山东大学 计算机科学与技术 学院

计算机图形学 课程实验报告

学号：202200130119	姓名：于斐	班级：学堂计机 22
实验题目：模型绘制与交互		
实验学时： 4	实验日期： 2024 年 11 月 23 日	

实验目的：

通过使用 OpenGL 编程实现一个真实感 3D 模型的绘制与交互，培养学生对计算机图形学的基本概念、技术和开发能力的掌握。通过从 OBJ 等格式文件中加载复杂模型并进行真实感显示，理解 3D 模型的渲染过程及其与纹理、光照的交互原理。通过实现平移、旋转、缩放等基本操作，熟悉 3D 模型的变换矩阵及其在 OpenGL 中的应用。最终，通过优化交互方式，提升用户对 3D 模型操作的自然性和直观性，为后续复杂的图形学应用开发奠定基础。

实验步骤与内容：

实验环境：OpenGL 4.6 及 GLFW，GLM 等附属库。

实验步骤：

0. 项目使用 CMake 管理，原则上任何支持 CMake 的编辑器均可使用。所有未包含在项目文件中的库均使用 CMake 的 FetchContent 导入，不需额外手动安装任何库。程序自动生成到 dist 目录下，但运行时 pwd 需包含 assets。因此需要在根目录运行 dist/Renderer 或将 assets 拷贝到 dist 下。
1. 建立 OpenGL 窗体及 argument parser，config parser。前者代码位于 Renderer::init()中，在 src/renderer/renderer.h，renderer.cpp 下可以找到。后者代码位于 src/utils 中。

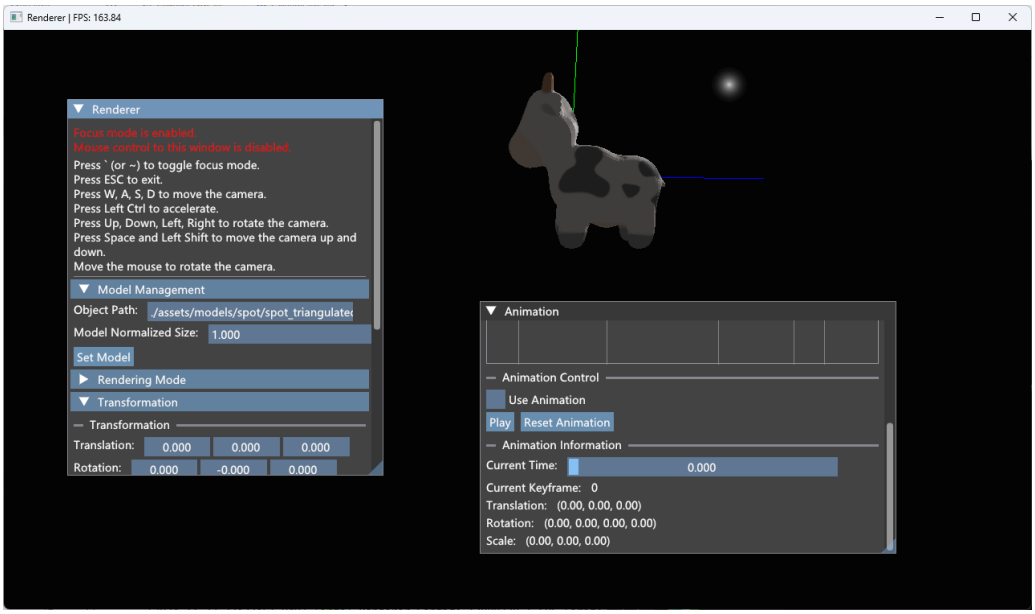
```cpp
void Renderer::init() {

    auto& globalConfig = getGlobalConfig();
    this->width = globalConfig.get<int>("window.width", 1280);
    this->height = globalConfig.get<int>("window.height", 720);
    this->fpsLimit = globalConfig.get<int>("window.fpsLimit", INT_MAX);

    spdlog::info("Renderer Config: {}x{} | FPS Limit: {}", width, height, fpsLimit == INT_MAX ? "Unlimited" : std::to_string(fpsLimit));


    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);

    window = glfwCreateWindow(width, height, "Renderer", nullptr, nullptr);
    if (!window) {
        spdlog::error("Failed to create GLFW window.");
        glfwTerminate();
        exit(-1);
    }
    glfwMakeContextCurrent(window);
```

```cpp
class ArgParser {
public:
    ArgParser(int argc, char* argv[]) {
        parser.add_argument("-v", "--verbose")
              .help("Enable verbose logging")
              .default_value(false)
              .flag();
        parser.add_argument("-t", "--trace")
              .help("Enable trace logging")
              .default_value(false)
              .flag();
        parser.add_argument("-c", "--config")
              .help("Path to config file")
              .default_value(std::string("./assets/con
        try {
            parser.parse_args(argc, argv);
        } catch (const std::exception& e) {
            spdlog::error("{}", e.what());
            exit(-1);
        }
    }

    template<typename T>
    T get(std::string_view arg_name) {
        return parser.get<T>(arg_name);
    }
```

```cpp
#ifndef YAML_LOADER_H__
class YAMLLoader {
    template<typename T>
    T get(const std::string& key) const {
        auto split = [](const std::string& s, char delim) -> s
            std::vector<std::string> result;
            std::stringstream ss(s);
            std::string item;
            while (std::getline(ss, item, delim)) {
                result.push_back(item);
            }
            return result;
        };
        auto keys = split(key, '.');
        YAML::Node node = YAML::Clone(config);
        for (const auto& k : keys) {
            node = node[k];
        }
        try {
            return node.as<T>();
        } catch (const YAML::Exception& e) {
            throw std::runtime_error("Failed to get key: " + k
        }
    }
```

部分实验代码

使用 ImGUI 构建交互选单，用于控制程序运行时的行为。相关代码位于 src/gui/ 下。

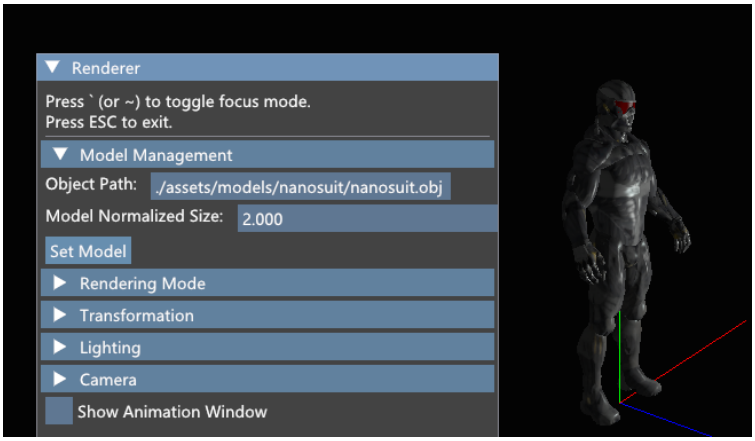


2. 构建 Model 类用于从 .obj 中导入模型（包含 vertices position, normal（如果没有则插值获得）, texcoord）。此处使用 tiny obj loader 库辅助实现。支持导入同一 obj 文件中的不同 object,并导入 diffuse, specular, normal texture。导入的数据首先在 CPU 侧储存，后建立 OpenGL Vertex Arrays, Buffer Data, Texture 送入 GPU 侧。

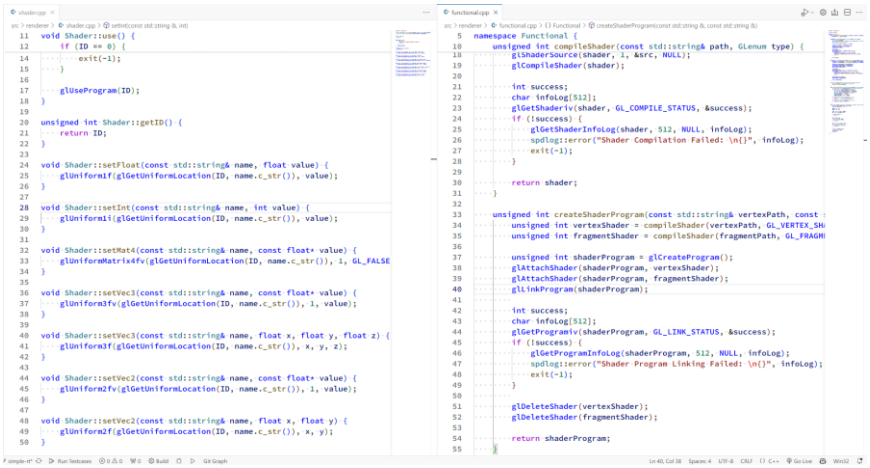
```cpp
void Mesh::setup() {
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeo

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size()

    // Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeo
    glEnableVertexAttribArray(0);
    // Normal
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeo
    glEnableVertexAttribArray(1);
    // Texture
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeo
    glEnableVertexAttribArray(2);

    glBindVertexArray(0);
}
```

```cpp
Texture::Texture(const std::string& path) {
    m_data = stbi_load(path.c_str(), &m_width, &m_height, &m_ch

    if (!m_data) {
        spdlog::error("Failed to load texture: {}", path);
        return;
    }

    glGenTextures(1, &m_id);
    glBindTexture(GL_TEXTURE_2D, m_id);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LI
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LI

    if (m_channels == 3) {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, m_width, m_height
    } else if (m_channels == 4) {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_width, m_heig
    } else {
        spdlog::error("Unknown number of channels: {}", m_chann
    }

    glGenerateMipmap(GL_TEXTURE_2D);

    // stbi_image_free(m_data);
}
```

程序支持运行时替换模型。读入的模型会被归一化为统一大小（默认为 XYZ -1 ~ 1）。



3. 构建 Shader 类用于加载 Shader Program。加载方式与大多 OpenGL 加载器相同，在此不再赘述。



本实验用到了 5 个 shader（位于 assets/shaders/ 下）：

a) rasterization/object.frag/vert：渲染物体的 shader，其中 vertex shader 做变换（model * view * projection），fragment shader 做 phong 模型的绘制，其中 ambient，diffuse 使用 diffuse 贴图 + 光照，specular 使用 specular 贴图。变换矩阵、光照通过 uniform 送入。

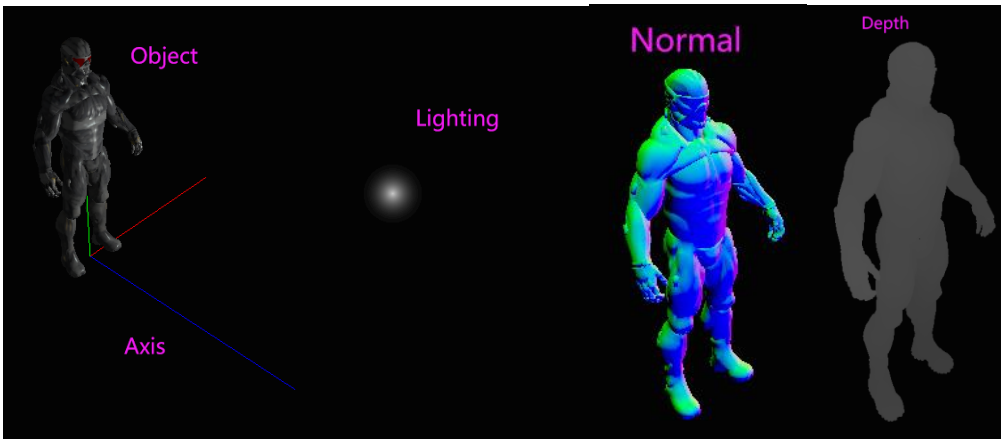
b) rasterization/lighting.frag/vert：在屏幕中绘制一个虚拟的光球，代表此

时的光源位置。通过 opengl 的点绘制（glDrawArrays GL_POINTS）实现。Uniform 变量同上。

c) rasterization/axis.frag/vert：坐标轴绘制，绘制出一个 x，y，z 坐标轴，类似一般的 object viewer。

d) normal/object.frag/vert：绘制物体的法线。将 fragment shader 中的输出颜色改为了 normalized normal。

e) depth/object.frag/vert：绘制物体的深度图。通过 gl_FragCoord.z 得到数据。



程序支持运行时更换渲染模式。支持运行时光照调整。

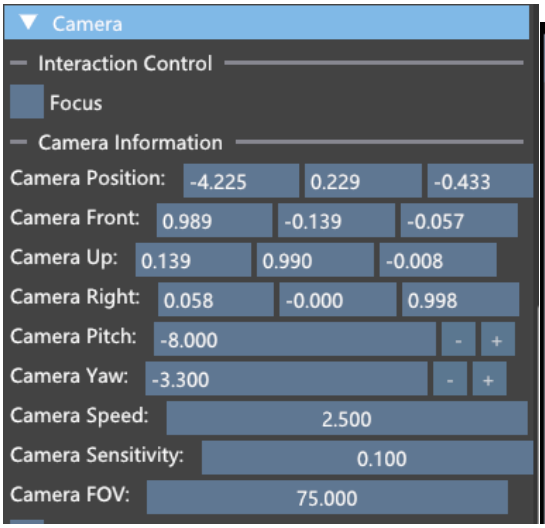
4. 构建 Camera 类，用于变换相机。可供变换的参数有控制姿态的 position，pitch，yaw、控制成像的 fov，near，far 和控制操作的 speed，sensitivity。

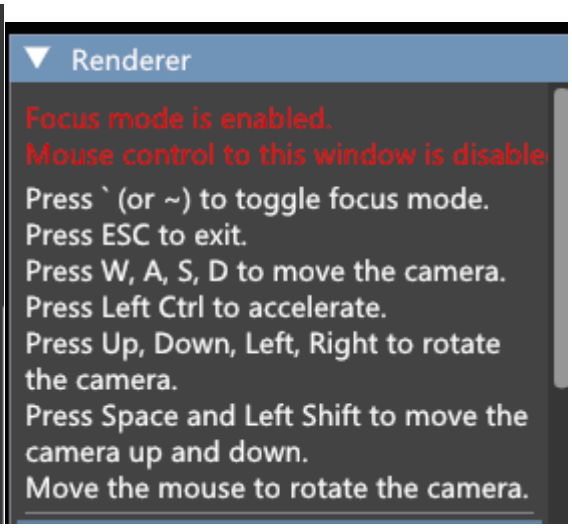
程序支持运行时相机调整，包含两种调整模式：FPS 鼠标调整、选单调整。使用 ` 键切换 FPS 模式（称 focus mode）。

在 focus mode 下，窗口将锁定鼠标，可使用类似 FPS 游戏的方式调整相机。

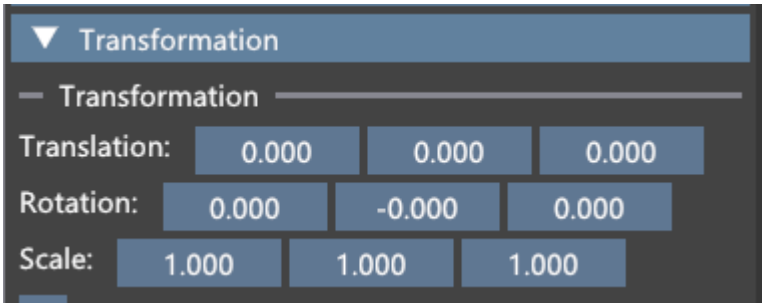
在选单中，可拖拽变换相机的 speed，sensitivity，fov。

相关代码位于 src/renderer/camera.cpp 下。

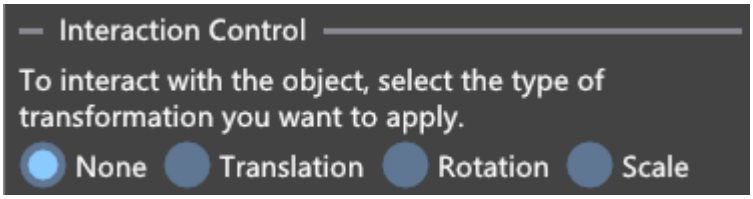




5. 构建物体变换方式。项目中构建了 Motion 类，支持 vec3 形式的 translation，scale 和 quaternion 形式的 rotation。在 gui 交互界面中，rotation 被表示为 euler angle，但是是由 quaternion 转译来的。



程序支持两种交互方式：通过 gui 设置数值，鼠标拖拽交互。
前者不再赘述，对于后者，提供了切换交互方式的选单，在选单中可以选择不同的交互方式。



在选择了某种交互方式后，使用鼠标拖拽模型，即可以符合人类直觉的形式自然交互。**此处“拖拽交互”形式是实验的创新点**。对三种交互方式，实现方式如下：

a) Translation：鼠标左键按下时，cast 一条 ray 与物体做 intersection test，得到交点。在鼠标左键按住的过程中，实时 cast 新的 ray，尝试将之前的交点以最短距离的方式移动到新的 ray 上。即，找到原交点在新的 ray 上的投影点，二者之差即可作为 translation。
b) Rotation：鼠标左键按下时记录指针的屏幕空间坐标，在鼠标左键按住的过程中，实时追踪指针的屏幕空间坐标。二者在 x，y 轴上的差距，映射到相机的 right 和 up 轴上。对物体的四元数做反向变换。
c) Scale：记录物体交点之差，令 scale 增量为差值即可。

实验总结：

通过本次实验，实现了一个 OpenGL 的渲染器，支持 3D Object 的读取和查看，并支持多种变形和多种视角。对 OpenGL 的理解加深，为后续复杂的图形学应用开发奠定了基础。