哈尔滨工业大学

计算机图形学与虚拟现实
实验指导手册

2024 年春季学期

授课教师：洪晓鹏

助教：陈鹏锦　魏宇鹏

2024 年 4 月

Contents

实验说明

本课实验一共 8 学时，分为 4 个实验，在各次实验课进行实验验收和答疑，并在**本次实验验收后至下一次实验课实验前**，上交实验报告和代码到 hitcgvr24@163.com。实验报告的模板见群文件。

四个实验均占相同的分数。它们分别是：

1. OpenGL 环境搭建
2. 金刚石的绘制
3. 光照绘制
4. 圆的绘制

实验手册使用 C++11 作为编程语言，使用 Visual Studio 作为 IDE，使用 OpenGL 作为图形库。同学们使用其他任何语言、IDE 和图形库进行实验都是可以的，但是实验指导手册中的代码示例可能不适用。

实验指导手册中的代码示例是为了帮助同学们更好地理解实验内容，只要实现了实验要求，代码风格和代码结构不作要求。指导手册是在 Windows 环境下编写的，但是 Windows、Linux、MacOS 都可以作为实验环境。

我们鼓励同学们使用 Vulkan 或者其他图形库进行实验，也鼓励同学自己实现更多的图形效果，例如阴影、PBR、SSAO 等。

关于环境配置：熟悉 CMake，PreMake 等工具的同学可以使用这些工具来配置环境，不熟悉的同学可以直接使用指导手册的方法。

一、 实验 1: OpenGL 环境搭建

（一） 实验目的

- 搭建 OpenGL 实验环境
- 了解 OpenGL 的基本概念，包括绑定，缓冲区，着色器等
- 用 OpenGL 绘制一个简单的三角形

（二） OpenGL 介绍

OpenGL (Open Graphics Library) 是由 Khronos 组织 (Khronos Group) 维护和推进的一个跨平台的图形 API(Application Programming Interface) 规范 (Specification)。该规范严格规定了每个函数该如何执行，以及它们的输出值，而具体 OpenGL 库由硬件开发者实现。OpenGL 提供了一系列函数和工具，使开发者能够直接与图形硬件进行交互，实现高性能的图形渲染和图形效果。

实验使用的 OpenGL3.3 的规范文档可以在这里找到：规范文档。

OpenGL 状态机 OpenGL 自身是一个巨大的**状态机** (State Machine)：一系列的变量描述 OpenGL 此刻应当如何运行。OpenGL 的状态通常被称为 OpenGL **上下文** (Context)。通常使用如下途径去更改 OpenGL 状态：设置选项，操作缓冲。最后，我们使用当前 OpenGL 上下文来渲染。

OpenGL 对象 在 OpenGL 中一个对象是指一些选项的集合，它代表 OpenGL 状态的一个子集。对象配置了 OpenGL 对应选项的实际状态。比如：

- 顶点数组对象 (Vertex Array Object)
- 缓存对象 (Buffer Object)
- 纹理对象 (Texture Object)
- 帧缓存对象 (Frame Buffer Object)

我们代码操作流一般是：生成一个对象并一个 id 保存它的引用 (*glGen...()*)。然后将其绑定 (*glBind...()*) 到当前上下文中，设置该对象所包括选项的一些内容，接下来调用 (*glSet...()*)，选项的设置将被记录到这一对象中，之后可以直接绑定该对象使用。最后我们将目标位置的对象 id 设回 0(*glBind...(..., 0)*)，解绑这个对象。如代码清单1所示。

Listing 1: OpenGL API 实例

```
1  // 创建对象
2  unsigned int objectId = 0;
3  glGenObject(1, &objectId);
```

```
    // 绑定对象至上下文
    glBindObject(GL_WINDOW_TARGET, objectId);
    // 设置当前绑定到 GL_WINDOW_TARGET 的对象的一些选项
    glSetObjectOption(GL_WINDOW_TARGET,
      GL_OPTION_WINDOW_WIDTH, 800);
    glSetObjectOption(GL_WINDOW_TARGET,
      GL_OPTION_WINDOW_HEIGHT, 600);
    // 将上下文对象设回默认
    glBindObject(GL_WINDOW_TARGET, 0);
```

（三） 创建窗口

Visual Studio 的安装和配置请参考官方文档。在 Visual Studio Installer 中，我们需要安装“使用 C++ 的桌面开发”。

创建项目 首先，打开 Visual Studio，创建一个新的项目。我们选择空项目 (Empty Project)。指导书使用 “learnopengl” 作为项目名称和解决方案名称。我们将在 64 位模式中执行所有操作，因此需要确认在顶部“Debug”旁选择“x64”。创建项目后的文件夹内容应该为:

```
learnopengl/
├── learnopengl.sln
└── learnopengl/learnopengl.vcxproj
```

在后文中,用. 表达在解决方案目录下（learnopengl.sln 所在的目录）。在./learnopengl 中创建文件夹 src，res，lib 和 include。我们将 lib 文件夹用于存放第三方库的 dll 和 lib 文件，include 文件夹用于存放头文件。src 文件夹用于存放源代码，res 文件夹用于存放资源文件。

进行如下设置：

1. 在解决方案管理器里右键项目名，进入属性页
2. 配置属性 →VC++ 目录中，包含目录，右边下拉 → 编辑，在包含目录下方白框中键入“$(ProjectDir)include”，如图1所示。
3. 配置属性 →VC++ 目录中，库目录，右边下拉 → 编辑，在库目录下方白框中键入“$(ProjectDir)lib”，如图1所示。
4. 确认。

GLFW 在GLFW 官网下载 GLFW 的对应平台版本，解压后将其 include 文件夹下的内容 (GLFW) 复制到你的项目的 include 文件夹下，将 lib-vc{你的 visual studio 版本} 文件夹下的内容 (*.dll, *.lib) 复制到你的项目的 lib 文件夹下。

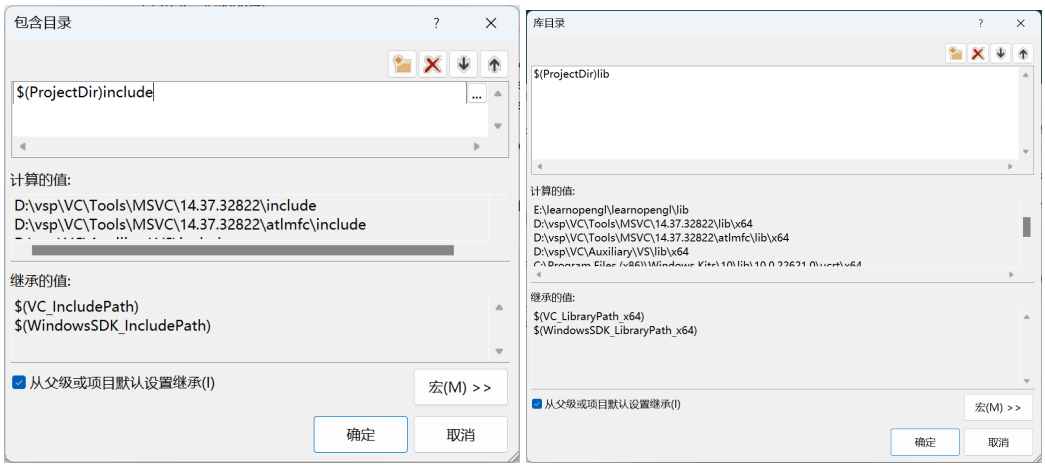

Figure 1: 包含目录、库目录的配置

GLAD 在GLAD 在线服务，将语言 (Language) 设置为 C/C++，在 API 选项中，选择 3.3 以上的 OpenGL(gl) 版本（我们的教程中将使用 3.3 版本，但更新的版本也能用）。之后将模式 (Profile) 设置为 Core，并且保证选中了生成加载器 (Generate a loader) 选项。选择完之后，点击生成 (Generate) 按钮来生成库文件。点击 glad.zip 下载。解压后，将 include 中的内容 (glad, KHR) 复制到你的项目的 include 文件夹下，将 src 中的 glad.c 复制到你的项目的 src 文件夹下。

1. 配置属性 → 链接器 → 输入 (Input) 中，附加依赖项，右边下拉 → 编辑，在附加依赖项下方白色框中键入“glfw3.lib;opengl32.lib”。
2. 解决方案资源管理器 → 源文件 → 右键 → 添加 → 现有项,找到./learnopengl/src/glad.c 文件。

创建窗口对象 接下来我们创建 main 函数，在这个函数中我们将会实例化 GLFW 窗口：

Listing 2: main 函数

```
#include <glad/glad.h>
#include <iostream>
#include <GLFW/glfw3.h>
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;
int main()
{
    glfwInit();
    // 设置OpenGL版本为3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // 设置OpenGL为核心模式 (只能使用OpenGL功能的一个子集)
    glfwWindowHint(GLFW_OPENGL_PROFILE,
       GLFW_OPENGL_CORE_PROFILE);
    // 或者可以使用向后兼容的OpenGL功能
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

```
16      // 创建窗口对象
17      // 参数：宽度，高度，窗口标题，不使用全屏，不共享资源
18      GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
            SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
19      if (window == NULL)
20      {
21          std::cout << "Failed to create GLFW window" << std
                ::endl;
22          glfwTerminate();
23          return -1;
24      }
25      glfwMakeContextCurrent(window);
26      // GLAD初始化
27      if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress
            ))
28      {
29          std::cout << "Failed to initialize GLAD" << std::
                endl;
30          return -1;
31      }
32      // 视口：渲染窗口大小
33      glViewport(0, 0, 800, 600);
34      return 0;
35  }
```

对于 GLFW 的初始化，我们调用 *glfwInit()* 函数。之后我们可以使用 *glfwWindowHint()* 函数来配置 GLFW。最后我们使用 *glfwCreateWindow()* 函数来创建一个窗口。

对于 GLAD 的初始化，我们调用 *gladLoadGLLoader()* 函数。然后，我们必须告诉 OpenGL 渲染窗口的尺寸大小，即**视口** (Viewport)，这样 OpenGL 才只能知道怎样根据窗口大小显示数据和坐标。我们可以通过调用 *glViewport* 函数来设置窗口的维度。

循环 我们希望程序在我们主动关闭它之前不断绘制图像并能够接受用户输入。因此，我们需要在程序中添加一个 while 循环，我们可以把它称之为渲染循环 (Render Loop)。在每一次循环中，我们检查 GLFW 是否被要求退出（*glfwWindowShouldClose()* 函数），如果是的话，我们就结束循环。我们加入了处理按键输入的函数 *processInput()*，目前只是检查了用户是否按下了 ESC 键，如果是的话，我们就把 *glfwWindowShouldClose()* 设置为 true。我们还需要在每次循环中调用 *glfwPollEvents()* 函数来处理所有事件。见代码清单3。

在该代码清单中，*glClearColor* 函数起了什么作用？实际上，它操作的是屏幕的颜色缓冲 (Color Buffer)，属于屏幕的帧缓冲 (Frame Buffer) 的一部分。

帧缓冲 帧缓冲是 OpenGL 中非常重要的一部分。在我们的例子中，我们使用了两个缓冲：颜色缓冲 (Color Buffer) 和深度缓冲 (Depth Buffer)。颜色缓冲是一个

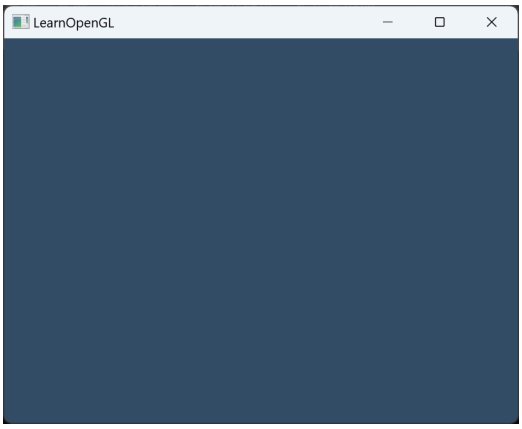

Figure 2: 创建窗口

大的数组，保存了每个像素的颜色值。深度缓冲是一个保存了深度值的缓冲，用于解决遮挡问题。我们目前所做的所有操作都是在默认帧缓冲中进行的。默认的帧缓冲是在创建窗口的时候生成和配置的（GLFW 帮我们做了这些）。在后续实验中，我们将使用自定义的帧缓冲来实现一些特殊的效果。

Listing 3: 渲染循环

```
    // 处理输入：如果按下ESC，退出
    void processInput(GLFWwindow *window)
    {
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    }
    // main 函数结尾添加循环
    while (!glfwWindowShouldClose(window))
    {
        // 输入
        processInput(window);
        // 清理颜色缓冲和深度缓冲
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // 渲染指令
        // ...
        // 检查并调用事件，交换缓冲
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
```

现在运行，可以看到一个 800×600 的黑色窗口。我们可以将窗口的背景颜色设置为其他颜色，只需要在 *glClearColor()* 函数中传入不同的颜色值即可。例如，*glClearColor(0.2f, 0.3f, 0.4f, 1.0f);*，如图2所示。

（四） 绘制三角形

现在，我们可以来绘制一个三角形了。为此，我们需要：

1. 准备好顶点数据，将顶点数据传递给 OpenGL（着色器）；
2. 编写着色器 (Shader);
3. 在渲染循环中，执行绘制指令。

在课程中，我们已经学习了光栅化渲染管线。回忆一下：

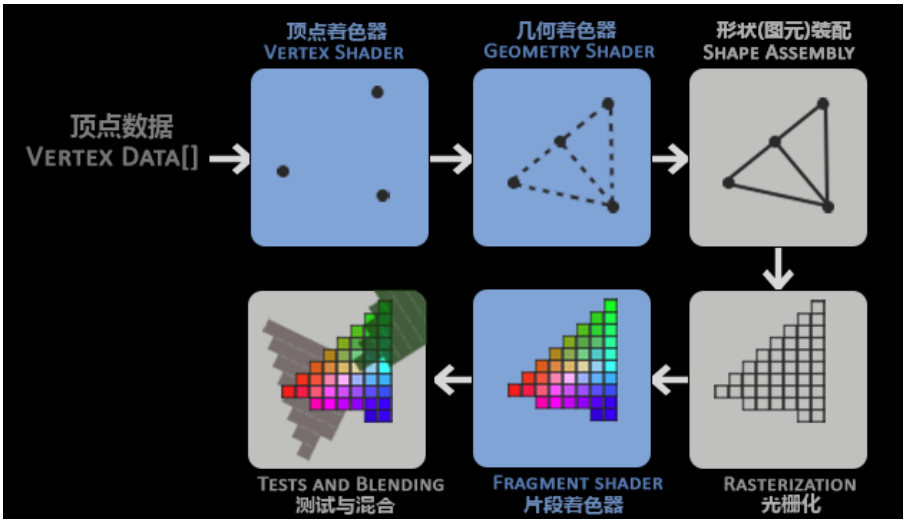

Figure 3: 管线，蓝色可以编程

我们需要编写顶点着色器将位于模型空间中的点转换为裁剪空间中的点，OpenGL 会通过透视除法 (除以 w 分量) 到 NDC (标准化设备坐标)，通过视口变换变到屏幕空间。我们还要用片段着色器为每个像素上色。

准备顶点数据 在本节中，我们不考虑上述变换，直接使用 NDC 的形式给出顶点坐标。在 OpenGL 标准中，这个空间被定义为 $[-1, 1]$ 的立方体。顶点数组形如4。

Listing 4: 顶点数据

```
1    float vertices[] = {
2        -0.5f, -0.5f, 0.0f, // 左下角
3        0.5f, -0.5f, 0.0f, // 右下角
4        0.0f, 0.5f, 0.0f // 顶部
5    };
```

我们通过顶点缓冲对象 (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。从 CPU 把数据发送到显卡相对较慢，所以只要可能我们都要尝试尽量一次性发送尽可能多的数据。当数据发送至显卡的内存中后，顶点着色器几乎能立即访问顶点，这是个非常快的过程。代码清单5主要分为三个步骤。

首先展示了如何创建一个 VBO，并把顶点数据复制到显存中。

Listing 5: 创建顶点缓冲对象

```
// main函数中，在初始化之后，while之前
//
// 1. 创建顶点缓冲对象，并获得其id，之后可以通过这个id来引用这个缓冲对象
//
unsigned int VBO;
glGenBuffers(1, &VBO);
// 在OpenGL上下文中绑定缓冲对象到GL_ARRAY_BUFFER目标上
// 此后，我们使用的任何（在GL_ARRAY_BUFFER目标上的）缓冲调用都会用来配置当前绑定的缓冲(VBO)
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// 这里我们就不用VBO了，直接对GL_ARRAY_BUFFER操作就行，因为已经绑定
// glBufferData函数会把之前定义的顶点数据复制到缓冲的内存中
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
//glBindBuffer(GL_ARRAY_BUFFER, 0); //解绑
//
// 2. 解释传入的缓冲数据
//
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

第二步，顶点着色器允许我们指定任何以顶点属性为形式的输入。这使其具有很强的灵活性的同时，它还的确意味着我们必须手动指定输入数据的哪一个部分对应顶点着色器的哪一个顶点属性。所以，我们必须在渲染前指定 OpenGL 该如何解释顶点数据。

glVertexAttribPointer 的参数：

1. 顶点属性的位置值 (Location)，这里是第一个属性
2. 顶点属性的大小，这里是 3
3. 数据的类型，这里是 GL_FLOAT
4. 是否希望数据被标准化，这里是 GL_FALSE
5. 步长，这里是 3 * sizeof(float)
6. 数据在缓冲中起始位置的偏移量，这里是 0

第三步，我们已经告诉 OpenGL 如何解释顶点数据。为了避免重复解释，我们把“数据”和“解释”保存为一个顶点数组对象 (Vertex Array Object, VAO)。在 OpenGL 上下文绑定 VAO 之后，再去绑定 VBO 和解释数据，就会自动保存在 VAO 中，之后我们只需要绑定 VAO 就可以了。

综上所述，这三步整体的代码如下6：

Listing 6: 创建顶点数组对象

```
    //
    // 3. 创建顶点数组对象
    //
    unsigned int VAO;
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);
    // 1. VBO ...
    // 2. glVertexAttribPointer ...
```

编写着色器 我们将着色器程序抽象为着色器类便于管理。新建 src/shader.h 和 src/shader.cpp 文件。这个封装的细节请阅读代码清单7。

Listing 7: 着色器类

```
//
// shader.h
//
#ifndef SHADER_H
#define SHADER_H
class Shader
{
public:
        // OpenGL ID
        unsigned int ID;
        // 构造器读取并构建着色器
        Shader(const char* vertexPath, const char*
           fragmentPath);
    // 使用着色器
    void use();
};
#endif
//
// shader.cpp
//

#include "shader.h"
#include <fstream>
#include <sstream>
#include <iostream>
Shader::Shader(const char* vertexPath, const char*
   fragmentPath)
{
        // 1. 从文件路径中获取顶点/片段着色器
        std::string vertexCode;
        std::string fragmentCode;
        std::ifstream vShaderFile;
        std::ifstream fShaderFile;
```

```
32          // 保证ifstream对象可以抛出异常:
33          vShaderFile.exceptions(std::ifstream::failbit | std
              ::ifstream::badbit);
34          fShaderFile.exceptions(std::ifstream::failbit | std
              ::ifstream::badbit);
35          try
36          {
37                  // 打开文件
38                  vShaderFile.open(vertexPath);
39                  fShaderFile.open(fragmentPath);
40                  std::stringstream vShaderStream,
                      fShaderStream;
41                  // 读取文件的缓冲内容到数据流中
42                  vShaderStream << vShaderFile.rdbuf();
43                  fShaderStream << fShaderFile.rdbuf();
44                  // 关闭文件处理器
45                  vShaderFile.close();
46                  fShaderFile.close();
47                  // 转换数据流到string
48                  vertexCode = vShaderStream.str();
49                  fragmentCode = fShaderStream.str();
50          }
51          catch (std::ifstream::failure e)
52          {
53                  std::cout << "ERROR::SHADER::
                      FILE_NOT_SUCCESFULLY_READ" << std::endl;
54          }
55          const char* vShaderCode = vertexCode.c_str();
56          const char* fShaderCode = fragmentCode.c_str();
57          // 2. 编译着色器
58          unsigned int vertex, fragment;
59          int success;
60          char infoLog[512];
61
62          // 顶点着色器
63          vertex = glCreateShader(GL_VERTEX_SHADER);
64          glShaderSource(vertex, 1, &vShaderCode, NULL);
65          glCompileShader(vertex);
66          // 打印编译错误（如果有的话）
67          glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
68          if (!success)
69          {
70                  glGetShaderInfoLog(vertex, 512, NULL, infoLog
                      );
71                  std::cout << "ERROR::SHADER::VERTEX::
                      COMPILATION_FAILED\n" << infoLog << std::
                      endl;
72          };
```

```
73
74          // 片段着色器也类似
75          fragment = glCreateShader(GL_FRAGMENT_SHADER);
76          glShaderSource(fragment, 1, &fShaderCode, NULL);
77          glCompileShader(fragment);
78          // 打印编译错误（如果有的话）
79          glGetShaderiv(fragment, GL_COMPILE_STATUS, &success
                );
80          if (!success)
81          {
82                glGetShaderInfoLog(fragment, 512, NULL,
                    infoLog);
83                std::cout << "ERROR::SHADER::FRAGMENT::
                    COMPILATION_FAILED\n" << infoLog << std::
                    endl;
84          };
85
86          // 着色器程序
87          ID = glCreateProgram();
88          glAttachShader(ID, vertex);
89          glAttachShader(ID, fragment);
90          glLinkProgram(ID);
91          // 打印链接错误（如果有的话）
92          glGetProgramiv(ID, GL_LINK_STATUS, &success);
93          if (!success)
94          {
95                glGetProgramInfoLog(ID, 512, NULL, infoLog);
96                std::cout << "ERROR::SHADER::PROGRAM::
                    LINKING_FAILED\n" << infoLog << std::endl;
97          }
98
99          // 删除着色器，它们已经链接到我们的程序中了，已经不再需要了
100         glDeleteShader(vertex);
101         glDeleteShader(fragment);
102   }
103   void Shader::use() {
104         glUseProgram(ID);
105   }
```

在本实验，我们使用两个简单的着色器。新建 res/expr1.vs 和 res/expr1.fs，在其中分别存放顶点着色器和片段着色器。

Listing 8: 顶点着色器

```
1     #version 330 core
2     layout (location = 0) in vec3 aPos;
3
4     void main()
5     {
6        gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
```

```
7     }
```

Listing 9: 片段着色器

```
1     #version 330 core
2     out vec4 FragColor;
3
4     void main()
5     {
6         FragColor = vec4(0.1f, 0.2f, 0.3f, 1.0f);
7     }
```

可以看到，GLSL 看起来很像 C 语言。每个着色器都起始于一个版本声明。OpenGL 3.3 以及和更高版本中，GLSL 版本号和 OpenGL 的版本是匹配的（比如说 GLSL 420 版本对应于 OpenGL 4.2）。我们同样明确表示我们会使用核心模式。

下一步，使用 in 关键字，在顶点着色器中声明所有的输入顶点属性 (Input Vertex Attribute)。现在我们只关心顶点位置，所以我们只需要一个顶点属性。由于每个顶点都有一个 3D 坐标，我们就创建一个 vec3 输入变量 aPos。该值来自于缓冲区的位置 0，这与我们在上边 glVertexAttribPointer 函数中设置的位置值相对应。

最后，为了设置顶点着色器的输出，我们必须把位置数据赋值给预定义的 gl_Position 变量，它是 vec4 类型的（齐次空间坐标）。在 main 函数的最后，我们将 gl_Position 设置的值会成为该顶点着色器的输出。因为在该程序中，输入和输入都在 NDC 空间下，因此第四个分量是 1.0。

对于片段着色器而言，只有一个输出：颜色值。我们把颜色值设置为一个 vec4 类型的变量（即 RGBA）FragColor。

绘制指令 首先，我们需要创建一个着色器对象，然后编译着色器。

```
1     // before int main()
2  # include "shader.h"
3     // in main()
4     // glEnableVertexAttribArray(0);
5     // 创建shader对象
6     Shader shader_triangle("res/expr1.vs", "res/expr1.fs")
          ;
7     // ...
8
9     // while(!glfwWindowShouldClose(window)) 中
10    // 渲染指令
11    shader_triangle.use();
12    // 绑定VAO
13    glBindVertexArray(VAO);
14    // 绘制三角形
15    glDrawArrays(GL_TRIANGLES, 0, 3);
16    // ...
```

我们最后得到了蓝色的三角形，如图4所示。

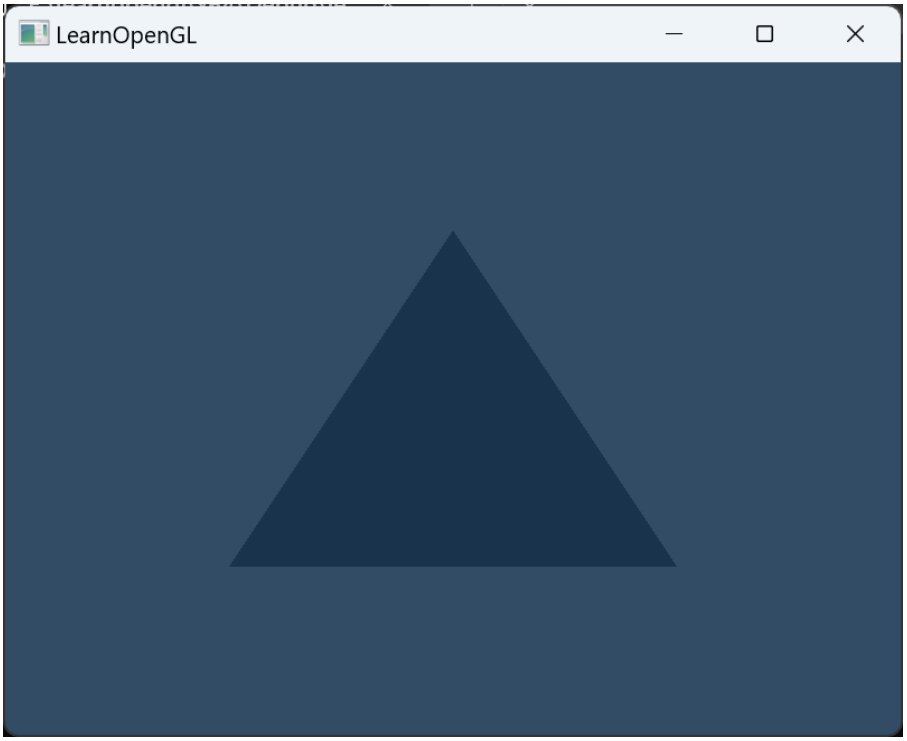

Figure 4: 绘制三角形

二、 实验 2: 金刚石的绘制

（一） 实验目的

- 了解坐标变换
- 了解使用纹理的方法
- 编写着色器，实现金刚石的绘制

（二） 摄像机和输入处理

为了更好地进行实验，我们引入摄像机类，用于处理摄像机的移动和旋转。首先，我们加入本次实验用到的库 glm 和 stb_image.h。这两个库都是只有头文件的。

1. 在GLM 下载处下载最新版本的 glm，将其中的 glm 文件夹复制到 include 文件夹下。
2. 在stb_image.h 下载处下载 stb_image.h，将其复制到 src 文件夹下。
3. 将 stb_image.h 添加到项目中（头文件-添加-现有项，或 src-添加-现有项）。

我们将摄像机抽象为摄像机类，其中包含摄像机的位置、方向、上方向、视野等信息。关于 Camera 类的实现，请参考代码清单10。

Listing 10: Camera.h

```
#ifndef CAMERA_H
#define CAMERA_H

#include <glad/glad.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <vector>

// Defines several possible options for camera movement. Used as abstraction to
   stay away from window-system specific input methods
enum Camera_Movement {
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT
};

// Default camera values
const float YAW = -90.0f;
const float PITCH = -15.0f;
```

```
21 const float SPEED = 2.5f;
22 const float SENSITIVITY = 0.1f;
23 const float ZOOM = 45.0f;
24
25
26 // An abstract camera class that processes input and calculates the
      corresponding Euler Angles, Vectors and Matrices for use in OpenGL
27 class Camera
28 {
29 public:
30     // camera Attributes
31     glm::vec3 Position;
32     glm::vec3 Front;
33     glm::vec3 Up;
34     glm::vec3 Right;
35     glm::vec3 WorldUp;
36     // euler Angles
37     float Yaw;
38     float Pitch;
39     // camera options
40     float MovementSpeed;
41     float MouseSensitivity;
42     float Zoom;
43
44     // constructor with vectors
45     Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f
         ), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), float
          yaw = YAW, float pitch = PITCH) : Front(glm::vec3
         (0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED),
         MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
46     {
47         Position = position;
48         WorldUp = up;
49         Yaw = yaw;
50         Pitch = pitch;
51         updateCameraVectors();
52     }
53     // constructor with scalar values
54     Camera(float posX, float posY, float posZ, float upX,
         float upY, float upZ, float yaw, float pitch) :
         Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(
         SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
55     {
56         Position = glm::vec3(posX, posY, posZ);
57         WorldUp = glm::vec3(upX, upY, upZ);
58         Yaw = yaw;
59         Pitch = pitch;
60         updateCameraVectors();
```

```
61      }
62
63      // returns the view matrix calculated using Euler Angles and the LookAt
            Matrix
64      glm::mat4 GetViewMatrix()
65      {
66          //位置，目标，上向量
67          return glm::lookAt(Position, Position + Front, Up);
68      }
69
70      // processes input received from any keyboard-like input system. Accepts
            input parameter in the form of camera defined ENUM (to abstract it from
            windowing systems)
71      void ProcessKeyboard(Camera_Movement direction, float
            deltaTime)
72      {
73          float velocity = MovementSpeed * deltaTime;
74          if (direction == FORWARD)
75              Position += Front * velocity;
76          if (direction == BACKWARD)
77              Position -= Front * velocity;
78          if (direction == LEFT)
79              Position -= Right * velocity;
80          if (direction == RIGHT)
81              Position += Right * velocity;
82      }
83
84      // processes input received from a mouse input system. Expects the offset
            value in both the x and y direction.
85      void ProcessMouseMovement(float xoffset, float yoffset
            , GLboolean constrainPitch = true)
86      {
87          xoffset *= MouseSensitivity;
88          yoffset *= MouseSensitivity;
89
90          Yaw += xoffset;
91          Pitch += yoffset;
92
93          // make sure that when pitch is out of bounds, screen doesn't get
                flipped
94          if (constrainPitch)
95          {
96              if (Pitch > 89.0f)
97                  Pitch = 89.0f;
98              if (Pitch < -89.0f)
99                  Pitch = -89.0f;
100         }
101
```

```
102         // update Front, Right and Up Vectors using the updated Euler angles
103         updateCameraVectors();
104     }
105 
106     // processes input received from a mouse scroll-wheel event. Only requires
        input on the vertical wheel-axis
107     void ProcessMouseScroll(float yoffset)
108     {
109     // yoffset = 1 / -1
110         Zoom -= (float)yoffset;
111         if (Zoom < 1.0f)
112             Zoom = 1.0f;
113         if (Zoom > 90.0f)
114             Zoom = 90.0f;
115     }
116 
117 private:
118     // calculates the front vector from the Camera's (updated) Euler Angles
119     void updateCameraVectors()
120     {
121         // calculate the new Front vector
122         glm::vec3 front;
123         front.x = cos(glm::radians(Yaw)) * cos(glm::radians
            (Pitch));
124         front.y = sin(glm::radians(Pitch));
125         front.z = sin(glm::radians(Yaw)) * cos(glm::radians
            (Pitch));
126         Front = glm::normalize(front);
127         // also re-calculate the Right and Up vector
128         Right = glm::normalize(glm::cross(Front, WorldUp));
              // normalize the vectors, because their length gets closer to 0 the
              more you look up or down which results in slower movement.
129         Up = glm::normalize(glm::cross(Right, Front));
130     }
131 };
132 #endif
```

摄像机类中，有几个我们需要注意的地方：

- 摄像机位置: Position，就是世界空间中一个指向摄像机位置的向量。
- 摄像机方向: Front，就是摄像机指向的方向。
- (见 updateCameraVectors) 为了定义摄像机空间，我们的方法是定义世界上方向（即 (0,1,0)），然后与摄像机的前向量叉乘得到右向量，再与前向量叉乘得到摄像机上向量。见图5。
- 摄像机的旋转角度 Yaw 和 Pitch，分别表示绕 y 轴和 x 轴的旋转角度。这是欧拉角表示法。

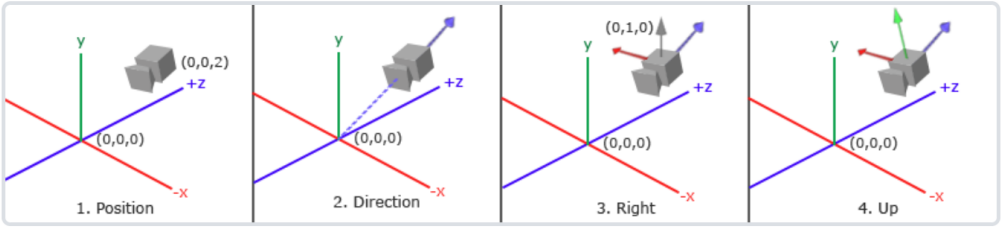

Figure 5: 摄像机坐标系

- lookAt 函数，用于计算摄像机的观察矩阵。

接下来，使用摄像机类

Listing 11: 加入摄像机类

```
    // learnopengl.cpp
#include "Camera.h"
    // SCR_WIDTH = ...
Camera camera(glm::vec3(0.0f, 1.0f, 0.0f));

float lastX = (float)SCR_WIDTH / 2.0;
float lastY = (float)SCR_HEIGHT / 2.0;
bool firstMouse = true;
float deltaTime = 0.0f; // 两帧的时间差
float lastFrame = 0.0f; // 上一帧的时间
    // 修改ProcessInput函数，加入移动摄像机的代码
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
}
    // 加入鼠标移动回调函数
void mouse_callback(GLFWwindow* window, double xposIn,
  double yposIn)
{
        float xpos = static_cast<float>(xposIn);
        float ypos = static_cast<float>(yposIn);

        if (firstMouse)
        {
              lastX = xpos;
```

```
            lastY = ypos;
            firstMouse = false;
        }

        float xoffset = xpos - lastX;
        float yoffset = lastY - ypos; // reversed since y-coordinates go from bottom to top

        lastX = xpos;
        lastY = ypos;

        camera.ProcessMouseMovement(xoffset, yoffset);
}
// glfw的鼠标滚轮回调函数
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
        camera.ProcessMouseScroll(static_cast<float>(yoffset));
}

// main() 函数中:
    // ...
    // glfwMakeContextCurrent(window);
    // 注册回调
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);   //关闭鼠标显示
```

（三） 导入纹理

OpenGL 纹理对象的创建和使用与前述 VBO 等对象是相似的。我们给出一个简单的纹理加载函数，用于加载纹理。

```
// 这个宏是需要的
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
// ...
unsigned int loadTexture(char const *path)
{
        unsigned int textureID;
        glGenTextures(1, &textureID);

        int width, height, nrComponents;
        unsigned char *data = stbi_load(path, &width, &height, &nrComponents, 0);
```

```
12          if (data)
13          {
14                  GLenum format;
15                  if (nrComponents == 1)
16                          format = GL_RED;
17                  else if (nrComponents == 3)
18                          format = GL_RGB;
19                  else if (nrComponents == 4)
20                          format = GL_RGBA;
21
22                  glBindTexture(GL_TEXTURE_2D, textureID);
23                  glTexImage2D(GL_TEXTURE_2D, 0, format, width,
                        height, 0, format, GL_UNSIGNED_BYTE, data)
                        ;
24                  glGenerateMipmap(GL_TEXTURE_2D);
25                  // gl_repeat在边缘与对面的边缘插值。为了避免边框，使用
26
27                  glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_WRAP_S, GL_REPEAT);
28                  glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_WRAP_T, GL_REPEAT);
29                  if (format == GL_RGBA) {
30                          glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
31                          glTexParameteri(GL_TEXTURE_2D,
                                GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
32                  }
33
34                  glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_MIPMAP_LINEAR);
35                  glTexParameteri(GL_TEXTURE_2D,
                        GL_TEXTURE_MAG_FILTER, GL_LINEAR);
36
37                  stbi_image_free(data);
38          }
39          else
40          {
41                  std::cout << "Texture failed to load at path:
                        " << path << std::endl;
42                  stbi_image_free(data);
43          }
44
45          return textureID;
46  }
```

在使用 stbi_load 函数加载图片文件后，数据被保存在 data 中。与之前相同，我们将 textureID 与 GL_TEXTURE_2D 绑定，然后使用 glTexImage2D 函数将数据

传递给 OpenGL 时，只需要操作 GL_TEXTURE_2D。

关于 Mipmap（多级纹理）请参考 ppt 的内容。

（四） 坐标变换

从一开始的模型空间到裁剪坐标空间 (gl_Position)，一般只有几步、

1. 乘以模型矩阵，将模型空间坐标转换到世界空间坐标。
2. 乘以观察矩阵，将世界空间坐标转换到观察空间坐标。
3. 乘以投影矩阵，将观察空间坐标转换到裁剪坐标。

其中，观察空间也就是摄像机空间，是以摄像机为原点的坐标系，这个是我们在之前通过摄像机类实现的。现在我们可以通过 *camera.GetViewMatrix()* 获得。

我们需要手工指定模型矩阵（模型在世界空间中的位置）。投影矩阵则与 FOV(视野) 有关，我们可以通过 glm::perspective 函数获得。接下来，我们需要将这些矩阵传递给着色器。在 shader.h 中，我们添加以下方法：

```
1     // 在 shader.h中
2  #include <glm/glm.hpp>
3  #include <string>
4     // class Shader
5     void setBool(const std::string &name, bool value)
        const;
6     void setInt(const std::string &name, int value) const;
7     void setFloat(const std::string &name, float value)
        const;
8     void setMat4(const std::string &name, glm::mat4 value)
         const;
9     void setVec3(const std::string &name, glm::vec3 value)
         const;
10    void setVec3(const std::string &name, float x, float y
        , float z) const;
11    void setVec2(const std::string& name, float x, float y
        ) const;
12    // 在shader.cpp中
13 #include <glm/gtc/type_ptr.hpp>
14 void Shader::setBool(const std::string& name, bool value)
      const
15 {
16       glUniform1i(glGetUniformLocation(ID, name.c_str()),
           (int)value);
17 }
18 void Shader::setInt(const std::string& name, int value)
     const
19 {
```

```
20        glUniform1i(glGetUniformLocation(ID, name.c_str()),
              value);
21  }
22  void Shader::setFloat(const std::string& name, float
      value) const
23  {
24        glUniform1f(glGetUniformLocation(ID, name.c_str()),
              value);
25  }
26  void Shader::setMat4(const std::string& name, glm::mat4
      value) const
27  {
28        glUniformMatrix4fv(glGetUniformLocation(ID, name.
            c_str()), 1, GL_FALSE, glm::value_ptr(value));
29  }
30  void Shader::setVec3(const std::string& name, glm::vec3
      value) const
31  {
32        glUniform3fv(glGetUniformLocation(ID, name.c_str())
            , 1, glm::value_ptr(value));
33  }
34  void Shader::setVec3(const std::string& name, float x,
      float y, float z) const
35  {
36        glUniform3fv(glGetUniformLocation(ID, name.c_str())
            , 1, glm::value_ptr(glm::vec3(x, y, z)));
37  }
38
39  void Shader::setVec2(const std::string& name, float x,
      float y) const
40  {
41        glUniform2fv(glGetUniformLocation(ID, name.c_str())
            , 1, glm::value_ptr(glm::vec2(x, y)));
42  }
```

这些函数是如何工作的？首先，我们要介绍 uniform 变量，这是一种在着色器中定义的变量，其定义如 “uniform vec3 lightColor;” 我们可以通过 glGetUniformLocation 函数 **通过变量名**获得这个变量的位置，然后通过 *glUniform3fv* 等函数将值传递给这个变量。

（五） 使用纹理

使用纹理很简单，我们只需要设置 uniform sampler2D 类型的变量，然后将纹理 ID 传递给这个变量即可。为了绘制金刚石，我们需要从定义顶点数据开始。此时，顶点数据需要增加一项：该顶点在纹理中的位置。纹理可以视为一个二维（UV 平面上的）图像，每个顶点都有一个对应的纹理坐标。这个坐标的范围是 [0,1]，(0,0) 是左下角，(1,1) 是右上角。

```
    // 顶点数据
    float diamondVertices[] = {
        // 顶点位置 // 纹理位置
        0.5f, 0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 0.5f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.5f, 1.0f, 1.0f,
        -0.5f, 0.0f, 0.0f, 0.0f, 0.0f,
        0.0f, -0.5f, 0.0f,1.0f, 0.0f,
        0.0f, 0.0f, -0.5f,1.0f, 1.0f,
};
```

这里，我们定义了一个金刚石的顶点数据，每个顶点有一个纹理坐标。但是，如果像上一节绘制三角形一样，我们会产生许多冗余的顶点数据。于是，我们使用索引缓冲对象，将顶点数据和索引数据分开。索引数据，也就是绘制三角形时的三个顶点的索引。

```
unsigned int diamondIndices[] = {
    1,0,2,
    3,1,2,
    4,3,2,
    0,4,2,
    4,0,5,
    0,1,5,
    1,3,5,
    3,4,5,
};
```

这里，我们定义了金刚石的索引数据。接下来，我们需要将这些数据传递给 OpenGL。我们使用的是 EBO（Element Buffer Object），也就是索引缓冲对象。与 VBO 类似，我们需要生成一个 EBO，然后绑定，然后传递，然后解绑等等。这些操作也可以记录到 VAO 中，以便之后的使用。我们的 VBO 也需要增加一个 *glVertexAttribPointer*，指定位置 1 的纹理坐标属性。考虑到遮挡关系，我们应该允许深度测试，即 *glEnable(GL_DEPTH_TEST)*。否则，后绘制的物体将遮挡先前绘制的物体，即使其深度在更后面的位置。

```
    glEnable(GL_DEPTH_TEST);
    // ...
    unsigned int VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(diamondVertices),
        diamondVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(
      diamondIndices), diamondIndices, GL_STATIC_DRAW);
```

```
12      // 位置属性
13      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
          sizeof(float), (void*)0);
14      glEnableVertexAttribArray(0);
15      // 纹理坐标属性
16      glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 *
          sizeof(float), (void*)(3 * sizeof(float)));
17      glEnableVertexAttribArray(1);
```

接下来，我们需要加载纹理，然后将纹理传递给着色器。在 res 文件夹中新建 expr2.vs 和 expr2.fs，**顶点着色器**代码：

Listing 12: expr2.vs

```
1  #version 330 core
2  layout (location = 0) in vec3 aPos;
3  layout (location = 1) in vec2 aTexCoords;
4
5  out vec2 TexCoords;
6
7  uniform mat4 model;
8  uniform mat4 view;
9  uniform mat4 projection;
10
11 void main()
12 {
13     TexCoords = aTexCoords;
14     gl_Position = projection * view * model * vec4(aPos,
         1.0);
15 }
```

与之前相比，我们增加了一个 layout(location = 1) in vec2 aTexCoords;，用于接收纹理坐标。我们还新建了 uniform 变量，model，view，projection，分别表示模型矩阵，观察矩阵，投影矩阵。

此外，我们还定义了一个 out vec2 TexCoords;，用于传递纹理坐标给片元着色器。**片元着色器**代码

Listing 13: expr2.fs

```
1      #version 330 core
2      out vec4 FragColor;
3
4      in vec2 TexCoords;
5
6      uniform sampler2D texture1;
7
8      void main()
9      {
10         vec4 texColor = texture(texture1, TexCoords);
11         FragColor = texColor;
```

```
12      }
```

与之前相比，我们增加了一个 uniform sampler2D 类型的变量 texture1，以及接受顶点着色器传递的纹理坐标 TexCoords。我们使用 texture 函数从纹理中获取纹理值。

接下来，我们需要在 main 函数中加载纹理，然后传递给着色器。我们在这里下载一个金刚石纹理，命名为 diamond.jpg，然后将其放在 res 文件夹下。接下来，我们需要在 main 函数中加载纹理，

```
1     Shader shader_diamond("res/expr2.vs", "res/expr2.fs");
2     unsigned int texture = loadTexture("res/diamond.jpg");
3     shader_diamond.use();
4     shader_diamond.setInt("texture1", 0);
```

在渲染循环中，我们执行绑定 Uniform 变量和绘制命令。

```
1     // while 中
2     // 更新帧时间，处理输入
3     {
4         float currentFrame = static_cast<float>(glfwGetTime
              ());
5         deltaTime = currentFrame - lastFrame;
6         lastFrame = currentFrame;
7
8         processInput(window);
9     }
10
11    shader_diamond.use();
12    shader_diamond.setMat4("model", glm::mat4(1.0f));
13    shader_diamond.setMat4("view", camera.GetViewMatrix())
          ;
14    shader_diamond.setMat4("projection", glm::perspective(
          glm::radians(camera.Zoom), (float)SCR_WIDTH / (float
          )SCR_HEIGHT, 0.1f, 100.0f));
15    glActiveTexture(GL_TEXTURE0);
16    glBindTexture(GL_TEXTURE_2D, texture);
17    glBindVertexArray(VAO);
18    glDrawElements(GL_TRIANGLES, 24, GL_UNSIGNED_INT, 0);
```

glDrawElements 命令的参数是绘制的图元类型，顶点数，索引数据的类型和偏移量。由于我们要绘制 24 个顶点，这里填 24。

运行程序，我们就看到一个金刚石，由两个四棱锥拼起来。有兴趣的同学可以绘制更精美的金刚石图像。

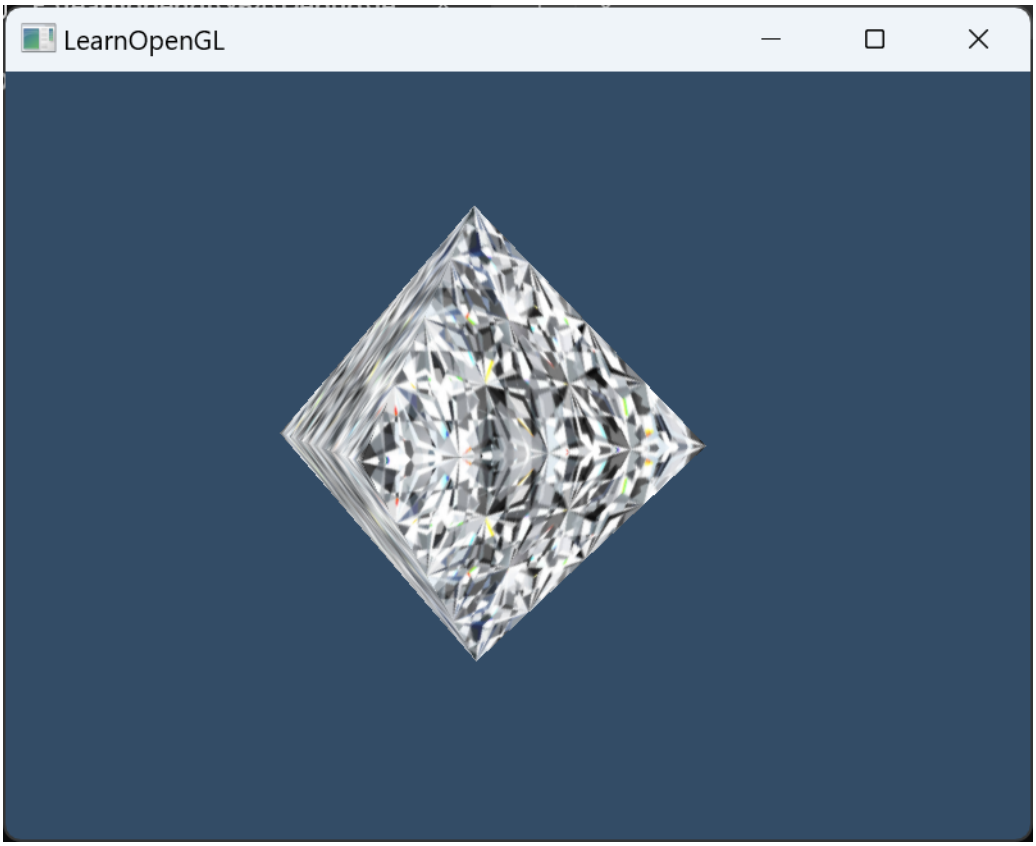

Figure 6: 金刚石

三、 实验 3: 光照

（一） 实验目的

- 了解冯氏光照模型的基本原理
- 实现基础的光照模型，包括环境光照、漫反射光照和镜面光照

（二） 基础光照

通过实验一及实验二，我们实现了绘制基本的立体图形的功能。为了实现更加真实的渲染效果，需要进一步为模型添加光照效果。现实世界的光照是极其复杂的，而且会受到诸多因素的影响，介绍和实现各种复杂的光照现象超出了本实验的范围。

但是，基于对光的物理特性的理解，也可以实现一些简单的光照模型，来对现实的光照效果进行近似，这些近似也能达到较好的视觉效果。

其中一个模型被称为**冯氏光照模型** (Phong Lighting Model)。冯氏光照模型的主要结构由 3 个分量组成：环境 (Ambient)、漫反射 (Diffuse) 和镜面 (Specular) 光照。下面这张图展示了这些光照分量看起来的样子：

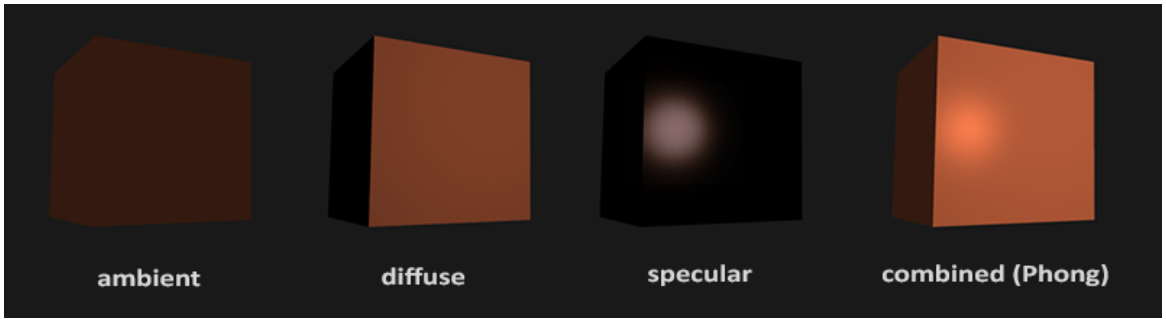

Figure 7: phong 光照模型

- 环境光照 (Ambient Lighting)：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，可以使用一个环境光照常量，它永远会给物体一些颜色
- 漫反射光照 (Diffuse Lighting)：模拟光源对物体的方向性影响 (Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮
- 镜面光照 (Specular Lighting)：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

（三） 环境光照

使用实验一二相同的方法，创建一个新的着色器来渲染模型。为模型添加环境光照的方法也非常简单，只需要将光源颜色乘以一个很小的常量环境因子，再乘以物体的颜色，然后将最终结果作为片元的颜色即可：

Listing 14: 环境光照

```
#version 330 core

out vec4 FragColor;

in vec2 TexCoords;
uniform sampler2D texture1;
uniform vec3 lightColor;

void main()
{
    vec3 objColor = texture(texture1, TexCoords).rgb;

    // 环境光照 = 环境光强度 * 光源颜色
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    vec3 result = ambient * objColor;
    FragColor = vec4(result, 1.0);
}
```

注意，光源颜色 lightColor 也需要通过绑定 uniform 变量进行传参，设为 glm::vec3(1.0f, 1.0f, 1.0f) 即可。物体的颜色是从纹理中获取的，所以需要传入一个纹理采样器。在这里可以下载到实验手册所用的黄色纹理。手册中加入的地板只是为了与背景色区分，并不需要同学们实现。

正确实现着色器，运行程序将得到下面类似的结果。物体会变得非常暗，但由于环境光照，仍然保留了一些颜色，而不至于全黑 (Figure 8)：

（四） 漫反射光照

环境光照本身不能提供最有趣的结果，但是漫反射光照就能开始对物体产生显著的视觉影响了。漫反射光照使物体上与光线方向越接近的片元能从光源处获得更多的亮度，如下图所示 (Figure 9)：

图左上方有一个光源，它所发出的光线落在物体的一个片元上。我们需要测量这个光线是以什么角度接触到这个片元的。如果光线垂直于物体表面，这束光对物体的光照将会最大。

为了测量光线和片元的角度，需要获得片元的**法向量** (Normal Vector)，它是垂直于片元表面的向量。使用法向量，光源与片元的角度就可以通过向量点乘直接计算。

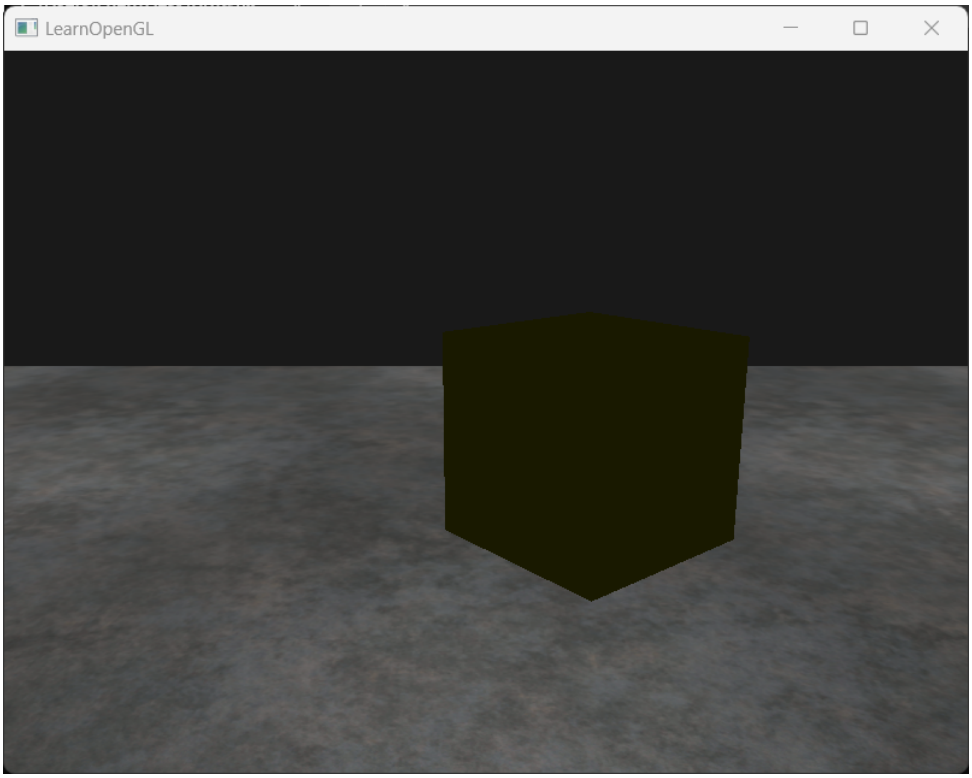

Figure 8: 环境光照

为立方体模型添加额外的法线信息，新的模型顶点如下。更新顶点信息后，还要修改绑定的顶点数组，正确的索引法线偏移和步长：

Listing 15: 带法线的立方体顶点信息

```
float cubeVertices[] = {
// positions    // texture Coords   // normals
-0.5f, -0.5f, -0.5f,  0.0f, 0.0f,    0.0f, 0.0f, -1.0f,
0.5f, -0.5f, -0.5f,  1.0f, 0.0f,    0.0f, 0.0f, -1.0f,
0.5f, 0.5f, -0.5f,  1.0f, 1.0f,    0.0f, 0.0f, -1.0f,
0.5f, 0.5f, -0.5f,  1.0f, 1.0f,    0.0f, 0.0f, -1.0f,
-0.5f, 0.5f, -0.5f,  0.0f, 1.0f,    0.0f, 0.0f, -1.0f,
-0.5f, -0.5f, -0.5f,  0.0f, 0.0f,    0.0f, 0.0f, -1.0f,

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

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

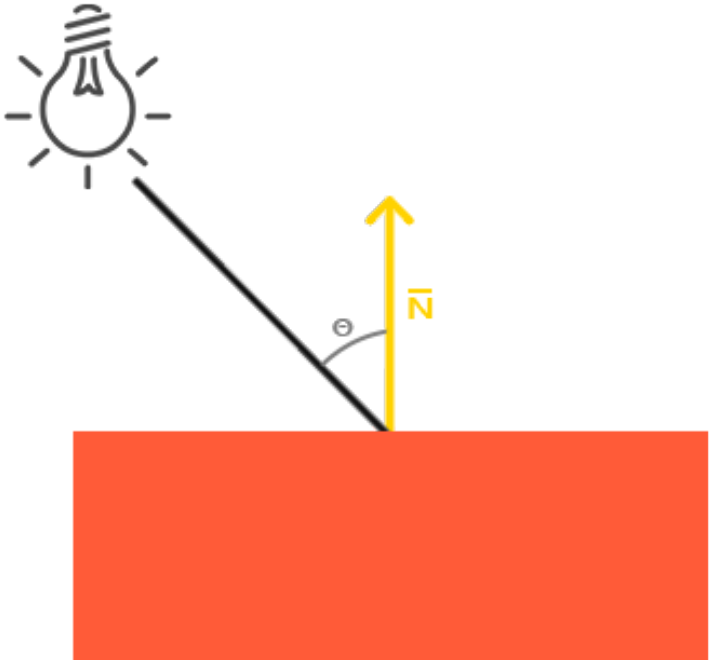

Figure 9: 漫反射模型

```
23
24 0.5f, 0.5f, 0.5f,  1.0f, 0.0f,   1.0f, 0.0f, 0.0f,
25 0.5f, 0.5f, -0.5f,  1.0f, 1.0f,   1.0f, 0.0f, 0.0f,
26 0.5f, -0.5f, -0.5f,  0.0f, 1.0f,   1.0f, 0.0f, 0.0f,
27 0.5f, -0.5f, -0.5f,  0.0f, 1.0f,   1.0f, 0.0f, 0.0f,
28 0.5f, -0.5f, 0.5f,  0.0f, 0.0f,   1.0f, 0.0f, 0.0f,
29 0.5f, 0.5f, 0.5f,  1.0f, 0.0f,   1.0f, 0.0f, 0.0f,
30
31 -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,   0.0f, -1.0f, 0.0f,
32 0.5f, -0.5f, -0.5f,  1.0f, 1.0f,   0.0f, -1.0f, 0.0f,
33 0.5f, -0.5f, 0.5f,  1.0f, 0.0f,   0.0f, -1.0f, 0.0f,
34 0.5f, -0.5f, 0.5f,  1.0f, 0.0f,   0.0f, -1.0f, 0.0f,
35 -0.5f, -0.5f, 0.5f,  0.0f, 0.0f,   0.0f, -1.0f, 0.0f,
36 -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,   0.0f, -1.0f, 0.0f,
37
38 -0.5f, 0.5f, -0.5f,  0.0f, 1.0f,   0.0f, 1.0f, 0.0f,
39 0.5f, 0.5f, -0.5f,  1.0f, 1.0f,   0.0f, 1.0f, 0.0f,
40 0.5f, 0.5f, 0.5f,  1.0f, 0.0f,   0.0f, 1.0f, 0.0f,
41 0.5f, 0.5f, 0.5f,  1.0f, 0.0f,   0.0f, 1.0f, 0.0f,
42 -0.5f, 0.5f, 0.5f,  0.0f, 0.0f,   0.0f, 1.0f, 0.0f,
43 -0.5f, 0.5f, -0.5f,  0.0f, 1.0f,   0.0f, 1.0f, 0.0f
44 };
45
46 ......
47
48 // 绑定顶点数组
49 glBufferData(GL_ARRAY_BUFFER, sizeof(cubeVertices), &
```

```
    cubeVertices, GL_STATIC_DRAW);
50  glEnableVertexAttribArray(0);
51  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 *
      sizeof(float), (void*)0);
52  glEnableVertexAttribArray(1);
53  glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 8 *
      sizeof(float), (void*)(3 * sizeof(float)));
54  glEnableVertexAttribArray(2);
55  glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 8 *
      sizeof(float), (void*)(5 * sizeof(float)));
```

完成对模型输入的修改后，就可以修改着色器来读取法线信息了。由于所有的光照计算均在片元着色器内进行，需要将法向量由顶点着色器传递到片元着色器。按照和输入纹理坐标相同的方式，声明顶点输入布局，并将读取的法线信息传递给片元着色器。另外，由于需要在世界空间计算光照，也需要给片元着色器传递片元的世界坐标。修改之后，完整的顶点着色器如下：

Listing 16: 顶点着色器，添加法线信息

```
1  #version 330 core
2  layout (location = 0) in vec3 aPos;
3  layout (location = 1) in vec2 aTexCoords;
4  layout (location = 2) in vec3 aNormal;
5
6  out vec2 TexCoords;
7  out vec3 Normal;
8  out vec3 FragPos;
9
10 uniform mat4 model;
11 uniform mat4 view;
12 uniform mat4 projection;
13
14 void main()
15 {
16     TexCoords = aTexCoords; // 纹理坐标
17     Normal = aNormal; // 法线
18     FragPos = vec3(model * vec4(aPos, 1.0)); // 片元坐标
19
20     gl_Position = projection * view * model * vec4(aPos,
         1.0);
21 }
```

所有需要的变量都设置完毕之后，就可以在片段着色器中添加漫反射光照计算了。

首先，需要计算光源和片段位置之间的方向向量。光源位置（世界空间坐标）按照和光源颜色相同的方法通过 uniform 输入即可，将其与片元的世界空间坐标作差，并归一化，就可以得到由片元指向光源的单位方向向量：

Listing 17: 片元着色器，光照向量计算

```
......
in vec2 TexCoords;
in vec3 Normal; // 顶点着色器输出的法线
in vec3 FragPos; // 顶点着色器输出的片元世界坐标

uniform vec3 lightColor; // 光源颜色
uniform vec3 lightPos; // 光源坐标

void main()
{
    ......
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    ......
```

接下来，对 norm 和 lightDir 向量进行点乘，计算光源对当前片段实际的漫反射影响。结果值再乘以光的颜色，得到漫反射分量。两个向量之间的角度越大，漫反射分量就会越小。

如果两个向量之间的角度大于 90 度，点乘的结果就会变成负数，这样会导致漫反射分量变为负数。为此，使用 max 函数返回两个参数之间较大的参数，从而保证漫反射分量不会变成负数（当夹角大于 90 度时，其实就意味着光照方向在片元的“背面”，那么光照就无法照亮该片元）。

在计算完成之后，再将得到的漫反射光照和环境光照合并即可：

Listing 18: 片元着色器，漫反射计算

```
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;

vec3 result = (ambient + diffuse) * objectColor;
FragColor = vec4(result, 1.0);
```

运行程序，将得到类似下面的结果 (Figure 10)：

如果代码正确，漫反射光照就顺利实现了。但在最后还有一个问题，片段着色器里的计算都是在世界空间坐标中进行的，而目前传递的法线是物体空间下的。如果模型进行了伸缩或者旋转变换，法线就会出错。如何修正这个问题呢？

法向量只是一个方向向量，不能表达空间中的特定位置，位移不应该影响法向量。对于法向量，应该只对它实施缩放和旋转变换。因此，如果把法向量乘以一个模型矩阵，需要要从矩阵中移除位移部分，只选用模型矩阵左上角 3×3 的矩阵。

另外，如果模型矩阵执行了不等比缩放，顶点的改变会导致法向量不再垂直于表面了。下面的图展示了应用了不等比缩放的模型矩阵对法向量的影响 (Figure 11)：

为了避免这类情况的发生，实际处理法线变换时使用的是一个特殊的**法线矩阵** (Normal Matrix)。法线矩阵被定义为“模型矩阵左上角 3x3 部分的逆矩阵的转置矩阵”（关于这个矩阵的数学推导，同学们可自行查阅资料）。

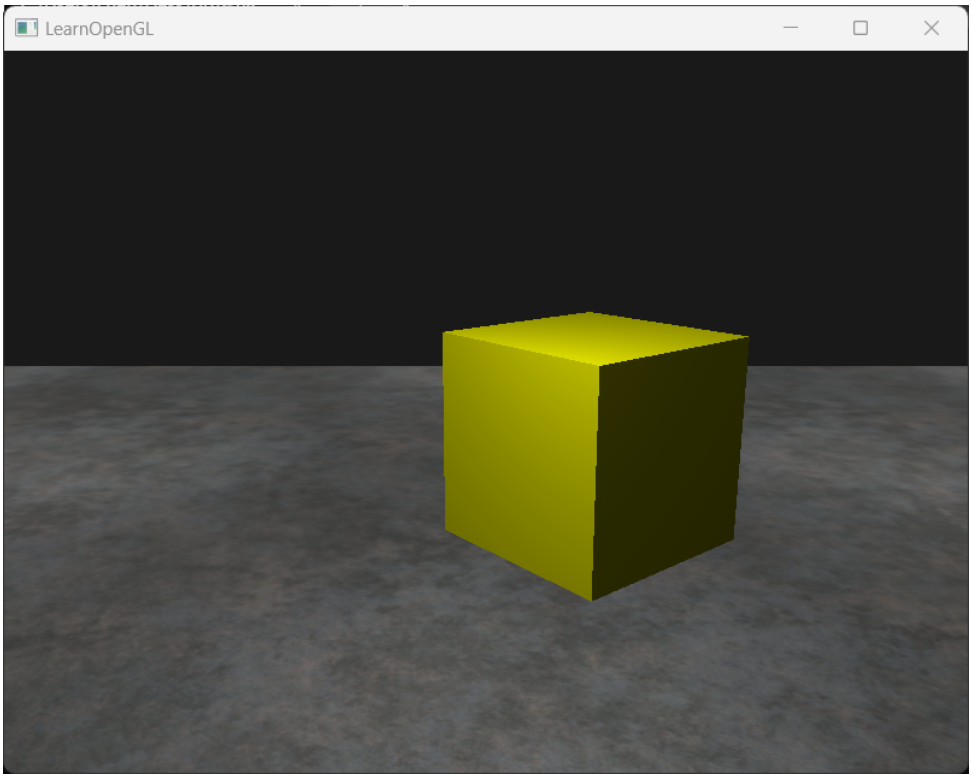

Figure 10: 漫反射光照

在顶点着色器中，可以使用 inverse 和 transpose 函数自己生成这个法线矩阵，再将矩阵转换为 mat3，移除位移，最终和法线计算，即可得到正确的世界空间法线：

Listing 19: 顶点着色器，法线计算

```
1 Normal = mat3(transpose(inverse(model))) * aNormal;
```

（五） 镜面光照

冯氏光照的最后一部分是镜面光照。和漫反射光照一样，镜面光照也由光的方向向量和物体的法向量决定，但是它也取决于观察方向。镜面光照决定于表面的反射特性。如果把物体表面设想为一面镜子，那么镜面光照最强的地方就是看到表面上反射光的地方。镜面光照的示意图如下 (Figure 12)：

我们通过根据法向量翻折入射光的方向来计算反射向量。然后计算反射向量与观察方向的角度差，它们之间夹角越小，镜面光的作用就越大。由此产生的效果就是，在看向在入射光在表面的反射方向时，会看到一点高光。

观察向量是计算镜面光照时需要的一个额外变量，可以传入相机的世界空间坐标，再与片元坐标作差得到。之后计算出镜面光照强度，用它乘以光源的颜色，并将它与环境光照和漫反射光照部分加和即可。

在计算镜面反射时，首先需要定义一个镜面强度 (Specular Intensity) 变量（也就是最大的镜面反射亮度），给镜面高光一个中等亮度颜色，让它不要产生过度的影响。之后，使用传入的相机坐标计算视线向量，再最后完成镜面高光的计算。

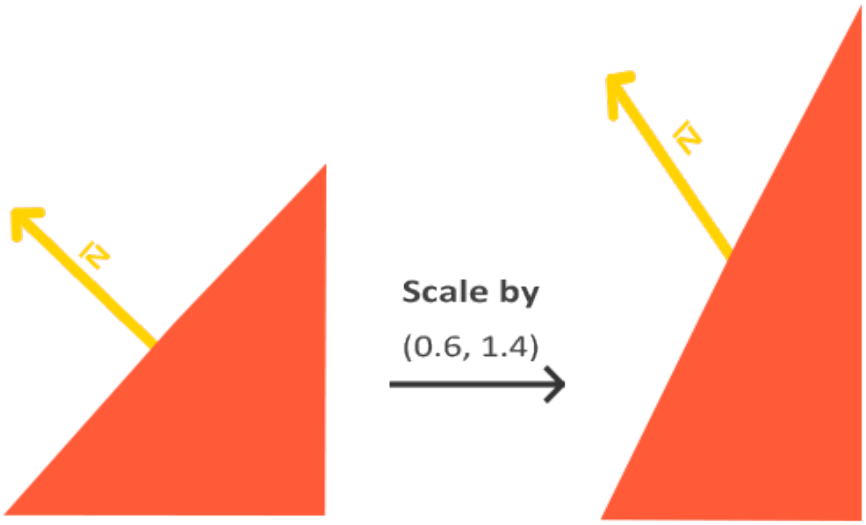

Figure 11: 缩放变换导致的法线失真

计算的相关代码如下：

Listing 20: 片元着色器，镜面反射计算

```
1  ......
2  uniform vec3 viewPos; // 传入相机的世界空间坐标
3
4  void main()
5  {
6      ......
7      float specularStrength = 0.5; // 镜面强度
8
9      vec3 viewDir = normalize(viewPos - FragPos); // 归一化的视线
           向量，由片元指向相机
10     vec3 reflectDir = reflect(-lightDir, norm); // 反射向量
11
12     float spec = pow(max(dot(viewDir, reflectDir), 0.0),
           32);
13     vec3 specular = specularStrength * spec * lightColor;
14     ......
```

计算镜面反射时，先计算视线方向与反射方向的点乘（保证非负），然后取它的 32 次幂。这个 32 是高光的反光度 (Shininess)。一个物体的反光度越高，反射光的能力越强，散射得越少，高光点就会越小。

不同反光度的视觉效果如下所示 (Figure 13)：

最后，将镜面光照也加入最终的结果，即可得到完整的冯氏光照。调整观察视角，能够得到如下的镜面反射效果 (Figure 14)：

cpp 部分的主要功能代码如下所示：

Listing 21: cpp 部分，着色器创建、绑定、传参及绘制

```
1  ......
```

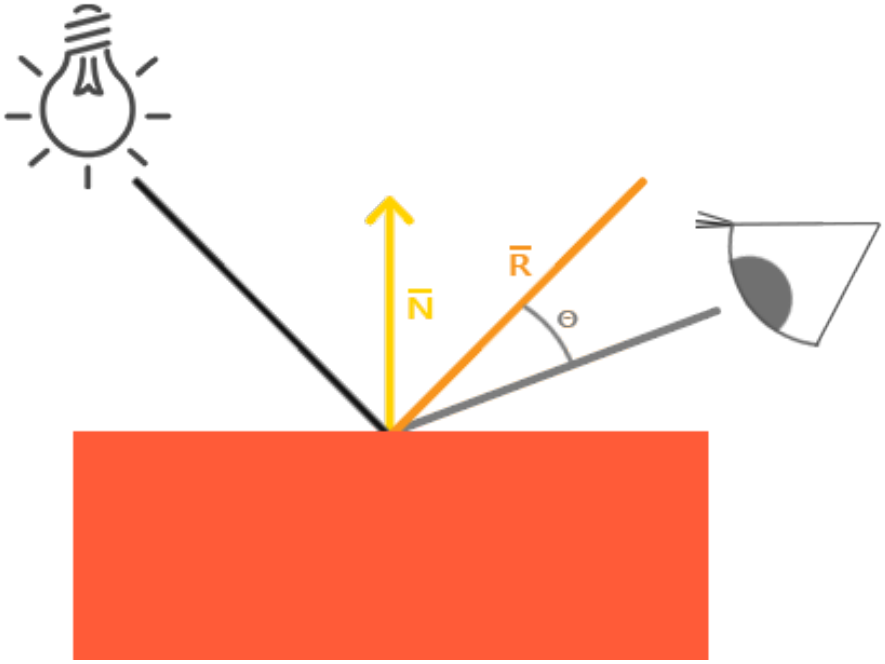

Figure 12: 镜面反射模型

```
2  Shader phongShader("./res/shader/phong.vert", "./res/
     shader/phong.frag");   // 创建着色器程序
3  ......
4
5  glm::vec3 lightColor = glm::vec3(1.0f, 1.0f, 1.0f);
6  glm::vec3 lightPos = glm::vec3(-0.1f, 1.5f, 0.3f);
7
8  glm::mat4 model = glm::translate(glm::mat4(1.0f), glm::
     vec3(0.5f, 0.0f, -1.0f));
9
10 phongShader.use();
11 phongShader.setMat4("model", model);
12 phongShader.setMat4("view", view);
13 phongShader.setMat4("projection", projection);
14
15 phongShader.setVec3("lightColor", lightColor);
16 phongShader.setVec3("lightPos", lightPos);
17 phongShader.setVec3("viewPos", camera.Position);
18
19 GLCall(glBindVertexArray(cubeVAO));
20 GLCall(glActiveTexture(GL_TEXTURE0));
21 GLCall(glBindTexture(GL_TEXTURE_2D, cubeTexture));
22 GLCall(glDrawArrays(GL_TRIANGLES, 0, 36));
```

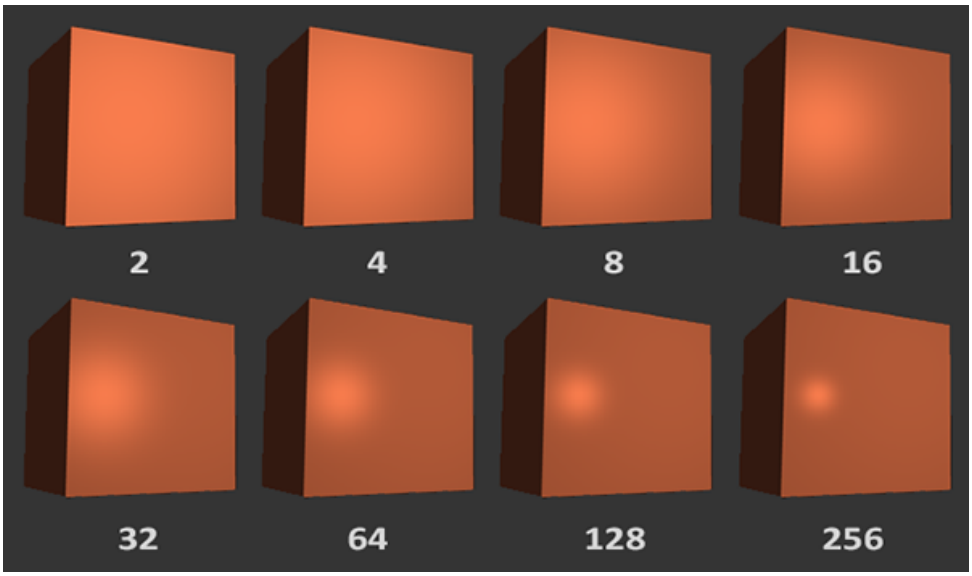

Figure 13: 不同反光度下的高光效果

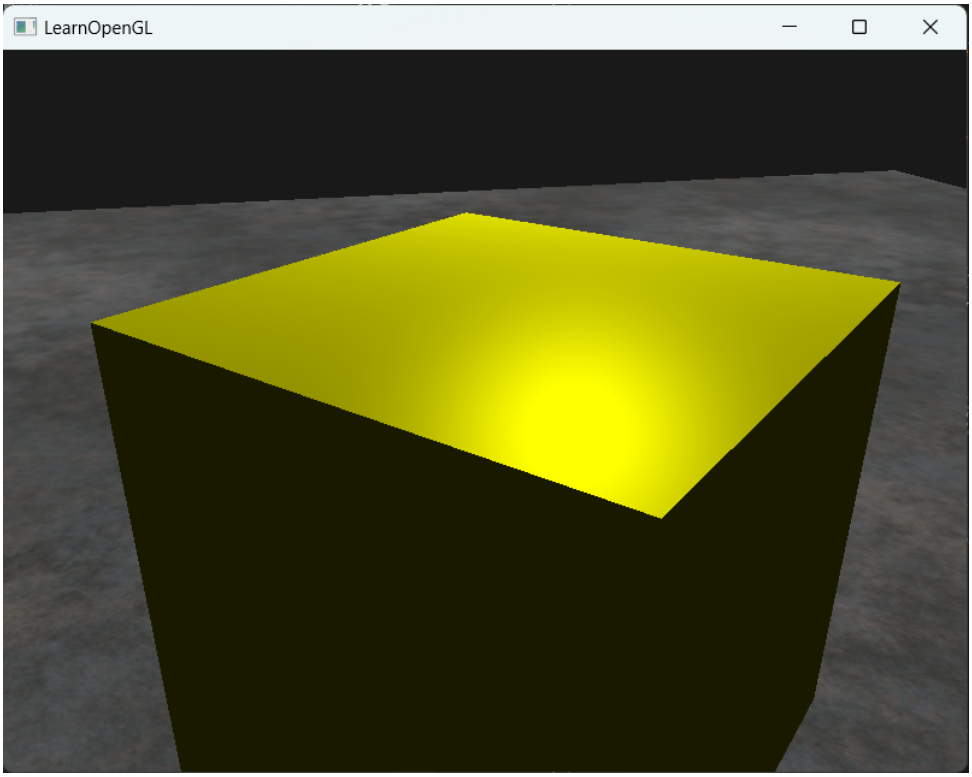

Figure 14: 镜面反射光照

四、 实验 4: 圆的绘制

（一） 实验目的

- 了解帧缓冲的概念
- 使用着色器绘制圆形

（二） 帧缓冲

通过前面的实验，我们已经能够顺利的完成各种三维物体的渲染了。但是，到目前为止，我们目前所做的所有操作都是在默认帧缓冲的渲染缓冲上进行的，默认的帧缓冲是在创建建窗口的时候，由 GLFW 生成和配置的。

用于写入颜色值的颜色缓冲、用于写入深度信息的深度缓冲和允许我们根据一些条件丢弃特定片段的模板缓冲，这些缓冲结合起来叫做**帧缓冲** (Framebuffer)，它被储存在内存中。OpenGL 允许我们定义我们自己的帧缓冲，也就是说我们能够定义我们自己的颜色缓冲，甚至是深度缓冲和模板缓冲。在能够自行实现帧缓冲后，就能够有更多方式来进行渲染。

本次实验，将首先介绍使用铺屏四边形绘制的方法，然后使用自行创建的帧缓冲来实现绘制过程。

（三） 使用铺屏四边形绘制圆

在渲染三维物体时，通常需要进行 MVP 的矩阵变换，将处于物体空间的模型顶点依次转移到世界空间，相机空间，裁剪空间，在此之后进行透视除法，转换到 NDC 坐标空间，实际渲染到屏幕。但是并非所有渲染都需要这个过程。对于渲染平面图形，直接为顶点指定坐标即可（就像绘制的第一个三角形那样）。

在真实的渲染程序中，为了实现各类屏幕空间效果，常常需要使用铺屏四边形来重新绘制整个场景。一个铺屏四边形就是顶点占据窗口四角的矩形，在线框模式下的绘制效果如下 (Figure 15)：

为了绘制这个四边形，我们将会新创建一套简单的着色器。着色器中将不会包含矩阵变换，因为输入时提供的是标准化设备坐标的顶点坐标，所以可以直接将它们设定为顶点着色器的输出。顶点着色器是这样的：

Listing 22: 顶点着色器，绘制铺屏四边形

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoords;
out vec2 TexCoords;

void main()
{
    gl_Position = vec4(position, 1.0f);
    TexCoords = texCoords;
```

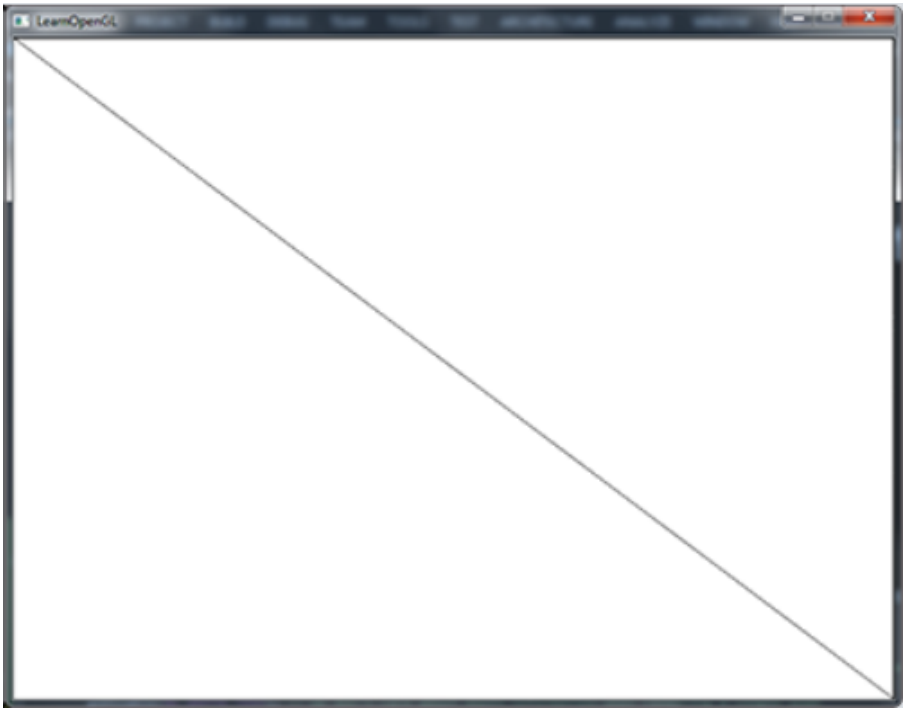

Figure 15: 线框模式绘制的铺屏四边形

```
}
```

在片元着色中，先简单的使用 UV 坐标作为颜色进行输出：

Listing 23: 片元着色器，输出 UV

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

void main()
{
    FragColor = vec4(TexCoords.xy, 0.0f, 1.0f);
}
```

在 cpp 程序中，完成模型及着色器创建，执行绘制指令的代码如下：

Listing 24: cpp 部分，模型创建及绘制

```
......
float quadVertices[] = { // 创建模型顶点VBO,VAO
        // positions // texture Coords
        -1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
        -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
         1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
         1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
};

unsigned int quadVAO, quadVBO;
```

```
11 glGenVertexArrays(1, &quadVAO);
12 glGenBuffers(1, &quadVBO);
13 glBindVertexArray(quadVAO);
14 glBindBuffer(GL_ARRAY_BUFFER, quadVBO);
15 glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), &
     quadVertices, GL_STATIC_DRAW);
16 glEnableVertexAttribArray(0);
17 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
     sizeof(float), (void*)0);
18 glEnableVertexAttribArray(1);
19 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 *
     sizeof(float), (void*)(3 * sizeof(float)));
20 glBindVertexArray(0);
21 ......
22 Shader circleShader("./res/shader/circle.vert", "./res/
     shader/circle.frag"); // 创建着色器程序
23 ......
24 circleShader.use(); // 绑定着色器与顶点，绘制
25 glBindVertexArray(quadVAO);
26 glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
27 glBindVertexArray(0);
```

顺利实现后，执行程序，应该得到以下的绘制效果：(Figure 16)：

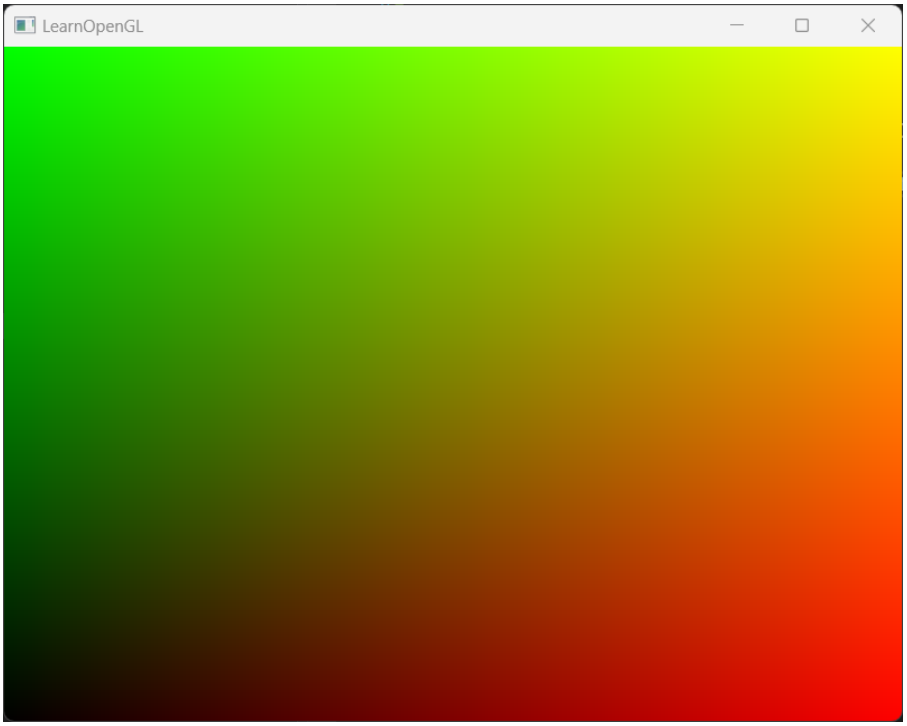

Figure 16: 铺屏四边形，输出颜色为 UV

在顺利实现绘制输出颜色为 UV 的铺屏四边形后，可以实现简单的几何绘制效果。以绘制圆形为例，将 uv 坐标转换到 [-1,1] 区间，那么一个以屏幕中心为圆心的圆环就可以使用点到屏幕中心点的距离函数表示：

Listing 25: 片元着色器，圆的距离函数表示

```
float step(float distance)
{
	if(distance < 0.99)    return 0; // [0.99, 1]为圆环宽度的绘制范围
	else if (distance <= 1)  return 1;

	return 0;
}
```

修改之前的片元着色器，在转换 UV 坐标之后，调用距离函数判断该片元是否处于圆环上，并进行相应的着色绘制，代码如下：

Listing 26: 片元着色器，绘制圆环

```
......
uniform float w_div_h;

void main()
{
	vec2 uv = TexCoords * 2.0 - 1.0; // 转换坐标，从UV的[0, 1]区间到[-1, 1]区间

	uv.x *= w_div_h; // 根据屏幕的宽高比重新计算x坐标

	float distance = sqrt(dot(uv, uv));

	vec3 color = vec3(step(distance)); // 调用距离函数绘制
	FragColor = vec4(color, 1.0);
}
```

注意，由于屏幕的宽高比问题，直接使用映射的 UV 坐标进行绘制得到的将会是椭圆形，所以需要传入屏幕宽高比以重新映射坐标比例。这里使用了 uniform 变量进行传参。

顺利执行上述着色器代码，得到的绘制效果如图 (Figure 17)：

（四） 创建帧缓冲

在了解了铺屏四边形的绘制方法后，我们就可以自行创建帧缓冲，并完成绘制过程了。我们将会将场景渲染到一个附加到帧缓冲对象上的颜色纹理中，之后将在一个横跨整个屏幕的四边形上绘制这个纹理。这样视觉输出和没使用帧缓冲时是完全一样的。

首先要创建一个帧缓冲对象，并绑定它，这些都很直观：

Listing 27: 创建帧缓冲对象

```
unsigned int framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

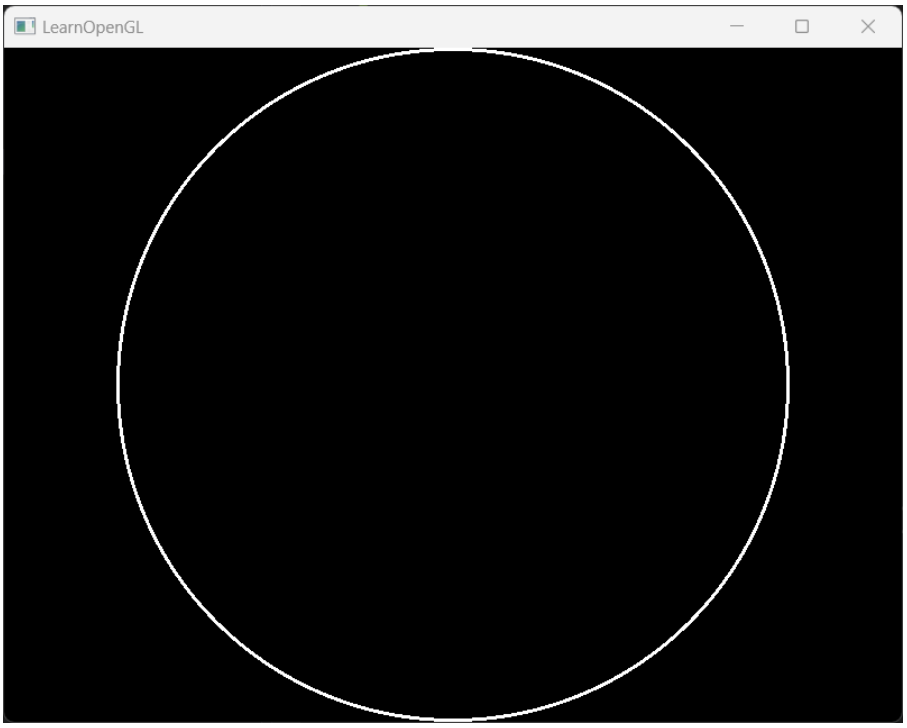

Figure 17: 铺屏四边形，绘制圆形

接下来需要创建一个纹理图像，将它作为一个颜色附件附加到帧缓冲上。我们将纹理的维度设置为窗口的宽度和高度，并且不初始化它的数据：

Listing 28: 创建颜色纹理并绑定

```
// 生成纹理
unsigned int texColorBuffer;
glGenTextures(1, &texColorBuffer);
glBindTexture(GL_TEXTURE_2D, texColorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0,
   GL_RGB, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
   GL_LINEAR );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
   GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);

// 将它附加到当前绑定的帧缓冲对象
glFramebufferTexture2D(GL_FRAMEBUFFER,
   GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texColorBuffer, 0)
   ;
```

如果需要进行深度或者模板测试，还需要添加一个深度附件到帧缓冲中。由于暂时不需要对深度缓冲进行采样，可以使用 opengl 的渲染缓冲对象，将它创建为一个深度和模板附件渲染缓冲对象。并将内部格式设置为 GL_DEPTH24_STENCIL8：

Listing 29: 创建深度模板缓冲并绑定

```
// 生成缓冲
```

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER,
   GL_DEPTH24_STENCIL8, 800, 600);
glBindRenderbuffer(GL_RENDERBUFFER, 0);

// 将它附加到当前绑定的帧缓冲对象
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
   GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

当所有缓冲都绑定完成后，检查帧缓冲的完整性，顺利执行后，就可以取消帧缓冲的绑定，帧缓冲的创建过程就完成了：

Listing 30: 检查帧缓冲完整性，取消帧缓冲的绑定

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
   GL_FRAMEBUFFER_COMPLETE)
    std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not
      complete!" << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

（五） 使用帧缓冲绘制场景

在完成帧缓冲的创建后，只需要绑定这个帧缓冲对象，让渲染到帧缓冲的缓冲中而不是默认的帧缓冲中。之后的渲染指令将会影响当前绑定的帧缓冲。所有的深度和模板操作都会从当前绑定的帧缓冲的深度和模板附件中（如果有的话）读取。如果在创建帧缓冲时忽略了深度缓冲，那么所有的深度测试操作将不再工作，因为当前绑定的帧缓冲中不存在深度缓冲。

使用帧缓冲绘制场景时，需要采取以下的步骤：

- 将新的帧缓冲绑定为激活的帧缓冲，和往常一样渲染场景
- 绑定默认的帧缓冲
- 绘制一个横铺屏四边形，将帧缓冲的颜色缓冲作为它的纹理。

绘制铺屏四边形将会使用和第一节中基本相同的着色器代码，不同之处在于，需要将创建的帧缓冲的颜色纹理作为着色器输入，并且采样和输出。顶点着色器和片元着色器代码如下：

Listing 31: 顶点着色器，绘制铺屏的帧缓冲纹理

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

```

```
7  void main()
8  {
9      gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
10     TexCoords = aTexCoords;
11 }
```

Listing 32: 片元着色器，绘制铺屏的帧缓冲纹理

```
1  #version 330 core
2  out vec4 FragColor;
3
4  in vec2 TexCoords;
5
6  uniform sampler2D screenTexture;
7
8  void main()
9  {
10     FragColor = texture(screenTexture, TexCoords);
11 }
```

准备好着色器程序后，就可以在 cpp 程序中补齐代码进行绘制了。先绑定创建好的帧缓冲，在其中进行绘制，再绑定默认帧缓冲，将创建的帧缓冲的颜色纹理作为着色器输入，绘制铺屏四边形，采样颜色纹理并输出：

Listing 33: cpp 部分，使用帧缓冲的绘制流程

```
1  // 绑定创建的帧缓冲，并进行场景绘制
2  glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
3  glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
4  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 我们
       现在不使用模板缓冲
5  glEnable(GL_DEPTH_TEST);
6
7  ......
8  // 在此处完成场景绘制
9  ......
10
11 // 绑定默认帧缓冲，将场景帧缓冲的颜色纹理作为输入，采样和渲染
12 glBindFramebuffer(GL_FRAMEBUFFER, 0);
13 glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
14 glClear(GL_COLOR_BUFFER_BIT);
15
16 quadShader.use();
17 glBindTexture(GL_TEXTURE_2D, texColorBuffer);
18 glDisable(GL_DEPTH_TEST); // 禁用深度测试
19 glBindVertexArray(quadVAO);
20 glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

绘制过程中还有一些细节需要注意。

第一，由于每个帧缓冲都有它自己一套纹理和内存，所以需要分别设置合适的帧缓冲指令，并调用 glClear 来完成帧缓冲清空。

第二，当绘制铺屏四边形时，并不关系场景的深度信息，此时需要禁用深度测试。在绘制普通场景的时候，也需要重新启用深度测试。

如果正确实现了上述过程，执行程序，将得到与未使用帧缓冲时完全相同的渲染结果。其区别在于，渲染的场景在最后一步时能够以纹理的方式被整体访问，在此基础上就可以实现各类丰富的后处理效果。

（一） debug 方法

在 OpenGL 实验中，我们经常会遇到一些问题，比如说图形显示不正确，或者只显示黑屏等等。这时候我们就需要使用一些 debug 方法来帮助我们找到问题所在。OpenGL 提供了一些 debug 方法，比如说使用 `glGetError` 函数来获取 OpenGL 的错误码。下面的代码清单展示了如何使用：

```
// 使用sstream来将多个参数转换为字符串
#include <sstream>
#define LOG GL_PRINT
template <typename ... Types>
void GL_PRINT(const Types&... args)
{
      std::ostringstream ss;
      std::initializer_list <int> { ([&args, &ss] {ss << (args) << " "; }(), 0)...};
      std::cout<<ss.str()<<std::endl;
}
inline static void GLClearError() { while (glGetError() != GL_NO_ERROR); }

static bool GLLogCall(const char* function, const char* file, const int line)
{
      while (GLenum error = glGetError())
      {
            LOG("[OpenGL] Error code:", error, "in function:", function, "at file:", file, "in line:", line);
            return false;
      }
      return true;
}
#define GLCall(x) GLClearError();\
                  x;\
                  GLLogCall(#x, __FILE__, __LINE__);
```

我们用 GLCall 宏来包装 OpenGL 的函数调用，这样就可以在每次调用 OpenGL 函数的时候检查是否有错误发生。例如，我们可以这样调用 OpenGL 函数：*GLCall(glGenVertexArrays(1, &VAO));*。

如果想用更高级的 debug 方法，可以使用一些工具，比如说 **RenderDoc**，**Nsight** 等等。这些工具可以帮助我们更方便地查找 OpenGL 的问题。

使用文本替换以快速的在 gl 函数外包装上 GLCall 函数。在替换界面中

```
    // 选择使用：正则表达式，在上方框中输入
    [\t ](gl[^fmca]+\(.+\));
    // 替换为
    \tGLCall($1);
```