

CSE 167: Introduction to Computer Graphics Lecture #12: GLSL

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Spring Quarter 2015

Announcements

- ▶ Project 6 due Friday at 1pm

GLSL

- ▶ Real Time 3D Demo C++/OpenGL/GLSL Engine
<http://www.youtube.com/watch?v=9N-kgCqy2xs>



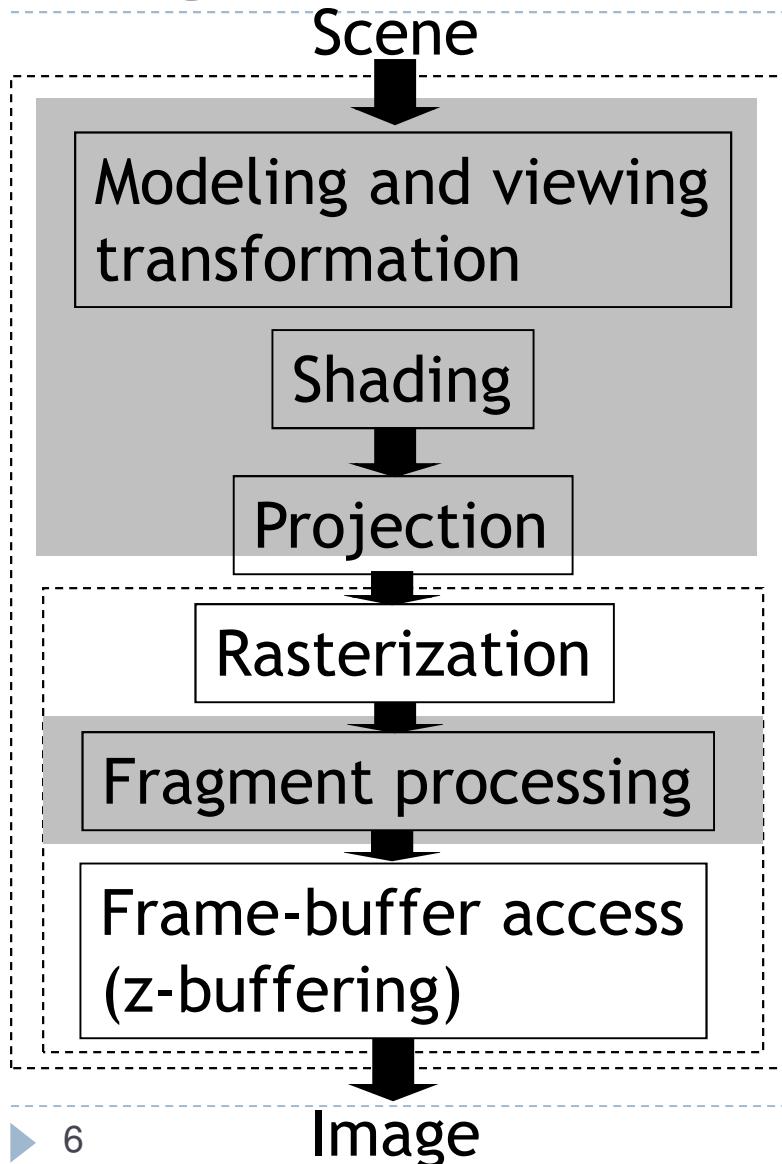
Lecture Overview

- ▶ **Programmable Shaders**
 - ▶ Vertex Programs
 - ▶ Fragment Programs
 - ▶ GLSL

Shader Programs

- ▶ Programmable shaders consist of shader programs
- ▶ Written in a **shading language**
 - ▶ Syntax similar to C language
- ▶ Each shader is a separate piece of code in a separate ASCII text file
- ▶ Shader types:
 - ▶ Vertex shader
 - ▶ Tessellation shader
 - ▶ Geometry shader
 - ▶ Fragment shader (a.k.a. pixel shader)
- ▶ The programmer can provide any number of shader types to work together to achieve a certain effect
- ▶ If a shader type is not provided, OpenGL's fixed-function pipeline is used

Programmable Pipeline



- ▶ Executed once per vertex:
 - ▶ Vertex Shader
 - ▶ Tessellation Shader
 - ▶ Geometry Shader

- ▶ Executed once per fragment:
 - ▶ Fragment Shader

Vertex Shader

- ▶ Executed once per vertex
- ▶ Cannot create or remove vertices
- ▶ Does not know the primitive it belongs to
- ▶ Replaces functionality for
 - ▶ Model-view, projection transformation
 - ▶ Per-vertex shading
- ▶ If you use a vertex program, you need to implement behavior for the above functionality in the program!
- ▶ Typically used for:
 - ▶ Character animation
 - ▶ Particle systems

Tessellation Shader

- ▶ Executed once per primitive
- ▶ Generates new primitives by subdividing each line, triangle or quad primitive
- ▶ Typically used for:
 - ▶ Adapting visual quality to the required level of detail
 - ▶ For instance, for automatic tessellation of Bezier curves and surfaces
 - ▶ Geometry compression: 3D models stored at coarser level of resolution, expanded at runtime
 - ▶ Allows detailed displacement maps for less detailed geometry

Geometry Shader

- ▶ Executed once per primitive (triangle, quad, etc.)
- ▶ Can create new graphics primitives from output of tessellation shader (e.g., points, lines, triangles)
 - ▶ Or can remove the primitive
- ▶ Typically used for:
 - ▶ Per-face normal computation
 - ▶ Easy wireframe rendering
 - ▶ Point sprite generation
 - ▶ Shadow volume extrusion
 - ▶ Single pass rendering to a cube map
 - ▶ Automatic mesh complexity modification (depending on resolution requirements)

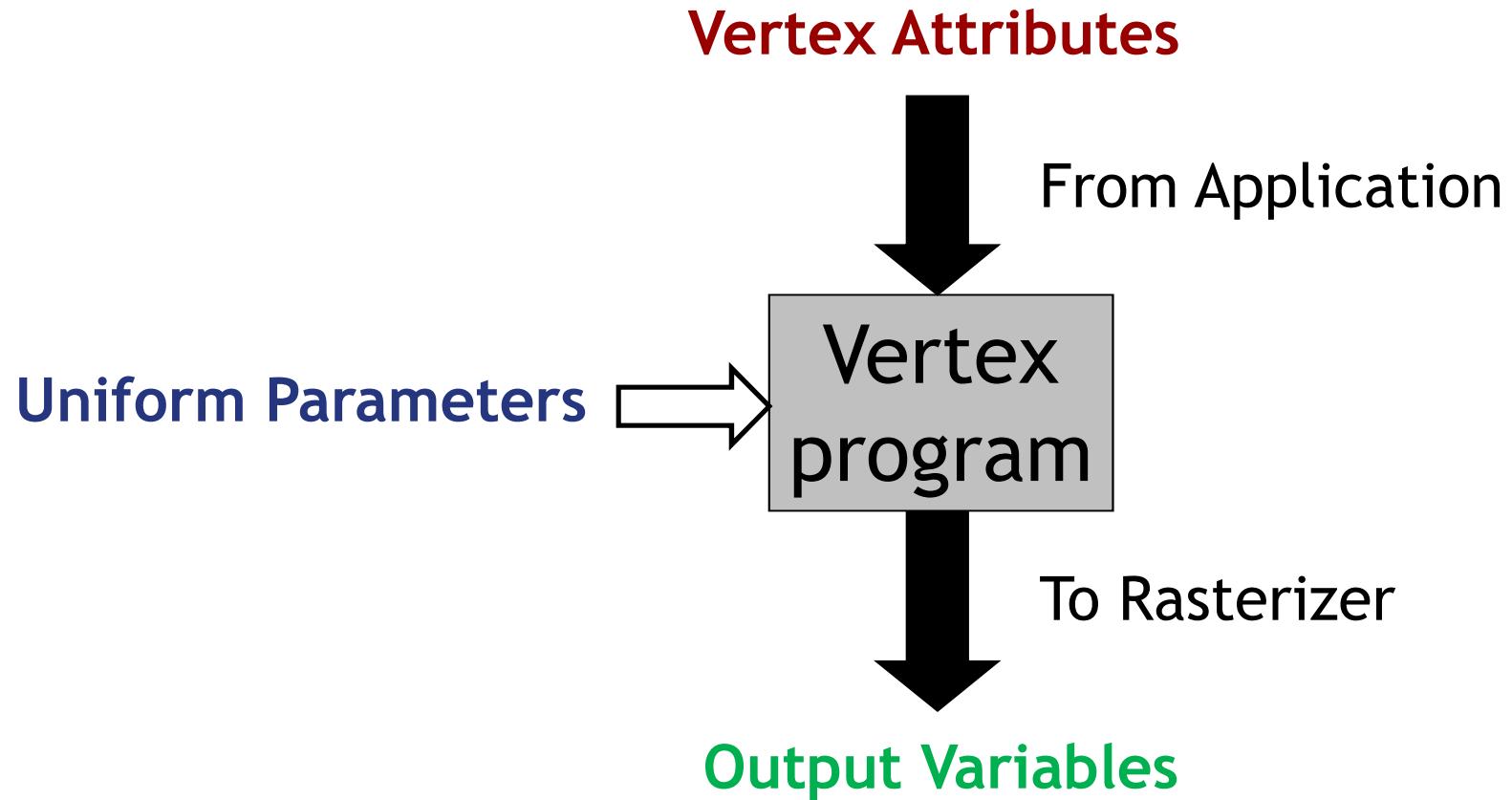
Fragment Shader

- ▶ A.k.a. Pixel Shader
- ▶ Executed once per fragment
- ▶ Cannot access other pixels or vertices
 - ▶ Makes execution highly parallelizable
- ▶ Computes color, opacity, z-value, texture coordinates
- ▶ Typically used for:
 - ▶ Per-pixel shading (e.g., Phong shading)
 - ▶ Advanced texturing
 - ▶ Bump mapping
 - ▶ Shadows

Lecture Overview

- ▶ Programmable Shaders
 - ▶ Vertex Programs
 - ▶ Fragment Programs
 - ▶ GLSL

Vertex Programs



Vertex Attributes

- ▶ Declared using the attribute **storage classifier**
- ▶ Different for each execution of the **vertex program**
- ▶ Can be modified by the **vertex program**
- ▶ Two types:
 - ▶ Pre-defined OpenGL attributes. Examples:

```
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
```
 - ▶ User-defined attributes. Example:

```
attribute float myAttrib;
```

Uniform Parameters

- ▶ Declared by `uniform` storage classifier
- ▶ Normally the same for all vertices
- ▶ Read-only
- ▶ Two types:
 - ▶ Pre-defined OpenGL state variables
 - ▶ User-defined parameters

Uniform Parameters: Pre-Defined

- ▶ Provide access to the OpenGL state
- ▶ Examples for pre-defined variables:

```
uniform mat4 gl_ModelViewMatrix;  
uniform mat4 gl_ModelViewProjectionMatrix;  
uniform mat4 gl_ProjectionMatrix;  
uniform gl_LightSourceParameters  
    gl_LightSource[gl_MaxLights];
```

Uniform Parameters: User-Defined

- ▶ Parameters that are set by the application
- ▶ Should not be changed frequently
 - ▶ Especially not on a per-vertex basis!
- ▶ To access, use `glGetUniformLocation`, `glUniform*` in application
- ▶ Example:
 - ▶ In shader declare

```
uniform float a;
```
 - ▶ Set value of a in application:

```
GLuint p;  
int i = glGetUniformLocation(p, "a");  
glUniform1f(i, 1.0f);
```

Vertex Programs: Output Variables

- ▶ Required output: homogeneous vertex coordinates

```
vec4 gl_Position
```

- ▶ **varying** output variables

- ▶ Mechanism to send data to the fragment shader
- ▶ Will be interpolated during rasterization
- ▶ Fragment shader gets interpolated data

- ▶ Pre-defined **varying** output variables, for example:

```
varying vec4 gl_FrontColor;  
varying vec4 gl_TexCoord[];
```

Any pre-defined output variable that you do not overwrite will have the value of the OpenGL state.

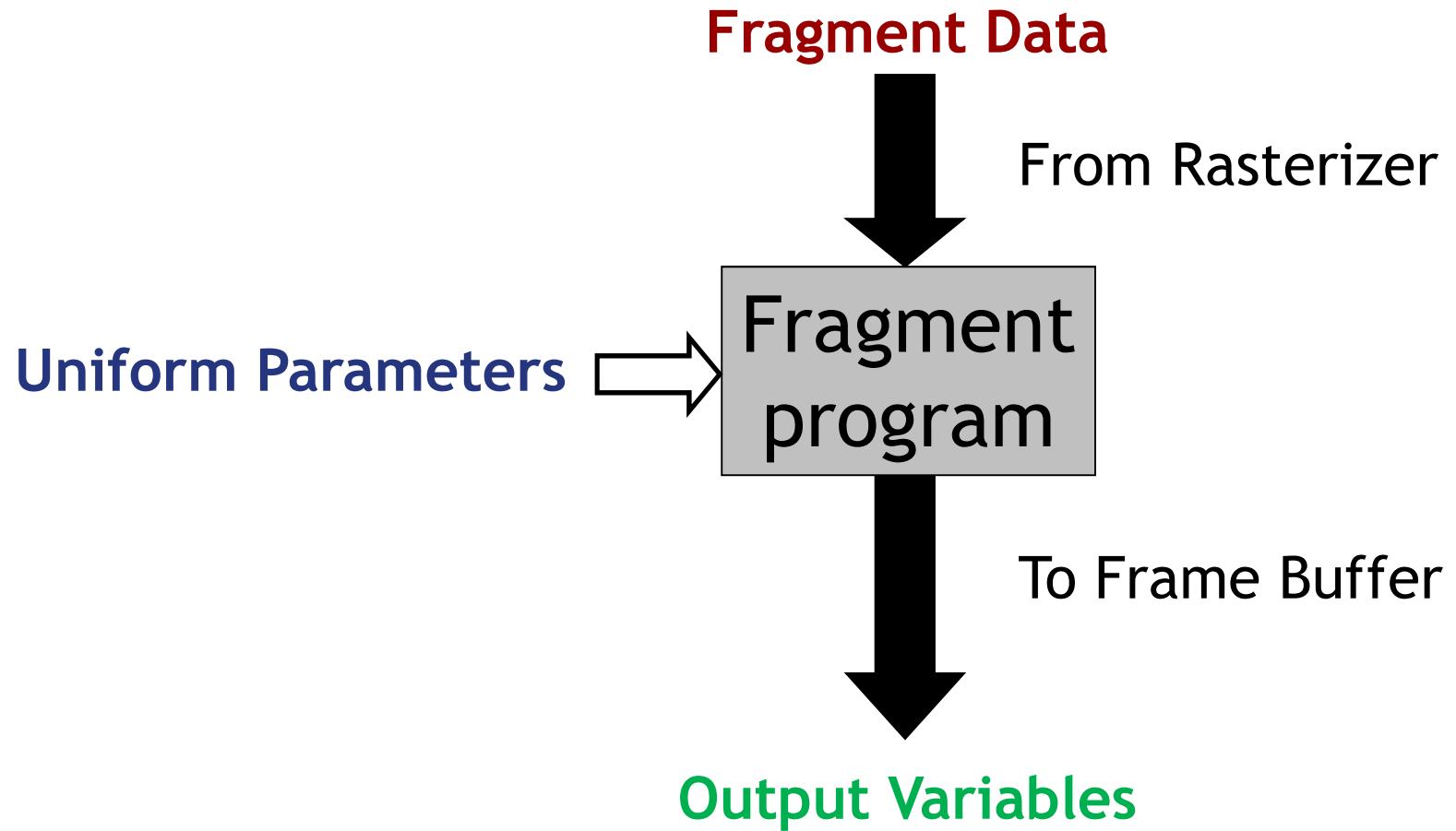
- ▶ User-defined **varying** output variables, e.g.:

```
varying vec4 vertex_color;
```

Lecture Overview

- ▶ Programmable Shaders
 - ▶ Vertex Programs
 - ▶ Fragment Programs
 - ▶ GLSL

Fragment Programs



Fragment Data

- ▶ Changes for each execution of the fragment program
- ▶ Fragment data includes:
 - ▶ Interpolated standard OpenGL variables for fragment shader, as generated by vertex shader, for example:

```
varying vec4 gl_Color;  
varying vec4 gl_TexCoord[];
```
 - ▶ Interpolated varying variables from vertex shader
 - ▶ Allows data to be passed from vertex to fragment shader

Uniform Parameters

- ▶ Same as in vertex programs

Output Variables

- ▶ Pre-defined output variables:
 - ▶ gl_FragColor
 - ▶ gl_FragDepth
- ▶ OpenGL writes these to the frame buffer
- ▶ Result is undefined if you do not set these variables!

Simple GLSL Shader Example

Vertex Shader

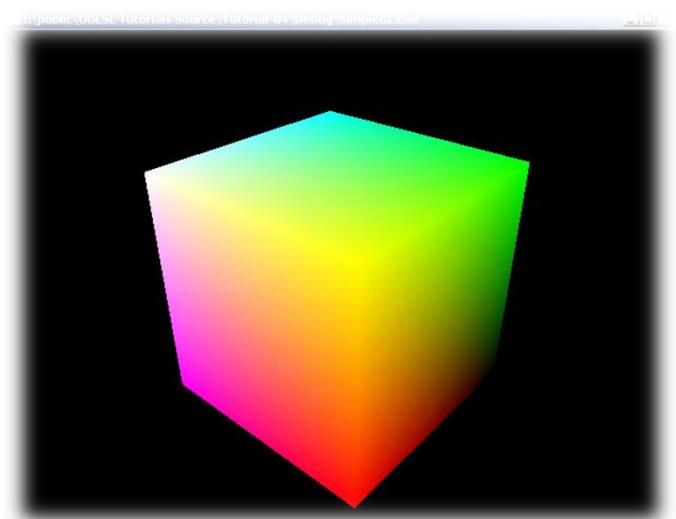
```
varying float xpos;
varying float ypos;
varying float zpos;
void main(void)
{
    xpos = clamp(gl_Vertex.x, 0.0, 1.0);
    ypos = clamp(gl_Vertex.y, 0.0, 1.0);
    zpos = clamp(gl_Vertex.z, 0.0, 1.0);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

```
varying float xpos;
varying float ypos;
varying float zpos;

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



William H. Hsu

► 23 Department of Computing and Information Sciences,
KSU



Diffuse Shader

Vertex Shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Transforming The Vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Transforming The Normal To ModelView-Space
    normal = gl_NormalMatrix * gl_Normal;

    // Transforming The Vertex Position To ModelView-Space
    vec4 vertex_in_modelview_space = gl_ModelViewMatrx * gl_Vertex;

    // Calculating The Vector From The Vertex Position To The Light Position
    vertex_to_light_vector = vec3(gl_LightSource[0].position - vertex_in_modelview_space);
}
```

F. Rudolf, NeHe Productions

Diffuse Shader

▶ Fragment Shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Defining The Material Colors
    const vec4 AmbientColor = vec4(0.1, 0.0, 0.0, 1.0);
    const vec4 DiffuseColor = vec4(1.0, 0.0, 0.0, 1.0);

    // Scaling The Input Vector To Length 1
    vec3 normalized_normal = normalize(normal);
    vec3 normalized_vertex_to_light_vector = normalize(vertex_to_light_vector);

    // Calculating The Diffuse Term And Clamping It To [0;1]
    float DiffuseTerm = clamp(dot(normal, vertex_to_light_vector), 0.0, 1.0);

    // Calculating The Final Color
    gl_FragColor = AmbientColor + DiffuseColor * DiffuseTerm;
}
```

F. Rudolf, NeHe Productions

Exercise

Vertex Shader

```
void main(void)
{
    vec4 a = gl_Vertex;
    a.x = a.x * 0.5;
    a.y = a.y * 0.5;
    gl_Position = gl_ModelViewProjectionMatrix * a;
}
```

Q: What does this do?

A: Incoming x and y components are scaled with a factor 0.5;
Scaled vertex is transformed with concatenated modelview and projection matrix.

Fragment Shader

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

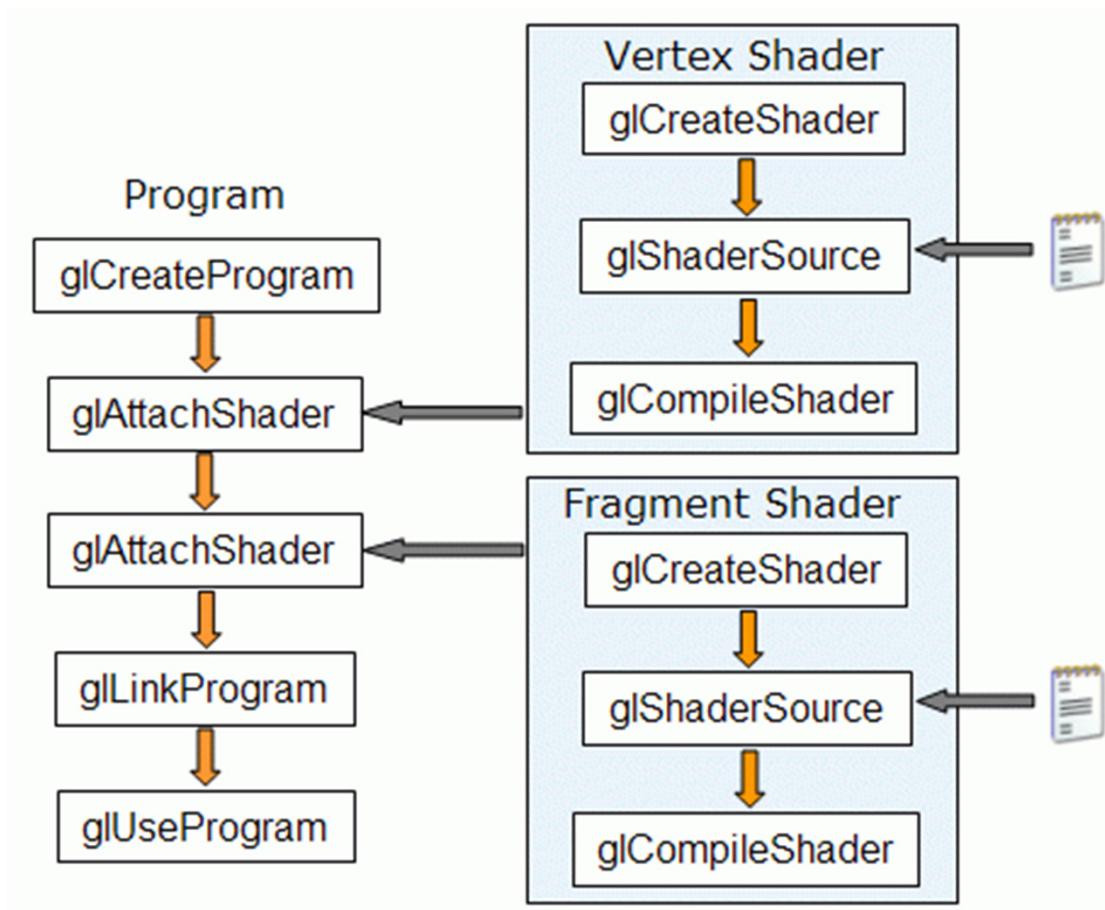
Q: What does this do?

A: Makes everything green!



M. Christen, ClockworkCoders.com

Loading Shaders in OpenGL



Gabriel Zachmann, Clausthal University

GLShader.h

```
▶ #ifndef GLSHADER_H  
▶ #define GLSHADER_H  
  
▶ #include "GL/glew.h" // http://glew.sourceforge.net  
  
▶ GLuint LoadShader(const char *vertex_path, const char  
 *fragment_path);  
  
▶ #endif
```

Source: Nexcius.net

GLShader.cpp

```
#include "GLShader.h"

#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

std::string readFile(const char *filePath)
{
    std::string content;
    std::ifstream fileStream(filePath, std::ios::in);
    if(!fileStream.is_open())
    {
        std::cerr << "Could not read file " << filePath << ". File does not exist." <<
                    std::endl;
        return "";
    }

    std::string line = "";
    while(!fileStream.eof())
    {
        std::getline(fileStream, line);
        content.append(line + "\n");
    }

    fileStream.close();
    return content;
}
```

Source: Nexcius.net

GLShader.cpp

```
GLuint LoadShader(const char *vertex_path, const char *fragment_path)
{
    GLuint vertShader = glCreateShader(GL_VERTEX_SHADER);
    GLuint fragShader = glCreateShader(GL_FRAGMENT_SHADER);

    // Read shaders
    std::string vertShaderStr = readFile(vertex_path);
    std::string fragShaderStr = readFile(fragment_path);
    const char *vertShaderSrc = vertShaderStr.c_str();
    const char *fragShaderSrc = fragShaderStr.c_str();

    GLint result = GL_FALSE;
    int logLength;

    // Compile vertex shader
    std::cerr << "Compiling vertex shader." << std::endl;
    glShaderSource(vertShader, 1, &vertShaderSrc, NULL);
    glCompileShader(vertShader);

    // Compile fragment shader
    std::cerr << "Compiling fragment shader." << std::endl;
    glShaderSource(fragShader, 1, &fragShaderSrc, NULL);
    glCompileShader(fragShader);

    std::cerr << "Linking program" << std::endl;
    GLuint program = glCreateProgram();
    glAttachShader(program, vertShader);
    glAttachShader(program, fragShader);
    glLinkProgram(program);

    glGetProgramiv(program, GL_LINK_STATUS, &result);
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &logLength);
    std::vector<char> programError( (logLength > 1) ? logLength : 1 );
    glGetProgramInfoLog(program, logLength, NULL, &programError[0]);
    std::cerr << &programError[0] << std::endl;

    glDeleteShader(vertShader);
    glDeleteShader(fragShader);

    return program;
}
```

Source: Nexcius.net

main.cpp

```
GLuint program =  
    LoadShader("shader.vert", "shader.frag");  
glUseProgram(program);
```

Source: Nexcius.net

Tutorials and Documentation

- ▶ OpenGL and GLSL Specifications
 - ▶ <https://www.opengl.org/registry/>
- ▶ GLSL Tutorials
 - ▶ <http://www.lighthouse3d.com/opengl/glsl/>
 - ▶ <http://www.clockworkcoders.com/oglsl/tutorials.html>
- ▶ OpenGL Programming Guide (Red Book)
 - ▶ <http://www.glprogramming.com/red/>
- ▶ OpenGL Shading Language (Orange Book)
 - ▶ http://wiki.labomedia.org/images/1/10/Orange_Book_-_OpenGL_Shading_Language_2nd_Edition.pdf
- ▶ OpenGL 3.2 API Reference Card
 - ▶ http://www.opengl.org/sdk/docs/reference_card/opengl32-quick-reference-card.pdf