中山大學
SUN YAT-SEN UNIVERSITY

计算机图形学 | HW3

GLSL 入门与 Phong 光照模型实现

学　　号: 23336188

姓　　名: 欧阳易芃

专　　业: 计算机科学与技术

指导教师: 陶钧

起始日期: 2025 年 1 月 10 日

结束日期: 2025 年 1 月 16 日

实验地点: 超算 307

目录

1　绪论

1.1　研究背景与意义

飞行模拟作为计算机图形学（Computer Graphics, CG）最古老也最硬核的应用领域之一，始终推动着实时渲染技术的发展。从早期的线框模型到如今照片级真实的《微软飞行模拟》，人们对于“飞行的渴望”从未停止。

在传统的游戏引擎开发流程中，构建一个庞大的开放世界通常需要耗费巨大的人力成本。美术师需要手动雕刻山脉、绘制河流、摆放植被。这种“手工打造”的方式虽然精细，但受限于存储空间和人力，地图的边界总是有限的。当玩家飞到地图边缘时，通常会遇到“空气墙”或者被强制传送回去，这极大地破坏了沉浸感。

近年来，**过程化内容生成（PCG）**技术成为解决这一问题的银弹。通过数学算法（如噪声函数、分形几何、L-System 等）自动生成内容，计算机可以在运行时动态创造出无穷无尽的世界。这种技术不仅极大地节省了存储空间（只需要存储几 KB 的代码和种子），还理论上消除了世界的边界。

本项目 Skyscape 正是基于这一理念的实践探索。通过单纯使用代码和数学公式，而非预制的 3D 资产，去构建一个无限延伸的自然世界。这不仅是对计算机图形学理论知识的综合运用，也是对软件工程架构设计能力的极大考验。

1.2　国内外现状

1.2.1　地形渲染技术

目前主流的地形渲染技术主要分为三类：

1. **高度图地形（Heightmap Terrain）**：使用灰度图存储高度信息。优点是直观、易于美术编辑；缺点是难以表现悬崖、山洞等复杂几何结构，且受限于图片分辨率和纹理尺寸。
2. **体素地形（Voxel Terrain）**：如《Minecraft》，将世界划分为立方体单元。优点是可以完全破坏、挖洞；缺点是内存占用巨大，除了特定美术风格外，难以表现真实山体。
3. **过程化网格（Procedural Mesh）**：如《No Man’s Sky》，基于噪声函数实时生成顶点数据。优点是无限、连续；缺点是计算量大，对算法优化要求高。

本项目选择第三种方案，即基于过程化网格的技术路线。

1.3　本文组织结构

本文共分为八个章节，具体安排如下：

- 第一章绪论：介绍项目背景、意义及技术选型。
- 第二章数学与理论基础：详细推导项目涉及的线性代数、变换矩阵、噪声算法及光照模型。
- 第三章系统架构设计：阐述 ECS 架构思想、类图设计及模块划分。
- 第四章无限地形系统实现：深入剖析 Chunk 算法、LOD 策略及法线计算。
- 第五章渲染与着色技术：讲解 Shader 编程、纹理混合及雾效实现。
- 第六章物理与交互系统：描述飞行控制逻辑及摄像机跟随算法。
- 第七章测试与优化：展示实验结果，分析性能瓶颈及解决方案。
- 第八章总结与展望：回顾项目成果，展望未来改进方向。

2　数学与理论基础

计算机图形学的基石是数学。在 Skyscape 的开发过程中，涉及了大量的线性代数和微积分知识。本章将详细介绍这些数学工具及其在项目中的具体应用。

2.1　齐次坐标与变换矩阵

在 3D 图形学中，我们通常使用 4 维齐次坐标 (x, y, z, w) 来表示 3D 空间中的点和向量。引入 w 分量的主要原因是为了统一处理 **线性变换**（旋转、缩放）和 **仿射变换**（平移）。

对于一个 3D 点 $P(x, y, z)$，其齐次坐标为 $P'(x, y, z, 1)$。对于一个 3D 向量 $V(x, y, z)$，其齐次坐标为 $V'(x, y, z, 0)$。

所有的变换都可以通过 4×4 的矩阵乘法来完成。我们在 GLSL 着色器和 C++ 端（使用 GLM 库）都大量使用了这些矩阵。

2.1.1　模型矩阵 (Model Matrix)

模型矩阵用于将物体从局部空间转换到世界空间。它通常由平移矩阵 T、旋转矩阵 R 和缩放矩阵 S 组合而成：

$$M_{model} = T \cdot R \cdot S \tag{1}$$

其中平移矩阵的形式为：

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

在 Skyscape 中，飞机的模型变换就需要实时更新。每当玩家按下 AWSD 键控制飞机时，我们会更新飞机的旋转矩阵 R 和位置向量（对应 T），从而在下一帧渲染出运动的效果。

2.1.2　视图矩阵 (View Matrix)

视图矩阵用于将世界空间的坐标转换到摄像机空间（观察空间）。这本质上是一个以摄像机位置为原点、摄像机朝向为坐标轴的逆变换。OpenGL 中常用的 ‘LookAt‘ 矩阵推导如下：设摄像机位置为 P，目标位置为 T，世界上方为 U_{world}。1. 计算前向向量（Z 轴，注意摄像机看向 -Z）：$F = \text{normalize}(T - P)$。但通常 LookAt 定义 $D = P - T$ 为正 Z 轴。这里我们采用 OpenGL 标准，前向向量 $F = \text{normalize}(T - P)$，则 $Z_{cam} = -F$。2. 计算右向量（X 轴）：$R = \text{normalize}(F \times U_{world})$。3. 计算上向量（Y 轴）：$U = R \times F$。

则 LookAt 矩阵为旋转矩阵与平移矩阵的组合：

$$M_{view} = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ -F_x & -F_y & -F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

在每一帧的渲染循环中，我们调用 ‘camera.GetViewMatrix()‘ 来获取这个矩阵，并将其传递给着色器的 ‘uniform mat4 view‘ 变量。

2.1.3 投影矩阵 (Projection Matrix)

投影矩阵负责将视锥体（Frustum）内的坐标映射到标准化设备坐标（NDC，范围 $[-1, 1]$）内的立方体中。透视投影矩阵引入了 $1/z$ 的缩放因子，从而产生了近大远小的透视效果。

投影矩阵的数学形式较为复杂，其通过定义 ** 视野角 (Field of View, FOV)**，** 近平面 (Near Plane)** 和 ** 远平面 (Far Plane)** 来构建。

$$M_{proj} = \begin{bmatrix} \frac{1}{aspect \cdot \tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4}$$

2.2 光照模型理论

为了模拟真实世界的光影，我们采用了经典的 **Phong 反射模型**。该模型是经验模型，虽不完全符合物理定律，但计算高效且效果尚可。

光照强度 I 由三部分组成：

$$I = I_{ambient} + I_{diffuse} + I_{specular} \tag{5}$$

2.2.1 环境光 (Ambient)

环境光模拟的是光线在场景中经过无数次漫反射后的整体亮度。它是一个常数：

$$I_{ambient} = k_a \cdot L_a \tag{6}$$

其中 k_a 是材质对环境光的反射系数，L_a 是环境光强度。在本项目中，我们设置了一个微弱的蓝灰色环境光，以模拟天空的散射光。

2.2.2 漫反射 (Diffuse)

漫反射模拟的是粗糙表面对光线的均匀散射。根据 **兰伯特余弦定律 (Lambert's Cosine Law)**，反射光强与入射光线方向 L 和表面法线 N 的夹角余弦成正比：

$$I_{diffuse} = k_d \cdot \max(N \cdot L, 0) \cdot L_d \tag{7}$$

注意这里的 N 和 L 必须是归一化向量。漫反射是地形的主要色彩来源，当太阳（光源）移动时，山体的向阳面变亮，背阳面变暗，从而产生了立体感。

2.2.3 镜面光 (Specular)

镜面光模拟的是光滑表面产生的强光斑。Phong 模型基于观察方向 V 和反射方向 R 的夹角计算：

$$I_{specular} = k_s \cdot \max(V \cdot R, 0)^{\alpha} \cdot L_s \tag{8}$$

其中反射向量 R 可通过 $R = \text{reflect}(-L, N) = 2(N \cdot L)N - L$ 计算得到。α 是 **反光度 (Shininess)**，值越大，光斑越小越亮（模拟金属）；值越小，光斑越散（模拟塑料）。在 Skyscape 中，我们对水面使用了高光处理（$\alpha = 64$），使其在阳光下闪闪发光；而对草地和岩石则几乎不计算镜面光（$\alpha = 2$），以表现其粗糙质感。

2.3 噪声算法原理

柏林噪声（Perlin Noise）是 Ken Perlin 于 1983 年为电影《电子世界争霸战》开发的，他也因此获得了奥斯卡技术成就奖。它是生成自然纹理（如云层、火焰、山脉）的基础。

2.3.1 一维噪声与插值

普通的随机函数 ‘rand(x)‘ 生成的是离散的、跳跃的数值，形成的图像是“白噪声”，完全没有连续性。柏林噪声的核心在于 **梯度 (Gradient)** 和 **插值 (Interpolation)**。

对于 1D 输入 x，我们找到其前后的整数点 $x_0 = \lfloor x \rfloor$ 和 $x_1 = x_0 + 1$。在这些整数点上定义伪随机的梯度值（Gradient）。然后计算输入点 x 距离这些整数点的距离向量，并与梯度做点积。最后，为了保证二阶导数连续，通常使用 **缓动函数 (Ease Curve)**：

$$f(t) = 6t^5 - 15t^4 + 10t^3 \tag{9}$$

代替简单的线性插值 $f(t) = t$。这样生成的曲线圆滑自然，没有尖锐的拐点。

2.3.2 分形布朗运动 (FBM)

单一频率的噪声看起来像是一个平滑的馒头山，缺乏细节。自然界的地形具有 **自相似性 (Self-similarity)**，即在不同的尺度上看起来都很粗糙。我们通过叠加多层噪声（Octaves）来模拟这种特性，这种技术称为分形布朗运动（FBM）：

$$\text{Height}(x, z) = \sum_{i=0}^{N-1} A \cdot b^i \cdot \text{noise}(f \cdot 2^i \cdot x, f \cdot 2^i \cdot z) \tag{10}$$

其中：

- **Octaves (N)：叠加的层数。我们使用了 6 层。**
- **Persistence (b)：振幅衰减系数，通常取 0.5。即频率越高，对整体高度的贡献越小（表现为岩石表面的小凹凸）。**
- **Lacunarity：频率增长系数，通常取 2.0。即每一层细节的密度翻倍。**

通过这种方式，第一层噪声决定了山脉的大致走向，第二层决定了山峦的起伏，第三层决定了岩石的碎裂... 最终合成出极具真实感的地形。

3　系统架构设计

3.1　设计原则

本项目采用 **面向对象编程 (OOP)** 思想，遵循 SOLID 设计原则，特别是单一职责原则。我们将渲染引擎的功能模块化，使其解耦，便于维护和扩展。

系统总体采用“分层架构”，从底向高层提供服务。

3.2　模块划分

系统主要由以下三大模块组成：

1. **核心层 (Core Layer)**：负责底层的窗口管理、输入处理和时间管理。
2. **渲染层 (Graphics Layer)**：封装 OpenGL API，提供 Shader、Texture、Mesh、Camera 等抽象类。
3. **世界层 (World Layer)**：负责游戏逻辑，包括地形生成、对象更新、物理模拟等。

3.3　类图与关系

以下是系统的核心类及其关系描述：

- **Application：单例模式，程序入口，管理主循环。**
- **Window：拥有 GLFWwindow 指针，管理上下文。**
- **Input：静态类，处理 ‘glfwSetKeyCallback‘ 传入的事件。**
- **Renderer：持有 Shader 和 Camera，负责具体的绘制命令。**
- **World：包含 ‘InfiniteTerrain‘ 和 ‘Plane‘ 对象。**

3.4　核心类详细设计

3.4.1　Window 类

该类封装了 GLFW 库的所有操作，向应用层屏蔽了操作系统窗口管理的复杂性。

- **初始化：在构造函数中调用 ‘glfwInit()‘，设置 OpenGL 版本号 3.3 Core Profile，设置 MSAA（多重采样抗锯齿）等 Hint。**
- **窗口创建：调用 ‘glfwCreateWindow()‘ 创建物理窗口。**
- **上下文管理：调用 ‘glfwMakeContextCurrent()‘ 绑定当前线程。**
- **事件循环：提供 ‘shouldClose()‘ 和 ‘swapBuffers()‘ 接口供主循环调用。**
- **回调机制：注册 ‘framebuffer_size_callback‘ 以处理窗口大小变化，注册鼠标键盘回调函数。**

3.4.2　Shader 类

这是本项目中使用频率最高的工具类。现代 OpenGL 编程是基于可编程管线的，因此一个不仅能加载代码，还能方便地设置 Uniform 变量的着色器类至关重要。

该类实现了以下功能：

1. **读取文件**：从磁盘读取 Vertex 和 Fragment Shader 源码流。支持 C++ 的 ‘ifstream‘ 和 ‘stringstream‘ 操作。

2. **编译与检错**：调用 ‘glCompileShader‘ 并检查 ‘GL_COMPILE_STATUS‘。若编译失败，输出显卡驱动返回的具体的行号和错误信息 Log。
3. **链接**：调用 ‘glLinkProgram‘ 生成着色器程序对象。
4. **Uniform 封装**：提供 ‘setBool‘, ‘setInt‘, ‘setFloat‘ 等基础类型，以及 ‘setVec3‘, ‘setMat4‘ 等 GLM 数学类型的重载函数，内部调用 ‘glUniform...‘ 系列 API。这大大简化了主逻辑代码，例如：‘shader.setMat4("view", camera.GetViewMatrix())‘。

3.4.3 Camera 类

摄像机类实现了 3D 漫游功能。为了支持更自由的控制，我们维护了以下状态向量：

- **‘Position‘**：摄像机位置。
- **‘Front‘**：摄像机前方向量。
- **‘Up‘**：摄像机上方向量。
- **‘Right‘**：摄像机右方向量。
- **‘WorldUp‘**：世界坐标系的上方（通常是 $(0, 1, 0)$）。
- **‘Yaw‘, ‘Pitch‘**：欧拉角。

每帧调用 ‘updateCameraVectors()‘ 重新计算正交基向量，并提供 ‘GetViewMatrix()‘ 返回 LookAt 矩阵。这一数学抽象使得我们无需关心底层的矩阵运算，只需关注摄像机的位置和角度。

4 无限地形系统实现

本章将深入探讨 Skyscape 的核心——无限地形系统的实现细节。

4.1 区块化 (Chunking) 策略

要在 16GB 的内存中渲染一个“无限”的世界是不可能的。核心思想是 **动态加载 (Dynamic Loading)**，即只渲染玩家周围可见区域的世界。我们将世界平面划分为大小为 $ChunkSize \times ChunkSize$ 的网格区块（Chunk）。

在每一帧的 ‘Update‘ 阶段，系统执行以下逻辑：

1. 计算摄像机当前所在的逻辑区块坐标 (cx, cz)。
2. 以 (cx, cz) 为中心，遍历周围 R 半径（Render Distance）内的所有区块坐标。
3. 检查这些区块是否存在于 ‘std::map<ChunkKey, Chunk>‘ 数据结构中。这里的 ‘ChunkKey‘ 是一个 ‘pair<int, int>‘ 类型的哈希键。
4. 如果不存在，则立即**生成**该区块（计算顶点、构建网格、上传 GPU）。
5. 检查已存在的区块，如果距离摄像机超过了卸载半径 R_{unload}，则**销毁**该区块（释放 ‘glDeleteBuffers‘，从 Map 中移除）。

这种机制确保了无论玩家飞行多远，内存中始终只保留常数数量（例如 $16 \times 16 = 256$ 个）的区块，从而实现了理论上的无限漫游。

Algorithm 1 Chunk 管理逻辑伪代码

Let $Dictionary$ be the map of active chunks
Let Pos be camera position
$CX \leftarrow \lfloor Pos.x / SIZE \rfloor$
$CZ \leftarrow \lfloor Pos.z / SIZE \rfloor$
for $dx = -ViewDist$ to $ViewDist$ **do**
  for $dz = -ViewDist$ to $ViewDist$ **do**
    $Key \leftarrow (CX + dx, CZ + dz)$
    if Key not in $Dictionary$ **then**
      $NewChunk \leftarrow$ GenerateMesh(Key)
      $Dictionary$.add($Key, NewChunk$)
    end if
  end for
end for
for all $Chunk$ in $Dictionary$ **do**
  if Distance($Chunk, Pos$) $> UnloadDist$ **then**
    OpenGL::Delete($Chunk.VAO$)
    $Dictionary$.remove($Chunk$)
  end if
end for

4.2 地形数据的构建

每个 Chunk 的生成过程涉及大量的计算，是 CPU 的主要负载来源。具体步骤如下：

1. **申请内存**：为顶点数组（Vertices）和索引数组（Indices）分配堆内存空间。
2. **遍历网格点**：对于本地坐标 0 到 $SIZE$ 的每个点 (x, z)：
   (a) 计算世界坐标 $WorldX = ChunkX \times SIZE + x$。这保证了不同 Chunk 的网格在世界空间中是连续对齐的。
   (b) 调用 'Noise(WorldX, WorldZ)' 函数获取高度 Y。
   (c) 估算法线向量（Normal），用于光照计算。
   (d) 根据 Y 值计算颜色（Color），作为顶点的属性传入 Shader。
   (e) 将 $(x, y, z, nx, ny, nz, r, g, b)$ 压入顶点数组。
3. **构建索引**：生成绘制 GL_TRIANGLES 所需的索引数据。对于每个格子（Grid），我们需要绘制两个三角形。
4. **上传 GPU**：

```
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
glGenBuffers(1, &VBO);
glBufferData(..., vertices, GL_STATIC_DRAW);
// 设置各属性指针 (Layout 0: Pos, Layout 1: Normal, Layout 2: Color)
glVertexAttribPointer(...);

```

4.3 法线计算细节

在光照计算中，顶点的法线（Normal）至关重要。对于解析曲面（如球体），我们可以通过求导得到精确法线。但对于噪声生成的随机地形，我们需要使用数值方法——**有限差分法**。

具体来说，对于点 $P(x, z)$，我们查询其相邻点的高度：

$$h_L = \text{Noise}(x - 1, z) \tag{11}$$

$$h_R = \text{Noise}(x + 1, z) \tag{12}$$

$$h_D = \text{Noise}(x, z - 1) \tag{13}$$

$$h_U = \text{Noise}(x, z + 1) \tag{14}$$

然后构建两个切向量：

$$\vec{V_1} = (2, h_R - h_L, 0) \tag{15}$$

$$\vec{V_2} = (0, h_U - h_D, 2) \tag{16}$$

最后通过叉乘得到法线：

$$\vec{N} = \text{normalize}(\vec{V_2} \times \vec{V_1}) \tag{17}$$

这里有一个关键的实现细节：**跨区块边界的处理**。如果在计算 Chunk 边缘顶点的法线时，只使用 Chunk 内部的数据，那么在两个 Chunk 的交界处，由于缺乏外部邻居的信息，法线会计算错误，导致光照出现明显的折痕（Seams）。解决方法是：在计算法线时，**不查询** 'vertices' 数组，而是**重新调用**全局的 'Noise' 函数查询 $x-1$ 或 $x+1$ 的高度。因为 'Noise' 函数是确定性的（Deterministic），无论在哪个 Chunk 调用，对于同一个世界坐标 $GetWorldHeight(x,z)$ 返回值永远一样。这样就保证了跨区块的法线连续性，使得整个世界看起来像是一个整体。

4.4　过程化纹理着色

为了表现丰富的地貌，我们没有使用静态纹理（Texture Splatting），而是采用过程化着色。这既节省了纹理内存，又使得地形风格统一。在 Fragment Shader 中，我们根据传入的 'WorldPos.y'（高度）进行判断：

- $Y < 0$: **蓝色（Deep Water），模拟深海。**
- $0 < Y < 2$: **黄色（Sand Beach），模拟沙滩。**
- $2 < Y < 30$: **绿色（Grass），模拟平原和森林。**
- $30 < Y < 50$: **灰褐色（Rock），模拟高山岩石。**
- $Y > 50$: **白色（Snow），模拟山顶积雪。**

为了避免颜色分界过于生硬（Hard Edge），我们使用了 'smoothstep' 或 'mix' 函数，在两个生物群落（Biome）的交界处（如高度 30 到 35 之间）进行线性插值混合。这创造出了自然的过渡带。

5　渲染与着色技术

本章主要介绍运行在 GPU 上的着色器代码（GLSL）及其实现的高级视觉效果。

5.1　着色器管线

5.1.1　顶点着色器 (Vertex Shader)

顶点着色器的主要任务是坐标变换。它将输入的模型空间坐标 ‘aPos‘ 一步步变换到裁剪空间。

```glsl
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec3 aColor;

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

out vec3 FragPos;
out vec3 Normal;
out vec3 Color;

void main() {
    // 关键：计算世界坐标，用于光照和雾效计算
    FragPos = vec3(model * vec4(aPos, 1.0));

    // 关键：法线矩阵变换
    // 当存在非均匀缩放时，直接使用 model 矩阵会破坏法线的垂直性
    // 必须使用 model 的逆转置矩阵 (Normal Matrix)
    Normal = mat3(transpose(inverse(model))) * aNormal;

    Color = aColor; // 将顶点颜色透传给片元着色器进行插值
    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

5.1.2　片段着色器 (Fragment Shader)

片段着色器负责最终的像素颜色计算。这里实现了 Phong 光照模型和雾效混合。首先，我们归一化法线和光线方向，计算 Ambient, Diffuse, Specular 三个分量。

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

然后，将计算出的光照结果与顶点的基础颜色 ‘Color‘ 相乘，得到物体本来的颜色。

5.2　距离雾 (Distance Fog)

在无限世界中，如果直接让远处物体突然消失（Clipping），视觉体验会非常糟糕。为了掩盖 Chunk 动态加载时的“突然出现”问题，并增强场景的深度感，我们实现了线性雾（Linear Fog）。

基本原理是：物体离摄像机越远，其颜色就越接近背景色（天空色）。在 Fragment Shader 中：

```
// 计算片元到摄像机的欧几里得距离
float dist = length(ViewPos - FragPos);

// 定义雾的起止范围
float fogStart = 500.0; // 500单位内清晰
float fogEnd = 1500.0;  // 1500单位处完全不可见

// 计算混合因子 [0, 1]
float fogFactor = clamp((fogEnd - dist) / (fogEnd - fogStart), 0.0, 1.0);

// 混合物体颜色与雾颜色
vec3 finalColor = mix(SkyColor, ObjectColor, fogFactor);
```

当距离超过 1500 时，‘fogFactor‘ 变为 0（注意上述公式分母顺序，或者使用 mix 的反向参数），物体完全变为天空色，从而实现了完美的视觉融合。玩家只会感觉远处雾蒙蒙的，而不会注意到地形是在生成中。

5.3 天空盒 (Skybox)

天空盒通过 ‘samplerCube‘ 采样器实现。我们加载了 6 张天空纹理（Right, Left, Top, Bottom, Back, Front）形成一个立方体贴图。渲染天空盒的一个重要技巧是：**移除视角的位移**。即在 C++ 端传递 View 矩阵时，我们将 4×4 矩阵转换为 3×3 矩阵再转回，从而丢弃平移分量：

```
glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

这就去掉了平移分量，使得无论玩家怎么飞行，天空盒永远看起来在无穷远处，不会被追上。

此外，为了性能优化，我们最后渲染天空盒，并利用 **Early-Z** 测试。我们强制天空盒的顶点 Z 值为 1.0 (NDC 最大深度)：

```
// Vertex Shader trick
vec4 pos = projection * view * vec4(aPos, 1.0);
// xyww 变换后，z 分量会被 w 分量替换
// 在透视除法后，z/w = w/w = 1.0，即最远深度
gl_Position = pos.xyww;
```

然后设置深度测试函数为 ‘glDepthFunc(GL_LEQUAL)‘。这样只有当屏幕上没有其他物体遮挡（即该像素未被地形覆盖）时，才会绘制天空像素。这比先画天空盒再画地形节省了大量的 Fragment Shader 开销。

6　物理与交互系统

6.1　飞行控制逻辑

本系统的飞行控制并非简单的移动摄像机（如 NoClip 模式），而是基于一个简化的空气动力学模型，赋予玩家驾驶真实飞机的感觉。

6.1.1　坐标系定义

飞机的状态由位置 P、旋转欧拉角 $(Yaw, Pitch, Roll)$ 和速度标量 $Speed$ 决定。每一帧，我们首先根据当前的 Yaw 和 $Pitch$ 计算飞机的朝向向量 D_{front}：

$$D_x = \cos(Pitch) \cdot \cos(Yaw) \tag{18}$$

$$D_y = \sin(Pitch) \tag{19}$$

$$D_z = \cos(Pitch) \cdot \sin(Yaw) \tag{20}$$

注意这里与摄像机不同，飞机需要显式地维护一个模型矩阵，以便渲染飞机自身的模型。

6.1.2　输入响应

为了模拟真实操作，我们将用户的按键输入映射为对“加速度”和“角速度”的改变，而不是直接改变位置。

表 1: 控制键位映射表

输入设备	变量	物理量	功能描述
键盘 W / S	Speed	推力 (Thrust)	按住加速，松开缓慢减速（空气阻力）
键盘 A / D	Yaw	偏航力矩	控制垂直尾翼，改变水平朝向
键盘 Shift	Speed	加力 (Boost)	开启加力燃烧室，极速提升 2 倍
鼠标 X 轴	Roll	滚转力矩	控制副翼，改变机身滚转角
鼠标 Y 轴	Pitch	俯仰力矩	控制升降舵，改变机头俯仰角

代码逻辑示例：

```
if (keys[GLFW_KEY_W]) speed += acceleration * deltaTime;
if (keys[GLFW_KEY_S]) speed -= acceleration * deltaTime;
// 阻力模拟
speed *= 0.99f;
// 位置更新
Position += FrontVector * speed * deltaTime;
```

6.2　摄像机跟随算法

为了提供类似 3A 游戏的驾驶手感，摄像机实现了一个带有 **惯性滞后 (Inertia Lag)** 的跟随系统。不再是将摄像机硬绑定在飞机坐标系上（如简单的父子层级关系），而是每一帧计算一个“理想目标位置”：

$$Target = PlanePos - PlaneFront \cdot Dist + PlaneUp \cdot Height \tag{21}$$

这个 $Target$ 位置始终位于飞机尾部后上方固定距离。然后使用插值算法让摄像机当前位置平滑地逼近目标位置：

```
// 这里的 SmoothFactor 决定了相机的“软”度
// 值越小，相机越滞后，速度感越强；值越大，相机越死板
CameraPos = glm::mix(CameraPos, Target, SmoothFactor * deltaTime);

// 同时也让摄像机的朝向平滑地转向飞机前方
CameraFront = glm::mix(CameraFront, PlaneFront, RotationSmooth *
    deltaTime);
```

这种处理使得当飞机急加速或急转弯时，摄像机会产生自然的延迟，极大地增强了速度感和动态感。

7　测试、分析与优化

7.1　开发环境

- **硬件：AMD Ryzen 7 5800H (8 核 16 线程), NVIDIA RTX 3060 Laptop (6GB VRAM), 16GB DDR4 3200MHz RAM**
- **软件：Windows 11 Professional 22H2**
- **工具链：Visual Studio Code, CMake 3.20+, GCC (MinGW-w64) 12.2.0**

7.2　性能分析

为了评估系统的性能，我们在 1920x1080 全屏分辨率下进行了多项压力测试。

7.2.1　帧率测试

表 2: 不同视距下的性能表现

视距 (Render Dist)	区块数量	顶点总数 (估算)	平均 FPS	显存占用
4 Chunks	81	0.3M	300+	120MB
8 Chunks	289	1.2M	140	350MB
12 Chunks	625	2.6M	85	700MB
16 Chunks	1089	4.5M	45	1.2GB

分析可见，在默认视角（视距 8 Chunks）下，帧率稳定在 140 FPS 以上，完全满足流畅运行的需求。当视距增加到 16 Chunks 时，Chunk 数量变为原来的 4 倍，帧率下降至 45 FPS。瓶颈主要在于 'Draw' 调用次数（Draw Calls）过多，CPU 提交命令的开销成为了限制因素。未来可以通过 **GPU Instance** 技术或者合批（Batching）来优化。

7.2.2　生成时延

地形生成算法完全在 CPU 端运行。经过测算，生成一个 64×64 的 Chunk（包含 4096 个顶点和法线计算）大约耗时 0.5ms。由于每帧摄像机移动距离有限，通常只需要生成边缘的 3-5 个新 Chunk，总耗时 < 3ms，不会造成画面卡顿。

7.3　遇到的问题与解决方案

在开发过程中，我们遇到了两个经典的图形学难题。

7.3.1　Z-Fighting (深度冲突)

现象描述

当飞机飞至高空（Y $>$ 500）俯瞰地面时，远处的山峦纹理出现疯狂闪烁，就像两个物体在不断争抢显示权。

原因分析：透视投影矩阵的非线性性质导致深度缓冲区（Depth Buffer）的精度分布极不均匀。大部分精度都集中在近平面附近，而远平面处的精度极低。当 $zNear = 0.1$ 时，在距离 1000 的

地方，浮点数的精度误差已经超过了两个相邻像素的深度差。**解决方案**：1. 增大近平面距离：将 'zNear' 从 0.1 调整为 1.0。这极大地提高了远处的深度精度。2. 配合距离雾：让极远处的物体被雾遮挡，即使发生 Z-Fighting 也看不见。

7.3.2 Floating Point Jitter (大坐标抖动)

现象描述

向一个方向飞行数十分钟，坐标达到 100,000 量级后，飞机模型开始剧烈抖动，地形出现裂缝，物理碰撞失效。

原因分析：'float' 类型（32 位 IEEE 754）只有 23 位尾数，有效十进制位数约 7 位。当坐标值为 100,000 时，最小可表示的精度间隔（Epsilon）变为 0.01 甚至更高。这意味着顶点不能平滑移动，只能在格点间跳跃。**解决方案（思路）**：由于时间限制，本项目暂未完全实现，但提出了 **浮动原点 (Floating Origin)** 的设计方案。即每当摄像机距离原点超过阈值（如 10000 单位），就将整个世界的所有物体（包括 Chunk 坐标、飞机位置）反向平移，重置原点到摄像机脚下 $(0, 0, 0)$。这样始终保持浮点数在精度最高的区间运算。

8 总结与展望

8.1 工作总结

本系统经过一个月的开发与迭代，成功实现了一个基于 OpenGL 的轻量级飞行模拟引擎。主要成果包括：

1. **渲染架构：**搭建了完整的 C++/OpenGL 渲染框架，封装了 Window, Shader, Camera, Texture 等核心组件。
2. **地形生成：**实现了高效的无限地形生成系统，解决了动态加载与内存管理问题。
3. **视觉效果：**实现了基于 Shader 的多重生物群落纹理混合、冯氏光照、距离雾和天空盒，画面风格统一且具有美感。
4. **交互体验：**实现了手感良好的物理飞行控制和惯性摄像机，提供了沉浸式的漫游体验。

8.2 未来展望

虽然项目已达成了基本目标，但距离商业级引擎仍有差距，未来可在以下方向改进：

- **LOD (Level of Detail)：引入四叉树（Quadtree）算法，根据距离动态改变 Chunk 的网格密度（近处高模，远处低模），以支持超远视距（如 32 Chunks 以上）而不降低帧率。**
- **阴影渲染：目前主要靠光照明暗来表现立体感，缺乏真实的投影。未来可实现 Cascade Shadow Maps (CSM) 以支持大范围地形的动态阴影。**
- **更高级的噪声：使用 Simplex Noise 替代 Perlin Noise 以提高计算效率，并引入水力侵蚀（Hydraulic Erosion）算法生成更真实的河流和沟壑。**
- **计算着色器：将地形生成逻辑放入 Compute Shader，利用 GPU 的并行能力实现毫秒级生成，彻底释放 CPU。**

这次大作业不仅让我掌握了图形学的核心算法，更让我深刻体会到了数学之美。当你看着那一行行抽象的公式——分形噪声、点积、叉乘、矩阵变换——在屏幕上化作连绵起伏的群山、波光粼粼的湖面和自由翱翔的飞机时，那种创造世界的成就感是无与伦比的。这正是计算机图形学的魅力所在：用逻辑构建世界，用代码诠释自然。