**实 验 一 报 告**

**（2020-2021学年第二学期）**

**3D游戏引擎架构设计基础**

**（Foundations of 3D Game Engine Architecture Design）**

|  |  |  |
| --- | --- | --- |
| 学生姓名： | 学号： | 年级，班级： |
| 成绩： | 任课教师签名： | 日期： |
| 实验题目： | | |
| 目录：   1. 实验内容描述: 引擎名称和版本,将分析的引擎模块名称 2. 将分析的引擎模块的主要功能描述 3. 将分析的引擎模块的主要类和类关系描述 4. 类的数据成员和成员函数描述 5. 总结 | | |
| 实验报告：   * 实验内容描述: 引擎名称和版本,将分析的引擎模块名称   本次分析的是 UnrealEngine 4.27.2 的模块管理部分   * 将分析的引擎模块的主要功能描述   这个模块主要作用是管理整个引擎各个模块的加载和卸载，以及跟踪所有的当前加载的模块。   * 将分析的引擎模块的主要类和类关系描述   ImoduleInterface：虚类，所有模块的基类，抽象出模块管理所需的接口  FModuleManager：管理各个模块  IInputDeviceModule：管理输入设备的模块，作为各种模块的例子分析  具体关系见下图   * 类的数据成员和成员函数描述   ImoduleInterface  成员函数：   * + - ~IModuleInterface()   析构函数   * + - StartupModule()   配置模块，DLL加载后会被调用   * + - ShutdownModule()   卸载模块，在模块被卸载之前调用   * + - PostLoadCallback()   在模块重新加载后调用   * + - PreUnloadCallback()   在模块被卸载之前调用   * + - SupportsDynamicReloading()   模块是否允许热重载   * + - SupportsAutomaticShutdown()   模块是否支持应用退出时关闭   * + - IsGameModule()   模块是否是游戏相关代码  FModuleManager  成员函数：   * + Manager实例管理     - FModuleManager()   构造函数   * + - ~FModuleManager()   析构函数   * + - Get()   获取单例   * + - TearDown()   销毁单例   * + 模块文件管理     - AddBinariesDirectory(InDirectory, bIsGameDirectory)   添加引擎二进制文件目录   * + - SetGameBinariesDirectory(InDirectory)   设置游戏二进制文件目录   * + - GetGameBinariesDirectory()   获取游戏二进制文件目录   * + - AddExtraBinarySearchPaths()   添加额外的文件搜索路径   * + - ResetModulePathsCache()   清除模块路径缓存   * + - FindModules(WildcardWithoutExtension, OutModules)   在磁盘上查找与指定通配符匹配的可加载模块的模块文件   * + - IsModuleUpToDate(InModuleName)   检查指定模块是否存在并与当前引擎版本兼容   * + - MakeUniqueModuleFilename(InModuleName, UniqueSuffix, UniqueModuleFileName)   通过添加随机后缀并检查文件冲突，为指定的模块名称生成唯一的文件名   * + - FindModulePaths(NamePattern, OutModulePaths)   查找与给定名称通配符匹配的模块路径   * + - FindModulePathsInDirectory(DirectoryName, bIsGameDirectory, OutModulePaths)   在指定目录查找与给定名称通配符匹配的模块   * + - GetModuleFilename(ModuleName)   获取模块的文件名   * + - SetModuleFilename(ModuleName, Filename)   设置模块的文件名，模块不会立即重载   * + 加载模块     - AddModule(InModuleName)   将一个模块添加到模块列表中，除非它是已知的   * + - LoadModule(InModuleName)   加载指定模块   * + - LoadModuleChecked(InModuleName)   加载指定的模块，并检查以确保它存在   * + - LoadModuleWithCallback(InModuleName, Ar)   加载一个在内存中的模块，然后调用它的 PostLoad   * + - LoadModuleWithFailureReason(InModuleName, OutFailureReason)   加载指定模块并返回失败原因   * + - LoadModuleChecked(InModuleName)   加载指定的模块，并检查以确保它存在   * + - LoadModulePtr(InModuleName)   加载指定模块   * + - AddModuleToModulesList(InModuleName, ModuleInfo)   将模块信息添加到列表中   * + - WarnIfItWasntSafeToLoadHere(InModuleName)   警告在非主线程中加载模块是不安全的   * + 查询模块     - GetModule(InModuleName)   获取指定模块   * + - IsModuleLoaded(InModuleName)   检查指定模块是否加载   * + - QueryModule(InModuleName, OutModuleStatus )   查询指定模块名称的信息   * + - QueryModules(OutModuleStatuses)   查询所有当前已知模块的信息   * + - GetModuleChecked(ModuleName)   获取指定模块，并检查以确保它存在   * + - GetModulePtr(ModuleName)   获取指定模块   * + - GetModulePtr\_Internal(ModuleName)   “获取指定模块”的内部实现   * + - GetModuleCount()   获取已加载模块的数量   * + - FindModule(InModuleName)   线程安全的查找指定模块   * + - FindModuleChecked(InModuleName)   线程安全的查找指定模块，并检查以确保它存在   * + - HasAnyOverridenModuleFilename()   确定是否加载了任何非默认模块，例如热重载的模块   * + - DoesLoadedModuleHaveUObjects(ModuleName)   查询指定模块是否包含 UObject   * + - ModuleExists(ModuleName)   指定名称的模块是否存在   * + 更新模块     - RefreshModuleFilenameFromManifest(InModuleName)   从清单中更新新模块的文件名   * + - RefreshModuleFilenameFromManifestImpl(InModuleName, ModuleInfo)   “从清单中更新新模块的文件名”的具体实现   * + 卸载模块     - AbandonModule(InModuleName)   放弃一个已加载的模块，将其留在内存中，但不再在模块管理器中跟踪它   * + - UnloadModule(InModuleName, bIsShutdown)   卸载特定模块   * + - UnloadOrAbandonModuleWithCallback(InModuleName, Ar)   调用模块的 PreUnload ，然后根据能否热重载，卸载或放弃模块   * + - AbandonModuleWithCallback(InModuleName)   调用模块的 PreUnload 然后放弃模块   * + - UnloadModulesAtShutdown()   按照配置顺序反向卸载所有模块   * + 其他     - OnModulesChanged()   返回模块更改事件   * + - OnProcessLoadedObjectsCallback()   处理加载的 UObjects 后的回调   * + - IsPackageLoadedCallback()   在加载包含 UObjects 的模块时的回调   * + - StartProcessingNewlyLoadedObjects()   由引擎调用，告知现在引擎可以正常的处理Uobjects   * + - GetUBTConfiguration()   获取UBT所需的构建配置   * + - SaveCurrentStateForBootstrap(Filename)   将当前状态保存到文件中以引导其他进程   * + - SerializeStateForBootstrap\_Impl(Ar)   “序列化当前状态”的具体实现   * + - Exec(Inworld, Cmd, Ar)   执行命令接口   * + - RegisterStaticallyLinkedModule(InModuleName, InInitializerDelegate)   为静态链接的模块注册一个初始化器   * + - ProcessPendingStaticallyLinkedModuleInitializers()   将挂起的模块初始化程序添加到容器中  数据成员：   * Modules   记录所有模块。将模块名称映射到模块信息，无论是否加载。   * PendingStaticallyLinkedModuleInitializers   待注册的模块名称   * StaticallyLinkedModuleInitializers   记录每个静态链接模块的初始化程序   * bCanProcessNewlyLoadedObjects   标记能否处理 UObjects   * bExtraBinarySearchPathsAdded   标记是否添加过额外的搜索路径   * ModulePathsCache   已知模块路径的缓存   * ModulesChangedEvent   模块更改的事件   * ProcessLoadedObjectsCallback   添加新 UObjects 时的回调   * IsPackageLoaded   链接到支持 UObjects 的应用时调用的委托   * EngineBinariesDirectories   引擎二进制目录   * PendingEngineBinariesDirectories   挂起的引擎二进制目录   * GameBinariesDirectories   游戏二进制目录   * PendingGameBinariesDirectories   挂起的游戏二进制目录   * BuildId   用于验证模块清单的 ID   * ModulesCriticalSection   控制对模块的访问  IInputDeviceModule  成员函数：   * GetModularFeatureName()   获取模块名称   * StartupModule()   初始化模块   * Get()   获取模块实例   * IsAvailable()   是否可用   * CreateInputDevice(InMessageHandler)   创建新的输入设备   * 总结   从模块管理的设计中，我看出了虚幻4，作为一个极为强大、运用极为广泛的引擎，它的模块化设计，它的模块加载、重载、热更新等功能。  从它的代码中，我学会了如何加载、储存、管理一个游戏引擎所需要的庞大的各种模块，了解到了这不是一个容易的事情。 | | |