Lesson 3
Shader

Minjing Yu
minjingyu@tju.edu.cn

Shader

- Shaders are written in the C-like language GLSL
  - Tailored for use with graphics
  - Contains useful features targeted at vector and matrix manipulation
- Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms and its main function
- Each shader's entry point is at its main function where we process any input variables and output the results in its output variables

Shader

- A shader typically has the following 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 input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

Shader

- A shader typically has the following 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 input(s) and do some weird graphics stuf
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

The vertex shader input variable is also known as a vertex attribute,
Need assign location information:
layout (localtion=x)

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
    gl_Position = vec4(position, 1.0f);
    ourColor = color;
    TexCoord = texCoord;
}
```

OpenGL Shading Language (GLSL)

Data Types

- GLSL has most of the default basic types from c language:
  int, **float**, **double**, **uint** and **bool**

- GLSL also has two container types:
  vectors and **matrices**

Data Types

- Vectors
  - A vector in GLSL is a 2,3 or 4 component container for any of the basic types

TYPE	MEANING
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 double components

Data Types

- Vectors
  - Components of a vector can be accessed via vec.x where x is the first component of the vector
  - You can use .x, .y, .z and .w to access their first, second, third and fourth component respectively
  - You can use **rgba** for colors or **stpq** for texture coordinates, accessing the same components

Data Types

- Vectors
  - The vector datatype allows for some interesting and flexible component selection called **swizzling**.

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

  - We can also pass vectors as arguments to different vector constructor calls, reducing the number of arguments required

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

Ins and outs

- GLSL defined the in and out keywords for shader’s input and output
- When an output variable matches with an input variable of the next shader stage they're passed along
- The vertex and fragment shader is different

Ins and outs

- Vertex shader
  - Differs in its input
  - Receives its input straight from the vertex data
  - Specifies the input variables with location metadata
    - So we can configure the vertex attributes on the CPU
    - Via glVertexAttribPointer
  - Requires an extra layout specification for its inputs
    - So we can link it with the vertex data
    - Example: layout (location = 0) in vec3 position

Ins and outs

- Fragment shader
  - Requires a vec4 color output variable
    - To generate a final output color
  - If don’t specify an output color in your fragment shader, OpenGL will render your object black (or white)

Ins and outs

- We should **declare an output in the sending shader and a similar input in the receiving shader** to send data from one shader to the other

- When the **types and the names are equal** on both sides OpenGL will link those variables together and then it is possible to send data between shaders

Ins and outs

- Example

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0
out vec4 vertexColor; // specify a color output to the fragment shader
void main(){
    gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
}
```

Fragment shader

```
#version 330 core
out vec4 FragColor;
in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)
void main(){
    FragColor = vertexColor;
}
```

Uniforms

- A way to pass data from our application on the CPU to the shaders on the GPU

- Slightly different compared to vertex attributes
  - (Global) A uniform variable is unique per shader program object, and can be accessed from any shader at any stage in the shader program

  - Uniforms will keep their values until they're either reset or updated

Uniforms

- Add the uniform keyword to a shader with a type and a name to declare a uniform in GLSL

- An example of setting the color of the triangle via a uniform:

```
#version 330 core
out vec4 FragColor;
uniform vec4 ourColor; // we set this variable in the OpenGL code
void main(){
    FragColor = ourColor;
}
```

Uniforms

- Add data to the uniform:
  - Use OpenGL function in main function
  - Find the index/location of the uniform attribute in shader via glGetUniformLocation
  - Set uniform value via glUniform{f/i/3f/fv}

Note

1. Finding the uniform location doesn't require using the shader program first
2. Updating a uniform does require you to first use the program (by calling glUseProgram), because it sets the uniform on the currently active shader program

Uniforms

- Add data to the uniform:
  - An example of gradually changing colors over time

```
GLfloat timeValue = glfwGetTime();
GLfloat greenValue = (sin(timeValue) / 2.0f) + 0.5f;
GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Uniforms

- Because OpenGL is in its core a C library, it doesn't support for type overloading
- OpenGL defines new functions for each type required
  - glUniform is an example of this:

Postfixes	Meaning
f	the function expects a float as its value
i	the function expects an int as its value
ui	the function expects an unsigned int as its value
3f	the function expects 3 floats as its value
fv	the function expects a float vector/array as its value

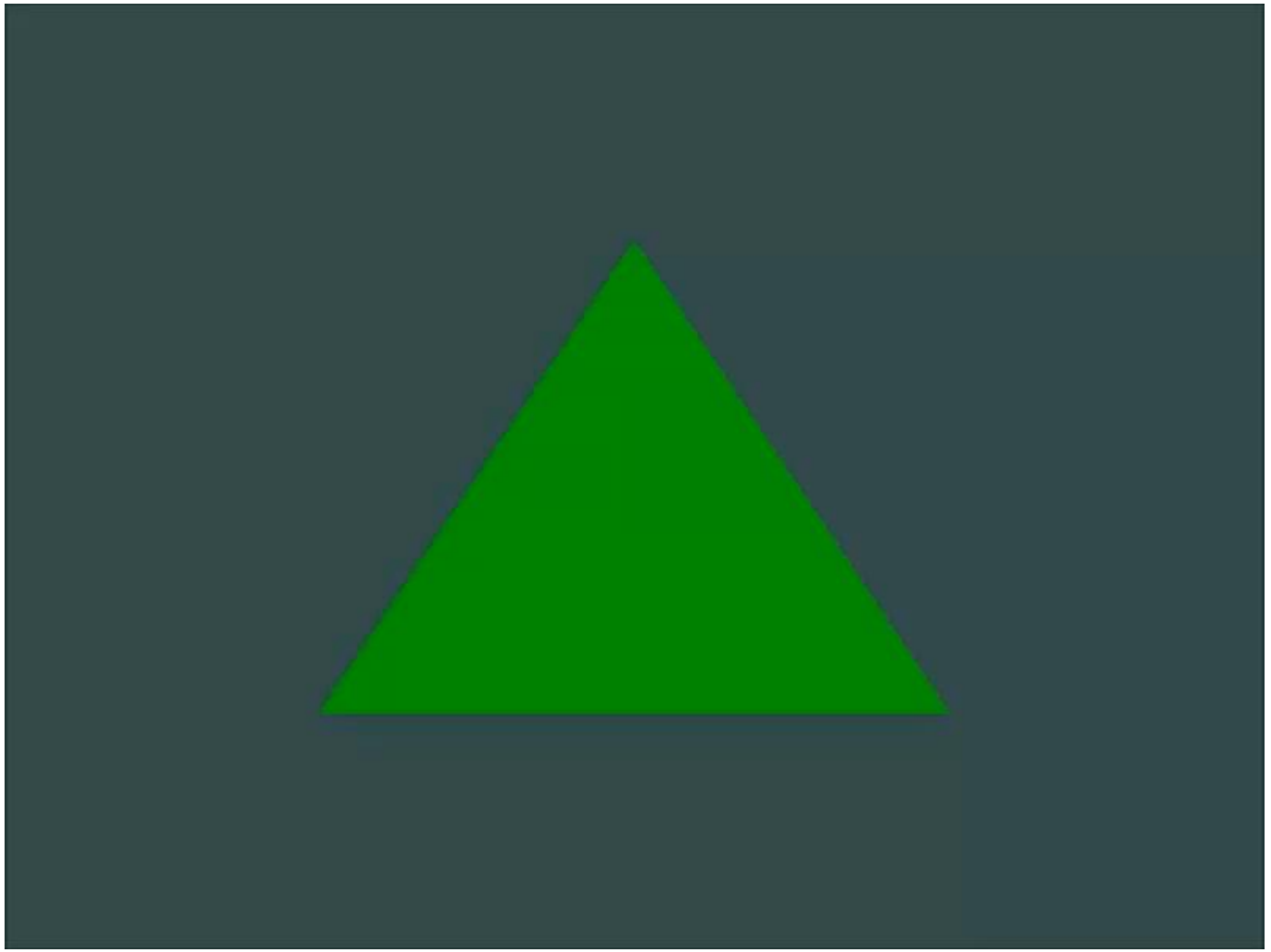
Uniforms

- Update this uniform in every render loop iteration

```
while(!glfwWindowShouldClose(window)){
    processInput(window);
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glUseProgram(shaderProgram);
    // update the uniform color
    float timeValue = glfwGetTime();
    float greenValue = sin(timeValue) / 2.0f + 0.5f;
    int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Uniforms

(SEE attached Tutorial3.cpp)



More attributes

- An example to add color data to the vertices array

```
float vertices[] = {
    // positions         // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // bottom left
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // top
};
```

Vertices array

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
void main(){
    FragColor = vec4(ourColor, 1.0);
}
```

Fragment shader

```
#version 330 core
layout (location = 0) in vec3 aPos;   // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1
out vec3 ourColor; // output a color to the fragment shader
void main(){
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

Vertex shader

More attributes

- An example to add color data to the vertices array

```
float vertices[] = {
    // positions         // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // bott
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // bott
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // top
};
```

Vertices array

glVertexAttribPointer()

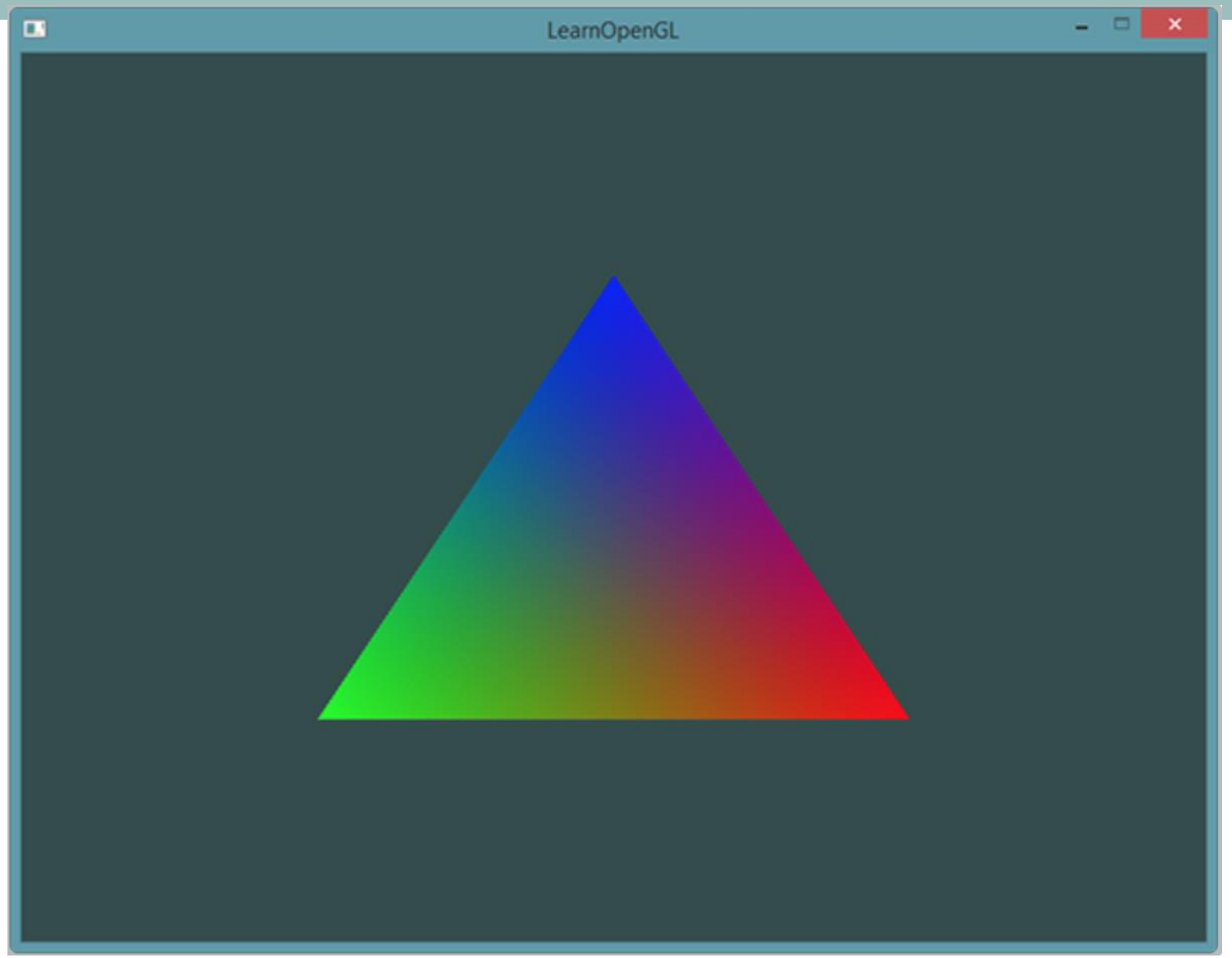
- First argument: which vertex attribute to configure
- Second argument: the size of the vertex attribute
- Third argument: the type of the data
- Fourth argument: if we want the data to be normalized
- Fifth argument: the space between consecutive vertex attributes
- Last parameter: the offset of where the position data begins in the buffer

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float)));
glEnableVertexAttribArray(1);
```

Update the vertex format

More attributes

(SEE attached Tutorial3_2.cpp)



Shader Class

- Writing, compiling and managing shaders can be quite complicated

- We can build a shader class that reads shaders from disk, compiles and links them, checks for errors

- An example is shader.h, we provide a shader class that reads shaders from disk, create shader object, compile and link the shader (SEE attached folder Tutorial3_3)

Thanks!