SHADERS

Shaders are programs that rest on the GPU and run for each specific section of the graphics pipeline.
Shaders are not allowed to communicate with each other, really; they can only communicate via their inputs and outputs.

GLSL

OpenGL shaders are written in the OpenGL Shading Language (GLSL), a language very similar to C.

- GLSL is tailored for use with graphics and contains useful features specifically targets at vector and matrix multiplication.

Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms, and its main function.

- The entry point of a shader is its main function, where we process any input variables and output the results in output variables.
- When referring to the vertex shader, each input variable is also known as a vertex attirbute.
  - OpenGL guarantees at least 16 4-component vertex attributes available, but some hardware may allow more.
    - Find the Max # of Vertex Attributes Allowed

```
#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 input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Types

GLSL has data types for specifying what kind of variable we want to work with.

- It has most basic C types: int, float, double, uint, and bool.
- It also has two very useful container types, vectors and matrices.

Vectors

A vector in GLSL is a 2-, 3-, or 4-component container for any of the basic types. They can take the following form (n represents the number of components):

- **vecn**: *the default vector of n floats.*
- **bvecn**: *a vector of n booleans.*
- **ivecn**: *a vector of n integers.*
- **uvecn**: *a vector of n unsigned integers.*
- **dvecn**: *a vector of n doubles.*

The first, second, third, and fourth component of a vector can be accessed via vec.x, vec.y, vec.z, and vec.w, respectively.
GLSL also allows you to use **rgba** for colors or **stpq** for texture coordinates, accessing the same components.

The vector datatype allows for flexible component selection called **swizzling**.

- Swizzling allows you to use some unique syntax to copy vector values from one vector to another.
  - Example of Swizzling

You can also pass vectors as arguments to other vectors, reducing the number of arguments required.

- Example of Passing Vectors as Arguments

Ins and Outs

In GLSL, the inputs and outputs of a shader are defined using `in` and `out`.
Whenever an output variable matches with an input variable of the next shader stage, that input value is passed into the output variable.
However, the vertex shader and fragment shader work a bit differently from normal shaders.

- The vertex shader should receive some form of input, otherwise it doesn't really do much.
  - Its input differs from normal shaders, in that it receives its input straight from the vertex data.
  - To define how the vertex data is organized, we specify the input variables with location metadata, using **layouts**, so we can configure the vertex attributes on the CPU.
    - You can achieve the same result by omitting the layout specifier and using `glGetAttribLocation` to query for the attribute locations in you OpenGL code, but layouts are easier to understand and save OpenGL some work.
- The fragment shader requires a vec4 color output variable, since fragment shaders need to generate a final output color.
  - An undefined output color will be interpretted by the color buffer as either black or white.

When the types and the names of the input and output variables are equal on both sides, OpenGL will link those variables together (during the program linking) and allow sending data between the shaders.

Alter the shaders to let the vertex shader decide the color for the fragment shader.

- Make Vertex Shader Control Fragment Shader Output

Uniforms

Uniforms are global, meaning that a uniform variable is unique per shader program object, and can be accessed from any shader at any stage in the shader program.
Uniforms also keep their values until they're either reset or updated.
NOTE: If you declare a uniform that isn't used anywhere in you GLSL code, it will be silently removed from the compiled version which may cause issues.

Set the color of the vertices using a uniform.

- Uniforms

More Attributes!

Add color data to the vertex data.

- Adding Color to Vertex Data

Our Own Shader Class

Create your own shader class that manages all shader operations used so far.

- You probably want to look at the example code just so the syntax isn't *too* different for later on.
- Our Own Shader Class

EXERCISES

1. Adjust the vertex shader so that the triangle is upside-down.
2. Specify a horizontal offset via a uniform and move the triangle to the right side of the screen in the vertex shader using th e offset value.
3. Output the vertex position to the fragment shader using the out keyword and set the fragment's color equal to the vertex posi tion (see how even the vertex position values are interpolated across the triangle). Once you managed to do this, try to answer the following question: Why is the bottom -left side of our triangle black?

Find the Max # of Vertex Attributes Allowed

Friday, March 18, 2022 2:02 PM

```
// Gets and prints the maximum number of vertex attributes allowed by your hardware
int maxAttribs;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &maxAttribs);
std::cout << "Maximum number of vertex attributes supported: " << maxAttribs<< std::endl;
```

Example of Swizzling

Friday, March 18, 2022 2:22 PM

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
// Adds the x, y, z, and w value of each vector together
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Example of Passing Vectors as Arguments

Friday, March 18, 2022 2:27 PM

```
// Typical vector construction
vec2 vect = vec2(0.5, 0.7);
// Vector construction using another vector as a parameter
vec4 result = vec4(vect, 0.0, 0.0);
// Vector contruction using swizzling of another vector
vec4 otherResult = vec4(result.xyz, 1.0);
```

Make Vertex Shader Control Fragment Shader Output

Friday, March 18, 2022 4:49 PM

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 aPos;

// Specify a color output to the fragment shader
out vec4 vertexColor;

void main()
{
    // You can directly give a vec3 to vec4's constructor
    gl_Position = vec4(aPos, 1.0);
    // Set the output variable to a teal color
    vertexColor = vec4(0.2, 0.8, 0.6, 1.0);
}
```

Fragment Shader

```
#version 330 core

out vec4 FragColor;

// The input variable from the vertex shader (same name and same type)
in vec4 vertexColor;

void main()
{
    FragColor = vertexColor;
}
```

NOTE: Don't forget to also change the vertex shader source and fragment shader source strings when updating the shader files.

Uniforms

Friday, March 18, 2022 6:10 PM

Fragment Shader

```
#version 330 core

out vec4 FragColor;

// we set this variable in the OpenGL code.
uniform vec4 ourColor;

void main()
{
    FragColor = ourColor;
}
```

main.cpp

```
// Stores the location of the ourColor uniform
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");

glUseProgram(shaderProgram);

// Sets the value of the ourColor uniform to the color grean
glUniform4f(vertexColorLocation, 0.0f, 1.0f, 0.0f, 1.0f);
```

Something a little extra you can try, if you want the color to change over time:

```
// Gets the time and changes greenValue over time
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
// Stores the location of the ourColor uniform
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
// Sets the value of the ourColor uniform to change in green intensity over time
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Adding Color to Vertex Data

Friday, March 18, 2022 7:54 PM

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec4 aColor;

out vec4 vertexColor;

void main() {
   gl_Position = vec4(aPos, 1.0);
   vertexColor = aColor;
}
```

Fragment Shader

```
#version 330 core

out vec4 FragColor;
in vec4 vertexColor;

void main() {
   FragColor = vertexColor;
}
```

main.cpp

```
// Vertices of a rectangle
float vertices[] = {
   // Coords (xyz)          // Colors (rgba)
    0.5f,  0.5f, 0.0f,      1.0f, 0.0f, 0.0f, 1.0f,      // top right
    0.5f, -0.5f, 0.0f,      0.0f, 1.0f, 0.0f, 1.0f,      // bottom right
   -0.5f, -0.5f, 0.0f,      0.0f, 0.0f, 1.0f, 1.0f,      // bottom left
   -0.5f,  0.5f, 0.0f,      1.0f, 1.0f, 1.0f, 1.0f,      // top left
};
```

```
// Tells OpenGL how to interpret the vertex data
// Coords
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// Colors
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 7 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Our Own Shader Class

Friday, March 18, 2022 8:03 PM

Shader.h

```
#pragma once

#include <glad/glad.h> // include glad to get all the required OpenGL headers

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader
{
public:
    // the program ID
    unsigned int ID;

    // constructor reads and builds the shader
    Shader(const char* vertexPath, const char* fragmentPath);
    // use/activate the shader
    void use();
    // utility uniform functions
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
};
```

Shader.cpp

```
#pragma once

#include "Shader.h"

class Shader
{
public:
    // constructor generates the shader on the fly
    Shader(const char* vertexPath, const char* fragmentPath)
    {
        // 1. retrieve the vertex/fragment source code from filePath
        std::string vertexCode;
        std::string fragmentCode;
        std::ifstream vShaderFile;
        std::ifstream fShaderFile;
        // ensure ifstream objects can throw exceptions:
        vShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
        fShaderFile.exceptions (std::ifstream::failbit | std::ifstream::badbit);
        try
        {
            // open files
            vShaderFile.open(vertexPath);
            fShaderFile.open(fragmentPath);
            std::stringstream vShaderStream, fShaderStream;
            // read file's buffer contents into streams
            vShaderStream << vShaderFile.rdbuf();
            fShaderStream << fShaderFile.rdbuf();
            // close file handlers
            vShaderFile.close();
            fShaderFile.close();
            // convert stream into string
            vertexCode   = vShaderStream.str();
            fragmentCode = fShaderStream.str();
        }
        catch (std::ifstream::failure& e) {
            std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ: " << e.what() << std::endl;
```

```
        }
        const char* vShaderCode = vertexCode.c_str();
        const char * fShaderCode = fragmentCode.c_str();

        // 2. compile shaders
        unsigned int vertex, fragment;
        // vertex shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
        glCompileShader(vertex);
        checkCompileErrors(vertex, "VERTEX");

        // fragment Shader
        fragment = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(fragment, 1, &fShaderCode, NULL);
        glCompileShader(fragment);
        checkCompileErrors(fragment, "FRAGMENT");

        // shader Program
        ID = glCreateProgram();
        glAttachShader(ID, vertex);
        glAttachShader(ID, fragment);
        glLinkProgram(ID);
        checkCompileErrors(ID, "PROGRAM");

        // delete the shaders as they're linked into our program now and no longer necessary
        glDeleteShader(vertex);
        glDeleteShader(fragment);
    }

    // activate the shader
    void use()
    {
        glUseProgram(ID);
    }

    // utility uniform functions
    // ------------------------------------------------------------------------
    void setBool(const std::string &name, bool value) const
    {
        glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
    }

    void setInt(const std::string &name, int value) const
    {
        glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
    }

    void setFloat(const std::string &name, float value) const
    {
        glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
    }
    // ------------------------------------------------------------------------

private:
    // utility function for checking shader compilation/linking errors.
    void checkCompileErrors(unsigned int shader, std::string type)
    {
        int success;
        char infoLog[1024];
        if (type != "PROGRAM")
        {
            glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
            if (!success)
            {
                glGetShaderInfoLog(shader, 1024, NULL, infoLog);
```

```
                std::cout << "ERROR::SHADER_COMPILATION_ERROR of type: " << type << "\n" << infoLog << std::endl;
            }
        }
        else
        {
            glGetProgramiv(shader, GL_LINK_STATUS, &success);
            if (!success)
            {
                glGetProgramInfoLog(shader, 1024, NULL, infoLog);
                std::cout << "ERROR::PROGRAM_LINKING_ERROR of type: " << type << "\n" << infoLog << std::endl;
            }
        }
    }
};
```