计算机图形学

实验指导书

（GLSL Version）

电子科技大学计算机科学与工程学院

2024 年 08 月

目录

实验环境配置

安装 GLFW

在使用着色器绘制图像之前，需要先创建一个 OpenGL 上下文(Context)和一个用于显示的窗口。这些操作在每个系统上都不一样，自己处理创建窗口、定义 OpenGL 上下文以及处理用户输入是很麻烦的。因此安装 GLFW 库，它能简化这些操作。

进入 GLFW 官网 https://www.glfw.org/download.html，下载 64 位 windows 二进制文件，如图 0-1 所示。下载后解压缩。

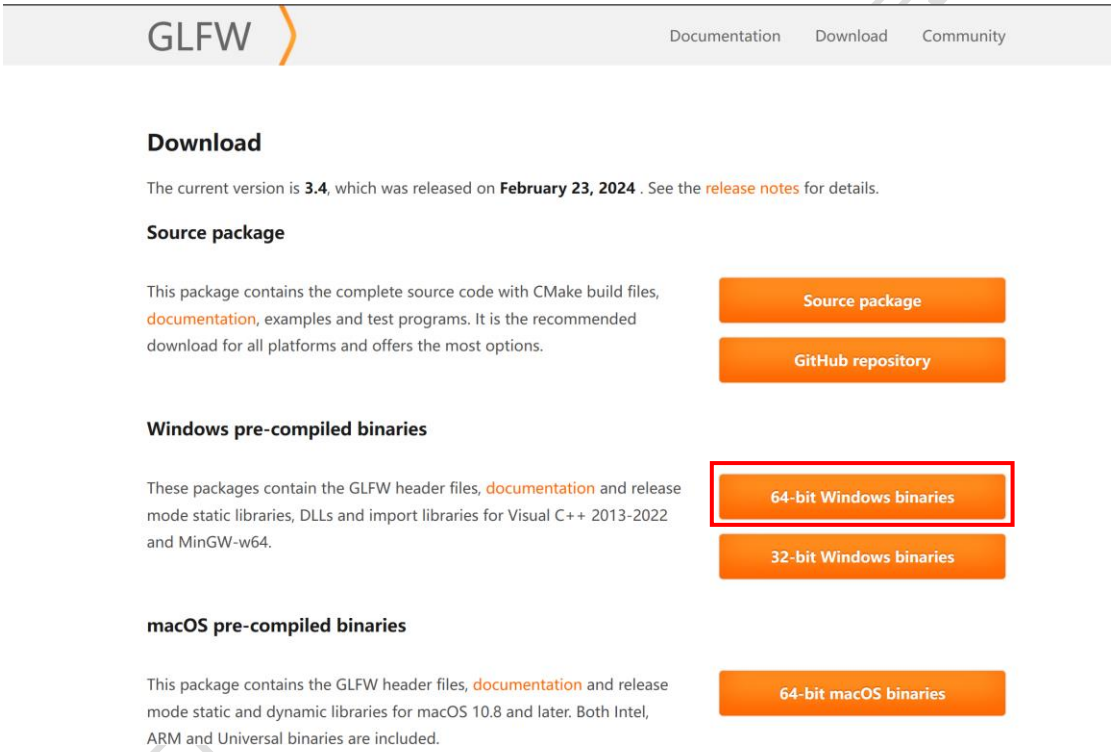


图 0-1 GLFW 下载界面

进入 CMake 官网 https://cmake.org/download，下载并安装 CMake 应用，如图 0-2 所示。

Get the Software

You can either download binaries or source code archives for the latest stable or previous release or access the current development (aka nightly) distribution through Git. This software may not be exported in violation of any U.S. export laws or regulations. For more information regarding Export Control matters please go to https://www.kitware.com/legal.

Training

CMake training covers how to efficiently write CMake scripts for small to larger projects along with best practices. For more information visit the training page. Our next CMake Training Course is April 25-26. Register Now.

Join the mailing list

CMake success stories

Attend a training course

Buy the book

Purchase support

Latest Release (3.24.2)

The release was packaged with CPack which is included as part of the release. The .sh files are self extracting gziped tar files. To install a .sh file, run it with /bin/sh and follow the directions. The OS-machine.tar.gz files are gziped tar files of the install tree. The OS-machine.tar.Z files are compressed tar files of the install tree. The tar file distributions can be untared in any directory. They are prefixed by the version of CMake. For example, the linux-x86_64 tar file is all under the directory cmake-linux-x86_64. This prefix can be removed as long as the share, bin, man and doc directories are moved relative to each other. To build the source distributions, unpack them with zip or tar and follow the instructions in README.rst at the top of the source tree. See also the CMake 3.24 Release Notes.

Source distributions:

Platform	Files
Unix/Linux Source (has \n line feeds)	cmake-3.24.2.tar.gz
Windows Source (has \r\n line feeds)	cmake-3.24.2.zip

Binary distributions:

Platform	Files
Windows x64 Installer:	cmake-3.24.2-windows-x86_64.msi
Windows x64 ZIP	cmake-3.24.2-windows-x86_64.zip
Windows i386 Installer:	cmake-3.24.2-windows-i386.msi
Windows i386 ZIP	cmake-3.24.2-windows-i386.zip
Windows ARM64 Installer:	cmake-3.24.2-windows-arm64.msi
Windows ARM64 ZIP	cmake-3.24.2-windows-arm64.zip
macOS 10.13 or later	cmake-3.24.2-macos-universal.dmg
	cmake-3.24.2-macos-universal.tar.gz
macOS 10.10 or later	cmake-3.24.2-macos10.10-universal.dmg
	cmake-3.24.2-macos10.10-universal.tar.gz
Linux x86_64	cmake-3.24.2-linux-x86_64.sh
	cmake-3.24.2-linux-x86_64.tar.gz
Linux aarch64	cmake-3.24.2-linux-aarch64.sh
	cmake-3.24.2-linux-aarch64.tar.gz

Summary files:

Role	Files
File Table v1	cmake-3.24.2-files-v1.json
Cryptographic Hashes	cmake-3.24.2-SHA-256.txt
PGP sig by 2D2CEF1034921684	cmake-3.24.2-SHA-256.txt.asc

Also see instructions on Download Verification.

图 0-2 CMake 下载界面

在 glfw 解压缩的文件夹中新建 build 文件夹，如图 0-3 所示。

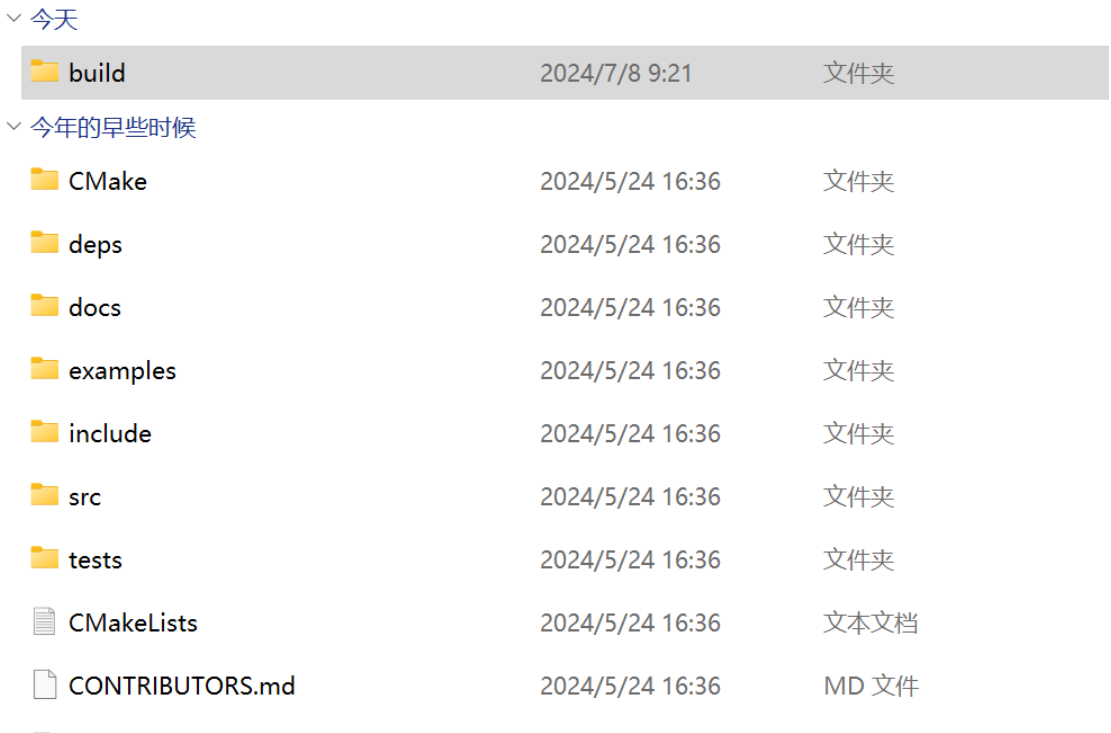


图 0-3 新建 build 文件夹

打开 CMake，见图 0-4 所示，source code 栏选择 glfw-3.4 文件夹（这里下载的是 3.4 版本的 glfw，自行根据实际安装版本查找），下方目标位置选择自己创建的 build 文件夹。然后点 Configure。

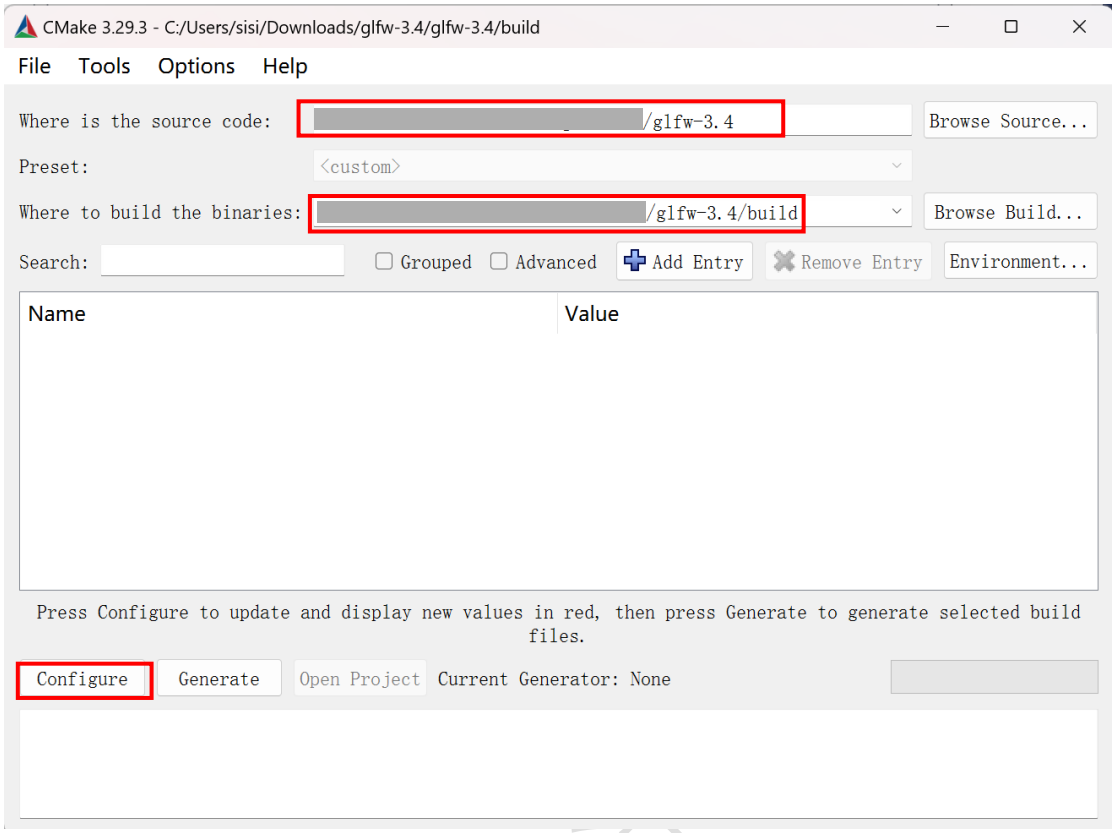


图 0-4 CMake 界面

见图 0-5 所示，在 Configure 中选择自己的 visual studio 版本，点 Finish 完成配置。

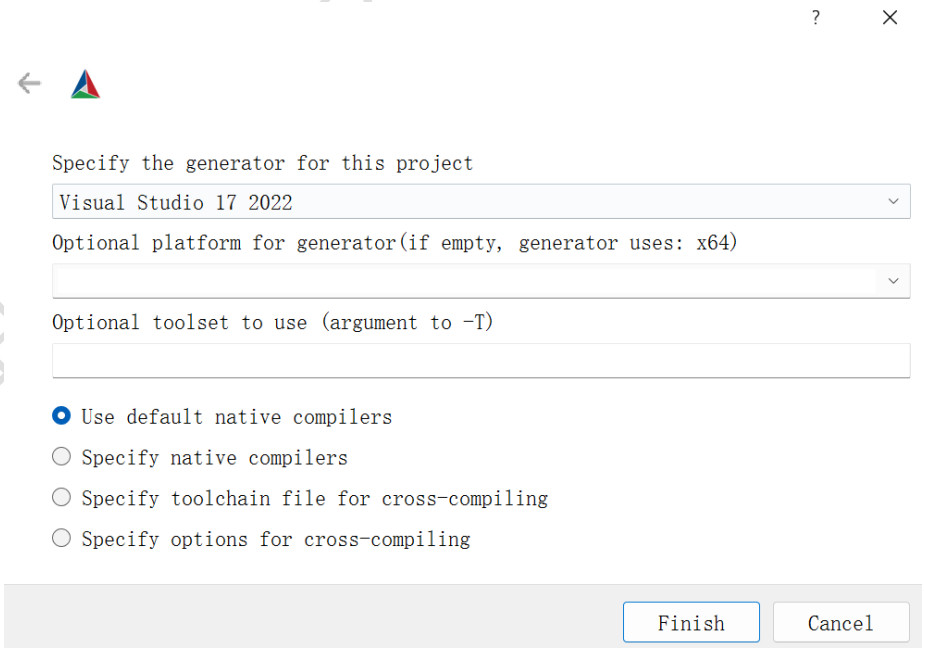


图 0-5 在 CMake 中选择 visual studio 版本

图 0-6 所示，等待下方输出栏出现 Configuring done 后，点 Generate 生成。

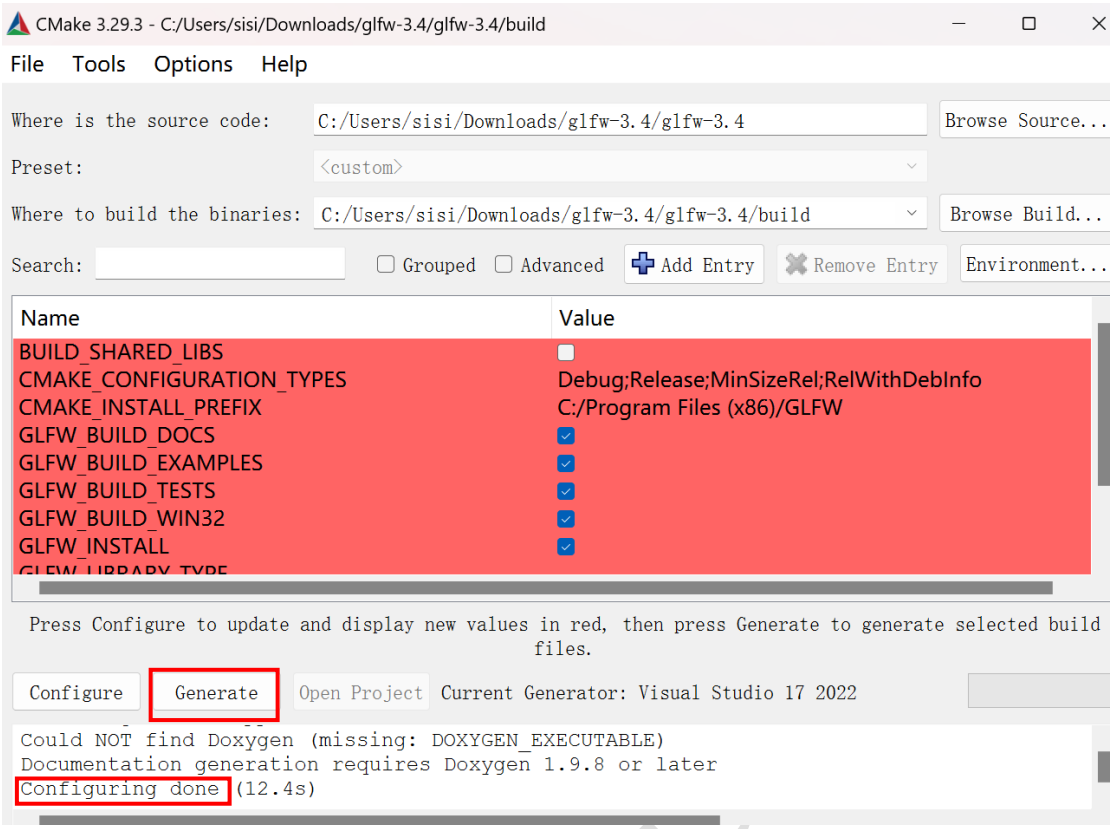


图 0-6 配置好 Cmake 后的界面

没有问题的话会出现 Generating done，如图 0-7 所示，此时可以退出 CMake。

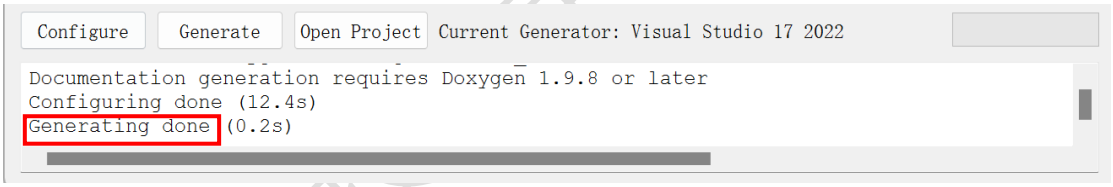


图 0-7 CMake 生成成功界面

打开 glfw-3.4/build 文件夹，使用 visual studio 打开 GLFW.sln，点击生成解决方案。

找合适位置新建一个文件夹，命名为 my-extern-lib，用于储存所有类似 glfw 的外部库，这样能方便地将外部库包含进新建的 visual studio 项目中。在 my-extern-lib 中新建 Include、Libs、src 三个文件夹。

打开 glfw-3.4/include/GLFW 文件夹，其中有 glfw3.h 和 glfw3native.h 两个头文件。如图 0-8 所示。

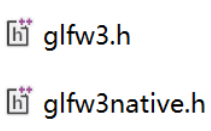


图 0-8 GLFW 头文件

将包含两个头文件的“GLFW”文件夹复制到刚刚新建的 my-extern-lib/Include 文件夹中。

再找到 glfw-3.4/build/src/Debug 文件夹，如图 0-9 所示，将其中的 glfw3.lib 文件复制到 my-extern-lib/Libs 文件夹中。

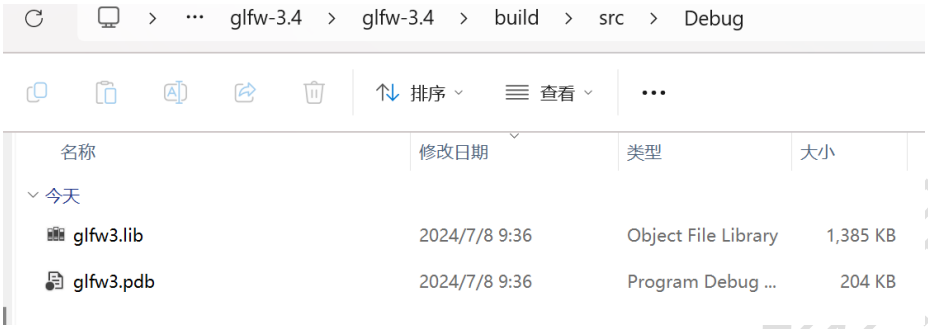


图 0-9 glfw3.lib 文件

安装 GLAD

安装 GLFW 之后，还需要安装 GLAD。因为 OpenGL 只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于 OpenGL 驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。GLAD 就是在运行时确定 GLFW 函数地址的一个开源库。

进入 GLAD 网站 https://glad.dav1d.de/，按图 0-10 所示选择，选好后点击 GENERATE。

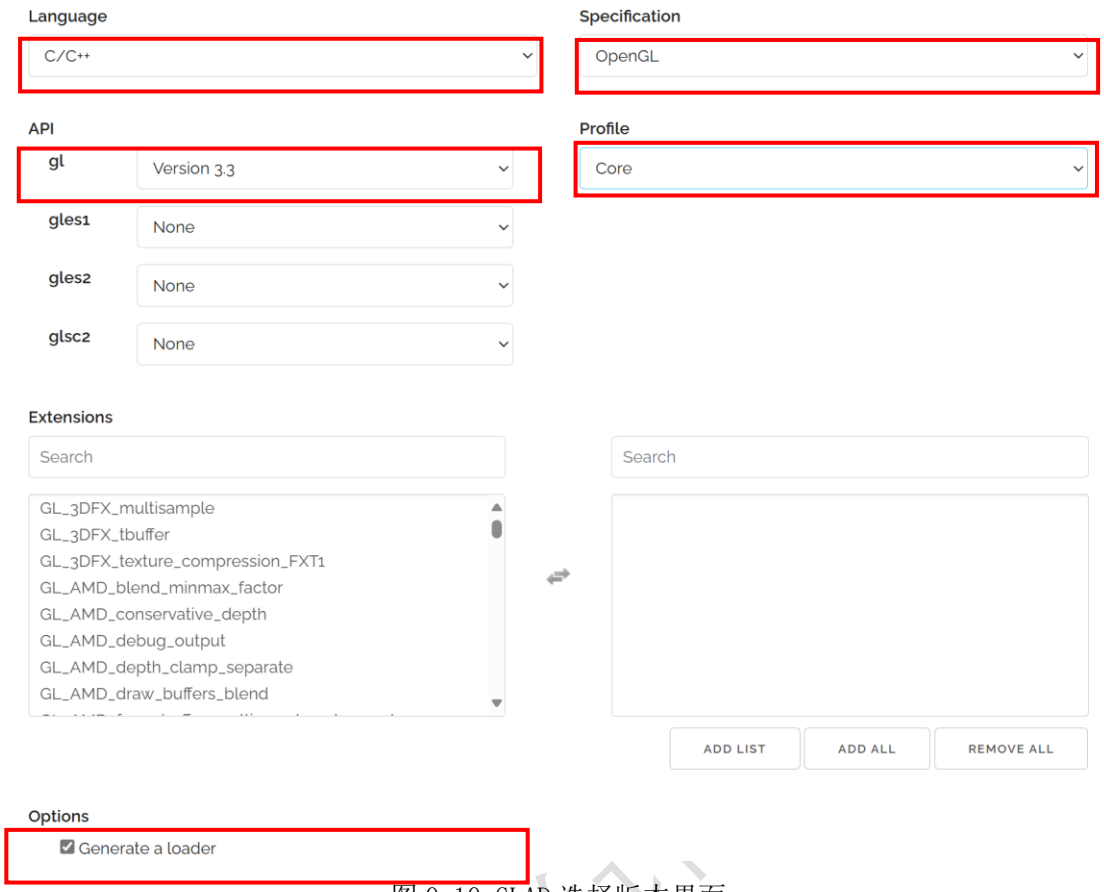


图 0-10 GLAD 选择版本界面

跳转到下载界面，点击 glad.zip 下载。如图 0-11 所示。

图 0-11 GLAD 下载界面

将 glad.zip 解压缩，打开 glad/include 文件夹，将其中的 glad 和 KHR 文件夹复制到 my-extern-lib/Include 中，将 glad/src/glad.c 复制到 my-extern-lib/src 中。

至此，GLFW 和 GLAD 资源文件全部放入 my-extern-lib 文件夹中，环境配置部分完成。在之后每次新建 visual studio 项目时，都需要参照实验一“创建工程”部分，将 my-extern-lib 中的头文件和库文件地址包含到项目中。

实验一 立方体绘制

1.1 实验目的

熟悉使用 GLSL 三维图形绘制，编写顶点着色器和片段着色器实现绘制立方体。

1.2 实验内容

学习在 visual studio 项目中配置 GLFW 环境，创建窗口，编写顶点和片段着色器，读取顶点数据，最终通过绘制三角形面绘制出立方体。

1.3 实验步骤

1.3.1 创建工程

打开 visual studio 创建一个新的空项目，如图 1-1 所示。

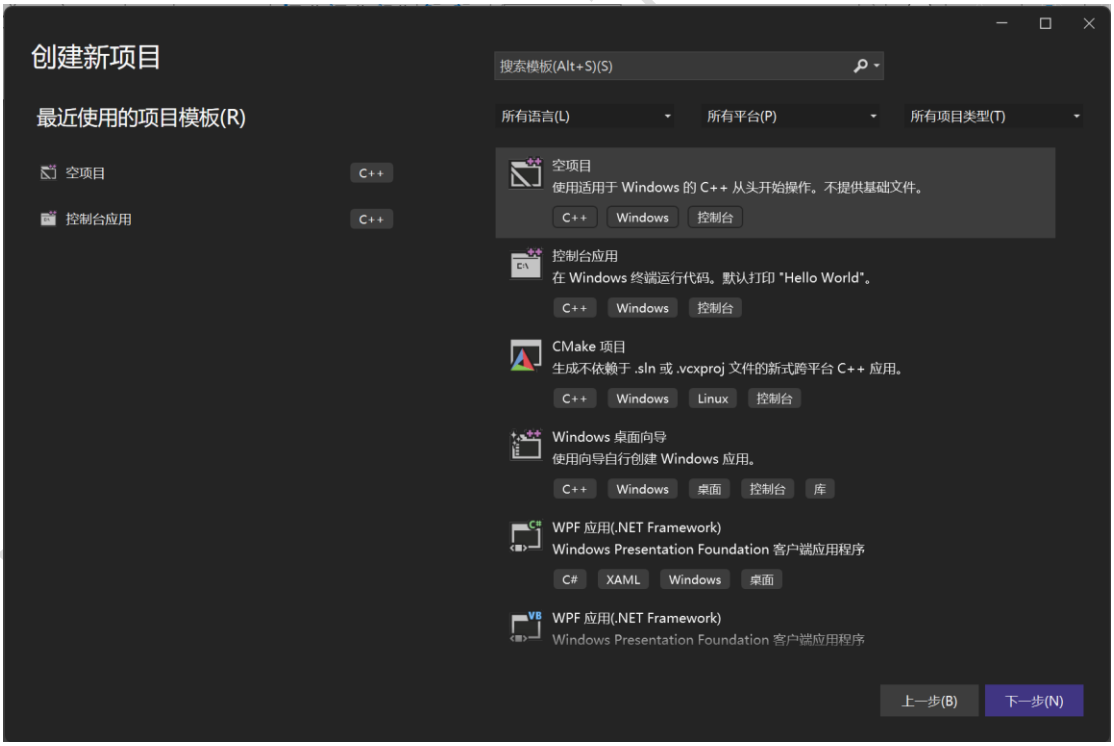


图 1-1 visual studio 新建项目界面

接下来配置环境。首先打开项目-属性，属性页上方配置调为 Debug，x64，如图 1-2 所示。然后点开项目-属性-VC++目录，打开后界面如图 1-3 所示。

图 1-2 选择编译器为 Debug，x64

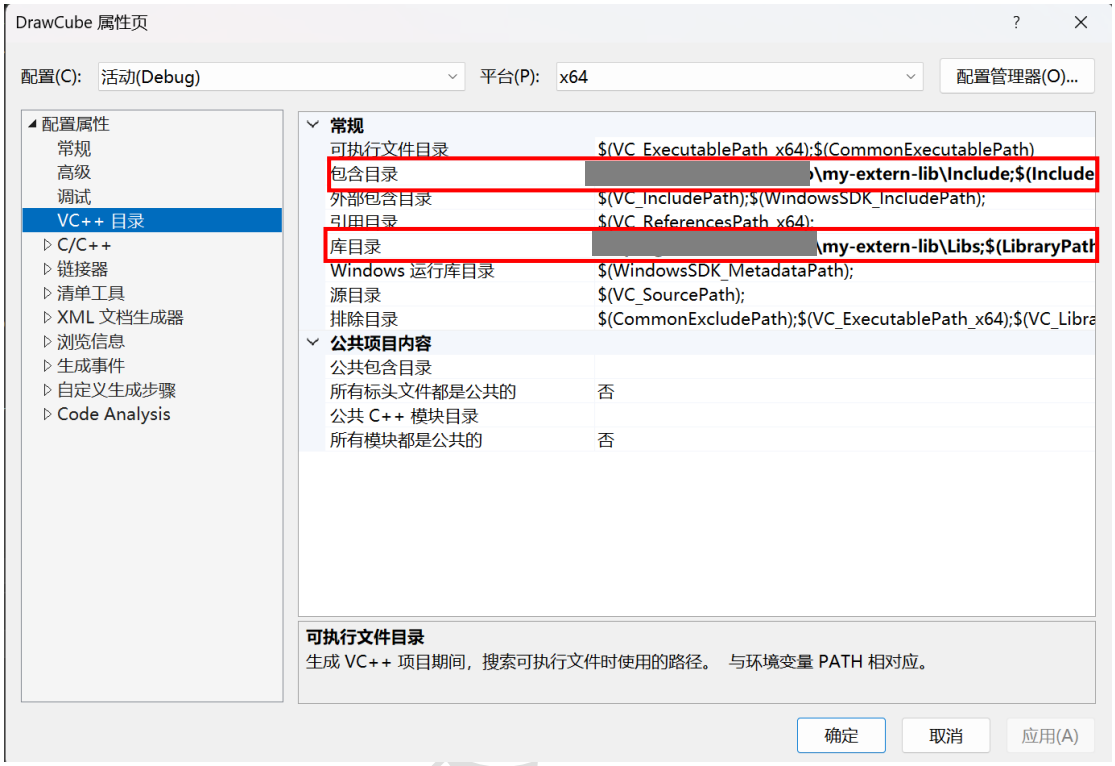


图 1-3 项目属性界面

在包含目录中编辑添加新行，路径设置为 my-extern-lib/Include 文件夹。举例说明写代码时引入头文件的路径怎么写：例如 glad.h 文件的相对路径为“my-extern-lib/Include/glad/glad.h”，则引入 glad.h 头文件时写为

```
1. #include <glad/glad.h>
```

类似地，在库目录中添加 my-extern-lib/Libs 文件夹路径。(.lib 文件需直接位于 my-extern-lib/Libs 文件夹内)

最后点开链接器-输入-附加依赖项，输入“glfw3.lib;opengl32.lib;”如图 1-4 所示。

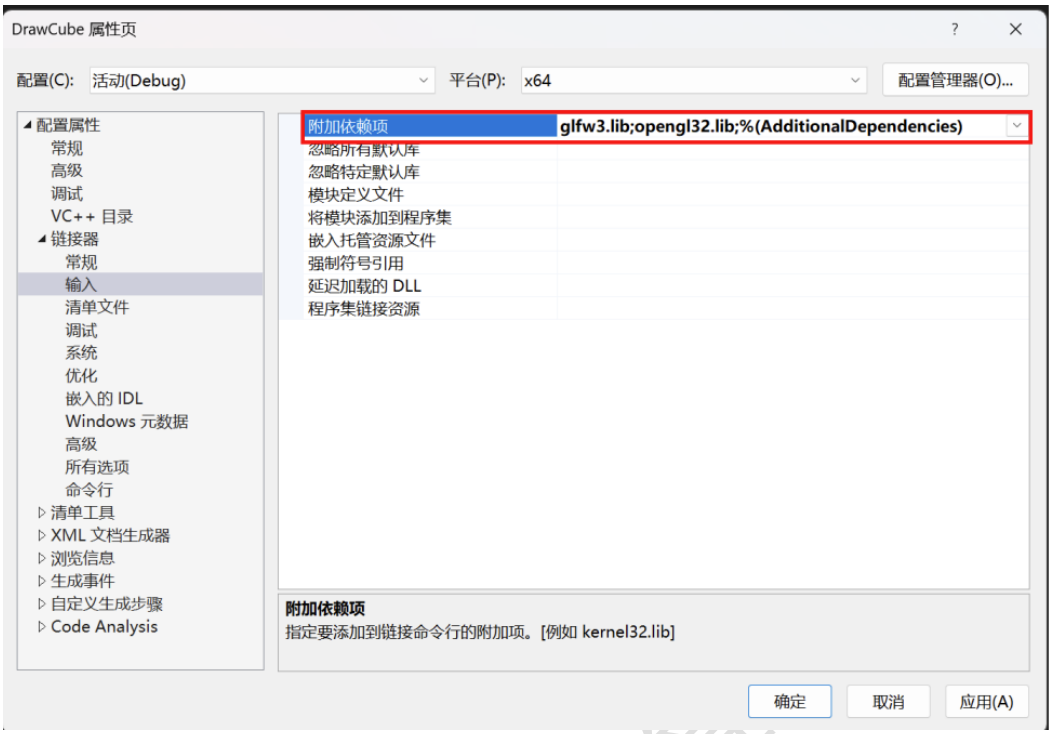


图 1-4 链接器-输入界面

点击确定保存设置。

选择文件-打开-文件，选择 my-extern-lib/src/glad.c，在项目中打开，再选择文件-将 glad.c 移入-项目。如图 1-5 所示。

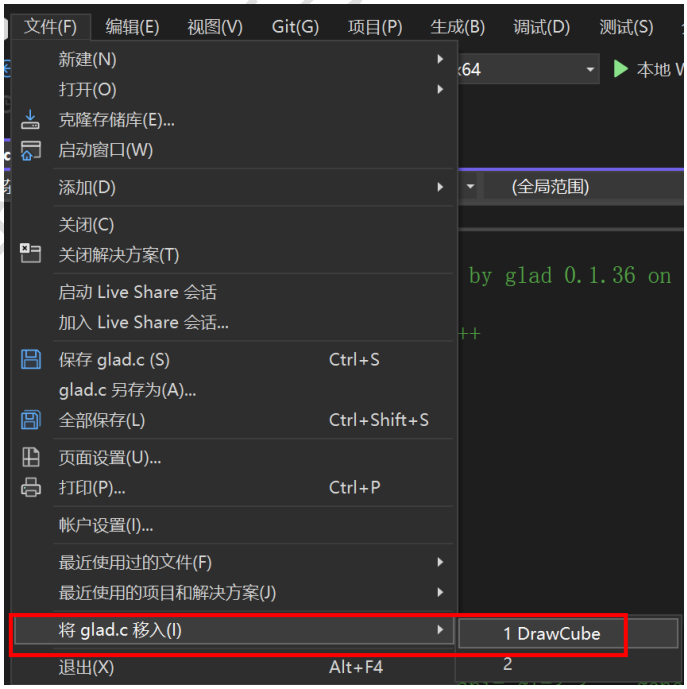


图 1-5 将文件移入项目

至此成功将外部库包含到项目中，接下来可以新建.cpp 文件编写代码了。

1.3.2 窗口创建代码的编写

新建.cpp 文件，首先引入 glad 和 glfw 的头文件。（如果此处报错，返回配置环境部分检查是否有忽略的步骤）

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>
```

接下来在 main 函数中进行 glfw 初始化，使用 glfwWindowHint 函数设置要使用的 OpenGL 版本为 3.3，模式为 Core-profile

```
// GLFW 初始化（设置使用 3.3 版本，核心模式的 OpenGL）
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

使用 glfwCreateWindow 函数创建窗口，该函数需要窗口的宽和高作为它的前两个参数，第三个参数表示这个窗口的名称，最后两个参数暂时忽略。这个函数将会返回一个 GLFWwindow 对象。

然后判断窗口对象是否为空，为空则代表创建窗口失败，输出失败提醒并使用 glfwTerminate 函数释放分配的资源，以便结束程序运行。

最后用 glfwMakeContextCurrent 函数将窗口的上下文设置为当前线程的主上下文。

```
// 创建窗口
GLFWwindow* window = glfwCreateWindow(800, 600, "DrawCube", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window)
```

使用 glad 导入 glfw 函数的代码地址，这样编译器才可以编译 glfw 函数。

```
// 使用 GLAD 导入 GLFW 函数地址
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    glfwTerminate();
    return -1;
```

```
7.  }
```

在 while 语句中进行每一帧的渲染。通过 glfwWindowShouldClose 函数检测窗口是否即将关闭，若即将关闭则退出循环。

这里先只写入 glfwSwapBuffers 函数和 glfwPollEvents 函数。前者的作用是交换缓冲，共有两个缓冲储存屏幕各像素颜色值，其中一个储存当前显示的画面，另一个储存下一帧正在计算中的画面，交换缓冲就可以显示下一帧的画面。后者用于检查事件触发。

```
1. // 渲染循环
2. while (!glfwWindowShouldClose(window))//窗口不关闭则进行渲染循环
3. {
4.     glfwSwapBuffers(window);//交换缓冲
5.     glfwPollEvents();//检查有没有触发事件（比如键盘输入、鼠标移动等）、更新窗口状态
6. }
```

最后在 main 函数结尾释放分配的资源。

```
1. glfwTerminate();
```

上述步骤是绘制图形前的必需流程。接下来使用 Debug，x64 的编译器运行代码，没有错误的话会弹出一个黑色窗口，如图 1-6 所示。

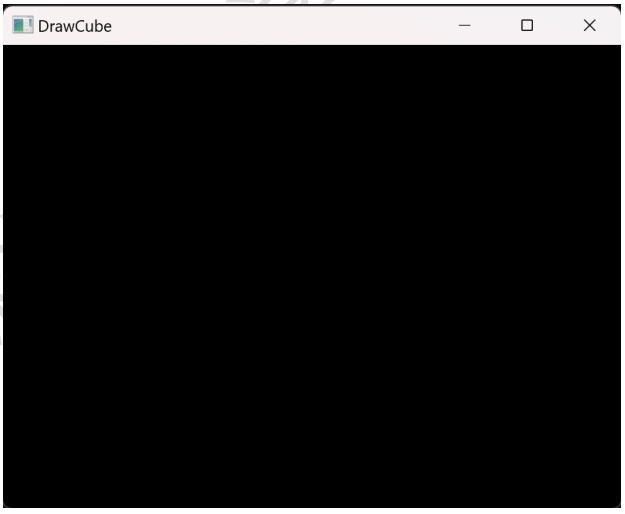


图 1-6 运行代码弹出的黑色窗口

1.3.3 编译着色器

本小节内容包括：从 txt 文件中读取着色器代码，编译着色器代码并进行错误检测输出。分为 C++和 C 语言两个版本，一个供学过 C++面对对象编程的同学学习，将创建一个 Shader 类用于编译、管理着色器；另一个供没学过 C++面对对象编程的同学学习，将直接在 main 函数中编译着色器以代替 Shader 类。

1.3.3.1 使用 C++的方式编译着色器

没学过C++的同学可以跳过该部分，看下一小节的“使用C语言的方式编译着色器”。

新建一个.h文件，命名为Shader.h。将其保存在项目中三个.vcxproj文件的同级目录下，其余代码文件也放在这里，这是为了避免 ifstream 函数读取文件时出错。如图 1-7 所示。

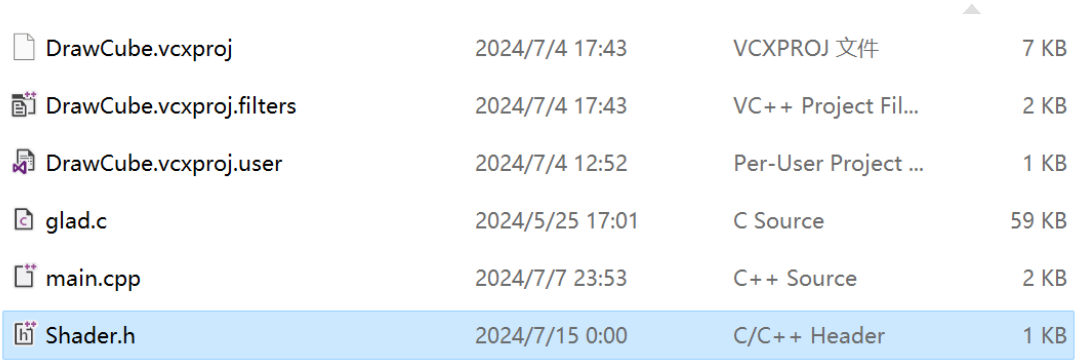


图 1-7 Shader.h 的创建位置示例图

在 Shader.h 文件中编写代码，首先引入下列头文件。

```
1. #include <glad/glad.h>
2. #include <string>
3. #include <fstream>
4. #include <sstream>
5. #include <iostream>
```

然后创建 Shader 类，该类用到的所有属性和方法如下：

```
1. class Shader
2. {
3. public:
4.     // 着色器程序 ID
5.     unsigned int ID;
6.
7.     // 构造函数，读取并编译着色器
8.     Shader(const char* vertexPath, const char* fragmentPath);
9.     // 激活程序
10.    void use();
11.    // uniform 工具函数，用于修改 uniform 变量的值
12.    void setBool(const std::string &name, bool value) const;
13.    void setInt(const std::string &name, int value) const;
14.    void setFloat(const std::string &name, float value) const;
15.};
```

接下来编写构造函数。构造函数有两个传入参数，分别是顶点着色器文件的相对路径，以及片段着色器文件的相对路径。构造函数负责使用 ifstream 读取着色器代码，并将它们编译，再使用 glCreateProgram 函数创建一个程序，将编译好的着色器连接到该程序上，再将该程序的 ID 储存在 Shader 的 public 变量 ID 中，以便后续通过程序 ID 调用。构造函数代码如下：

```
1. Shader(const char* vertexPath, const char* fragmentPath)
2. {
3.     // 1. 读取着色器代码
4.     std::string vertexCode;
5.     std::string fragmentCode;
6.     std::ifstream vShaderFile;
7.     std::ifstream fShaderFile;
8.     // 确保 ifstream 函数可以抛出异常
9.     vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::b
   adbit);
10.    fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::b
   adbit);
11.    try
12.    {
13.        // 打开着色器的 txt 文件
14.        vShaderFile.open(vertexPath);
15.        fShaderFile.open(fragmentPath);
16.        std::stringstream vShaderStream, fShaderStream;
17.        // 读取文件中的字符内容到 stringstream 中
18.        vShaderStream << vShaderFile.rdbuf();
19.        fShaderStream << fShaderFile.rdbuf();
20.        // 关闭 txt 文件
21.        vShaderFile.close();
22.        fShaderFile.close();
23.        // 转换 stringstream 为 string 类型
24.        vertexCode = vShaderStream.str();
25.        fragmentCode = fShaderStream.str();
26.    }
27.    catch (std::ifstream::failure& e)
28.    {
29.        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ: " <
   < e.what() << std::endl;
30.    }
31.    const char* vShaderCode = vertexCode.c_str();
32.    const char* fShaderCode = fragmentCode.c_str();
33.    // 2. 编译着色器代码
34.    unsigned int vertex, fragment;
35.    // 编译顶点着色器代码并错误检查
```

```
36.     vertex = glCreateShader(GL_VERTEX_SHADER);
37.     glShaderSource(vertex, 1, &vShaderCode, NULL);
38.     glCompileShader(vertex);
39.     checkCompileErrors(vertex, "VERTEX");
40.     // 编译片段着色器代码并错误检查
41.     fragment = glCreateShader(GL_FRAGMENT_SHADER);
42.     glShaderSource(fragment, 1, &fShaderCode, NULL);
43.     glCompileShader(fragment);
44.     checkCompileErrors(fragment, "FRAGMENT");
45.     // 创建程序，将着色器连接到程序上，并错误检查
46.     ID = glCreateProgram();
47.     glAttachShader(ID, vertex);
48.     glAttachShader(ID, fragment);
49.     glLinkProgram(ID);
50.     checkCompileErrors(ID, "PROGRAM");
51.     // 释放顶点和片段着色器
52.     glDeleteShader(vertex);
53.     glDeleteShader(fragment);
54. }
```

构造函数中用到了 checkCompileErrors 函数，用于进行错误检查并输出，该函数代码直接给出，代码如下：

```
1. private void checkCompileErrors(unsigned int shader, std::string type)
2. {
3.     int success;
4.     char infoLog[1024];
5.     if (type != "PROGRAM")
6.     {
7.         glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
8.         if (!success)
9.         {
10.            glGetShaderInfoLog(shader, 1024, NULL, infoLog);
11.            std::cout << "ERROR::SHADER_COMPILATION_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;
12.        }
13.    }
14.    else
15.    {
16.        glGetProgramiv(shader, GL_LINK_STATUS, &success);
17.        if (!success)
18.        {
19.            glGetProgramInfoLog(shader, 1024, NULL, infoLog);
```

```
            std::cout << "ERROR::PROGRAM_LINKING_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;
        }
    }
}
```

然后是 use 函数实现。使用 glUseProgram 函数激活程序，激活后着色器才会工作。代码如下：

```
public void use()
{
    glUseProgram(ID);
}
```

接下来实现设置 uniform 类型变量值的 setxxx 函数。Uniform 类型相当于全局变量，能在 main 函数和着色器中使用，例如在 main 函数中修改某 uniform 类型的变量值，在顶点着色器中也能读取到该修改后的值。需要先通过下列函数获取 Uniform 变量的 ID：

```
glGetUniformLocation(GLuint 着色器程序 ID, const GLchar *变量名);
```

然后通过 glUniform 系列函数设置值：

```
glUniformxx(Uniform 变量 ID, value);
```

常用的类型有 Bool、Int、Float 这三种，分别对应的 set 函数实现代码如下：

```
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);
}
```

至此，Shader 类的部分编写完成。

1.3.3.2 使用 C 语言的方式编译着色器

该部分内容用于使用 C 语言代替 Shader 类，学过 C++的同学可以跳过该部分。

首先引入下列头文件。

```
#include <string>
#include <fstream>
#include <sstream>
```

接着声明一个全局变量 ID，用于记录着色器程序的 ID。

```
unsigned int ID;
```

接下来编写 compileShader 函数。该函数有两个传入参数，分别是顶点着色器文件的相对路径，以及片段着色器文件的相对路径。该函数负责使用 ifstream 读取着色器代码，并将它们编译，再使用 glCreateProgram 函数创建一个程序，将编译好的着色器连接到该程序上，再将该程序的 ID 储存，以便后续通过程序 ID 调用。函数代码如下：

```
void compileShader(const char* vertexPath, const char* fragmentPath)
{
    // 1. 读取着色器代码
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    // 确保 ifstream 函数可以抛出异常
    vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        // 打开着色器的 txt 文件
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        // 读取文件中的字符内容到 stringstream 中
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        // 关闭 txt 文件
        vShaderFile.close();
        fShaderFile.close();
        // 转换 stringstream 为 string 类型
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch (std::ifstream::failure& e)
    {
```

```
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ: " << e.what() << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char* fShaderCode = fragmentCode.c_str();
    // 2. 编译着色器代码
    unsigned int vertex, fragment;
    // 编译顶点着色器代码并错误检查
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
    glCompileShader(vertex);
    checkCompileErrors(vertex, "VERTEX");
    // 编译片段着色器代码并错误检查
    fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fShaderCode, NULL);
    glCompileShader(fragment);
    checkCompileErrors(fragment, "FRAGMENT");
    // 创建程序，将着色器连接到程序上，并错误检查
    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    checkCompileErrors(ID, "PROGRAM");
    // 释放顶点和片段着色器
    glDeleteShader(vertex);
    glDeleteShader(fragment);
}
```

compileShader 函数中用到了 checkCompileErrors 函数，用于进行错误检查并输出，该函数代码直接给出，代码如下：

```
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 << "\n -- --------------------------------------------------- -- " << 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 << "\n -- --------------------------------------------------- -- " << std::endl;
        }
    }
}
```

然后是 useShader 函数实现。使用 glUseProgram 函数激活程序，激活后着色器才会工作。代码如下：

```
void useShader()
{
    glUseProgram(ID);
}
```

接下来实现设置 uniform 类型变量值的 setxxx 函数。Uniform 类型相当于全局变量，能在 main 函数和着色器中使用，例如在 main 函数中修改某 uniform 类型的变量值，在顶点着色器中也能读取到该修改后的值。需要先通过下列函数获取 Uniform 变量的 ID：

```
glGetUniformLocation(GLuint 着色器程序 ID, const GLchar *变量名);
```

然后通过 glUniform 系列函数设置值：

```
glUniformxx(Uniform 变量 ID, value);
```

常用的类型有 Bool、Int、Float 这三种，分别对应的 set 函数实现代码如下：

```
void setBool(const std::string& name, bool value)
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}
void setInt(const std::string& name, int value)
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}
void setFloat(const std::string& name, float value)
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}
```

至此，编译着色器要用到的相关函数编写完成，在之后的章节中会调用这些

函数，后续教程只展示用 Shader 类的写法，例如 ourShader.setFloat(xxx)，使用 C 语言的同学直接用 setFloat(xxx)函数代替即可。

1.3.4 着色器代码的编写

在项目中三个.vcxproj 文件的同级目录下创建两个.txt 文件，分别命名为 vectorShader 和 fragmentShader，如图 1-8 所示。分别在这两个文件中编写顶点着色器和片段着色器代码。

图 1-8 新建的两个着色器文件

顶点着色器参考代码如下：

```
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aColor;

out vec3 ourColor;

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

片段着色器参考代码如下：

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;

void main()
{
```

```
8.     FragColor = vec4(ourColor, 1.0f);
9. }
```

上述代码中，第一行表明使用 3.3.0-Core 版本的 OpenGL。

每个着色器都有输入和输出，这样才能进行数据交流和传递。GLSL 定义了 in 和 out 关键字来实现这个目的。每个着色器使用这两个关键字设定输入和输出，只要一个输出变量与下一个着色器阶段的输入匹配，它就会传递下去。

顶点着色器的输入变量有些特殊，因为顶点着色器直接从顶点数据而不是上一个着色器中接收输入，需要通过 layout (location = 0)把输入变量链接到顶点数据。location 相当于顶点数据的 ID，这里储存顶点位置数据的 location 是 0，顶点颜色的 location 是 1。

如果变量类型、名称都相同则视为同一个变量，例如在顶点着色器中输出了 out vec3 ourColor 的变量，那么在片段着色器中就可以通过 in vec3 ourColor 的方式将顶点着色器输出的变量读入，从而达到获取各顶点颜色值的目的。

每个着色器需要完成不同的任务，例如顶点着色器需要设置 gl_Position 的值，作为点的位置；片段着色器需要设置 FragColor 的值，作为点的颜色。不设置这些值，就画不出图像。

1.3.5 读取顶点数据

需要读取顶点数据，着色器才能有输入用于工作。

首先定义 float 数组 vertices 用于储存顶点信息。这里的代码中储存了三个顶点，每一行代表一个顶点。每一行的前三个数据代表顶点坐标 xyz 的值，后三个数据是顶点颜色值 rgb 的值。在 main 函数中定义，写在调用 glad 函数之后，渲染循环之前。代码如下：

```
1. float vertices[] = {
2.      0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,
3.     -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,
4.      0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f
5. };
```

接下来依次创建、绑定 VBO 和 VAO，使用 glBufferData 将顶点数据写入缓冲，再用 glVertexAttribPointer 分别创建指向位置数据和颜色数据的指针。代码如下：

```
1. unsigned int VBO, VAO;
2. glGenVertexArrays(1, &VAO); // 给 VAO 分配内存
3. glGenBuffers(1, &VBO); // 给 VBO 分配内存
4.
```

```
5. glBindVertexArray(VAO); // 绑定 VertexArray 为 VAO，后续修改或调用 VertexArray 的操作，都是作用于当前被绑定的对象
6. glBindBuffer(GL_ARRAY_BUFFER, VBO); // 绑定 GL_ARRAY_BUFFER 为 VBO，后续修改或调用 GL_ARRAY_BUFFER 的操作，都是作用于当前被绑定的对象
7.
8. glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); // 将 vertices 数组中的数据写入 GL_ARRAY_BUFFER
9.
10.// 位置数据
11.glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);//该函数从左到右参数：数据的 Location（在当前 VertexArray 中具有唯一性的 ID，可理解为变量名），数据数量（位置数据是 xyz 三个值，所以是 3），数据类型，是否标准化，距离下一个数据的距离（每行第一个数据是位置信息的起始，读完需要跳到下一行开头读取，即相隔 6 个 float 数据），偏移量（从什么位置开始读取）
12.//该函数创建 Location=0 的数据，用于储存顶点位置信息，并规定从偏移量为 0 处开始读取，间隔 6 个 float，每次读取 3 个数据
13.glEnableVertexAttribArray(0); //规定完 Location=0 的数据的阅读方式，需要使用该函数 enable 该数据
14.// 颜色数据
15.glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float))); //该函数创建 Location=1 的数据，用于储存顶点颜色信息，并规定从偏移量为 3*sizeof（float）处开始读取，间隔 6 个 float，每次读取 3 个数据
16.glEnableVertexAttribArray(1); //规定完 Location=1 的数据的阅读方式，需要使用该函数 enable 该数据
```

至此顶点数据读取完成。

1.3.6 激活着色器

接下来调用之前写好的用于编译着色器的函数，使用着色器绘制一个三角形。

先在 main.cpp 中添加引用 Shader.h 的头文件：（没用 C++方式的同学跳过该步骤）

```
1. #include "Shader.h"
```

使用 C++的同学，在 main 函数中，调用 glad 函数之后，读取顶点数据之前的位置，加入下列代码，使用构造函数创建着色器：

```
1. Shader ourShader("vectorShader.txt", "fragmentShader.txt");
```

使用 C 语言的同学在相同位置调用 compileShader 函数代替：

```
1. compileShader("vectorShader.txt", "fragmentShader.txt");
```

再在读取顶点数据之后，渲染循环之前的位置加入下列代码激活着色器：（使

用 C 语言的同学在相同位置调用 useShader 函数代替）

```
ourShader.use();
```

按照下列代码修改渲染循环中的代码：

```
// 渲染循环
while (!glfwWindowShouldClose(window))//窗口不关闭则进行渲染循环
{
    //设置窗口背景色为墨绿色
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    //清空颜色缓冲
    glClear(GL_COLOR_BUFFER_BIT);
    //使用前三个顶点绘制三角形
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glfwSwapBuffers(window);//交换缓冲
    glfwPollEvents();//检查有没有触发事件（比如键盘输入、鼠标移动等）、更新窗口状态
}
```

运行代码，没有错误的话会绘制出图 1-9 所示的三角形。

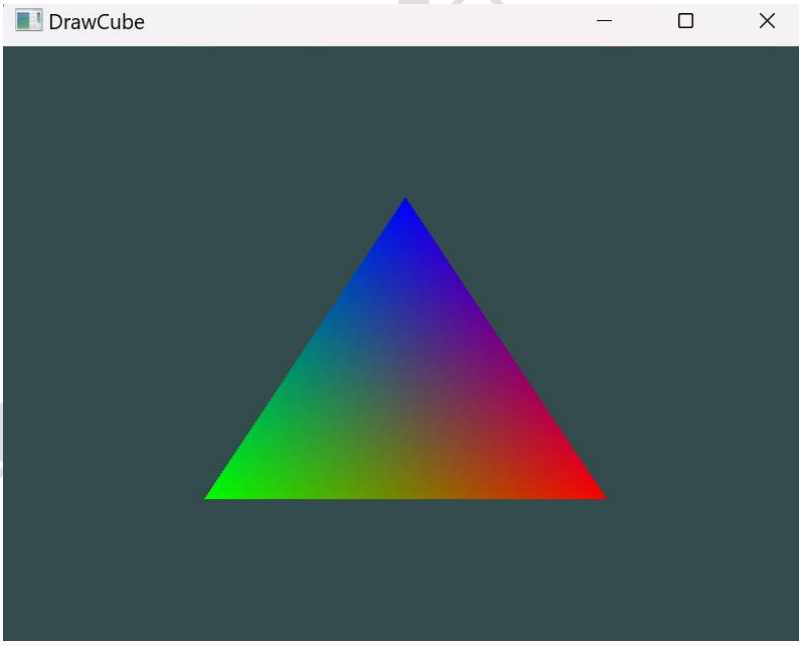


图 1-9 绘制的三角形

1.3.7 绘制立方体

接下来在绘制三角形的代码基础上进行修改，使其能绘制出一个立方体。

绘制立方体涉及到矩阵操作，需要对每一个顶点的坐标向量都左乘一个透视矩阵进行透视变换，变换后就会产生近大远小的透视效果。

矩阵操作要用到 GLM 库里的函数，GLM 是 OpenGL Mathematics 的缩写，它是一个只有头文件的库，不用通过 CMake 链接和编译。在下述网站上下载头文件：https://glm.g-truc.net/0.9.8/index.html。下载好后把头文件的根目录复制到 my-extern-lib/Include 文件夹就可以了。

使用 glm 库中的函数时，需要引入下面三个头文件：

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

在顶点着色器中添加一个 uniform mat4 类型的变量，用于储存透视矩阵：

```
uniform mat4 projection;
```

在 Shader 类中添加 setMat4 函数，用于修改 uniform 矩阵类型的变量：（使用 C 语言的同学将该函数写在其余 setXXX 函数下方，去掉最后的 const 关键字）

```
void setMat4(const std::string& name, glm::mat4 value) const
{
    glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE, glm::value_ptr(value));
}
```

在 main 函数中，激活着色器之后，渲染循环之前，添加下列代码，用于创建一个透视变换矩阵，并使用 Shader 类的 setMat4 函数设置顶点着色器中 projection 的值为该透视变换矩阵（使用 C 语言的同学用 setMat4 函数代替 ourShader.setMat4）：

```
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
ourShader.setMat4("projection", projection);
```

接下来修改顶点数据为正方体的顶点数据，共 36 个顶点。因为 OpenGL 通过绘制三角形面画出所需物体，正方体每个面是由两个三角形面组成的，6 个面就是 12 个三角形面，每个三角形有三个顶点，因此一共用到了 36 个顶点。下方给出顶点数据，共 36 行，每行代表一个顶点信息，前三个数据是坐标信息，后三个是颜色信息：

```
float cubeVertices[] = {
    -0.5f, -0.5f, -3.5f,  0.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -3.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -3.5f,  1.0f, 1.0f, 0.0f,
     0.5f,  0.5f, -3.5f,  1.0f, 1.0f, 0.0f,
    -0.5f,  0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, -3.5f,  0.0f, 0.0f, 0.0f,
```

```
    -0.5f, -0.5f,  -2.5f,  0.0f, 0.0f, 0.0f,
     0.5f, -0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  -2.5f,  1.0f, 1.0f, 0.0f,
     0.5f,  0.5f,  -2.5f,  1.0f, 1.0f, 0.0f,
    -0.5f,  0.5f,  -2.5f,  0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f,  -2.5f,  0.0f, 0.0f, 0.0f,

    -0.5f,  0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -3.5f,  1.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f,  -2.5f,  0.0f, 0.0f, 0.0f,
    -0.5f,  0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,

     0.5f,  0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -3.5f,  1.0f, 1.0f, 0.0f,
     0.5f, -0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
     0.5f, -0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
     0.5f, -0.5f,  -2.5f,  0.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,

    -0.5f, -0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
     0.5f, -0.5f, -3.5f,  1.0f, 1.0f, 0.0f,
     0.5f, -0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
     0.5f, -0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f,  -2.5f,  0.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, -3.5f,  0.0f, 1.0f, 0.0f,

    -0.5f,  0.5f, -3.5f,  0.0f, 1.0f, 0.0f,
     0.5f,  0.5f, -3.5f,  1.0f, 1.0f, 0.0f,
     0.5f,  0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
     0.5f,  0.5f,  -2.5f,  1.0f, 0.0f, 0.0f,
    -0.5f,  0.5f,  -2.5f,  0.0f, 0.0f, 0.0f,
    -0.5f,  0.5f, -3.5f,  0.0f, 1.0f, 0.0f
};
```

修改顶点数据后，记得自行修改渲染循环中的 glDrawArrays 函数，将最后一个参数修改为 36，即绘制前 36 个顶点。

运行代码，结果如图 1-10 所示。

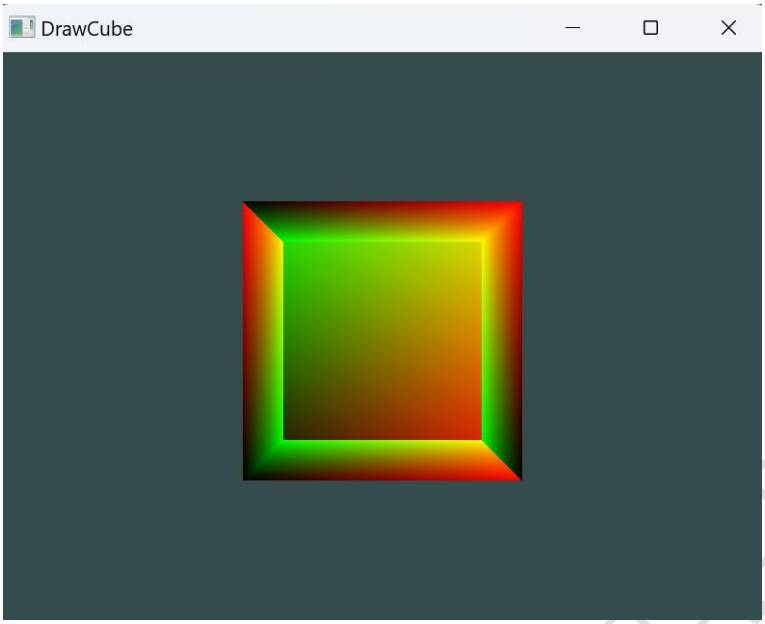


图 1-10 未开深度测试的立方体绘制结果

这是因为没开启深度测试。开启深度测试后，OpenGL 会将被其他面遮挡的面剔除不绘制。开启深度测试代码如下，添加到 main 函数，渲染循环之前：

```
1. glEnable(GL_DEPTH_TEST);
```

开启深度测试后，还需要每帧清空深度缓冲，否则缓冲中永远保存离相机最近的面的深度信息，所有面都会被剔除不绘制。将渲染循环中的 glClear 函数修改，添加对深度缓冲的清空操作：

```
1. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

1.4 实验结果

运行代码，现在能绘制出正确的立方体。如图 1-11 所示。

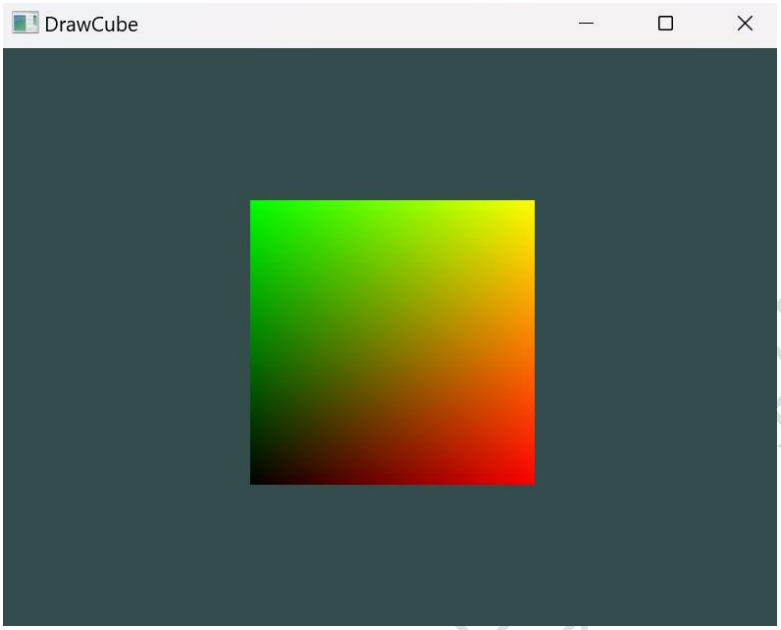


图 1-11 开启深度测试后的立方体绘制结果

实验完成后可以思考一下下列问题：

1. 为什么绘制出的是正方形？
2. 为什么只显示一个面的图像，不显示多个面的图像？

实验二 图形几何变换

2.1 实验目的

熟悉矩阵变换函数，实现图形的平移、旋转和缩放效果。

2.2 实验内容

在上一章立方体绘制的基础上，学习使用矩阵进行图形几何变换。本实验会增加键鼠输入、图形平移旋转缩放的相关代码，详见下一节。

2.3 实验步骤

2.3.1 键鼠输入

在上一章实验代码的基础上进行编写。首先使用枚举类型记录当前处于什么变换模式。

```
enum TransMode {
    Translate,
    Rotate,
    Scale
};
TransMode transModeSelected = Rotate;
```

接下来编写 keyCallBack 函数，用于读取键盘输入。代码如下：

```
void keyCallBack(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    //按 Esc 键退出
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    //按 0 重置变换
    if (glfwGetKey(window, GLFW_KEY_0) == GLFW_PRESS)
    {
        //自行编写代码,通过修改变换矩阵达到重置效果
    }
    //按 1 切换填充/线框模式
    if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS)
        shouldDrawLine = !shouldDrawLine;//要先声明 bool 类型全局变量 shouldDrawLine
    //按 R 开启旋转模式
```

```
    if (glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS)
        transModeSelected = Rotate;
    //按T开启平移模式
    if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS)
        transModeSelected = Translate;
    //按S开启缩放模式
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        transModeSelected = Scale;
}
```

在 main 函数中绑定键盘输入函数，绑定后 glfw 会自动调用我们编写的 keyCallBack 函数。绑定键盘输入代码如下：

```
glfwSetKeyCallback(window, keyCallBack);
```

创建两个 float 变量，用于记录上一帧鼠标的位置坐标。初始值设为一个很大的数（大于窗口宽度），表示位置坐标还未初始化，上一帧鼠标位置不明。

```
float lastX=10000, lastY=10000;
```

编写 mouseCallBack 函数，

```
void mouseCallBack(GLFWwindow* window, double xpos, double ypos)
{
    //当鼠标左键被按住时
    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_1) == GLFW_PRES
S)
    {
        //lastX,lastY无效，将其重置
        if (lastX > SCR_WIDTH)//SCR_WIDTH是窗口宽度
        {
            lastX = xpos;
            lastY = ypos;
            return;
        }
        if (transModeSelected == Translate)
        {
            //自行编写代码，结合鼠标坐标变化量平移立方体
        }
        else if (transModeSelected == Rotate)
        {
            //自行编写代码，结合鼠标坐标变化量旋转立方体
        }
        else if (transModeSelected == Scale)
        {
            //自行编写代码，结合鼠标坐标变化量缩放立方体
        }
        model = translation * rotation * scale;
```

```
        ourShader.setMat4("model", model);//更新顶点着色器的model
        lastX = xpos;
        lastY = ypos;
    }
    else //鼠标左键没被按下，将上一帧坐标无效化
    {
        //将lastX无效化
        lastX = 10000;
    }
}
```

像绑定键盘输入一样绑定鼠标输入，代码如下：

```
glfwSetCursorPosCallback(window, mouseCallBack);
```

上述代码中出现了一些需要自行编写代码的部分，将在接下来的章节中给出编写思路。

2.3.2 变换矩阵

使用矩阵进行几何变换，下面是要用到的六个矩阵，声明为全局变量：

```
glm::mat4 model = glm::mat4(1.0f);
glm::mat4 view = glm::mat4(1.0f);
glm::mat4 projection = glm::mat4(1.0f);
glm::mat4 translation = glm::mat4(1.0f);
glm::mat4 rotation = glm::mat4(1.0f);
glm::mat4 scale = glm::mat4(1.0f);
```

glm::mat4(1.0f)表示 4*4 单位矩阵。

先看前三个矩阵，我们使用 model 矩阵储存物体在世界坐标系中的位置、旋转、缩放信息，用 view 矩阵储存摄像机在世界坐标系中的位置、旋转、缩放信息，用 projection 矩阵储存透视信息。其中，projection 的值可以直接由透视函数生成，具体代码与实验一相同。view 矩阵保持单位矩阵的值即可，在下一章实验才会用到。

修改顶点着色器的代码，先声明这三个矩阵为 uniform 类型：

```
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
```

再给顶点坐标左乘这三个矩阵，这样得到的结果就是变换后的顶点位置。三个矩阵相乘的顺序不能变，最靠右的矩阵最先做乘法：

```
gl_Position = projection * view * model * vec4(aPos, 1.0f);
```

再看后三个矩阵 translation、rotation、scale，它们分别用于储存物体的

平移信息、旋转信息、缩放信息，用三个矩阵的乘积给 model 赋值，此处相乘顺序也不能改动，代码如下：（写在渲染循环中，再自行编写代码将 model 的值用 setMat4 函数更新到顶点着色器）

```
model = translation * rotation * scale;
```

可以想一想，为什么 translation 矩阵要放在最左边？

下面介绍 opengl 世界坐标系和屏幕坐标系，如图 2-1 所示。物体坐标使用世界坐标系，例如将物体向靠近屏幕方向移动 3 单位长度，就是移动（0，0，3）。鼠标坐标使用屏幕坐标系，窗口左上角为（0，0），右下角为（窗口宽度，窗口高度）。

图 2-1 opengl 世界坐标系（左）和屏幕坐标系（右）

由于本章实验引入了 model 矩阵储存立方体的位置信息，上一章使用的中心位于（0，0，-3）的立方体顶点数据需要换成中心位于（0，0，0）的立方体顶点数据，然后使用平移矩阵将其平移至（0，0，-3）的初始位置。新的顶点数据如下，使用其替换上一章的顶点数据：

```
float vertices[] = {
-0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
 0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
-0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
-0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,

-0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
 0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
 0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
 0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
-0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
-0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,

-0.5f,  0.5f,  0.5f,  0.0f, 0.0f, 1.0f,
-0.5f,  0.5f, -0.5f,  0.0f, 0.0f, 1.0f,
-0.5f, -0.5f, -0.5f,  0.0f, 0.0f, 1.0f,
```

```
-0.5f, -0.5f, -0.5f,  0.0f, 0.0f, 1.0f,
-0.5f, -0.5f,  0.5f,  0.0f, 0.0f, 1.0f,
-0.5f,  0.5f,  0.5f,  0.0f, 0.0f, 1.0f,

 0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 0.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 0.0f,
 0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 0.0f,
 0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 0.0f,
 0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 0.0f,
 0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 0.0f,

-0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 1.0f,
 0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 1.0f,
 0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 1.0f,
 0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 1.0f,
-0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 1.0f,
-0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 1.0f,

-0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 1.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 1.0f,
 0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 1.0f,
 0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 1.0f,
-0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 1.0f,
-0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 1.0f
};
```

接下来讲解变换矩阵的赋值操作。

下面三行代码分别是平移、旋转、缩放函数，在传入参数的原矩阵基础上进行变换，传出变换后的矩阵。

```
1. glm::translate(glm::mat4 原矩阵, glm::vec3 平移向量);
2. glm::rotate(glm::mat4 原矩阵, float 旋转角度, glm::vec3 旋转轴);
3. glm::scale(glm::mat4 原矩阵, glm::vec3 缩放向量);
```

请使用这三个函数，结合鼠标坐标变化量，对 translation、rotation、scale 三个矩阵进行更改，补全 mouseCallBack 函数中缺少的部分，以及 keyCallBack 函数中按 0 重置变换部分。

要求：变换应当每帧实时更新，而不是松开鼠标左键后才统一变换；平移只需进行 x，y 轴方向的平移，不涉及 z 轴；旋转只进行绕 x，y 轴的旋转，不涉及绕 z 轴旋转，旋转方向应与鼠标移动方向一致，例如鼠标向右移动，则绕 y 轴正向旋转，鼠标向上移动，则绕 x 轴反向旋转；缩放操作为鼠标向上移动则放大立方体，向下则缩小；重置操作应将物体重置到（0，0，-3）的坐标处，清空旋转和缩放信息（记得将该状态也设置为物体的初始状态）。

（提示：平移和缩放操作都可以将 translation/scale 矩阵作为原矩阵传入，相当于对上一帧的物体状态直接进行平移/缩放，但旋转操作不行。旋转操作需要使用新的单位矩阵进行旋转，再将得到的结果与 rotation 矩阵相乘来将本帧的新旋转量累加上去。这是因为旋转后物体自身坐标轴方向也会变化，因此不能在 rotation 的基础上直接旋转。）

下面给出平移操作的参考代码：

```
1. if (transModeSelected == Translate)
2. {
3.     translation = glm::translate(translation, glm::vec3((xpos - lastX) * 0.005, -(ypos - lastY) * 0.005, 0.0f));//0.005 用于控制平移速度
4. }
```

编写完代码后运行，应能按 t/r/s 切换平移/旋转/缩放模式，按住鼠标左键进行相应变换，按 0 重置回初始位置，结果示意图如图 2-2 所示：

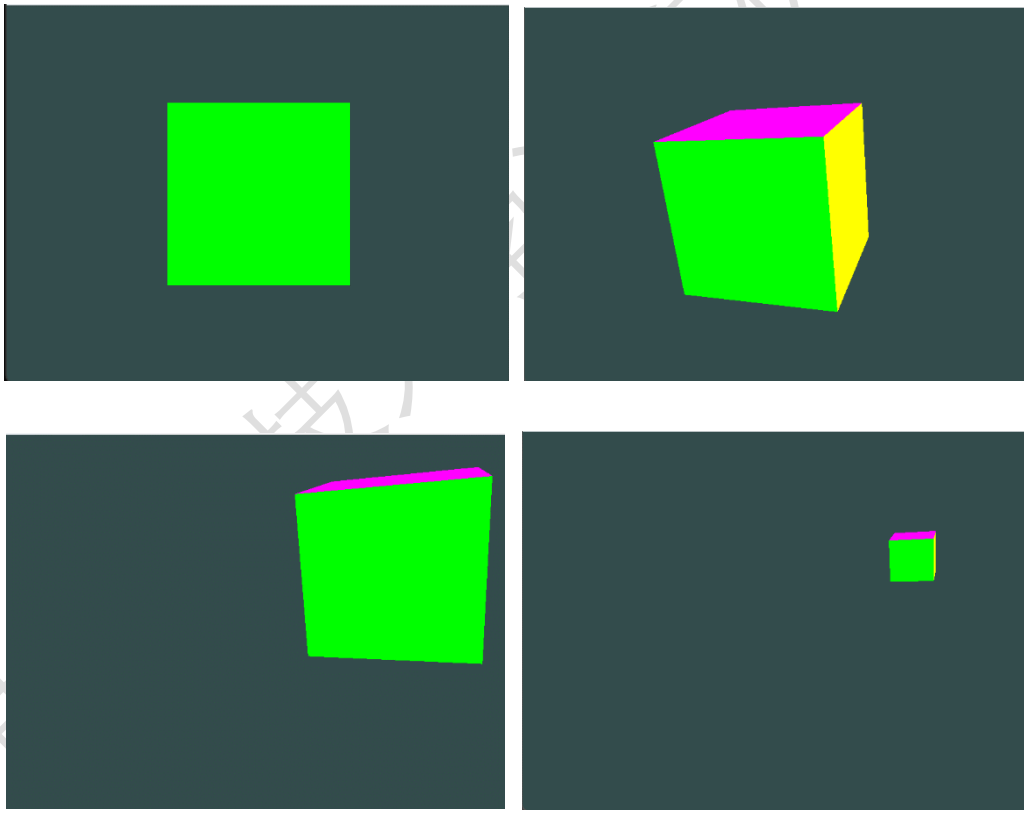


图 2-2 初始位置（左上） 旋转后（右上）
平移后（左下） 缩放后（右下）

2.3.3 线框绘制

下面实现按 1 切换填充绘制和线框绘制模式。填充绘制是将立方体的六个面全部绘制，线框绘制则只画出每个面的边框。

OpenGL 有 glPolygonMode 函数用于切换线框模式，但这样绘制出的线框是以三角形面为基础构成的，如图 2-3 所示。

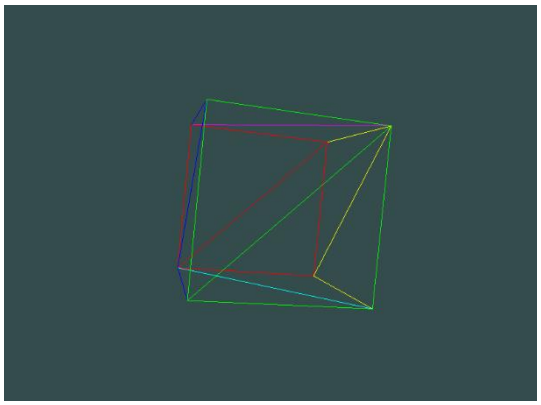


图 2-3 三角形线框图

要画出没有对角线的线框立方体，需要将 glDrawArrays 函数的第一个传入参数由 GL_TRIANGLES 改为 GL_LINE_LOOP，这样该函数绘制的就不是三角形面，而是多边形线框。

目前使用的顶点数组 vertices，共有 36 个顶点数据，是按三角形面的顶点顺序记录的。要想绘制多边形线框，需要创建一个新的顶点数组 frameVertices，依序储存立方体每个面上的四个顶点，一共六个面 24 个点。这样我们使用 glDrawArrays 函数绘制线框图时，需要绘制六次，每次绘制一个面边框的四条线，代码如下：

```
1. glDrawArrays(GL_LINE_LOOP, 0, 4);
2. glDrawArrays(GL_LINE_LOOP, 4, 4);
3. glDrawArrays(GL_LINE_LOOP, 8, 4);
4. glDrawArrays(GL_LINE_LOOP, 12, 4);
5. glDrawArrays(GL_LINE_LOOP, 16, 4);
6. glDrawArrays(GL_LINE_LOOP, 20, 4);
```

新的线框顶点数据如下，共 24 个顶点，将其加到 vertices 数组后面：

```
float frameVertices[] = {
-0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
 0.5f, -0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,
-0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 0.0f,

-0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
 0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
 0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,
-0.5f,  0.5f,  0.5f,  0.0f, 1.0f, 0.0f,

-0.5f,  0.5f,  0.5f,  0.0f, 0.0f, 1.0f,
```

```
-0.5f,  0.5f, -0.5f,  0.0f, 0.0f, 1.0f,
-0.5f, -0.5f, -0.5f,  0.0f, 0.0f, 1.0f,
-0.5f, -0.5f,  0.5f,  0.0f, 0.0f, 1.0f,

 0.5f,  0.5f,  0.5f,  1.0f, 1.0f, 0.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 1.0f, 0.0f,
 0.5f, -0.5f, -0.5f,  1.0f, 1.0f, 0.0f,
 0.5f, -0.5f,  0.5f,  1.0f, 1.0f, 0.0f,

-0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 1.0f,
 0.5f, -0.5f, -0.5f,  0.0f, 1.0f, 1.0f,
 0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 1.0f,
-0.5f, -0.5f,  0.5f,  0.0f, 1.0f, 1.0f,

-0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 1.0f,
 0.5f,  0.5f, -0.5f,  1.0f, 0.0f, 1.0f,
 0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 1.0f,
-0.5f,  0.5f,  0.5f,  1.0f, 0.0f, 1.0f
};
```

因为需要按 1 切换三角面和线框图，两种顶点数据都是要用到的，我们新建一个 frameVBO 和 frameVAO，用于对线框顶点数据进行绑定。仿照上一章实验中读取 vertices 顶点数据到 VBO，再绑定到 VAO，请自行完成读取 frameVertices 顶点数据到 frameVBO，再绑定到 frameVAO。

这样我们就有了 VAO 和 frameVAO 两个顶点数组。

用创建的 bool 类型的 shouldDrawLine 变量记录当前是不是线框模式。再在渲染循环中修改 glDrawArrays 的代码，使 shouldDrawLine 为 true 时，使用 glBindVertexArray 函数切换绑定的顶点数组为 frameVAO，再画出线框立方体；shouldDrawLine 为 true 时则切换绑定为 VAO，然后画出立方体。自行实现该部分代码。

2.4 实验结果

运行程序，可以得到如下结果：画出立方体，按 1 后切换为线框图，如图 2-4 所示。

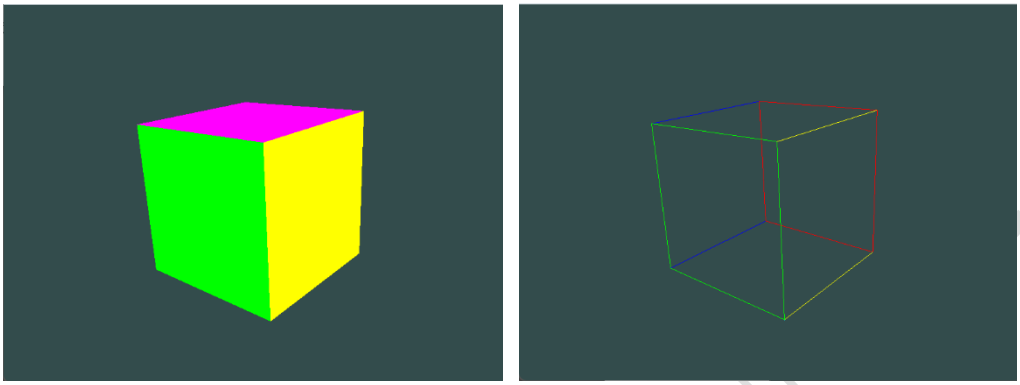


图 2-4 立方体原图（左） 线框图（右）

实验三 图形观察变换

3.1 实验目的

熟悉观察变换函数，实现相机漫游效果。

3.2 实验内容

在上一章的基础上，学习使用矩阵进行图形观察变换。

3.3 实验步骤

3.3.1 地面绘制

在立方体下方绘制一个灰色平面当地面，以便于区分立方体的几何变换和观察变换。可以直接将地面顶点数据分别加到 vertices 和 frameVertices 顶点数组最后。

地面顶点数据如下，将其添加到 vertices 数组最后：

```
-1.0f, 0.0f, -1.0f,  0.5f, 0.5f, 0.5f,
 1.0f, 0.0f, -1.0f,  0.5f, 0.5f, 0.5f,
 1.0f, 0.0f,  1.0f,  0.5f, 0.5f, 0.5f,
 1.0f, 0.0f,  1.0f,  0.5f, 0.5f, 0.5f,
-1.0f, 0.0f,  1.0f,  0.5f, 0.5f, 0.5f,
-1.0f, 0.0f, -1.0f,  0.5f, 0.5f, 0.5f
```

地面框架顶点数据如下，添加到 frameVertices 数组最后：

```
-1.0f, 0.0f, -1.0f,  0.5f, 0.5f, 0.5f,
 1.0f, 0.0f, -1.0f,  0.5f, 0.5f, 0.5f,
 1.0f, 0.0f,  1.0f,  0.5f, 0.5f, 0.5f,
-1.0f, 0.0f,  1.0f,  0.5f, 0.5f, 0.5f
```

添加好后，在创建 model 矩阵的代码旁创建 groundModel 矩阵，再将 model 改名为 cubeModel（右键 model，在弹出来的菜单中选择重命名）。分别用这两个矩阵储存立方体和地面的几何变换信息。（不要修改顶点着色器中的 model）。代码如下：

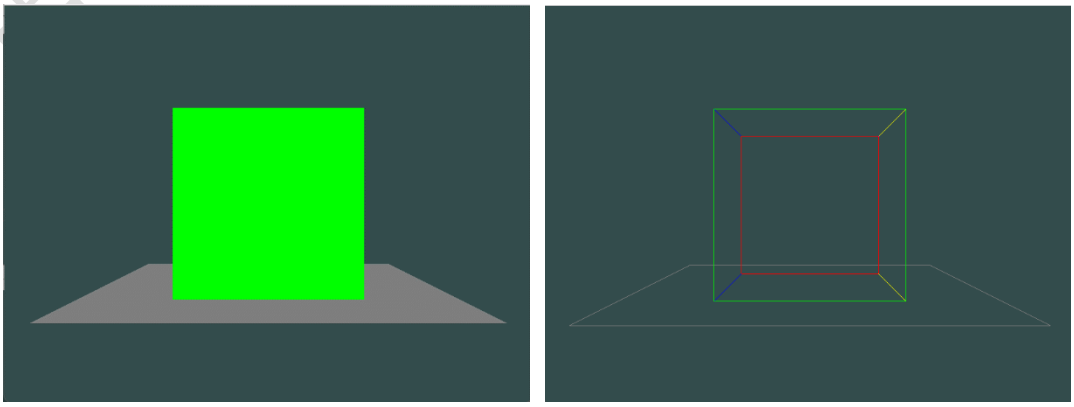
```
1. glm::mat4 cubeModel = glm::mat4(1.0f);
2. glm::mat4 groundModel = glm::mat4(1.0f);
```

记得在渲染循环前使用 glm::translate 函数将 groundMoudel 平移（0，-0.5，-3），以放在立方体正下方。

然后修改绘制函数，添加绘制地面的部分，并在每次绘制立方体和地面前，分别将 cubeModel 和 groundModel 的值写入顶点着色器中的 model 变量。渲染循环中的相应代码改为如下：

```
if (shouldDrawLine) //绘制线框
{
    glBindVertexArray(frameVAO);

    ourShader.setMat4("model", cubeModel); //改为立方体变换矩阵，再绘制立方体
    glDrawArrays(GL_LINE_LOOP, 0, 4);
    glDrawArrays(GL_LINE_LOOP, 4, 4);
    glDrawArrays(GL_LINE_LOOP, 8, 4);
    glDrawArrays(GL_LINE_LOOP, 12, 4);
    glDrawArrays(GL_LINE_LOOP, 16, 4);
    glDrawArrays(GL_LINE_LOOP, 20, 4);

    ourShader.setMat4("model", groundModel); //改为地面变换矩阵，再绘制地面
    glDrawArrays(GL_LINE_LOOP, 24, 4);
}
Else //绘制面
{
    glBindVertexArray(VAO);

    ourShader.setMat4("model", cubeModel);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    ourShader.setMat4("model", groundModel);
    glDrawArrays(GL_TRIANGLES, 36, 6);
}
```

运行代码，出现如图 3-1 所示结果，并且对立方体进行变换不会影响地面。



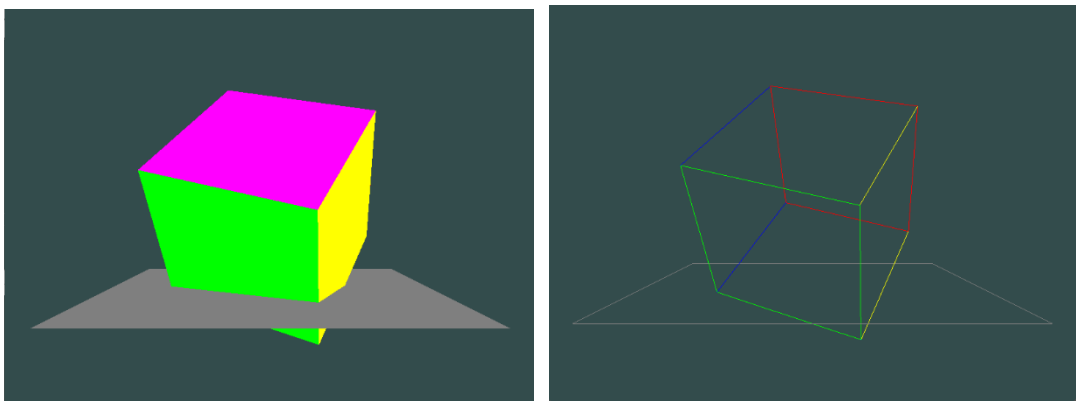


图 3-1 立方体和地面原图（左上） 线框图（右上）
旋转立方体（左下） 旋转立方体的线框图（右下）

3.3.2 五视图实现

本小节实现按键切换不同视图，包括下列功能：

- 数字键 2，前视图
- 数字键 4，左视图
- 数字键 6，右视图
- 数字键 8，后视图
- 数字键 5，顶视图

本章使用 view 矩阵模拟相机功能。上一个实验中也用到了 view 矩阵，不过始终保持单位矩阵的值，不对结果产生影响。

首先新建三个全局 vec3 变量 cameraPosition、cameraTarget、cameraUp。其中 cameraUp 表示相机的上向量，一般（0，1，0）即可。

```
1. glm::vec3 cameraPosition=glm::vec3(0.0f,0.0f,0.0f);//相机位置
2. glm::vec3 cameraTarget=glm::vec3(0.0f,0.0f,-3.0f);//相机看向的位置
3. glm::vec3 cameraUp=glm::vec3(0.0f,1.0f,0.0f);//相机上向量，需要标准化
```

然后在渲染循环中每帧更新 view 的值，lookAt 函数使用相机位置、相机看向位置、上向量三个参数，输出 view 矩阵：

```
1. view = glm::lookAt(cameraPosition,cameraTarget,cameraUp);
2. ourShader.setMat4("view", view);
```

在 keyCallBack 函数中添加五视图功能，以前视图和顶视图为例：

```
1. if (glfwGetKey(window, GLFW_KEY_KP_2) == GLFW_PRESS)//前视图，
   GLFW_KEY_KP_2 表示数字键 2
2. {
3.     cameraPosition = glm::vec3(0.0f,0.0f,0.0f);
4.     cameraTarget = glm::vec3(0.0f,0.0f,-3.0f);//看向立方体坐标（0，0，
   -3）
```

```
}
if (glfwGetKey(window, GLFW_KEY_KP_5) == GLFW_PRESS)//顶视图
{
    cameraPosition = glm::vec3(0.0f, 3.0f, -2.99f);//由于相机在竖直方向旋转超过90°会产生视野翻转，因此相机z坐标设置为-2.99而不是-3，其余视图不用考虑视野翻转
    cameraTarget = glm::vec3(0.0f, 0.0f, -3.0f);
}
```

自行实现其余视图，提示一下：物体坐标为（0，0，-3），各个视图的摄像机坐标应距离物体 3 个单位长度。

3.3.3 相机移动

本小节实现 WASD 按键前后左右移动相机，Q 向下移动，E 向上移动。

创建相机移动速度变量 cameraSpeed：

```
const float cameraSpeed = 0.05f;
```

计算出相机向前的方向向量 cameraFront 和向右的方向向量 cameraRight（这两个值需要用之前更新）：

```
glm::vec3 cameraFront = glm::normalize(cameraTarget - cameraPosition);
glm::vec3 cameraRight = glm::normalize(glm::cross(cameraFront, cameraUp));
```

在 keyCallBack 函数中实现 WASDQE 六个键控制相机移动，以向前移动为例：

```
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) //按w向前移动
{
    cameraPosition += cameraSpeed * cameraFront;
    cameraTarget += cameraSpeed * cameraFront;
}
```

同时平移 cameraPosition 和 cameraTarget，自行实现其余按键移动代码，向上和向下沿 y 轴方向即可。再修改之前的代码，将控制缩放的按键由 S 键换为 Y 键。

3.3.4 相机旋转

本小节实现按住鼠标右键旋转相机视角。

相机旋转分为三个方向，如图 3-2 所示。实验中只涉及到 pitch 俯仰角和 yaw 偏航角。

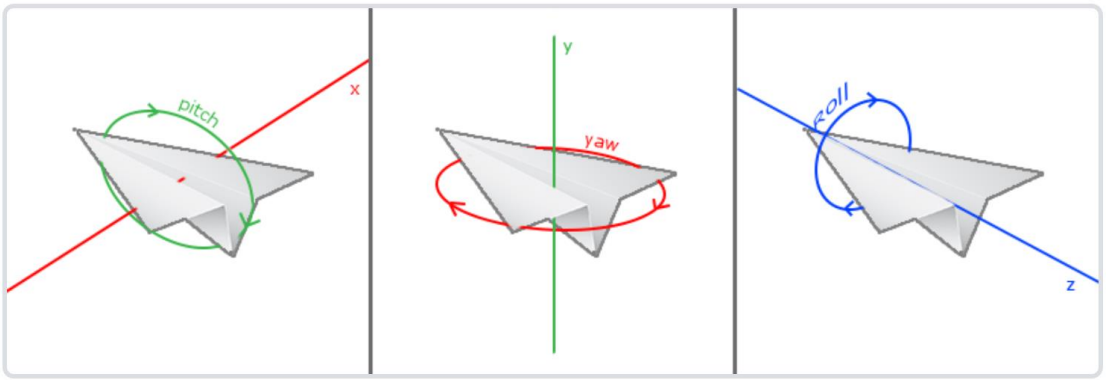


图 3-2 相机旋转的三个方向

修改 mouseCallBack 函数，添加按住右键旋转视角功能。像上一章实验的立方体几何变换一样，要用到鼠标坐标变化量。

先进行相机水平旋转，算出相机向前的方向向量 cameraFront，将其绕 cameraUp 旋转，旋转角度与鼠标 x 坐标变化量相关，即可得到水平旋转后的新相机方向向量 newCameraFront。

再进行垂直旋转，算出相机向右的方向向量 cameraRight，将 newCameraFront 绕 cameraRight 旋转，旋转角度与鼠标 y 坐标变化量相关，即可得到垂直旋转后的相机方向向量。与水平旋转不同的是，需要限制俯仰角 pitch 介于-89.9° 至 89.9° 之间，以防视野翻转（因为 90° 时已经产生视野翻转，所以设为 89.9° ）。

```
1. void mouseCallBack(GLFWwindow* window, double xpos, double ypos)
2. {
3.     if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_1) == GLFW_PRES
   S)//按住鼠标左键移动立方体
4.     {
5.         //lastX,lastY 无效，将其重置
6.         if (lastX > SCR_WIDTH)
7.         {
8.             lastX = xpos;
9.             lastY = ypos;
10.            return;
11.        }
12.        if (transModeSelected == Translate)
13.        {
14.            translation = glm::translate(translation, glm::vec3((xpo
   s - lastX) * 0.005, -(ypos - lastY) * 0.005, 0.0f));
15.        }
16.        else if (transModeSelected == Rotate)
17.        {
```

```
18.             rotation = glm::rotate(glm::mat4(1.0f), (float)(xpos - l
    astX) * 0.01f, glm::vec3(0.0f, 1.0f, 0.0f)) * rotation;
19.             rotation = glm::rotate(glm::mat4(1.0f), -
    (float)(ypos - lastY) * 0.01f, glm::vec3(-
    1.0f, 0.0f, 0.0f)) * rotation;
20.         }
21.         else if (transModeSelected == Scale)
22.         {
23.             float deltaScale = 1 - (ypos - lastY) * 0.01f;
24.             scale = glm::scale(scale, glm::vec3(deltaScale, deltaSca
    le, deltaScale));
25.         }
26.         cubeModel = translation * rotation * scale;
27.         ourShader.setMat4("model", cubeModel);
28.         lastX = xpos;
29.         lastY = ypos;
30.     }
31.     else if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_2) == GLFW
    _PRESS)//按住右键旋转视角
32.     {
33.         //lastX,lastY 无效，将其重置
34.         if (lastX > SCR_WIDTH)
35.         {
36.             lastX = xpos;
37.             lastY = ypos;
38.             return;
39.         }
40.
41.         //相机水平旋转
42.         glm::vec3 cameraFront = glm::normalize(cameraTarget - camera
    Position);
43.         glm::vec3 newCameraFront = glm::rotate(glm::mat4(1.0f), (flo
    at)-
    (xpos - lastX) * cameraSensitivity, cameraUp) * glm::vec4(cameraFron
    t,1.0f);
44.         //相机垂直旋转
45.         glm::vec3 cameraRight = glm::normalize(glm::cross(newCameraF
    ront, cameraUp));
46.         float pitch = 90 - glm::degrees(glm::acos(glm::dot(cameraUp,
     newCameraFront)));
47.         if (pitch + -(ypos - lastY) * cameraSensitivity > 89.9f)
48.             newCameraFront= glm::rotate(glm::mat4(1.0f), (float)(89.
    9f-pitch), cameraRight) * glm::vec4(newCameraFront, 1.0f);
```

```
        else if (pitch + -(ypos - lastY) * cameraSensitivity < -89.9f)
            newCameraFront = glm::rotate(glm::mat4(1.0f), (float)(-89.9f-pitch), cameraRight) * glm::vec4(newCameraFront, 1.0f);
        else
            newCameraFront = glm::rotate(glm::mat4(1.0f), (float)(-(ypos - lastY) * cameraSensitivity),cameraRight) * glm::vec4(newCameraFront, 1.0f);
        //更新 cameraTarget
        cameraTarget = cameraPosition + newCameraFront;

        //更新 lastX、lastY
        lastX = xpos;
        lastY = ypos;
    }
    else
    {
        //将 lastX 无效化
        lastX = 10000;
    }
}
```

3.4 实验结果

运行代码测试，应能使用 WASDQE 按键移动相机，按住鼠标右键旋转相机。如图 3-3 所示。

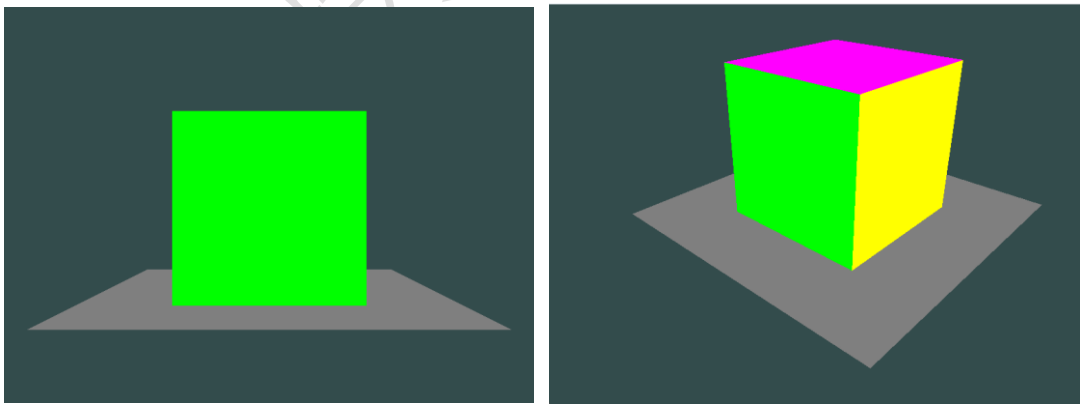


图 3-3 原始视角（左） 自由移动后的视角（右）

实验四 模型导入

4.1 实验目的

熟悉.obj 类型格式，掌握场景、网格、面片之间的关系与储存形式，读入并绘制模型。

4.2 实验内容

在前三章知识的基础上，学习使用 Assimp 库读取.obj 格式文件，将其数据转换为顺序存储后绘制模型，并加入光照效果。

4.3 实验步骤

4.3.1Assimp 库安装

Assimp 库用于读取模型文件，按照下述网站的教程安装：https://blog.csdn.net/derbi123123/article/details/105783048。（最后在项目-属性-链接器-输入-附加依赖项中添加 Assimp 的 lib 文件名时，根据实际的 lib 文件名填写，例如 assimp-vc143-mtd.lib）

安装好后新建项目，步骤参见 1.3.1，引入 assimp 头文件：

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
```

4.3.2 读取模型文件

先创建下列结构体用于储存顶点和网格信息：

```
struct Vertex {
    glm::vec3 Position; //顶点坐标
    glm::vec3 Normal; //顶点法向量
};

struct Mesh {
    vector<Vertex>       vertices;
    vector<unsigned int> indices;
    unsigned int VAO, VBO, EBO;
};
```

一个模型（model）可能存在不止一个网格（mesh），每个网格是一些顶点的

连通集，不同网格之间没有边或面进行连接。

Mesh 结构体用于储存网格信息，vertices 储存顶点坐标，indices 储存顶点下标，使用顶点下标即可获取对应顶点的坐标，从而绘制出三角形面。VAO 和 VBO 之前介绍过了，EBO 用于绑定顶点下标。

创建一个 Mesh 类型的 vector 用于储存模型的所有网格：

```
vector<Mesh> meshes;
```

创建 LoadModel 函数用于加载模型。使用 Assimp 库的 Importer 可以很方便地读取模型文件，但读取到的数据是像树一样的层次存储，需要被我们改成顺序存储才能绑定到顶点数组。LoadModel 代码如下：

```
void loadModel(string const& path) //path 是模型文件相对路径
{
    //读取模型文件
    Assimp::Importer importer;
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_GenSmoothNormals | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
    // 错误检查
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
        return;
    }
    //遍历所有节点，储存各节点中的网格
    processNode(scene->mRootNode, scene);
}
```

这里读取的是.obj 类型的模型文件，这种模型文件包含一个场景（scene），场景内包含所有网格（mesh），通过节点（node）像文件夹那样一层一层地储存这些网格。场景有一个根节点（scene->mRootNode），创建 processNode 函数使用递归方式从根节点遍历所有节点，并将遍历到的网格储存到 meshes 向量中。processNode 函数如下：

```
//用递归方式遍历所有节点
void processNode(aiNode* node, const aiScene* scene)
{
    // 遍历并储存当前节点下的所有网格
    for (unsigned int i = 0; i < node->mNumMeshes; i++)
    {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
```

```
        meshes.push_back(processMesh(mesh, scene)); //将网格存入 meshes 向量
    }
    // 递归查找当前节点下的所有子节点
    for (unsigned int i = 0; i < node->mNumChildren; i++)
    {
        processNode(node->mChildren[i], scene);
    }
}
```

编写 processMesh 函数用于将网格信息存入 meshes 向量：

```
Mesh processMesh(aiMesh* mesh, const aiScene* scene)
{
    // 要填写到Mesh 中的变量
    vector<Vertex> vertices;
    vector<unsigned int> indices;

    // 遍历网格的所有顶点
    for (unsigned int i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        glm::vec3 vector; //用于临时存储信息
        //顶点位置
        vector.x = mesh->mVertices[i].x;
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;
        // 顶点法向量
        if (mesh->HasNormals())
        {
            vector.x = mesh->mNormals[i].x;
            vector.y = mesh->mNormals[i].y;
            vector.z = mesh->mNormals[i].z;
            vertex.Normal = vector;
        }
        //将填好数据的顶点加入 vertices 中
        vertices.push_back(vertex);
    }
    // 遍历网格所有面片
    for (unsigned int i = 0; i < mesh->mNumFaces; i++)
    {
        aiFace face = mesh->mFaces[i];
        // 储存顶点下标信息
        for (unsigned int j = 0; j < face.mNumIndices; j++)
            indices.push_back(face.mIndices[j]);
```

```
    }

    Mesh returnMesh; //作为函数返回值
    returnMesh.vertices = vertices;
    returnMesh.indices = indices;
    //绑定VAO 和EBO
    glGenVertexArrays(1, &returnMesh.VAO);
    glGenBuffers(1, &returnMesh.VBO);
    glGenBuffers(1, &returnMesh.EBO);

    glBindVertexArray(returnMesh.VAO);

    glBindBuffer(GL_ARRAY_BUFFER, returnMesh.VBO);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, returnMesh.EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

    //顶点坐标
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
    //顶点法向量
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));

    glBindVertexArray(0);//清除当前绑定的VAO，以便再次调用该函数绑定其他VAO
    return returnMesh;
}
```

4.3.3 绘制模型文件

编写 Draw 绘制函数用于绘制模型：

```
void Draw()
{
    for (unsigned int i = 0; i < meshes.size(); i++)
    {
        glBindVertexArray(meshes[i].VAO);
        glDrawElements(GL_TRIANGLES, static_cast<unsigned int>(meshes[i].indices.size()), GL_UNSIGNED_INT, 0);
```

```
        glBindVertexArray(0);
    }
}
```

除此之外的代码，例如常规步骤的 glfw 和 glad 初始化、创建窗口、渲染循环、开启深度检测等代码保持不变。

变换矩阵 model、view、projection 都与之前实验类似，model 可进行适当变换以将模型绘制在屏幕中央，例如平移(0.0f, -0.5f, -3.0f)。view 保持单位矩阵，projection 与之前实验相同即可。

顶点着色器代码如下：

```
#version 330 core
layout(location = 0) in vec3 aPos;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection*view*model*vec4(aPos, 1.0f);
}
```

片段着色器代码如下：

```
#version 330 core
out vec4 FragColor;

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

这段着色器代码会将模型的每个顶点颜色绘制为白色。

在渲染循环前调用 loadModel 函数加载模型，路径为模型的 obj 文件相对 main 函数代码文件的位置：

```
loadModel("bunny/bunny_10k.obj");
```

在渲染循环中调用 Draw 函数绘制模型。

运行代码，能绘制一个白色兔子，如图 4-1 所示。

图 4-1 运行结果

运行结果只能看出一个兔子的剪影，看不出立体感，是因为每个顶点都被画成纯白色，需要加上光照效果。

修改顶点着色器代码如下：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 Normal;
out vec3 FragPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    Normal = aNormal;   //顶点法向量
    FragPos = vec3(model * vec4(aPos, 1.0)); //用于传递给片段着色器顶点的真实位置（即只考虑 model 矩阵的变换，不考虑透视、相机的变换）
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

修改片段着色器代码如下：

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 FragPos;

```

```
7. void main()
8. {
9.     vec3 lightDir=vec3(0.0,0.0,-1.0); //光的方向（必须是单位向量）
10.    vec3 lightColor=vec3(1.0,1.0,1.0); //光的颜色
11.    vec3 objectColor=vec3(1.0,1.0,1.0); //物体颜色
12.
13.    // ambient，环境光
14.    float ambientStrength = 0.1;
15.    vec3 ambient = ambientStrength * lightColor;
16.
17.    // diffuse，漫反射光
18.    vec3 norm = normalize(Normal);
19.    float diff = max(dot(norm, -lightDir), 0.0);
20.    vec3 diffuse = diff * lightColor;
21.
22.    vec3 result = (ambient + diffuse) * objectColor;
23.    FragColor = vec4(result,1.0);
24.}
```

光照效果由环境光和漫反射光叠加，环境光是环境反射到物体表面的微弱的光，没有环境光的物体照不到光的面就会是纯黑色。可以自行修改光的方向、颜色、物体颜色试试。

4.4 实验结果

运行结果如图 4-2 所示。

图 4-2 加了光照的运行结果