**绘制四边形**

日期\_2020/4/18

1. 实验目的

1、掌握利用OpenGL绘制四边形;

2、掌握多边形扫描转换算法、区域填充算法、直线属性的填充以及反走样过程;

3、熟练区分OpenGL中不同窗口和图形的颜色填充函数。

二、实验要求

1、完成四边形绘制;

2、设置窗口背景颜色为红色;

3、对完成的四边形进行区域颜色填充为白色;

三、实验内容(代码及注释)

#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <iostream>

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);

void processInput(GLFWwindow\* window);

// settings

const unsigned int SCR\_WIDTH = 800;

const unsigned int SCR\_HEIGHT = 600;

// 顶点着色器：使用glsl语言编写顶点着色器；该顶点着色器对输入数据没有做任何处理就直接传到了着色器的输出，真是程序里输入数据通常不是标准化设备坐标

// 所以，首先必须把它们转换至OpenGL的可视区域内

const char\* vertexShaderSource = "#version 330 core\n" //版本声明为3.3，它与opengl版本相匹配

"layout (location = 0) in vec3 aPos;\n" //in关键字声明输入顶点属性，创建一个vec3的输入变量aPos，现只关心位置属性。layout (location = 0)设置输入变量的位置值

"void main()\n"

"{\n"

" gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n" // glsl的向量数据类型float gl\_Position，main函数最后，gl\_Position设置的值会成为该顶点着色器的输出

"}\n";

// 片段着色器

const char\* fragmentShaderSource = "#version 330 core\n"

"out vec4 FragColor;\n" // out关键字声明输出变量FragColor

"void main()\n"

"{\n"

" FragColor = vec4(0.5f, 0.5f, 1.2f, 1.0f);\n" // RGBA A=1.0f表示完全不透明，RGB搭配为黄色

"}\n";

int main()

{

glfwInit();//初始化glfw

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);//配置glfw，告诉glfw需要使用的opengl版本号为3.3

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);//配置glfw，告诉glfw需要使用的opengl版本号为3.3

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);//告诉glfw使用的是核心模式，即只能使用opengl功能的一个子集，不需要3.3之后版本的兼容功能。

GLFWwindow\* window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "testOpengl", NULL, NULL);//创建窗口对象，该对象存放所有与窗口相关数据，且会被GLFW其他函数调用。

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) //初始化glad

{

std::cout << "faild to initialize GLAD" << std::endl;

return -1;

}

//创建并编译着色器程序

//---------------

//创建顶点着色器

int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

//将创建的着色器附加到着色器对象上，然后编译它

//第2个参数指定传递的源码字符串数量，只有一个。

//第3个参数是顶点着色器真正的源码，在上面声明的。

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

//编译着色器（运行时）

glCompileShader(vertexShader);

//检测着色器编译是否出错

int success;

char infoLog[512];

glGetShaderiv(vertexShader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);

std::cout << "Error::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog << std::endl;

}

//创建片段着色器：计算像素最后的颜色输出;与顶点着色器类似，除了着色器类型为GL\_FRAGMENT\_SHADER不同之外。

int fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);

glCompileShader(fragmentShader);

glGetShaderiv(fragmentShader, GL\_COMPILE\_STATUS, &success);

if (!success)

{

glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);

std::cout << "Error::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog << std::endl;

}

//创建着色器程序对象：它将多个着色器合并后并链接的一个对象。若要使用上面编译好的着色器

//必须把它们链接为一个着色器程序对象，然后在渲染对象时，激活这个着色器程序。以激活的着色器程序的

//着色器将在我们发送渲染调用时被使用

int shaderProgram = glCreateProgram();

glAttachShader(shaderProgram, vertexShader);//vertexShader附加到着色器程序对象上

glAttachShader(shaderProgram, fragmentShader);//fragmentShader附加到着色器程序对象上

glLinkProgram(shaderProgram);//链接

//检测链接是否出错

glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

if (!success)

{

glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);

std::cout << "ERROR::SHARED::PROGRAM::LINKING\_FAILED\n" << std::endl;

}

//在着色器对象链接到对象程序对象后，删除着色器对象，不再需要它们了

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

//设置顶点数据（缓冲）以及配置顶点属性

#if 1//矩形顶点数据，索引缓冲对象：存储不同的顶点，并设定绘制这些顶点的顺序，因此只要存储4个顶点就能绘制矩形

//之后只要指定绘制顺序即可。

// 因此要定义不重复的顶点和绘制出矩形所需的索引

float vertices[] = {

0.5f, 0.5f, 0.0f, //右上角

0.5f, -0.5f, 0.0f, // 右下角

-0.5f, -0.5f, 0.0f, // 左下角

-0.5f, 0.5f, 0.0f // 左上角

};

unsigned int indices[] = { //注意索引从0开始！

0, 1, 3, // 第一个三角形

1, 2, 3 // 第二个三角形

};

#else // 三角形顶点数据

float vertices[] = {

-0.5f, -0.5f, 0.0f, //left

0.5f, -0.5f, 0.0f, //right

0.0f, 0.5f, 0.0f //top

};

#endif

unsigned int VBO, VAO, EBO;

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glGenBuffers(1, &EBO);

//首先绑定顶点数组对象，随后绑定并设置顶点缓冲对象，以及随后配置顶点属性

// 复制顶点数组到缓冲中供opengl使用

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW);

// 第1个参数0指定要配置的顶点属性。它是顶点着色器中使用的layout (location=0)定义的position定点数行的位置值。

// 它可把顶点属性的位置值设置为0

// 第2个参数指定顶点属性的大小，顶点属性是一个vec3，它由3个值组成，因此大小为3

// 第3个参数指定数据的类型，这里是GL\_FLOAT(glsl中vec\*都是由浮点数值组成的)

// 第4个参数是否希望数据被标准化。若设为GL\_TRUE，则所有数据都会被映射到0（对有符号型的signed数据是-1）到1之间。因此把它

// 设为GL\_FALSE

// 第5个参数为步长(stride)，简单点说就是从这个属性第二次出现的地方到整个数组0位置之间有多少字节。它指出顶点属性组之间的间隔。

// 由于下个组位置数据在3个float之后，因此把步长设为3\*sizeof(float)，

// 需要注意的是：1. 由于我们已知该数组是紧密排列的（在两个顶点属性之间没有空隙），我们也可设置为0让opengl决定具体步长为多少（只有

// 当数值紧密排列才可用）。 2. 当有更多顶点属性时，需要更小心地定义每个顶点属性之间的间隔。

// 第6个参数为void\*，因此需要进行强制类型转换。它表示位置数据在缓冲中起始位置的偏移量，由于位置数据在数组的开头，所以这里是0

// 每个顶点属性从一个VBO管理的内存中获得它的数据，而具体是从哪个VBO获取则是通过在调用glVetexAttribPointer时绑定到GL\_ARRAY\_BUFFER的

// VBO决定的。由于在调用glVetexAttribPointer之前绑定的是先前定义的VBO对象，顶点属性0现在会链接到它的顶点数据。

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);

// 以顶点属性位置值作为参数启用顶点属性；顶点属性默认是禁用的。

glEnableVertexAttribArray(0);

//到此为止，所有东西都已设置好：我们使用一个顶点缓冲对象将顶点数据初始化到

// 缓冲中，建立一个顶点和一个片段着色器，并告诉OpenGL如何把顶点数据链接到顶点着色器的顶点属性上。在OpenGL中绘制一个物体。

//glBindBuffer(GL\_ARRAY\_BUFFER, 0); // 对glVertexAttribPointer的调用已经使得VBO注册成为顶点属性的绑定顶点缓冲对象，因此可安全解绑定

// 解绑VAO以防止其他的VAO调用意外的修改该VAO的值，但是这很少发生。修改其他的VAO要求对glBindVertexArray进行调用，

// 因此，当不是一定有必要，一般不解绑VAO（或VBO）。

//glBindVertexArray(0);

glViewport(0, 0, 800, 600);//告诉opengl渲染窗口的尺寸大小，视口（-1~1）对应GLFW的（0,800）以及(0,600);该维度也可设置的比glfw小

// 以线框模式绘制三角形

// 第1个参数表示将其应用到所有的三角形的正面和背面

// 第2个参数表示用线来绘制

// 之后的绘制调用会一直以线框模式绘制三角形，直到用glPolyonMode(GL\_FRONT\_AND\_BACK,GL\_FILL)将其设置回默认模式

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_LINE);

glPolygonMode(GL\_FRONT\_AND\_BACK, GL\_FILL);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);// 告诉glfw，每当窗口调整大小时，调用此函数

while (!glfwWindowShouldClose(window))//每次循环开始前检查一次GLFW是否被要求退出，若是则函数返回true，则渲染循环便结束。

{

processInput(window);//处理输入事件

glClearColor(0.2f, 0.3f, 0.3f, 1.0f); //设置清空屏幕所用的颜色（状态设置函数）

glClear(GL\_COLOR\_BUFFER\_BIT);//清楚颜色缓冲区之后，整个颜色缓冲会被填充glClearColor里所设置的颜色，深蓝绿色（状态使用函数）。

//画第一个三角形

glUseProgram(shaderProgram);//使用着色器程序对象

glBindVertexArray(VAO); //绑定顶点数组对象

//glDrawArrays(GL\_TRIANGLES, 0, 3); //它使用当前激活的着色器，之前定义的顶点属性配置，和VBO的顶点数据（通过VAO间接绑定）来绘制图元。

glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0);

glfwPollEvents();//检查是否触发事件（如键盘输入，鼠标移动等），更新窗口状态，并调用对应回调函数

glfwSwapBuffers(window);//交换颜色缓冲，存储GLFW窗口没个像素颜色值的大缓冲，它在迭代中用来绘制图像，并输出到显示屏上

}

glfwTerminate(); //清理申请的资源，退出

return 0;

}

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height) //调整视口大小

{

glViewport(0, 0, width, height);

std::cout << "viewport width:" << width << std::endl;

std::cout << "viewport height:" << height << std::endl;

}

void processInput(GLFWwindow\* window)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS) //按下escape键，关闭glfw

{

glfwSetWindowShouldClose(window, true);

}

}

四、实验总结

绘制四边形先初始化OpenGL,接下来对四边形顶点数据进行处理，通过VAO、VBO 将其发送至GPU中，并设置属性指针告诉GPU会如何解释这些数据，然后在着色器中通过顶点和片段着色器对数据进行处理，最后进行渲染，渲染之后做好善后工作，一个四边形就绘制成功了。

五、实验结果

