPollEvents();
   }}
```

vshader_simple.glsl

```
in vec4 vPosition;

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

Associate shader variables with vertex attributes

- Vertex attributes are named in the shaders
- Linker forms a table via `glLinkProgram()`
- Application can get index from table and tie it to an application variable (`loc`)
- Similar process for uniform variables

```
GLuint loc = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0) );
```

vshader_simple.glsl

```
in vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
```

```
#define BUFFER_OFFSET( offset ) ((GLvoid*) (offset))
```

Data Types

- C types: `int, float, bool`

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

- Matrices: `mat2, mat3, mat4`
  - Stored by columns
  - Standard referencing `m[row][column]`

- C++ style constructors
  - `vec3 a = vec3(1.0, 2.0, 3.0)`
  - `vec2 b = vec2(a)`

Pointers

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types, they can be passed into and out from GLSL functions, e.g.

```
mat3 func(mat3 a)
```

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length

- Overloading of vector and matrix operations

```
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
```

Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - `x, y, z, w`
  - `r, g, b, a`
  - `s, t, p, q`
  - `a[2], a.b, a.z, a.p` are the same

- Swizzling operator lets us manipulate components

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Qualifiers

- GLSL has many of the same qualifiers such as `const` as C/C++
- Need also other qualifiers due to the nature of the execution model such as
  - `in, out, uniform`
  - `varying, attribute` (deprecated or removed)
- Variables (depending on the qualifier type) can change value
  - per vertex
  - per fragment
  - at any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualified

- Attribute-qualified variables can change per vertex or per fragment, such as `in`
- There are a few built-in variables such as `gl_Position` but most others have been deprecated
- User defined (in application program)
  - We use `in` qualifier to get to shader
    - `in float temperature`
    - `in vec3 velocity`

Varying Qualified

- Variables that are passed from vertex shader to fragment shader, such as `out`
- Automatically interpolated by the rasterizer
- Old style used the `varying` qualifier:

  `varying vec4 color;` ✗

- Now use `out` in vertex shader and `in` in the fragment shader:

  `out vec4 color;`

Example: Vertex Shader

```
in vec4 vPosition;
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec4 color_out;

void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

Required Fragment Shader

```
in vec4 color_out;
out vec4 fragcolor;

void main(void)
{
    fragcolor = color_out;
}
```

Uniform Qualifier

- Variables that are constant among all processed vertices or fragments, such as `uniform`
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the projection matrix (which does not change per vertex)

Uniform Variable Example

```
GLint angleParam;
angleParam = glGetUniformLocation(myProgObj,"angle");
/* angle defined in shader */

/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */

glUniform1f(angleParam, my_angle);
```

Vertex Shader Applications

- Moving vertices
  - Rotation, translation, scale, etc
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Wave Motion Vertex Shader

```
in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h; // height scale
uniform float time;

void main()
{
  vec4 t = vPosition;
  t.y = vPosition.y
     + h*sin(time + xs*vPosition.x)
     + h*sin(time + zs*vPosition.z);
  gl_Position = t;
}
```

Particle System Vertex Shader

```
in vec3 vPosition;
uniform mat4 ModelView;
uniform mat4 Projection;
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 =
     ModelView*Projection*vec4(object_pos,1);
}
```

Built-in Shader Variables

- gl_Position
  - Output vertex position from vertex shader
- gl_ FragCoord
  - Input fragment position in fragment shader
- gl_FragDepth
  - Input depth value in fragment shader
- others