CSE 167:
Introduction to Computer Graphics
Lecture #3: GLSL

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2020

Announcements

- Homework Project 1 due October 25

GLSL in Practice

- 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

Programmable Shaders in OpenGL

- Initially, OpenGL only had a fixed-function pipeline for shading
- Programmers wanted more flexibility, similar to programmable shaders in raytracing software (term "shader" first introduced by Pixar in 1988)
- First shading languages came out in 2002:
  - **Cg** (C for Graphics, created by Nvidia)
  - **HLSL** (High Level Shader Language, created by Microsoft)
- They supported:
  - **Vertex shaders**: allowed modification of geometry
  - **Fragment shaders**: allowed per-pixel shading

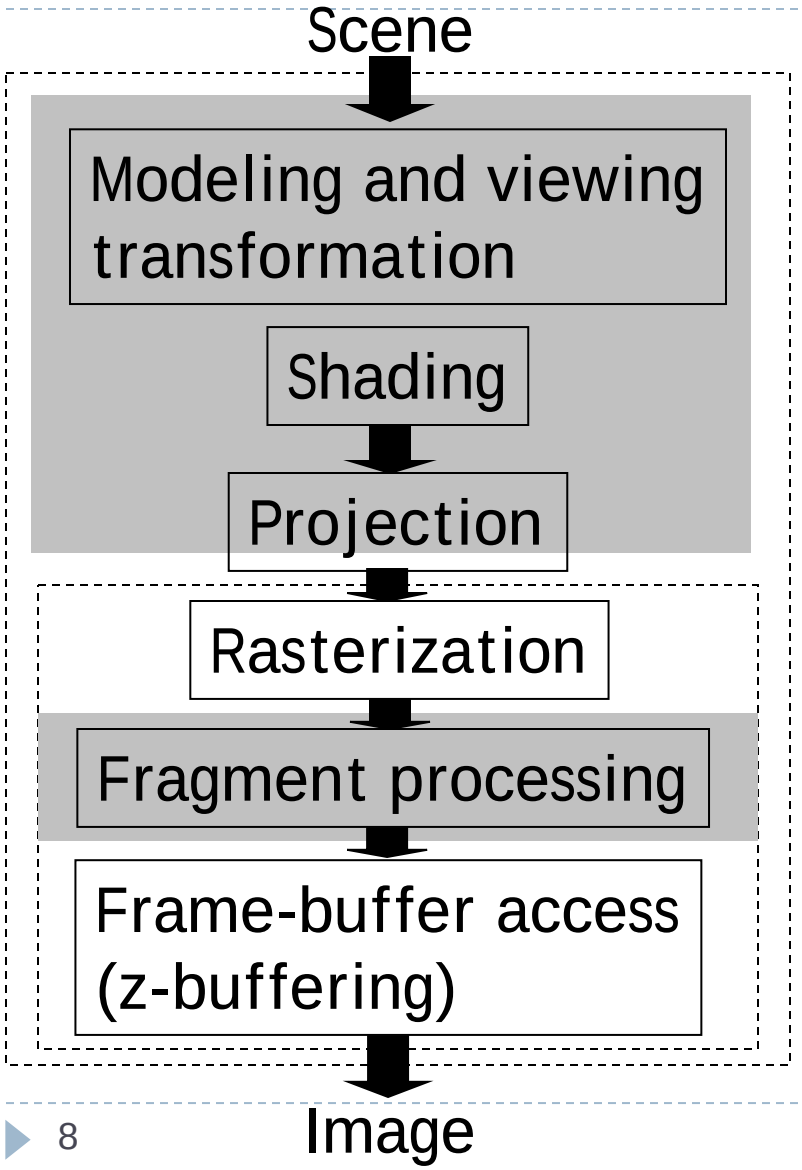
Programmable Shaders in OpenGL

- OpenGL 2.0 supported the OpenGL Shading Language (GLSL) in 2003
- OpenGL 3.0 (2011) deprecates fixed rendering pipeline and immediate mode
- **Geometry shaders** were added in OpenGL 3.2
- **Tessellation shaders** were added in OpenGL 4.0
- **Compute shaders** were added in OpenGL 4.2
- Programmable shaders allow real-time: Shadows, environment mapping, per-pixel lighting, bump mapping, parallax bump mapping, HDR, etc.

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 desired effect

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 (triangle, quad, etc.)
- Generates new primitives by subdividing each line, triangle or quad primitive
- Typically used for adapting visual quality to the required level of detail → recursive subdivision
  - For instance, for automatic tessellation of Bezier curves and surfaces

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 it can remove the primitive
- Typically used for:
  - Per-face normal computation
  - Easy wireframe rendering
  - Point sprite generation: turn OpenGL points into geometry
  - Shadow volume extrusion
  - Single pass rendering to a cubic shadow map
  - Marching cubes for iso-surfaces
  - Automatic mesh complexity modification

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

Compute Shader

- Not part of the graphics pipeline
- Have no user-defined inputs and no outputs
- Results written to an image or shader storage block
- More info:
  - https://www.opengl.org/wiki/Compute_Shader

GLSL Shader Structure

```
#version version_number
in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;

void main()
{
  // process inputs and apply graphics algorithms
  ...
  // store algorithm result in output variable
  out_variable_name = algorithm_result;
}
```

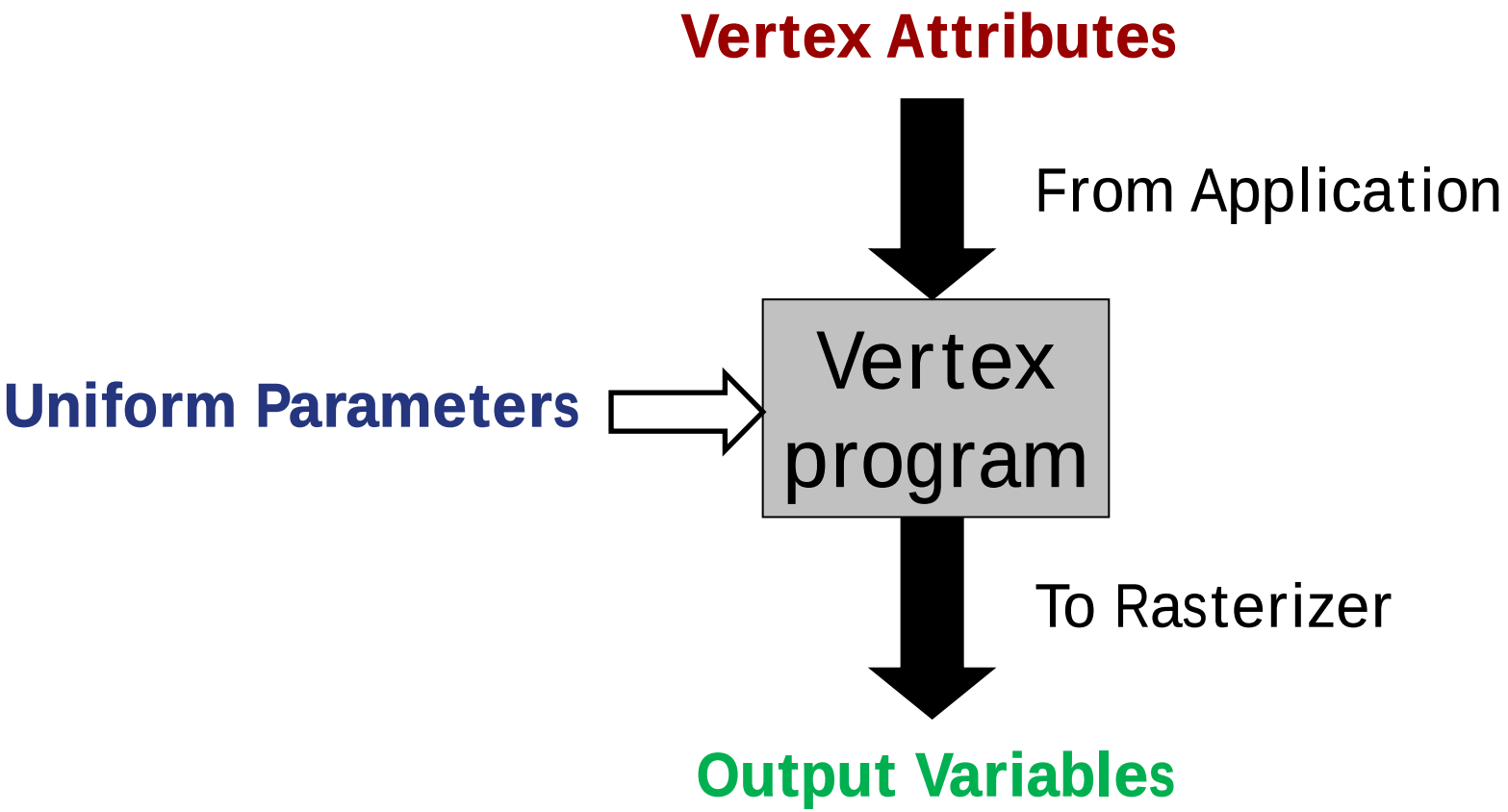
GLSL Data Types

- float
  - vec2, vec3, vec4: floating point vector in 2D, 3D, 4D
  - mat2, mat3, mat4: 2x2, 3x3, 4x4 floating point matrix
- int
  - ivec2, ivec3, ivec4: integer vector
- bool
  - bvec2, bvec3, bvec4: boolean vector
- sampler: represent textures
  - sampler1D, sampler2D, sampler3D: 1D, 2D and 3D texture
  - samplerCube: Cube Map texture
  - sampler1Dshadow, sampler2Dshadow: 1D and 2D depth-component texture

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
- Example:
  - `attribute float myAttrib;`

Uniform Parameters

- Declared by `uniform` storage classifier
- Normally the same for all vertices
- Read-only

Uniform Parameters

- 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 = …;  // handle of shader program
    ```
  - ```
    GLint i = glGetUniformLocation(p, "a");
                                  // returns location of a
    ```
  - ```
    glUniform1f(i, 1.0f); // set value of a to 1
    ```

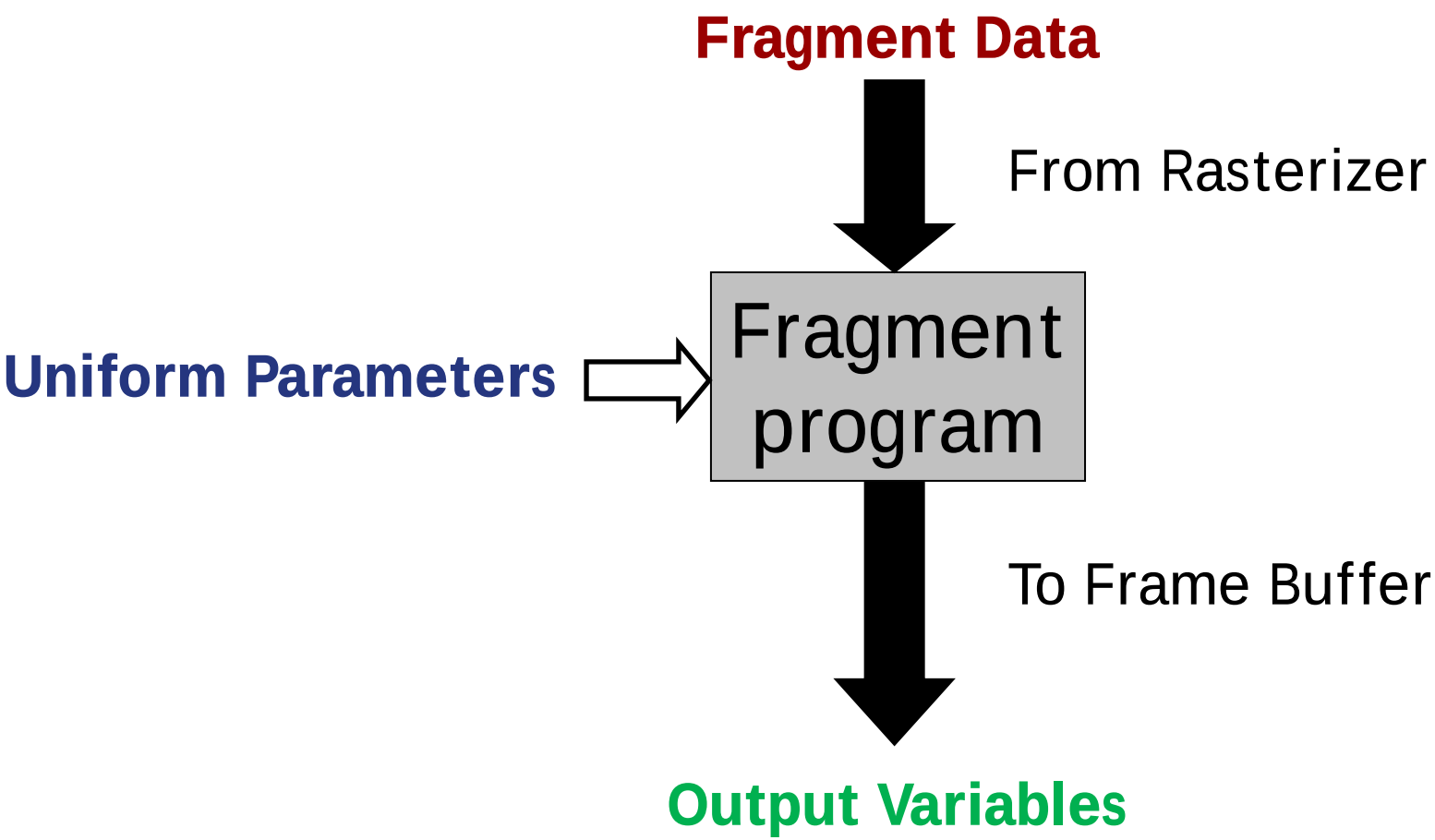
Vertex Programs: Output Variables

- Required output: homogeneous vertex coordinates
  `vec4 gl_Position`
- **`out`** output variables
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Fragment shader gets interpolated data
  - Example: `out 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 variables from vertex shader
  - Allows data to be passed from vertex to fragment shader
  - Specified with `in` parameter

Uniform Parameters

- Same as in vertex programs

Output Variables

- Pre-defined output variables:
  - `vec4 gl_FragColor`
  - `float gl_FragDepth`
- OpenGL writes these to the frame buffer

Built-In GLSL Functions

- dot: dot product
- cross: cross product
- texture2D: used to sample a texture
- normalize: normalize a vector
- clamp: clamping a vector to a minimum and a maximum

Simple GLSL Shader

Vertex Shader: vertex.glsl

```
in vec3 vPosition;

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

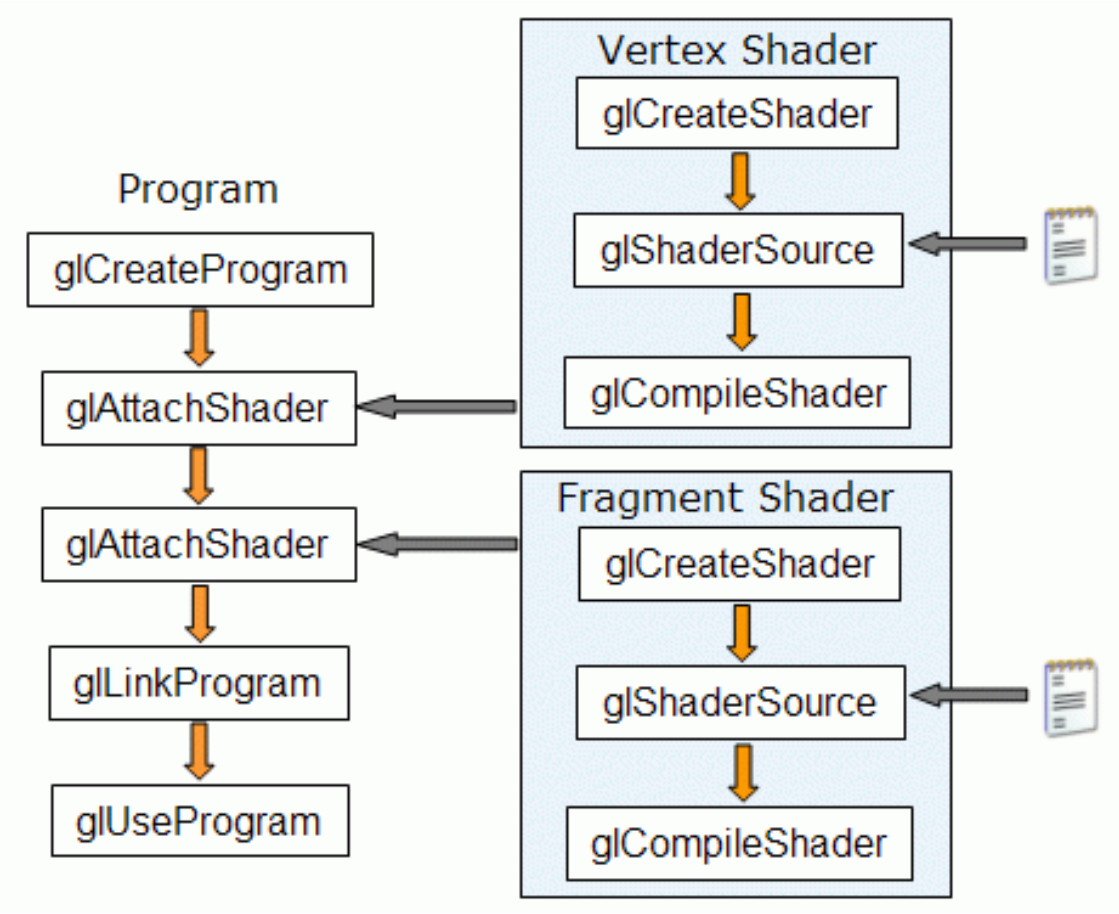
Fragment Shader: fragment.glsl

```
out vec4 color;

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

Source: http://www.codeincodeblock.com/2013/05/introduction-to-modern-opengl-3x-with.html

Loading Shaders in OpenGL



Gabriel Zachmann, Clausthal University

Loading a Shader

```
GLuint loadShader(char *shaderFile, GLenum type)
{
    std::ifstream in(shaderFile);
    std::string src= "";
    std::string line="";
    while(std::getline(in,line))
    src += line + "\n";
    std::cout << src;
    GLuint shader;
    GLint compiled;
    shader = glCreateShader(type);

    const char* source = src.c_str();
    glShaderSource(shader,1,&source,NULL);
    glCompileShader(shader);
    if(!shader)
    {
        std::cerr << "Could not compile the shader";
        return 0;
    }
    return shader;
}
```

Create Shader Program

```
GLuint createShaderProgram()
{
    GLuint vertexShader,fragmentShader;
    GLint linked;

    vertexShader = loadShader("vertex.glsl",GL_VERTEX_SHADER);
    fragmentShader = loadShader("fragment.glsl",GL_FRAGMENT_SHADER);
    if(!vertexShader || !fragmentShader) return 0;

    programId=glCreateProgram();
    if(!programId)
    {
        std::cerr << "could not create the shader program";
        return 0;
    }
    glAttachShader(programId,vertexShader);
    glAttachShader(programId,fragmentShader);

    glBindAttribLocation(programId,0,"vPosition");
    glLinkProgram(programId);
    glGetProgramiv(programId,GL_LINK_STATUS,&linked);
    if(!linked)
    {
        std::cerr << "could not link the shader";
        return 0;
    }
    glUseProgram(programId);

 return programId;
}
```

Load a Triangle

```
static void LoadTriangle()
{
    // make and bind the VAO
    glGenVertexArrays(1, &gVAO);
    glBindVertexArray(gVAO);

    // make and bind the VBO
    glGenBuffers(1, &gVBO);
    glBindBuffer(GL_ARRAY_BUFFER, gVBO);

    // Put the three triangle verticies into the VBO
    GLfloat vertexData[] = {
        //  X     Y     Z
         0.0f, 0.8f, 0.0f,
        -0.8f,-0.8f, 0.0f,
         0.8f,-0.8f, 0.0f,
    };
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

    // unbind the VBO and VAO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

Render

```
// draws a single frame
static void Render(GLFWwindow* MainWindow)
{
    // clear everything
    glClearColor(0, 0, 0, 1); // black
    glClear(GL_COLOR_BUFFER_BIT);
    // bind the VAO (the triangle)
    glBindVertexArray(gVAO);
    // draw the VAO
    glDrawArrays(GL_TRIANGLES, 0, 3);
    // unbind the VAO
    glBindVertexArray(0);
    // swap the display buffers
    glfwSwapBuffers(MainWindow);
}
```

Initialize OpenGL Window

```
// initialize GLFW
if(!glfwInit()) {cerr << "glfwInit failed" << endl; exit();}

// open a window with GLFW
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_RESIZABLE, GL_TRUE);
MainWindow = glfwCreateWindow((int)SCREEN_SIZE.x, (int)SCREEN_SIZE.y,
    "Intro OpenGL with Shader",NULL,NULL);

if(!MainWindow) {cerr << "glfwOpenWindow failed” << endl; exit();}

// GLFW settings
glfwMakeContextCurrent(MainWindow);

// initialize GLEW
if(glewInit() != GLEW_OK) {cerr << “glewInit failed“ << endl; exit(); }

// make sure OpenGL version 3.2 API is available
if(!GLEW_VERSION_4_2) {cerr << “OpenGL 4.2 API is not available.“ << endl; exit();}
```

Load Shaders and Render

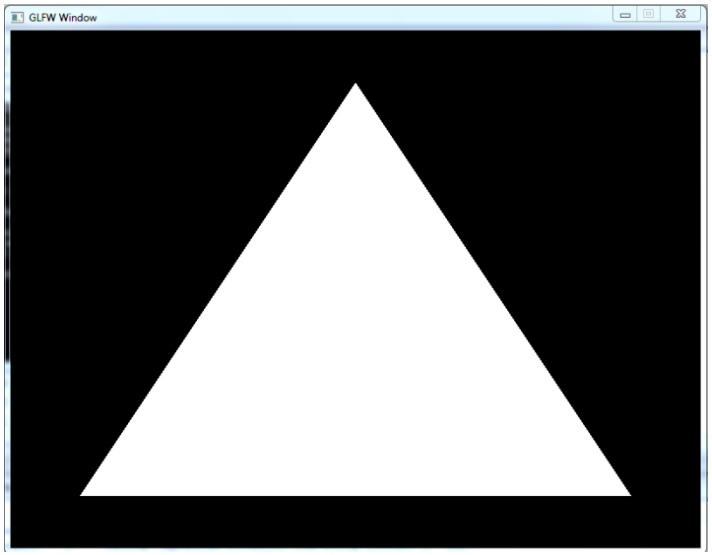
```
// load vertex and fragment shaders into opengl

LoadShaders();
if(!createShaderProgram())
{
   cerr << "Could not create the shaders";
}

// create buffer and fill it with the points of the triangle
LoadTriangle();

// run while the window is open
while(glfwGetWindowAttrib(window,GLFW_FOCUSED))
{
    while(!glfwWindowShouldClose(MainWindow))
    {
        // process pending events
        glfwPollEvents();
        // draw one frame
        Render(MainWindow);
    }

    // clean up and exit
    glfwTerminate();
}
```



Lighting with GLSL

- Tutorial for diffuse lighting with a point light
  - http://www.tomdalling.com/blog/modern-opengl/06-diffuse-point-lighting/