基于多 Agent 系统的游戏 AI 引擎

A Game AI Engine Based on Multi-Agent System

学科专业：计算机应用技术
研 究 生：付恒
指导教师：冯志勇 教授

天津大学计算机科学与技术学院
二零零七年一月

摘 要

在当前的电脑游戏中，图形质量的发展已经到了近乎极至的水平，人工智能已经成为决定游戏成功的重要因素，越来越多的游戏开发者和研究者开始将重点转移到游戏中的人工智能研究。由于游戏 AI 与具体的游戏设计紧密相关，目前还没有出现被广泛使用的 AI 引擎。本文针对团体竞技类型的游戏，研究和开发了一个基于多 Agent 系统的游戏 AI 引擎，主要工作包括：

根据游戏中存在大量自主角色的特点和游戏软件对架构灵活性的要求，设计了游戏 AI 引擎中的多 Agent 系统模型，包括个体 Agent 的感知系统模型、运动系统模型和决策系统模型，以及多 Agent 的通讯系统模型和团队协作模型。以有限状态机技术为支持，集成了消息处理机制，实现了 AI 引擎的隐式决策树架构和算法封装。在此基础上，将足球相关知识和技能集成进来，增加了游戏控制、输入处理、图形渲染、实体管理等实际的游戏需要的必要功能，实现了一个完整的足球模拟游戏，以验证该 AI 引擎的有效性。

在利用该 AI 引擎构建的足球模拟游戏中，Agent（球员）可以灵敏地感知比赛形势并做出正确合理的动作，多 Agent 之间也可以进行良好的协作来完成同一任务，表现出令人满意的智能效果。

综上所述，本文将多 Agent 系统用于游戏 AI 引擎设计，实现了完整的引擎和基于该引擎的足球模拟游戏，是构建通用游戏 AI 引擎的初步尝试。我们相信，利用该引擎，开发者可以简单快速地实现游戏中的人工智能架构和自主的智能角色。

关键词： 游戏引擎 多 Agent 系统 人工智能

ABSTRACT

Until recently, technology in games was driven by a desire to achieve real-time, photo-realistic graphics. To a large extent, this has now been achieved. As game developers look for new and innovative technologies to drive games development, AI is coming to the fore. Because of the dependence on game design, there are not so much popular AI engines as graphical engines in game developing. In this paper, a game AI engine suits team-sports game is developed based on multi-agent system.

The Multi-Agent system in game AI engine is modeled, according to the special requirements of game engine. The models includes perceptive system model, movement system model and decision-making system model of single agent and communication system model and collaboration model of multi-agent. Based on a Finite State Machine, the architecture of the AI engine is established, integrating message handling. Further more, an entire soccer-simulating game including input control, graphical render and other aspects is developed based on our game AI engine.

In our game, the intelligence of the agents is satisfied, as well as the collaboration among the agents. In this paper, the effect of using multi-agent model in the design of game AI engine is validated and the experiment is a successful attempt to develop a common game AI engine. We believe that using our game AI engine, developers could develop the AI in games efficiently.

KEY WORDS： Game Engine, Multi-Agent System, Artificial Intelligence

目　　录

第一章 绪论

1.1 研究背景

自 1961 年首套电子游戏《宇宙战争》发行以来的四十多年的时间里，电子游戏的发展速度让全世界都为之震惊，如今，电子游戏已经成为一个产业，对许多国家乃至世界的经济都有举足轻重的影响[1]。

2000 年中国的网络游戏销售额仅为 0.38 亿元，2001 年即达 3.25 亿元，而 2002 年中国网络游戏市场规模达到 10.2 亿元，增长率为 213.8%。到 2007 年中国网络游戏用户数将达到 4180 万，从 2002 年到 2007 年这 5 年的年复合增长率将达到 37.8%，届时网络游戏用户将占到 Internet 用户的 29.5%（图 1-1）。据新闻出版总署公布的《2004 年中国游戏市场报告》显示，2004 年我国网络游戏市场的总体规模为 24.7 亿元人民币，较 2003 年增长了 47.9%。并且预计到 2009 年中我国网络游戏市场的销售收入将达到 109.6 亿元人民币，从 2004 至 2009 年的年复合增长率将达到 34.7%。而《2005 年中国游戏市场报告》显示 2005 年我国网络游戏用户总数达到 2634 万，较 2004 年增长了 30.1%。报告还显示从 2006 年到 2010 年的 5 年间我国网游用户的复合增长率为 13.7%，超过同期网民数量的增长[2]。

可见，游戏产业在国内的发展形势蒸蒸日上，随着政策的不断优化和技术的进步，国内的游戏产业还会有更进一步的发展，游戏产业，已经成为国内新一代的一大朝阳产业。

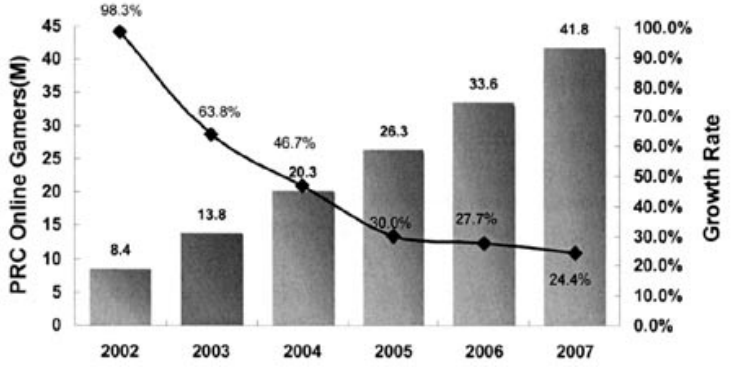

图 1-1 中国网络游戏用户数及增长

市场的发展必然促进技术的发展，短短几年前，游戏开发还被多数计算机工作者视为“不务正业”，而如今，游戏开发技术已经成为软件开发者所追逐的目标，经验丰富的游戏开发人员也成为业界新星。

70 年代的游戏，如《Pong》，仅仅用几个象素或线条表示物体，运动的规则也非常简单。80 年代，随着专用游戏机的出现和硬件能力的提高，游戏的表现力有了显著的提高，代表作品有《超级马里奥》等。到 90 年代，三维图形技术被广泛应用于电脑游戏，游戏表现力进一步提升。如今，随着专用三维图形硬件的发展，游戏的图形质量已经到了近乎极至的地步。从图 1-2 中可以看到不同时期游戏画面的巨大差异[1]。

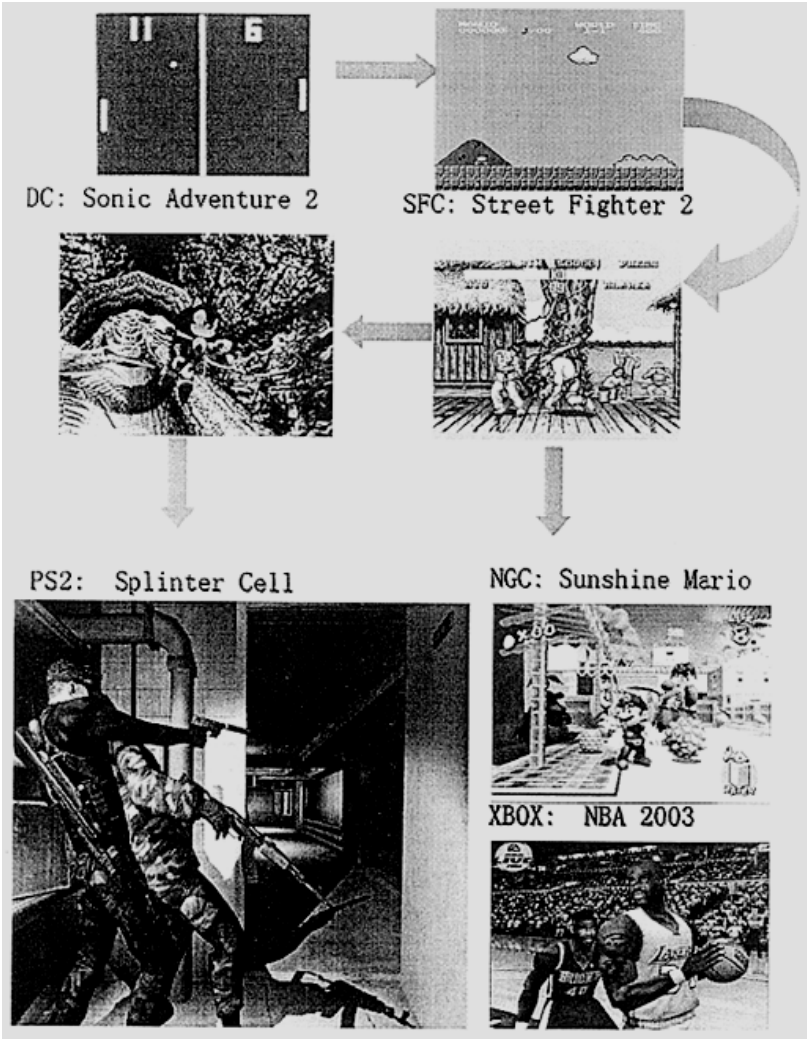

图 1-2 游戏画面的进化

在相当长的一段时间里，游戏开发技术的发展就是图形技术的发展。1992 年，3D Realms 公司/Apogee 公司发布了一款只有 2 兆多的小游戏——《德军司令部》（Wolfenstein 3D），首次实现了具有 3D 效果的画面。1994 年，一个叫做 ID 的游戏公司做出了 PC 历史上第一个 FPS 对战游戏 DOOM，DOOM 开创了 3D 游戏的新时代，其画面效果让所有玩家为之震撼。仅在 1994 年当年，DOOM 就获得了一系列令人羡慕的殊荣，这其中包括：欧洲计算机贸易展示会年度最佳电脑游戏、交互艺术与科学协会年度最佳动作冒险游戏、PC Gamer 年度最佳游戏；

Computer Gaming World 年度最佳游戏、PC Magazine 技术卓越奖和其它大大小小的奖项。伟大的游戏 DOOM 不仅仅开创了 3D 游戏的时代，也开创了一种新的游戏开发模式[3]，基于引擎的游戏开发。

1.2 游戏引擎技术

无论是 2D 游戏还是 3D 游戏，无论是角色扮演游戏、即时策略游戏、冒险解谜游戏或是动作射击游戏，都有很多相似功能的代码，它们在每个游戏中完成大致相似的工作，图像和声音的渲染、用户输入的接收和处理、物理运动的模拟等。渐渐地，一些有经验的开发者摸索出了一条“偷懒”的方法，他们借用上一款类似题材游戏中的部分代码作为新游戏的基本框架，以节省开发时间和开发费用。这部分可以在不同游戏中重复使用的代码，就是游戏引擎的雏形。

可以把游戏的引擎比作赛车的引擎，大家知道，引擎是赛车的心脏，决定着赛车的性能和稳定性，赛车的速度、操纵感这些直接与车手相关的指标都是建立在引擎的基础上的。游戏也是如此，玩家所体验到的剧情、关卡、美工、音乐、操作等内容都是由游戏的引擎直接控制的，它扮演着中场发动机的角色，把游戏中的所有元素捆绑在一起，在后台指挥它们同时、有序地工作。简单地说，引擎就是“用于控制所有游戏功能的主程序，从计算碰撞、物理系统和物体的相对位置，到接受玩家的输入，以及按照正确的音量输出声音等等”[3]。

事实上，游戏引擎的出现是软件工程在游戏开发领域的发展，游戏引擎实际上是一种“中间件”，它使得游戏的开发过程变得更加简单和快速，开发者的分工也更加明确。游戏引擎是模块化软件开发的典型，它将游戏中常用的图形、声音、物理、人工智能等方面的代码独立出来，成为一个通用的核心模块和软件框架。基于已有的引擎开发游戏的成本要比完全重新开发一个游戏的成本低的多，这种游戏开发方式已经成为当今主流的游戏开发方式。经过不断的进化，如今的游戏引擎已经发展为一套由多个子系统共同构成的复杂系统，从建模、动画，到光影、粒子特效;从物理系统、碰撞检测，到文件管理、网络特性;还有专业的编辑工具和插件，几乎涵盖了开发过程中所有重要环节。

但是，在 3D 游戏的开发过程中，引擎的制作往往会占用非常多的时间，游戏《马科斯·佩恩》的 MAX-FX 引擎从最初的雏形 Final Reality 到最终的成品共花了四年多的时间，而 LithTech 引擎的开发共花了整整五年时间，耗资 700 万美元。出于节约成本、缩短周期和降低风险这三方面的考虑，越来越多的开发者倾向于使用第三方的现成引擎制作自己的游戏，一个庞大的引擎授权市场已经形成。

ID Software 公司的游戏《雷神之锤 2》使用的 Quake II 引擎一举确定了自己在 3D 引擎市场上的霸主地位。Quake II 引擎可以更充分地利用 3D 加速和 OpenGL 技术，在图像和网络方面与前作相比有了质的飞跃，Raven 公司的《异教徒 2（Heretic II）》和《军事冒险家（Soldier of Fortune）》、Ritual 公司的《原罪（Sin）》、Xatrix 娱乐公司的《首脑：犯罪生涯（Kingpin：Life of Crime）》以及离子风暴工作室去年夏天刚刚发布的《安纳克朗诺克斯（Anachronox）》都采用了 Quake II 引擎。

EPIC 公司出品的 Unreal II（国内称为“虚幻”引擎）是游戏行业中最成功、应用游戏最广泛的、最昂贵（单个产品授权超过 60 万美金）的商业引擎[4]。从最早的《虚幻》、《虚幻》资料片《重返纳帕利》，第三人称动作游戏《北欧神符》（Rune）、角色扮演游戏《杀出重围》（Deus Ex）以及《永远的毁灭公爵》，到近期的《勇者无惧：越战》，Unreal II 被广泛使用在 FPS 游戏上。随着网络游戏的 3D 化，很多网络游戏也开始使用 Unreal II 制作，例如《凯旋》、《天堂 2》、《创世纪 10 网络版：奥德赛》、《铁血三国志》等等。

这些引擎大部分都是以出色的图像渲染为卖点的，辅助一些便于开发的工具和程序框架。而游戏不仅仅是图形图像上的视觉表现，可以做到引擎里的通用部分也不单单是图形。随着游戏的功能越来越多，游戏引擎也变的越来越强大和庞大，依靠一个单独的引擎实现游戏中所有的方面开始变得困难，渐渐地，原来单一的引擎开始分化不同的组成部分，如图形引擎、3D 引擎、声音引擎、物理引擎、网络引擎等，游戏开发者可以根据自身的需要选择不同的模块进行组合。这一点在开源社区表现的尤为突出，例如最为著名的 3D 引擎 OGRE[5]，在一开始的目标也是一个功能较为完善的图形渲染引擎，如今已经抛弃其他方面，专门集中在 3D 图形渲染上；专门做图形用户界面的 CEGUI[6]，在游戏中的用户界面上表现突出，可以和 OGRE 很好的结合；高度仿真的数学物理引擎 ODE[7]；以及由 CORBA 专家领军的网络通讯引擎 ICE [8]，是一个稳定、高效和成熟的网络中间件。这些都是被广泛使用的开源引擎，甚至有一些商业游戏也开始使用它们。

可以看到，基于引擎的游戏开发已经成为目前一种可靠的开发模式，其中图形引擎技术已经相当成熟和稳定，而作为游戏中另外一个重要因素的人工智能，相应的引擎技术还很不成熟。

基于以上的讨论，本文设计了一个用于团体竞赛类游戏的 AI 引擎。第二章讨论了游戏中的人工智能的特点以及相关的研究与目前的状况。第三章设计了 AI 引擎中的多 Agent 系统模型。第四章研究了基于多 Agent 系统模型的游戏 AI 引擎的设计与实现。第五章论述了将具体的游戏与 AI 引擎相集成的方法。第六章是一个基于我们的 AI 引擎的完整的游戏的实现。

第二章 游戏中的人工智能与 AI 引擎

迄今为止，游戏开发中的技术提升主要都在图像表现方面，游戏总是追求实时的真实感图像效果，在广义上说，基于目前的硬件水平和技术，已经可以达到这个目标了[9]。游戏的图像发展到《虚幻》已经达到了一个天花板的高度，接下去的发展方向很明显不可能再朝着视觉方面进行下去[2]。至少在图形图像的质量上，已经不会有像《DOOM》刚出现时那种巨大的飞跃式的进步了。

基于这样的现实，智能性表现已经成为影响一个游戏成功的越来越重要的因素。长期以来不被游戏开发者重视的人工智能已经从默默无闻上升到了业界新星的地位，一场智能游戏的变革正在发生[10]。

2.1 游戏中的人工智能

1956 年，约翰·麦卡锡（John McCarthy）提出了“人工智能（AI）”的概念，被认为是人工智能诞生的标志。此后的十多年时间里，这个新兴的领域取得了很多的进展，特别是在定理的证明和推导以及符号计算方面。在六十年代末到七十年代初，人工智能在机器翻译等方面的应用遇到困难，相关研究进入低潮。随后，专家系统的出现使得人工智能的研究趋于理性。到八十年代中期，神经元网络在许多应用上的成功使得人工智能的研究出现复苏[11]。

学术界所谓的人工智能涵盖了大量不同的领域和分支，如专家系统、有限状态机、产生式系统、多 Agent 系统、遗传算法、神经网络、模糊逻辑等。具有智能表现的游戏都在不同的场合或多或少的使用了这些技术，当然他们获得的效果也各不相同。

在电子游戏刚刚出现的时候，如 Pong、Pac-Man、Space Invaders 等游戏中，使用了大量的简单规则、动作序列的描述以及随即决策等技术来使游戏的发展变得难以预测[10]。这可以认为是人工智能技术在游戏中最早的应用。

在很长的一段时间内，学术界喜欢利用棋类游戏来进行人工智能的研究。从早期的西洋跳棋程序一直到击败了特级大师加里·卡斯帕罗夫的国际象棋程序“深蓝”[12]，都是人工智能用于游戏的例子。

然而，使用人工智能技术开发一个游戏与使用游戏来研究人工智能技术有着很大的区别。在游戏开发中，现代视频游戏环境的复杂程度和游戏机制决定了它们无法像“深蓝”那样采用穷举搜索游戏树的策略。取而代之，现在的游戏多采

用了其他的搜索技术，如 A*算法几乎在所有类型的游戏中都是最佳的搜索算法[13,14]。

策略游戏是较为广泛的采用人工智能技术的一类游戏，智能性是这种游戏的卖点之一。这种游戏与棋类游戏有很大不同，它不是一个由电脑控制的难以击败玩家，而是由电脑在游戏世界中模拟出具有一定智能表现的虚拟人物，这种智能被称为对象级（unit-level）的人工智能。这种游戏的一个典型就是《Civilization》[15]。

近来出现的一些即时战略（RTS）游戏使用了更为先进的人工智能技术。如《WarCraft》与《Age of Empire 2》，都具有强大的 RTS 智能技术，可以实时为游戏中的数百个对象进行路径搜索，也就是游戏中常说的寻径（pathfinding）。

在很多类型的游戏，特别是 RTS 游戏中，寻径都是人工智能成功与否的关键。关于游戏中的寻径，有大量的文献和智能算法[16,17]，我们前面说到的 A*算法就是用来解决寻径问题最为有效的算法。

在第一人称射击游戏中，Valve Software 公司的《Half-Life》实现了卓越的战术智能，而 Epic Games 公司的 Unreal 引擎以其高度扩展性和高超的策略而闻名。Looking Glass 开发的《Thief》游戏逼真的模拟了虚拟角色的感知能力，并采用分级报警技术使玩家能感受到虚拟角色的智能水平。Sierra 的《SWAT3》游戏很好的表现了虚拟人物的动作和交互方式，并且采用了随机化的智能行为[10]。

在模拟类游戏 SimCity 中，成功采用了人工生命（A-Life）技术，在此基础上，游戏 The Sims 中使用了具有不同个性的智能体（Agent）技术。这款游戏的流行证明了有限状态机与人工生命技术的巨大潜力[18]。

另外，著名的 RoboCup （Robot World Cup）[19]机器人世界杯组织是学术界为了鼓励人工智能和智能机器人技术研究于 1995 年在瑞士成立的一个非赢利性组织，它提供了一个展示和评估各种智能技术的平台，它的终极目标是：到 2050 年的时候，构建一个机器人球队，能够战胜人类世界杯冠军球队。

RoboCup 自创建以来，每一年都有很多大学和研究机构构建的机器人足球队参赛，在这里，由不同理论和技术实现的机器人球队可以同场竞技，各种人工智能技术可以得到充分的试验和展示。最为著名的卡耐基-梅隆大学的球队 CMUnited[20]使用了一种分层的机器学习技术，使得球员可以在一个共同目标的约束下自动学习如何进行协作。在 1999 年与 2000 年取得季军的球队 Essex Wizards[21]，使用了多线程技术[22]和 Q 学习方法[23]。2000 年的欧洲冠军和世界冠军 FC Portugal[24]在 CMUnited 的基础上实现了一个良好的策略机制。另外还有很多采用各种技术的球队如 AT Humboldt[25]、YowAI[26]等。在国内，RoboCup 联合会中国分会于 1999 年成立，著名的参赛院校有浙江大学、中国科技大学、清华

大学等[27]，其中清华大学的 TsinghuAeolus 球队在 2001 年和 2002 都获得了世界冠军[28]。

可以发现，游戏中的人工智能和主流的人工智能理论研究在目标上并不完全相同。人工智能理论研究的目的在于解决极端困难的问题，比如模拟人的认知或者理解自然语言。相比之下，游戏中的人工智能是为了更多的增加游戏的趣味性。因此，在开发游戏中的人工智能的时候，要把注意力放在那些智能角色的表现上，而不能陶醉于人工智能技术本身[10]。

与人工智理论研究相比，游戏中的人工智似乎能显得有点简单。实际上，如果看一下传统的游戏开发的特点就可以理解这一点[9]：

1) 更少的 CPU 资源：在游戏中，人工智能的地位一直都在图形渲染之下，大部分的 CPU 资源都让给图形渲染，以尽可能提高游戏的图像质量，人工智能技术总是在不影响图形质量的前提下进行设计和开发，这限制了大计算量技术的使用。
2) 相关理论的适用性：很多不确定方法，如神经网络和遗传算法，它们在实验室内可以做出很好的结果，然而对于游戏这样一个庞大的工程来说，很难找到一个切实可行的方法来推广这些应用。另外，从这些方法产生的效果与其需要的大量计算和开发投入来说，在很长的一段时间内并没有展示出决定性的优势。
3) 较少的开发时间和人力：多年来，游戏开发者总是为 AI 分配较少的开发时间，一般来说，游戏中的 AI 部分都是在其他部分完成之后的时间开发的。对图形质量的一味追求使得游戏开发者忽略了包括 AI 在内的其他方面。

于是，在商业游戏开发中，人工智能技术总是倾向于采用简单的、易于实现的和确定的方法与技术。从理论性讲，游戏中的人工智能似乎显得较为简单，然而从具体的应用与游戏的要求来说，游戏中的人工智能方法都是切实可行的、有效的方法。

在 2000 年的游戏开发者大会（GDC）的 AI 圆桌会议上，展现出了今年来游戏开发者对游戏中人工智能的态度转变，从图 2-1 可以看出在游戏中 AI 的所占的比重以及开发者对游戏 AI 的重视程度相比前几年都有很大的提高。这次会议将游戏中的人工智能推向了一个新的高度[29]。

之后，关于游戏人工智能的研究逐渐升温，更多的开发者开始尝试将一些不确定方法引入游戏开发。John Manslow 给出了一个在游戏中使用神经网络的简单示例[30]；Thomas Rolfes 更是别出心裁，利用图形处理器 GPU 的编程来处理人工神经网络[31]，这已经可以说明游戏中 AI 的地位已经超越了图形渲染。关于游戏

中使用神经网络的研究还有 Andre [32]与 Alex J.Champandard [33]。Francois 则尝试了在游戏 AI 中使用遗传算法[34]，另外，还有一些对游戏中使用学习理论的研究[35, 36, 37]，EA 公司的一款非常成功的商业游戏《Black & White》[38]就是使用机器学习技术的典范。

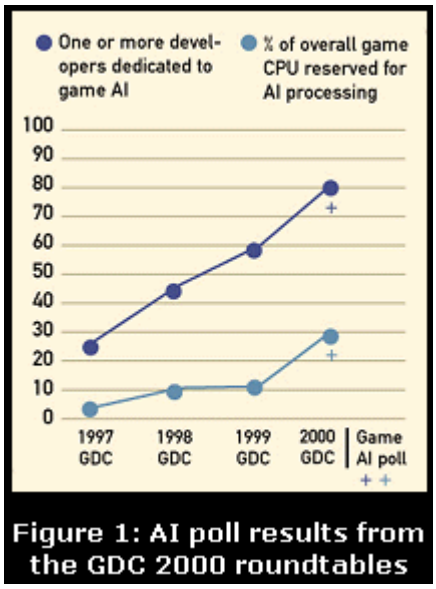

图 2-1 GDC2000 上 AI 调查结果

2.2 游戏 AI 引擎

从第一章我们知道，游戏中的通用特性往往被做成了各种游戏引擎，既然已经有这么多关于游戏中人工智能的研究，那么为什么没有流行的 AI 引擎呢？

实际上，在 2000 年的游戏开发者大会（GDC2000）上，一个被广泛讨论的话题就是游戏 AI 引擎的适用性[29]。当前也确实已经存在几种面向游戏 AI 的引擎（或游戏 AI SDK），它们是：

1) DirectAI 使用有限状态机构建了具有一定智能行为的智能体框架。
2) Spark 这是一个基于模糊逻辑的开发包，提供一些编辑工具。
3) OpenAI 是一个开源的游戏 AI 开发包，提供了很多智能算法的支持，如神经网络与遗传算法等。

然而在 GDC2000 这些引擎并没有得到多少好评，多数开发者认为它们不能适合自己所开发的游戏，DirectAI 的固有模式使得它难以用于具体的游戏开发，SparK 与 OpenAI 提供的通用智能算法在具体的游戏中显得效率低下，不利于进行针对性的优化。在讨论中，更多开发者更希望 AI 引擎仅仅支持最为基本的功能，如运动支持与寻径就可以了。多样的游戏设计使得以上的 AI 引擎显得不够

灵活，对于开发者来说，一个 AI 引擎或 SDK 的灵活性比它强大的功能更重要。

另外，密歇根大学人工智能实验室开发了一个名为 Soar 的智能引擎[39]，Soar 实际上是一个接口系统，将具体的知识库与原有的游戏引擎如 Quake II 连接起来，从而增强了游戏的智能，并使得各种智能算法可以方便地添加到游戏中。Soar 引擎的目的并不是实现一个用于游戏开发的 AI 引擎，而是为了利用 QuakeII 引擎来试验一些智能算法。

这些现有的“AI 引擎”，要么是提供一些零散的算法支持（这实际上是 SDK），要么不具有足够的灵活性来支持不同的具体游戏。实际上，一个智能系统是难以使用于各种场合的，更何况新的任务总是随着游戏开发而不断出现[40]，更好的做法是实现一个基本的智能系统来提供常用的动作支持，复杂的游戏相关的任务在具体的游戏开发中使用动作脚本调用基本动作来完成。

2.3 多 Agent 系统用于游戏 AI 引擎

人工智能的研究进入九十年代后，研究者开始重新审视“完整智能体”的问题。Agent（智能体）是处在某个环境中的计算机系统，该系统有能力在这个环境中自主行动以实现其设计目标[41]。利用多个 Agent 构建的协同工作的系统称为多 Agent 系统，多 Agent 系统已经成为人工智能研究的一种思考方式，更倾向于实现类似人类的智能。

Agent 并不是一个固定的概念，任何可以感知环境并做出反应的东西都可以被看作是一个 Agent[11]。事实上，Agent 可以具有一些特定领域的知识和技能，并且在设计时往往被赋予一定的使命或目标。在多 Agent 系统中，多个 Agent 往往具有相同的目标，它们通过相互合作来达到这个目标。多 Agent 的系统的这些特点使得它在用于构建复杂系统时具有独特的优势：

1) 提供并行计算的能力。复杂的系统可以被划分为不同的子系统交给不同的 Agent 来并行处理。
2) 高度的鲁棒性。在多个 Agent 构建的系统中，Agent 直接的耦合非常小，且 Agent 通过消息来通讯，使得系统灵活性非常高，一个 Agent 出现问题一般不会影响到其他的 Agent。
3) 简单的编程模型。基于 Agent 的程序设计层次清晰，分工明确，易于编程实现。
4) 多 Agent 系统的可扩展性。可以方便的向多 Agent 系统增加新的 Agent 以增强系统功能，而需要对原有系统做过多改动。

在团体竞技的游戏中，人工智能要负责维护由多个 NPC（Non-Player Character

非玩家角色）组成的队伍，来与玩家或其他电脑维护的队伍进行竞技或者合作。显然，多 Agent 系统的上述特性非常适合用于对此类游戏的人工智能进行建模。久负盛名的游戏《SimCity》就成功地采用了多 Agent 系统的思想[10]。

综上所述，使用多 Agent 系统构建游戏 AI 引擎，可以方便地实现游戏中的自主角色，同时也能得到较为灵活的软件架构。然而对于游戏软件，执行效率是一个至关重要的因素，这使得使用效率上并不占优势的多 Agent 系统来构建游戏 AI 引擎成为一个挑战。本文针对团体竞技类型的游戏，设计了高效的 Agent 个体模型和多 Agent 通讯模型，并使用了如延迟消息与 AI 规整器等优化策略来提高引擎的执行效率。

第三章 多 Agent 系统建模

3.1 个体 Agent 模型结构

在本文的 AI 引擎中，提供对应用广泛的对象级（unit-level）智能的支持，将游戏中的每个具有智能的对象视为一个智能 Agent，需要为单个 Agent 建模，并提供基本的智能行为支持。这样，在具体的游戏中就可以方便的创建一个具有一定智能的游戏对象。

Agent 可以分为纯反应式 Agent 和有状态的 Agent[41]，更详细地，可以分为简单反射型 Agent、基于模型的反射型 Agent、基于目标的反射型 Agent 和基于效用的 Agent[11]。其中简单反射型 Agent 直接对感知信息做出反应，而基于模型的反射型 Agent 保持内部状态，追踪记录当前感知信息中不明显的世界各方面。基于目标的 Agent 的行动是为了达到目标，而基于效用的 Agent 试图最大化它们自己期望的“快乐”。

根据游戏的特点分析一下这几种 Agent 模型。其中，简单反射型 Agent 仅基于当前的感知选择自己的行动，而不会记忆历史。这种 Agent 模型简单，但是基于这种模型的 Agent 的智能相当有限，对于游戏来说，这类似于 80 年代游戏中普遍采用的简单规则和运动模式系统，基于这种方法的 Agent 在游戏中表现出来的动作较为机械，很容易使玩家掌握其动作规律和模式，从而轻松的击败电脑。这种 Agent 模型难以表现出令人满意的智能效果。

基于模型的反射型 Agent 在内部保持当前无法感知的历史信息，从而获取比简单反射型 Agent 更丰富的信息。用这种模型构造出来的游戏 Agent 具有更高的智能，可以在复杂的环境中做出合理的决策。由于对历史状态的维护，这种 Agent 具有一定的记忆性。在游戏具有明确的目标和任务的时候，这种 Agent 的表现令人满意。但是用这种模型构建一个完全自主的游戏角色时，很多情况 Agent 会面临多个类似的选择，或者说根据当前环境和历史发展并不能唯一确定一个明确的动作，Agent 就无法合理确定进一步的选择，因为这种 Agent 对自身的目标和价值没有意识。

游戏的要求需要让 Agent 知道自己想要达到的目标，在一个完全自主的环境中可以明确自己的任务，有目的进行下一步动作，而不是单纯的根据环境做出反映。基于目标的 Agent 符合这个要求，这种模型会把目标信息与可能的动作结果联合起来考虑，从而选择达到目标的最佳动作。例如，在后面创建的足球游戏中，

Agent 要选择一个动作来处理球，它不仅需要知道当前的比赛形式，还要了解它的动作对比赛形式产生的影响，也就是需要了解它自己和团队的目标，这样它才能做出一个正确的动作，例如快速发起进攻或是消极的浪费时间。

一个基于目标的 Agent 结构如图 3-1 所示[11]：

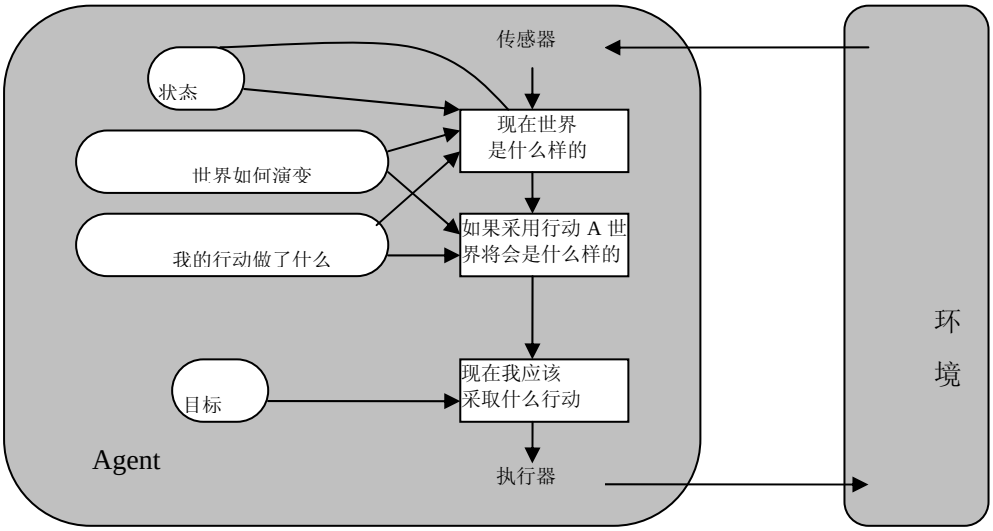

图 3-1 基于目标的 Agent 结构

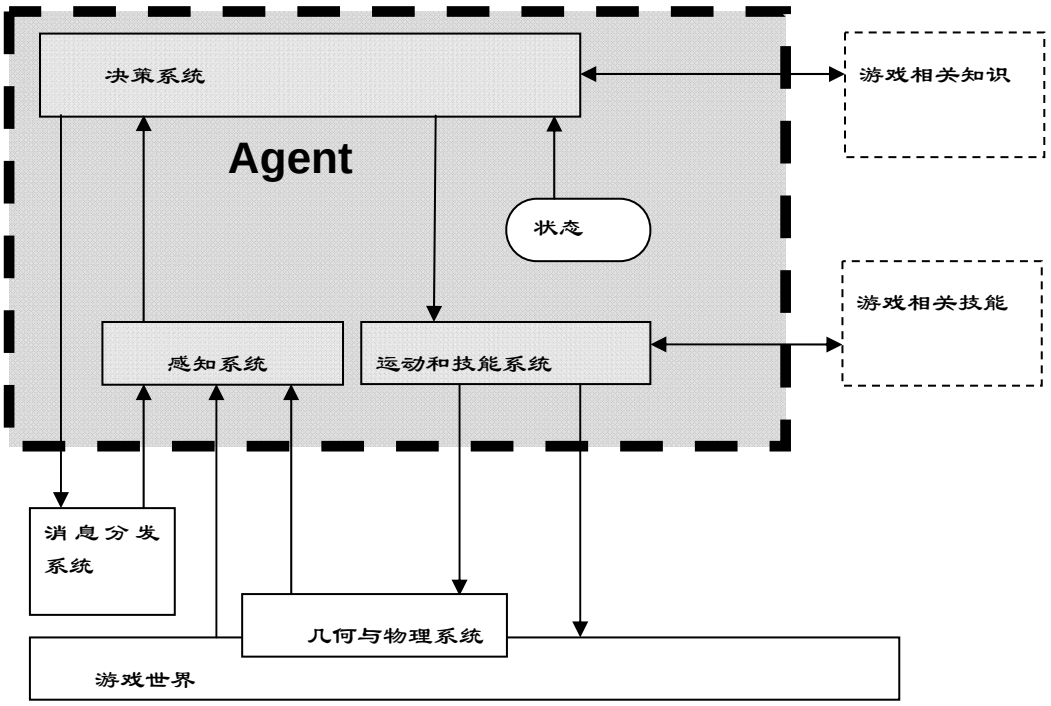

图 3-2 AI 引擎中的 Agent 结构

图 3-1 突出了 Agent 的决策过程，表现出一个基于目标的 Agent 是怎样“思

考”的。这种过程是一种高度抽象的、不确定的过程，其中的部分在应用中都会被具体的操作所代替，可以把这里表现出来的决策过程作为编写 Agent 决策部分的指导思想。从软件设计和开发者的角度，如果把决策的细节封装到一个决策系统中，再把感知系统和运动系统明确的表示出来，就得到我们 AI 引擎中的 Agent 结构模型（图 3-2）。

在游戏中，一个 Agent 所处的环境并不是真实的物理环境，而是模拟出来的游戏世界，基本游戏世界的模型由引擎提供，并且实现一定的几何与物理效果，即为 Agent 的环境提供可感知性。Agent 通过感知系统来感知游戏世界，并维护自身状态，同时记录历史信息。决策系统利用感知的信息和自身的状态信息分析当前游戏世界的状态，确定游戏的发展方向，并结合自身被设定的目标来决策出一系列动作，然后由运动系统来执行这些动作，从而可以改变自身的状态和游戏世界的状态。在我们的 Agent 模型中，Agent 还可以从消息系统接受消息来作为决策的依据之一，消息可以来自其他的 Agent 或游戏世界甚至是 Agent 群体，以完成多 Agent 之间的协作。

一般来说，在不同的游戏中，Agent 需要完成不同的任务，即使在同一个游戏中，也存在各种不同类型的 NPC，也就是 Agent，他们的任务也千差万别。不可能把各种智能行为都统一到一个 Agent 模型中，一个合理的游戏引擎，既要提供对游戏开发的足够支持，又要保持相当高的灵活性，所以，应该在 Agent 模型中提供基本智能决策的架构和一些通用智能算法，同时提供对游戏相关知识和游戏相关技能集成的支持，在开发具体的游戏时，可以编写相应的游戏相关知识来改变 Agent 的决策方式，以及编写游戏相关的技能来增强 Agent 的智能行为。

3.2 环境特征

根据 Agent 的定义：Agent 是处在某个环境中的计算机系统，该系统有能力在这个环境中自主行动以实现其设计目标[41]。

Agent 与其所处的环境是不可分割的，一个智能体表现好坏取决于其所在环境的本性[11]。要想构建一个良好的 Agent 模型，必须明确其所处的环境具有哪些特征。

Russell 和 Norvig 对 Agent 所处的环境做出如下分类[41]：可观察的与不可观察的、确定性的与非确定性的、静态与动态的、离散的与连续的。

根据这种分类，可以对游戏中 Agent 所处的环境的特点做如下分析：

可观察性。在游戏中，Agent 所处的环境不是真实的环境，而是由计算机模拟出来的游戏世界。一般来说，将游戏世界设计为不可观察的是没有意义的，因

此，对于可观察与不可观察性来说，游戏世界总是被认为是可观察的，并且，游戏世界本身会提供一些支持来使得 Agent 对它的观察更加容易。

确定性。同样，由于游戏的虚拟的特点，动作对环境的影响也是通过计算模拟出的，显然，可以容易的确定动作的结果。在游戏中，为了增加真实性，很多时候也会人为地添加一些随机因素，使得一些结果不那么确定，但从 Agent 所处的环境特点来说，它显然还是确定的。

静态与动态。一般来说，游戏中游戏世界会按照客观规律模拟变化，如黑夜和白昼的变换和四季交替等。除了这种变化之外，能对环境产生影响的就是玩家控制的角色和 NPC。这样，除了当前关注的 Agent（一般是游戏中的 NPC）以外，游戏世界也会改变某些状态和信息，也就是说，在游戏中，Agent 所处的环境是动态的。

离散的与连续的。对于棋类游戏，一般来说只有有限数量的状态，虽然状态的数量可能非常大，也可以认为它是离散的环境。但是对于 RPG 游戏和和本文要实现的群体竞技类型的游戏来说，都是在模拟一个真实的连续环境，所以理论上并不把它作为离散的环境来对待，然而由于环境本身是由计算机模拟的，这与真实的环境还存在很大的差别，实际上，这为 Agent 的设计提供了一种便利，使得 Agent 可以更容易得获得关于环境的更多更准确的信息。

综上所述，在游戏中，Agent 所处的环境是可观察的、确定的、动态的和连续的。依据这些特征，就可以来构建 Agent 的各个系统了。

3.3 感知系统

感知系统是 Agent 了解环境的方式。Agent 所做的一切决策依赖于从感知系统获得的信息，获取足够的正确信息是 Agent 做出正确决策的必要条件。从游戏角度说，良好的感知系统有助于产生更好的智能表现。

在构建一个机器人 Agent 的时候，感知系统一般是位于机器人身上的摄像机或者一系列传感器；在软件 Agent 中，感知系统一般是获得环境信息的一系列系统命令。而在游戏中，由于 Agent 一般表现为一些精灵、怪兽或人物，这样，就要通过软件的方式模拟出类似于生物或人的各种感觉器官的功能。一般来说，游戏中使用最为广泛的感觉模型就是视觉，当然有些游戏也使用了听觉或触觉模型。

3.3.1 视觉模型

在具有一定智能表现的游戏中，几乎都能找到视觉感知上的表现。在具有良

好视觉模型的游戏中，对手不像老式的游戏中那样可以随时知道玩家的动向，玩家可以有意识地远离敌人从而不被发现，甚至可以绕到敌人背后进行突袭或暗杀。显然，良好视觉模型给游戏增加了更多的趣味。

实际上，构建一个视觉模型是非常简单的，在我们的引擎中，一个 Agent 的视觉模型可以用如下三元组来描述：

$$\left\{ \vec{h}, \alpha, d \right\}$$

其中 $\vec{h}$ 是一个二维向量，表示当前 Agent 所面对的方向，也就是 Agent 的“眼睛”的朝向；α 表示当前 Agent 的视角，任意时刻，Agent 只能“看”到 $\vec{h}$ 方向左右各 $\alpha/2$ 弧度内的其他物体；d 表示 Agent 的可视距离，即任意时刻，Agent 智能“看”到距离它在 d 内的其他物体。另外，由于物体之间的相互遮挡，还需要从 Agent 视野中排除被遮挡的物体。图 3-3 展示了我们引擎中的 Agent 的视觉模型。

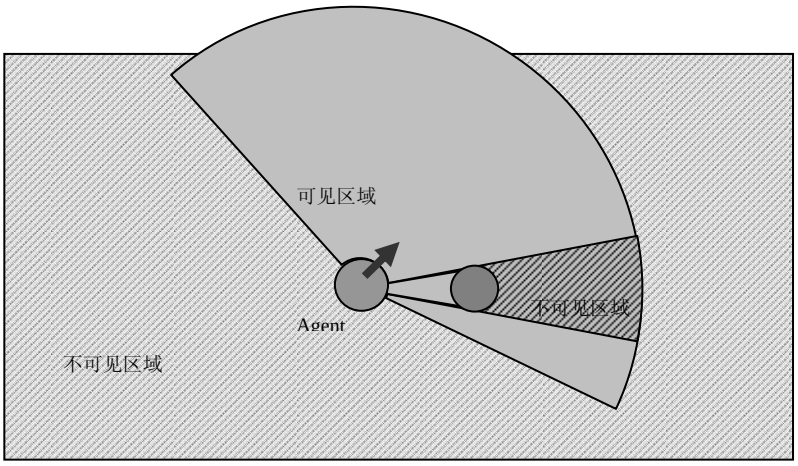

图 3-3 AI 引擎中的 Agent 视觉模型

3.3.2 听觉模型与消息感知

由于听觉模型的应用远不如视觉模型广泛，在我们的引擎中，没有单独实现听觉模型，简单地，可以通过消息机制实现听觉的效果。

在我们的引擎中，将消息的接收作为感知系统的一部分，使得 Agent 能够感知来自环境或其他 Agent 的消息。在利用消息机制实现听觉效果时，只需要对声音波及范围内的所有 Agent 发送一个特定的声音消息就可以了。关于消息处理的详细内容，在本文 3.6 节消息通讯种叙述。

3.4 运动系统

一个 Agent 想要影响其所在的环境，必须对环境做出某些动作。在游戏中，Agent 的动作包括基本的移动以及与具体游戏相关的特定动作序列，动作的结果通过游戏画面表现出来，直接反映了 Agent 的智能性，是影响游戏可玩性的重要因素。因此，需要将运动系统设计的尽量真实，在引擎中，给出了一系列游戏中常见动作的封装，对于游戏相关的特定动作和技能，可以方便的通过引擎提供的基本动作的组合来实现。

3.4.1 基本动作模型

可以方便地采用牛顿力学作为运动理论，来实现基本的物理上的移动。在我们引擎中，使用以下五元组来描述一个 Agent 的运动状态：

$$\{ \vec{p}, \vec{v}, \vec{a}, \omega, \beta \}$$

其中 $\vec{p}$ 是一个二维向量，表示在二维空间中 Agent 的坐标位置，$\vec{v}$ 和 $\vec{a}$ 也都是二维向量，分别表示 Agent 当前的速度和加速度，ω 和 β 则表示 Agent 当前的角速度和角加速度。根据运动学方程，在引擎中实现一些的基本运动接口，例如向前跑、停止、加速、减速、以某种速度向前跑、向左转、向右转等等。由于实现基于牛顿力学的运动细节不是本文重点，关于游戏中的物理模拟更详细的介绍可以参见其他资料[42]。

3.4.2 运动行为模型

基本的动作模型仅仅提供了实现运动的一些支持，在很多情况下，游戏开发者希望仅仅发出一个运动的命令，Agent 就能按照预定的想法去运动，而不需要去关心运动的所有细节，更不希望去计算牛顿方程得到力和加速度等等。这样，就需要更高一层的抽象来描述 Agent 的运动，可以使用运动行为模型[43]来封装对运动细节的计算。运动行为模型包含一系列的运动模式，我们的引擎中实现了以下几种：

Seek 模式 Seek 模式是很简单的一种模式，它是当一个 Agent 向某个目标位置移动时采用的动作，它计算一个使 Agent 向某个目标运动的驱动力（图 3-4）。

Arrive 模式 Seek 模式以及可以使 Agent 正确地移动到目标点，然而可以发现在使用 Seek 模式的时候，当 Agent 在到达目标点时速度突然变为零，这种表现在游戏中显得很不优雅。我们希望找到一种方法，能使得 Agent 平稳的到达目标点，也就是希望 Agent 在快要到达目标点的时候就开始减速，然后平稳的停在

目标位置。Arrive 模式在 Seek 模式的基础上增加了减速的计算，使得 Agent 移动起来更加优雅（图 3-5）。

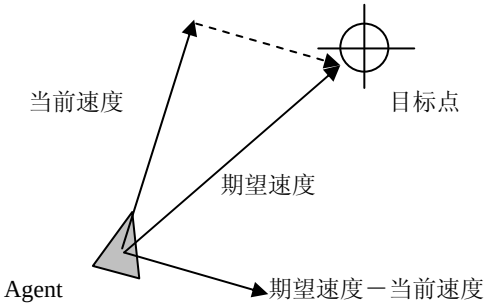

图 3-4 Seek 运动模式

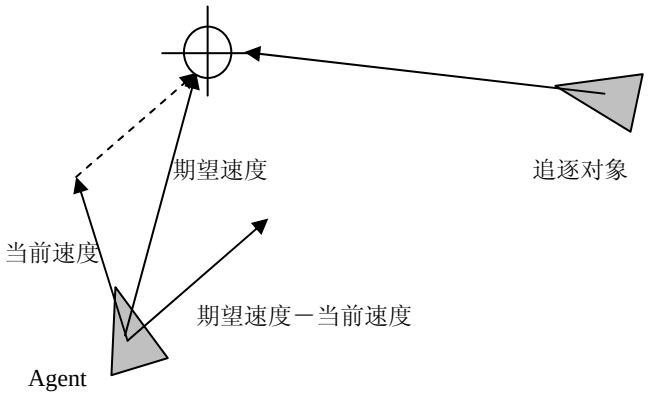

图 3-5 Pursuit 运动模式

Pursuit 模式 在一个 Agent 希望有效的追逐一个运动中的其它 Agent 或物体时，应该使用 Pursuit 运动模式，这种模式可以根据被追逐对象的速度和追逐者的速度计算出一个最佳的目标位置，追逐者朝向这个目标位置运动便能有效的追逐被追逐者。

Obstacle Avoidance 模式 在游戏中，对于碰撞检测的计算几乎是必不可少的，实际上，碰撞检测是一个游戏物理引擎的主要工作[44]。碰撞检测模拟的是物理世界的运动规律，对于运动中的 Agent 来说，它与一个运动的无生命的物体有很大不同，一个聪明的 Agent 当然不会像一块飞行的木头一样对即将发生的碰撞视而不见，相反，Agent 应该能够聪明地避免碰撞，这样，利用物理引擎的碰撞检测来处理 Agent 运动中的碰撞，显然无法实现这种效果。Obstacle Avoidance 就是这样一种主动避免碰撞的运动模式，在这种模式下，Agent 会在碰撞到障碍物之前主动的改变运动方向以达到避免碰撞的效果。这也是 Agent 智能性的一个表现（图 3-6）。

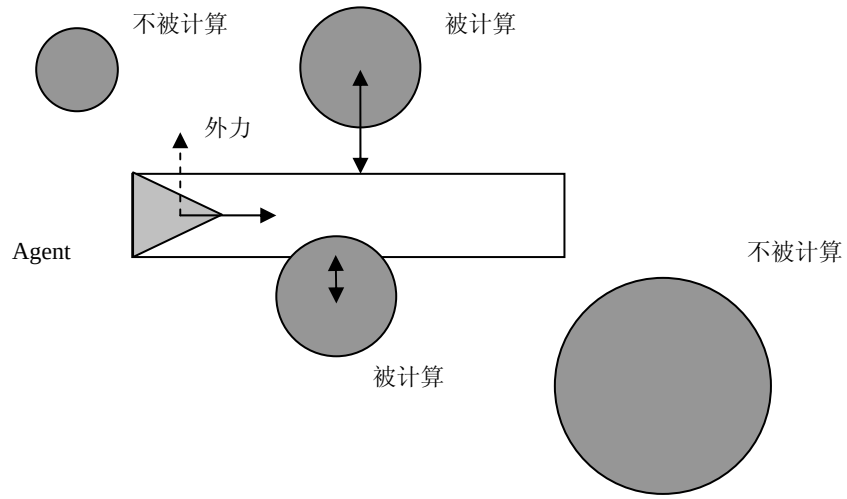

图 3-6 Obstacle Avoidance 模式

如图 3-6 所示，Obstacle Avoidance 模式会检测位于 Agent 前方一定范围内的物体，计算他们是否会与 Agent 相碰撞，根据计算结果，给出一个驱使 Agent 避免碰撞的外力，从而改变 Agent 的运动方向。

运动行为模型的使用使得具体的运动细节得到很好的封装，让 Agent 的运动具有更加智能的表现，也让游戏相关的代码变得更加简洁。在我们的引擎中实现了这些基本的运动模型，这样，在具体的游戏中，就可以方便地基于这些基本的运动接口和模式来实现更为复杂的动作。

3.5 决策系统

现在已经为 Agent 构建了能够感知世界信息的感知系统，也构建了使 Agent 有能力对世界做出反应的运动系统。要使 Agent 能够根据周围环境的变化做出自己的反应，我们还需要为它构建一个“大脑”，也就是 Agent 的决策系统。

一般来说，一个 Agent 的决策系统都是根据 Agent 的存在目标有针对性的设计的，可是在游戏 AI 引擎中，由于具体的游戏千变万化，不能假设 Agent 的目标。也就是说，引擎中决策系统设计的难点在于为游戏开发提供足够的灵活性，使得具体的游戏可以针对自己的特点做充分的扩展。因此，在引擎中，不能使用固定的、显式的决策系统，而应该提供一种隐式的决策结构，以便于具体游戏的开发。

在基于多 Agent 系统的 AI 引擎中，一个决策系统的结构大致是如图 3-7 所示的样子。决策算法的输入应该至少考虑三部分内容：

1) 环境。当前的世界是什么样的，这是 Agent 做出决策的最基本的条件。

2) 个体倾向。当前 Agent 的个性特点和动作倾向，这是体现每个 Agent 自

身特点的必要条件，否则，所有 Agent 将变得完全一致，这样开发出来的游戏也就没什么意思了。

3) 团队协作。当前的 Agent 团队对协作的要求，这是实现多 Agent 协作的基础，Agent 应该具有一定的团队意识，协调个体与团队利益。

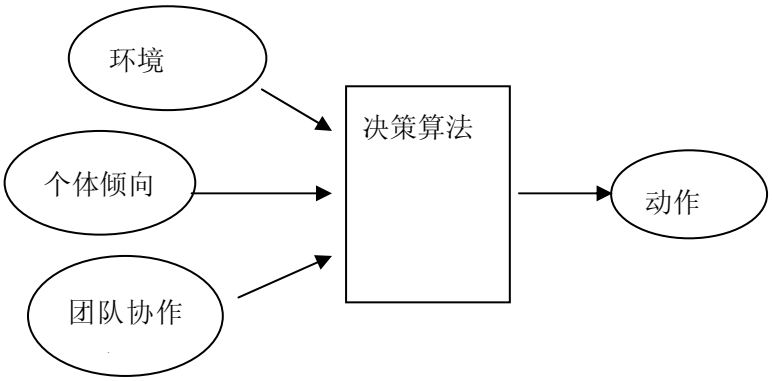

图 3-7 AI 引擎中的决策系统结构

在一个具体的游戏中，无论使用什么决策算法，基本的游戏规则总是存在的。也就是说，基于规则的决策树不能避免。但是引擎中又无法得知具体游戏的决策树结构，那么如何提供对决策系统的支持呢？

可以通过在引擎中实现一个有限状态机框架来支持具体游戏的决策逻辑。状态模式[45]建议应该使用对象而不是函数的方式来实现 AI 的行为或状态，这样，就允许游戏开发者创建能容纳任意多行为的隐式决策树而不用修改框架中的代码。关于我们引擎中有限状态机的具体设计和实现在 4.2 节：集成消息处理的有限状态机。

从图 3-7 可以看到，决策算法的输出是一个动作，实际上，这并不是指由决策算法来生成一个动作，而是指决策算法在当前 Agent 支持的动作集合中选择出一个合适的动作，有时候还要为这个动作提供一些执行参数。那么，决策问题就是确定哪一个动作是“最好的”动作。

在决策理论中，一个传统定义“最好”的方法是最大期望值。一个动作的期望值是它的平均代价。数学上，一个动作的数学期望定义为，动作可能产生的所有后果乘上各自对应的概率并累加[46]：

$$E(V)=\sum_{i} p_i r_i \qquad \text{公式（3-1）}$$

根据公式（3-1），可以针对每个动作编写一个计算其期望值（称为效用值）的函数。该效用函数和具体的游戏有关，在我们的引擎中要做的是提供一些相关的计算支持函数和依据效用值选择动作的框架，一般来说，可以选择效用值最大

的动作。

值得注意的是，在游戏中，应该允许 Agent 偶然做出不是最优的甚至是错误的选择，这样才会使得 Agent 更接近现实中的人，使游戏更加有趣。要实现这种效果的方法就是将效用值看作概率，也就是说具有较高效用值的动作具有较高的被选择的概率，反之亦然。同时，还应该能够调节选择最优动作的概率，以使得 Agent 表现出不同的“聪明”程度。在我们的 AI 引擎中，设计了一个动作选择器来实现上述功能，详细介绍参见 4.3 节。

3.6 消息通讯系统

尽管已经为 Agent 构建了感知系统、决策系统和运动系统，基于这些子系统，已经能够构建出一个智能 Agent，它可以通过自己对世界的观察来做出合理的动作。仅仅是这样的话，在多 Agent 的环境中，Agent 还只是把其他 Agent 作为一个客观的个体，而不是和自己一样有决策能力的智能体，也就是说，现有的 Agent 还没有能力与其他的 Agent 合作。因此，需要构建一个系统来使得 Agent 之间可以相互通讯，也就是构建一个多 Agent 合作的基础。

在我们的引擎中，使用一个消息分发器来实行 Agent 的通讯，Agent 不仅可以通过消息分发器与其他 Agent 进行消息发送，消息分发器也作为 Agent 群体和游戏世界与个体 Agent 交互的一种方式（图 3-8）。

在说明消息分发器的具体设计之前，先来仔细分析一下消息分发器在游戏中的重要作用。在游戏中，每帧内各种游戏对象都要进行逻辑更新，实际上游戏对象在游戏世界中更新有两种基本方式：通过积极地观察世界（论询），或者通过坐等消息到来（事件驱动）[47]。直观上，让每个 Agent 去感知世界从而获得其感兴趣的信息可能更符合现实状况，然而在游戏中，让每个游戏对象在每帧内检查每一种需要处理的事件是否发生却不是一个好的编程方式。例如，在某个游戏中，游戏中的 Agent 需要对发生在它周围的爆炸做出反应，按照论询的方法，每个 Agent 需要在每一帧内检查它周围的爆炸事件，而对于大多数 Agent 来说，可能直到整个游戏结束都不会经历一次爆炸。随着 Agent 需要处理的事件数目的增加，它每帧中需要检测的事件就越来越多，这时，论询方法就变得臃肿而低效。

尽管不是那么直观，使用消息机制的事件驱动方式确实是一个好的解决方案。在设计 Agent 感知系统时说过，使用消息方式，可以方便的实现 Agent 的听觉，这将使得 Agent 表现的更加智能。在事件驱动的模型中，每个 Agent 只需要针对感兴趣的消息编写处理代码并等待消息到来，而不用在更新函数中不停的检查各种事件的发生，这使得代码变得非常清晰易读。

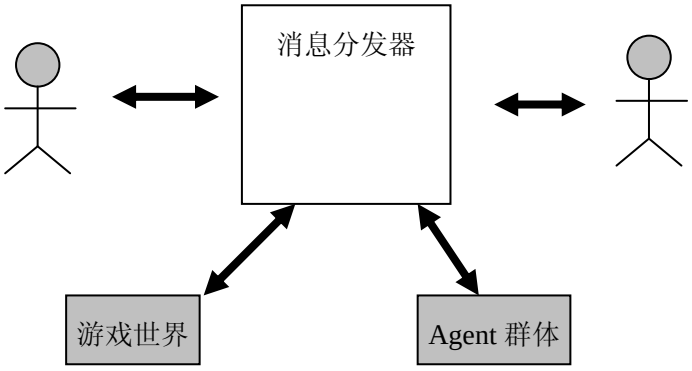

图 3-8 Agent 消息通讯模型

基于以上的讨论，先来确定一个消息的概念。一个消息应该至少含有如下几个域：

消息名字（消息 ID）、发送者、接受者、发送时间、该消息的特定参数。

以爆炸消息为例，一个爆炸消息的内容应该是这样的：

消息 ID：爆炸（整型 ID）；发送者：炸弹（对象指针）；接受者：Agent（对象指针）；发送时间（浮点数）；消息参数（void 指针）：爆炸地点、破坏力等。

消息的参数为 void 指针，因为每个消息都有各不相同的参数，具体的消息处理代码可以确定参数的类型和意义。

在我们的引擎中，消息分为两种，即时消息和定时消息（延迟消息）。即时消息是在发送者发送消息后马上转发给接收者的消息，从发送到接受的延迟不会超过一帧的时间。而定时消息则是指定在一定时间之后才发送到接受者的消息。即时消息可以马上发送，而对于定时消息来说，由于每次更新消息分发器都要确定哪些消息是要发送的，这就需要一个良好的设计来使得消息的查询更为快捷。

在消息分发器中，维护一个定时消息的排序链表，在这个链表中，消息按照其预期的被发送时间排序。这样，在每次更新时，消息分发器只要顺序取出要发送的消息并发送就可以了，当有新的消息到来时，也只需要将消息插入到这个有序链表中。

在我们的引擎中，为了方便使用，消息分发器被设计为一个单件[45]。

3.7 多 Agent 协作

有了消息分发器这样一个通讯的基础，并不代表多 Agent 系统就能良好的进行合作了。事实上，任何形式的群体组织，无论是一群蜜蜂，一群狼，一个运动

团体，或是一支军队，都需要对组织中的每个成员进行协调。每个成员在组织中都扮演一定的角色，每个角色完成一个特定的底层行为。一项高层的目标，通常需要不同的底层行为的配合，这种配合通常是隐式的。除了自然于社会组织外，上述结论对于计算机游戏中各种模拟 Agent 群体同样成立。当系统中 Agent 的数目随着潜在的角色数量而增加，如何控制以及协商他们之间的行为就成为一个难点。

黑板体系结构[48]给出了协调不同 Agent 之间行为的一种方法。它不但易于实现，而且实践证明，这种体系结构能够有效的解决各类协同问题，包括人工角色控制、自然语言识别以及各种推理问题。实际上，从 20 世纪 70 年代黑板体系结构提出的那一天开始，它就被作为规则推理和问题解决的一种技术，一直发展到今天。

一般来说，黑板体系结构包括黑板、知识源和仲裁器几个部分。其中，黑板（Black Board ）是一块公共的可读/写信息区。通常，它包含一栏逻辑表达式，作为体系结构中其他部分运行的依据。这些逻辑表达式占据一定的空间，他们可以简单的排列在一起，但是更多的时候，是通过某种作用在表达式空间上的结果进行组织，这种组织结构使得当需要访问这些虚拟黑板上的表达式时，只需要考虑黑板的特定区域，就能获得感兴趣的内容。知识源（KS）是围绕在黑板周围的一系列组件，根据黑板蕴涵的信息，做出相应的动作。可以认为这些知识源是协同工作，可以解决一个特定问题的专家。知识源可以用多种形式改变黑板的内容，比如增加新的逻辑表达式，修改现有的逻辑表达式或者发送控制信号给其他知识源。给定一组黑板内容，可以存在不止一个知识源具有相关性。对此，仲裁器（Arbiter）需要做出判断，以决定哪一个知识源对黑板进行操作。对于整个黑板体系结构而言，仲裁器的重要性是不言而喻的，它具有全部的控制策略。

黑板体系结构并不是一个固定的模式，它可以依据不同的应用有着不同的实现。在大部分早期的系统中，仲裁器以如下方式工作，选择其中一个知识源，保证在任意时刻，不会有两个相抵触的动作被同时执行。选择具有最高相关性的知识源最简单的一种选择策略。高级的仲裁器甚至可以根据总体目标，计算相关的期望值，而不仅仅通过知识源返回的相关性值进行判断。

黑板体系结构的这些特征使得它非常适合处理多 Agent 系统中的团队协作问题[49]。，特别的，对于游戏开发，在 Improv 动画教本系统[50]中，“事件黑板”被用于发送关于演员状态及其行为的信息。如果一个演员讲了一个笑话，系统将在黑板上布置“讲了一个笑话”的标记，以便于其他的演员做出响应的反映。演员根据自己的个性和对讲笑话人的态度，将做出大笑或者嗤之以鼻等表情或动作。

在我们的引擎中，使用了一个类似于黑板的体系结构（图 3-9），所不同的是，

在引擎中，仲裁器是被具体的游戏隐式地实现的，这是由于仲裁的依据和决策方式都与具体的游戏紧密相关。从图 3-9 中可以看出，知识源由游戏中的 Agent 充当，每个 Agent 具有各自不同的技能，例如执行某种预定战术的能力，可以用熟练程度来描述 Agent 对每种技能的掌握，也作为团队合作选择参与者的依据。

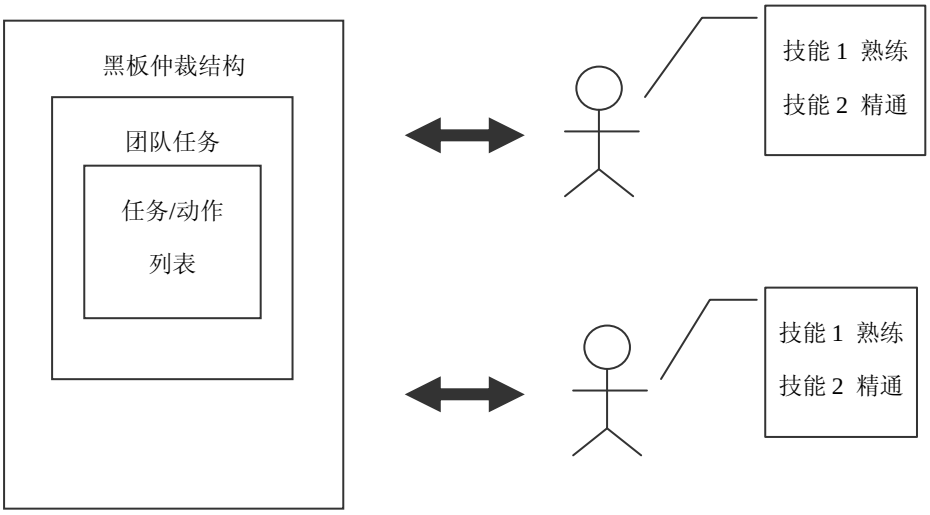

图 3-9 引擎中的黑板体系结构

在特定的游戏中，一个 Agent 群体或者群体种的某一个 Agent 对当前的环境进行分析并做出发动一次团队合作的决策。它会将具体的任务布置在黑板上，每个 Agent 看到黑板上的任务后判断自身能否参与合作并在黑板上留下个人信息，Agent 群体选择出合适的执行者并根据每个 Agent 的特点确定任务参数，然后将其布置到黑板上。最后，每个 Agent 从黑板得到自己需要完成的任务和参数。这样，一次协同性的合作任务就被部署到每个 Agent 身上。

3.8 本章小结

本章设计了基于多 Agent 系统的 AI 引擎中的 Agent 模型，包括感知系统、运动系统、决策系统以及多 Agent 通讯与协作系统模型，基于这种模型，就可以来构建一个具体的 AI 引擎了。

第四章 构建 AI 引擎

4.1 基本类关系

前面介绍的 Agent 模型可以用一个类 GameAgent 来封装，另外，还要增加一些类构建一个完整的游戏 AI 引擎， 图 4-1 表示了该引擎中的几个比较核心的类的关系。

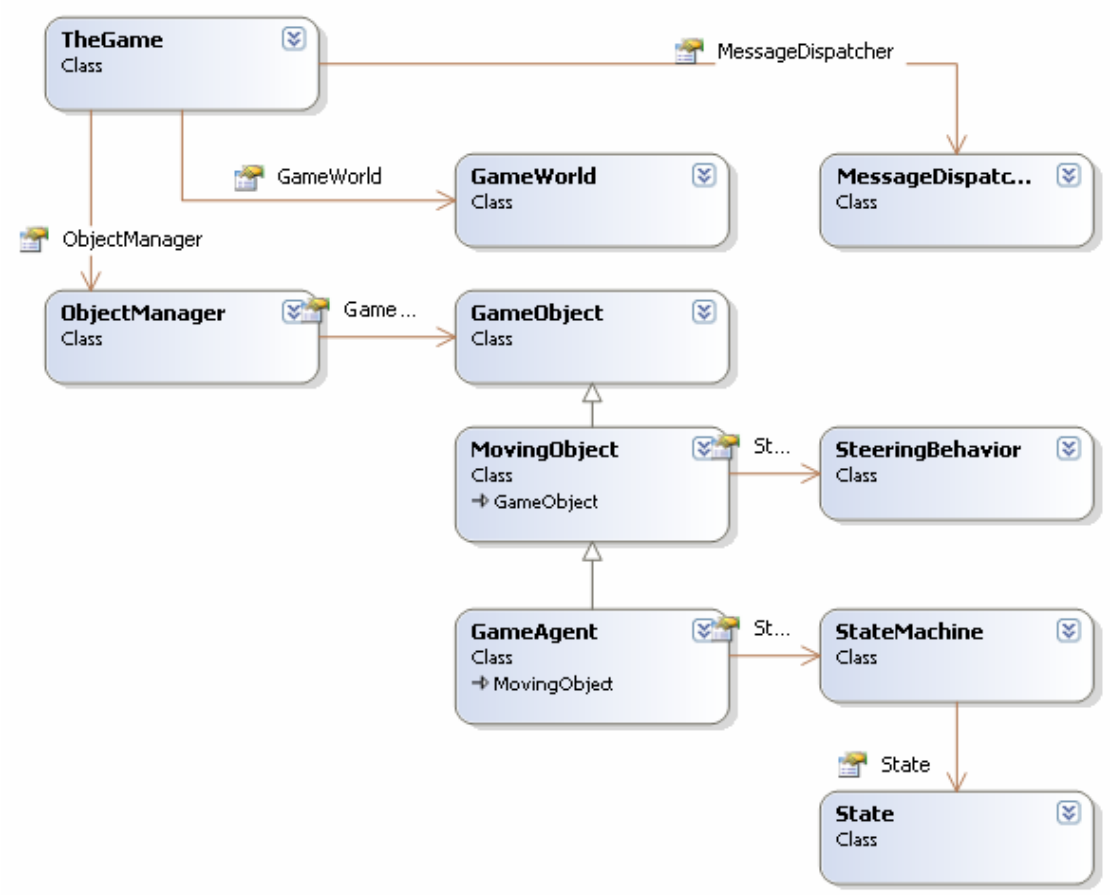

图 4-1 AI 引擎中 Agent 相关类的关系图

从图 4-1 中可以看到，类 TheGame 是描述整个游戏的类，具体的游戏可以从 TheGame 派生以利用已经实现的功能，即使不从 TheGame 类派生自己的游戏类，依然可以利用引擎中提供的其它类。类 GameWorld 是对游戏世界的模拟，类 MessageDispatcher 就是前面已经提到的消息分发器，它是一个单件。

在引擎中，Agent 由类 GameAgent 来描述，类 GameAgent 派生自类 MovingObject，类 MovingObject 是描述所有运动物体的类，这个类里面含有一

个 SteeringBehavior 类，SteeringBehavior 类实现了前面介绍的 Agent 的运动系统，由于这个类可以给运动计算带来很大方便，将其关联到类 MovingObject 而不是直接关联到 GameAgent。类 MovingObject 又从 GameObject 派生而来，GameObject 是游戏中所有物体的基类，所有的 GameObject 对象都由一个 ObjectManager 来统一管理，类 ObjectManager 是一个直接隶属于 TheGame 的单件。

可以看到，每个 GameAgent 都含有一个 StateMachine（有限状态机）对象，每个 StateMachine 又可以关联多个 State（状态）对象。在前面介绍 Agent 的决策系统设计时，前面已经说过，有限状态机提供了一个支持智能决策的框架，各种决策逻辑和算法都可以封装在一个个不同的 State 中，下面就来说明我们的 AI 引擎采用的有限状态机的结构。

4.2 集成消息处理的有限状态机

有限状态机（Finite State Machine, or FSM）在人工智能学科中具有形式化的定义，在此且不去关心它背后的理论，简单的说，有限状态机 FSM 是一个有着有限数目的机器，其中的一个状态是当前状态。FSM 可以接受输入，这将导致一个基于某些状态变迁函数的从当前状态到另外一个状态的变迁，这个新的状态就成了当前的状态[51]。

有限状态机是在游戏中应用最为广泛的实现 AI 效果的工具，几乎在每一款具有一定智能表现的游戏中，你都能找到各种各样的有限状态机的身影。即便在今天，各种智能技术已经取得不少的进步，有限状态机仍然是游戏开发者的首选。不难发现，有限状态机具有一系列吸引开发者的优点。

实现快速而简单。有限状态机原理简单，容易理解，可以用各种方法来实现一个有限状态机，基于已经实现的有限状态机来开发智能逻辑就更容易了，以至于可以用脚本来控制状态转换。

易于调试。没有什么比调试一个难以发现的 bug 更让开发者郁闷的了，一个易于调试的方法无疑具有相当大的诱惑力。有限状态机将复杂的逻辑分离成一个个离散的状态，很大程度上降低了问题的规模和复杂性，使得代码更加清晰且易于调试。

符合人类的思维方式。有限状态机的结构跟人类思考问题的方式是一致的，这使得在直观上理解和设计一个有限状态机变得相对容易。以另外一个常用的智能技术人工神经网络为例，在开发一个人工神经网络时，面对的是非常底层的输入和权值等数据，看到的只是简单的计算，而无法从直观上理解结果产生的机理，虽然人工神经网络是在某种层度上模拟我们大脑的工作方式，然而我们的大脑却无法直观的理解它。

灵活且易于扩展。一个有限状态机可以方便的被调整来适应新的逻辑变化，新的状态和转换规则也可以容易的被添加到一个有限状态机而不会引起大量的代码改动。更多的时候，灵活且易于扩展的特点，使得有限状态机被用来作为游戏智能逻辑的主框架，各种其他的智能技术可以被方便的集成到有限状态机中，这个特点也是一个游戏引擎所必需的。

关于有限状态机有各种各样不同的实现方式，Eric[51]实现了一个通用状态机类 FSMclass，在 FSMclass 中维护了状态机中所有状态的一个集合。它在每一个状态 FSMstate 的对象中维护一个输入和输出的对应表，在状态机的更新函数中，以当前的输入为参数，调用 GetOutput（int Input）函数就可以获得当前的状态在当前的输入下要转移到的状态 ID。

Eric 用不同的 ID 来区分不同的状态对象，而在 Charles Farris 的实现中[52]，每个状态都是被一个对象表示的。在这个对象中，状态函数调用将保存在函数指针里，当状态初始化的时候函数映射就会生成。FSM 通过维护一个指向当前状态的指针来跟踪当前状态，而不是 Eric 方法中的状态 ID。

在我们的有限状态机中，状态的作用是封装 Agent 在不同情形下的决策逻辑和算法，而状态机则是负责状态的管理和转换，将 Agent 的一部分作用代理到自身处理。设计一个精致和健壮的状态机，一般来说有这样一些要求[47]：

1) 状态机可以有任意数目的状态；
2) 容易的定义和设置；
3) 当进入一个状态时，能够执行任何初始化代码；
4) 当退出一个状态时，能够执行任何清除代码；
5) 可以容易的倾听消息并执行任何响应代码；

由于想要实现的是一个 AI 引擎中，具体的状态代码应该在游戏中来完成，引擎中的状态类应该是一个接口。另外，状态类必须不能依赖于任何具体的 Agent 类型，可以使用模板来构建这个状态接口，简化后的代码是这样的：

```
template <class entity_type>
class State
{
public:
   virtual void Enter(entity_type*)=0;      // 状态进入时的初始化代码
   virtual void Execute(entity_type*)=0;    // 状态的执行逻辑
   virtual void Exit(entity_type*)=0;       // 状态退出时的清除代码
};
```

同理，我们的状态机类也应该是一个抽象模板，我们列出它支持的接口函数：

表 4-1AI 引擎中有限状态机支持的接口函数

函数	作用
Update	在每次更新时被调用
ChangeState	改变当前状态
RevertToPreviousState	回到前一个状态
isInState	判断是否处于某个状态
SetCurrentState	设置当前状态
SetPreviousState	设置前一个状态
CurrentState	返回当前状态
PreviousState	返回前一个状态

这样，基本的有限状态机的结构就设计好了。前面讨论过消息机制的重要性，为了便于构建出的 Agent 能够方便地处理消息，我们希望构建出一个支持消息处理的状态机。大家知道，状态机是将具体的任务交给当前的状态来执行的，一种支持消息处理的方法就是在状态中添加消息处理函数，这种方法看似合理，然而，在实际应用中，多数情况是这样的：无论当前的 Agent 处于什么状态，我们都希望它能响应某个消息，而且在大多数状态下，这种响应都是一样的。如果使用状态中增加消息处理函数的方法，就需要在每个状态中都要添加对消息的支持，而且对于多数状态来说，这部分代码都是一样的。

一个好的方法是引入一个特殊的状态，专门用于消息响应，无论 Agent 当前处于什么状态，这个特殊的状态都能得到执行，如果 Agent 的当前状态需要以一种不同的方式响应某个消息，则当前状态可以截获消息并进行特殊处理。可以把这个特殊的状态称为 Global 状态，因为它在每次状态机更新时都会得到执行。状态机含有一个 Global 状态后，就可以为状态机增加消息响应函数，状态机的消息响应函数用伪码表示如下：

```
bool  HandleMessage(const Telegram& msg)const
{
   if ( 当前状态存在并且当前状态处理了消息 )
   {
      return true;
   }
   if ( 如果Global状态存在并且处理了消息 )
   {
      return true;
   }
   return false;
}
```

这个函数用返回值的真假表示是否处理了该消息，因此，在状态类中也要增加一个接口函数：

```
virtual bool OnMessage(entity_type*, const Telegram&);
```

在引擎中提供一个该函数的默认实现，那就是直接返回 false，不处理消息，需要处理消息的具体状态只要重载这个函数并返回 true 就可以了。

Global 状态的另一个作用就是可以提供一个良好的位置来安放那些每个状态都需要执行的代码，从而使得程序结构更为清晰易懂。

对于想要使用有限状态机的具体游戏，只需要根据其实现的 Agent 类型来实例化我们的有限状态机和状态类的模板并派生自己状态机和状态类，以及在具体 Agent 的更新函数中同步更新状态机就可以了。

4.3 动作选择器

在 Agent 决策系统的设计中，已经提到，决策过程就是选择出一个合适动作的过程。由于已经设计了一套机制来确定每个动作在当前状况下的效用值，下面就来看一下如何根据这些效用值来合理的选择出一个具体的动作。

最为简单的选择方法是每次都直接选择效用值最高的动作，这样做的缺点是会使得 Agent 过于“聪明”，对于游戏来说，这会使得一个对手无懈可击，有时候这反而会影响到游戏的可玩性。理想的状态是，使开发者可以设定 Agent 的“智商”，从而可以控制其选择出最佳动作的概率。

使用一个概率选择器 Selector 类可以满足这个要求，简化后的类声明如下：

```
template < class T >
class Selector
{
public:
   Selector();                                              // 构造函数
   void PushData(const T & ind, double probability);        // 添加数据
   T & DoSelect(void);                                      // 执行选择
   void Clear(void);                                        // 清除数据
   void SetContrast(double contrast);                       // 设置对比度
};
```

可以看到，概率选择器被设计为一个模板，这样就可以接受各种类型的待选目标，而不仅仅是一个固定的类型，如动作 ID。

首先创建一个选择器对象，然后使用 PushData()函数将待选项全部压入选择器，其中参数 ind 为待选对象，probability 为其效用值，也就是选择概率，该值可以为任意正数，选择器内部会对其进行单位化。压入所有对象和效用值后，调用 DoSelect()函数做出一次选择，函数返回一个选出的对象。

函数 SetContrast()用于调整选择出较优选项的概率，命名为 SetContrast 是因为这里使用了与图像处理中调整图像对比度相似的算法，参数范围为 0 到 1，当“对比度”设置较低时，各个选项间的区分度不大，选择出较优选项的概率不高，Agent 表现较为“愚钝”；当“对比度”设置为较高值时，效用值高的选项概率增大，效用值低的选项概率降低，选出较优选项的概率升高，这时的 Agent 也显得更加“聪明”。

4.4 AI 优化策略

为了提高游戏的画面效果，在游戏每一帧的有限时间里，渲染系统总是尽可能多的使用大量的 CUP 资源，这样，留给 AI 计算的时间更加有限，一个高效的 AI 引擎必须要对 AI 计算进行不断地优化。

4.4.1 延迟消息机制对 AI 优化的作用

任何人和生物对待各种事件都需要一定的反映时间，总是快如闪电的 Agent 不仅不会显得更加智能，反而会降低游戏的真实感。这个事实提供了一个很好的减少 AI 计算的契机。

在我们的引擎中，已经实现了延迟消息机制，利用一个发给自身的延迟消息，Agent 可以容易的将某个复杂的计算延迟到以后进行，从而降低当前的计算负担。在游戏中，往往经常出现这种紧迫的时刻（如足球中禁区内的攻防），多个 Agent 都需要进行大量复杂的计算，也就是常说的 AI 处理高峰。同样的，只需要在允许的时间内随机延迟一段时间，就可以有效地避免在同一时刻出现多个 Agent 都需要计算的处理高峰。

4.4.2 规整器设计

除了上面说的方法，在我们的引擎中，还实现了一个规整器来规整 AI 的计算。规整器的基本思想是这样的：一般来说，游戏模拟的是一个连续的世界环境，这种环境在很短的一段时间内不会产生很大的变化，因此，对于一些依赖与环境的计算，在一定的时间内只需要计算一次就可以了。

例如，某个类有一个这样的函数 f，用于对环境信息的计算，在一帧内，许多 Agent 都需要这个函数的计算结果，没有优化的方式就是直接在每个 Agent 的代码中调用函数 f，这样会造成很多的额外计算。当然，在 Agent 内保存计算的结果也可以达到减少额外计算的目的，然而确定什么时候重新计算又给 Agent 增加了多余的职责，特别地，当存在多个这种函数时，这些中间结果和控制代码会使得整个 Agent 变得臃肿不堪。好的解决方案就是使用规整器。

一个规整器是一个类，它含有一个开关函数 IsReady()，该函数在设定的时间段内仅第一次调用返回 true，其余的调用都返回 false，直到时间到期进入下一个时间段。

简化后的规整器类声明如下：

```
class Regulator
{
public:
   Regulator(double NumUpdatesPerSecondRqd);    // 构造函数
   bool IsReady();                               // 开关函数
};
```

在构造函数中设定要求的时间间隔，然后就可以直接调用 IsReady()函数判断规整器是否就绪。举例说明一下规整器的使用，有一个负责计算的类：

```
class MyCalculator
{
public:
   double CalculateSomething();     // 计算函数
};
```

其中的函数 CalculateSomething()开销较大，希望对它的调用进行规整。只要添加如下代码：

```
class MyCalculator
{
public:
   double GetSomething();
private:
   double CalculateSomething();
   Regulator m_RegulateSomthing;
   double m_SomethingResult;
};
```

将原来的函数 CalculateSomething()变为私有，用一个同类型的函数 GetSomething()代之，同时增加一个规整器成员和一个保存计算结果的成员。简化的 GetSomething()函数实现是这样的：

```
double GetSomething()
{
   if ( m_RegulateSomthing.IsReady() )
      return m_SomethingResult = CalculateSomething();
   return m_SomethingResult;
}
```

4.5 本章小结

本章利用上一章设计出的多 Agent 系统模型构建出了一个游戏 AI 引擎，实现了一个基于 C++模板的有限状态机，并集成了消息通讯机制；实现了一个通用的动作选择器作为对 Agent 动作决策的支持；采用了延迟消息机制与规整器来提高该 AI 引擎的效率。

第五章 游戏相关知识集成

现在已经设计好了一个基于多 Agent 系统的游戏 AI 引擎，本章以足球模拟游戏为例来说明怎样利用这个引擎构建一个具体的游戏。在我们的 AI 引擎中，已经具有了实现游戏 AI 的基本元素，但是，游戏 AI 是和具体游戏紧密相关的，也就是说，针对一个具体的游戏，要将游戏相关的知识与我们的 AI 引擎集成起来。

5.1 足球模拟游戏说明

本文要实现的是一个足球模拟游戏，其中，两方球队中所有的球员都是计算机模拟的 Agent，玩家可以针对两边的球队设置一些阵型、战术要求等参数。该游戏主要用于展示我们的 AI 引擎的智能效果，在游戏的设计与可玩性等方面不做过多讨论。

5.2 游戏与 AI 引擎中的类关系

利用现有引擎提供的类，只需要派生一些游戏中的类，足球游戏中主要的类如图 5-1 所示。在图 5-1 中，位于虚线框内的类是游戏中实现的类，基本上都是从我们引擎里的类派生来的。其中实现了两种 Agent，普通球员（FieldPlayer）和守门员（GoalKeeper），这里为这两种 Agent 分别实现了各自的有限状态机类和状态类。球场（SoccerPitch 类）从游戏世界（GameWorld）派生来，用于描述整个球场环境。球队（SoccerTeam）用来维护 Agent 群体，事实上，在游戏中也为球队实现了有限状态机和状态。

5.3 状态设计

有了有限状态机，只需要构建足球游戏中的相关状态就可以了。在足球游戏中，构建了三个有限状态机。分别是球队状态机，球员状态机和守门员状态机。将守门员单独独立成一个状态机是因为守门员的行为与普通球员的行为差别较大，使用同一个状态机容易造成状态的冗余和混乱。

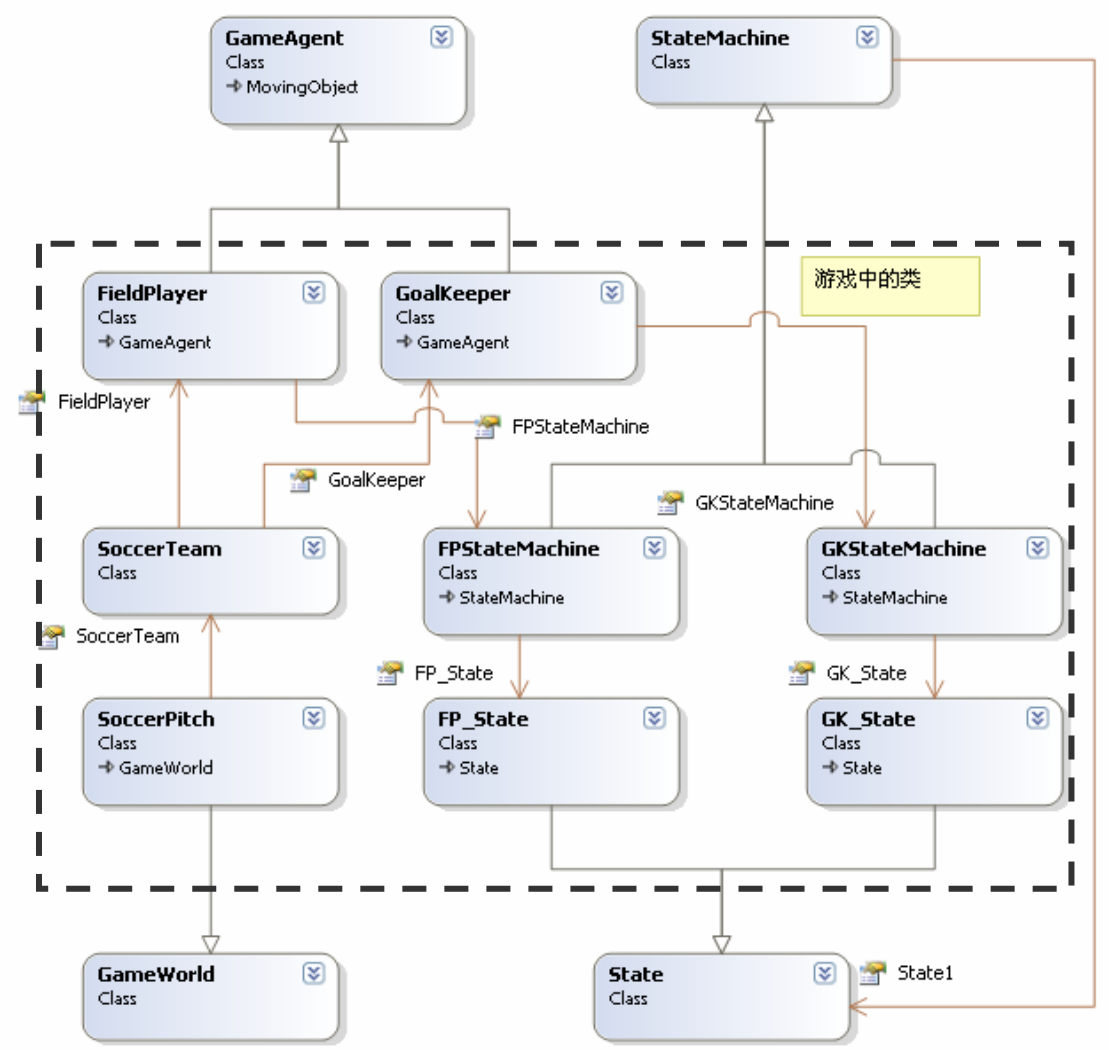

图 5-1 足球游戏类关系图

球队实现的状态机中的主要状态如表 5-1 所示。普通球员和守门员的状态分别表示为表 5-2 和表 5-3。

表 5-1 球队主要状态

状态	说明	备注
TS_Attacking	进攻状态	
TS_CornerBall	角球状态	
TS_Defending	防守状态	
TS_FreeKickDefence	任意球防守	
TS_FreeKickOffend	任意球进攻	
TS_PenaltyKick	罚点球	（攻/防）
TS_PrepareForKickOff	开球	（攻/防）

TS_ThrowIn	界外球	（攻/防）

表 5-2 普通球员主要状态

状态	说明	备注
FPS_ChaseBall	追球跑状态	
FPS_CornerKick	开角球的状态	
FPS_CornerShoot	接应角球准备射门的状态	
FPS_CornerWait	防守对方角球/本方角球时除了开球和准备接应射门以外的球员的状态	
FPS_Dribble	带球跑状态	
FPS_FreeKick	任意球开球球员状态	
FPS_FreeKickWait	接应任意球/防守任意球的状态	
FPS_FreeKickWall	防守任意球时人墙球员的状态	
FPS_KickBall	踢球（处理球）状态，这里进行射门，传球，带球的决策	
FPS_KickOffKick	开球时踢球的球员状态	
FPS_KickOffRecv	开球时接球的球员状态	
FPS_None	空状态，无意义，在代码中起标志作用	
FPS_PenaltyKick	点球球员状态	
FPS_PenaltyWait	点球接应球员状态	（攻/防）
FPS_ReceiveBall	接球状态	
FPS_ReturnHome	返回自身标准位置的状态	
FPS_SupportAttacker	助攻状态	
FPS_ThrowIn	掷界外球的状态	
FPS_ThrowInWait	响应/防守界外球的状态	
FPS_Wait	无球跑状态	（攻/防）

表 5-3 守门员主要状态

状态	说明	备注
GKS_GoalKick	开球们球的状态	
GKS_InterceptBall	出击截球	
GKS_PutBallBackInPlay	将球返回给本方球员	
GKS_ReturnHome	返回守门员的标准位置	
GKS_TendGoal	守门状态	

5.4 球员个人技术

构建一个球员，也就是构建能够踢足球的 Agent，需要赋予基本 Agent 一些足球相关的知识和技能，帮助 Agent 在特定的环境即足球比赛中做出合理的决策和动作。

5.4.1 技术表示

先来研究一下怎样定量的表示出足球相关的一些技术动作。

一个技术动作，应该是一系列的特定基本动作的组合，其中每个动作应有几个相关的参数来描述。最后，任何技术动作都需要一定的环境和条件才能完成。可以这样来描述一个足球中的技术动作：

{动作和参数序列，条件}

例如一个简单的带球跑动作，可以描述如下：

动作序列：踢球（力度，方向）；跑动（速度，方向）

条件：受威胁程度小于一定值

5.4.2 技术层次

容易注意到，有些技术无法用基本动作的组合完成，如踢球，把这种技术称为原子技术。而另外一些技术可以用基本动作和原子技术组合而成，如带球、过人等。显然可以用组合模式[45]来表示技术动作（图 5-2）。

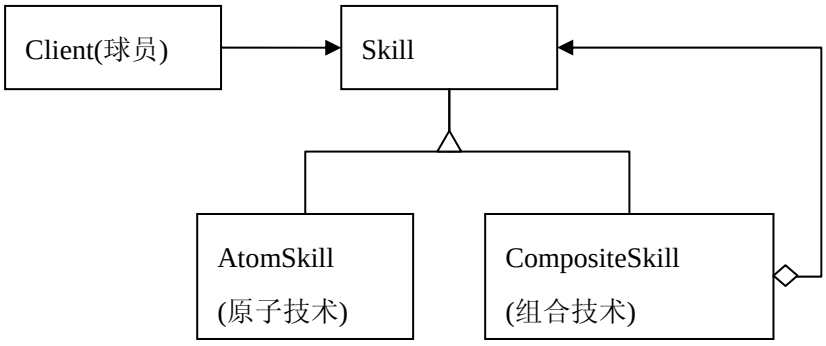

图 5-2 球员技术的组合模式

这样，对于球员来说，就可以将各种技术用统一的方式来对待，而不用关系每种技术的细节。在抽象的 Skill 层次，每种技术还对应一个抽象相关参数类，具体技术的参数由抽象参数类派生而来。

5.4.3 个人参数与角色参数

由于采用的是同样的 Agent 模型来构建游戏中的所有的 Agent，到现在为止，这些 Agent 还是完全一样的，在游戏中，他们会有着相同的模样、对同样的情形做出同样的反应，他们实际上就是同一个东西的拷贝！玩家当然不希望看到这样的游戏，他们希望游戏中每个 NPC 具有自己的个性特征，对同样的情形能产生不同的反应，这样的游戏才有意思。

5.4.3.1 个人参数

通过为每个 Agent 赋予几组参数来刻画他们的不同个性，让构建出来的游戏看起来更自然。

首先，与现实中的运动员一样，游戏中的球员应该具有一些基本的身体参数，包括：

身高；体重；弹跳；速度；力量；敏捷度；耐力。

这些参数将对球员执行某种技能的效果产生影响，例如过人技术需要足够的灵敏度和速度，否则就容易失败。在设计这些参数时，要注意其逻辑性，比如，一个身高不足 170cm，体重超过 80kg 的选手不应该拥有较高的敏捷度和较低的力量，等等。

在足球中，不同的选手对同样的技术掌握的熟练层度是不一样的，这里使用执行能力来刻画该选手对某个技术的掌握程度。相关技术的执行能力有射门技术、头球技术、角球技术、点球技术，长传技术等。

以上这些参数基本上都是客观上的物理参数，为了使游戏更真实，游戏中还为 Agent 提供了一组性格参数，用来刻画 Agent 的性格。如表 5-4 所示。

表 5-4 球员的性格参数

参数名称	说明
稳定性	球员在球场上发挥其正常水平的概率。
积极性	刻画球员跑位、防守的积极程度。
自信	用于影响球员对球的处理。

另外，对足球队员来说，还会具有一定的个体倾向，表现为射门倾向、传球倾向、带球倾向、犯规倾向等。

实际上，上面讨论的这些参数往往不是独立变化的随机变量，而是相互关联的，关于参数的变化规律这里不做深入研究，只是介绍在足球模拟游戏中使用这些参数来实现 Agent 的个性化。

5.4.3.2 角色参数

在足球比赛中，球员是具有不同的角色的，例如前锋、中场、后卫等，不同的角色对球的处理有着不同的倾向，用角色参数来描述不同角色之间的差异（表5-5）。

表 5-5 不同角色的动作倾向

角色 / 参数	射门倾向	传球倾向	带球倾向
前锋	高	中	高
中场	中	中	中
后卫	低	高	低

5.5 团队战术

足球是一项集体运动，一个球员技术再优异，没有团队的配合，也无法赢得比赛。要使得我们的游戏看起来是一个团队运动而不是个人秀，必须把团队战术集成进来。

5.5.1 阵型约束

对于足球来说，阵型是团队战术中一个非常重要的因素，一个训练有素的球队总是可以维持一个良好的阵型，阵型中不同的球员各尽其职。

5.5.1.1 阵型表示

对于足球比赛中的阵型，首先要解决其表示问题，对于任何一个阵型来说，在不同的状态下都会有不同的变化，对于每种阵型，又将其划分为若干个子阵型，如开球子阵型（攻），开球子阵型（防），进攻子阵型，防守子阵型等。另外，对于定位球，设计了如角球子阵型（攻），角球子阵型（防），点球子阵型（攻），点球子阵型（防），任意球子阵型（攻），任意球子阵型（防）等等。

对于每种子阵型来说，要表示出的信息实际上是某种类型的球员位于什么位置上。因此，可以用这样的结构来描述阵型中的一个位置信息：

```
struct FormationElement
{
坐标位置;        // 用于控制阵型的一个基准位置
活动范围;        // 这个位置上的球员的大致活动范围
球员角色;        // 处于这个位置上的球员对应的角色（前锋、后卫）
};
```

这样，就可以用一个位置信息的数组来描述一个子阵型，再用若干子阵型构成的数组描述一个完整的阵型，阵型数据可以保存在一个文件中供游戏调用。游戏中还专门开发了一个辅助工具阵型编辑器来简化阵型文件的创建。关于阵型编辑器的设计见 6.6 节辅助工具开发。

5.5.1.2 阵型保持

在足球比赛进行中，总是要求既要保持一定的阵型结构，又要对具体的形式灵活反应。首先，对于阵型的灵活性来说，可以使关键的球员暂时不考虑自己的阵型位置，例如在进攻和防守的关键时刻。困难的是对于普通情况下的无球队员，如果将其固定在阵型位置上，则过于呆板；如果让他们随机移动，则不利于阵型的保持。在足球模拟游戏中，使用 HomePosition 和 StrategyPosition[53]的概念来保持阵型，同时不失灵活性。HomePosition 就是阵型位置中提供的基准坐标点，球员一般不会固定在这个位置上不动，StrategyPosition 是根据 HomePosition 和目前场上形式计算出来的一个策略位置。

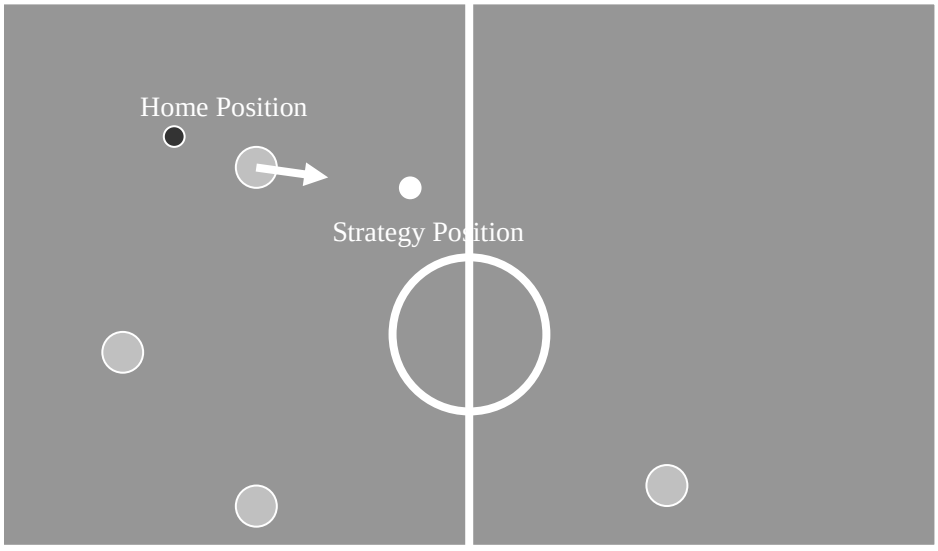

图 5-3 阵型中的 Home Position 和 Strategy Position

5.5.2 战术配合

对于足球游戏来说，个个球员之前的战术配合是必不可少的。从前面 AI 引擎中的 Agent 设计已经知道，已经为多 Agent 协作提供了一些基本的支持，下面就来研究怎样利用已有的 AI 引擎在足球模拟游戏中将战术表现出来。

5.5.2.1 战术表示

实际上，一个战术可以认为是一个团队动作，就是我们引擎中说的群体任务。一个战术应该包含一定的适用条件，若干个参与者以及每个参与者的任务。因此

可以这样表示一个战术：

```
TacticsInfo
{
   战术名称（ID）；
   适用条件的描述；
   参与者描述数组；
}
```

其中，参与者描述数组中每个元素表示如下

```
ActorInfo
{
   参与者ID；
   对该参与者的要求描述；
   该参与者的动作序列（任务）；
}
```

5.5.2.2 团队协作

有了战术的表示方法，再来看一下执行一个战术的流程（图 5-4）。

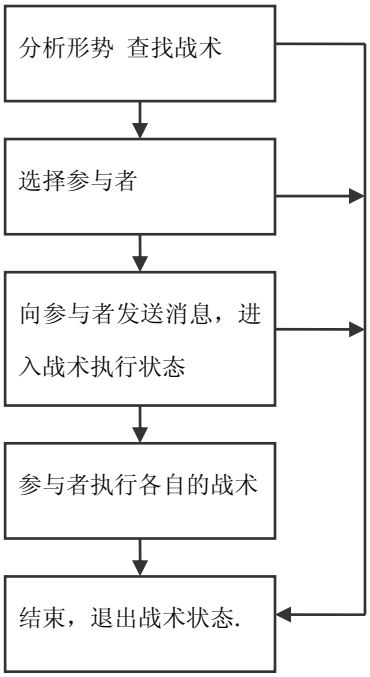

图 5-4 战术执行流程

可以看出，一次战术的执行是这样的：首先，在球队的更新任务中，分析和检查当前的比赛形势，查询满足条件的战术；在找到合适的战术之后，根据战术信息选择响应的参与者，然后向这些参与者发送消息，使其进入战术执行状态，每个参与者从团队的“黑板”上读取到自己需要执行的任务，开始执行这些任务，任务执行完毕后战术结束，参与者回到正常状态。值得一提的是战术的执行是一

个严密的过程，其中每一个环节都会出错，例如某个参与者的某个动作执行失败，都将导致整个战术的失败。

5.6 用于决策的影响力地图

在足球游戏中，虽然射门是赢得比赛的关键，然而真正反映整个球队战略和战术的不仅仅是射门，各种情况下的传球和带球至关重要，并且直接影响到比赛的效果与观赏性，对于足球模拟游戏来说，这反映了游戏的智能程度。因此，让游戏中的 Agent 做出正确的决策至关重要。日常经验告诉我们，要做出有效的决策，不仅仅要有最佳的数据，而且还要以正确的方式表示这些数据。原始数据只有被转换为上下文相关的场景信息后才能发挥最大作用。

影响力地图[54]就是一种数据的维护和表示方式，被许多游戏广泛采用。对于战术评估而言，建立影响力地图是一种宝贵的、得到实践检验的技术。影响力地图是 Agent 关于游戏世界知识的空间表示，让 Agent 能够根据游戏环境的物理或地理表示获悉当前游戏状态的战术视图。从影响力地图中，Agent 可以得知对方的位置信息、兵力分布、敌我边界、防守薄弱环节等可以直接影响决策的非常重要的信息。

影响力地图实际上是一个数据网格，并不存在创建影响力地图的标准方法，影响力地图的创建和使用在很大层度上取决于具体的游戏设计，这也是没有将影响力地图放到我们的 AI 引擎中的原因。

下面针对足球模拟游戏，来创建一个用于决策的影响力地图。

5.6.1 构造单元格

创建影响力地图的第一步是确定合适的单元格大小，一般来说，影响力地图的单元格大小可以是任意的。然而，如果单元格太大，则影响力地图将难以表示一些过小的细节信息；如果单元格太小，则需要更多的计算来维护单元格数据的更新，并需要更多的内存来存储这些数据。

首先来看一下足球模拟游戏的特点。大家知道，在足球比赛时，当本方控球且球位于后场时，动作的选择一般较为简单，这时候球员处理球时一般不会选择射门，比赛的节奏也较为缓慢，对方的防守一般来说也不算紧迫，各种动作的执行有较大的余地，对技术精度要求不高；相反，当本方控球进入前场或是对方禁区附近时，比赛节奏变快，对方防守积极，球员对球的处理更加谨慎，考虑的因素也更多，这时候一个小的疏忽都可能造成一次射门机会甚至得分机会的丧失。

针对足球游戏的这种特点，我们提出一种不均匀网格的影响力地图的方法。

如图 5-5 所示，以左侧的球队为例，在其后场，采用较大的单元格来构建影响力地图，而在前场，采用中型的单元格大小，在对方禁区附近，则采用较为精细的单元格大小。通过这种方式，使得单元格得到最大限度的使用。注意这里三种单元格大小都是倍数关系，这是为方便后面的影响力传播算法设计的。

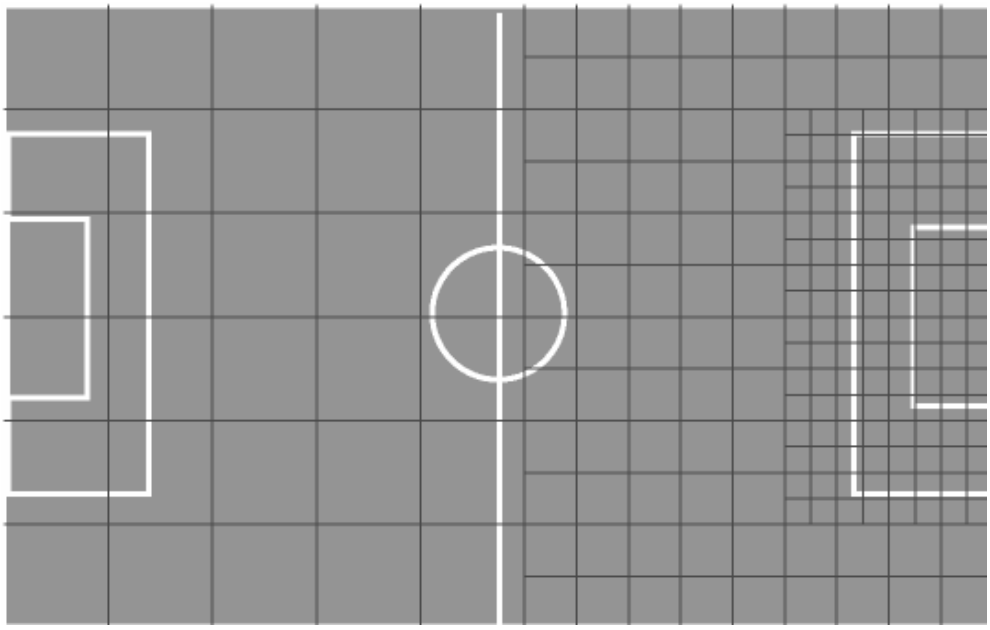

图 5-5 左侧球队的影响力地图

在确定单元格大小之后，我们来确定单元格的数据成员。影响力地图之所以被如此广泛的应用，就在于使用者可以给单元格设定各种感兴趣的数据成员，这实际上将影响力地图变成了一个小型的实时的动态数据库。

在影响力地图中的每个单元格中，设定以下数据成员：

占有率。表明该单元格被本方队员控制的程度，完全没有球员占领的单元格占有率为 0，被本方球员控制的单元格占有率为一个正数，且控制该单元格的本方队员越多，占有率越高。相反，被对方球员控制的单元格占有率为负数。

射门概率。该单元格所在位置的射门得分的概率。

传球评估。对到该单元格的一个传球的效果评估，这个数据又包含几个组成部分：

1) 传球距离评估 该传球的距离是否合理，过短或过长，是否符合团队战术的要求等。
2) 传球方式评估 该传球的方式是否合理，传球是否会被对方抢断，是否迅速有效等。
3) 传球方向评估 该传球是向前传球还是向后传球，是否满足当前比赛形式的要求。

5.6.2 不均匀单元格的影响力传播算法

在计算完影响力地图中每个单元格的初始值后，接下来要将每个单元格的值传播到附近的单元格。这一个过程被称为修匀（smoothing）或模糊化（bluring），因为它与标准的图像模糊化技术有很多共同之处[55]。

影响力传播实际上是使用“衰减规则”将每个单元格的影响力扩展到相邻的单元格。常用的衰减规则有指数衰减、线性衰减和高斯滤波[55]。

一般来说，对于单元格大小完全相同的均匀的影响力地图，影响力的传播可以直接根据到给定单元格的距离和衰减规则计算出来。对于不均匀的单元格大小，传播算法还需要做适当的调整。

从图 5-5 中可以看到，在影响力地图中，单元格的大小是具有区域性的，也就是说，在同样大小单元格的区域内，可以使用原来的算法计算影响力的传播；不同之处存在不同大小的单元格的相邻的地方，来看一个具体情况(图 5-6)。

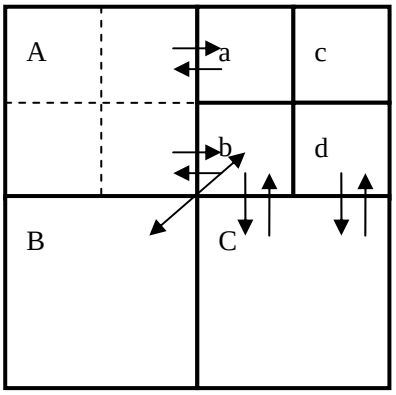

图 5-6 不同大小单元格之间的影响力传播

从源单元格到目标单元格的影响力传播计算公式为：

$$I' = I \times \alpha^{d} \qquad \text{公式(5-1)}$$

其中 I 为源单元格的影响力，α 为衰减系数，d 为源单元格到目标单元格的距离。I'为从源单元格传播到目标单元格的影响力。这里的单元格距离 d 指两个单元格中心的距离。

以图 5-6 为例，在计算较大单元格之间的影响力传播时，如 A 到 B 的影响力，可以直接使用公式（5-1）进行计算。在计算较大单元格向较小单元格的影响力时，如 A 到 a，依然可以使用公式（5-1）进行计算，只是要注意这里的距离 d 是两个单元格中心的距离。

不同的是计算较小单元格向较大单元格的影响力传播，如 a 到 A 和 b 到 A，

这时，如果仍然按照公式（5-1）进行计算，则 a 和 b 对 A 的影响都会被叠加到 A 的影响力中，这显然是不正确的。在这种情况下，需要根据单元格的大小为影响力的传播增加一个权值，这时候的计算公式如下：

$$I' = \frac{S}{S'} I \times \alpha^{d} \qquad \text{公式(5-2)}$$

其中 S 是源单元格的面积，S’是目标单元格的面积。将公式(5-1)与公式(5-2)统一起来就是：

$$I' = \begin{cases} I \times \alpha^{d} & S \geq S' \\ \dfrac{S}{S'} I \times \alpha^{d} & S < S' \end{cases} \qquad \text{公式(5-3)}$$

5.7 本章小结

本章将具体的足球游戏知识和技能集成到通用的游戏 AI 引擎中，设计了足球比赛中球员技术和球队战术的表示和模拟方法，提出了一套描述球员个性的参数。在球员的动作决策上，还提出了一种不均匀影响力地图的方法，并给出了相关的影响力传播算法。将游戏相关的知识和技能表示出来后，下一章就可以开始构建一个完整的游戏了。

第六章 完整的游戏

6.1 游戏架构

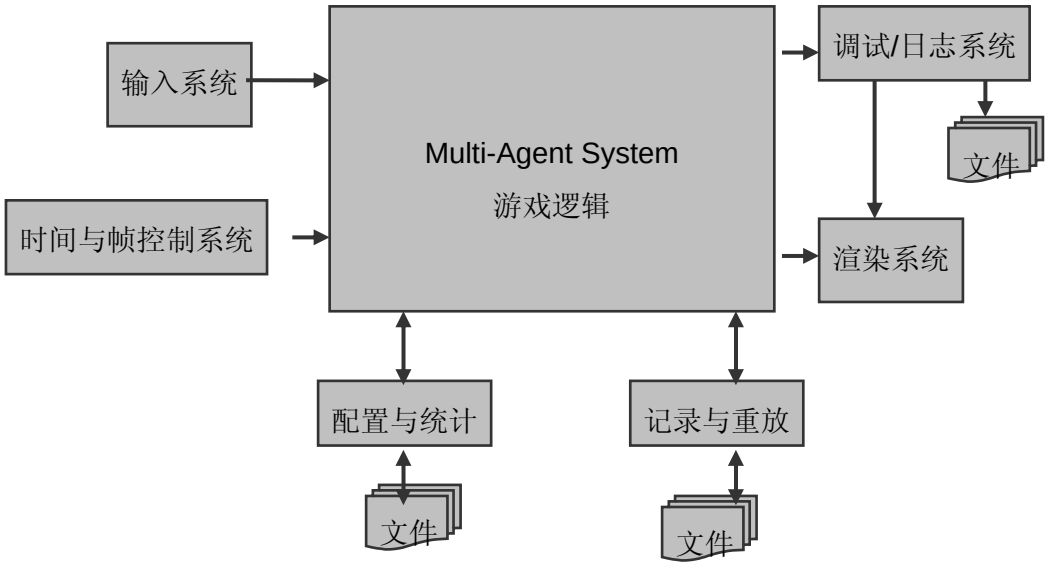

图 6-1 足球模拟游戏主要模块架构

足球模拟游戏整体上的结构如图 6-1 所示，其中最主要的部分就是游戏逻辑，也就是前几章重点讲述的部分，它是基于我们的 AI 引擎实现的一个逻辑模块。游戏通过输入系统接受用户的输入，通过渲染系统将游戏画面表现出来，同时，使用时间与帧控制系统来控制游戏的执行速度与时间模拟。游戏的配置信息从文件读入，游戏的运行数据可以记录到数据文件，也可以通过日志系统输出到日志文件或直接输出到屏幕。下面就从游戏循环开始介绍我们游戏的一些实现上的方法和技巧。

6.2 游戏循环

对于游戏程序，总有一个顶层的循环结构来驱动整个系统的运行，这个循环被程序游戏循环。这里简单介绍一下游戏循环的结构以及我们的游戏中采用的游戏循环模型。

一个游戏程序显然是一个软件应用程序，更具体的说，游戏属于实时交互应

用程序[56]。一个实时应用程序需要在一定的时间限制内完成要求的操作，以游戏为例，为了达到用户（玩家）能接受的交互效果和画面显示，通常要求画面每秒钟更新 25 次以上，也就是帧率 25 以上，对于理想的交互效果，帧率一般在 30 甚至更高。

一般来说，在实时交互式应用程序中，包含三个需要同时执行的任务。首先，程序内部维护的状态需要持续的更新；其次，需要实时处理用户的输入；最后，结果要通过输出系统及时地反馈给用户。相似地，在游戏中，一帧之内要完成的任务就是接受用户输入、更新游戏内部状态以及渲染图像和声音。

在游戏开发中，很多情况下，可以把输入看作状态更新的条件之一，合并到更新任务中，这样，游戏程序的每帧内要处理两大任务就是更新和渲染。理想的状况下，这两大任务须要并行执行，并且执行频率越高效果越好。但是在目前的硬件水平限制下，理想状况无法得到，需要一些技巧来处理这两大任务的执行。

最为简单的方法是在一个循环内同时执行这两个任务（图 6-2）这是一种将更新和渲染两大任务耦合起来的做法，实际上，更新任务的执行频率决定了系统的速度，而渲染任务的执行频率决定了系统的表现力。这种方法将两个具有不同频率要求的任务耦合在了一起，强制性的使它们具有了相同的执行频率。我们希望渲染任务能够根据系统中硬件的能力尽可能频繁的执行，以达到较好的视觉效果，而更新任务则应该在不同的硬件平台上都保持同样的执行频率，以使得游戏具有相同的执行速度。这种耦合的结构显然不能满足要求。

另外一个方法是使用两个线程来分别处理这两大任务（图 6-3），这样就可以单独控制每个任务的执行频率，使两个任务按各自的要求执行，二者完全解耦合。然而这种方法存在实现上的问题，渲染任务应该保证在一次更新任务完全执行完毕后进行，而不应该在更新任务执行到一半时进行，这样就需要维护两个线程的同步。另外，对于游戏来说，对时间的要求非常精确，就目前的硬件功能和操作系统来说，还难以在多线程的情况下提供精确的时间控制。由于这些实现上的问题，这种双线程的游戏循环模型在实际的游戏开发中几乎见不到。

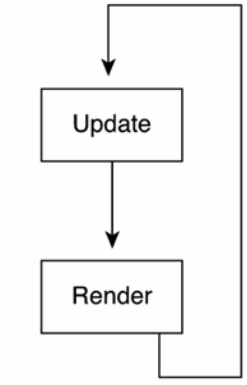

图 6-2 游戏循环的耦合方法

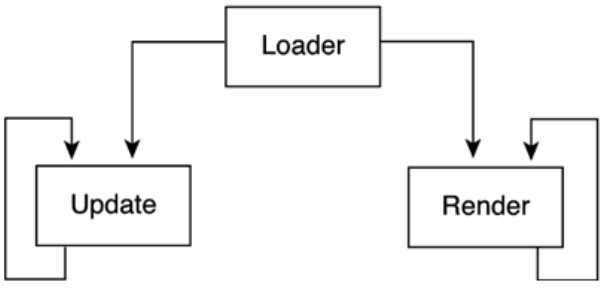

图 6-3 游戏循环的双线程方法

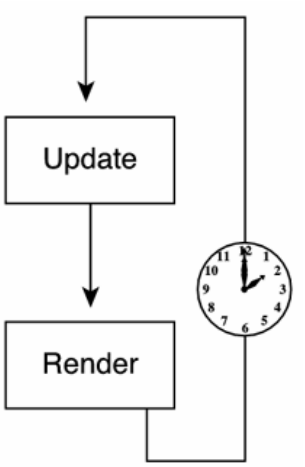

图 6-4 单线程非耦合方法

在实际中应用最为广泛的游戏循环就是使用单线程模型配合一个时钟（图6-4），这种方法的主要思想是依然线性地执行更新任务和渲染任务，使用时钟来控制更新任务的执行频率，使其在不同硬件平台上具有同样的效果。对于渲染任务，则尽可能频繁的给予执行。通过这种方式，更新任务和渲染任务也被很好的解耦合了。

这种方法的控制算法可以用伪码表示为：

```
上次调用时间 = 当前时间();
while (!结束)
{
   if (当前时间－上次调用时间)>1000/更新任务预定执行频率)
   {
      执行更新任务;
      上次调用时间 = 当前时间();
   }
   执行渲染任务;
}
```

6.3 游戏中的控制与管理

游戏是一个很复杂的软件系统，游戏中存在大量不同种类的对象，他们的继承关系和所有者关系错综复杂，另外，游戏的基于帧的结构也使得游戏与一般的应用软件大不相同，对于游戏来说，一个设计良好的控制与管理机制是保证软件质量的必要条件。

6.3.1 时间管理与帧控制

在游戏开发的早期，多数程序员根本不使用时钟的概念，他们总是将尽可能多的东西放在一帧内执行，这时候帧的长度由 CPU 的计算速度决定。如果游戏总是在相同配置的电脑上运行，这样做一点问题也没有。但是如果有人试图在较快的 CPU 上运行该游戏，则整个游戏都会变快，这里变快并不是说游戏的运行更加流畅，而是游戏中的一切都变快了，比如游戏中物体的运动等，这导致有时候游戏根本没法玩了。

前面已经指出，采用的基于时钟的游戏循环可以很好的解决上述问题，在这种游戏循环模型中，关键的问题就是构建一个足够精确且可靠的时钟。多数情况下，操作系统会提供定时器的支持和一些时间相关的函数，这对于一般的应用来说以及足够了，但是对于游戏这种对时间要求很高的程序来说就显得精度不够，关于如何构建游戏中的时钟已经不是一个新的话题，游戏中的时钟也有很多种具体的实现[57, 58]。我们游戏中采用了一种比简单的方法[43]。

在游戏的开发过程中，特别是在调试阶段，能够在运行时人为改变帧率对 bug 定位具有很大帮助。在我们所采用的游戏循环模型中，只需要对游戏循环做简单的改进就可以方便的提供对帧率的控制。这样，在调试过程中，（甚至最终的游戏中），可以降低帧率以便于详细查看游戏的执行细节，也可以增加帧率让游戏飞快地执行到某种状态。

6.3.2 游戏实体管理

6.3.2.3 实体管理器

一般来说，在游戏中存在各种游戏对象和资源对象，而这些对象的创建和销毁是动态的，特别是在一个灵活的基于对象组合的架构中，想要知道什么时候一个对象对另外的对象具有所有权是一件非常困难的事情。为解决这个问题，可以引入一个实体管理系统来统一管理对象资源，这样，某个对象需要访问其他对象的时候可以连接到实体管理系统来获取其他对象的访问权，这就很好的避免对象所有权上的混乱。

Bilas 实现了一个基于句柄的模板化的资源管理器[59]，Hawkins 则讨论了一种利用智能指针的对象管理机制[60]，而 Natalya 设计了一个非常复杂但极其强大的对象管理器[61]。通过这些实现，可以总结出一个好的实体管理器应该具有的功能和特点：

1) 高效的存储各种实体对象
2) 快速检索一个实体对象

3) 方便的添加和删除一个实体对象

从这几个方面考虑，针对足球模拟游戏，实现了一个实体管理器类，简化后的类声明如下：

```
class EntityManager
{
private:
  EntityMap m_EntityMap;     // 存储ID到实体指针的映射表
public:
  void RegisterEntity(BaseGameEntity* NewEntity);    // 注册新的实体
  BaseGameEntity* GetEntityFromID(int id)const;      // 获取一个实体指针
  void RemoveEntity(BaseGameEntity* pEntity);        // 移除一个实体
  void Reset(){m_EntityMap.clear();}
};
```

实际上，在我们的游戏中，该实体管理器还是一个单件[45]，这使得它的使用更加简单且不容易出错。

6.3.2.4 自动列表设计模式

在游戏开发过程中，经常需要选择性的访问同一类型的一系列对象，例如类型为 T 的对象，它时从类型 O 派生出来的。而在游戏实体管理中，往往只保留一个列表，包含所有从 O 派生出来的对象，一般的做法是先获得这个列表，遍历一遍，从中取出所有类型为 T 的对象。这种做法需要通过基类的指针来判断其对象的类型，如果为了效率问题而不用 RTTI 的话，则需要人工添加动态类型识别的代码。显然，这不是一个优雅的解决方案。

另外一种方法是为每一种类型的对象创建一个单独的列表，这样就可以快速的访问需要的对象了，然而，这种方法也存在各种缺陷：

1) 这个列表本身的存储。随着游戏中对象类型的增多，各种对象的列表也变得越来越多，列表用来管理对象，而对这些列表本身的管理又成为了新的问题。
2) 保证对象在创建时加入列表，在销毁时从列表中删除。使用这种方法构建的游戏引擎，必须要求客户程序员在创建一个对象的时候将其添加到列表，而在销毁一个对象的时候将其从列表中删除。例如，类 T 从类 B 派生二来，而类 B 从类 O 派生来，程序员创建了一个 T 类型的对象，这时候，需要将该对象添加到 T 类型的列表、B 类型的列表、O 类型的列表，同理，在该对象销毁时还需要从相应的列表中删除该对象。在游戏开发过程中控制对象的创建和销毁本身就是一件麻烦的事，这种做法无疑给程序员带来了更多的麻烦。

3) 防止对列表的任意添加和删除。一旦程序员可以访问对象的列表，就需要防止对列表的非法访问，由于在引擎中无法确定用户程序员的对象类型，这就又增加了用户程序员的职责，也就降低了引擎的易用性。

另外，实现这样的列表来将一个类维护需要在很多个位置修改代码，这也增加了出错的可能并为调试带来麻烦。理想的情形是：当客户程序员需要某个特定类型的列表时，他只需要在某一个地方做出某种标志或简单的修改，而不需要在其他任何地方修改代码，就可以使用到前面所说的访问此类型的对象列表的功能，同时能避免上述的各种缺陷。

一个被称为自动列表[62]的设计模式提供了这样一个解决方案。用简化后的代码来说明它的原理：

```
template <class T>
class AutoList
{
public:
  typedef std::list<T*> ObjectList;
private:
  static ObjectList m_Members;
protected:
  AutoList()
  {
    m_Members.push_back(static_cast<T*>(this));
  }
  ~AutoList()
  {
    m_Members.remove(static_cast<T*>(this));
  }
public:
  static ObjectList& GetAllMembers(){return m_Members;}
};
```

可以看到，自动列表类是一个 C++模板类，它有一个静态的成员变量，类型为 T 对象指针的列表。在构造函数中，它把 this 转换为 T*类型，从而可以把自己加入列表。在销毁时，它把自己从列表中删除。当客户程序员需要把某个类型的对象加入列表时，他只需要从该模板派生自己的类型：

```
class CListMe : public AutoList<CListMe>
{
   //....
};
```

由于 CListMe 的构造函数自动调用基类的构造函数，因此在 CListMe 的对象创建时，该对象就自动被添加到 AutoList<CListMe>类的列表中，同样的道理，

对象销毁时也会自动从列表中移除，而不需要程序员做任何其他动作。这种以派生类实例化基类模板的技术被称为 Curiously Recurring Template Pattern （CRTP）[63]，有人翻译为“诡异而循环的模版模式”。

当程序员需要访问某个列表时，只需调用 AutoList<CListMe>::GetAllMembers()即可，利用静态成员来存储不同类型的列表，不需要考虑列表的维护问题。另外，可以在 AtuoList 中方便的加入成员函数来限制程序员对列表的访问，这里就不再具体说明了。

6.4 图形与物理系统

对于电脑视频游戏来说，图形显示系统可以说是最为重要的方面之一，但是由于我们的研究重点是游戏中的人工智能，并且在我们游戏中，图形表现较为简单，仅仅使用简单的几何图形来表示游戏中的场地和球员。对于图形系统，可以使用较为方便的高层系统 API 并对其进行简单的封装。

在我们的游戏中，需要大量的几何与物理方面的计算，例如 Agent 运动方程的计算、各种距离计算、位置的选择等，这需要一个强大的几何物理系统来支持，本文采用了一个开源的几何与物理系统[43]，并根据具体需要做了相应的修改。

6.5 辅助系统

游戏程序是一个十分复杂和庞大软件，一个正规的商业游戏开发往往耗费几百个甚至上千个人月。对于游戏开发来说，强大的辅助系统和辅助工具无疑会大大提高开发和调试的效率，降低成本以及提高游戏的结构化和可重用性。

6.5.1 调试输入支持

对于调试和重现 bug 来说，在游戏运行时修改游戏的中的变量值的功能时非常有用的。在大型的游戏中，一般会实现一个游戏内置的控制台，调试者可以在游戏运行的时候输入控制台命令，从而改变游戏中的状态和数据[64]。在我们的游戏中，也提供了类似的功能来支持调试输入。

可以使用责任链设计模式来构建调试输入的结构。首先，设计一个输入事件接口 EventListener，该接口包含处理鼠标和键盘事件的方法，希望提供调试输入支持的类都需要实现该接口。在足球游戏中，处理调试输入的类关系如图 6-5 所示。其中每个类都实现了 EventListener 接口，如果当前的对象没有处理某个事件，

则交给下一级对象来处理。可以看到，这里实际上是两个含有公共部分的责任链。

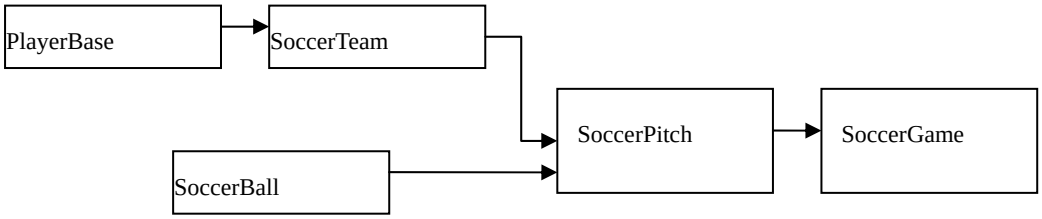

图 6-5 调试输入支持的责任链模式

6.5.2 游戏状态的保存与恢复

由于游戏软件本身的复杂性，很多时候再现一个 bug 异常困难。试图纠正一个随机出现的 bug 常会使人感到挫败，而且通常不过是浪费时间[64]。如果在游戏运行到某一时刻时能将其状态完整的保存下来，对于 bug 的再现和调试的帮助是显而易见的。

游戏状态的保存，对于面向对象的程序来说，就是对象的序列化。序列化不是什么新鲜的话题，经常使用的序列化技术就是针对不同的类定义一个结构体，然后对其中的数据进行填充后保存文件，一般的情况下，这种技术可以满足要求，但是它也存在几个明显的缺陷：

1) 不能方便的处理指针 如果对象中存在指针数据成员，简单保存指针值是完全错误的，无法预期对象在恢复时出现在相同的内存地址。
2) 结构体的字节对齐问题 当使用不同编译器编译项目时，字节对齐是令程序员非常头疼的一个问题，而且它很难被发现。
3) 变长数据的处理 当对象中含有如数组或字符串等类型的成员时，这种序列化方法就显得非常笨拙甚至会出错。

Jason Beardsley[65]提出了一种基于模板的对象序列化方法，这基本是一种全能型的解决方案，并且可以支持网络数据传输。对于我们的引擎来说，并不需要如此重量级的解决方案，可以借鉴它的思想，使用操作符重载的方式来实现对象的序列化。以 BaseGaemEntity 为例，可以通过增加两个友元操作符使该类支持 std::iostream：

```
friend std::ostream& operator<< ( std::ostream& os, BaseGameEntity & en ) ;
friend std::istream& operator>> ( std::istream& is, BaseGameEntity &en ) ;
```

6.6 辅助工具

一个完整的游戏开发过程总是需要多方面的辅助工具，例如 RTS 游戏一般需要地图编辑器，3D 场景的游戏需要场景编辑器，3D 角色需要角色编辑器等等。针对足球比赛模拟游戏，我们开发了一个可视化的阵型编辑器，用来编辑不同的阵型和球员的角色、参数等。编辑好的阵型保存在阵型文件中，游戏从阵型文件中读取相应的阵型信息。

在阵型编辑器的设计上采用了 Model/View 的结构（图 6-6），在阵型编辑器中，我们可以切换同一阵型里的不同子阵型的视图，在一个子阵型里对位置信息的更改会直接反映在另外的子阵型中。

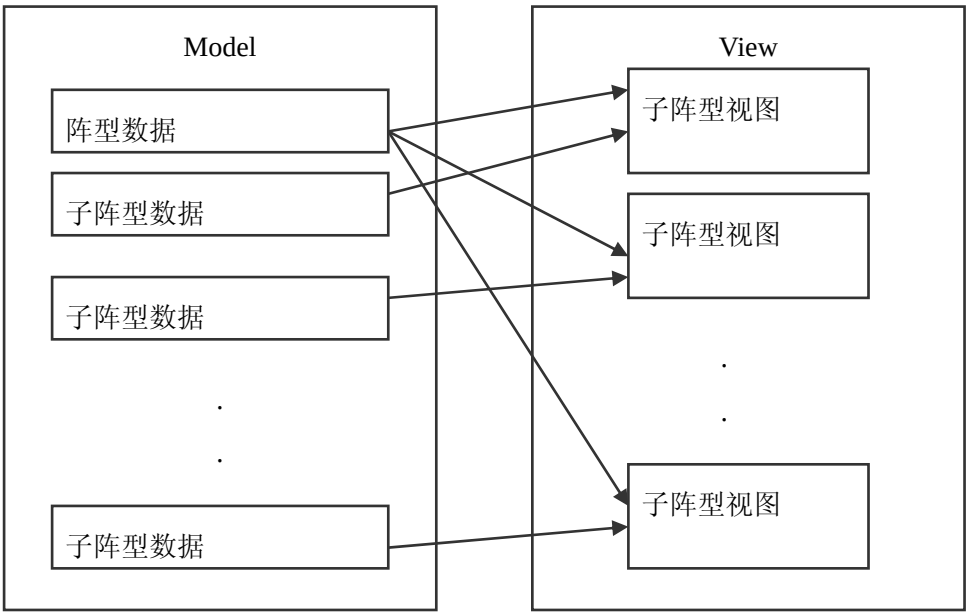

图 6-6 阵型编辑器的 Model/View 结构

6.7 实验结果

本文中的游戏 AI 引擎与游戏开发均采用 Microsoft Visual Studio 2005 为开发环境，编程语言为 C++，系统平台为 Microsoft Windows XP。游戏运行界面截图如图 6-7 与图 6-8。

结果表明，在个人技术方面，游戏中的 Agent（球员）在大多数情况下能迅速感知比赛形势，做出合理的跑位、带球、传球和射门等动作。在团队协作方面，同一球队中的 Agent 可以较好的保持比赛阵型，并在必要的时候完成团队协作以增加射门机会。

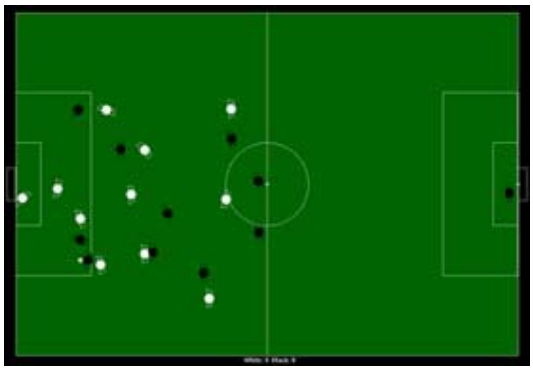

图 6-7 比赛中

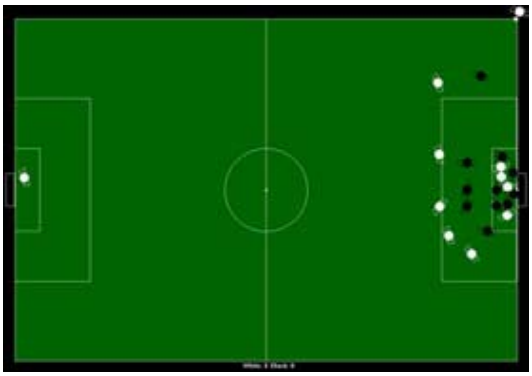

图 6-8 角球

帧率是用来衡量游戏效率的最重要因素，本游戏的平均帧率为 80 左右，最高帧率 120 左右。游戏的图形渲染采用 Windows GDI 接口，画面分辨率为 1024*768，硬件配置为 CPU: AMD Athlon 1.8G，RAM: 512M, Video Card: VIA KM400 Pro.，一般来说，对于图形系统采用 DirectX 接口的游戏，帧率达到 30 以上即可满足要求，本文采用比 DirectX 效率低若干倍的 GDI 接口，平均帧率能达到 80 以上，足以说明我们的 AI 引擎的执行效率。

6.8 本章小结

在前面的基础上，本章着重研究了一个游戏实现所需的各个方面，如游戏循环、管理与控制系统、图形系统、几何物理系统以及必不可少的辅助系统等。实现了一个完整的足球模拟游戏来验证 AI 引擎的有效性。

第七章 结论与展望

7.1 全文总结

本文给出了一个从基于多 Agent 系统的游戏 AI 引擎的设计，到游戏相关知识的集成，再到完整的游戏实现的一整套解决方案。

根据游戏引擎对执行效率和架构灵活性的要求，有针对性地设计了引擎中的多 Agent 系统模型，包括个体 Agent 的感知系统模型、运动系统模型和决策系统模型，以及多 Agent 的通讯系统模型和团队协作模型，其中提出了一种简化的黑板结构来支持团队协作。在此基础上，实现了一个基于模板的有限状态机作为 AI 引擎的框架，并通过一个全局状态提供消息处理机制，同时采用延迟消息与规整器来提高 AI 引擎的执行效率。在游戏相关知识的集成中，设计了各种足球技术、战术的表示方法，对于球员的动作决策，提出了一种不均匀单元格影响力地图的方法，并设计了相应的影响力传播算法。最后，实现了如输入控制、图形系统、实体管理等一个真实的游戏所必须的各种要素，完成了一个完整的足球模拟游戏，验证了该 AI 引擎的有效性。另外，同游戏开发的一贯作风一样，本文也尝试了大量的编程技巧和优化技术来提高游戏的效率。

本文是将多 Agent 系统用于构建通用游戏 AI 引擎的初步尝试，在充分利用多 Agent 系统优势的同时保证了引擎的执行效率。基于本 AI 引擎的足球模拟游戏展示出了令人满意的智能效果，验证了本 AI 引擎的有效性。本 AI 引擎为团体竞技游戏中的人工智能开发提供了强有力的支持，使得开发过程更加简单和快速。

7.2 下一步工作

游戏的类型多种多样，不同类型的游戏对 AI 的要求也是各不相同。本文验证了我们的 AI 引擎在团体竞技类型的游戏中的良好效果，对于其他类型的游戏如 RTS 或 RPG 等，我们没有足够的时间去尝试。另外，本文的重点工作在整个引擎与游戏的设计和实现上，对智能算法的实现没有太多深入的研究。因此，下一步的工作主要有：

1) 继续尝试其他类型的游戏，增强 AI 引擎的通用性。

2) 在 AI 引擎中集成和更多的智能算法，实现更好的智能效果。

相信随着技术的发展以及无数开发者和研究者的努力，在不远的将来，一定会出现多种各具特色的游戏 AI 引擎，基于这些引擎，也将更容易的开发出更加智能、更加有趣的游戏软件来！

参考文献

[1] 顾煜，游戏开发过程：[硕士学位论文]，上海：华东师范大学，2004

[2] http://www.it.hc360.com/scyj/2004/200404/new/0404-12.htm

[3] http://dev.gameres.com/Program/Abstract/GEanatomy1.htm

[4] http://www.blogchina.com/new/source/164.html

[5] http://www.ogre3d.org/

[6] http://www.cegui.org.uk/wiki/index.php/Main_Page

[7] http://www.ode.org/

[8] http://www.zeroc.com/ice.html

[9] Chris Fairclough, Michael Fagan, Brian Mac Namee, Research Directions for AI in Computer Games，Proceedings of the Twelfth Irish Conference on Artificial，2001

[10] Paul Tozour，智能游戏的变革，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，1~11

[11] Stuart Russell, Peter Norvig，人工智能一一种现代方法（第二版），姜哲，金奕江、张敏、杨磊等译，北京：人民邮电出版社，2004

[12] RD Keene，Man Versus Machine: Kasparov Versus Deep Blue D Goodman, - H3 Publications，1997

[13] Higgins Daniel F.，How to Achieve Lightning-Fast A*，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，99~108

[14] Rabin Steve. A * Speed Optimizations.，Mark Deloura，Game Programming Gems.，Charles River Media，2000，240~251

[15] Sid Meier’s Civilization, MicroProse,1991

[16] Mike Mika, Chris Chariaz Steve Rabin，简单低耗的路径搜索，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，115~150

[17] Dan Higgins，设计路径搜索的体系结构，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，92~99

[18] SimCity, http://www.simcity.com

[19] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E.: RoboCup: The Robot World Cup Initiative. In: Proc. of the IJCAI-95 Workshop on Entertainment and

AI/Alife. 1995

[20] P.Stone. Layered Learning in Multi-Agent Systems. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, Dec.1998

[21] K. Kostiadis and H. Hu. Essex Wizards. In RoboCup-99 Team Descriptions for the Simulation League, Linkoping University Press, 1999，17～26.

[22] K. Kostiadis and H. Hu. A Multi-Thread Approach to Simulated Soccer Agents for the RoboCup Competition. In M. Veloso, E.Pagello, and H. Kitano, editors, RoboCup-99: Robot Soccer World Cup III. Springer Verlag, 2000

[23] L.P.Kaelbling, A.R. Cassandra, and M.L.Littman. Acting Optimally in Partially Observable Stochastic Domains. In Proceedings of the Twelfth National Conference on Artifical Intelligence(AAAI-94),1994

[24] L.P.Reis and J.N.Lau. FC Portugal Team Description: RoboCup-2000 Simulation League Champion. In P.Stone, T. Balch, and G.Kraetszchmar, editors, RoboCup-2000: Robot Soccer World Cup IV, Berlin ： Springer Verlag，2001，29～40.

[25] H.D.Burkhard, M.Hannebauer, and J. Wendler. AT Humboldt-Development, Practice and Theory. In H.Kitano, editor, RoboCup-97:Robot Soccer World Cup I, Springer Verlag，1998，357～372.

[26] T.Suzuki. Team YowAI Description. In RoboCup-99 Team Descriptions for the Simulation League, Page 31-35. Linkoping University Press,1999

[27] 郭叶军，机器人足球仿真比赛中多智能体系统的构建：[硕士学位论文]，杭州：浙江大学，2004

[28] Y Jinyi, C Jiang, C Yunpeng, L Shi Architecture of TsinghuAeolus-Robocup-2001: Robot Soccer World Cup V, Springer Verlag, 2002

[29] Woodcock， Game AI: The State of the Industry, http://www.gamasutra.com/features/20001101/woodcock_01.htm

[30] John Manslow，在游戏中使用神经元网络的例子，Mark A. Deloura，游戏编程精粹 2（袁国忠，陈蔚），北京：人民邮电出版社，2003，304~309

[31] Thomas Rolfes，利用可编程图形硬件处理人工神经网络，Mark Deloura，游戏编程精粹 4（沙鹰等），北京：人民邮电出版社，2005，308～312

[32] Andre LaMothe 神经网络初探，Mark Deloura, 游戏编程精粹 1（王淑礼，张磊），北京：人民邮电出版社，2004，292~309

[33] Alex J.Champandard，神经网络的魔法，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，470～479

[34] Francois Dominic Laramee 遗传算法：进化出完美的洞穴巨人，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，463～470

[35] PJohn Manslow，学习与适应性，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，407～414

[36] Richard Evans，学习的多样性，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，415～424

[37] Janson Hutchens, Jonty Barnes 实用自然语言学习，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，442～452

[38] Black & White. Lionhead Studios/Electronic Arts, 2001 http://www.bwgame.com

[39] Michael van Lent, John Laird, Josh Buckman, Intelligent Agents in Computer Games, Joe Hartford, Steve Houchard, Kurt Steinkraus, Russ Tedrake

[40] Jeff Orkin，实战中的 12 个诀窍，Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，20～25

[41] Michael Wooldridge, 多 Agent 系统引论（石纯一，张伟，徐晋辉），北京：电子工业出版社，2003

[42] Wendy Stahler，beginning math and physics for game programmers，New Riders Publishing，2004

[43] David M. Bourg, Glenn Seemann, AI for Game Developers, O'REILLY, 2005

[44] Kevin Kaiser, 3D 碰撞检测, Mark Deloura, 游戏编程精粹 1（王淑礼，张磊），北京：人民邮电出版社，2004，346~357

[45] Erich Gamma, Richard helm, Ralph Johson, etc，Design Patterns Elements of Reuseable Object-Oriented Software，北京：机械工业出版社，2002

[46] John Hancock 一个基于效用的面向对象决策架构, Mark Deloura,游戏编程精粹 4（沙鹰等），北京：人民邮电出版社，2005，277～283

[47] Steve Rabin，设计一个通用健壮的 AI 引擎，Mark Deloura, 游戏编程精粹 1（王淑礼，张磊），北京：人民邮电出版社，2004，193～207

[48] Carver N and Lesser V.， The Evolution of Blackboard Control Architectures. CMPSCI Technical Report, 1992，92～71

[49] Damian Isla, Bruce Blumberg, 黑板体系结构, Steve Rabin，人工智能游戏编程真言（庄越挺，吴飞），北京：清华大学出版社，2005，251～259

[50] Perlin k，Goldberg A.， Improv: A system for scripting interactive actors in virtual worlds. Computer Graphics (SIGGRAPH'96 Proceedings) 1996，205～

216

[51] Dysband, Eric, “A finite State Machine Class”, Game Programming Gems, Charles River Media, Inc., 2000，237~248

[52] Charles Farris，基于函数指针的内嵌式有限状态机中，Dante Treglia, 游戏编程精粹 3（张磊），北京：人民邮电出版社，2003，226～236

[53] Remco de Boer，Jelle Kok，The Incremental Development of a Synthetic Multi-Agent System: The UvA Trilearn 2001 Robotic Soccer Simulation Team：[硕士学位论文]，Holand：University of Amsterdam，2002

[54] Paul Tozour，创建影响力地图，Mark A. Deloura，游戏编程精粹 2（袁国忠，陈蔚），北京：人民邮电出版社，2003，252~260

[55] Evans, Alex. Four Tricks for Fast Blurring in Software and hardware, http://www.gamasutra.com/features/20010209/evans_01.htm.

[56] Daniel Sánchez-Crespo Dalmau，Core Techniques and Algorithms in Game Programming，New Riders Publishing，September 08, 2003

[57] Michael Harvey，调度游戏中的事件，Dante Treglia, 游戏编程精粹 3（张磊），北京：人民邮电出版社，2003，4～11

[58] Noel Llopis，时钟：游戏的脉搏尽在掌握 ，Mark Deloura，游戏编程精粹 4（沙鹰等），北京：人民邮电出版社，2005，23～29

[59] Bilas, Scott, A Generic Handle-Based Resource Manager, Game Programming Gems, Charles River media, 2000

[60] Hawkins, Brian, Handle-Based Smart Pointers, Game Programming Gems 3, Charles River Media,2002

[61] Natalya Tatarchuk, 一个易用的对象管理器，Mark Deloura，游戏编程精粹 4（沙鹰等），北京：人民邮电出版社，2005，86～92

[62] Ben Board 等，自动列表设计模式，Dante Treglia, 游戏编程精粹 3（张磊），北京：人民邮电出版社，2003，59～63

[63] David Vandevoorde, Nicolai M. Josuttis，C++ Template – the Complete Guide，北京：中国电力出版社，2004

[64] Steve Rabin，调试游戏程序的学问，Mark Deloura，游戏编程精粹 4（沙鹰等），北京：人民邮电出版社，2005，4～10

[65] Jason Beardsley 基于模板的对象序列化，Dante Treglia, 游戏编程精粹 3（张磊），北京：人民邮电出版社，2003，465～475

发表论文和参加科研情况说明

发表的论文：

[1] Shi LS, Fu H，“A delimitative and combinatorial algorithm for discrete optimum design with different discrete sets”, Computational and Information Science, Proceedings Lecture Notes in Computer Science 3314: 443-448 2004

[2] 付恒，冯志勇，“一种利用形状描述子的选择性图像分割方法”，《电子测量技术》，2007 年 9 月

致　谢

本论文的工作是在我的导师冯志勇教授的悉心指导下完成的，冯志勇教授严谨的治学态度和科学的工作方法给了我极大的帮助和影响。在此衷心感谢三年来冯志勇老师对我的关心和指导。

冯志勇教授悉心指导我们完成了实验室的科研工作，在学习上和生活上都给予了我很大的关心和帮助，在此向冯志勇老师表示衷心的谢意。

在实验室工作及撰写论文期间，王飞、王家昉等同学对我论文中的 Agent 建模研究工作给予了热情帮助，在此向他们表达我的感激之情。

另外也感谢家人，他们的理解和支持使我能够在学校专心完成我的学业。