**实 验 二 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：   1. 本次实验分析的是：虚幻4的世界、关卡管理 2. 这个模块主要管理虚幻4中的世界、关卡，提供添加AActor，添加Ulevel等功能。 3. 主要关系类为UWorld，ULevel，AActor。关系描述：    1. UWorld 是表示地图或沙盒的顶级对象，一个世界包含一个或多个ULevel对象。    2. ULevel是一组 Actor（灯光、体积、网格实例等），包含一个的 AActor 列表。每个关卡都有一个 World 作为其 Outer。    3. AActor 是可以在关卡中放置或生成的 Object 的基类。AActor 可能包含一组 UActorComponent，可用于控制 Actor 如何移动、如何渲染等。 4. 类的数据成员和成员函数描述   UWorld: 包含一个单一的持久关卡，还有可选的流式关卡列表   1. Level 管理    1. TArray<class ULevel\*> Levels   当前在这个世界上的一系列 ULevel   * 1. const TArray<class ULevel\*>& GetLevels() const;   返回 ULevel 列表   * 1. bool AddLevel( ULevel\* InLevel );   添加 ULevel 到列表中   * 1. bool RemoveLevel( ULevel\* InLevel );   从列表中删除指定的 ULevel   * 1. ULevel\* GetLevel( int32 InLevelIndex ) const;   返回指定序号的 ULevel   * 1. bool ContainsLevel( ULevel\* InLevel ) const;   返回是否包含指定的 ULevel   1. 持久 Level    1. ULevel\* PersistentLevel;   包含世界信息、默认画笔和在游戏过程中生成的 AActor 等内容的持久关卡   * 1. class ULevel\* GetCurrentLevel() const;   获取当前世界的 ULevel   1. 流式 Level 2. TArray<ULevelStreaming\*> StreamingLevels;   流式关卡列表。Level 集合。 ULevel 由 FName（包名）引用以避免序列化引用。 还包含世界单位的偏移量   1. Level 显示管理 2. class ULevel\* CurrentLevelPendingVisibility;   指向队列中要显示的当前 ULevel 的指针，如果没有待处理，则为 NULL   1. ULevel\* GetCurrentLevelPendingVisibility() const   返回显示过程中的 ULevel（如果有）   1. class ULevel\* CurrentLevelPendingInvisibility;   指向队列中要设为不可见的当前 ULevel 的指针，如果没有待处理，则为 NULL   1. ULevel\* GetCurrentLevelPendingInvisibility() const   返回隐藏过程中的 ULevel（如果有）   1. utils 2. void ModifyLevel(ULevel\* Level) const;   修改指定的 ULevel   1. 编辑器 Utils 2. void AddToWorld( ULevel\* Level, const FTransform& LevelTransform = FTransform::Identity, bool bConsiderTimeLimit = true );   将传入的 ULevel 与 UWorld 相关联   1. void RemoveFromWorld( ULevel\* Level, bool bAllowIncrementalRemoval = false );   将传入的 ULevel 与 UWorld 分离   1. Actor 管理 2. AActor\* SpawnActor( UClass\* InClass, FVector const\* Location=NULL, FRotator const\* Rotation=NULL, const FActorSpawnParameters& SpawnParameters = FActorSpawnParameters() );   创建 Actor   1. AActor\* SpawnActor( UClass\* Class, FTransform const\* Transform, const FActorSpawnParameters& SpawnParameters = FActorSpawnParameters());   创建 Actor   1. AActor\* SpawnActorAbsolute( UClass\* Class, FTransform const& AbsoluteTransform, const FActorSpawnParameters& SpawnParameters = FActorSpawnParameters());   创建 Actor   1. T\* SpawnActorDeferred(UClass\* Class, FTransform const& Transform, ...)   创建 Actor   1. bool ContainsActor( AActor\* Actor ) const;   返回传入的actor是否是任何加载的关卡actor数组的一部分   1. void RemoveActor( AActor\* Actor, bool bShouldModifyLevel ) const;   从 AActor 列表中删除传入的 AActor   1. bool DestroyActor( AActor\* Actor, bool bNetForce, bool bShouldModifyLevel);   从其关卡的actor列表中删除actor并清理引擎的内部状态   1. bool EditorDestroyActor( AActor\* Actor, bool bShouldModifyLevel );   编辑器中调用的 DestroyActor() 包装器   1. 相机 Actor    1. void RegisterAutoActivateCamera(ACameraActor\* CameraActor, int32 PlayerIndex);   注册一个为 PlayerController 自动激活的 CameraActor。   1. 网络 Actor    1. void AddNetworkActor( AActor\* Actor );   添加网络 Actor   * 1. void RemoveNetworkActor( AActor\* Actor ) const;   删除网络 Actor  ULevel包含一个的 AActor 列表。每个关卡都有一个 World 作为其 Outer。   1. Actor 管理    1. TArray<AActor\*> Actors;   此 ULevel 中所有 AActor 的数组, 由 FActorIteratorBase 和派生类使用   * 1. TArray<AActor\*> ActorsForGC;   在此级别中要暴露于 GC 的 AActor 数组。 所有其他actor都将通过 ULevelActorContainer 引用   1. Utils    1. static bool IsNetActor(const AActor\* Actor);   在内部用于确定哪些 AActor 应该进入 UWorld 的 NetworkActor 列表   * 1. void CreateReplicatedDestructionInfo(AActor\* const Actor);   在 DestroyedReplicatedStaticActors 列表中填充一个 Actor 的实例   * 1. void RegisterActorForAutoReceiveInput(AActor\* Actor, const int32 PlayerIndex);   注册一个应该在创建玩家输入堆栈时添加到玩家输入堆栈的actor  AActor在此流程中作为数据结构流动，无交互。  交互流程：  虽然Level持有Actor对象，但是只提供了基本的CRUD接口。Actor对象的主要交互功能由World管理执行。  其中较为复杂、关键的逻辑为World的添加Actor对象的接口。World共提供了四种不同的添加Actor的接口，其中三个为另一个的包装版本。剩下的接口负责往Level添加Actor对象。这里的关键代码为：（UWorld::SpawnActor LevelActor.cpp Line 308 ~ 594）  // 先进行 Class 检查  // actually make the actor object  AActor\* const Actor = NewObject<AActor>(LevelToSpawnIn, Class, NewActorName, ActorFlags, Template, false, nullptr, ExternalPackage); // Line 508  // 再进行 Actor 检查  LevelToSpawnIn->Actors.Add( Actor ); // Line 539  LevelToSpawnIn->ActorsForGC.Add(Actor); // Line 540  return Actor; // Line 593  从中我们可以看出他的主要逻辑，先进行合法性检查，然后new出这个Object，再添加到Level的列表中   1. 总结   从UE4的World对Actor的管理流程中，我了解了一个复杂系统中，所有权设计的重要性，有着优秀的设计，在代码层面上可以极大地简化编码的心智负担，也可以令系统有着极大的可拓展性，极大的可维护性。 | | |