



计算机图形学期末大作业实验报告

雪中户外场景渲染与天气交互系统

Snowy Outdoor Scene Rendering & Weather Interaction System

简称：SOSRWIS

成员 _____

学号 _____

学院 _____ 计算机学院 _____

专业 _____ 计算机科学与技术 _____

2026 年 1 月 18 日



图 1: SOSRWIS 系统最终运行效果图

摘要

本项目设计并实现了一个基于 OpenGL 的户外雪景实时渲染与天气交互系统 (SOSRWIS)。用户可以在场景中自由移动，同时能够支持调节太阳位置实现不同时间场景的显示，以及选择开启不同级别的下雪效果。

该系统旨在搭建一个冬季的户外可交互场景，核心技术涵盖了**多光源冯氏光照模型 (Phong Lighting)**、**基于阴影贴图 (Shadow Mapping)**的实时动态阴影、以及基于**Billboard**技术的**粒子降雪系统**。在场景构建方面，系统通过 Assimp 库解析 glTF 模型，解决了复杂的节点变换与纹理适配问题，并结合天空盒与基于物理轨迹的动态光照模拟算法，实现了平滑的昼夜循环 (Day/Night Cycle)。

在工程实现上，本项目采用了模块化的架构设计，实现了 FPS/God Mode 双视角漫游、基于 AABB 的物理碰撞检测以及对象化的场景管理。报告具体说明了整个项目的实现思路以及功能实现过程，同时讨论了在开发过程中遇到的技术挑战。

最终运行结果表明，该系统能够在保证较高帧率的前提下，提供沉浸式的视觉体验与交互反馈。

关键词： OpenGL, 实时渲染, 阴影映射, 粒子系统, glTF 模型, 昼夜循环

目录

| | |
|---------------------------------------|-----------|
| 1 项目概述 | 1 |
| 1.1 项目背景 | 1 |
| 1.2 开发环境与工具 | 2 |
| 1.3 项目架构与工程化构建 | 2 |
| 1.3.1 目录结构组织 | 3 |
| 1.3.2 构建与自动化环境自检流程 | 3 |
| 2 核心渲染引擎搭建 | 4 |
| 2.1 摄像机与漫游系统 | 4 |
| 2.1.1 数学模型 | 4 |
| 2.1.2 双模式漫游交互 | 5 |
| 2.2 模型加载与数据处理 | 5 |
| 2.2.1 glTF 格式的适配与修复 | 5 |
| 2.2.2 纹理坐标适配与翻转控制 | 6 |
| 2.2.3 对象化的场景管理与批量渲染架构 | 6 |
| 3 光照与阴影技术 | 7 |
| 3.1 多光源 Phong 模型 | 7 |
| 3.1.1 基于物理衰减的点光源模型 | 7 |
| 3.1.2 着色器实现细节 | 8 |
| 3.2 动态昼夜循环系统 (SunSystem) | 10 |
| 3.2.1 太阳本体的几何生成算法 | 10 |
| 3.2.2 轨道模型与光色状态机 | 10 |
| 3.2.3 太阳外发光效果 (Glow Effect) | 11 |
| 3.3 实时阴影映射 (Shadow Mapping) | 12 |
| 3.3.1 阴影映射核心流程 | 12 |
| 3.3.2 阴影瑕疵优化措施 | 12 |
| 3.4 本部分小结 | 13 |
| 4 高级场景与特效 | 14 |
| 4.1 粒子降雪系统 | 14 |
| 4.1.1 雪花粒子的数据结构与生成机制 | 14 |
| 4.1.2 雪花粒子运动物理公式 | 15 |
| 4.1.3 公告板渲染与透明度测试 | 16 |
| 4.1.4 最终雪花效果 | 17 |
| 4.2 天空盒与纹理修复 | 17 |
| 4.2.1 立方体贴图构建 | 17 |

| | | |
|----------|--------------------------------------|-----------|
| 4.2.2 | 视觉无限远与深度优化 | 18 |
| 4.2.3 | 渲染状态管理与修复 | 18 |
| 4.2.4 | 天空盒最终效果 | 19 |
| 4.3 | 碰撞检测系统 | 19 |
| 4.3.1 | 算法原理与数据结构 | 20 |
| 4.3.2 | 位置预测与响应机制 | 20 |
| 4.3.3 | 空气墙效果 | 21 |
| 5 | 系统优化与性能分析 | 22 |
| 5.1 | 渲染性能优化 | 22 |
| 5.1.1 | 阴影贴图分辨率与带宽权衡 | 22 |
| 5.1.2 | 背面剔除策略 | 22 |
| 5.1.3 | 粒子系统渲染优化 | 23 |
| 5.2 | 资源管理架构 | 23 |
| 5.2.1 | 模型数据的复用机制 | 23 |
| 5.2.2 | 预分配与生命周期管理 | 24 |
| 6 | 遇到的主要问题与解决方案 | 24 |
| 7 | 总结与展望 | 24 |
| 7.1 | 项目成果总结 | 24 |
| 7.2 | 不足与改进方向 | 25 |
| 7.2.1 | 从 Phong 模型向 PBR 的演进 | 25 |
| 7.2.2 | 阴影技术的升级 (CSM) | 26 |
| 7.2.3 | 后处理特效 | 26 |
| 7.2.4 | 批量渲染 (Instanced Rendering) | 26 |
| 7.2.5 | 高级空间划分与剔除 | 26 |

小组分工情况

| 成员信息 | 具体分工与贡献 |
|-------|--|
| *** | 负责项目顶层架构设计与核心渲染管线搭建。 |
| ***** | <ul style="list-style-type: none">设计基于 CMake 的工程化构建系统，确立 Core/Renderer/Scene 的项目整体分层架构。实现 Camera 漫游系统 (FPS/God Mode) 及底层键鼠交互逻辑。负责主要场景构建，包括天空盒渲染、Assimp 模型加载适配及 glTF 格式修复等。 |
| *** | 负责动态环境光照与时间系统的算法实现。 |
| ***** | <ul style="list-style-type: none">搭建 SunSystem，基于三角函数模拟太阳/月亮的运行轨迹。实现动态昼夜循环逻辑，完成不同时段 (黎明/正午/深夜) 的光色插值。负责场景环境光 (Ambient) 与直射光的实时动态平衡调节。 |
| *** | 负责粒子特效系统与天气物理模拟。 |
| ***** | <ul style="list-style-type: none">基于 Billboard 技术实现高性能粒子渲染系统。编写粒子物理逻辑，模拟雪花受风力扰动飞舞及重力下落的随机性细节。设计天气状态机，实现了“小雪/中雪/暴雪”三级天气切换功能。 |

表 1: SOSRWIS 项目小组成员分工表

1 项目概述

1.1 项目背景

本项目旨在利用计算机图形学相关知识，构建一个温馨、真实的冬季户外村庄场景。用户可以以第一人称视角 (FPS) 或上帝视角 (God Mode) 在场景中漫游，体验昼夜交替的光影变化和下雪的天气氛围。

本项目基于现代计算机图形学理论与 OpenGL 可编程管线技术，构建了一个高保真的沉浸式冬季户外村庄以及天气交互系统。在自然场景的渲染中，涉及几何处理、光照计算及大气特效模拟等多个维度。同时以“雪夜村庄”为主题，重点挑战在实时渲染环境下对动态自然光照 (昼夜交替)、复杂几何场景 (glTF 模型) 以及天气粒子特效 (降雪) 的综合模拟。

1.2 开发环境与工具

本项目采用标准的工业界图形开发流程，主要软硬件环境配置如下：

- **开发 IDE:** Visual Studio 2022 (MSVC v143)
- **构建系统:** CMake (跨平台构建管理)
- **图形 API:** OpenGL 3.3 Core Profile
 - 采用可编程管线技术，完全摒弃了过时的立即渲染模式，确保了渲染效率与现代图形硬件的兼容性。
- **核心依赖库:**
 - GLFW: 窗口管理与上下文创建
 - GLAD: OpenGL 函数指针加载
 - GLM: 数学运算库 (矩阵/向量)
 - Assimp: 模型解析与加载
 - ImGui: 实时交互界面调试
- **版本控制与协作:** Git, GitHub
 - 项目采用 Feature Branch Workflow 工作流进行多人协作开发，所有代码均已开源托管至 GitHub¹。
- **实验硬件环境 (基准测试平台):**
 - CPU: Intel Core i5-13500H
 - GPU: Intel Iris Xe Graphics
 - RAM: 16GB
 - 注：针对集显设备的显存带宽限制，本项目在阴影贴图分辨率与粒子数量上进行了专门的性能调优。

1.3 项目架构与工程化构建

基于本次期末大作业是一个小组合作编写的项目，并且随着图形渲染功能的不断丰富，扁平化代码结构极易导致渲染逻辑与业务逻辑的高度耦合，造成代码难以维护。为了解决这一问题，本项目严格遵循软件工程原则，采用了模块化的分层架构设计。

我们将系统划分为核心基础设施层 (Core)、渲染抽象层 (Renderer) 以及场景逻辑层 (Scene)，实现了底层 API 调用与上层游戏逻辑的有效分离，为后续的多人协作开发奠定了坚实基础。

¹项目开源仓库地址: https://github.com/Miraitowa-XF/-_SOSRWIS-

1.3.1 目录结构组织

我们将核心引擎逻辑、渲染管线与具体场景逻辑分离，工程目录结构如下所示：

```
SOSRWIS/
|-- CMakeLists.txt          # 根构建脚本（依赖管理与编译配置）
|-- main.cpp                # 程序入口与主渲染循环（Main Loop）
|-- test.cpp                # 环境自检与单元测试模块
|-- src/
|   |-- Core/                # [核心层] 摄像机(Camera)、碰撞检测(AABB)、输入控制
|   |-- Renderer/            # [渲染层] 模型加载(Model)、网格(Mesh)、天空盒(Skybox)
|   |-- Scene/                # [场景层] 粒子系统、太阳系统(SunSystem)、对象管理
|-- assets/
    |-- models/              # 3D模型库（包含 .gltf 模型及配套 .bin 等数据）
    |-- textures/             # 纹理库（雪花贴图、天空盒贴图等）
    |-- shaders/              # [核心算法] GLSL 着色器脚本
```

1.3.2 构建与自动化环境自检流程

为了确保开发环境的稳定性与跨平台兼容性，本项目在构建流程中设计了“自动化环境自检”代码程序。方便使用者一键检测自身环境是否都已配置成功。

1. 构建系统设计 项目使用 CMake 进行跨平台构建管理。通过 Target 分离技术，我们将核心业务逻辑（src/）与测试逻辑（test.cpp）解耦。`'glMain'` 负责主渲染循环，而 `'glTest'` 作为一个独立的轻量级可执行文件，用于在不启动图形窗口的情况下快速验证底层依赖库的链接状态。

2. 单元测试与环境自检 我们在 `test.cpp` 中实现了一套标准化的自检流程，涵盖了从数学库运算到资源路径检查的各个环节。核心测试逻辑包括：

- **C++17 文件系统检查**: 验证 `std::filesystem` 是否能正确识别 `assets/` 资源目录，防止因工作路径（Working Directory）配置错误导致的资源加载失败。
- **GLM 数学库验证**: 执行一次矩阵变换运算（位移 + 向量乘法），验证 GLM 库是否正确链接且计算结果符合预期。
- **Assimp 链接测试**: 尝试实例化 `Assimp::Importer` 对象，验证静态库是否正确链接，排除符号未定义错误。
- **ImGui 上下文初始化**: 验证 UI 库的上下文创建与后端绑定流程。

3. 运行结果验证 当所有依赖库配置正确时，运行 'glTest.exe'，控制台将显示所有核心模块（GLM, Assimp, GLFW, GLAD）均通过测试，且 ImGui 调试窗口正常渲染，（如下图 2 所示）。这一机制极大地降低了多人协作时的环境配置排错成本。

The screenshot shows a terminal window titled 'F:\OpenGL_project\SOSRWIS\ >' and a separate ImGui window titled 'SOSRWIS - All Libs Test'. The terminal output displays a series of '[OK]' messages indicating successful tests for CoreLib, Assets, GLM, Assimp, stb_image, GLFW, GLAD, and ImGui. The ImGui window shows a message: 'If you see this, ImGui is working! Application average 0.648 ms/frame (1543.2 FPS)'. It also has a 'Close App' button.

```

=====
SOSRWIS 全面系统自检启动 =====

[ OK ] CoreLib (Src): add(100, 200) = 300
[ OK ] Assets: 检测到 assets 文件夹
    -> models 目录存在
[ OK ] GLM: 矩阵变换计算正确
[ OK ] Assimp: Importer 实例化成功 (静态链接正常)
[ OK ] stb_image: 函数链接正常 (测试加载空文件返回 NULL)
[ OK ] GLFW: 窗口创建成功
[ OK ] GLAD: OpenGL Version: 3.3.0 - Build 31.0.101.4255
[ OK ] ImGui: UI 上下文初始化完成

===== 所有库检测完毕, 进入渲染循环 =====

```

图 2: 环境自检程序 (glTest) 的成功运行结果

2 核心渲染引擎搭建

这一节详细阐述了 SOSRWIS 系统的底层渲染架构，包括基于欧拉角的摄像机系统、模型加载管线以及着色器管理机制。

2.1 摄像机与漫游系统

为了实现灵活的场景漫游，我们封装了 Camera 类。摄像机的核心是通过位置向量、前向向量和上向量构建观察矩阵 [1]。

2.1.1 数学模型

系统通过欧拉角（偏航角 ψ 和俯仰角 θ ）来计算摄像机的方向向量 D_{front} ：

$$D_{front} = \begin{bmatrix} \cos(\psi) \cos(\theta) \\ \sin(\theta) \\ \sin(\psi) \cos(\theta) \end{bmatrix} \quad (1)$$

基于计算出的方向向量，我们利用 Gram-Schmidt 正交化过程计算右向量和上向量，最终构建 LookAt 矩阵将世界坐标转换为观察空间坐标。

2.1.2 双模式漫游交互

为了兼顾沉浸感与调试便利性，系统实现了两种漫游模式的无缝切换，这里我们是具体通过 TAB 键来实现两种模式的自由切换：

- **FPS 模式 (地面漫游)**：模拟重力效果，摄像机的高度被锁定在地面上方（这里设置的眼睛高度为 $y = 3.0f$ ），移动向量被限制在 XZ 平面，适用于模拟真实的步行体验。
- **God Mode (上帝视角)**：解除重力与碰撞限制，摄像机可以在三维空间内自由飞行，便于观察场景整体布局。

2.2 模型加载与数据处理

对于整个场景中的模型，我们没有采用从轮子开始造车的方式一个个模型的绘制，对于大多数模型我们是从 Sketchfab 网站中直接下载发布的成品模型，然后在项目中通过引入 Assimp 库来处理复杂的 3D 模型格式（这里具体是处理.gltf 类型的模型文件 [2]）。

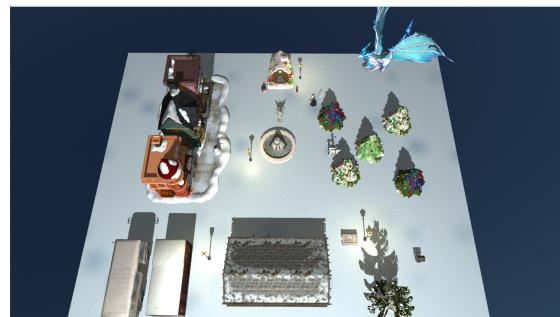
2.2.1 glTF 格式的适配与修复

相较于传统的 OBJ 格式，现代 glTF 格式通常采用复杂的节点层级来组织网格。在开发初期，我们直接加载 glTF 模型导致出现“零件散落”的现象（即子节点的局部变换矩阵未应用到顶点数据中），如图 3(a) 所示。

为了解决这一问题，我们在 Assimp 加载阶段启用了 `aiProcess_PreTransformVertices` 标志位。该指令会强制在加载阶段遍历场景图，将各级节点的变换矩阵直接“烘焙”到顶点坐标中，从而正确还原模型的几何形态（见图 3(b)）。



(a) 直接加载导致的节点层级丢失



(b) 应用预变换后的正确模型

图 3: glTF 模型加载问题的排查与修复对比。

2.2.2 纹理坐标适配与翻转控制

由于 OpenGL 纹理坐标系原点位于左下角，而常规图像数据（如 PNG/JPG）存储原点位于左上角，直接加载会导致贴图倒置。为此，我们在底层实现了智能的翻转控制策略：

- 针对传统 .obj 模型，启用 `stb_image` 的垂直翻转功能进行 Y 轴校正；
- 针对现代 .gltf 模型及天空盒，因其标准已预定义坐标方向，则显式关闭翻转。

该策略有效解决了混合格式加载时的纹理错位问题。

2.2.3 对象化的场景管理与批量渲染架构

随着场景内模型数量增加至 20 余个（包括房屋、植被、角色等），早期开发中采用的“硬编码渲染”模式（即在主循环中为每个物体单独手动设置矩阵并调用 Draw 函数）导致代码冗余度极高且难以维护。

为了解决这一问题，我们重构了渲染管线，引入了**对象化数据驱动**的设计思路 [3]：

1. **数据与状态分离**（享元模式思想 [4]）：我们定义了 `SceneObject` 结构体。其中，`Model*` 指针指向重型的几何与纹理数据（只加载一次，驻留内存），而 `position`, `rotation`, `scale` 等轻量级属性则存储每个实例的独特状态。这种设计极大地降低了内存开销，如图 4(a) 所示
2. **容器化批量管理**：引入 `std::vector<SceneObject>` 容器来统一管理场景中所有的静态与动态物体。场景的构建过程从编写逻辑代码转变为“配置数据列表”，如图 4(b) 所示。
3. **统一渲染循环**：重构后的渲染循环不再包含具体的物体逻辑，而是简单地遍历容器。这使得新增一个模型只需一行 `push_back` 代码，无需修改渲染核心逻辑，极大地提升了系统的可扩展性。

```

1 // 1. 遍历并绘制列表中的所有物体
2 for (const auto& obj : objects)
3 {
4     // 初始化模型矩阵为单位矩阵
5     glm::mat4 model = glm::mat4(1.0f);
6
7     // [平移] 将物体移动到指定的世界坐标位置
8     model = glm::translate(model, obj.position);
9
10    // [旋转] 根据旋转轴和角度进行旋转
11    model = glm::rotate(model,

```

```

12     glm::radians(obj.rotationAngle), obj.rotationAxis);
13
14     // [缩放] 调整物体的大小
15     model = glm::scale(model, obj.scale);
16
17     // 计算好的模型矩阵传递给 Shader 渲染提交
18     shader.setMat4("model", model);
19
20     // 调用模型自身的绘制函数
21     obj.model->Draw(shader);
22 }
```

Listing 1: 统一渲染绘制循环

```

// 定义一个结构体, 用来管理场景里的每一个物体
struct SceneObject {
    Model* model; // 模型指针
    glm::vec3 position; // 位置
    glm::vec3 scale; // 缩放
    float rotationAngle; // 旋转角度 (度)
    glm::vec3 rotationAxis; // 旋转轴

    SceneObject(Model* m, glm::vec3 pos, glm::vec3 s, float rot, glm::vec3 axis) {
        : model(m), position(pos), scale(s), rotationAngle(rot), rotationAxis(axis) {
    }
};

// 存储所有场景对象的列表
std::vector<SceneObject> allObjects;
```

(a) 对象数据化结构体 SceneObject

(b) 配置数据列表

图 4: 对象化管理与批量渲染架构

3 光照与阴影技术

为了营造温馨且真实的冬季村庄氛围，我们构建了一套动态、多层次的光照渲染系统。该系统集成了基于物理衰减的多光源模型、动态昼夜循环逻辑以及高质量实时阴影。

3.1 多光源 Phong 模型

本项目采用了改进的 Phong 反射模型，在 `basic.frag` 中实现了定向光（天体光源）与多盏点光源（路灯）的线性叠加。

3.1.1 基于物理衰减的点光源模型

为了模拟村庄道路两侧的照明效果，我们定义了结构体 `PointLight` 并通过 Uniform 数组传递 4 盏路灯的数据。

```

1 #define NR_POINT_LIGHTS 4 // 定义路灯数量：4盏
2 struct PointLight {
3     vec3 position;
4     vec3 color;
5
6     // 衰减系数
7     float constant;
8     float linear;
9     float quadratic;
10 };
11 uniform PointLight pointLights[NR_POINT_LIGHTS]; // 数组

```

Listing 2: 点光源结构体定义

衰减模型 每盏路灯对片段颜色 C_{final} 的贡献遵循以下物理公式 [5]:

$$C_{point} = \frac{1}{K_c + K_l \cdot d + K_q \cdot d^2} \times (I_{diff} + I_{spec}) \quad (2)$$

其中：

- d : 片段到光源的距离
- K_c, K_l, K_q : 分别为常数项、线性项和二次项衰减系数
- I_{diff}, I_{spec} : 漫反射和镜面反射强度

3.1.2 着色器实现细节

点光源计算函数 在片段着色器中，我们通过 `CalcPointLight` 函数遍历累加各点光源贡献。

```

1 // basic.frag 中实现的点光源贡献累计
2 vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos,
3     vec3 viewDir, vec3 objectColor)
4 {
5     vec3 lightDir = normalize(light.position - fragPos);
6
7     // 漫反射
8     float diff = max(dot(normal, lightDir), 0.0);
9     // 镜面反射
10    vec3 reflectDir = reflect(-lightDir, normal);
11    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);

```

```

12 // 衰减
13 float distance = length(light.position - fragPos);
14 float attenuation = 1.0 / (light.constant
15                         + light.linear * distance
16                         + light.quadratic * (distance *
17                         distance));
18
19 // 合并
20 vec3 diffuse = light.color * diff * objectColor;
21 vec3 specular = light.color * spec * 0.5; // 0.5是镜面强度
22
23 diffuse *= attenuation;
24 specular *= attenuation;
25
26 return (diffuse + specular);
}

```

Listing 3: 点光源计算函数

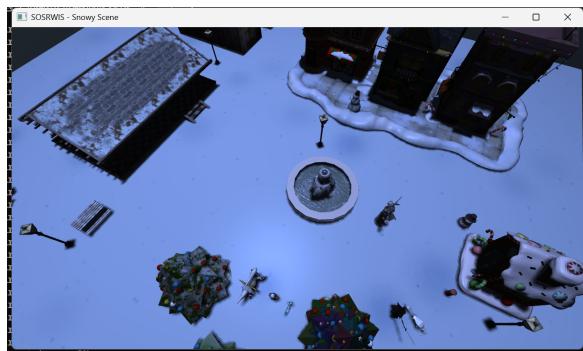
透明纹理处理 为了正确处理植被、围栏等带透明通道的纹理，我们引入了 Alpha 测试机制：

```

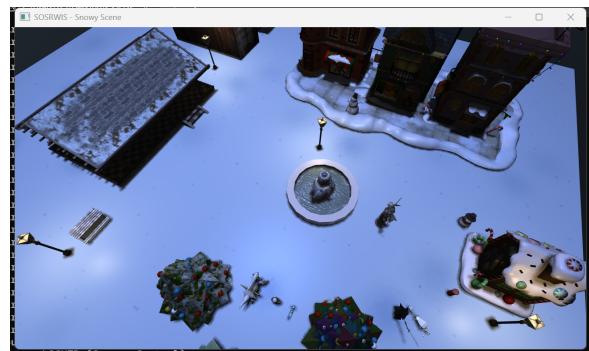
1 // basic.frag 处理纹理透明度
2 vec4 textColor = texture(texture_diffusel, TexCoordds);
3 if(textColor.a < 0.01) discard; // 丢弃透明像素
4 // 防止干扰阴影和光照计算

```

Listing 4: Alpha 测试处理透明像素



(a) 路灯关闭（仅环境光与月光）



(b) 路灯开启（点光源叠加效果）

图 5: 村庄路灯系统开启前后的光照效果对比

3.2 动态昼夜循环系统 (SunSystem)

SunSystem 模块通过模拟天体运行轨迹，实现了从黎明、正午、黄昏到深夜的实时平滑过渡。

3.2.1 太阳本体的几何生成算法

为了在天空中渲染太阳实体，我们利用 **UV Sphere 算法** [6] 动态生成一个气体几何网格。给定经度细分数 M 和纬度细分数 N ，球面上的任意一点的坐标 (x, y, z) 可通过球面坐标系参数方程计算：

$$\begin{cases} x = R \cdot \cos(\phi) \cdot \sin(\theta) & \theta \in [0, \pi] \text{(纬度角)} , \\ y = R \cdot \cos(\theta) & \phi \in [0, 2\pi] \text{(经度角)} , \\ z = R \cdot \sin(\phi) \cdot \sin(\theta) & R \text{(单位半径)} . \end{cases} \quad (3)$$

我们在 SunSystem::Init 中动态构建顶点缓冲 (VBO) 和索引缓冲 (EBO)，网格密度配置为 $M = 32$ 、 $N = 32$ ，确保了天体形状的圆滑度。

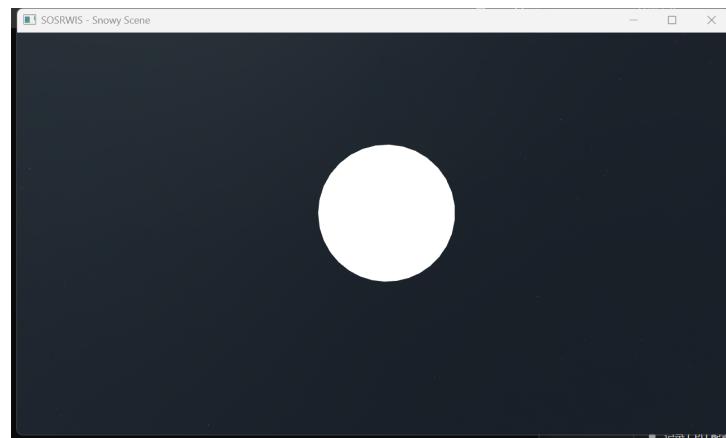


图 6: 太阳本体渲染效果

3.2.2 轨道模型与光色状态机

太阳高度 h 由简谐运动模型模拟。

$$h(t) = A \cdot \sin\left(\frac{2\pi}{T}t + \phi_0\right), \quad A = 1.0, \quad T = 240\text{s}, \quad \phi_0 = -\frac{\pi}{2}. \quad (4)$$

基于高度 h 的光照状态机设计如下：

表 2: 昼夜循环四阶段状态机

| 阶段 | 高度范围 | 光色 | 环境光 | 、 textbf 描述 |
|------|-----------------|-----------------------|---------|----------------------------|
| 正午模式 | $h > 0.1$ | 暖白 (1.0, 0.95, 0.9) | 0.8 | 光照最强，色温 5500K |
| 晨昏过渡 | $[-0.15, 0.1]$ | 橙红 \rightarrow 暗蓝插值 | 0.3–0.7 | <code>glm::mix</code> 线性插值 |
| 至暗时刻 | $[-0.3, -0.15]$ | 深蓝 (0.1, 0.1, 0.3) | 0.1 | 环境光最低，营造压抑感 |
| 月夜模式 | $h < -0.3$ | 冷蓝 (0.4, 0.5, 0.9) | 0.2 | 太阳向量取反作月亮光源 |

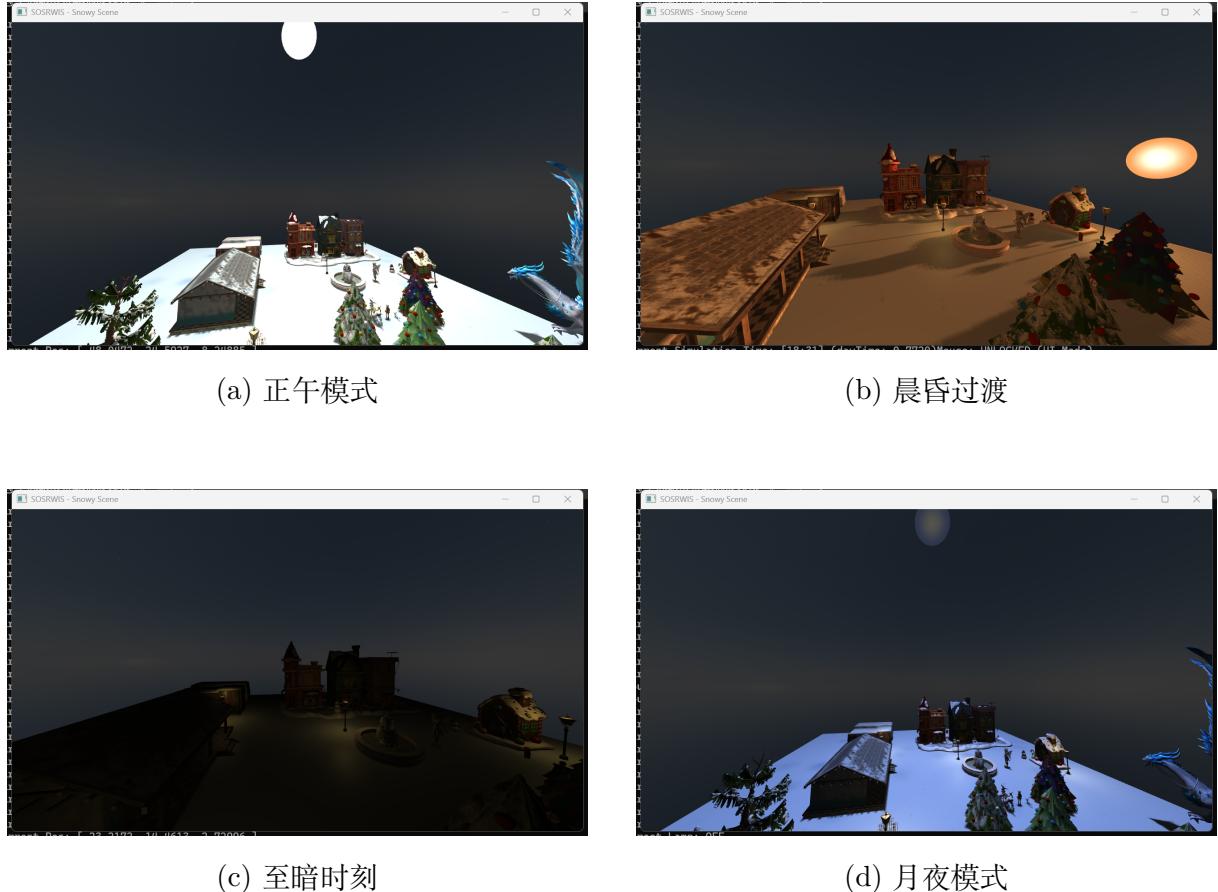


图 7: 动态昼夜循环系统的四阶段视觉表现对比

3.2.3 太阳外发光效果 (Glow Effect)

在 `sun.frag` 片段着色器中，我们利用视线向量 \vec{v} 与球体法线 \vec{n} 的点积实现径向渐变发光 [7]：

$$I_{\text{glow}} = (\vec{n} \cdot \vec{v})^\gamma, \quad \gamma = 4.0, \quad (5)$$

$$C_{\text{final}} = C_{\text{base}} + I_{\text{glow}} \cdot C_{\text{glow}}. \quad (6)$$

其中 C_{base} 为基色（随状态机变化）， $C_{\text{glow}} = (1.2, 1.1, 1.0)$ 为高光叠加色。该设计使太阳中心呈现亮白过曝，边缘保持物理色温，增强体积感。

3.3 实时阴影映射 (Shadow Mapping)

为了实现大场景下高质量的动态阴影，我们采用基于深度贴图双通行渲染（Two-pass Rendering）技术。在保证实时性的同时，结合多种优化手段解决阴影渲染中的常见视觉瑕疵。

3.3.1 阴影映射核心流程

阴影映射 [8] 的核心是将片段从世界空间变换到光源视角空间进行深度比较。算法流程如下：

步骤 1：深度图生成阶段：以光源为视点渲染场景，将最近深度存储到深度纹理中。

步骤 2：阴影渲染阶段：对每个片段计算其在光源裁剪空间中的坐标：

$$P_{\text{light}} = M_{\text{proj}} \cdot M_{\text{view}} \cdot P_{\text{world}} \quad (7)$$

其中 M_{view} 为光源视图矩阵， M_{proj} 为光源投影矩阵。

步骤 3：深度比较：执行透视线除法并将坐标归一化至 $[0, 1]$ 范围，与深度图采样值进行比较：

$$\text{shadow} = \begin{cases} 0, & \text{当前深度} \leq \text{深度图采样值} + \text{bias} \\ 1, & \text{其他} \end{cases} \quad (8)$$

3.3.2 阴影瑕疵优化措施

针对阴影渲染中的常见问题，我们实施了一系列优化措施。

阴影痤疮 (Shadow Acne) 消除 深度图的数值精度限制会导致自遮挡条纹伪影。我们采用动态偏置技术 [9] 解决此问题：

$$\text{Bias} = \max \left(\alpha \cdot (1 - \vec{n} \cdot \vec{l}), \beta \right) \quad (9)$$

其中 \vec{n} 为表面法线， \vec{l} 为光照方向， $\alpha = 0.005$ 控制斜率相关性， $\beta = 0.0005$ 为最小偏置值。该公式确保陡峭表面获得更大偏置，有效消除伪影。

边缘柔化 (PCF 滤波) 硬阴影的锯齿状边缘通过百分比渐近过滤改善 [10]。在片段着色器中实现 3×3 核的加权采样：

$$\text{shadow}_{\text{soft}} = \frac{1}{9} \sum_{i=-1}^1 \sum_{j=-1}^1 \text{PCF}(P_{\text{light}} + (\Delta x_i, \Delta y_j)) \quad (10)$$

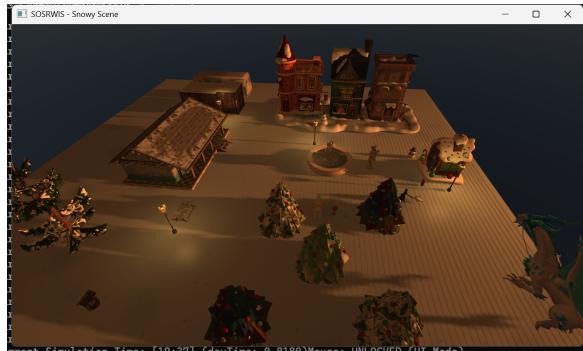
其中 $\Delta x, \Delta y$ 为纹理坐标偏移量。此方法能够在雪地等表面上产生更自然的柔和阴影过渡。

正面剔除优化 深度图生成阶段启用正面剔除 [11]，避免自遮挡伪影：

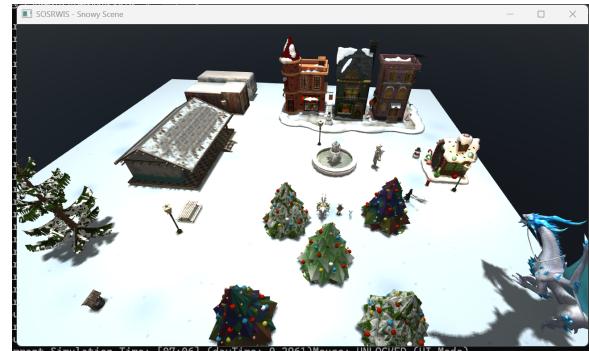
```
glCullFace(GL_FRONT); // 仅渲染背面
renderSceneToDepthTexture();
glCullFace(GL_BACK); // 恢复默认
```

Listing 5: 深度图渲染配置

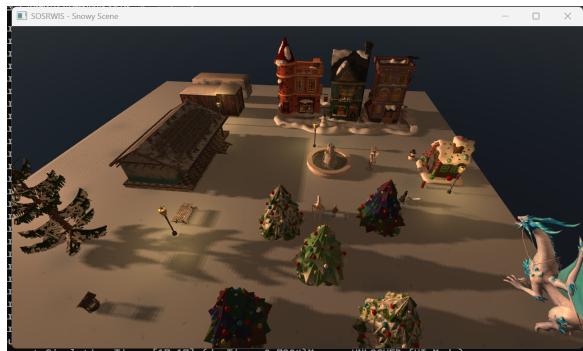
该方法利用物体背面深度生成阴影，能够有效提升算法的鲁棒性。



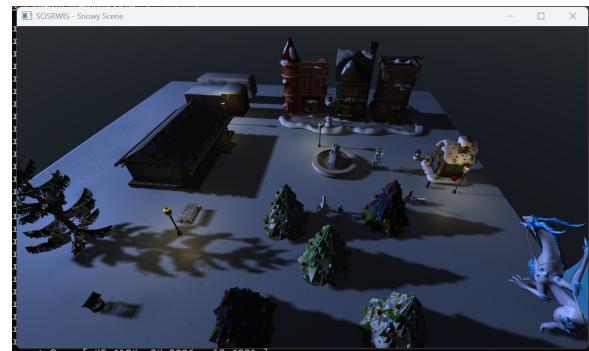
(a) 晨间阴影（斜射长阴影）



(b) 午间阴影（直射短促锐利）



(c) 傍晚阴影（斜射长阴影）



(d) 夜间阴影（斜射长阴影）

图 8: 实时阴影映射在不同时段与光源下的视觉表现

3.4 本部分小结

本章通过数理几何构建天体、基于物理公式模拟光照衰减，并利用深度图技术实现阴影渲染。最终的光照系统能够支持 4 盏路灯与 1 个动态天体光的实时交互，并能够呈现稳定清晰的画面，实现了高质量的视觉输出。

4 高级场景与特效

4.1 粒子降雪系统

为了增强场景的沉浸感，我们开发了 `ParticleSystem` 模块，利用大量独立的四边形纹理（Quad）模拟真实的降雪效果。该系统集成了物理运动模拟、生命周期管理以及基于视角的公告板（Billboarding）渲染技术。降雪系统一共三个下雪场景：小雪、中雪、大雪

4.1.1 雪花粒子的数据结构与生成机制

单个雪花粒子由 `SnowParticle` 结构体定义，包含位置 (Position)、速度 (velocity)、生命周期 (lifetime)、大小 (size)、以及用于模拟空气湍流的相位参数 (phase、swaySpeed) 等。

为了高效管理数千个粒子，我们使用 `std::vector` 作为容器，并采用 **Swap-and-Pop** 策略移除消亡粒子，将时间复杂度从 $O(N)$ 降低至 $O(1)$ 。

粒子的生成受到累加器 `spawnAccumulator` 的控制，确保在不同帧率下生成的粒子数量 ($R_{spawn} = 400/s \quad 800/s \quad 1600/s$) 保持恒定。具体生成、更新函数如下

```

1 void ParticleSystem::SpawnParticle() {
2     SnowParticle p;
3     //下雪范围: x: -50~50, y: 25~80, z: -50~50
4     p.position = glm::vec3(
5         glm::linearRand(-50.0f, 50.0f),
6         glm::linearRand(25.0f, 80.0f),
7         glm::linearRand(-50.0f, 50.0f)
8     );
9     //初始下落速度
10    p.velocity = glm::vec3(
11        wind.x + glm::linearRand(-0.2f, 0.2f),
12        glm::linearRand(-0.5f, -1.0f),
13        wind.z + glm::linearRand(-0.2f, 0.2f)
14    );
15
16    p.lifetime = glm::linearRand(8.0f, 18.0f);
17    p.size = glm::linearRand(0.1f, 0.2f);
18
19    p.phase = glm::linearRand(0.0f, 6.2831f);
20    p.swaySpeed = glm::linearRand(0.5f, 1.5f);
21    p.angle = glm::linearRand(0.0f, 6.2831f);
22    p.angularSpeed = glm::linearRand(-1.0f, 1.0f);

```

```
23
24     particles.push_back(p);
25 }
26
27 void ParticleSystem::Update(float deltaTime, bool smallSnow) {
28     if (!active) return;
29
30     spawnAccumulator += deltaTime * spawnRate; //累加器
31     //按速率精确生成新粒子
32     while (spawnAccumulator >= 1.0f) {
33         SpawnParticle();
34         spawnAccumulator -= 1.0f;
35     }
36
37     //倒序更新，以便删除粒子
38     for (int i = (int)particles.size() - 1; i >= 0; --i) {
39         auto& p = particles[i];
40         //更新粒子p如下参数： lifetime、 velocity、 position
41
42         if (p.lifetime <= 0.0f || p.position.y <= -1.0f) {
43             std::swap(p, particles.back());
44             particles.pop_back();
45             //particles.erase(particles.begin() + i);
46         }
47         else {
48             //更新粒子p如下参数： position，模拟雪花在空气中飘舞的轻盈感
49         }
50     }
51 }
```

Listing 6: 雪花例子生成、更新函数

4.1.2 雪花粒子运动物理公式

雪花的下落并非简单的自由落体，而是受到重力、风力以及空气阻力导致的摆动影响。在 Update 阶段，我们采用半隐式欧拉积分计算粒子轨迹。同时，我们将下雪的场景分为了小雪、中雪、大雪；在小雪场景中，为了衬托小雪时的宁静，雪花运动不受风力的影响。具体公式如下：

1. 基础运动与重力: 设重力加速度 $G = -0.8$, 风力向量为 \vec{W} , 粒子速度 \vec{v} 更新如下:

$$\vec{v}_y(t + \Delta t) = \vec{v}_y(t) + G \cdot \Delta t \quad (11)$$

$$\mathbf{P}(t + \Delta t) = \mathbf{P}(t) + (\vec{v}(t) + \vec{W}) \cdot \Delta t \quad (12)$$

2. 三角函数扰动 (Swaying): 为了模拟雪花在空气中飘舞的轻盈感, 我们在水平方向叠加了基于正弦波的时间扰动。每个粒子拥有独立的相位 ϕ 和摆动速度 ω , 粒子位置偏移量计算如下: A_x, A_z 为摆动幅度

$$\begin{cases} \Delta x = A_x \cdot \sin(t \cdot \omega + \phi), & A_x = 0.06 \\ \Delta z = A_z \cdot \cos(t \cdot \omega + \phi), & A_z = 0.03 \end{cases} \quad (13)$$

这种非线性的位移叠加有效地消除了粒子运动的机械感。

4.1.3 公告板渲染与透明度测试

为了使二维的雪花纹理在三维空间中始终朝向观察者, 我们实现了 **Billboarding** 技术。在 `Render` 函数构建模型矩阵 (Model Matrix) 时, 我们将视图矩阵 (View Matrix) 的转置矩阵的左上 3×3 部分赋值给模型矩阵, 消除了旋转差异:

$$M_{model}[i][j] = M_{view}[i][j], \quad i, j \in \{0, 1, 2\} \quad (14)$$

此外, 在片段着色器 `particle.frag` 中, 为了解决纹理边缘的矩形伪影, 我们结合了混合模式 (Blending) 与 Alpha 测试:

```
// Fragment Shader Alpha Test
vec4 texColor = texture(particleTexture, TexCoords);
if(texColor.a < 0.2)
    discard; // 剔除透明度过低的像素
```

4.1.4 最终雪花效果

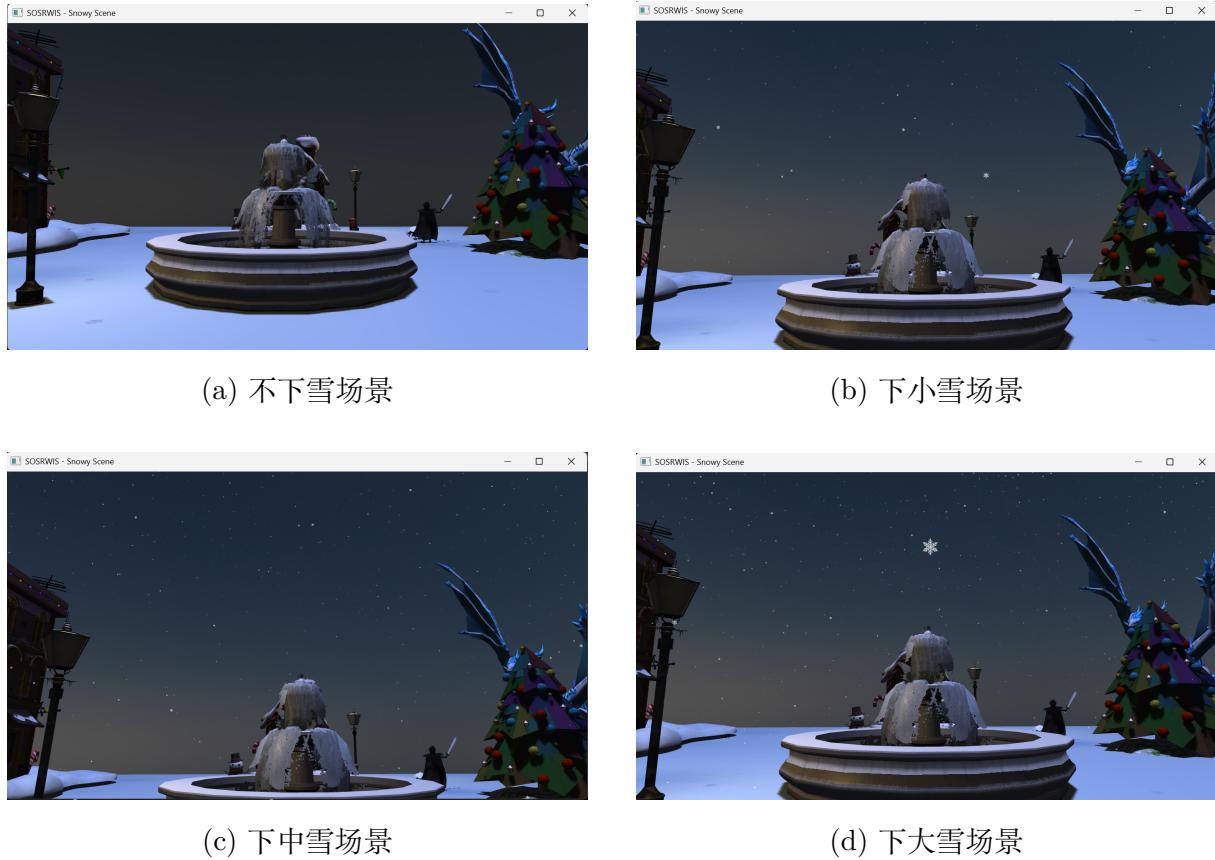


图 9: 不同下雪场景展示

由于图片传递信息效果有限，不能很好体现出雪花在空气中的飘舞视觉效果，更佳效果请查看视频。

4.2 天空盒与纹理修复

为了构建具有深邃感的环境背景，我们采用了立方体贴图 (CubeMap) 技术实现天空盒。该模块不仅负责渲染背景，还集成了针对 OpenGL 渲染状态的自动化管理与修复机制，以确保与场景中其他物体的渲染互不冲突。在加载过程中，我们发现开启全局纹理翻转会导致天空盒接缝错乱。因此在加载天空盒纹理前，我们显式关闭了 `stb_image` 的垂直翻转功能。

4.2.1 立方体贴图构建

天空盒本质上是一个包裹摄像机的巨大立方体。我们通过 `loadCubemap` 函数加载 6 张连续的环境纹理，利用 `GL_TEXTURE_CUBE_MAP` 目标将其映射到立方体的六个面上。

为了消除天空盒纹理拼接处可能出现的黑色接缝 (Seams)，我们将纹理环绕方式强制设为 `GL_CLAMP_TO_EDGE`，防止纹理坐标插值时采样到边缘以外的颜色：

```
// Skybox.cpp: 防止纹理边缘插值产生接缝
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

4.2.2 视觉无限远与深度优化

为了模拟天空“无限远”的视觉效果，我们在渲染时移除了视图矩阵 (View Matrix) 中的位移分量，仅保留旋转分量。这样无论摄像机如何移动，天空盒始终保持相对静止，只有旋转视角时背景才会变化。

$$M_{view_no_trans} = \begin{bmatrix} R_{3 \times 3} & 0 \\ 0 & 1 \end{bmatrix} \leftarrow \text{glm::mat4(glm::mat3(view))} \quad (15)$$

在性能优化方面，我们将天空盒的渲染置于渲染循环的最后阶段，并将深度测试函数修改为 `GL_EQUAL`。这意味着天空盒仅在深度缓冲区值为 1.0（最大深度）的像素上进行绘制，从而利用 Early-Z 测试自动剔除被场景物体遮挡的部分，大幅减少片段着色器的开销。

```
// Skybox.cpp: 深度测试优化与视图矩阵剥离
glDepthFunc(GL_EQUAL); // 允许在深度为 1.0 处绘制
glm::mat4 viewNoTrans = glm::mat4(glm::mat3(view)); // 移除位移
skyboxShader->setMat4("view", viewNoTrans);
```

4.2.3 渲染状态管理与修复

由于天空盒是从内部观察的封闭立方体，且为了在最后阶段渲染，我们修改了全局深度函数和面剔除 (Face Culling) 设置。为了防止这些修改污染后续帧或半透明物体的渲染 (如 `SnowScene`)，我们在 `Draw` 函数结束前必须执行“状态修复”：

1. **深度函数复位**：将 `GL_EQUAL` 恢复为默认的 `GL_LESS`。

2. **面剔除复位**：天空盒绘制时为了确保内部可见可能关闭了剔除，绘制结束后必须重新启用 `GL_CULL_FACE`，否则会导致场景中其他模型（如房屋、地面）的渲染出现异常。

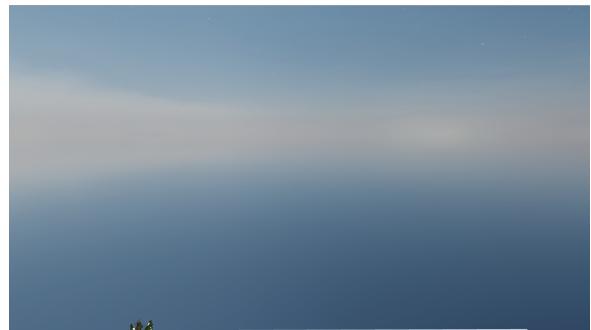
```
// Skybox.cpp: 关键的状态修复步骤
 glBindVertexArray(0);
 glDepthFunc(GL_LESS); // 恢复默认深度测试
 glEnable(GL_CULL_FACE); // 【关键修复】恢复面剔除，防止模型渲染错误
```

此外，在 `main.cpp` 中，我们还引入了动态亮度参数 `skyBrightness`，将天空盒与 `SunSystem` 的昼夜循环强度挂钩，防止在“至暗时刻”天空盒过于死黑破坏沉浸感。

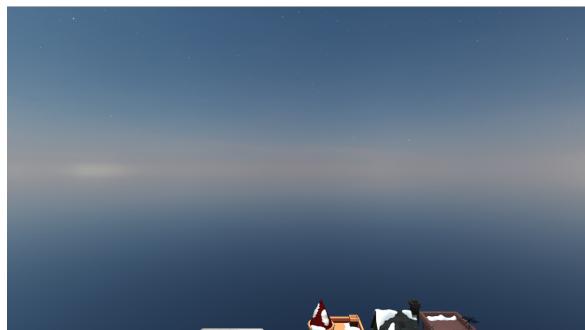
4.2.4 天空盒最终效果



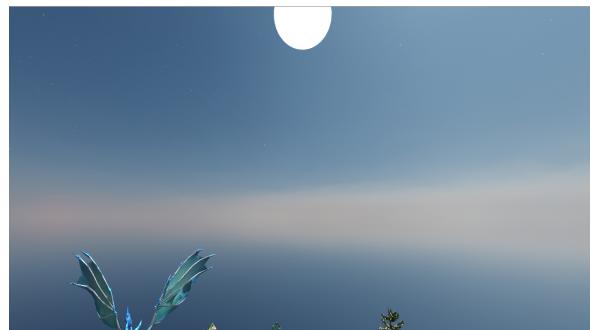
(a) front



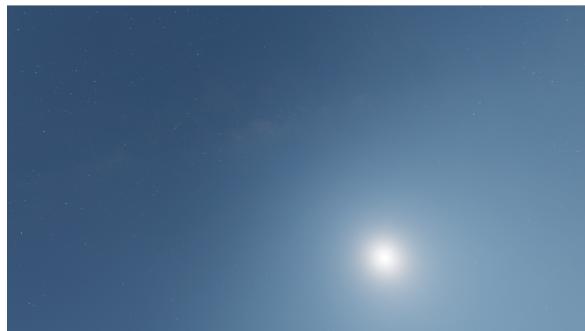
(b) back



(c) left



(d) right



(e) top



(f) bottom

图 10: 不同方位天空盒图片展示

通过上述图片，我们可以看到天空盒的边缘之间没有明显的黑缝 (Seams)，边缘之间衔接效果良好且连续。

4.3 碰撞检测系统

碰撞检测采用 **AABB** (轴对齐包围盒) 算法。为了平衡性能与游戏性，我们采用了“手动定义空气墙”的策略，即在房屋、水井等不可通行区域手动放置不可见的 AABB

盒子。核心碰撞逻辑如 Listing 7 所示。

4.3.1 算法原理与数据结构

我们定义了一个名为 AABB 的结构体，仅需存储包围盒的最小坐标点 (\vec{P}_{min}) 和最大坐标点 (\vec{P}_{max}) 即可确立空间体积。

两个 AABB (A 和 B) 发生碰撞的充要条件是它们在 X、Y、Z 三个轴上的投影均存在重叠。该算法极其高效，仅涉及简单的浮点数比较，非常适合实时渲染循环中的频繁调用。判定逻辑如下代码所示：

```

1 // Collision.h
2 struct AABB {
3     glm::vec3 min, max;
4     // 检测是否重叠：三个轴向均需满足 min <= other.max && max >=
5     // other.min
6     bool checkCollision(const AABB& other) const {
7         return (min.x <= other.max.x && max.x >= other.min.x) &&
8             (min.y <= other.max.y && max.y >= other.min.y) &&
9             (min.z <= other.max.z && max.z >= other.min.z);
10    }
11 };

```

Listing 7: AABB 碰撞核心判定逻辑

4.3.2 位置预测与响应机制

在每帧处理输入 (processInput) 时，系统并不直接修改摄像机坐标，而是采用“**位置预测**”策略：

1. **构建空气墙集合**：在程序初始化阶段，我们在房屋、水井及地图边界处硬编码了一组不可见的静态 AABB 对象 (walls)。
2. **构建玩家包围盒**：基于当前的键盘输入计算出玩家的预期位置 (nextPos)，并以此为中心构建一个微小的动态 AABB (模拟玩家躯干)。
3. **碰撞裁决**：遍历所有静态空气墙，一旦玩家包围盒与任意墙体发生重叠，即判定为碰撞，**放弃**更新位置；反之则应用位移。

这种“拒绝移动”的响应方式简单有效，避免了复杂的物理回弹计算，同时保证了角色不会穿模进入物体内部。

```

// main.cpp: 预测位置与碰撞检测循环
// 定义玩家的身体大小 (0.3宽, 1.8高)

```

```

glm::vec3 playerHalfSize(0.3f, 0.9f, 0.3f);
AABB playerBox(camera.Position - playerHalfSize,
                camera.Position + playerHalfSize);

bool hit = false;
for (const auto& box : sceneColliders) {
    if (playerBox.checkCollision(box)) {
        hit = true;
        break;
    }
}

// 碰撞回退机制 (main.cpp)
if (hit) {
    // 检测到碰撞，强制回退到上一帧位置
    camera.Position = oldPosition;
}

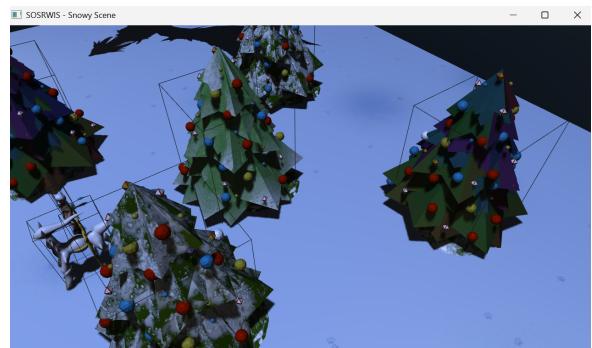
```

4.3.3 空气墙效果

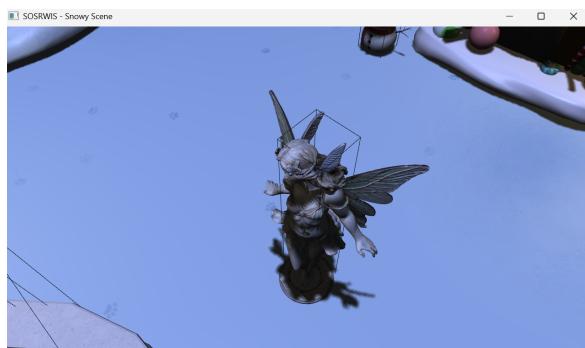
按 F1 键可视化物品的空气墙，以下是空气墙的效果：



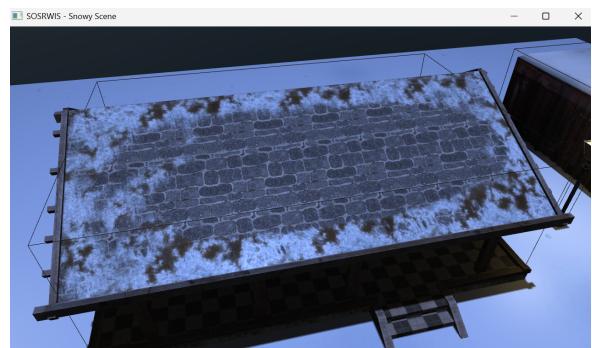
(a) 喷泉空气墙



(b) 树的空气墙



(c) 雕像空气墙



(d) 房子空气墙

图 11: 物品空气墙展示

5 系统优化与性能分析

针对本项目开发环境 (Intel Iris Xe 集成显卡, 共享系统内存) 的硬件特性, 我们在渲染管线和资源管理层面进行了深度的性能调优 [1], 以平衡视觉质量与运行帧率。

5.1 渲染性能优化

5.1.1 阴影贴图分辨率与带宽权衡

在实时阴影映射的实现初期, 我们采用了 4096×4096 的高分辨率深度贴图以追求极致的阴影锐利度, 如图 12(a) 所示。然而在集显设备上, 这导致了严重的性能瓶颈:

1. **显存带宽耗尽**: 集显使用系统内存作为显存, 高分辨率纹理的频繁读写导致带宽饱和, 引发系统卡顿甚至短暂死机。
2. **光栅化压力**: 每帧渲染数千万像素的深度数据远超 GPU 的填充率极限。

优化方案: 我们通过基准测试, 将阴影贴图分辨率调整为 1024×1024 。虽然阴影边缘出现了轻微的锯齿 (见图 12(b)), 但渲染开销降低了约 93% (像素量从 1600 万降至 100 万), 使得场景漫游帧率稳定在 60 FPS 以上。



(a) 高分辨率显示的效果



(b) 降低分辨率导致阴影边缘锯齿化

图 12: glTF 模型加载问题的排查与修复对比。

通过进行上述的优化后, 虽然降低分辨率导致如右图所示的阴影边缘锯齿化, 但极大缓解了集显的带宽压力, 消除了运行时可能存在的卡死现象。

5.1.2 背面剔除策略

为了减少片元着色器的无效计算, 我们在渲染不透明物体 (如房屋、地面) 时开启了 `GL_CULL_FACE`, 剔除背对摄像机的三角形。

- **策略差异化**: 针对树叶和天空盒等特殊对象, 我们在渲染循环中动态关闭剔除, 以防止植被在侧视角度下消失或天空盒内部不可见。

5.1.3 粒子系统渲染优化

降雪系统包含成千上万个半透明粒子，若处理不当会产生严重的过度绘制问题。

- **Alpha 测试优化：**我们在 Shader 中引入了 `discard` 指令 (`if(alpha < 0.1) discard`)。对于完全透明的像素片段，直接在大规模深度测试和颜色混合计算之前丢弃，有效减轻了光栅化阶段的压力 [12]。

5.2 资源管理架构

为了解决场景中大量重复模型（如多棵树木、多盏路灯）带来的内存开销，我们重构了资源管理逻辑，实现了基于“享元模式”的对象管理系统 [4]。

5.2.1 模型数据的复用机制

在朴素的实现中，每创建一个新物体都重新加载一次 `.gltf` 文件会迅速耗尽内存。我们采用了“数据与状态分离”的设计：

- **固有状态：**包括顶点数据、纹理贴图、材质属性。这些数据由 `Model` 类统一加载，并在内存中仅保留一份副本。
- **外部状态：**包括位置、旋转、缩放。这些数据存储在轻量级的 `SceneObject` 结构体中。

如图 13 所示，场景中的所有路灯对象均指向同一个 `Model` 指针。渲染时，系统遍历轻量级对象列表，仅更新模型矩阵并复用已绑定的 VAO/VBO，极大地降低了显存占用和 CPU 到 GPU 的总线传输开销。

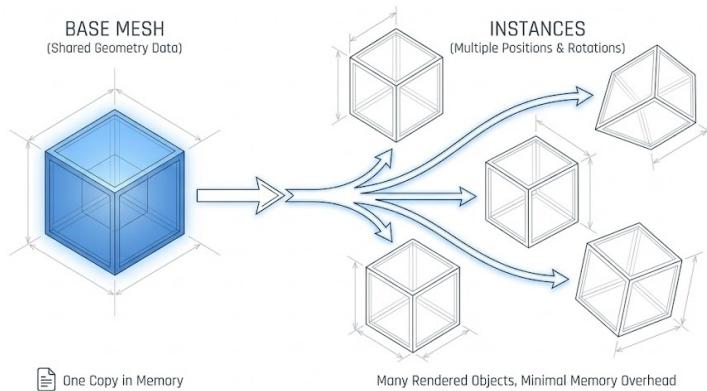


图 13：基于指针引用的资源复用架构示意图

5.2.2 预分配与生命周期管理

针对 C++ 运行时内存分配导致的性能抖动，我们严格遵循 **RAII** (资源获取即初始化) 原则：

- **杜绝运行时分配**: 所有的纹理加载 ('`glGenTextures`')、缓冲区创建 ('`glGenBuffers`') 和模型解析均在 '`while`' 渲染循环之前的初始化阶段完成。
- **容器预留**: 对于 `std::vector<SceneObject>`，在初始化时进行 '`reserve`' 操作，避免运行时的内存重新分配与拷贝。

这一策略彻底解决了程序运行初期因频繁 '`new/malloc`' 导致的掉帧问题。

6 遇到的主要问题与解决方案

在图形渲染管线的搭建过程中，我们遇到了涉及模型格式兼容性、底层二进制编码以及光照算法精度等多方面的技术挑战。通过查阅 OpenGL 规范文档、利用 RenderDoc 进行帧调试以及数学原理分析，我们逐一解决了这些阻碍系统运行的关键 Bug。

主要遇到的问题及其针对性的工程解决方案汇总如表 3 所示。这些问题的解决不仅保证了系统的稳定性，也加深了小组成员对图形学底层原理的理解。

7 总结与展望

7.1 项目成果总结

经过本学期的开发与迭代，小组成功构建了 **SOSRWIS (Snowy Outdoor Scene Rendering & Weather Interaction System)**。本项目不仅仅是对 OpenGL API 的简单调用，更是一次从零构建微型图形引擎的完整实践。主要的工程与学术成果总结如下：

1. **高保真的环境模拟能力**: 系统成功实现了物理意义上的动态昼夜循环。通过 SunSystem 的轨迹演算与多光源混合算法，场景能够精确呈现从晨曦、正午到午夜月色的光影流转。配合基于 Billboard 技术的粒子降雪系统，有效营造了沉浸式的冬季户外氛围。
2. **完整的渲染管线搭建**: 我们从底层搭建了可扩展的渲染架构。实现了基于 Assimp 的通用模型加载器，解决了 glTF 格式的节点变换与纹理适配难题；构建了 Two-pass Shadow Mapping 管线，并攻克了“阴影痤疮”(Shadow Acne) 与“悬浮”(Peter Panning) 等经典图形学伪影问题。

| 问题现象 | 解决方案 |
|-------------------------------|---|
| glTF 模型加载后零件散落/爆炸 | 启用 Assimp 的 <code>aiProcess_PreTransformVertices</code> 标志，将节点变换烘焙至顶点。 |
| Shader 编译报错”Unexpected Token” | 重写文件读取函数，以二进制流打开并自动跳过文件头的 UTF-8 BOM 标记。 |
| 物体表面出现条纹阴影 (Shadow Acne) | 在生成阴影贴图时开启 <code>GL_CULL_FACE</code> (正面剔除)，并微调 Shadow Bias。 |
| 单通道灰度贴图显示为鲜红色 | 设置纹理参数 <code>GL_TEXTURE_SWIZZLE_RGBA</code> ，将 R 通道值映射至 G、B 通道。 |
| 动漫模型面部出现黑斑/破碎 | 开启 <code>GL_CULL_FACE</code> (背面剔除)，隐藏用于描边的反向法线外壳。 |
| 玻璃材质不透明 | 开启 <code>GL_BLEND</code> 混合模式，并在 Shader 中依据 Alpha 值丢弃透明像素。 |
| 可以读取着色器的相对路径，却无法读取雪花图片的相对路径 | 可能是目录设置存在问题，先运行 <code>glTest.exe</code> ，再去运行 <code>glMain.exe</code> 即可解决问题。 |

表 3: 开发过程中的关键问题与解决方案汇总

3. **工程化与性能优化：**针对集显设备的硬件限制，我们实施了严格的资源管理策略。通过“享元模式”重构场景对象管理，大幅降低了内存开销；通过权衡阴影分辨率与剔除策略，在保证视觉质量的前提下，实现了稳定 60 FPS 的流畅漫游体验。

7.2 不足与改进方向

尽管项目达到了预期的功能目标，但对比工业界成熟的渲染引擎，本项目在光照真实感与物理交互深度上仍有较大的提升空间。未来的改进方向主要集中在以下三个维度：

7.2.1 从 Phong 模型向 PBR 的演进

目前系统使用的是经典的 Blinn-Phong 光照模型，材质表现力有限（尤其是金属与粗糙表面的质感）。

- **改进思路：**引入基于微表面理论的 PBR (Physically Based Rendering) 工作流。将目前的 Diffuse/Specular 贴图升级为 Albedo/Normal/Metallic/Roughness 贴图组，并实现 IBL (Image Based Lighting) 环境光照，以实现更真实的材质反射

效果。

7.2.2 阴影技术的升级 (CSM)

目前的阴影贴图采用单一的正交投影矩阵，存在“覆盖范围”与“分辨率”的矛盾。当视野拉远时（如调整到 500.0f），近处的阴影锯齿感会变强。

- **改进思路：**实现 CSM (Cascaded Shadow Maps, 级联阴影贴图)。将视锥体沿深度划分为多个层级（近景、中景、远景），分别为其生成不同分辨率的阴影贴图。这样既能保证近处阴影的高精度，又能覆盖远处的场景。

7.2.3 后处理特效

目前的渲染结果是直接输出到屏幕的原始颜色，缺乏电影级的视觉质感。

- **改进思路：**引入帧缓冲后处理管线。
 - Bloom (泛光)：提取高亮区域（如路灯、太阳）进行模糊叠加，制造光晕感。
 - HDR (高动态范围) + Tone Mapping：允许光照强度超过 1.0，再映射回屏幕空间，还原真实的明暗对比。
 - SSAO (屏幕空间环境光遮蔽)：增强物体角落和缝隙处的阴影感，提升场景的立体层次。

7.2.4 批量渲染 (Instanced Rendering)

目前降雪系统中，每个雪花粒子都单独 `glDrawArrays`, Draw Call 数量太多，影响性能，可能导致掉帧等问题。

- **优化思路：**具体分为以下两步
 - 使用 OpenGL 实例化渲染 (`glDrawArraysInstanced`)，一次性绘制所有粒子。
 - 把每个粒子的 model 矩阵数据上传到一个 VBO，作为实例属性。

7.2.5 高级空间划分与剔除

目前虽然使用了简单的视锥剔除（由 OpenGL 硬件完成），但在场景物体极多时，CPU 端的提交开销依然存在。未来可引入 Octree (八叉树) 或 BVH (层次包围盒) 算法对场景进行空间划分，实现更高效的视锥剔除 (Frustum Culling) 和遮挡剔除 (Occlusion Culling)。

参考文献

- [1] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*. A K Peters/CRC Press, 2018.
- [2] The Khronos Group, *glTF 2.0 Specification*. The Khronos Group Inc., June 2017. Available: <https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html>.
- [3] J. Gregory, *Game Engine Architecture, Third Edition*. Boca Raton, FL: CRC Press, 2018.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [5] C. Yuksel, “Point light attenuation without singularity,” in *ACM SIGGRAPH 2020 Talks*, SIGGRAPH 2020, (New York, NY, USA), ACM, August 2020.
- [6] P. P. Srinivasan, S. J. Garbin, D. Verbin, J. T. Barron, and B. Mildenhall, “Nuvo: Neural uv mapping for unruly 3d representations,” *arXiv preprint arXiv:2312.05283*, 2023. Accessed: 2024-12-19.
- [7] J. F. Blinn, “Models of light reflection for computer synthesized pictures,” in *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 192–198, 1977.
- [8] L. Williams, “Casting curved shadows on curved surfaces,” in *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 270–274, 1978.
- [9] S. Datta, D. Nowrouzezahrai, C. Schied, and Z. Dong, “Neural shadow mapping,” in *ACM SIGGRAPH 2022 Conference Proceedings*, 2022.
- [10] R. Fernando, “Percentage-closer soft shadows,” in *ACM SIGGRAPH 2005 Sketches*, pp. 35–es, 2005.
- [11] M. Stamminger and G. Drettakis, “Perspective shadow maps,” in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’02, pp. 557–562, 2002.
- [12] W. T. Reeves, “Particle systems—a technique for modeling a class of fuzzy objects,” *ACM Transactions on Graphics (TOG)*, vol. 2, no. 2, pp. 91–108, 1983.