三维游戏引擎的研究与设计①

高 兴, 郑 智, 全 宇

(中国医科大学附属盛京医院 计算机中心, 沈阳 110004)

摘 要: 展示了一款基于设计模式的采用UML设计的三维游戏引擎, 该引擎使得开发者更加容易的制作3D游戏. 借助使用最频繁的一些设计模式和优良的面向对象设计原则来保持从高度抽象的观点进行设计. 设计目的是具备通用性、可复用性以及可扩展性的高性能游戏引擎, 分析了其架构, 介绍了引擎结构的设计思路, 并分析了各种模式的使用所带来的效果.

关键词: 三维仿真; 三维引擎; 设计模式; OpenGL; 解耦合

Research and Design of 3D Game Engine

GAO Xing, ZHENG Zhi, QUAN Yu

(Shengjing Hospital of China Medical University, Shenyang 100004, China)

Abstract: This paper presents a 3D game engine based on design patterns, which is designed by using UML, easier for developers to produce 3D games. With the use of the most frequent some design patterns and good object-oriented design principles to keep from the point of view of highly abstract design. The architecture of the high performance engine which is designed to be in common use, reusable and extensible is analyzed, the main concept of the structure of engine is introduced and the benefits of the use of each pattern are also analyzed.

Key words: three dimensional simulation; three dimensional engine; design patterns; OpenGL; decoupled

1 引言

游戏引擎最早出现在上个世纪90年代, 它并非和游戏诞生在同一时刻, 而是当游戏开发发展到一定程度之后才产生的. 游戏引擎比较准确定义是: “游戏引擎是电子游戏或者其它交互式图像应用程序的核心软件组织, 它提供游戏运行的底层技术, 简化了游戏开发过程, 支持多种硬件平台和操作系统, 包括游戏主机和运行游戏的桌面系统. 它主要包括以下功能模块: 三维渲染引擎、物理引擎碰撞检测、声音、人工智能、网络、内存管理、线程、场景组织”.

从 20 世纪 90 年代初开始, 欧美等发达国家就开始大力发展游戏引擎, 目前在研发水平上居世界领先的著名游戏引擎例如 Quake III、Unreal 等均出自欧美的游戏公司. 国内只有完美时空、网易等少数几家公司具有游戏引擎的研发能力, 而且以自用为主. 国内高校在游戏引擎领域的研究较为薄弱, 尚处于起步探索的阶段, 如浙江大学的 CAP 小型三维游戏引擎.

如今的游戏引擎应该更倾向于数据驱动设计, 更专注于游戏内容的创造上[1]. 游戏引擎允许迅速整合大量的内容, 允许艺术家和游戏设计者站在足够高的层次上去考虑问题而不用考虑程序员去怎么完成细节. 开发人员常常需要花费大量的时间用于游戏应用程序中对象的交互和管理. 更糟糕的是,程序员并不总是能够重用现有的软件而是花费大量的时间制造“重复的轮子”. 他们花费大部分的时间用于创造针对特定游戏的接口, 而不是直接设计一个通用的接口.

本文设计一款集通用性、可复用性以及可扩展性于一体的用来减轻上述问题的游戏引擎 GameCore(以下简称 GC). GC 不是一个简单的类库, 它是一个具有扩展性的架构, 用户可以在该架构的基础上, 增加自己的内容达到自己的目的, 由此用户可以把注意力放在游戏功能的实现上, 从而使用户更快的建立自己的

① 收稿时间:2012-12-07;收到修改稿时间:2013-05-06

游戏. GC 架构充分考虑到融入设计模式, 设计模式在软件设计中起到记录经验的作用,是经过提炼的出色的设计方法,对于很多情况下碰到的面向对象程序开发中的常见问题, 它都是合理并可复用的解决方法. 它们所描述的除了问题以外,还有久经考验的解法以及变化形式. 通常, 运用设计模式的系统要比没有运用模式的系统更具健壮性,在可复用性和扩展性上对系统的提升尤其显著[2-4].

2 三维游戏的元素

一个三维游戏本质上是一个持续不断的循环, 它执行游戏逻辑并在屏幕上绘制图像——通常以 30～60 帧/秒或者更高的速度进行绘制. 这类似与电影的放映方式, 与其不同的是可以控制游戏的情节的发展. 图 1 是一个简化的游戏循环, 接下来对该图进行介绍.

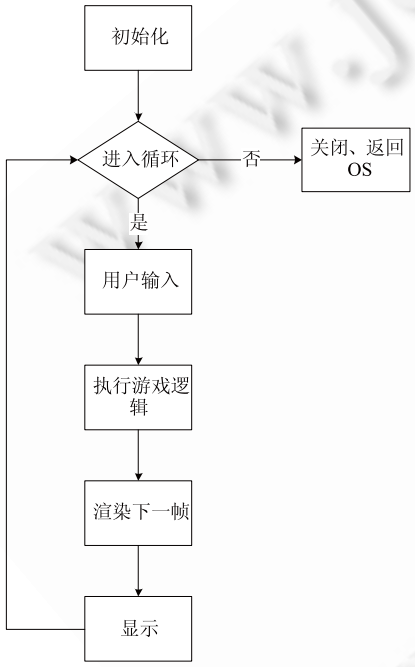

图 1 三维游戏循环流程图

初始化部分和一般的程序类似, 都要进行内存的分配、资源获取、从外存中加载数据等. 三维游戏一般读取配置好的场景文件, 在游戏中也称为地图, 这是个对场景中各种资源进行配置的文件, 包含了各种模型的三维位置信息和模型信息等.

游戏的循环部分, 这部分是主游戏循环, 用户在这里不断的执行动作和游戏逻辑进行交互, 直到用户退出游戏.

用户输入部分是用户和游戏用鼠标、键盘等输入设备进行交互的部分, 它接收用户的输入信息, 交给游戏的逻辑使用.

执行游戏逻辑部分包含了游戏代码的主体部分, 游戏的主题功能聚集在此处, 其中可以包含 AI.

渲染下一帧其实就是把逻辑执行的结果生成到下一个游戏的动画帧, 这部分渲染过程包含着三维图形的绘制, 往往借助于第三方渲染引擎来完成, 将一个 3D 图形按照不同的转换方式渲染成不同的多边形. 例如基于 OpenGL 或者 Direct3D 的渲染引擎中, 大部分是由硬件来承担的.

显示部分是把游戏的画面比较流畅的显示在用户眼前, 保持游戏的帧频是很重要一件事, 一般来事 30 帧/秒是可接受的最小帧频, 60 帧/秒是比较理想的帧频, 这根人的大脑可接受的信息是相关的.

3 三维引擎架构的设计

基于以上三维游戏引擎构成的基本要素, 我们了解到一个游戏引擎应该包括: 场景的渲染、用户的交互、游戏功能逻辑等等[5,6]. 更重要的是把各个方面有机的结合在一起. 而一款游戏引擎首先要提供满足以上基本要素, 首先要比较方便的用来开发一款游戏. 接下来将介绍三维游戏引擎的核心架构 GC, 它同时也是游戏引擎的主要逻辑, 它担负起管理消息、网络传输、场景角色模型及游戏逻辑的组织, 加载场景等工作. 它可以帮助开发者又快又好的建立一款新的游戏.

3.1 GameCore 内容

GC 是这个架构的核心, 它是一个有力的组织者, 把系统的一切组织在一起. 如图 2 所示, 它包括了三部分重要的内容:

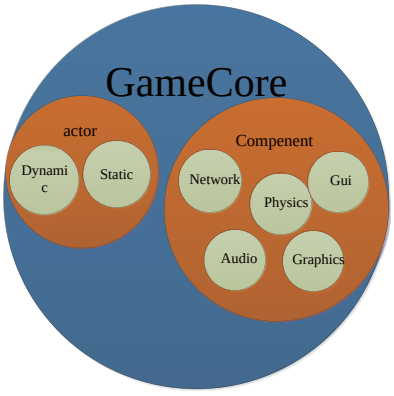

图 2 GameCore 组成结构图

(1) 角色

角色是游戏场景中我们所看见的物体, 主要分外两种, 静态的和动态的. 静态的就是一些没有任何交互的物体,譬如建筑物、路面等; 动态的就是有交互的物体, 譬如人、动物等. 其中动态角色可以接收部分的消息.

(2) 组件

组件是比较重要的功能模块, GC 的基础层, 包括图形渲染, 物理引擎, 声音引擎, 网络, GUI 等, 其中很多模块可以直接使用第三方提供的成熟软件包和中间件. 开发者建立游戏逻辑也是对组件的基础功能进行扩展. 其中组件可以接收所有的消息.

(3) 消息

游戏引擎的整体运转是依靠消息驱动的, 是各个模块进行交互的媒介, 消息把系统中各个模块联系在一起. 用户在开发游戏的时候, 游戏逻辑之间的交互也是通过消息进行的.

3.2 GameCore 功能设计

GC 把游戏引擎中的核心内容组织到了一起, 为了使游戏有效的运转起来, 它具有如下核心的功能:

(1) 管理角色

GC 维护存储了场景中的所有的角色, 包括静态角色和动态角色. 维护角色的状态, 对角色进行管理. 用户创建一个新的角色, 必须在 GC 里进行注册. GC 可决定对角色赋予接收消息的权限, 在内部对发送给角色的消息进行过滤, 角色只接收注册过的消息, 并可对消息进行处理.

(2) 管理组件

GC 维护了一个组件列表, 并可赋予组件不同的优先级. 用户在创建了一个新的组件后, 必须在GC中注册, 申请一定的优先级, GC 将根据组件优先级的顺序把 GC 中的消息分发给所有的组件. 组件具有接收一切消息的特性, GC 对消息不加处理, 分发给所有的组件.

(3) 消息路由

系统中 GC 是唯一具有发送消息的功能, 消息在系统中是一个事件, 它可以由组件或者角色形成, 在每一帧, 角色或者组件把要发送的消息放在 GC 的消息队列中. 在下一帧的逻辑处理阶段, GC 把消息分别分发给角色和组件. 消息结构如图 3 所示:

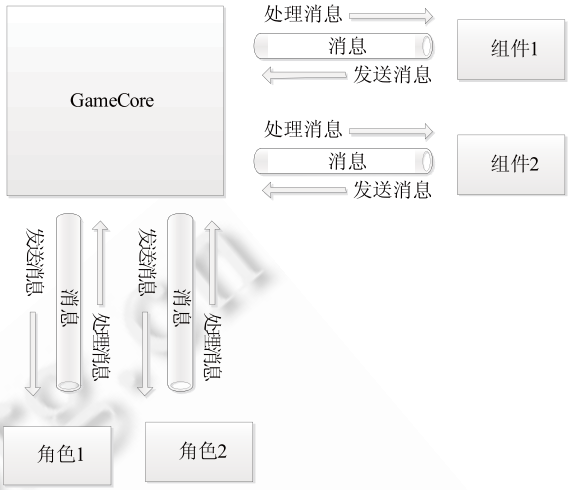

图 3 GameCore 消息机制图

4 组件封装

GC 中 Component 可以接收游戏中所有的消息. 所以我们把底层所调用的其他引擎封装到 Component 中. 由于图形引擎是渲染引擎是游戏引擎中最复杂的部件, 也是最为关键的部分, 在此重点介绍如何封转一个面向对象的图形渲染引擎, 并成功运用到组件中. 参见图 4.

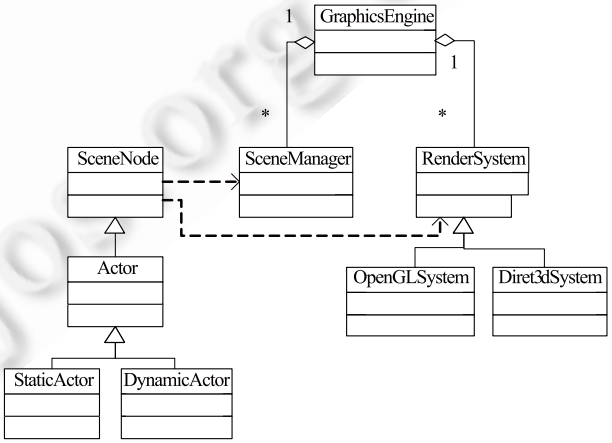

图 4 图形引擎架构图

图形渲染的作用是高效地显示清晰的画面, 它的强大与否直接决定着最终的输出质量, 往往要求在时间和处理资源有限的情况下, 满足玩家对图形质量的要求, 开发时通常把 DirectX 和 OpenGL 作为底层的 API. 传统的这种直接的调用底层 API 的模式越来越暴露出开发复杂性大、周期性强长、维护困难的的缺陷. 因此我们设计一套面向对象的高度封转底层细节

的图形引擎, 隐藏底层系统库 Diretx 和 OpenGL 的所有细节, 并支持多种高级特性, 提供一个基于现实世界对象和其他直观类的接口, 使开发者不再困恼于底层实现环节, 直接调用高层 API 就可以实现三维模型的渲染.

4.1 GraphicsEngine

GraphicsEngine 直接作为图形 Component 的一个引用类存在, Component 接收到的消息直接传给 GrahicsEngine. GraphicsEngine 的实例是整个渲染系统的入口点. 一旦启动图形引擎,就应该确保它是第一个生成,最后一个被销毁的对象. 通过调用 Graphics Engine 实例,可以配置整个系统,并且可以得到其它核心类(如 RenderSystem 类,SceneManager 类)的实例的指针. 它还有一个重要职能就是,提供执行渲染的方法,当设置好场景及其中的元素后,则调用该方法让应用程序开始连续地进行渲染.

显然,GraphicsEngine 是整个图形系统中独一无二的部件. 如果采用常规的实例化对象的手法,即 New 直接实例化对象,这样的全局变量会遇上两个问题:第一,直接 New 的方式具有较高的耦合性,缺乏相应的灵活度;第二,类实例化的主动权被客户程序掌握,而不为类自身所控制. 为了避免这两个问题,我们采用 Singleton 模式. Singleton 模式的意图是,保证类仅有一个实例对象, 并提供对它的全局访问点. 鉴于引擎中还有其他地方用的单例模式, 故引入模板技术, 实现模板单例 Singleton<T>,作为所有可采用单例模式的类的基类.

4.2 渲染系统

由于屏蔽了底层 OpenGL 和 Direct3d 或者其他渲染引擎的差异, 可以在 RenderSystem 中定义一些统一的接口, 客户程序可以在不做任何更改下选择不同的底层渲染引擎进行渲染场景. 接口形式可以用设计模式中的抽象工厂模式来实现, 通过 Abstract Factory 模式创建一系列针对特定底层渲染引擎的接口.

4.3 场景管理

游戏场景中有很多的 SceneNode, 如何对这些 SceneNode 进行管理, 进行最合理有效的渲染, 这就需要 SceneManger 来实现. SceneManger 针对不同的场景类型将采用不同的算法来决定哪些 SceneNode 送给 RenderSystem 实例进行渲染. 譬如常见, 基于 BSP(Binary Space Partition)算法实现室内场景的管理, 基于 Octree 实现室外场景的管理, 当然除此之外, 还可以使用其他的一些算法. 这里我们采用 Factory 模式, 设置 GraphicsEngine::CreateScene()方法, 传递场景类型 (BSP_TYPE, OCTREE_TYPE), 返回相应的 SceneManger 类的指针, 这个 ScenManger 的 SceneNode, 就调用相应的算法进行渲染.

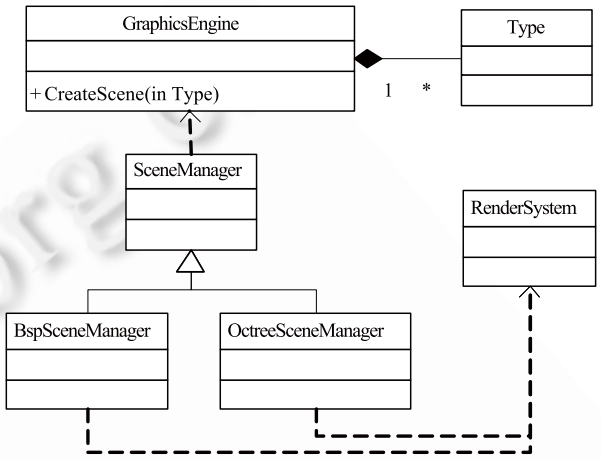

图 5　场景管理

4.4 其他功能组件

通过这样一套面向对象的高度封转底层细节的图形引擎, 把 GraphicsEngine 直接作为一个图形 Component 的一个引用, 通过 GraphicsEngine 来渲染游戏场景中的角色. 其他底层引擎都可以通过这样的方式直接封装在 Component 中, 例如网络引擎可以直接封装开源引擎 GNE, 音效引擎可以封装 OpenAL 库, 物理引擎可以封装 bullet, 如果添加新的功能, 可以以组件的形式封装到 GameCore 中, 统一了整个接口, 使变化含在其中.

5　仿真实验

本节利用设计的三维游戏引擎建立一个简单的场景, 鼠标选中车辆或者小人行走. 如图 6 时序图所示.

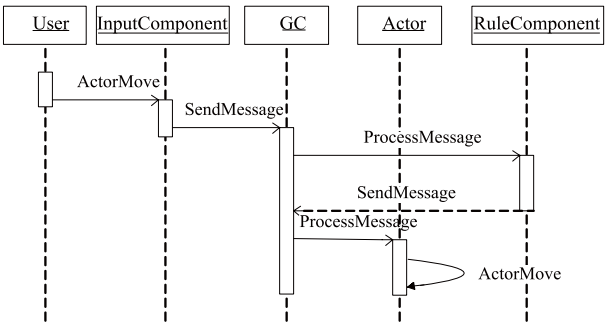

图 6　角色运动时序图

用户用键盘点击场景中角色，则角色处于选中状态，然后用鼠标点击目标位置，则输入组件(InputCompont)向逻辑组件(RuleComponent)发送的消息首先存储在 GC 中，GC 在下一帧逻辑处理阶段把该消息发送给逻辑组件(RuleComponent)，逻辑组件经过逻辑计算，计算出了角色下一帧的位置，并把消息传给 GC，GC 在下一帧传给角色(Actor)，则对角色下一帧的位置进行渲染，循环知道角色到达目标地点位置，渲染画面如图 7 所示. 该仿真验证程序的为 60 帧/秒，能够很好的满足人的视觉感受，画面比较流畅.

图 7 角色运动图

6 结语

本文运用设计模式设计了一款具备通用性、灵活性以及可扩展性的面向对象三维游戏引擎. 其主要特点是:体系结构简洁清晰，面向对象技术将引擎划分为多个定义清晰而功能相对单一的子系统，并大大降低各个功能模块的耦合度. 通过合理使用设计模式,使得系统易于修改、维护、升级以及复用，部分模式在一定程度上提高了系统的性能. 目前，该引擎被用于开发空战游戏系统. 该引擎还将在执行效率方面作进一步提升，以适应不断发展而提出的新的要求.

参考文献

1 孙咏.基于 OCP 软件应用架构的设计与实现[博士学位论文].北京:中国科学院研究生院,2009.

2 阎宏.JAVA 与模式.北京:电子工业出版社,2002.41–44.

3 Gamma E.设计模式-可复用面向对象软件基础(双语版).北京:机械工业出版社,2009.1–21.

4 Shalloway A, Trott JR. Design Patterns Explained.第 2 版.北京:人民邮电出版社,2010.53–188.

5 张秀山.虚拟现实技术及编程技巧.长沙:国防科技大学出版社,1999.20–90.

6 Berenbrink P, Brinkman A, Scheduler C. SIMLAB-A Simulation Environment for Storage Area Networks. IEEE, 2001, 27(4):227–234.

(上接第 8 页)

detection. Proc. of the 5th ASIACCS, 2010.

7 Hu X, Knysz M, Shin KG. Measurement and analysis of global IP-usage patterns of Fast-Flux botnets. IEEE INFOCOM technical program, 2011.

8 Caglayan A, Toothaker M, Drapaeau D, Burke D, Eaton G. Behavioral analysis of Fast Flux Service Network. Proc. of the 5th CSIIRW, 2009.

9 汪洋.Fast-Flux 服务网络检测方法研究[硕士学位论文].武汉:华中科技大学,2009.

10 Hawkinson J. RFC1930: Guidelines for creation, selection, and registration of an Autonomous System (AS), March 1996. http://www.ietf.org/rfc/rfc1930.txt

11 Knysz M, Hu X, Shin KG. Good guys vs.bot guise:mimicry attacks against Fast-Flux detection system.IEEE INFOCOM technical program, 2011.

12 Mockapetris P. RFC 1035: Domain name-implementation and specification, November 1987. http://www.ietf.org/rfc/rfc1035. txt.

13 Perdisci R, Corona I, Dagon D, Lee WK. Detecting malicious flux service networks through passive analysis of recursive DNS traces. Proc. of ACSAC'09, 2009.

14 Mockapetris P. RFC 1034: Domain name-concepts and facilities, November 1987. http://www.ietf.org/rfc/ rfc1035.txt.

15 Xiao JZ, Li X. A Research of the Partition Clustering Algorithm. International Symposium on Intelligence Information Processing and Trusted Computing 2010.