绘制三角形

日期 2020.4.6

**一、实验目的**

1、熟练OpenGL环境配置；

2、掌握OpenGL基本绘制图形函数。

**二、实验要求**

1、OpenGL环境配置；

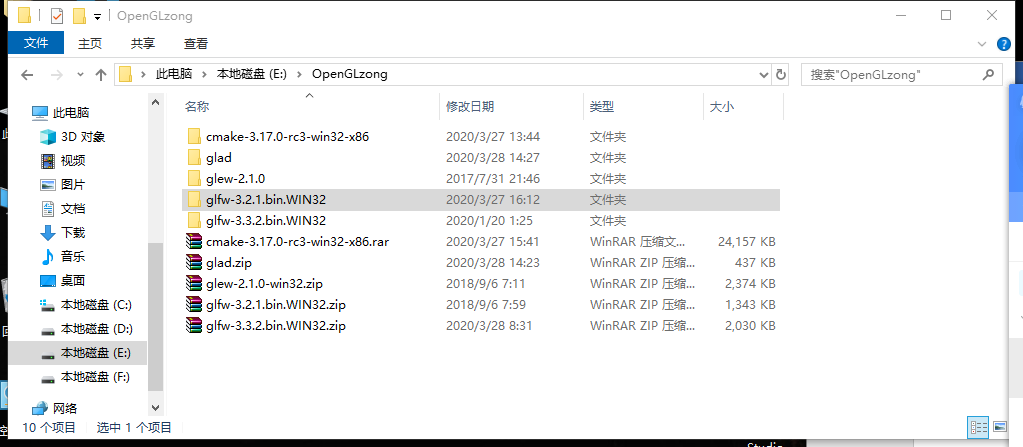
2、利用OpenGL绘制三角形；

3、对绘制好的三角形进行颜色填充。

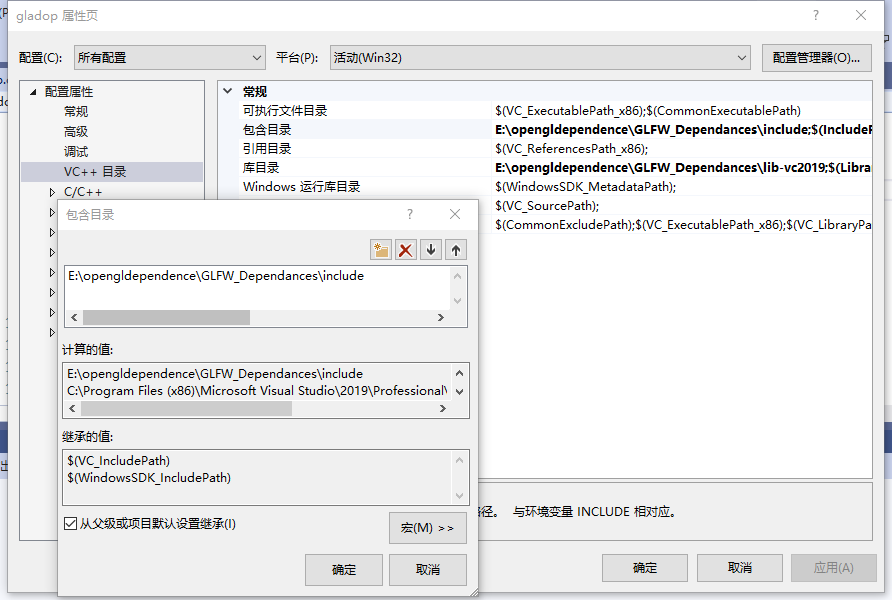
**三、实验内容**

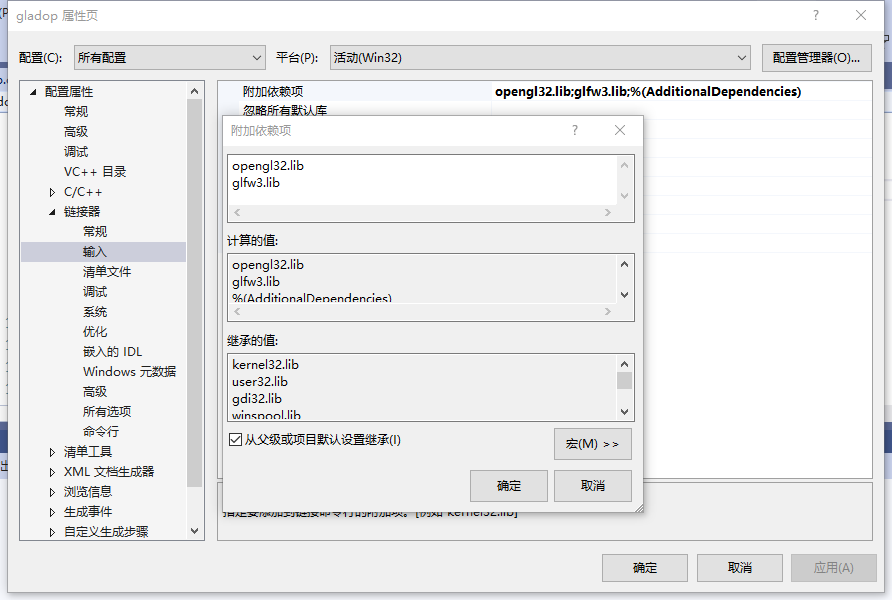
1、OpenGL环境配置步骤：

1. 下载好glfw和glad压缩包

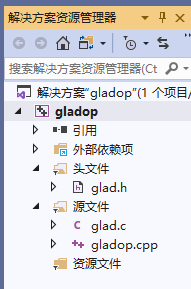


1. 配置vs2019
2. 新建一个空项目
3. 配置项目属性中的包含(头)文件和库文件的路径及附加依赖项的库。





c、创建gladop这个c++项目。



2、三角形绘制代码及注释：

#include <iostream>

#include <glad/glad.h>

#include <GLFW/glfw3.h>

//using namespace std;

//三角形顶点数据

const float triangle[] = {

//位置

-0.5,-0.5,0.0,//左下

0.5f,-0.5f,0.0f,//右下

0.0f,0.5f,0.0f//正上

};

//屏幕宽，高

int screen\_width = 1280;

int screen\_height = 720;

int main() {

//初始化GLFW

glfwInit();

//OpenGL版本为3.3，主次版本号均设为3

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_ANY\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);//使用核心模式，无需向后兼容

glfwWindowHint(GLFW\_RELEASE, false);

//创建窗口（宽、高、窗口名称）

auto window = glfwCreateWindow(screen\_width, screen\_height, "Triangle", nullptr, nullptr);

if (window == nullptr)//如果窗口创建失败

{

std::cout << "Failed to Create OpenGL ConText" << std::endl;

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);//将窗口的上下文设置为当前线程的主上下文

//初始化GLAD,加载OpenGL函数指针地址的函数

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

{

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

}

//指定当前视口尺寸(前两个参数为左下角位置，后两个参数是渲染窗口宽、高)

glViewport(0, 0, screen\_width, screen\_height);

//生成并绑定VAO和VBO

GLuint vertex\_array\_object;//VAO

glGenVertexArrays(1, &vertex\_array\_object);

glBindVertexArray(vertex\_array\_object);

GLuint vertex\_buffer\_object;//VBO

glGenBuffers(1, &vertex\_buffer\_object);

glBindBuffer(GL\_ARRAY\_BUFFER, vertex\_buffer\_object);

//将顶点数据绑定至当前默认的缓存中

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

//设置顶点属性指针

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

glEnableVertexAttribArray(0);

//解绑VAO和VBO

glBindVertexArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

//顶点着色器和片段着色器源码

const char\* vertex\_shader\_source =

"#version 330 core\n"

"layout (location=0) in vec3 aPos;\n"

"void main()\n"

"{\n"

" gl\_Position=vec4(aPos,1.0);\n"

"}\n\0";

const char\* fragment\_shader\_source =

"#version 330 core\n"

"out vec4 FragColor;\n"

"void main()\n"

"{\n"

" FragColor=vec4(0.5f,1.5f,0.2f,1.0f);\n"//三角形的颜色 "}\n\0";

//生成并编译着色器

//顶点着色器

int vertex\_shader = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertex\_shader, 1, &vertex\_shader\_source, NULL);

glCompileShader(vertex\_shader);

int success;

char info\_log[512];

//检查着色器是否编译成功，如果编译失败，打印错误信息

glGetShaderiv(vertex\_shader, GL\_COMPILE\_STATUS, &success);

if (!success)

{ glGetShaderInfoLog(vertex\_shader, 512, nullptr, info\_log);

std::cout << "ERROR::SHADER::VERTEX::COMPILATION\_FAILED\n" << info\_log << std::endl;

}

//片段着色器

int fragment\_shader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragment\_shader, 1, &fragment\_shader\_source, nullptr);

glCompileShader(fragment\_shader);

//检查着色器是否成功编译，如果编译失败，打印错误信息 glGetShaderiv(fragment\_shader, GL\_COMPILE\_STATUS, &success);

if (!success)

{ glGetShaderInfoLog(fragment\_shader, 512, nullptr, info\_log);

std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION\_FAILED\n" << info\_log << std::endl;

}

//链接顶点和片段着色器至一个着色器程序

int shader\_program = glCreateProgram();

glAttachShader(shader\_program, vertex\_shader);

glAttachShader(shader\_program, fragment\_shader); glLinkProgram(shader\_program);

//检查着色器是否成功链接，如果链接失败，打印错误信息 glGetProgramiv(shader\_program, GL\_LINK\_STATUS, &success);

if (!success) {

glGetProgramInfoLog(shader\_program, 512, nullptr, info\_log);

std::cout << "ERROR::SHADER::PROGRAM::LINKING\_FAILED\n" << info\_log << std::endl;

}

//删除着色器

glDeleteShader(vertex\_shader);

glDeleteShader(fragment\_shader);

//线框模式

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

//渲染循环

while (!glfwWindowShouldClose((window))) {

//清空颜色缓冲

glClearColor(1.0f, 0.34f, 0.57f, 1.0f);//背景色

glClear(GL\_COLOR\_BUFFER\_BIT);

//使用着色器程序

glUseProgram(shader\_program);

//绘制三角形

glBindVertexArray(vertex\_array\_object);//绑定VAO

glDrawArrays(GL\_TRIANGLES, 0, 3);//绑定三角形

glBindVertexArray(0);//解除绑定

//交换缓冲并且检查是否有触发事件(比如键盘输入、鼠标移动等)

glfwSwapBuffers(window);

glfwPollEvents();

}

//删除VAO和VBO

glDeleteVertexArrays(1, &vertex\_array\_object);

glDeleteBuffers(1, &vertex\_buffer\_object);

//清理所有的资源并正确退出程序

glfwTerminate();

return 0;

}

3、三角形颜色填充代码及注释：

const char\* fragment\_shader\_source =

"#version 330 core\n"

"out vec4 FragColor;\n"

"void main()\n"

"{\n"

" FragColor=vec4(0.5f,0.5f,0.5f,1.0f);\n"//三角形的颜色

"}\n\0";

1. **实验总结**

通过本实验使我明白了OpenGL在vs2019上的配置，以及OpenGL函数的调用。

其中VBO 即 Vertex Buffer Object，是一个在高速视频卡中的内存缓冲，用来保存顶点数据,也可用于包含诸如归一化向量、纹理和索引等数据。通过顶点缓冲对象(Vertex Buffer Objects, VBO)管理这个内存，它会在GPU内存（通常被称为显存）中储存大量顶点。使用这些缓冲对象的好处是我们可以一次性的发送一大批数据到显卡上，而不是每个顶点发送一次。VAO 即 Vertex Array Object ，是一个包含一个或多个VBO的对象，被设计用来存储一个完整被渲染对象所需的信息。图形渲染管线可以被划分为几个阶段，每个阶段将会把前一个阶段的输出作为输入。为了让OpenGL知道我们的坐标和颜色值构成的到底是什么，OpenGL需要你去指定这些数据所表示的渲染类型。

**四、实验结果**

