电子科技大学
UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER DISSERTATION

论文题目 游戏仿真层架构研究与设计

学科专业 信息安全
指导教师 陈雷霆 教授
作者姓名 胡浩源
班 学 号 200420605005

分类号 TP391.1

UDC

学 位 论 文

游戏仿真层架构研究与设计

（题名和副题名）

胡浩源

（作者姓名）

指导教师姓名 陈雷霆 教授

电子科技大学 成都

（职务、职称、学位、单位名称及地址）

申请学位级别 硕士 专业名称 信息安全

论文提交日期 2006.12 论文答辩日期 2007.1

学位授予单位和日期 电子科技大学

答辩委员会主席

评阅人

2006 年 12 月 5 日

注 1 注明《国际十进分类法 UDC》的类号

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：胡洪涛　日期：2006 年 12 月 4 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技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、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：胡洪涛　导师签名：

日期：2006 年 12 月 4 日

摘要

游戏中的仿真层是一个独立于图形系统的逻辑层次，代表了游戏内容驱动游戏图形的层次。随着游戏复杂程度提高，游戏对仿真层的设计也提出了更高的要求。本文研究大规模网络游戏引擎中仿真层的设计，通过游戏仿真层架构的研究，进一步提高游戏开发效率和游戏开发的灵活性。

游戏仿真层涉及多个学科内容，本文先从人文学科的角度入手，分析虚拟世界的本质，划分出虚拟世界的结构，包括对象、消息、规则、时间等。将此理念融入游戏仿真层架构的传统设计，并结合创新的计算机技术，得到最终可实现的版本。其中，对象的实现引入了包括扩展的有限状态机、数据集合等若干创新技术；消息采用了基于数据集合的方式，较之传统的消息机制更为便于使用和设计；规则在对象行为中得以体现；以及更加灵活的时间调度机制等。此外为了便于二次开发，还实现了若干辅助模块，如应用程序框架、用户输入模型等。至此游戏仿真架构已相当完整，能够有效地虚拟现实世界中事物的组织关系与行为模式，并使游戏仿真层的开发效率有极大提高。

基于虚拟世界的特殊性，文中特别提出并证明了可回溯的状态机这一创新点。此创新说明了虚拟世界中的状态机模型可以基于消息流，在时间中任意的向前推进或者向后回溯。该技术可以为游戏设计带来全新的理念，并且能够解决网络游戏中同步的难题。另一创新点——数据集合，是可以在运行时根据需要动态变化的数据结构，内嵌的类型检查机制确保了其数据操作的安全性。数据集合可以广泛的应用在仿真层架构的各个模块，并为脚本化奠定了坚实基础。此外，数据集合在其他需要动态数据结构的场合都可以广泛地应用。

最后，文中的内容在两个 863 项目中得以很好地实现。

关键词：游戏仿真层，虚拟世界的本质，可回溯的状态机，消息机制

2

Abstract

Game Simulation Layer which indicates the game context driver mechanism is an independent logic module from Graphics Layer. As the complexity of game logic growing, Game Simulation Layer requires a better design. The document researched the design of Game Simulation Layer in massive multiplayer online game developing platform for a concern of efficiency and flexibility.

Game Simulation Layer is known as cross-subject knowledge. The virtual world has been researched in some humanities ways, such as philosophy.

The major objective of this thesis is the design and implementation of Game Simulation Layer. The principles of the virtual world have been found, including Objects, Messages, Rules and Time. This idea is integrated with traditional Game Simulation design, importing new programming technologies. And finally, an implement able architecture is designed. Objects are found on new method such as Data Set and FSM. Message mechanism based on Data Set is more flexible than before. Rules were implemented in objects behaviors and also a well-designed time scheduler. As for second development, supporting tools are designed too, such as Application Framework, User Input Model.

As the game industry growing, computer technology in game industry shows a brand new scene. Being a virtual-real part, the game simulation technology seems more and more important in games. A goal to make game developing more efficient and flexible is our concern. This architecture is mostly perfect. It can simulate the real world as objects, relationships and behaviors. And it brings a great effort to game logic development.

As an addition, Traceable FSM is a totally new concept in game simulations. It proves that with message flows FSM could go forward or backward randomly without any errors. It brings some amazing features for the future game design. And it solved a difficult problem in network synchronization. Another creation is the Data Set, which is a dynamic data structure can be changed at will in runtime. An embedded data type checking mechanism ensures the safety of data operations. It has a wide usage in Game

Simulation Layer. And it builds a stable base for scripting technology. As well, it could be widely used in any situations which needs dynamic data structure.

At last, all the contexts in this document are implemented in the two 863 Nation High-tech Project.

Key words: Game Simulation, Essence of Virtual World, Traceable FSM, Message Mechanism

目录

第一章 绪论

1.1 课题背景和研究意义

国家科技部863计划于2006年发布了“中国数字媒体技术发展白皮书”，对数字媒体技术发展现状进行了深入分析，文中描述到：从目前我国数字媒体技术的发展状况来看，内容制作效率低、内容缺乏创新、标准滞后、网络整合困难、终端不足、市场不成熟、价格缺乏制度依据、核心技术有待继续研究、产业链尚待进一步完善、商业模式有待探索以及体制问题等诸多因素成为制约我国数字媒体产业的健康发展主要瓶颈。这些瓶颈问题严重影响了我国数字媒体产业和技术的发展，所有这一切都有待在未来五年的产业与科技规划中得到改善和彻底解决。数字媒体技术产业具有巨大的市场空间与成长机会。中国作为一个制造业大国，应在该领域有所建树，并不断通过扶持和鼓励、引导市场对相关项目和产品的研发，提升我国数字媒体产业的竞争力，进而确立我国在这一产业中的国际地位。在这样的政策指导下，国家大力推动了数字媒体相关的研究项目。本文描述的系统就是在以下两个863项目中完成的：

- 国家高技术研究发展计划(863 计划)《网络游戏公共技术平台关键技术研究》(No.2005AA114080)，该项目的主要任务为进行大型实时网络三维游戏引擎设计，整合网络，图形，物理，AI和游戏逻辑，音效等功能模块，并开发以场景编辑器为主的数字内容创作工具。最终实现了多通道大视景的大型室外场景实时模拟。
- 国家高技术研究发展计划(863 计划)《数字媒体公共技术平台研发》(No.2005AA114030)，该项目的主要任务为研究数字媒体创作中的若干关键技术。实现一个统一的数字媒体技术开发公共平台。主要涉及实时计算机图形学高级算法、网络以及大规模软件工程。

在这两个项目中，我基于已有的相关技术基础上，通过结合新的研究方法，完成了本文所描述的游戏仿真层架构，并与其他相关模块进行了良好的整合。

1.2 关于游戏

曾几何时，我们钟情于捉迷藏等游戏，它们给我们的童年带来了无数乐趣。随着年龄的增长，我们已不再专注于这些童稚的游戏。然而对于决大多数人而言，生活中总是不能缺少形形色色的乐趣，谁愿意过着了无生趣的生活呢？世界在进步，随着计算机技术的逐步发展，电子游戏作为一种新兴的获得“乐趣”的形式从无到有，从简单到复杂，从只在小范围内流传到被社会大部分人所接受，到如今已经成为很多人生活中不可或缺的一环。并为丰富人类的生活不断的贡献着自己的力量。

游戏的组成是复杂的，它包括了最前沿计算机科技的应用，如图形学、人工智能、网络技术等；它也继承了古老人类文明的优秀内容——文学、美术、音乐、哲学等多种内容。某种程度上来说，游戏已经成为继人类所创造的八大艺术之后，第九种伟大的艺术。不仅仅是艺术，而且是先进的科技与优秀的艺术的集合体。不可否认的，和所有的艺术形式一样，游戏也存在良莠不齐的情况，然而这并不阻碍它作为或者成为一门伟大的艺术。[31]

作为承载先进技术的艺术形式，我们可以说，艺术只是它的血肉，而计算机科技才是它的头脑和骨骼，控制它的行为，支撑它的一举一动。构成游戏的各类计算机技术，分别掌管游戏的各个方面。比如图形学，负责将游戏转变为画面展现在人们眼前，网络技术负责将千里之外的人通过计算机网络联系在一起，而本文的主角，游戏仿真，通常也称作游戏逻辑，则负责将游戏所要表现的世界在计算机中虚拟出来，这也是所谓“仿真”的意义之所在。[3]

1.3 国内外研究现状

这一小节中，我们对本文使用的主要技术的国内发展作一个全面的综述。主要介绍关于游戏逻辑相关技术的发展和研究现状。

1.3.1 游戏开发技术的历史

追溯到游戏开发的早期，计算机不论从种类还是从能力上都非常有限。这些计算机的 8 位处理器很慢，运行速度只有几兆赫兹，图形处理功能也很有限，内存在 48KB 到 64KB 之间。为了绕开平台的硬件限制，所有的游戏都是用汇编语言来编程的，这样可以满足执行速度和存储空间的要求。那时还没有好的 C 编译器，已有的那些 C 编译器无法为游戏程序生成短小紧凑的的代码。这意味着每一

款游戏都需要彻底重新编写代码，甚至在平台移植时。

然而这些限制也带来了一些好的影响。因为内存、图形、声音硬件都是受到限制的，由此更多的精力被放在了游戏的可玩性方面，这样使游戏玩起来更加良好。即是说，开发的重心在游戏逻辑部分，各种如今仍然活跃在开发舞台上的技术在此时已经被广泛的应用，比如有限状态机技术。这为后来的技术的发展奠定了良好的基础，在后来的技术发展中，有限状态机技术经过多次扩展，但其核心的思想却贯穿始终。

那么我们现在的情况如何呢？在今天计算机上完成开发的实际情况，一句话，更加容易，因为由编译器和操作系统自动完成的工作量大大增加了。通过开发一些函数库和结构就可以访问硬件接口，这样游戏开发人员就可以把更多的精力集中在游戏的逻辑设计上，这是比较有趣的一部分工作。所有烦琐的工作，包括机器初始化、硬件设置等，几乎全由库函数来实现，开发人员可以更多地关注眼前的任务，而不必和成百上千的硬件设置打交道。[2]

1.3.2 研究虚拟世界的方法

时下比较流行跨学科的研究手段，各个学科都通过融入其他学科来获得新的活力。数学，作为理工科中抽象程度最高，最通用的研究手段和工具，早已渗透于人文学科当中，比如说经济学。但人文学科中的研究方法在理工科中的应用却是少之又少。在方法上，长期以来关于游戏逻辑的研究都是基于理工科方法的。然而作为对现实世界的抽象，我们需要引入一些研究现实世界抽象的方法，比如说哲学。

实际上数学和哲学都是对于现实世界的抽象（事实上我认为哲学的抽象程度较之数学要高）。数学的优点在于其形式化的表达方式，更适合于理工学科。而哲学自身充满着种种矛盾，不可能具有形式化的表达方式。然而作为一类指导性质的学问，这并不影响把它作为一种研究方法引入到理工科学研究中来。

除了跨学科的初衷之外，软件架构本身也是非常具有人文学科特性的。它并不着重于过程化的算法，算法在软件架构看来只是组成整体的元素；也非着重于基于统计的预测，更多的是人的经验，比如说经典的“设计模式”。软件架构的核心在于组成软件的各个“元素”之间的联系，这并不适合运用数学的方法来研究。可以假设一下，如果我们把这些联系用数学方式来表达的话，那么将是一个具有成百上千变量的多元高次联立方程组，这是远远超越人的阅读和理解能力的，但是我们却可以用简单的哲学原理来描述这样的联系。

当然，对于软件架构这种工程性极强的学科，研究原理的目的还是为了实现。或许不能够给出具体的数学算法，但是如果能够得到具有广泛意义的方法论，那对于实现也是相当有益的。

1.3.3 复用能力

复用能力的概念是，制作的软件模块可以运用在多个项目里。可能有的人会认为，使用共享模块会让不同开发者生产出来的游戏产品看上去大同小异。事实上，这是“单片集成电路式开发”与“分立元器件式开发”的区别。“单片集成电路式开发”就是以大模块的方式开发程序。这种大模块包括一些分立的小块，但这些小块之间相互交织，具有不可分离的关系。系统的每一部分之间相互依赖，各部分是依靠内部消息（而不是外部接口）来实现其功能。这种对内部消息的依赖性标志着这些部分是不变的。各部分间彼此关联，必然使其错综复杂。

与“单片集成电路式开发”模式对应的的“分立元器件式开发”模式，我们可以用 COM 为例。这种开发模式被微软公司力推了很长一段时间。虽然这种模式有点笨拙，而且不能完全支持所有的硬件平台，但微软公司认为，不能因为这些缺点就认为这种模式不好。

COM 的主要优点是容易实现版本转换。一个 COM 对象可以为用户提供多个对外输出接口。COM 对象至少有一个输出接口，IUnknown。这个接口是所有 COM 接口的父接口，它只支持 3 种主要运算：引用计数加一，引用计数减一和检索接口。单独使用这种父接口时其用处不大，因而一个稍微复杂点的 COM 对象有多个输出接口。这些额外的派生接口是通过如下方法获得的：首先得到一个 IUnknown 指针，然后由 Iunknown 提供一个指向所用接口的指针（使用 QueryInterface 方法）。关于 COM 对象的一个金科玉律是：一旦发布了一个接口，它就无法改变。如果需要为一个正在被其他程序使用的接口添加额外的功能时，就必须为该对象增加新的接口。这个新接口能够支持所有旧的功能，也能够支持想要增加的新的功能。这样旧程序和新程序都能够有效的工作。[8]

虽然在最后的实现中并没有采用 COM 技术。但实际上，复用的思想对我们的系统设计影响很大。可复用性也是将来游戏开发中极度重要的原则。

1.3.4 脚本技术

脚本技术实际上是形式化描述方法的实现。

通常脚本包括两类，一类是描述数据，另一类是描述过程。其中描述数据的代表是大家都很熟悉的 XML。现在在游戏开发主要应用的描述过程的脚本语言有 Python 与 Lua 等。其中 Lua 由于其轻量级与灵活的特点，更多被应用在游戏逻辑的开发中。Lua 可以集成于 C/C++中，它并不作 C/++已经擅长的工作，相反它是 C/C++的一个补充，提供诸如：硬件抽象、动态数据结构、无冗余、易于调试和测试等等。除了作为 C/C++的扩展，Lua 也是一种“胶水”语言。它能够将各种不同类型、层次的组件组织在一起。[17][29]

以下是 Lua 脚本语言的特性列表：

- 可扩展性：Lua 从设计理念上就是为了扩展而存在的。通过 Lua 可以方便地与 C/C++，Fortran，Java，Smalltalk，Ada 甚至其他脚本语言交互。
- 简单：Lua 是一个小而简单的脚本语言，只包含很少的一些概念。这个特性使其非常的易于学习和传播，同时也适合资源有限的环境。
- 移植性：Lua 几乎支持所有的操作系统，除了 Windows，Linux，Unix 之外，还有 NextStep, OS/2, PlayStation II (Sony), Mac OS-9 and OS X, BeOS, MS-DOS, IBM mainframes, EPOC, PalmOS, MCF5206eLITE Evaluation Board, RISC OS 等等。这是因为 Lua 是基于 ANSI C 的。

我们的目的是效法已有的成熟脚本技术，实现一个自主的脚本引擎。然而由于实现一个完整的脚本引擎是相当复杂的工程，因此本文中所叙述的游戏逻辑框架并没有脚本引擎的实现。但是在设计时，充分考虑到为将来扩展脚本引擎做好了各方面的铺垫。因此脚本相关依然是本文主要的内容之一。

1.4 论文的主要工作与创新点

本文对游戏逻辑架构进行了深入的研究与分析，内容涉及：计算机虚拟世界的理论、有限状态机理论的证明与应用、大规模软件工程中的框架设计、与其他游戏中相关模块的整合。此外还包括与游戏逻辑相关的应用程序框架、C++中关于模板技术的研究。并且在研究的基础上，设计出了一个适合大规模多人在线网络游戏开发平台中仿真层的架构，在实际的 863 项目中实现，并且以若干实际开发应用例子来验证了设计的正确性。

当然目前的设计与理想中的还有差距（比如脚本引擎），这会在将来的工作中

逐步的增加、完善。

总的说来本文的主要研究成果有下面这些：

- 实现了一个面向对象的、可扩展的、为脚本化准备充分的游戏仿真层框架。并融合到整个 863 项目《大型多人在线网络游戏开发平台》的引擎之中。
- 创立了一套计算机虚拟世界架构的理论。
- 为现今常见的游戏逻辑中的有限状态机理论增添了新的特性。

而本文的主要创新点在于：

- 引入了跨学科横向研究的方法。
- 证明了可回溯的有限状态机理论并引入到实际的应用中。
- 提出了一种基于半动态数据结构的脚本思想。

1.5 论文的章节安排

全文共分六章，对游戏仿真层技术进行详细的分析与研究，章节安排如下：

第一章是全文的绪论。主要介绍本文的课题背景，对国内外研究现状进行综述，并指出论文的主要工作和创新点。

第二章阐述了计算机中虚拟世界的性质，同时基于这些性质给出了一个概念上的设计方案。

第三章详细地介绍了仿真层设计中所涉及到的各方面技术细节。

第四章专门用来介绍具有创新意义的部分，包括了其证明过程和实际应用中所需要注意的一些问题。

第五章对全文进行了总结并对之后的工作进行了展望。

第二章 计算机中的虚拟世界

2.1 虚拟世界的本质

2.1.1 虚拟世界的时空观

计算机科学创造出的独立世界，就好像科幻电影《Matrix》里所描述的那样，未来的计算机科学可以创造一个纯精神的世界。在那个世界里面，可以说人只剩下了思想和感觉，而物质只是一堆感觉材料。整个世界的物质性完全失去了；而人的躯体则完全在另一个世界里，那已经不能叫作躯体了，只能称作“有机物的组合”。这部电影给我们展现了计算机科学发展到了一种极端状态的画面。而在这篇文中，我并不打算由电影来讲些什么，那只是科幻而已，即便它有可能实现，那恐怕也是未来多少代的事情了。在这里我只是针对现代计算机科学的内容来进行一些思考和探讨。

随便哪一本计算机入门书上都会讲，一台计算机是由硬件和软件所构成的。硬件作为物质基础，是必要，但却不是核心的。正有如碳氢氧可以组成简单的无机物，也是构成人这样的复杂的有思想的有机体的基本元素。而软件则是计算机的抽象层面。黑格尔说世界的发展经过三个阶段：逻辑、自然、精神。单就从字面意义上来理解的话，我们不妨来简单的套一下这个公式。计算机硬件除了是其物质基础之外，也是计算机逻辑的最基本的载体。硬件的电路提供了最基本的逻辑：01 逻辑，即是与非的逻辑。那么我们再来看计算机软件构成的核心——数学。从逻辑证实主义者的观点来看，数学其实是可以用最基本的是非逻辑推导出来的，因此我们也可以看作计算机软件的核心也是逻辑。那么，抛开硬件在现实世界的物质意义，计算机世界的第一个阶段，或者说第一个层次，已经展现在我们面前了。[14]

那么我们再来看第二个——自然。对于自然，我以为不妨当作实体来理解。不过首先我们来看看计算机世界中的时空观。计算机世界中的时间和现实世界的时间相同的是，都具有无限延展性，时间没有头尾的。这是因为这两者是同步的，计算机时间基于现实时间，但也有不同。现实世界中的时间具有无限可分性，而计算机世界中的时间却是有限可分的。康德曾经说现实世界的时间不具有无限可分性，是由一个一个瞬间组成的，这个瞬间可以看作时间的最小单位。但是后来

有一个数学哲学家证明了时间是无限可分的，反驳了康德。在计算机世界里，终于可以如康德的愿了。只不过在计算机世界里，我们把时间的最小单位叫做“时钟频率”。在每一个最小时间片之内，一切实体都是静止的，只有在时间片交迭的时候才会发生运动和变化。那么实际上计算机世界就是由无数个连续的静止画面组成的幻灯片，对于每一个幻灯片我把它叫做一个“状态”。状态是绝对静止的，所谓“人不可能两次踏进同一条河流”“事物无时无刻不处在运动中”的论断在计算机世界是不成立的。而且由于有状态的存在，我们可以保存某个状态，然后在将来的某个时刻将该状态完全重现，甚至可以让某个状态停留直至永远。有了这样的时间观，就让这个世界有“静止”、“绝对”、“永恒”的存在。[5]

由于有了不连续的时间，机械运动也就有了不连续性。但不能因此证明空间就有不连续性。所谓的空间连续性的概念在数学上可以这样来说，空间中任意两点（不是同一点）之间的距离都不是最短，那么这个空间具有连续性。我们再来看计算机世界的空间，计算机世界里的空间和现实世界的很大的不同，现世世界中的空间是指方位上的意义，而计算机中的所谓的“空间”具有的是存在意义。我们可以简单的把计算机中的存储器看作是它的空间，具有有限延展性和有限可分性两个性质。这些空间是由 01 逻辑构成的，每一个 01 逻辑便是最小的空间单位，01 逻辑的数量决定了空间的大小。可以这样说，在逻辑的基础上，构建了计算机世界的空间。如果我们承认计算机中空间的特殊意义，即存在意义的话，那么我们可以得出这样一个结论：逻辑决定存在。[7]

接下来我们再来证明为什么计算机中的空间是决定其存在意义。所谓存在，是一切实体所具有的共同性质。现实世界中实体可以分为两类——物质和精神，物质依托于空间而存在，精神可以独立于空间而存在。在计算机中，实体都要依托于空间而存在。从计算机编程的角度来看，存储空间中主要有两种东西——数据和代码。数据又分为数据类型和数据内容，犹如现实世界中的共相与殊相。共相论讲，共相是一种独立于时间和空间永恒存在的实体。然而在计算机中，数据类型也是需要占用存储空间的。代码就现实世界而言，相当于是事物变化的原因和过程，或是事物之间的相互联系相互影响，简单的讲就是变化与联系。这两者在现实世界中是非常抽象的东西，然而在计算机内存里也要占用空间。也就是说共相、变化、联系这些现实世界中存在而又抽象的东西，在计算机中都通过硬件逻辑把它们实际化了。变成了实实在在的一种“存在量”——计算机中的存储空间，即我所谓的“空间”。而计算机中现实世界的方位被抽象成了数据而存在。[4][26]

那么我们可以说，在计算机中，逻辑和存在都是可以衡量的，逻辑的量是有限的，而逻辑决定存在，那么存在也是有限量，即逻辑量决定存在量。

由此，我们得到一条准则，计算机所能描述的是一个有限存在量，有限可分的世界，是现实世界的一个子集。它的时间却与现实世界时间大大不同，有限可分，可静止，甚至可以倒退，如果不考虑时间的有限可分性来说，它是现实时间的超集。

2.1.2 虚拟世界的组成

由上文可以看到，计算机世界与现实世界的本质有很大的差别，为了能够更好的虚拟现实世界，我们需要让两个世界在某个层次上接近一致。

在游戏中，大部分情况下我们并不需要把虚拟世界精确到最小可分性上去。更多的高层的概念才是着重考虑的。通常要视游戏设计的粒度而定。比如说赛车的游戏，是把赛车视作一个单独的具有速度、加速度、碰撞体积等属性的独立个体考虑？还是由引擎、动力传导装置、轮胎等若干独立个体的组合考虑？无论如何回答这个问题，有一点可以明确的是，我们着重考虑的是游戏的中一个一个具有独立性的个体。这些个体通常被称作对象，正如同 C++里对象的概念一般，而且这样也更容易由程序语言来实现。

在现实世界中，我们也很容易找到对象这样的概念。一个人、一个班级、一个学院、一所学校、一座城市等等。这些概念在虚拟世界中也能得到很好的体现。我们可以说这些概念是世界本质在更高层次的抽象，无论本质如何，它们最终都可以构成更高层次的概念。

在对象层次上，计算机中的虚拟世界与现实世界得到了某种程度的统一。那么下面来详细地阐述虚拟世界的构成。

2.1.2.1 对象

对象的存在，属性

如空间位置，体积，质量等。

对象的变化，行为

对象属性的变化，如位移，缩放，增减质量等。

对象属性种类的变化，对象由进入某个环境时获得新的属性，或者由离开某个环境时失去某个原有属性。如某学生上学后得到了学分这个属性，或是离校后失去学分属性。

对象的构造，聚合对象

若干对象通过联系聚合在一起后构成的具有完整性的新对象，如树木构成森林。

2.1.2.2 对象间的关系

联系

对象与对象间的相互作用，如 A 对 B 说了一句话。

消息

对象与对象间联系的方式，如 A 给 B 说的话的内容。

2.1.2.3 规则

单体规则

对象行为的一般性原理，如空间位置变化，符合几何原理。

群体规则

对象聚合的一般性原理，如何由单个对象构造为复杂对象，以及聚合对象的新特性的组成原理，如由树木数量产生的森林覆盖度。

时间规则

基于时间的一般性原理。所有对象会随着时间流逝产生变化，比如说孤立对象内部会呈现熵增加的状态（无序化）。

2.2 虚拟世界的设计

2.2.1 逻辑对象

逻辑对象包含了状态机。

一方面是现实世界层次结构的体现；另一方面是抽象状态的复杂数值逻辑的描述。

首先我们要让每一个逻辑对象 Logic Object 都成为一个独一无二的个体，现实中也是如此。我们通过分配独一无二的 ID 来解决，就好像身份证号和学号一般。因此要有一个 ID 生成器 ID Generator，保证生成独一无二的 ID。这个 ID 生成器我们是通过空闲域管理的方式来实现的。然后每生成一个逻辑对象，便对其分配 ID，当逻辑对象销毁时，我们便把这个 ID 重新插入空闲域中。因为是采用域的方式，而不是单个 ID 节点，因此可以保证 ID 生成器所占用的内存最小。

当逻辑对象有了 ID 之后，我们便可以明确的来标识它们了。

下一步就是把逻辑对象纳入树状管理中。本来现实中对象间的关系并非树状的，确切地说，应该是网状的结构。之所以要做树状管理，有三个理由：一是树状结构使用更方便，相关各种算法更成熟；二是网状结构一般都可以通过某种方法转换为树状结构；三是这个结构主要是体现层次或者说从属关系的，旁的非主要的关系我们可以忽略。

那么我们怎么样来设计这个从属关系呢？我认为，下层的逻辑对象应该是上层逻辑对象的一个复杂数据逻辑的抽象。比如说某个学校的一个班，班里有 20 个学生。在考察这个班的成绩的时候，我们可以完全把学生看作一组数据，语文成绩多少，数学成绩多少，英语成绩多少。然后经过一个计算来评价这个班的成绩或者说状态是优、良、中、差。但我们加入学生这个对象，把每个学生的成绩分为优良中差。那么在统计班级的成绩时就简单了许多。也就是说在设计层次结构的时候，核心的方法就是把每一个层次的数值逻辑简化到可以控制的地步，上层对象只需要关心下层对象反映出来的简单状态，而忽略掉下层对象的细节，那么在设计的时候就可以做到分工明确。

因此，在实现中，每一个逻辑对象都包含有一个子对象的容器 Childrens。如何组织并不在框架里涉及，而是交由具体逻辑设计负责。

逻辑对象的数值逻辑，实际上就是逻辑对象的状态逻辑对应的真实数值。比如上述的天气，光照辐射度和气温就是相应的数值逻辑。同样，这也是决策过程

所需要参考的数据。

每一个逻辑对象都是从父类 Logic Object 派生出来。Logic Object 并不包含任何数值逻辑。由其派生类来负责数值逻辑。这是现在的做法。以后可能会把逻辑对象的数值抽取出来，作为一个属性域，交由上述的数据集合来管理。这样进一步实现了逻辑对象的抽象性和灵活性。开发者通过名字设定逻辑对象的属性和其类型。这样也便于将来的形式化和脚本化描述。

关于逻辑对象的行为，事实上需要通过消息参数和逻辑对象本身的数值逻辑来进行决策。现在是通过回调逻辑对象的方法的实现，并且回调带有消息参数。因此回调函数可以给予逻辑对象本身的属性和消息参数来作完整的决策判断。

2.2.2 状态机

系统论中讲到，世界是由若干个层次的系统组成，每一件事物都可以当作是一个独立的系统，系统与系统之间通过信息来相互联系。那么状态机便是对系统的描述。系统在没有信息输入的时候，会呈现熵即混乱度增加的状态，会向无序化演变，直到一个混乱度平衡的状态。如果有接受到信息，则会向相反的方向发展，即有序化。如果我们把状态机视作系统考虑，那么实际上是忽略了封闭系统自演化过程。没有信息输入的时候，我们是视其为完全静止的。也就是说状态机系统是现实世界的静态抽象，这样更易于人的理解。[18]

状态机中的状态，是对于系统在某一时刻所有状况的总体描述。比如说我们来考虑这么一句话："今天天气很好。"在这句话里，主语天气是一个系统，即一个状态机，定语今天描述的是时间，谓语很好则是这状态机的一个状态。作为思维的过程，我不需要去考虑阳光的辐射度有多少，也不需要考虑气温有多高。但实际上确实是由阳光的辐射度和气温来决定了今天的天气很好。那么很好这个状态便从天气的复杂数据逻辑抽象出来了。

如何来分析状态？这涉及到我们需要模拟现实世界到何种程度。仍就天气来说。从人的感觉来分，可以分为好、一般和坏。如果要进一步详细化，那么可以分为晴、阴、雨、雪、风、雾等等。当这种详细化的工作到了一定层次，超越了人的思维处理能力的时候，就需要设计子状态机和子状态，把平行的复杂程度划分为层次关系上的复杂程度。

因此，理想的状态机应该是自包含层次结构的，从最低抽象层次到最高层次。但是此处我们在实现上是用了经典的状态机理论。他本身只包含单个状态机的描述。所以我们把层次结构放到了后面的逻辑对象及结构里。

关于状态机，我们前面一般都要加上有限二字作为限定词。这是因为我们处理状态的能力是有限的，因此状态机所包含的状态也是有限的。在这里我用一个状态集合的概念来描述状态机所能处理的状态。这个状态集合是严格状态机相关的，即某一类型的状态机只能拥有同一类型的状态集合。

那么，由上面的原理，我们来设计出相应的程序结构。

首先我们通过全局对象创造出某个状态集合。这里面没有用到类继承的技术，状态集合完全是基于名字管理的。对于给定的名字，创造出相应的状态集合。也可以通过名字来查找状态集合。

当我们创造好状态集合后，便通过状态集合的对象来创建里面所包含的状态类。同样，状态类也是通过名字管理的。如此我们便创建了某种类型的状态集合，以及所包含的若干种状态。

状态机类只是一个容器，包含有一个状态集合对象和一个当前状态对象。逻辑对象类会从状态机类派生出来。由此，我们的状态机类便有了上述的所有属性：静态，非层次结构，状态抽象，某一时刻的状态描述，有限状态集合，独一无二的状态集合。[13][15]

2.2.3 消息

消息调度机制主要包括消息管理（分配）以及消息两个部分。其作用在于将对象间复杂的调用关系简化为对象间的协议，或者说将对象间复杂的多对多关系简化为对象与消息调度器的简单的一对多关系。但是这样的缺点是，对象之间的关系被模糊化了，导致在设计上的不直观，以及运行时更高的代价。因此在应用时根据应用级别的不同而适当选取消息调度或者直接调用。然而目前的消息机制设计的并未完善，因为我们尚不能预测未来可能的应用场景。因此该机制还处于不断的改进之中。

那么在此我不能够完整的描述消息机制的类的组成，只能够描述其概念和方法。

事实上，对象之间的调用关系是不同系统间通过信息而达到控制的具象表现。也就是说，对象调用关系实际上是一个对象通过向另一个对象传递信息而达到控制对象行为的关系。那么我们可以把这些被传递的信息单独抽离出来，称之为消息，并给这些消息一条唯一的通道，消息管理器。那么我们就可以很轻松的把对象与对象之间的复杂的网状联系简化为对象与消息管理的星形联系。那么在设计类时，就可以忽略对象间调用而产生的复杂度，而只用考虑对象与对象之间传递

信息的内容，以及发送者与接收者如何产生以及解释消息。我们把这个关系称作对象间的协议。

设计上的几个问题

一、是否存在通用的消息格式

因为消息管理器并不能够无限的扩展来适应可能无限扩展的消息类型。因此我们有必要考虑一种通用的消息格式。在这里我们考虑的是，消息是2个固定对象之间的定义好的协议，消息管理器无需了解其内容，产生和解析都是在对象内部完成的，因此消息里只需要一个void*的内存块即可。

二、消息是否应该生存周期

消息是不存在生存周期的，对象行为的时间由对象自身或者其他对象控制，消息都是瞬时的。这一点在上一个设计中是不完善的，因为消息被作为有生存周期的一个存在，从而导致了设计上根本性的困难。

三、消息的效率以及应用范围问题

消息传递实际上是一个对象到消息管理器再到对象的过程，较之对象调用实际不止两倍的消耗，因为还有很多额外的处理。

要决定什么情况下要使用消息机制，在没有大量的实际应用经验之前，很难有个标准。这个标准只能是个经验型的标准。在此，我们先定义为，用于大模块之间的交互，所谓大模块，即比如场景模块，输入模块等等。之后再根据实际应用情况来调整。然则，这套消息机制是支持到最微小的对象级别的，因此在应用中，可以灵活的根据需要来使用，并且来验证以及测试该机制的可适应性。

2.2.4 时间

虚拟世界内的对象大部分都是活动的，因此存在生存周期的问题。消息虽然是实时的，没有生存周期的概念，但是消息也有产生时间的性质。因此我们需要通过一个时间管理器来管理虚拟世界内的时间。

由前文可知，虚拟世界里的时间是按一个个短小的片断构成的。对象的行为被每一个时间片断划分为若干个行为片断。我们需要通过时间管理器在每一个时间片断内继起的对象的行为，并且在时间片断结束时停止对象的行为。同样，消息也根据其产生的时间片断被划分开来。在同一个时间片断内产生的消息根据产生的时间逻辑顺序在消息队列中排列。并且会在该时间片断内被悉数处理完毕。

那么以上是完全按照现实世界中时间正向流动的方式管理虚拟时间的。然而正如同前文所述，虚拟时间除了可以正向流动之外，也可以静止或者倒流。所谓

静止，即我们把虚拟的世界的状态停留在当前，停止更新对象的行为，停止产生新的消息，停止处理已有的消息。也可以把当前所有对象的状态和所有消息作一个备份，以当前时间命名。那么在需要的时候，可以把系统整个回复到备份的状态。

在对象的行为模型里，行为是以时间参数与消息驱动的。当我们给对象一个正数的时间以及消息队列里下一个消息的话，那么对象会按照时间正向流动行为。我们维护一整个时间过程的消息队列。给于对象行为模型一个负数的时间参数以及上一个消息，那么对象则会按照时间负向流动行为。当然要实现时间逆向流动，还有很多细节上需要考虑的因素。

2.3 本章小结

本章分析了虚拟世界的性质，以及虚拟世界的构成。

并由虚拟世界的构成入手，简要地阐述了实现虚拟世界的设计思路。在下一章里，将会按照这些思路详细地介绍游戏仿真层设计的各个细节。

第三章 仿真层架构设计与实现

这一章会详细的叙述仿真层架构的实现。其中内容包括一些编程的技巧，设计模式的改良，数据结构的设计与实现，部分辅助类的设计，架构模型的详细分析以及开发过程管理的原则等等。

3.1 全局对象实例

全局对象实例 globe_instance 实际上是作为 singleton 模式的一个补充。通过全局对象实例可以对任何已有类型构建一个唯一的全局对象，在任意位置访问，并且不需要任何额外的代码。它构造的全局对象自创建时期起具有全局生存周期，直到手动销毁或者程序结束。[33]

它主要被用来实现仿真层框架中几大需要全局生存周期的管理对象，比如逻辑对象管理，时间管理等等。传统的 singleton 模式往往需要通过继承的方式来实现对象的全局访问。如果有很多对象需要全局访问的特性的话，那么将会额外增加许多代码。使用全局对象实例可以避免这种情况的发生。它采用了模板的技术，可以为任何一个已知的类或者类型创建一个全局的实例。只需要用简单的代码即可访问，如已知类 A，那么访问它的全局对象的代码为：globe_instance<A>::InstancePtr()。即可得到一个全局的 A 的对象的指针。

使用全局对象实例还有个好处在于，脚本里有很多情况会调用 C++中的函数。通常这些接口都是通过全局对象暴露出的。通过全局对象实例可以方便的管理所有的全局对象，从而使得脚本引擎的开发更为灵活。

全局对象实例的缺点在于，不能像基于继承的 singleton 模式那样自动销毁对象。因此必须在合适的时候手动调用 Release()函数释放全局对象。此外，全局对象实例只能保证一个类只能有一个并且是唯一一个全局对象，当需要二个或者以上的同类型全局对象时，globe instance 变不再适用。

它的代码很简单，如下：

```
template< typename T >
class globe_instance
{
protected:
    static T* _instance;
```

```
public:
    //! 全局对象指针
    static T* InstancePtr( void ){
        if( !_instance )
            _instance = new T();
        return _instance;
    }
    //! 手动销毁
    static void Release( void ){
        SAFE_DELETE( _instance );
    }
};
template< typename T >
    T* globe_instance< T >::_instance = NULL;
```

3.2 数据集合

3.2.1 设计思路

在以往的设计中，对象的数据成员都是在编译阶段已经固定的。这是出于效率考虑。然而在实际应用中，我们往往需要可变的数据成员，尤其是在游戏逻辑中。此外，考虑到大规模的游戏逻辑时，一个可以动态定义数据结构的灵活框架，可以让开发的效率大大的提高。此处的核心问题是运行效率与开发效率的矛盾。然而随着硬件能力的不断提高，运行效率已经渐渐不再作为着重考虑的方面，随着软件规模的不断增大，开发效率逐步成为十分重要的问题。

数据集合便是本着用灵活的数据结构简化系统内部设计从而提高开发效率的思路而设计的。

比如 Lua 脚本语言，它的数据结构相当的灵活。每一个变量没有类型的限定，可以赋予它任何类型的值，整数、浮点数、字符串或者函数。Lua 中还有个相当重要的数据结构类型 table，它的数据成员数量是可变的，数据成员也如同普通变量一般，也没有类型的限定。因此 Lua 具有相当高的灵活性。

对于灵活性，也需要有个额定的限度。在我们的设计中，与 Lua 不同的是，对变量的类型有着严格的限定，由此我们称之为半动态数据结构。它的灵活性是体现在数据结构的灵活性上面。与 Lua 中 table 对应的 data set，它的数据成员是可变的。可以运行时建立空的 data set，然后再根据需要来设置它的数据成员。由此引伸到脚本，在将来扩展到脚本时，脚本中定义的数据结构，运行时脚本引擎

载入脚本，按照定义动态的装备数据结构。

之所以要加入类型限定，主要的考虑是在于运行时的安全性。游戏逻辑系统并不是一个模糊的系统，它对的数据的类型匹配要求很高，一旦某一个小的环节由于类型不匹配出现了错误，即便通过出错处理不会立即使程序崩溃，但由于逻辑联系，小小的错误可能会引发整个逻辑系统雪崩般的连锁错误，从而使整个系统得出的结果与预期大相径庭。此外附带的好处在于，加入类型限定后，运行时少了类型变化检查的开销。

函数并没有类似 Lua 一般作为类型加入到考虑中。这是一个尚缺乏灵活性的地方。因为函数和一般数据类型在实现上有很多细节不同，或者说更有难度。因此还没有实现。在接下来的设计中，函数相关的东西将被纳入重点考虑。

3.2.2 在仿真层的应用

数据集合在仿真层主要有 2 个用途。

一是逻辑对象的数据结构实现。

逻辑对象继承于数据集合。通常在实现逻辑对象的数据结构时，我们采用两种方法结合的方式。即传统的静态声明方式与运行时动态设置两种。对于一些固定的不需要变化的属性，我们把它直接声明为对象的数据成员。

另一部分需要运行时动态改变的，举个例子。某个年轻人当他进入大学后，他获得了学分这个属性；他大学毕业时，便失去了学分这个属性。在以往的设计中，我们可以这样做，假设此人为类 Man。那么 Man 具有 2 个数据成员：

```
class Man{
bool m_bStudent;
int m_nScore;
};
```

我们要确定 score 的有效性，必须先判断 student 的有效性。这样无疑多了一层环节，设计时会因为这样的额外判断让复杂度大大的增加，并且更繁琐。

采用动态数据的话，只需要在他入学后为对象 Man 动态增加一个新的数据成员 int score 即可。同理，在离校时把 score 数据域删除即可。数据集合会提供一套机制来确保访问数据的有效性，当访问不存在的数据域时，会得到一个“空”值。

数据集合的第二个应用关于消息参数。

关于消息的最大问题就是消息参数要如何定义，通常有三种方式。

- 统一固定参数。所谓统一固定参数，即所有消息的参数个数，类型都是相同的，比如 Windows 消息循环。根据消息类型的不同，每个参数的意义也不同。这样的好处在于统一固定参数后消息接受函数的格式也统一了。不会因为不同格式而导致大量的不同格式的函数声明。但这种方式难于设计，统一所有的消息为一种格式不是简单的活儿。扩展性也很差，当有复杂消息加入的时候，可能会考虑改变整个系统。
- 不同消息自定义固定参数。即每个消息单独声明格式。这样灵活性就很高。但随之而来的便是随着消息的增多，消息声明也越来越多，占用大量的代码篇幅，可读性极差。同时相应的消息接受函数也愈来愈多。整个程序结构趋于混乱。
- 传递指针。传递一个 void*指针，指向所需要的消息参数内存块。这样消息格式统一了，消息参数类型也是可变的。但存在一个严重的问题——安全性。因为并不能保证指针指向的内容是一定正确的。

采用数据集合作为消息格式，既能使消息格式统一（一个数据集合对象），消息参数类型也是灵活多变的。同时内在的类型限定机制保证了安全性。当一个对象需要发送消息时，只需要产生一个数据集合，然后将目的对象感兴趣的参数动态地装配到数据集合中，在发送给目的对象；目的对象接收到消息后，通过数据集合的类型机制检查自己需要的参数是否完整，即可进行相应的操作。

3.2.3 类型封装与 RTTI

首先我们会通过一个抽象数据类 G_Data 来封装所有的数据类型。

G_Data 具有如下几个成员函数：

```
//! 获取数据值
virtual void SetData( void *pData )  = 0;
//! 设置数据值
virtual void GetData( void *pData )  = 0;
//! 数据类型
virtual tstring DataType( void ) = 0;
```

G_Data 类实际上只给出了相应的接口。实际包含数据以及实现的类是一个模板类 data_t。

```
template< typename T >
class data_t : public G_Data{
private:
    T m_data;
```

```
    public:
        virtual void SetData( void *pData ){
            if( pData ){
                m_data = *( ( T* )pData );
            }
        }
        virtual void GetData( void *pData ){
            if( pData ){
                *( ( T* )pData ) = m_data;
            }
        }
        virtual G_Data* Clone() {
            return ( G_Data* )( new data_t< T >( m_data ) );
        }
        virtual tstring DataType( void ) {
            return CLASS_NAME( T );
        }
    };
```

其中设置数据与获得数据函数中有一个不安全操作，即强制转换类型与强行赋值。由于 data_t 对开发者是不可见的，需要通过 data set 类对其操作。在 data set 类的操作函数中对类型的要求使得 data_t 中的强制转换类型得到确保二者一定是同一类型。对于强行赋值的问题，这里有一个限制。只有保证=操作安全的类型，才可以作为抽象数据类型的实现存在。比如说基本数据类型，或者确保拷贝构造函数安全的类。

此处对于数据类型 Data Type，采用了 RTTI（运行时类型信息）的机制。因为 C++中 typeid 操作符得到的结果会因编译器的不同而变化。因此我们自己实现了一套简单的 RTTI 机制。[19]

```
template< class T >
class class_name_t{
public:
    static tstring _class_name( void ){
        return _T( "Class Name" );
}};
#define CLASS_NAME_DECLARE( Class )\
template<>\
class class_name_t< Class >{\
public:\
    static tstring _class_name( void ){\
```

```
        return _T( #Class );\
}};
#define CLASS_NAME( Class ) class_name_t< Class >::_class_name()
```

在 data_t 中，通过 DataType()函数，将模板的静态数据类型保存起来。用于之后的动态类型检验。

3.2.4 数据管理器

数据集合是一个基于名字映射表的数据管理器。它有如下几个方法：

- 创建数据。创建一个（名字、数据）的二元映射。
- 删除数据。删除某个名字的数据。
- 验证数据类型。判断某个名字的数据类型是否为给定类型。
- 设置数据。将某个名字的数据设置为给定值。
- 获得数据。得到某个名字的数据的值。

这里，名字在将来的脚本中，作为数据域可以直接用“类型名.名字”访问。

3.3 成员函数回调与对象行为

通常实现对象行为的有二种方式，一是基于继承的，二是基于回调函数的。这二种方式各有其优缺点，业界关于这二种方式的争论已经持续了很久，并且会在相当长的一段时间内继续争论下去。在我而言，自己更倾向于基于回调函数的形式。因为回调函数的方式更灵活，适合于脚本的使用。

下面我们简要的来阐述一下回调函数的作用方式。

这里我们把需要回调的函数原型封装成为仿函数对象。所谓仿函数对象即是重载了()操作符，可以用类似函数调用的格式的对象。我们以 0 参数回调函数作为例子。[12]

```
template< typename RetType, typename ClassType >
        class class_functor0{
        private:
            RetType ( ClassType::*m_functor )( void );
        public:
            class_functor0( RetType ( ClassType::*func )( void ) )
                : m_functor( func ){
            }
            bool empty( void ){
                return !m_functor;
            }
```

```
        void bind( RetType ( ClassType::*func )( void ) ){
            m_functor = func;
        }
        RetType operator() ( ClassType *pObject ){
            return ( pObject->*m_functor )();
        }
    };
```

其中数据成员 RetType (ClassType::*m_functor)(void)是指向 0 参数成员函数的指针。可以通过构造函数或者 bind 函数为其赋值。通过重载的()操作符，传入一个相应对象的实例，调用相应的成员函数。

然而回调函数类不能直接应用到到逻辑对象中，需要通过一层封装。封装的原因在于要在接口上忽略回调函数的类类型，而实际调用过程中需要类类型。因此我们通过更上一层 Action 类的封装，把类型信息通过虚函数屏蔽掉。由此达成了非常灵活的回调函数机制。

逻辑对象的 Action 分为两类，一类是由消息继起的消息响应动作，另一类是由时间循环继起的定时更新动作。他们分别由 Action 类的 2 个子类 action_t 与 action_update_t 实现。Action 类通过消息类型在逻辑对象内构建起一张二元映射表。

3.4 ID 管理

ID 产生器用于产生，管理一组独一无二的 ID。

包括产生，删除两个方法

void Generate ();

bool Delete (unsigned int ID);

ID 为无符号非零整数。

通过管理一组空闲 ID 域来实现。

这组空闲 ID 域用一个链表来表示。

链表的每个节点由一个二元组 N 表示，如 N[1，0xffffffff]，每个二元组表示一个空闲的 ID 空间。

二元组表之间关系必须满足

Min(N)<=Max(N)，如 N[1,1]或者 N[1,100]，表示该二元域值空间不能为 0；

Max(Nn)<Min(Nn+1)+1，比如 N1[1,10],N2[13,15]，这表示二元域组之间是不连续的，否则需把两个二元组整合成一个。如 N1[1,10],N2[11,15]等价于 N1[1,15]；

初始化时，空闲域为整个无符号非零整数，即 N[1,0xFFFFFFFF]。

当需要产生新的 ID 时，从链表中第一个节点的空闲域中选取最小值，同时缩小域空间，或者删除该域（域空间为 0 时）。

当删除某个 ID 时，查找该 ID 处于某 2 个连续的空闲域之间，如 Max(Nn)<ID<Min(Nn+1)，对于第一节点 Max(N0)设作 0，最末节点 Min(Nlast)设作无穷大。

在此 2 个空闲域之间插入[ID,ID]的新空闲域，并视是否可以与前空闲域或者后空闲域合并。

域合并运算

N1,N2 有交集或是连续的则可以合并。

有交集或连续可以用如下表达式描述

(((N1.Max >= N2. Min-1)&&(N1.Min < N2. Min))||((N1.Max > N2. Max)&&(N1.Min-1 <= N2. Max)))

或运算前面的表达式表示 N2 的左边界在 N1 之内或连续，后面的表达式表示 N2 的右边界在 N1 之内或连续。

3.5 消息管理

3.5.1 消息产生

消息管理器是一个全局对象实例（见前文）。所有对象或者外部界面（如图形用户界面）都可以访问消息管理器。通过消息管理器的相应接口产生一个制定类型的空消息。然后在通过数据集合（见前文）的接口，创建消息参数并赋值。产生的消息会被立即放到消息队列的最末，等待下一个时钟周期的处理。同时，创建消息的对象需要指定一个或者若干发送目的地。这个目的地是通过逻辑对象 ID 标识的，前文说过，逻辑对象通过 ID 管理会得到一个独一无二的 ID，因此消息管理器会根据目的寻址，精确无误的将消息传送到需要的对象上。如果没有指定需要目的对象 ID，即目的 ID 列表为空，那么消息管理器会对所有对此类型消息感兴趣的对象广播消息。[9][10]

3.5.2 对象组织

对象在消息管理器中会根据它们感兴趣的消息类型组织为一个索引表。其定义如下：

```
hash_map< tstring, hash_map< unsigned int, G_LogicObject* >* >  m_registerMap
```

前一个表的 key 类型是 string，用于消息类型（字符串类型）的索引。数据类型是另一个表，此表是 key 为 ID，数据为逻辑对象指针的映射表，标明了所有对此消息感兴趣的对象。

对象在创建时，在绑定消息与动作阶段，会通过一段代码自动的把对象与其感兴趣的消息在消息管理器中注册。因此开发者只需要着重与设计逻辑对象本身。这样大大减少了开发者的额外负担。

之所以要把对象预先注册，是为了减少消息寻址的时间。以往没有注册对消息感兴趣的对象时，在遇到需要广播的消息时，会进行很多无意义而又浪费时间的操作。这样的设计大大减少了消息寻址的时间。此外在对定向寻址的消息处理时，如果该消息的目的地址不存在于注册表中，那么也可以避免在继续把消息发送给对象而引起很多不必要的操作。

3.5.3 消息处理

在每一个时钟周期到来时，消息管理器便会对消息队列中的所有消息进行一次分派。如前文所说，消息没有生存周期，当前周期产生，分派完毕便被销毁。如果对象在相应消息的动作中又产生了新的消息，那么该消息会被立即放在消息队列的末尾。

当所有的消息被处理完毕，结束当前时钟周期的消息处理，转入其他的工作。由此这里存在一个缺陷，如果某个对象的消息相应不断的产生新的消息，那么此次循环会被无限制的进行下去。因此在设计对象的消息相应动作时，正如同设计递归一般，需要有个终结的状态。由此消息管理本身不负责这种无限循环的逻辑错误。[23]

3.6 状态、转换规则与状态机

3.6.1 复杂状态机的问题

我们可以用一个例子来说明，假设有一盏灯，一个开关控制灯是亮还是灭，那么：

状态集 S={On, Off}

输入字母表 I={Turn on, Turn off}

转换函数 f

f(On, Turn off)= Off

f(Off, Turn on)= On

用图表的方式表示就是：

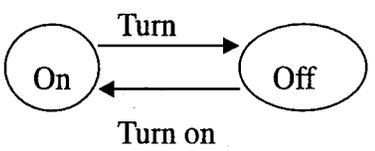

图 4-1

上面所举的例子很简单，在程序上也很容易实现，那我们再考虑一个更复杂的例子。

同样有一盏灯，有 2 个开关 S0，S1，当且仅当两个开关同时被按下时灯才亮，那么对应的有限状态机设计如下：

状态集 S={灯不亮且无按下，灯不亮 S0 按下，灯不亮 S1 按下，灯亮 S0S1 均按下}

输入集 I={S0 on, S0 off, S1 on, S1 off}

转换函数 f

f(灯不亮且无按下, S0 on)= 灯不亮 S0 按下

f(灯不亮且无按下, S1 on)= 灯不亮 S1 按下

f(灯不亮 S0 按下, S1 on)= 灯亮 S0S1 均按下

f(灯不亮 S0 按下, S0 off)= 灯不亮且无按下

f(灯不亮 S1 按下, S0 on)= 灯亮 S0S1 均按下

f(灯不亮 S1 按下, S1 off)= 灯不亮且无按下

f(灯亮 S0S1 均按下, S0 off)= 灯不亮 S1 按下

f(灯亮 S0S1 均按下, S1 off)= 灯不亮 S0 按下

我们再考虑由 N 个开关控制的灯的有限状态机模型，状态会有 2 的 N 次方个，输入有 2*N 个，转换函数有 2^N*N 个。由此可以看到，随着 N 的增大，模型会变得极为复杂。

通常，为了让模型简化，我们会把它改造成一个非确定有限状态机(NFA)，或者说带判断过程的非确定有限状态机，如下面的例子：

有 2 个开关的灯，在接收到有开关操作输入的情况下进行判断，是否 2 开关均被按下，如果均按下，则灯亮，否则灯灭。

同样对于有 N 个开关的灯，只需要改变判断，即 N 个开关是否均被按下。

实际上，这种解决方案是把条件反射式的简单状态机输入处理的转换函数扩展为带判断的复杂处理过程。在处理过程中，依据状态机自身属性和输入参数作为判断，根据判断修改属性、产生新的输入以及跳转状态。[6]

3.6.2 输入处理过程

如果我们从客观世界中对象处理输入来看，状态机模型实际上是一种“条件反射”的方式，即由输入和当前状态立即产生下一个状态，本身缺乏处理输入的“思维过程”。虽然无论多么复杂的思维过程都可以转化为有限状态机模型，但正如同前面的例子一般，这样的有限状态机模型也是非常复杂而难于设计和实现的。[1]

而程序语言其长处正在于处理复杂的逻辑过程。因此我们考虑将有限状态机条件反射般的转换函数扩展为具有判断能力和行为能力的处理过程。那么对于有 2 个开关的灯，在接收到有开关操作输入的情况下进行判断，是否 2 开关均被按下，如果均按下，则灯亮，否则灯灭。同样对于有 N 个开关的灯，只需要改变判断，即 N 个开关是否均被按下。

其中，灯是接收输入的对象，开关操作是输入，所有开关是否均被按下是接收输入后的判断，灯亮灯灭是判断后产生的动作。

我们可以用这样一个图来描述此模型：

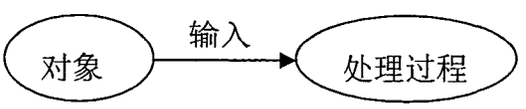

图 4-2

其中处理过程包括，判断和动作。

3.6.2.1 属性

为了能够更好的描述事件处理过程，我们再扩展 2 开关灯的例子：一盏有 2 个开关 S0，S1 的灯 L，在灯没有被损坏的情况下，当且仅当 2 开关同时被按下时，灯亮。任何情况下可以破坏灯，但修复灯需要在两个开关都没有被按下时才可进行（考虑到安全性）。

其中，灯 L 是可接受输入的对象，它包括 3 个属性：开关 S0；开关 S1；是否损坏。所谓的属性，是对象某一方面情况的反映，比如说某对象的运动速度，某空间对象的长宽高等等。属性是用于判断的基础，可以由动作改变。

开关 S0，S1 可以取值有 2，按下或未按下，是否损坏属性可以取值是否，此处为了方便，统一用 Y 和 N 表示。那么我们可以用一张表格来描述对象 L 的属性：

S0	S1	Broken	描述
Y	Y	Y	灯灭，不可修复
Y	Y	N	亮
Y	N	Y	灭，不可
Y	N	N	灭
N	Y	Y	灭，不可
N	Y	N	灭
N	N	Y	灭，可
N	N	N	灭

表 4-1

3.6.2.2 输入

所谓输入是外界传递给对象的信息，对象得到输入包含的信息以及自身属性来进行分析处理。

那么对象 L 可以接受的输入有：

S0 turn on

S0 turn off

S1 turn on

S1 turn off

Destroy

Repair

输入包含了两类信息，一类是类型信息，一类是属性相关的信息，比如说 S0 turn on，turn on 是类型信息，S0 是属性相关信息。

那么事件可以简化为

Turn on (S0, S1)

Turn off (S0, S1)

Destroy

Repair

3.6.2.3 处理过程（动作与判断）

那么事件可以简化为

对象在接收到输入之后进入处理过程。

其中动作是指要改变对象属性的过程，判断是基于对象属性和输入信息的分析过程。比如，对象 L 在接收到 turn on(S0)后，首先会执行一个动作将属性 S0 置为 Y，然后判断 2 个按钮是否均被按下，再判断灯是否损坏。然后根据判断结果来将灯置为亮或者灭。

下面是对象 L 完整的输入处理列表：

Turn on	设置相应的开关，判断是否所有开关均被按下以及灯是否完好，根据结果设置灯亮。
Turn off	设置相应的开关，设置灯灭。
Destroy	设置灯损坏。
Repair	判断是否没有开关被按下，根据结果设置灯完好。

表 4-2

3.6.2.4 输入处理过程与原有转换函数的比较

前面用例子阐述了输入处理过程，我们可以看到，较之原有转换函数，输入处理过程拥有了针对输入的“思维能力”，能更加接近客观世界对象的行为方式。

从复杂程度而言，加入输入处理过程并没有本质的，仍然具有被模拟的客观世界对象本身的复杂度。不同的是，有限状态机是把复杂度拆解到海量的状态、

输入以及转换函数上，而输入处理是把复杂度还原到对象本上的属性，事件的信息含量以及处理输入的逻辑过程上，尤其是处理输入的逻辑过程。而逻辑过程本身更符合人的思维过程，因此在设计上会更加简单，在程序的实现上也相对方便。

3.6.3 状态抽象

由前面的例子可以看到，加入事件处理过程后，状态统统可以用属性值来表达，我们似乎完全可以忽略“状态”了。

在前面的例子中，有一处概念尚且比较模糊，就是关于灯亮灯灭的概念。它不属于对象的属性，因为它虽然会被改变，但却不是判断所需的内容，那要如何定义灯亮灯灭呢？

我们再从另一个角度来考虑。假设此灯是某道路上的交通指示灯，灭表示不可通过，亮表示可以通过。那么对要通过该道路的人而言，他们所关注的只在于灯是亮是灭，而不会考虑到灯是否坏了，或是哪一个开关没有被按下。由此，我们可以说灯亮或灭是灯给予外界的参考数据，让外界可以忽略灯的具体属性。

那么我们又找到了需要“状态”的地方。此处的状态与传统有限状态机的区别在于，前者是抽象概念，有更底层的属性数据作支撑，而后者则是纯粹的底层数据。

状态抽象除了做为提供给外部的抽象数据之外，同时，在程序实现输入处理过程上也有相当的价值，它可以提供对于输入的预判断作用。

在原有的输入处理模型中，对象对于接收输入来者不拒，无论此输入于当前属性有无意义，都要处理一番。那么如果在程序中我们也采用这样的方式，无疑程序的开销会大很多，尤其是在处理过程相当复杂的情况下。

从 2.1 中的属性表格里我们可以看到，在对象处于某种状态时，有的事件是可以被忽略掉的。比如，在灯亮的时候，Turn on 和 Repair 事件是可以忽略掉的。即接收到 Turn on 和 Repair 事件后不作相应处理。

假设灯具有亮和灭两个状态的话，我们把相应的可接收的输入列举出来：

亮	Turn off, Destroy
灭	Turn on, Turn off, Destroy, Repair

表 4-3

这是对于路人而言灯所具有的状态。

我们可以通过对状态的重新设计来让该模型更直观，比如加入一个“损坏”

的状态（即灭表示灯尚完好），此状态下无需继续损坏，也没有必要按下开关，可接收的输入列表如下：

亮	Turn off, Destroy
灭	Turn on, Turn off, Destroy
坏	Turn off, Repair

表 4-4

这可以视作对灯的维修人员而言的状态。

3.7 其它

3.7.1 逻辑与图形的二元关系

首先摆在面前的问题是逻辑与图形的结合，我还不能明确逻辑与图形是如何结合，但有一点可以肯定：逻辑和图形间是有联系的。那么我们可以初步为此建立一个 2 元关系。

逻辑对象<——>图形对象

我们从对象间的关系来考察大的模块间的关系，目的是为了简化考虑。我们忽略两对象本身的内容，只考虑二者的联系。这里我们简称逻辑对象为 Lo，图形对象为 Go。

- 如果二者视作本质与表现，那么 Lo 是本质，Go 是表现；
- Lo 作用于 Go，反映在程序中为主动调用的关系；
- Lo 先于 Go 创建，后于 Go 销毁，Lo 负责创建和销毁对应的 Go；
- Lo 知道 Go 的存在，Lo 随着自己的变化改变对应的 Go；

3.7.1.1 时间调度与更新

Lo 何时改变自己？何时改变 Go？改变后的 Go 何时被绘制出来？

以上的问题通过基于时间的调度来实现。每一个时间周期，Lo 更新自己，如果有改变，则作用于 Go，Go 被改变后，在同一个时间周期内所有 Lo 得到更新后，统一被绘制出来。

3.7.1.2 基本的逻辑和图形系统

在这个系统中，我们考虑只有一个 Lo 和一个 Go，然后还有一个定时调度系统。Lo 和 Go 都有 Update 这个方法，在定时调度时调用。Lo 提供处理自身属性的方

法，Go 提供绘制相关的方法。由此来模拟逻辑和图形的关系。

为了方便定时调度系统调用，分别设计一个逻辑对象管理器和一个图形对象管理器。因为图形对象对调度器是不可见的，所以需要通过将图形对象注册到对象管理器中方便调度器调用。在这里管理器被简单的设计为一个容器，只负责遍历对象。

3.7.2 应用程序框架

应用程序框架部分采用了类似 MFC 的构架，将应用程序主函数封装到库中，后续开发者只需要重载应用程序类。同时将窗口程序和控制台程序统一封装到应用程序类中，只需要通过修改项目属性和添加预编译指令就可以在二种程序类型间转换。同时封装了窗口类，用回调机制来实现窗口的消息处理，使得窗口类的行为更加方便。

主要包含二个类：

3.7.2.1 CApplication 类

Initialize() // 窗口和控制台有不同参数

返回值为 0 表示初始化失败，退出程序。

Run()

使用：

重载 CApplication 类

对于控制台程序一般只需重载 Run 函数。

窗口程序按需求而定。

3.7.2.2 CWindowCustom 类

（因为 CWindow 是 VC 的保留类名）

Create() // 创建窗口实体，构造函数只负责初始化数据

BeforeCreate() // 在窗口创建之前所作的动作，比如说构造消息映射表等，可以重载

AfterCreate() // 在窗口创建之后所作的动作，一般是与窗口句柄相关的操作，可以重载

上二者是保护对象。

GetHWND() // 得到窗口句柄

WindowProcess() // 消息分配函数，外部的系统回调函数调用此函数作为

向对象传递消息，同样，可以让用户手动传送消息

BindMessageFunc() // 绑定消息处理函数

m_messageMap // 消息映射表，是消息与对应处理函数的 hash_map

3.7.2.3 构造消息映射表的方法

比如说要添加 WM_KEYDOWN 的消息处理函数

那么在 CWindowCustom 的派生类中添加一个原型为

LRESULT xxx(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

的函数。

然后在重载的 BeforeCreate()函数中添加

BindMessageFunc(WM_KEYDOWN, &CMyWnd::xxx, this);

即可。

3.7.2.4 定义应用程序类型的方法

通过预编译指令以及项目属性来配置

#define _WINAPP_ // 定义为窗口应用程序

#undef _WINAPP_ // 定义为控制台应用程序

二者互斥，一般在程序入口处定义或者标准系统包含文件处定义。

同时修改菜单项目->属性->链接器->System->子系统为对应的选项。

3.7.3 定时调度器

3.7.3.1 CTimer 定时器类[32]

End(); // 停止定时器

Expire(); // 定时器触发的操作，后续开发者一般不要重载此方法

GetXXX(); // 得到定时器 XXX 的数据

Start(); // 开始定时器

onTimer // 定时器触发时调用的用户指定回调函数，一般用此即可

3.7.3.2 CTimerWin Windows 定时器类

由 CTimer 派生，用于低分辨率的定时要求

3.7.3.3 CTimerMM 多媒体定时器类

由 CTimer 派生，用于高分辨率的定时要求

3.7.4 用户输入

很明显这部分的作用是接收用户的输入信息，包括鼠标、键盘和游戏杆。同时还要把这些信息转化为程序内部可以识别的消息并发送给消息分配器。用户输入有两种运行机制，一是基于轮询，另一是基于多线程事件响应，这两种机制会通过暴露出的接口予以设置，再者设置一般在应用程序初始化时完成，不推荐在运行时更改。此外，根据应用级别的不同，会分为普通 Windows 输入和 Direct Input 输入两种类型。前者不支持多线程的方式，后者两种方式都支持，但开销更大。

主要有如下的类：

3.7.4.1 CInput 抽象输入类

```
AcquireInputData();          // 读取用户输入到输入类的存储区
GenerateInputMessage();      // 将输入信息转化为消息
SetAutoUpdate();             // 设置自动更新模式，也就是多线程模式
onAutoUpdate;                // 自动更新时的回调函数
```

其下有若干派生类，主要用于相关类型的初始化工作

CDirectInput

CwinInput

3.7.4.2 辅助类

CKeyboardInput

CMouseInput

CJoystickInput

这三者主要封装了相关输入设备数据的存储和解析。

由此派生出

CDIKeyboard, CDIMouse, CDIJoystick, CWinKeyboard, CWinMouse, CWinJoystick

其中 CDIKeyboard 是由 CDirectInput 和 CKeyboardInput 多重继承而来，其余同理。

使用：

在应用程序中声明一个或者多个上述 6 个类的对象的实例。

方式 1：

在定时器的回调函数中写

Keyboard. AcquireInputData();

MessageDispatcher.HandleMessage(Keyboard.GenerateInputMessage());

方式 2：

将上面一段代码写到回调函数中。

然后调用

Keyboard.SetAutoUpdate(true);

即可。

3.7.5 声音播放

声音部分主要包括声音文件的载入，播放，流文件的播放，音频特效（包括 3D 音效、回音等等）。由于音频特效尚处于学习研究中，因此设计上并没有考虑完善。目前主要是针对载入和播放的设计。所有的解决方案都是基于 Microsoft Direct Sound，因此主要是针对底层 Wave form 的音频，对于更高级的诸如音频编码解码，MIDI 等目前也未有相关的支持。然则 Sound 部分是设备级的底层音频操作，因此要扩展到支持高级特性也是很方便的。

主要包含有四个类：

3.7.5.1 CSoundManager 类

作为音频设备的管理者以及音频的创建者

```
Initialize();       // 初始化
CreateSound();      // 创建一个音频
CreateStream(); // 创建一个音频流
```

3.7.5.2 CSound 音频类

```
LoadFromWaveFile();   // 从 Wave 文件读入
Play();               // 播放
Stop();               // 暂停
Reset();              // 回到开始
Seek();               // 设置播放点
```

3.7.5.3 CStream

继承于 CSound 的音频流类。

方法与 CSound 相同，但是实现不同

3.7.5.4 CWaveFile 音频文件类

因为 Wave 文件是 RIFF 格式，因此需要用 Windows Multimedia I/O 来相应的操作。因此通过把这些操作封装到 CWaveFile 类中，同时还可以起到代理模式的作用。在需要时加载相应的文件。内存文件操作也封装到这个类里面。

方法包括 Open，Write，Read，Close 等常用的文件读写操作。

使用：

应用程序初始化时创建 CSoundManager 的实例。

通过 CWaveFile 来管理 Wave 文件，需要播放时，创建 CSound，调用 CSound 的 LoadWaveFile(CWaveFile*)方法将文件实际载入内存。然后调用 CSound 的方法播放。音频流的使用同 CSound;

3.7.6 软件开发过程

对于软件开发过程来讲，有两个主要的问题：

怎么样安排任务的顺序？

什么时候能够完成？

在开发中，最糟糕的事情莫过于在项目的尾声还在写核心部分的代码，这样不仅仅会对项目组造成极大的压力，而且还会严重的影响到软件交互后的稳定性。因此，我们需要安排合理的开发顺序来避免上述情况的发生。理论上来讲，我们是可以预知项目的某个部分可以优先或者某个部分可以延后，从而优化我们的开发顺序。

所有软件都是由有限个部分组成的，因此我们可以通过描述这些有限的组成部分来描述项目的完成情况。然而对于游戏引擎来讲，或者说对于游戏来讲，我们能够直接看到听到的，只是其中的一部分，还有很多的内部实现是我们看不到的，因此在定义一个游戏引擎的组成部分时，只定义可见部分是不完善的，那么我们要怎么来定义一个完整的游戏引擎组成部分呢？

在拿到需求文档，进行概要设计的时候，我们通常会把项目划分为若干个模块，而模块又是若干个细节组成的。模块是基于软件中的逻辑联系来划分，而细节更着重于时间的划分，也就是说我们主要考虑一个细节需要多少“天”来完成。

做到这一步并非简单的事情，需要对技术有足够的掌握以及相当的开发经验。通常我们会考虑由经验丰富的老手来协助新手完成这个划分。

除此之外，我们还要考虑到，很少有模块可以一次性完成，往往会对同一个模块反复的操作：修改 bug，增加或减少细节，优化甚至重写。然而这种重复工作并不往往都是恶性的，绝大多数优秀软件的开发过程中都会有这样的情况存在，因此我们在定义模块开发时间的的时候，通常会给一个+2 天的活动时间。[25]

3.7.6.1 增量式的开发

几乎所有的软件开发都是基于某种里程碑（mile stone）系统管理。相比于缺乏管理的混乱的软件开发过程，里程碑式的管理有它好的一面，也有不好的一面。毫无疑问，它的严格交互时间会让项目组产生很大的担忧。此外，它还会让复杂的团队协作变得更加复杂，尤其是在团队中的一环出现延迟或者提前完成的情况下。为了解决这些情况，项目管理需要通过各种有效的手段来调节，比如说文档上的协调等等。

我们来看看一些基本的假设：

如果有人没有达到里程碑，则不可能按时交付；

如果设置了不现实的里程碑，则不可能按时交付；

管理者需要了解团队对于开发周期的看法和重要的里程碑，否则不能够安排合理的计划；

由此可以清楚地看到，明确每一个里程碑，比让团队去按某一个里程碑执行更为重要。在划分里程碑的时候，我们也需要对里程碑进行一个分类：内部的和外部的。内部里程碑通常是中层或者底层细节相关的，时间可能是天或者周来衡量；外部里程碑是可见的，交付相关的，时间可能是周甚至月来衡量。

那么对于项目的管理者而言，他可以不需要了解内部里程碑，也就是说把开发团队视作一个“黑盒子”以及相应的“接口”。项目管理者可以对开发人员提两个要求：

“给出最新版本软件”

“给出最新的时间表”

实际上这是管理者和开发人员的交互最简单的一个模型，实际上不可能只是这样。管理者还需要了解很多细节，比如延期的原因，或者近期项目相关的描述等等。

还有一个相当重要的问题，即管理与开发对于里程碑的认知上的不同，这是

基于可见性这样一个概念的。管理者通常会忽略他们看不到的细节，尤其是游戏引擎这样的倚重于可见内容的项目，比如 AI、物理、内存管理、脚本系统、数学库、优化、bug 修改等。管理者对于这些细节的忽视会造成开发者相当大的困扰，数周或者数月的底层工作而没有新的可见的东西是相当常见的情况。因此对于开发过程而言，也是管理者的一个学习和积累的过程，这对于管理者乃至整个项目都是重要的过程。

在开发者方面，我们可以通过一定的策略来规避这个问题。开发者需要建立一个版本管理的机制。在交付的时候并不交付当前版本，而是交付上一个版本，并且附上详细的版本信息。这样一来，一是可以在交付时出现问题后可以及时地 Roll-back，二是让管理者清晰的看到每个版本的变化。那么为什么不交付当前版本呢？因为当前版本不一定是完善的，不能够反映出版本的变化。在进行当前版本的工作时，同时也是对上一个版本的完善，规范化和整理成文。相比来说，前一个版本较之当前版本，在版本管理良好的情况下，更符合交付件的品质。

3.7.6.2 迭代式的开发

除了按照外部里程碑交付之外，内部我们还需要按照一定的步骤来工作，那么我们怎么来安排这个工作步骤呢？

我们假设有一份对于项目软件的全面的设计文档（这是开发事先应该做的工作），它简要的描述了这款软件的若干特性。那么我们要做的第一步是按照面向对象的方法为软件绘制一张对象图。在这张图里，我们不关心对象间的具体的逻辑联系，主要的目标是一份大而完整的对象列表，而这些对象是直接与软件中的若干概念相对应的。辅助类比如容器类和数学类可以不用考虑，我们只需要考虑逻辑层面的对象。

在得到这么一张列表之后，开发者需要做的是将这若干类进行分类，通常我们会划分为如下三种类型：

核心的类：核心类是构成整个软件的基础。拿游戏引擎来说，包括应用程序相关，渲染，内存管理，声音，输入，脚本支持，资源管理等。对一款游戏而言，玩家角色、状态、NPC 也是核心类的成员。

有了这些类，软件并不一定是可用的，但是却是可执行的。

必需的类：这些类扩展了核心类的功能，使软件成为可用。通常他们较之核心类更为抽象。作为引擎，这些类决定了引擎的类型——第一人称射击类，即时战略类或者其他，比如说 Terrain 类，Movable Object 类等。作为游戏，这些类决

定了核心类的行为，比如说 AI 等。

包含核心的和必需的类之后，软件成为可用的，即可以交付了。也就是说完成一款软件的最小要求就是完成核心类和必需的类。但是我们还是可以做更多的工作来完善这软件。

需要的类：这些类的作用是来让游戏增光添彩的。比如说视觉特效、音频特效、隐藏要素等。这些类不会改变整个软件的行为方式，但是他却可以提升软件应用的深度和广度。

那么由此我们可以给软件一个划分版本的方式。

1、 空版本：只有类的声明而没有实现

2、 基本版：完成了核心类，或者说用临时性的功能代替了核心类的功能。即可执行的实现

3、 可用版：完成了核心类和必需类，所有要求的功能得以实现，即可用的实现

4、 优化版：所有类都已完成。

我们在外部里程碑管理或者版本管理上就可以依照这种划分的方式，从而为软件开发定义了较为明晰的迭代过程。

3.7.6.3 迭代开发的时间管理

迭代开发的焦点往往会从“什么时候能够完成”转移到“什么时候能够做到足够好”上。因此我们有时很难定义一个充分的完成点，但是我们却可以定义所谓的“足够好”，这是由用户需求决定的。开发者的时间管理往往由于不能够抓住“足够好”的点，而导致项目的无限扩大，从而推迟了项目的完成。因此根据用户需求合理的结束项目，也是管理中相当值得注意的问题。

3.8 本章小结

本章从若干细节入手，详细地阐述了仿真层的架构设计，并且给出了其中若干部分的实现。总的来说此架构在结构上已经趋向于完整。开发者可以通过上述若干核心类和辅助类方便的进行开发。同时，有的基础类功能十分强大，除了游戏仿真层，更可以应用于广泛地用其他领域。

第四章 可回溯的有限状态机的研究

4.1 数学模型

一个确定型有限状态机(DFA)我们可以由下面的五元组来表示：

状态集合 S;

输入集合 I;

转换函数 f, $f(s,i)=s'$;

初始状态 S_0;

输出状态集合 F, F 是 S 的子集。

在这个模型中，状态由输入导致变化，可以由输入字符串来推导出未来的状态，但无法回溯得到过去的状态。

在实际应用中，经常需要由已知的过去输入回溯到过去的状态，比如：

在网络的同步问题中，在网络延迟差异存在的情况下，延迟较大的客户端如果在额定时间内没有完成对所有输入的响应，为了保证和延迟较小的客户端内容一致，需要把所有客户端回溯到延迟最大的客户端的状态。

还有的时候，我们为了让游戏增加一些有趣的特性，比如时光倒流，也需要将状态机回退到过去的状态。

由此，我们需要把有限状态机加入回溯的能力。[11]

4.1.1 带逆转换函数的确定型有限状态机

带逆转换函数的确定型有限状态自动机形式化描述如下：

状态集合 S;

输入集合 I;

转换函数 f, $f(s,i)=s'$, i 表示某个输入符号;

逆转换函数 f^{-1}, $f^{-1}(s',i)=s$;

初始状态 S_0;

结束状态集合 F, F 是 S 的子集。

要如何保证这样的有限状态机模型的回溯过程与正向处理是恰好吻合的呢？我们来简单的证明下：

把上面的模型描述为处理输入串的模型，即扩展转换函数和逆转换函数：

假设串$w=ax$，也就是说，a是串w的起始符号，x是包含除起始符号外所有符号的串。

串w^{-1}为w的逆串，$w^{-1}=x^{-1}a$ w’，x’是x的逆串。

转换函数$f(s,w)=f(f(s,a),w)=s'$

逆转换函数

$f^{-1}(s,w^{-1})=f^{-1}(f^{-1}(s,x^{-1}),a)$

$f(s,\varepsilon)=s, f^{-1}(s',\varepsilon)=s'$，$\varepsilon$表示空串，即没有输入。

归纳可得

$f^{-1}(f^{-1}(s',x),a)=s$；（证明过程略）

由上式可以看出，对象在已知的状态下，对于已知的输入消息流，更新输出状态是可确定的。同样，回溯输出状态也是可确定的。并且是与更新输出状态恰好逆转的。

最小单位的转换函数，即基于单个输入符号的转换函数，使得对于输入流的更新或者回溯无论进行到哪一步，都可以进行相应的逆操作使对象达到预期的状态。

4.1.2 逆转过程的非确定有限状态机

上述的模型描述了一种理想的可回溯的状态机，即转换函数与逆转换函数是恰好一一对应的。实际上一般的确定有限状态机的逆转通常是一个非确定的有限状态机(NFA)。[28][30]

如下例所示：

某DFA

$S=\{s_0,s_1,s_2\}$

$I=\{i_0\}$

$f(s_0,i_0)=s_2$

$f(s_1,i_0)=s_2$

那么其逆转函数应为

$f^{-1}(s_2,i_0)=\{s_0,s_1\}$

可见，逆转函数的输出为状态集合，即是说在s_2状态下，接收到逆转输入i_0时，

那么下一个状态可能是s_0，也可能是s_1，具体跳转到哪个状态是不可预见的。为了确保有限状态机的可回溯，那么必须要对 DFA 模型有所改造。

下面是非确定有限状态机的定义：

状态集合 S;

输入集合 I;

转换函数 f, $f(s,i)$ 返回值是 S 的子集;

初始状态 S_0 ;

输出状态集合 F, F 是 S 的子集。

为了符合 2.1 中的带逆转函数的有限状态机模型，我们需要让所有的逆转换函数的输出都是一个固定的状态而不是一个状态集合。即是一个将 NFA 转换为 DFA 的过程。

从有限状态机的理论可以知道，任意的 NFA 都是可以转换为 DFA 的。那么我们要做的就是把逆转换 NFA 转换为 DFA, 并且相应的改变原有的状态机。

下面给出一个一般的由 NFA 构造 DFA 的方法，称之为子集构造：

子集构造从一个 NFA $N=(S_N,I,f_N,s_0,F_N)$ 开始，目标是描述一个 DFA $D=(S_D,I,f_D,\{s_0\},F_D)$，使得 $L(D)=L(N)$。这两个自动机的输入字母表是一样的，D 的初始状态是只包含 N 的初始状态的集合。D 的其他部分构造如下。

- S_D 是 S_N 的子集的集合，即 S_D 是 S_N 的幂集合。如果 S_N 有 n 种状态那么 S_D 就有 2^n 种状态。通常这些状态不都是从 S_D 的初始状态可达的。可以丢弃掉不可达状态，所以实际上，D 的状态数可能远小于 2^n。
- F_D 是使得 $Q\cap F_N\neq\varnothing$ 的 S_N 的子集合 Q 的集合。也就是说 F_D 是所有至少含有一个 N 的接受状态的 N 的状态集合的集合。
- 对于每个集合 $Q\subseteq S_N$ 以及 I 中每个输入符号 a，

$$f_D(Q,a)=\bigcup_{p\subset Q} f_N(p,a)$$

也就是说，为了计算 $f_D(Q,a)$，检查 Q 中所有状态 p，看看 N 在输入 a 上从 p 进入哪些状态，取所有这些状态的并集。[16]

使用理论上的一般方法或许有些麻烦，针对逆转函数，可以采用一些更方便的办法，下面由一个例子来说明：

对于 DFA

$S=\{s_0,s_1,s_2,s_3\}$

$I=\{i_0,i_1\}$

$f(s_0,i_0)=s_2$

$f(s_1,i_0)=s_2$

$f(s_2,i_1)=s_3$

$f^{-1}(s_2,i_0)=\{s_0,s_1\}$

$f^{-1}(s_3,i_1)=\{s_2\}$

首先找到所有$\left|f^{-1}(s,i)\right|>1$的逆转函数，如上例的

$f^{-1}(s_2,i_0)=\{s_0,s_1\}$

把$f^{-1}(s,i)$扩展为$\left|f^{-1}(s,i)\right|$个函数，每个新的函数的输出为$f^{-1}(s,i)$的元素，它们的并集为$f^{-1}(s,i)$的输出集合。同时，将原$f^{-1}(s,i)$中的状态 s 划分为$\left|f^{-1}(s,i)\right|$个新的状态，由此使得每一个新逆转换函数中的参数 s 都不同。如，

$f^{-1}(s_{20},i_0)=s_0$

$f^{-1}(s_{21},i_0)=s_1$

然后再将被改造的逆转函数中的参数状态相关的其他转换函数或者逆转函数相应改变。如，

$f^{-1}(s_3,i_1)=\{s_2\}$

变为

$f^{-1}(s_3,i_1)=\{s_{20},s_{21}\}$

在继续转换为

$f^{-1}(s_{30},i_1)=s_{20}$

$f^{-1}(s_{31},i_1)=s_{21}$

那么最后得到的可逆转的 DFA 为

$S=\{s_0,s_1,s_{20},s_{21},s_{30},s_{31}\}$

$I=\{i_0,i_1\}$

$f(s_0,i_0)=s_{20}$

$f(s_1,i_0)=s_{21}$

$f(s_{20},i_1)=s_{30}$

$f(s_{21},i_1)=s_{31}$

$f^{-1}(s_{20},i_0)=s_0$

$f^{-1}(s_{21},i_0)=s_1$

$f^{-1}(s_{30},i_1)=s_{20}$

$f^{-1}(s_{31},i_1)=s_{21}$

这便是符合 2.1 节中可逆有限状态机模型的有限状态机。为了达到可回溯的目的，作为代价，原状态机的复杂程度也会相应增加不少。

4.1.3 有输出的转换函数

通常有限状态机处理的字符串都是外界输入的，但是有的时候也会在转换过程中产生新的输入，同样在回溯时也需要把这些新产生的输入删除掉。

由于有新输入的产生和删除，是否能保持输入串的一致？以及由此带来的回溯的可确定性问题。下面将会证明是可确定的：

设原始输入串 $w=xiy$，i 表示字符串中某 1 个字符，x 表示 i 左边的串，y 表示 i 右边的串，当前输入指针指向 i

那么新的转换函数为：

$f(s,i)=\{s',o\}$

其中 o 表示转换过程中产生的字符串，可以为空串，转换后的输入字符串为 $f(i)=a$。

由于确定型有限状态机的状态跳转是与输入符号完全相关的，那么只要有串的可确定性，即有状态的可确定性。因此我们只考虑因此把各式的状态省去，那么可以简化转换函数如下：

$f(i)=o$

逆转换函数

$f^{-1}(i)=o^{-1}$；o^{-1} 表示 o 的逆串。

把o从输入字符串中删除，输入指针左移一位，新的输入字符串为xiy，指针在i；

串转换函数

$f(w)=f(i_0)f(i_1)...f(i_n)=P$

新的输入串为wP，输入指针在P的第一位；

串的逆转换函数

$f^{-1}(w^{-1})=f^{-1}(i_n)f^{-1}(i_{n-1})...f^{-1}(i_0)=P^{-1}$

新的输入串为w，输入指针在w的第一位；

推论

$f(w)^{-1}=f^{-1}(w^{-1})$

设xy是2个串x与y的连接：

$f(xy)=f(x)f(y)=P_xP_y$

$f^{-1}(y^{-1}x^{-1})=P^{-1}{}_yP^{-1}{}_x$

串处理函数δ表示处理到不再产生新的输入为止，Q表示最后除了w剩余的串。

$\delta(w)=Q=f(w)\delta(f(w))=P\delta(P)$

新输入串为Q，指针在Q的末位；

$\delta^{-1}(w^{-1})=f^{-1}(i_n)f^{-1}(i_{n-1})...f^{-1}(i_0)$

每一位按基本定义回推

推论

$\delta^{-1}(w)=f^{-1}(w)$

设xy是2个串x与y的连接：

$\delta(xy)=f(xy)\delta(f(xy))=P_xP_y\delta(P_xP_y)$

$\delta^{-1}(xy)=\delta^{-1}(x)\delta^{-1}(y)$

4.1.4 相关证明

那么要证明回溯的可确定性，只需证明回溯产生的新输入为Q^{-1}，新的输入串为w，即

$\delta^{-1}(Q^{-1}w^{-1})=Q^{-1}$

证明：

$\because \delta^{-1}(w) = f^{-1}(w)$

$\therefore \delta^{-1}(Q^{-1}w^{-1}) = f^{-1}(Q^{-1}w^{-1})$

$\because f^{-1}(w^{-1}) = f(w)^{-1}$

$Q^{-1}w^{-1} = wQ$

$\therefore f^{-1}(Q^{-1}w^{-1}) = f(wQ)^{-1}$

因此要证明 $\delta^{-1}(Q^{-1}w^{-1}) = Q^{-1}$；

需要证明 $f(wQ)^{-1} = Q^{-1}$；

即 $f(wQ) = Q$；

我们来考察 δ 的定义

$\delta(w) = f(w)\delta(f(w))$

可以看出，$\delta(w)$ 是个递归的定义：

把递归展开得到：

$\delta(w) = f(w)\delta(f(w))$

$= f(w)f(f(w))\delta(f(f(w)))$

$= f(w)f(f(w))f(f(f(w)))\delta(f(f(f(w))))$

$= \ldots$

此处为了简化书写，$f^n(w)$ 视作 $f(w)$ 的 n 次递归，如：

$f^0(w) = w$

$f^1(w) = f(w)$

$f^2(w) = f(f(w))$

$f^3(w) = f(f(f(w)))$

等等，依此类推

那么 $\delta(w)$ 的等式可以写作：

$Q = f^1(w)f^2(w)f^3(w)\ldots f^n(w)\delta(f^n(w));(n \to \infty)$

把上式代入等式 $f(wQ) = Q$

可得

$f(wf^1(w)f^2(w)f^3(w)\ldots f^n(w)\delta(f^n(w))) = f^1(w)f^2(w)f^3(w)\ldots f^n(w)\delta(f^n(w))$

由 $f(xy) = f(x)f(y) = P_xP_y$，上式左边可化为：

$f^1(w)f^2(w)f^3(w)\ldots f^n(w)\delta(f^n(w))$

与右边式子相等，由此得证。

由定义可得，δ 是收敛的，即 δ 是有穷的。

那么定然存在一个正整数 n，使得 $\delta(f^n(w)) = \varepsilon$ （ε 表示空串），

即 $\delta(w) = f^1(w)f^2(w)f^3(w)...f^n(w)$

4.2 应用与实现

4.2.1 网络协同

论文中涉及到的863项目中，网络协同化是着重要求的一个环节。所谓网络协同，是指将信息即时发送给所有参予此活动的单位，以避免由于信息传递延迟造成活动中大量的浪费。其中网络协同主要包括两大组成部分：

一、将某单位的操作实时发送给所有参与活动的单位；

二、保证所有单位当前状态的一致性。

我们除了从网络层给予技术支持之外，游戏仿真层也需要作相应的支持。

比如，其一，要将游戏仿真层中信息发送给其他单位，仿真层必须具有将信息序列化到网络流中的能力。

其二，在网络同步中，由于不同的客户端存在延迟上的差异，每个客户端收到消息流的时间不同；此外，不同客户端处理速度可能也各有区别。因此从消息流产生到每个客户端处理完所有的消息，往往存在着较大的时间差异。随着时间的增长，不同客户端上的内容就可以会有较大的不同，从而造成了客户端内容的不一致，即网络同步问题。

而解决同步问题的核心在于，保持所有客户端内容的一致性。通常对于这类由时间即延迟引起的不同步，解决方法有二，一是加快延迟较大的客户端内容处理速度，然而这个方法不适用于延迟明显的网络环境，当时间差异是由于消息在网络中传递造成时，无论如何提高客户端的处理速度都是徒劳的。第二种方法是让延迟小的客户端等待延迟大的客户端。在不清楚其它的客户端当前处理进度的情况下，某特定客户端首先按自己已收到的消息逐步处理。当它知道其它的客户端当前的处理进度后，将它的状态回溯到与处理进度最慢的客户端一致。由此便可以实现所有客户端内容同步。[24]

为了解决这些问题，那么游戏仿真层中二个核心组成部分需要相应的改进。包括状态机部分与消息机制部分。

4.2.2 扩展的状态机

解决网络同步问题的基本策略是状态回溯，前文通过数学模型证明了状态机状态回溯的可行性，下面我们将基于此可行性来考虑其实现。

4.2.2.1 简单状态机与复杂状态机的关系

在第三章中我们提到了，实际仿真层中的对象不能是简单的状态机模型，因为那样会使状态机设计的复杂度大大超越人可以驾驭的范畴。然而在第四章数学模型中讨论的是简单状态机的情况，这是否会使实际仿真层中的状态机对象具有回溯能力？

答案是有此能力的。

正如同第三章中阐述的，复杂状态机是简单状态机的抽象。它在输入处理过程中所作的逻辑判断，实际上是简单状态机中由输入映射到转换函数的非形式化描述。可以这样比方，简单状态机实质上是一组 switch...case 语句，而复杂状态机的输入处理过程是若干组 switch...case 语句的组合，若某些 switch...case 相对简单的话，则直接将其还原为 if...then...else 组合。由此可见，简单状态机的一个重要作用是简化程序中的流程逻辑，但是基于更形式化的方式。当这种形式化的方式描述某些问题显得过于复杂时，我们把它拆解为更好描述的程序逻辑流程。

再从另一个角度来考虑。复杂的状态机对象（即由若干属性组成），以及复杂的消息。它们的这些属性（于消息而言是消息参数），如果是不可再细分的简单数据，则构成了一个简单状态机。因为程序是单一流程的，同一时间只能有一个不可再分的简单消息参数作用于一个不可再分的状态机对象简单属性，那么这个属性的变化完全符合简单状态机模型。

由此，复杂状态机是由简单状态机组成的，它本身也符合简单状态机的性质。那么它就可以拥有回溯能力以及带输出的回溯能力。

4.2.2.2 逆转换能力

在数学模型中，可回溯状态机首先具有逆转换函数，那么我们需要把逆转换函数在程序中实现出来，称之为状态机对象的逆转换能力。

逆转换能力包含两个部分：状态转换和消息回收。逆状态转换与正向转换相应，即将状态回退到相应消息的上一个状态。消息回收是指若在状态转换函数中产生了新的消息，那么在其逆转换函数中需要从消息队列里把此消息收回，即销毁掉，以保证消息队列的正确性。

由于是复杂的状态机对象，转换函数和逆转换函数也不再是简单的条件反射式的跳转，而是复杂的处理过程，分析消息，改变属性，产生新消息，转换状态等。即是说复杂状态机的转换函数是若干简单转换函数的组合，其中可能存在分支使得某些简单转换函数无法执行。其中，使系统状态产生变化的动作有：改变

属性，产生消息，转换状态三种。那么逆转换能力可以更明确的被定义为，恢复属性，回收消息，逆转换状态。

如果状态机被设计为程序中的“对象”，那相应的，状态转换函数则是此对象的某个成员函数，逆转函数也应该是对象的成员函数。状态机对象应该包含两个消息接收入口，一个正向转换入口，一个逆向转换入口。消息分配器根据当前系统状态是前进还是回溯来确定把消息分发给对象的对应入口。每个入口有消息映射表把消息映射到合适的转换函数中去。

4.2.2.3 逆转换状态机的自动生成

在过去的游戏仿真层开发中，通常只定义对象的正向转换状态机。由前文可见，逆向转换状态机是可以由数学方法求得的。那么我们是否要在实现的系统内加入由数学方法自动求出逆状态机的功能，从而大大减低开发者的工作量呢？然而即便理论上是可行的，在现阶段，处于工程能力的考虑，只能局部实现。

目前，在系统中定义的最细粒度是状态机对象，状态机对象包括转换函数、逆转换函数、状态集合等等。虽然是复杂状态机对象，但符合所有可回溯状态机的定义。因此在逆转换函数能被明确定义的情况下，完全可以实现自动生成逆转换状态机。

然而对于更细粒度，即状态机的输入处理过程，它是由若干微小简单状态机组成的。目前它就是一段由开发者手动编写的代码，若要让这些代码也可以自动生成逆转换代码，理论上来说，需要把这些代码划分为对系统状态无改变的普通代码与对系统状态有改变的转换代码，通过分析代码，然后将自动生成相应的逆转代码。这涉及到复杂的编译器功能，因此不纳入现阶段实现的范围中。还有另一种不需要编译器级处理能力的解决方法，将转换代码封装为基本单位，我们称之为代码元，即输入处理过程就是代码元的组合，实现每一类代码元的逆代码，也可以达到自动生成的目的。不过同样的，这也是相当复杂的工程。

4.2.3 扩展的消息机制

在状态机做出改进后，消息机制也要进行相应的改进，才能从整个仿真层上支持网络协同的功能。

4.2.3.1 数据集合序列化

网络协同的组成之一是信息的网络序列化。在仿真层中，只需要给作为信息传递方式的消息加入序列化能即可。第三章中讲过，消息是基于数据集合实现的，

它的目的是为了改善过去种种消息机制存在的缺陷。同样令人惊喜的是，在序列化的能力上，数据集合也远胜于过去种种消息机制。

对于以往的种种消息机制而言，序列化是一项机械而又繁重的工作。因为针对不同的消息格式，都要撰写对应的序列化代码。当消息类型越来越多，序列化代码也会越来越膨胀。因此序列化封装器应运而生。序列化封装器是一种工具类，提供对各种不同数据类型的序列化方法。数据集合本身就是一种序列化封装器，它作为一种异质类型容器，对不同类型的进行统一的封装。在封装不同类型时即可将此类型的序列化方法加入其中。与普通的序列化封装器不同的是，由于数据集合实现了不同类型数据的统一容器管理，那么它只需要一个序列化方法就可以序列化所有容器内的数据。而以往的消息机制仍然需要对某一具体类型的消息使用序列化封装器进行序列化代码的编写。

在反序列化，即由二进制流转换到实际数据结构时，数据集合由于其独到的数据类型检查机制，使得其安全性大大增加，而不会出现过去用消息 A 的反序列化方法去反序列化消息 B 的二进制数据流产生的错误。

4.2.3.2 全局消息队列

网络协同需要系统能够返回到过去时间的任意状态。为此有两种方法，一是记录所有时间系统的状态，二是记录所有时间系统中的消息。其中方法一是不现实的。因为记录系统状态意味着巨大的存储资源开销。相反，记录消息的开销则要小很多，一是消息本身只包含系统变化的信息，二是可能多个对象共享同一条消息。

那么在消息机制中，除了原有的消息消息队列（待处理）外，还要包括一条处理过的消息队列，或者把二条队列合为一条，通过当前指针来管理。状态机对象便可以基于消息队列跳转到任意一个时间点的状态去。由此，网络中的同步机制可以有多级策略选择，比如选择网络延迟最大的客户端来跳转到相应时间点的状态上，或者是某个标杆型客户端来跳转，抑或是全程回溯等。

当系统的运行时间到达一定程度时，消息队列可能会大到不适合继续全部保存在内存中。因此我们必须考虑将消息队列局部保存到硬盘的策略。目前采用的方法是设置消息队列的最大长度，当消息队列的长度超出时，便把队尾的消息写入到硬盘消息队列文件的末尾。此处同样也需要用到消息的序列化能力，区别在于是序列化到硬盘而不是网络数据流。当消息队列回溯到为空时，便从硬盘中取出不大于最大长度的消息继续回溯。

4.2.3.3 消息排序

由于某些缘故，如网络延迟，或者系统设定的需要延时处理，有些消息进入消息队列的顺序不可能完美的按照产生的时间。为了解决这个问题，可以在输入中加入一个时间戳，来表示输入的产生时间。在管理输入队列的时候，采用按时间戳排序的方法，便可以让输入的先后次序正确。

时间戳的另一个用途在于作为回溯时的标志，如指定系统回复到某一个时间的状态，只需要给出一个时间点，系统自动查找到最接近的输入位置，并回复到当时的状态。这样较之指定回溯到某个输入更为便利，因为输入队列有多条，给出时间可以在每一条队列中找到合适的位置。当然也保留直接回溯到某个输入的功能，其实现无非是根据该输入的时间再在其他输入队列中查找相应的位置而已。

第五章 总结与展望

至此，我们已经完成了整个游戏仿真层框架的设计。

通过对框架中的各个部件的合理运用，可以轻松的实现游戏中对现实世界的仿真。当然，游戏逻辑本身存在自己的复杂性，仍需要开发者自己去掌握。此框架的目的在于，让游戏开发者能够尽可能的规避程序结构上的复杂性，从而能够把更多的精力放到游戏本身内在逻辑的复杂度中去。这也是863项目网络游戏开发平台的意义之所在。

但是这套框架目前尚只能应用于C++代码的开发。对于通常的网络游戏而言，后期维护的过程几乎等量于前期的开发，而后期维护一个主要组成部分是增加的新的游戏逻辑内容。如果仍然采用C++代码的开发方式，必然会大大的降低开发效率。为了能够及时的添加新内容，必须要采用更有效率的方法，比如说脚本。我们的架构中没有能够加入脚本的确是个遗憾。

不过脚本技术本身是个复杂的工作，需要相当长时间的积累。我们在长期计划中确定了对脚本技术的开发。并且我们在设计中为将来脚本化进行了的充分考虑。因此我想，这个缺陷会在未来得到弥补。

最后我们再来展望一下未来的游戏仿真层将会如何发展。

如同我们在第二章中所分析的，除了计算机学科的理论之外，我认为越来越多其他学科的内容会被引入进来。因为这已经是对计算机中虚拟世界的本质的研究。那么可以预见的是，游戏仿真的研究会逐步扩展到对于计算机世界本质的研究，让我们能更深刻地认识到计算机世界的内部，从而对计算机技术的发展起到不可磨灭的重大推动作用。

致谢

在论文即将完成之际，我首先要感谢我的导师陈雷霆教授。感谢陈老师在我攻读硕士期间在学术上给予我的悉心指导，陈老师一丝不苟的工作作风、严谨的治学态度和对工作的无私奉献精神，深深地影响了我，使我受益匪浅。在此，谨向近三年来培养、帮助和关怀我的陈老师致以由衷的敬意和诚挚的谢意。

我也特别感谢数字媒体研究所的何明耘老师、蔡洪斌老师、房春兰老师、白忠建老师和邱航老师。感谢他们在工作和学习上给予我无私的帮助和支持。

同时我也要感谢数字媒体研究所游戏引擎小组的所有同学，有各位同学的协作和共同努力，我们才能如此顺利地完成项目。

参考文献

1. André.LaMothe, Building Brains into Your Games, www.gamedev.net, 1995.
2. Andrew. Rollings, Dave. Morris, Game Architecture and Design, Beijing, Beijing Hope Electronic Press, 2003, 243-256.
3. Andrew. Rollings, Ernest. Adams, Game Design, Beijing, Beijing Hope Electronic Press, 2003, 42-54.
4. Aristotle the Stagirite, Metaphysics, Beijing,, The Commercial Press, 2003, 1-84.
5. Bertrand. Russell, a History of Western Philosophy, Beijing, The Commercial Press, 1945, 117-278.
6. Brownlee. Jason, A Finite State Machine Framework, http://ai-depot.com/FiniteStateMachines/FSM.html, 2000.
7. Cantor, 集合论, 数学杂志, 1874.
8. Charlie. Kindle, Grady. Booch, Essential COM, 中国电力出版社, 2001, 30-78.
9. Fang-Chang Lin, Ssu-Chi Chiu, A Less Message Mechanism for Improving, http://csu.edu.au/~mantolov/CD/...papers, 2002.
10. Choe. Y. K, Agrawal. D. P, Green. C. R, A Hierarchical Message Mechanism For Distributed Systems Software, Distributed Computing Systems in the 1990s, 1988. Proceedings., Workshop on the Future Trends of, 1988, 132-138.
11. Emmanuel. Roche, Finite-State Language Processing. Cambridge, Massachusetts, USA, MIT Press, 1997, 24-37.
12. Erich. Gamma, Richard. Helm, Ralph. Johnson, John. Vlissides, Design Pattern, Element Of Reusable Object-oriented Software, Beijing, China Machine Press, 2002, 127-134.
13. Geoff. Howland. A Practical Guide to Building a Complete Game AI, www.gamedev.net, 2002.
14. Georg. Wilhelm. Priedrich. Hegel, 逻辑学, 北京, 商务印书馆, 2003, 23-42.
15. Gibson. D, Finite State Machines - Making simple work of complex functions, http://www.microconsultants.com/tips/fsm/fsmartcl.htm.
16. John. E. Hopcroft, Rajeev. Motwani, Jeffrey. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 北京, 机械工业出版社, 2005, 25-55.

17. Keith. Fieldhouse, Introduction to Lua,http://www.onlamp.com/, 2006.
18. L. Von. Bertalanffy, General System Theory; Foundations, Development, Applications, George Braziller, 1968, 13-24.
19. Doug. Lea, Run-Time Type Information and Class Design: Annotations to Stroustrup & Lenkov, http://g.oswego.edu/dl/rtti/rtti.html, 1997.
20. Maguire. Steve, Writing Solid Code, Microsoft Press, 1993, 58-66.
21. Maguire. Steve, Debugging the Development Process, Microsoft Press, 1994, 27-44.
22. McConnell. Steve, Software Project Survival Guide, Microsoft Press, 1997, 134-182.
23. Nelson. Mark, Priority Queues and the STL, http://www.dogma.net/markn/articles/pq_stl/priority.htm, 1996.
24. P.Reed.D, Teatime Framework Design, Hewlett-Packard Labs & MIT Media Lab, 2005, 1-12.
25. Shari. Lawrence. Pfleeger, Software Engeering, Theory and Practice, Beijing, Higher Education Press, 2001, 45-76.
26. Plato, The Republic, Beijing,, The Commercial Press, 2003, 177-272
27. Prof. Dr. Peter Brucker, Juniorprof. Dr. Sigrid Knust, Complexity results for scheduling problems, http://www.mathematik.uni-osnabrueck.de/research/OR/class/, 2006.
28. Michael. O. Rabin, Complexity of Computations, ACM Turing Award Lectures——The First 20Years：1966—1985，ACM Pr., 1977, 47-62.
29. Roberto. Lerusalimschy. Programming in Lua, Lua.org, 2003, 8-12.
30. Dana. S. Scott, 10Sic and Programming Language, ACM Turing Award Lectures——The First 20Years：1966—1985，ACM Pr., 1977, 319-338.
31. Sheff. David, Game Over, Cyberactive Media Group, 1999, 3-16.
32. Jack Stankovic, Real-Time Systems: Scheduling, http://www.cs.virginia.edu/~stankovic/sched.html, 2000.
33. Vsrajeshvs, Singleton Pattern & its implementation with C++, http://www.codeproject.com/, 2002.
34. Deborah. Wallach, Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation, http://mit.edu/~kerr/papers, 1995.
35. Woodcock. Steve, Game AI: The State of the Industry, Game Developer, August 1999, 32-44.

个人简历及硕士期间发表的论文

个人简历

胡浩源，1983 年生，男，四川绵阳人。2004 年 9 月毕业于电子科技大学计算机科学与工程学院，获工学学士学位。于 2004 年 9 月进入电子科技大学计算机科学与工程学院攻读工学硕士学位。主要研究方向为：数字媒体相关，大规模软件工程。

发表论文

胡浩源，陈雷霆等，《可回溯有限状态机的研究与应用》，计算机应用研究(已录用).