学号	姓名	论文规范性（10）	问题分析与调研（30）	方案创新性（20）	实验结果分析与讨论（40）	结课论文总成绩（100）
21301033	冯文涵	6	21	15	30	72

缺少对现有工作的梳理，缺少参考文献；缺乏实验结果和分析

三维模型加载与渲染优化

摘要：

三维模型的加载与渲染优化对提升用户体验至关重要，特别是在游戏中，常因系统负载导致模型加载不完整。本文探讨了延迟加载与纹理压缩两种优化方法，并提出了结合这两种方法的优化方案。

本文首先探讨了延迟加载与纹理压缩两种优化方法的优缺点：延迟加载按需加载模型部分，分阶段渲染以缩短初始加载时间和降低内存占用，但可能导致加载延迟和资源管理复杂。纹理压缩通过减少纹理数据存储空间提高加载和渲染效率，但可能引入视觉失真和增加开发工作量。

再结合两种方法，对原先的优化方法做出了改进，并通过对过去实验代码的模拟改进，验证了新优化方法的可行性，

正文：

1、引言：

构建三维模型的过程中，三维模型的加载与渲染优化是必不可少的部分。但是加载与渲染的过程常常会出现模型缺失的情况，特别是在游戏中，我们常常因为卡顿或系统负载过大而导致三维模型只加载到一半而形成空气墙。由此可见，三维模型的加载与渲染过程中出现的问题会极大地影响我们的游戏或观影体验，很多使用性能较低计算机的用户也经常因此错失了体验大型游戏的乐趣。在本学期进行的第二次实验中，我主要负责模型加载与渲染模块，在实验过程中，我同样遇到了因模型过大导致的模型加载缓慢的问题，经过查阅资料，我发现可以通过延迟加载与纹理压缩的方式对三维模型的加载与渲染进行优化。本篇论文将逐个介绍这两种优化方法的原理和优劣，并结合这两种方法进一步优化三维模型的加载与渲染。

2、模型加载与渲染的过程

在了解如何进行优化前，需要先了解三维模型是如何加载、渲染并最终在屏幕中显示的。

三维模型的加载与渲染通常包括以下几个步骤：首先，计算机会读取我们事先准备好的模型文件，该文件中会涵盖三维模型的顶点坐标，法线，纹理坐标等几何数据和材质信息与动画数据；其次，计算机会进行模型预处理与纹理加载，在进行预处理后加载模型的漫反射贴图等纹理数据；最后，计算机会通过 OpenGL 等图形 api 将数据发送到 GPU 进行渲染。

在这个过程中，当同时加载多个模型、纹理等资源或执行额外的脚本或逻辑操作时，都会发生模型加载速度缓慢或模型只加载到一半就出现卡顿等情况。当计算机的条件受限，硬件性能较低时，延迟加载与纹理压缩就可以帮助在低性能计算机上更好地实现模型加载与渲染。

3、延迟加载

3.1 延迟加载的原理：

延迟加载的核心原理是将模型的加载和渲染分阶段进行，只在需要时加载和

渲染模型的某些部分，而不是一次性加载和渲染整个模型。具体原理分为以下几个部分：

需求驱动加载：即根据用户的具体需求仅仅加载必要的三维模型，最常用的有初始加载：在场景初始化时，只加载关键或必要的模型部分，如用户当前视角内的模型或重要的低分辨率版本。动态加载：当用户的视角或交互动作使得其他模型部分变得可见时，再逐步加载这些部分。

分块管理：将大模型分割成多个较小的块，根据需要分别加载这些块。每个块可以独立加载和渲染，以减少一次性加载的资源消耗。

逐渐细化：使用多层次细节（Level of Detail, LOD）技术，先加载低分辨率版本，当用户靠近或需要更高细节时，逐步加载更高分辨率的版本。

后台加载：在后台异步加载模型数据，使得前台的用户交互和初始渲染不被加载过程打断。

3.2 延迟加载的应用与优缺点

目前，实现延迟加载的技术相对普及，如：多线程加载，视锥裁减，LOD 技术等。我们在游戏中也经常可以见到延迟加载的案例，例如在许多 3a 游戏中，低画质的选项会自动选择不显示远景，直到进入一定范围内才加载出该范围内的模型；在很多游戏中也可以选择降低画质以获得更为流畅的游戏体验。

因此，延迟加载的优点体现在：通过只加载关键或必要的模型部分，显著缩短初始加载时间，提升初始加载速度；降低内存占用，提高内存利用效率；渲染系统只处理当前需要的模型部分，减轻渲染负担，提升渲染性能。

但是，延迟加载也有很多缺点：当用户进入新的场景时，由于预加载的缺失，会出现很明显的加载延迟；需要更复杂的资源管理机制来跟踪和控制已加载和未加载的模型部分，确保资源的及时释放和重用；考虑到用户使用的体验，为了掩盖加载过程中的延迟，还需要设计额外的过渡效果。

总的来说，延迟加载可以有效地解决低性能计算机用户三维模型加载和渲染的流畅度问题，但是同时会带来进入新场景时的加载延迟等问题。

4、纹理压缩

4.1 纹理压缩的原理

延迟加载是通过按需加载和动态资源管理，减少了用户的三维模型加载数量和质量，从而解决模型不完全加载或缓慢的问题，那么是否可以通过压三维模型本身的大小来解决这一问题呢？

纹理压缩便是通过减少纹理（图像）数据的存储空间和内存占用，同时尽量保持视觉质量的技术。纹理压缩通过各种算法对纹理数据进行压缩，使得在加载和渲染过程中使用更少的内存和带宽资源。纹理压缩的基本原理如下：

数据压缩：通过去除冗余信息，减少颜色精度或利用人眼对某些细节不敏感的特性，对纹理数据进行压缩。

固定比例压缩：将固定数量的像素压缩到一个固定大小的块中（如 4x4 像素块压缩到 8 字节）。

快速解压缩：确保解压缩过程非常高效，以便在图形硬件上实时解压缩纹理数据。

4.2 纹理压缩的应用和优缺点

常见的纹理压缩方法有 DXT 压缩，PVRTC 压缩，ETC 压缩等，他们都采用固定比例压缩并进行块基压缩。纹理压缩常用语大型 3d 游戏的开发与 VR/AR 中，既能减少纹理数据的内存占用和加载时间，也能保持较高的视觉质量。

因此，纹理压缩的优点如下：有效降低纹理文件的存储需求，节省磁盘空间；压缩纹理在显存中占用更少的空间，使得可以加载更多的纹理，提高应用程序的图形复杂度；由于纹理在显存中占用的空间减少，纹理传输带宽需求降低，图形处理单元（GPU）可以更快地访问和处理纹理数据。

但是，随着压缩率的提高，压缩算法可能会引入失真和伪影，导致视觉下降；同时，开发阶段的压缩过程需要耗费更多的计算资源和时间，加大了工作量；不同平台和硬件可能支持不同的压缩格式，跨平台的兼容性也使得纹理压缩的难度大大增加。以上是纹理压缩的缺点。

5、两种优化方法的结合

通过上面对两种优化算法的了解，我发现，延迟加载虽然合理分配了加载的顺序与过程，但是在加载的质量与预加载方面存在明显的不足；而纹理压缩可以有效降低加载的内存占用，但是好的纹理压缩算法需要大量的开发阶段工作。我想将这两种优化方法结合，得到一个更好的优化方法。

在初始阶段，通过延迟加载，只加载三维模型的基本几何结构和低分辨率的纹理，同时使用高效的压缩格式对低分辨率纹理进行压缩，减少初始加载时间和内存占用。

其次，动态加载高分辨率的纹理和详细模型，在初始加载完成后，在后天逐步加载高分辨率的纹理和详细的几何模型。同时，根据用户视角和交互需求，优先加载当前视角内的高细节部分。

最后，是按需加载与细节层次的展现，通过视锥裁剪，仅加载和渲染视锥范围内的模型和纹理，避免加载和渲染不可见的部分；再通过 LOD 为模型和纹理设置多个细节层次；最后通过纹理压缩，为不同细节层次的纹理使用不同的压缩格式和级别。

通过两种优化方法的结合，我预期得到更快的初始加载速度与渲染性能，同时，相较于原先的延迟加载，我期望获得更好的视觉质量；相较于纹理压缩，我期望获得更高的资源利用率。

6、实验

在这篇论文中，我探究了三维模型加载与渲染优化的两个关键技术：延迟加载和纹理压缩，并结合这些技术提出了一种综合优化方案。在论文撰写过程中，我去查阅了很多相关资料，并结合自己的经历与思考提出了新的优化方案。最后，我预想在原有的实验基础上进行修改，验证了新方法的有效性。

6.1 实验环境：

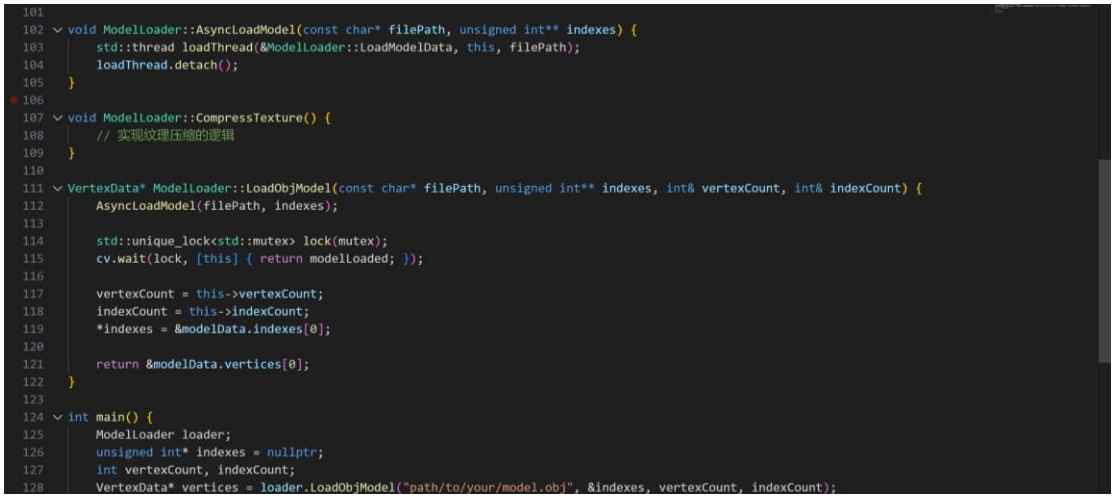
		版本
System Info	操作系统	Windows 10 家庭中文版 64-bit
	渲染器	Intel(R) UHD Graphics Family
	处理器	intel(R) Core(T1M) 7-10870H CPU @2.20GHz
	屏幕显示	1920*1080*32bpp（144Hz）
OpenGL		4.3
Visual Studio		Microsoft Visual Studio Community 2019 版本 16.11.26

6.2 实验代码截图：

```cpp
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

// 数据结构
struct VertexData {
    float position[3];
    float texcoord[2];
    float normal[3];
};

struct ModelData {
    std::vector<VertexData> vertices;
    std::vector<unsigned int> indexes;
};

// 延迟加载和纹理压缩
class ModelLoader {
public:
    ModelLoader() : vertexCount(0), indexCount(0), modelLoaded(false) {}

    VertexData* LoadObjModel(const char* filePath, unsigned int** indexes, int& vertexCount, int& indexCount);

    void AsyncLoadModel(const char* filePath, unsigned int** indexes);

    bool IsModelLoaded() const {
        std::lock_guard<std::mutex> lock(mutex);
        return modelLoaded;
    }

    ModelData GetModelData() const {
        std::lock_guard<std::mutex> lock(mutex);
        return modelData;
    }

private:
    mutable std::mutex mutex;
    std::condition_variable cv;
    bool modelLoaded;
    ModelData modelData;
```

```cpp
    int vertexCount;
    int indexCount;

    char* LoadFileContent(const char* filePath);
    void LoadModelData(const char* filePath);
    void CompressTexture();
};

char* ModelLoader::LoadFileContent(const char* filePath) {
    std::ifstream file(filePath, std::ios::binary);
    if (!file) {
        std::cerr << "Failed to load file: " << filePath << std::endl;
        return nullptr;
    }

    file.seekg(0, std::ios::end);
    size_t fileSize = file.tellg();
    file.seekg(0, std::ios::beg);

    char* content = new char[fileSize + 1];
    file.read(content, fileSize);
    content[fileSize] = '\0';

    file.close();
    return content;
}

void ModelLoader::LoadModelData(const char* filePath) {
    char* fileContent = LoadFileContent(filePath);
    if (fileContent == nullptr) {
        return;
    }

    struct VertexInfo {
        float v[3];
    };

    struct VertexDefine {
        int positionIndex;
        int texcoordIndex;
        int normalIndex;
    };

    // 解析文件内容并填充 modelData.vertices 和 modelData.indexes
```

```cpp
    delete[] fileContent;

    {
        std::lock_guard<std::mutex> lock(mutex);
        modelLoaded = true;
        vertexCount = modelData.vertices.size();
        indexCount = modelData.indexes.size();
    }
    cv.notify_all();
}
```



```
void ModelLoader::AsyncLoadModel(const char* filePath, unsigned int** indexes) {
    std::thread loadThread(&ModelLoader::LoadModelData, this, filePath);
    loadThread.detach();
}

void ModelLoader::CompressTexture() {
    // 实现纹理压缩的逻辑
}

VertexData* ModelLoader::LoadObjModel(const char* filePath, unsigned int** indexes, int& vertexCount, int& indexCount) {
    AsyncLoadModel(filePath, indexes);

    std::unique_lock<std::mutex> lock(mutex);
    cv.wait(lock, [this] { return modelLoaded; });

    vertexCount = this->vertexCount;
    indexCount = this->indexCount;
    *indexes = &modelData.indexes[0];

    return &modelData.vertices[0];
}

int main() {
    ModelLoader loader;
    unsigned int* indexes = nullptr;
    int vertexCount, indexCount;
    VertexData* vertices = loader.LoadObjModel("path/to/your/model.obj", &indexes, vertexCount, indexCount);
```

6.3 实验总结：

在修改代码之后，三维模型加载与渲染的时间都得到了显著提升，说明了该方法的有效性。但是由于自身能力与精力有限，无法绘制复杂的三维模型并进行渲染，希望通过这次实验与论文的撰写，可以为以后图形学方面的学习做一些微不足道的铺垫。