**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：胡嘉鸿 | 学号：201930380501 | 年级，班级：2019级1班 |
| 成绩： | 任课教师签名： | 日期：2022.4.12 |
| 实验题目：OGRE或Panda3D开源游戏引擎场景、渲染、资源等核心类的分析 | | |
| 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. 实验内容描述：   本实验选择的引擎为OGRE，分析的引擎模块名称为：场景组织体系模块。  本次实验在实验一的基础上，即（Node、Scene Node、Bone）这三个类的基础上，对Node类的结构与代码实现进行更进一步的详细分析。  2、该模块主要负责管理出现在屏幕上的所有物体的承载节点，保持对这些节点的跟踪。  3、Node、SceneNode、Bone这三个类的关系如下图所示：    4、  Node类的类数据成员如下：   |  |  |  | | --- | --- | --- | | 名称 | 类型 | 描述 | | mParent | Node\* | 指向父节点的指针 | | mChildren | vector<Node\*> | 指向孩子的指针集合 | | mName | String | 节点的名字 | | mNeedParentUpdate | Bool | 用于指示来自父级的自身转换已过期的标志 | | mNeedChildUpdate | Bool | 指示所有子项都需要更新的标志 | | mParentNotified | Bool | 指示父级已收到更新请求通知的标志 | | mQueuedForUpdate | Bool | 指示节点已排队等待更新的标志 | | mInheritOrientation | Bool | 存储此节点是否从其父节点继承方向 | | mInheritScale | Bool | 存储此节点是否从其父节点继承缩放 | | mCachedTransformOutOfDate | Bool | 存储变换矩阵是否已过期 | | mOrientation | Quaternion | 存储节点相对于其父节点的方向。 | | mPosition | Vector3 | 存储节点相对于其父节点的位置/平移。 | | mScale | Vector3 | 存储应用于此节点的比例因子 | | mCachedTransform | Affine3 | 缓存的变换矩阵 | | mListener | Listener\* | 节点侦听器 | | msQueuedUpdates | Static vector<Node\*> | 等待更新的排队队列,静态变量 |   与挂载、删除父节点、子节点相关的函数   |  |  |  | | --- | --- | --- | | 名称 | 返回值类型 | 描述 | | setParent(Node\* parent) | Void | 设置父亲节点 | | createChild(const Vector3& inTranslate, const Quaternion& inRotate) | Node\* | 创建无名子节点，进行挂载 | | createChild(const String& name, const Vector3& inTranslate, const Quaternion& inRotate) | Node\* | 创建有名子节点，进行挂载 | | addChild(Node\* child) | Void | 将一个节点挂载在当前节点名下 | | getChild(unsigned short index) | Node\* | 通过下标获取该子节点 | | removeChild(unsigned short index) | Node\* | 将该下标对应的子节点分离出来 | | removeChild(Node\* child) | Node\* | 将该子节点分离出来 | | removeAllChildren(void) | Void | 移除所有子节点 | | getChild(const String& name) | Node\* | 通过名字获取该子节点 | | removeChild(const String& name) | Node\* | 通过名字分离该子节点 |   与节点的位置、方向、缩放比例相关的函数   |  |  |  | | --- | --- | --- | | 名称 | 返回值类型 | 描述 | | \_getFullTransform(void) | Affine3 | 更新变换矩阵（若得知该变换矩阵已过期） | | setOrientation( const Quaternion & q ) | Void | 设置相对于父节点的方向 | | setOrientation( Real w, Real x, Real y, Real z) | Void | 设置相对于父节点的方向 | | resetOrientation(void) | Void | 重置方向 | | setPosition(const Vector3& pos) | Void | 设置相对于父节点的位置 | | translate(const Vector3& d, TransformSpace relativeTo) | Void | 选择变换所涉及的空间，进行节点移动 | | rotate(const Quaternion& q, TransformSpace relativeTo) | Void | 选择变换所涉及的空间，进行节点的旋转 | | convertWorldToLocalPosition( const Vector3 &worldPos ) | Void | 转换世界坐标系的位置到本地坐标系 | | convertWorldToLocalOrientation( const Quaternion &worldOrientation ) | Void | 转换世界坐标系的方向到本地坐标系 | | scale(const Vector3& inScale) | Void | 缩放 | | setInitialState(void) | Void | 设置初始状态 |   与节点更新有关的函数   |  |  |  | | --- | --- | --- | | 名称 | 返回值类型 | 描述 | | \_update(bool updateChildren, bool parentHasChanged) | Void | 更新节点本身 | | \_updateFromParent(void) | Void | 若有父节点，则要先看看父节点有没有更新，获取父节点更新后的状态，才能更新自己本身 | | needUpdate(bool forceParentUpdate) | Void | 将该节点设置为需要更新的状态 | | requestUpdate(Node\* child, bool forceParentUpdate) | Void | 请求更新某一个子节点 | | cancelUpdate(Node\* child) | Void | 取消对某个子节点的更新 | | queueNeedUpdate(Node\* n) | Void | 将节点推入待更新的队列 | |  |  |  |  1. 总结：   本次实验主要分析了Node类的源码如何实现，例如各个函数内部是怎么组织代码的。在这过程中令我印象最深的就是2个：   1. Node类的成员变量中，有很多的bool值变量，专门用来判断节点是否需要更新、变换矩阵是否需要更新、父节点是否需要更新这些操作，大大减少了不必要的计算。 2. 为了防止整个场景图在更新，导致里面包含的Node节点无法更新这种情况，Node用了一个static的队列容器，将遇到这种情况时的Node节点，暂且放置于容器内，等到后续在进行操作。   同时，中途遇到了很多有趣的问题，比如：   1. 为什么本身节点就是相对于父节点的位置、方向，还要设置类似派生方向、派生位置这种的变量，感觉多此一举。后来，在老师的讲解下，理解了这是一种保护措施，由于有些节点可能是没有父节点的，如有些节点可能会从父节点中分离出去，变成孤立节点，使用这种方式能很好的处理这种问题。   最后，下次实验计划在老师的建议下：“不要细究于某一个类代码是如何实现，要抓大放小，先分析各个关联类之间的函数是如何相互调用的”准备第三次实验对模块的时序图进行绘制。  要求：  1. 尽可能用图、表等设计工具描述代码，包括类关系图、时序图、状态转换图、流程图、功能模块图等；  2. 描述类功能时，不要简单列出类的成员函数和解释，要按功能类别分析和描述；  3. 最好能解释系统这样设计的原因，如：为什么这样设计，为什么用这样的设计模式等。  总之，想象你是开发团队的一份子，你的报告是要让其他开发成员看懂你的设计思想，而不是读懂你的代码。 | | |