# 计算机图形学期末大作业-个人报告

# 题目：opengl 搭建广州塔城市街景

**22365043 江颢怡 专业：计算机科学与技术 指导老师：陶钧**

## 项目概述

我们小组的选题是**用OpenGL搭建并渲染一个场景**。最初想到要搭建场景，就联想到最具有广州特色的广州塔，所以组员之间一拍即合决定构建广州塔城市街景。在这个场景里，你可以轻松实现**漫游**，看到许多具有城市风格的建筑，并且可以根据自己的要求灵活**缩放旋转**广州塔模型，让他与城市街景融合的更完美，同时我们还加上了天空之城的**背景音乐**，为城市的场景漫游提供更沉浸式的体验。

主要涉及的技术功能包括加载和渲染3D模型、处理用户输入以及播放音频。

开发的过程

- 搭建一个基于OpenGL的城市街景，具备良好的交互性和沉浸感。
- 完成从物体建模、纹理贴图、光照阴影到用户视角和操作交互等一系列核心功能的实现。
- 通过对场景的漫游操作和背景音乐的加入，提升用户的游戏体验和参与感。

## 主要功能实现

### 初始化和设置

在main.cpp中，首先初始化GLFW和GLAD库，并创建一个窗口。然后设置了帧缓冲大小回调、鼠标回调和滚轮回调函数。（详见函数的分析见小组报告）

```
if (!glfwInit())
    return -1;

GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL,
NULL);
if (!window)
{
    glfwTerminate();
    return -1;
}

glfwMakeContextCurrent(window);
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}

glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
glfwSetCursorPosCallback(window, mouse_callback);
glfwSetScrollCallback(window, scroll_callback);
```

## 加载和渲染模型

使用Assimp库加载3D模型，并使用GLM库进行矩阵变换。**模型的渲染使用了自定义的着色器程序。**

```
Model ourModel("path/to/model.obj");
ourShader.use();
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, -1.75f, 0.0f));
model = glm::scale(model, glm::vec3(0.2f, 0.2f, 0.2f));
ourShader.setMat4("model", model);
ourModel.Draw(ourShader);
```

## 处理用户输入

通过GLFW库处理用户输入，包括**键盘和鼠标输入，用于控制摄像机的移动和视角的变化。**

```
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);
}
```

## 纹理贴图

在本项目中，实现了用Assimp库加载obj文件。当使用Assimp导入一个模型的时候，它通常会将整个模型加载进一个**场景**(Scene)对象，它会包含导入的模型/场景中的所有数据。Assimp会将场景载入为一系列的节点(Node)，每个节点包含了场景对象中所储存数据的索引。我们将创建我们自己的model和mesh类来加载并储存导入后的模型。

并且贴图的相关设置定义在mtl文件，通常与obj文件一起使用，用于定义3D模型的材质属性

```
# 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。
```

## 播放音频

使用IrrKlang库播放背景音乐和音效。

```
ISoundEngine* SoundEngine = createIrrKlangDevice();
if (!SoundEngine)
    return -1;

SoundEngine->play2D("path/to/sound.mp3", true);
```

## 天空盒实现

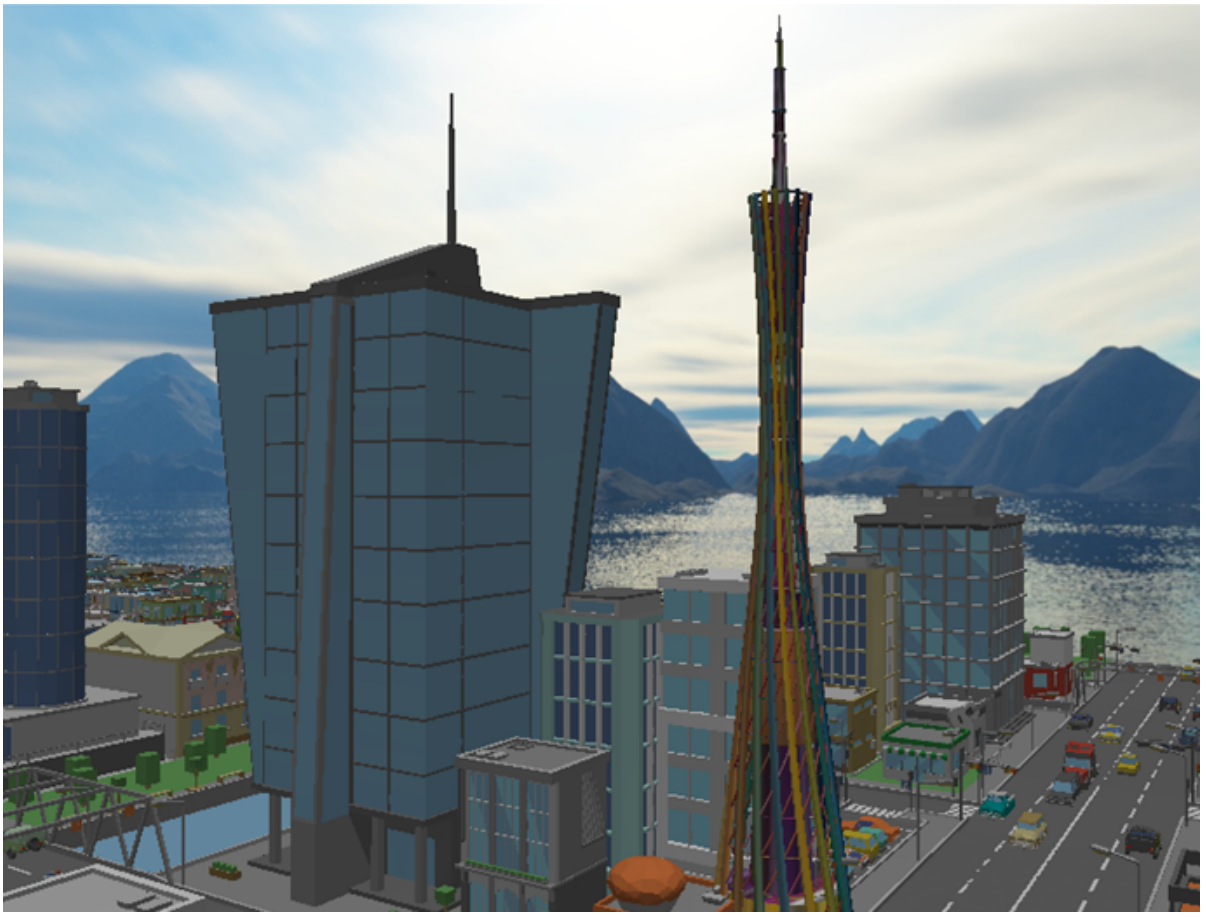
天空盒背景是通过立方体贴图实现的。所谓立方体贴图，就是一个由6个2D纹理组成的纹理，6个2D纹理分别对应立方体的6个面。将场景置于一个大的立方体内，6个面的纹理相互吻合形成全景世界，给予观察者一种处在比实际大得多的环境当中，这就是天空盒的效果。

```
 //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
```

最终效果



## 三种光照效果实现

可以看到这里定义的三种光照都有**环境光**（Ambient），**漫反射**（Diffuse），和**镜面反射**（Specular）三个成分，每种成分模拟了光在物体表面不同交互方式下的反射。这里其实用到了实验2的**基本光照模型 Phong 模型的**原理 很好的体现了实验的一个连续性

```
struct DirLight {
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct PointLight {
    vec3 position;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct SpotLight {
    vec3 position;
    vec3 direction;
    float cutOff;
    float outerCutOff;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

实验2自己的报告对光照模型原理的一个简单解释

1. 环境光 (Ambient)

环境光是所有光源的背景光，它的作用是给物体提供一个基本的亮度值，使得物体即使在没有任何直接光照的地方也能有一些可见度。环境光的亮度通常是固定的。

$$I_{\text{ambient}} = k_a \cdot I_a$$

2. 漫反射 (Diffuse)

漫反射模拟了光线与物体表面微小不规则结构的交互，导致光线在所有方向上均匀散射。漫反射的强度取决于光照方向和表面法线的关系。光照越与法线对齐，漫反射的强度越强。

漫反射分量使用 **Lambertian 反射模型** 计算：

$$I_{\text{diffuse}} = k_d \cdot I_d \cdot \max(0, \mathbf{L} \cdot \mathbf{N})$$

3. 镜面反射 (Specular)

镜面反射模拟了光在光滑表面（例如镜子）上的反射行为。光线在镜面表面上的反射方向通常会集中到一个小的区域，形成高亮的反射效果。镜面反射的强度不仅依赖于光照方向和法线的关系，还依赖于观察方向与反射光线的关系。

镜面反射的强度可以使用 **Phong 镜面反射模型** 计算：

$$I_{\text{specular}} = k_s \cdot I_s \cdot (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\alpha}$$

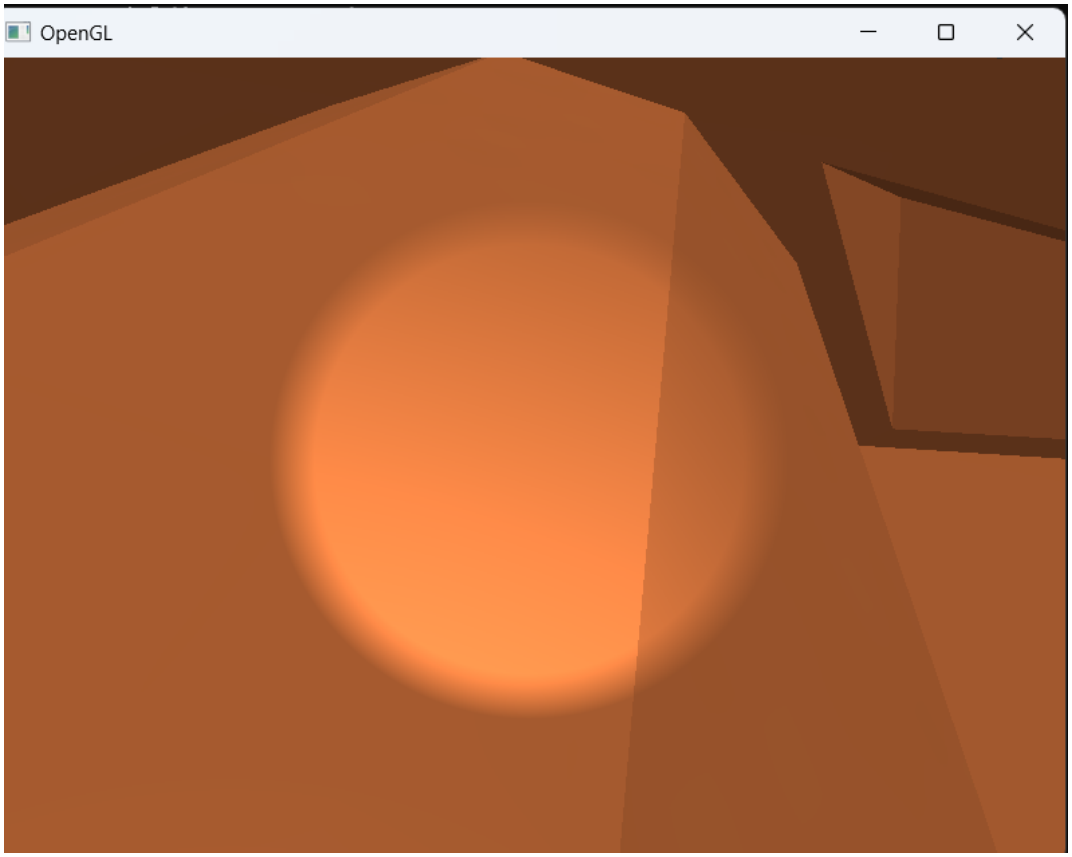
综合以上三个分量，Phong 光照模型可以将环境光、漫反射和镜面反射合成，计算每个像素的总光照：

$$I_{\text{total}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

或者：

$$I_{\text{total}} = k_a \cdot I_a + k_d \cdot I_d \cdot \max(0, \mathbf{L} \cdot \mathbf{N}) + k_s \cdot I_s \cdot (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\alpha}$$

这里的光照效果实际上是跟着摄像机移动的




# 技术心得

1. **OpenGL的使用**: OpenGL是一个功能强大的图形库，但其API较为底层，使用起来需要对图形学有一定的了解。通过本项目的实践，我熟悉了OpenGL的基本使用方法，包括上下文创建、着色器编写、VAO/VBO的使用等。
2. **第三方库的集成**: 项目中集成了多个第三方库，如GLFW、GLAD、GLM、Assimp和IrrKlang。每个库都有其独特的使用方法和注意事项，通过阅读相关的文档，我们较好地将这些库集成到项目中。

3. **矩阵变换**: 对于摄像机的变换实现场景漫游功能，我使用了GLM库进行矩阵变换，使得模型的平移、旋转和缩放变得更加方便。GLM库提供了丰富的数学函数，极大地简化了矩阵运算的复杂度。
4. **模型加载和渲染**: 使用Assimp库加载3D模型，并使用自定义的着色器进行渲染。通过学习Assimp的使用方法，能够加载多种格式的3D模型，并进行基本的处理和渲染。
5. **音频播放**: 使用IrrKlang库进行音频播放，能够方便地添加背景音乐和音效，增强了项目的互动

# 实验心得

**1.opengl都是状态机函数** 我在搭建场景的最开始，出现了场景叠加混乱的问题，最后发现是缓冲区没有及时清理， 在每个新的渲染迭代（即每次游戏循环）开始之前都需要先清屏glClearColor是状态设置函数 glClear是状态应用的函数

**2.我们需要开启深度测试**，灵活设置深度值，因为在深度缓冲区（z-buffer）中，深度值越大越会被掩盖，深度值越小越会显示在前面。

**3.着色的一系列步骤顺序：**顶点着色器->几何着色器->光栅化->片段着色器->深度测试（即渲染时考虑深度的大小）并且最后要把不同的着色器对象链接到一个用来渲染的着色器程序中

**4.双缓冲渲染窗口应用程序**：应用程序使用单缓冲绘图时可能会存在图像闪烁的问题。 这是因为生成的图像不是一下子被绘制出来的，而是按照从左到右，由上而下逐像素地绘制而成的。最终图像不是在瞬间显示给用户，而是通过一步一步生成的，这会导致渲染的结果很不真实。为了规避这些问题，我们应用双缓冲渲染窗口应用程序。前缓冲保存着最终输出的图像，它会在屏幕上显示；而所有的的渲染指令都会在后缓冲上绘制。当所有的渲染指令执行完毕后，我们交换(Swap)前缓冲和后缓冲，这样图像就立即呈显出来，之前提到的不真实感就消除了。

总之，通过这次期末大作业，在技术层面，我深入掌握了 OpenGL 的核心概念和编程技巧，包括着色器编程、纹理映射、光照模型实现以及图形变换等。学会了如何高效地利用各种库来简化开发流程，提高开发效率，如使用 LearnOpenGL 库提供的类来快速构建着色器、相机和模型等组件。

在算法方面，对光照计算算法（如 Phong 光照模型）有了更深入的理解和实践应用经验，能够根据场景需求准确设置光照参数，实现逼真的光照效果。同时，在处理天空盒的深度缓冲和纹理坐标时，掌握了相关的优化技巧，确保天空盒的正确显示和性能表现。

**不足之处与改进方向**

- 尽管场景在功能和效果上取得了一定成果，但在性能优化方面仍有提升空间。例如，在复杂场景下可能出现帧率下降的情况，尤其是当场景中物体数量增多或光照计算复杂时。未来可以考虑采用更高效的算法和数据结构来优化场景渲染，如空间划分算法（如八叉树）来减少不必要的光照计算和渲染操作，提高渲染效率。
- 在纹理细节方面，虽然已经应用了纹理贴图来增强物体真实感，但部分纹理在近距离观察时可能显得不够精细。可以进一步探索更高分辨率的纹理资源或者采用纹理压缩技术，在不显著增加内存占用的前提下提升纹理质量。