SimpleRenderer 项目申请书

何卓昊

2024/05/31

摘要

本报告总结了SimpleRenderer项目的当前进展，并提出了项目的完善计划。完善计划主要从四个方面展开：扩展项目功能、提升渲染性能、提高代码质量、加强文档支持。在报告的最后提出了项目的未来开发计划。通过进一步的开发和优化，SimpleRenderer可以成为一个轻量且易用的渲染器，并作为学习图形学和3D编程的优秀资源。

项目当前进度总结

通过阅读项目代码和文档，对SimpleRenderer项目实现了一系列功能进行了总结。

核心功能和进展

数学基础

项目包括用于3D变换的数学工具。这些工具包括：

- **向量和矩阵:** 基本操作，如加法、减法、点积、叉积和矩阵乘法。
- **变换:** 用于缩放、旋转和平移3D对象的变换矩阵函数。

光栅化管道

光栅化过程包括：

- **深度缓冲 (Z-buffering):** 处理深度信息并正确渲染重叠的对象。
- **基本图元绘制:** 绘制基本形状（如线条和三角形）的函数。

纹理和材质管理

系统支持：

- **纹理映射:** 将2D纹理应用于3D表面。
- **材质属性:** 处理漫反射、镜面反射和法线贴图，以增强渲染图像的视觉质量。
- **PBR纹理:** 基本支持基于物理的渲染(PBR)纹理，包括反照率、金属度和粗糙度贴图。

光照实现

系统实现了多种光源，包括：

- **点光源:** 模拟从单个点向所有方向发射的光。
- **方向光:** 代码中未明确提及，但框架建议可能支持。
- **光照计算:** 可能使用基本模型（如兰伯特反射）进行漫反射光照计算。

渲染状态

渲染器支持不同的渲染模式，包括：

- **线框模式:** 仅渲染形状的边缘。
- **纹理模式:** 将纹理应用于形状的表面。
- **颜色模式:** 使用纯色渲染形状。

项目完善思路的思考

尽管项目已经实现了多项功能，但仍有一些方面需要改进，以增强功能性和提升性能：

- **功能增强:** 完善目前项目中已经实现的功能，并添加新的功能，如材质支持和相机支持，在此基础上可以进一步扩展。
- **性能优化:** 当前实现可能需要优化，特别是在处理复杂场景或大型纹理时，这会消耗大量资源。
- **用户交互:** 提升用户体验并简化测试和调试过程。

代码结构和注释:

- **代码结构:** 代码的整体结构是功能性的，但可以通过重构来提高可读性和可维护性。例如更清晰的模块化和面向对象编程，将提升项目的可扩展性并简化未来的开发工作。
- **注释:** 代码中的注释并不总是清晰或专业。有些是中英文混合，可能不容易被所有团队成员理解。提高注释的质量和一致性将使代码库更易于访问和导航。
- **文档:** 项目文档可以进一步的补充和完善，例如添加中英双语文档；并仿照tinyrenderer项目，将文档设计为多个章节循序递进的模式，更利于初学者学习和使用。

项目完善方案

根据目前的项目进度，项目完善方案主要从以下四、个方面展开：

1. 完善并拓展项目功能
2. 提升渲染性能
3. 提高代码质量
4. 完善文档支持

一、完善并拓展项目功能

1. 完善贴图支持

1) 高分辨率贴图加载

当前状态: 项目目前通过函数如 `device_set_texture` 和 `device_set_texture_by_photo` 支持基本的纹理加载。

改进步骤:

- **审查和优化纹理加载代码:**
  - 确保 `device_set_texture` 函数能有效地处理高分辨率纹理。
  - 通过检查 `framebuffer` 和 `zbuffer` 中纹理的存储方式来优化内存管理。
- **实现多级纹理贴图（Mipmapping）:**

- 在纹理加载函数中引入多级纹理贴图，以提高性能并减少锯齿现象。这可以通过扩展 `device_set_texture` 函数为每个纹理生成多级纹理贴图来实现。

2) 高效的纹理管理

当前状态: 纹理管理相对简单，但系统可以通过更高效的管理方式得到改进。

改进步骤:

- **纹理缓存:**
  - 实现一个纹理缓存系统以重用已加载的纹理，避免重复加载。这可以通过在 `device_t` 结构中维护一个哈希图来实现。
- **纹理图集:**
  - 将多个小纹理组合到一个大纹理中，以减少纹理绑定次数并提高渲染性能。修改纹理管理函数以支持使用纹理图集。

3) 将原始指针更改为智能指针

当前状态: 项目使用原始指针来管理资源，如纹理和缓冲区。

改进步骤:

- **引入智能指针:**
  - 使用 `std::shared_ptr` 或 `std::unique_ptr` 替换原始指针。这一改变将自动管理内存，减少内存泄漏和悬空指针的风险。
- **重构代码库:**
  - 更新代码库中相关部分以使用智能指针，确保与现有逻辑兼容。这包括重构处理资源分配和释放的函数和类。

2. 添加材质支持

1) 扩展材质结构

当前状态: 项目具有对材质的基本支持，可能包括漫反射和镜面反射纹理。

改进步骤:

- **定义材质属性:**
  - 扩展材质结构，包含额外的属性，如环境光颜色、自发光颜色、光泽度和透明度。
- **更新材质类:**
  - 修改现有的材质类以处理这些新属性，包括适当的设置器和获取器。确保现有的材质相关函数也能管理这些新属性。

2) 更新着色器程序

当前状态: 着色器处理基本的光照和纹理。

改进步骤:

- **增强着色器输入:**
  - 更新顶点和片段着色器以接受新的材质属性作为输入。这涉及修改着色器程序以包含 `ambient_color`、`emissive_color`、`shininess` 和 `transparency` 等uniform变量。

- **修改光照计算:**
  - 调整着色器中的光照计算以考虑新的材质属性，包括：
    - 使用 `ambient_color` 进行环境光照计算。
    - 使用 `emissive_color` 进行自发光计算。
    - 修改镜面光照计算以使用 `shininess` 属性。

3) 实现材质管理

当前状态: 材质使用简单的数组或列表。

改进步骤:

- **材质库:**
  - 创建一个材质类来管理不同的材质及其属性。这可以通过扩展现有的 `device_t` 结构中的数组来管理新的材质结构。
- **材质绑定:**
  - 实现将材质绑定到对象的函数。这涉及在绘制每个对象之前在着色器中设置适当的材质属性。

3. 引入 camera 支持

1) 定义摄像机结构

当前状态: 项目支持基本的变换，但缺乏专门的摄像机系统。

改进步骤:

- **创建摄像机类:**
  - 定义 `camera_t` 表示摄像机，包括位置、方向、上向量、视场（FOV）、纵横比、近裁剪面和远裁剪面等属性。
- **初始化摄像机:**
  - 实现初始化和配置摄像机的函数。

2) 实现视图和投影矩阵

当前状态: 项目处理变换，但未明确定义视图和投影矩阵。

改进步骤:

- **计算视图矩阵:**
  - 实现函数以使用摄像机的位置、目标和上向量计算视图矩阵。
- **计算投影矩阵:**
  - 实现函数以使用摄像机的视场、纵横比、近裁剪面和远裁剪面计算投影矩阵。

3) 将摄像机集成到渲染管线中

当前状态: 渲染管线使用变换，但未针对摄像机进行专门处理。

改进步骤:

- **更新着色器:**
  - 修改顶点着色器以使用视图和投影矩阵，将顶点从世界空间变换到屏幕空间。
- **将摄像机绑定到渲染管线:**

- 确保在渲染之前将摄像机的视图和投影矩阵传递到着色器。

二、提升项目性能

1. 优化渲染性能

1) 使用OpenCL进行渲染批处理

- **当前状态:** 每个对象各自发出绘制调用。
- **实现方法:**
  - 设置OpenCL环境，包括初始化上下文、命令队列和设备。
  - 将对象按材质和纹理分组，以便在批处理中一起处理。
  - 将组合后的顶点数据和索引数据传输到OpenCL缓冲区。
  - 执行OpenCL内核来处理顶点和片段数据。
  - 从OpenCL缓冲区读取处理后的数据并渲染最终图像。

2) 优化纹理使用

当前状态: 项目使用基本的纹理管理

改进步骤:

- **使用纹理图集:**
  - 将多个小纹理组合到一个大纹理（纹理图集）中，以减少纹理绑定次数。这涉及更新纹理坐标和修改纹理加载函数。

3) 高效的内存管理

当前状态: 项目使用原始指针进行资源管理。

改进步骤:

- **使用智能指针:**
  - 用智能指针（`std::shared_ptr` 或 `std::unique_ptr`）替换原始指针，以自动化内存管理并防止内存泄漏。

三、提高代码质量

1. 代码重构

采用面向对象的结构，以下列出主要的类：

- **Application**: 用于管理渲染器的生命周期和主循环
- **Window**: 用于创建和管理窗口和处理用户输入。
- **Renderer**: 用于实现光栅化算法和渲染逻辑，在软件渲染器中，这将包含主要的光栅化和绘制函数。
- **Shader**: 在软件渲染器中，着色器逻辑可以用C++函数来模拟，处理光照和材质计算。
- **Light**: 用于定义场景中的光源属性，如点光源、方向光和环境光。
- **Texture**: 用于管理纹理数据，可以实现纹理采样函数，以支持纹理映射。

- **Material**: 用于定义材质属性，如漫反射、镜面反射和环境反射。
- **Mesh**: 用于存储和管理网格的顶点、法线、纹理坐标等信息。
- **Model**: 用于管理包含多个网格的3D模型。
- **Camera**: 用于定义摄像机的视角和位置，并计算视图矩阵和投影矩阵。
- **CameraController**: 用于处理用户输入并更新摄像机的位置和方向。
- **Data**:用于存储和管理顶点数据、索引数据等渲染所需的数据。

2. 添加测试

引入 camera 支持

当前状态: 项目中缺少摄像机系统和 MVP（模型-视图-投影）变换的支持。

改进步骤:

- 定义 Camera 类: 实现一个 Camera 类，用于管理摄像机的视角、位置和方向，并计算视图矩阵和投影矩阵。
- 实现 MVP 变换: 在渲染管线中集成模型矩阵、视图矩阵和投影矩阵的计算和应用，使得每个顶点的变换都能够考虑摄像机的视角。
- 更新着色器: 修改顶点着色器以接受 MVP 矩阵，并应用这些矩阵进行顶点坐标变换。

支持用户输入响应

当前状态: 项目中缺少用户输入响应的支持，无法通过键盘和鼠标控制摄像机和其他交互元素。

改进步骤:

- 实现输入处理: 使用一个输入处理类或在现有的渲染循环中添加输入处理逻辑，支持键盘和鼠标输入。
- 摄像机控制: 通过输入处理更新摄像机的位置和方向，实现基本的摄像机移动和视角旋转。
- 其他用户交互: 支持其他可能的用户输入，如切换渲染模式、开启/关闭特效等。

四、完善文档支持

1. 完善项目文档

- 详细的API文档: 提供每个函数和类的详细说明，帮助开发者理解和使用代码。
- 开发指南: 编写开发指南，介绍项目的设计理念和使用方法。

2. 添加中英双语文档

- 双语支持: 提供中英双语文档，方便更多开发者理解和贡献项目

3. 设计多章节文档用于学习

- 分章节的学习文档: 参考项目 tiny-renderer 设计多章节教程，循序渐进地介绍项目的各个部分，帮助初学者系统学习

项目时间计划

1. 代码重构（8天） 2024/07/01 - 2024/07/08
2. 完善贴图支持（8天） 2024/07/09 - 2024/07/17
3. 添加材质支持（8天） 2024/07/18 - 2024/07/26
4. 引入 camera 支持（9天） 2024/07/26 - 2024/08/03
5. 添加测试（5天） 2024/08/04 - 2024/08/09
6. 优化纹理使用（5天） 2024/08/10 - 2024/08/15
7. 高效的内存管理（5天） 2024/08/15 - 2024/08/20
8. 使用OpenCL进行渲染批处理（10天） 2024/08/20 - 2024/08/30
9. 完善项目文档（5天） 2024/08/31 - 2024/09/04
10. 添加中英双语文档（5天） 2024/09/05 - 2024/09/09
11. 设计多章节文档用于学习（7天） 2024/09/10 - 2024/09/16
12. 配合 SimpleGameEngine 完成测试（7天） 2024/09/17 - 2024/09/23
13. 预留时间（7天） 2024/09/24 - 2024/09/30