计算机图形学期末大作业-小组报告

组员1 22365043 江颢怡

组员2 22302075邱丽梦

题目：opengl 搭建广州塔城市街景

一.实验概述

本实验旨在利用 OpenGL 技术构建一个具有广州特色的城市街景，其中广州塔作为标志性建筑成为场景的核心元素。通过整合多种opengl技术，模拟真实世界中的光照、纹理、视角切换和用户交互等效果，为用户带来身临其境的感受。

```
使用的是vs2022版本 直接release模式下运行main函数即可
相关按键：
ESC：退出程序。
W/S/A/D：控制摄像机移动。
上下左右方向键：控制物体在 X 和 Z 轴方向上的平移。
N/M：控制物体的缩放。
J/K：控制物体的旋转。
```

二. 技术路线与方案

本项目采用GLFW+GLAD进行OpenGL开发，渲染一个有广州塔的广州3D城市场景。

1. **物体绘制** 利用OpenGL的 Assimp库加载虚拟场景中的三维物体建筑，构建场景中的建筑物和广州塔。
2. **纹理贴图** 加载并应用材质纹理，确保物体表面效果真实自然。通过体素贴图将真实的场景图片赋予物体纹理，使得场景中的建筑和环境更加贴近实际。天空盒和地面也将使用真实场景的图片作为背景，提升沉浸感。
3. **光照与阴影效果** 使用顶点着色器和片元着色器着色 实现了三种光照**（点光源、平行光源、手电筒光源）和阴影**效果，以增强物体的立体感和场景的真实感。动态光照和阴影的交互将使得场景的氛围更加生动。
4. **视角切换与场景漫游** 实现多种视角切换功能，允许用户通过键盘或鼠标操作来进行场景漫游。视角包括：拉远/拉近，左右上下旋转视角，以及第一人称行走视角，进一步增强用户的互动体验。
5. **用户交互操作** 通过鼠标和键盘响应用户的交互操作，实现用户对场景中建筑物的控制功能（如移动、旋转、缩放）。
6. **背景音乐的添加** 为了提升3D体验，项目将配上**天空之城的轻音乐**作为背景音乐，以增强玩家在虚拟场景中的沉浸感和探索感。
7. **广州塔的模型旋转实现**：为了使广州塔这个模型更具真实感 我们还为这个模型单独提供了左右旋转功能

使用的相关技术和依赖的库

```
1. OpenGL：一个跨平台的API，用于渲染2D和3D矢量图形。
2. GLFW：一个用于创建窗口、处理输入和其他窗口相关任务的库。
```

3. GLAD：一个用于加载OpenGL函数指针的库。
4. GLM：一个基于OpenGL着色语言（GLSL）的3D图形软件数学库。
5. STB Image：一个用于加载图像的库。
6. Assimp：一个用于加载和处理3D模型的库。
7. LearnOpenGL：一个自定义库，提供着色器、相机、动画器和模型的类。
8. irrKlang：一个加载背景音乐的库

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <learnopengl/shader.h>
#include <learnopengl/camera.h>
#include <learnopengl/model.h>
#include <irrklang/irrKlang.h>
```

三.计划安排及组内成员分工

1. **项目计划**

o **第一阶段**：完成项目的整体框架设计，搭建OpenGL开发环境，完成基本的物体绘制与纹理贴图功能。

o **第二阶段**：实现光照和阴影效果，开发视角切换功能，初步完成用户交互功能。

o **第三阶段**：加入背景音乐模块，完善水波流动效果和场景细节，完成所有主要功能的调试和测试。

o **第四阶段**：进行最终优化，修复BUG，撰写项目报告。

2. **组内成员分工**

江颢怡：

开发负责人：负责总体技术架构设计与代码实现。

光照与效果负责人：负责光照、阴影效果以及水波流动等自然效果的实现。

用户交互与GUI负责人：负责实现用户交互操作与场景编辑功能。

邱丽梦：

物体建模与纹理负责人：负责物体的建模、纹理制作及贴图实现。

音乐与音效负责人：负责背景音乐的选择与实现。

四.分模块实验代码解释

1.三种不同的照明效果

三种照明效果分别指 **点光源，平行光源，手电筒光源**,并都具有位置、环境光、散射光和镜面光等属性，与我们的第二次作业

这里以点光源为例

①一共设置了四个**点光源**位置

```
// positions of the point lights
glm::vec3 pointLightPositions[] = {
    glm::vec3(7,  20,  20),
    glm::vec3(23, 60, -40),
    glm::vec3(-40.0f,  50.0f, -120.0f),
    glm::vec3(0.0f,  100.0f, -30.0f)
};
```

②然后为调用shader里的片段着色器传递所需参数，设置ambient，diffuse，specular等参数

```
// point light 1
cityShader.setVec3("pointLights[0].position", pointLightPositions[0]);
cityShader.setVec3("pointLights[0].ambient", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[0].diffuse", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[0].specular", 1.0f, 1.0f, 1.0f);
cityShader.setFloat("pointLights[0].constant", 1.0f);
cityShader.setFloat("pointLights[0].linear", 0.09f);
cityShader.setFloat("pointLights[0].quadratic", 0.032f);
// point light 2
cityShader.setVec3("pointLights[1].position", pointLightPositions[1]);
cityShader.setVec3("pointLights[1].ambient", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[1].diffuse", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[1].specular", 1.0f, 1.0f, 1.0f);
cityShader.setFloat("pointLights[1].constant", 1.0f);
cityShader.setFloat("pointLights[1].linear", 0.09f);
cityShader.setFloat("pointLights[1].quadratic", 0.032f);
// point light 3
cityShader.setVec3("pointLights[2].position", pointLightPositions[2]);
cityShader.setVec3("pointLights[2].ambient", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[2].diffuse", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[2].specular", 1.0f, 1.0f, 1.0f);
cityShader.setFloat("pointLights[2].constant", 1.0f);
cityShader.setFloat("pointLights[2].linear", 0.09f);
cityShader.setFloat("pointLights[2].quadratic", 0.032f);
// point light 4
cityShader.setVec3("pointLights[3].position", pointLightPositions[3]);
cityShader.setVec3("pointLights[3].ambient", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[3].diffuse", 0.3f, 0.3f, 0.3f);
cityShader.setVec3("pointLights[3].specular", 1.0f, 1.0f, 1.0f);
cityShader.setFloat("pointLights[3].constant", 1.0f);
cityShader.setFloat("pointLights[3].linear", 0.09f);
cityShader.setFloat("pointLights[3].quadratic", 0.032f);
```

2.相机视角切换

相机视角切换主要是通过processInput函数实现的，函数通过 glfwGetKey 检测键盘按键状态，根据输入对窗口状态、摄像机位置、对象平移、缩放及旋转参数等进行更新.

1. 核心功能：
   ESC：退出程序。
   W/S/A/D：控制摄像机移动。
   方向键：控制物体在 X 和 Z 轴方向上的平移。

N/M：控制物体的缩放。
J/K：控制物体的旋转。

2. 具体实现代码如下：

```
void processInput(GLFWwindow* window){
  // 如果用户按下 ESC 键，设置窗口关闭标志，准备退出应用程序
  if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
      glfwSetWindowShouldClose(window, true);

  // 检测 W/S/A/D 键，更新摄像机的位置
  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);

  // 检测方向键，更新物体在 X 和 Z 轴上的平移
  if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS)
      transx -= 1.0; // 按下左方向键，物体沿 X 轴负方向移动
  if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS)
      transx += 1.0; // 按下右方向键，物体沿 X 轴正方向移动
  if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
      transz -= 1.0; // 按下上方向键，物体沿 Z 轴负方向移动
  if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
      transz += 1.0; // 按下下方向键，物体沿 Z 轴正方向移动

  // 检测 N 和 M 键，调整物体的缩放比例
  if (glfwGetKey(window, GLFW_KEY_N) == GLFW_PRESS)
      scale += 0.01; // 按下 N 键，增大物体缩放比例
  if (glfwGetKey(window, GLFW_KEY_M) == GLFW_PRESS)
      scale -= 0.01; // 按下 M 键，减小物体缩放比例
```

1. 检测 ESC 键是否按下，若按下则通过 glfwSetWindowShouldClose 设置窗口应关闭，退出程序主循环。
2. 调用了摄像机类的 ProcessKeyboard 方法，根据键值 (FORWARD, BACKWARD, LEFT, RIGHT) 调整摄像机的位置，deltaTime 用于确保移动速度不受帧率影响（实现帧率独立的运动）。

```
W：向前移动摄像机（前进）
S：向后移动摄像机（后退）
A：向左平移摄像机（左移）
D：向右平移摄像机（右移）
```

3. 变量 transx 和 transz 记录物体的平移状态，每次按键改变对应变量的值。

```
方向键 LEFT / RIGHT：用于调整物体在 X 轴方向 的位置（左右平移）。
方向键 UP / DOWN：用于调整物体在 Z 轴方向 的位置（前后平移）。
```

4. 变量 scale 控制物体的缩放比例，用于在渲染时调整模型的大小。

```
N 键：增大物体缩放比例，每次按下增加 0.01。
M 键：减小物体缩放比例，每次按下减少 0.01。
```

3.鼠标输入处理实现和用户交互

①这里将鼠标光标模式设置为GLFW_CURSOR_DISABLED，即隐藏鼠标光标并且鼠标的移动输入会被程序完全捕获，让鼠标操作完全服务于场景内的视角控制等功能，而不会显示常规的鼠标光标在窗口上，为用户提供更沉浸式体验

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

  // 将鼠标移动事件与自定义的mouse_callback函数进行关联，当鼠标在指定窗口（window）内移动时，GLFW会自动调用mouse_callback函数来处理相关逻辑
  glfwSetCursorPosCallback(window, mouse_callback);

  // 将鼠标滚轮滚动事件与自定义的scroll_callback函数进行关联，当鼠标滚轮在指定窗口（window）内滚动时，GLFW会调用scroll_callback函数来处理相应操作
  glfwSetScrollCallback(window, scroll_callback);
```

②mouse_callback函数

主要负责处理鼠标移动的输入，更新摄像机的视角。当鼠标移动时，该回调函数会被 GLFW 自动调用。

```
// glfw: 当鼠标移动时，该回调函数会被调用
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
{
    // 将鼠标双精度坐标转换为单精度浮点数
    float xpos = static_cast<float>(xposIn);
    float ypos = static_cast<float>(yposIn);

    // 如果是鼠标的第一次移动，初始化上一帧的坐标
    if (firstMouse)
    {
        lastX = xpos; // 记录鼠标初始 X 坐标
        lastY = ypos; // 记录鼠标初始 Y 坐标
        firstMouse = false; // 标记鼠标已初始化
    }

    // 计算鼠标在 X 和 Y 轴上的偏移量
    float xoffset = xpos - lastX;   // X 轴的偏移量
    float yoffset = lastY - ypos;  // Y 轴的偏移量，注意 Y 轴反向

    // 更新上一帧的鼠标位置
    lastX = xpos;
    lastY = ypos;

    // 调用摄像机的处理函数，根据鼠标的偏移量调整摄像机的视角
    camera.ProcessMouseMovement(xoffset, yoffset);
}
```

③scroll_callback函数

会在用户使用鼠标滚轮时被调用。它的作用是处理滚轮的输入，通常用于调整视野（如缩放场景或摄像机）。

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    // 调用摄像机类的 ProcessMouseScroll 方法来处理滚轮的输入
    camera.ProcessMouseScroll(static_cast<float>(yoffset));
}
```

4.主渲染循环--着色器加载（顶点和片段）

cityshader主要用于广州塔的着色处理

ourShader主要用于剩余街景的着色处理

skyboxShader主要用于天空盒的处理

这里以广州塔的着色处理为例

①创建了一个名为 cityShader 的 Shader 类对象，并且通过传入两个参数 "model.vs" 和 "model.fs"，尝试从对应的文件中加载顶点着色器（Vertex Shader）和片段着色器（Fragment Shader）代码来构建一个完整的着色器程序，用于后续在 OpenGL 渲染中对图形进行渲染处理。

```
Shader cityShader("model.vs", "model.fs");
```

②同理还创建了Shader ourShader("model.vs", "model.fs"); Shader skyboxShader("skybox.vs", "skybox.fs");**一共三个着色器**

③将cityshader用于我们加载的Canton Tower.obj模型的渲染

```
 glm::mat4 model = glm::mat4(1.0f);
 //对 Tower 模型进行模型变换（平移、旋转、缩放）的设置
 model = glm::translate(model, glm::vec3(transx, 0.0f, transz));
 model = glm::rotate(model, float(glm::radians(moverot)), glm::vec3(.0f, 1.0f,
0.0f));
 model = glm::scale(model, glm::vec3(scale));
 cityShader.setMat4("model", model);//将变换矩阵传递给着色器，然后调用模型的绘制方法
 Tower.Draw(cityShader);//将cityshader用于我们加载的Canton Tower.obj模型的渲染
```

5.天空盒

在场景中最后渲染，以提供背景环境，使用立方体贴图和专门的着色器。

1. 首先定义了一个名为 `skyboxVertices` 的数组，用于存储天空盒的顶点数据，为后续调用skybox.vs顶点着色器代码对顶点数据进行相应的坐标变换做准备
2. 然后创建 `skyboxShader` 着色器，加载顶点和片段着色器代码。

```
    //整个数组按照一定顺序罗列了构成天空盒（立方体）六个面的所有顶点坐标
    Shader skyboxShader("skybox.vs", "skybox.fs");
    GLfloat skyboxVertices[] = {
        // Positions
        -1.0f,  1.0f, -1.0f,
        -1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f,  1.0f, -1.0f,
```

```
        -1.0f,  1.0f, -1.0f,

        -1.0f, -1.0f,  1.0f,
        -1.0f, -1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f, -1.0f,
        -1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,

        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,

        -1.0f, -1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f, -1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,

        -1.0f,  1.0f, -1.0f,
        1.0f,  1.0f, -1.0f,
        1.0f,  1.0f,  1.0f,
        1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f,  1.0f,
        -1.0f,  1.0f, -1.0f,

        -1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f,  1.0f,
        1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f,
        -1.0f, -1.0f,  1.0f,
        1.0f, -1.0f,  1.0f
    };
```

3. 生成 `skyboxVAO` 和 `skyboxVBO`，配置顶点属性并上传顶点数据，之后解绑 `skyboxVAO`。

```
  //  声明了两个无符号整数类型（GLuint）的变量 skyboxVAO 和 skyboxVBO，它们分别用于表示天空盒的顶点数组对象 VAO 和顶点缓冲对象 VBO。
  GLuint skyboxVAO, skyboxVBO;

  glGenVertexArrays(1, &skyboxVAO);
  glGenBuffers(1, &skyboxVBO);

  glBindVertexArray(skyboxVAO);//将之前生成的 skyboxVAO 顶点数组对象绑定为当前活动的 VAO。意味着后续对顶点属性的配置都会应用到这个绑定的 VAO 所关联的顶点数据上

  glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);//将 skyboxVBO 顶点缓冲对象绑定到指定的缓冲类型 GL_ARRAY_BUFFER（表示这是一个用于存储顶点属性数据的缓冲类型）上，后续对缓冲数据的操作（比如通过 glBufferData 函数向缓冲中存储数据）就会作用在这个绑定的 VBO 上。
```

```
  glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices,
GL_STATIC_DRAW);

  glEnableVertexAttribArray(0);

  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat),
(GLvoid*)0);

  glBindVertexArray(0);//这行代码用于解除当前绑定的顶点数组对象（VAO）。将之前绑定的
skyboxVAO 解绑后，就不会再有默认的活动 VAO，这样可以避免后续一些意外的操作影响到已经配置好的
skyboxVAO 的状态
```

4. 准备立方体贴图纹理各面文件路径，调用loadCubemap 函数加载生成纹理贴到天空盒的六个面上，使天空盒能够正确显示带有纹理的效果。

```
  //这里定义了一个string 类型的向量（vector），名为 faces，用于存储构成天空盒（立方体贴图）的
六个面所对应的纹理图片文件路径。每个元素对应天空盒一个面的纹理图片路径，例如
"resource/right.jpg" 表示天空盒右侧面的纹理图片路径
  vector<std::string> faces
  {
    "resource/right.jpg",
    "resource/left.jpg",
    "resource/top.jpg",
    "resource/bottom.jpg",
    "resource/front.jpg",
    "resource/back.jpg"
  };

  unsigned int cubemapTexture = loadCubemap(faces);//尝试从对应的文件路径加载六个面的纹
理图片，生成一个立方体贴图纹理，并将生成的纹理对象的标识符存储到 cubemapTexture 变量中

  skyboxShader.use();
  skyboxShader.setInt("skybox", 0);//使天空盒能够正确显示带有纹理的效果
```

5. 天空盒的渲染（不同于街景和广州塔的shader设置）

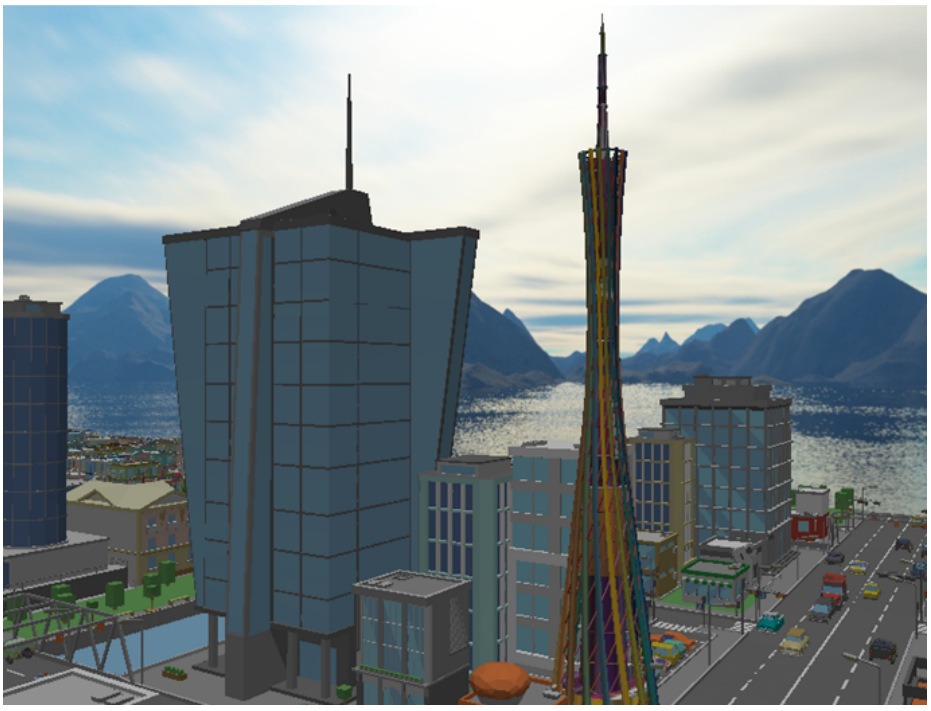
```
//获取相机的观察矩阵（通过 camera.GetViewMatrix()），然后将这个矩阵转换为只包含旋转和缩放信息
的新矩阵（去除了平移部分）。因为天空盒是作为背景，通常只需要根据相机的旋转和缩放来调整显示角度和大
小，不需要跟随相机的平移而移动，这样处理后的矩阵更符合天空盒的渲染需求。
// draw skybox as last
glDepthFunc(GL_LEQUAL);  // change depth function so depth test passes when
values are equal to depth buffer's content
skyboxShader.use();
view = glm::mat4(glm::mat3(camera.GetViewMatrix())); // remove translation from
the view matrix
skyboxShader.setMat4("view", view);
skyboxShader.setMat4("projection", projection);
// skybox cube
glBindVertexArray(skyboxVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthFunc(GL_LESS); // set depth function back to default
```

6. loadCubemap 函数实现加载一个立方体贴图（Cubemap），包含 6 张纹理图像的纹理，用于天空盒。它会将每张图像绑定到立方体贴图的一个面上，并设置相关参数。

```
  GLuint loadCubemap(vector<std::string> faces)
  {
      unsigned int textureID;
      // 生成立方体贴图的纹理对象
      glGenTextures(1, &textureID);
      glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

      int width, height, nrChannels;
      // 遍历 6 张图像并加载
      for (unsigned int i = 0; i < faces.size(); i++)
      {
          unsigned char* data = stbi_load(faces[i].c_str(), &width, &height,
&nrChannels, 0);
          if (data)
          {
              // 将每张图像数据绑定到立方体贴图的对应面上
              glTexImage2D(
                  GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,  // 指定立方体贴图的一个面
                  0,                                   // Mipmap 级别
                  GL_RGB,                              // 内部格式
                  width, height,                       // 图像的宽度和高度
                  0,                                   // 边框大小，必须为 0
                  GL_RGB,                              // 数据格式
                  GL_UNSIGNED_BYTE,                    // 数据类型
                  data                                 // 图像数据
              );
              stbi_image_free(data); // 释放图像数据
          }
          else
          {
              // 图像加载失败，输出错误信息并释放内存
              std::cout << "Cubemap texture failed to load at path: " << faces[i]
<< std::endl;
              stbi_image_free(data);
          }
      }

      // 设置立方体贴图的纹理参数
      glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR); //
缩小时的线性过滤
      glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR); //
放大时的线性过滤
      glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
// S轴（水平）边缘包裹方式
      glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// T轴（垂直）边缘包裹方式
      glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
// R轴（深度）边缘包裹方式

      // 返回立方体贴图的纹理 ID
      return textureID;
  }
```

最终天空盒效果

（这里因为时间太赶 没有找到合适的天空盒贴图 后续可以优化）



6.顶点着色器和片段着色器的详细解释

skybox

1. 顶点着色器 skybox.vs

   负责将天空盒的立方体顶点坐标转换为裁剪空间，并输出纹理坐标，用于片段着色器采样立方体贴图。

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

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;  // 直接将顶点位置作为纹理坐标
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;  // 特殊处理 z 分量，确保天空盒始终在最远处
}
```

2. 片段着色器 skybox.fs

使用采样器 `samplerCube` 根据片段的纹理坐标从立方体贴图中采样颜色，并输出。

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;
```

```
uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);  // 从立方体贴图中采样颜色
}
```

Skybox 技术要点：

1. 使用立方体贴图（samplerCube）渲染天空盒背景。
2. 丢弃深度缓冲中的 z 分量（gl_Position = pos.xyww），使天空盒始终处于视野的最远端。
3. 天空盒的顶点位置直接作为纹理采样坐标，无需额外变换。

广州塔模型

顶点着色器（`model.vs` 内的代码）和片段着色器（`skybox.fs` 文件内的代码）负责加载并渲染广州塔模型，包括光照、纹理贴图等效果。

1.顶点着色器 model.vs

处理模型的顶点位置、法向量以及纹理坐标，完成从 模型空间 到 裁剪空间 的转换。
```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;   // 片段的世界空间位置
out vec3 Normal;    // 片段的世界空间法向量
out vec2 TexCoords; // 纹理坐标

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

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));   // 模型空间 -> 世界空间
    Normal = mat3(transpose(inverse(model))) * aNormal;  // 法向量转换到世界空间
    TexCoords = aTexCoords;  // 传递纹理坐标

    gl_Position = projection * view * vec4(FragPos, 1.0); // 世界空间 -> 裁剪空间
}
```

2. 片段着色器 model.fs

实现基于光照的颜色计算，并采样纹理贴图，为模型赋予真实感。
```
#version 330 core
out vec4 FragColor;
```

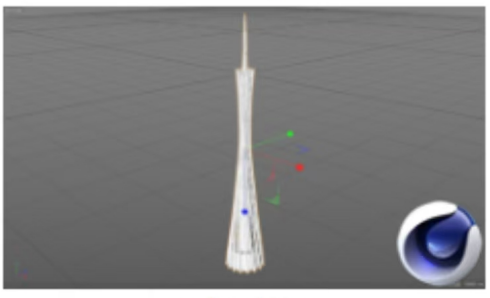
```
in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

uniform vec3 lightPos;    // 光源位置
uniform vec3 viewPos;     // 摄像机位置
uniform vec3 lightColor;  // 光源颜色
uniform sampler2D texture_diffuse1; // 纹理采样器

void main()
{
    // 环境光
    vec3 ambient = 0.1 * lightColor;

    // 漫反射
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    // 镜面高光
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = spec * lightColor;

    // 结合光照与纹理
    vec3 lighting = (ambient + diffuse + specular);
    vec3 textureColor = texture(texture_diffuse1, TexCoords).rgb;
    vec3 finalColor = lighting * textureColor;

    FragColor = vec4(finalColor, 1.0); // 输出片段颜色
}
```

模型渲染技术要点：

1. 光照计算：实现基础的 Phong 光照模型（包括环境光、漫反射、镜面反射）。
2. 法向量转换：法线向量需要通过法线矩阵（transpose(inverse(model))）进行变换。
3. 纹理采样：使用 sampler2D 从纹理贴图中获取颜色，与光照计算结果结合。

7.三维模型的贴图和纹理加载

刚开始加载的城市街景和广州塔的3D模型都是没有颜色的



canton tower

但我们知道实际的广州塔是有多种颜色的 为了更接近真实效果，我们手动选择贴图纹理

广州塔mtl文件的代码如下：

```
# Blender 3.4.1 MTL File: 'Canton Tower.blend'
# www.blender.org

newmtl Material_#26.006
Ns 37.321308
Ka 1.000000 1.000000 1.000000  //指定材质的环境光颜色（Ambient Color），这里表示白色，用于模拟材质在环境光照射下的基础颜色贡献。
Ks 1.000000 1.000000 1.000000
Ke 0.000000 0.000000 0.000000
Ni 1.450000
d 1.000000  //控制材质的透明度（Dissolve），值为 1 表示完全不透明。
illum 3  //指定光照模型（Illumination Model），用于确定材质与光照的交互方式，值为 3 表示使用具有镜面反射和反射光的光照模型。
map_Kd hippo.png  //将名为 “hippo.png” 的图像文件映射到材质的漫反射颜色（Diffuse Color）上，用于为材质表面提供纹理细节，使模型看起来更具真实感或特定的外观效果。
```

这个 MTL 文件与 Blender 中的模型文件配合使用，告诉渲染引擎如何渲染 “Canton Tower.blend” 模型中使用该材质的部分，包括材质的颜色、光照特性以及纹理外观等，从而实现最终的视觉效果。

8.添加天空之城的背景音乐

通过 irrKlang 库加载并循环播放 “天空之城” 轻音乐，为整个虚拟场景营造出更加轻松愉悦的氛围，增强了沉浸感。

```
 ISoundEngine* SoundEngine = createIrrKlangDevice();//创建一个音频播放引擎对象，以便在程序中实现音频播放相关的功能
 SoundEngine->play2D("resource/day.mp3", GL_TRUE);//GL_TRUE 开启循环播放功能
```

9.广州塔模型的左右旋转

为广州塔模型单独设计了左右旋转功能，通过检测键盘 J/K 键按下事件，调整旋转角度变量 moverot，使广州塔模型在场景中能够动态旋转，增加了场景的动态效果和趣味性

变量 moverot 控制物体的旋转角度，可用于在渲染时应用旋转变换。

```
// 检测 J 和 K 键，调整物体的旋转角度
if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS)
    moverot += 1; // 按下 J 键，顺时针旋转物体，角度增加
if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS)
    moverot -= 1; // 按下 K 键，逆时针旋转物体，角度减少
```

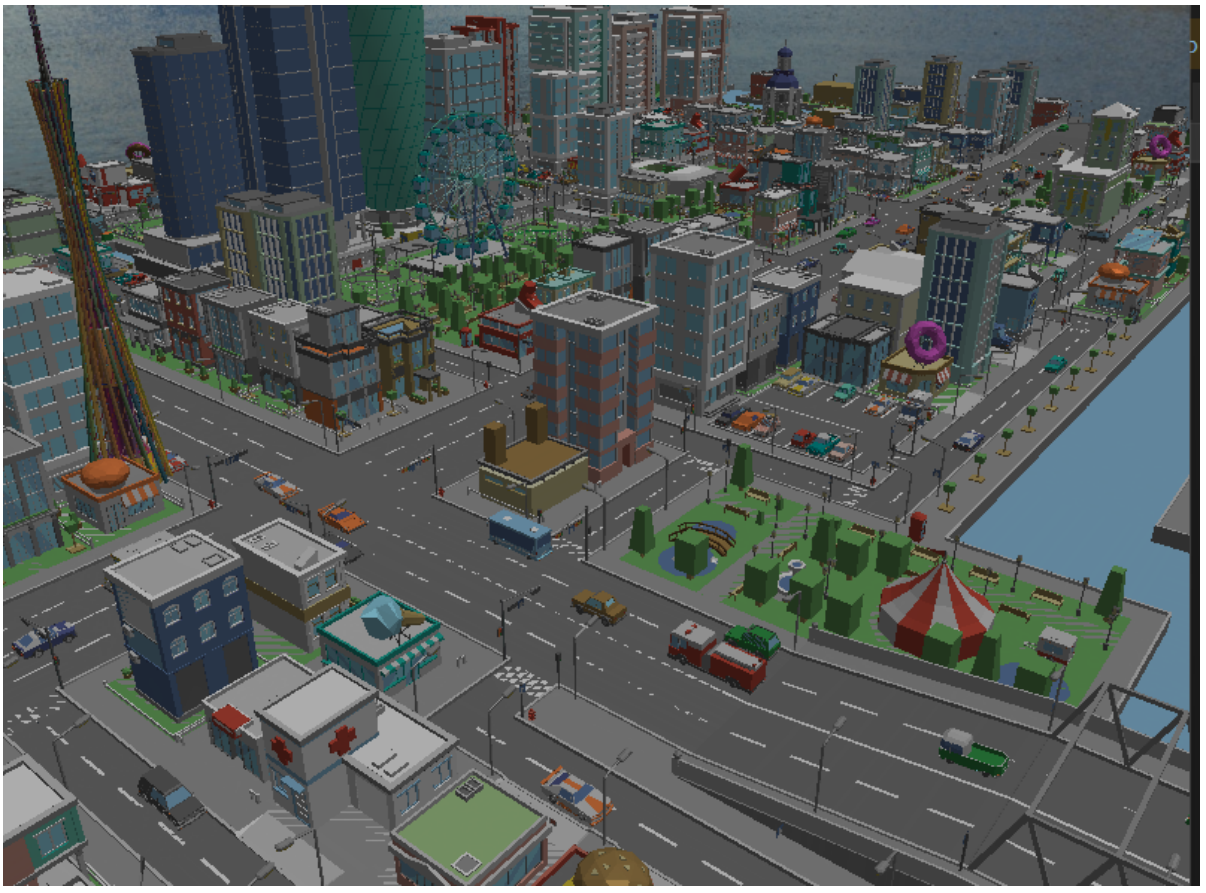
J键：顺时针旋转物体，增加 moverot 的值。
K 键：逆时针旋转物体，减少 moverot 的值。

五.实验效果

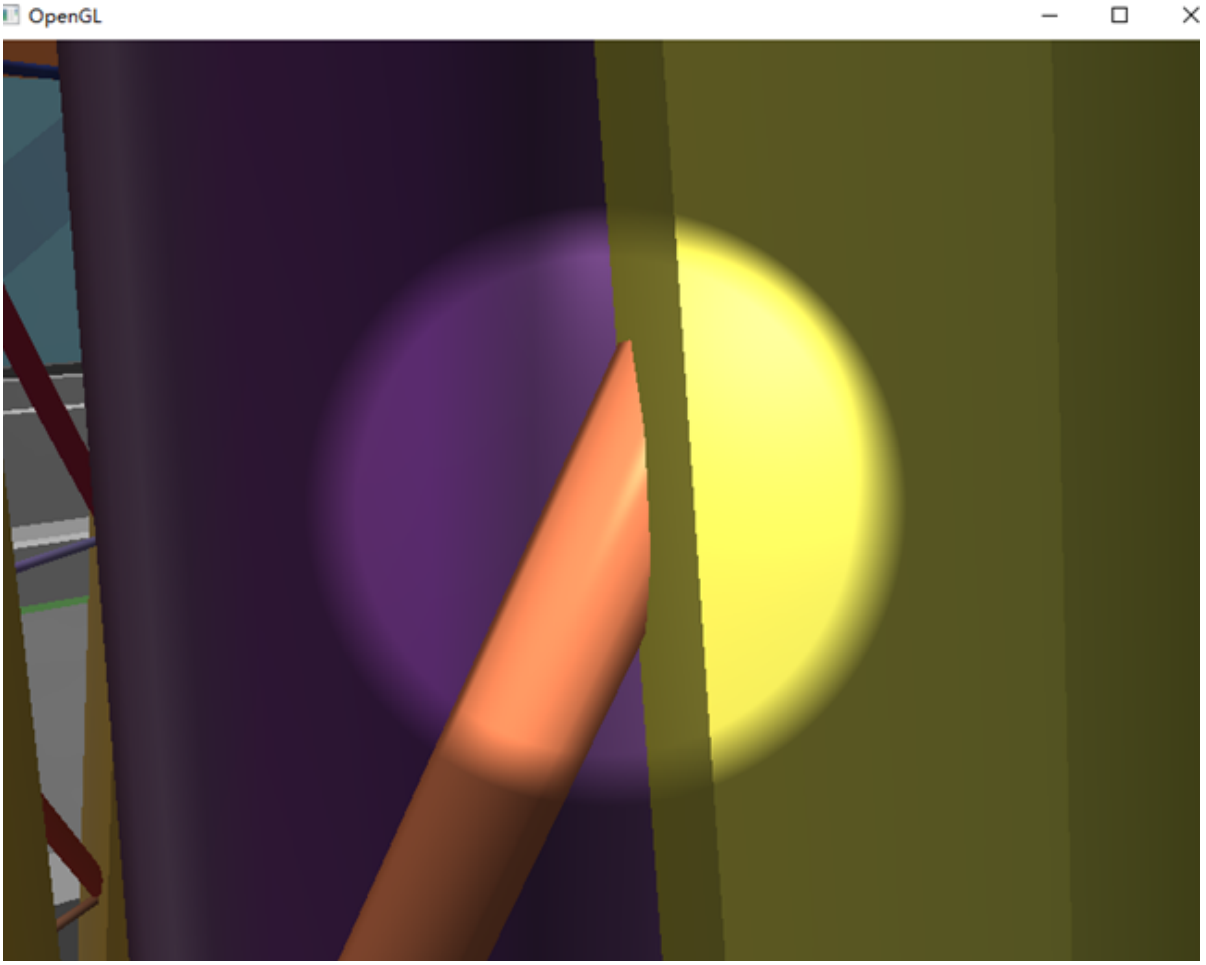
1.广州塔模型缩放场景截图

可以按NM键缩放广州塔在街景中的大小 灵活控制

2.场景截图



3.三种光照效果展示



4.音乐效果需要直接运行main函数启动项目看到