河北师范大学

本科生毕业论文（设计）

题目：《万维望远镜》渲染引擎的分析研究

学生姓名：王轩宇

指导教师：王威

学　　院：计算机与网络空间安全学院

专　　业：计算机科学与技术

年　　级：2019 级

完成日期：2023 年 4 月 23 日

学位论文原创性声明

本人所提交的学位论文《万维望远镜》渲染引擎的分析研究，是在导师的指导下，独立进行研究工作所取得的原创性成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中标明。

本声明的法律后果由本人承担。

论文作者（签名）：王轩宇　　指导教师确认（签名）：王威

2023 年 5 月 21 日　　2023 年 5 月 21 日

学位论文版权使用授权书

本学位论文作者完全了解河北师范大学有权保留并向国家有关部门或机构送交学位论文的复印件和磁盘，允许论文被查阅和借阅。本人授权河北师范大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其它复制手段保存、汇编学位论文。

论文作者（签名）：王轩宇　　指导教师（签名）：王威

2023 年 5 月 21 日　　2023 年 5 月 21 日

摘　要

万维望远镜(Worldwide Telescope)作为一款集合了全世界海量天文数据的可视化开源软件，它在天文观测和天文数据可视化有着广泛作用。然而，作为一款开源软件，万维望远镜渲染引擎存在技术文档缺失、注释不全和代码可读性差等问题，阻碍了开发人员对其渲染引擎进行二次开发。因此，本研究的研究目的是增强渲染引擎的代码可读性，从而方便开发人员进行二次开发。

由于渲染引擎源代码是使用 C#代码编写的，本研究主要采用 Visual Studio 和 Reshaper 分析源代码，运用软件工程和计算机图形学的理论知识，来剖析渲染引擎。具体工作流程为：首先分析渲染引擎的渲染流程，然后划分出渲染引擎的主要功能模块，最后找出各个功能模块的调用关系、层次结构和主要功能。

本研究完成了渲染引擎源代码的注释和技术文档的补全工作。通过对代码的深入分析，理清了渲染流程，深入挖掘了主要功能模块的调用关系、层次结构和重要功能。不仅提升了代码的可读性，也为开发人员更好地对渲染引擎进行二次开发打下了基础。

关键词：　万维望远镜　渲染引擎　软件工程　计算机图形学　C#

Abstract

Worldwide Telescope is an open source visualization software that collects massive astronomical data from all over the world. It plays a wide role in astronomical observation and astronomical data visualization. However, as an open source software, there are some problems such as missing technical documentation, incomplete comments and poor code readability, which hinder developers from secondary development of its rendering engine. Therefore, the purpose of this study is to enhance the code readability of the rendering engine, so as to facilitate the secondary development of developers.

Because the source code of the rendering engine is written by C# code, this study mainly analyzes the source code by Visual Studio and Reshaper, and applies the theoretical knowledge of software engineering and computer graphics to analyze the rendering engine. The specific working process is as follows: Firstly, the rendering process of the rendering engine is analyzed, and then the main functional modules of the rendering engine are divided. Finally, the calling relationship, hierarchical structure and main functions of each functional module are found out.

This study completed the annotation of the source code of the rendering engine and the completion of technical documents. Through the in-depth analysis of the code, the rendering process is clarified, and the calling relationship, hierarchy and important functions of the main functional modules are dug in depth. Not only does it improve the readability of the code, but it also lays a foundation for developers to better develop the rendering engine.

Key words: Worldwide Telescope rendering engine software engineering computer graphics C#

目 录

《万维望远镜》渲染引擎的分析研究

第 1 章 绪论

1.1 研究背景

随着世界天文学研究以及天文观测设备的进步与发展，接连诞生了许多更先进的观测技术和观测设备，现代天文学家可以借助这些技术和设备获得更多天文观测数据。由于世界上天文台、天文望远镜、天文观测设备和探测器数量繁多，其产生的天文观测数据的数据量大且分布不均匀，导致天文学家不能便捷的获取天文观测数据。所以天文学家急需一款有效的数据平台，可以统合世界上各大天文台、天文望远镜、观测设备和探测器的科学观测数据。从而有效地将世界上的科学观测数据进行统合并归档汇集到统一的天文数据库。

2008 年微软研究院推出了万维望远镜（WorldWide Telescope），其首次公开发布以来，已经被来自全球各国的职业和业余天文学家下载了数百万次[8]。万维望远镜就是一个集合了全世界海量天文数据的数据可视化平台，它将世界上各大天文台、天文望远镜、观测设备和探测器的科学数据都统合在了一起。

作为一款天文数据可视化软件，其渲染引擎主要负责将天文数据中的天体模型以真实、准确和逼真的方式绘制在屏幕上，帮助科研人员更直观的了解天文数据。因此，渲染引擎是万维望远镜数据可视化的核心。

1.2 研究现状

经过访问 GitHub 网站，发现微软已经将万维望远镜的代码仓库开源，并允许开发人员在此基础上进行二次开发[11]。浏览仓库参数发现，万维望远镜是使用 C#语言结合 SharpDX 包进行开发的。但是，在将其代码拉取到本地后，发现其技术文档和注释并不完备。通过访问万维望远镜社区，发现近几年并未有组织和个人补全万维望远镜的技术文档和代码注释。直观来讲，万维望远镜仅获得了 176 人的关注，以及 57 位开发者的维护[11]。

为了了解是否存在相同的问题，需要对其他天文数据可视化功能的开源软件进行访问。通过查询得知，Celestia 也是一款开源的天文数据可视化平台，采用 C++语言结合 OpenGL 进行开发[12]。经过将其代码拉取到本地，发现其技术文档和注释十分完备，具有较强的可读性。通过访问 Celestia 社区，发现开发者和社区成员积极贡献 Celestia 的

代码仓库。直观来讲，Celestia 获得了 1300 人的关注，以及 241 位开发者参与维护[12]。

1.3 研究问题和目的

经过对万维望远镜和 Celestia 项目的比较分析，发现万维望远镜渲染引擎技术文档缺失、重要模块注释不完善和代码可读性较差等因素，确实给开发人员带来了二次开发难度过大的问题。

因此，本研究的目的在于通过对万维望远镜渲染引擎进行系统性分析，达到两个目标：补全渲染引擎中重要着色器和渲染流程的技术文档，将代码注释率提高到软件工程推荐的 20%以上。完成以上目标就能解决开发者在进行渲染引擎二次开发时难度过大的问题，进而促进更多开发人员参与万维望远镜渲染引擎的二次开发。

1.4 研究内容

为了达到补全缺失的技术文档和提高代码注释率的目标，需要采取以下方法：首先，通过阅读有关文献，全面地了解渲染的基本知识、软件开发流程和万维望远镜的基本概况；其次，运用定性分析法，结合软件工程的理论知识对万维望远镜渲染引擎部分进行系统性分析，包括渲染引擎的需求分析和功能分析、梳理渲染引擎的渲染流程、划分渲染引擎的主要模块和分析各个主要模块的调用关系、层次结构和模块功能；最后，通过实验法，对渲染引擎进行黑盒测试，从而验证渲染引擎分析的可靠性。

1.5 研究方法

1.5.1 文献研究法

通过参考 DirectX 技术文档、相关书籍，以及 SharpDX 技术文档、软件工程相关书籍、关于万维望远镜的已有文献、万维望远镜官方社区网站等，辅助对万维望远镜渲染引擎进行系统性分析。

1.5.2 定性分析法

结合软件工程的理论知识，对天文数据可视化软件的需求、功能、架构、模块等进行深入剖析和分析，通过构建软件的功能模型、流程图、层次结构等工具，揭示软件内在的结构、功能和关联，从而了解软件的设计思想、工作原理和实现机制。

1.5.3 实验法

在本研究中，主要通过控制输入的天体参数，测试行星着色器的渲染效果，并分析渲染引擎输出的 HLSL 代码。通过观察渲染效果来评估行星着色器代码的正确性，并通过阅读渲染引擎输出的 HLSL 代码来验证行星着色器的可靠性，从而验证对渲染引擎分

析的准确性。

1.6 论文结构

论文的第一章是绪论，主要包括对本研究的研究背景、研究现状、研究问题和目的、研究内容和研究方法进行阐述。第二章主要进行渲染引擎的功能分析。包括万维望远镜渲染引擎的需求分析、数据建模、功能建模和行为建模。第三章主要介绍渲染引擎的渲染流程，包括创建渲染上下文、读取纹理、创建着色器和绘制几何图形。此外，还将对渲染引擎的渲染流程和理论渲染流程进行对比。第四章将主要分析渲染引擎的功能模块，首先划分出各个功能模块，然后对每个模块的层次结构进行分析。随后，将对渲染上下文、纹理和着色器模块进行系统性的功能分析，包括绘制类图和流程图。最后，第五章将对渲染引擎进行功能测试，主要包括对行星着色器进行黑盒测试，并根据测试结果得出结论。

第 2 章 渲染引擎的功能分析

2.1 渲染引擎的需求分析

万维望远镜作为一款天文数据可视化软件，其主要功能就是对天文数据进行可视化展示，以便用户能够直观地观察和分析天体的位置、形态、运动等信息。根据以上信息，其渲染引擎应包含以下功能：

实时渲染，即渲染引擎能够将天体数据绘制到场景中，且实时生成天体的渲染效果，以便用户能够在软件中实时地观察天体的变化和运动。

符合规定的渲染效果，即能够根据用户设备情况和天体属性，生成对应质量的渲染效果，包含光照效果、阴影效果、遮蔽效果和自发光等效果，以便用户能够更真实地观察和分析天体特性。

高性能渲染，即具备处理大规模的天文数据的能力，并通过尽量降低本地纹理数据读取时间和多采用预编译的着色器来降低生成渲染效果的时间，以确保在不降低渲染质量的情况下保持实时的渲染效果。

2.2 建立渲染引擎的数据模型

通过阅读源码发现，万维望远镜的渲染引擎核心部分在 WWTCore 下，其主要功能是由 RenderContext11.cs、Shaders.cs、Shaders11.cs 和 Texture11.cs 实现的。

为了对渲染引擎的数据进行抽象和描述，从而更好地理解和利用数据，进而了解渲

染引擎的工作方式，需要确定渲染引擎各个实体之间的关系。

通过阅读和分析渲染引擎的源代码，确定了渲染引擎核心部分的实体，属性和关系。其实体，属性和关系的描述如下。

（1）外部存储区实体

外部存储区的属性：存储区地址(下划线代表键)、本地位图、本地纹理数据和本地预编译 HLSL 代码。

外部存储区的关系：外部存储区中存储了多个纹理数据和位图数据，这些数据在渲染引擎中统称为“纹理数据”，外部存储区可以获得多个纹理库的生成纹理数据；外部存储区中还存储了本地预编译的 HLSL 代码，外部存储区可以获得着色器编译器传入的 HLSL 代码。

（2）纹理库实体

纹理库的属性：纹理库 ID、纹理信息和纹理库大小。

纹理库的关系：多个纹理库通过访问多个外部存储区来获得纹理信息；纹理库可以为渲染上下文提供多个纹理数据。

（3）着色器编译器实体

着色器编译器的属性：着色器键、HLSL 代码、索引数据、顶点数据。

着色器编译器的关系：着色器编译器可以获得多个本地预编译 HLSL 代码数据；着色器编译器可以获取顶点缓冲区、索引缓冲区、常量缓冲区中多个数据；通过编译后的着色器会生成二进制代码，着色器编译器可以向着色器库输出多个着色器键和其对应的二进制代码信息；着色器编译器可以获取渲染上下文的纹理信息、采样器参数和着色器参数。

（4）渲染上下文实体

渲染上下文的属性：交换链信息、设备信息、功能水平、输入布局、采样器参数信息、着色器参数信息和纹理信息。

渲染上下文的关系：渲染上下文可以获取多个纹理库中的纹理信息；渲染上下文可以获取编译器中的纹理信息、采样器参数和着色器参数。

（5）着色器库实体

着色器库的属性：着色器库 ID、着色器键、着色器二进制代码信息和着色器库大小。

着色器库的关系：着色器库可以向着色器编译器提供着色器键和其对应的二进制代

码信息。

（6）顶点缓冲区实体

顶点缓冲区的属性：顶点缓冲区 ID、顶点缓冲区大小和顶点数据。

顶点缓冲区的关系：顶点缓冲区可以向着色器编译器提供顶点信息。

（7）索引缓冲区实体

索引缓冲区的属性：索引缓冲区 ID、索引缓冲区大小和索引数据。

索引缓冲区的关系：索引缓冲区可以向着色器编译器提供索引信息。

（8）常量缓冲区实体

常量缓冲区的属性：常量缓冲区 ID、常量缓冲区大小和常量数据。

常量缓冲区的关系：常量缓冲区可以向着色器编译器提供常量数据。

根据以上对渲染引擎的实体、属性和关系的描述，从而绘制出渲染引擎的实体-关系图。如图 2-1 所示为渲染引擎的实体-关系图。

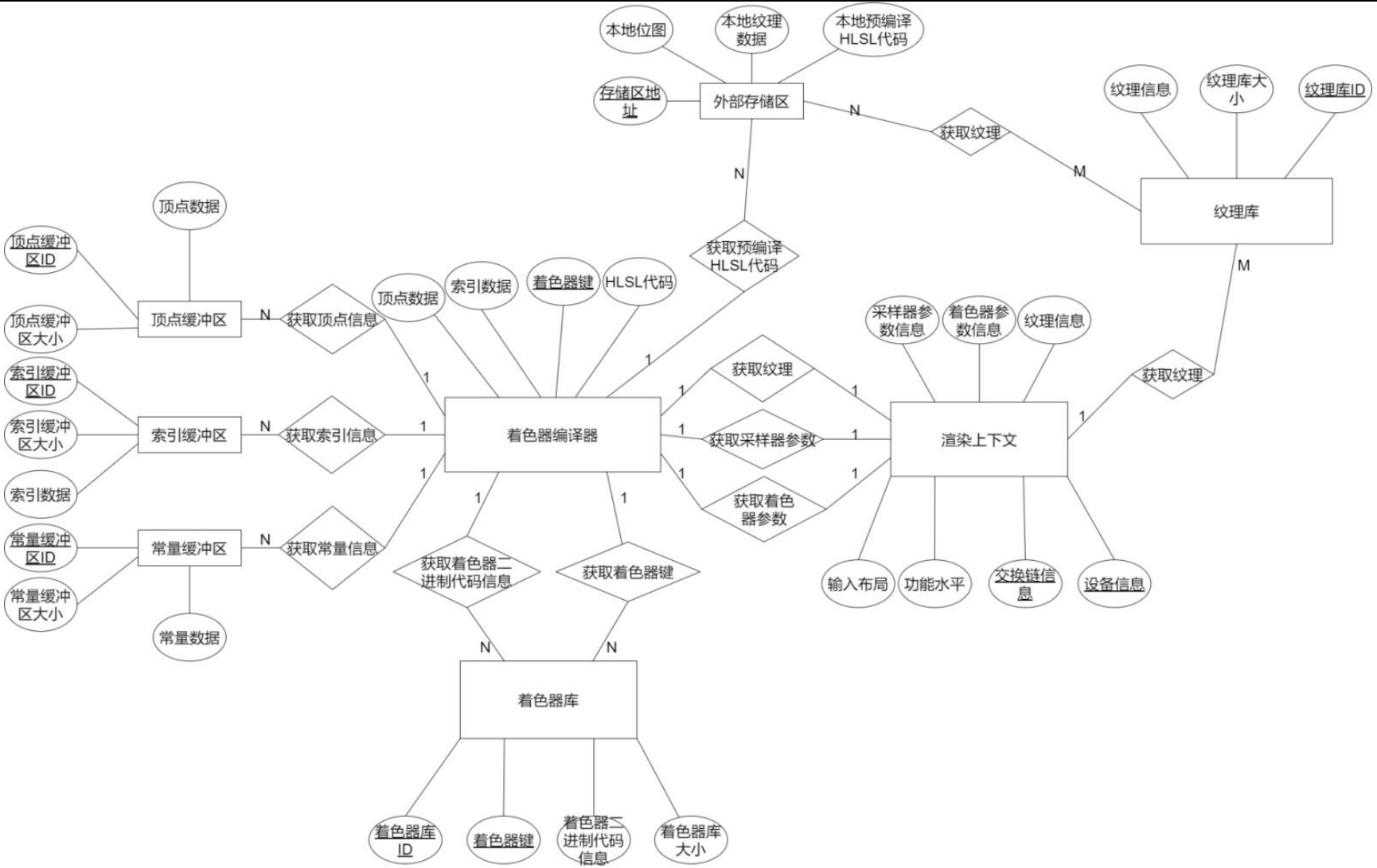
本地位图
本地纹理数据
本地预编译HLSL代码
存储区地址
外部存储区
N
获取纹理
M
纹理信息
纹理库大小
纹理库ID
纹理库
N
获取预编译HLSL代码
M
顶点数据
顶点缓冲区ID
顶点缓冲区大小
顶点缓冲区
N
获取顶点信息
顶点数据
索引数据
着色器键
HLSL代码
1
索引缓冲区ID
索引缓冲区大小
索引数据
索引缓冲区
N
获取索引信息
1
着色器编译器
1
获取纹理
1
采样器参数信息
着色器参数信息
纹理信息
获取纹理
1
1
获取采样器参数
1
渲染上下文
1
常量缓冲区ID
常量缓冲区大小
常量缓冲区
N
获取常量信息
1
1
获取着色器参数
1
1
1
获取着色器二进制代码信息
获取着色器键
输入布局
功能水平
交换链信息
设备信息
常量数据
N
N
着色器库
着色器库ID
着色器键
着色器二进制代码信息
着色器库大小

图 2-1 实体-关系图

2.3 建立渲染引擎的功能模型

在建立了渲染引擎数据模型之后，为了能够对渲染引擎的功能进行抽象和描述，从而更好地理解和渲染引擎的功能需求和设计，需要建立渲染引擎的功能模型。具体来说，需要详细的确定渲染引擎中信息流和数据从输入移动到输出的过程中所经过的变换，即需要绘制渲染引擎的数据流图。

首先，需要绘制渲染引擎的顶层数据流图，其中数据输入源为存储器，数据输出源为用户界面。存储器包括外部存储器和内部存储器，其存储了待渲染的顶点数据、索引数据和纹理数据。这些数据需要输入到渲染引擎中进行处理。渲染引擎的主要作用是接收顶点数据、索引数据和纹理数据，并通过计算机图形渲染将它们转换成图像。最终，用户界面会接收由渲染引擎渲染出的图像，并呈现给用户。如图 2-2 所示，为渲染引擎的顶层数据流图。

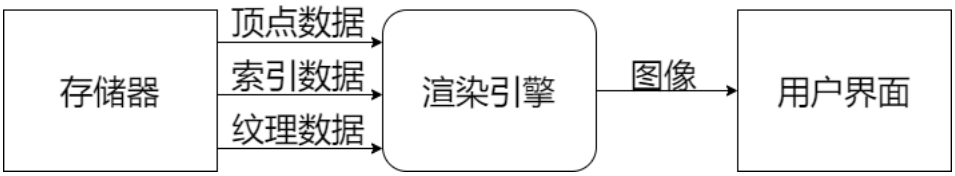

图 2-2 渲染引擎的顶层数据流图

接下来，根据顶层数据流图，需要画出中层数据流图。中层数据流图将渲染引擎的功能进一步细化，分解出多个功能、数据存储和数据流。存储器中存储的顶点数据会被顶点缓冲区读取、索引数据会被索引缓冲区读取，而纹理数据则会存储在纹理库中。渲染上下文会获取纹理库中的纹理，通过纹理指定着色器参数。同时，渲染上下文根据传入纹理的参数，指定着色器键和 HLSL 代码。之后，渲染上下文会将 HLSL 代码传递给着色器编译器，着色器键传递给着色器库中。在编译着色器时，首先需要从数据存储中获取顶点数据、纹理数据、索引数据和常量数据。然后，编译渲染上下文传入的 HLSL 代码。编译完成后，将生成的着色器二进制代码存储在着色器库中。与此同时，用户界面将接收由渲染引擎渲染出的图像，并呈现给用户。如图 2-3 所示，为渲染引擎的中层数据流图。

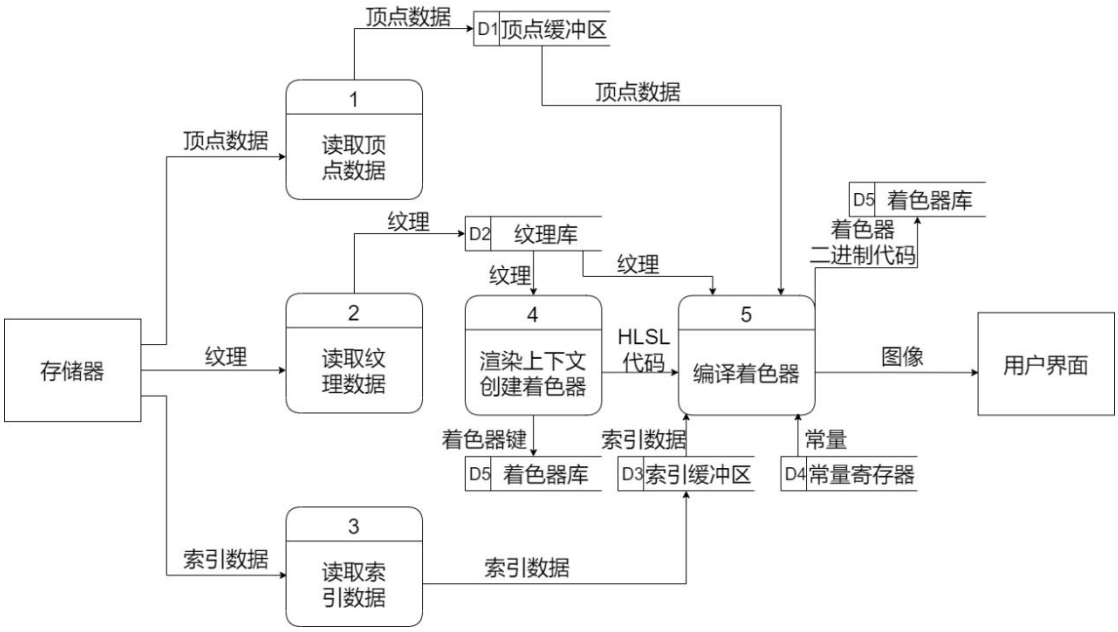

图 2-3 渲染引擎的中层数据流图

最后，根据中层数据流图，为了进一步细化各个功能和信息流需要绘制底层数据流图。首先，渲染上下文会解析输入纹理样式，得出着色器样式，从而生成着色器键。之后，渲染上下文根据着色器键生成 HLSL 代码。着色器编译器将根据 HLSL 代码编译，然后汇编出一段着色器二进制代码。接下来，着色器二进制代码、顶点数据、纹理、索引数据和常量数据会交给 GPU 运算，从而计算出每个像素的颜色和属性。最终，每个像素的颜色和属性会被 GPU 组合成为最后的图像，并且用户界面将接收由渲染引擎渲染出的图像并呈现给用户。如图 2-4 所示，为渲染引擎的底层数据流图。

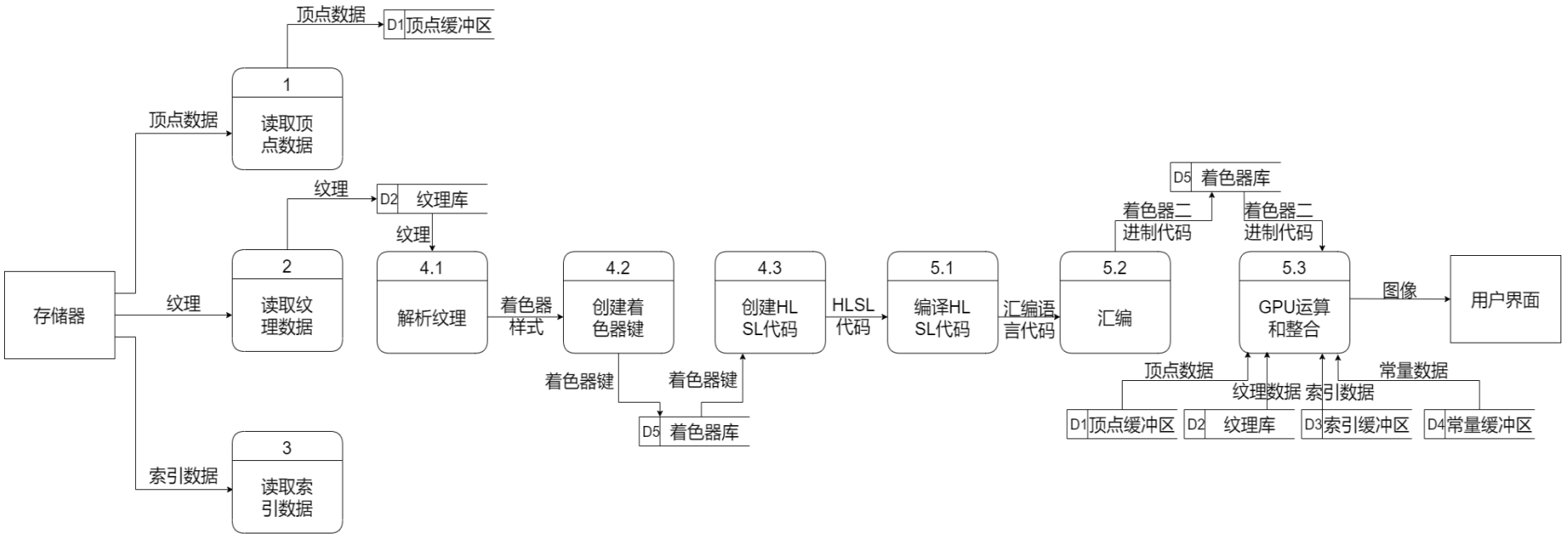

图 2-4 渲染引擎的底层数据流图

2.4 建立渲染引擎的行为模型

建立行为模型的意义在于能够对渲染引擎的行为进行描述、分析和预测，从而更好地理解和规划渲染引擎行为模式和行为结果。为了更加详细的确定渲染引擎中关键的状态变化过程，以便了解渲染引擎对各个事件的响应方式，进而清晰的描述渲染引擎状态之间的转换顺序和状态之间的关系，需要绘制渲染引擎的状态转换图。

首先列出渲染引擎的关键状态，包括：初态、等待纹理、读取纹理、创建着色器键、创建 HLSL 代码、预编译状态、编译代码、读取缓冲区、渲染状态、等待纹理和终态。接下来，需要理清状态之间的转换关系。

若渲染引擎未收到初态发来的纹理信息，则等待读取纹理指令，此时渲染上下文的当前状态为待机状态。当渲染引擎收到了读取纹理的指令，其状态转换到读取纹理状态，此时渲染上下文开始读取存储器中的纹理数据。当读取结束，如果收到设定着色器参数的指令，则渲染引擎进入创建着色器键的状态，此时渲染上下文获取输入值并创建着色器键。

渲染引擎首先会按照着色器键查找存储器是否存在与之对应的预编译 HLSL 代码。如果渲染引擎从本地获取了与着色器键相同的HLSL代码，则会进入预编译状态。反之，如果没有获取与着色器键相同的 HLSL 代码信息，则会进入创建 HLSL 代码状态。在创建 HLSL 代码状态下，渲染上下文会获取输入值，创建 HLSL 代码，并发送编译命令。在预编译状态下，渲染上下文会获取本地预编译 HLSL 代码，并发送编译命令。着色器编译器在接收到编译命令和 HLSL 代码后，会将渲染引擎的状态调整为编译代码状态。这时，着色器编译器编译输入的 HLSL 代码。

渲染引擎在接收到编译器输出的渲染指令和着色器二进制代码后，会将渲染引擎调整为渲染状态。渲染引擎首先向缓冲区发送读取指令。缓冲区在收到读取指令后，会读入渲染所需的基本数据，并发出输入基本数据命令给 GPU。GPU 在接收到输入基本数据命令后，会读入基本数据，并等待数据准备就绪后计算每个像素的颜色和属性。当 GPU 工作完成后，会向渲染引擎发出终止指令。渲染引擎进入等待纹理状态，若长时间未响应，则进入终态。如图 2-5 所示，为渲染引擎的状态转换图。

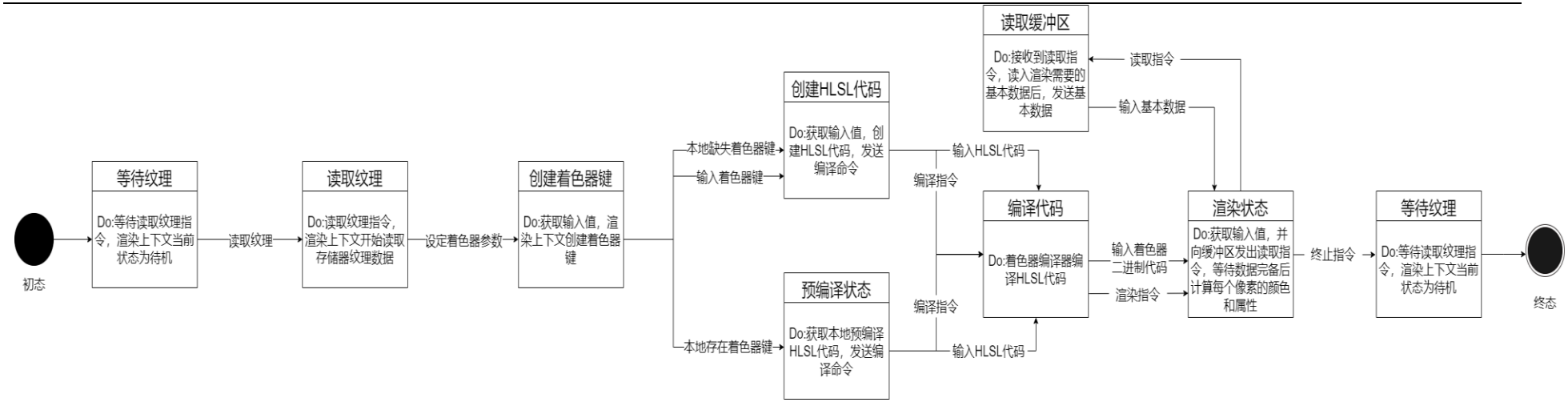

图 2-5 渲染引擎的状态转换图

第 3 章 渲染引擎的渲染流程

3.1 万维望远镜实现的渲染流程

3.1.1 创建渲染上下文

万维望远镜的渲染上下文实现创建交换链、创建设备和设备上下文、创建渲染目标视图、创建深度/模板缓冲区、创建混合模板状态和设置视窗的功能。与 Direct3D 给出的理论渲染方式相同的是，在创建交换链、创建设备和设备上下文、顶点缓冲区和索引缓冲区和常量缓冲区，万维望远镜采用了相同的方法。但在创建渲染目标视图、创建深度/模板缓冲区、创建混合模板状态和设置视窗方面，万维望远镜采用了不同的方法。

为了实现这些功能，万维望远镜的渲染上下文创建了多个类和枚举类型，其中包括混合模板状态、深度模板方式、光栅器模板方式和材质类型以及它们对应的状态数组。例如，在设定深度模板方式时，渲染引擎首先会初始化当前深度模板方式、标准深度模板方式数组和深度模板视图接口。然后，渲染引擎会根据当前渲染方式和设备水平指定深度模板的格式，并将其赋值给对应的深度模板视图接口。确定使用枚举类型的哪一状态。然后渲染引擎指定深度模板的格式，并赋值给对应的深度模板视图接口。具体来说，深度模板视图接口在深度模板测试期间访问纹理资源，在输出合成器中，将渲染目标和深度/模板缓冲区组合起来形成渲染目标集，需要同时设置两者的渲染目标视图和深度模板视图。渲染目标视图可以绑定到设备上，以便在渲染时使用。最后将深度模板视图接口指定给输出合成器上，并释放深度模板视图接口资源。混合模板状态、光栅器模板方式和材质类型与深度模板方式相同。

3.1.2 读取纹理

渲染引擎的纹理类包括纹理标识、2D 纹理以及访问资源中的数据的着色器资源视图。纹理读取本地文件分为两种类型：一类是直接读取图像文件，一类是以流的形式读取内存流的位图文件。

首先为直接读取图像文件部分：渲染引擎获取文件扩展名以确定文件格式。根据设备性能和用户需求，决定是否将纹理格式转换为 SRGB 格式。接着，初始化图像加载信息以控制从文件加载纹理的方式。使用 SharpDX 的方法读取本地图像。将创建好的纹理传递到渲染上下文中，并释放内存资源。

然后是以流的形式读取内存流的位图文件：首先将内存流中的位图转换为 PNG 格式，并将 PNG 数据写入内存流中，设置内存流的位置指针到开始处。接着，判断位图

的宽和高是否为 2 的整数次幂，以决定使用哪种纹理类型。如果宽和高均为 2 的整数次幂，则使用原始的纹理图像。如果宽或高不是 2 的整数次幂，表示需要生成一张缩小一半的图像。接下来，根据上面设置的图像加载信息和内存流的图像数据，创建一个 2D 纹理，并将其作为参数传递给纹理构造函数。将创建好的纹理传递到渲染上下文中，并释放内存资源。

3.1.3 创建着色器

渲染引擎设定了很多着色器类型，大致可以分为两种，分别为行星着色器和简单着色器。

首先介绍如何创建行星着色器。当用户需要创建行星着色器时，渲染上下文会按照用户需求创建行星着色器键，并设置行星着色器键的基本效果、行星表面样式、行星表面纹理映射方式以及行星着色器类型。行星着色器键的主要功能是指定行星着色器的功能和生成 HLSL 代码的唯一标识符。在编译 HLSL 代码之前，行星着色器类首先会检查生成的行星着色器键是否与内存中着色器库的着色器二进制代码或本地存在的预编译 HLSL 代码的功能相同。如果相同，则渲染引擎直接执行着色器库中的着色器二进制代码或编译本地的 HLSL 代码。如果没有检查出相同的行星着色器键，则根据行星着色器键生成 HLSL 代码，并进行编译以输出着色器二进制代码，然后渲染引擎执行二进制代码。最后，将行星着色器键及其对应的二进制代码保存到着色器库中，并将生成的 HLSL 代码保存到本地。

然后介绍如何创建简单着色器。当用户需要创建一个简单着色器时，渲染上下文会根据用户的需求创建着色器键。与行星着色器不同的是，简单着色器键是渲染引擎中的“固定”着色器，预编译的着色器缓存键只是类名。因此，创建的简单着色器不用指定其类型，只需生成对应功能的 HLSL 代码即可。后续与行星着色器方法相同。

3.1.4 绘制几何图形

这一部分是由渲染上下文完成的，与 Direct3D 提供的理论方法一致。当着色器编译完成后，指定绘制的索引以及索引缓冲区中的偏移量，最终会根据当前设置的渲染状态和着色器程序对顶点数据进行处理，生成像素数据并输出到后台缓冲区中进行显示。

3.2 万维望远镜渲染流程与理论渲染流程的比较

3.2.1 万维望远镜渲染流程的优点

首先从渲染上下文的设置来看，存在以下优点。第一点是灵活性高、兼容性强。渲染引擎会根据设备情况和用户需求，设定不同的渲染效果，从而能够保证让大多数设备

和用户都能使用。第二点是减少不必要的资源浪费。例如，在创建混合模板时，如果用户设置枚举类型的值为不指定混合，则渲染引擎不会进行深度混合、深度模板测试、光栅器剔除和指定材质。从而大大节省了渲染引擎的运行时间，提高了渲染速度。

从纹理角度来看，存在以下优点。读取纹理分为直接读取图像文件和以流的形式读取内存流的位图文件两种方式，保证了读取纹理的稳定性。在读取文件的过程中，如果存在异常，程序可以及时中断，从而降低了内存泄漏的风险。在转化位图格式时，根据用户的需求和设备情况指定图像加载信息，根据图像加载信息判断纹理数据格式是否兼容，从而增强了兼容性。

最后从着色器角度来看，存在以下优点。虽然着色器使用的渲染管线与 Direct3D 中的渲染管线相同，但是对着色器进行了分类和存储。在创建着色器时，渲染引擎会优先考虑本地是否存在预编译的 HLSL 代码，以及着色器库中是否有相同着色器键的二进制代码。这种方法可以显著性提高渲染引擎创建着色器和渲染的速度。

根据如上所述绘制表格如下，如表 3-1 展示了万维望远镜渲染流程的优点和描述。

表 3-1 万维望远镜渲染流程的优点

优点	描述
灵活性高、兼容性强	渲染引擎根据设备情况和用户需求，设定不同的渲染效果。
减少不必要的资源浪费	根据用户设备信息设定渲染效果，提高了渲染速度。
稳定的纹理读取	分为直接读取图像文件和以流的形式读取内存流的位图文件两种方式，保证了读取纹理的稳定性。
较强的纹理兼容性	在转化位图格式时，根据用户的需求和设备情况指定图像加载信息，根据图像加载信息判断纹理数据格式是否兼容，从而增强了兼容性。
提高着色器创建和渲染速度	在创建着色器时，渲染引擎会优先考虑本地是否存在预编译的 HLSL 代码，以及着色器库中是否有相同着色器键的二进制代码。

3.2.2 万维望远镜渲染流程的缺点

首先从渲染上下文的设置来看，存在以下不足。第一点是渲染上下文的低内聚性和高耦合性，即渲染上下文包含了多个理论渲染流程的功能实现，这些功能实现可能彼此之间并没有直接关系。当修改其中某一个功能时，很容易影响到其他的功能，导致代码修改和测试的复杂度增加。例如在创建设备时，若擅自修改属性值，可能会影响交换链的功能，从而影响渲染效果。第二点是代码膨胀，渲染上下文中包含多个功能实现，这些实现可能包含大量代码。当需要添加新的功能时，需要修改和大量的代码，导致代码的碰撞和维护困难。

从纹理角度来看，存在以下不足。第一点是性能问题，由于着色器获取纹理需要经过渲染上下文，多次读取使用纹理会增加 GPU 负载，从而降低性能。特别是在需要频繁访问纹理的场景下，如渲染地球实时地图时，需要进行大量的纹理采样和渲染操作。第二点时内存占用高，每次读取纹理都需要将其加载到内存中，如果纹理质量较大或需要多次读取，则会占用大量内存。

最后从着色器角度来看，主要存在一个问题，就是代码量过大。行星着色器需要实现多种光照效果、表面问题、高光效果和环境光遮蔽贴图等，导致其代码量过大，使得调试和维护的成本较高。

根据如上所述绘制表格如下，如表 3-2 展示了万维望远镜渲染流程的缺点和描述。

表 3-2 万维望远镜渲染流程的缺点

缺点	描述
低内聚性和高耦合性	渲染上下文包含了多个功能实现，这些功能实现可能彼此之间并没有直接关系。当修改其中某一个功能时，很容易影响到其他的功能，导致代码修改和测试的复杂度增加。
代码膨胀	渲染上下文中包含多个功能实现，这些实现可能包含大量代码。当需要添加新的功能时，代码修改难度过大。
性能问题	由于着色器获取纹理需要经过渲染上下文，多次读取使用纹理会增加 GPU 负载，从而降低性能。
内存占用高	每次读取纹理都需要将其加载到内存中，如果纹理质量较大或需要多次读取，则会占用大量内存。
着色器代码量过大	行星着色器需要实现多种效果时，导致其代码量过大，使得调试和维护的成本较高。

第 4 章 渲染引擎的功能模块分析

在第三章中，重点介绍了渲染引擎的渲染流程。在第四章中首先根据渲染流程划分了主要模块，并分析出它们的层次结构。接下来，将根据实现各个模块的类来深入了解这三个模块的功能和实现原理。

4.1 万维望远镜的功能模块及其层次结构

4.1.1 万维望远镜的功能模块

根据上述渲染引擎的渲染流程，我们可以将万维望远镜的功能模块划分为以下三个部分：渲染上下文、纹理和着色器。

4.1.2 功能模块的层次结构

根据上述功能模块和渲染流程，画出渲染引擎的层次结构图。如图 3-1 所示，为渲染引擎功能模块的一级层次结构图。

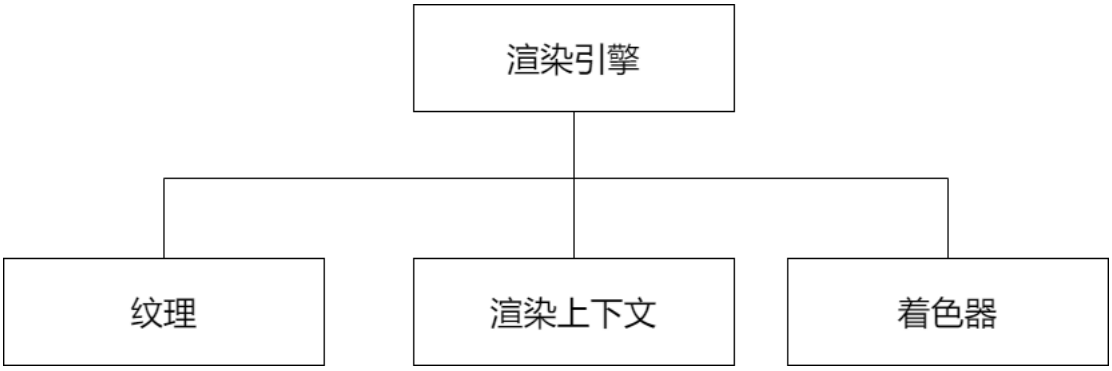

图 4-1 功能模块的一级层次结构图

根据功能模块的一级层次结构图，细化出渲染引擎的二级层次结构图。如图 3-2 所示，为渲染引擎功能模块的二级层次结构图。

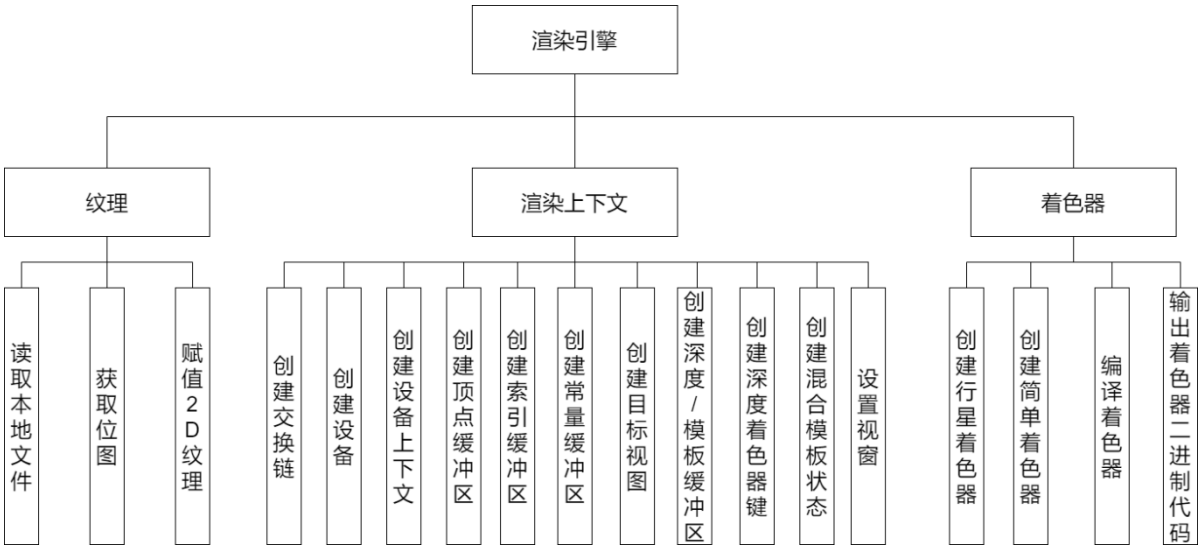

图 4-2 功能模块的二级层次结构图

4.2 渲染上下文

渲染上下文的主要功能是初始化渲染引擎的渲染环境，以及通过读取纹理数据来创建着色器键，从而指定着色器的功能。

渲染上下文是由 RenderContext11 类来实现的，如图 4-3 是该类的类图，类图展示了类中的主要方法和属性。下面将介绍该类的主要方法以及其实现原理。

（1）渲染上下文的构造函数

构造函数的功能是创建一个 Direct3D 的渲染环境，以便在 Windows Forms 控件上进行渲染。下面将描述该构造函数的实现过程。传入参量包括控件、使用 SRGB 编码标志和异常信息。

首先创建交换链，并指定交换链的描述信息，包括缓冲区个数、模块描述、是否窗口化、采样标准和使用方式。

然后，设定设备 Direct3D 版本信息，并根据交换链创建设备。如果创建失败，则表明 GPU 负载不能支持多重采样抗锯齿，因此不采用多重采样抗锯齿。

接下来，获取当前设备的即时上下文，并将准备设备指定为当前设备。即时上下文用于执行要立即提交到设备的呈现。对于大多数应用程序，即时上下文是用于绘制场景的主要对象。上述完成后，设定 Direct3D 的功能级别，主要是指定硬件设备的功能级别。如果设备低于目前指定的功能级别，则不启用 SRGB 编码，即 GPU 驱动程序不要对颜色进行任何颜色空间转换或 gamma 处理，直接使用颜色的原始值进行计算和输出。

接着，渲染上下文通过工厂将所有目前执行的窗口文件忽略，并将万维望远镜程序置于图层最上方。根据交换链创建后置缓冲区，然后根据后置缓冲区创建渲染目标视图。同样的，根据设备创建深度缓冲区，并指定缓冲区数组大小、格式、采样级别、宽度和高度。之后根据深度缓冲区创建深度模板视图。

根据当前版本，创建了一个采样器状态接口，其中包含了采样器状态的说明，可以绑定到管线的任何着色器阶段，供纹理采样操作引用。在默认采样器中，它使用 Clamp 采样模式对 UV 坐标进行采样，W 方向上使用 Wrap 采样模式。如果采样超出了[0,1]的范围并且设备上下文指定使用 Border 采样，那么就会使用 BorderColor 的颜色，这里设置为黑色。如果设备上下文指定使用位图，那么最大的细节层次为 float 类型的最大上限。如果当前的设备版本较低，则最大的细节层次为 16。

最后，设备上下文的像素着色器接口设置采样器并准备所有阶段。同时，初始化视口并将深度模板缓冲区和一组渲染目标绑定到输出合并。最后，调用初始化环境方法来

初始化万维望远镜渲染引擎使用的各种类型的所有状态块。

（2）设置显示渲染目标

该方法实现了设置显示渲染目标的功能。下面将描述该函数的实现过程。

首先通过设备上下文的“Flush”方法，将命令缓冲区的排队命令发送到 GPU 中。

然后判断是否存在渲染视图。如果存在渲染视图，则调用输出装配器清空输出合并阶段的目标视图和深度模板缓冲区。然后设置视窗为显示窗口，并通过调用输出装配器设置目标方法将深度模板缓冲区和渲染视图绑定到输出合并阶段。最后将当前的渲染目标是图设置为获取的渲染视图，当前的深度模板视图设置为获取的深度视图。

如果不存在渲染视图，而是有外部目标视图，则也先调用输出装配器清空输出合并阶段的目标视图和深度模板缓冲区。然后设置视窗为外部视窗，并通过调用输出装配器设置目标方法将深度模板缓冲区和渲染视图绑定到输出合并阶段。最后将当前的渲染目标视图设置为外部目标视图，当前的深度模板视图设置为外部深度视图。

（3）指定离屏渲染目标

该方法实现了指定离屏渲染目标的功能。以下将描述该函数的实现过程。

指定离屏渲染目标方法有两个重载，第一个重载传入了目标纹理和深度缓冲区对象，第二个重载传入了目标纹理视图、深度模板视图、视图宽度和视图高度。两个重载的实现大致相同，下面将介绍第一个重载的实现方式。

在第一个重载中。首先根据传入的参数设置当前目标视图和深度视图。如果深度缓冲区不为空或者使用外部投影，则当前深度视图指定为深度模板视图。然后，通过输出装配器清空输出合并阶段的目标视图和深度模板缓冲区，初始化视图。最后指定渲染目标和深度缓冲区。

（4）更新渲染器的视图变换矩阵、光照常量和阴影效果

首先判断是否需要更新渲染器渲染效果的标志，只有当视图变换矩阵、光照效果和阴影效果中有一个被修改了才会更新。

如果使用的行星着色器不为空且需要更新渲染器渲染效果，则首先获取当前坐标系的世界矩阵，并对其求逆矩阵。将太阳位置从世界坐标系转换为天球坐标系，并标准化太阳光方向向量，最后将其传递给着色器中的太阳光方向。

如果着色器键的光源个数大于 1，则计算反射光的方向向量并标准化，将其传递给着色器中的反射光方向向量。计算半球光方向向量并标准化，将其传递给着色器中的半球光方向向量。如果需要重新计算光照效果，则将光照颜色设置到着色器中，包括太阳

光、反射光和半球光。

（5）设置材质和光照效果

该方法通过传入的材质、颜色贴图、高光贴图、法线贴图和不通透率，设置行星材质及其纹理贴图，并返回一个用于绘制行星表面的着色器键。以下将描述该函数的实现过程。

根据材质中高光颜色的值是否为黑色，确定行星表面的类型，有颜色贴图、高光贴图和自发光三种类型。在禁用照明时，强制设定行星表面类型为自发光材质。根据反射光颜色的值是否为黑色，确定光源个数为 1 或 2 个。

根据传入的纹理贴图，确定行星表面的纹理映射方式，包括颜色贴图、高光贴图和法线贴图。设定行星着色器键对象，将行星表面类型和纹理映射方式设置为获取的表面类型和纹理映射方式。返回一个用于绘制行星表面的着色器对象。

设定着色器的漫反射颜色向量，其中 RGB 值为材质的漫反射颜色的 RGB 值除以 255.0f，Alpha 值等于材质不透明度乘以传入的不透明度。

如果行星表面类型为高光贴图或者镜面反射，设定着色器中的高光颜色向量和高光强度。根据传入的纹理贴图，设定着色器中的主要贴图、高光贴图和法线贴图资源视图。

（6）设置基本效果

该方法通过传入的基本效果、不透明度和颜色，设置着色器键的漫反射颜色向量，使其具有用户指定的颜色和不透明度。以下将描述该函数的实现过程。

首先定义纹理映射方式为自发光材质，并声明校正后的颜色向量，用于存储经过校正后的颜色向量值。

判断是否使用 SRGB 编码，如果是，则对 alpha 值进行 SRGB 校正，将不透明度的值取 2.2 次方，将传入参数颜色的 RGB 值除以 255.0f 后取 2.2 次方，alpha 值为不透明度，计算出校正后的颜色向量。如果不适用，则不需要进行 SRGB 校正。

根据输入的基本效果，确定基本效果的类型，并分别执行以下步骤：

如果是仅仅设定纹理：使用指定的不透明度创建行星表面效果，设定着色器的漫反射颜色向量为（1.0, 1.0, 1.0, opacity）。

如果仅仅设定颜色：着色器不使用纹理，使用指定的不透明度创建行星表面效果，设定着色器的漫反射颜色向量为修正后的颜色向量。

如果采用纹理和颜色：使用指定的不透明度创建行星表面效果，设定着色器的漫反射颜色向量为修正后的颜色向量。

如果使用颜色文字效果：使用 alpha 通道混合纹理，使用指定的不透明度创建行星表面效果，设定着色器的漫反射颜色向量为修正后的颜色向量。

如果参数基本效果不是上述四种情况，则将着色器和着色器键对应基本效果设为空。

（7）初始化渲染引擎使用的各种类型的所有状态块

该方法将初始化混合状态，深度模板状态，初始化光栅器状态和采样器状态。将当前光栅器状态初始化为剔除所有逆时针方向定义的三角形，并将当前光栅器状态初始化为标准光栅化状态数组中对应状态值。

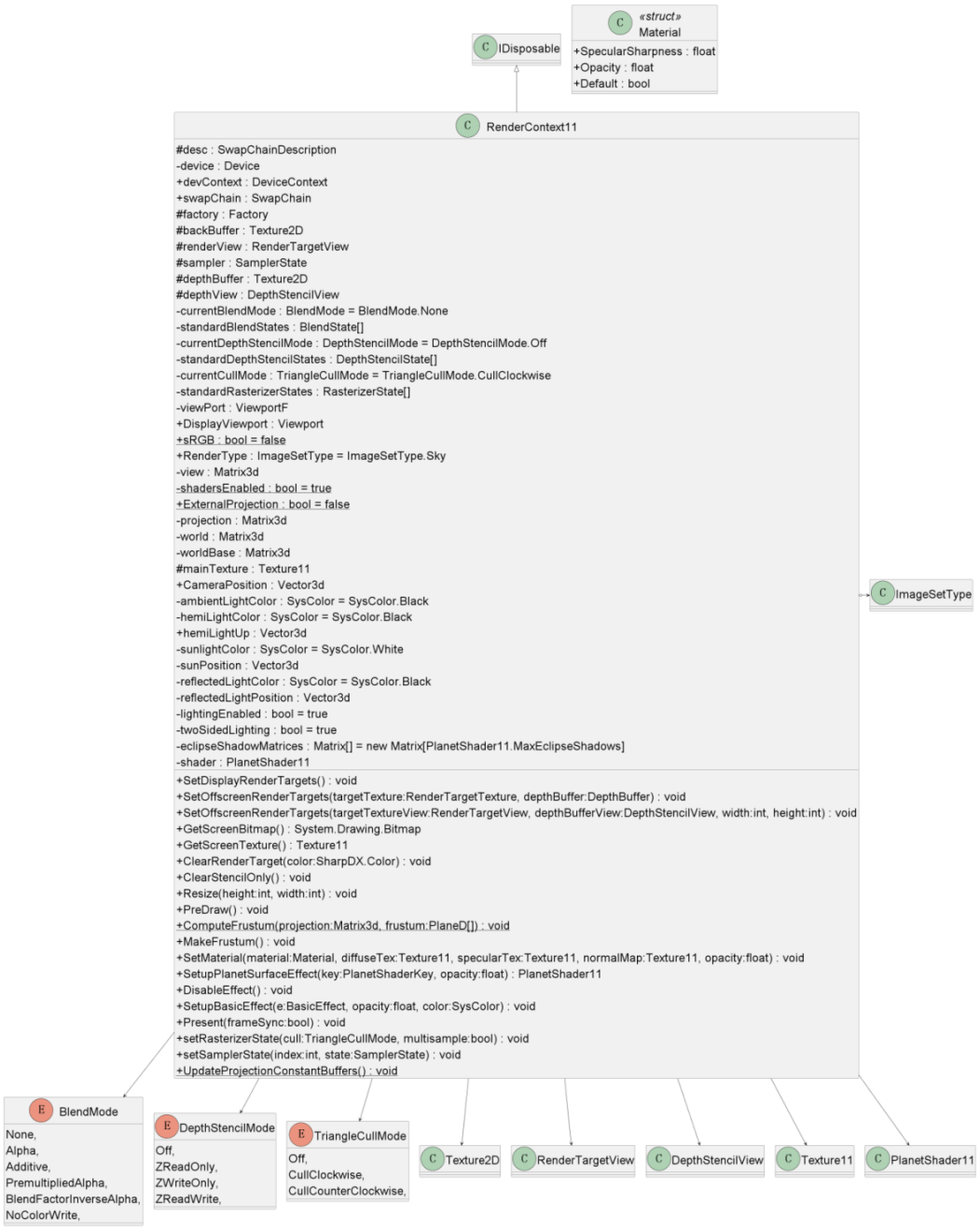

图 4-3 渲染上下文类图

通过对渲染上下文的静态分析，可以了解渲染上下文的结构和方法，从而更好地理解其工作流程。基于此，绘制出渲染上下文的流程图，以更加清晰地展示渲染上下文在运行时的工作流程。如图 4-4 为渲染上下文的流程图。

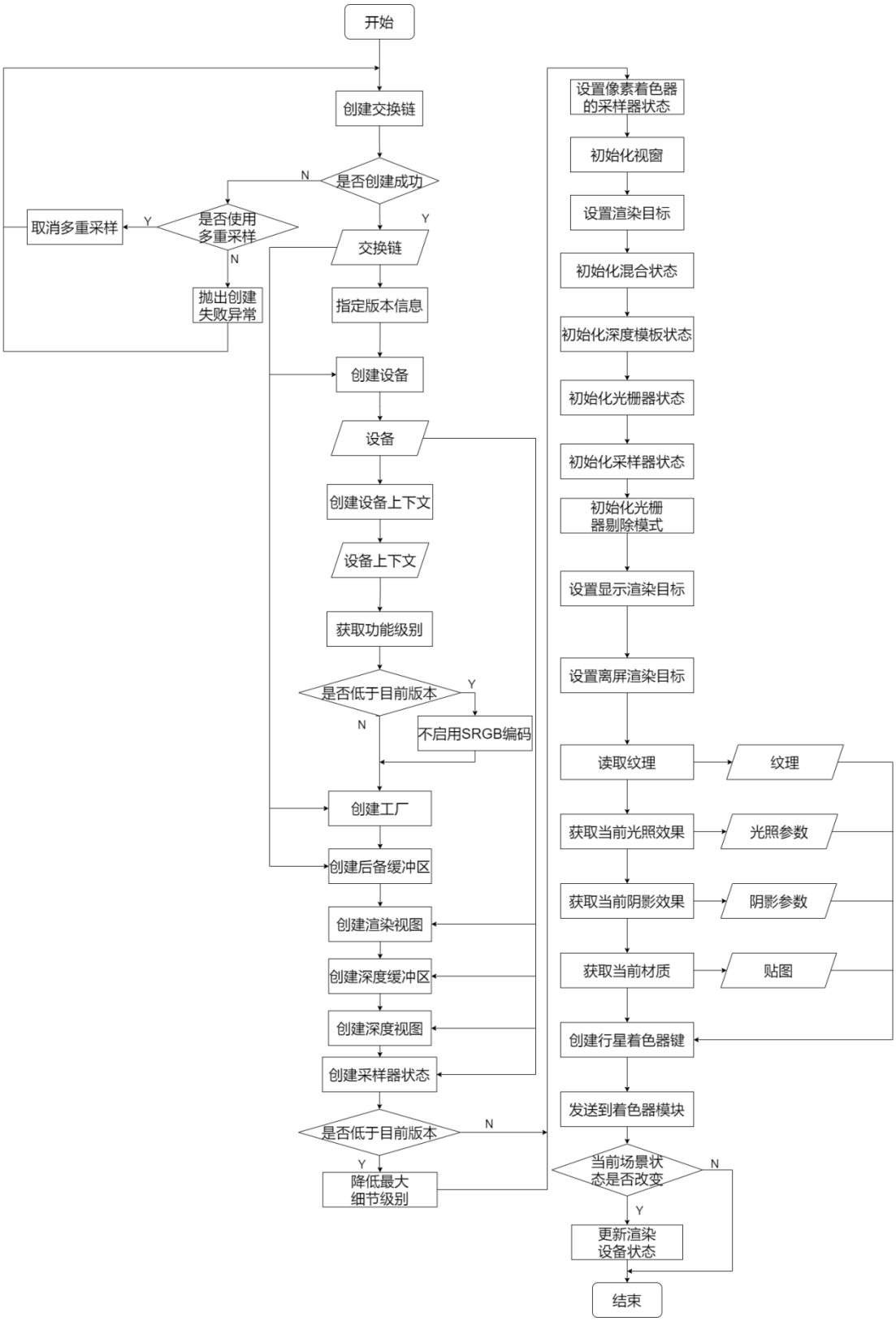

图 4-4 渲染上下文流程图

4.3 纹理

纹理在渲染中具有重要作用，它的主要功能是通过读取本地文件的位图，将其规范化为 Direct3D 所使用的位图类型。接着将规范化后的位图转化为 2D 纹理，并将 2D 纹理提供给渲染上下文使用。

渲染上下文是由 Texture11 类来实现的，如图 4-5 是该类的类图，类图展示了类中的主要方法和属性。下面将介绍该类的主要方法以及其实现原理。

（1）纹理构造函数

纹理构造函数包含两个重载。两个重载的功能都是实例化纹理对象。以下将分别分析两个重载函数。

在第一个构造函数中传入了一个 2D 纹理对象，将其保存到类成员变量 2D 纹理中，并创建一个着色器资源视图，用于在着色器中读取纹理资源。

在第二个构造函数的作用与第一个相同，但多了一个参数不存在资源视图标志，资源视图标志表示构造函数是否创建着色器资源视图。如果资源视图标志为 true，则不创建着色器资源视图对象，此时该构造函数的作用是创建一个不包含资源纹理的纹理对象。

（2）读取本地图像

读取本地文件方法是一个静态方法，用于从文件中加载图像。函数的第一个参数是设备，表示本地设备信息。第二个参数是要加载的文件名。第三个参数是一个枚举值，表示加载选项，默认为纹理使用 SRGB 格式。以下将介绍该函数的工作流程。

首先，根据参数信息初始化纹理加载配置信息，其中包括绑定管线阶段、CPU 访问缓冲区权限、纹理深度、过滤器类型、多纹理采样类型、纹理格式、纹理高度和宽度。检查文件名后缀是否是支持的图像格式，如果是，则设置加载配置中的过滤器、格式和 SRGB 标志。如果需要将格式升级为 SRGB 格式，则将加载配置信息中的格式设置为 SRGB 格式。

如果文件名的后缀不是支持的图像格式，则使用 SharpDX 的图像信息类实例化文件信息，并检查是否需要升级为 SRGB 格式。如果需要，则将加载配置信息中的格式设置为 SRGB 格式。

调用 SharpDX 中 2D 纹理的读取文件方法，从文件中读取纹理数据，并将纹理数据实例化为 2D 纹理对象。将 2D 纹理对象传递给纹理构造函数，实例化一个新的纹理对象，最终传递到渲染上下文中。

如果在上述步骤发生异常，则将处理失败的纹理文件作为参数再次调用 SharpDX 中

2D 纹理的读取文件方法，使用默认纹理格式创建新的纹理对象。如果仍然失败，则返回 null。

（3）读取位图

读取位图方法是一个静态方法，用于加载位图并转化为纹理对象。函数的第一个参数是设备，表示本地设备信息。第二个参数是要加载的位图。以下将介绍该函数的工作流程。

首先，根据传入的位图对象，创建一个内存流。然后将位图保存为 PNG 格式并存入内存流中，并把内存流的查找位置调整为起始索引。接着，创建一个图像加载信息对象，并设置其属性。然后判断位图的宽和高是否为 2 的整数次幂，如果宽高均为 2 的整数次幂，将 MipLevel 设置为 0，表示使用原始的纹理图像。如果不是 2 的整数次幂，将 MipLevel 设置为 1，表示生成一张缩小一半的位图，这样在渲染时可以根据需要使用更小的图像来提高性能和质量。

接下来，将图像加载信息对象的纹理格式设置为渲染上下文提供的默认纹理格式。当默认纹理格式是 SRGB 属性时，将过滤器设置为 SRGB。

最后，根据内存流的图像数据和图像加载信息实例化一个 2D 纹理对象。将 2D 纹理对象传递给纹理构造函数，实例化一个新的纹理对象，最终传递到渲染上下文中。


IDisposable
Texture11
-nextID : int = 0
-texture : Texture2D
-resourceView : ShaderResourceView
+Width : int
+Height : int
#savedStrem : byte[] = null
+SaveStream : bool = false
+SaveStream : bool = false
+Texture11(t:Texture2D)
+Texture11(t:Texture2D, noResourceView:bool)
+Dispose() : void
+FromFile(fileName:string) : Texture11
#promoteFormatToSRGB(format : Format) : Format
#isSRGBFormat(format : Format) : bool
+FromFileImediate(device:Device, filename:string, options:LoadOptions) : Texture11
+FromFile(device:Device, fileName:string, options:LoadOptions) : Texture11
+FromBitmap(bmp:object) : Texture11
+FromBitmap(bmp:object, transparentColor:uint) : Texture11
+FromBitmap(device:Device, bitmap:object) : Texture11
#IsPowerOf2(x : uint) : bool
+FromStream(stream:Stream) : Texture11
+SaveToFile(filename:string) : void
+FromStream(device:Device, stream:Stream) : Texture11
LoadOptions
None= 0x0,
AssumeSRgb= 0x1,
Texture2D
ShaderResourceView

图 4-5 纹理类图

通过对纹理模块的静态分析，可以了解纹理模块的结构和方法，从而更好地理解其工作流程。基于此，绘制出纹理的流程图，以更加清晰地展示渲染上下文在运行时的工作流程。如图 4-6 为纹理模块的流程图。

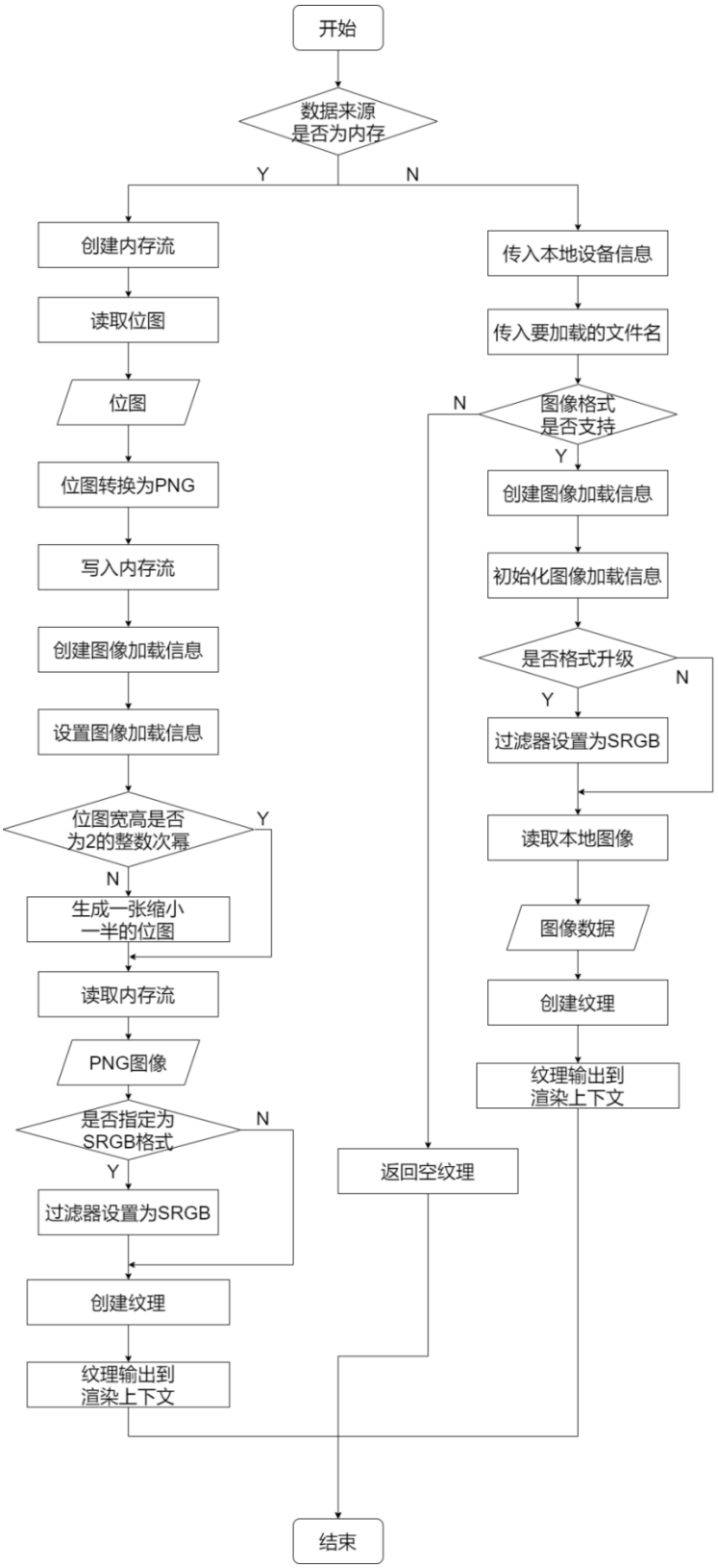

图 4-6 纹理模块流程图

4.4 着色器

着色器的主要功能是在渲染管线中对几何图形进行处理和渲染，以产生符合用户需求的图像。

分析渲染引擎的着色器模块，需要首先要了解渲染管线，才能对着色器代码进行可靠的分析。由于着色器系统庞大，对整个系统直接使用静态分析和动态分析可能会比较繁琐。因此，需要将着色器系统分成行星着色器、简单着色器和着色器库三个子模块进行分析。通过介绍每个子模块的功能和实现原理，可以更好地理解着色器系统的工作流程。

4.4.1 渲染管线

在渲染引擎中，渲染管线是指：在给定的一个 3D 场景的几何描述及一架已确定位置和方向的虚拟摄像机时，根据虚拟摄像机的视角生成 2D 图像的一系列步骤（渲染管线由许多步骤组成，每个步骤称为一个阶段）[4]。如图 4-7 所示，为渲染引擎渲染管线的各个阶段，以及与各个阶段相关的内存资源[3]。从内存指向阶段的箭头表示该阶段可以从内存读取数据，其中圆角矩形表示渲染引擎提供给开发人员可编程的阶段。下面将介绍渲染管线的重要阶段。

（1）输入装配阶段（Input Assembler Stage）

输入装配阶段从内存中读取顶点和索引数据，并将这些数据组合为几何图元。通常用到的图元拓扑为点、直线和三角形。

（2）顶点着色器阶段（Vertex Shader Stage）

完成图元装配后，顶点将被送往顶点着色器阶段。它的主要作用是对输入的几何形状进行变换和处理，并输出变换后的结果给下一个阶段。在顶点着色器阶段中，每个输入的顶点都会被处理一次，该阶段的输出通常是每个顶点的位置和其他属性（如颜色、纹理坐标等）。

（3）曲面细分阶段（Tessellation Stage）

曲面细分是指通过添加三角形的方式对一个网格的三角形进行细分，这些新添加的三角形可以偏移到一个新的位置，让网格的细节更加丰富。

（4）几何着色器阶段（Geometry Shader Stage）

几何着色器是渲染管线中对完整的图元进行计算和处理的可编程阶段。在渲染引擎中，几何着色器由开发人员指定，并且其使用是可选的。渲染引擎根据是否需要外部视图来决定是否需要添加几何着色器。

在图 4-5 中可以看到“流输出阶段（Stream Output Stage）”箭头，几何着色器可以将顶点数据流输出到内存中的一个顶点缓冲区内，顶点可以在渲染管线的随后阶段中渲染出来。

（5）光栅化阶段（Rasterizer Stage）

光栅化阶段的主要任务是将图元转化为像素，并对每个像素进行处理，例如深度模板测试和多重采样。

（6）像素着色器阶段（Pixel-Shader Stage）

像素着色器是由开发人员编写的在 GPU 上执行的程序。像素着色器会计算和处理每个光栅化阶段输入的像素片段，由此计算出一个颜色。在渲染引擎中，像素着色器是由开发人员指定的。

（7）输出合并阶段（Output-Merger Stage）

当像素片段由像素着色器生成之后，它们会被传送到渲染管线的输出合并阶段。在该阶段中，某些像素片段会被丢弃。未丢弃的像素片段会被写入后台缓冲区。

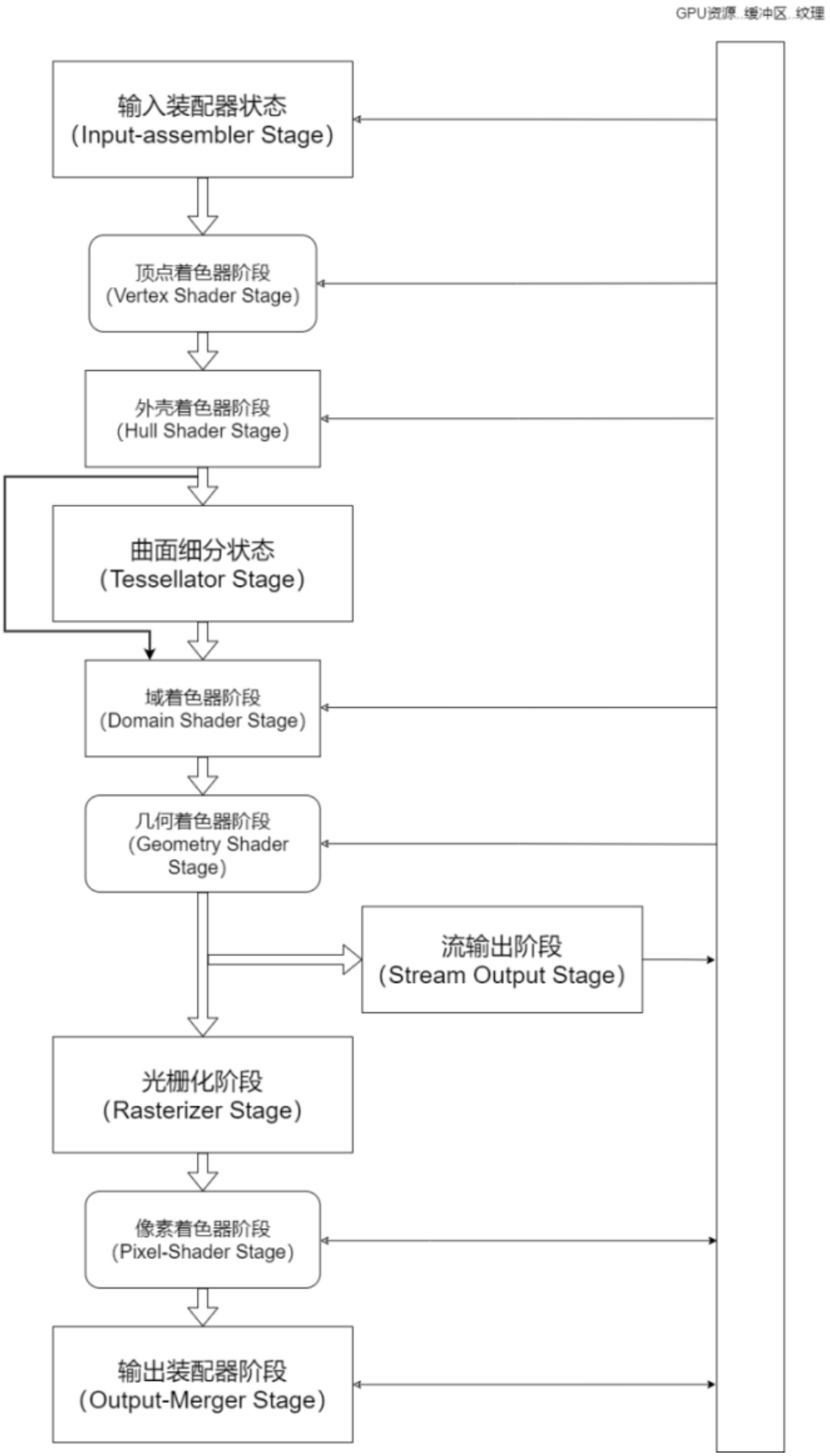
GPU资源.缓冲区.纹理
输入装配器状态
(Input-assembler Stage)
顶点着色器阶段
(Vertex Shader Stage)
外壳着色器阶段
(Hull Shader Stage)
曲面细分状态
(Tessellator Stage)
域着色器阶段
(Domain Shader Stage)
几何着色器阶段
(Geometry Shader Stage)
流输出阶段
(Stream Output Stage)
光栅化阶段
(Rasterizer Stage)
像素着色器阶段
(Pixel-Shader Stage)
输出装配器阶段
(Output-Merger Stage)

图 4-7 渲染管线

4.4.2 行星着色器

行星着色器的主要功能是根据渲染上下文传入的着色器键，创建相应功能的顶点着色器、几何着色器和像素着色器。

行星着色器是由 Shader 和 PlantShader11 类来实现的，如图 4-8 是该类的类图，类图展示了类中的主要方法和属性。下面将介绍一下行星着色器的主要方法。

（1）输入着色器键

该方法的功能是从渲染上下文中获取着色器键，并将着色器键传入到创建着色器代码的方法中。着色器键主要包含以下参数：纹理参数、太阳方向向量和颜色、光源个数、阴影个数和索引、半球光方向向量和颜色、摄像机方向向量、漫反射颜色向量、高光强度和颜色以及行星大气参数。

（2）创建着色器代码

该方法的功能是根据传入的着色器键参数创建相应的着色器代码。其中包括创建顶点着色器代码、像素着色器代码。如果使用外部视图，还需要创建几何着色器。

（3）保存并编译着色器代码

该方法的功能是将创建后的着色器代码编译并保存到着色器库中。在该方法中，会向 GPU 发出编译着色器的指令，将着色器代码编译为着色器二进制代码，并将着色器键序列化。接着，将序列化后的着色器键和着色器二进制代码保存到着色器库中，并将预编译的着色器代码保留在本地。

（4）向渲染引擎输出着色器

该方法的功能是将编译好的着色器代码转换为适用于 DirectX 的着色器代码，并将着色器代码输出到渲染引擎中，以供渲染使用。

为了深入理解行星着色器的实现原理，需要分别对其打包常量寄存器、纹理常量缓冲区、顶点着色器、几何着色器、采样器和像素着色器进行功能分析。

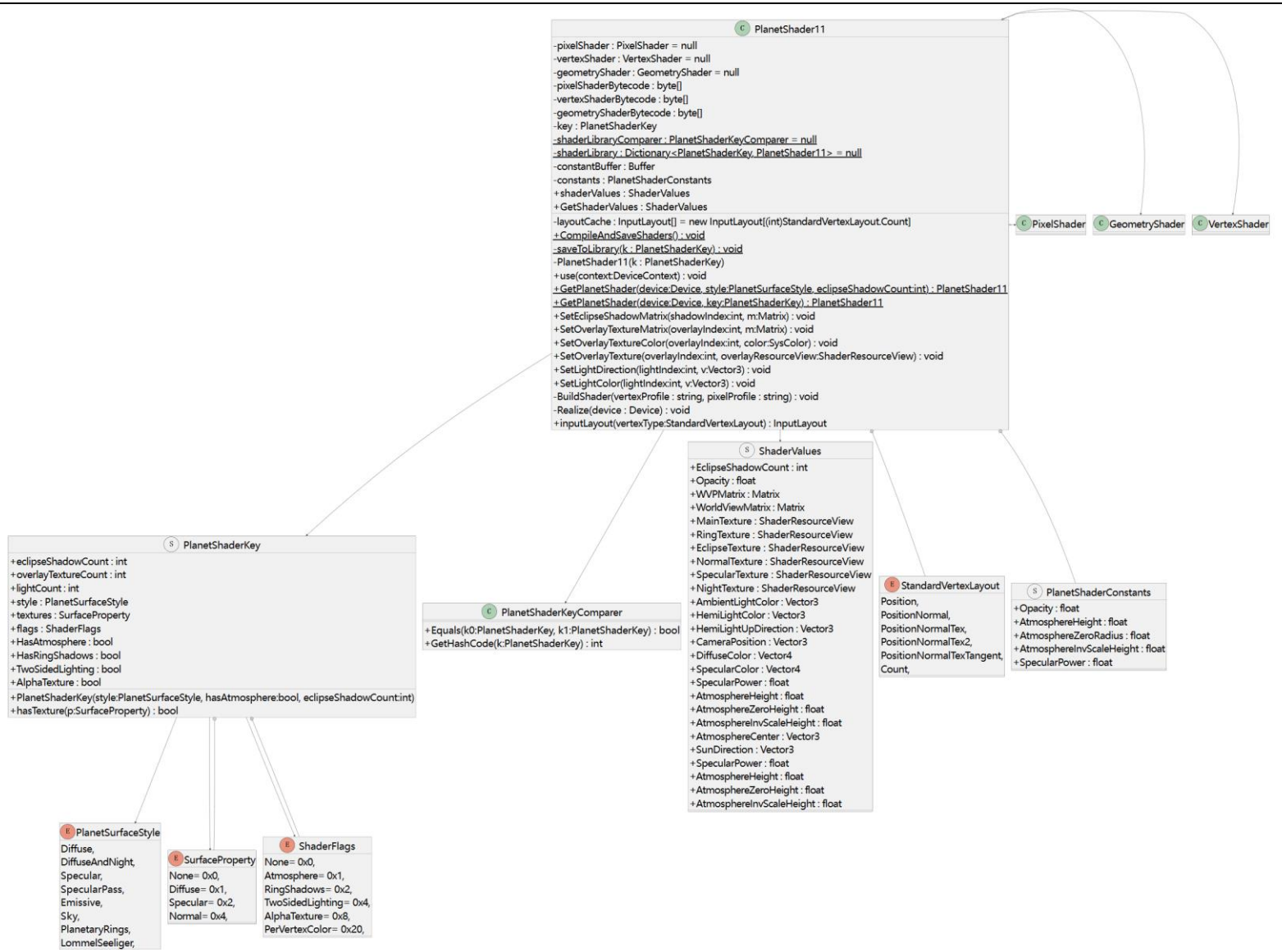

图 4-8 行星着色器的类图

（5）打包常量寄存器

打包常量寄存器是包含两层含义，第一是常量寄存器，另一个是常量寄存器的打包规则。首先常量寄存器是将多个常量值存储在寄存器中，以便在图形渲染中提高性能。常量寄存器的主要功能是为顶点着色器、几何着色器和像素着色器提供可编程的常量数据。然后就是常量寄存器的打包规则，其为 HLSL 中常量变量遵循的打包规则，它规定了数据存储时排列的紧密程度。在行星着色器中通过 packoffset 关键字进行手动打包。

常量寄存器中的常量分成三部分，一部分是顶点着色器使用的常量，另一部分是像素着色器使用常量，还有一个外部视图定义的一组视图和投影矩阵。如图 4-9 所示为常量寄存器分布图。

顶点着色器常量	float4x4 matWVP	C0
		C1
		C2
		C3
	float4x4 matWV	C4
		C5
		C6
		C7
	float opacity	C8
	float sunDirection	C9
	float cameraPosition	C10
	float3 rayleighScatterCoeff	C11
	float atmosphereHeight	C12
	float atmosphereZeroRadius	C13
	float atmosphereInvScaleHeight	C14
	float3 atmosphereCenter	C15
	float4x4 matEclipseShadow0	C16
		C17
		C18
		C19
	float4x4 matEclipseShadow1	C20
		C21
		C22
		C23
	float4x4 matEclipseShadow2	C24
		C25
		C26
		C27
	float4x4 matEclipseShadow3	C28
		C29
		C30
		C31
	float4x4 matOverlayTexture0	C32
		C33
		C34
		C35
	float4x4 matOverlayTexture1	C36
		C37
		C38
		C39
	float4x4 matOverlayTexture2	C40
		C41
		C42
		C43
	float4x4 matOverlayTexture3	C44
		C45
		C46
		C47
像素着色器常量	float3 lightDirection0	C48
	float3 lightColor0	C49
	float3 lightDirection1	C50
	float3 lightColor1	C51
	float3 lightDirection2	C52
	float3 lightColor2	C53
	float3 lightDirection3	C54
	float3 lightColor3	C55
	float3 ambientLightColor	C56
	float4 diffuseColor	C57
	float3 hemiLightColor	C58
	float3 hemiLightUp	C59

图 4-9 常量寄存器分布图

（6）顶点着色器

顶点着色器的书写分为顶点着色器输入缓冲区，顶点着色器缓冲区输出和主函数。

顶点着色器输入缓冲区的功能就是存储传入的顶点数据。顶点着色器输入缓冲区结构体的定义如下。如果采用自发光材质，则输入缓冲区包含对象空间位置、纹理坐标、经纬度坐标系下的坐标、顶点颜色和实例 ID 等信息；如果不采用自发光材质，则输入缓冲区包含对象空间位置、纹理坐标、对象空间法向量、经纬度坐标系下的坐标、对象空间切线向量、顶点颜色和实例 ID。

顶点着色器输出缓冲区的功能就是保存处理和变换后的顶点数据。顶点着色器输出缓冲区结构体的定义如下。输出缓冲区包含投影空间位置、纹理坐标、表面法线、表面切线、对象重心坐标。根据渲染上下文传入的着色器键，如果需要逐点着色器，则需要加上顶点颜色；如果行星存在大气，则需要设置颜色插值器；如果存在行星间相互遮蔽，则需要设置阴影纹理坐标；如果设置环阴影或者存在星环，则需要设置环阴影纹理坐标；如果需要设置叠加纹理，则需要设置叠加纹理坐标。

根据顶点着色器输入缓冲区和输出缓冲区的定义，介绍主函数的功能和工作流程。顶点着色器的主要功能是根据传入的着色器参数实现对输入顶点数据的特定处理和变换。具体来说，主函数使用瑞利散射模型处理顶点数据，达到模拟行星大气效果的目的。如果设置阴影效果，则通过阴影纹理坐标和叠加纹理坐标模拟行星之间和行星环的遮挡效果。如图 4-10 为顶点着色器主函数流程图。

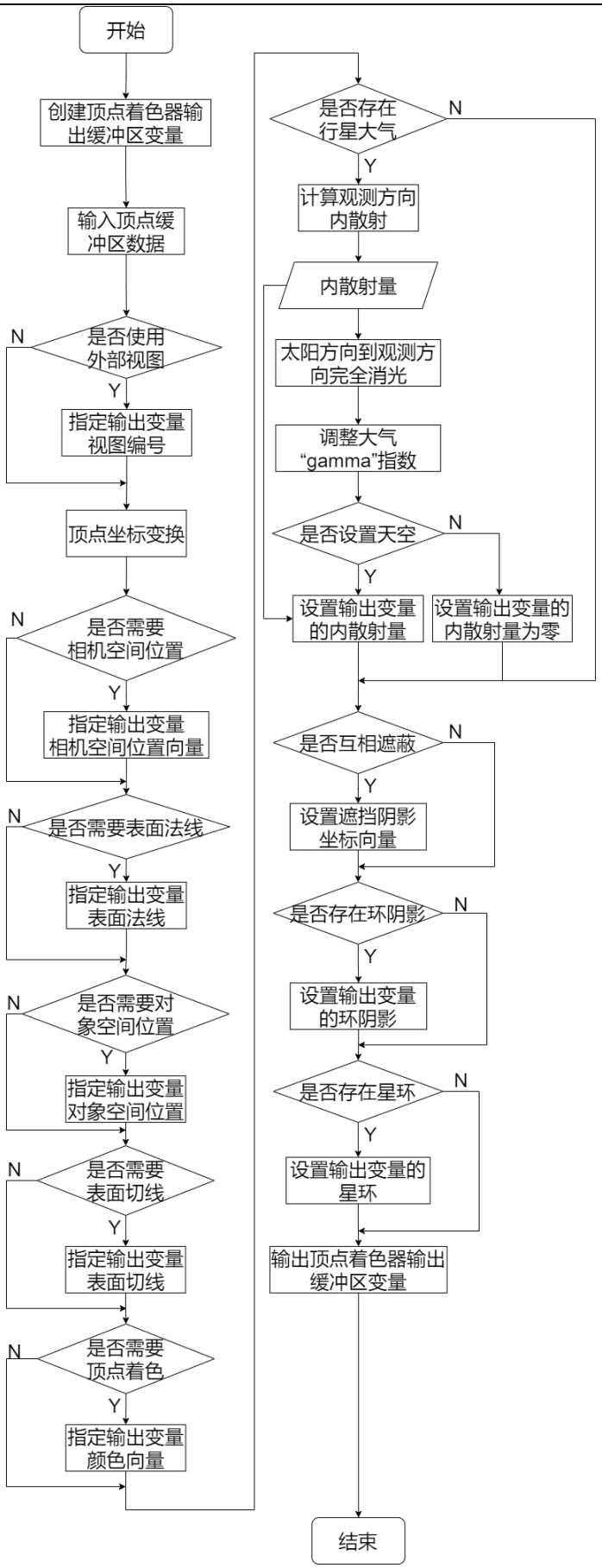

图 4-10 顶点着色器主函数流程图

（7）几何着色器

在渲染引擎中，当使用外部视图实现混合现实的立体渲染时，才使用几何着色器。几何着色器的书写分为几何着色器输入缓冲区，几何着色器缓冲区输出和主函数。

几何着色器输入缓冲区的功能是存储顶点着色器传入的顶点数据。几何着色器输入缓冲区结构体的定义与顶点缓冲区输出缓冲区结构体的定义相同。

几何着色器输出缓冲区的功能是存储几何着色器处理后的几何数据的，包括图元的顶点数据以及可能的附加信息。几何着色器输出缓冲区结构体的定义与顶点缓冲区输出缓冲区结构体的定义相同。

根据几何着色器输入缓冲区和输出缓冲区的定义，介绍主函数的功能和工作流程。几何着色器的主要功能是通过遍历三个输入顶点，将每个顶点的属性复制到输出顶点中。输出的图元为三角形，三角形流传递到光栅化阶段。如图 4-11 为几何着色器主函数流程图。

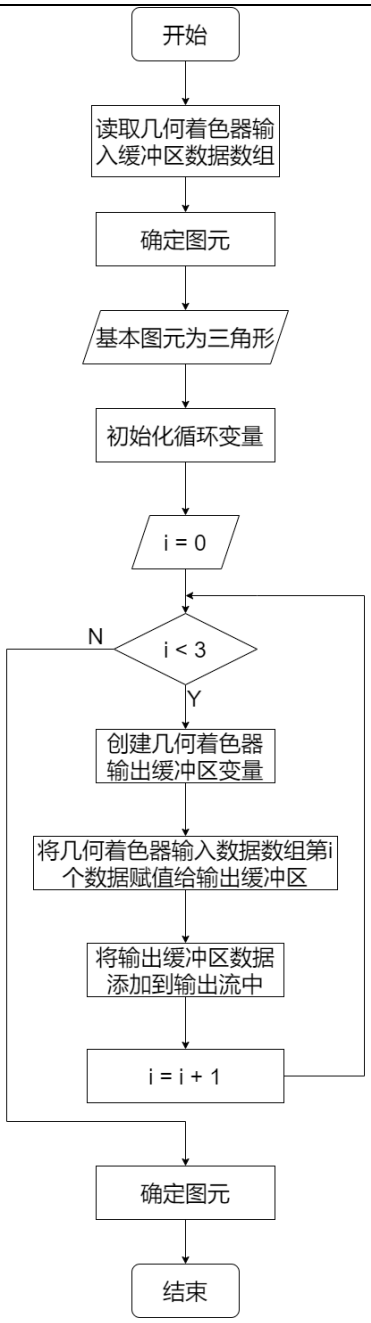

图 4-11 几何着色器主函数流程图

（8）纹理常量缓冲区

纹理常量缓冲区是用于存储渲染上下文传入的纹理数据的缓冲区。其主要作用是在像素着色器中进行纹理采样时，将需要采样的纹理数据传入像素着色器中，以便像素着色器对像素进行着色处理。

纹理常量缓冲区是必须包含一个主纹理，而其他纹理则根据纹理映射方式的要求进行设定。如果使用夜间贴图，则需要设定夜间纹理；如果使用高光贴图，则需要设定高光纹理；如果需要法线贴图，则需要设定法线纹理；如果阴影参数不为零，则需要设定遮挡阴影纹理；如果由行星有环，则需要设定环阴影纹理；如果设定叠加纹理，则需要

设定所有传入的叠加纹理。

（9）采样器

采样器是用于在 GPU 上进行纹理采样的对象。其主要作用是将纹理常量缓冲区传入的纹理进行纹理采样，并将采样值传入像素着色器中。

行星着色器主要定义了主要纹理采样器、夹具纹理采样器、环阴影纹理采样器和透明边界纹理采样器。

主要纹理采样器的过滤器为使用线性内插进行缩小、方法和 mipmap 采样。UV 方向的寻址方式均采用“Clamp”方式。即范围[0.0， 1.0]之外的纹理坐标分别设置为 0.0 或 1.0 的纹理颜色。主要纹理采样器也是默认的纹理采样方式。

夹具纹理采样器与主要纹理采样器的过滤器和寻址方式均相同。

环阴影纹理采样器的过滤器为使用线性内插进行缩小、方法和 mipmap 采样。UV 方向的寻址方式均采用“Border”方式。即范围[0.0， 1.0]之外的纹理坐标颜色设置为渲染上下文中指定的边框颜色，根据渲染上下文模块的描述边框颜色设定为黑色。

透明边界纹理采样器的过滤器为使用线性内插进行缩小、方法和 mipmap 采样。UV 方向的寻址方式均采用“Border”方式。即范围[0.0， 1.0]之外的纹理坐标颜色设置为 HLSL 中指定的边框颜色，这里是（0，0，0，0）表示完全透明。

（10）像素着色器

像素着色器的书写分为像素着色器输入缓冲区和主函数。

像素着色器输入缓冲区的功能是接收从光栅化器传入的像素数据。像素着色器输入缓冲区结构体的定义与顶点缓冲区输出缓冲区结构体的定义相同。

根据像素着色器输入缓冲区的定义，介绍主函数的功能和工作流程。像素着色器的主要功能是根据光栅化后传入的像素值，通过特定方式计算出像素的颜色值。具体来说，就是根据纹理、贴图、阴影和光照效果，计算出行星表面顶点的像素。如图 4-12 为像素着色器主函数流程图。

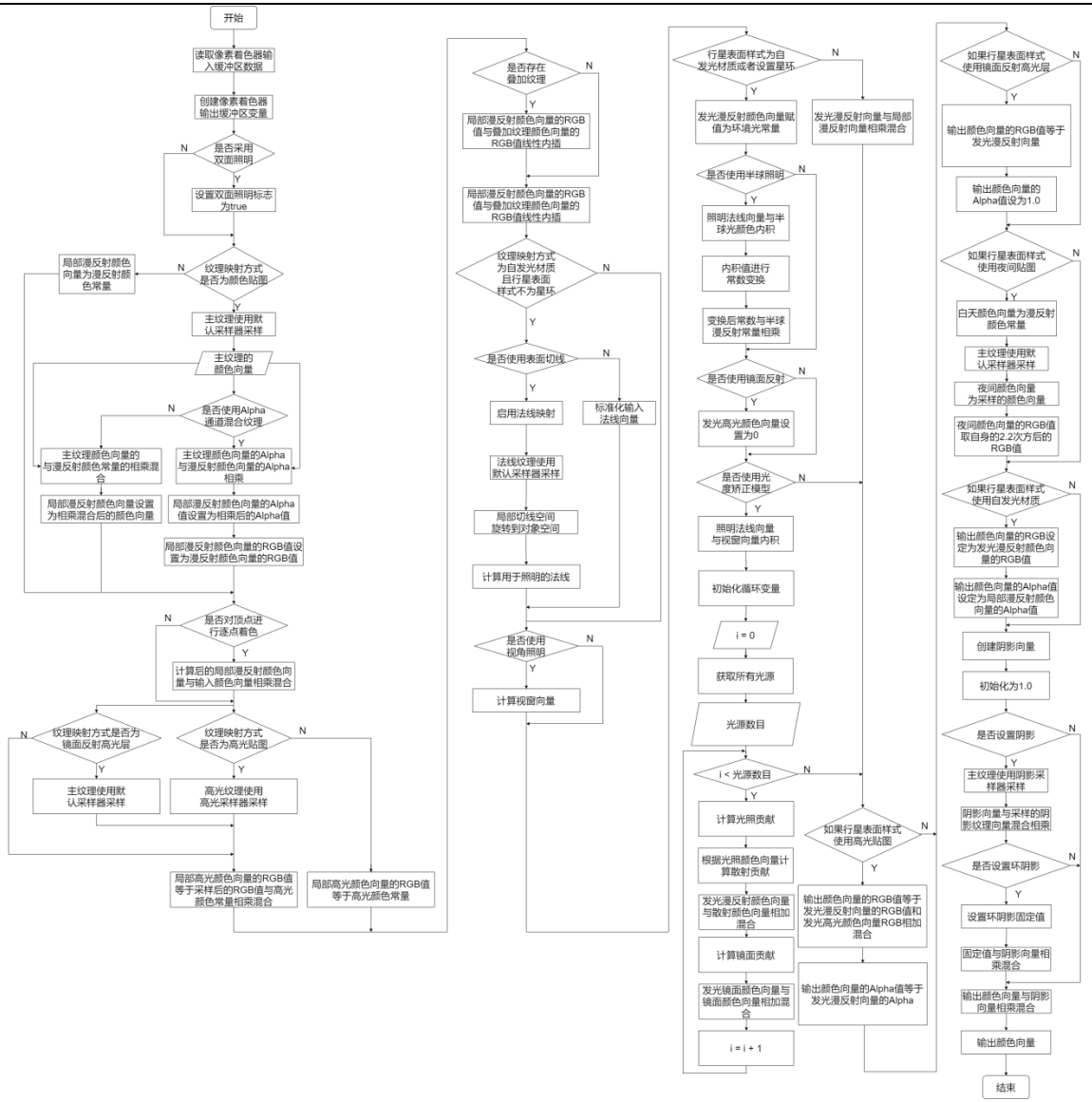

图 4-12 像素着色器主函数流程图

通过对行星着色器的静态分析和可编程着色器的动态分析，可以了解行星着色器的结构和方法，从而更好地理解其工作流程。基于此，绘制出行星着色器的流程图，以更加清晰地展示行星着色器在运行时的工作流程。如图 4-13 为行星着色器的流程图。

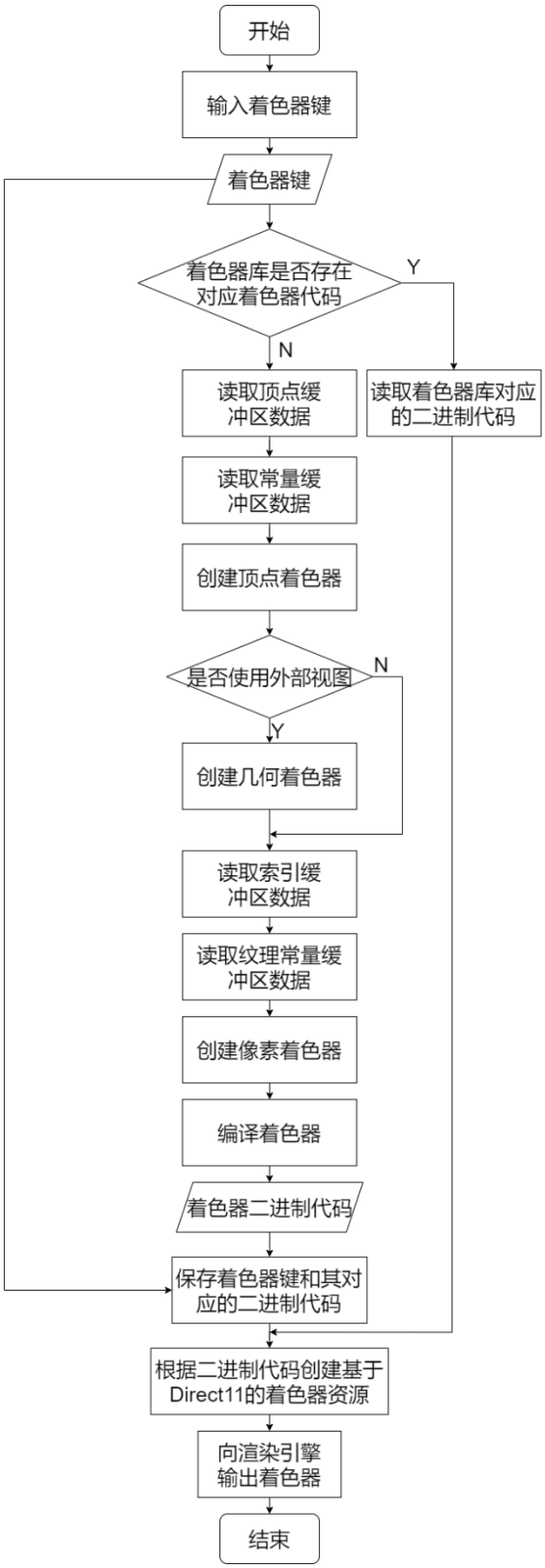

图 4-13 行星着色器流程图

4.4.3 简单着色器

简单着色器的主要功能是根据渲染引擎的功能指令创建相应功能的顶点着色器、几何着色器和像素着色器。与行星着色器不同，简单着色器的顶点着色器、几何着色器和像素着色器的 HLSL 代码是直接编辑好的，只要渲染引擎给出对应需求的功能指令，它就会创建对应的着色器代码。渲染上下文只负责指定常量缓冲区参数和接收渲染消息。

简单着色器是由 ShaderBundle 抽象类实现的，由子类实现方法，如图 4-14 是以线条着色器类为例，绘制简单着色器的类图，类图展示了类中的主要方法和属性。

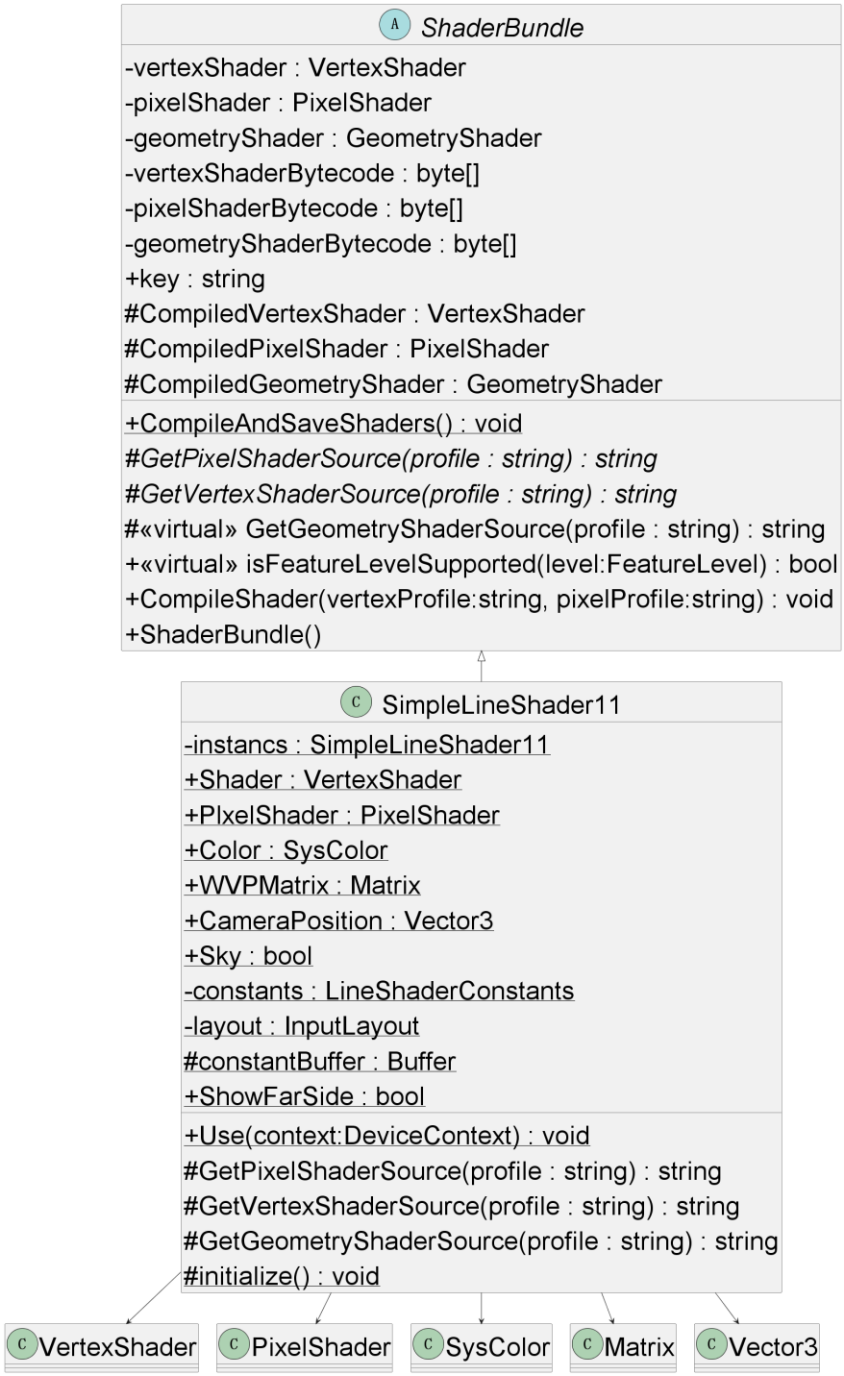

图 4-14 简单着色器类图

虽然简单着色器框架下创建的各个子着色器的功能和类定义各不相同，但其工作方式基本相同。如图 4-15 为简单着色器的流程图。

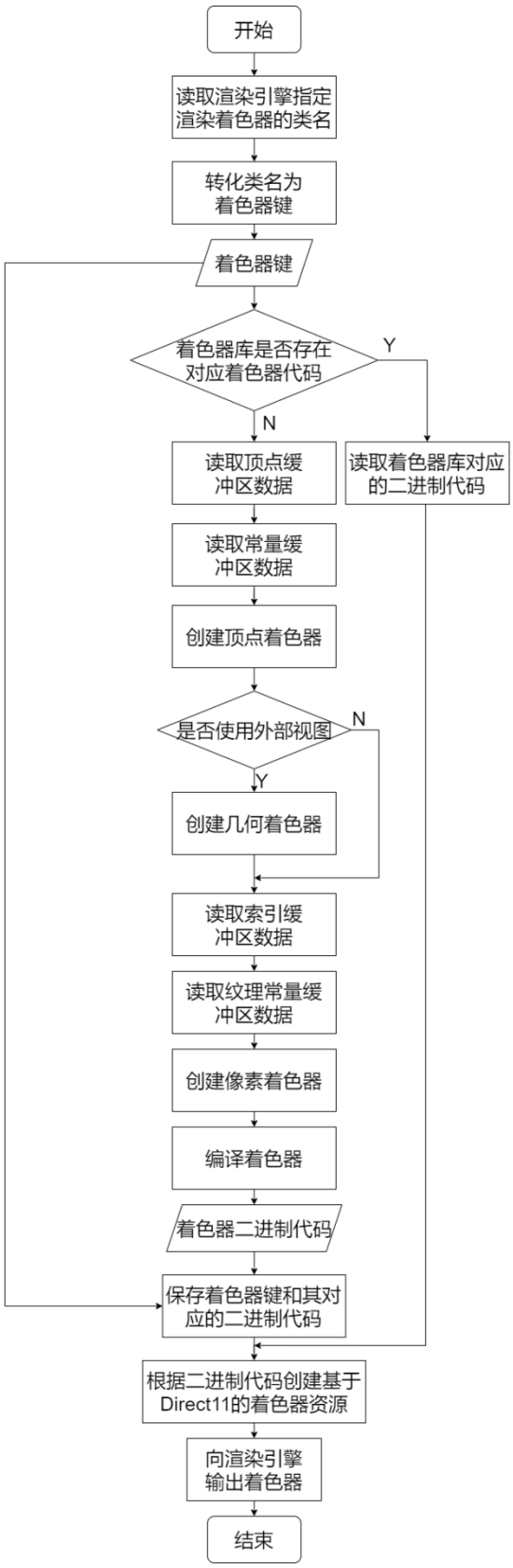

图 4-15 简单着色器流程图

4.4.4 着色器库

着色器库的主要功能是根据着色器键，保存编译好的着色器二进制代码。从而在下一次使用中直接运行着色器二进制代码，提高渲染引擎的渲染效率。

着色器库是由 ShaderLibrary 类实现的，如图 4-16 为着色器库类图，类图展示了类中的主要方法和属性。

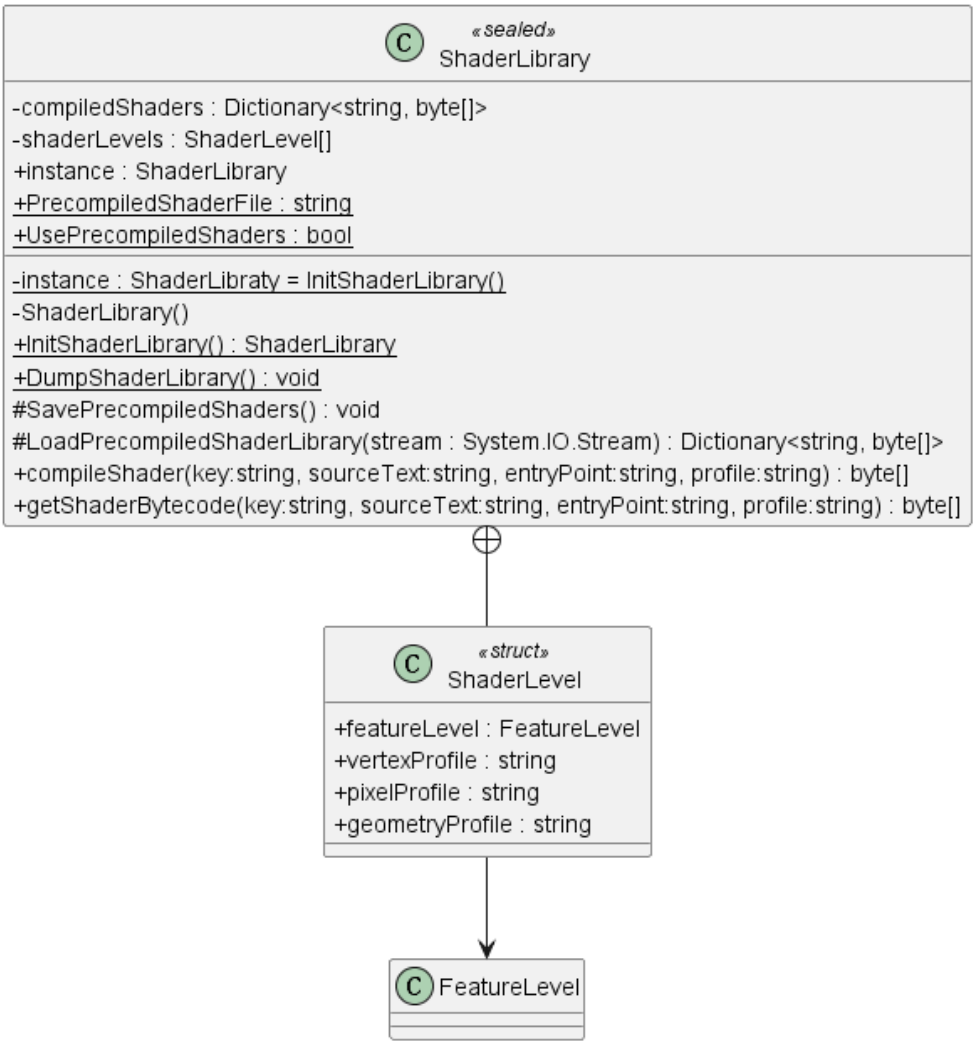

图 4-16 着色器库类图

通过对着色器库的静态分析，可以了解着色器库的结构和方法，从而更好地理解其工作流程。基于此，绘制出着色器库的流程图，以更加清晰地展示着色器库在运行时的工作流程。如图 4-17 为着色器库的流程图。

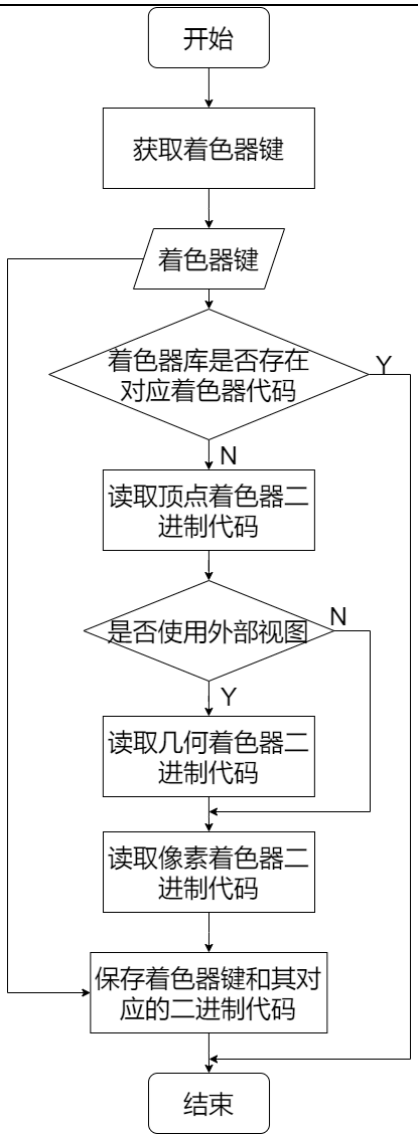

图 4-17 着色器库流程图

第 5 章 测试

5.1 渲染效果测试

本章主要是对着色器的效果进行功能测试，来证实前三章论述的可靠性。主要对行星着色器进行黑盒测试。

5.1.1 测试目的和测试条件

（1）测试目的

测试行星着色器在不同参数下是否产生正确的输出。

（2）测试条件

在编译成功的个人计算机上进行万维望远镜项目的测试。

5.1.2 测试方法和测试步骤

（1）测试方法

首先，需要预先选择需要输入的天体类型，并确保这些天体类型能够覆盖渲染效果的所有情况。接着，向渲染引擎输入不同的天体类型，并在万维望远镜界面观察各个天体的渲染情况，并记录下天体的渲染情况的截图。最后，通过查看本地存储的预编译的 HLSL 代码，来确认天体类型是否正确。

（2）测试步骤

首先设置渲染引擎的渲染目标。然后，根据每个测试行星输出行星着色器的顶点着色器、几何着色器和像素着色器。最后根据本地输出的着色器代码判断行星着色器的设置是否符合万维望远镜渲染出的行星样式。

5.1.3 测试用例设计

（1）太阳

设置太阳用例的目的是为了测试行星着色器在渲染自发光材质时是否能正常工作，并通过输出的 HLSL 代码验证代码功能为设置自发光效果。

如表 5-1 所示，为太阳渲染效果测试用例表。

表 5-1 太阳渲染效果测试用例表

测试用例信息	描述
测试用例编号	Worldwide_Telescope_Rendering_Engine_PlantShader_Sun_001
测试项目	太阳渲染效果
测试标题	检测太阳渲染效果和输出的 HLSL 代码
重要级别	高
预置条件	万维望远镜已将太阳渲染到界面中
输入	太阳位置坐标
操作步骤	1 根据太阳位置设置摄像机朝向 2 观察太阳渲染效果 3 查看本地预编译 HLSL 代码
预期输出	太阳呈现自发光的渲染效果，HLSL 代码功能为设置自发光效果

（2）水星

设置水星用例的目的是为了测试行星着色器在不加任何渲染效果时是否能正常工作，并通过输出的 HLSL 代码验证代码功能为设置颜色贴图。

如表 5-2 所示，为水星渲染效果测试用例表。

表 5-2 水星渲染效果测试用例表

测试用例信息	描述
测试用例编号	Worldwide_Telescope_Rendering_Engine_PlantShader_Mercury_001
测试项目	水星渲染效果
测试标题	检测水星渲染效果和输出的 HLSL 代码
重要级别	高
预置条件	万维望远镜已将水星渲染到界面中
输入	水星位置坐标
操作步骤	1 根据水星位置设置摄像机朝向 2 观察水星渲染效果 3 查看本地预编译 HLSL 代码
预期输出	水星仅呈现自身颜色贴图的渲染效果，HLSL 代码功能为设置颜色贴图

（3）地球

首先设置地球无云层测试用例，设置地球无云层测试用例是为了验证行星着色器是否正常设置大气效果，并通过输出的 HLSL 代码验证代码功能为设置大气。

如表 5-3 所示，为地球无云层测试用例表。

表 5-3 地球无云层渲染效果测试用例表

测试用例信息	描述
测试用例编号	Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_001
测试项目	地球无云层渲染效果
测试标题	检测地球渲染效果和输出的 HLSL 代码
重要级别	高
预置条件	万维望远镜已将地球渲染到界面中
输入	地球位置坐标，取消地球云层
操作步骤	1 根据地球位置设置摄像机朝向 2 观察地球渲染效果 3 查看本地预编译 HLSL 代码
预期输出	地球呈现大气的渲染效果，HLSL 代码功能为设置大气

然后设置地球有云层测试用例，设置地球有云层测试用例是为了验证行星着色器是否正常设置云层遮蔽效果，并通过输出的 HLSL 代码验证代码功能为设置云层遮蔽。

如表 5-4 所示，为地球有云层测试用例表。

表 5-4 地球有云层渲染效果测试用例表

测试用例信息	描述
测试用例编号	Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_002
测试项目	地球有云层渲染效果
测试标题	检测地球渲染效果和输出的 HLSL 代码
重要级别	高
预置条件	万维望远镜已将地球渲染到界面中
输入	地球位置坐标，启动地球云层
操作步骤	1 根据地球位置设置摄像机朝向 2 观察地球渲染效果 3 查看本地预编译 HLSL 代码
预期输出	地球呈现云层遮蔽效果，HLSL 代码功能为设置云层遮蔽

（4）月球

设置月球测试用例的目的是为了验证行星着色器是否正常设置天体间阴影遮蔽效果，并通过输出的 HLSL 代码验证代码功能为设置天体间阴影遮蔽。

表 5-5 月球渲染效果测试用例表

测试用例信息	描述
测试用例编号	Worldwide_Telescope_Rendering_Engine_PlantShader_Moon_001
测试项目	月球渲染效果
测试标题	检测月球渲染效果和输出的 HLSL 代码
重要级别	高
预置条件	万维望远镜已将月球渲染到界面中
输入	月球位置坐标
操作步骤	1 根据月球位置设置摄像机朝向 2 观察月球渲染效果 3 查看本地预编译 HLSL 代码
预期输出	月球呈现阴影遮蔽效果，HLSL 代码功能为设置天体间阴影遮蔽

（5）土星

设置土星测试用例的目的是为了验证行星着色器是否正常设置星环效果，并通过输出的 HLSL 代码验证代码功能为设置星环效果。

表 5-6 土星渲染效果测试用例表

测试用例信息	描述
测试用例编号	Worldwide_Telescope_Rendering_Engine_PlantShader_Saturn_001
测试项目	土星渲染效果
测试标题	检测土星渲染效果和输出的 HLSL 代码
重要级别	高
预置条件	万维望远镜已将土星渲染到界面中
输入	土星位置坐标
操作步骤	1 根据土星位置设置摄像机朝向 2 观察土星渲染效果 3 查看本地预编译 HLSL 代码
预期输出	土星呈现星环效果，HLSL 代码功能为设置星环

5.1.4 测试结果

（1）太阳测试结果

执行测试用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Sun_001，经过万维望远镜渲染后的效果如图 5-1 所示。

通过输出的渲染效果可以看到，万维望远镜正确地渲染出了太阳的自发光效果。其中节选输出的功能代码如图 5-2 所示，该节选代码的 localDiffuseColor 设置了局部漫反射向量，litDiffuseColor 设置了自发光漫反射向量，最后将这两个颜色向量混合，生成像素点的颜色。这符合设置自发光效果的功能。

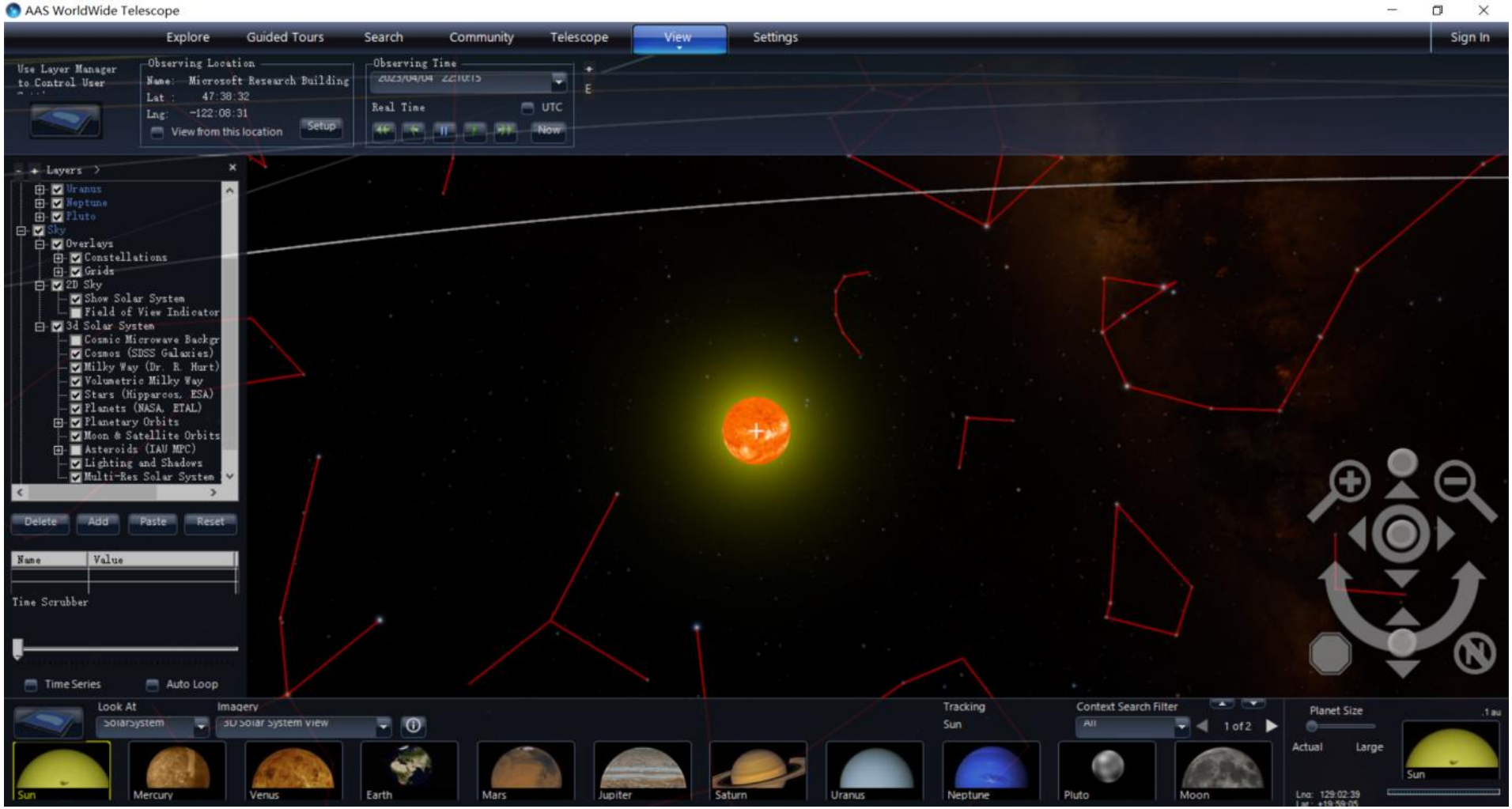

图 5-1 用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Sun_001 太阳渲染效果图

```
float4 PSMain(VS_OUT In) : SV_TARGET
{
    float4 localDiffuseColor = diffuseColor * mainTexture.Sample(MainTextureSampler, In.TexCoord);
    float3 litDiffuseColor = localDiffuseColor.rgb;
    float4 color = float4(litDiffuseColor, localDiffuseColor.a);
    return color;
}
```

图 5-2 用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Sun_001 太阳 HLSL 代码

（2）水星测试结果

执行测试用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Mercury_001，经过万维望远镜渲染后的效果如图 5-3 所示。

通过输出的渲染效果可以看到，万维望远镜正确地渲染出了水星的颜色贴图效果。其中节选输出的功能代码如图 5-4 所示，该节选代码首先使用采样器采样了主要纹理（mainTexture）的颜色值，然后通过与太阳的距离值计算出光源颜色向量 litDiffuseColor，最后将光源颜色向量与采样值混合，生成像素点的颜色。这符合设置颜色贴图的功能。

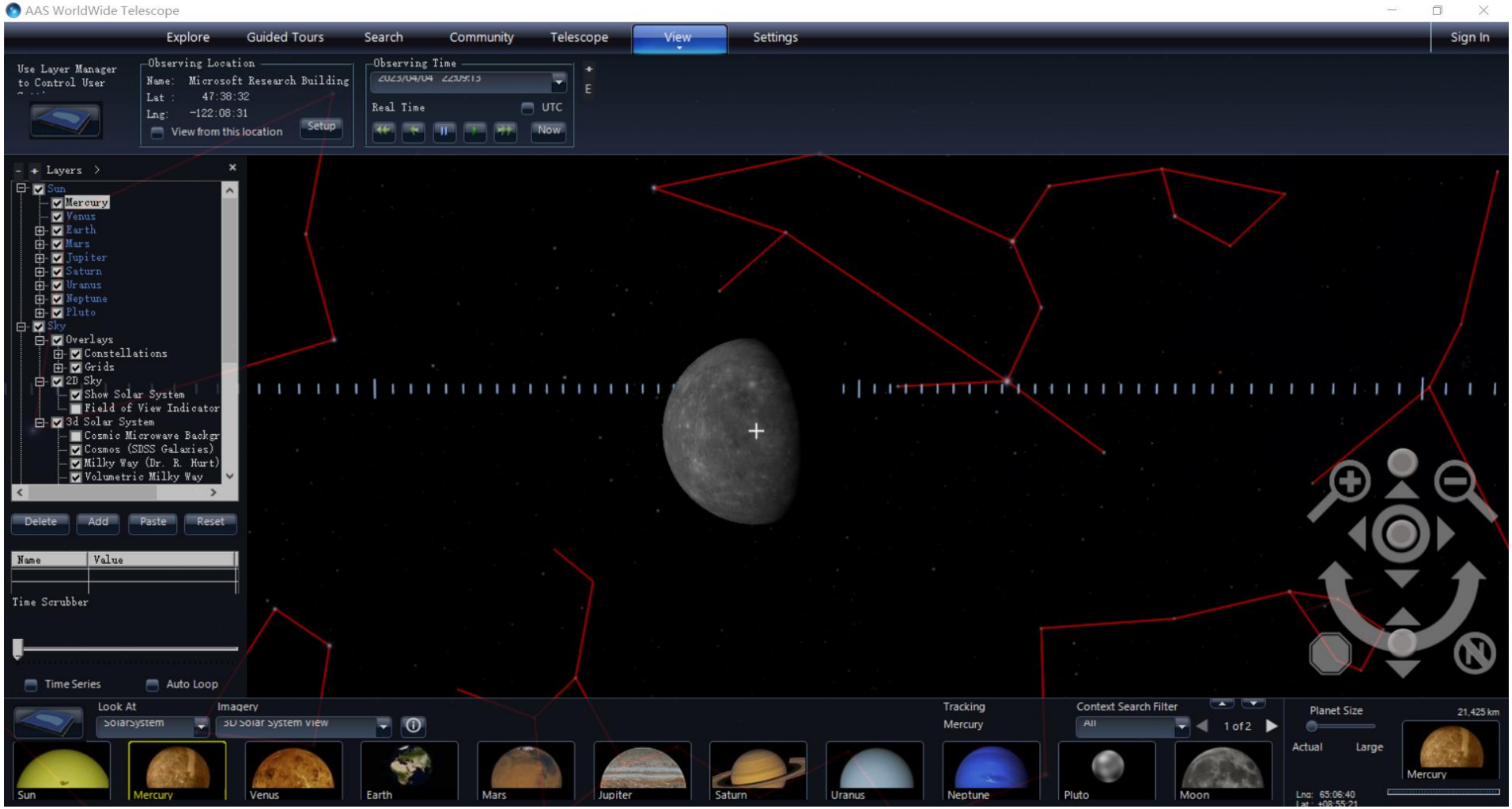

图 5-3 Worldwide_Telescope_Rendering_Engine_PlantShader_Mercury_001 水星渲染效果图

```
float4 PSMain(VS_OUT In) : SV_TARGET
{
    float4 localDiffuseColor = diffuseColor * mainTexture.Sample(MainTextureSampler, In.TexCoord)
;
    float3 N = normalize(In.ObjNormal);
    float3 litDiffuseColor = ambientLightColor;
    litDiffuseColor += hemiLightColor * (0.5 * (1.0 + dot(N, hemiLightUp)));
    {
        float3 L = lightDirection0;
        float3 lightColor = lightColor0;
        float NdotL = dot(N, L);
        litDiffuseColor += max(0.0, NdotL) * lightColor;
    }
    litDiffuseColor *= localDiffuseColor.rgb;
    float4 color = float4(litDiffuseColor, localDiffuseColor.a);
   float4 shadow = 1.0;
    return color;
}
```

图 5-4 Worldwide_Telescope_Rendering_Engine_PlantShader_Mercury_001 水星 HLSL 代码

（3）地球测试结果

首先执行测试用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_001，经过万维望远镜渲染后的效果如图 5-5 所示

通过输出的渲染结果可以看到，万维望远镜正确地渲染出了地球的大气效果。其中节选输出的功能代码如图 5-5 所示，这段代码使用了瑞利散射原理来模拟了大气对顶点位置的偏移，并且考虑了观测者视角的变化，从而实现了对地球模型顶点的偏移。这符合设置大气效果的功能。

然后执行测试用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_002，经过万维望远镜渲染后的效果如图 5-7 所示。

通过输出的渲染结果可以看到，万维望远镜正确地渲染出了地球的云层遮蔽效果。其中节选输出的功能代码如图 5-8 所示，这段代码在获取光源和主要纹理颜色值的方法上与水星相同，但是在获取颜色值之后还考虑了大气影响（即考虑了 AtmosphereExtinction、EclipseShadowCoord 和 AtmosphereInscatter），并根据其计算出阴影颜色（shadow），将阴影颜色与颜色值混合，生成像素点的颜色。这符合设置云层遮蔽效果的功能。

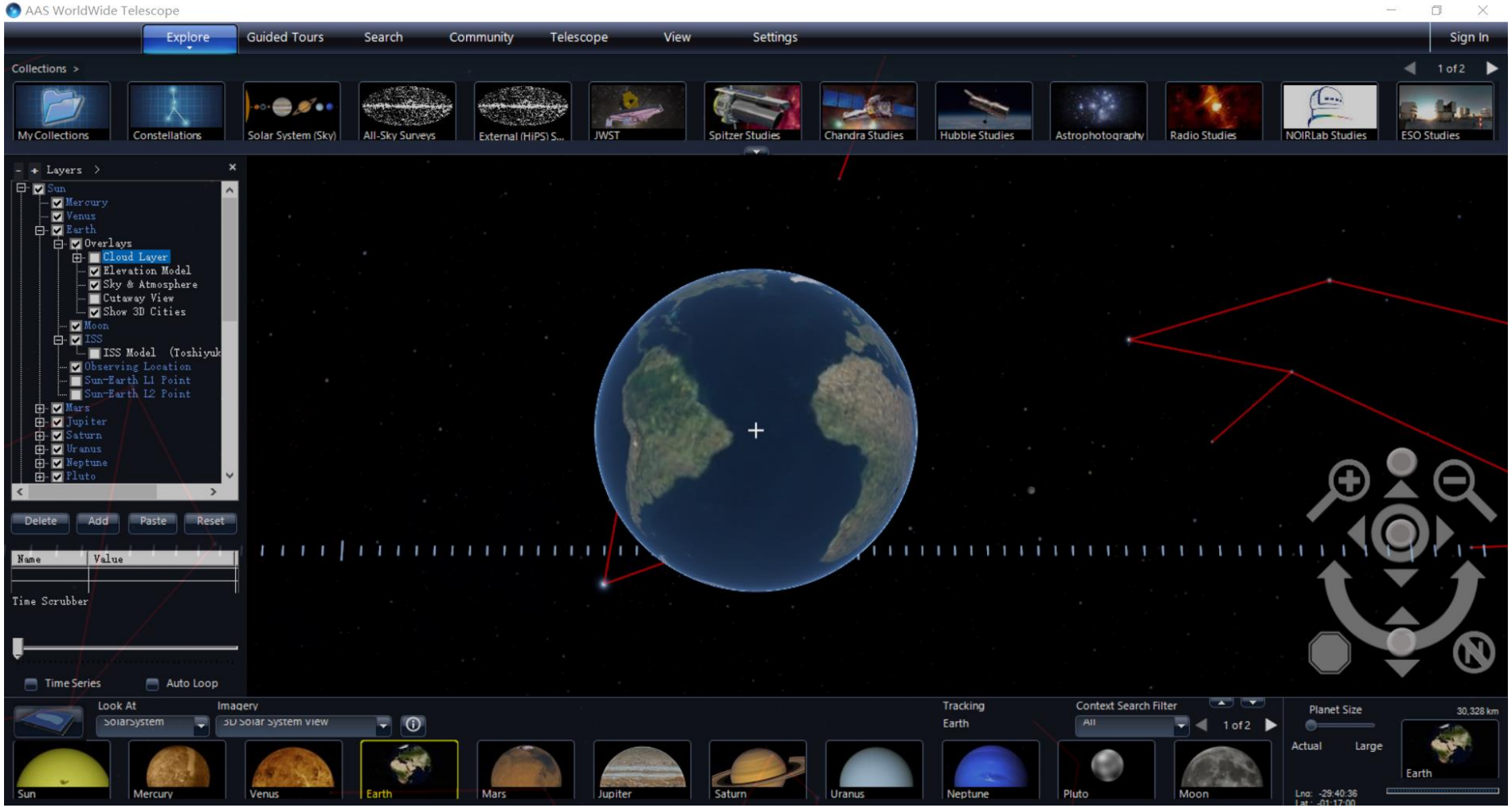

图 5-5 Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_001 地球（无云层）渲染效果图

```
VS_OUT VSMain(VS_IN In)
{
    float3 extinctionCoeff = rayleighScatterCoeff; // +mieScatterCoeff
    float3 objPos = In.ObjPos + atmosphereCenter;
    float3 camPos = cameraPosition + atmosphereCenter;
    float atmRadius = 1.0 + atmosphereHeight;
    float3 v = objPos - camPos;
    float a = dot(v, v);
    float b = dot(v, camPos);
    float c = dot(camPos, camPos) - atmRadius * atmRadius;
    float t = max(0.0, (-b - sqrt(b * b - a * c)) / a);
    float3 atmIntersection = camPos + t * v;
    float atmDistance = distance(atmIntersection, objPos);
    float3 inscatter = 0.0;
    float opticalDepth = 0.0;
    float3 p = atmIntersection;
    float3 step = (objPos - p) / 5.0;
    for (int i = 0; i < 5; ++i) {
        float pDotS = dot(p, sunDirection);
        b = pDotS;
        c = dot(p, p) - atmRadius * atmRadius;
        float u = -b + sqrt(max(0.0, b * b - c));
        float hMid = length(p + sunDirection * (u * 0.5)) - atmosphereZeroRadius;
        float tau = u * exp(min(50.0, -atmosphereInvScaleHeight * hMid));
        float distToCenter = length(p - pDotS * sunDirection);
        float blockedFraction = 1.0 - clamp((distToCenter - 0.8 * atmosphereZeroRadius) / (0.2 * atmosphereZeroRadius), 0.0, 1.0);
        c = dot(p, p) - atmosphereZeroRadius * atmosphereZeroRadius;
        if (b * b - c >= 0.0 && -sqrt(abs(b * b - c)) > b) tau = max(tau, 50.0 * blockedFraction);
        float h = length(p) - atmosphereZeroRadius;
        float d = exp(-atmosphereInvScaleHeight * h);
        inscatter += d * rayleighScatterCoeff * exp(-(opticalDepth + tau) * extinctionCoeff);
        opticalDepth += d * atmDistance / 5.0;
        p = p + step;
    }
    float hMid = length((atmIntersection + objPos) * 0.5) - atmosphereZeroRadius;
    Out.AtmosphereExtinction.rgb = exp(-opticalDepth * extinctionCoeff);
    Out.AtmosphereExtinction.a = hMid;
    float mu = dot(v, sunDirection) / length(v);
    float3 inscatterColor = pow(inscatter * atmDistance / 5.0, 1.0);
    Out.AtmosphereInscatter = float4(inscatterColor, opacity * dot(float3(2.0, 3.0, 4.0), inscatterColor));
    return Out;
}
```

图 5-6 Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_001 地球（无云层）HLSL 代码

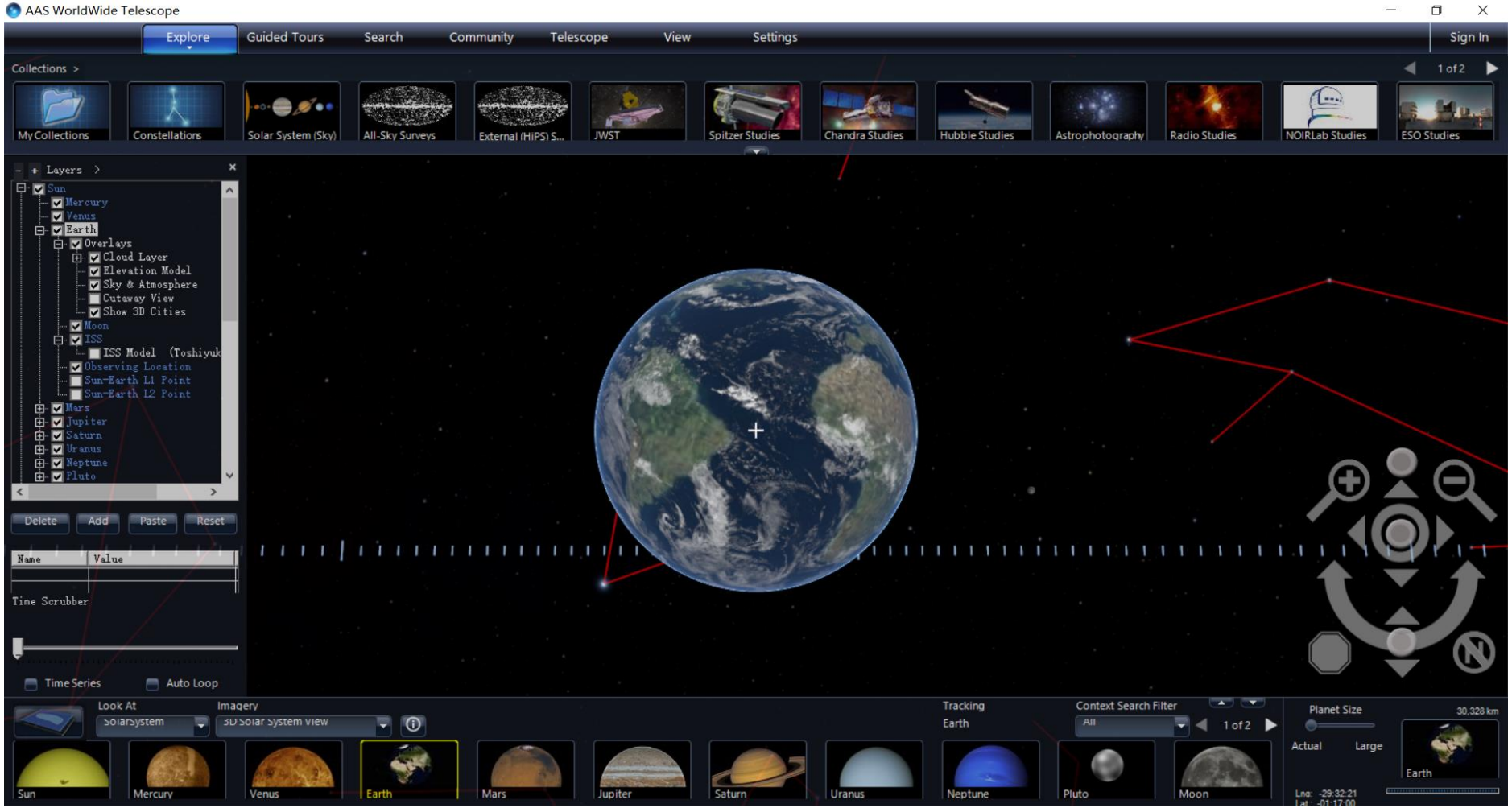

图 5-7 Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_002 地球（有云层）渲染效果图

```
float4 PSMain(VS_OUT In) : SV_TARGET
{
    float4 localDiffuseColor = diffuseColor * mainTexture.Sample(MainTextureSampler, In.TexCoord);
    float3 N = normalize(In.ObjNormal);
    float3 litDiffuseColor = ambientLightColor;
    litDiffuseColor += hemiLightColor * (0.5 * (1.0 + dot(N, hemiLightUp)));
    {
        float3 L = lightDirection0;
        float3 lightColor = lightColor0;
        float NdotL = dot(N, L);
        litDiffuseColor += max(0.0, NdotL) * lightColor;
    }
    litDiffuseColor *= localDiffuseColor.rgb;
    float4 color = float4(litDiffuseColor, localDiffuseColor.a);
   float4 shadow = 1.0;
    shadow *= eclipseTexture.Sample(ClampTextureSampler, In.EclipseShadowCoord0.xy / In.EclipseShadowCoord0.w);
   color *= shadow;
   color = color * float4(In.AtmosphereExtinction.rgb, 1.0) + In.AtmosphereInscatter * shadow * (In.AtmosphereExtinction.a < -0.001 ? 0.0 : 1.0);
    return color;
}
```

图 5-8 Worldwide_Telescope_Rendering_Engine_PlantShader_Earth_002 地球（有云层）HLSL 代码

（4）月球测试结果

首先执行测试用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Moon_001，经过万维望远镜渲染后的效果如图 5-9 所示。

通过输出的渲染效果可以看到，万维望远镜正确地渲染出了天体间阴影遮蔽效果。其中节选输出的功能代码如图 5-10 所示，其中 VSMain 使用了叠加纹理（EclipseShadowCoord0）来模拟了天体间遮挡导致的顶点位置偏移。

PSMain 在获取光源和主要纹理颜色值的方法上与水星相同，但是在获取颜色值之后还考虑了对叠加纹理（eclipseTexture）进行采样器采样的颜色值，将叠加纹理采样的颜色值与之前的颜色值进行混合，生成像素点的颜色。这符合设置天体间阴影遮蔽效果的功能。

图 5-9 Worldwide_Telescope_Rendering_Engine_PlantShader_Moon_001 月球渲染效果图

```
VS_OUT VSMain(VS_IN In)
{
    VS_OUT Out;
    Out.ProjPos = mul(In.ObjPos, matWVP);
    Out.TexCoord = In.TexCoord;
    float4 cameraSpacePos = mul(In.ObjPos, matWV);
    Out.ObjNormal = In.ObjNormal;
    Out.ObjPos = In.ObjPos.xyz;
    Out.EclipseShadowCoord0 = mul(cameraSpacePos, matEclipseShadow0);
    return Out;
}
float4 PSMain(VS_OUT In) : SV_TARGET
{
    float4 localDiffuseColor = diffuseColor * mainTexture.Sample(MainTextureSampler, In.TexCoord);
    float3 N = normalize(In.ObjNormal);
    float3 litDiffuseColor = ambientLightColor;
    litDiffuseColor += hemiLightColor * (0.5 * (1.0 + dot(N, hemiLightUp)));
    {
        float3 L = lightDirection0;
        float3 lightColor = lightColor0;
        float NdotL = dot(N, L);
        litDiffuseColor += max(0.0, NdotL) * lightColor;
    }
    litDiffuseColor *= localDiffuseColor.rgb;
    float4 color = float4(litDiffuseColor, localDiffuseColor.a);
    float4 shadow = 1.0;
    shadow *= eclipseTexture.Sample(ClampTextureSampler, In.EclipseShadowCoord0.xy / In.EclipseSha
dowCoord0.w);
    color *= shadow;
    return color;
}
```

图 5-10 Worldwide_Telescope_Rendering_Engine_PlantShader_Moon_001 月球 HLSL 代码

（5）土星测试结果

执行测试用例 Worldwide_Telescope_Rendering_Engine_PlantShader_Saturn_001，经过万维望远镜渲染后的效果如图 5-11 所示。

通过输出的渲染效果可以看到，万维望远镜正确地渲染出了星环效果。其中节选输出的功能代码如图 5-12 所示，通过 VSMain 代码可以观察到，代码使用了环阴影纹理坐标（RingShadowCoord）来模拟了天体间遮挡导致的顶点位置偏移。

通过 PSMain 代码可以观察到，由于土星距离太阳过远，所以仅考虑太阳光向量（litDiffuseColor），因此颜色值用太阳光向量与主要纹理采样后的颜色值混合即可。在获取颜色值之后还考虑了对环阴影纹理进行采样器采样的颜色值，将环阴影纹理采样的颜色值与之前的颜色值进行混合，生成像素点的颜色。这符合设置星环的功能。

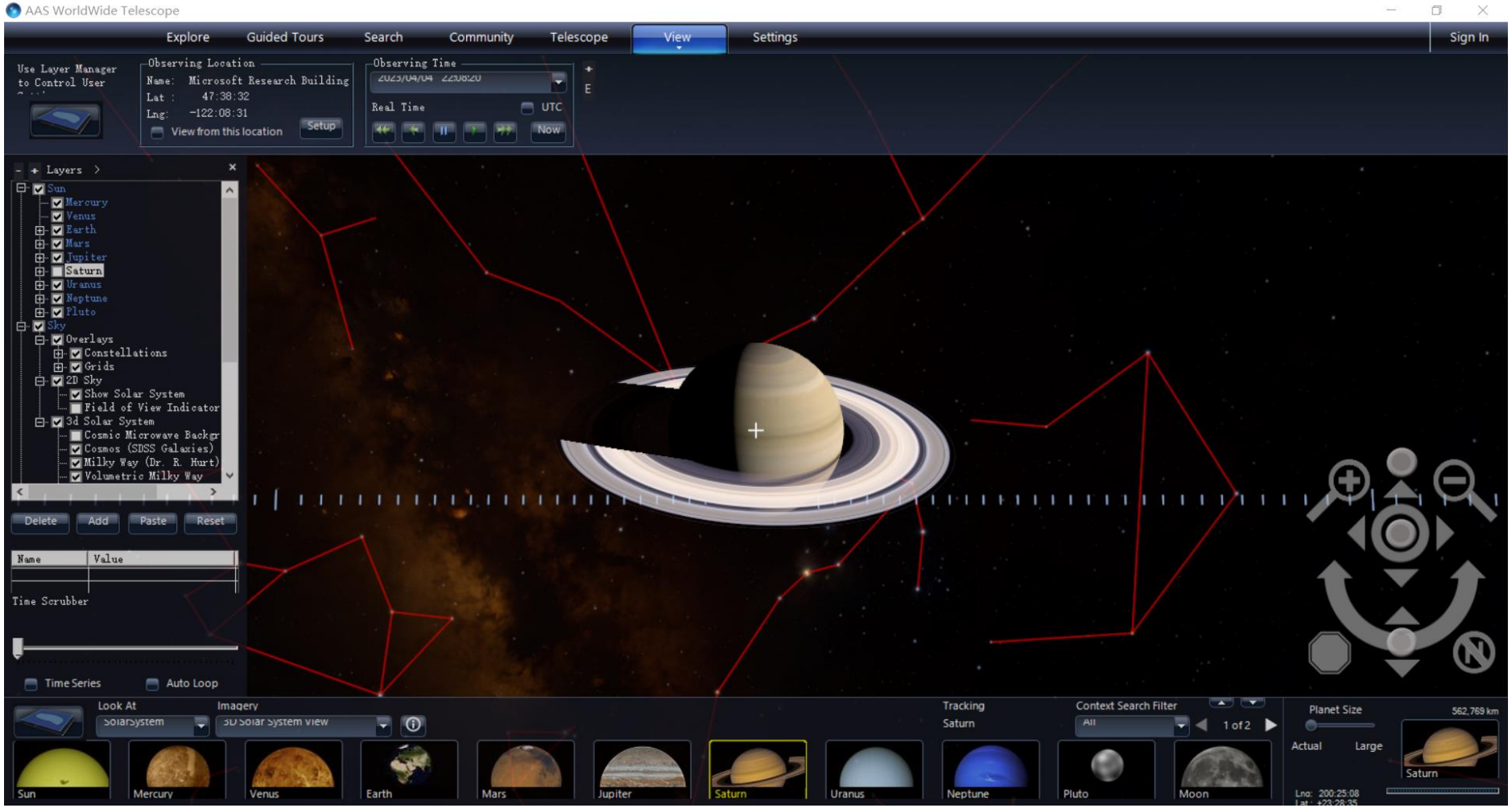

图 5-11 Worldwide_Telescope_Rendering_Engine_PlantShader_Saturn_001 土星渲染效果图

```
1.  VS_OUT VSMain(VS_IN In)
2.  {
3.      VS_OUT Out;
4.      Out.ProjPos = mul(In.ObjPos, matWVP);
5.      Out.TexCoord = In.TexCoord;
6.      Out.ObjNormal = In.ObjNormal;
7.      Out.ObjPos = In.ObjPos.xyz;
8.      float3 v1 = cross(sunDirection, float3(0, 1, 0));
9.      v1 = normalize(v1);
10.     float3 v2 = cross(v1, sunDirection);
11.     Out.RingShadowCoord = float4(dot(v1, In.ObjPos), dot(v2, In.ObjPos), dot(sunDirection, In.Obj
    Pos), 0.0);
12.     return Out;
13. }
14. float4 PSMain(VS_OUT In) : SV_TARGET
15. {
16.     float4 localDiffuseColor = diffuseColor * mainTexture.Sample(MainTextureSampler, In.TexCoord)
    ;
17.     float3 litDiffuseColor = localDiffuseColor.rgb;
18.     float4 color = float4(litDiffuseColor, localDiffuseColor.a);
19.     float4 shadow = 1.0;
20.     shadow.rgb *= In.RingShadowCoord.z > 0.0 ? 1.0 : step(1.0, length(In.RingShadowCoord.xy));
21.     color *= shadow;
22.     return color;
23. }
```

图 5-12 Worldwide_Telescope_Rendering_Engine_PlantShader_Saturn_001 土星 HLSL 代码

5.2 测试结论

经过测试，万维望远镜项目的行星着色器能够正常渲染太阳、火星、地球和月球，输出的 HLSL 代码符合预期的行星着色器样式。

在测试过程中，通过提供不同的输入参数来测试不同的行星，以确保行星着色器的稳定性和可靠性。记录下所有着色器库中的 HLSL 代码以及其对应的效果图，可以为后续测试提供参考，也有利于着色器的管理和维护。

第 6 章 总结

本研究对万维望远镜渲染引擎进行了系统性的分析，对重要模块进行了关键注释和技术文档的补全，以达成以下两个目标：补全缺失的万维望远镜渲染引擎技术文档，完备了重要模块的注释。具体而言，完成了补全万维望远镜渲染流程技术文档和行星着色器 HLSL 实现细节文档；将代码注释量添加到 4711 行，总代码量为 13066 行，代码注释率达到 36.1%，高于软件工程推荐的 20%~30%注释率区间。因此，本研究达成了方便开发人员对万维望远镜渲染引擎进行二次开发的研究目的，并成功增强了万维望远镜渲染引擎代码的可维护性和可读性。

在系统性分析渲染引擎代码中，首先应从渲染引擎的功能下手，对渲染引擎进行需求分析、数据建模、功能建模和行为建模，从而构建出渲染引擎的主要框架。然后根据构建出的主要框架，选择性地阅读渲染引擎的源代码，以找出实现主要功能的代码位置。之后先粗略地阅读这些代码，理清渲染引擎的渲染流程和工作方式。然后根据渲染引擎的工作方式划分模块，并仔细阅读每个模块的代码，从而理清渲染引擎重要功能模块的工作原理和层次结构。最后对主要功能进行测试，以验证系统分析的可靠性。

参考文献

[1] Alexandre Mutel.SharpDX Wiki [DB/OL]. http://sharpdx.org/wiki/, 2016

[2] Albahari J , Albahari B . C# 7.0 in a Nutshell, 7th Edition. 2019.

[3] Microsoft.Direct3D11[DB/OL].https://learn.microsoft.com/zh-cn/windows/win32/api/_direct3d11/,2023

[4] Luna F. Introduction to 3D Game Programming with DirectX 12[M]. Mercury Learning & Information, 2020

[5] Xu Y, Cui C, Fan D, et al. IVOA HiPS Implementation in the Framework of WorldWide Telescope:, 10.1016/j.ascom.100380[P]. 2020

[6] Michaelis M . Essential C# 7.0, 6th Edition, 2018

[7] IanSommerville. Software engineering,10th Edition, 2018

[8] NADC.WWT[DB/OL].https://nadc.china-vo.org/wwt, 2019

[9] Microsoft.CSharp[DB/OL].https://learn.microsoft.com/zh-cn/dotnet/csharp/,2023

[10] Kyle Halladay. Shader 开发实战,1th Edition,2021

[11] AAS. WorldWide Telescope Windows Client [DB/OL]. https://github.com/WorldWideTelescope, 2022

[12] Celestia. Celestia Windows Project[DB/OL]. https://github.com/CelestiaProject/Celestia, 2022

附　录

代码注释打包在与论文同级的附录文件夹下的代码文件夹中，其中包括 RenderContext11.cs 、 Shaders.cs 、 Shaders11.cs 和 Texture11.cs 这四个文件。RenderContext11.cs 实现了渲染上下文模块的功能，Shaders.cs 和 Shaders11.cs 实现了着色器模块，Texture11.cs 实现了纹理模块。

相关技术文档打包在与论文同级的附录文件夹下的技术文档文件夹中，其中包括打包常量寄存器技术文档.md、行星着色器采样器代码技术文档.md、行星着色器顶点着色器技术文档.md、行星着色器几何着色器技术文档.md、行星着色器纹理常量寄存器技术文档.md、行星着色器像素着色器技术文档.md 和渲染流程.md 七个文件。具体情况如附录表 A 所示。

附录表 A 技术文档描述表

技术文档名称	内容描述
打包常量寄存器技术文档.md	介绍了渲染引擎中行星着色器的手动打包原理
行星着色器采样器代码技术文档.md	介绍了行星着色器采样器的定义和实现原理
行星着色器顶点着色器技术文档.md	包含行星着色器顶点着色器的注释和实现流程分析
行星着色器几何着色器技术文档.md	包含行星着色器几何着色器的注释和实现流程分析
行星着色器纹理常量寄存器技术文档.md	介绍了行星着色器使用的几个纹理类型和来源
行星着色器像素着色器技术文档.md	介绍了像素着色器的注释和实现流程分析
渲染流程.md	介绍了万维望远镜的渲染流程

致　谢

在此，我要首先向我的本科指导老师王威表示最诚挚的感谢。在我的本科学习生涯中，王老师给予了我非常重要的指导和帮助，帮助我解决了许多学习和工作上的难题，使我在学术上和个人发展上都取得了显著的进展。

在本篇论文即将完成之际，我感到内心十分激动。在整个研究过程中，我得到了无数可敬的师长、同学和朋友们的无私帮助与支持，他们的付出和贡献是我能够顺利完成这篇论文的关键，在这里请接受我诚挚的谢意!