游戏AI开发实用指南

[葡萄牙] 麦克·达格雷斯（Micael DaGraça）著 杨奕 马遥 译

Practical Game AI Programming

- 游戏AI开发入门级实用指南，详细阐述各种类型游戏中AI的实现思路与方法
- 技术与理念并重，既涵盖游戏AI开发的各种实用技术及实践，也探讨了许多游戏设计的历史与理念

机械工业出版社
China Machine Press

Table of Contents

游戏开发与设计技术丛书

游戏AI开发实用指南

Practical Game AI Programming （葡）麦克·达格雷斯（Micael DaGraça） 著

杨奕 马遥 译

ISBN：978-7-111-58940-2

Micael DaGraça: Practical Game AI Programming（ISBN: 978-1-78712-281-9）.

目录

译者序

在游戏开发领域，不乏很多优秀的引擎技术书籍，但着重介绍Gameplay方面的技术书籍却少之又少，而游戏AI恰恰又是Gameplay中非常重要的一个环节。故当读到这本专门介绍游戏AI的书籍时，我们感到非常开心，便不遗余力地翻译了本书。

本书详述了各种类型的游戏中AI的实现思路与方法。其中涵盖了可能性图、概率图、状态机等基本方法，也谈到了环境交互、寻路、动画等游戏AI必要的组成部分，最后还介绍了群体交互、碰撞避免、感知等较高层次的设计方法。

本书的特色是技术与理念并重。很多时候开发游戏AI并不是一项纯技术性的工作，开发者如果能对游戏玩法以及玩家心理感受有更好的理解，那么做出来的AI效果肯定能更上一层楼。有时一些很简单的AI逻辑也能让玩家津津乐道——比如经典的《吃豆人》，而复杂的AI要和游戏玩法紧密结合才能实现激动人心的效果。

本书的代码部分是依托于Unity引擎来写的，但本书并没有介绍Unity引擎的基础知识，所以便需要读者有一定程度的C#或Unity开发经验。当然，本书内容依旧是循序渐进的，读者不必担心过于复杂。如果你正对游戏中AI的实现而烦恼，本书无疑会是非常不错的选择。

本书的作者是一位独立游戏开发者，故书中除了对技术话题的探讨外，还涉及许多游戏设计的历史与理念。审校者Davide Aversa也是一位AI、机器学习以及游戏开发领域的权威学者，这为本书的技术部分提供了有力保障。

为了帮助大家更好地学习，我们还在知乎专栏“游戏开发入门指南”中补充了更多关于游戏AI的内容。希望通过这个专栏，可以让大家进一步掌握游戏AI开发的关键性技术。

最后，我们想在这里衷心感谢机械工业出版社华章公司的伙伴们在翻译过程中给予的帮助。同时也感谢我们的家人和朋友给予的支持，正因为有你们的帮助和支持，我们才能够顺利完成本书的翻译。还要特别感谢我们的同事李天颀，在后期大力参与我们的校对工作，其良好的英文素养与游戏经历大大推进了我们的翻译进度。

由于译者水平有限，书中恐有疏漏和错误之处，还请读者指正。也希望本书能够帮助更多的朋友开发出有趣的游戏。

杨奕　马遥

2017年9月于成都

前言

游戏开发对有些人来说充满了激情，我相信这是因为，我们可以创造一个完全由我们想象出来的世界。这有点像开创一个新的世界，我们放置的AI角色就像是这个世界的居民，他们生活在我们创造的这个世界里。我们可以自由地想象他们的行为，基于想象创造一个社会体系，创造一个甜美又温柔的角色，也可以创造一个前所未有的恶魔——这种可能性是无限的，这也是为什么我们总是冒出新的游戏创意来。无论我们打算做什么类型的游戏，这个世界和这些角色都是我们将会看到的基本要素，我们的游戏也因此独一无二。理想情况下，我们应当能够将脑海中所有的东西都原封不动地创造出来。本书就是想达到这个目的——让所有人都能够实现自己的想法，而不应当约束我们的想象力，因此本书将会涵盖创造人工智能角色的基础。当你读完以后，我们可以继续深入探索这些你学过的主题，创造出完全符合我们想象的AI角色。

本书内容

第1章是对电子游戏产业与游戏AI的综述。

第2章重点介绍如何创建和使用AI的可能性图和概率图。

第3章描述了怎样对AI角色创建一系列必要的规则以实现其目标。

第4章聚焦于游戏中的角色与他们所处环境之间的交互。

第5章展示了在游戏中实现动画的最佳实践。

第6章主要讨论实现实时计算的AI移动方案的最佳实践。

第7章主要介绍使用A*算法寻找最短路径。

第8章主要讲述在当同一场景中有大量角色时，AI应该如何表现。

第9章探讨AI的预期，事先知道当到达某个位置或面对某个问题它们将会做什么。

第10章主要讨论创建潜入类游戏的技术：感知系统。

读前准备

推荐安装一个使用C#语言的游戏引擎（Unity3D有免费版本，本书的例子中也使用了它）。

读者对象

本书面向的读者是这些开发者：他们已经用C#创作了一个游戏并且在探索用AI扩展游戏内容，从而创建具有自主行为的群体、敌人或是盟友。

下载示例代码

你可以从http://www.packtpub.com通过个人账号下载你所购买书籍的样例源码。如果你在其他地方购买了本书，你可以访问http://www.packtpub.com/support并注册账户，相关文档就会直接发送到您的邮箱中。

你也可以访问华章图书官网http://www.hzbook.com，通过注册并登录个人账号，下载本书的源代码。

下载本书彩图

我们还提供了一份具有彩色插图的PDF文件，包含了书中的屏幕截图和图表的彩色版。这些彩图会帮助你更好地理解程序输出的变化。可以从以下网址下载：https://www.packtpub.com/sites/default/files/downloads/PracticalGameAIProgramming_ColorImages.pdf。

关于作者

Micael DaGraa是一位游戏设计师以及AR开发者，现居葡萄牙波尔图市。他曾为多个游戏工作室工作，致力于创建与众不同的独立游戏和交互式应用程序。

Micael的童年伴随电子游戏一起长大，对游戏的热情也从未消去过。因此，在他后来的生活中，他决定学习如何制作游戏。一开始，他没有任何编程或3D动画方面的知识，他便从制作一些简单的游戏开始，在这个过程中每次都学到了更多的游戏制作经验。当游戏可以正常运行并且变得有趣时，他开始计划与一个老朋友合作发布一款游戏。Micael负责游戏的技术方面，确保游戏可以按照预期进行开发，而他的朋友则负责游戏的所有美术工作。最终，游戏成功发布，并获得了其他独立游戏开发者的一些积极反馈。由于游戏产生了一些收入，成为游戏设计师的梦想也随之变成了现实。

现在，Micael为其他工作室工作，帮助他人实现他们的游戏创意，并整合了一家专注于制作游戏以及开发健康类App的公司。尽管他没有时间继续从事个人项目，但他还有一些在朋友的帮助下仍在开发中的游戏项目。

“我要感谢父母多年来给予我无条件的支持，因为没有他们，我不可能成为一个游戏设计师；非常感谢我的妹妹Alexandrina，感谢她在我最需要帮助的时候帮助我，让我在她的办公室制作游戏，从而使我可以开始做游戏设计师。感谢我的老朋友兼老对手Vicente，逼着我不断突破自己的极限，使我成为一个更好的专业人士。感谢我的女朋友Marta，她的微笑总是让我很开心，她处理好了生活中的每件事，让我可以专注于我的工作。最后，我想把本书献给我的爷爷，是他激励我成为今天的自己。”

第1章　不同的问题需要不同的解决方案

1.1　游戏AI解决方案的历史简述

为了更好地理解游戏开发者如何战胜当前正在面对的问题，我们需要探究一点电子游戏开发的历史，看看当时那些非常重要的问题以及它们的解决方案。它们中的一些方法是相当超前的，实际上改变了整个电子游戏设计的历史，甚至直到今天我们仍然在使用相同的方法，去创造独特且有趣的游戏。

在讨论游戏AI时，最值得一提的标志性事件之一总是AI与人类的国际象棋对决。这也是开始研究人工智能时的理想选择，因为国际象棋通常需要很多思考和预先计划，这对当时的电脑来说还是无法做到的事情。AI为了顺利地进行比赛甚至赢得胜利，还需要一些人类特性，因此，首先是使电脑能够处理游戏规则，并且为了能够实现最终目标——将死对方获胜，要能自己进行适当的思考，为下一步做出良好的判断。问题在于国际象棋有很多的可能性，因此，即便电脑有一个完美的策略来赢得比赛，也需要对这个既有策略进行重新计算、适应、改变，甚至是在首选策略发生错误时，创造新的策略。

人类每次下棋都是全然不同的，这让程序员输入所有可能性到计算机中以获胜的这种做法，变成一项巨大无比的任务。所以，为全部的可能性编写代码不是一个可行的方案。正因如此，程序员们需要重新思考这个问题。接着，某一天他们终于想出了一个更好的办法，那就是让计算机自己决定每一轮怎么做，去选择看上去最好的选项。这种方法让计算机能够适应比赛中的任何一种情况。不过，这带来另一个问题——计算机只能想到短期有利的走法，而不是建立计划以便在将来的步骤中击败人类。所以人类想要打败这种计算机很容易，但是至少我们让事情前进了一步。直到几十年以后，才有人定义了人工智能（AI）这个词语，他尝试做出了一台具有打败人类玩家的能力的计算机，解决了我们前面提到的困扰了许多研究者的问题。Arthur Samuel正是那个人，他创造了能够自我学习而且能记住所有可能组合的计算机。这样，不必有任何的人工干预，计算机就能自我思考，这是一个巨大的进步，即使以今天的标准来看依然让人印象深刻。

1.2 电子游戏中的敌人AI

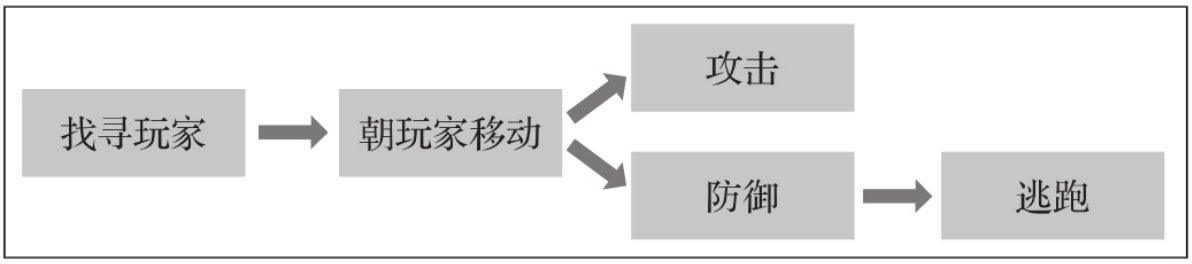
现在，让我们来到电子游戏产业中，分析一下第一个敌人和障碍物是如何编写的，以及是否与我们现在的做法有所差别。下面让我们试着找出答案。

在单机游戏中，敌人AI第一次出现是在20世纪70年代，接着很快，一些游戏开始提高对电子游戏AI定义的质量和期望。其中的一些游戏开始发布到街机上，如来自Taito的《极速赛车手》（Speed Race，一个竞速电子游戏），以及来自Atari的《小鸭子大冒险》（Qwak，一只鸭子使用光枪狩猎的游戏）和《追击》（Pursuit，一个战斗机游戏）。其他值得关注的例子是个人电脑上的文字游戏，例如《捕获狮头象》（Hunt The Wumpus）和《星际旅行》（Star Trek），这些游戏都有着敌人。这些游戏之所以如此令人愉快，正是由于AI敌人没有像以前那样做出固定的反应，它们有着随机的元素，同时还混合了预先存储的模式，从而变得无法预测，因此，在你每次玩游戏时都有着独特的体验。无论如何，当时微处理器的加入极大地扩展了程序员的能力。《太空入侵者》（Space Invaders）带来的移动模式和《小蜜蜂》（Galaxian）改进增加的更多变化，使得AI愈加复杂。《吃豆人》（PAC-MAN）后来也带来了迷宫类型上的移动模式。

《吃豆人》在AI设计上带来的影响与其在游戏自身上带来的影响同样重要，这个经典的街机游戏使得玩家相信游戏中的敌人真的在追逐他们，且不是以粗糙的没有任何谋略的方式。鬼魂们以不同的方式追逐玩家（或躲避玩家），就好像它们具有独特的个性一样。这给予了人们一种错觉——与他们在对阵的是四、五个完全不同的鬼魂，而不是同一个电脑敌人的多个副本。

之后，《空手道冠军》（Karate Champ）引入了第一个AI战斗角色，《勇者斗恶龙》（Dragon Quest）引入了RPG类型游戏的战术系统。多年来，探索人工智能并用其创造独特游戏概念的游戏名单不断增多，所有这些都来自同一个问题：我们如何使电脑能够在游戏中战胜人类？

上面提及的所有游戏都是不同类型的，它们都有自己独特的风格，但是它们的AI都使用着同样的方法——有限状态机（FSM）。在这里，程序员输入好所有电脑挑战玩家所需的行为，就好像第一次下棋的电脑一样。程序员准确地定义了电脑应该如何在不同的场合有序地移动、躲避、攻击，或者执行任何其他挑战玩家的行为，即使在最新的大制作游戏中，这种方法也同样被使用。


找寻玩家
朝玩家移动
攻击
防御
逃跑

1.3 从简单到聪明的类人AI

程序员在开发AI角色时面临许多挑战，不过其中最大的挑战是如何使得AI的动作和行为能够与玩家当前正在做或未来将要做的事相关。之所以存在这种困难是因为AI是按照预定的状态来编程的，其中使用了概率图和可能性图来有序地根据玩家的动作和行为调整自身的动作和行为。如果程序员拓展了AI决策的可能性，这一技术将变得非常复杂，就好像是国际象棋中的所有可能情况发生在了游戏中。

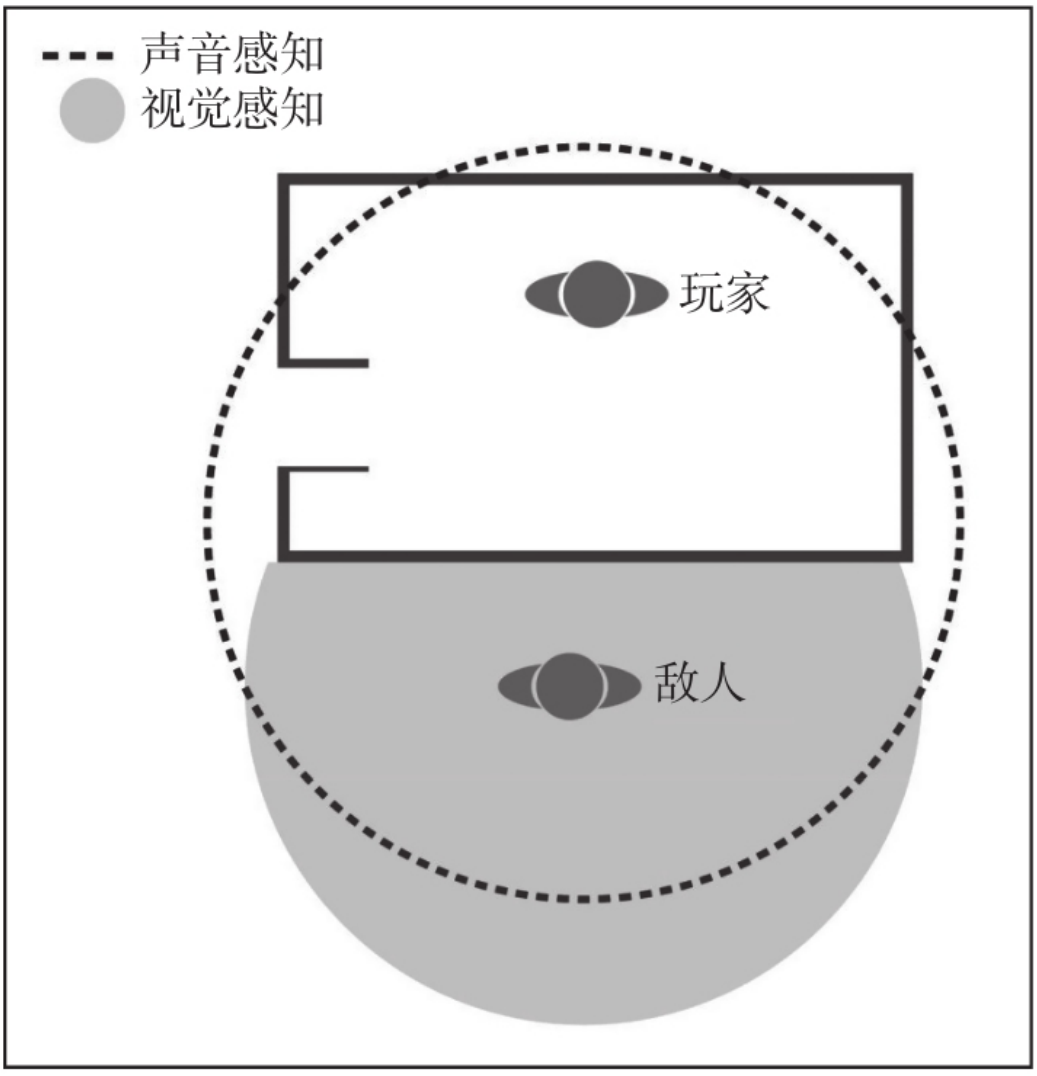
这对于程序员来说是一项巨大的任务，因为有必要确定玩家可以做什么以及AI如何对玩家的每一个动作做出反应，而这需要大量的CPU计算能力。为了克服这个挑战，程序员开始混合使用可能性图与概率图，以及执行其他技术，让AI能够自己决定如何响应玩家的动作。这些因素在开发AI时应该被慎重考虑，因为正如我们发现的那样，这些因素提高了游戏的质量。

游戏不断进化，玩家对优质游戏的期望也变得更加迫切，这不仅体现在视觉质量，还包括敌人和盟军的AI能力。考虑到玩家的期望，在实现新的游戏时，程序员开始为每个角色添加更多的状态，创造新的可能性和更多有趣的敌人以及重要的友军角色，这意味着玩家需要做更多的事情，并且这些新创造的功能还帮助其重新定义了新的游戏类型。当然，所有这些新增的事物是可能实现的，因为技术在不断改进，开发者可以在电子游戏中探索更多的人工智能技术。一个非常值得一提的例子是《合金装备》（Metal Gear Solid），这款游戏通过在当时流行的普通射击游戏上实现了潜行要素，给电子游戏产业带来了新的游戏类型。然而由于当时的硬件限制，小岛秀夫（Hideo Kojima）预期中的这些要素并没有得到完全体现。从第三代到第五代游戏主机设备上，科乐美（Konami）和小岛秀夫使用了相同的标题，不过这时候游戏不再受到硬件的制约，有了更多的交互、可能性以及游戏中AI敌人的行为，使这款游戏成为了电子游戏史中非常成功且重要的一款。我们也很容易看到《合金装备》在问世后对后辈游戏造成的影响。

Metal Gear Solid-Sony Playstation 1

1.4 视觉和声音的感知

先前截图中的游戏实现了AI敌人对视觉和声音的感知，我们会在本书后面的章节中探索这个功能的细节，这个功能造就了我们今天所知道的潜行游戏。从电子游戏产业起步之初就已经开始使用路径查找和有限状态机了。但是他们也会创造新的特性，例如场景交互，导航行为，视觉/声音感知，以及AI互动。很多东西在当时不存在，但是今天这些特性广泛地被不同类型的游戏使用，例如体育、赛车、格斗以及第一人称射击（FPS）游戏，下



面也会介绍到。

在游戏设计迈出巨大的一步之后，开发者仍然面临其他问题，或者我应该说，这些新的可能性带来了更多的问题，因为它们并不完美。AI仍然没有像真正的人那样做出反应，不仅在潜行游戏中是这样，在其他所有类型的游戏中也都是这样，特别是当需要改善这些AI以使得游戏感觉更加逼真时，还有许多额外的要素是必须实现的。

我们讨论一下体育游戏，尤其是那些试图模拟真实世界团队行为的游戏，例如足球和篮球。玩家的交互并不是我们唯一需要关注的事情，很早前我们就已经没有面对国际象棋那种1V1的情况了。现在我们想要的更多，看到其他游戏有着真实的AI行为，体育狂热者们开始要求他们喜欢的游戏也应该有那样棒的AI，毕竟这些游戏是基于真实世界的事物，因此AI应该尽可能真实地做出反应。

在这一点上，开发者和游戏设计师开始考虑AI互动本身，像《吃豆人》中的敌人会使玩家得到这样一种印象，即游戏中每个角色都有自我思考，对其他角色的反应都有不同。如果我们仔细分析，体育游戏中的AI和FPS或者RTS游戏中的AI是非常接近的，都是使用不同的动画状态，通用的移动、交互、单独决策，以及战术和集体决策。因此，体育游戏与其他在AI开发方面已经非常成熟的游戏类型一样，也可以拥有足够真实的AI，这不足为奇。不过，只有体育游戏在当时面临几个问题：如何使这么多同屏角色产生不同的反应，但同时又能让这些角色共同努力实现相同的目标。考虑到这个问题，开发者开始改进每一个角色的独立行为，不仅仅包括玩家的对立阵营AI，还有与玩家同阵营的AI。有限状态机在这里又一次成为人工智能的一个重要组成部分，而且，潜行游戏中的感知和预测对于创造尽可能真实的体育游戏也非常有帮助。电脑需要计算玩家正在做什么，球在哪里，以及所有这些东西的坐标，且还得让同阵营的AI看起来像是在团队协作。在这些数量众多的同屏角色的身上加入潜行游戏中的新特性，可以改进已有的体育类型游戏，多年来这种做法已经获得了广泛的应用。这也有助于让我们明白，我们几乎可以使用相同的方法来创造任何类型的游戏，即使它们看起来完全不同。我们看到电脑AI下棋时使用的核心原则，其对于30年后发行的体育游戏仍然是有价值的。

让我们继续最后一个例子，这个例子对于如何使得AI角色的行为更加真实有着非常高的价值：这个游戏是Monolith Productions开发的《极度恐慌》（F.E.A.R.）。使这个游戏在人工智能方面显得相当特别的是敌人角色之间的对话。虽然从技术角度来看这并不是什么进步，但它确实有助于展示角色AI的所有开发工作，假如AI之间没有这么说话，便不会使游戏显得与众不同。这在创建具有真实感的AI时是一个非常重要的考量因素，AI的这种行为会让人觉得其自身的反应以及其与玩家的交互行为跟人类非常接近，使人产生错觉。对话不仅有助于创造类人的气氛，还有助于让角色显得更加立体，如果没有这些对话的话，玩家很难注意到这些。当AI第一次察觉到玩家时，它会大喊发现玩家了；当AI视野内丢失玩家时，它也会喊出来。当AI小队试图搜寻或伏击玩家时，AI小队会进行讨论，让玩家想象敌人真的有能力去制定策略来迎战玩家。为什么这点这么重要？因为如果这些角色只有数字和数学公式般的机械行为，那么它们会按照既定的做，但是并没有人的特性，为了使这些AI角色更加人性化，就需要让它们有一些过失、错误以及对话，这可以让玩家觉得自己不是在与苍白的机器对战。

整个电子游戏的历史中人工智能距离完美还很遥远，我们可能需要花费几十年时间去一点点地改进。从20世纪50年代初期到今天，我们已经取得了很大的突破，所以不要害怕探索，你应该去学习、融合，甚至是打碎重建一些事物，以得到不同的结果。因为伟大的游戏在过去都这么做了，且它们获得了巨大的成功。

1.5 总结

在这一章中，你学习了AI对电子游戏历史的影响，了解了所有的一切都是如何从简单的想法开始到有电脑AI来和人类在传统游戏中竞争比赛，接着是如何演变到了电子游戏的世界。你还了解了自第一天来出现的挑战和困难，以及程序员不停地解决问题，但还是会有相同的问题产生的过程。在下一章节，我们将从细节开始，讨论一些更加实用的技术，以及这些技术在过去、现在和未来游戏中的争论和演化。

第2章　可能性图与概率图

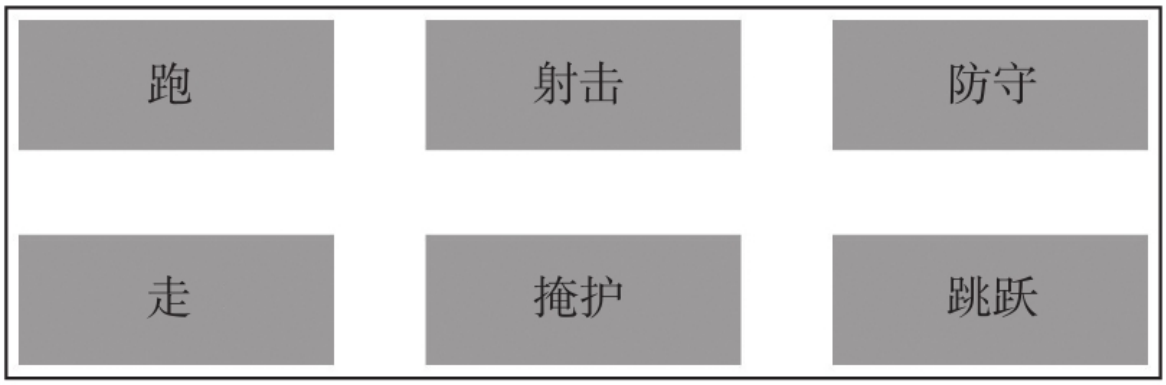
本章将会讨论可能性图与概率图，理解如何使用以及何时使用它们。我们还会学习一种创建AI的最佳实践，它不仅能够回应玩家的行为，同时还能做出最优选择，正如我们正在探索的问题：创建一个能够像人类一样做决策的角色。

就像我们前面看到的，电子游戏往往非常依赖于提前确定AI在不同情景下的行为，这些情景可能是游戏本身设计的，也可能是由玩家的行动造成的。这种方法的应用从第一天开始一直持续到现在，使得可能性图与概率图变成了一种用于创造高质量AI角色的极其重要的方法。在解释每种图有什么作用的细节，以及演示用这两种图开发出好的AI行为之前，最好大致地知道什么是可能性图与概率图、知道何时何地使用它们。

作为玩家，我们倾向于整体性地享受作品，怀着热情，全身心投入体验游戏的每个部分，同时忘记了游戏技术性的那方面。基于这个理由，我们有时会忘了即便游戏中最简单的事件也是按照预定的设计发生的，而且在事件发生的那一刻背后，有许多的考虑与规划。我们常听到：事出必有因，这个词也可以用在电子游戏里。从你按下游戏开始键的一刻起，一直到你打出最后一套精彩的连招消灭了最终BOSS，一切都是按照计划发生的，对程序员来说，需要输入游戏中所有可能发生的情况。如果你按下A键游戏角色跳了起来，这是因为设定就是如此。同样的事情对游戏里具有AI的敌人或盟友也是一样，当它们的一些行为消灭了你或是帮助了你，这种行为也是需要编程来实现的，要做到这一点，要用到状态。

2.1 游戏状态

要理解如何创建可能性图与概率图，我们需要先知道它们的基本组成部分，即游戏状态，简称状态。游戏状态，就是游戏中的预定动作，它按场合定义，并且可以应用于玩家或敌人角色。有些行为很简单，例如跑、跳跃或攻击，也可以再扩展一点，例如当角色在空中且无法攻击的状态，或者角色魔法量低且无法释放魔法攻击的状态。这些例子里，角



色要么从一个状态切换到另一个状态，要么由于正处在另一个状态中而无法切换状态。

2.2 可能性图

现在深入看看第1章的例子里的可能性图，从国际象棋到《合金装备》游戏。如我们所见，这是一项当下仍然在使用的技术，而且不用它的话我们几乎不可能做出游戏AI来。

顾名思义，可能性图允许程序员定义游戏中玩家或AI角色可以采用的行为。所有游戏中可以发生的事情需要事先设计并编码，但是当你允许角色做许多事情时会怎么样呢——角色能够同时做这些事情吗？假设游戏有不同的关卡，角色在所有的关卡都按同样的方式做出反应吗？除了允许或者限制所有这些可能的行为之外，我们还需要思考游戏中可能发生的情景，当你把所有这些放在一起，就称之为可能性图。

2.2.1 怎样使用可能性图

让我们看一个常规的第一人称射击（FPS）游戏中的简单例子，为此我们会使用之前图中所展示的状态。

想象一下我们是游戏中的敌人角色，目标是射击玩家并杀死他，只能使用走、跑、掩护、跳跃、射击和防守。我们要考虑到，玩家会尽全力杀死敌人，并且可能会出现各种各样的情景。我们从基础开始——从一个点走到另一个点来守卫地盘，当玩家靠近我们地盘的时候，我们的目标就从守卫地盘变成了更明确的目标，即杀死玩家。我们接下来做什么？射击？向玩家跑过去并从近处射击他？寻找掩护并等待玩家靠近？如果玩家先发现了我们准备射击我们怎么办？可能会发生很多情况，不过很多情况可以只用几个状态来应对。列举所有可能的情况，并计划在每种情况下我们应当怎样行动或者应对玩家。在这个游戏中，我会选择以下几种例子：

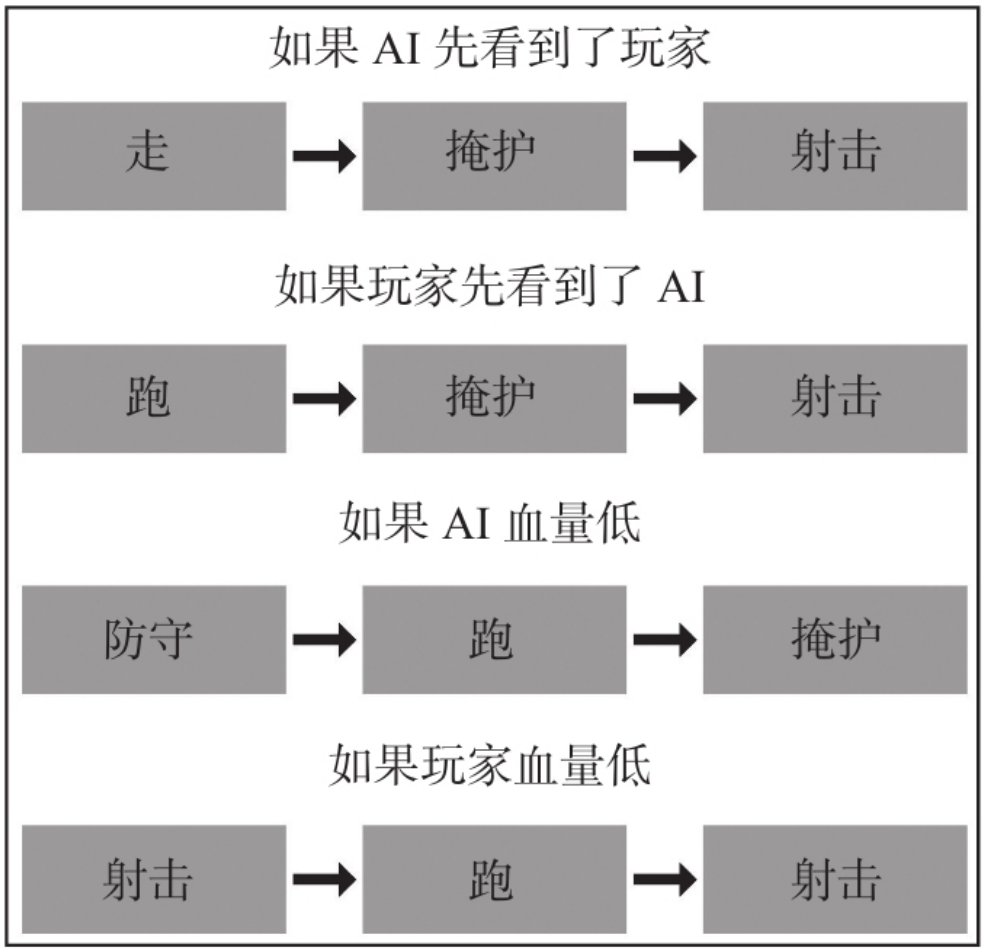
· 慢慢走到掩护位置，等待玩家靠近，然后射击他。

· 跑到掩体后面，从那里射击。

· 一边防御（远离被子弹射击的位置）一边跑向掩护位置。

· 与玩家对射，跑向他同时持续射击。

根据要制作的游戏类型，可以使用同样的状态，然后调整为不同的样式。还需要考虑正在制作的角色的个性。如果它是个机器人，那么它应该不怕一直对着玩家开火，即便是被玩家消灭的可能性高达99%；另一种情况，如果这是一个没有经验的新兵，它可能会非常害怕被击中并立即跑向掩体。像这样，可能的情况可以通过修改角色的个性一直扩展下去。


如果 AI 先看到了玩家
走
掩护
射击
如果玩家先看到了 AI
跑
掩护
射击
如果 AI 血量低
防守
跑
掩护
如果玩家血量低
射击
跑
射击

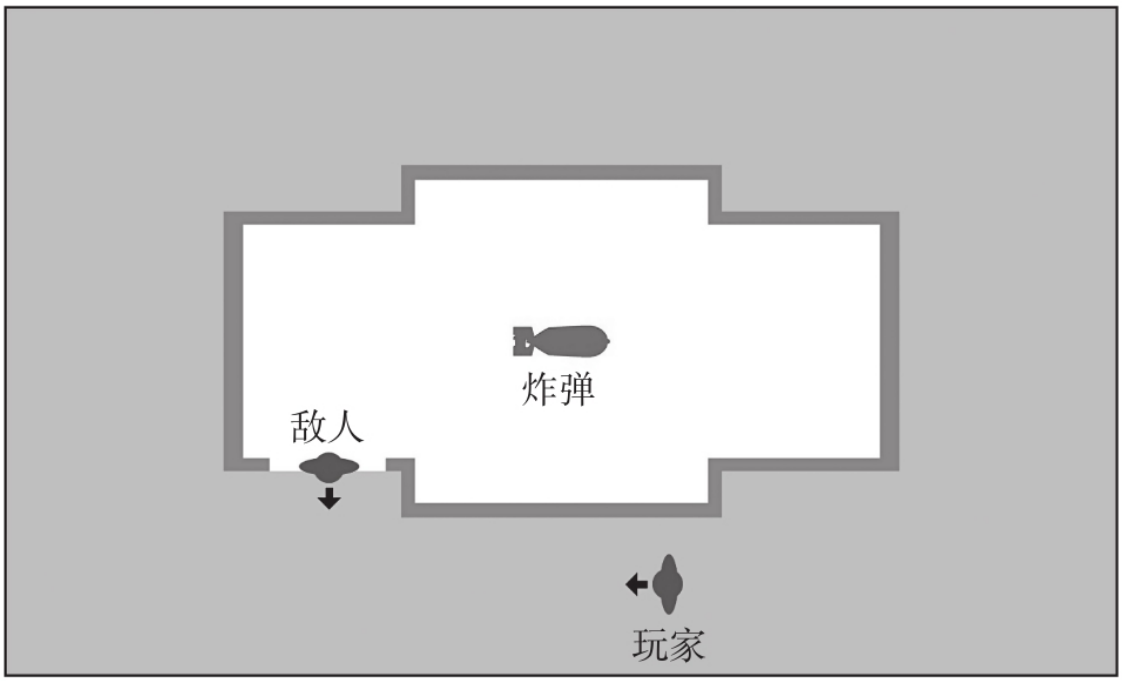
2.2.2 准备一个可能性图（FPS游戏）

到这里，我们已经理解了可能性图是什么，怎样用它来创建能够根据不同状况做出反应的AI角色。现在创建一个实际的例子，给我们的AI角色编程，让它成功击败玩家。为实现这个例子，我会使用两个模型，一个代表我们需要编程的AI敌人，另一个代表玩家。

我们将建立一个常规的例子：AI角色正在保护一幢建筑物的入口，玩家需要进入建筑物，然后解除炸弹以完成本关卡。假设玩家角色已经编码设计完毕了，接下来要关注具有

AI的敌人角色，见下面的截图。

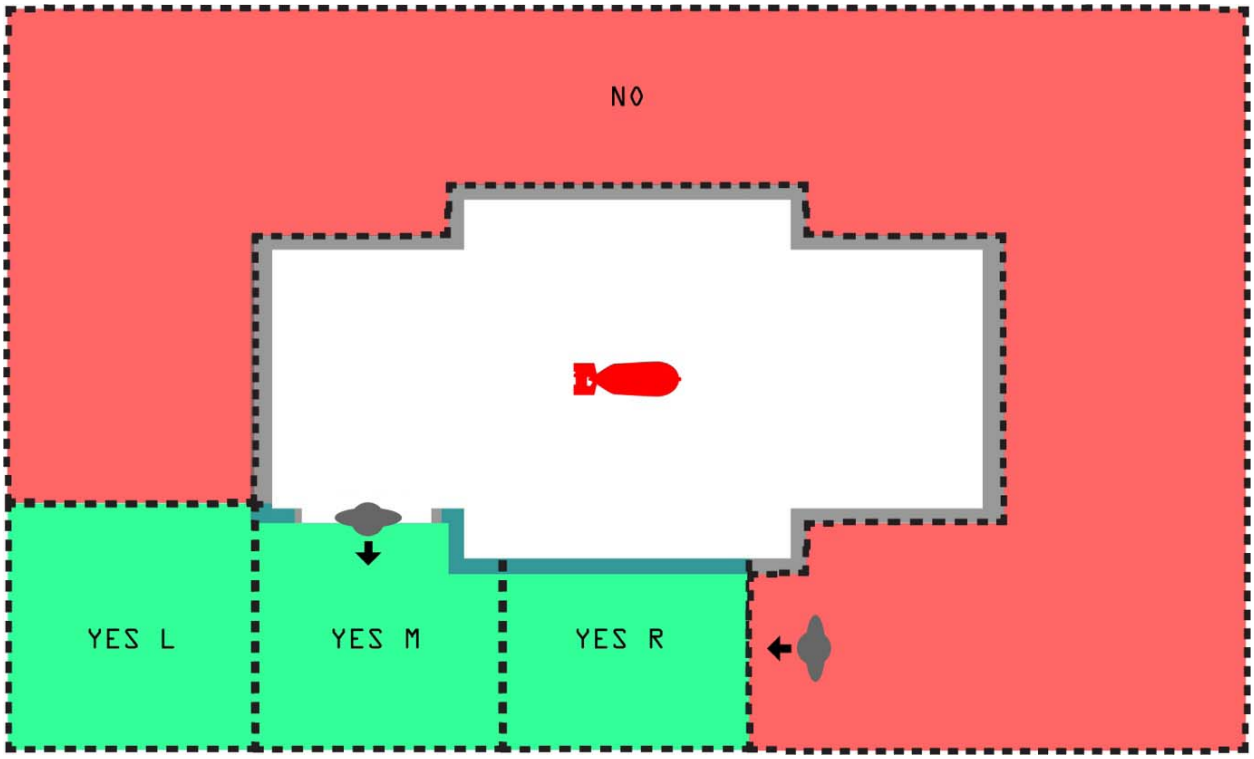
在写第一行代码之前，我们需要思考可能发生的情景，以及AI应该怎样做出反应。首先，简化一下场景，把场景变成简单的2D视图以供参考，以便确定距离和其他相关的参数。



简化了场景之后，接着规划所有可能的情况。玩家可以在建筑物周围移动。注意，只有一个入口，并且入口被AI敌人保护着。途中箭头代表角色面对的方向，方向是我们规划的重点之一。

2.2.3 创建一个可能性图（FPS游戏）

稍后会学习如何创建一个有感知的AI角色，现在只用简单的布尔类型变量来代表玩家是否靠近我们的位置，以及玩家面对的方向。考虑到以上几点，把图像切割成几个触发区域，来定义什么时候AI敌人应当做出反应。

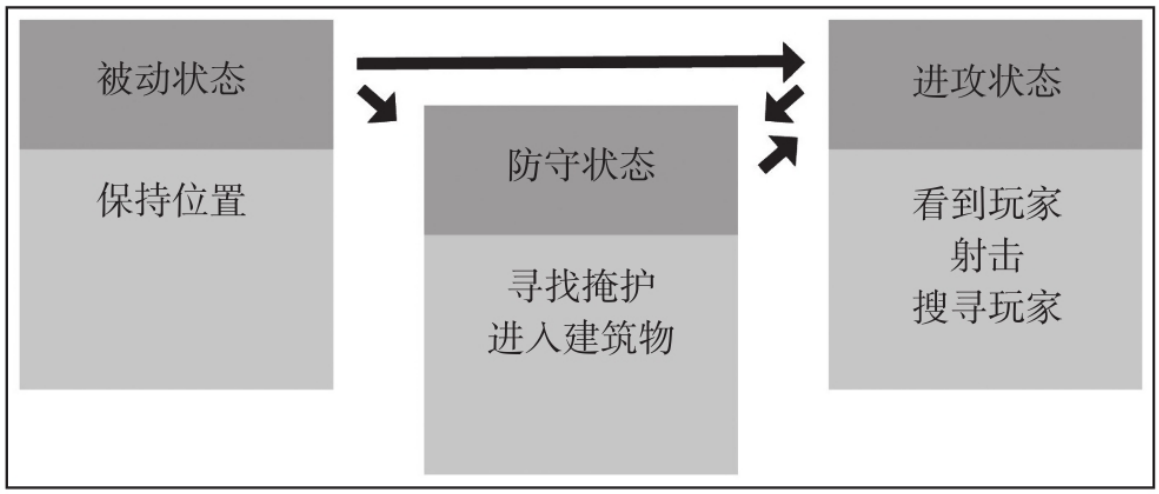


标记为YES的区域会触发AI改变行为，从被动状态变为攻击性状态。标记为NO的区域不会对AI的行为造成直接影响。我把YES区域切成了三部分，因为针对玩家的不同位置，AI角色应当做出不同的反应。如果玩家从右侧接近（YES R区域），敌人有一堵墙可以用来掩护；如果玩家从左侧接近（YES L区域），就没法使用那堵墙了；如果玩家从中间接近（YES M区域），AI角色可以简单地后退到建筑物里即可。

下面准备一下AI敌人的脚本。在这个例子里，我们会使用C#语言，但是你也可以将脚本修改为任何你喜欢的编程语言，其实思路是一样的。将要用到的变量有Health、statePassive、stateAggressive和stateDefensive：

```
public class Enemy : MonoBehaviour {
private int Health = 100;
private bool statePassive;
private bool stateAggressive;
private bool stateDefensive;
// Use this for initialisation
void Start () {
}
// Update is called once per frame
void Update () {
}
}
```

既然我们已经了解了制作AI的基本信息，接下来就要想想什么时候用到这些状态以及AI如何选择以下三种状态之一。为此，我们将使用一个可能性图。我们已经知道了触发角色行动的区域，并且确定了有三种状态。所以是时候来规划状态转移以及AI对玩家行为和位置的反应了。



AI敌人可以从被动状态（PASSIVE）

切换到防守状态（DEFENSIVE）或是进攻状态（AGGERESSIVE），可以从进攻状态切换到防守状态，还可以从防守状态切换到进攻状态，但是一旦AI角色知道了玩家就在附近，它就再也不会回到被动状态了。

2.3 定义状态

接下来定义每一种状态由什么触发，还有AI角色怎样在不同的情景下选择不同的状态。被动状态是角色默认的状态，游戏将从默认状态开始，直到玩家靠近它才变化。防守状态在两种情况下被采用——玩家从右侧接近的情况，以及已经遭遇玩家而且血量较低的情况。最后，当玩家从左侧接近或者已经到达中间区域时，会触发进攻状态。

```
public class Enemy : MonoBehaviour {
private int Health = 100;
private bool statePassive;
private bool stateAggressive;
private bool stateDefensive;
private bool triggerL;
private bool triggerR;
private bool triggerM;
// Use this for initialisation
void Start () {
 statePassive = true;
}
// Update is called once per frame
void Update () {
 // The AI will remain passive until an interaction with the player occurs
 if(Health == 100 && triggerL == false && triggerR == false && triggerM
 == false)
 {
  statePassive = true;
  stateAggressive = false;
  stateDefensive = false;
 }
 // The AI will shift to the defensive mode if player comes from the
 right side or if the AI is below 20 HP
 if(Health<= 100 && triggerR == true || Health<= 20)
 {
  statePassive = false;
  stateAggressive = false;
  stateDefensive = true;
 }
 // The AI will shift to the aggressive mode if player comes from the
 left side or it's on the middle and AI is above 20HP
 if(Health> 20 && triggerL == true || Health> 20 && triggerM == true)
 {
  statePassive = false;
  stateAggressive = true;
  stateDefensive = false;
  }
 }
}
```

上面添加了三个触发变量triggerL、triggerR和triggerM，还定义了AI何时应当从一个状态切换到另一个状态。到这里为止，在游戏过程中因玩家位置不同而形成的各种情景下，敌人角色已经知道了应该怎么做。

现在需要确定每种状态下具体应当做什么，这才是进攻状态区别于防守状态的原因。对当前这个具体的敌人角色来说，它的主要功能是保护建筑物的入口，我们希望它能坚守岗位而永远不会跑到玩家后面去。AI并不知道入侵者是否只有一个，所以不能冒险只去追逐一个敌人，而使得建筑物入口失守。这种设计让敌人的行为显得更真实一些。当敌人角色感觉到快要失败或者快要死亡的时候，会切换到防守状态，当利用建筑物本身来对付玩家更有优势的时候，也会如此。最后，当AI角色发现明显的优势可以乘机消灭玩家或是别无选择的时候，会切换到进攻状态。

2.3.1 防守状态

下面开始看看玩家从右侧接近而敌人已经发现他的情况。我们希望AI充分利用墙的优势来保护自己，给玩家制造点麻烦，同时也表现得更像人类，而不是简单地开火射击。敌人会朝着墙向前移动，然后停在那里射击玩家，直到玩家到达墙的位置。

敌人会从被动状态切换到防守状态而不是进攻状态，仅仅是因为这样会带给它相比玩家多一点优势。在初次遭遇时采取防守姿态让AI展示出来一些个性，这在让电脑角色显得更加可信方面非常重要。后面的章节会介绍怎样使用环境来帮助我们更深层次地定义AI角色：

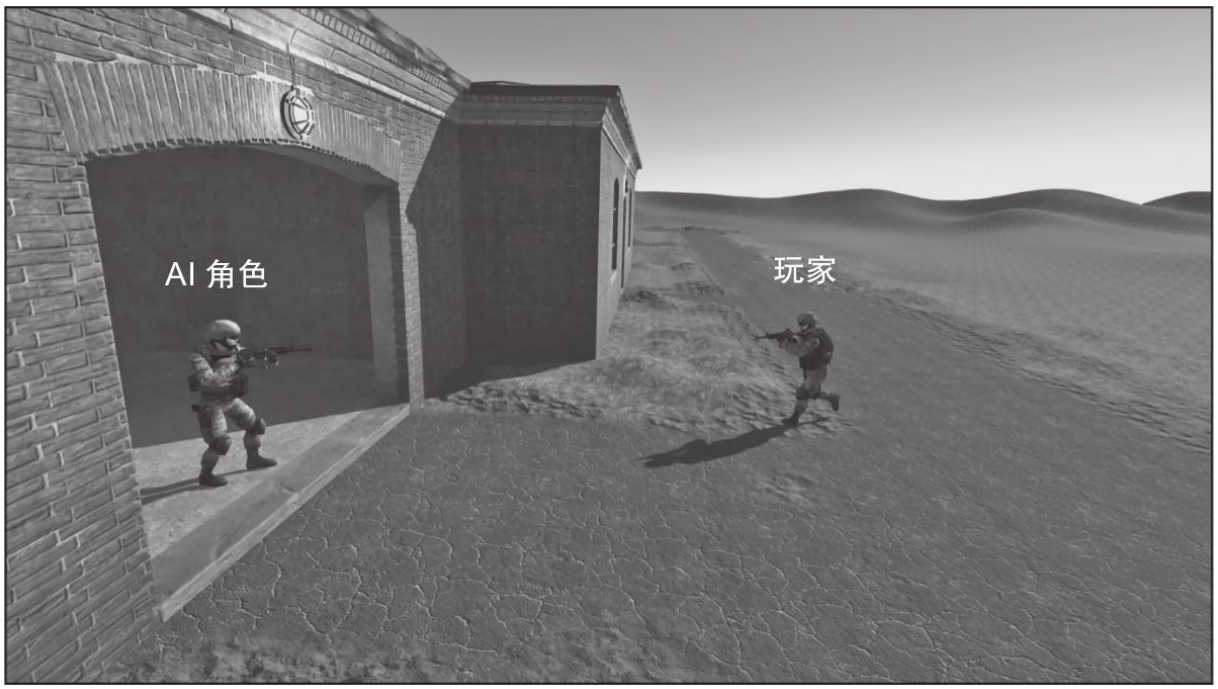
```
Void Defensive () {
if(playerPosition == "triggerR")
{
// Check if player is currently
located on the triggerR position
transform.LookAt(playerSoldier);
// Face the direction of the player
if(cover == false)
 {
  transform.position = Vector3.MoveTowards(transform.position,
  wallPosition.position, walk);
 }
if(cover == true)
 {
  coverFire();}
 }
}
```

添加了防守状态的核心逻辑，然后还要实现当玩家从右边接近时，敌人AI的具体逻辑。添加了新的变量，分别是speed、cover、playerSoldier和coverFire。首先需要检查玩家是否位于右边triggerR区域，如果检查结果为真，角色应当移动到掩护位置。一旦敌人AI到达了掩护位置，它就可以开火射击玩家了（coverFire函数）。现在输入接下来的情况——如果玩家依然活着，敌人需要移动到另一个位置去，否则就会被困在角落里，对敌人来说，那可不是什么好的状况。添加这些逻辑到脚本中。

我们希望角色会一边面朝玩家射击，一边退回到建筑物里面。可以使用另一种战术，或是更具进攻性地直接与玩家交火，但是现在还是坚持采用更简单的策略。后面再添加更

```
if (playerPosition == "triggerM")
{
 transform.LookAt(playerSoldier); // Face the direction of the player
 transform.position = Vector3.MoveTowards(transform.position,
 buildingPosition.position, walkBack);
 backwardsFire();  }
```

多复杂的行为：



这部分代码中添加了玩家从右侧接近然后到中间时仍然活着的情况，所以还需要修改之前的代码。AI角色从掩护位置移动到新的位置也就是建筑物内部，全程都保持对玩家射击。这里，敌人会持续后退直到其中一方的角色阵亡——玩家或是AI角色。这样玩家从右边接近的情况就处理完毕了。现在这部分完成了，接着添加最后一种情况以完成整个情景，也就是玩家绕了建筑物一圈然后从左边接近的情况。AI需要适配这种情况并做出不同的应对，所以让我们来做完这部分，然后结束这个例子。

2.3.2 进攻状态

在写程序之前，我们定义过这个敌人AI一共需要哪些状态，然后选择了其中三种：被动、防守和进攻。现在两种行为状态已经准备就绪了（被动和防守），还需要最后一种来完成敌人AI，让它能守住建筑物。

我们之前确定了，当这个角色无法利用墙体作为掩体时，直接与玩家交火。这种情况

发生在玩家从左侧接近，他的突然出现惊动了敌人的时候。

再一次，需要先检查玩家是否位于左边的区域，如果在的话，会激活敌人AI从被动状态切换到预定的进攻状态。然后需要定义这种情况如何处理。把这个逻辑写进脚本：

```
Void Aggressive () {
if(playerPosition == "triggerL" || playerPosition == "triggerM")
{
 transform.LookAt(playerSoldier); // Face the direction of the player
 frontFire();
}
else {
 transform.position = Vector3.MoveTowards(transform.position,
 triggerLPosition.position, walk);
}
}
```

这次添加了玩家从左边进攻时两种可能的状况：第一种是玩家从左边过来然后继续朝敌人移动或者站在那里；第二种情况是玩家看到敌人后立即撤退，在这种情况下选择让敌人搜寻玩家，移动到左边triggerL区域，也就是玩家出现的位置。

下面是完成版的脚本，使用了本章一直在讨论的可能性图。看一下完整的脚本：

```
Private int Health = 100;
Private bool statePassive;
Private bool stateAggressive;
Private bool stateDefensive;
Private bool triggerL;
Private bool triggerR;
Private bool triggerM;
public Transform wallPosition;
public Transform buildingPosition;
public Transform triggerLPosition;
private bool cover;
private float speed;
private float speedBack;
private float walk;
private float walkBack;
public Transform playerSoldier;
staticstring playerPosition;
```

上面这段代码里，我们可以看到目前所有用到的变量。下面是代码的其余部分：

```
// Use this for initialization
Void Start () {
statePassive = true;
}
// Update is called once per frame
Void Update () {
// The AI will remain passive until an interaction with the player occurs
if(Health == 100 && triggerL == false && triggerR == false && triggerM ==
false)
{
 statePassive = true;
 stateAggressive = false;
 stateDefensive = false;
}
// The AI will shift to the defensive mode if player comes from the right
side or if the AI is below 20 HP
if(Health<= 100 && triggerR == true || Health<= 20){
 statePassive = false;
 stateAggressive = false;
 stateDefensive = true;
}
// The AI will shift to the aggressive mode if player comes from the left
side or it's on the middle and AI is above 20HP
if(Health> 20 && triggerL == true || Health> 20 && triggerM == true){
 statePassive = false;
 stateAggressive = true;
 stateDefensive = false;
}
walk = speed * Time.deltaTime;
     = speedBack * Time.deltaTime;
}
Void Defensive () {
     if (playerPosition == "triggerR")
     {
     // Check if player is currently located on the triggerR position
     transform.LookAt(playerSoldier); // Face the direction of the
     player
     if(cover == false)
      {
        transform.position = Vector3.MoveTowards(transform.position,
        wallPosition.position, walk);
       }
     if(cover == true)
     {
      coverFire();
     }
     }
    if(playerPosition == "triggerM")
     {
      transform.LookAt(playerSoldier); // Face the direction of the
      player
      transform.position = Vector3.MoveTowards(transform.position,
      buildingPosition.position, walkBack);
      backwardsFire();
     }
    }
```

```
Void Aggressive () {
if (playerPosition == "triggerL" || playerPosition == "triggerM")
 {
  transform.LookAt(playerSoldier); // Face the direction of the player
  frontFire();
 }
else {
  transform.position = Vector3.MoveTowards(transform.position,
  triggerLPosition.position, walk);
 }
}
Void coverFire () {
// Here we can write the necessary code that makes the enemy firing while
in cover position.}
Void backwardsFire () {
// Here we can write the necessary code that makes the enemy firing while
going back.}
voidfrontFire() {
}
```

2.3.3 可能性图小结

终于完成了第一个可能性图的例子。本章中展示的原理可以广泛地用在其他的游戏类型中。实际上，几乎所有你将来打算创作的游戏都会从可能性图中受益。如我们所见，这种技术用来规划玩家所能制造的各种局面以及AI角色应当怎样应对。通过对此进行仔细的设计，可以避免游戏过程中的很多问题，同时也可能在AI角色行为方面缺少了一些多样性。另外值得一提的是，可以试着为游戏中不同的角色创建不同的可能性图，就像不同的人，反应也不完全相同。计算机AI也应当遵循同样的规则。

2.4 概率图

概率图是一种更复杂、更细化版本的可能性图，因为它基于概率来改变角色的行为，而不是基于简单的开/关触发器。它和可能性图相似的地方是它也被用来提前规划角色可能的行为。而这次，添加了一个百分比，依靠它AI会计算哪一种行为会被采用。想象一下接下来的例子，还是继续使用之前的场景——敌人AI在白天会比晚上更具有进攻性。所以创建一个语句用来告诉敌人现在是否是晚上，晚上发现玩家的概率更低，基于同样的理由也应当采取更保守的行动而不是进攻性的。或者，为简化起见，可以定义敌人消灭玩家的概率与两人之间的距离相关。如果玩家更靠近敌人，AI撤退与存活的概率都会下降，可以把这个规则添加进AI中。

看一下人类的行为方式，我们做的选择——通常，会根据以往的经历和当时的行动来做出当下的决定。如果我们感觉到饿，决定出去吃饭，我们的朋友能猜到我们会去哪家餐馆吗？朋友可能会计算各种选项的概率，然后选择概率最高的一个。这正是我们将要对AI朋友做的事情：我们需要为它的行动设置一些概率，比如当它守卫建筑物时睡着了，这种情况发生在白天和晚上的概率各是多少？当它血量低时，逃跑的概率是多少？为AI角色设定行为的概率，这对创建和人类一样不可预测的行为很有帮助，也会让游戏更自然更引人入胜。

2.4.1 怎样使用概率图

这个例子里，继续使用之前创建的场景，AI角色守卫着一个建筑，玩家要去让建筑物里面的原子弹失效。建筑物唯一的入口被AI角色把守。

想象一下我们就是门口的卫兵，被命令一直待在那里16小时——我们可能需要吃饭、喝水、走动一下，以便于持续保持警觉状态。把这些行为添加到角色里，让它的行为对玩家来说显得更难预测。如果AI决定吃饭或喝水，它应当在建筑物里面，如果它决定出去走

时间	守卫	吃饭/喝水	走动
早上	0.87	0.1	0.03
下午	0.48	0.32	0.2
晚上	0.35	0.40	0.25

走，它应当从左边区域到右边区域巡逻。大部分时间里，它应当站在守卫的位置上。

上面是一张概率图，定义了角色切换到每一种状态的概率。它表明了每一次玩家看到AI敌人，敌人会在做这些事情之一。玩家出现在一天不同的时间段时，表现会有明显的不同。如果玩家在早上出现，会有0.87的概率发现敌人在建筑物前守卫的位置上，有0.10的概率发现它在建筑物里吃东西或喝水，最后有0.03的概率发现它在外面从一个点走到另一个点。如果玩家在下午到达，有0.48的概率发现敌人在建筑物前守卫的位置，有0.32的概率发现它在建筑物里吃东西或喝水，最后有0.2的概率发现它在外面从一个点走到另一个点。晚上，有0.35的概率发现敌人在守卫的位置上，有0.40的概率发现它在建筑物里吃东西或喝水，最后有0.25的概率发现它在走动。

这对创建角色的不可预测性很有帮助，让角色的行为不那么明显，好比每次玩这关的时候它都站在同样的位置上。我们还可以每5分钟左右更新一次概率，以针对玩家一直站在那里等待敌人变换位置的情况。这种技术被应用于很多游戏，但最多的还是用在潜入类的游戏里，这种游戏里观察是关键。这是因为玩家有机会待在一个安全位置观察敌人的行动，类似于抢劫题材的电影里，匪徒等待守卫换岗的时机潜入银行。由于这种常见的行为在电影里的流行，玩家也喜欢在游戏中获得相同的体验，因此概率图改变了我们玩游戏的方式。

概率如何在脚本中应用的例子如下所示。这里使用了被动状态并添加了我们之前打算要用的概率：

```
Void Passive () {
rndNumber = Random.Range(0,100);
If(morningTime == true &&  13)
{
// We have 87% of chance
goGuard();
}
if(morningTime == true && rndNumber =< 13 && rndNumber< 3)
{
// We have 10% of chance
goDrink();
}
if(morningTime == true && rndNumber<= 3)
{
// We have 3% of chance
goWalk();
}
if(afternoonTime == true && rndNumber> 52)
{
// We have 48% of chance
goGuard();
}
if(afternoonTime == true && rndNumber =< 34 && rndNumber< 2)
{
// We have 32% of chance
goDrink();
}
if(afternoonTime == true && rndNumber<= 2)
{
// We have 2% of chance
goWalk();
}
if(nightTime == true && rndNumber> 65)
{
// We have 35% of chance
goGuard();
}
if(nightTime == true && rndNumber =< 65 && rndNumber< 25)
{
// We have 40% of chance
goDrink();
}
if(nightTime == true && rndNumber<= 25)
{
```

```
// We have 25% of chance
goWalk();
}
}
```

为计算百分比，我们需要先获得一个0到100的随机数，然后我们写一个语句用随机数去检查应当切换到哪种状态。比如第一种情况，我们有87%的概率让AI待在守卫点，如果随机数大于13，就满足了这个条件，角色被设置为待在守卫区域。大于3小于等于13的数，代表了10%的概率，小于3的数代表了3%的概率。

2.4.2 接下来做什么

现在我们理解了怎样使用可能性图与概率图，我们可能会问自己一个相关的问题，那就是这些能用来做什么？我们已经看到了使用可能性图来定义角色行为的重要性，也看到了概率图怎样改善了这些行为的不可预测性，其实我们还可以用学到的东西做很多事情，这取决于我们要做的游戏类型以及我们想要怎样的AI。记住缺点是我们生而为人的一部分，我们被概率包围，即使只有0.000001%的概率的行为也是有可能发生的，因此人无完人。有趣的一点是，在创建AI角色时，给它一些几率去做人类才会做的事情，或是简单做一些好的或糟糕的决定，这会给你创造的计算机角色建立一种个性。

概率图可以用来做的另一个特别的事情，是给AI自己学习的机会，让玩家每次玩游戏之后，AI都会变得更聪明。玩家和AI敌人都会学习，使得挑战随着游戏的进行时间而升级。如果玩家习惯于使用某种武器或从相同的方向过来，计算机都可以更新这些信息并在将来使用。如果玩家面对计算机敌人100次并且60%的情况下他都使用了手榴弹，AI应当记住这条信息然后根据这个概率制定对策。这会促使玩家思考其他的战术、探索其他打败敌人的方式，而不要表现得这么明显。

2.5 总结

本章描述了可能性图与概率图，同时我们学习了怎样让AI根据玩家的行动做出自己的决定。可能性图与概率图是AI角色的基础，我们会继续探索用这种技术去创建新的独特的人工智能，用在自己的游戏里。下一章，我们将学习不使用概率图的情况下，AI在有多种选项的时候应该如何表现。我们希望AI角色分析状况、思考如何行动，并且考虑多种因素，例如健康度、距离、武器、子弹数量和其他任何相关参数。

第3章　产生式系统

在这一章，我们将讨论完善AI角色的不同方法，以及如何让同一种技术适应于不同类型的游戏。与此同时我们也会讨论以下话题：

· 自动有限状态机（AFSM）

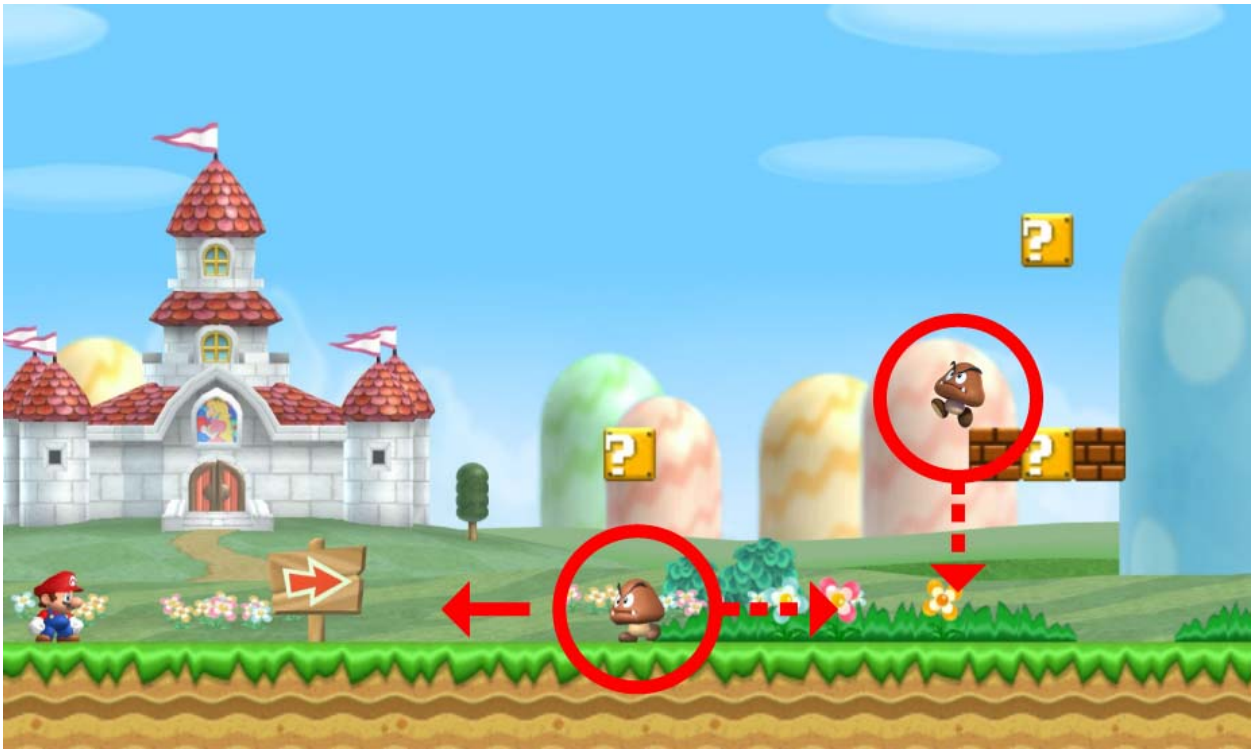
· 可能性计算

· 基于效用的方法

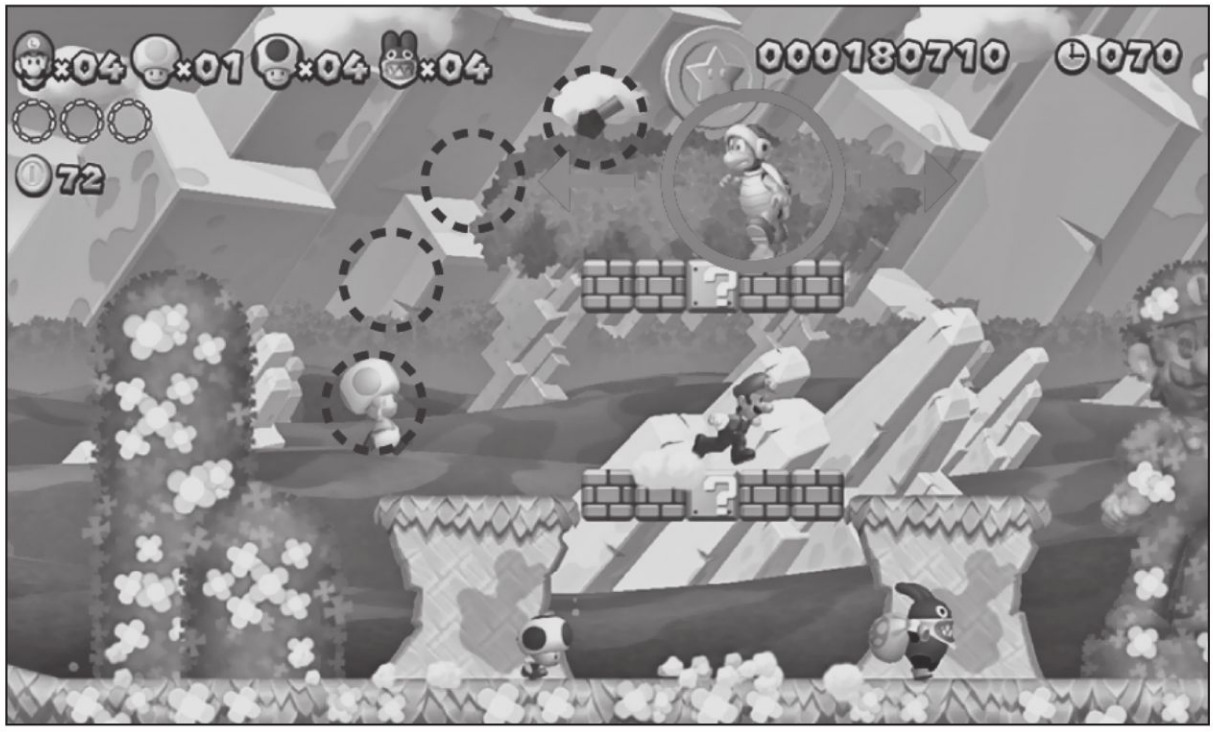
· 游戏AI的动态平衡

在探索完可能性图和概率图后，我们需要明白如何结合其他技术和策略来使用它们，帮助我们创建更加拟人的AI角色。可能性图或者概率图独立使用也都可以创建出有趣的游戏；事实上，许多电子游戏都只依赖于这两种图来创建它们的AI角色，这样做也依然很成功。一个很好的例子是常见平台类游戏，如任天堂（Nintendo）的《超级马里奥兄弟》（Super Mario Bros）。他们并不需要为了使敌人更加有挑战性而创建复杂的AI系统，这也是为什么他们10年来在AI敌人的开发上都使用着相同的方式，对于这种类型的游戏来说，这些方式就已经能够完美实现需求了。需要谨记的重要一点是：我们决定使用某种技术或者某种技术是否比其他技术好，这完全取决于我们将要创建的是怎样的游戏。在创建AI角色时也是一样的，我们应该知道AI在游戏的每一秒中，该做什么，什么时候做。

让我们继续以《超级马里奥兄弟》为例子，并分析通常这些敌人是怎么做的：



在这个游戏截图中的敌人叫板栗仔（Goomba）。它一出现在游戏中，你就会注意到它从右边移动向左边，一旦它撞到某物（除了玩家以外）就会改变移动方向从左边移动到右边。如果它处在高处的平台上，它也会从右向左的移动，直到它坠落到低处的平台后，又会保持从右向左的移动。此敌人从不会尝试主动攻击玩家，而且其行动是非常可预测的。因此我们可以确定，它只有一个目标，就是移动，我们可以将它放置在场景的任何一处，它都会准确执行相同的行为。现在让我们转到下一个敌人：



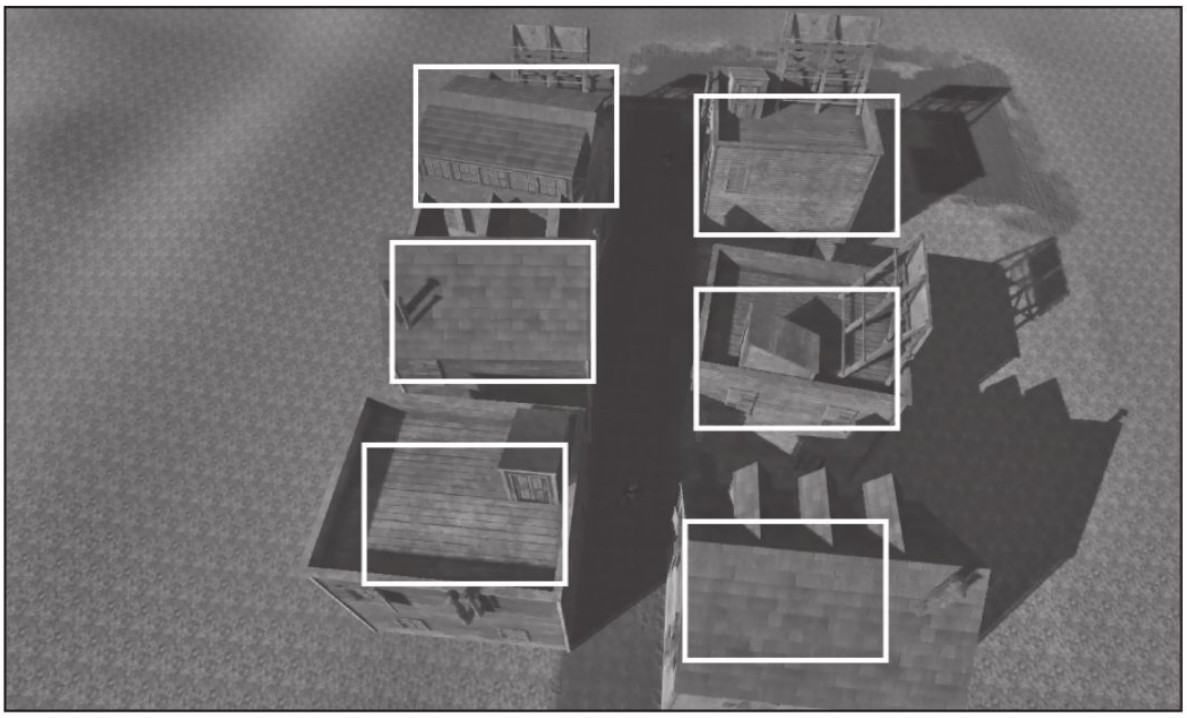
在第二个例子中，敌人叫做锤哥（Hammer Bro），它与板栗仔不同的地方是：它可以向左移动也可以向右，且总是面朝玩家移动，并且它会朝玩家的方向丢出锤子。因此，它在游戏中的主要目的就是击败玩家。与板栗仔相同的地方是：这个AI角色也可以被放置在游戏的任何地方，并会按照它的目标行事。现在想象一下，我们在上一章中开发的AI如果被放置在这里的场景中，它会有什么反应？因为我们并没有给它写任何关于它在这个场景中应该要怎么做的代码，所以其实它不会作出任何反应。这也说明了，要使我们开发的AI角色按照我们的意图做出反应，是依赖于我们正在创建的具体游戏本身的。有时它会被固定在一个位置上，但大多数情况下，同样的AI需要在游戏中不同的位置做出同样的反应。想象一下，如果《超级马里奥兄弟》的制作者每次在给游戏加入敌人时，都需要重新定义他们的AI，这将花费大量的时间，产生大量多余的工作量。因此让我们来学习如何使用有限状态机来确保我们的角色对游戏中的每种状况都能良好适应。

3.1 自动有限状态机

我们注意到在《超级马里奥兄弟》的例子中，不论AI敌人处在什么位置，它们都知道如何正确地反应。显然，它们不需要执行复杂的任务或预先计划它们将要做什么，但这依然是一个完美的范例，尤其是当与其他不同类型的电子游戏相比时。举个例子，我们可以看到一些相同原则被用于《光环》（Halo）里的咕噜人（Grunt，一种小怪物）。它们简单地从一边移动到另一边，如果它们发现了玩家，就会朝玩家开火；而当咕噜人与玩家的战斗失败时，它们则会逃走。为此，开发者会使用一个状态来进行判断：每当角色血量低于一定程度时，它们就会开始逃跑。我们用FSM来构建概率图和可能性图，这也是角色在面临不同情况时应该做的事情。现在让我们创建自动有限状态机，在这里AI角色将通过计算一些因素来选择最佳决策（如位置、玩家血量、当前武器等等）。如果我们计划在不同的舞台或者包含开放世界要素的游戏中使用相同的角色，那么这个方法是非常有效的。

在规划AFSM阶段，将AI角色要执行的动作拆分为两到三列信息是一个很好的开始。在第一列，我们放上目标的主要信息，如方位、速度、目的地；在其他列中，我们放上用来达成目标所需执行的动作，如移动、射击、装弹、寻找掩体、隐匿以及使用物品等等。这么做我们能确保我们的角色可以根据第一列的内容做出反应，并且整个过程独立于它当前所在的位置。试想一下AI的目标是守卫我们指定的位置，这个目标是首要的，因此会被放置在第一列。现在想象AI角色开始游戏后，距离它需要守卫的位置很远。此时，它会使用第二列的动作去完成第一列的目标。我们在第二列放入什么动作，完全取决于我们想要开发的游戏类型。接下来，让我们来创建一个示例来展示如何较好地完成这些事。

在这个例子中，我们将继续使用FPS类型作为演示的主要舞台，但相同的原理可以被应用在绝大多数电子游戏中。这也是为什么我们之前将《超级马里奥兄弟》作为参考，用以证明无论我们想要创建哪种类型的游戏，当中的AI开发都遵循着相同的开发过程：



例子中所使用的地图有着6个建筑，玩家和敌人可以去到除建筑内部以外的任意位置。游戏的主要目的，是在有限的时间内尽量多地击败对手；弹药点和补血点会间歇性地在游戏内不断生成。现在让我们来开发一个可以在任何地图上以相同的方式进行反应的AI。为了达成这点，我们需要给AI角色分配一个主要的目的，以及让它知道实现目的的所有可能性，同时还要确保它在游戏进行时每秒都有事情做。

简单来说，AI角色有两个主要目的：击败玩家（DEFEAT PLAYER）和生存下来（SURVIVE）。我们需要确保AI在有机会时可以干掉玩家，没有机会时也要生存下来。现在我们简化一下规则，通过AI角色当前血量的多寡来定义将要执行的目标。如果AI当前血量大于总血量的20%，那么主要目的便是击败玩家，如果当前血量小于总血量的20%，那么主要目的就变为存活下来：

>20HP DEFEAT PLAYER		
<20HP SURVIVE		

一旦定义了上面的内容，我们就可以转战第二列填写次要目的了，AI将根据第一列的首要目标来选择对应的次要目的。在这个例子中，我们将给予AI角色三个次要目的：寻找玩家（FIND PLAYER），寻找掩体（FIND COVER），寻找道具点（FIND POINTS）。通过使用它们，AI将有能力实现主要目的，且总会有事情需要做，不会去等待玩家的行为。如下

>20HP DEFEAT PLAYER	FIND PLAYER	
	FIND COVER	
<20HP SURVIVE	FIND POINTS	

图：

现在我们已经定义了次要目标，接下来将写下AI在游戏中所有可能的动作，例如，移动至（MOVE TO）、射击（FIRE）、使用物品（USE OBJECT）和蹲下（CROUCH）。玩家或敌人AI在游戏中所能做的所有事情都是我们在进行游戏设计时预先定义的，这些内容都应该填入这一列中。这也是一个可以用来分析游戏角色的所有动作是否与主要或次要目的相关的重要策略。如果一个复杂的动作并不能有助于达成主要目的，那么便没有必要编写这个动作，这将节省我们的时间。如同在超级马里奥兄弟中，开发者并没有给敌人AI开发复杂的动作，因为游戏类型决定了没必要做这些。在上面的示例中，角色可以自由移动，射击，使用物品（装弹药或者使用补血点）以及蹲下：

>20HP DEFEAT PLAYER	FIND PLAYER	MOVE TO
	FIND COVER	FIRE
<20HP SURVIVE		USE OBJECT
	FIND POINTS	CROUCH

现在我们已经填完了全部三列内容，包含AI在当前情况下进行决策时所需要的所有信息。我们马上就会看到，这个方法与前面章节中采用的方法完全不同，因为当时我们使用图来指示AI角色应该去做什么事，只根据位置来下达指令。而在这个例子中，我们想要AI角色无论被放在哪个位置、哪张地图，都可以做出最佳决策。这种方法会让我们的AI开发水平迈上一个全新的台阶，因为人类的真实行为其实很少会像之前那样，根据一个统一、固定的标准做出决策。

这个相同的处理也适用于我们当前的敌人：它们会根据不同的标准做出最佳决策，我们需要确保它们选择的最佳决策是基于它们自己的决定。例如，只有1%生命时的紧迫度，比弹药低于1%时更高，甚至高于完成游戏的主要目的。

准备好了这三列内容，我们就可以继续下一步，将第三列的每个动作连接至第二列的每个行为，然后将第二列的每个行为连接到第一个列的主要目的。做这件事时需要考虑在AI想要寻找玩家、掩体或是道具点时，需要做哪些具体的事情。同时，还需要定义它应该在什么时候寻找玩家、掩体或是道具点。为了找到玩家，需要使用MOVE TO动作，此时角色就会到处移动，直到它找到玩家，最后朝玩家开火。如果是寻找掩体，我们也将使用MOVE TO动作，使得角色在抵达可以作为掩护的墙附近之前一直移动。之后我们也可以选择是否让它蹲下，这取决于它想要达到的具体目标。至于寻找道具点也是类似，先MOVE TO，然后决定自己是否要使用物品（USE OBJECT）。现在让我们考虑，当AI试图击败玩家或者试图生存下来时，会选择哪些行为。为了击败玩家，AI角色需要寻找玩家，所以我们使用FIND PLAYER这个行为；如果AI已经找到了玩家，并且自身处于墙附近，我们将会让它选择FIND COVER行为。而为了实现存活这一目标，我们会在当其遭受玩家攻击时选择FIND COVER行为，还会通过FIND POINTS行为来找到生存道具以恢复生命值。

可能性计算

现在我们已经配置好了所有东西，接下来需要将所有这些信息输入到代码中。在这里我们会使用布尔值来定义主要目的，还会写一些逻辑让AI角色在所有不同的选项中间抉择。我们已经定义了主要的语句来切换AI角色的目的（从击败玩家切换至生存下来），但是现在我们需要添加更多的细节到我们的AI行为中，原因很简单：如果敌人AI在面对玩家时，有足够的生命值（HP），但是没有足够的弹药怎么办？如果有足够的弹药，但是在上一次

攻击尝试中失败了又怎么办？角色需要对每一个选项进行比较，并挑出最有可能实现主要目标的选项，给予其最高的优先级。

让我们从击中玩家的可能性开始：假设AI已经射出10发子弹，且只有4发击中了玩家。可以认为（以样本的概率分布为基础）它在下一次射击中有40%的可能性击中玩家。现在假设它的枪里只有两发子弹。此时它会怎么做？是在命中率不高且弹药不多的情况下朝敌对玩家开火，之后陷入弹尽粮绝的境地？还是跑向道具点补充弹药？为了帮助它做出这个决定，AI角色需要计算被玩家射到的可能性：如果玩家射到AI角色的可能性较小，AI将会冒险去尝试攻击玩家，否则它会尝试先补充弹药再做其他考虑。现在尝试将这些东西添加

```
Private int currentHealth = 100;
Private int currentBullets = 0;
private int firedBullets = 0;
private int hitBullets = 0;
private int pFireBullets = 0;
private int pHitBullets = 0;
private int chanceFire = 0;
private int chanceHit = 0;
public GameObject Bullet;
private bool findPlayer;
private bool findCover;
private bool findPoints;
```

进代码里，你可以在下面这个例子中看到，这些代码是如何工作的。

这些是我们将会用到的变量。firedBullets表示角色在整个游戏过程中已经射出了多少子弹；hitBullets表示有多少子弹击中了目标；pFiredBullets和pHitBullets与上面两个变量意思相同，但指代的是玩家的弹药情况。接下来计算击中目标或被目标击中的可能性。chanceFire表示击中目标的概率百分比，chanceHit表示被击中的概率百分比：

```
void Update ()
{
 chanceFire = ((hitBullets / firedBullets) * 100) = 0;
 chanceHit = ((pHitBullets / pFiredBullets) * 100) = 0;
 if(currentHealth > 20 && currentBullets > 5)
 {
  Fire();
 }
 if(currentHealth > 20 && currentBullets < 5 && chanceFire < 80)
 {
  MoveToPoint();
 }



 if(currentHealth > 20 && currentBullets < 5 &&chanceFire>80)
 {
  Fire();
 }
 if(currentHealth > 20 && currentBullets > 5 && chanceFire < 30&&
 chanceHit > 30)
 {
  MoveToCover();
 }
 if(currentHealth < 20 && currentBullets > 0 && chanceFire > 90 &&
 chanceHit < 50)
 {
  Fire();
 }
}
```

我们使用击中与被击中的可能性来决定AI在特定情形下应该怎么做。如果它的生命值大于20且枪中弹药大于5发，那么它会与玩家开火交锋，直到上述两个条件中的任意一个不满足了为止。一旦只有5发弹药时，那么是时候考虑下一步的移动了，在这个例子中，如果它成功击中玩家的概率小于80%，它会尝试不攻击，并朝弹药点移动。如果它成功击中玩家的概率大于80%，那么它会尝试朝玩家开火。而在战斗中如果AI的命中率不到30%，而玩家大于30%，那么AI会立即寻找掩体。最后，如果AI角色当前生命值低于20%，但它有90%的几率击中玩家，且被玩家击中到的概率小于50%，那么它会选择开火。

如果想要更精确的百分比，还可以额外引入一个时间变量，AI会考虑最近的两分钟而不是所有时间，亦或是将两种百分比综合起来比较并分析哪一个的值更高：

```
if(recentPercentage > wholePercentage)
```

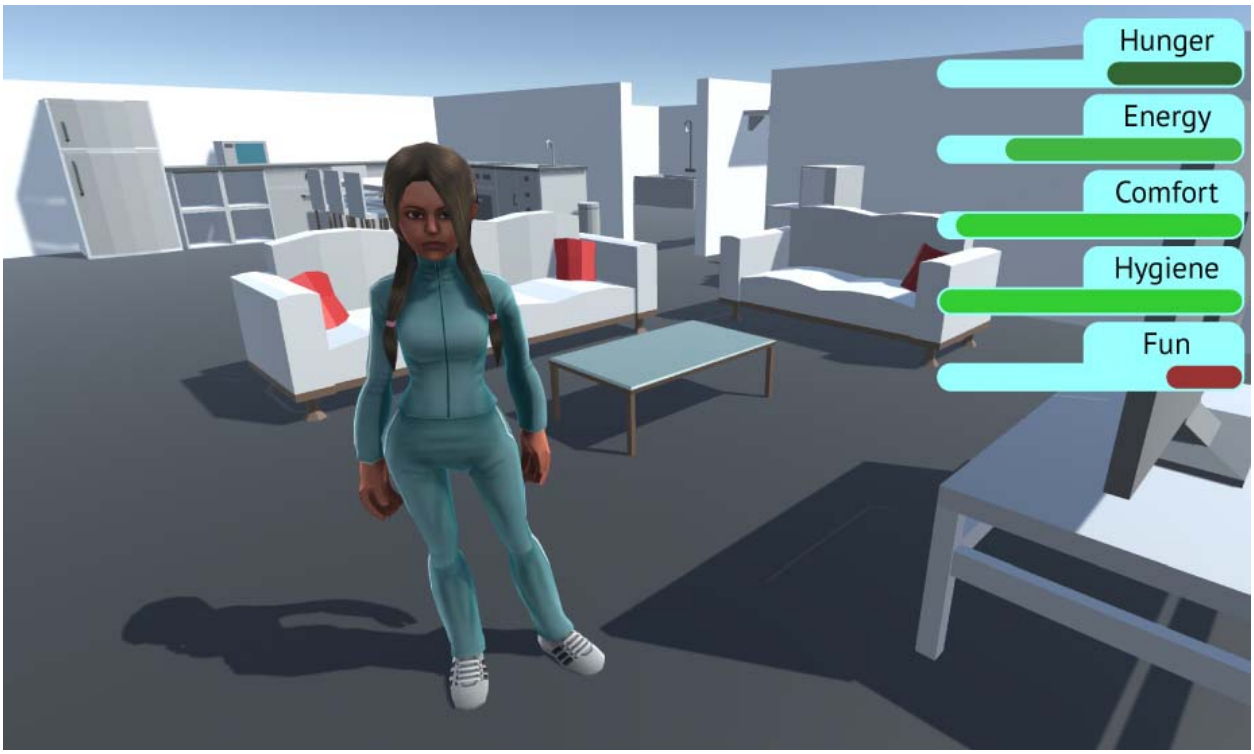
通过可能性计算，我们给了AI足够多的方法去得出它自己的下一步行动，它可以自由地选择在特定时刻下更重要的目的，并选择对应的为达成此目的所需的动作。这样同时还会帮助AI在两种或更多的可行方案中选出对自己最有利的选项。这样做出来的将是更为智能的角色，它们可以为自己思考。同时我们还可以定义角色的个性，通过简单地变更百分比值来赋予它们更冒进或者更保守的习性。

3.2 基于效用的函数

知道了如何计算可能性以及如何使用AFSM，现在是时候来更深入地探索它们，以便让我们的角色看起来更加聪明。这一次，我们使用一个类似《模拟人生》（The Sims）游戏中具有自主行为的AI角色为例。这是一个测试AI性能的完美环境，因为其模拟了真实生活中的需求和选择。

在《模拟人生》中，玩家控制的是与真实人类高度近似的角色，主要游戏目标是让这些角色总是处在好的情况中，使得他们的个人与职业生活总是积极的状态。与此同时，跟真实世界一样，随着时间慢慢流逝角色也会慢慢老去，直至死亡。玩家对这些人的生活负责，但如果玩家没有给这些角色任何行动指令，他们也会自动做出满足自己需要的行动。在电子游戏中，这些AI角色的表现方式是革命性的；人们联想到这些虚拟角色，会觉得他们就像真的过着独立自主的生活一样。而创建这样一个与《模拟人生》中完全一样的角色需要的东西，我们之前已经有所接触。

不啰嗦了，转到我们的例子：



首先让我们给例子中的角色起一个名字：索菲。她是一个虚拟的人类。她的家里有日常生活所需的所有物品：沙发、浴室、电视、床和烤箱等等。同时与真实世界类似，她也有着典型的人类需求，例如饥饿（HUNGER）、活力（ENERGY）、舒适（COMFORT）、卫生（HYGIENE）和愉悦（FUN）。随着时间的流逝，她需要满足这些需求来维持健康。现在的问题是要能够让索菲完全自主地决定自己需要去做什么。为此让我们来研究一个解决方案：

HUNGER ENERGY COMFORT HYGIENE FUN	GO TO INTERACT WITH

我们可以简单地分解目的到以上这两列：左边一列放入游戏的主要目的，右边一列则放入角色可以执行的动作。例如，如果她饿了，她需要移动到厨房，与冰箱交互。这与我们在FPS示例中创建的自动有限状态机的原理是完全相同的，不过这次我们将更深入地探讨这个概念。

首先考虑饥饿感。吃了早餐后，由于满足了饥饿的需要，我们便不再感到饥饿了。在这个点上我们可以说，我们饥饿感的需求值是100%。经过一小段时间之后，我们饥饿感的需求值变成了98%；为什么我们不立刻补充这减少的2%呢？因为只有当需求值减少至介于空与满之间的某个特定值时，才会判定为饥饿的状态，在这之前其他需求会更加优先一些。所以在开发自主AI角色时：需要让AI也意识到上述原则，使得它不会一旦减少1%的饥饿度就立刻去吃东西，那样看起来实在不像是人类的行为。游戏内能做的所有事情都需要平衡，否则会变成睡5分钟，然后再工作5分钟，这样既不能说健康也不能说有效率。因此，我们往往设定为，睡足够长的时间来补偿清醒时的时间，吃合理数量的食物来维持一段时间的饱腹状态。我们需要判断出“有一点饥饿，但需要首先完成工作”的情形。除此之外还需要对一系列的决策进行比较，如：虽然饿，但是更累。让索菲这样的虚拟角色懂得真实人类会做出怎样的行为决策是非常重要的，否则，会让索菲看起来像个苍白的机器人。

为了帮助索菲决定在某个特定时刻什么才是最重要的，我们需要使用到比例系数，索菲会根据系数去比较和决定想要做的事。在事情变得复杂之前，让我们先在代码里写一些基本的信息：

```
Private float Hunger = 0f;
Private float Energy = 0f;
Private float Comfort = 0f;
Private float Hygiene = 0f;
Private float Fun = 0f;
private float Overall = 0f;
public Transform Fridge;
public Transform Oven;
public Transform Sofa;
public Transform Bed;
public Transform TV;
public Transform Shower;
public Transform WC;
void Start ()
{
   Hunger = 100f;
   Energy = 100f;
   Comfort = 100f;
   Hygiene = 100f;
   Fun = 100f;
}
void Update ()
{
 Overall = ((Hunger + Energy + Comfort + Hygiene + Fun)/5);
   Hunger -= Time.deltaTime / 9;
   Energy -= Time.deltaTime / 20;
   Comfort -= Time.deltaTime / 15;
   Hygiene -= Time.deltaTime / 11;
   Fun -= Time.deltaTime / 12;
}
```

上面写下了一些与角色需求相关的基本变量。随着时间的流逝，这些值会慢慢降低，不同的属性值都依赖于不同的需求。与此同时，我们会维护一个Overall变量，用来计算角色的整体情况，并完美呈现虚拟角色的心情。这个要素非常重要，它将辅助索菲决定哪些选项对她来说是最好的。

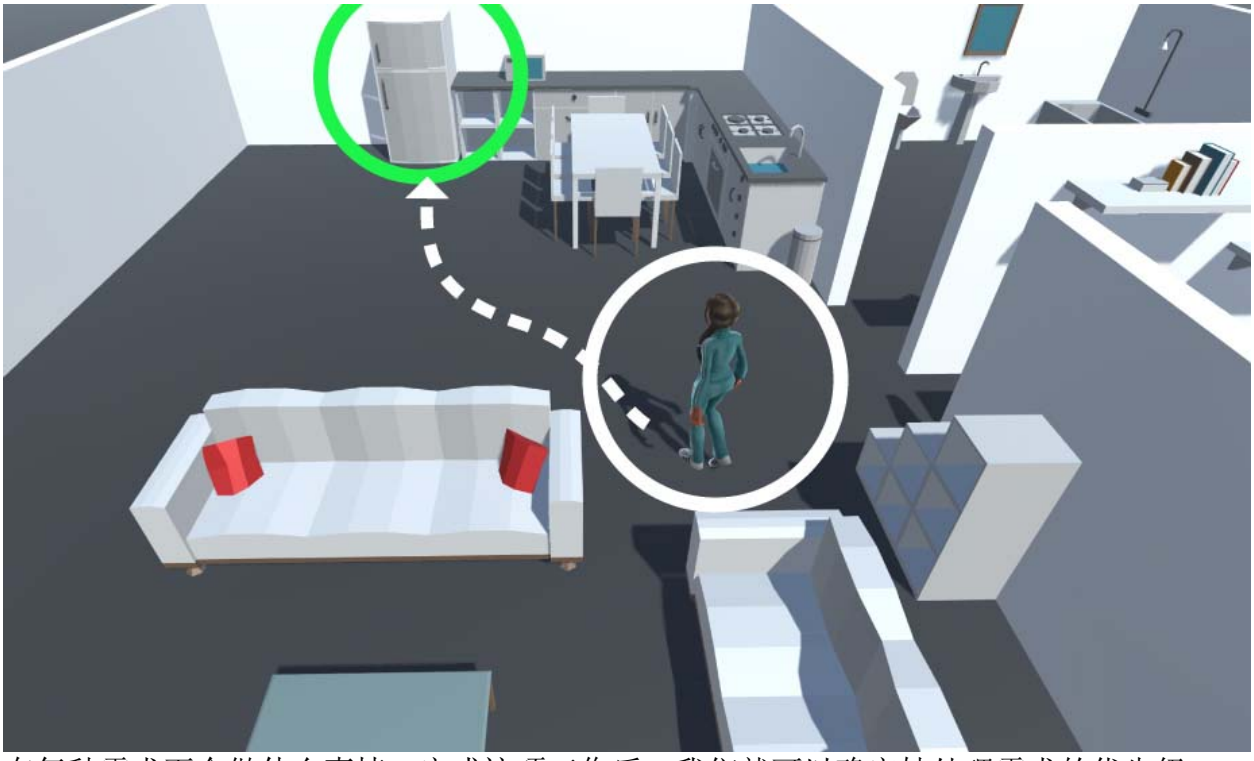
现在我们分别处理每个需求，为每一个需求都创建一个决策树。为了完成这点，需要规划一下决策过程，索菲在选择任何动作之前都会考虑一遍这样的决策过程。还是先从饥饿需求开始：

如果索菲感到饥饿并将其作为了优先事项，她将遵循下面的步骤：首先感到饥饿，接着确认自己是否有足够的食物；要得到这个答案，她需要移动到冰箱前检查冰箱。如果有，她会去做饭并用餐。在这中间，如果任何一个事件没有被完成，整个过程就会被中断，她也会转而去做其他优先事项。比方说，她冰箱里的食物快没了，如果她去上班，她就会得到两天的食物。如果她想保持健康和活力，上班很快就会成为优先考虑的事情：

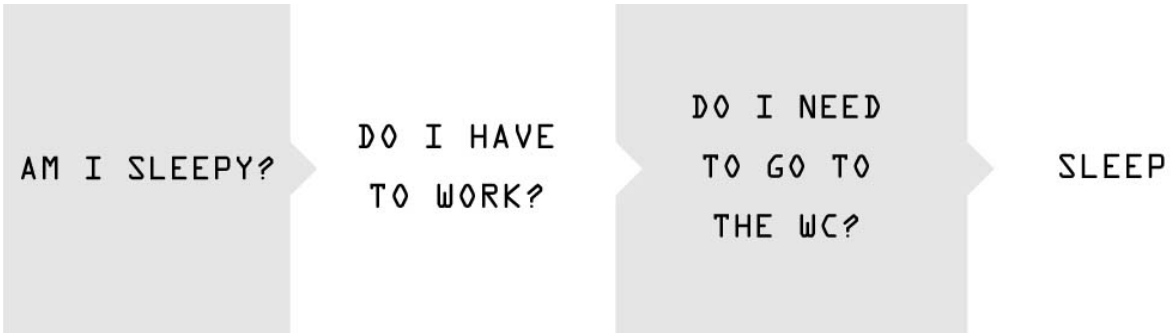
```
Private float Hunger = 0f;
Private float Energy = 0f;
Private float Comfort = 0f;
Private float Hygiene = 0f;
Private float Fun = 0f;
private float Overall = 0f;
public Transform Fridge;
public Transform Oven;
public Transform Sofa;
public Transform Bed;
public Transform TV;
public Transform Shower;
public Transform WC;
private int foodQuantity;
public float WalkSpeed;
public static bool atFridge;
void Start ()
{
   Hunger = 100f;
   Energy = 100f;
   Comfort = 100f;
   Hygiene = 100f;
   Fun = 100f;
}
void Update ()
{
   Overall = ((Hunger + Energy + Comfort + Hygiene + Fun)/5);
   Hunger -= Time.deltaTime / 9;
   Energy -= Time.deltaTime / 20;
   Comfort -= Time.deltaTime / 15;
   Hygiene -= Time.deltaTime / 11;
   Fun -= Time.deltaTime / 12;
}
```

```
void Hungry ()
{
   transform.LookAt(Fridge); // Face the direction of the Fridge
   transform.position vector3.MoveTowards(transform.position.
   Fridge.position, walkSpeed);
   //checks if already triggered the fridge position
   if(atFridge == true)
   {
    //interact with fridge
    if(foodQuantity > 1)
    {
     Cook();
    }
    else()
    {
     // calculate next priority
    }
 }
}
```

在先前的代码中展现了一个决策树如何在代码中实现的例子。之后我们也将继续写她



在每种需求下会做什么事情，完成这项工作后，我们就可以确定她处理需求的优先级。



下面列表中是与活力有关的部分：

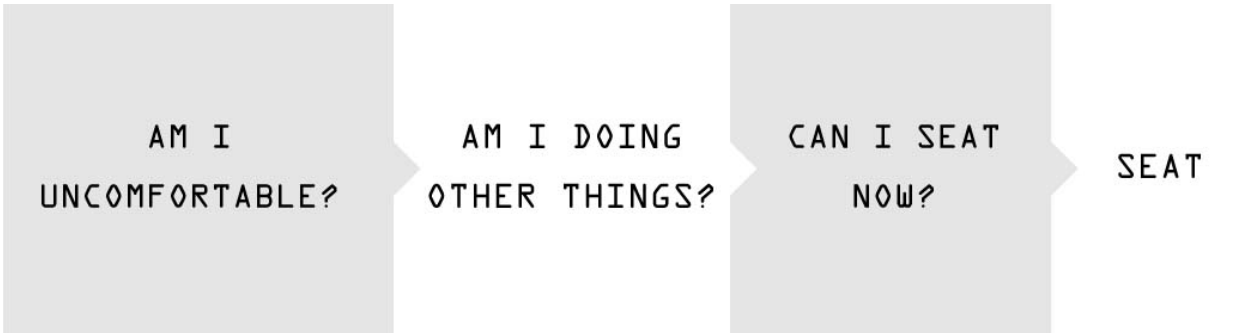
如果索菲感觉困了并将睡觉作为了优先事项，她将遵循下面的步骤：首先感到困了，然后确认自己是否还有工作要做。如果没有，那么她在睡觉前会确认自己是否需要去趟浴室。一旦每个状态都达成了，她就完成了目标并最终睡觉。下面的代码展示了本例中描述

```
void Sleepy ()
{
   if(hoursToWork > 3&&Energy < Hygiene)
{
        transform.LookAt(Bed); // Face the direction of the Bed
        transform.position = vector3.MoveTowards(transform.position.Bed.
        position, walkSpeed);
          //checks if already triggered the bed position
          if(atBed == true)
           {
           //interact with the bed
           }

 }
if(hoursToWork > 3 && Energy > Hygiene)
{
  useWC(); //Go to the bathroom
}
 if(hoursToWork < 3)
{
   //choose another thing to do
}
}
```

的内容：

这里假定，索菲每睡眠1个小时，就会获取10点活力值，但减少10点卫生值。那么首先需要确认她是否需要在睡觉前使用浴室，为此这里比较了她所需要的活力值和卫生值：

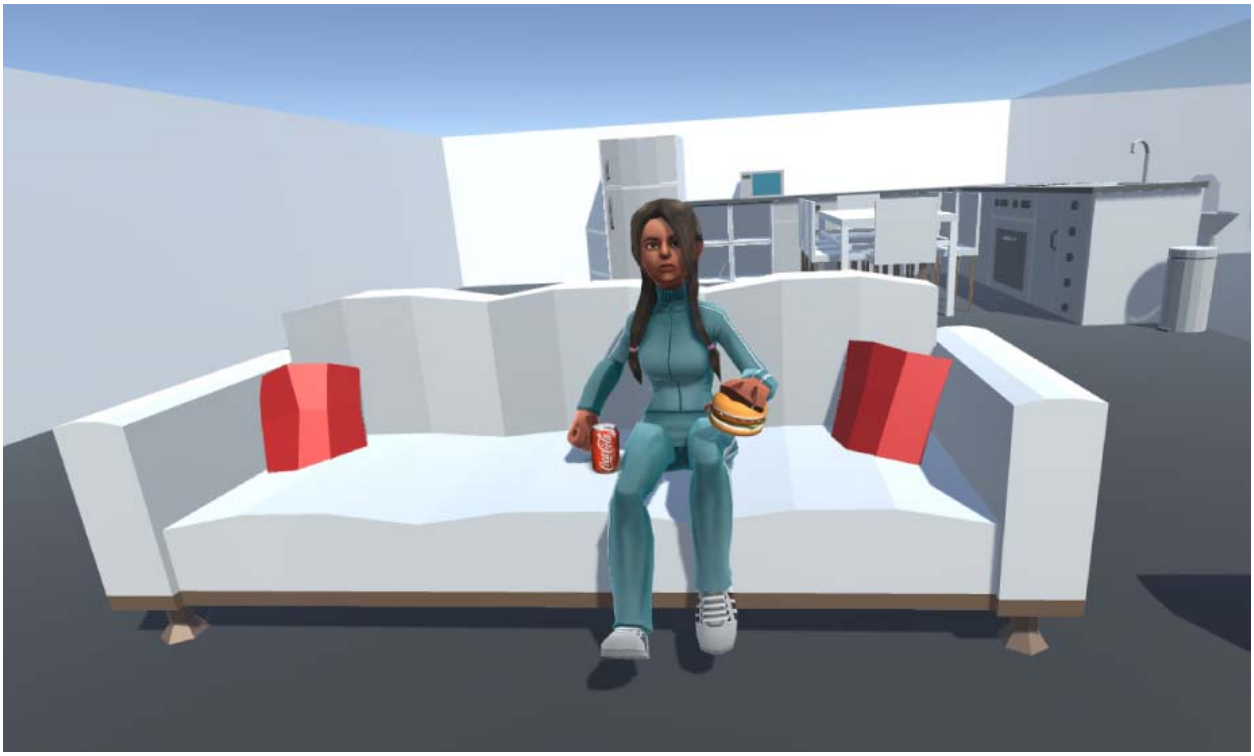
接下来考虑舒适度需求。这一块有点特殊，因为我们可以同时指派两个获取舒适度的目的。例如，她将会决定自己是否想坐在椅子上用餐。看电视的时候也同样如此。这是一个非常重要的例子，可以应用在许多游戏上：当一个角色觉得自己在同一时间内去做两件事会获得更好回报时，它就会去做。在下面的例子中我们将考虑到这一点：



如果索菲感到不舒适，她会首先检查当前自己是否正在做事。如果是，她会思考是否可以坐着去做这件事，不能坐的话她就会先完成正在做的事，然后问自己相同的问题。如果条件允许，她最终就可以坐下来并感到舒适。下面的代码展示了如何处理这个例子：

```
private bool isEating;
private bool isWatchingTV;
private bool Busy;
...
void Uncomfortable ()
{
   if(isEating == true || isWatchingTV == true)
    {
         transform.LookAt(Sofa); // Face the direction of the Sofa
         transform.position = vector3.MoveTowards(transform.position.
         Sofa.position, walkSpeed);
         //checks if already triggered the bed position
         if(atSofa == true)
         {
         //interact with the sofa
         }
     }
else
{
  if(Comfort < Overall&& Busy == false)
{
    transform.LookAt(Sofa); // Face the direction of the
    Sofa
    transform.position =
    vector3.MoveTowards(transform.position.Sofa.position, walkSpeed);
               //checks if already triggered the bed position

           if(atSofa == true)
           {
           //interact with the sofa
           }
    }
   if(Busy == true && isEating == false && isWatchingTV == false)
    {
     //Keep doing what she is doing at that moment
    }
}
}
```

上面添加了3个额外的变量：isEating、isWatchingTV和Busy。将这3个变量考虑进去会帮助她更好地做出决策。当她在吃东西或看电视时如果感到不舒适，她可以坐着完成当前的事情。而如果当前正在做的事情不允许坐下，她就会做一番比较，判断是坐下还是其他事情更重要。举个例子，假如她正在洗澡或工作，她会忽略感到不舒服的事实，而且只要有机会，她就会坐下来并恢复舒适度：



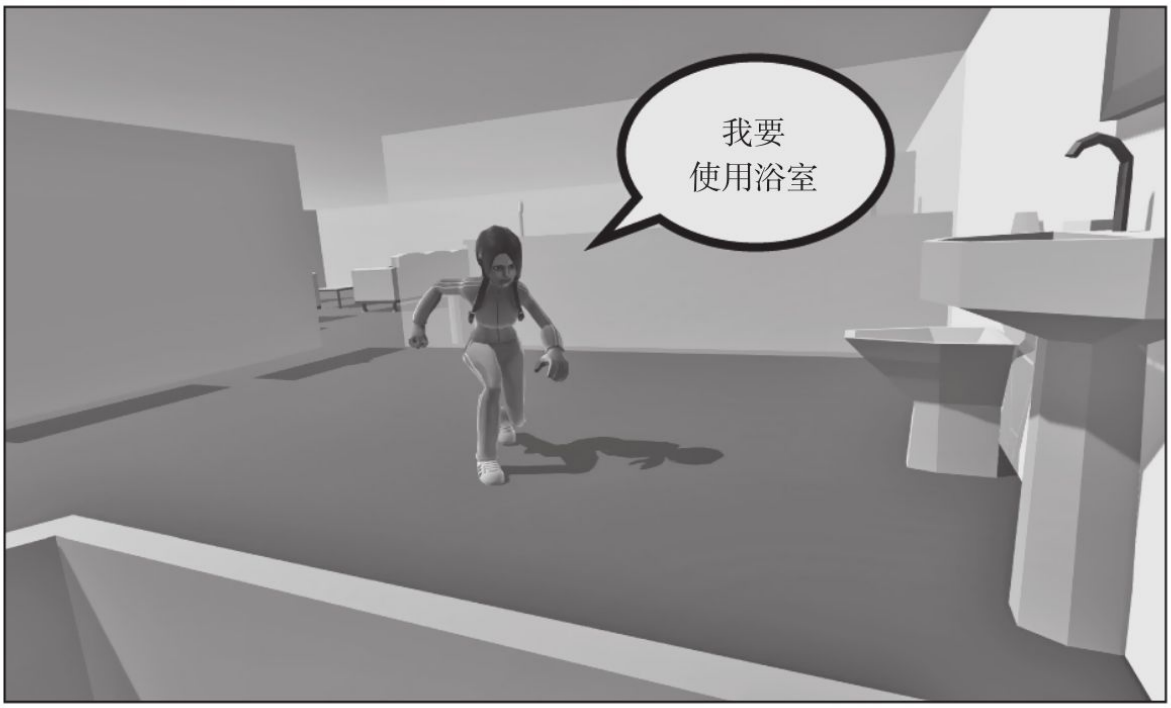
至此还剩两个需求需要完成，很快我们便会有一个可以独自生活，不需要他人来控制和帮忙决策的AI角色。虽然这个AI系统是工作在模拟类游戏中的，但是相同的方法和原理也可以用在其他类型的游戏中。例如在一个实时策略游戏中，工人可以自主地做出决定，不会无所事事地被动等待命令，且当有其他优先事项来临时，它们立即去做更高优先级的工作。

下面是卫生。简单起见，我们让这个需求只与浴室相关：

卫生需求比之前的都要简单，它只有一个问题：索菲要不要去洗澡。我们将要用到的一个额外因素是，不管什么情况，去浴室是最重要的，一旦有需求她会立刻停下手头的一切事，径直走向浴室。

她洗澡时能做的事就是刷牙或与卫生相关的其他一些事情。不过现在，让我们坚持尽可能少的选项来测试AI；一旦基本的函数能正常工作了，我们就可以着手添加更多的动作。作为原型设计，这同时也是一个很好的方法论：首先拥有一个可以工作的基本函数，再逐步添加更多细节。现在看一下针对卫生功能实现的例子：

```
                        void useBathroom ()
                        {



if(Hygiene<10)
{
       transform.LookAt(Bathroom); // Face the direction of the
       Bathroom
       transform.position = vector3.MoveTowards(transform.position.
       Bathroom.position, walkSpeed);
       //checks if already triggered the bed position
if(atBathroom == true)

{

   //choose randomly what to do in the bathroom

}

}

}
```



现在看最后一个需求，愉悦度。这是所有需求中最灵活的一个，我们可以边用餐边看电视，可以坐着看电视，还可以坐着边用餐边看电视。AI可以选择同时做三件不同的事情。乍看之下，让角色在三个不同的需求维度上同时获得点数的提升似乎比较理想，我们晚点会讨论这一点。现在我们只关注愉悦度的因素，规划她决定当前是否需要，以及是否有条

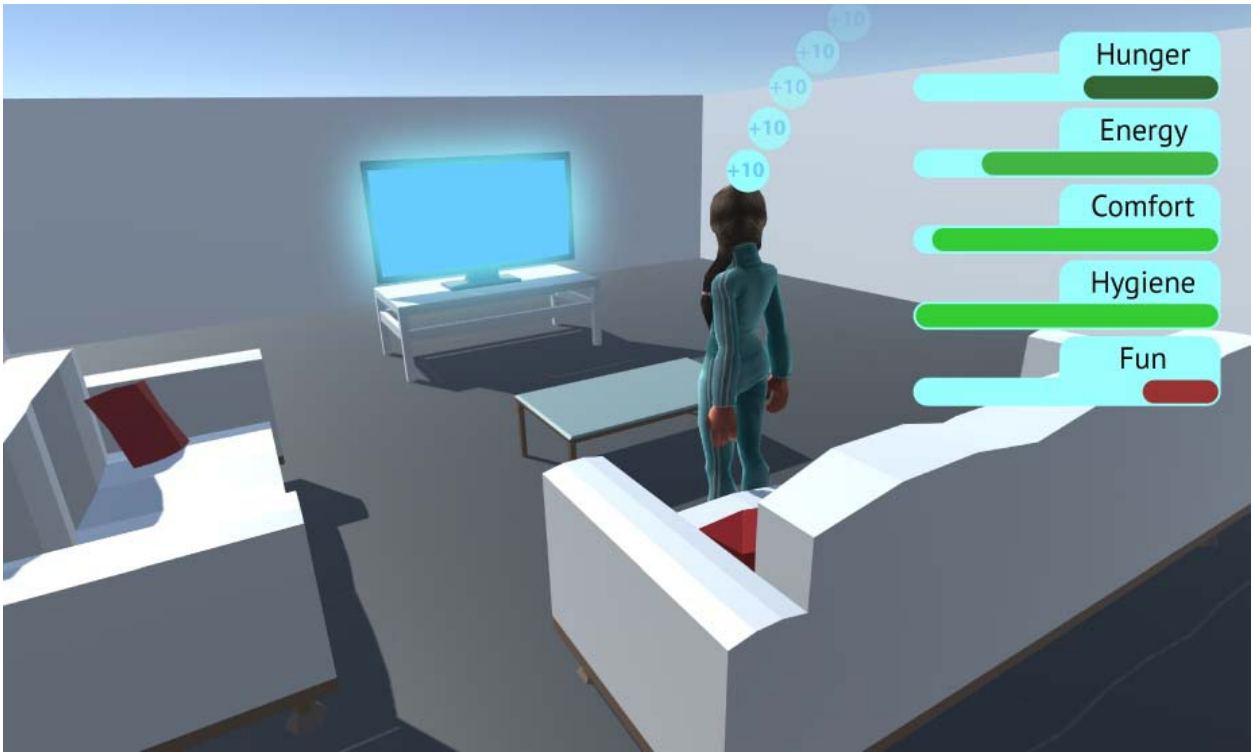
件看电视的步骤：

如果索菲感到无聊，她会问自己是否可以看电视。如果答案是肯定的，那么她可以自由地进行观看，默认状态下她会坐着进行这一行为。但是如果她此时正在忙于某事，那么她需要问自己是否可以在做这件事的同时看电视（在这个例子中，这件事代表吃饭，坐着，

或者两者同时）。简单起见，让我们假设她不需要先走到电视跟前打开电视。我们可以在下面的例子中看到这些是如何在代码中呈现的：

```
private bool isSeat;
private bool televisionOn;
void Bored ()
{
if(Fun<Overall&& Busy == false)
{
   televisionOn = true; // turns on the television
         transform.LookAt(TV); // Face the direction of the television
         transform.position = vector3.MoveTowards(transform.position.
         Sofa.position, walkSpeed);
         //checks if already triggered the bed position
         if(atSofa == true)
         {
          //interact with the sofa
         }
 }
if(Fun < Overall && Busy == true)
{
   f(isEating == true) {
         televisionOn = true; // turns on the television
               transform.LookAt(TV); // Face the direction of the
         television
      }
if(isSeat == true)
  {
         televisionOn = true; // turns on the television
         transform.LookAt(TV); // Face the direction of the television

   }
if(isSeat == true && isEating == true)
{
         televisionOn = true; // turns on the television
         transform.LookAt(TV); // Face the direction of the television
         }
}
else()
{
   //continue doing what she is doing
}
}
```

现在我们得到结论：每一个与需求相关的习惯可以对应不同的实现方式。这么做能确保，AI角色清楚其在游戏中的不同情形下应该做什么，且总会自己去找事做。当然，此AI设计中有一些缺陷是需要完善的。例如，如果索菲当前感到饿或者累，她将必然不能选择继续看电视。为了改善这一点，我们只需简单添加一个概率图，然后使用Overall变量来定义她是否开心。比如说，Overall在50%以上即代表开心。那么在这种情况下，即使她觉得累了，她也可以选择继续看电视。所有这些限制条件现在都可以添加至代码，与此同时，我们也可以不断地向AI角色的行为中添加更多的细节。不过现在，我们只需要关注其中2个：AI平衡性和动态解决问题。


+10
+10
+10
+10
+10
Hunger
Energy
Comfort
Hygiene
Fun

3.3 游戏AI的动态平衡

另一个有趣且非常有用的主题是关于游戏难度的。如果我们和一个人类玩家比赛，比赛的难度将完全取决于对手的经验。如果他对这款游戏非常熟练，显然其相比一个新手玩家会有更大的优势。通常电子游戏会逐步增加难度，以让玩家更好地适应难度变化，不会因为难度突然增加而感到沮丧，也不会因为游戏没有挑战性而感到无聊。而动态游戏难度平衡则是通过为每个玩家创造有趣的体验来解决这个问题。为了使用此方法平衡AI角色，我们会考虑一些游戏中可以根据玩家经验进行调整的动态游戏元素。这些属性可以是如下这些：

· 速度

· 生命值

· 魔法值

· 战斗力

通常，我们使用这些属性来定义AI角色的难度，并通过调节这些属性来让难度变成我们想要的。另一个平衡难度的方法则是调整玩家在游戏中所面对的敌人的数量。在调整难度时我们也要小心，不要制造像橡皮筋那样的敌人。举个例子，如果AI车在玩家后面，那么它会马上变得更快来向玩家提供挑战性。而当玩家在对手车后面时，对手车又会马上降低速度，这种很直观的反应如果不加以调整，那么可能会变得乏味。

在格斗游戏中，开发者通常会这样定义AI的战斗：如果玩家已经靠近，AI会使用拳打或者脚踢；否则，AI会朝玩家靠近。然后使用AI攻击的百分比和时间间隔来调整难度。

在FPS游戏中，游戏AI是由于在开发阶段考虑到玩家的表现而调整的，程序员会调整AI状态值和战术直到其与玩家整体能力相匹配。举个例子，如果玩家的命中率是70%左右，AI角色将使用这个值以保持相对接近人类的表现。

《古惑狼》（Crash Bandicoo）使用的动态游戏难度平衡不会直接改变AI角色的行为，而是表现在动画速度上。如果玩家过关有困难，那么会放慢动画速度。这种难度的调整是根据玩家死亡次数来进行的。这是一个通过考虑玩家想要尝试通关的次数来调整难度的简单方法。

《生化危机4》（Resident Evil 4）作为Capcom在2005年发行的游戏，难度调整的原理相同，不过采用了更复杂的系统。根据玩家的表现，系统会在后台对玩家进行评分，范围在1～10之间。1意味着玩家在游戏中玩的并不顺畅，10代表玩家非常熟练。根据这些评分，敌人的行为会有所不同，其攻击性和造成的伤害值都会有所增减。评分会持续更新，且为了确定玩家熟练与否会考虑各种各样的标准，包括但不限于：玩家杀死一只丧尸需要耗费多少子弹，玩家承受了多少次伤害等等。

《求生之路》（Left 4Dead）也有考虑到玩家的评分，但在这款游戏里不只是增加敌人AI的难度，还会改变敌人的出生地，让玩家在游玩同一个关卡时都会产生不同的挑战。如果玩家刚上手游戏，敌人往往出现在更容易被消灭的地方；而如果玩家之前已经通过了该关卡，敌人就会出现在对玩家来说更难被消灭的位置。

总结游戏开发人员在调整AI角色的难度时所选择的方法，我们需要注意到，难度并不总是要根据玩家的行为上调或下降。一个很好的例子是模拟游戏，其中的关键是需要使得难度变化更贴近真实生活，而不是使玩家的游戏过程变得更困难或更容易，否则它将不再像一个模拟游戏。其他一些例子如《魔界村》（Ghosts'n Goblins），以及最近的《黑暗之魂》（Dark Souls），开发者则明显是在不改变AI行为的情况下，让游戏从头至尾都一直非常有难度。

3.4 总结

在这一章，我们探究了如何使用AFSM来创建AI角色，不论我们将这些角色放置在哪，他们都能自主决策。接着我们学习了如何计算可能性，如何结合之前学过的技巧来创建一个在决策时能够考虑优先级的角色。然后我们讨论了如何使用基于效用的方法去创建一个行为自主的类人角色。最后，我们讨论了一些基于玩家表现调整游戏难度平衡的话题。在下一章，我们会深入讨论环境和AI，以及在不同类型的游戏中，AI应该如何利用地图上的可用空间来为玩家创造挑战，如何与环境互动，如何使环境对自己有利等等。

第4章　环境与人工智能

当我们创建电子游戏AI的时候，最重要的方面之一是它所处的位置。在前面我们了解到，AI角色所处的位置会彻底影响它的行为以及将来的决策。本章中，我们会继续深入探索游戏环境怎样影响AI，以及如何以合适的方式使用它。这将涵盖多种多样的、具有截然不同地图结构的游戏类型，如开放世界类、街机类和赛车类。

作为玩家，我们更喜欢探索生动的世界，里面有很多事情可做，有很多东西可以交互。对游戏开发者或设计师来说，那通常意味着更多的工作量，毕竟玩家可以交互的所有东西都需要仔细地设计，以确保它们能正常工作，还要避免游戏过程中出现bug或者任何意外状况。同样的问题对AI角色本身来说也是如此。如果我们允许角色与场景交互，那需要增加很多工作，想象、规划以及编码，让它能正常工作。玩家或者AI可以选择的事情的数量往往等同于可能带来的问题的数量，所以我们创造游戏时需要特别关注环境。

并不是所有游戏都必须要有地图或者地形，但是动作发生的位置始终在玩法中占据重要地位，AI本身也要清楚这一点。当然有时候，玩家可能没有在意环境和位置对角色产生的微妙影响，但是大多数情况下，这些微妙的变化会为游戏带来良好而愉快的体验。与环境交互是创作电子游戏时一个很重要的方面，因为只有它才能够赋予角色生命，如果没有它，角色就永远只是单纯的模型而已。

另外，我们也不能忘记相反的方面，即角色对游戏环境的影响。如果我们能生活在游戏世界里，我们对环境产生的影响也会是游戏涉及的一个方面。举个例子，我们在森林里扔了一根香烟，有很大可能会点燃树叶造成大火，给森林里的动物们带来不安，还可能会引发更加严重的一系列后果。所以，看看环境对游戏中发生的事情有什么反应也很有趣。在游戏设计过程中，我们可以选择与环境的交互是游戏玩法的一部分，或是仅仅为了改善视觉体验，但有一点可以肯定，无论哪种设计的终极目的都是创建一个人人都喜爱的丰富多彩的环境。本章中，我们会深入探索上面提到的所有内容，还将开始探索不会改变游戏玩法的一些基本交互。最后本章将讨论对游戏体验有深远影响的高级交互。

4.1 视觉交互

视觉交互相对比较基础，它不会对游戏玩法产生直接的影响，但是对打磨游戏与角色的品质有帮助，所以它也是环境的一部分，对提升用户玩游戏的沉浸感有显著贡献。这个话题的例子非常多，几乎在所有的游戏类型中都可以找到。从中我们也可以看到环境作为游戏一部分的重要性，而不仅仅是为了把屏幕填满而已。现在这种类型的交互在游戏中越来越普遍，玩家也乐于接受。如果游戏中能看到一个物体，那么应该能对它做点什么，重不重要都无所谓。这会让环境更加生动和鲜活，显然是件好事。

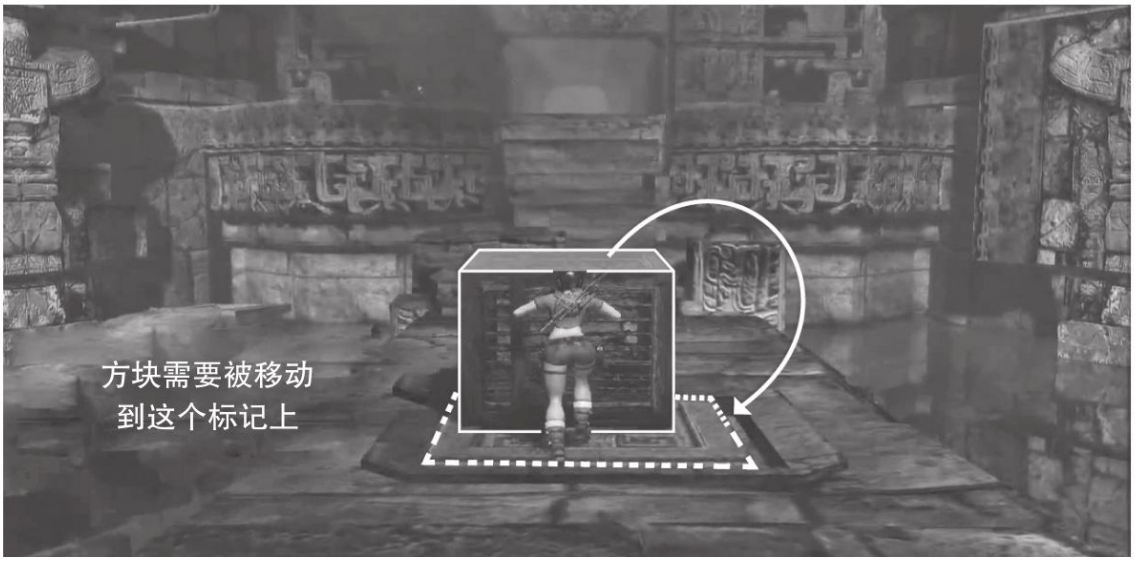
第一批游戏环境交互的例子，首先是元祖版《恶魔城》（Castlevania），它于1986年在任天堂的NES游戏机上发售。从游戏一开始，玩家就可以使用鞭子破坏蜡烛和火炬，它们本来是用作背景的一部分。

该游戏和当时发售的其他一些游戏，为我们对背景、环境与角色之间关系的认知打开了新的大门，同时也带来了很多新的可能。显然，由于主机的硬件机能限制，创建这些在我们现在看来稀松平常的东西，在当年要难得多。但是每一代主机的机能都在持续增长，吸引着我们这样充满热情的开发者创作出令人惊叹的游戏。

所以，我们的第一个视觉交互的例子，就是背景里可以被破坏的物体，它不会对游戏玩法造成直接影响。这种交互类型的例子可以在很多游戏中见到，和用代码实现在其被攻击时播放动画一样简单。然后，我们还可以决定被破坏的物体是否会掉落一点分数或者收集品，以鼓励玩家探索游戏。另外一个例子是，游戏中的物体在角色经过它们时会播放动画或者移动。这和可破坏的物体的原理是一样的，但这次是角色移动到物体位置附近时的一次微妙的交互。这可以用在游戏里不同的东西上，从草地、尘土到水洼，还有会飞走的鸟，或是做鬼脸的人，可能性太多了。当分析这些交互的时候，我们可以轻松地发现它们不必具有人工智能，因为大部分情况下，只需要一个布尔函数来触发预定义的动作即可。但是它们仍然是场景的一部分，因此如果想要实现良好的环境与AI交互的效果，也必须要考虑它们。

4.2 基本环境交互

前面说到，环境成为游戏体验的一部分，它为未来的游戏点亮了新的概念和灵感。而接下来的一步就是把这些小的改动合并到游戏玩法之中，用它们来勾勒出玩家在游戏中的行为。这毫无疑问对电子游戏历史的发展产生了积极的贡献，场景中的一切物体开始获得了生命，玩家开始察觉到周围这些丰富事物的存在。利用环境来达成游戏目标开始成为游戏体验的一部分。



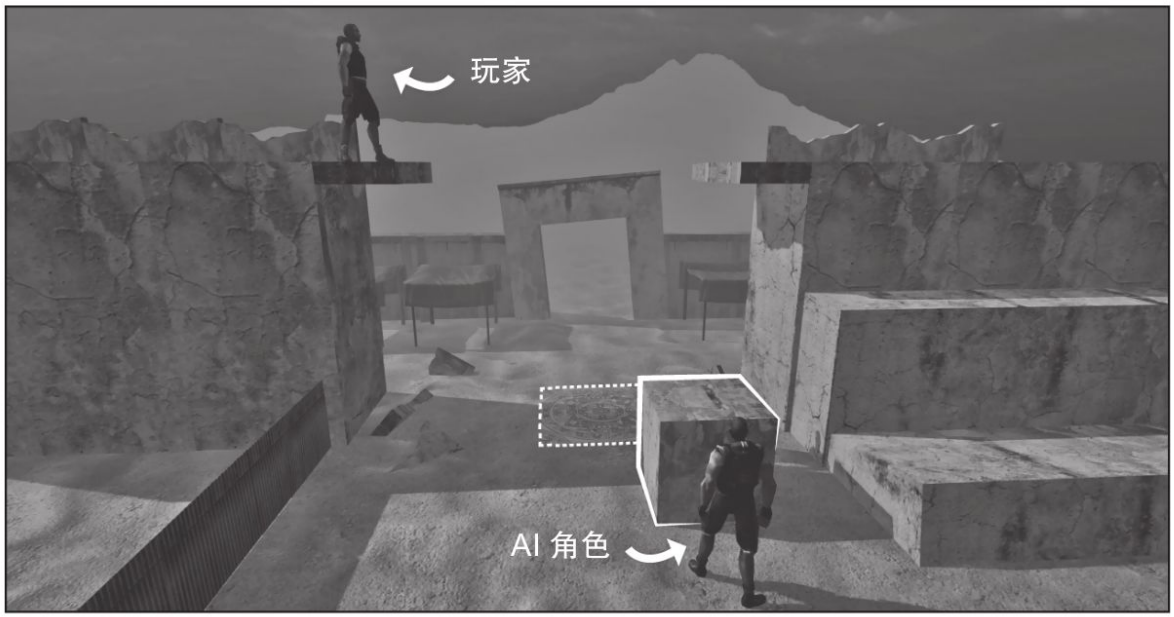
要展示环境元素直接影响游戏玩法的一个例子，《古墓丽影》系列就完美地符合要求。在本例中，主角劳拉·克劳馥（Lara Croft）需要推动方块到标记位置的顶部。这会改变环境并打开一条新的路径，从而使得玩家可以继续推进关卡。在别的游戏中可以找到很多这种类型的挑战，都是必须触发地图中特定的机关以令其他地方的某个事件发生，这可以用于完成游戏中特定的目标。所以，当我们规划地图或者关卡时，会考虑这些交互行为，然后我们会为每一种交互创建相应规则。例如：

```
if(cube.transform.position == mark.transform.position)
{
    openDoor = true;
}
```

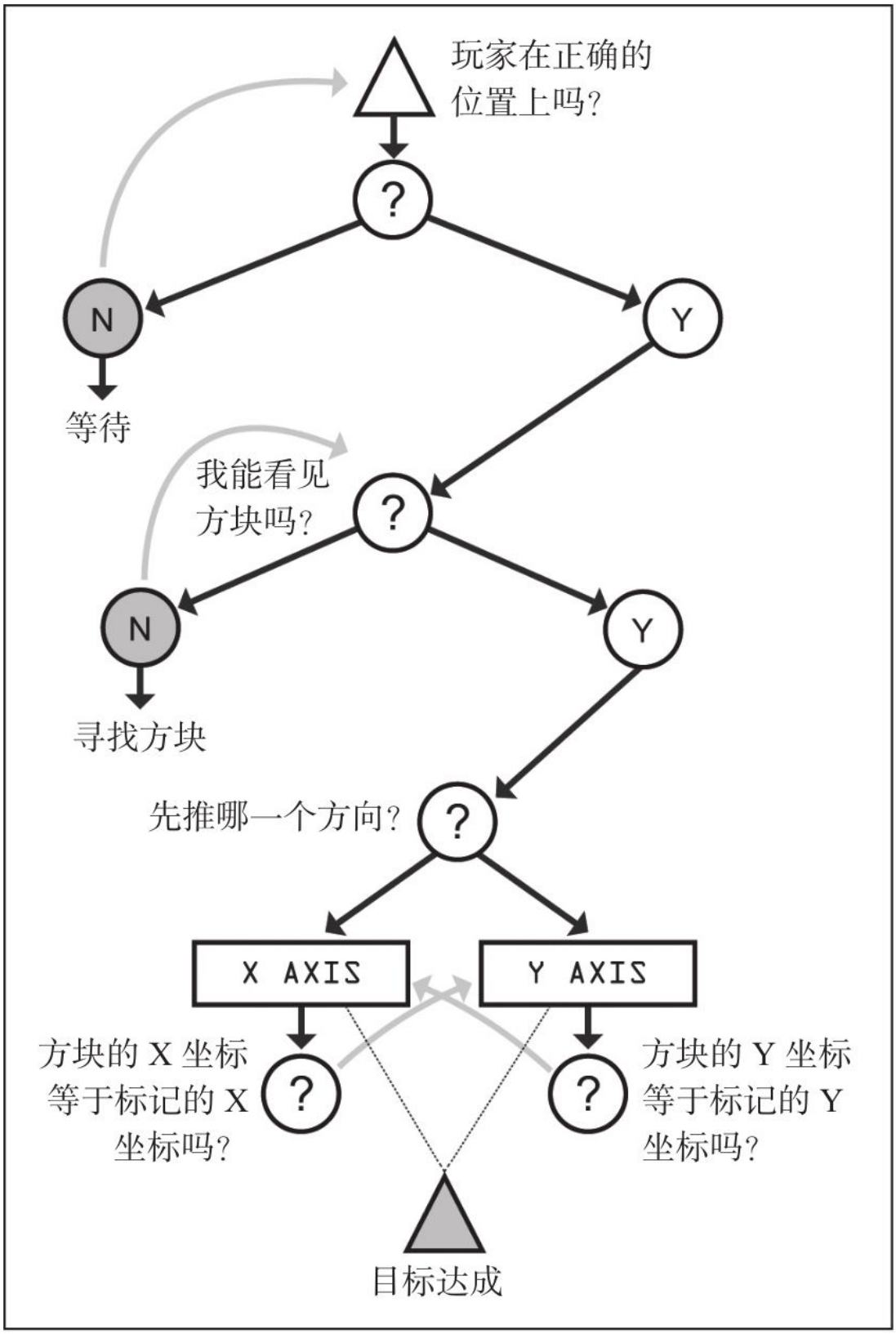
想象一下，劳拉·克劳馥有一个伙伴，它的任务是帮助劳拉把箱子放在合适的位置。这正是本章中将要讨论的一个交互的例子，AI角色需要明白环境是怎样工作的，以及如何利用它。

4.2.1 移动环境中的物体

让我们回到上面提到的环境中，试着重现AI角色帮助玩家达到目标的情形。在这个例子里，我们假设玩家被困在一个地方，在那里玩家够不着需要交互的物体，因而无法独自脱困。我们要创建的AI角色需要能够找到特定方块，并推动它到既定位置。



现在已经有了所有角色和物体，让我们来计划一下在这个场景中AI应当怎样行动。首先，它需要知道玩家在附近，才可以开始搜索和移动方块到正确的位置上。假设方块放在标记上面之后，一个新的障碍物会从沙子里升起，从而使玩家可以继续推进关卡。AI角色可以向不同的方向推箱子，左、右、前或后，以确保方块和位置标记完全重合。


玩家在正确的
位置上吗?
?
N
Y
等待
我能看见
方块吗?
?
N
Y
寻找方块
先推哪一个方向?
?
X AXIS
Y AXIS
方块的 X 坐标
等于标记的 X
坐标吗?
?
?
方块的 Y 坐标
等于标记的 Y
坐标吗?
目标达成

正如前面行为树中所展示，AI角色需要询问和确认自己的每一个行动。要达成目标，首先也是最重要的，是确保玩家处于正确的标记位。如果玩家还未到达该位置，AI需要保持待机状态；如果玩家已经到达了，则AI角色开始询问自己是否在方块附近。如果不是，AI会尝试朝方块的方向移动。当本次行动完毕后，再询问自己同样的问题。如此循环，直到答案变为肯定，此时AI需要算出首先应该朝哪个方向推方块。接下来，AI就会沿Y轴或者X轴推方块，直到方块与标记点对齐，此时目标也宣告完成。

```
public GameObject playerMesh;
public Transform playerMark;
public Transform cubeMark;
public Transform currentPlayerPosition;
public Transform currentCubePosition;

public float proximityValueX;
public float proximityValueY;
public float nearValue;

private bool playerOnMark;


void Start () {

}
void Update () {

// Calculates the current position of the player
currentPlayerPosition.transform.position = playerMesh.transform.position;

// Calculates the distance between the player and the player mark of the X
axis
proximityValueX = playerMark.transform.position.x -
currentPlayerPosition.transform.position.x;

// Calculates the distance between the player and the player mark of the Y
axis
proximityValueYplayerMark.transform.position.y -
currentPlayerPosition.transform.position.y;

// Calculates if the player is near of his MARK POSITION
if((proximityValueX + proximityValueY) < nearValue)
{
   playerOnMark = true;
}
}
```

现在我们开始添加代码，使得AI角色能确认玩家是否在标记位置附近。为此，我们定义计算玩家和标记位置之间距离所需的所有变量。playerMesh代表玩家的3D模型，我们从中提取位置信息保存在currentPlayerPosition变量中。要想知道它是否在标记附近，我们需要一个变量代表标记的位置，在此例中我们创建了playerMark变量来代表该位置。然后我们添加了3个变量用于判断玩家是否在附近，proximityValueX用于计算玩家和标记之间距离的X轴分量。proximityValueY用于计算玩家和标记之间距离的Y轴分量。之后是nea

rValue，用于定义玩家离标记位置具体多远才算是附近，用于触发AI角色开始工作的条件。一旦玩家在标记附近，playerOnMark布尔变量变为真。

 为了计算玩家和标记位置之间的距离，我们使用了以下方法：玩家与标记的距离等于（mark.position-player.position）向量的长度。

现在，为了确定AI角色是否在方块附近，我们会使用相同的方法，计算AI与方块的距离。如下，我们完成了代码，包含两标记的位置（玩家与方块标记）。

```
public GameObject playerMesh;
public Transform playerMark;
public Transform cubeMark;
public Transform currentPlayerPosition;
public Transform currentCubePosition;

public float proximityValueX;
public float proximityValueY;
public float nearValue;

public float cubeProximityX;
public float cubeProximityY;
public float nearCube;

private bool playerOnMark;
private bool cubeIsNear;


void Start () {
    Vector3 playerMark = new Vector3(81.2f, 32.6f, -31.3f);
    Vector3 cubeMark = new Vector3(81.9f, -8.3f, -2.94f);
    nearValue = 0.5f;
    nearCube = 0.5f;
 }

 void Update () {

 // Calculates the current position of the player
 currentPlayerPosition.transform.position = playerMesh.transform.position;

 // Calculates the distance between the player and the player mark of the X
axis
 proximityValueX = playerMark.transform.position.x -
currentPlayerPosition.transform.position.x;

 // Calculates the distance between the player and the player mark of the Y
axis
 proximityValueY = playerMark.transform.position.y -
currentPlayerPosition.transform.position.y;

 // Calculates if the player is near of his MARK POSITION
 if((proximityValueX + proximityValueY) < nearValue)
 {
    playerOnMark = true;
```

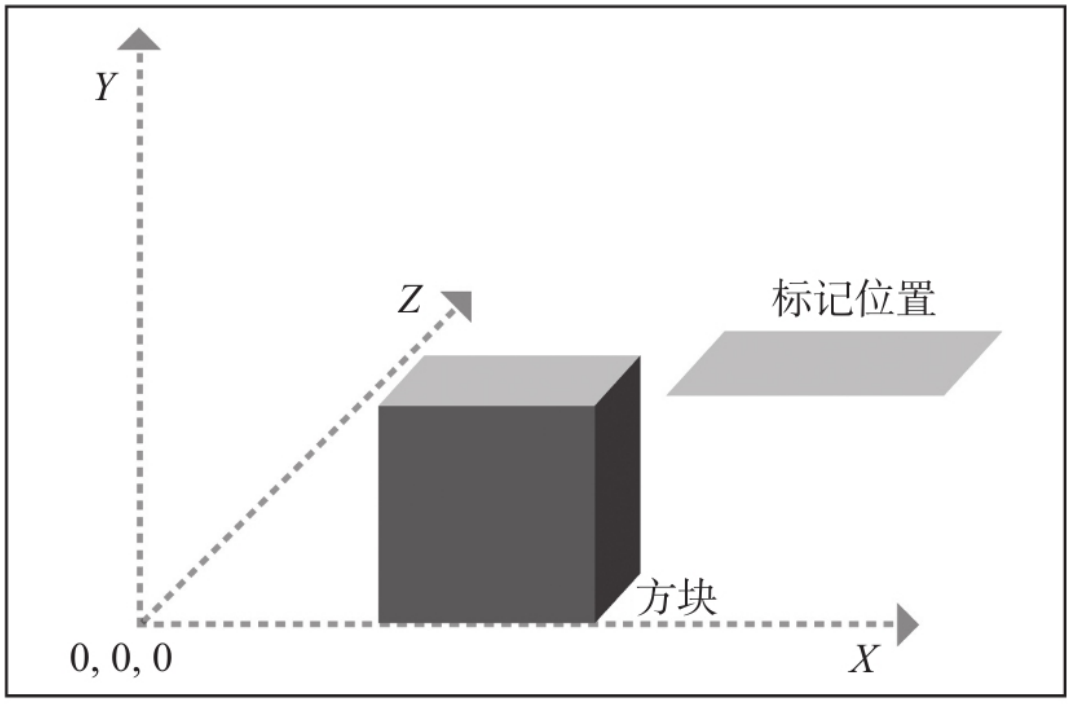
```
 }

 cubeProximityX = currentCubePosition.transform.position.x -
this.transform.position.x;
 cubeProximityY = currentCubePosition.transform.position.y -
this.transform.position.y;

 if((cubeProximityX + cubeProximityY) < nearCube)
 {
    cubeIsNear = true;
 }

else
 {
    cubeIsNear = false;
 }
}
```

现在，AI角色知道了自己是否在方块附近，这可以确定它能否继续执行我们接下来预定好的环节。那么，如果AI不在方块附近会怎样？它应该主动走到方块附近。因此我们把对应的逻辑添加到代码里：

```
public GameObject playerMesh;
public Transform playerMark;
public Transform cubeMark;
public Transform cubeMesh;
public Transform currentPlayerPosition;
public Transform currentCubePosition;

public float proximityValueX;
public float proximityValueY;
public float nearValue;

public float cubeProximityX;
public float cubeProximityY;
public float nearCube;

private bool playerOnMark;
private bool cubeIsNear;

public float speed;
public bool Finding;


void Start () {

 Vector3 playerMark = new Vector3(81.2f, 32.6f, -31.3f);
 Vector3 cubeMark = new Vector3(81.9f, -8.3f, -2.94f);
 nearValue = 0.5f;
 nearCube = 0.5f;
 speed = 1.3f;
}

void Update () {

// Calculates the current position of the player
currentPlayerPosition.transform.position = playerMesh.transform.position;
```

```
 // Calculates the distance between the player and the player mark of the X
axis
 proximityValueX = playerMark.transform.position.x -
currentPlayerPosition.transform.position.x;

 // Calculates the distance between the player and the player mark of the Y
axis
 proximityValueY = playerMark.transform.position.y -
currentPlayerPosition.transform.position.y;

 // Calculates if the player is near of his MARK POSITION
 if((proximityValueX + proximityValueY) < nearValue)
 {
     playerOnMark = true;
 }

 cubeProximityX = currentCubePosition.transform.position.x -
this.transform.position.x;
 cubeProximityY = currentCubePosition.transform.position.y -
this.transform.position.y;

 if((cubeProximityX + cubeProximityY) < nearCube)
 {
     cubeIsNear = true;
 }

 else
 {
     cubeIsNear = false;
 }

 if(playerOnMark == true && cubeIsNear == false && Finding == false)
 {
    PositionChanging();
 }

 if(playerOnMark == true && cubeIsNear == true)
 {
    Finding = false;
 }

 }

 void PositionChanging () {

 Finding = true;
 Vector3 positionA = this.transform.position;
 Vector3 positionB = cubeMesh.transform.position;
 this.transform.position = Vector3.Lerp(positionA, positionB,
Time.deltaTime * speed);
 }
```

到目前为止AI角色已经能够计算自己和方块的距离了。而且如果方块太远，它会主动朝其靠近。一旦接近方块的目标完成，它会继续进行下一阶段，也就是推动方块。最后，它需要计算方块离标记点有多远，然后根据哪一边离得更近来决定应该推方块的哪一面。



方块只能从X轴或者Z轴方向推动，在这里我们不涉及旋转，方块一旦到达按钮上面，按钮就会被激活。考虑到这一点，AI需要计算方块离目标距离的X轴分量和Z轴分量。然后其会比较两个值并选择较远的那个方向，从那里开始推。角色会一直推到在该方向上和目标位置对齐为止，然后再切换到另一个方向，再次推动方块直到完全到达目标点上面。

```
public GameObject playerMesh;
public Transform playerMark;
public Transform cubeMark;
public Transform cubeMesh;
public Transform currentPlayerPosition;
public Transform currentCubePosition;

public float proximityValueX;
public float proximityValueY;
public float nearValue;

public float cubeProximityX;
public float cubeProximityY;
public float nearCube;

public float cubeMarkProximityX;
public float cubeMarkProximityZ;

private bool playerOnMark;
private bool cubeIsNear;

public float speed;
public bool Finding;


void Start () {

        Vector3 playerMark = new Vector3(81.2f, 32.6f, -31.3f);
        Vector3 cubeMark = new Vector3(81.9f, -8.3f, -2.94f);
        nearValue = 0.5f;
        nearCube = 0.5f;
        speed = 1.3f;
}

void Update () {
```

```
 // Calculates the current position of the player
 currentPlayerPosition.transform.position = playerMesh.transform.position;

 // Calculates the distance between the player and the player mark of the X
axis
 proximityValueX = playerMark.transform.position.x -
currentPlayerPosition.transform.position.x;

 // Calculates the distance between the player and the player mark of the Y
axis
 proximityValueY = playerMark.transform.position.y -
currentPlayerPosition.transform.position.y;

 // Calculates if the player is near of his MARK POSITION
 if((proximityValueX + proximityValueY) < nearValue)
 {
    playerOnMark = true;
 }

 cubeProximityX = currentCubePosition.transform.position.x -
this.transform.position.x;
 cubeProximityY = currentCubePosition.transform.position.y -
this.transform.position.y;

 if((cubeProximityX + cubeProximityY) < nearCube)
 {
    cubeIsNear = true;
 }

 else
 {
    cubeIsNear = false;
 }

 if(playerOnMark == true && cubeIsNear == false && Finding == false)
 {
     PositionChanging();
 }

 if(playerOnMark == true && cubeIsNear == true)
 {
     Finding = false;
   }

  cubeMarkProximityX = cubeMark.transform.position.x -
currentCubePosition.transform.position.x;
  cubeMarkProximityZ = cubeMark.transform.position.z -
currentCubePosition.transform.position.z;

  if(cubeMarkProximityX > cubeMarkProximityZ)
  {
    PushX();
  }

  if(cubeMarkProximityX < cubeMarkProximityZ)
  {
    PushZ();
  }
```

```
    }

    void PositionChanging () {

    Finding = true;
    Vector3 positionA = this.transform.position;
    Vector3 positionB = cubeMesh.transform.position;
    this.transform.position = Vector3.Lerp(positionA, positionB,
   Time.deltaTime * speed);


   }
```

随着最后的动作添加到代码中，AI角色已经可以确定它的目标，寻找并推动方块到想要的位置，至此玩家已经完成本环节的全部流程。在这个例子里，我们专注于怎样计算场景里物体和角色间的距离。这对创建相似类型的交互来说很有帮助，适用于把物体放在某个特定位置的大部分情况。

上面的例子展示了一个能够帮助玩家的AI同伴角色，同样的原理可以用在相反的情况下（用在敌人角色上），敌人会尽可能快地找到方块来阻止玩家。

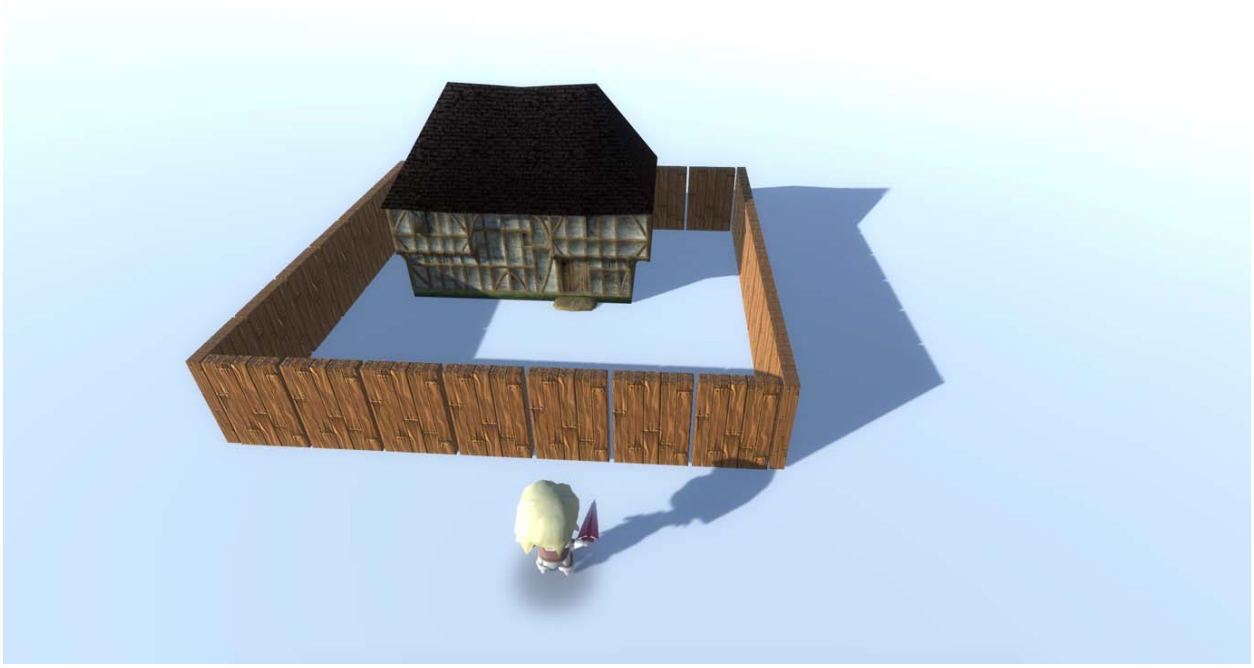
4.2.2 环境中的障碍物

如我们前面所见，可以通过在游戏中使用或移动物体来达成目标，但是有物体挡住路线时会怎么样？障碍物可能是被玩家放在那的或是本来就在那里，无论哪种，AI角色都应该能知道如何应对这种局面。

我们可以先观察一下这种行为。举个例子，Ensemble工作室开发的策略游戏《帝国时代2》。每当由于敌军被坚固的城墙包围导致角色无法到达敌军位置的时候，AI角色就会集合并开始破坏城墙的一角，以便进入城墙。这样的交互非常聪明也非常重要，因为不这样做的话，他们就只能在城墙周围走动以寻找入口，这样就显得不那么智能了。由于城墙是玩家建造的，可以是任何形状，并被放在任何地方，因此在制作AI对手时思考相应的应

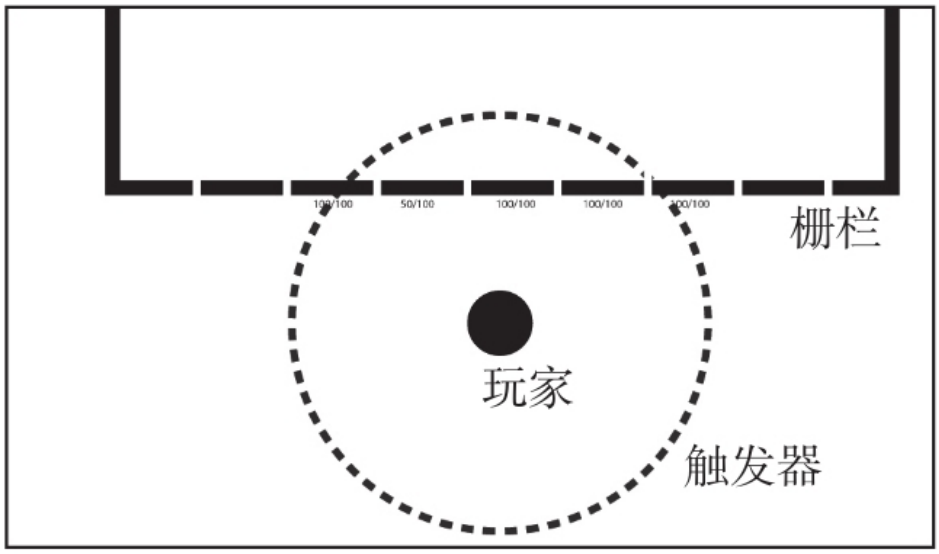
对方式很有必要。

这个例子也与本章主题有关，因为在决策过程中，当我们创建行为树的时候，需要思考有某样东西阻挡了角色的路线以至于它无法达成自己目标的情况。我们在以后的章节会深入探索这个问题，但是现在，我们要简化这个问题，只分析当某个物体干扰了AI角色的行为时，AI应当如何应对。



在这个例子里，AI角色需要进入房屋，但当它到达附近后却发现房屋被木栅栏包围着，无法进入。这时，我们希望角色选择一块栅栏攻击，直到这部分栅栏被毁坏，这样它就可以找到一条路进去了。

对此例而言，需要在考虑栅栏的距离与栅栏当前血量的前提下，计算角色应当攻击哪块栅栏。血量较低的栅栏比满血的具有较高被攻击的优先级，在计算时要把这点考虑在内。



我们希望在角色周围定义一个圆形区域，区域内离角色最近的栅栏会把自身信息给到角色，这样AI就能决定哪一块木板最容易被摧毁了。可以通过不同的方法来实现这一点。比如在栅栏上开启碰撞检测并由角色触发，也可以通过计算栅栏与角色之间的距离。我们定义了距离变量以便让角色感知到栅栏的情况。这里，我们将会计算距离并用它去提醒角色，告诉它栅栏的血量。

```
public float HP;
public float distanceValue;
private Transform characterPosition;
private GameObject characterMesh;

private float proximityValueX;
```

让我们从栅栏（fence）对象的代码开始写起，所有的栅栏都具有以下相同的脚本：

```
    private float proximityValueY;
    private float nearValue;

    // Use this for initialization
    void Start () {

    HP = 100f;
    distanceValue = 1.5f;

    // Find the Character Mesh
    characterMesh = GameObject.Find("AICharacter");
    }

    // Update is called once per frame
    void Update () {

    // Obtain the Character Mesh Position
    characterPosition = characterMesh.transform;

    //Calculate the distance between this object and the AI Character
    proximityValueX = characterPosition.transform.position.x -
  this.transform.position.x;
    proximityValueY = characterPosition.transform.position.y -
  this.transform.position.y;

    nearValue = proximityValueX + proximityValueY;
    }
```

在这个脚本里添加了基本信息，即血量和距离，用它们来联系栅栏本身和AI角色。我们将计算距离的脚本放在了环境物体上而不是角色身上，这带来了更多灵活性，允许我们用它来实现更多的东西。比如，如果角色负责建设栅栏，栅栏就会拥有多种不同的状态，比如正在建造、建造完成和被毁坏，角色能让这些信息为自己所用。

我们继续定义能与环境互动的AI角色。它的主要目标是到达房屋，但是当它到达附近时，它会发现由于周围有栅栏而无法进入。分析了情况之后，我们想要它摧毁一块栅栏以便于最终达成目标，进入房屋。

在角色脚本里，我们加入了静态函数，栅栏可以通过静态函数传递它们当前的血量信息，以帮助AI角色选择较容易的一块栅栏去破坏。

```
                public static float fenceHP;
                public static float lowerFenceHP;
                public static float fencesAnalyzed;
                public static GameObject bestFence;

                private Transform House;

                private float timeWasted;
                public float speed;



                void Start () {

                        fenceHP = 100f;



        lowerFenceHP = fenceHP;
        fencesAnalyzed = 0;
        speed = 0.8;

        Vector3 House = new Vector3(300.2f, 83.3f, -13.3f);

}

void Update () {

        timeWasted += Time.deltaTime;

        if(fenceHP > lowerFenceHP)
         {
             lowerFenceHP = fenceHP;
         }

         if(timeWasted > 30f)
         {
             GoToFence();
         }
 }

 void GoToFence() {

         Vector3 positionA = this.transform.position;
         Vector3 positionB = bestFence.transform.position;
         this.transform.position = Vector3.Lerp(positionA, positionB,
Time.deltaTime * speed);
 }
```

现在我们已经给角色添加了基本信息。fenceHP是静态变量，每个被角色触发的栅栏都会把当前血量赋值给这个静态变量。之后AI就可以分析收集到的信息，与最低的血量比较，最低血量用变量lowerFenceHP表示。角色有一个timeWasted变量代表它在寻找栅栏时已经花费的时间。fencesAnalyzed用于知道是否已经有被发现过的栅栏了，如果没有，会加入第一个发现的栅栏，如果发现了相同血量的栅栏，角色会先攻击它。现在我们更新一下栅栏代码，以便让它们能够访问角色脚本并传递一些有用的信息。

```
public float HP;
public float distanceValue;
private Transform characterPosition;
private GameObject characterMesh;

private float proximityValueX;
private float proximityValueY;
private float nearValue;
void Start () {

        HP = 100f;
        distanceValue = 1.5f;

        // Find the Character Mesh
        characterMesh = GameObject.Find("AICharacter");
}
```

```
 void Update () {

         // Obtain the Character Mesh Position
         characterPosition = characterMesh.transform;

         //Calculate the distance between this object and the AI Character
         proximityValueX = characterPosition.transform.position.x -
this.transform.position.x;
         proximityValueY = characterPosition.transform.position.y -
this.transform.position.y;

         nearValue = proximityValueX + proximityValueY;

         if(nearValue <= distanceValue){
             if(AICharacter.fencesAnalyzed == 0){
                 AICharacter.fencesAnalyzed = 1;
                 AICharacter.bestFence = this.gameObject;
             }

             AICharacter.fenceHP = HP;
            if(HP < AICharacter.lowerFenceHP){
                AICharacter.bestFence = this.gameObject;
            }
        }
}
```

到此例子终于完成，栅栏将当前血量与角色身上的最低血量（lowerFenceHP）比较，如果它的血量低于玩家身上记录的最低血量，那么这个栅栏就是最优栅栏（bestFence）。

这个例子展示了如何让AI角色适应不同的动态物体，同样的原理可以推广，用来和几乎所有的物体交互。它也可以用于物体主动和角色交互，将它们的信息相互联系起来。

4.2.3 用区域阻断环境

当我们创建地图时，常常会有两个或多个不同的区域被用来改变玩法，区域可以包含水、流沙、飞行区、山洞等等。如果我们希望创造一个AI角色能够在任何关卡、任何地方使用，我们需要考虑这点并让AI感知地图上所有不同的区域。通常这意味着我们需要输入更多信息到角色的行为中，包括怎样根据它当前所处的位置来做出反应，或者在某些情况下它能够选择自己该去哪。

它应当避开某些区域吗？它应该更偏好其他区域吗？这些类型的信息也与我们的设计有关，因为这让角色能够感知到周围的环境，选择或适应的同时考虑到它自己的位置。如果不正确地设计这些，会导致一些不自然的决策。例如，在Bethesda Softworks工作室开发的《上古卷轴5：天际》里，我们能够看到一些AI角色在遇到不知道该如何应对的区域时，只是简单地转身往回走，特别是遇到山或河流。

视角色遇到的区域不同，它可能会有不同的反应，或者更新它自己的行为树以适应环境。我们在前面曾创建了一个士兵，根据自己的血量、瞄准成功与否、玩家的血量来改变他反应的方式。现在我们正在研究的是角色如何去利用环境来定义自己需要做什么，这种方法也可以被用来更新前面模拟真实生活的例子。如果索菲来到一个房间中的特定区域，她应当使用环境信息来更新她行为的优先级，并补充和那个区域有关的必要数据。以厨房为例，一旦她到达那儿，她会先准备好早餐，然后顺便带走垃圾。如我们所见，角色周围

的环境会重新定义行为的优先级，或者彻底改变角色的行为。

这就有点像Jean-Jacques Rousseau谈论人性时说过的：“我们天性善良，但被社会腐化。”作为人类，我们自己就是周围环境的体现，因此，人工智能也应当遵循同样的原理。

我们选一个之前就制作好的角色更新它的代码，以令代码在一个不同的场景下工作。这里选择的例子是之前的士兵，我们希望根据三种不同的区域改变它的行为，包括沙滩、河流和森林。为此我们创建了三个公共静态布尔变量，分别是Beach、Forest和River，然后我们定义了地图上的三个区域会开启或关闭它们。

```
public static bool Beach;
public static bool River;
public static bool Forest;
```

因为在这个例子中，每次只有一种情况是真的，所以我们将添加一个简单的代码行，

```
if(Beach == true)
{
 Forest = false;
 River = false;
 }

 if(Forest == true){
 Beach = false;
  River = false;
 }

 if(River == true){
 Forest = false;
```

一旦其被激活，其他选项就会被禁用。

```
        Beach = false;
    }
```

做完了上面的部分，我们可以开始定义每个区域里的不同行为了。例如，在沙滩区域，角色没有可以用来掩护的地方，所以“掩护”行为需要被删除并被更新成新的行为。河流区域可以穿越到对岸，所以角色可以隐蔽并从那个位置攻击。最后，我们可以把角色定义得更谨慎，用树林来掩护自己。根据区域的不同，我们可以改变变量的值来更好地适配环

```
if (Forest == true)
 {// The AI will remain passive until an interaction with the player occurs
 if (Health == 100 && triggerL == false && triggerR == false && triggerM ==
false)
 {
 statePassive = true;
 stateAggressive = false;
 stateDefensive = false;
 }

 // The AI will shift to the defensive mode if player comes from the right
side or if the AI is below 20 HP
 if (Health <= 100 && triggerR == true || Health <= 20)
 {
 statePassive = false;
 stateAggressive = false;
 stateDefensive = true;
 }

 // The AI will shift to the aggressive mode if player comes from the left
side or it's on the middle and AI is above 20HP
 if (Health > 20 && triggerL == true || Health > 20 && triggerM == true)
 {
 statePassive = false;
 stateAggressive = true;
 stateDefensive = false;
 }

 walk = speed * Time.deltaTime;
 walk = speedBack * Time.deltaTime;
 }
```

境，或者创建新的方法以允许我们使用那个区域特有的一些性质。

这部分会在后面谈论AI规划和决策以及战术和感知时深入探讨。

4.3 高级环境交互

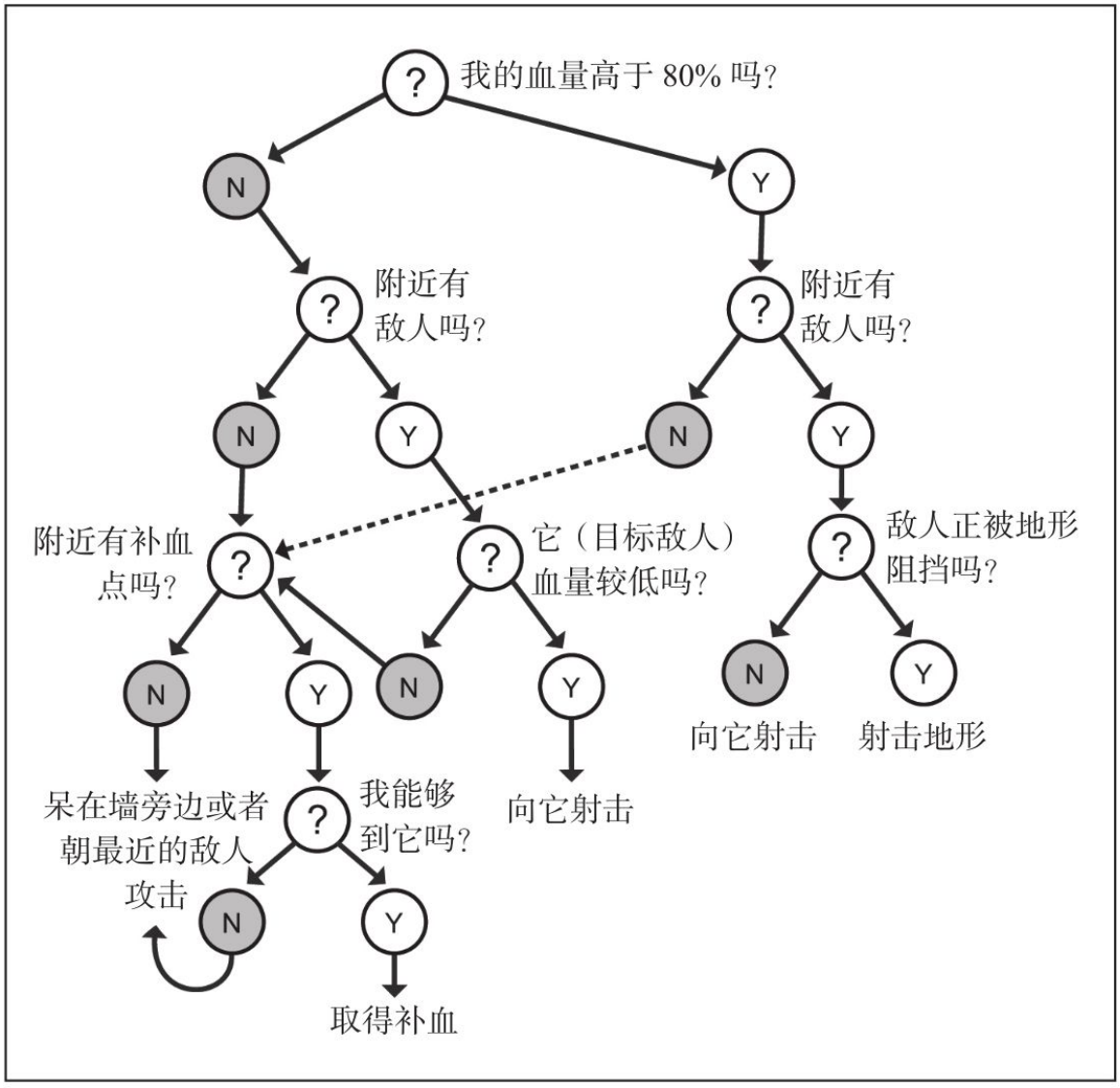
随着电子游戏产业和与之相关的技术不断发展，新的玩法不断出现，很快游戏角色与环境的交互变得越来越有趣，特别是用到了物理元素的时候。这意味着环境变化的结果可以是完全随机的，这就要求AI角色能够持续适应不同的情况。在这里提到的例子是Team17开发的游戏《百战天虫》。在这个游戏里，地图是可以完全被摧毁的，游戏里的AI角色可

以适应这种地形并持续做出聪明的决策。

这个游戏的目标是通过消灭对方所有的虫子来击败敌方队伍，坚持到最后的一方胜利。一开始，角色能够在地图上找到额外的生命点数或弹药，随着时间流逝，还会从天上掉下来更多补给。所以，角色有两个主要目标，即生存和消灭。为了生存，它需要保持一定的血量并远离敌人，另外一方面是选择最佳目标进行攻击，尽可能多地消耗其血量。与此同时，地图会被炸弹和所有角色使用的武器破坏，这对人工智能来说是一种挑战。

4.3.1 适应不稳定的地形

现在，我们来分解这个例子，创建一个能用在这个游戏里的角色。我们从观察地图开始。在底部，有水面会自动杀死虫子。然后，有地形可以供虫子走动，也可以被破坏。最后，地形的空隙，空白区域不能够走上去。游戏一开始，角色们（虫子们）被随机地放置在地图各个位置上，它们可以走动、跳跃、射击。



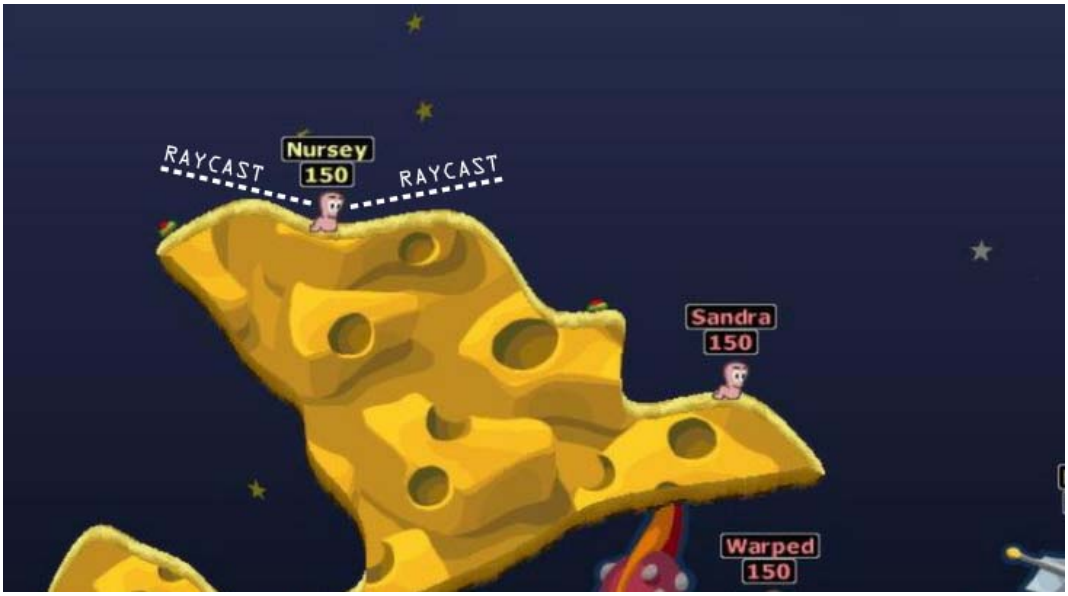
游戏中的角色应可以持续适应会变化的地形，所以我们需要考虑到这一点并将其作为行为树的一部分。如上图所示，角色需要知道它现在的位置、对手的位置、血量和道具。

由于地形会阻挡住角色，AI角色有几率被困在一个位置，既不能攻击也无法取得道具。所以，在它处于诸如此类情况时，都需要给它一些选项。但最重要的一点是，我们需要定义当它无法达成任何一个选项的时候应当怎么做。因为地形可能会形成不同的形态，在游戏过程中有很多时候几乎无法做任何事情，这就是为什么我们需要在这些情况下提供不同

的选项。

例如，在上图中，虫子可能没有足够空间移动，没有一个够得着的道具来获取，也没有一个敌人能正确攻击到。那么它应该怎么做？首先需要让角色获取到周围的信息，以便它能对当前的情形作一个好的判断。在本例中，我们定义角色对最近的敌人展开射击，然后靠近墙面。由于它可能会离攻击最近敌人时产生的爆炸过于接近，因此它应该选择等待在一个安全的角落，直至下一个回合到来。

4.3.2 使用射线检测评估决策



理论上，每回合开始的时候，角色有两条射线，左边一条右边一条，它们会检测两边是否有墙阻挡。这样可以帮忙确定角色应该朝哪边移动才不会被攻击到。当角色想要攻击的时候，我们会再用一条射线指向瞄准的方向，看看是否有什么东西阻挡了攻击的通路。如果有一些物体位于中间，角色可以计算该物体离自己的距离来判断能否安全射击。

因此，每一个角色都应该能访问一个共享的列表，里面有所有虫子所在的位置，这样它们就能比较和每一只虫子的距离并选择最近的一只来射击。另外，我们添加了两条射线来检查两边是否有东西阻挡，而且还有基本的信息来让角色适应持续变化的地形。

```
 public int HP;
 public int Ammunition;

 public static List<GameObject> wormList = new List<GameObject>();
 //creates a list with all the worms
 public static int wormCount; //Amount of worms in the game
 public int ID; //It's used to differentiate the worms

 private float proximityValueX;
 private float proximityValueY;
 private float nearValue;
 public float distanceValue; //how far the enemy should be

 private bool canAttack;

 void Awake ()
 {
        wormList.Add(gameObject); //add this worm to the list
        wormCount++; //adds plus 1 to the amount of worms in the game
 }

 void Start ()
 {
        HP = 100;
        distanceValue = 30f;
 }

void Update ()
 {
        proximityValueX = wormList[1].transform.position.x -
this.transform.position.x;
        proximityValueY = wormList[1].transform.position.y -
this.transform.position.y;
        nearValue = proximityValueX + proximityValueY;

        if(nearValue <= distanceValue)
        {
            canAttack = true;
        }

        else
        {
            canAttack = false;
        }

        Vector3 raycastRight =
```

```
transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, raycastRight, 10))
            print("There is something blocking the Right side!");

        Vector3 raycastLEft =
transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, raycastRight, -10))
            print("There is something blocking the Left side!");
 }
```

4.4 总结

本章我们探索了与场景交互不同的方法。本章中展示的技术可以扩展到更广泛的游戏类型中，用来解决从基础到高级的AI角色与场景交互问题。现在我们懂得了怎样创建能够被AI角色利用的交互动态物体，这会让我们每次进行游戏都有全新体验。本章还稍微涉及了后面章节才会深入探讨的话题，例如与其他AI角色之间的交互与决策。

下一章，我们会探讨动画行为。动画构成了玩家对人工智能角色的视觉感受的一部分，用它来展现出AI角色的行为有多么逼真，这一点很重要。我们会讨论动画状态图，游戏玩法与动画，动画行为，以及动画结构。

第5章　动画行为

在想到AI的时候，我们经常会联想起聪明的机器人或是机械的物件，它们能完美地完成一系列动作，而相同的联想也发生在电子游戏的AI上。我们倾向于认为敌人或盟友会以聪明的决策来表现出、反馈出、想出或者做出很多事情，这也没错，但是另一个重要的方面常常被忽略，那就是动画。为了创建可信的、真实的AI角色，动画是最重要的方面之一。动画定义了视觉交互，也就是角色在做某件事情时，它看起来是怎样的。为了使角色真实可信，让动画能够完美地匹配角色的某个功能是非常重要的。在本章中，我们会看到一些技术和方案，它们可以用来创造出完美匹配角色行为的动画。创建和使用动画的方法在玩家角色和AI角色上是一样的，但本章会重点讨论怎样把之前学到的关于创建AI的技术与动画结合起来。

5.1　2D动画与3D动画的对比

电子游戏动画可以分为2D动画和3D动画。它们各有特点，要充分考虑到它们的特性让它们在游戏开发中为我们所用。下面来看看它们的主要不同点。

5.1.1　2D动画-精灵

随着主机和电脑开始允许游戏开发者使用动画，依靠漂亮的视觉效果来表现角色的移动与动作，使得游戏变得更加丰富起来了。这也打开了许多创新思路，人们开始创造新的游戏类型、改进旧的游戏类型，以使游戏更加吸引人。从那开始，几乎所有的游戏都开始在动画上投入巨大精力了。

2D动画的处理类似于迪斯尼制作电影的方法。先绘制影片的每一帧，每秒大约有12帧。当时游戏无法使用真正的图画，但是可以用坐标来绘制画面上的每一个点（像素），以此来表现出想要的人物或者动物。剩下的处理是类似的。他们需要用这种方法来创建动画的每一帧，但是由于这是一个艰难而漫长的过程，他们会忽略大量的细节与复杂度。最后他们制作好让角色动起来所必需的每一帧。对程序来说，则需要按特定顺序读取每一帧，并

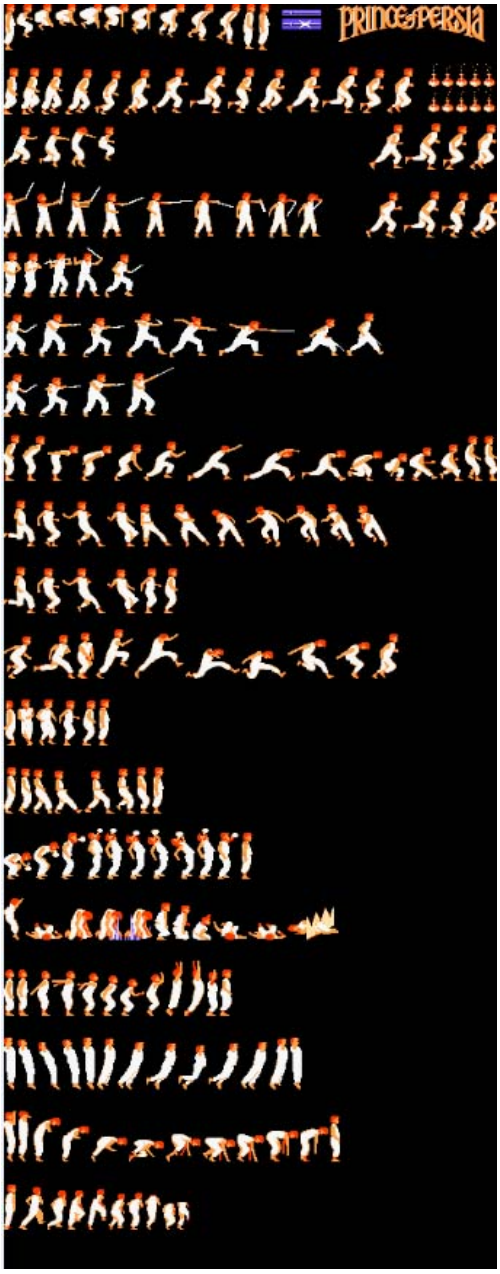
按顺序播放出来。

上图中，可以看到8位机时代的一个例子，上面展示了《超级马里奥》里的角色马里奥的所有动作。如我们所见，有跑步、跳跃、游泳、死亡、刹车和下蹲，某些动作只有一帧。这里没有出现流畅的动作衔接，动画与游戏玩法是结合在一起的。因此，如果我们想要包含更多动画，那么就必须创建更多的帧。对动画的复杂度来说也是一样的，如果想要

动画有更多细节，那么就必须有更多动画帧和过渡帧。这让创建动画的过程非常地长，但是随着硬件机能的发展，实现这个过程需要的时间变短了，而效果也更好。

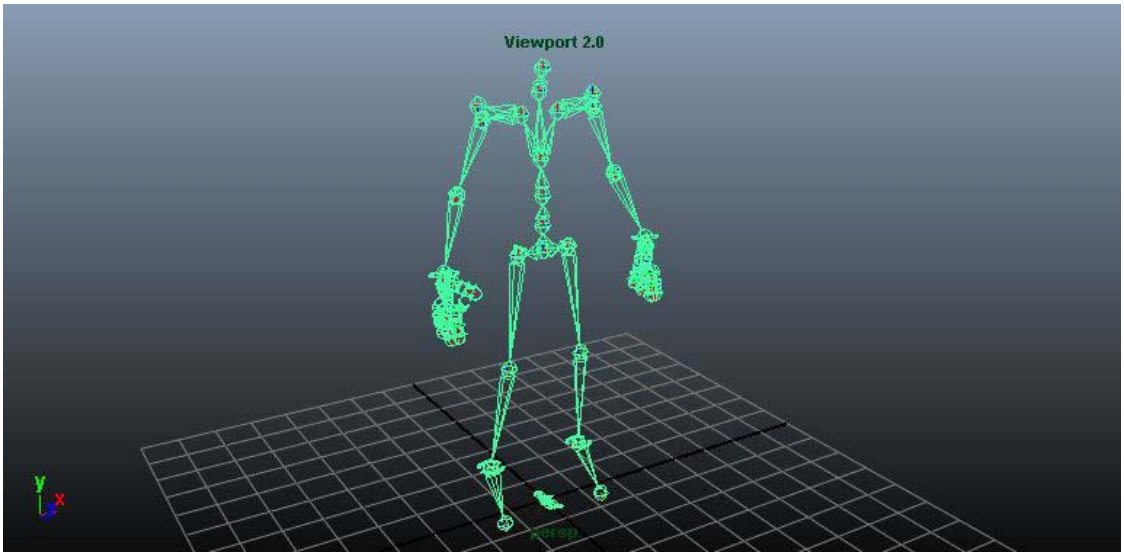
2D动画的另一个例子是1989年发售的电子游戏《波斯王子》（下图展示了游戏中的动画）。由于参考了现实世界的人物动作，其品质、细节以及平滑的过渡效果非常惊人，也把未来游戏的品质标准提得更高了。因此在那个时代，游戏开发者开始考虑动画的切换、平滑，以及如何在不添加大量帧的情况下创建更多的动作：

今天，我们仍然在使用和2D游戏一样的处理流程，我们拥有包含所有动画的精灵表单，然后写代码来让它们在角色做动作时播放出来。使用精灵表单（Sprite sheet）不像使用3D骨骼动画那么舒服，但是其有一些有用的技巧可以用来创建平滑的过渡，并且把动画代码从游戏玩法代码中拆出来。


PRINCE of PERSIA

5.1.2　3D动画-骨骼结构

使用3D模型与骨骼动画创建游戏是目前非常流行的选择，主要原因之一是需要花费的时间更短。我们只需要创建一次3D模型，之后就可以绑上骨骼结构来让它运动了。还可以在另一个3D模型上使用相同的骨骼结构与皮肤，就能获得和之前同样的动画。使用3D动画显然在大型项目中更合适，其可节约大量辛苦工作的时间，而且骨骼动画还允许我们在不用重做的情况下更新角色的局部动画。可以只设计一个特定部位的动作让其他部分静止或做另外的动作，这种方法有助于提高动画品质，以及节约时间和资源。对平滑地切换动作来说这也十分有用，且这还可以用来同时播放两个动作：



主要的区别

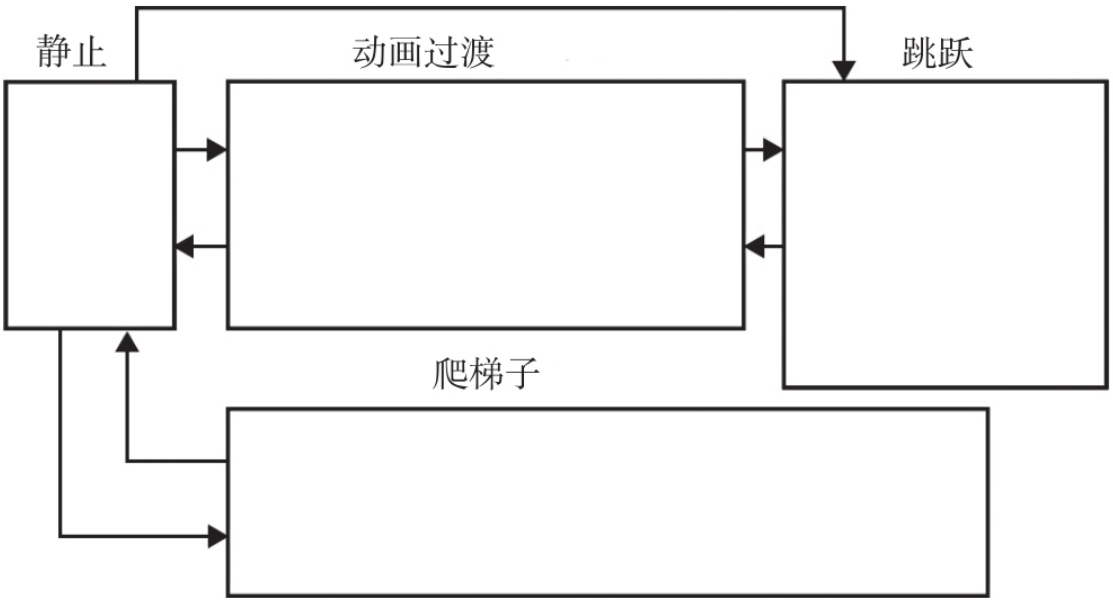
精灵表单与骨骼动画是两种截然不同的动画类型，这两种形式在我们编写Gameplay内容时，会带来完全不同的变化。使用精灵，会受制于动画帧的数量，而且无法用代码改变动画的效果；而使用骨骼动画，可以在代码中定义需要让角色的哪个部分动起来，还可以使用物理引擎根据环境来调整动画。

最近出现了一些新技术，允许在2D模型中使用类似的骨骼动画，但是相比3D骨骼动画来说还是有很大限制。

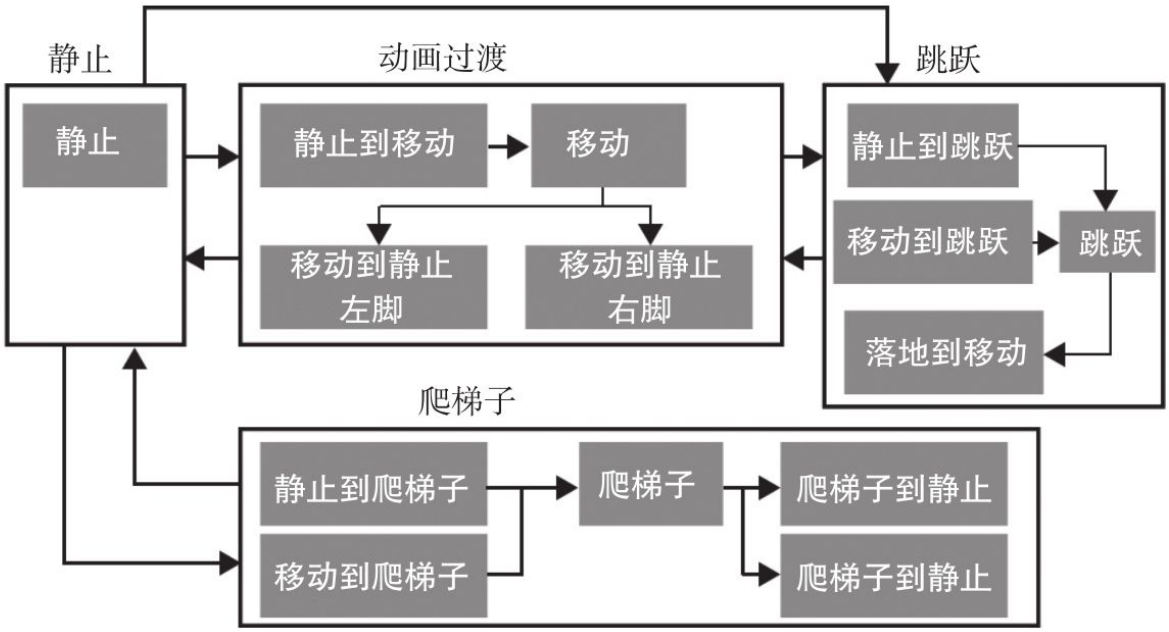
5.2 动画状态机

我们已经讨论了行为状态，其中定义了角色所有可能的行为以及如何把它们串联起来。动画状态机是一个相似的过程，但不是定义行为，而是定义动画。开发角色并创建行为状态的时候，可以在代码里给行为分配动画，定义在哪个状态角色开始奔跑，当奔跑开始时，走路的动画停止、奔跑的动画开始播放。这种将动画与Gameplay代码结合起来的方法看起来很简单，但是这不是最好的方式，而且以后要修改代码时，其会让问题变得复杂。

这个问题的解决方案是为动画创建单独的状态机。这种方案允许我们对动画有更好的控制，而不必担心改变角色本身的代码。这对于程序员与动画师协同工作来说也是个好主意，因为动画师可以给动画状态机添加更多动画而不会对代码产生影响。

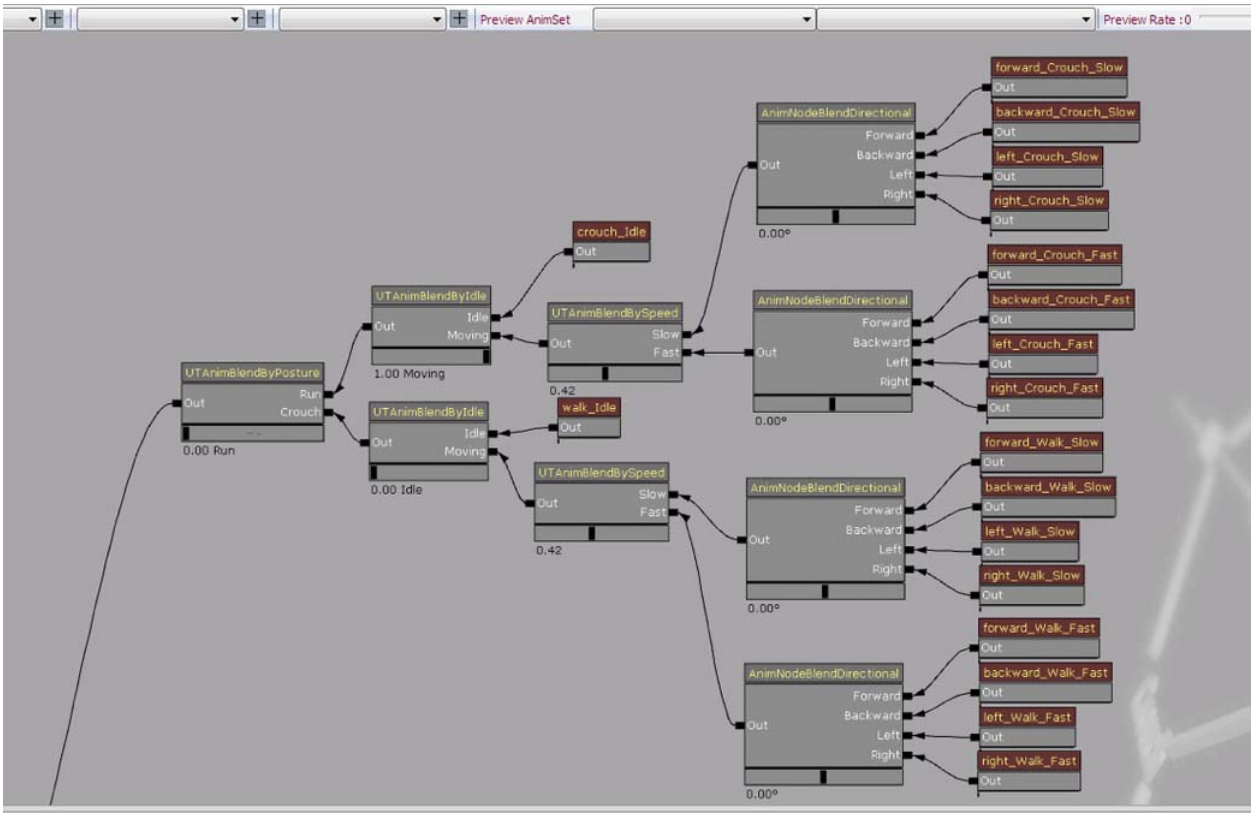


在下图中，我们可以看到简单的行为状态机例子，其中角色可以静止不动、移动、跳跃或者使用梯子。一旦这部分完成，就可以开始设计实现与它配合工作的动画状态机了：

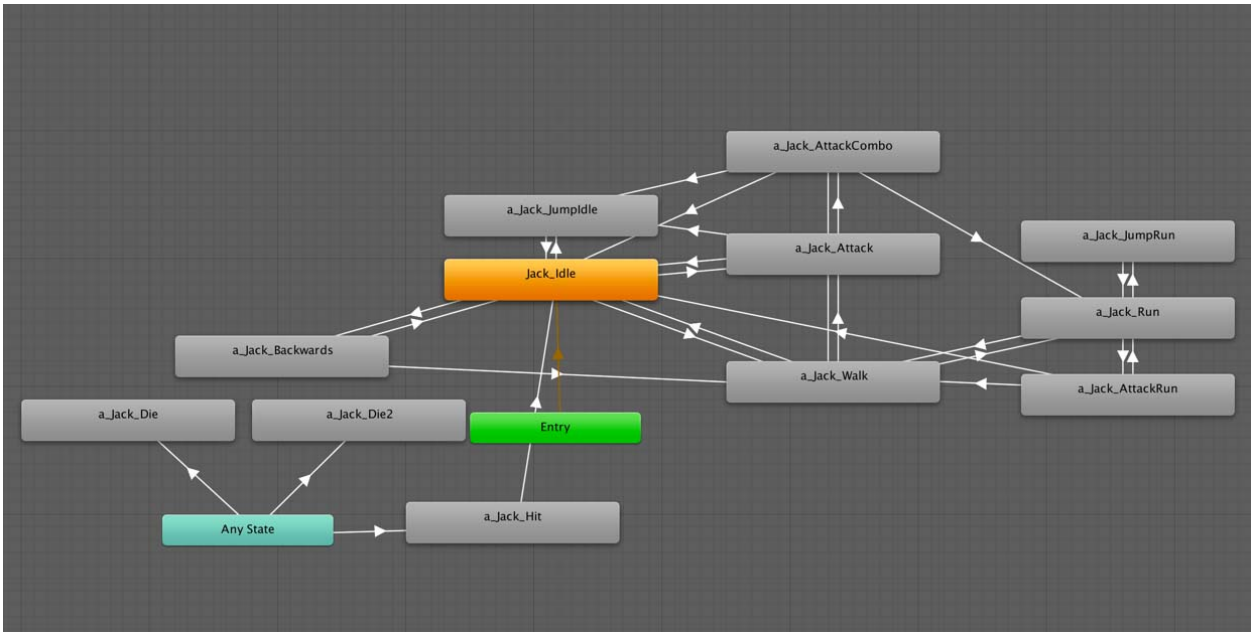


如上图所见，动画状态的数量比行为状态要多，这就是为什么角色结合动画最好的方式，是把Gameplay逻辑和动画分离开来。开发游戏时，使用的是表达式和值，所以走路和跑步的唯一区别是角色速度变量的值。而使用动画状态的原因，是要把走路或跑步这个信息转化为视觉效果，动画是根据行为状态来工作的。使用这种方法并不意味着动画不能干预游戏逻辑，因为根据需要，仍然可以把相关动画信息反馈给游戏逻辑代码，来改变游戏逻辑。

这种方法可以用在2D或3D动画中使用。流程都是完全一样的，也可以在最流行的游戏引擎中使用，比如CryENGINE、Unity和Unreal Development Kit。要让它正常工作，需要导入所有动画到游戏中，然后为动画分配状态：



既然我们已经导入了所有动画到动画状态机窗口中，接下来就需要根据代码来配置每个动画何时播放。这里的表达式可以用整数、浮点数、布尔值和触发器。有了这些，可以定义每个动画播放的时机。给动画连线的时候，就使用了表达式的值来决定什么时候切换动画的状态：

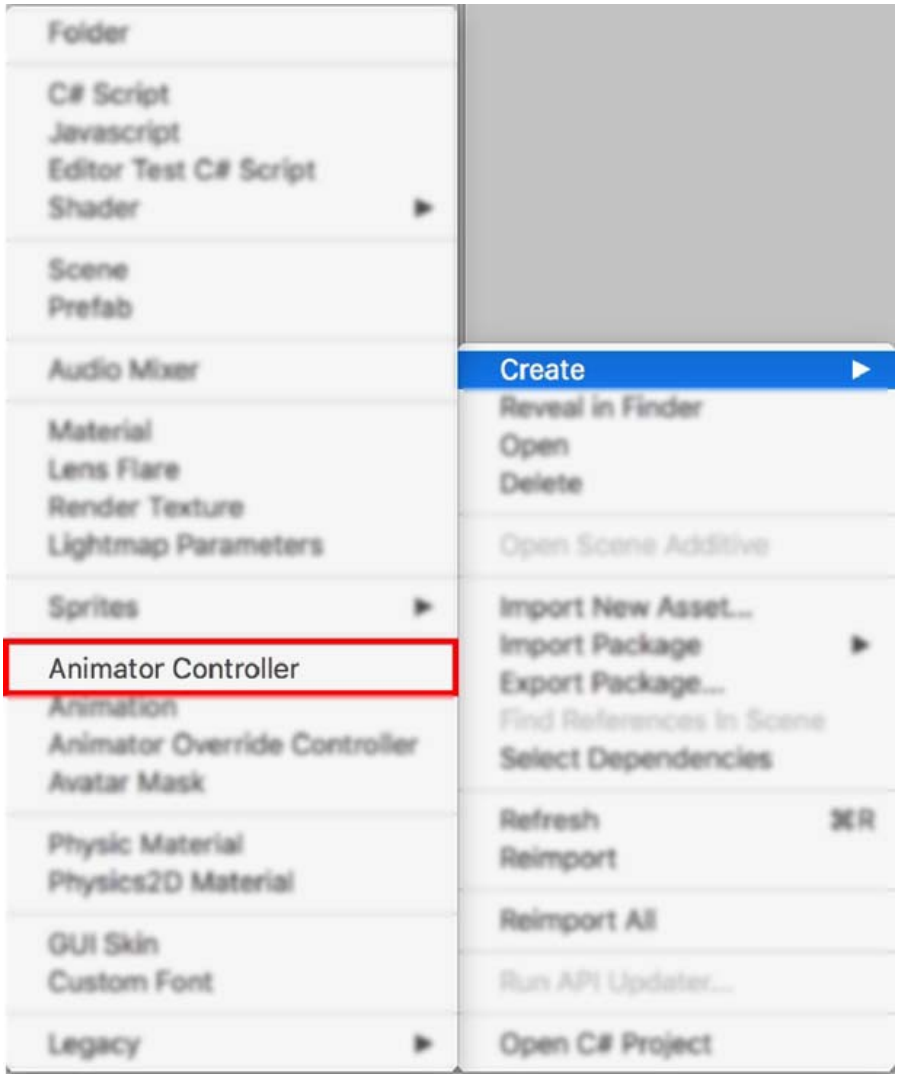


这里就是定义走路和跑步之间区别的地方。如果角色的移动速度达到某个值，就会开始播放跑步动画，而当减小到这个值以下时，就会改为播放走路动画。

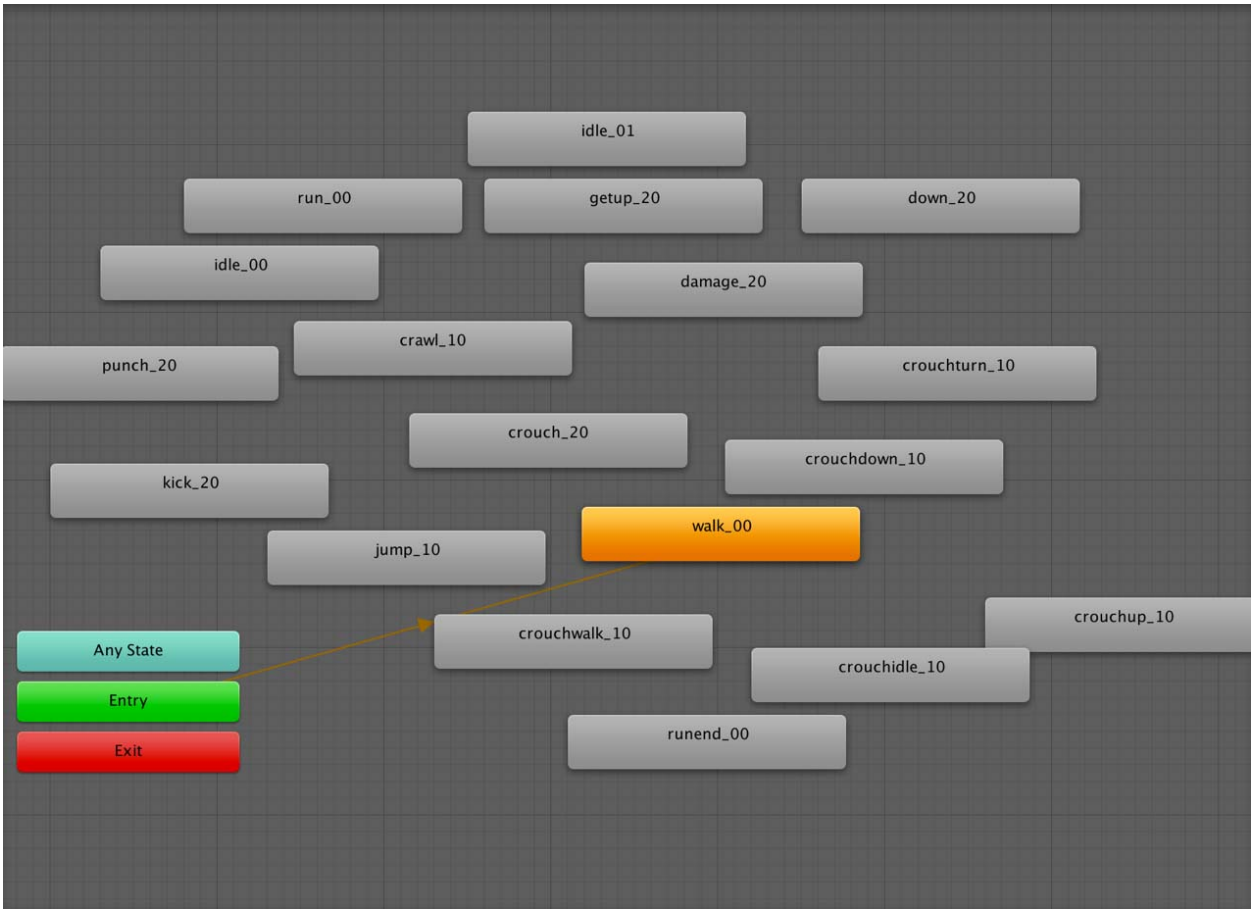
想要多少状态就可以定义多少，和Gameplay状态无关。看看移动的例子。可以定义为角色在很慢地移动时动画效果看起来像它在潜行，再快一点就是走路，再快一点角色就开始跑步，最后如果移动速度足够快，它就会长出一对翅膀好像它在飞。如我们所见，将动画与玩法分离非常方便，因为可以随意修改、删除、新增动画，而不需要修改代码。

现在继续这个例子。使用前面所有章节学到的东西，将配置角色根据Gameplay里的行为和环境来播放动画。从导入模型和所有动画开始。之后创建一个新的动画状态机，这里叫做animator。然后需要把动画状态分配给角色。

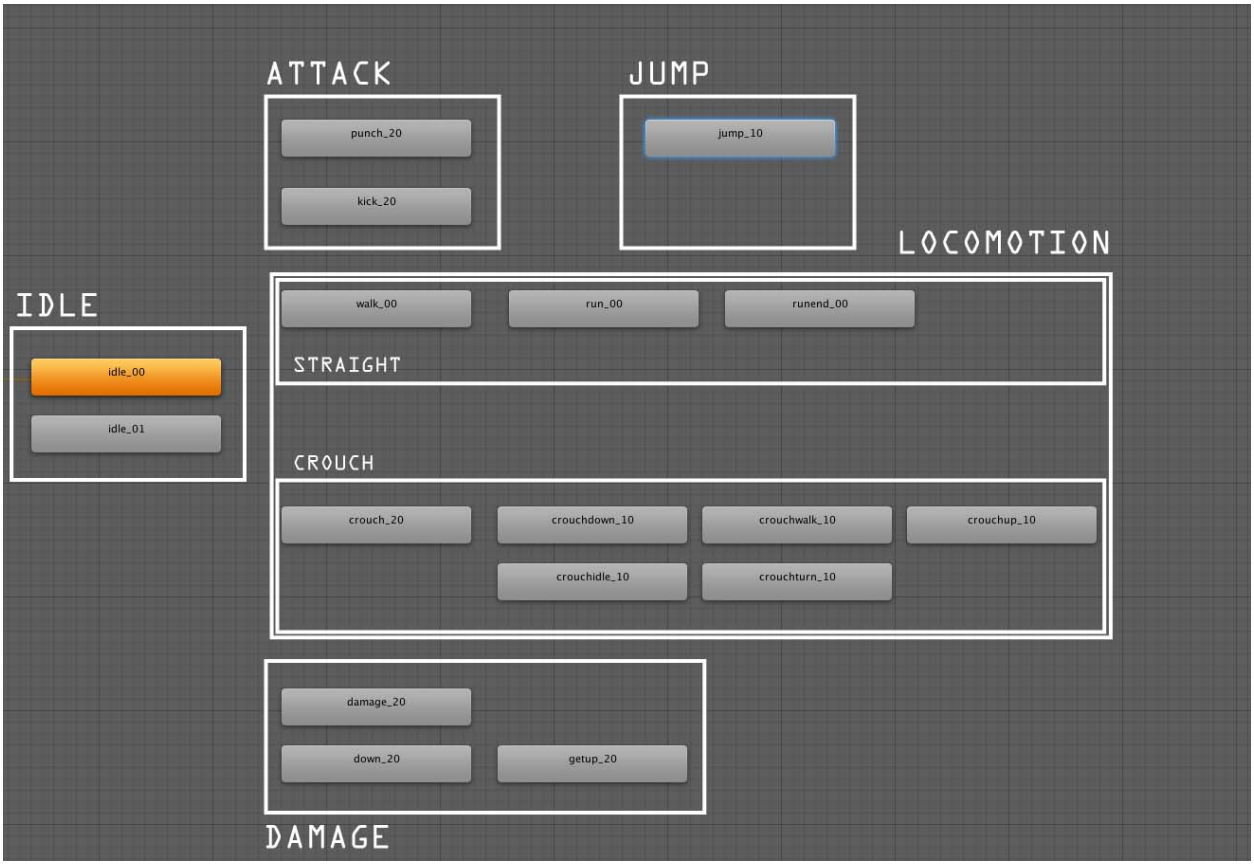
导入到游戏里的角色理论上应该处于默认的姿势，比如T形姿势（上面的截图展示的）。然后导入动画并将它们添加到Animator Controller中。



现在，如果点击角色character，并打开刚才创建的动画状态机，里面会是空的。这是正常现象，因为需要手动添加要用的动画。

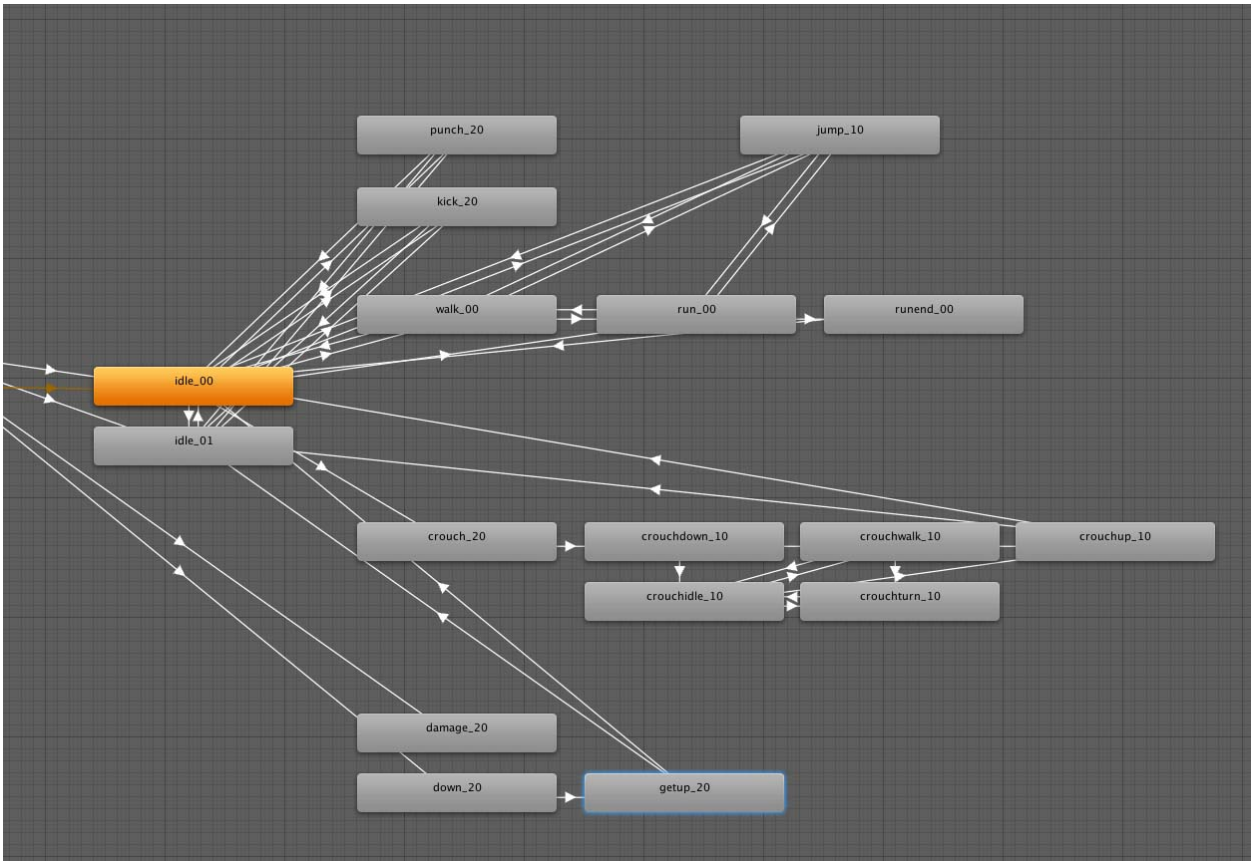


添加完毕后，就需要排列一下所有的状态，以便连接它们：



这样就根据游戏状态分离了不同的动画，比如IDLE（静止）、ATTACK（攻击）、JUMP（跳跃）、LOCOMOTION（动画过渡）和DAMAGE（伤害），如上图所示。有两个不同的动画用于IDLE状态，还有两个用于ATTACK。我们想要它们随机播放，与Gameplay代码无关，所以可以添加很多动画来增加多样性。在运动状态中，有两个分组STRAIGHT和CROUCH，也就是站着走和蹲着走。把它们都归在运动状态里，因为需要根据控制器的摇杆位置来确定动画的状态。

现在可以开始连接这些动画了，做这个的时候可以暂时忘了动画如何激活，专注于考虑播放顺序即可：



按需连接好之后，可以开始定义它们如何播放了。这里，我们需要看看角色代码并利用其中的变量来控制动画。在代码中，有访问动画状态机的变量，这个例子里它们是动画状态机Animator，整型变量Health和Stamina，浮点变量movementSpeed、rotationSpeed、maxSpeed、jumpHeight、jumpSpeed和currentSpeed，最后有一个布尔变量用来检查玩家是否还活着：

```
public Animator characterAnimator;
public int Health;
public int Stamina;
public float movementSpeed;
public float rotationSpeed;
public float maxSpeed;
public float jumpHeight;
public float jumpSpeed;

private float currentSpeed;
private bool Dead;

void Start () {

}
```

```
void Update () {

    // USING XBOX CONTROLLER
    transform.Rotate(0,Time.deltaTime * (rotationSpeed *
    Input.GetAxis ("xboxlefth")), 0);

    if(Input.GetAxis ("xboxleft") > 0){
        transform.position += transform.forward * Time.deltaTime *
        currentSpeed;
        currentSpeed = Time.deltaTime * (Input.GetAxis
        ("xboxleft") * movementSpeed);
    }

    else{
        transform.position += transform.forward * Time.deltaTime *
        currentSpeed;
        currentSpeed = Time.deltaTime * (Input.GetAxis
        ("xboxleft") * movementSpeed/3);
    }

    if(Input.GetKeyDown("joystick button 18") && Dead == false)
    {

    }

    if(Input.GetKeyUp("joystick button 18") && Dead == false)
    {

    }

    if(Input.GetKeyDown("joystick button 16") && Dead == false)
    {

    }

    if(Input.GetKeyUp("joystick button 16") && Dead == false)
    {

    }
    if(Health <= 0){
        Dead = true;
    }
}
```

我们把这些变量传给动画状态机。移动和currentSpeed的值由左摇杆控制，所以如果只推动摇杆一点点，角色会播放走路动画。如果推满摇杆就会播放跑步动画。

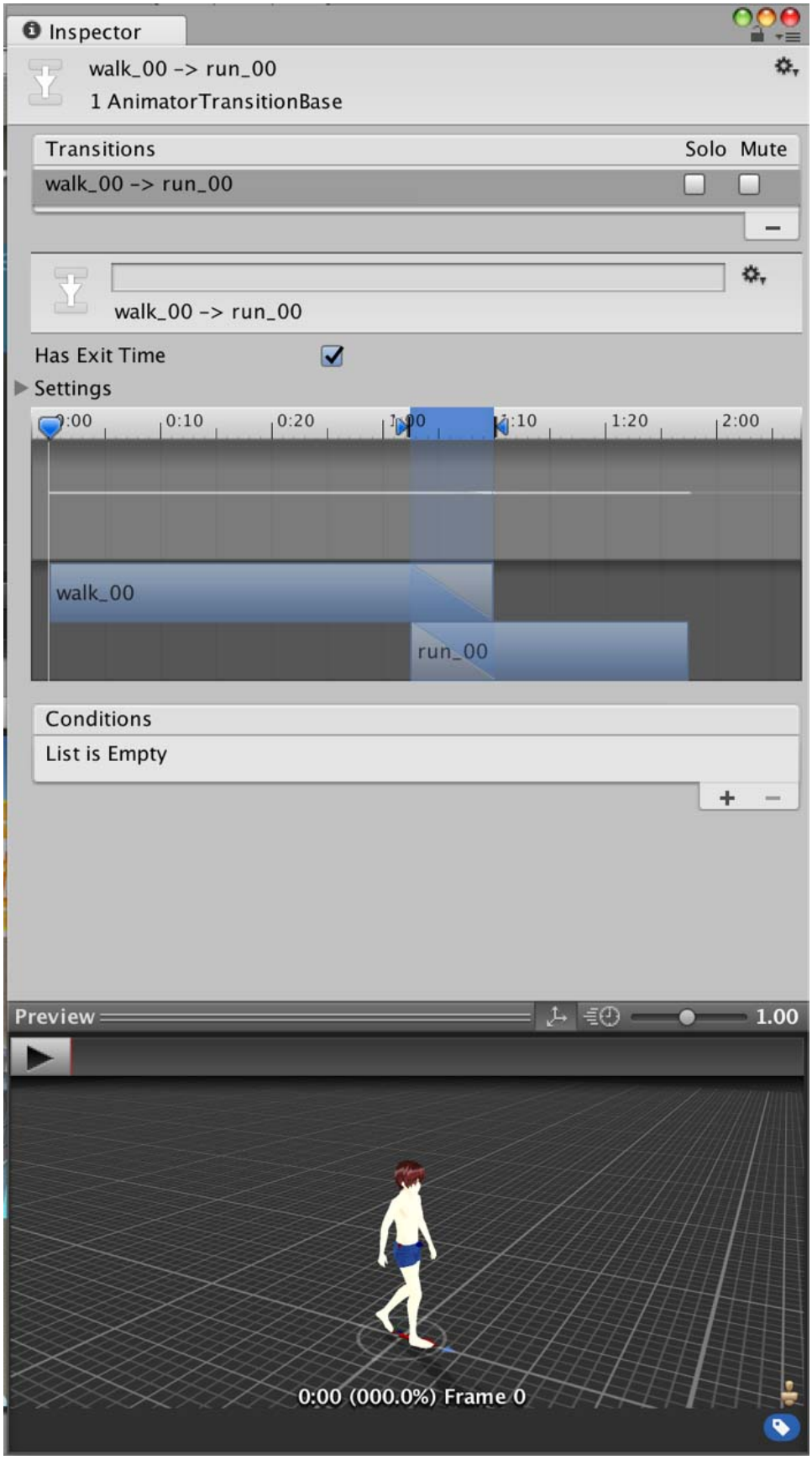
在Animator窗口中，可以选择四种类型的参数，针对这个角色的运动，选择浮点类型Float。现在我们需要把这个值和代码中的currentSpeed变量联系起来。在Update函数的开始进行处理：

```
public Animator characterAnimator;
public int Health;
public int Stamina;
public float movementSpeed;
public float rotationSpeed;
public float maxSpeed;
public float jumpHeight;
public float jumpSpeed;

private float currentSpeed;
private bool Dead;

void Start () {

}

void Update () {

    // Sets the movement speed of the animator, to change from
   idle to walk and walk to run
    characterAnimator.SetFloat("currentSpeed",currentSpeed);

    // USING XBOX CONTROLLER
    transform.Rotate(0,Time.deltaTime * (rotationSpeed *
    Input.GetAxis ("xboxlefth")), 0);

    if(Input.GetAxis ("xboxleft") > 0){
        transform.position += transform.forward * Time.deltaTime *
        currentSpeed;
        currentSpeed = Time.deltaTime * (Input.GetAxis
        ("xboxleft") * movementSpeed);
    }

    else{
        transform.position += transform.forward * Time.deltaTime *
        currentSpeed;
        currentSpeed = Time.deltaTime * (Input.GetAxis
        ("xboxleft") * movementSpeed/3);
    }

    if(Input.GetKeyDown("joystick button 18") && Dead == false)
    {

    }

    if(Input.GetKeyUp("joystick button 18") && Dead == false)
    {

    }

    if(Input.GetKeyDown("joystick button 16") && Dead == false)
    {

    }
```

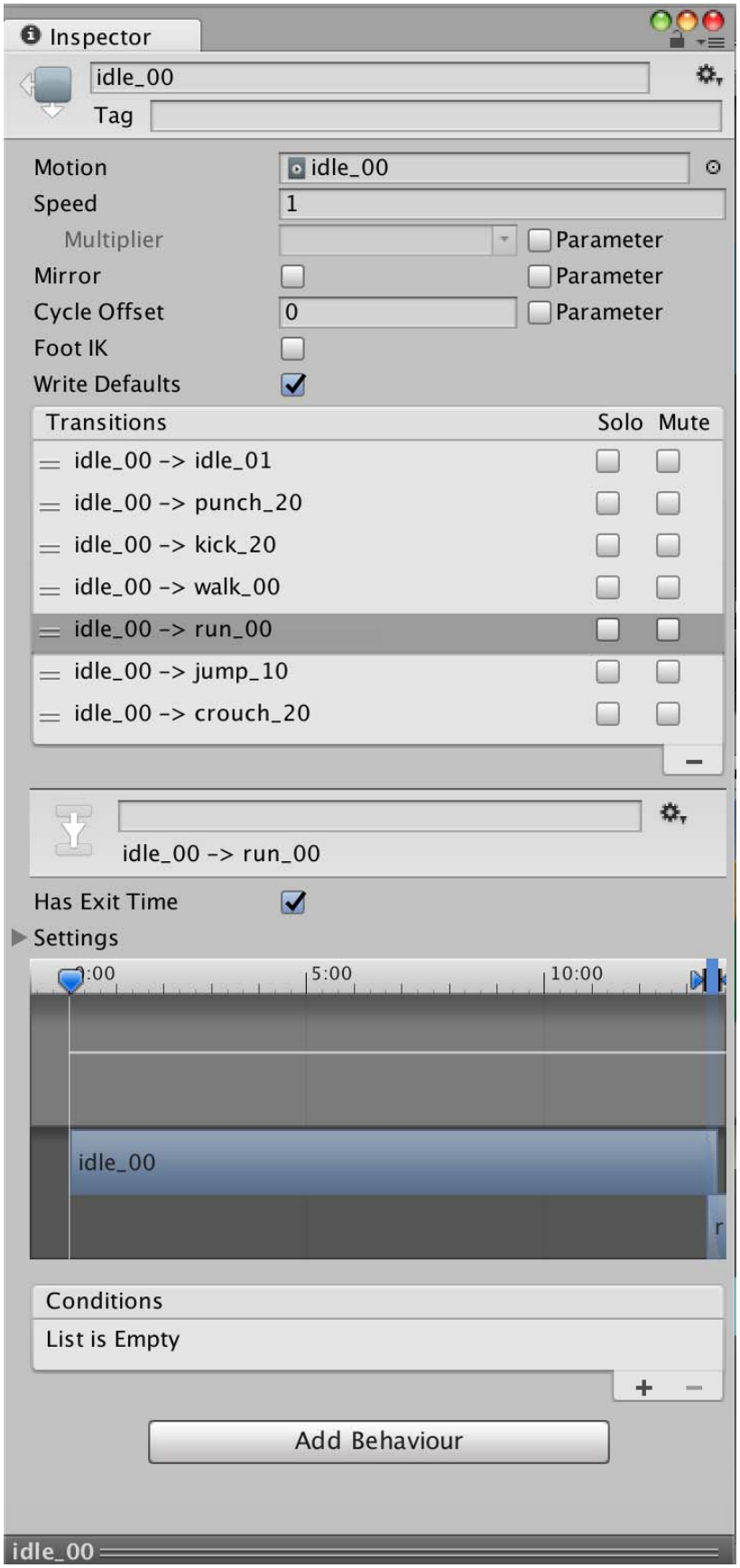
```
        if(Input.GetKeyUp("joystick button 16") && Dead == false)
        {

        }

        if(Health <= 0){
            Dead = true;
        }
    }
```

我们已经连接了两个参数。这样，动画状态机可以使用与currentSpeed相同的值。在动画状态机中用了完全相同的名称，这不是必需的，但是这样可以更容易地看出它代表哪个变量。

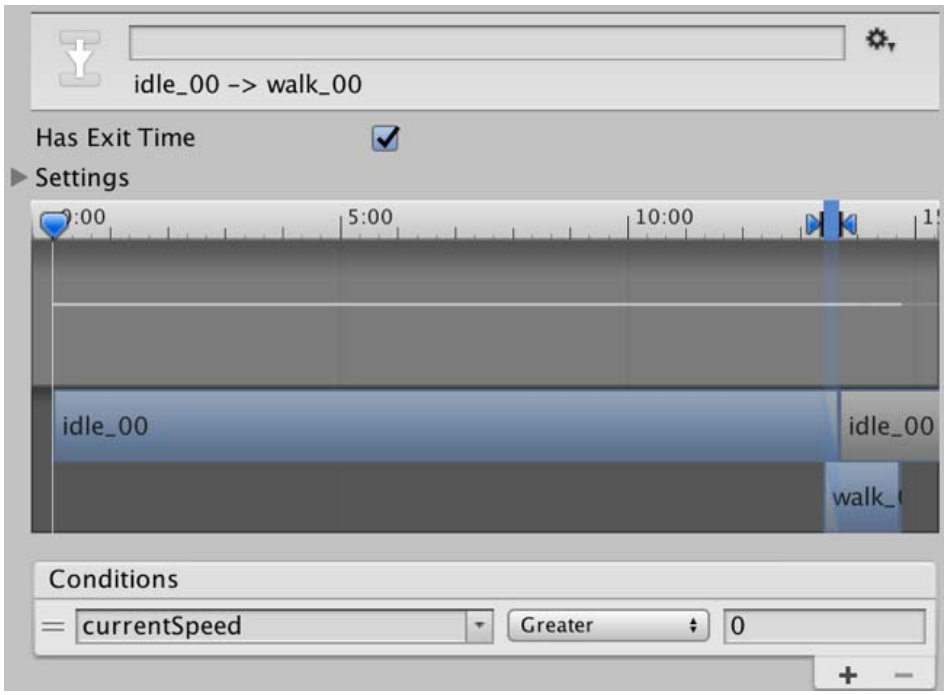
到这里，可以开始定义连线的值了。这个例子中，点击线条会弹出一个新的窗口，因此把会引起动画状态改变的值填写到窗口中。


Inspector
walk_00 -> run_00
1 AnimatorTransitionBase
Transitions
Solo
Mute
walk_00 -> run_00
walk_00 -> run_00
Has Exit Time
Settings
0:10
0:20
1:20
2:00
walk_00
run_00
Conditions
List is Empty
Preview
1.00
0:00 (000.0%) Frame 0

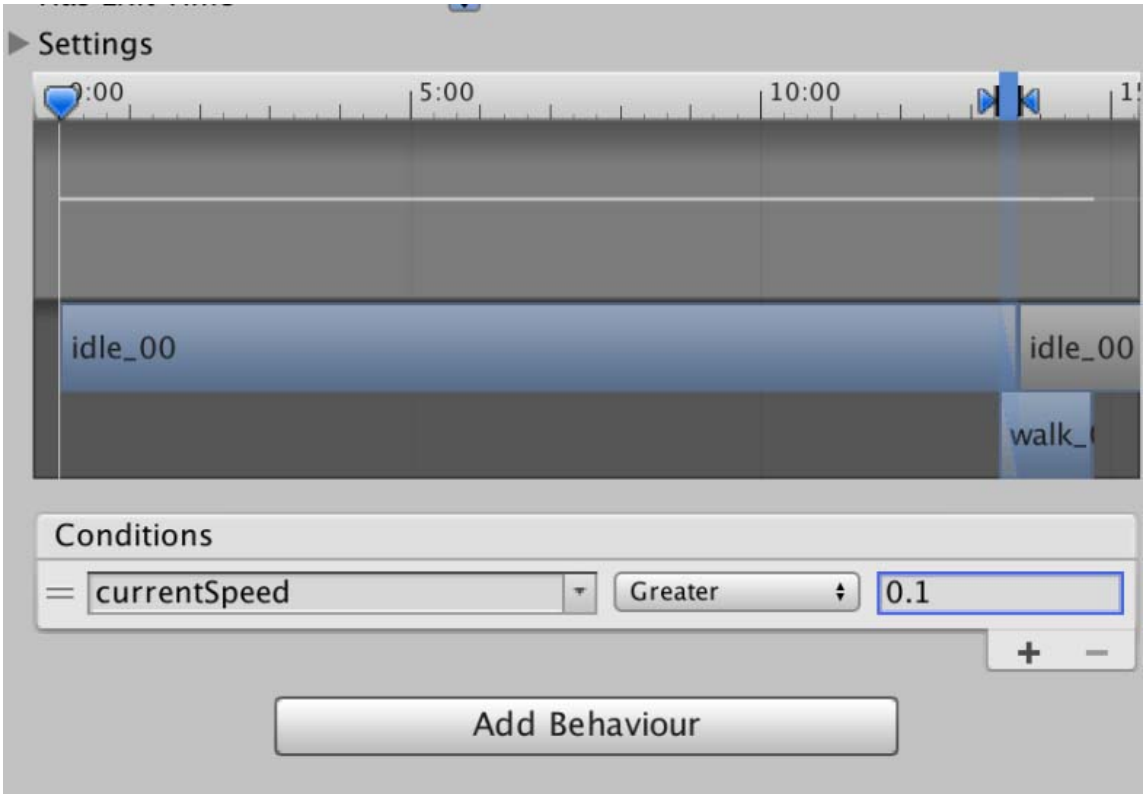
也可以点击想要配置的动画，比如IDLE，然后会弹出新的窗口，里面会有所有相关联的状态。可以选择其中想要修改的一条。如下图所示：


Inspector
idle_00
Tag
Motion
idle_00
Speed
1
Multiplier
Parameter
Mirror
Parameter
Cycle Offset
0
Parameter
Foot IK
Write Defaults
Transitions
Solo
Mute
idle_00 -> idle_01
idle_00 -> punch_20
idle_00 -> kick_20
idle_00 -> walk_00
idle_00 -> run_00
idle_00 -> jump_10
idle_00 -> crouch_20
idle_00 -> run_00
Has Exit Time
Settings
5:00
10:00
idle_00
Conditions
List is Empty
Add Behaviour
idle_00

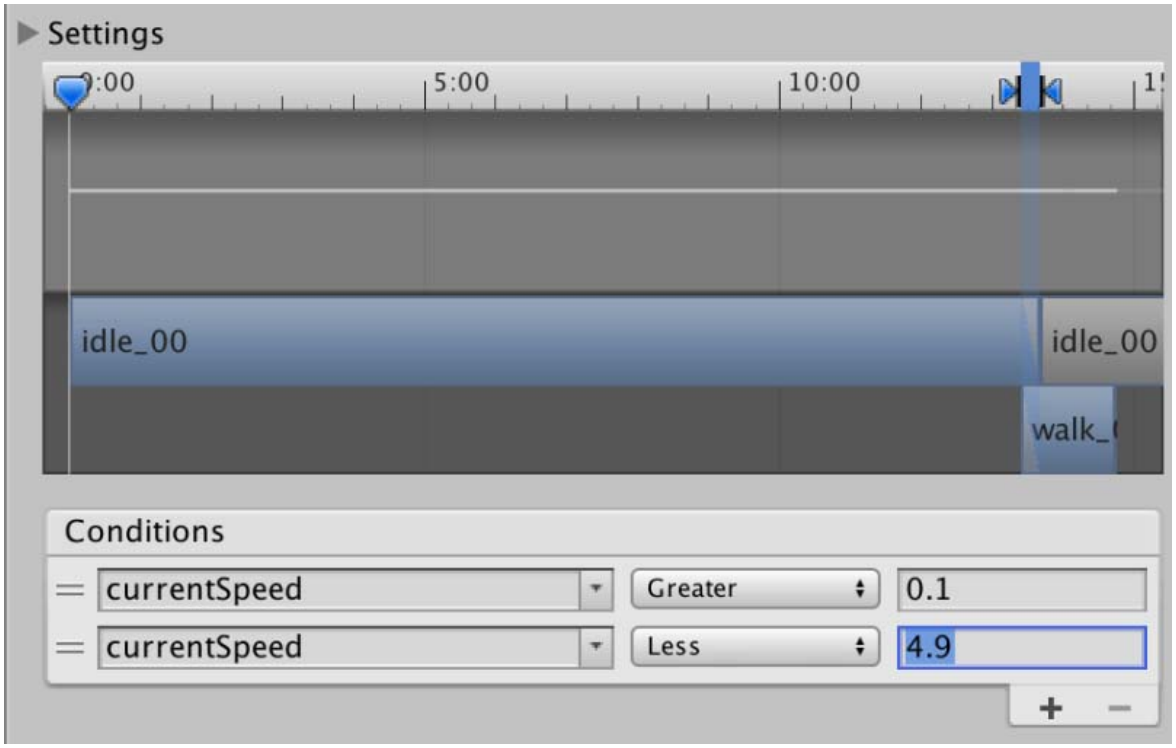
这里我们点击了IDLE到WALK（走路）的线并添加了之前创建的参数作为条件currentSpeed：



这里我们可以选择的参数应该大于或小于想要的值，以此来控制动画的播放。对这个例子，设置值应当大于0.1，这样当角色开始移动时，会停止IDLE动画并播放WALK动画，我们不需要写任何代码，动画状态机和代码是相互独立的：

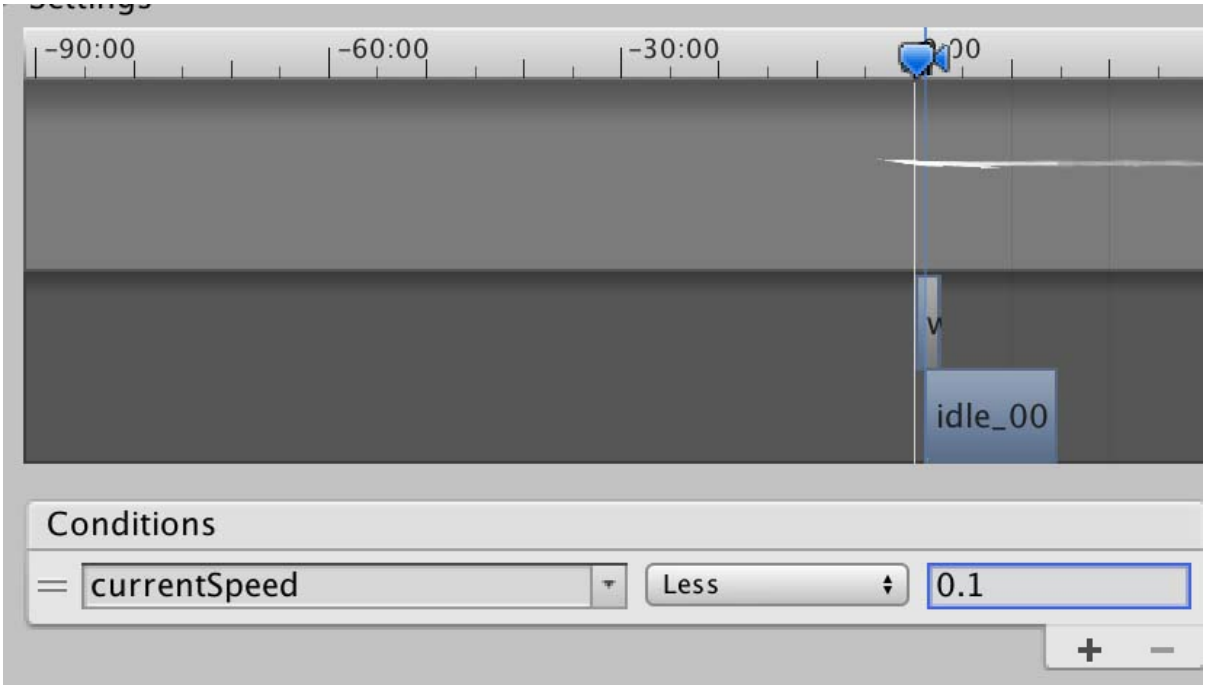


但是，由于有另一个动画在WALK动画之后，我们需要设置WALK的上限。这里假设角色在currentSpeed为5以上时跑步，那么角色应当在4.9时停止走路。所以我们添加另一个条件来告诉角色它在currentSpeed达到4.9时停止走路。



现在我们定义了何时角色开始走路，相反地，还需要定义何时停止走路并播放IDLE动画。要知道这样的设置不会对游戏逻辑有影响，这意味着如果我们这时开始进行游戏，角色会开始播放WALK动画，因为已经设置了。但是就算没有设置，角色也会移动，即使没有动画。我们简单地使用代码中的值来连接动画状态，需要定义动画在某个特定值的时候播放。如果我们忘了设置所有动画的播放条件，角色依然能够正常产生行为，但是没有正确的动画效果。所以，还是要检查所有的动画状态的连线是否都被正确地赋值了。

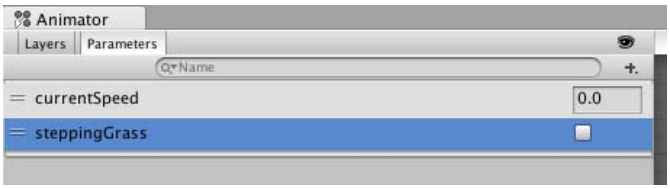
现在令角色在停止移动后回到IDLE动画，点击从WALK到IDLE的连线，添加新的条件用来表示如果currentSpeed小于0.1，就停止WALK动画播放IDLE动画。



现在可以在其余的动画中使用currentSpeed值了，以完成所有运动动画。这些都准备好之后，我们看下蹲组的动画。它们也使用currentSpeed的值，但是需要定义附加的条件来禁止在WALK动画状态中启用CROUCH动画。有两种下蹲移动的方式：按下下蹲按钮同时前进，或者定义地图上需要角色下蹲的区域，直接进入下蹲状态。本例中，由于我们是在制作AI角色，所以需要使用第二种方法并在地图上定义下蹲区域，以此让角色进入下蹲模式。

本例中我们假设角色不能在草地上行走，所以他会试着蹲下来前进。也可以选择一块它无法直立前进的小区域，来让角色自动蹲下移动。

为实现这一点需要在地图上创建触发区域，根据角色当前位置修改动画状态。在代码中创建一个新的布尔变量叫做steppingGrass，就放在用来联系动画的currentSpeed变量后面。接下来在动画状态机里面也添加一个参数，把新的布尔变量和动画关联起来。我们从创建新的变量开始：

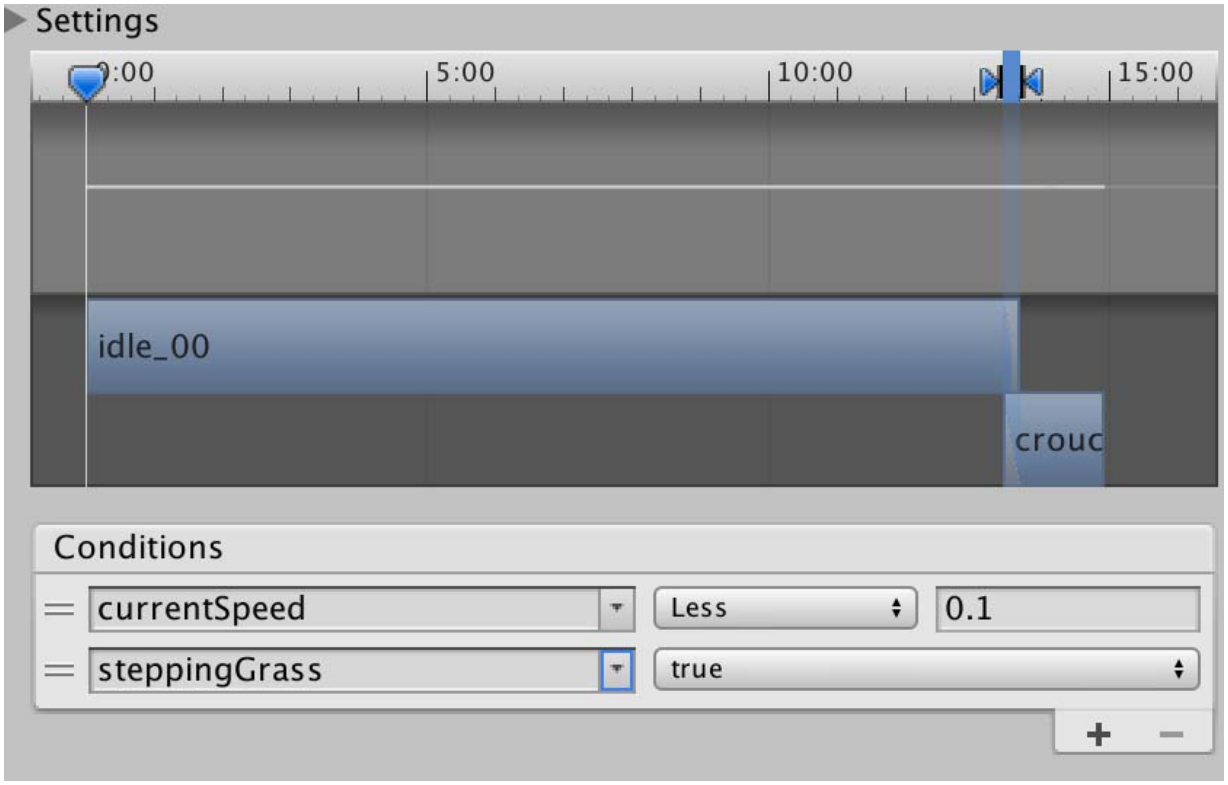


在代码中，我们添加碰撞检测来让这个值在角色进入草地区域时开启，离开那个区域时关闭：

```
public Animator characterAnimator;
public int Health;
public int Stamina;
public float movementSpeed;
public float rotationSpeed;
public float maxSpeed;
public float jumpHeight;
public float jumpSpeed;

private float currentSpeed;
private bool Dead;
private bool steppingGrass;

void Start () {

}

void Update () {

    // Sets the movement speed of the animator, to change from
    idle to walk and walk to run
    characterAnimator.SetFloat("currentSpeed",currentSpeed);
    // Sets the stepping grass Boolean to the animator value
    characterAnimator.SetBool("steppingGrass",steppingGrass);

    // USING XBOX CONTROLLER
    transform.Rotate(0,Time.deltaTime * (rotationSpeed *
    Input.GetAxis ("xboxlefth")), 0);

    if(Input.GetAxis ("xboxleft") > 0){
        transform.position += transform.forward * Time.deltaTime *
    currentSpeed;
        currentSpeed = Time.deltaTime * (Input.GetAxis
    ("xboxleft") * movementSpeed);
    }

    else{
        transform.position += transform.forward * Time.deltaTime *
        currentSpeed;
        currentSpeed = Time.deltaTime * (Input.GetAxis
        ("xboxleft") * movementSpeed/3);
    }

    if(Input.GetKeyDown("joystick button 18") && Dead == false)
    {

    }
```

```
        if(Input.GetKeyUp("joystick button 18") && Dead == false)
        {

        }

        if(Input.GetKeyDown("joystick button 16") && Dead == false)
        {

        }

        if(Input.GetKeyUp("joystick button 16") && Dead == false)
        {

        }

        if(Health <= 0){
            Dead = true;
        }
    }

    void OnTriggerEnter(Collider other) {
        if(other.gameObject.tag == "Grass")
        {
            steppingGrass = true;
        }
    }

    void OnTriggerExit(Collider other) {

        if(other.gameObject.tag == "Grass")
        {
            steppingGrass = false;
        }
    }
```

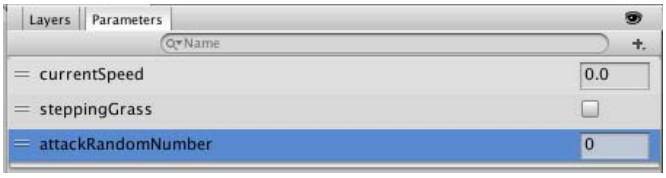
现在，我们可以继续添加新的参数到CROUCH动画中。从选择IDLE和CROUCH之间的连线开始，设置currentSpeed的值和steppingGrass的值。由于我们有Crouch Idle动画，尽管角色没有移动，还是会播放该动画而不是普通的IDLE动画：



我们设置currentSpeed小于0.1，也就是角色不移动的时候，steppingGrass设置为true，这会停止普通的IDLE动画开始播放Crouch Idle动画。其余的CROUCH动画和这个的原理一致。当角色开始移动，currentSpeed就会变化，Crouch Idle动画停止Crouch Walk开始播放。最后，连接Crouch Idle到IDLE，Crouch Walk到WALK，确保角色离开草地时，WALK动画不会停止且角色保持直立走动。

关于攻击，我们会使用1到10的随机整数，如果数字大于5，则播放KICK动画；如果数字小于5，则播放PUNCH动画。所以当角色战斗时，会播放不同的动画。未来可以使用同样的方式添加更多动画，增加攻击动作的多样性。

再一次地，创建一个新的参数，在这里我们要创建一个新的整型参数attackRandom-Number：



在代码中，我们添加新的变量并起同样的名字（不是必须使用相同的名字，但是这样显得更有组织性）。在之前连接变量与动画参数的地方，添加新的变量来关联attackRandomNumber的值。之后我们创建一个函数在角色战斗时获得随机数：

```
public Animator characterAnimator;
public int Health;
public int Stamina;
public float movementSpeed;
public float rotationSpeed;
public float maxSpeed;
public float jumpHeight;
public float jumpSpeed;

private float currentSpeed;
private bool Dead;
private bool steppingGrass;
private int attackRandomNumber;

void Start () {

}

void Update () {

    // Sets the movement speed of the animator, to change from
    idle to walk and walk to run
    characterAnimator.SetFloat("currentSpeed",currentSpeed);
// Sets the stepping grass Boolean to the animator value
characterAnimator.SetBool("steppingGrass",steppingGrass);

// Sets the attackrandomnumber to the animator value
characterAnimator.SetInteger("attackRandomNumber",
```

```
   attackRandomNumber);

// USING XBOX CONTROLLER
transform.Rotate(0,Time.deltaTime * (rotationSpeed *
Input.GetAxis ("xboxlefth")), 0);

if(Input.GetAxis ("xboxleft") > 0){
    transform.position += transform.forward * Time.deltaTime *
currentSpeed;
    currentSpeed = Time.deltaTime * (Input.GetAxis
("xboxleft") * movementSpeed);
}

else{
    transform.position += transform.forward * Time.deltaTime *
currentSpeed;
    currentSpeed = Time.deltaTime * (Input.GetAxis
("xboxleft") * movementSpeed/3);
}

if(Input.GetKeyDown("joystick button 18") && Dead == false)
{
    fightMode();
}

if(Input.GetKeyUp("joystick button 18") && Dead == false)
{

}

if(Input.GetKeyDown("joystick button 16") && Dead == false)
{

}

if(Input.GetKeyUp("joystick button 16") && Dead == false)
{

}

if(Health <= 0){
    Dead = true;
}
}

void OnTriggerEnter(Collider other) {

    if(other.gameObject.tag == "Grass")
    {
        steppingGrass = true;
    }
}

void OnTriggerExit(Collider other) {

    if(other.gameObject.tag == "Grass")
    {
```

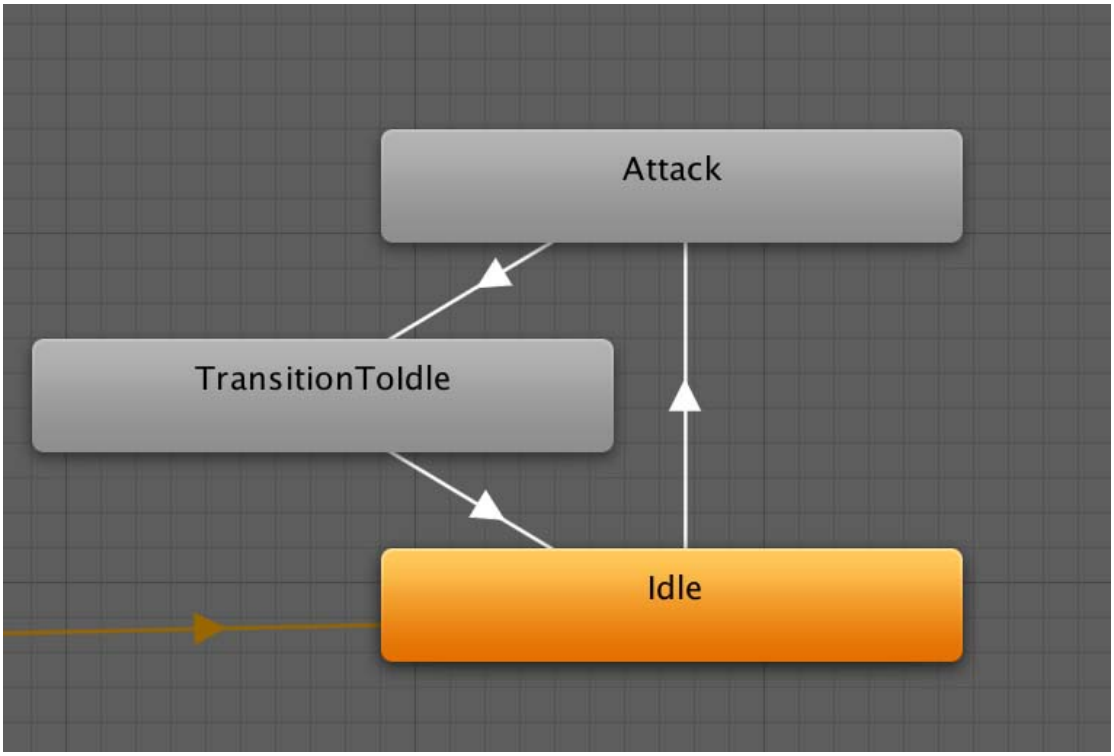
```
            steppingGrass = false;
        }
    }

    void fightMode ()
    {
        attackRandomNumber = (Random.Range(1, 10));
    }
```

做完这块之后，我们需要在动画里给参数赋值。操作过程和之前的动画一样，只是值不同而已。如果attackRandomNumber大于1，代表它在攻击且攻击动画开始播放。我们有两个不同的攻击动作且打算随机播放，如果是玩家操作，就需要在代码中直接指定变量的值了，当玩家按下特定按钮，角色会出拳或者踢腿。

5.3 平滑过渡

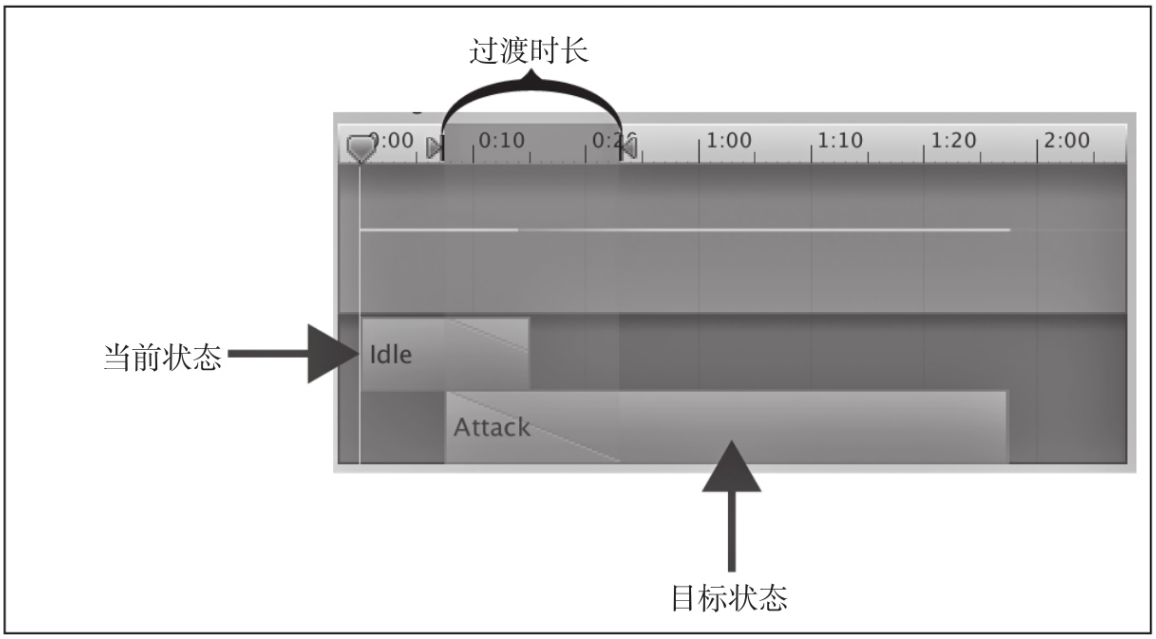
另一个重要的点是动画间的平滑过渡。这对保持动画的完整性很重要，以便让角色的每个动作看起来都很流畅，增强玩家的沉浸感。

在这一点中，2D和3D动画考虑的方式不同。如果我们使用2D精灵，就需要为每一种动作转换画出必要的帧，每次角色切换动画时，切换的相应动画就会被播放。



另一方面，对3D角色我们可以使用骨骼动画自动创建过渡，每个骨骼的坐标都会从前一个位置移动到新的位置。就算我们使用骨骼动画来辅助创建过渡，某些时候也必须手动创建一些过渡动画，而且有时为了效果更好也需要这么做。比如一个很常见的情况，角色在使用物品或武器时，在进行下一个动作前要维持当前的动画状态。

要创建平滑的动画，我们需要让后一个动画的第一帧和前一个动画的最后一帧相同。需要从相同的位置开始播放下一个动画。这在避免出现跳跃性的动画过渡方面是关键性的方法。我们可以使用游戏引擎中动画的转换系统来帮助我们创建平滑的动画过渡，我们可以通过调整过渡时间，可以设置为快速过渡或是慢一些，通常需要根据视觉效果来进行实验。



有时我们可能需要牺牲平滑的动画切换来换取更好的游戏体验。举个例子在格斗游戏中，快速切换就比流畅切换要重要得多，因此需要根据实际情况考虑动画切换所要花费的时间。

5.4 总结

本章描述了怎样使用2D或3D动画来提升角色的动作。动画在开发一个真实可信的AI角色方面具有重要地位，通过正确地使用动画，会让玩家感受到角色就像活了一样，而且还会自主地做出反应。

下一章里，会探讨导航行为和寻路，也就是如何给AI角色编程，来让它自动走到需要的位置并选择最优路线。

第6章　导航行为和寻路

在本章中，我们将详细解释AI角色应当如何移动，以及如何让AI知道它可以去哪里、不能去哪里。这个问题对于不同类型的游戏有不同的解决方案，我们将在本章中讨论这些解决方案，探索可以用来开发能够在地图上正确移动的角色的常用方法。同时，我们希望角色能够计算出到达指定地点的最佳路线，避开障碍并完成任务。我们将介绍如何创建一个简单的导航行为，然后会介绍点到点的移动，最后深入探讨如何创建一个复杂的点到点移动系统（RTS/RPG类游戏中的移动系统）。

6.1 导航行为

当我们谈论导航行为时，实际指的是一个角色面对特定情况时的行为，在这种情况下，它们需要计算去哪里或去做什么。一张地图可以有许多个特殊的地点，在这些地点可能会需要跳跃或爬楼梯才能最终到达目的地。角色应该知道如何运用这些动作来保持正确移动，否则它可能会因为没有跳过地洞而跌到洞里，或者走到墙边时因为没有爬楼梯而一直卡在墙下。为了避免这种情况，我们需要为它计划好所有可能性，确保它能通过跳跃或其他动作来保持正确地移动。

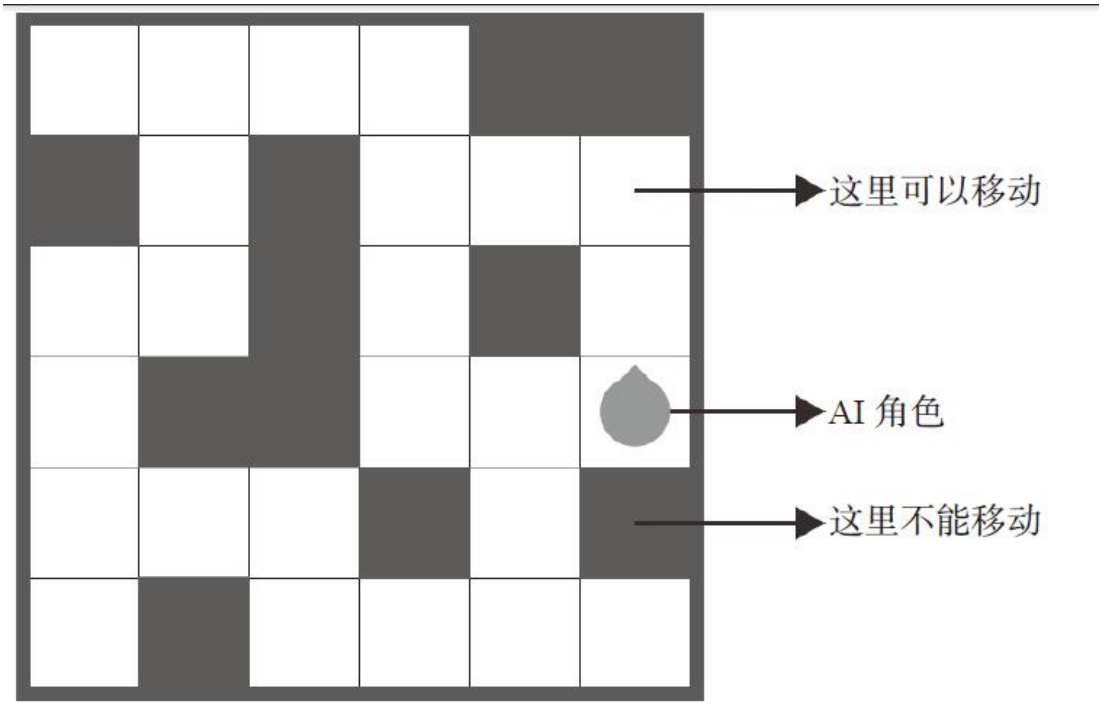
6.1.1 选择新的方向

当AI角色遇到一个挡住它的路的障碍时，它可以选择新的方向以绕过障碍，这对AI角色来说是非常重要的一个方面。角色应该意识到前方有物体挡住了去路，当它不能继续朝那个方向前进时，它需要选择一个新的方向以避免撞到障碍物。

避免撞墙

如果我们的角色面对一堵墙，它需要知道自己不能穿过那堵墙，并且应该选择其他可能的路线。除非允许角色爬墙或摧毁墙，否则角色应当面朝一个没有被阻挡的新方向移动。

我们从一个简单且常见的方法开始，对某些游戏类型来说，这个方法也许是最佳选择。在下面演示的例子中，这个角色需要像《吃豆人》中的敌人那样在关卡中来回移动。从基本的例子开始，我们让AI角色可以自由选择一个方向移动，然后我们将给角色的AI提供更



多的信息，这样它将可以在地图上追逐一个特定的目标。

我们创建了一个网格，并且画了一些黑色方块来代表AI角色不能通过的区域。现在来编写我们的AI角色，让它一直向前移动直到其遇到一个黑色方块。然后，它需要随机决定选择右转或左转。这样就能让角色在地图上自由移动，没有任何特定的模式。代码如下：

```
public float Speed;
public float facingLeft;
public float facingRight;
public float facingBack;
public static bool availableLeft;
public static bool availableRight;

public bool aLeft;
public bool aRight;

void Start ()
{

}

void Update ()
{

    aLeft = availableLeft;
    aRight = availableRight;

    transform.Translate(Vector2.up * Time.deltaTime * Speed);
```

```
    if(facingLeft > 270)
    {
        facingLeft = 0;
    }

    if(facingRight < -270)
    {
        facingRight = 0;
    }

}

void OnTriggerEnter2D(Collider2D other)
{

    if(other.gameObject.tag == "BlackCube")
    {
        if(availableLeft == true && availableRight == false)
        {
            turnLeft();
        }

        if(availableRight == true && availableLeft == false)
        {
            turnRight();
        }

        if(availableRight == true && availableLeft == true)
        {
            turnRight();
        }

        if(availableRight == false && availableLeft == false)
        {
            turnBack();
        }
    }
}

void turnLeft ()
{
    facingLeft = transform.rotation.eulerAngles.z + 90;
    transform.localRotation = Quaternion.Euler(0, 0, facingLeft);
}

void turnRight ()
{
    facingRight = transform.rotation.eulerAngles.z - 90;
    transform.localRotation = Quaternion.Euler(0, 0, facingRight);
}

void turnBack ()
{
    facingBack = transform.rotation.eulerAngles.z + 180;
    transform.localRotation = Quaternion.Euler(0, 0, facingBack);
}
```

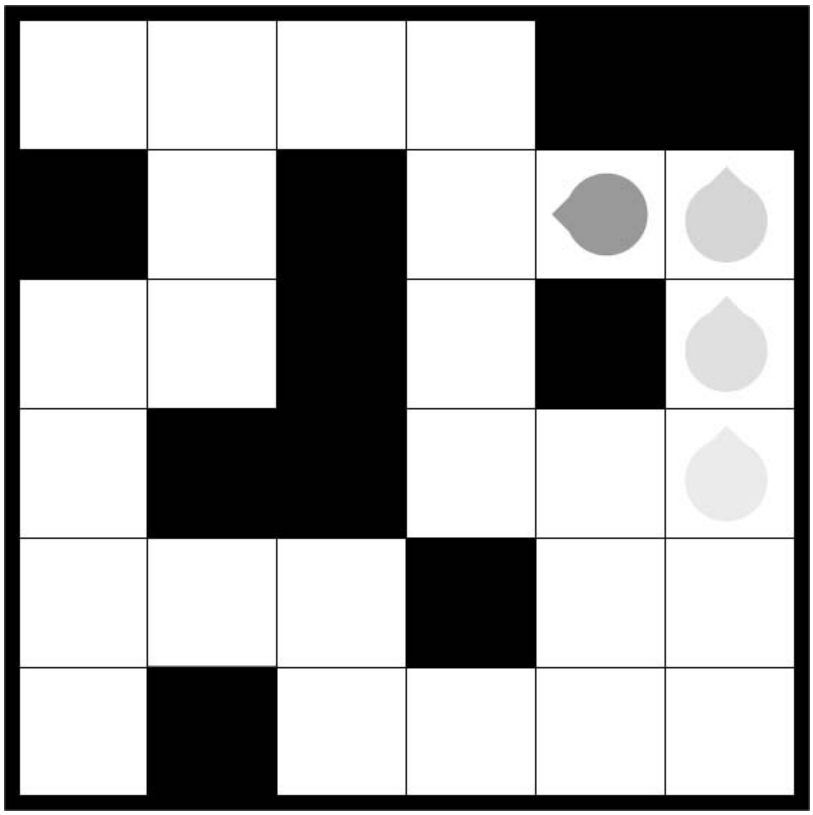
在这个例子中，我们在黑色方块上添加了碰撞器，能够让角色准确地知道是否碰到了障碍物。这样，它首先会持续移动，直到撞上一个黑色的方块，然后在当前相撞的位置上将有三个选项：左转，右转，或原路返回。为了知道哪个方向是畅通的，我们在AI角色身上创建了两个独立的碰撞器。每个碰撞器都有一个脚本，用于给角色提供信息，让它知道这两边的方向是否是畅通的。

availableLeft变量对应左边，availableRight变量对应右边。如果左边或右边的碰撞器与黑色正方形接触到，则该值会被设置为false，否则设置为true。我们使用aLeft和aRight这两个值存下两边碰撞检测的结果，用来随时检查两边的情况。

```
public bool leftSide;
public bool rightSide;

void Start ()
{
     if(leftSide == true)
     {
         rightSide = false;
     }

     if(rightSide == true)
     {
         leftSide = false;
     }
}
void Update () {
}

 void OnTriggerStay2D(Collider2D other)
 {

     if(other.gameObject.tag == "BlackCube")
     {
         if(leftSide == true && rightSide == false)
         {
             Character.availableLeft = false;
         }

         if(rightSide == true && leftSide == false)
         {
             Character.availableRight = false;
         }
     }
 }

 void OnTriggerExit2D(Collider2D other)
 {

     if(other.gameObject.tag == "BlackCube")
     {
         if(leftSide == true)
            {
                Character.availableLeft = true;
            }

            if(rightSide == true)
```

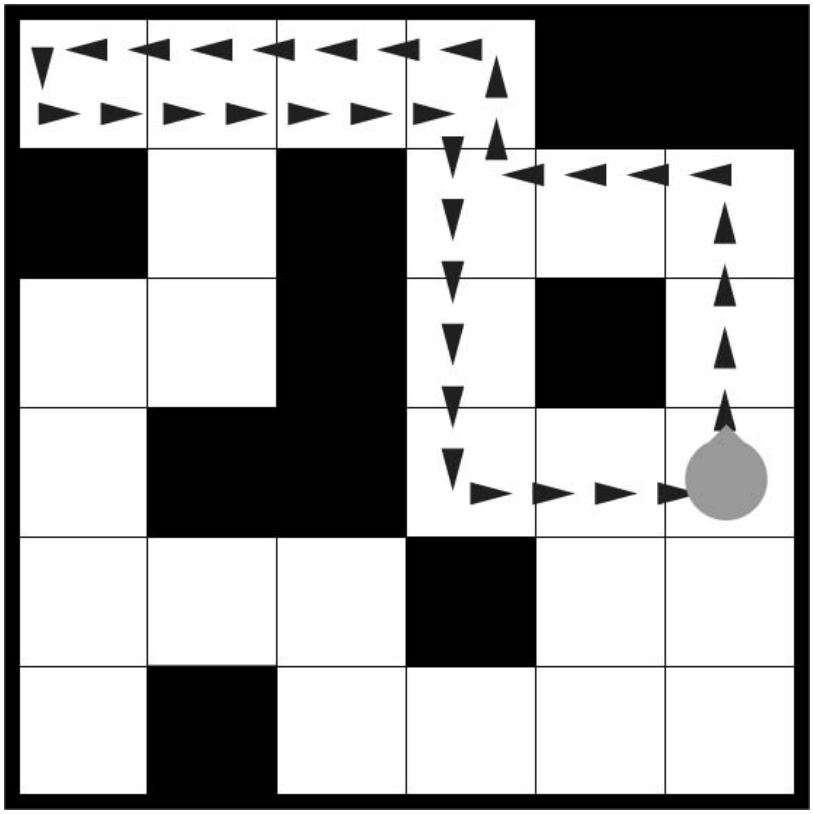
```
                {
                    Character.availableRight = true;
                }
            }
    }
```

当我们开始运行游戏，我们可以看到AI角色开始在地图上移动，每当遇到黑色方块时



就会左转或右转：

但是如果我们让游戏运行几分钟，我们就会发现这个角色一直在做同样的决策，它只会绕着地图的一小部分走。这是因为它只在与黑方格碰撞时做出决定，而忽略了其他可以转向的机会：



如上图所见，角色总是遵循相同的模式，这并不是一个理想的情况，我们希望他能一直选择不同的路径。

选择其他的路线

我们的角色在每次接近墙壁时都选择了一个新的方向，现在我们希望它能够在地图上更自由地移动。为了做到这一点我们将添加一些逻辑，让角色在有一个向左或向右转的机会时可以自由旋转，即使前方的道路是通畅的。我们可以用概率来决定角色是否会改变方向，在这个例子中我们指定了有90%的概率会选择一个新的方向。这样，我们很快就能看到结果发生了变化：

```
public float Speed;
public float facingLeft;
public float facingRight;
public float facingBack;
public static bool availableLeft;
public static bool availableRight;

public static int probabilityTurnLeft;
public static int probabilityTurnRight;public int probabilitySides;

public bool forwardBlocked;

public bool aLeft;
public bool aRight;
```

```
void Start ()
{
    availableLeft = false;
    availableRight = false;
    probabilityTurnLeft = 0;
    probabilityTurnRight = 0;
}
```

在添加了变量之后，我们可以转到在第一帧时会被调用的Start方法。

接着我们转到每帧被调用的Update方法。

```
void Update ()
{
    aLeft = availableLeft;
    aRight = availableRight;

    transform.Translate(Vector2.up * Time.deltaTime * Speed);

    if(facingLeft > 270)
    {
        facingLeft = 0;
    }

    if(facingRight < -270)
    {
        facingRight = 0;
    }

    if (forwardBlocked == false)
    {
        if (availableLeft == true && availableRight == false)
        {
           if (probabilityTurnLeft > 10)
           {
                turnLeft();
           }
        }

        if (availableLeft == false && availableRight == true)
        {
           if (probabilityTurnRight > 10)
```

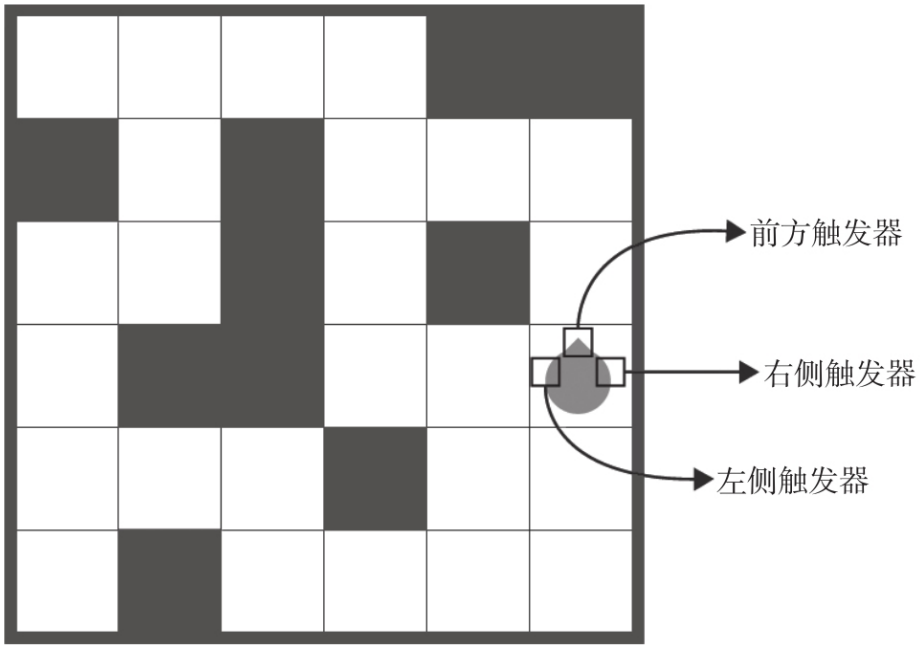
```
            {
                turnRight();
            }
        }

        if (availableLeft == true && availableRight == true)
        {
            probabilityTurnLeft = 0;
            probabilityTurnRight = 0;
        }
    }

}
```

这里我们添加触发器的函数，当有物体进入或离开自身的碰撞区域时会被调用：

```
void OnTriggerEnter2D(Collider2D other)
{

    if(other.gameObject.tag == "BlackCube")
    {
        forwardBlocked = true;

        if(availableLeft == true && availableRight == false)
        {
             turnLeft();
        }

        if(availableRight == true && availableLeft == false)
        {
            turnRight();
        }

        if(availableRight == true && availableLeft == true)
        {
            probabilitySides = Random.Range(0, 1);
            if(probabilitySides == 0)
            {
                turnLeft();
            }

            if(probabilitySides == 1)
            {
                turnRight();
            }

        }

        if(availableRight == false && availableLeft == false)
        {
            turnBack();
        }
    }
}

void OnTriggerExit2D(Collider2D other)
{
    forwardBlocked = false;
}
```

```
void  turnLeft ()
{
    probabilityTurnLeft = 0;
    facingLeft = transform.rotation.eulerAngles.z + 90;
    transform.localRotation = Quaternion.Euler(0, 0, facingLeft);
}

void turnRight ()
{
    probabilityTurnRight = 0;
    facingRight = transform.rotation.eulerAngles.z - 90;
    transform.localRotation = Quaternion.Euler(0, 0, facingRight);
}

void turnBack ()
{
    facingBack = transform.rotation.eulerAngles.z + 180;
    transform.localRotation = Quaternion.Euler(0, 0, facingBack);
}
```

我们已经在AI角色的脚本中增加了四个新的变量，probabilityTurnLeft静态变量表示角色左转弯的概率；probabilityTurnRight表示角色右转弯的概率；probabilitySides决定了当两者都可用且前方的路线被阻挡时，角色会怎么选择转向；最后，是一个布尔值，forwardBlocked，用来检查前方路线是否被阻挡。角色需要检查前方路线是否被阻挡以知道它是否可以转弯。这样可以防止角色在面对黑色方块时转向多次。



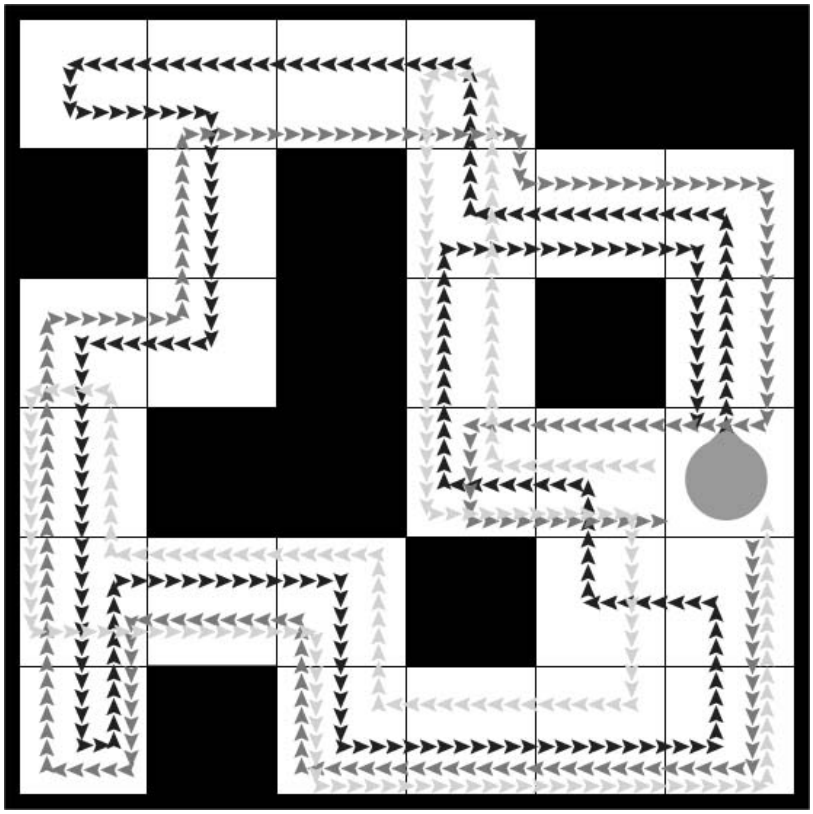
在控制侧面触发器的脚本中，我们添加一个新的变量：probabilityTurn，这给角色提供了一些概率信息。每当有物体退出碰撞盒时，它就计算概率并发送消息给角色，告诉他可以转向这一边：

```
public bool leftSide;
public bool rightSide;
public int probabilityTurn;

void Start ()
```

```
{
    if(leftSide == true)
   {
       rightSide = false;
    }

    if(rightSide == true)
    {
       leftSide = false;
    }
 }

 void Update ()
 {

 }

 void OnTriggerEnter2D(Collider2D other)
 {
    if(other.gameObject.tag == "BlackCube")
    {
        if(leftSide == true && rightSide == false)
        {
            Character.availableLeft = false;
            probabilityTurn = 0;
            Character.probabilityTurnLeft = probabilityTurn;
        }

        if(rightSide == true && leftSide == false)
        {
            Character.availableRight = false;
            probabilityTurn = 0;
            Character.probabilityTurnRight = probabilityTurn;
        }
     }
  }
void OnTriggerStay2D(Collider2D other)
{

      if(other.gameObject.tag == "BlackCube")
      {
          if(leftSide == true && rightSide == false)
          {
              Character.availableLeft = false;
              probabilityTurn = 0;
              Character.probabilityTurnLeft = probabilityTurn;
          }

          if(rightSide == true && leftSide == false)
          {
              Character.availableRight = false;
              probabilityTurn = 0;
              Character.probabilityTurnRight = probabilityTurn;
          }
      }
  }

  void OnTriggerExit2D(Collider2D other)
```

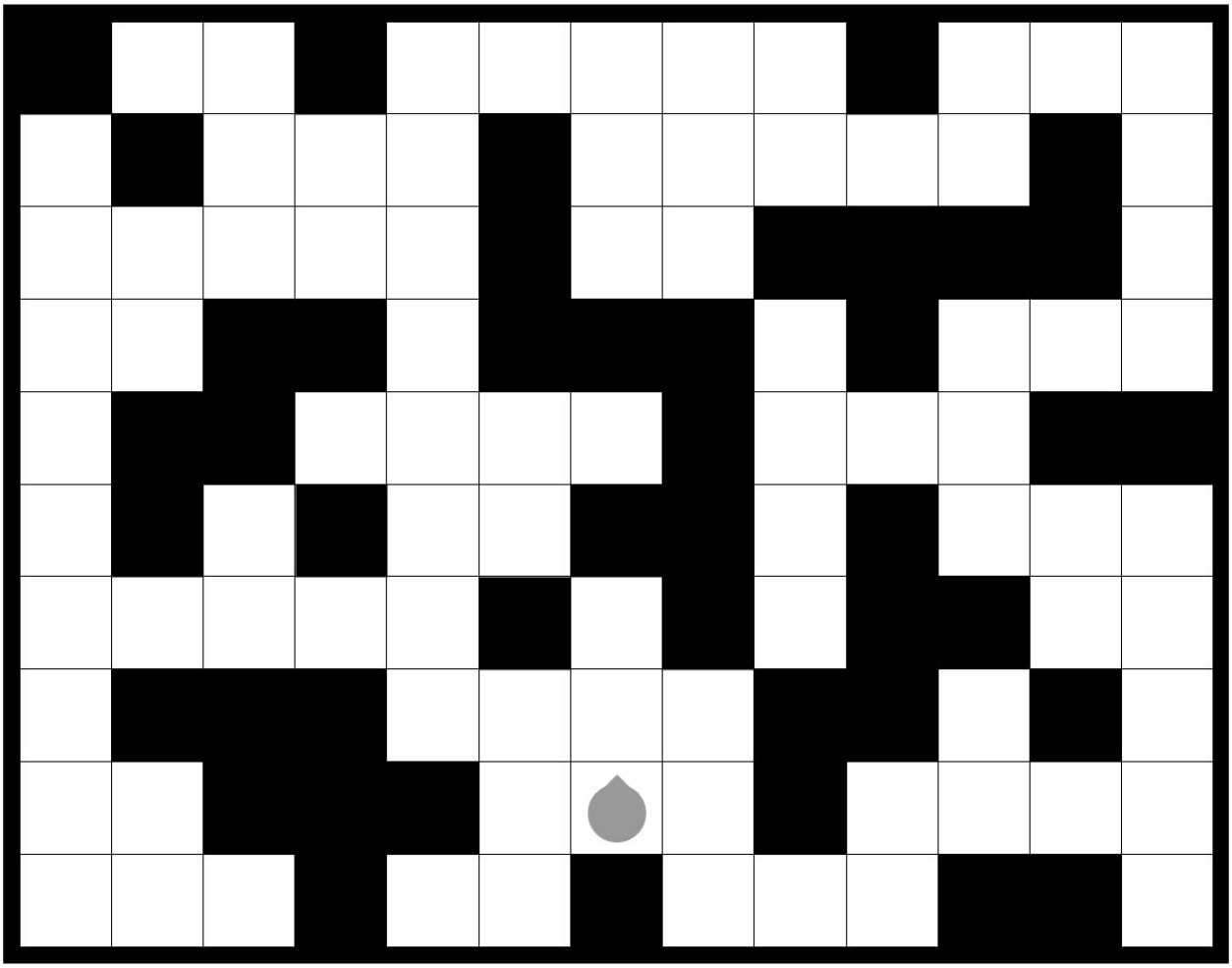
```
    {

        if(other.gameObject.tag == "BlackCube")
        {
            if(leftSide == true)
            {
                probabilityTurn = Random.Range(0, 100);
                Character.probabilityTurnLeft = probabilityTurn;
                Character.availableLeft = true;
            }

            if(rightSide == true)
            {
                probabilityTurn = Random.Range(0, 100);
                Character.probabilityTurnRight = probabilityTurn;
                Character.availableRight = true;
            }
        }
     }
```

如果我们这时运行游戏，我们可以看到角色有了新的变化。现在AI角色的移动路线更加不可预测，它每次都会选择不同的路线，在地图上四处移动，与之前的行为有着显著的不同。完成了这部分之后，我们就可以创建各种各样的地图了，因为角色总能找到移动的路径并且避免与墙壁相撞。



在更大的地图上进行测试，角色会以同样的方式在整个地图上移动。这意味着我们的主要目标已经完成了，现在我们可以很容易地创建新的地图，使用AI角色作为游戏的主要敌人，而且这些敌人将总是采取不同的方式移动，也不遵循任何模式。

我们可以根据想要的反应方式来调整百分比值，也可以加入更多的变量来让游戏变得更加独特。

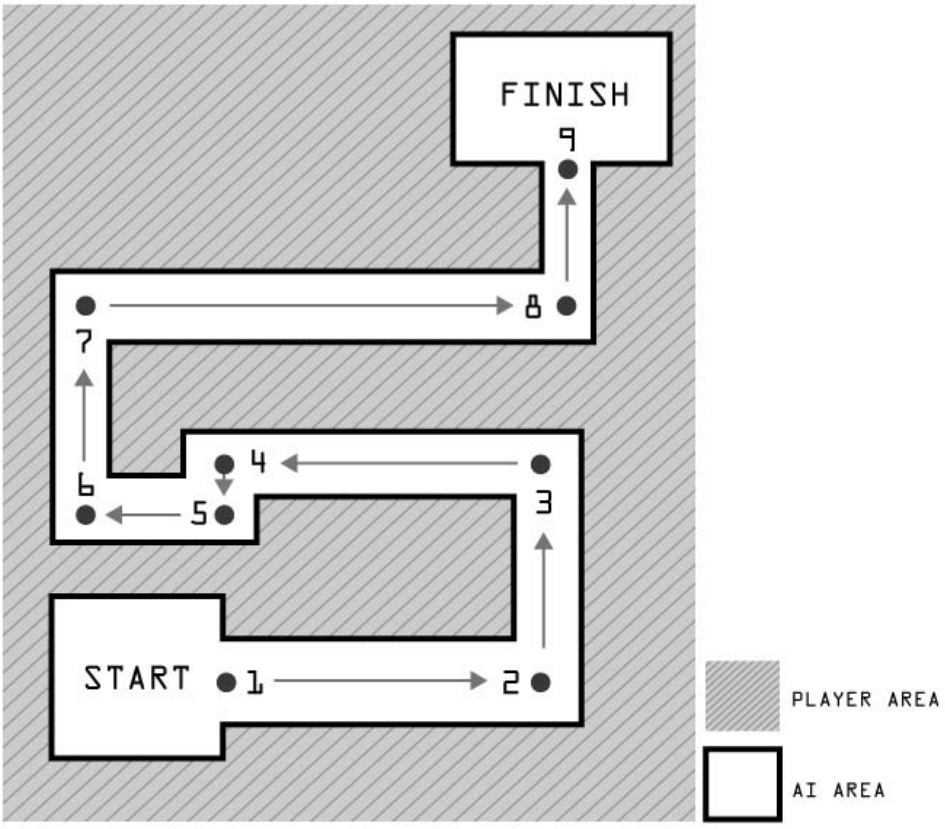


6.1.2 点到点的移动

现在我们了解了如何创建一个可以在迷宫类游戏中自由移动的角色的基本知识，接下来我们将看点不一样的东西：如何创建点到点的移动模式。它在AI运动问题中也是一个重要的方面，后面我们会综合运用这两种技术，来创建一个能够点到点移动同时避开墙壁与障碍物的AI。

塔防类型

这里我们将介绍让角色点到点移动的方法，和之前的方法一样，这种方法也能同时适用于2D和3D游戏。在这个例子中，我们将探索如何创建一个塔防游戏的主要特性：敌人的行为模式。目标是让敌人在一个起始点生出来，并沿着一条路径到达终点。塔防游戏中的



敌人通常都会像这样移动，所以这是用来演示如何创建点对点移动的完美的例子。

塔防游戏通常由两个区域组成：敌人从起始位置到最后位置的行进路线区域，以及玩家可以建造攻击敌人的塔的区域，玩家通过建造这些攻击塔来阻止敌人到达最后的位置。由于玩家不能在敌人经过的路径内建造任何东西，所以AI不需要知道它们周围的环境，它

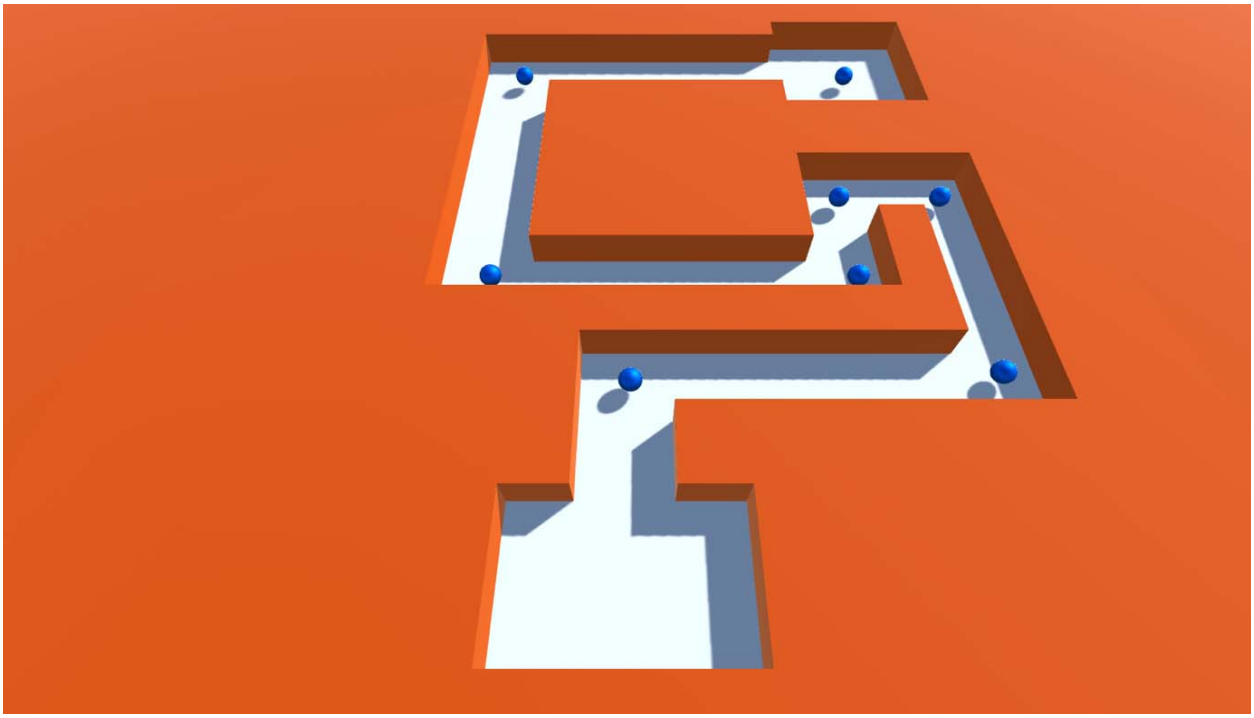
们总是处在可以自由通过的路线内。正因如此，我们只需要把重点放在角色的点到点移动上。

在导入了游戏中使用的地图和人物后，我们需要配置寻路点（waypoint），以便让角色知道自己要去往哪里。我们可以手动在我们的代码中添加这些坐标，但为了简化这一过程我们将在场景中创建物体作为我们的寻路点。创建后要删除物体的三维网格组件，因为它们不需要显示出来。

现在将这些在场景中创建的所有寻路点分到一组里，并将该组命名为waypoints。放置好这些寻路点并分组后，我们就可以开始创建代码了，代码将控制我们的角色去跟随这些寻路点。此代码是非常有用的，因为我们可以创建不同的地图、使用任意数量的寻路点，而不需要修改角色代码：

```
public static Transform[] points;

void Awake ()
{
     points = new Transform[transform.childCount];
     for (int i = 0; i < points.Length; i++)
     {
          points[i] = transform.GetChild(i);
      }
}
```

此代码将我们已经创建的寻路点分配到一个数组里，并对这些寻路点计数和排序。



我们可以从上图中看到一些蓝色的球，它们就是用来表示寻路点的。对于这个例子来说，角色需要跟随这8个寻路点以抵达终点。现在让我们转到AI角色的代码，看看如何使用这些寻路点来令角色从一个点移动到另一个点。

我们首先创建角色的基本属性，即生命和速度。然后，我们可以创建一个新的变量，它将告诉角色需要移动到的下一个位置。另一个变量则用来表示当前需要跟随哪一个寻路

```
public float speed;
public int health;

private Transform target;
private int wavepointIndex = 0;
```

点：

现在我们有了一个基本的变量来让敌人角色从一个点移动到另一个点，直到它死亡或者到达终点。让我们看看如何使用这些变量：

```
public float speed;
public int health;

private Transform target;
private int wavepointIndex = 0;


void Start ()
{
     target = waypoints.points[0];  speed = 10f;
}

void Update ()
{
     Vector3 dir = target.position - transform.position;
     transform.Translate(dir.normalized * speed * Time.deltaTime,
        Space.World);

     if(Vector3.Distance(transform.position, target.position) <= 0.4f)
     {
         GetNextWaypoint();
     }
}

void GetNextWaypoint()
{
     if(wavepointIndex >= waypoints.points.Length - 1)
     {
         Destroy(gameObject);
         return;
     }

     wavepointIndex++;
     target = waypoints.points[wavepointIndex];
}
```

在Start函数中，第一个需要玩家抵达的寻路点就是我们之前创建的waypoints数组中下标为0的那个元素，也就是waypoints数组中第一个Transform类型的元素。此外我们还确定了角色的速度，这个例子中我们设置为10f。

接下来在Update函数中，角色会计算当前位置朝向下一个寻路点的距离，并保存在Vector3类型的变量dir里。角色需要不断沿着这个方向移动，所以我们添加了一行代码控制角色移动，这里用到了transform.Translate函数。知道距离和速度信息后，角色会知道它离下一个位置有多远，一旦它走完到达那个点所需的距离，它就可以移动到下一个点。要做到这一点，我们使用了一个if语句，如果当前角色的位置与目标点的距离小于等于0.4f的话，意味着角色已经抵达目标点，可以开始前往下一个目标点了，调用GetNextWaypoint（）即可获得下一个寻路点。

在GetNextWaypoint（）函数中，角色会去确定是否它已经到达了最终目标点。如果角色到了，那么这个AI角色就可以被销毁了；如果没有，它会继续切换到下一个目标点。这里让wavepointIndex自增一次，即wavepointIndex++，以便在之后的代码中获取下一个目标点。

现在我们将代码挂在角色上并把角色放在起始位置，然后测试游戏能否正常运行：

一切都像预期的那样：角色将从一个点移动到另一个点直到它到达最后一个点，然后它从游戏中消失。但是，仍然有一些需要改进的地方，因为我们的角色在移动过程中总是面朝着同一个方向。我们在改正这个问题时，正好借此机会使用实例化的方式在地图中自动生成敌人。

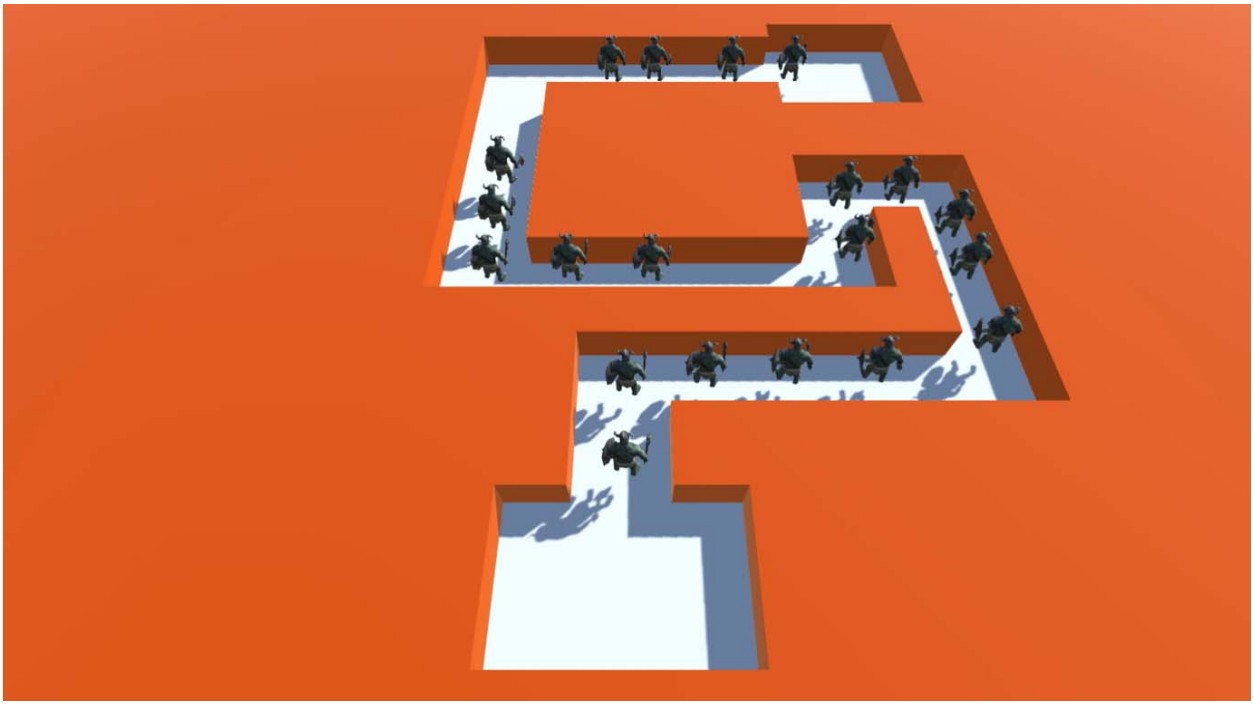
我们使用和在场景中创建寻路点相同的方法，即简单地用物体来代表位置，来创建一个敌人的出生点，并让敌人从这个点开始移动。为了做到这一点，我们写了一些简单的代码以便测试游戏的功能，这帮助我们不必手动地将角色添加到游戏中：

```
    public Transform enemyPrefab;
    public float timeBetweenWaves = 3f;
    public Transform spawnPoint;

    private float countdown = 1f;
    private int waveNumber = 1;

    void Update ()
    {
        if(countdown <= 0f)
        {
            StartCoroutine(SpawnWave());
            countdown = timeBetweenWaves;
        }



    countdown -= Time.deltaTime;
}

IEnumerator SpawnWave ()
{
    waveNumber++;

    for (int i = 0; i < waveNumber; i++)
    {
        SpawnEnemy();
        yield return new WaitForSeconds(0.7f);
    }
}

void SpawnEnemy()
{
    Instantiate(enemyPrefab, spawnPoint.position,
          spawnPoint.rotation);
}
```

到这里，我们已经有了一个可以一波接一波生成敌人的生成器，它将每隔三秒生成一波新的敌人。这对我们观察AI角色的行为很有帮助。我们添加了5个变量，enemyPrefab是我们要创建的角色的预制体，用于在代码中生成敌人角色。timeBetweenWaves表示多久可以生成一波新的敌人。spawnPoint变量确定了角色生成时的起始位置。countdown则是每生成一波后的冷却时间。最后是waveNumber，表示我们当前一共生成了几波敌人（通常它也可以用来区分每一波的难度）。

现在让我们运行游戏，可以看到游戏中出现的角色数量远远不止一个，且每三秒会变得更多。在开发AI角色的同时实现这种游戏逻辑是十分有用的，因为如果我们的某个AI角色有特殊的能力，或者他们的速度不同时，我们就可以在开发期间尽快更新测试。不过我们只是在创建一个小例子，预计它应当可以顺利地工作。

现在让我们再测试一下看看效果：



它看起来更有趣了！我们可以看到，点对点的移动是按照预期的方式工作的，所有进入游戏的角色都知道他们需要去往哪里，并沿着正确的路径走。

我们现在可以更新角色代码，以便在转弯时可以面朝下一个点位置。为了实现这个功能，我们在敌人的脚本中添加一些代码：

```
public float speed;
public int health;
public float speedTurn;

private Transform target;
private int wavepointIndex = 0;


void Start ()
{
     target = waypoints.points[0];
     speed = 10f;
     speedTurn = 0.2f;
}

void Update ()

{
     Vector3 dir = target.position - transform.position;
     transform.Translate(dir.normalized * speed * Time.deltaTime,
           Space.World);

     if(Vector3.Distance(transform.position, target.position) <= 0.4f)
     {
         GetNextWaypoint();
     }

     Vector3 newDir = Vector3.RotateTowards(transform.forward, dir,
           speedTurn, 0.0F);

     transform.rotation = Quaternion.LookRotation(newDir);
}

void GetNextWaypoint()
{
     if(wavepointIndex >= waypoints.points.Length - 1)
     {
         Destroy(gameObject);
         return;
     }

     wavepointIndex++;
     target = waypoints.points[wavepointIndex];
}
```

如上面的代码所示，我们添加了一个变量speedTurn，表示我们角色的转向速度，在Start函数中，我们将其设为0.2f。接着在Update函数中，将这个速度乘以Time.deltaTime，以排除FPS变化对速度的影响。Vector3类型的局部变量newDir代表这一帧角色转向的方向。

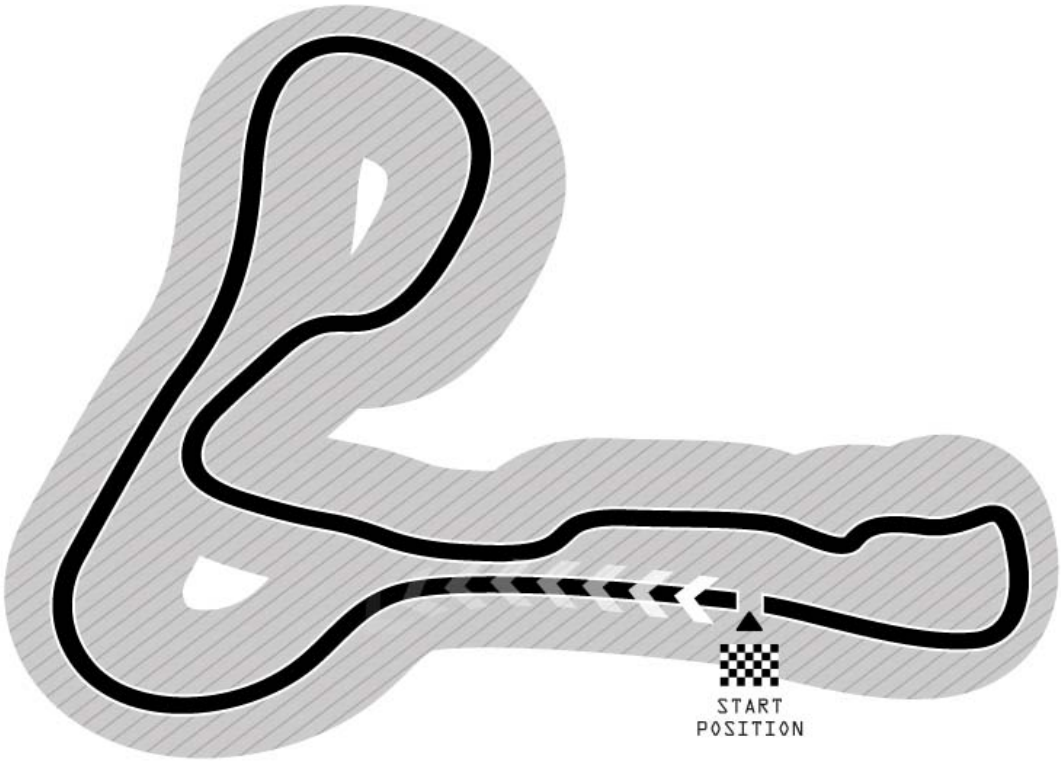
现在如果再运行一下游戏，就可以看到角色会转向下一个寻路点了：

这里，我们可以看到AI角色的反应是正确的，他们从一个点移动到另一个点，并转向下一个位置。现在我们有了塔防游戏的基础，可以继续添加具有独特性的代码来创建一个全新且有趣的游戏了。

赛车类型

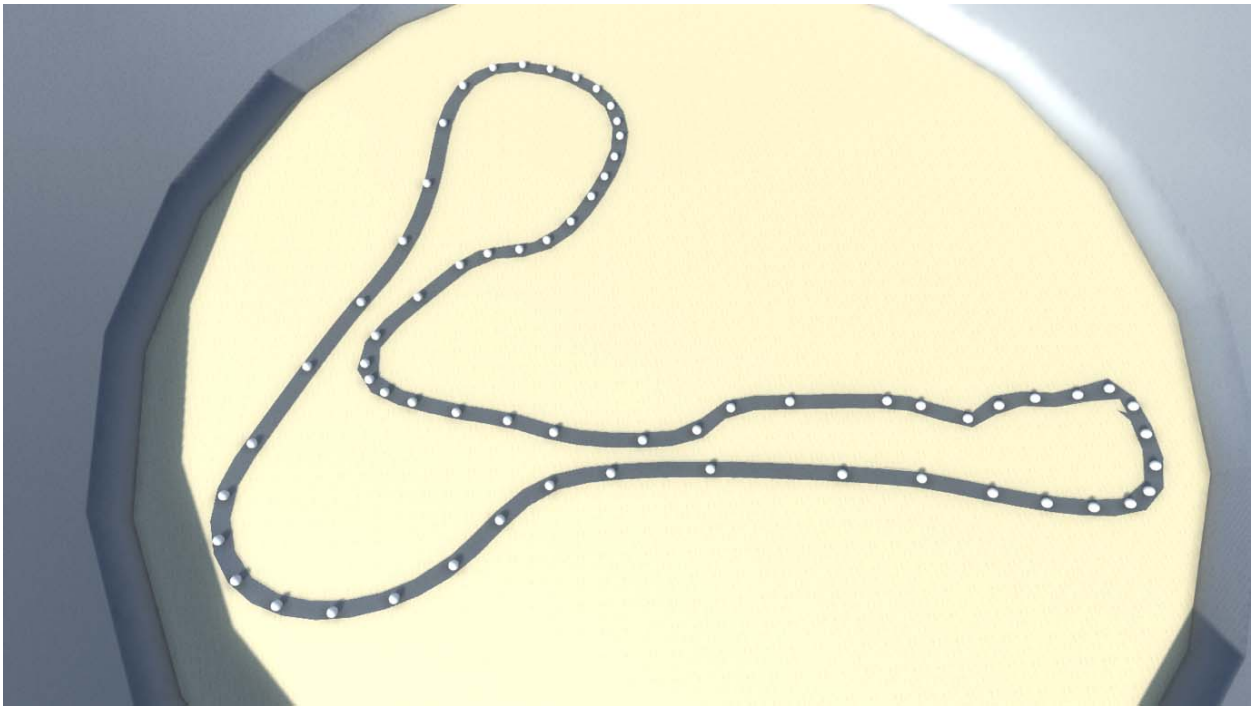
点到点移动是一种几乎可以应用于任何游戏类型的方法，多年来得到了广泛的应用。我们的下一个例子是赛车类型的游戏，AI驾驶员使用点到点移动的方式来对抗玩家。为了实现它，我们需要一条道路和一位驾驶员，然后我们将寻路点放置在道路上，并告诉我们的AI驾驶员沿着这条路线走。这和我们以前做过的很相似，但是这里在我们的角色中有一些行为上的差异，因为我们不希望他转弯时看起来很僵硬，而且在同一张地图上还有其他司机，它们彼此之间不能重叠。

不需要犹豫，让我们开始吧，首先我们需要建立地图，在这种情况下设计赛道：



在设计了赛道之后，我们需要沿着赛道放置很多寻路点，由于这些赛道都是曲线，我们需要创造比以前更多的点，使汽车平滑地沿着道路行驶。

目前，我们做的和之前是一样的，在场景中创建一些空对象，只用它们来表示每一个寻路点的位置：



上图是地图和其中的寻路点，我们可以看到，在赛道的弯道处有更多的点。如果我们要实现从一个点到另一个点的平滑过渡，那么这是十分重要的。

现在，跟之前一样，我们把所有寻路点分到一组里，不过这次我们的代码会有一些不同之处。我们将不会创建管理所有寻路点的代码，我们将在AI驾驶员的脚本中实现这部分计算，并且还要创建一段简单的代码挂在每个寻路点上，用来指定下一个寻路点。

有很多方法可以用来开发这部分内容，根据我们的偏好或者游戏类型的不同，有些方法可能会比其他方法更好。在这个例子中，我们发现之前为塔防游戏编写的代码就不太适合这种赛车类型。

接下来我们开始编写AI驾驶员的代码，我们使用了10个变量，如下所示：

```
public static bool raceStarted = false;

public float aiSpeed = 10.0f;
public float aiTurnSpeed = 2.0f;
public float resetAISpeed = 0.0f;
public float resetAITurnSpeed = 0.0f;

public GameObject waypointController;
public List<Transform> waypoints;
public int currentWaypoint = 0;
public float currentSpeed;
public Vector3 currentWaypointPosition;
```

第一个，raceStarted是一个静态布尔类型的变量，它可以告诉我们AI是否已经开始比赛了。绿灯亮时，比赛才算是开始；如果绿灯没亮的话，raceStarted就为false。下一个是aiSpeed，表示车辆的速度。这是一个用来测试的简化版本，正常情况下，我们需要通过一个计算速度的函数来确定汽车在当前状况下的速度。aiTurnSpeed表示车辆在转向时的速度。接下来是waypointController，它关联着场景中的waypoints组，它可以用来从组中获取寻路点。

在这里，currentWaypoint会告诉我们的驾驶员当前正在跟随的是哪个寻路点。currentSpeed变量表示了车辆当前的速度。最后Vector3类型的变量currentWaypointPosition

```
void Start ()
{
        GetWaypoints();
        resetAISpeed = aiSpeed;
        resetAITurnSpeed = aiTurnSpeed;
}
```

是当前车辆正在跟随的寻路点的位置。

在Start函数中，我们只有三行代码：GetWaypoints（）用来获取寻路点组里存在的所有寻路点，给resetAISpeed和resetAITurnSpee赋值来指定初始速度，它们会影响车辆上挂载的刚体组件。

```
void Update ()
{
      if(raceStarted)
      {
          MoveTowardWaypoints();
      }
}
```

在Update函数中，我们简单的添加了一个if语句，它会检查比赛有没有开始。如果已经开始，那么就进入下一个步骤，对我们的AI驾驶员来说也是最重要的函数MoveTowardWaypoints（）。对于这个例子来说，在比赛开始前赛车还不能跑，但是我们可以启动引擎以及让汽车预加速。

```
void GetWaypoints()
{
  Transform[] potentialWaypoints = waypointController.
      GetComponentsInChildren<Transform>();

  waypoints = new List<Transform>();

  for each(Transform potentialWaypoint in potentialWaypoints)
   {
      if(potentialWaypoint != waypointController.transform)
      {
         waypoints.Add(potentialWaypoint);

      }
   }
}
```

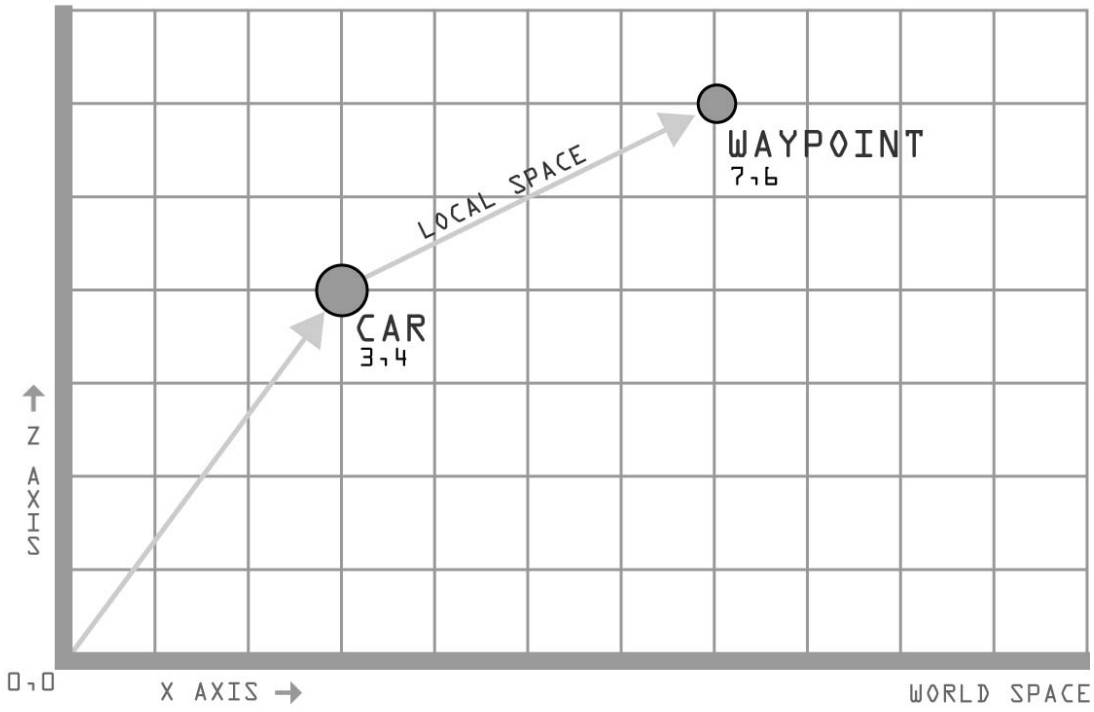
下一步，是我们的GetWaypoints（）函数在Start函数中被调用。这里我们访问waypointController并检索所有在它里面存储的寻路点。

```
void MoveTowardWaypoints()
{
   float currentWaypointX = waypoints[currentWaypoint].position.x;
   float currentWaypointY = transform.position.y;
   float currentWaypointZ = waypoints[currentWaypoint].position.z;

   Vector3 relativeWaypointPosition = transform.
      InverseTransformPoint (new Vector3(currentWaypointX,
      currentWaypointY, currentWaypointZ));
   currentWaypointPosition = new Vector3(currentWaypointX,
       currentWaypointY, currentWaypointZ);

   Quaternion toRotation = Quaternion.LookRotation
      (currentWaypointPosition - transform.position);
   transform.rotation = Quaternion.RotateTowards
      (transform.rotation, toRotation, aiTurnSpeed);

   GetComponent<Rigidbody>().AddRelativeForce(0, 0, aiSpeed);

   if(relativeWaypointPosition.sqrMagnitude < 15.0f)
   {
      currentWaypoint++;

      if(currentWaypoint >= waypoints.Count)
      {
         currentWaypoint = 0;
      }
   }

   currentSpeed = Mathf.Abs(transform.
     InverseTransformDirection
    (GetComponent<Rigidbody>().velocity).z);

   float maxAngularDrag = 2.5f;
   float currentAngularDrag = 1.0f;
   float aDragLerpTime = currentSpeed * 0.1f;
   float maxDrag = 1.0f;
   float currentDrag = 3.5f;
   float dragLerpTime = currentSpeed * 0.1f;

   float myAngularDrag = Mathf.Lerp(currentAngularDrag,
      maxAngularDrag, aDragLerpTime);
   float myDrag = Mathf.Lerp(currentDrag, maxDrag, dragLerpTime);

   GetComponent<Rigidbody>().angularDrag = myAngularDrag;
   GetComponent<Rigidbody>().drag = myDrag;
 }
```

最后，我们有一个MoveTowardsWaypoints（）函数，由于车辆在运动方面比简单的塔防中的角色更具深度，我们打算扩展这部分代码并实现更多的内容。

首先，我们获取一下当前正在被使用的寻路点的位置。我们选择了分别获取其中每个轴的信息，currentWaypointX对应X轴，currentWaypointY对应Y轴，currentWaypointZ对应Z轴。

接着，我们创建了一个新的Vector3类型的方向relativeWaypointPosition，它被用来计算下一个寻路点离当前位置的距离，而且要将这个方向从世界空间转换到本地空间，这里使用InverseTransformDirection函数来实现。

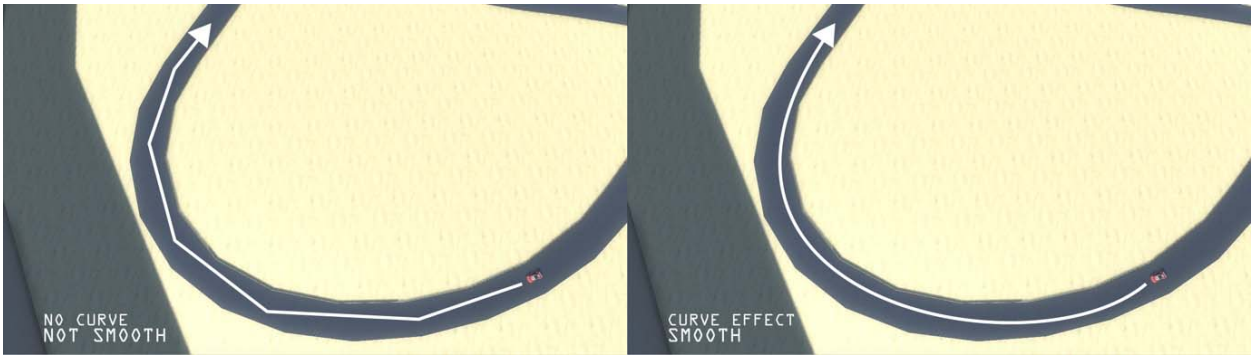


如上图所示，我们想要计算在本地坐标系下车辆与寻路点之间的距离，并告诉我们的驾驶员寻路点在它的右边还是左边。这个功能是必要的，毕竟车轮的旋转控制着汽车的速度且四个车轮有独立的旋转速度，如果我们继续深入开发这个游戏，这些基本功能都是必要的。

为了在寻路点之间做出平滑的旋转，我们使用下面的代码：

```
Quaternion toRotation = Quaternion.LookRotation
        (currentWaypointPosition - transform.position);
transform.rotation = Quaternion.RotateTowards
        (transform.rotation, toRotation, aiTurnSpeed);
```

这是塔防游戏中的代码修改后的版本。它将使我们的车顺利地走向寻路点，同时平滑地转向寻路点。这对车辆转向的效果有明显影响，如果没有做平滑处理的话，赛车就会立即转向下一个寻路点，那样的话看起来会不真实：



正如我们看到的，直线不适合我们目前正在创建的游戏类型。这在其他类型的游戏中没什么，例如塔防游戏，但是对于竞速游戏来说，必须要对我们的代码做一些修改以适应新的情况。

其余的代码也是这样，调整我们正在创建的代码以适应在赛道上行驶的赛车。还会有一些作用力的因素，例如drag代表汽车与路面间的摩擦力。当我们的车辆转向时，它将根据当时速度产生滑动，这些细节在这里都会被考虑，以创造一个更真实的、更符合物理学的点到点运动。

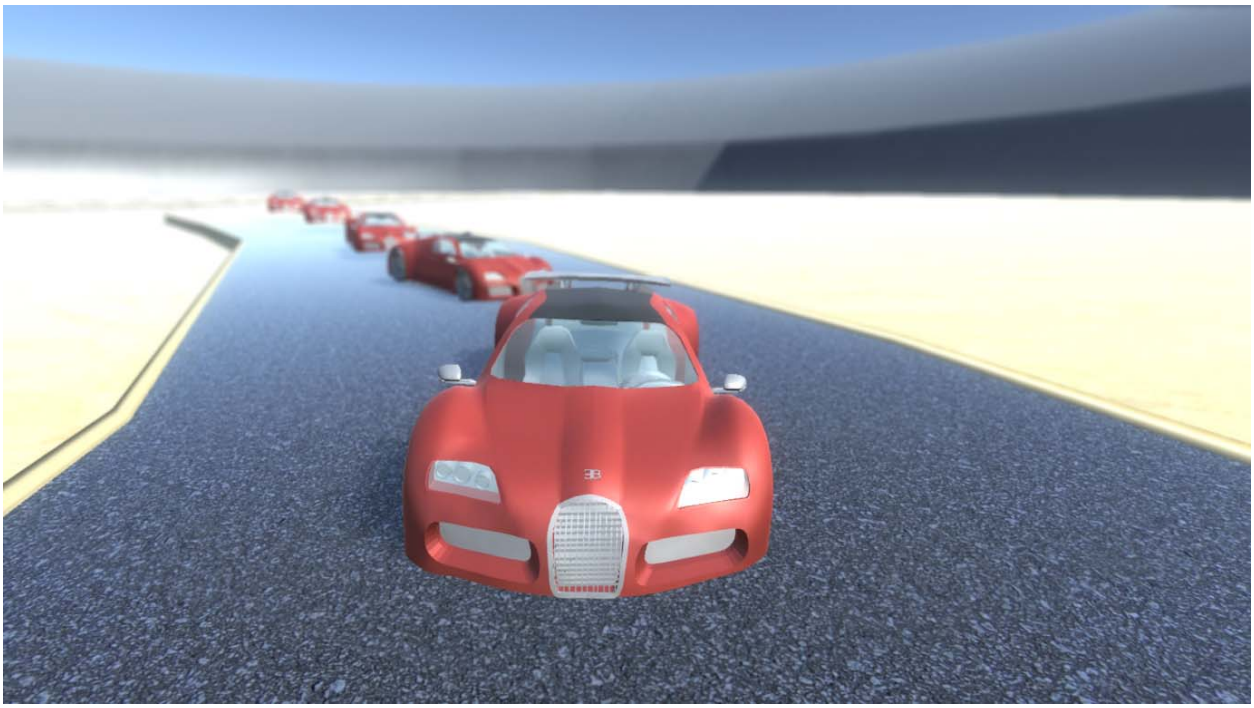
这些是我们在这个例子中使用的全部代码：

```
public static bool raceStarted = false;

public float aiSpeed = 10.0f;
public float aiTurnSpeed = 2.0f;
public float resetAISpeed = 0.0f;
public float resetAITurnSpeed = 0.0f;

public GameObject waypointController;
public List<Transform> waypoints;
public int currentWaypoint = 0;
public float currentSpeed;
public Vector3 currentWaypointPosition;

void Start ()
{
     GetWaypoints();
     resetAISpeed = aiSpeed;
     resetAITurnSpeed = aiTurnSpeed;
}

void Update ()
{
   if(raceStarted)
   {
       MoveTowardWaypoints();
   }
}

void GetWaypoints()
 {
     Transform[] potentialWaypoints =
       waypointController.GetComponentsInChildren<Transform>();

     waypoints = new List<Transform>();

     foreach(Transform potentialWaypoint in potentialWaypoints)
     {
         if(potentialWaypoint != waypointController.transform)
         {
             waypoints.Add(potentialWaypoint);
         }
     }
}

void MoveTowardWaypoints()
```

```
{
    float currentWaypointX = waypoints[currentWaypoint].position.x;
    float currentWaypointY = transform.position.y;
    float currentWaypointZ = waypoints[currentWaypoint].position.z;

    Vector3 relativeWaypointPosition = transform.
       InverseTransformPoint (new Vector3(currentWaypointX,
       currentWaypointY, currentWaypointZ));
    currentWaypointPosition = new Vector3(currentWaypointX,
       currentWaypointY, currentWaypointZ);

    Quaternion toRotation = Quaternion.
       LookRotation(currentWaypointPosition - transform.position);
    transform.rotation = Quaternion.RotateTowards
       (transform.rotation, toRotation, aiTurnSpeed);

    GetComponent<Rigidbody>().AddRelativeForce(0, 0, aiSpeed);

    if(relativeWaypointPosition.sqrMagnitude < 15.0f)
    {
        currentWaypoint++;

        if(currentWaypoint >= waypoints.Count)
        {
            currentWaypoint = 0;
         }
    }

    currentSpeed = Mathf.Abs(transform.
        InverseTransformDirection
        (GetComponent<Rigidbody>().velocity).z);

    float maxAngularDrag = 2.5f;
    float currentAngularDrag = 1.0f;
    float aDragLerpTime = currentSpeed * 0.1f;

    float maxDrag = 1.0f;
    float currentDrag = 3.5f;
    float dragLerpTime = currentSpeed * 0.1f;

    float myAngularDrag    = Mathf.Lerp(currentAngularDrag,
        maxAngularDrag, aDragLerpTime);
    float myDrag = Mathf.Lerp(currentDrag, maxDrag, dragLerpTime);

    GetComponent<Rigidbody>().angularDrag = myAngularDrag;
    GetComponent<Rigidbody>().drag = myDrag;
}
```

如果现在开始测试游戏，我们可以看到它工作得很好。汽车自动行驶，平滑转弯，能够按预定轨道跑完全程。

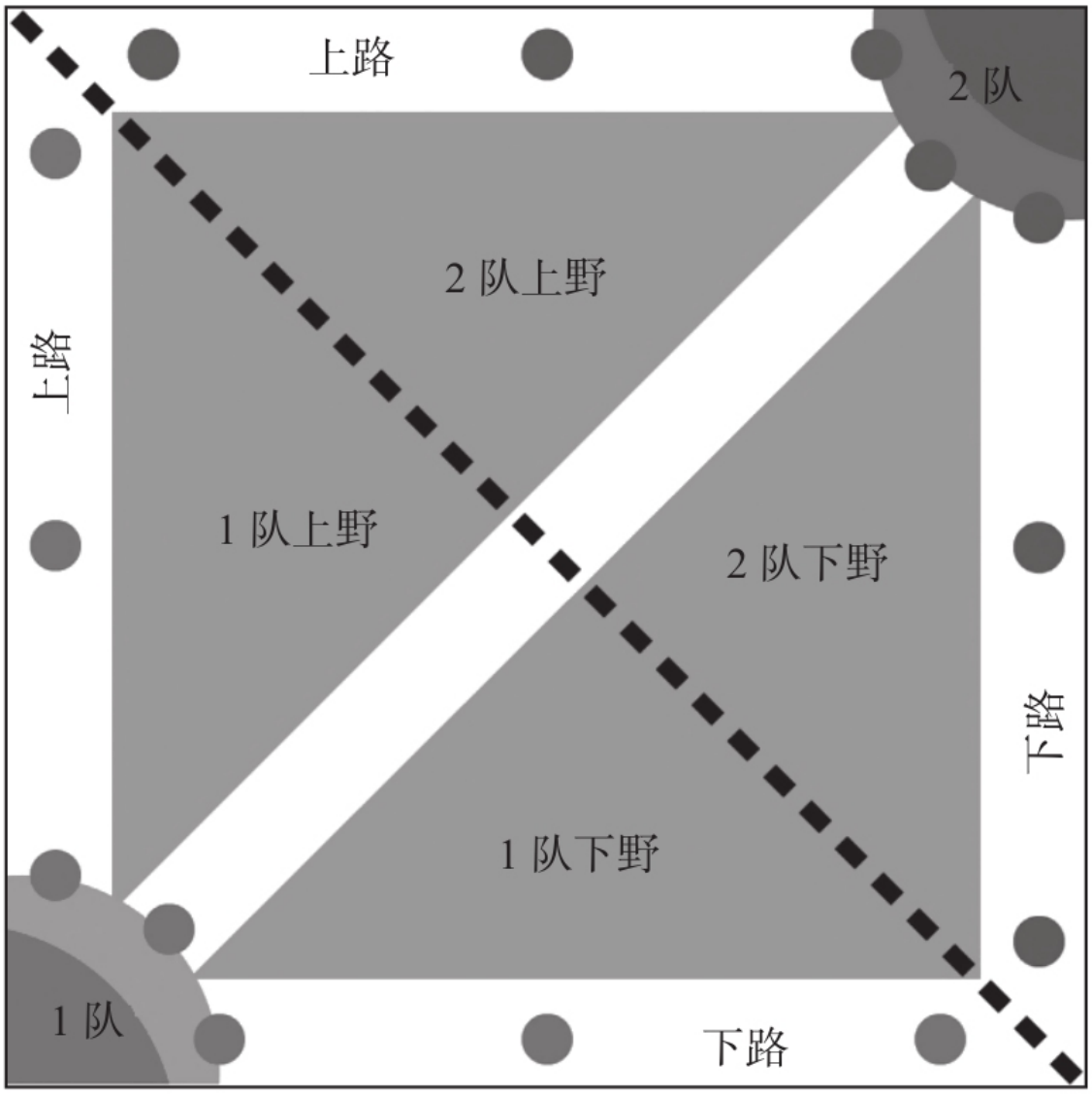
现在我们有了一个基本的点到点移动的系统，我们可以为这段AI程序实现更多的功能，并开始开发我们想要的游戏。建议在开发任何细节之前，先从制作游戏的主要功能开始。这将有助于我们判断这个想法究竟有没有想象中的那么好。



MOBA类型

点到点移动是控制角色移动最常用的方法之一。它能够被广泛使用，是因为让角色从一个点移动到另一个点通常正是我们想要的移动方式，角色能以这种方式到达某个目的地或跟随另一个角色。还有一种游戏类型这种移动方式也适用，那就是多人在线战术竞技游戏（MOBA），这种游戏类型最近非常受欢迎。通常，NPC角色在起始位置生成，并按照预定的路径到达敌人的塔，类似于塔防游戏中的敌人，但在这种游戏中AI角色的移动区域和玩家的移动区域是同样的，可能会互相干扰。

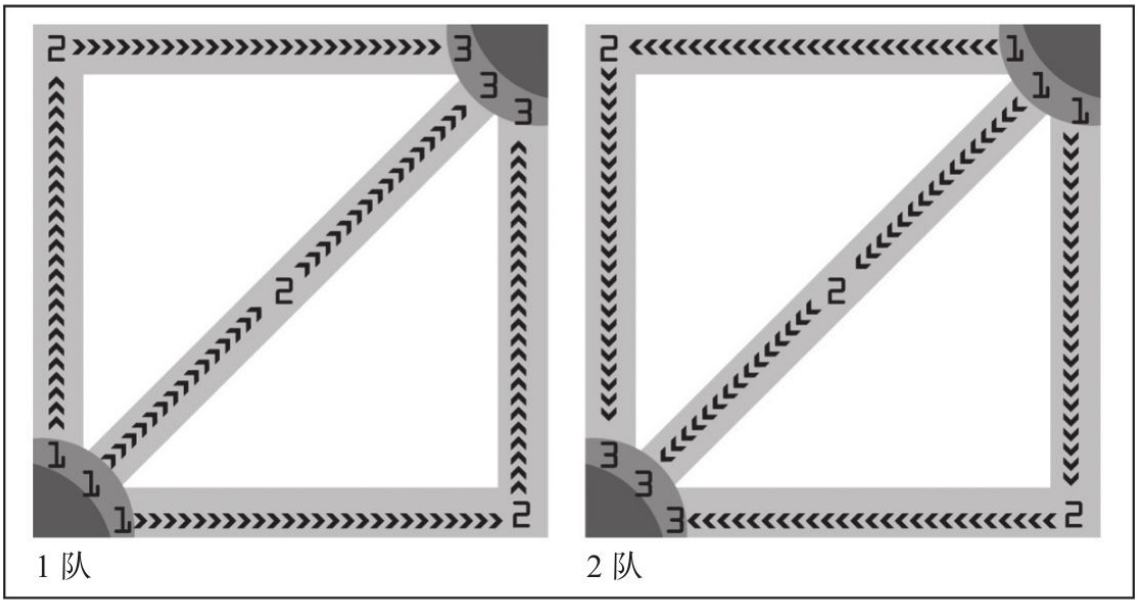
MOBA的地图分为相等的两部分，双方互相攻击并且每边会产生不同的怪物组，这些怪物组由一系列的小兵组成。当它们跟随自己的行进路线移动时，如果一个怪物组遇到另一个敌对怪物组，它们就会停止行进，开始攻击。战斗结束后，幸存者继续行进：



在这个例子中，我们会重现游戏中的一部分内容：怪物组会在起始位置生成出来，并沿着自己的行进路线移动，当它们找到敌人时会停下来，直到它们赢得战斗后又会继续向着下一个目标移动。然后，我们将创建由玩家或计算机控制的英雄角色的基本动作：二者都能在地图上自由移动，角色需要跟随玩家或计算机指示的方向移动，同时避开所有障碍。

我们首先会导入地图进游戏中。我们选择了一个常用的MOBA风格的地图，如下图所示：

下一步是在地图上添加寻路点。这里有六种不同的寻路点组，因为每个小组有三个不同的路径，每个队伍只能沿着一条路走。我们从基地位置开始，然后添加更多的寻路点，直到我们到达敌人的基地。下图展示了我们创建的示例。



我们需要为每个队伍创建3个不同的寻路点组，这是因为会有3个不同的生成位置，它们将彼此独立工作。在配置好寻路点后，我们可以对它们进行分组并在代码中获取它们的

引用、收集位置信息。在这个例子中，我们可以使用与塔防游戏相同的代码，因为在角色跟随路径这个部分，两者是差不多的：

```
public static Transform[] points;

void Awake ()
{
     points = new Transform[transform.childCount];
     for (int i = 0; i < points.Length; i++)
     {
         points[i] = transform.GetChild(i);
     }
}
```

由于有6个不同的寻路点组，这就需要新建6份这样的代码并为其分别命名。由于我们生成出怪物后需要为它们选择正确的行进路线，因此，在代码里清晰地命名这些寻路点组，可以让我们更容易地理解哪个组代表哪个路线，举个例子：命名为1_Top/1_Middle/1_Bottom和2_Top/2_Middle/2_Bottom。前缀的数字表示它们的队伍，后缀表示它们的位置。在这种情况下，我们将代码中的points改为表示每个路线的正确名称。

```
public static Transform[] 1_Top;

void Awake ()
{
     1_Top = new Transform[transform.childCount];
     for (int i = 0; i < 1_Top.Length; i++)
     {
         1_Top[i] = transform.GetChild(i);
     }
}
```

1队上路：

1队中路：

```
public static Transform[] 1_Middle;

void Awake ()
{
     1_Middle = new Transform[transform.childCount];
     for (int i = 0; i < 1_Top.Length; i++)
     {
         1_Middle[i] = transform.GetChild(i);
     }
}
```

```
public static Transform[] 1_Bottom;

void Awake ()
{
     1_Bottom = new Transform[transform.childCount];
     for (int i = 0; i < 1_Top.Length; i++)
     {
         1_Bottom[i] = transform.GetChild(i);
     }
}
```

1队下路：

2队上路：

```
public static Transform[] 2_Top;

void Awake ()
{
      2_Top = new Transform[transform.childCount];
      for (int i = 0; i < 1_Top.Length; i++)
      {
          2_Top[i] = transform.GetChild(i);
      }
}
```

```
public static Transform[] 2_Middle;

void Awake ()
{
      2_Middle = new Transform[transform.childCount];
      for (int i = 0; i < 2_Middle.Length; i++)
      {
          2_Middle[i] = transform.GetChild(i);
      }
}
```

2队中路：

```
public static Transform[] 2_Bottom;

void Awake ()
{
      2_Bottom = new Transform[transform.childCount];
      for (int i = 0; i < 2_Bottom.Length; i++)
      {
          2_Bottom[i] = transform.GetChild(i);
      }
}
```

2队下路：

现在我们已经在代码中为每一条线路的寻路点创建了一个数组，接下来我们可以转到AI角色部分的代码。我们可以为每个队伍创建一份单独的代码，或者也可以将每个队伍都写在同一份代码中并使用if语句来判断角色应该朝哪条线路行进。这里我们选择写在同一份代码中。这样，只要每次更新角色代码，就可以同时对两个队伍都生效。和之前一样，我们可以在塔防游戏中的敌人代码的基础上做一些调整，以使其适应我们目前正在做的游戏[1]。

```
public float speed;
public int health;
public float speedTurn;

private Transform target;
private int wavepointIndex = 0;
```

```
{
     target = waypoints.points[0];
     speed = 10f;
     speedTurn = 0.2f;
}

void Update ()
{
     Vector3 dir = target.position - transform.position;
     transform.Translate(dir.normalized * speed * Time.deltaTime,
        Space.World);

     if(Vector3.Distance(transform.position, target.position) <= 0.4f)
     {
         GetNextWaypoint();
     }

     Vector3 newDir = Vector3.RotateTowards(transform.forward, dir,
        speedTurn, 0.0F);

     transform.rotation = Quaternion.LookRotation(newDir);
}

void GetNextWaypoint()
{
     if(wavepointIndex >= waypoints.points.Length - 1)
     {
         Destroy(gameObject);
         return;
     }

     wavepointIndex++;
     target = waypoints.points[wavepointIndex];
}
```

这份代码可以让角色沿着指定的路线移动，且能够平滑地转向。这里我们只需要调整这些代码以适应我们所做的游戏类型即可。要做到这一点，首先需要考虑的是更改我们之前创建的寻路点的命名，并添加if语句来选择角色需要遵循的方向。

我们通过添加一些信息来区分某个角色是属于1队还是2队。为了达成这点，我们需要

```
public bool Team1;
public bool Team2;
```

创建两个新的布尔类型的变量：

这让我们能够判断某个角色是1队的还是2队的，这两者不会同时为true。现在我们可以在角色代码中实现更多细节，让他知道自己应该走哪条路：

```
public bool Top;
public bool Middle;
public bool Bottom;
```

我们还需添加三个bool类型的变量，用来表示角色需要在哪一条路线上行进。在确定角色是属于哪一队后，我们添加一个if语句来决定角色会走哪一条线路。

一旦我们添加了这些变量，我们需要为我们的角色指定需要跟随的寻路点组，我们可以在Start函数里实现这些：

```
if(Team1 == true)
 {
          if(Top == true)
          {
              target = 1_Top.1_Top[0];
          }

          if(Middle == true)
          {
              target = 1_Middle.1_Middle[0];
          }

          if(Bottom == true)
          {
             target = 1_Bottom.1_Top[0];
          }
 }

 if(Team2 == true)
 {
          if(Top == true)
          {
             target = 2_Top.2_Top[0];
          }

          if(Middle == true)
          {
              target = 2_Middle.2_Middle[0];
          }

         if(Bottom == true)
         {
             target = 2_Bottom.2_Top[0];
         }
}
```

这允许角色查询自己所属的队伍在哪条路生成，以及他将在哪条路线上行进。我们需要调整代码的其余部分，以便能够被应用于这个示例。下一步我们会修改GetNext-Waypoint（）函数。需要添加if语句来让角色知道它的下一个寻路点，这与我们在Start函数里做的类似：

```
void GetNextWaypoint()
{
    if(Team1 == true)
    {
       if(Top == true)
       {
          if(wavepointIndex >= 1_Top.1_Top.Length - 1)
          {
             Destroy(gameObject);
              return;
           }

           wavepointIndex++;
```

```
            target = 1_Top.1_Top[wavepointIndex];
        }

        if(Middle == true)
        {
            if(wavepointIndex >= 1_Middle.1_Middle.Length - 1)
            {
                Destroy(gameObject);
                return;
            }

            wavepointIndex++;
            target = 1_Middle.1_Middle[wavepointIndex];
        }

        if(Bottom == true)
        {
            if(wavepointIndex >= 1_Bottom.1_Bottom.Length - 1)
            {
                Destroy(gameObject);
                return;
            }

            wavepointIndex++;
            target = 1_Bottom.1_Bottom[wavepointIndex];
    }
}

if(Team2 == true)
{
  if(Top == true)
  {
      if(wavepointIndex >= 2_Top.2_Top.Length - 1)
      {
          Destroy(gameObject);
          return;
      }

      wavepointIndex++;
      target = 2_Top.2_Top[wavepointIndex];
    }

    if(Middle == true)
    {
        if(wavepointIndex >= 2_Middle.2_Middle.Length - 1)
        {
            Destroy(gameObject);
            return;
        }

        wavepointIndex++;
        target = 2_Middle.2_Middle[wavepointIndex];
    }

    if(Bottom == true)
    {
        if(wavepointIndex >= 2_Bottom.2_Bottom.Length - 1)
```

```
            {
                Destroy(gameObject);
                return;
            }

            wavepointIndex++;
            target = 2_Bottom.2_Bottom[wavepointIndex];
        }
    }
}
```

到了这一步，如果我们添加一个角色到游戏中，并指定对应的AI脚本代码，它将遵循

所选择的路线行进：

看起来这能够正常工作，然而我们准备实现更多的功能以创造一组能够完美地沿着一条通往敌方炮塔路线行进的怪物，且当遭遇敌方怪物或者英雄时停止前行并开始战斗。现在我们的怪物组有了基本的移动功能，可以给它们添加更多的细节或独特性。这里，我们附上怪物组AI角色的完整代码：

```
public float speed;
public int health;
public float speedTurn;

public bool Team1;
public bool Team2;

public bool Top;
public bool Middle;
public bool Bottom;

private Transform target;
private int wavepointIndex = 0;
```

在前面的变量更新之后，我们转到Start方法：

```
void Start ()
{
   if(Team1 == true)
   {
      if(Top == true)
      {
          target = 1_Top.1_Top[0];
       }

       if(Middle == true)
       {
            target = 1_Middle.1_Middle[0];
        }

        if(Bottom == true)
        {
            target = 1_Bottom.1_Top[0];
        }
    }

    if(Team2 == true)
    {
        if(Top == true)
        {
           target = 2_Top.2_Top[0];
         }

        if(Middle == true)
        {
           target = 2_Middle.2_Middle[0];
        }

         if(Bottom == true)
         {
             target = 2_Bottom.2_Top[0];
          }
  }
  speed = 10f;
  speedTurn = 0.2f;
}
```

接下来是被每帧调用的Update方法：

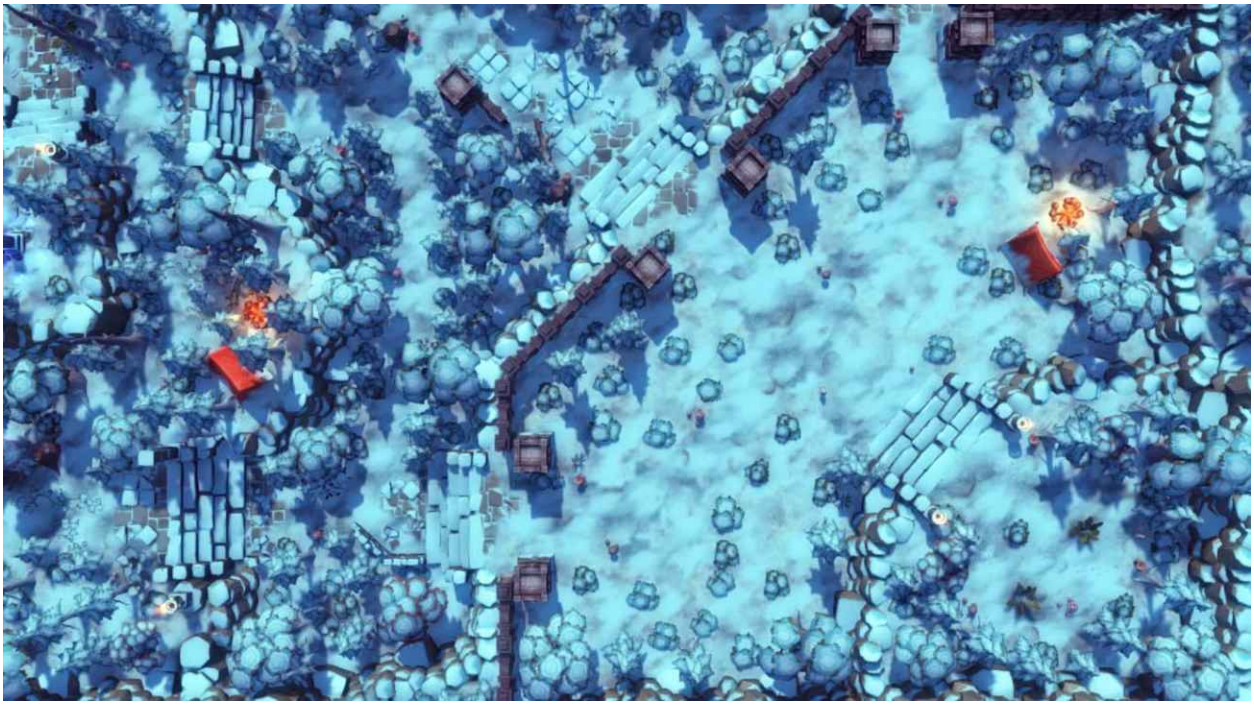
```
void Update ()
{
   Vector3 dir = target.position - transform.position;
   transform.Translate(dir.normalized * speed * Time.deltaTime,
      Space.World);

   if(Vector3.Distance(transform.position, target.position) <= 0.4f)
   {
       GetNextWaypoint();
   }

   Vector3 newDir = Vector3.RotateTowards(transform.forward, dir,
      speedTurn, 0.0F);

   transform.rotation = Quaternion.LookRotation(newDir);
}
```

```
void GetNextWaypoint()
{
    if(Team1 == true)
    {
      if(Top == true)
      {
        if(wavepointIndex >= 1_Top.1_Top.Length - 1)
        {
          Destroy(gameObject);
          return;
        }

        wavepointIndex++;
        target = 1_Top.1_Top[wavepointIndex];
      }

      if(Middle == true)
      {
        if(wavepointIndex >= 1_Middle.1_Middle.Length - 1)
        {
           Destroy(gameObject);
           return;
         }

         wavepointIndex++;
         target = 1_Middle.1_Middle[wavepointIndex];
      }
  if(Bottom == true)
  {
     if(wavepointIndex >= 1_Bottom.1_Bottom.Length - 1)
     {
          Destroy(gameObject);
          return;
      }

      wavepointIndex++;
      target = 1_Bottom.1_Bottom[wavepointIndex];
  }
}

if(Team2 == true)
{
   if(Top == true)
   {
     if(wavepointIndex >= 2_Top.2_Top.Length - 1)
     {
          Destroy(gameObject);
          return;
      }

       wavepointIndex++;
       target = 2_Top.2_Top[wavepointIndex];
    }

    if(Middle == true)
    {
       if(wavepointIndex >= 2_Middle.2_Middle.Length - 1)
```

```
        {
            Destroy(gameObject);
            return;
        }

        wavepointIndex++;
        target = 2_Middle.2_Middle[wavepointIndex];
      }

      if(Bottom == true)
      {
        if(wavepointIndex >= 2_Bottom.2_Bottom.Length - 1)
        {
            Destroy(gameObject);
            return;
        }

        wavepointIndex++;
        target = 2_Bottom.2_Bottom[wavepointIndex];
      }
    }
}
```

另一个对于MOBA游戏来说非常重要的方面就是英雄的移动。即使英雄由玩家控制，角色也需要AI来决定它的移动路径，以到达指定的目的地。为了完成这些任务，我们将首先介绍点到点的移动方法。然后我们会继续处理相同的例子，但是会用更先进的方法使我们的角色选择最佳路径来移动到目的地，而且还不需要使用任何寻路点。

这个例子也阐述了如何创建一个可以跟随玩家的角色。要做到这一点，我们需要配置所有可能的路径。我们希望AI能够避免与物体碰撞或穿过墙壁，例如：



让我们重点关注一下这块地图上的区域。正如我们所看到的，墙壁和树木挡住了地图的一部分，而这些就是角色不能穿过的地方。使用寻路点的方法，我们需要在地图上创建角色需要跟随的路径。和之前的例子不同，路点不会有任何特定的顺序，因为角色可以向任何方向移动，所以我们无法预测它将选择哪条路径。

我们通过在可行走的区域上安置寻路点来防止角色在不可行走区域移动。

上图中，地图上的星星是我们已经创建的寻路点，应当将这些寻路点只放在角色能够行走的区域。如果角色要从一个位置移动到另一个位置，它必须跟随这些寻路点，直到它到达离目的地最近的寻路点。

在这个游戏机制中，角色想要到达某个目的地有多种原因，例如跟随玩家，到基地恢复生命值，向敌人的防御设施移动并摧毁它以及许多其他原因。移动的功能与AI的其他功能是独立的，它只要确保在地图上正确地移动即可，这个寻路点系统在任何情况下都能够正常工作。

下面我们可以看到完成这项工作所需的完整代码。之后我们将详细解释代码的工作过程，以便更好地理解如何复用此代码，让它可以在不同的游戏类型中正常工作：

```
public float speed;
private List <GameObject> wayPointsList;
private Transform target;
private GameObject[] wayPoints;

void Start ()
{

    target = GameObject.FindGameObjectWithTag("target").transform;
    wayPointsList = new List<GameObject>();

    wayPoints = GameObject.FindGameObjectsWithTag("wayPoint");

    for each(GameObject newWayPoint in wayPoints)
    {
       wayPointsList.Add(newWayPoint);
     }
 }

 void Update ()
 {
    Follow();
 }
```

```
 void Follow ()
 {
    GameObject wayPoint = null;

    if (Physics.Linecast(transform.position, target.position))
    {
       wayPoint = findBestPath();
     }

     else
     {
        wayPoint = GameObject.FindGameObjectWithTag("target");
      }

      Vector3 Dir = (wayPoint.transform.position -
              transform.position).normalized;
      transform.position += Dir * Time.deltaTime * speed;
      transform.rotation = Quaternion.LookRotation(Dir);
   }

GameObject findBestPath()
{
    GameObject bestPath = null;
    float distanceToBestPath = Mathf.Infinity;

    for each(GameObject go in wayPointsList)
    {
       float distToWayPoint = Vector3.
          Distance(transform.position, go.transform.position);
       float distWayPointToTarget = Vector3.
          Distance(go.transform.position,
          target.transform.position);
       float distToTarget = Vector3.
          Distance(transform.position, target.position);
       bool wallBetween = Physics.Linecast
          (transform.position, go.transform.position);

       if((distToWayPoint < distanceToBestPath)
           && (distToTarget > distWayPointToTarget)
               &&  (!wallBetween))
        {
            distanceToBestPath = distToWayPoint;
            bestPath = go;
        }

        else
        {
            bool wayPointToTargetCollision = Physics.Linecast
               (go.transform.position, target.position);
            if(!wayPointToTargetCollision)
            {
                 bestPath = go;
             }
        }
    }
    return bestPath;
}
```

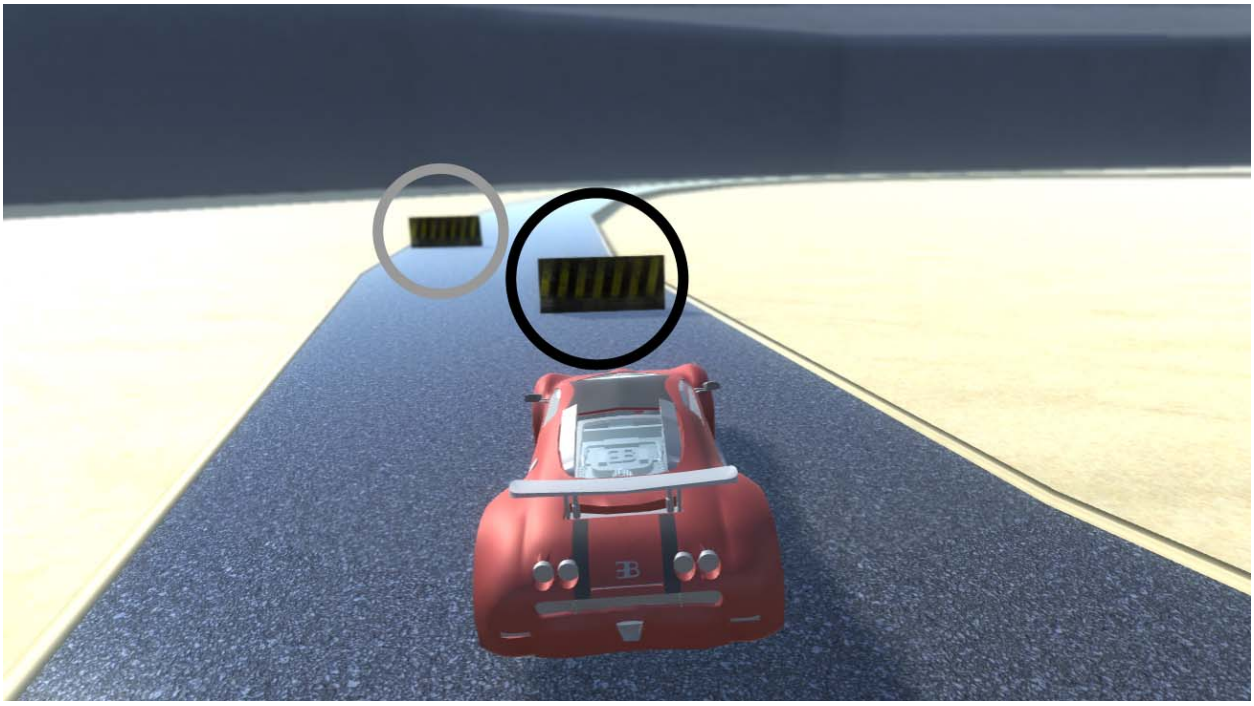
如果我们把这些代码挂到角色身上，并运行游戏，就可以看到它已经可以正常工作了。角色使用寻路点的位置信息在地图上到达了我们设置的目的地。这种方法同样适用于NPC角色和玩家角色，因为这两种角色都需要避免与墙壁和障碍物碰撞。

如果我们继续增加寻路点以使其遍及整个地图，我们就有了一个基本的可以正常工作的MOBA游戏了，每个小兵营地生成的怪物都会沿着正确的行军路线移动，英雄角色则可以在地图上自由移动，且不会撞到墙上。

点到点移动和躲避动态障碍

现在我们有了可以跟随正确路线移动并绕开静态障碍的角色，我们已经准备好进入下一个阶段，让这些角色在从点到点移动时也可以躲开动态的障碍。我们将修改本章中创建的三个不同的示例，并了解如何在这些示例中添加躲避动态障碍物的技术。

这三种方法作为主要移动方式几乎覆盖了每一种使用点对点移动的游戏类型，我们将以这些例子作为指导，创造出新的方法：



让我们从赛车游戏开始：我们有一辆赛车一直在赛道上行驶，直到它完成比赛。如果汽车单独行驶，没有道路障碍，就没有必要避开任何障碍，但通常障碍物会使比赛变得更有趣、更具挑战性。特别是当这些障碍不是预先决定而是随机出现的时候。一个很好的例子就是《马里奥赛车》（Mario Kart），在那里它们抛出香蕉和其他物体来对付其他玩家，而物体没有任何预先定义的位置，所以角色无法预测它们将要在什么位置躲避。因此，AI驾驶员有必要实现能够实时躲避这些动态障碍的功能。

假设两个物体突然出现在AI角色与下一个寻路点之间的路上，我们希望角色能够预料到碰撞并转变方向以避免撞到物体。我们将在这里使用一种结合寻路点移动与迷宫移动的方法，而同一时刻AI只能选择两种方法其中的一个，这正是我们需要添加到代码中的策略，AI角色可以根据它面临的现状选择最合适的方法：

```
    public static bool raceStarted = false;
    public float aiSpeed = 10.0f;
    public float aiTurnSpeed = 2.0f;
    public float resetAISpeed = 0.0f;
    public float resetAITurnSpeed = 0.0f;

    public GameObject waypointController;
    public List<Transform> waypoints;
    public int currentWaypoint = 0;
    public float currentSpeed;
    public Vector3 currentWaypointPosition;

    public static bool isBlocked;
    public static bool isBlockedFront;
    public static bool isBlockedRight;
    public static bool isBlockedLeft;
```

```
void Start ()
{
        GetWaypoints();
        resetAISpeed = aiSpeed;
        resetAITurnSpeed = aiTurnSpeed;
 }
```

在更新上面这些变量后，我们可以转到在第一帧中被调用的Start方法：

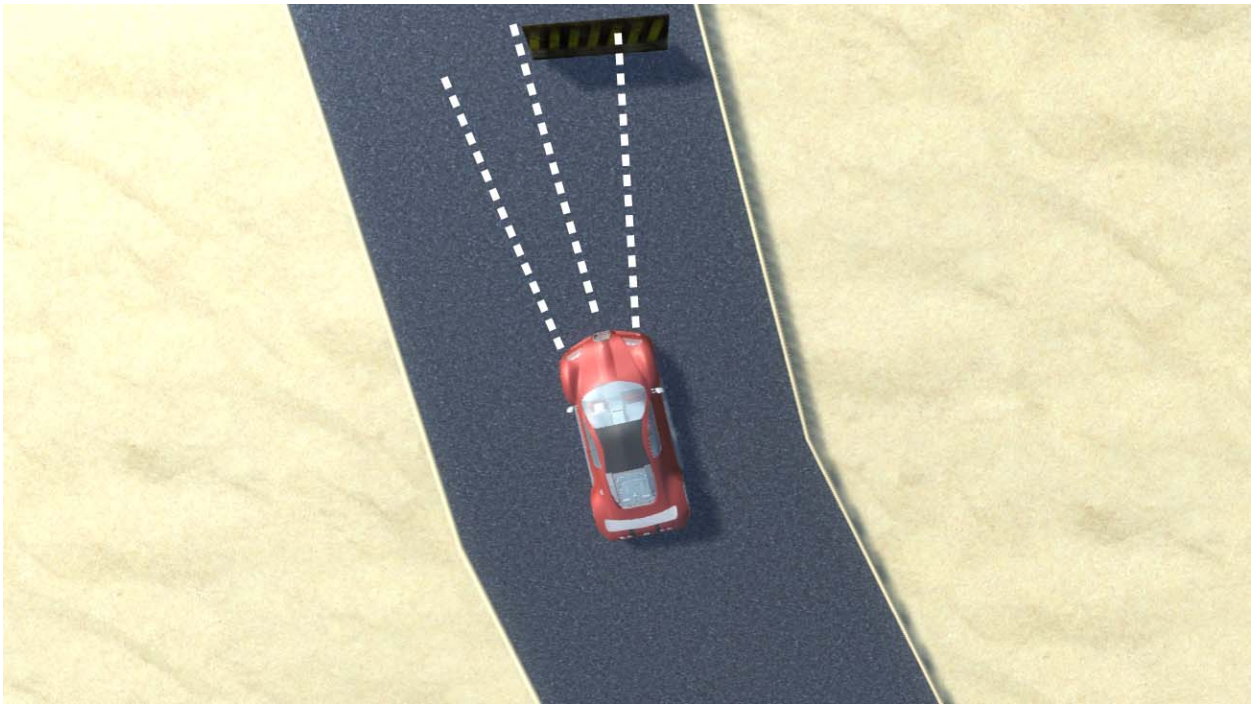
接下来是每帧被调用的Update方法：

```
void Update ()
 {
    if(raceStarted && isBlocked == false)
    {
       MoveTowardWaypoints();
     }

    if(raceStarted && isBlockedFront == true
        && isBlockedLeft == false && isBlockedRight == false)
     {
         TurnRight();
     }
```

```
    if(raceStarted && isBlockedFront == false
        && isBlockedLeft == true && isBlockedRight == false)
    {
        TurnRight();
    }

  if(raceStarted && isBlockedFront == false
      && isBlockedLeft == false && isBlockedRight == true)
  {
      TurnLeft();
  }
}

void GetWaypoints()
 {
      Transform[] potentialWaypoints = waypointController.
          GetComponentsInChildren<Transform>();

      waypoints = new List<Transform>();

      for each(Transform potentialWaypoint in potentialWaypoints)
       {
           if(potentialWaypoint != waypointController.transform)
           {
               waypoints.Add(potentialWaypoint);
            }
        }
}

void MoveTowardWaypoints()
{
  float currentWaypointX = waypoints[currentWaypoint].position.x;
  float currentWaypointY = transform.position.y;
  float currentWaypointZ = waypoints[currentWaypoint].position.z;

  Vector3 relativeWaypointPosition = transform.
     InverseTransformPoint (new Vector3(currentWaypointX,
     currentWaypointY, currentWaypointZ));
  currentWaypointPosition = new Vector3(currentWaypointX,
     currentWaypointY, currentWaypointZ);

  Quaternion toRotation = Quaternion.
     LookRotation(currentWaypointPosition - transform.position);
  transform.rotation = Quaternion.
     RotateTowards(transform.rotation, toRotation, aiTurnSpeed);

   GetComponent<Rigidbody>().AddRelativeForce(0, 0, aiSpeed);

   if(relativeWaypointPosition.sqrMagnitude < 15.0f)
   {
        currentWaypoint++;

        if(currentWaypoint >= waypoints.Count)
        {
                 currentWaypoint = 0;
         }
     }
```

```
        currentSpeed = Mathf.Abs(transform.
            InverseTransformDirection(GetComponent<Rigidbody>().
            velocity).z);

        float maxAngularDrag = 2.5f;
        float currentAngularDrag = 1.0f;
        float aDragLerpTime = currentSpeed * 0.1f;

        float maxDrag = 1.0f;
        float currentDrag = 3.5f;
        float dragLerpTime = currentSpeed * 0.1f;

        float myAngularDrag = Mathf.Lerp(currentAngularDrag,
              maxAngularDrag, aDragLerpTime);
        float myDrag = Mathf.Lerp(currentDrag, maxDrag,
              dragLerpTime);

        GetComponent<Rigidbody>().angularDrag = myAngularDrag;
        GetComponent<Rigidbody>().drag = myDrag;
    }

    void TurnLeft()
    {
        //turning left function here
    }

    void TurnRight()
    {
        //turning right function here
    }
```

我们添加了4个新的静态变量到代码中：isBlocked、isBlockedFront、isBlockedRight和isBlockedLeft。这些变量用来检查汽车前方的道路是否有障碍物。该车将持续跟随寻路点移动直到车辆需要左转或右转以避开障碍。为了实现这个需求，我们需要在车前方至少添加3个感应器。当这些感应器探测到障碍物时，会给AI驾驶员提供相关信息，而AI驾驶员就会根据这些信息来做出最佳决策：



就如我们在上图中看到的，车辆现在被附加了3个感应器。在这个例子中，右感应器将报告它被障碍物阻塞，司机将左转直到右侧不再被阻挡。一旦3个感应器都没有报告收到前方路线上的阻挡信息，汽车将持续沿着寻路点前进。假如我们发现驾驶员没有准确识别障碍物，那么建议增加传感器的数量以覆盖前方区域。

现在让我们转到MOBA例子中的小兵。在这里，我们需要创建一个不同的方法，因为小兵在遇到敌方角色之前会一直走向下一个寻路点，只是这一次我们不想让它们遇到东西就走开。相反，我们想要它们走向敌方角色。

为了实现这个，我们会在角色身上添加一个圆形或球形的碰撞器用来检测碰撞。如果有什么东西触发了这个区域，角色将停止朝寻路点移动，并且以触发到碰撞器的角色的位置作为目标：

```
public float speed;
public int health;
public float speedTurn;

public bool Team1;
public bool Team2;
public bool Top;
public bool Middle;
public bool Bottom;

private Transform target;
private int wavepointIndex = 0;

static Transform heroTarget;
static bool heroTriggered;
```

```
void Start ()
{
        if(Team1 == true)
```

在添加了上面的变量之后，我们可以转到在第一帧被调用的Start方法：

```
{
   if(Top == true)
   {
         target = 1_Top.1_Top[0];
    }

     if(Middle == true)
     {
           target = 1_Middle.1_Middle[0];
     }

      if(Bottom == true)
      {
          target = 1_Bottom.1_Top[0];
       }
}

 if(Team2 == true)
 {
   if(Top == true)
   {
     target = 2_Top.2_Top[0];
   }

    if(Middle == true)
    {
       target = 2_Middle.2_Middle[0];
    }

     if(Bottom == true)
     {
         target = 2_Bottom.2_Top[0];
      }
 }
 speed = 10f;
 speedTurn = 0.2f;
}
```

接下来是每帧被调用的Update方法：

```
    void Update ()
    {
       Vector3 dir = target.position - transform.position;
       transform.Translate(dir.normalized * speed * Time.deltaTime,
          Space.World);

       if(Vector3.Distance(transform.position, target.position) <=
             0.4f && heroTriggered == false)
        {
            GetNextWaypoint();
         }

         if(heroTriggered == true)
         {
              GetHeroWaypoint();
          }

          Vector3 newDir = Vector3.RotateTowards(transform.




                forward, dir, speedTurn, 0.0F);

       transform.rotation = Quaternion.LookRotation(newDir);
}
```

GetNextWaypoint方法用来获取角色需要跟随的下一个寻路点的信息：

```
void GetNextWaypoint()
{
   if(Team1 == true)
   {
      if(Top == true)
      {
         if(wavepointIndex >= 1_Top.1_Top.Length - 1)
         {
              Destroy(gameObject);
              return;
          }

          wavepointIndex++;
          target = 1_Top.1_Top[wavepointIndex];
        }

        if(Middle == true)
        {
      if(wavepointIndex >= 1_Middle.1_Middle.Length - 1)
      {
        Destroy(gameObject);
        return;
       }

       wavepointIndex++;
       target = 1_Middle.1_Middle[wavepointIndex];
    }

    if(Bottom == true)
    {
       if(wavepointIndex >= 1_Bottom.1_Bottom.Length - 1)
       {
          Destroy(gameObject);
          return;
       }

        wavepointIndex++;
        target = 1_Bottom.1_Bottom[wavepointIndex];
      }
}

if(Team2 == true)
{
   if(Top == true)
   {
      if(wavepointIndex >= 2_Top.2_Top.Length - 1)
      {
         Destroy(gameObject);
         return;
      }

      wavepointIndex++;
```

```
            target = 2_Top.2_Top[wavepointIndex];
        }

        if(Middle == true)
        {
            if(wavepointIndex >= 2_Middle.2_Middle.Length - 1)
            {
               Destroy(gameObject);
               return;
             }

             wavepointIndex++;
             target = 2_Middle.2_Middle[wavepointIndex];
          }
                if(Bottom == true)
                {
                   if(wavepointIndex >= 2_Bottom.2_Bottom.Length - 1)
                    {
                       Destroy(gameObject);
                       return;
                     }

                     wavepointIndex++;
                     target = 2_Bottom.2_Bottom[wavepointIndex];
                  }
              }
    }
```

在GetHeroWaypoint方法中，我们可以添加当角色在跟随敌方英雄时应当怎么做，比

```
void GetHeroWaypoint()
{
     target = heroTarget.transform;
}
```

如攻击或者其他功能：

我们给角色添加了一个球形碰撞器，如有其他英雄触发这个碰撞器，那么AI角色会收到那个英雄的信息。如果没有英雄触发角色的碰撞器，角色会继续跟着寻路点，否则角色会把注意力集中到英雄上面，以英雄作为目标点。

在这个例子中我们学习了MOBA游戏中AI移动的核心特性，现在我们可以重新实现这个流行的游戏类型了。从本章开始我们将会创建从简单到复杂的导航系统，并使用它们来让我们的AI角色在游戏中更加活跃，即使目标正在移动也能一直跟随目标。

[1] 下面是之前塔防游戏的代码。——译者注

6.2 总结

本章我们介绍了点到点移动，这种方法在今天被广泛应用于许多游戏，我们可以修改我们创建的代码以便在任何游戏中都能很好地适配。在这一点上，我们能够重新创造许多流行的游戏，给它们加上自己的个人想法。在下一章中，我们将继续讨论移动问题，但我们将集中在一个较为进阶的算法，叫做Theta算法。那将是我们在本章所学到的东西的延续，我们将能够创造一个角色AI，在没有任何事先指定的信息和位置的情况下，它也能够找到最佳的路径来到达目的地。

第7章　高级寻路

在这一章，我们会来看一些有关高级寻路的方法，这些方法可用于非常多的游戏中。这一章的主要目标是学习创建高级AI的基本要素，即分析地图并处理所有必要的信息来找到最佳路径。高级寻路算法可以在许多流行的游戏中看到，其中的AI角色需要实时地选择最佳路径，我们还将分析一些非常流行的例子，并看看如何能够再现同样的效果。

7.1 简单寻路与高级寻路

前一章探讨过，AI角色利用寻路来探索在地图中正确移动的方法。不论是简单还是复杂的寻路系统都非常有用，选择用哪种寻路系统依赖于游戏本身。在之前的章节中，一个简单的寻路系统足以完成我们需要的任务，但在其他情况下，特别是当AI角色需要实现很高的复杂度和真实感时，需要一个不同的方法。

在讨论任何有关创建寻路系统的方法之前，先探索一下为什么需要使用它，在什么情况下需要升级AI角色，使其更智能、更有意识。在先前的例子中，能够看到简单的寻路方法存在一些局限。理解这些简单寻路系统中的局限性，有助于让我们认识到创建一个更复杂的系统时将面临的问题和挑战。因此学习如何搭建一个简单的寻路系统是一个很好的开始，接着再一点一点地向复杂的寻路系统进化。因为游戏本身也是伴随着这些技术的进化而进化的，所以第一个例子将是一个老游戏，然后我们会看到这个游戏是如何进化的，特别是在AI寻路方面。

开放世界大地图现在非常常见，许多不同类型的游戏都使用这种技术来创造丰富的体验，但一开始并不是这样。以初代的《侠盗猎车手》（GTA）为例。分析在地图上行驶的汽车，可以看出它们并没有复杂的系统，这些汽车固定在预先指定的路径或者环形路线上。当然，这里的AI寻路系统在当时看来还是非常先进的，甚至放在现在来看也还不错。我们并不会对这些AI角色感到失望，对这款游戏来说它们表现得非常出色。

这些AI驾驶员总是沿着指定的路线移动，一旦玩家挡住了它们的路其就会停下来。这表明在每辆车前面都有一个碰撞检测器，告诉它们是否有东西挡住了道路。如果有东西挡住了路它们就会立即停下，直到道路通畅才会继续行驶。通过这个例子可以看出AI驾驶员具有某种寻路系统，它还不能很好地处理这样的特殊情况。当这类事故发生时，程序员选择让AI驾驶员停下来。

上面讨论的是司机在无法继续前进时就会停下来的问题，在后续作品中其被成功解决而且还成为最强有力的特性。在GTA游戏系列中很多东西都在进化，毫无疑问AI是其中之一。它们改进了AI驾驶员，让它们能够意识到周围的环境和发生的事件。让我们来分析一下《GTA圣安地列斯》，现在也可以在手机上玩到了。在这个游戏中，如果玩家的车在AI驾驶员面前停下来，将得到完全不同的结果。根据AI驾驶员不同的个性，这些结果会产生不同的反应。举个例子，它们中的一些会简单地鸣笛并等待一会，如果玩家一直挡着它的路，那么它会绕过玩家。有些还会有另一些更具侵略性的反应，比如下车找玩家单挑。

如果AI驾驶员听到了枪声进而意识到当前所处的环境很危险，它们会加快速度选择最快路线以离开当前位置。类似这样的行为证明了：AI角色可以将更加复杂且精准的寻路系统与可能性图相结合，最终它们选择的路线会受周围情况的影响。

如我们看到的，这些AI驾驶员已经比第一个游戏中的要引人注目得多了。前一章介绍了如何创建一个简单的寻路系统，与初代GTA游戏很接近。现在，将深入研究如何创建一个可以脱离任何意外情况的AI角色。

这属于那些至今还没有完美解决的问题之一，许多开发者仍然在尝试使用新的方式创建AI角色，让它们在陷入同样的困境时的行为像人类一样。一些公司已经能够做到和Rockstar Games的GTA系列非常接近了，因此选择从这个例子开始讲起。

7.2 A*搜索算法

一些不可预测的情况通常会导致编写代码的绝大部分时间花费在不断扩展的可能性上。因此，我们需要思考新的方式来创建更好的寻路系统：人物可以实时分析周围环境，选择最佳路径。有一种非常流行的实现这个效果的方式是使用A*算法，它允许AI角色不断搜索最佳路径，而不用手动设置需要跟踪的点。

A*搜索算法是一种被广泛使用的搜索算法，可以用来解决很多问题，寻路只是其中之一。由于这种算法具有开销一致性以及启发式特性，用这种算法来解决寻路问题是很常见的。A*搜索算法在到达目的地的途中不断检查地图，帮助角色决定是否可以使用某一个位置，同时尝试到达预期的目的地。

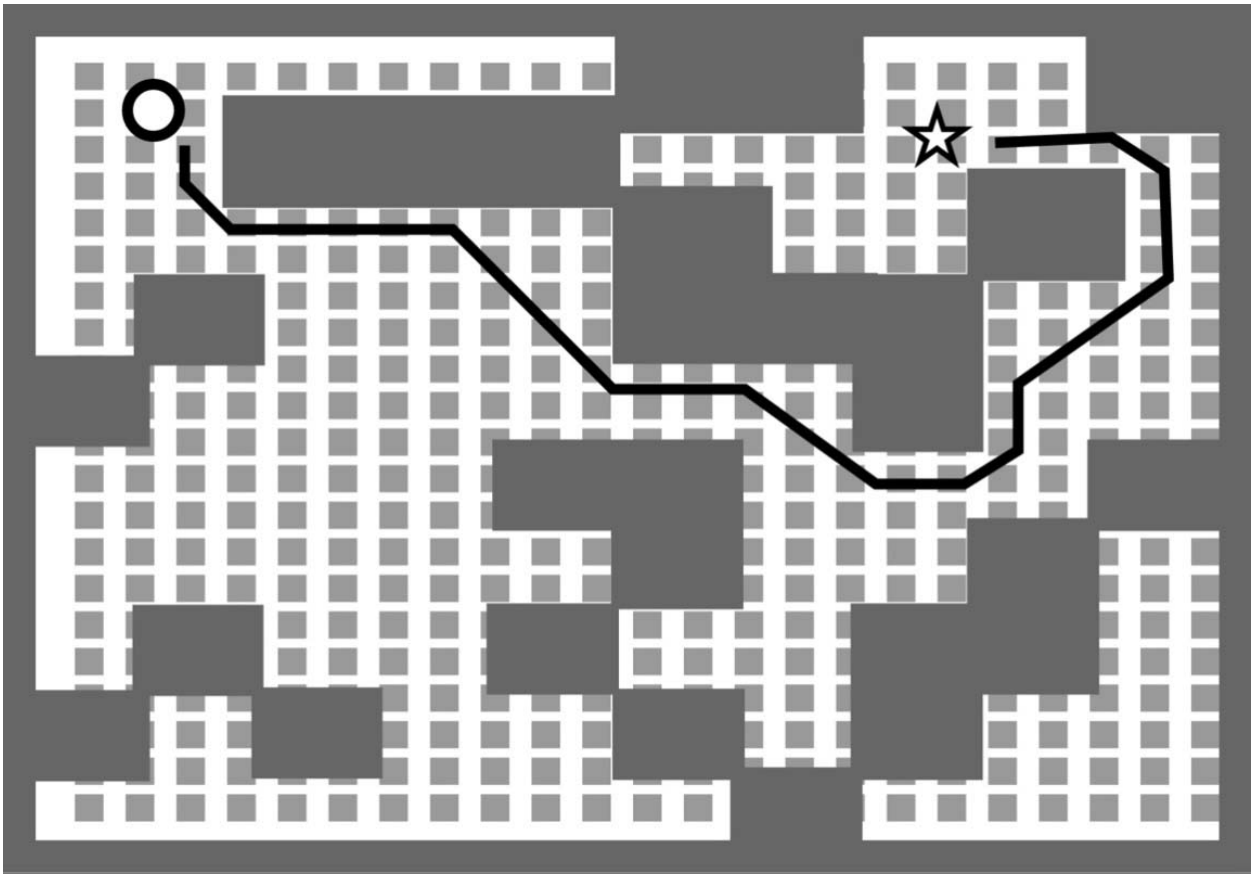
这个算法如何工作

游戏中的地图和场景在使用A*算法前需要提前分析和准备。环境与其包含的所有资源会被当作一张图来处理。这意味着地图将会分为不同的点和位置，我们称之为节点。这些节点用来记录整个搜索的过程。在记录地图位置时，每个节点都有其自身的属性，包括?tness、goal和heuristic，通常将它们写作f、g和h。通过综合考虑这三个属性可以看出当前节点和路径的优劣程度。

路径上的每个节点都被赋予了不同的值。这些值通常也表示两个节点间的距离。不过两个节点间的值也不一定是距离。这些值也可以是时间，这样就可以找到最快的路径而不是最短的路径。A*算法使用两个列表——开启列表和关闭列表。开启列表里包含需要检查的所有节点。还可以用标记数组来检查某个节点是否在开启列表或关闭列表中。

这意味着AI角色将会不断搜索最佳节点，以得到最快或最近的结果。如下图所示，地图是预先分析过的，可行走区域用小的灰色方块来表示，大块的灰色方块用来表示不可行走区域。AI角色用圆环表示，目的地用星星表示。如果中间遇到了不可行走区域，角色会迅速绕开然后再朝目的地行进。

正如我们看到的，寻路算法的理论与我们之前做的非常相似，角色一步一步地跟着这些路径点移动，直到抵达最终目的地。其中主要的不同之处是A*算法是由AI自动生成的路径点，在开发者要处理复杂的大型场景时，这会是非常好的选择。



A*算法的缺点

在开发各种各样的游戏时，A*算法并不总是完美的，需要时刻留意这点。由于AI角色在持续不断地搜索最佳路径，因此这会给CPU带来很多工作负担。现在平板电脑和移动设备非常流行，为这些平台开发游戏时需要特别注意CPU和GPU的使用。A*寻路算法的缺陷也恰恰在这里。

而且硬件的限制并不是唯一的缺点。让AI自行工作不受人工干预时，非常容易产生bug。这也是在偏好开放式世界的现代游戏中总会遇到很多bug和奇怪的AI行为的原因之一，因为在一个巨大的游戏区域中，要限制所有的可能性是非常困难的。

“最新演示中出现了bug，这对开放式游戏来说是很自然的。”

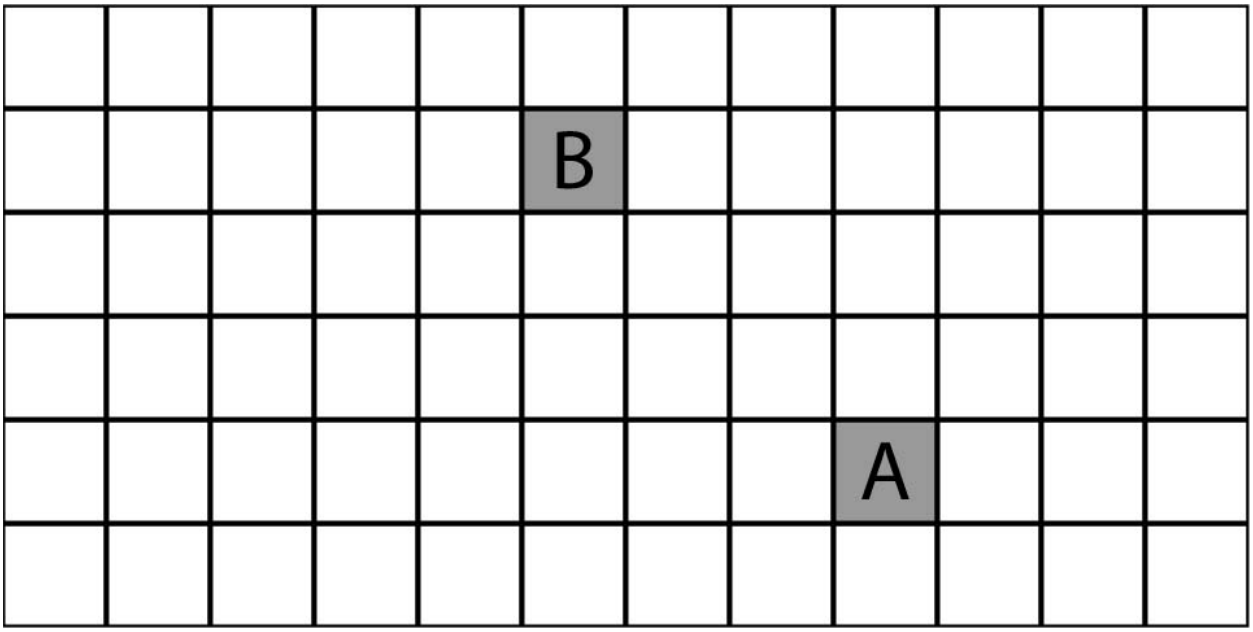
——《最终幻想15》导演

《最终幻想15》的导演对这个问题发表了评论，指出在每一个开放世界的游戏中都必定会有bug。这也完美地总结了在开放世界的游戏中，使用A*算法的AI寻路是非常流行且有效的方法，但它并不是完美的方法而且注定会有bug。

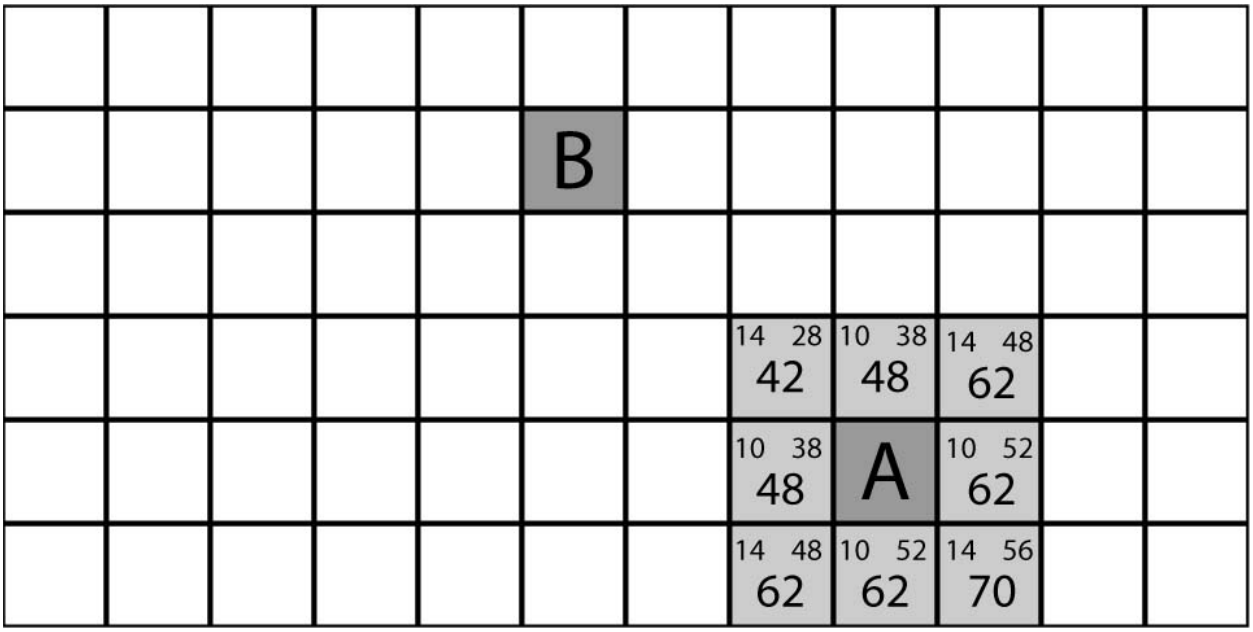
现在我们已经基本理解了A*算法及其优劣，接下来实践一下。

从A直接到B（无障碍物的情况）

我们从一个简单的例子开始，在两点间没有任何障碍物。这有助于可视化地理解算法是如何找到最佳路线的。后面还会添加一些障碍物，并观察算法是如何在有障碍物的情况下找到最佳路线的。



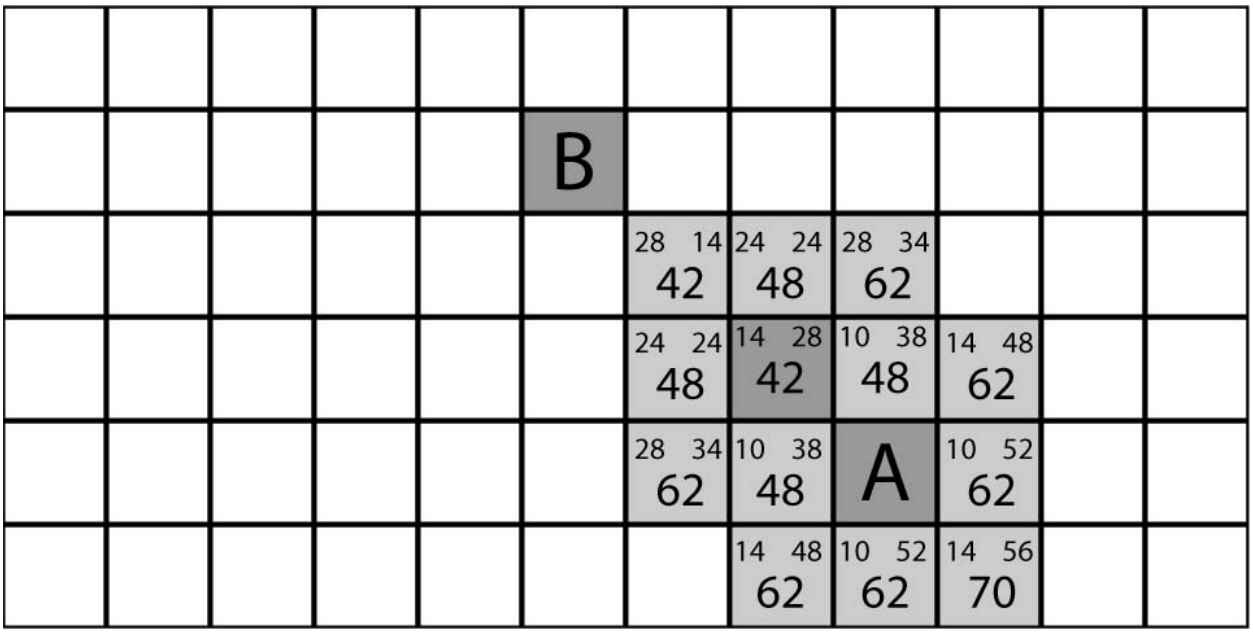
在这个网格中，有两个点，A是起点，B是终点，我们想要找到这两点间的最短路线。



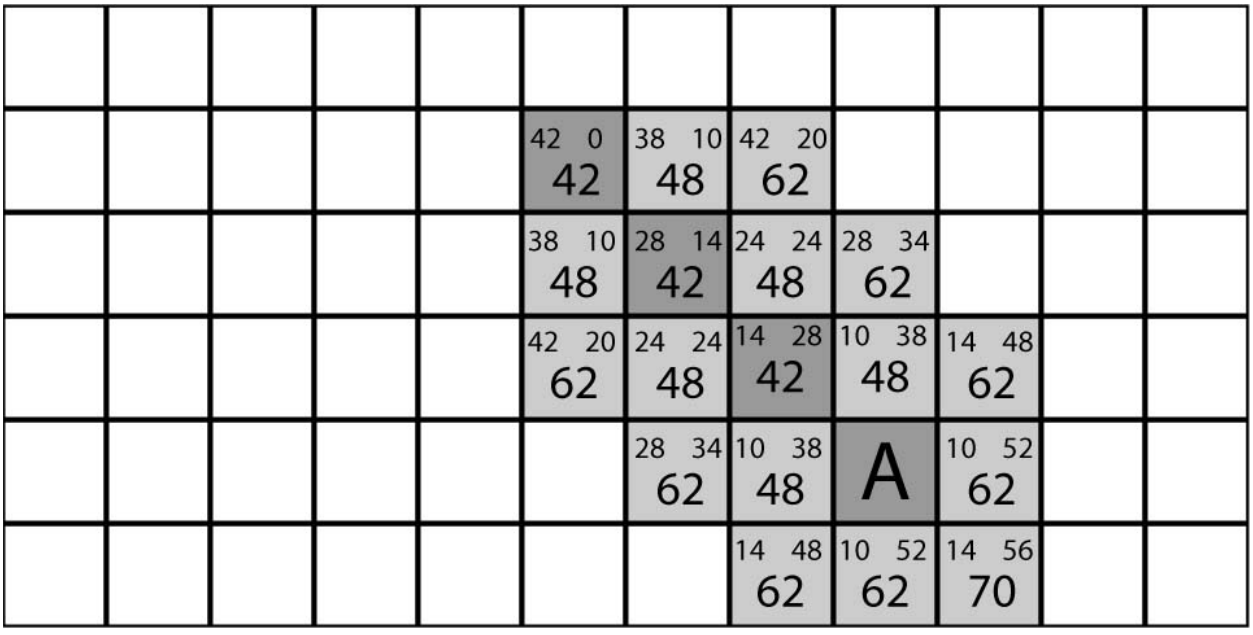
解决这个问题需要使用A*算法，下面看看它是怎么帮我们找到最短路线的。

这个算法每一步都会进行计算以找到最短路径。为了计算最短路径，算法使用两个节点G和H。G表示从起点A移动到当前方格的距离有多远，H表示从当前方格移动到终点B的预计距离有多远。将G和H相加就得到了F，F用来得出最佳路径[1]。

如下图所示，最小的值是42，因此移动到42的位置，并且再一次计算每个可用的节点[2]。



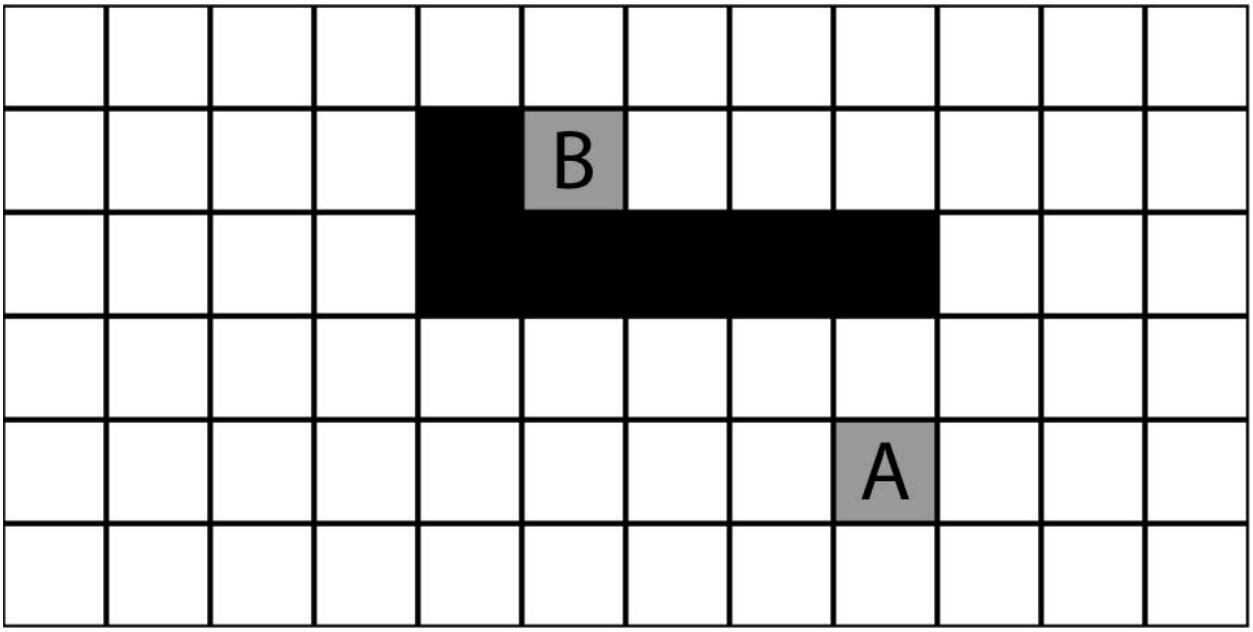
再一次，算法会计算出从当前所处位置开始的最佳选择。当越来越靠近B时，节点里H的值会越来越小，而G的值会越来越大，这正是想要的结果。从当前方格周围所有可能的方格来看，42依然是最小的，也就是说，42依然是最好的选择。因此，继续朝这个方向前进。



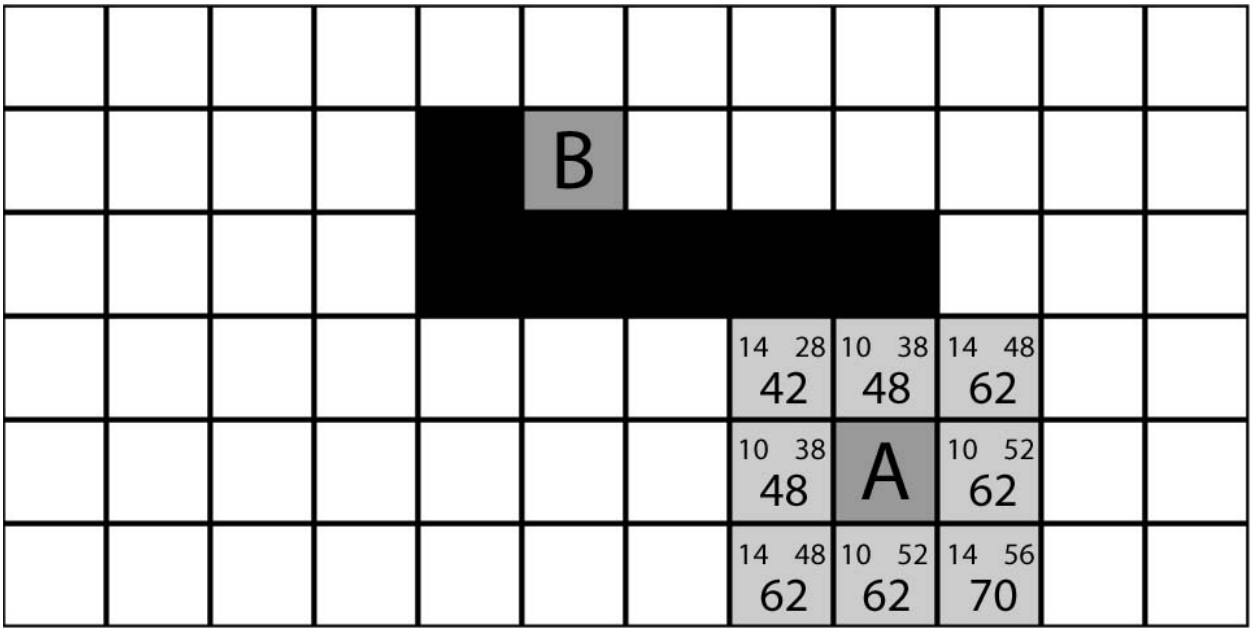
最终到达终点B。一旦算法找到某个节点的H值为0时，也就意味着我们已经到达了终点，不再需要继续搜索了。

从A点到B点（有障碍物的情况下）

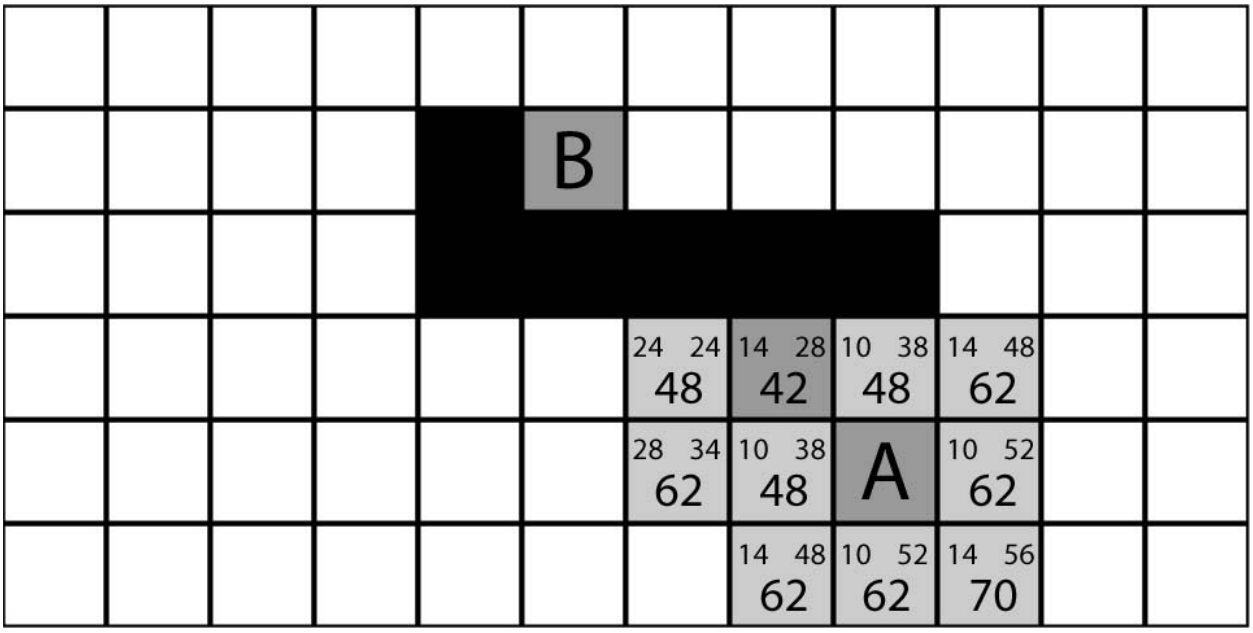
以上正是A*算法的工作原理，它从一个点到另一个点不断地评估最佳选项，选择最短路径直到抵达目的地。之前的例子过于简单，现在来看看当地图中有障碍物时算法如何工作。



在与之前相同的地图中，加入了一些黑色的方块来代表这些区域不允许通过。现在，事情开始变得有趣了，因为当猜测H的值时，它有可能会因为黑色区域不能通过，而变得无法准确猜测。下面计算一下最佳选项是什么：



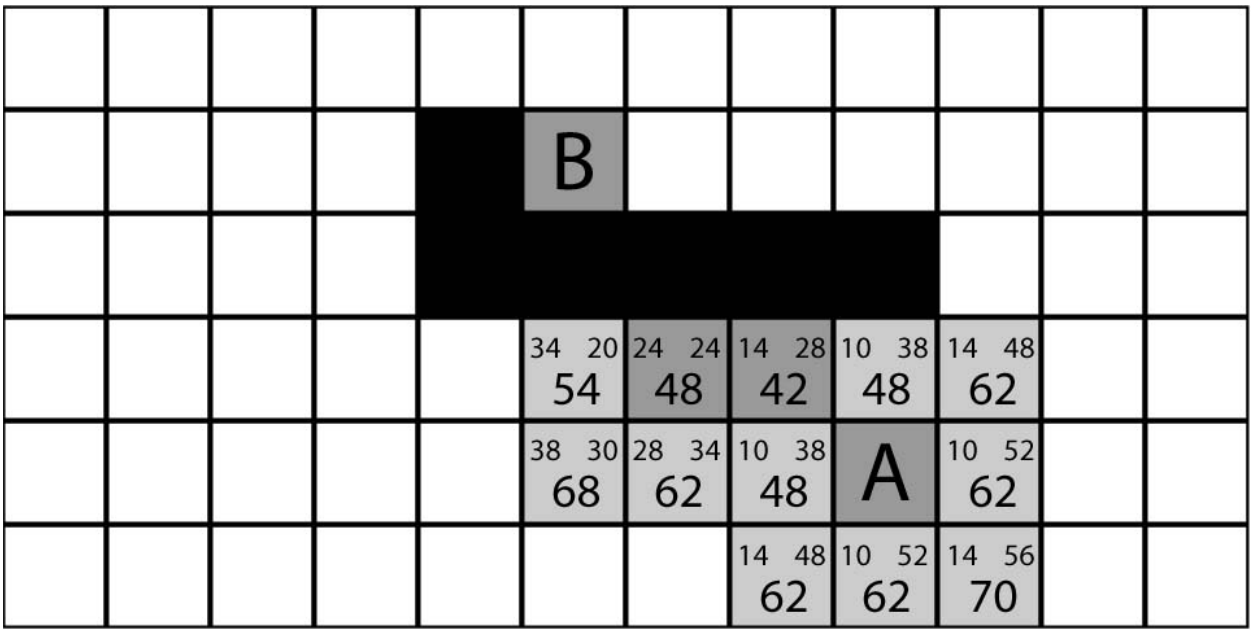
第一步，得到的结果与第一次测试的结果完全相同，这是正常的，因为围绕A点的8个



方块中没有一个是黑色方块。这次，也移动到F值最小的42的格子上去。

现在已经移动了一步，并计算了从那一点开始的所有选项，现在的处境就非常有趣了。此刻，有了3个最佳选项[3]，需要选择其一。要找到可以到达B点的最短路线，由于这3个F值都是相同的，因此只需要根据节点的H值来做出决定，这里H表示当前位置与B点位置之

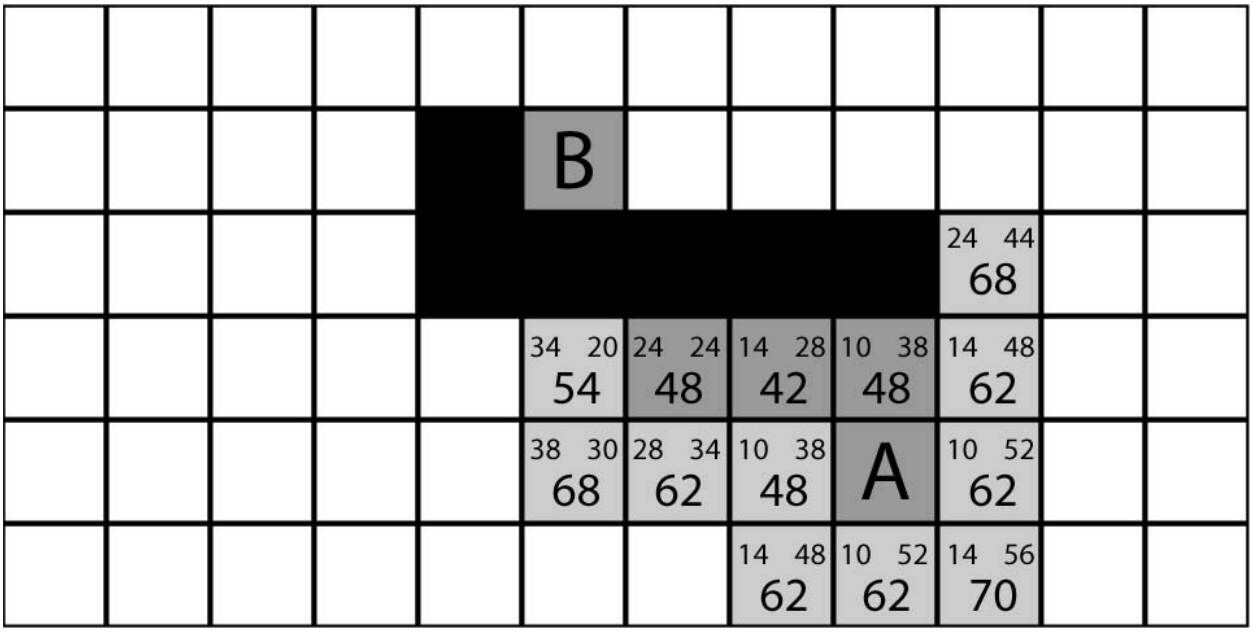
间的距离远近。有两个方块的H值为38，只有一个方块的H值为24，则选择那个H值最小的方块。接下来，朝这个方向移动，可以看到，离终点又近了一步。



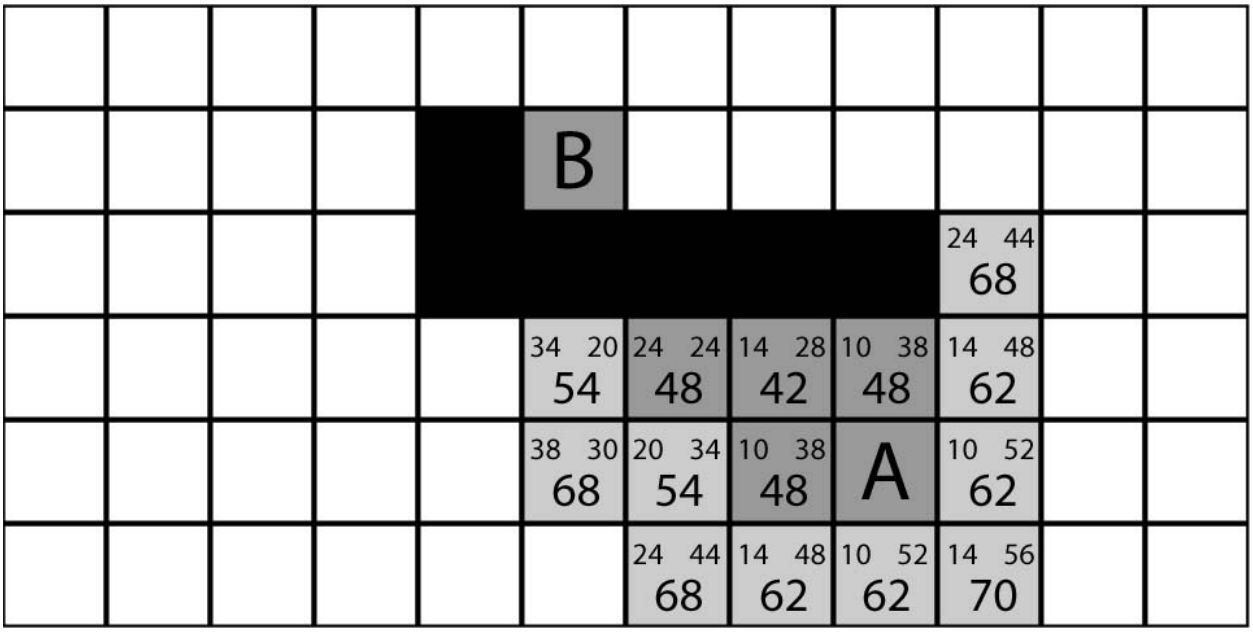
现在注意，方块上代表最短路线的F值越来越大。由于在地图中加入了黑色方块导致需要绕路，这就增加了最终路径的长度。这就是AI感知墙壁的方式，它们知道离最后的目的地很近，但是要到达那里又不能穿过墙壁。因此它们需要绕过墙，直到遇到开着的门或者其他类似的东西。

现在，最小的值是在另一个方向，这意味着需要返回之前的方块才能找到最佳路线。这是此算法中非常有用的一方面——如果角色在四处行走的同时寻找最佳路径，我们会得到一个更人性化的效果。看起来像是它正在寻找正确的路线，而不是一开始就知道最佳路线。另一方面，角色也可以被编写为在开始行动前就做完这些计算，这样会使角色一直都走在正确的路线上。这两种方法都是有用的，可以根据不同的游戏选择最适合的方式。

接下来继续寻路，我们总是需要不断地选择最小的值，因此这时需要返回之前的路线，并在两个最小的值48中选择一个。两者都有相同的g和h值，所以找出哪一个是最佳路径的唯一方法是随机选择其中一个点，或者预先做其他处理。随机选择一个来看看所产生的效果。

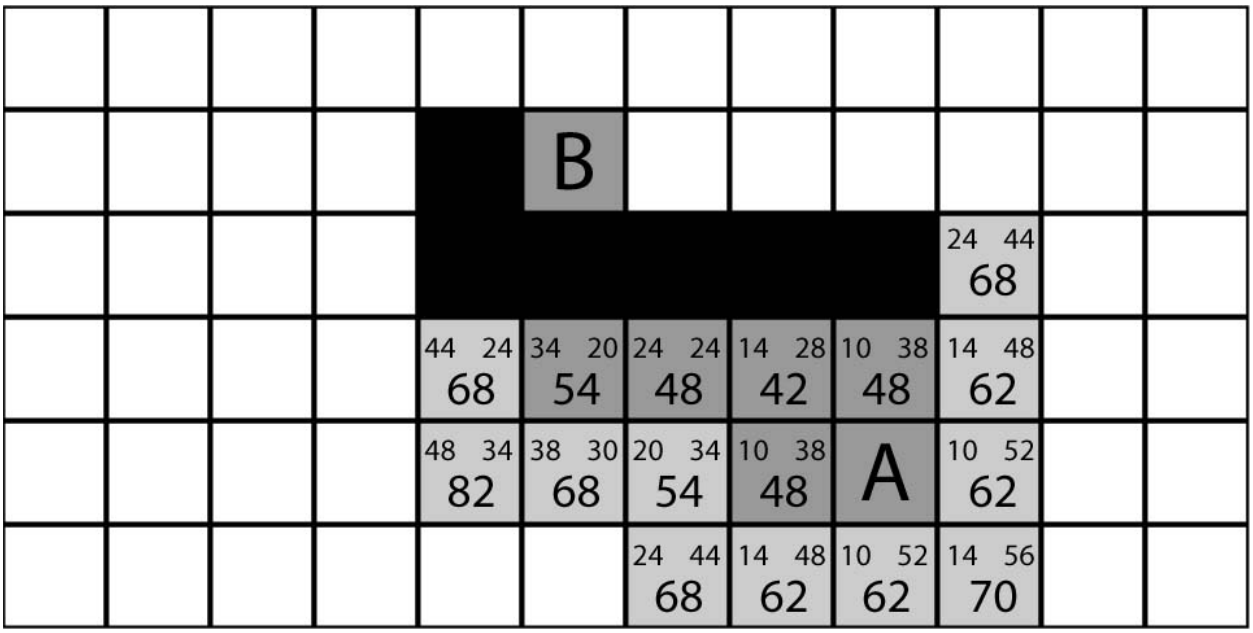


在选择了其中一个之后，我们却发现值更大了，因此需要回去计算另一个值，看看它后面是否有一个较小的值。因为我们能看到地图也已经知道B点的位置，所以可以看出继续算下去的话，最小值实际上比刚刚出现的68还要大。不过在实际计算时我们并不知道B点在哪里，所以仍然需要检查48这个值。看目的地是否在这个位置附近。这就是AI角色在



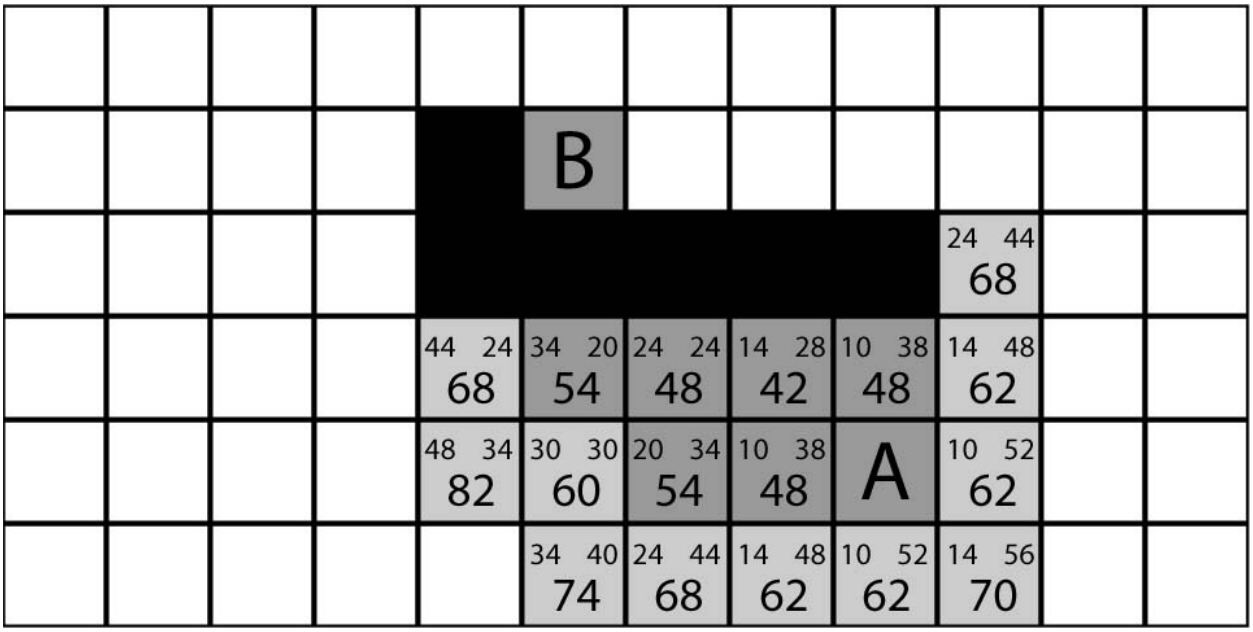
游戏中所做的，它将不断地检查最小的F值。

在选择了新的位置后，可以看到并没有出现更好的路线，需要继续寻找，这种情况下，要回到先前已经检查过但没有计算过结果的一个方格。这样又遇到了两个最小的F值，我们会选择其中H值最小的那个，即H值为20的方格。

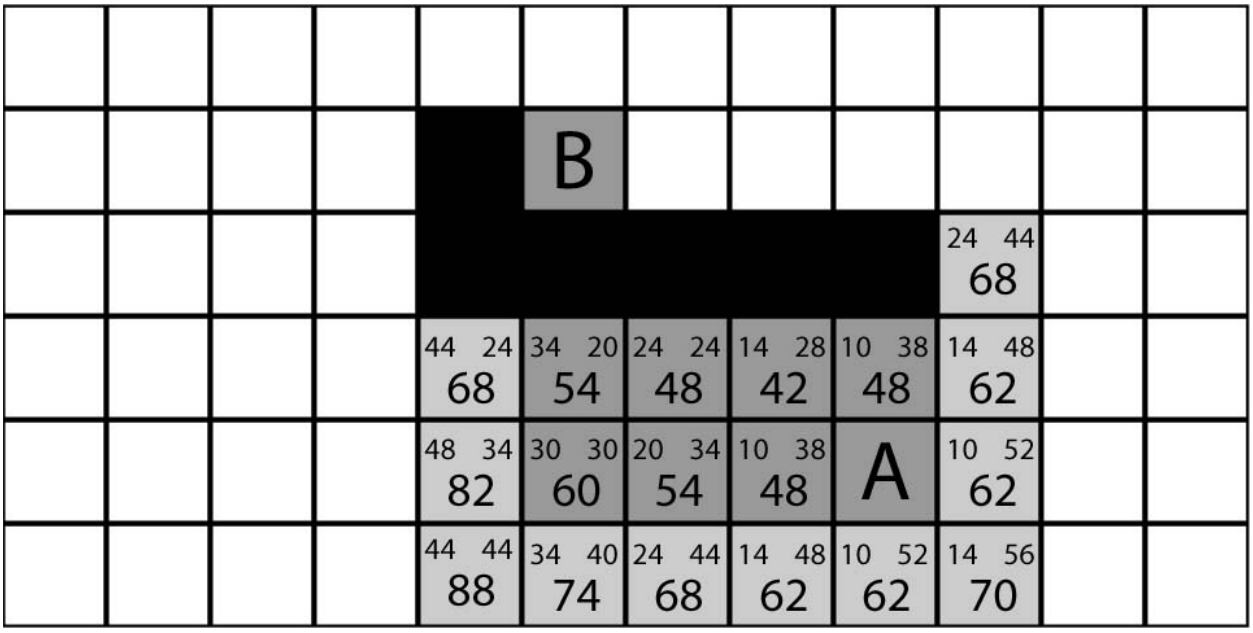


在计算新的可能性之后，注意，需要再次选择54，看看是否会发现更好的路线。这正是在对AI进行编程时，为了找到最短路径而采取的方法。这种计算是实时的，并且随着时间的推移它们会变得非常复杂。这个计算过程完全依赖于CPU硬件，因此它会耗费相当大的CPU运算资源。

现在选择54，因为其是这个地图上目前可选的最小值了。

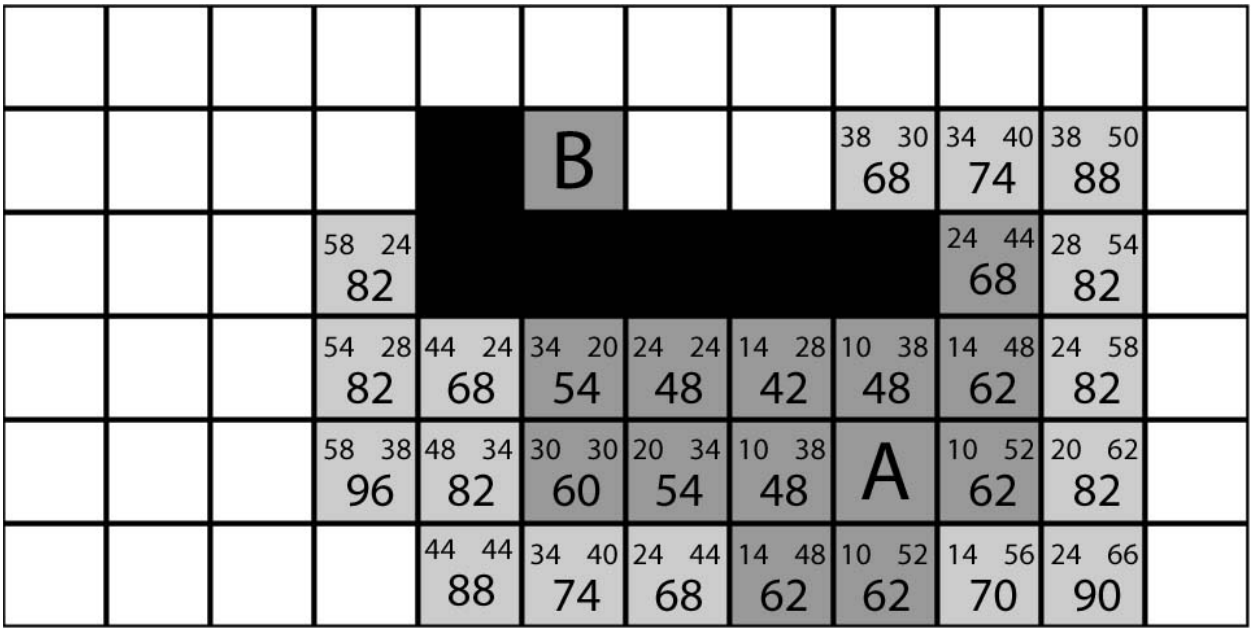


如果继续往下走，这些值就会越来越大，也意味着离目标越来越远了。如果我们是AI，

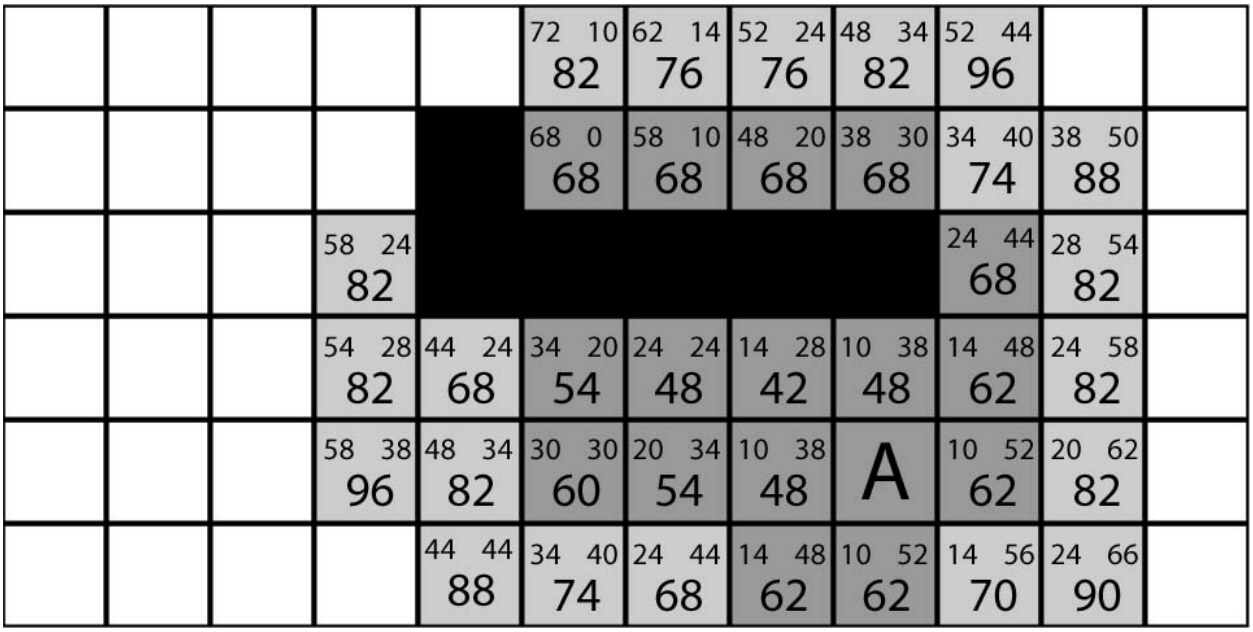


不知道最终目的地在上方，那么需要检查60。因为这是现在最小的值了。因此，计算结果。

现在，我们可以看到有很多相同的最小值，即62，所以需要探索所有这些方格并继续计算，直到找到正确的路径。为了实现这个例子的目标，将检查地图上每一个最小的数字。



在探索了所有的可能性之后，可以看到我们正在接近最终的目的地。在这一点上，可



用的最小值是68，这之后就很容易到达目的地了。

最终，我们到达了目的地B。这些就是A*算法的可视化效果，其中较深的灰色区域表示计算机处理过的位置，较淡的灰色区域表示曾经计算过数值的区域。[4]

电脑可以实时计算最佳路线，或者开发者也可以预先计算出最佳路线，这样在游戏开始时AI会自动知晓最佳路线，可以少占用一些CPU资源。

为了解释这种方法在代码中是如何运行的，我们将使用伪代码来展示这个例子。通过这种方式，我们可以从头至尾地理解如何在任意一种编程语言中实现搜索方法，以及如何

```
OPEN // the set of nodes to be evaluated
CLOSED // the set of nodes already evaluated

Add the start node to OPEN

loop
   current = node in OPEN with the lowest f_cost
   remove current from OPEN
   add current to CLOSED

   if current is the target node // path has been found
         return

   foreach neighbor of the current node
         if neighbor is not traversable or neighbor is in CLOSED
               skip to the next neighbor

if new path to neighbor is shorter OR neighbor is not in OPEN
   set f_cost of neighbor
   set parent of neighbor to current
   if neighbor is not in OPEN
         add neighbor to OPEN
```

调整它们：

让我们来分析一下这个例子中的每一行代码。网格地图里的格子划分为了两种类型：OPEN和CLOSED。OPEN的方块是我们已经探索过的方块，用深灰色表示。CLOSED的方块是尚未探索的方块，用白色表示。这么做会帮助AI区分已经彻底探索和待探索的方块。以帮助

Add the start node to OPEN

AI找到最佳路线：

接着我们将起点的方块分配给OPEN，这一步设置了探索的起点，它将自动开始计算从该位置开始的最佳选项：

```
loop
   current = node in OPEN with the lowest f_cost
   remove current from OPEN
   add current to CLOSED
```

然后我们需要创建一个循环，在我们的循环中，有一个叫做current的临时变量；这相当于在OPEN列表里具有最低F成本的节点。然后它将从OPEN列表中移除，并添加到关闭列表中[5]：

```
if current is the target node // path has been found
        return
```

```
foreach neighbor of the current node
        if neighbor is not traversable or neighbor is in CLOSED
             skip to the next neighbor
```

如果当前节点是目的地节点，那么我们认为目的地已经找到，就可以退出循环了。

否则，我们必须检查当前节点的每个相邻节点。如果其中某个相邻节点是不能通过的，或者它是在CLOSED列表中的，代码便跳到下一个相邻节点。这一部分设置了所有可以移动的位置，同时也告诉了AI不要考虑先前已经探索过的位置。

```
if new path to neighbor is shorter OR neighbor is not in OPEN
   set f_cost of neighbor
   set parent of neighbor to current
   if neighbor is not in OPEN
        add neighbor to OPEN
```

如果没有跳过的话，那么我们可以继续检查几件事情。如果到相邻方块的新路径比旧路径短，或者相邻方块并不在OPEN列表上，那么我们通过计算g_cost和h_cost来设置相邻节点的f cost。设置相邻节点的父节点为当前节点。我们看到新的可能块有来自当前块的子节点，因此我们可以跟踪正在执行的步骤。最后如果相邻节点不在OPEN列表中，我们将其添加进去。

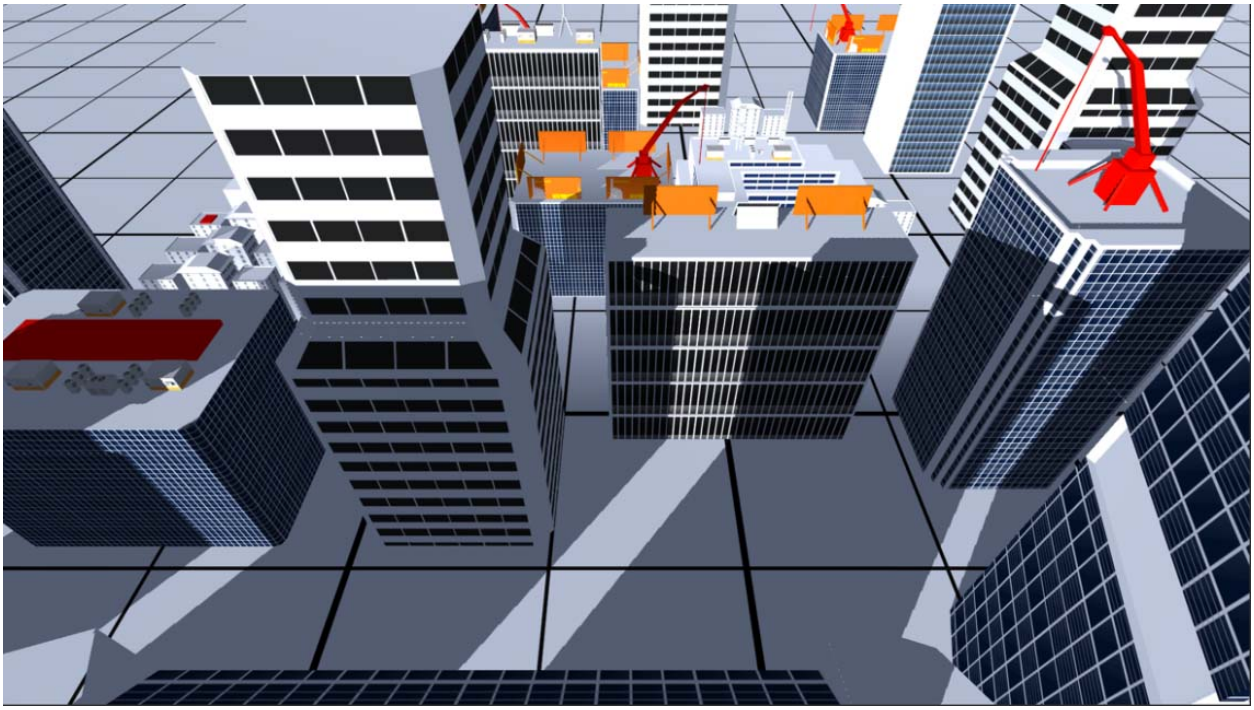
循环这个过程，代码会持续搜索最佳选项，并慢慢向目标节点靠近。

刚才所学的原理同样可以在《侠盗猎车5》的行人中找到。许多其他游戏也使用了这个方法，不过我想用这个游戏来作为展示两种寻路系统的例子。如果我们把这个系统应用到AI警察身上，以让它搜索和找到玩家，在实际游戏中差不多会得到相同的结果。

它不仅仅要寻找最终的目的地，这只是代码功能的一小部分，我们还看到一个AI角色在避开墙壁的同时，一步一步地接近玩家的位置。除此之外，有必要在AI代码中添加更多的内容，让角色知道在可能发生的多种情况下应该做什么，比方说在路径被水、楼梯、移动的汽车，或是其他什么东西阻挡的情况。

生成网格中的节点

现在将把我们学到的所有东西应用到一个实践练习中。让我们从创建或者导入场景到编辑器里开始。



在这个例子中，建筑是不可以走过去的物体，它也可以是任何其他东西，然后我们需要将这些物体与地面区分开。要做到这一点，我们要给它们分配一个单独的层（layer），并将该层命名为unwalkable。

然后开始创建第一个类Node，从这个类开始写起：

```
public bool walkable;
 public Vector3 worldPosition; public Node(bool _walkable, Vector3
         _worldPos, int _gridX, int _gridY) {
         walkable = _walkable;
         worldPosition = _worldPos;
```

可以看到，节点有两个不同的状态，可步行或者不可步行，所以创建了一个布尔变量walkable。然后，我们需要知道这个节点在世界中表示的位置，因此我们创建了一个Vector3的变量worldPosition。现在，当创建一个节点时，我们需要一个构造函数来初始化这些值，所以我们创建了一个Node的构造函数，它会初始化关于节点的所有重要信息。

在创建这个类的基本部分之后，转到grid对象：

```
Node[,] grid;
public LayerMask unwalkableMask;
public Vector2 gridWorldSize;
publicfloatnodeRadius;
void OnDrawGizmos()
{
        Gizmos.DrawWireCube(transform.position,new
   Vector3(gridWorldSize.x,1,gridWorldSize.y));
}
```

首先，我们需要一个表示网格上的所有节点的二维数组，因此，这里创建了一个二维数组grid，元素类型是节点。接着我们创建一个Vector2的成员变量gridWorldSize，它定义了这个格子在世界中所占据的区域大小。我们还定义了一个？oat型变量nodeRadius，它表示了每个节点覆盖的空间大小。接着我们创建了一个LayerMask变量unwalkableMask，用来表示不可行走的层。

为了在游戏编辑器中可视化地看到我们创建的网格，我们使用了OnDrawGizmos函数；这一步不是必须的，只是为了调试方便：

```
public LayerMask unwalkableMask;
public Vector2 gridWorldSize;
public float nodeRadius;
Node[,] grid;

float nodeDiameter;
int gridSizeX, gridSizeY;

void Start() {
    nodeDiameter = nodeRadius*2;
    gridSizeX = Mathf.RoundToInt(gridWorldSize.x/nodeDiameter);
    gridSizeY = Mathf.RoundToInt(gridWorldSize.y/nodeDiameter);
    CreateGrid();
}

void CreateGrid(){
    grid = new Node[gridSizeX,gridSizeY];
    Vector3 worldBottomLeft = transform.position - Vector3.right *
gridWorldSize.x/2 - Vector3.forward * gridWorldSize.y/2;
}
```

接下来创建Start方法，我们会在其中添加一些基本的计算。需要弄清楚的主要问题是，我们可以在网格中容纳多少个节点。我们添加了一个？oat类型变量nodeDiameter，以及两个整型变量gridSizeX和gridSizeY。在Start方法内，我们将nodeRadius*2赋值给nodeDiameter，表示节点的直径。将gridWorldSize.x/nodeDiameter赋值给gridSizeX，同时，我们还对其使用Mathf.RoundToInt函数进行四舍五入得到了一个整数。这里gridSize

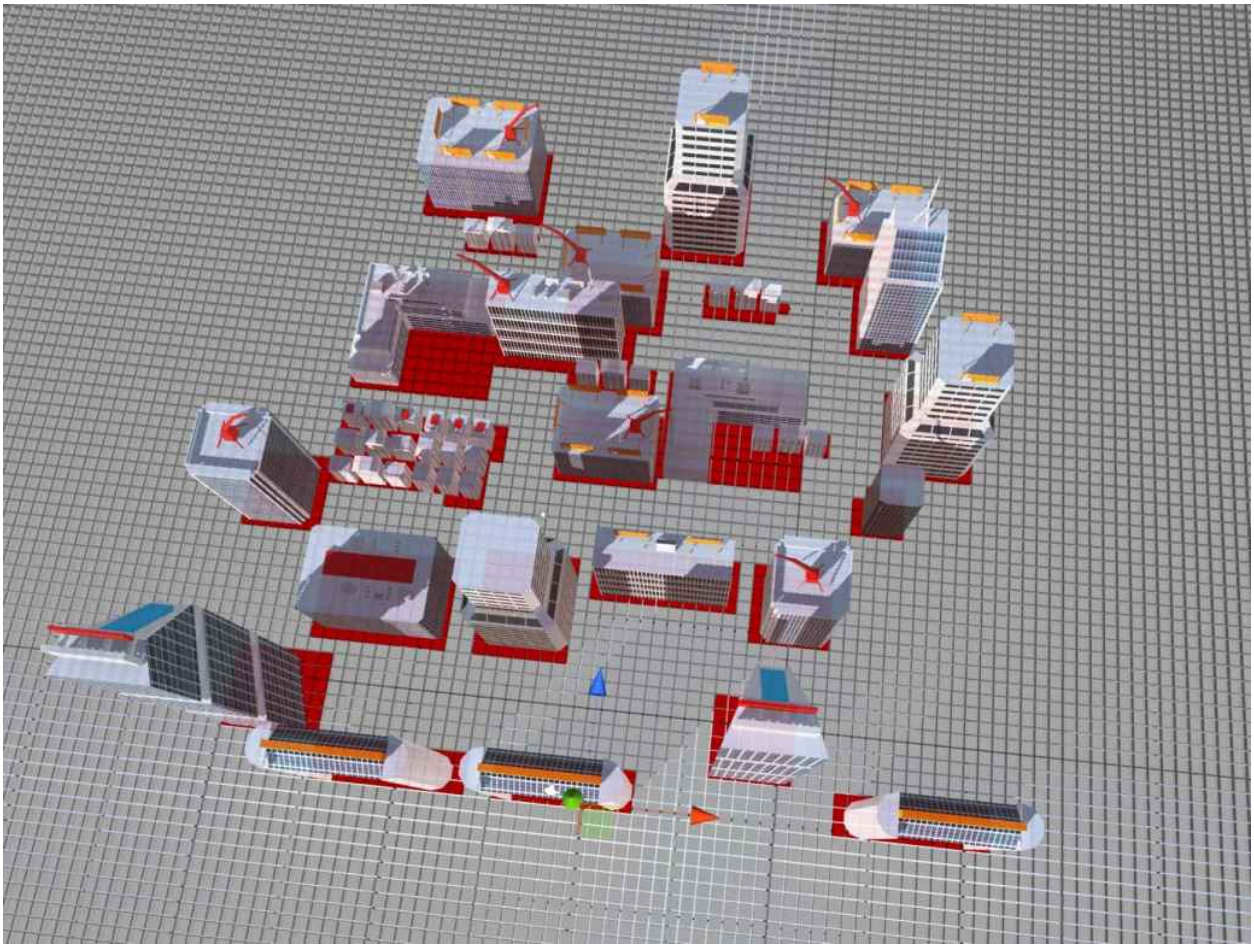
X就表示了gridWorldSize.x一共有多少个节点。X轴计算完后，我们用同样的方式计算Y轴。最后我们创建了一个新的方法CreateGrid（）：

```
public LayerMask unwalkableMask;
public Vector2 gridWorldSize;
public float nodeRadius;
Node[,] grid;

float nodeDiameter;
int gridSizeX, gridSizeY;

void Start(){
        nodeDiameter = nodeRadius*2;
        gridSizeX = Mathf.RoundToInt(gridWorldSize.x/nodeDiameter);
        gridSizeY = Mathf.RoundToInt(gridWorldSize.y/nodeDiameter);
        CreateGrid();
}

void CreateGrid()
{
        grid = new Node[gridSizeX,gridSizeY];
        Vector3 worldBottomLeft = transform.position - Vector3.right *
        gridWorldSize.x/2 - Vector3.forward * gridWorldSize.y/2;

        for (int x = 0; x < gridSizeX; x ++) {
            for (int y = 0; y < gridSizeY; y ++) {
                Vector3 worldPoint = worldBottomLeft + Vector3.right *
                (x * nodeDiameter + nodeRadius) + Vector3.forward * (y
                * nodeDiameter + nodeRadius);
                bool walkable = !(Physics.CheckSphere(worldPoint,
                                  nodeRadius,unwalkableMask));
                grid[x,y] = new Node(walkable,worldPoint);
            }
        }
}
```

我们在这里初始化了grid数组，grid=new Node[gridSizeX，gridSizeY]；。现在我们需要添加碰撞检测，来设定地图中的可行走区域与不可行走区域。为此，我们创建了一个循环，可以在前面演示的代码中看到。我们添加了一个Vector3的变量，用来表示地图的左下角，叫做worldBottomLeft。接着我们对每个方块使用Physics.CheckSphere来检测当前方块是否是可行走区域，并创建grid中的每一个节点。

```
    void OnDrawGizmos() {
            Gizmos.DrawWireCube(transform.position,new
            Vector3(gridWorldSize.x,1,gridWorldSize.y));


            if (grid != null) {



            foreach (Node n in grid) {
                Gizmos.color = (n.walkable)?Color.white:Color.red;
                Gizmos.DrawCube(n.worldPosition, Vector3.one *
                (nodeDiameter-.1f));
            }
        }
}
```

在开始测试这些代码之前，需要写一下OnDrawGizmos函数，让我们可以在地图上看到这些网格。为了让网格显示，使用红色和白色来标识方块，每一个方块的尺寸使用nodeDiameter来确定。白色方块表示可行走区域，红色方块表示不可行走区域。现在来测试看看：



结果看起来不错，现在我们有了一个可以自动分析地图并表示可行走区域的网格。完成此部分后，其余部分将更容易实现。在我们开始下一步之前，我们需要添加一个方法，这个方法可以告诉我们角色处在哪个节点中。现在我们在代码中添加这个方法NodeFromWorldPoint：

```
public LayerMask unwalkableMask;
public Vector2 gridWorldSize;
public float nodeRadius;
Node[,] grid;

float nodeDiameter;
int gridSizeX, gridSizeY;

void Start(){
        nodeDiameter = nodeRadius*2;
        gridSizeX = Mathf.RoundToInt(gridWorldSize.x/nodeDiameter);
        gridSizeY = Mathf.RoundToInt(gridWorldSize.y/nodeDiameter);
```

```
        CreateGrid();
}

void CreateGrid()
{
        grid = new Node[gridSizeX,gridSizeY];
        Vector3 worldBottomLeft = transform.position - Vector3.right *
              gridWorldSize.x/2 - Vector3.forward * gridWorldSize.y/2;

        for (int x = 0; x < gridSizeX; x ++) {
            for (int y = 0; y < gridSizeY; y ++) {
                Vector3 worldPoint = worldBottomLeft + Vector3.right *
                (x * nodeDiameter + nodeRadius) + Vector3.forward * (y
                * nodeDiameter + nodeRadius);
                bool walkable = !(Physics.CheckSphere(worldPoint,
                                  nodeRadius,unwalkableMask));
                grid[x,y] = new Node(walkable,worldPoint);
            }
        }
}
public Node NodeFromWorldPoint(Vector3 worldPosition) {
        float percentX = (worldPosition.x + gridWorldSize.x/2) /
        gridWorldSize.x;
        float percentY = (worldPosition.z + gridWorldSize.y/2) /
        gridWorldSize.y;
        percentX = Mathf.Clamp01(percentX);
        percentY = Mathf.Clamp01(percentY);

        int x = Mathf.RoundToInt((gridSizeX-1) * percentX);
    int y = Mathf.RoundToInt((gridSizeY-1) * percentY);
    return grid[x,y];
} void OnDrawGizmos() {
    Gizmos.DrawWireCube(transform.position,new
    Vector3(gridWorldSize.x,1,gridWorldSize.y));


    if (grid != null) {
        foreach (Node n in grid) {
            Gizmos.color = (n.walkable)?Color.white:Color.red;
            Gizmos.DrawCube(n.worldPosition, Vector3.one *
           (nodeDiameter-.1f));
        }
    }
}
```

我们终于完成了例子中的第一部分。我们这段代码可以工作在任何场景——我们只需要定义地图的尺寸，可行走和不可行走的区域，以及每一个节点的尺寸。改变这些参数可以调整寻路的精度，但要记住，如果增加地图上的节点，就会给CPU带来更大的压力。

寻路的实现

接下来的工作是让角色能够寻找到最终目的地。创建一个新的类，叫做pathfinding。这个类会用来管理寻路的过程。它会实时计算角色需要跟随的最短路径，每秒更新一次，所以如果目的地移动了，它也会跟着变化重新计算最佳路径。

我们从在编辑器窗口里添加一个AI角色开始，它将会搜寻游戏中的另一个角色。为了测试的目的，我们只添加一些基本功能给我们的角色，让它在地图上移动，但是我们也可以使用简单的立方体来测试寻路系统是否正常工作。

在我们导入角色进游戏之后，我们可以开始创建一个类，这个类在稍后将挂在它身上：

```
Grid grid;

void Awake(){
      requestManager = GetComponent<PathRequestManager>();
      grid = GetComponent<Grid>();
 }

void FindPath(Vector3 startPos, Vector3 targetPos)
{
Node startNode = grid.NodeFromWorldPoint(startPos);
 Node targetNode = grid.NodeFromWorldPoint(targetPos);
}
```

我们创建了一个FindPath方法，它保存了需要用来计算的起点位置和目标位置。然后我们添加了一个grid变量，这与我们之前创建的一样，接着我们在Awake函数里给grid赋值：

```
void FindPath(Vector3 startPos, Vector3 targetPos)
{
Node startNode = grid.NodeFromWorldPoint(startPos);
Node targetNode = grid.NodeFromWorldPoint(targetPos);

List<Node> openSet = new List<Node>();
HashSet<Node> closedSet = new HashSet<Node>();
openSet.Add(startNode);
}
```

接着我们需要创建一个列表，该列表将包含在游戏中的所有节点，正如我们之前所演

```
public bool walkable;
public Vector3 worldPosition;
```

示的那样。一个列表里面包含所有OPEN节点，另一个包含所有CLOSED节点：

```
public int gCost;
public int hCost;
public Node parent;
public Node(bool _walkable, Vector3 _worldPos, int _gridX, int _gridY)
{
        walkable = _walkable;
        worldPosition = _worldPos;
}

public int fCost
{
        get {
              return gCost + hCost;
        }
}
```

现在我们来到了Node类，我们增加了新的变量gCost和hCost。这被用来计算最短路径，和之前一样可以获取fCost来表示最短路径，对一个节点的g和h求和，其结果就是f。


f（n）=g（n）+h（n）。

```
Grid grid;

void Awake()
{
        grid = GetComponent<Grid> ();
}

void FindPath(Vector3 startPos, Vector3 targetPos)
{
   Node startNode = grid.NodeFromWorldPoint(startPos);
   Node targetNode = grid.NodeFromWorldPoint(targetPos);

   List<Node> openSet = new List<Node>();
   HashSet<Node> closedSet = new HashSet<Node>();
   openSet.Add(startNode);

   while (openSet.Count > 0)
{
        Node node = openSet[0];
        for (int i = 1; i < openSet.Count; i ++) {
        if (openSet[i].fCost < node.fCost || openSet[i].fCost ==
        node.fCost) {
        if (openSet[i].hCost < node.hCost)
        node = openSet[i];
   }
}
```

Node类编辑完后，我们回到pathfinding类继续完善它：

回到pathfinding类，我们需要定义角色当前所在的节点。我们在遍历OPEN列表的循环中加入了一句Node currentNode=openSet[0]；这会设置默认节点为第一个节点。接着我们创建一个循环，里面会比较节点的fCost值来选择一个最佳选项，openSet[i].fCost<node.fCost||openSet[i].fCost==node.fCost。这段代码可以达到理想的结果，但如果有必要仍然可以优化它。

```
Grid grid;
void Awake()
{
        grid = GetComponent<Grid> ();
}

void FindPath(Vector3 startPos, Vector3 targetPos)
{
   Node startNode = grid.NodeFromWorldPoint(startPos);
   Node targetNode = grid.NodeFromWorldPoint(targetPos);

   List<Node> openSet = new List<Node>();
   HashSet<Node> closedSet = new HashSet<Node>();
   openSet.Add(startNode);

   while (openSet.Count > 0)
{
        Node node = openSet[0];
        for (int i = 1; i < openSet.Count; i ++)
{
        if (openSet[i].fCost < node.fCost || openSet[i].fCost ==
        node.fCost){
              if (openSet[i].hCost < node.hCost)
        node = openSet[i];
          }
        }

        openSet.Remove(node);
        closedSet.Add(node);

        if (node == targetNode) {
              RetracePath(startNode,targetNode);
              return;
        }
```

继续写这个循环，我们现在将当前节点移到OPEN列表或者CLOSED列表中去，并检查当前节点是否是目标节点，如果if（currentNode==targetNode）语句的结果是真的话，那么就意味着角色已经到达了目的地。

```
public List<Node> GetNeighbors(Node node)
{
   List<Node> neighbors = new List<Node>();

   for (int x = -1; x <= 1; x++) {
         for (int y = -1; y <= 1; y++) {
               if (x == 0 && y == 0)
                     continue;

               int checkX = node.gridX + x;
               int checkY = node.gridY + y;

               if (checkX >= 0 && checkX < gridSizeX && checkY >= 0 &&
               checkY < gridSizeY) {




                     neighbors.Add(grid[checkX,checkY]);
                     }
               }
         }
 }
```

现在需要遍历当前节点的每个相邻节点。我们将其添加到grid类的代码中，因此需要打开示例一开始时创建的grid类，并添加上面展示的获取临近节点列表的函数。接下来我们需要添加Node类必须的值（gridX，gridY）：

```
public bool walkable;
public Vector3 worldPosition;
public int gridX;
public int gridY;

public int gCost;
public int hCost;
public Node parent;
public Node(bool _walkable, Vector3 _worldPos, int _gridX, int _gridY)
{
   walkable = _walkable;
   worldPosition = _worldPos;
   gridX = _gridX;
   gridY = _gridY;
}
public int fCost
{
   get
{
         return gCost + hCost;
   }
}
```

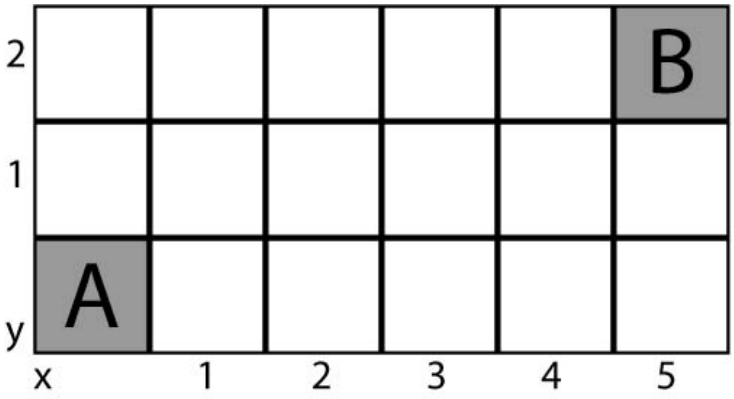
这里，我们已经添加了Node类的最后内容：将会被grid类使用的gridX和gridY。现在

```
foreach (Node neighbor in grid.GetNeighbors(node)) {
if (!neighbor.walkable || closedSet.Contains(neighbor))
{
                continue;
}
}
```

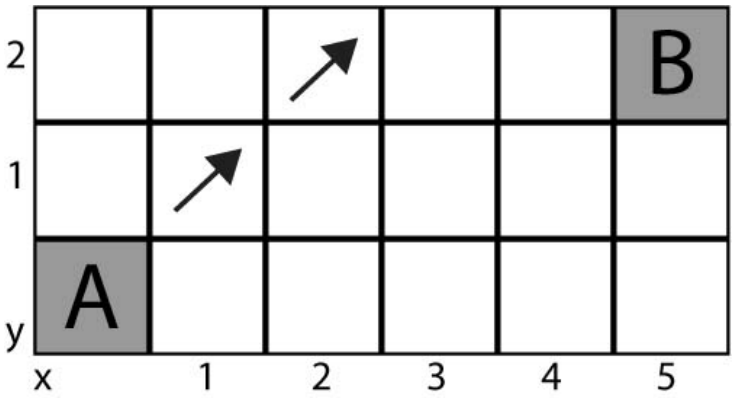
再回头处理pathfinding类：

这里我们添加了一个foreach循环，它将遍历每一个相邻节点，检查它们是否是可行走区域，以及是否属于CLOSED列表。

为了更好地理解我们接下来要做的事，用一些例图来表示我们想要完成的寻路系统：

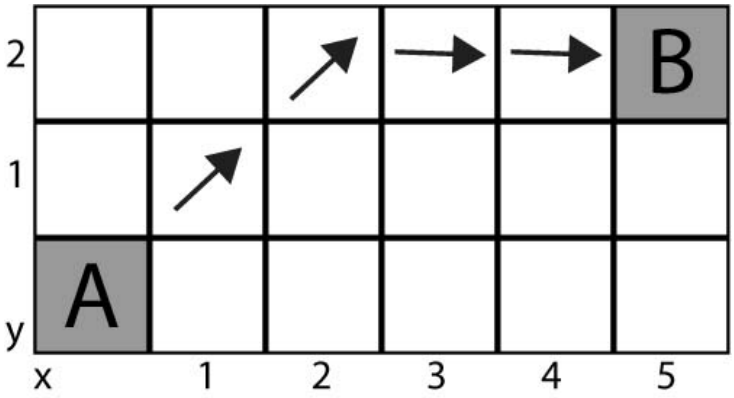


我们首先需要沿着X轴计算到最后位置有多少个节点，然后我们沿着Y轴计算出到最后



位置有多少个节点：

在这个例子中可以看到，为了到达B位置，我们需要向上移动两个点。因为我们是在寻找最短的路径，在向上移动的同时也会在X轴上移动。



要计算到达B位置所需的垂直或水平移动步数，只需用较大的数减去较小的数就可以了。举个例子，要抵达B所在的那条轴，可以通过计算5-2=3，就得到了需要水平移动多少步才能到达最终位置。

现在，我们可以回到pathfinding类添加我们刚刚学到的公式：

```
int GetDistance(Node nodeA, Node nodeB)
{
   int dstX = Mathf.Abs(nodeA.gridX - nodeB.gridX);
   int dstY = Mathf.Abs(nodeA.gridY - nodeB.gridY);

   if (dstX > dstY)
         return 14*dstY + 10* (dstX-dstY);
   return 14*dstX + 10 * (dstY-dstX);
}
```

这里只需要添加一行代码来告诉AI需要在水平和垂直轴上移动多少步才能到达目的地。现在，再回顾一下本章开头所创建的伪代码，检查一下还有哪些代码需要添加。可以看到我们实现了相同的结构，而且几乎已经完成了。伪代码如下所示：

```
OPEN // the set of nodes to be evaluated
CLOSED // the set of nodes already evaluated

Add the start node to OPEN

loop
   current = node in OPEN with the lowest f_cost
```

```
    remove current from OPEN
    add current to CLOSED
    if current is the target node // path has been found
          return

    foreach neighbor of the current node
          if neighbor is not traversable or neighbor is in CLOSED
                skip to the next neighbor

 if new path to neighbor is shorter OR neighbor is not in OPEN
    set f_cost of neighbor
    set parent of neighbor to current
    if neighbor is not in OPEN
          add neighbor to OPEN
```

所以，让我们继续为代码添加一些重要内容来完善我们的pathfinding类。

我们需要设置临近节点的f_Cost，计算这个值我们需要使用临近节点的g_Cost和h_Cost：

```
 foreach (Node neighbor in grid.GetNeighbors(node))
 {
    if (!neighbor.walkable || closedSet.Contains(neighbor)) {
          continue;
    }

    int newCostToNeighbor = node.gCost + GetDistance(node, neighbor);
    if (newCostToNeighbor < neighbor.gCost ||
    !openSet.Contains(neighbor)) {
          neighbor.gCost = newCostToNeighbor;
          neighbor.hCost = GetDistance(neighbor, targetNode);
          neighbor.parent = node;
    }
```

在pathfinding类中，我们添加以下代码，来检查这些临近节点的f_Cost：

```
    void RetracePath(Node startNode, Node endNode) {
            List<Node> path = new List<Node>();
            Node currentNode = endNode;

            while (currentNode != startNode) {
                  path.Add(currentNode);
                  currentNode = currentNode.parent;
            }
            path.Reverse();

            grid.path = path;
    }
```

在退出循环前，我们将会调用RetracePath函数，将参数startNode和targetNode传递进去。然后我们创建这个新函数，并分配一个我们已经探索过的节点列表。为了可视化地看到寻路系统是否正常工作，还需要在grid类中创建path：

```
public List<Node> path;
void OnDrawGizmos()
{
Gizmos.DrawWireCube(transform.position,new
Vector3(gridWorldSize.x,1,gridWorldSize.y));
```

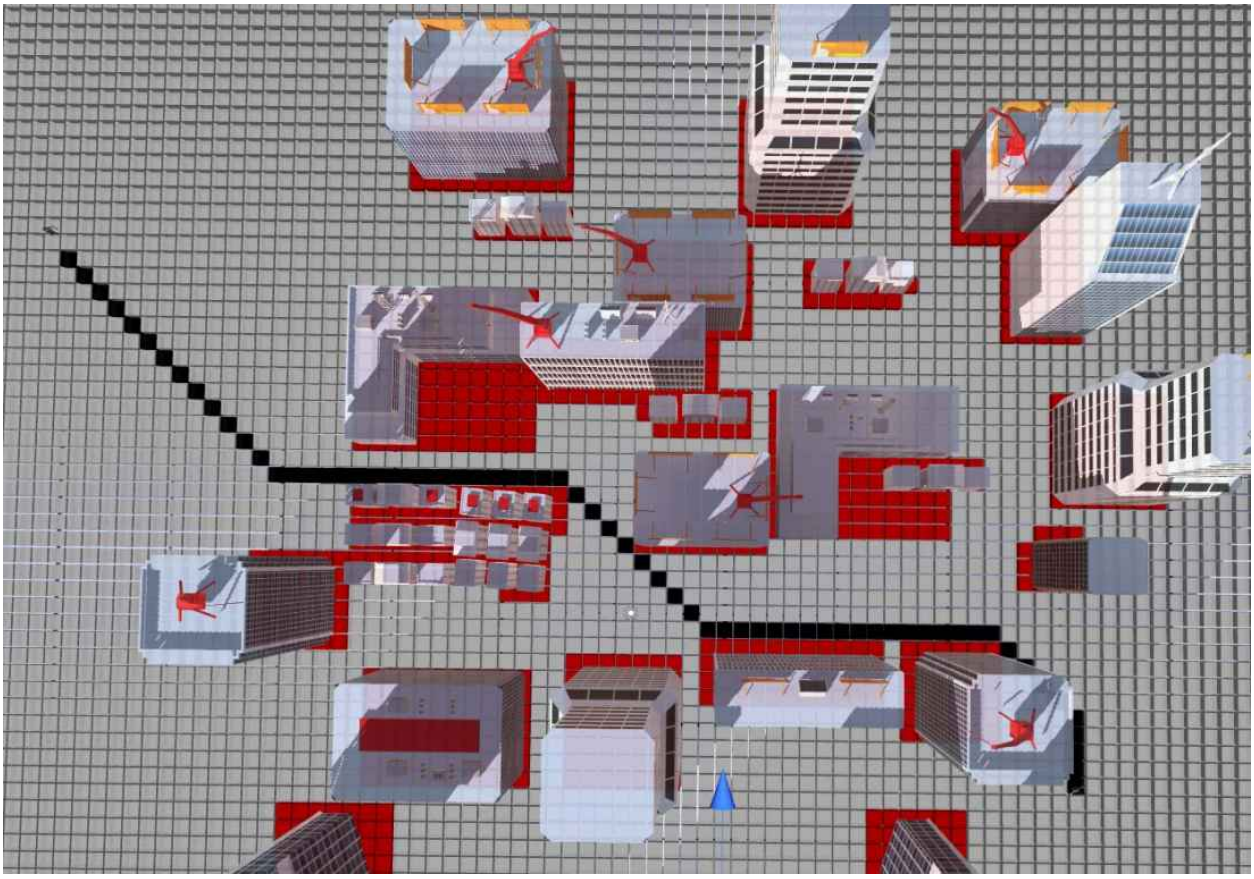
```
        if (grid != null) {
              foreach (Node n in grid) {
                    Gizmos.color = (n.walkable)?Color.white:Color.red;
                    if (path != null)
                          if (path.Contains(n))
                                Gizmos.color = Color.black;
                    Gizmos.DrawCube(n.worldPosition, Vector3.one *
(nodeDiameter-.1f));
              }
   }
}
```

我们更新了grid类中的这段代码，现在包含了列表、路径和将在编辑器窗口中显示的新Gizmo，用来表示从AI位置到目标位置之间的路径。

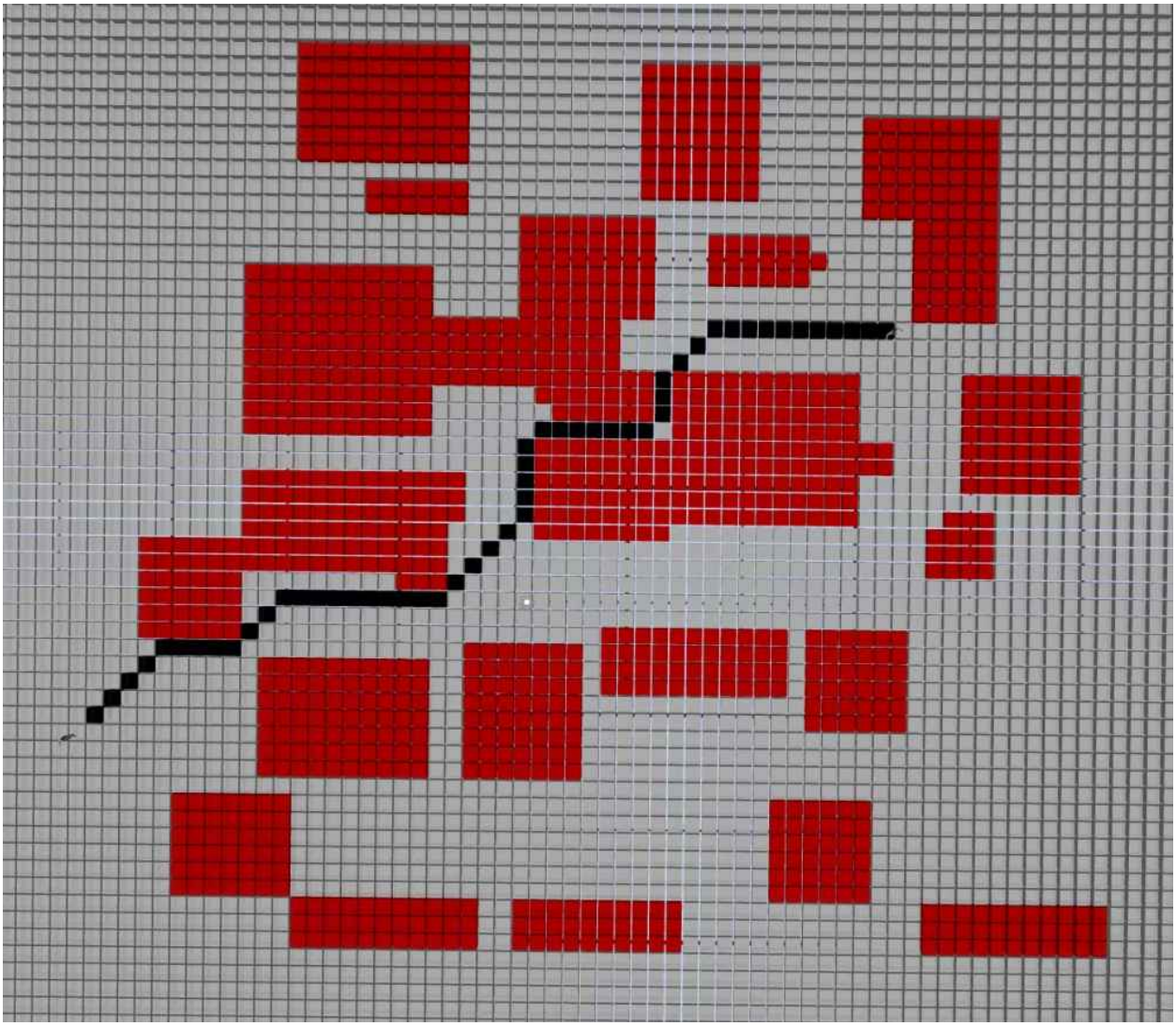
```
public Transform seeker, target;

Grid grid;

void Awake()
{
   grid = GetComponent<Grid> ();
}

void Update()
{
   FindPath (seeker.position, target.position);
}
```

最后，在pathfinding类中加入一个Update函数来结束这个例子，这会让AI不断寻找目标位置。

现在可以转到游戏编辑器窗口了，为我们创建的网格指派pathfinding类。接着我们把AI角色和目的地简单地布置在我们想要的位置：



现在测试pathfinding系统，结果看起来还不错。在上面的截图中，左上角是AI角色的位置，右下角是目标位置。我们可以看到角色找出了最短路线而且避开了建筑物：



然后我们禁用建筑的mesh组件来隐藏建筑，就可以更清楚地看到地图上的可行走区域和不可行走区域。可以看到，角色只能选择可行走区域并绕开所有障碍。用静止的图像来演示这个结果非常不方便，如果我们在运行时改变目标位置，可以看到寻路系统会调整角色的路线，并且总是在找最短路径。

我们刚刚创建了可以在很多流行的游戏中看到的高级寻路系统，这些游戏人人都喜爱。现在我们已经学会了如何创建复杂的寻路系统，我们能够重新创造一些现代游戏中最先进的AI角色，如《侠盗猎车》（GTA）或《刺客信条》（Assassins Creed）。说到《刺客信条》，这将是我们的下一个游戏。下一章的内容会以它作为参考，因为里面的AI角色完美地融合了A*寻路以及拟真的群体交互，正如我们在上面的截图中看到的。

[1] 方格左上角数字代表G的代价，右上角数字代表H的代价，中间数字代表F的代价。——译者注

[2] 可用是指开启列表内的节点。——译者注

[3] F值都为48。——译者注

[4] 较深的灰色区域代表已经彻底探索过的方格，较浅的灰色区域代表计算过数值但是没有再继续探索的方格。注意在A*算法中，已经彻底探索过的方格也可能因为后来发现了更近的路线而重置为浅灰色并重新计算数值。——译者注

[5] 因为下面的操作过后，这个节点就是已经被彻底探索过的了。——译者注

7.3 总结

本章再次讨论了如何创建点到点的运动，我们没有使用简单的方法，而是研究了那些获得巨大成就的游戏工作室是如何解决AI最复杂的特性之一——寻路。我们还学习了如何使用A*算法来重现一种人类特征，来帮助我们搜索并朝着正确的方向移动以到达目的地。

在下一章里，我们将讨论拟真的人群交互。想要让AI角色变得尽可能真实，这是一个非常重要的方法。我们将研究不同类型游戏中使用的不同方法，还会研究人类和动物在它们所处的环境中是如何交互的，以及如何在AI代码中实现它们。

第8章　群体交互

在知道了怎样让AI角色在地图上自由移动并寻找最佳路径之后，我们可以开始学习角色之间的交互了。本章我们将看看真实世界中的群体交互，如何开发可信的群体交互行为，以及角色如何感知到人群中的其他个体。本章的目标是要做到持续给AI角色提供周围的信息，特别是其他智能个体的信息。在本章里，我们还将讨论AI协调、通信与群体碰撞避免。

8.1 什么是群体交互

群体交互是一个真实存在的课题，通常代表多个生物共享同一个空间的情形。一个规模较大的例子是人类社会，人类如何与其他人、其他空间交互。我们作出的决定大多时候都涉及他人，从简单的决定到超前的和复杂的决定。假设我们需要买电影票，电影3点钟开始。如果我们是仅有的几个喜欢这部电影的人，那么在电影开始前2分钟买票还来得及观看电影。但是如果100个以上的人对同一部电影感兴趣，我们就需要提前出发早早到达电影院以便有充足时间买电影票。当到达电影院的时候也有既定的规则，我们要等至轮到自己的时候才能买票。通常我们排在队伍中最后一个人的后面等待。这种行为是一个群体交互的例子。我们的生活环境被其他人包围着，因此我们需要适应它以达成自己的目标。

在电子游戏中也可以找到这种类型的交互，从简单的到高级的、复杂的行为都有。如果游戏中有一个以上的AI角色同处于同一个空间内，那么它们之间就可能发生碰撞。具体情况取决于开发者是怎么想的，比如两个角色同时想做同一件事时会怎么样，可能在设计考虑的范围之内，也可能会发生bug。为解决这些问题，我们需要仔细思考，做出对角色共享同一个空间有帮助的决定，避免错误并让行为更有现实感。

8.2 电子游戏与群体交互

如前面所见，群体交互是现实存在的事物，但是它也出现在电子游戏里，特别是那些模仿人类行为的游戏。由于开放式游戏的流行，群体交互成为了游戏开发中很重要的一方面，因为AI角色一直共享着同一个空间。这意味着几乎所有的开放世界游戏都有必要规划群体交互系统。

8.2.1 《刺客信条》

说到电子游戏中的群体交互，一个非常著名的例子是《刺客信条》系列。非玩家角色在地图上结伴走动，他们会用一种简单的方式避免碰撞并与环境交互。这帮助我们在游戏中创建一个真实的宇宙，这是一个让游戏变得可信的关键点并且能让玩家沉浸到虚拟世界中。

群体交互不仅能在这个游戏的平民角色身上看到，也可以在守卫那里特别是战斗场景中看到。游戏中，玩家经常需要和守卫战斗，并且通常会有多于一个的守卫准备攻击玩家。一个有趣的特点是多个守卫不会同时攻击玩家，它们会审时度势等待更好的攻击机会。

这种理念给玩家带来了一种多个角色之间在交互的感受。


ATTACK
GRAB
STRAFE
24033
23

8.2.2 《侠盗猎车》（GTA）

《侠盗猎车》系列是很多有趣设计理念的源头，可以当作教科书来学习。制作者们不懈尝试让游戏变得更逼真更可信的方法来改进游戏体验，这最终引起了玩家对周围环境的关注，而不单单是只关注主角。为了让周围环境更加吸引人和真实，游戏的创作者们开始花更多的时间在AI开发方面，包括它们如何移动、如何做出反应以及它们之间如何交互。这个游戏中AI角色之间的交互方式在当时是具有开创性的。

玩家可以看到角色们停下来和他人聊天，在戏剧性事件发生时面对面交流，所有这些都让环境更加生动。

如上图所见，游戏中的街道上有各种各样的人正在与其他人交流。我们可以看到一个男人在遛狗，两个女孩在交谈，一个年轻的女人在给另一个人拍照，所有这些对游戏本身玩法并没有任何贡献，但是它让游戏体验更加生动和真实。

8.2.3 《模拟人生》

另一个关于群体交互的伟大先例，是真实生活模拟游戏《模拟人生》。我们又一次提到这个游戏是因为它重塑了开发者们创作游戏的方式，并且在AI这一课题上也贡献巨大。

非玩家角色并不意味着它们只需要站在原地不动，等待什么事情发生。这里我们可以看到所有的角色都有着独特的个性而且它们能够与其他人互动。就算玩家把控制器放在一边光是看着游戏进行，依然会有很多有趣的事情发生，所有这些事情都源于AI角色。

本书之前的章节分析过《模拟人生》游戏中角色的属性，我们还知道它们会在有更紧急的事情时决定不去做某件事情。现在我们已经知道了寻路系统是如何工作的，这里甚至可以为角色实现一个更高级的系统，例如让它们组织事情的优先级，考虑需要花费的时间

来达成特定的目标并最终完成任务。所有这些将会在稍后探索。

8.2.4 FIFA/实况足球

另一个非常重要的例子是体育游戏中的AI角色。从表面看来这种游戏并不是很复杂的类型，事实上体育游戏在AI方面可能是最先进的。

原因是这些游戏是基于真实体育项目的，其中的很多是团队项目。开发既真实又具有特定功能的团队体育游戏难点非常多，因此这也是一个很好的研究样本。

上图展示了FIFA 17游戏中玩法的一角。从中我们可以看到只有一个角色正带着球，同时其他所有角色分散了开来，要么是等待传球，要么是试图接近那个角色的位置以试着抢球。总的来说，这让二十二个人为了一个球而行动（每支队伍十一个人）。这就是为什么体育游戏需要良好的AI角色的原因，因为它们需要持续地工作，就算没有带球的时候也是如此。它们各自拥有自己的位置，进攻或防守，左边、右边或中间，等等。在团队中，它们需要一起按照战术行动同时还要遵守比赛规则。如果队友拿球并且在向前跑，那么就要朝同样的方向跑动以协助它，让它更容易突破防线，或者可以选择留守后方，因为如果它丢球需要有人把球抢回来。

角色之间的交互在持续不停地发生，不仅仅是简单地追球跑动看谁先抢到，而是需要分享大量信息以取得比赛的胜利。

8.3 规划群体交互

在创作游戏的时候，有时我们会把工作计划抛在脑后，总想着创造一个好的游戏只需要一个很棒的创意就够了，然后所有事情都会自然地从头脑中出现。其实成功的游戏之所以成功，是因为开发中的每一步都规划到了最末尾的细节，在开发游戏时我们应当牢记这一点。目前我们已经掌握了强有力的技术和知识，可以通过很多AI功能来开发出既有挑战性又有趣味的游戏，因此我们的下一步是整合这些技术，制定一份计划，来创造一个看起来更好的游戏。

我们已经分析了一些关于群体交互的流行的例子，现在可以看看如何规划这些类型的交互。我们将仿照之前的例子来看看如何规划类似的群体交互行为并加入到自己的游戏里。

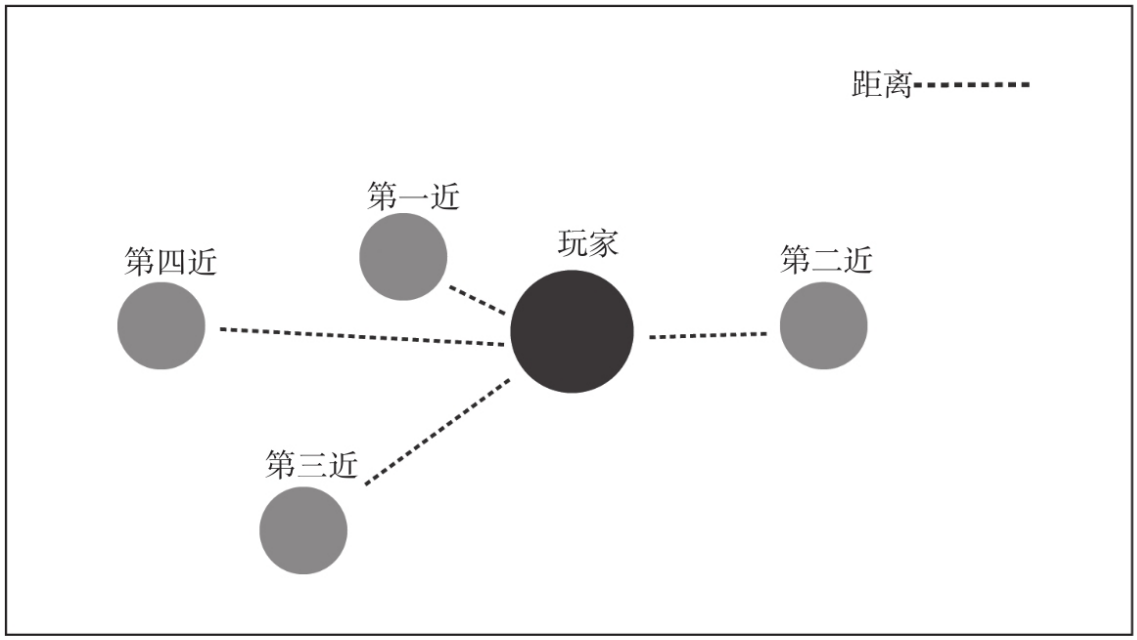
8.3.1 小组战斗

我们来创建一个场景，其中有多个AI角色正在攻击玩家。从编写战斗功能开始，比如单手攻击、双手攻击、防御以及尾随玩家等等。当实现了这些以后，角色已经可以攻击玩家了。假如不规划任何东西仅仅是让四个角色和玩家战斗，它们会同一时间发动攻击以击败玩家。

那样的话可能会产生这样或那样的bug，但是如果没有时间改进也可以这么做。我们想要的是实现一些AI角色之间的交互，这样它们就不会看起来很傻地同时攻击玩家而完全不分析当前情况：

所以，现在游戏已经能够运行了，而且里面的敌人角色能够跟随并攻击玩家，我们想要规划AI角色之间的交互来让它们决定谁来先攻击，以及其他人何时也可以发动攻击。

我们可以从许多因素中选择一部分，让角色利用它们做出决策。我们规划得越详细，AI角色的完成度就越高，挑战性也越强：



对于这个例子，我们通过检测AI角色和玩家之间的距离来决定首先由哪个角色发起攻击。我们希望近处的角色优先攻击，其他人在等这个角色血量较低时再攻击。一旦角色血量偏低，第二近的角色就会加入战斗并攻击玩家。

现在我们已经设定了第一个判断标准以决定谁先攻击，接下来要确定当其他角色等待时应该做些什么。要考虑到，玩家可以在任何时间攻击任何一个敌人，而且我们不希望AI角色仅仅由于没有等到攻击的时机而一直处于静止状态。因此解决方法是思考可能发生的不同状况并计划AI应当如何反应，特别是角色之间应当如何交互：

```
public static int attackOrder;
public bool nearPlayer;
public float distancePlayer;
public static int charactersAttacking;
private bool Attack;
private bool Defend;
private bool runAway;
private bool surpriseAttack;

void Update ()
{

    if(distancePlayer < 30f)
    {
        nearPlayer = true;
    }

    if(distancePlayer > 30f)
    {
```

```
            nearPlayer = false;
        }

        if(nearPlayer == true && attackOrder == 1)
        {
            Attack = true;
        }

        else
        {
            Defend = true;
        }

    }
```

我们从简单的代码开始，根据遇到的情况决定角色的行为，在这之后就可以一直添加更多必要的内容来让它如我们所期望的进行工作。在这个例子中，我们创建了一个静态整型变量attackOrder用来表示每个角色的攻击顺序，所以它们知道何时发动攻击。在这之后，有个布尔类型变量nearPlayer，用来表示角色是否靠近玩家。地图上有30个角色但是我们只希望最近的一个攻击玩家。这个例子中，其他角色只是简单地忽略玩家。为了判定AI角色是否离玩家很近，创建了一个浮点类型的变量distancePlayer，代表玩家与AI角色之间的距离。之后添加了一个公开的静态整型变量charactersAttacking，每次有角色靠近玩家时都会自增1。我们可以使用它来给所有角色提供一共有多少人正在攻击玩家的信息。

像这样短小而简单的代码，也会给群体交互行为带来巨大的差异，因为可以用共有多少人在攻击玩家的信息来决定它们将会做什么。例如，如果只有两个角色在攻击玩家，可以让其中一个一直防御玩家的进攻并让另一个角色攻击玩家，当玩家从一个目标切换到了另一个，它们就交换职责继续进行同样的行为，来给玩家制造麻烦。

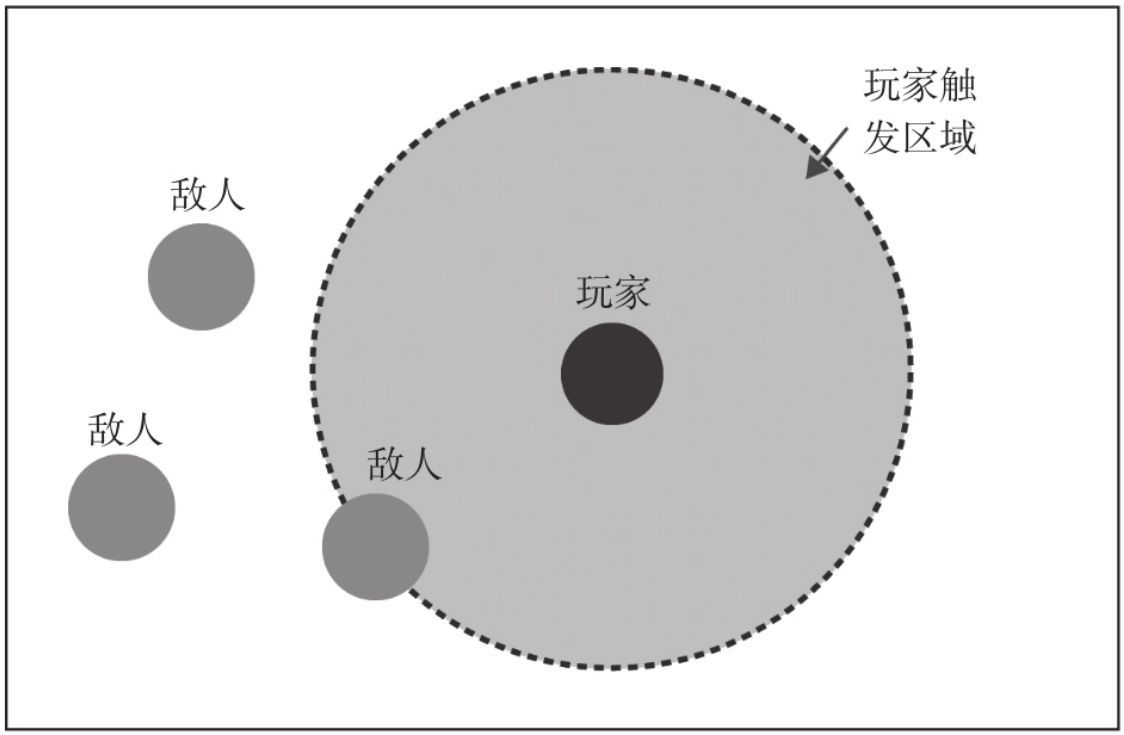
上述内容可以从上图中看出来，其中一个骷髅告诉另一个骷髅它将进行防守，另一个骷髅可以从后面攻击玩家。这正是群体交互的含义，一个角色给另一个角色提供信息来让它知道能做什么、怎么做。共享的信息越多，角色的选择就越多，而交互的真实性就越强，因为它们不是一个人在战斗。

如我们所见，就算用简单的代码也能达成复杂的结果，但是提前思考和规划一切还是十分有必要的，另外很显然，随着我们添加更多的细节和选项，代码也会变得更长。

8.3.2 通信（警告区域）

继续上面的例子，地图上有一些骷髅，在玩家靠近时开始攻击玩家，可以添加更多功能进去以让它们能够进行更多交互。一种有助于它们更好地协作而不是单打独斗的方法，就是通信。例如，骷髅们只在玩家接近时发起攻击，但是如果离玩家较近的骷髅喊出它发现了玩家呢？可以假设那个区域的所有AI角色都能听到叫声并赶来帮助它们的朋友。

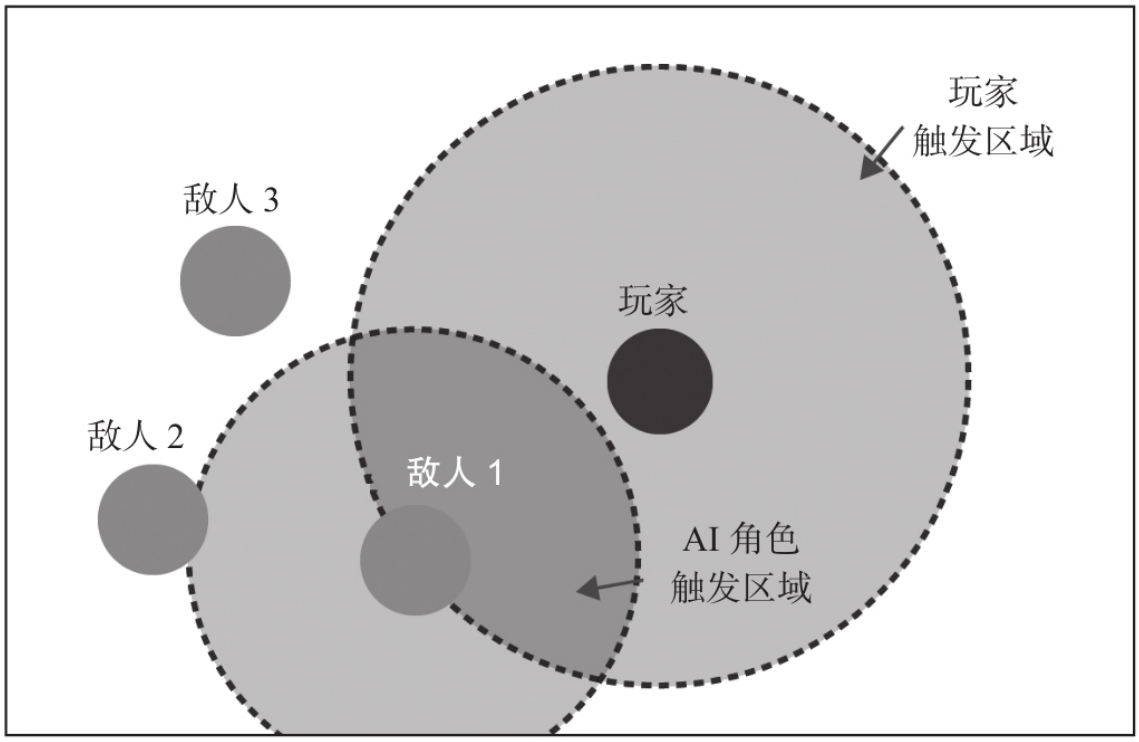
我们再次使用简短的代码来达到这个目标，但是如果不预先计划好交互行为以及角色应当如何像团队一样行动，上述要点就会从AI角色中漏掉，它们就会表现得更独立而不那么聪明了。



如上图，这是系统目前的样子。AI角色并不与其他人通信，因此察觉到玩家位置的骷髅仅仅是离玩家足够近的那些。如果试图创建一个群体系统，我们需要针对类似的情况做出规划。其他骷髅不能仅仅因为看不到玩家，就假装没有事情发生而不做出反应。

让我们想想真实生活中的情景。例如，有一个人在房子里，其他人在房子外面。外面的一个人看到了一个不可思议的美丽的小鸟而房子里面的人没有看见，所以他还是待在房子里面。如果看到小鸟的人不与其他人通信，那么房间里的人永远也不会知道这件事。所以，通常情况下看到鸟的人会告诉其他人也来外面看那只美丽的小鸟。这才是真实的行为，值得添加到群体交互系统中去。

要让这种没有交互的情况变成更真实的样子，我们需要给角色添加更多功能来让他们能相互交流。这里，我们只需要简单的通信，所以可以使用和之前判断角色能否看到玩家类似的代码：

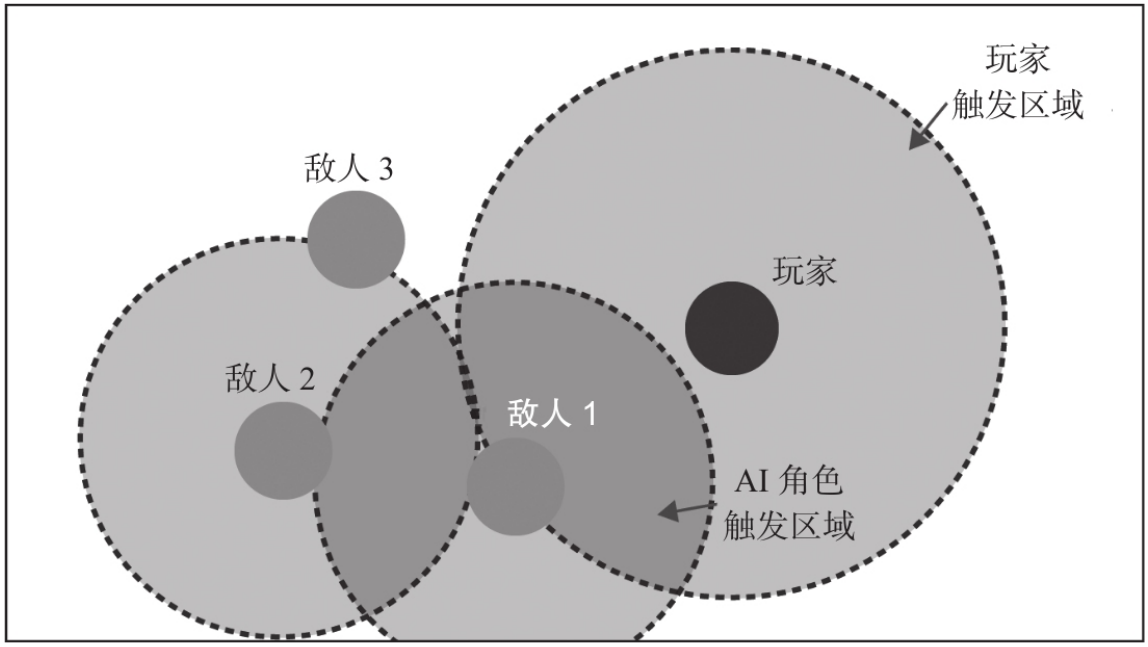


现在AI角色已经进入了玩家的触发区域，因此它会发出叫喊让附近所有的AI角色知道玩家的存在。上图中我们可以看见不仅玩家有触发区域，发现玩家的敌人也有自己的触发区域。这个新的触发区域是用来警告其他角色的，也就是代表喊叫的范围。因此当游戏中的敌人发现了玩家，我们会听到喊叫，这带来一种AI角色们在交流的感觉：

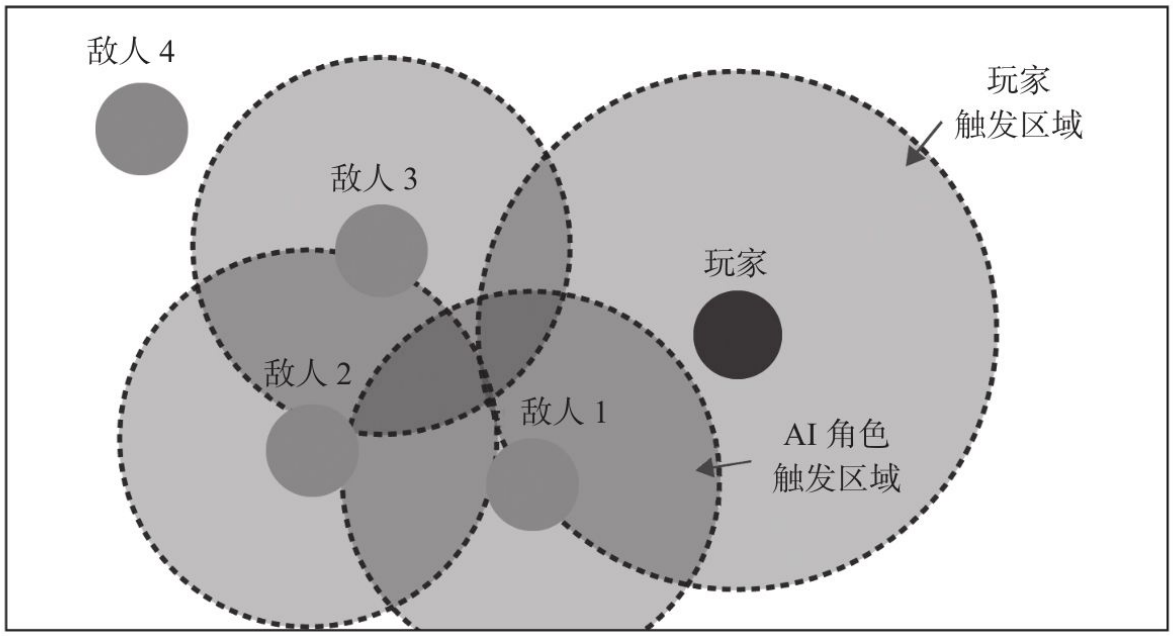
```
public static int attackOrder;
public bool nearPlayer;
public bool nearEnemyAttacked;
public float distancePlayer;
public static int charactersAttacking;
private bool Attack;
private bool Defend;
private bool runAway;
private bool surpriseAttack;

void Update ()
{

    if(distancePlayer < 30f)
    {
          nearPlayer = true;
    }

    if(distancePlayer > 30f)
    {
           nearPlayer = false;
    }

            if(nearPlayer == true && attackOrder == 1)
                {
                    Attack = true;
                }

             else
                {
                    Defend = true;
                }



          if(nearEnemyAttacked == true)
          {
              runPlayerDirection();
          }
    }
```

要实现这一点，我们简单地添加新的布尔变量nearEnemyAttacked。为了将逻辑联系起来，我们添加了触发器判断来检查附近有没有骷髅已经发现了玩家。如果触发了，布尔变量就变为true，否则就保持false。

一旦触发器被触发了，就轮到这个AI角色呼叫周围其他人了：



如上图所见，通过上面实现的通信系统，我们现在有了三个角色能够充分感知到玩家的位置。最后一个角色还会喊叫，试图通知其他人玩家的位置，但是如果它的触发区域并没有覆盖其他角色，那么什么都不会发生：



例如，敌人4离触发区域很远，所以它会保持呆在那个区域直到玩家靠近，另一方面，它也不会知道现在发生了什么。

这个例子的诀窍是让角色互相交谈、喊叫或是其他方式来引起周围其他角色的注意。这会带来一种通信的感觉，把简单独立的动作变成了更吸引人的团队交互。

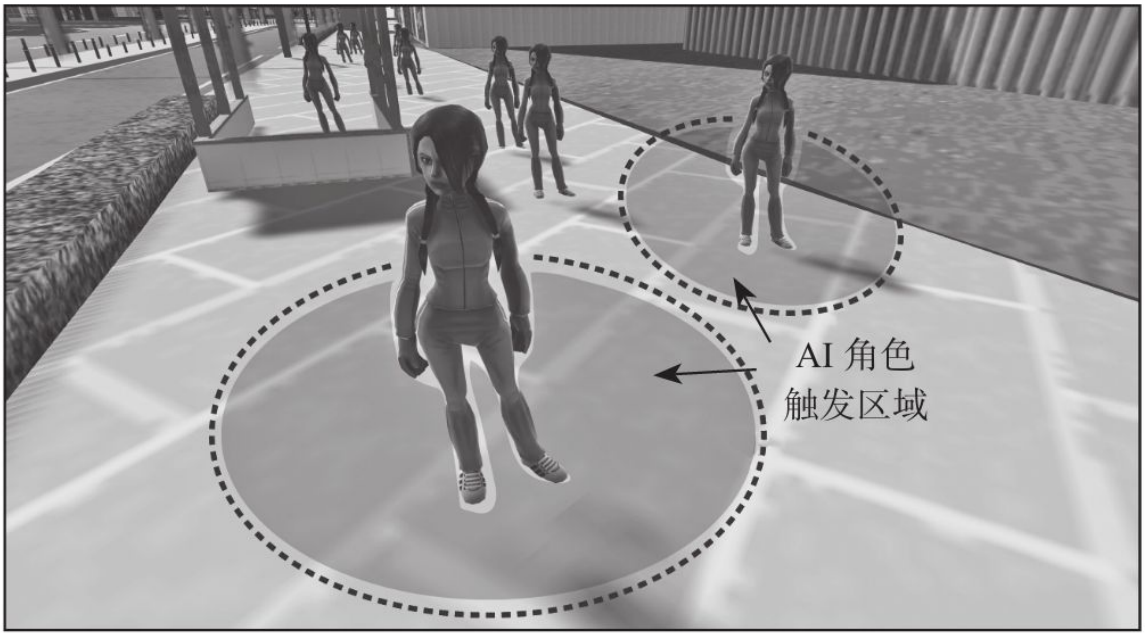
8.3.3 通信（与其他AI角色交谈）

有很多例子可以用于演示通信的概念，因为总能找到新的方式来让角色通信，如同现实生活中我们就一直在寻找新的交流方式。但是这里我们会坚持采用最基本的方式——交谈。

如果我们打算在游戏中放置很多AI角色，很快交互行为就会成为游戏中很大的一部分内容，玩家的注意力也会直接或间接地转向它们。不是每个游戏都有直接对玩家行为做出反应的角色，也许由于玩家只是游戏中的一个普通角色而不会被特殊对待。因此这里将排除玩家的影响仅仅规划AI角色的交互：

我们创建一个拥有很多人的城市，并给人物添加一些细节以便让他们表现得更像真实的群体。可以从添加基本的移动信息开始，比如走路、跑步、停止和寻路。这些实现了之后，可以再添加一个独立的角色，他能在城市中的人行道上走动同时能避免和建筑碰撞。

关于这个例子的第一个建议是添加简单的触发器检测来让角色能够感知到周围人物经过：



添加了触发区域之后，下一步是处理他们之间的交互。我们打算在这里使用可能性图

```
If(probabilityFriendly > 13)
{
  // We have 87% of chance
  talkWith();
}
```

来判断是否遇到了可以交谈的熟人：

要实现这点，我们添加了一个整型变量probabilityFriendly来代表发现熟人的概率。当新的角色进入了触发区域，就会产生一个随机数，如果这个数字满足设定的条件，两个角色都会停止走动并开始交谈。这之后，就可以持续为情景添加更多细节，比如当交谈结束后，我们让他们一边并肩走动一边说话，无穷多的选项可以从这个小小的触发器与概率图中得到。

这个例子背后的思想是让角色随机地与其他人交互。从玩家的视角看来，这就好像这些角色本来就是朋友，他们停下来交谈只是因为他们认识对方。这对创建一个真实的世界很有帮助，而且这和技术的关系也不是很大，仅仅是规划每一种可能的交互而已。

8.3.4 团队竞技

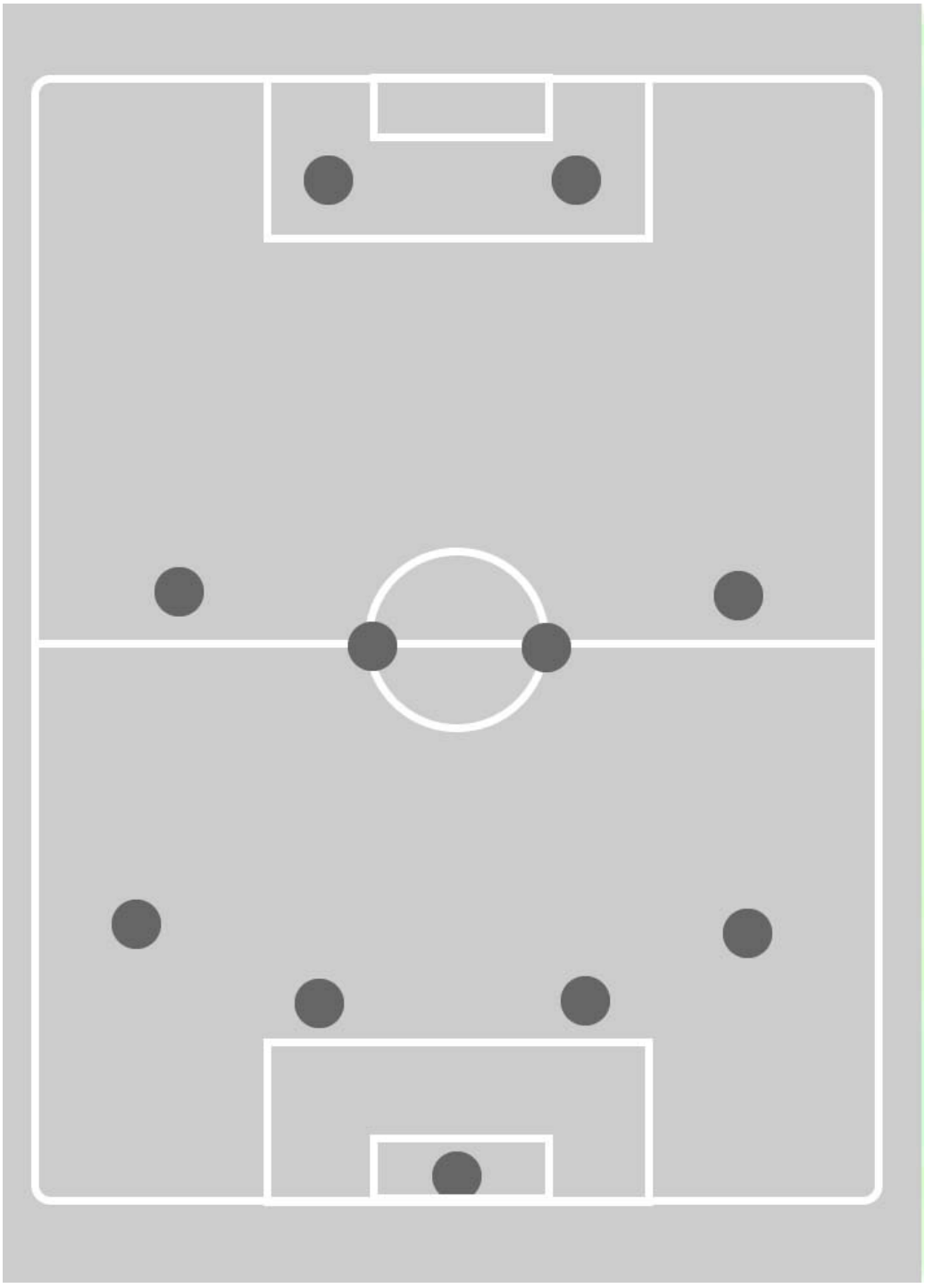
之前解释一些游戏群体交互的例子时谈到过，体育游戏往往具有优秀的AI系统，能在团队竞技方面工作得很好。现在我们深入看看电子游戏的核心功能，看看它们如何做到这一点，能让AI角色如此具有挑战性而又有真实感。

我们分析一下现实中的足球，两支队伍每队由11个队员组成。为了赢得比赛，队伍需要获得比对方更高的分数，因此游戏过程可以分成两个基本的状态：进攻状态，目标是得分；防守状态，目标是防止对方得分。比赛中只有一个足球，因此大量时间球员都是处于无球状态，而对比赛结果来说无球的时间反而更加重要。从对方那里抢球或是提前站位以接到球，是球员无球状态下的两种基本策略。

电子游戏试图去模仿比赛的每一个细节，由于它是一场团队竞技，在AI团队交互方面有大量的开发工作。AI角色的头脑中需要有更多的团队协作和更少的单打独斗。因此它们只做出特定的、和团队目标相一致的决策。

我们观看足球比赛时，会听到运动员告诉其他人把球传给自己，或是让别人向前、向后跑动，等等。这里的思路是把类似这样的交流也放在游戏中。并不是必须实现为语言的交流，只需要做出类似的动作让游戏更加真实即可。

我们来一步一步分析游戏过程中AI所做出的基本决策。我们从考虑角色在赛场上的组织开始，如下图所示：



这是一个简单的例子，展示了队伍在球场上的阵形。最下方的圆点代表守门员，也就是防守球门的人。这个角色是唯一一直待在这个区域的人。其他所有人可以在场上随意移动。现在我们已经看到了球队如何被部署到球场上，接下来看下个例子。

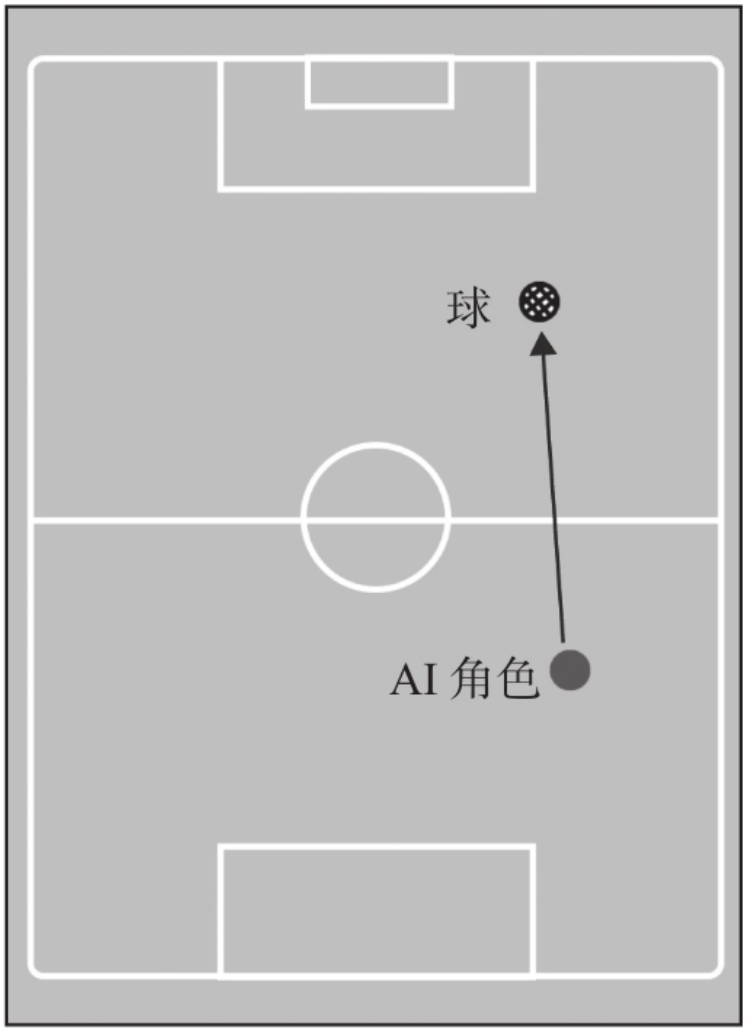
游戏中每个角色都有着独立的目标。也许是传球给进攻的球员，也许是尽可能多地破门得分，也许是待在后方防守，等等。每个人有自己独立的目标，同时也需要考虑团队的目标来决定某个时刻哪一个目标更重要，做出的决定是否对达到最终目标有帮助。

我们继续创建一个单独的球员。从基本功能开始，向球的方向跑动。要实现这一点，

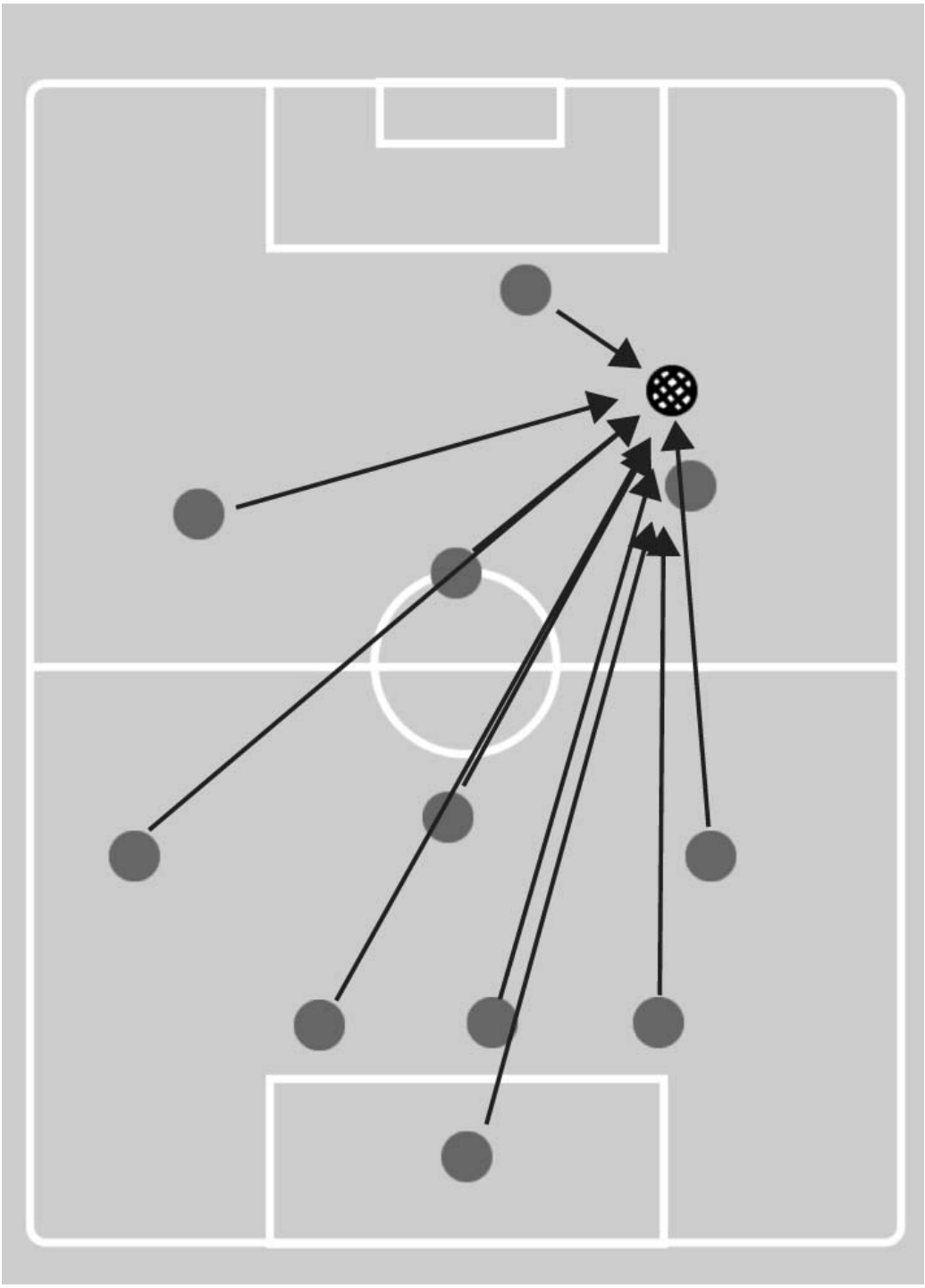
```
public float speed;
public Transform ball;
public bool hasBall;

void Start ()
{
    speed = 1f;
}

void Update ()
{
    if(hasBall == false)
    {
       Vector3 positionA = this.transform.position;
       Vector3 positionB = ball.transform.position;
       this.transform.position = Vector3.Lerp(positionA, positionB,
         Time.deltaTime * speed);
    }

    if(hasBall == true)
    {

    }
}
```

我们可以使用本书之前解释过的技术，例如朝目标位置走动：

这里的代码能够让角色向球的方向跑动。这里先只对一个角色进行设计，之后我们会逐步添加队伍交互以便至少得到一个基本框架，如我们在成熟的体育类游戏中看到的样子。因此如果只运行这一部分代码，只能看到角色朝着球的位置跑动，这是足球游戏的基本玩法之一：



到这里，已经能让单独的球员合理地工作了，这只是目前希望看到的。如果我们添加更多角色到游戏中，所有人都会朝球的方向跑动，忽视其他所有东西，没有通信和互动的效果就是如此：



如果我们让所有角色都朝着球的位置跑动，如上图所示，那么看起来就像每个人都不知道其他人的存在。为避免这种状况，可以获取离球最近的角色并让他通知其他角色，其他角色就可以不再朝球的位置跑动了。为创建这个功能，我们持续计算每个角色和球的距离：

```
    public float speed;
    public Transform ball;
    public bool hasBall;
    public float ballDistance;

    void Start ()
    {
        speed = 1f;
    }

     void Update ()
   {

        if(hasBall == false)
        {
            Vector3 positionA = this.transform.position;
            Vector3 positionB = ball.transform.position;
            this.transform.position = Vector3.Lerp(positionA,
               positionB, Time.deltaTime * speed);
        }

        if(hasBall == true)
        {

        }



      ballDistance =Vector3.Distance(transform.position,ball.position);

 }
```

为实现这点，我们使用本书讲过的计算距离的方法。因此现有三个新变量，其中浮点型变量ballDistance代表球和角色之间的举例。

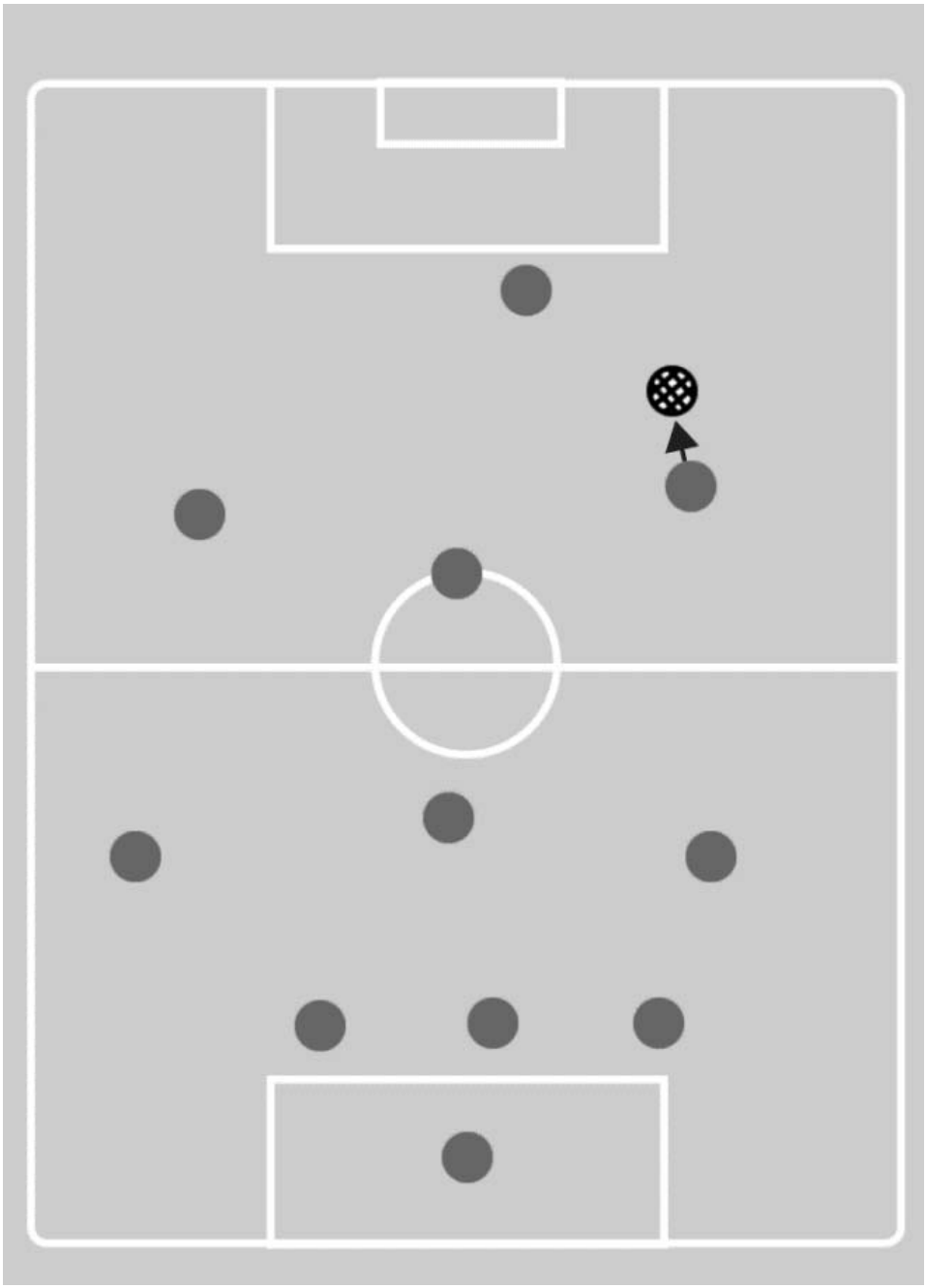
完成之后，接着需要让角色验证自己是否是离球最近的人，如果是，则他接着朝球的位置跑动：

```
public float speed;
public Transform ball;
public bool hasBall;
public float ballDistance;
public static float teamDistance;

void Start ()
{
   speed = 1f;
}

void Update ()
{

    if(hasBall == false)
    {
        Vector3 positionA = this.transform.position;
        Vector3 positionB = ball.transform.position;
        this.transform.position = Vector3.Lerp(positionA, positionB,
           Time.deltaTime * speed);
    }

    if(hasBall == true)
    {

    }

    ballDistance =Vector3.Distance(transform.position,ball.position);
    if(teamDistance < ballDistance)
    {
         teamDistance = ballDistance;
    }

}
```

在这个例子里，我们简单地添加一个所有角色共享的变量，也就是静态浮点型变量teamDistance。它会存储最近的角色离球的距离，通过它角色可以知道自己是否是离球最近的人。下一步就比较容易了，每次让角色检查自己和球的距离，如果比当前最近的距离更近，就朝球的方向跑动。这是我们添加到AI里面的第一个团队协作的元素。球员们会依照设计让离球最近的那个人去抢球。我们进一步分解了问题，这样球员们就能互相检查以做出决定了。另外，这个例子里我们还有一个前提，即所有人的跑动速度是一样的：

```
    public float speed;
    public Transform ball;
    public bool hasBall;
    public float ballDistance;
    public static float teamDistance;
    public bool nearTheBall;

    public float teamdist;

    void Start ()
    {
         speed = 0.1f;
         teamDistance = 10;
    }

    void Update ()
   {
       teamdist = teamDistance;

       if(hasBall == false && nearTheBall == true)
       {
           Vector3 positionA = this.transform.position;
           Vector3 positionB = ball.transform.position;
           this.transform.position = Vector3.Lerp(positionA, positionB,
              Time.deltaTime * speed);
       }

       if(hasBall == true)
       {
       }

       ballDistance =Vector3.Distance(transform.position,ball.position);

       if(teamDistance > ballDistance)
       {
           teamDistance = ballDistance;
       }

       if(teamDistance == ballDistance)
       {
           nearTheBall = true;
       }

       if(teamDistance < ballDistance)
       {
          nearTheBall = false;
       }

   }
```

现在测试一下游戏，可以看到只有一个角色在追着球。所有其他角色都知道有另一个人离球更近，他会去抢球。到这里，我们已经有了一个团队交互的基本框架，也走在了正确的方向上：



下一步，我们需要实现球的移动，球在整个游戏中都是不断移动的，之前的代码假设了一个静态的情况，但是如果球移动了，距离检测就要重置，因为之前的代码中抢球的角色会离球越来越近，而这个距离永远不会增大，所以我们需要更新代码。从给球添加一个脚本开始：

```
 public Vector2 curPos;
 public Vector2 lastPos;

 public bool ballMoving;

 void Update ()
{

    curPos = transform.position;

    if(curPos == lastPos)
    {
         ballMoving = false;
    }

    else
    {
         ballMoving = true;
         characterAI.teamDistance = 10;
    }

    lastPos = curPos;
}
```

添加了这段代码之后，每当球移动的时候，距离检测就要更新一次。现在我们来让球移动起来。要实现这一点，我们需要允许角色踢球。

首先，我们更新球的脚本。需要添加一个变量保存球在被角色踢出之后的落点位置：

```
public Vector2 curPos;
public Vector2 lastPos;
public static Transform characterPos;
public float speed;

public bool ballMoving;

void Start ()
{
     characterPos = this.transform;
     speed = 2f;
  }

  void Update ()
  {

      curPos = transform.position;

      if(curPos == lastPos)
      {
          ballMoving = false;
      }

      else
      {
          ballMoving = true;
          characterAI.teamDistance = 10;
      }

      lastPos = curPos;

      Vector2 positionA = this.transform.position;
      Vector2 positionB = characterPos.transform.position;
      this.transform.position = Vector2.Lerp(positionA, positionB,
         Time.deltaTime * speed);
}
```

像这样，这里实现的是给出球的落点信息。为了实现这个功能，我们添加了公开静态Transform类型变量characterPos。另外我们获取了其他角色的位置，因为要让角色把球传给其他人而不是简单地踢开球：

```
public float speed;
public Transform ball;
public bool hasBall;
public float ballDistance;
public static float teamDistance;
public bool nearTheBall;
public List<Transform> teamCharacters;
public int randomChoice;
public float teamdist;
```

之后给脚本添加了一些变量。其中有一个列表变量，包含了队伍中所有玩家的坐标信息。思路是让角色选择一个队友，并把球踢给他。

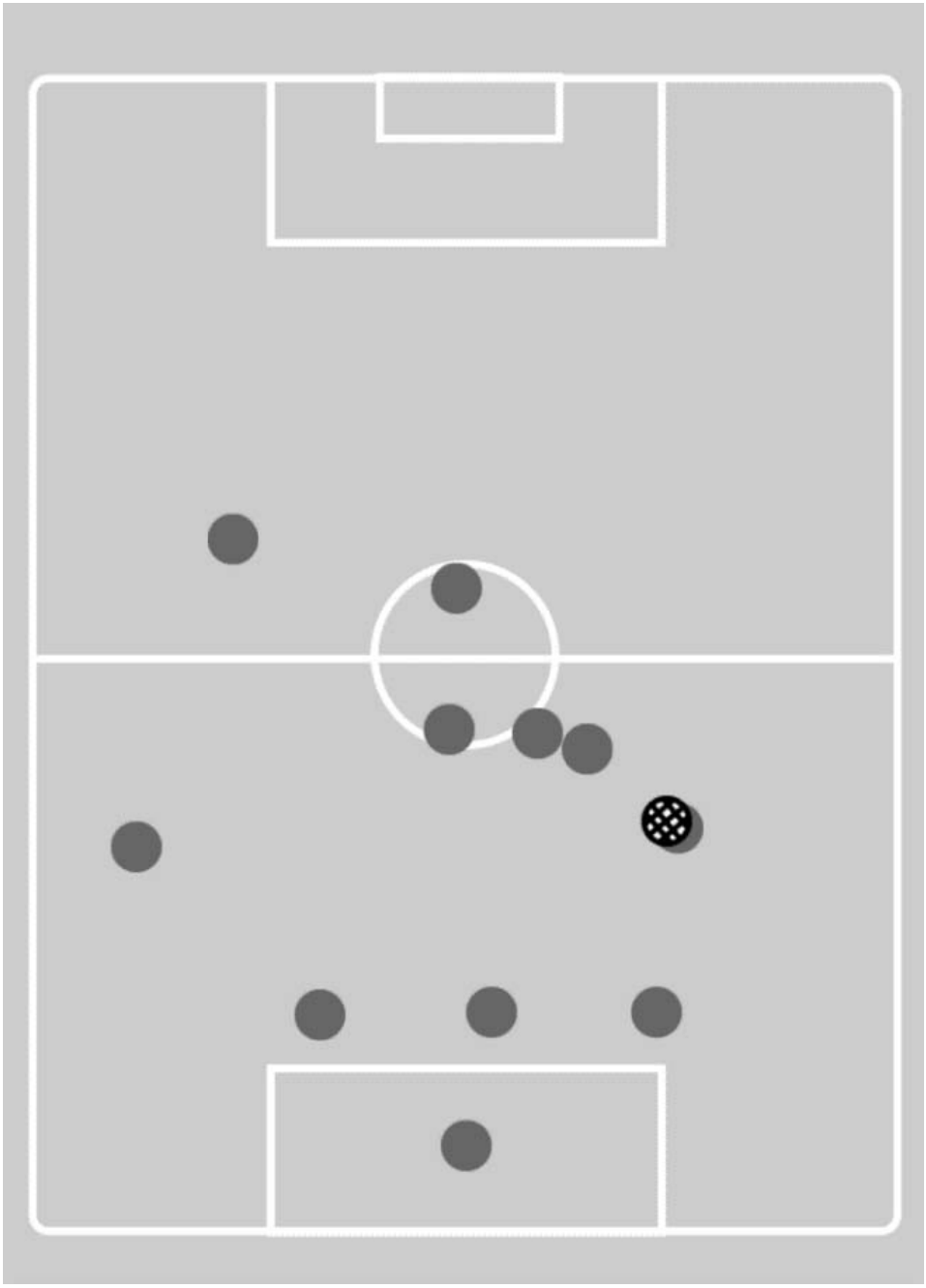
针对这个例子，我们选择了角色的坐标作为球的落点，要想让这个效果更真实，可以添加更多抛物线轨道的细节，比如重力和风力的影响：

```
void Update ()
{
     teamdist = teamDistance;

     if(hasBall == false && nearTheBall == true)
     {
         Vector3 positionA = this.transform.position;
         Vector3 positionB = ball.transform.position;
         this.transform.position = Vector3.Lerp(positionA, positionB,
            Time.deltaTime * speed);
     }

     if(ballDistance < 0.1)
     {
         hasBall = true;
     }

     if(hasBall == true)
     {
          passBall();
          hasBall = false;
     }

     ballDistance =Vector3.Distance(transform.position,ball.position);

     if(teamDistance > ballDistance)
     {
         teamDistance = ballDistance;
     }

     if(teamDistance == ballDistance)
     {
         nearTheBall = true;
     }

     if(teamDistance < ballDistance)
     {
          nearTheBall = false;
     }

}

void passBall ()
{
      randomChoice = Random.Range(0, 9);
      ballScript.characterPos = teamCharacters[randomChoice];
}
```

之后使用刚刚添加的变量，我们在角色离球足够近的时候，把方向信息发送给球。void passBall（）函数在每次角色传球时调用。这里我们只是想让角色传球给其他人，因此我们从列表中随机选择了一个队友。

测试这个游戏，可以看到发生了很多移动与交互行为。从中可以观察到最近的角色会抢到附近的球，抢到之后，他会把球传给另一个角色。球会朝角色移动，它们之间的距离

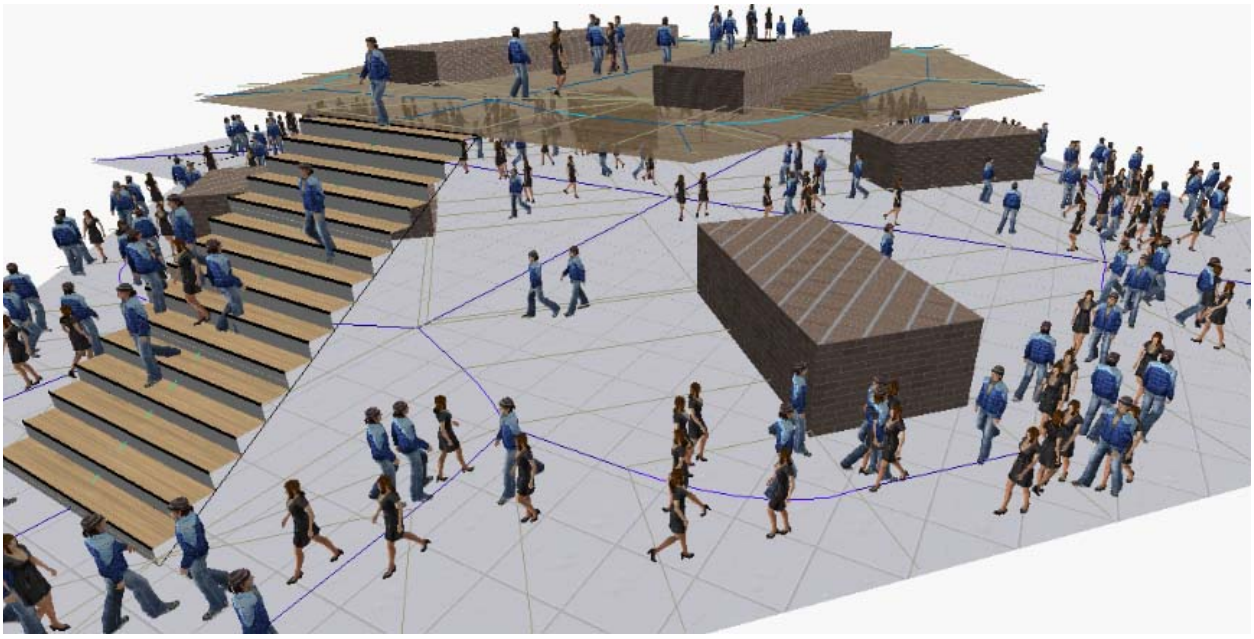
缩短，这样角色就将抢到球，然后把球再传出去。当前，这种行为会永远循环下去，角色拿球、传球，另一个角色拿球、传球，以此类推。



现在已经实现了简单足球游戏的基本框架，可以继续添加类似的更多功能来让角色交互，来看谁会抢到球并传球给队友。

8.4 群体碰撞避免

最后我们讨论一下群体碰撞避免来结束这一章。让许多人出现在同一块地图上的设计



思路，已经成为了开发世界游戏的标准。但是这经常带来一个问题，如何避免碰撞：

我们已经研究过了高级寻路是如何工作的，并且我们知道了在开发AI移动功能时，它是一种强有力的方法。但是如果有许多角色想同时到达同一个目标点，就会造成它们互相碰撞，可能会挡住到达目标必要的路径。如上图所见，所有东西都能够顺畅地运行，而不会产生异常情况，因为角色在沿着不同的方向移动且很少干涉到其他人。

但是如果所有角色试图同时到达同一个目标，会怎么样呢？例如都尝试进入屋子里。门一次只允许通过一个人，这意味着其他人应当排队等待通过门。

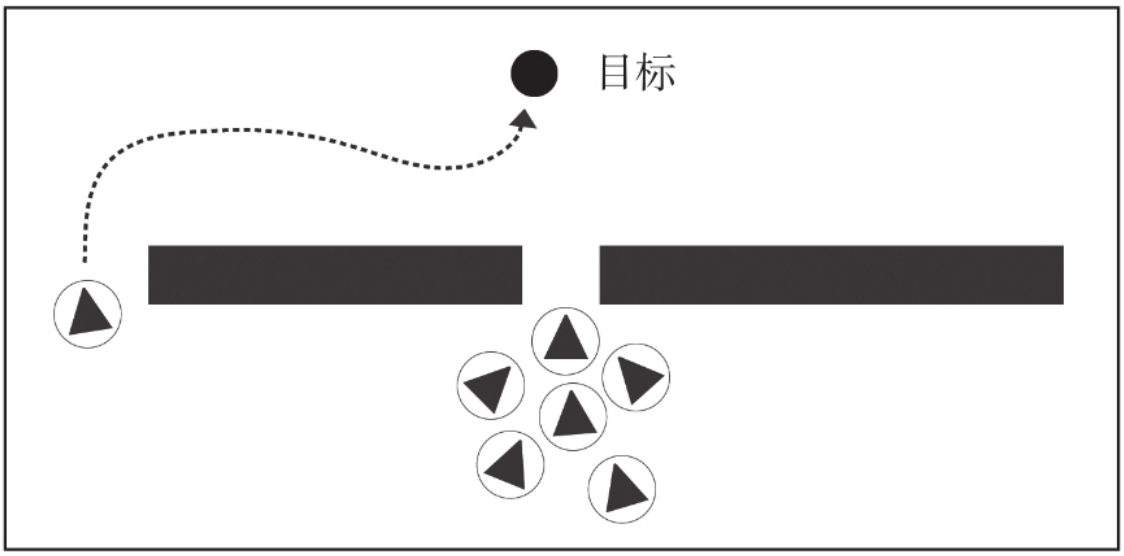
这个问题的解决方案还正在研究中，而且还没有得到绝对适用任何情况的答案，但是有一些方法可以参考。

当前，群体运动的解决方案通常涉及两个不同的层面，一个是寻路，另一个是局部碰撞避免。通过尝试这些方法得到了一些成果，能够用来制作一个高品质的移动系统同时在局部避免碰撞，在很多游戏中这种方法都很常见。

要得到满意的结果，需要对这个方法做出不同的修改。一个流行的方法是结合A*算法与velocity obstacle方法。这种方法可以让我们得到角色与可能发生碰撞的另一个角色之间的距离。

在高密度群体的场景中，仅仅依靠局部碰撞避免和理想化的寻路算法会引起角色在热点、路径交汇点聚积。碰撞避免算法只能帮助我们避免在进行寻路时的局部的碰撞。游戏经常依赖于这些算法在高密度群体的场景里将角色转移到不太拥挤的、不太直接的路线上去。在特定的情况下，碰撞避免可以到达这个目的，尽管通常这只是碰撞避免算法的副作用，而不是有意为之。

将群体移动与群体密度纳入到寻路计算中的研究已经完成了。根据群体密度优化寻路的方法并没有将群体的移动或移动方向考虑进去，这会导致矫枉过正的现象，如下图所示：



Congestion maps在很多方面与现有的合作寻路算法类似，例如Direction Maps（DMs），但是在几个关键方面略有不同。DMs使用平均群体移动来鼓励角色跟随群体移动。因此，很多Congestion maps方法中会局部振动的地方被平滑地解决了。另一方面，这种随时间平滑的方法无法让DMs在环境突然变化时实现快速精准的响应。Congestion maps与DMs都使用了总体人群移动趋势的信息，以强化寻路规划过程，这点是类似的。但是，Congestion maps会考虑到每个个体的大小与形状，而DMs通常假设所有个体是一致的。

DMs和Congestion maps的最后一个重要区别在于，Congestion maps将移动惩罚的比重与人群密度关联了起来。由于没有考虑密度，DMs显示出过于悲观的寻路行为，它会鼓励个体移动时围绕阻挡了路径的群体，就算阻挡的群体比较稀疏。

8.5 总结

本章探索了一些电子游戏中流行的群体交互系统的例子，并且认识到规划每一种能想到的交互的重要性，因为这正是把几行简单的代码转变成拟真游戏的方法。为总结这一章的内容，我们重新审视了高级寻路系统，学习了游戏中的多个角色拥有同一个最终目标时如何处理，比如寻找另一条路来避免碰撞，或是排队等待，让其他角色优先前进。

下一章会讨论AI计划与决策。我们将看到如何让AI预测事物，在到达一点之前，或是面对一个特定问题之前，提前知道应该怎么做。

第9章　AI规划与碰撞避免

本章涵盖的内容将会帮助我们把AI角色带向更高、更复杂的层次。本章的理念是赋予角色规划与决策的能力。之前的章节已经探索过一部分必要的技术知识，现在将继续深入研究如何创建能提前做出规划并决策的AI角色。

9.1 搜索

我们从讨论电子游戏中的搜索行为开始。搜索可能是角色做出的第一个决策，因为大部分情况下，我们总是希望角色搜索一些东西，可能是玩家，也可能是能让角色胜利的其他事物。

让角色能够顺利地找到某样东西非常有用，也极其重要。这个特性在大部分电子游戏中都可以找到，我们会想要用到它的。

正如我们在前面章节内的例子中所见，大部分情况下，玩家在地图上四处走动。当他遇到一个敌人，敌人就从空闲状态切换到进攻状态。现在我们想要敌人主动一些，持续搜索玩家而不是守株待兔。这里我们可以从思考敌人搜索玩家所需的必要过程开始。首先，这个过程需要计划，而计划需要提前装在AI的头脑中。一般来讲我们希望AI的思维方式和我们的一样，因为这样看起来更真实，也正是我们想要的效果。

其他时候，即当角色有其他主要目标时，搜索可能是次要的。这在即时战略游戏中很常见，AI角色会探索地图，并在某些特定点发现敌人基地。在这里搜索并不是这种AI的优先目标，尽管如此，这依然是游戏的一部分——探索地图并获得对手的位置。在发现了玩家位置以后，AI角色可以决定是继续深入探索其他部分，还是放弃探索另外决定接下来要做什么。

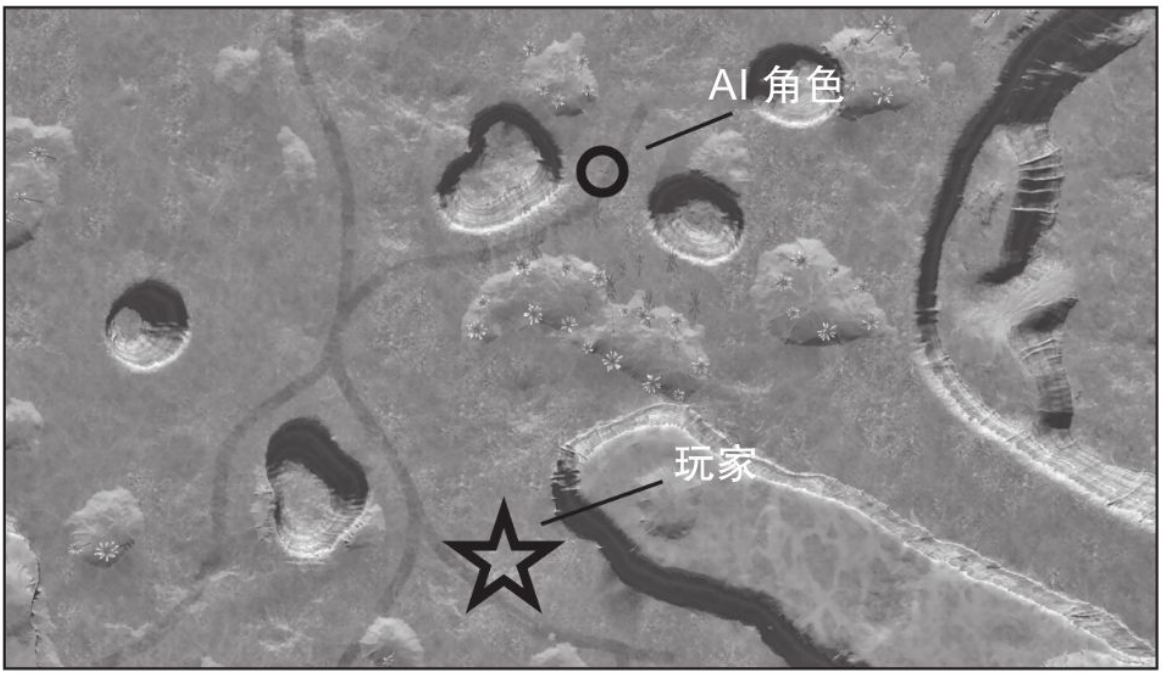
同样，我们可以为打猎游戏创建拟真的动物，例如让动物的主要目标是进食和喝水，这样它们就不得不一直搜索食物和水，一旦它们吃饱喝足，则会搜索温暖舒适的地方并待在那里。但是，一旦动物发现了猎人，它们的优先级会立即发生改变，变为搜索安全的位置。

有很多决策可以依赖于搜索系统，这也是一种模拟真实世界中人和动物的方法。下面讨论的会涵盖大部分游戏中常见的搜索类型，最终目标是让AI角色能够搜索并找到任何东西。

积极搜索

第一种搜索类型是积极搜索。积极搜索，意思是让搜索成为AI角色的主要目标。游戏中的角色出于某种原因必须找到玩家，类似于捉迷藏，其中一个玩家要去抓其他藏起来的玩家。

在这里，有一张地图可供角色随意走动，只是需要注意避免碰撞（树、山和岩石）。



因此，第一步是创建一个系统，角色能够在上面走动。在这个例子里，我们选择创建一个路点（waypoint）系统，角色可以从一个点走到另一个点，探索整个地图。

导入要用到的地图和角色之后，需要设置路点以便让角色知道需要去哪里。我们固然可以手动把路点坐标输入到代码里，但是为了简化过程，这里将在场景里创建物体并删除其3D模型来作为路点。

现在，我们把所有路点编在一组里，命名为waypoints。把这些路点放好以后，就可以开始编写代码来告诉角色有多少路点需要跟随了。这个代码非常有用，因为此方法可以用来创建不同的地图，而且需要多少路点就放多少而不必修改角色的代码：

```
public static Transform[] points;

    void Awake ()
    {
        points = new Transform[transform.childCount];
        for (int i = 0; i < points.Length; i++)
```

```
        {
            points[i] = transform.GetChild(i);
        }
    }
```

这个代码挂在刚才创建的组的对象上，它会计算出路点总数并给路点排序。

上图中的蓝色球体就是用来代表路点的3D模型，在这个例子里，角色会跟随8个点移动直到它走完整个路径。现在，我们转到AI角色的代码，看看如何让AI角色在路点间移动。

我们从创建角色的基本属性开始——血量和速度——然后我们创建新的变量，用来给它下一个位置的相关信息，除此之外，还有一个变量用来表示正在跟随的路点的编号。

```
    public float speed;
    public int health;

    private Transform target;
    private int wavepointIndex = 0;
```

现在，我们已经有了一些基本的变量，这些变量足以让敌人角色一个接一个路点地移

```
private float speed;
public int health;

private Transform target;
private int wavepointIndex = 0;


void Start ()
{
     target = waypoints.points[0];  speed = 10f;
}

void Update ()
{
     Vector3 dir = target.position - transform.position;
     transform.Translate(dir.normalized * speed * Time.deltaTime,
Space.World);
```

动，直到发现玩家。让我们看看怎样使用它们来让游戏运作：

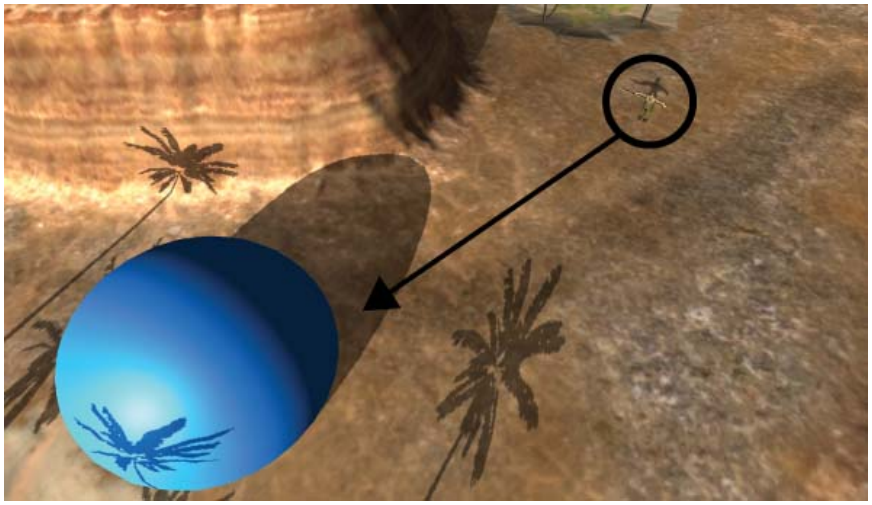
```
        if(Vector3.Distance(transform.position, target.position) <= 0.4f)
        {
            GetNextWaypoint();
        }
    }

    void GetNextWaypoint()
    {
        if(wavepointIndex >= waypoints.points.Length - 1)
        {
            Destroy(gameObject);
            return;
        }

        wavepointIndex++;
        target = waypoints.points[wavepointIndex];
    }
```

在Start函数里，我们给变量赋值了角色需要跟随的第一个路点，也就是waypoints.points[0]，前面创建的路点列表的第一个元素。除此之外还确定了角色的移动速度，在这个例子里是10。

之后，在Update函数里，角色会使用Vector3类型的变量dir计算当前的路点到下一个路点的距离。由于角色会持续保持移动的状态，所以这里创建了角色的移动代码，用到transform.Translate方法。知道了距离和速度信息，角色就知道了离下一个位置有多远，一旦它走过了需要的距离，它就会再去寻找下一个点。为了实现这一部分，我们使用一个if语句来告诉角色是否离目标点的距离小于0.4，如果满足条件则表示他已经到达了目标点，可以向新的目标点移动了，新的目标点通过函数GetNextWaypoint（）得到。

在GetNextWaypoint（）函数中，角色会检查是否已到达最后一个路点，如果是，那么路标物体被销毁；如果不是，就跟随下一个路点。waypointIndex++语句会在角色每到达一个路点时让变量值加1，像0>1>2>3>4>5这样。

现在我们把代码挂在角色身上，然后把角色放在起点位置开始测试，看看是否能正常工作：



现在角色从一个点走到另一个点，这是开发搜索系统必需的第一步——角色需要能在整个地图上移动。现在我们只需要让它转向前进的方向，之后再考虑搜索的问题：

```
public float speed;
public int health;
public float speedTurn;

private Transform target;
private int wavepointIndex = 0;


void Start ()
{
     target = waypoints.points[0];
     speed = 10f;
     speedTurn = 0.2f;
}

void Update ()
{
     Vector3 dir = target.position - transform.position;
     transform.Translate(dir.normalized * speed * Time.deltaTime,
Space.World);

     if(Vector3.Distance(transform.position, target.position) <= 0.4f)
     {
         GetNextWaypoint();
     }

     Vector3 newDir = Vector3.RotateTowards(transform.forward, dir,
speedTurn,
     0.0F);

     transform.rotation = Quaternion.LookRotation(newDir);
}

void GetNextWaypoint()
{
     if(wavepointIndex >= waypoints.points.Length - 1)
     {
         Destroy(gameObject);
         return;
     }

     wavepointIndex++;
     target = waypoints.points[wavepointIndex];
}
```

现在，我们已经让角色面向了正确的方向，接下来添加搜索系统。

我们已经有了能够沿着路点移动的角色，不过现在就算它发现了玩家，也不会停下脚步，什么也不会发生。这就是我们现在需要补充的。

为了达成想要的目标，我们尝试在角色周围添加一个圆形触发区域，如下图所示。这个角色会在地图上走动，当触发器检测到玩家时，角色就找到了它的首要目标。来添加这部分代码：

```
public float speed;
public int health;
public float speedTurn;
private Transform target;
private int wavepointIndex = 0;
private bool Found;

void Start ()
{
    target = waypoints.points[0];
    speed = 10f;
    speedTurn = 0.2f;
}

void Update ()
{
    Vector3 dir = target.position - transform.position;
    transform.Translate(dir.normalized * speed * Time.deltaTime,
    Space.World);

    if(Vector3.Distance(transform.position, target.position) <= 0.4f)
    {
        GetNextWaypoint();
    }

    Vector3 newDir = Vector3.RotateTowards(transform.forward, dir,
    speedTurn, 0.0F);

    transform.rotation = Quaternion.LookRotation(newDir);
}

void GetNextWaypoint()
{
    if(wavepointIndex >= waypoints.points.Length - 1)
    {
        Destroy(gameObject);
        return;
    }

    wavepointIndex++;
    target = waypoints.points[wavepointIndex];
}
```

```
void OnTriggerEnter(Collider other)
{
    if(other.gameObject.tag =="Player")
    {
        Found = true;
    }
}
```

所以，现在我们添加了OnTriggerEnter函数验证触发区域是否接触到其他物体。要检查触发区域碰到物体是否是玩家，我们用了if语句来检查碰到的物体是否具有Player标签。如果为真，则布尔变量Found变为true。这个布尔变量接下来会非常有用。

让我们测试游戏，检查角色在走过玩家时，变量Found是否会从false变为true：

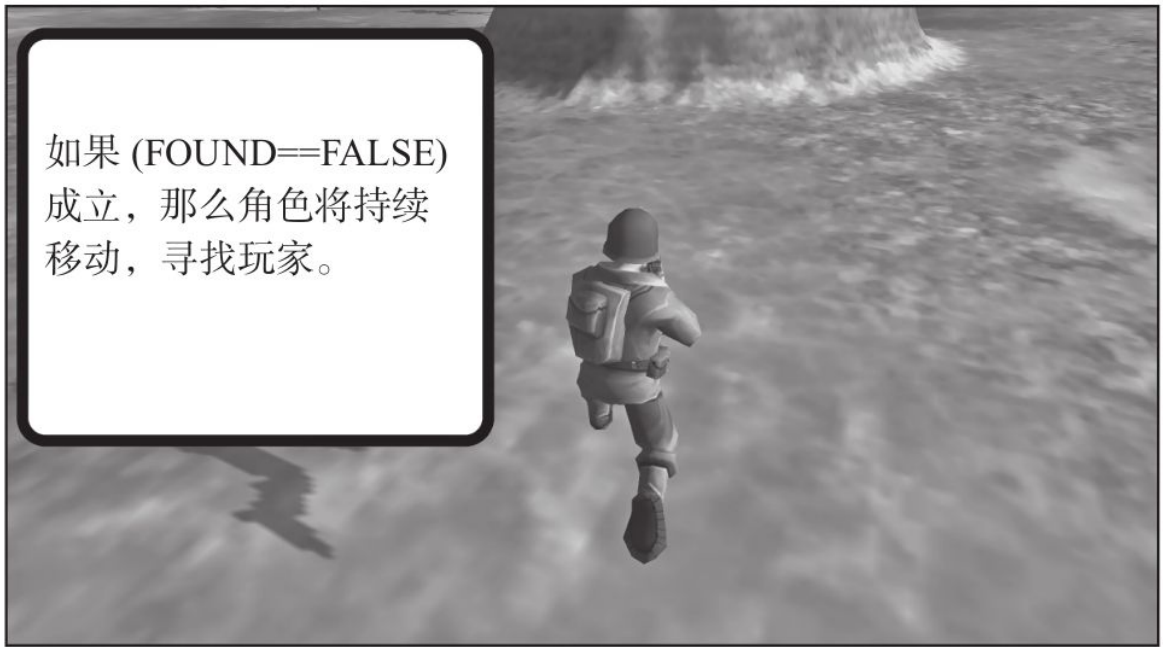
刚刚实现的搜索系统工作得很好，角色会在整个地图上移动搜索玩家，也能够准确地找到玩家。下一步是告诉角色在发现玩家后停止搜索。

```
public float speed;
public int health;
public float speedTurn;
private Transform target;
private int wavepointIndex = 0;
public bool Found;

void Start ()
{
    target = waypoints.points[0];
    speed = 40f;
    speedTurn = 0.2f;
}

void Update ()
{
    if (Found == false)
    {
        Vector3 dir = target.position - transform.position;
        transform.Translate(dir.normalized * speed *
        Time.deltaTime,
        Space.World);
        if (Vector3.Distance(transform.position, target.position)
        <= 0.4f)
```

```
            {
                GetNextWaypoint();
            }

            Vector3 newDir = Vector3.RotateTowards(transform.forward,
            dir,
            speedTurn, 0.0F);

            transform.rotation = Quaternion.LookRotation(newDir);
        }
    }

    void GetNextWaypoint()
    {
        if(wavepointIndex >= waypoints.points.Length - 1)
        {
            Destroy(gameObject);
            return;
        }

        wavepointIndex++;
        target = waypoints.points[wavepointIndex];
    }

    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            Found = true;
        }
    }
```

在最后的修改之后，我们已经获得了一个AI角色，它能够在地图上持续走动，直到找到玩家。而当它最终找到了玩家后，会停止移动并计划接下来怎么做。

这里做的是使用布尔变量Found来确定角色是否应当继续寻找玩家。



上图展示了某个时刻角色的状态，我们已经准备好了实现更多功能来让它们具有决策能力并选择最优方案。

这个搜索系统可以应用于大量不同的游戏类型，设置起来也非常迅速，这让规划AI角色有了一个完美的开端。现在让我们继续，开发预测玩家的功能。

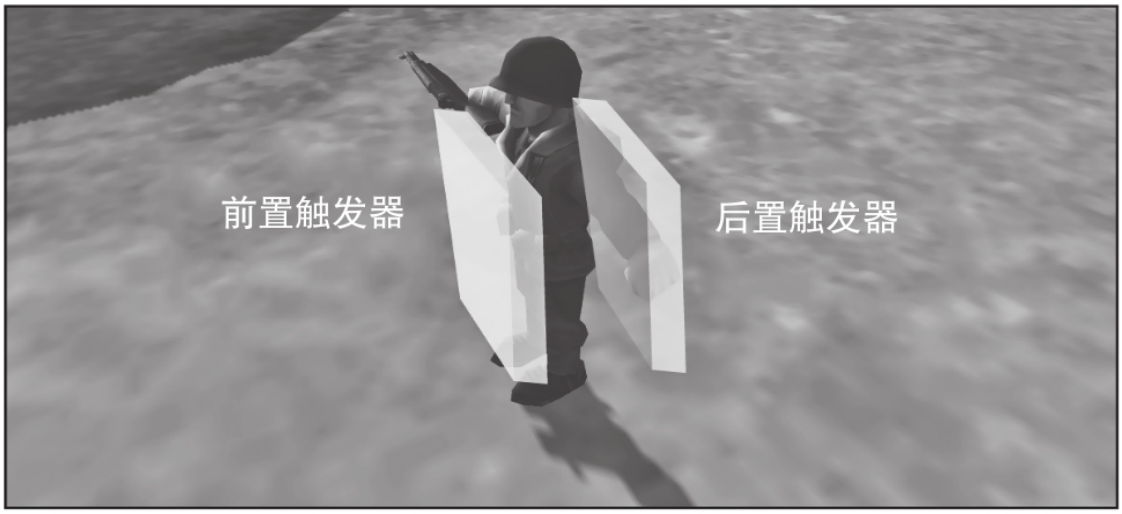
预测对手行动

现在，我们要令角色能够预测将要发生的事情，甚至是在对抗玩家之前。这是AI要开始计划最优的可用选择来达成目标所不可或缺的一部分。

让我们看看如何为AI实现一个预测系统。我们会继续使用前面的例子，一个士兵在地图里搜索另一个人。在这里，有一个角色在地图上四处移动，在发现玩家时停止。

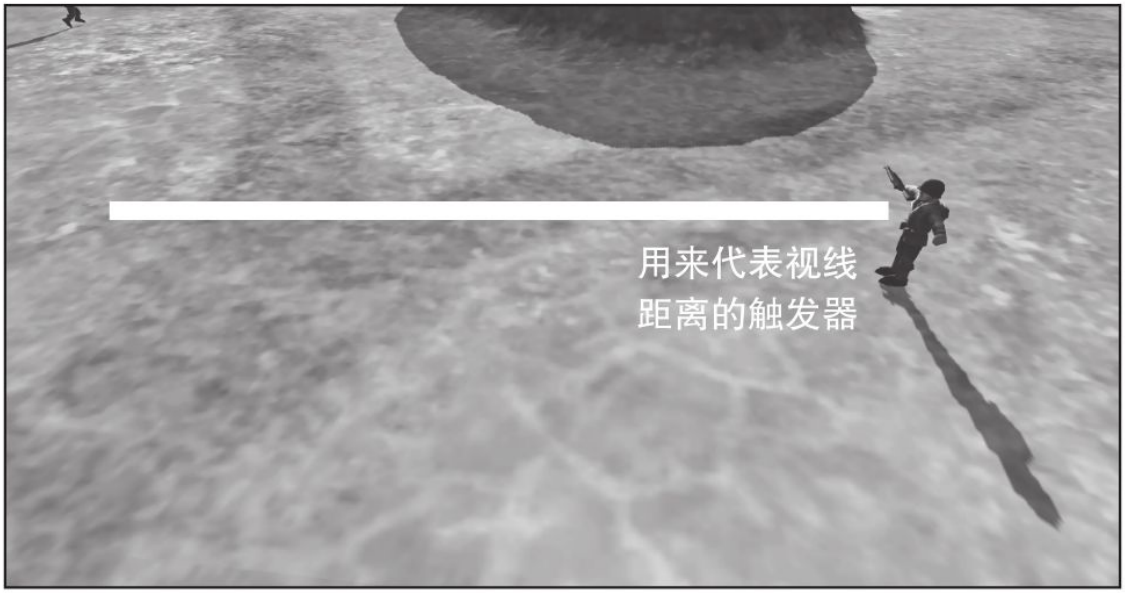
如果角色发现了玩家，最有可能的情况是玩家同时也发现了AI角色，这样双方都发觉了对方的存在。玩家攻击AI角色的概率有多大？有多大概率玩家缺少足够的弹药？这些都非常主观且不好预测。但是，我们希望角色记住它们来预测玩家可能的行为。

从一个简单的问题开始：玩家的脸朝向角色吗？让角色检查这一点对做出判断非常有帮助。为了达到这个目标，我们添加一个触发器放在角色背部，再添加一个放在角色前面，包括玩家角色也这样做，如下图所示：

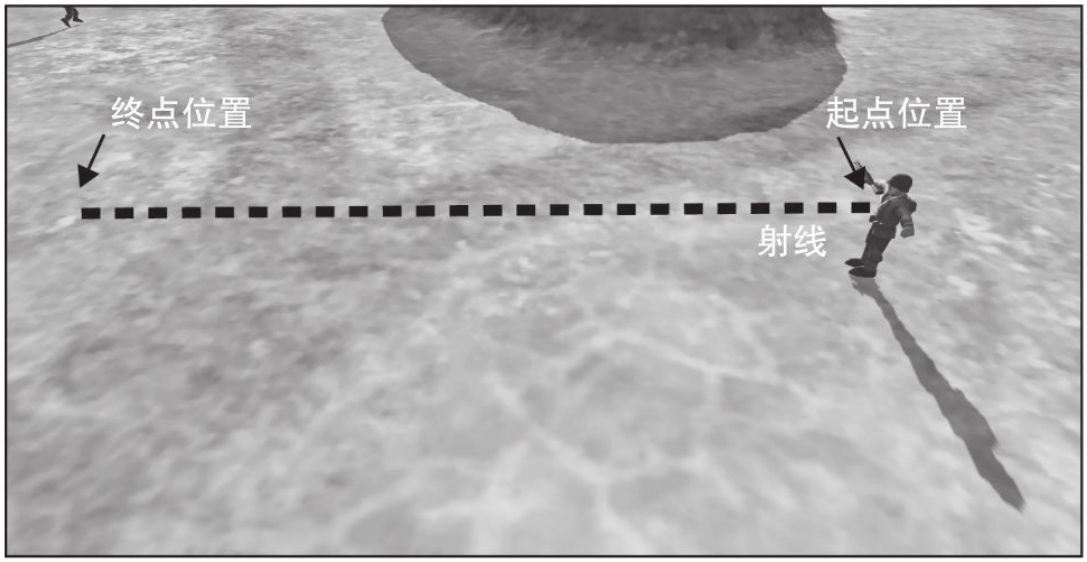


在角色前后放上触发器是为了帮助另一个角色识别看到的是前面还是后面。所以，我们为游戏中的每个角色添加触发器，并将触发器命名为back和front。

现在，让角色能区分出后置和前置触发器，这可以通过两种方式做到——第一种是在角色前面添加一个拉长的触发器代表观察范围：



另外一种方法，可以创建从角色位置到视野终点的射线，如下图所示：



两种方法各有优缺点。再一次强调，我们不必一直使用复杂的方法来获得更好的结果。所以，这里的建议是使用更舒服的方法，在此例中使用触发器来代表视野范围可能是一个好主意。

我们添加角色的前置触发器，之后编写代码来让它能够检测到角色的前面或后面。首先需要修改的地方是：当角色发现玩家时，让角色面对玩家。角色不面对玩家的话是无法做出任何预测的，所以我们先修改这一点：

```
void Update ()
     {
         if (Found == false)
         {
             Vector3 dir = target.position - transform.position;
             transform.Translate(dir.normalized * speed *
             Time.deltaTime,
             Space.World);

             if (Vector3.Distance(transform.position, target.position)
             <= 0.4f)
        {
            GetNextWaypoint();
        }

        Vector3 newDir = Vector3.RotateTowards(transform.forward,
        dir,
        speedTurn, 0.0F);

        transform.rotation = Quaternion.LookRotation(newDir);
    }

    if (Found == true)
    {
        transform.LookAt(target);
    }
}

void GetNextWaypoint()
{
    if(wavepointIndex >= waypoints.points.Length - 1)
    {
        Destroy(gameObject);
        return;
    }
```

```
        wavepointIndex++;
        target = waypoints.points[wavepointIndex];
    }

    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            Found = true;
            target = other.gameObject.transform;

        }
    }
```

现在AI角色已经能够在发现玩家时持续面向玩家了。要让代码正常工作，我们在if（found==true）判断内部添加了第一行代码。这里使用了transform.LookAt函数来让AI面向玩家角色。当玩家被玩家发现时，玩家将自动变成目标：

现在，AI角色已经面朝玩家了，接下来检查是面对玩家的前方还是后方。

对我们来说，角色竟然不知道看到的是玩家的后面还是前面，这有点无法理解，但是在开发AI角色时，所有的东西都要写在代码里，特别是这种对预测、计划、决策至关重要的细节。

所以，我们用之前添加的触发器来检查AI角色到底面朝玩家的后面还是前面。先从添

```
public bool facingFront;
public bool facingBack;
```

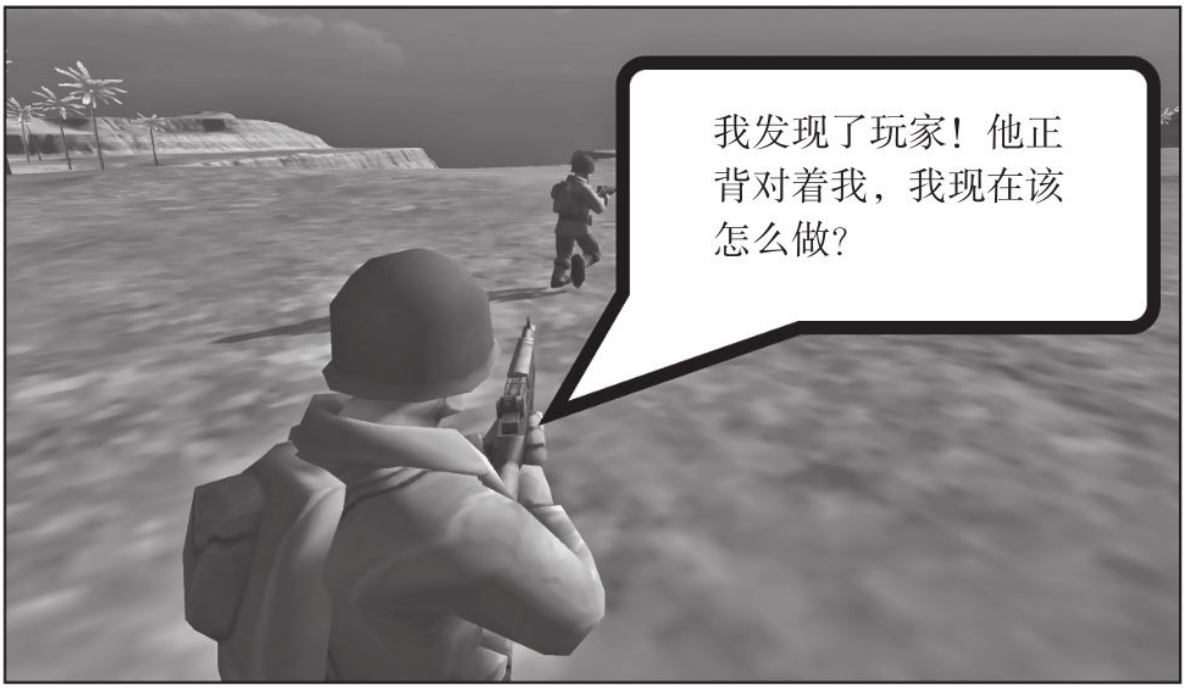
加以下变量开始：

这些变量是布尔类型的facingFront和facingBack。触发器会设置其中之一为true，那样，角色AI就知道了他面向的是哪一面。接着，我们来配置触发器：

```
void Update ()
     {
         if (Found == false)
         {
        Vector3 dir = target.position - transform.position;
        transform.Translate(dir.normalized * speed *
        Time.deltaTime,
```

```
            Space.World);

            if (Vector3.Distance(transform.position, target.position)
            <= 0.4f)
            {
                GetNextWaypoint();
            }

            Vector3 newDir = Vector3.RotateTowards(transform.forward,
            dir,
            speedTurn, 0.0F);

            transform.rotation = Quaternion.LookRotation(newDir);
        }

        if (Found == true)
        {
            transform.LookAt(target);
        }
    }

    void GetNextWaypoint()
    {
        if(wavepointIndex >= waypoints.points.Length - 1)
        {
            Destroy(gameObject);
            return;
        }

        wavepointIndex++;
        target = waypoints.points[wavepointIndex];
    }

    void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            Found = true;
            target = other.gameObject.transform;
        }

        if(other.gameObject.name == "frontSide")
        {
            facingFront = true;
            facingBack = false;
        }

        if(other.gameObject.name == "backSide")
        {
            facingFront = false;
            facingBack = true;
        }
    }
```

这里做的事情是让触发器检查是否碰撞到了玩家的后方或前方。为达到这个目的，我们让触发器一直询问是否检测到了碰撞、碰撞的是不是frontSide对象或者backSide对象。它们之中同时只能有一个为true。



现在，角色已经能够区别出玩家的后面和前面了，我们希望它还能分析两种情况可能会导致的危险。所以，根据看到的是角色的前面还是后面，我们要做的第一件事会有极大的不同。如果面对的是前面，玩家已经准备好了射击我们的AI角色，那么这是更危险的情况。我们创建一个危险度表并将这种情况添加进去：

```
public float speed;
public int health;
public float speedTurn;
private Transform target;
private int wavepointIndex = 0;
public bool Found;
public bool facingFront;
public bool facingBack;

public int dangerMeter;
```

在变量定义的这段，我们添加了一个新的整型变量dangerMeter。现在，我们将为变量赋值以辅助判断AI角色被攻击的风险是高还是低：

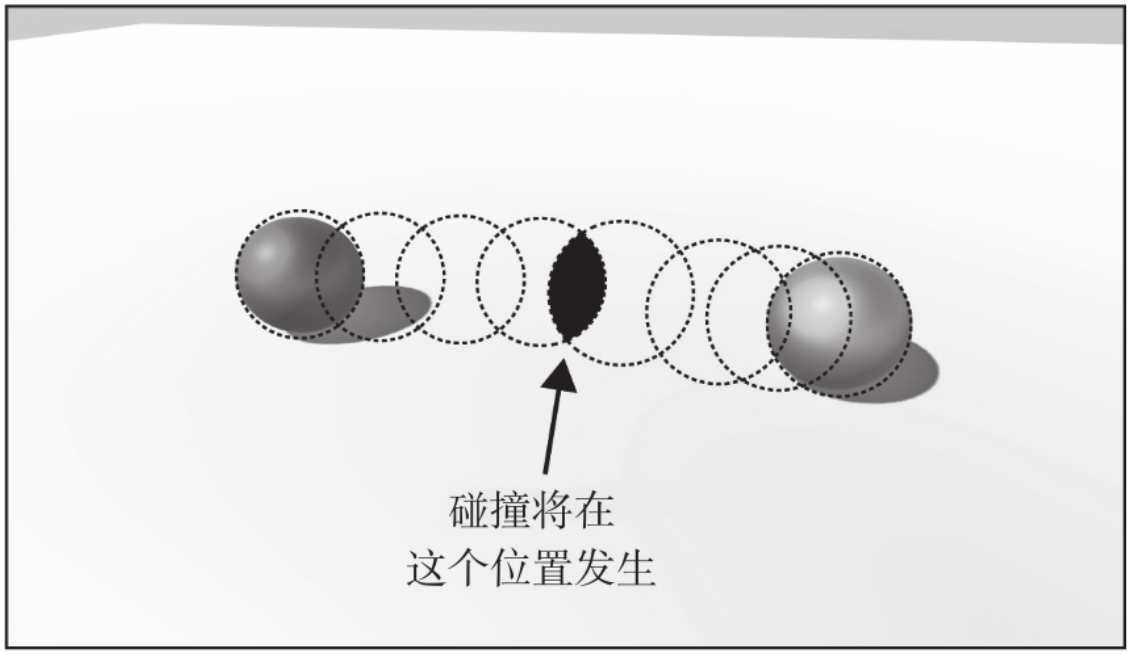
```
void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            Found = true;
            target = other.gameObject.transform;
        }

        if(other.gameObject.name == "frontSide")
        {
            facingFront = true;
            facingBack = false;
            dangerMeter += 50;
        }

        if(other.gameObject.name == "backSide")
        {
            facingFront = false;
            facingBack = true;
            dangerMeter += 5;
        }
    }
```

所以，根据当前情况，我们加上一个较小的数来代表一个小的风险，或者一个较大的数代表大的风险。如果危险度很高，AI角色会预感到生命受到威胁，因此可能会做出一些比较有戏剧性的决定。另一方面，如果角色面对的是低危险度的情况，它可以做出更细致、更有效的计划。

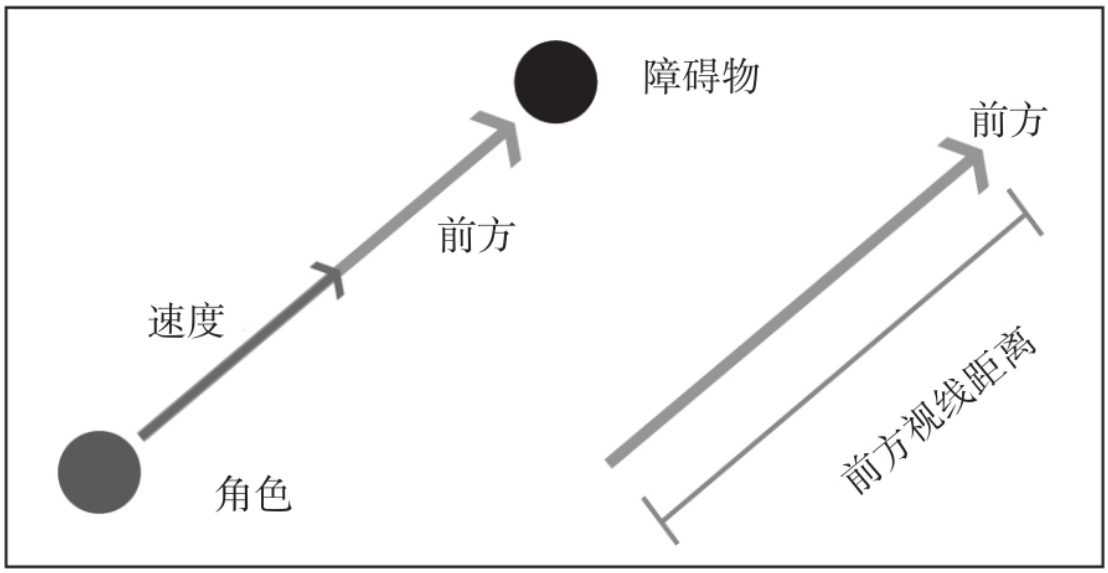
实际上，有更多的因素可以被考虑到dangerMeter中，如角色相对玩家所处的位置。为实现这个，我们需要将地图划分为不同的区域，并为每个区域指定不同的风险等级。例如，如果角色在森林中，那么可以认为其处在一个较低风险的区域，而如果在一片开阔场地，则被认为是处在更高风险的区域。其他诸如持有多少弹药、剩多少血量等等，很多要素都可以被添加至dangerMeter当中。补全这些要素可以帮助到角色更好地预测到可能发生在自己身上的各种情形。

碰撞避免

预测碰撞对实现AI角色来说是一个很有用的方法，它也可以用在集群系统中来让群落更有组织地移动，就像在之前的章节中，一个角色可以根据另一个角色的方向来移动。现在，让我们试试用一个简单的方式来实现这个功能：



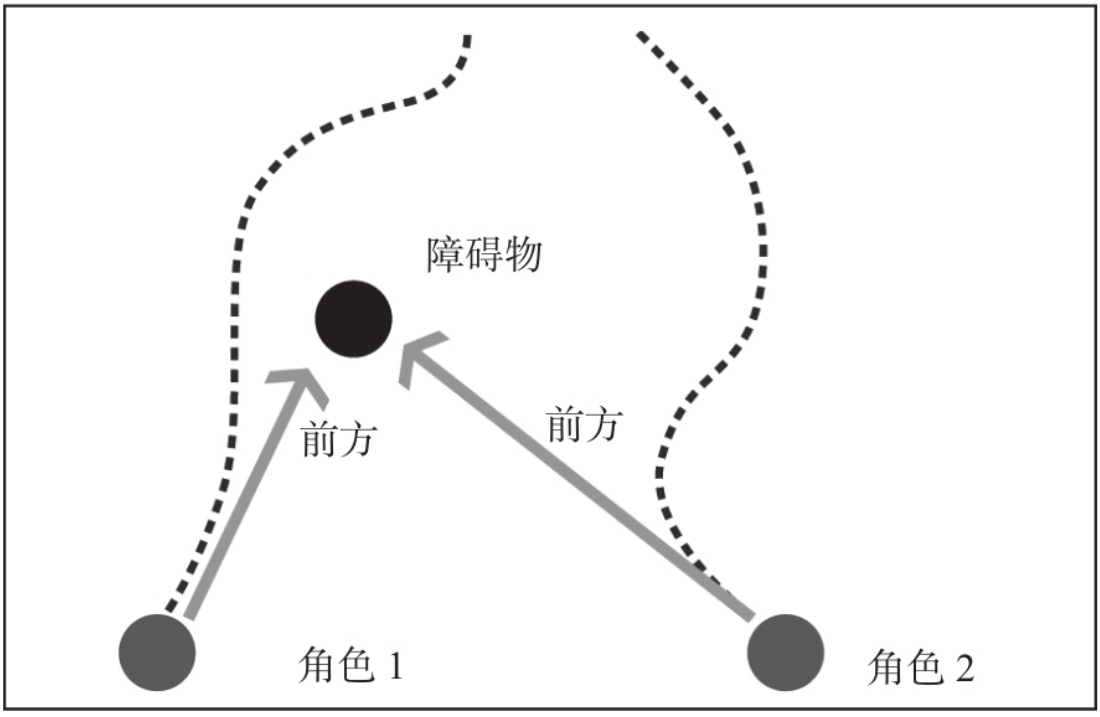
要预测碰撞，至少需要两个物体或角色。在上图中，有两个球代表两个角色，虚线代表它们的移动轨迹。如果蓝色球体向着红色球体移动，在某个点上，它们会碰撞到对方。这里的主要目标是预测碰撞何时发生并调整球的运动轨迹，以达到避免碰撞的目的。



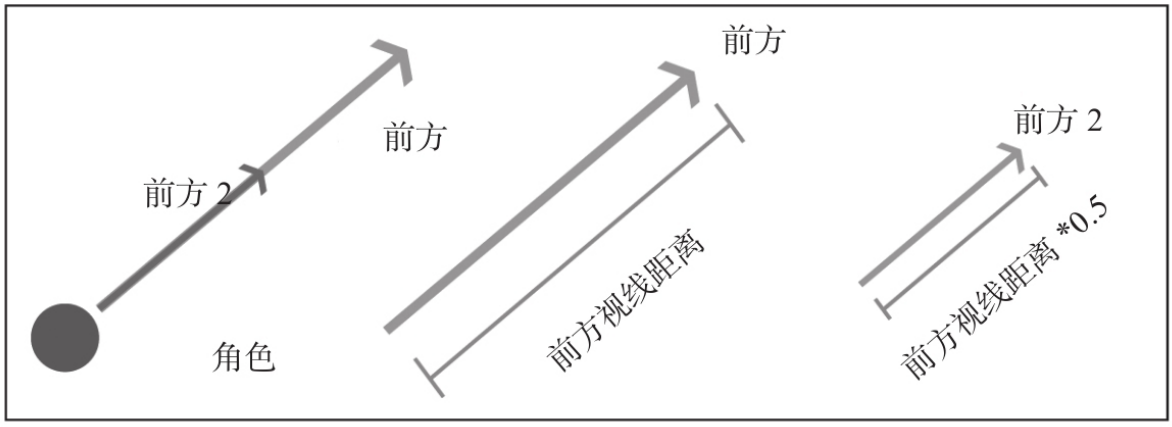
上图中展示的正是我们为了避免角色碰撞到障碍物所需要做的。我们需要知道速度向量，它可以代表角色的方向，同时它也用来生成一个新的向量ahead，它是速度向量的拷贝，但是具有更长的长度。解释一下，ahead向量代表了角色的视线，一旦看到了障碍物，它就会调整前进的方向以避免碰撞。计算ahead向量的方法如下：

```
ahead = transform.position + Vector3.Normalize(velocity) * MAX_SEE_AHEAD;
```

在上式中，ahead和velocity均为Vector3类型变量，MAX_SEE_AHEAD是一个浮点类型变量用来表示角色能看多远。如果我们增大MAX SEE AHEAD的值，角色就会更早调整它的方向，如下图所示：



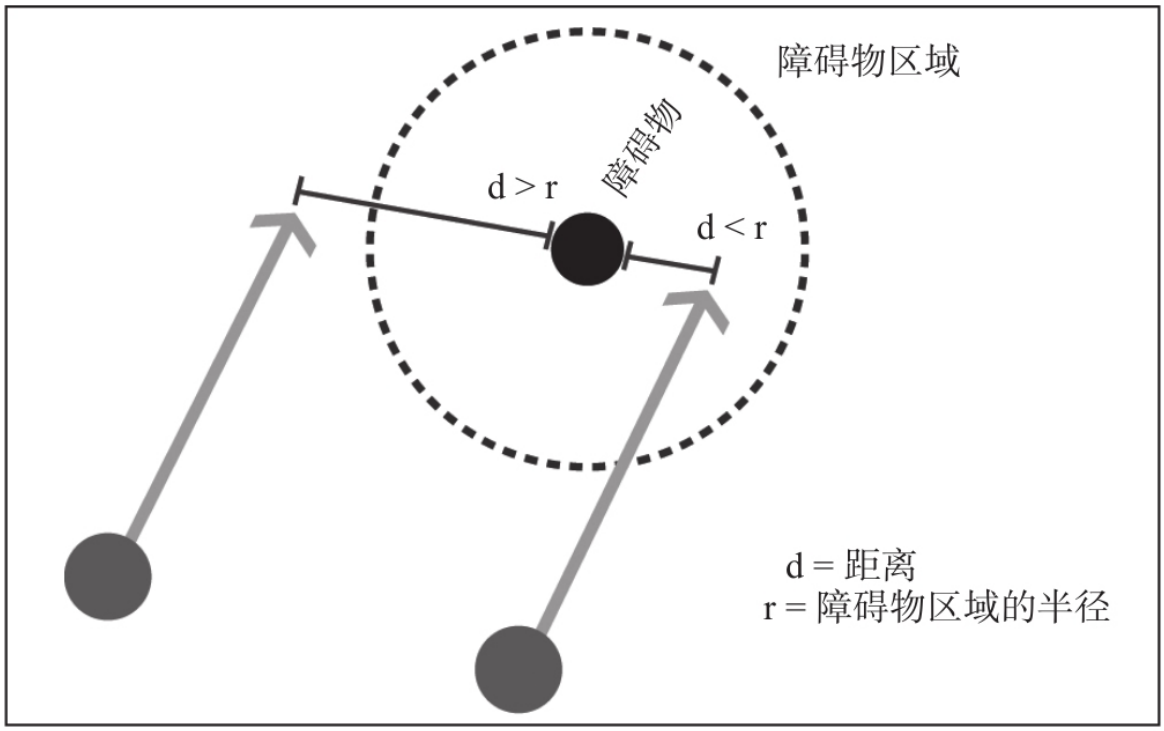
为了检测碰撞，一种解决方案是使用线球相交，线即ahead向量，球即障碍物。这个方法有效，但是我们要用一种更简化、更容易理解的方式，且能得到相同的结果。向量ahead要与另一个向量相乘，这个向量的长度是ahead的一半。



从上图中可以看到，ahead和ahead2方向相同，唯一的区别是它们的长度。

```
ahead = transform.position + Vector3.Normalize(velocity) * MAX_SEE_AHEAD;
ahead2 = transform.position + Vector3.Normalize(velocity) * (MAX_SEE_AHEAD
* 0.5);
```

我们需要检测碰撞，以便知道这两个向量是否在障碍物区域。要算出这个结果，我们可以比较向量到障碍物中心的距离。如果距离小于等于障碍物半径，意味着向量在障碍物区域内，即检测到了一处碰撞。



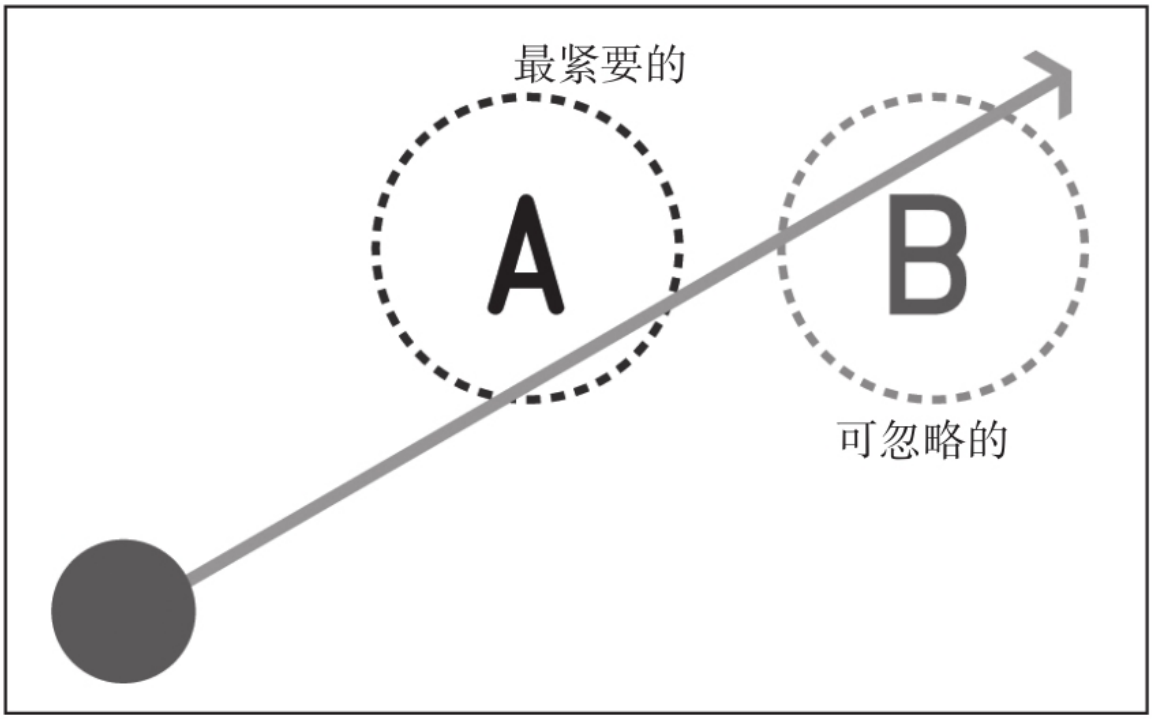
为了简化起见，向量ahead2没有出现在上图中。

如果两个向量任意一个与障碍物区域相交，这意味着障碍物挡住了路径，为了解决这个问题，我们将计算两点之间的距离：

```
 public Vector3 velocity;
 public Vector3 ahead;
 public float MAX_SEE_AHEAD;
 public Transform a;
 public Transform b;

 void Start (){

     ahead = transform.position + Vector3.Normalize(velocity) *
MAX_SEE_AHEAD;
     }

 void Update ()
 {

     float distA = Vector3.Distance(a.position, transform.position);
     float distB = Vector3.Distance(b.position, transform.position);

     if(distA > distB)
     {
         avoidB();
     }

     if(distB > distA)
     {
         avoidA();
     }
     }

     void avoidB()
     {

     }
```

```
                void avoidA()
                {

                }
        }
```

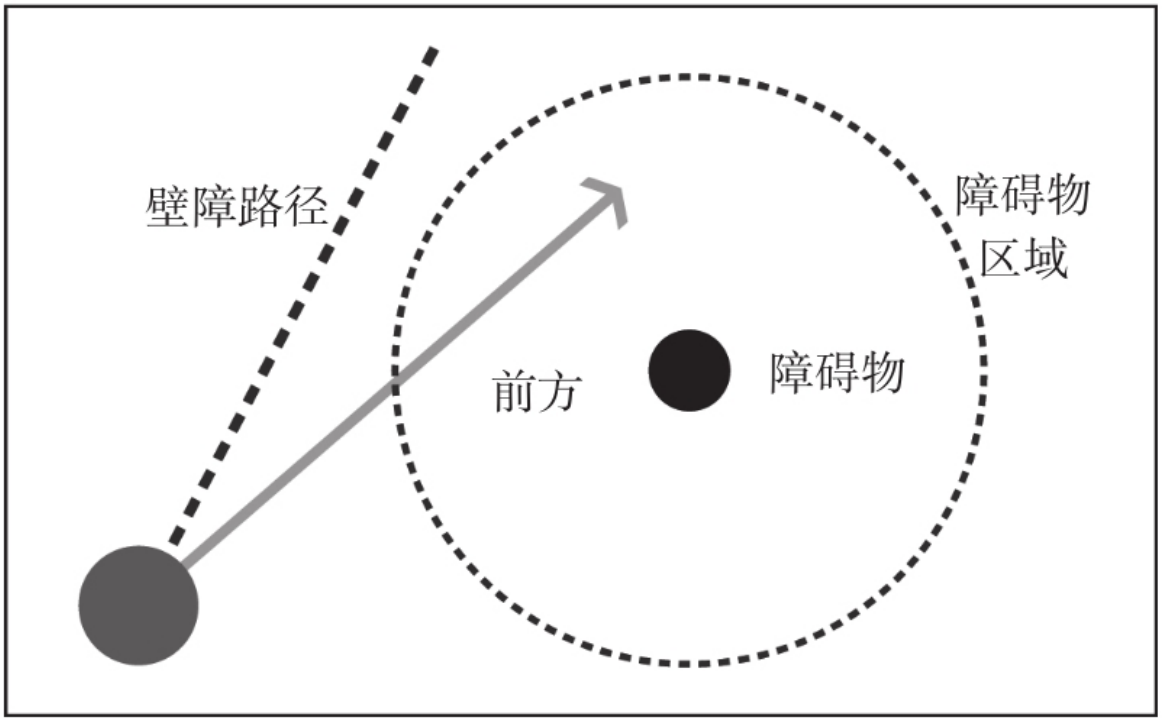
对有两个障碍物阻挡路径的情况，我们需要检查哪一个离角色更近，之后就要先考虑避免较近的障碍物，再考虑较远的：



最近的障碍物具有最大的威胁，先对它进行计算。现在，我们看看怎样计算以进行规避：

```
 public Vector3 velocity;
 public Vector3 ahead;
 public float MAX_SEE_AHEAD;
 public float MAX_AVOID;
 public Transform a;
 public Transform b;
 public Vector3 avoidance;

 void Start () {

     ahead = transform.position + Vector3.Normalize(velocity) *
MAX_SEE_AHEAD;
 }

 void Update ()
 {

     float distA = Vector3.Distance(a.position, transform.position);
     float distB = Vector3.Distance(b.position, transform.position);

     if(distA > distB)
     {
         avoidB();
     }

     if(distB > distA)
     {
         avoidA();
     }
 }



  void avoidB()
  {
      avoidance = ahead - b.position;
      avoidance = Vector3.Normalize(avoidance) * MAX_AVOID;
  }

  void avoidA()
  {
      avoidance = ahead - a.position;
      avoidance = Vector3.Normalize(avoidance) * MAX_AVOID;
  }
 }
```

计算好avoidance向量之后，将其标准化再缩放到MAX AVOID的大小，MAX_AVOID用来定义规避距离的长度。MAX AVOID值越大，规避距离就越长，角色就被推到离障碍物越远。




任何实体的位置都可以用向量表示，所以它们可以用在和其他向量、力的计算中。

现在，角色能够做预测，也能在遇到障碍物时转向以避免碰撞，我们已经有了实现它们的基本框架。将这种方法与寻路相结合，可以让角色任意在地图上自由地移动。

9.2 总结

本章中，我们已经探索了怎样让我们的AI角色创建并执行一个计划，以完成一个确定的目标。核心思路是让角色提前思考将会发生什么情况，并做出准备。为完成这个目标，我们还探索了怎样让AI角色提前预测到和障碍物或另一个角色的碰撞。这个方法不仅是让角色在地图上自由移动的基础，而且还是规划该做什么事情的一个标准。下一章里，我们将讨论感知，即如何开发潜入类型的游戏里最标志性的功能，同时通过拟真视野的机制，让AI角色具有自行感知周围环境的能力。

第10章　感知

最后一章中，我们会看到如何开发使用战术与感知来达成目标的AI。这里会用到所有之前学到的东西，了解如何结合所有技术来创建人工智能角色，它可以被用于潜入类游戏或其他依赖战术与感知的游戏。

10.1 潜入类游戏

潜入类是一个非常流行的游戏子类型，游戏的主要目标是玩家借助潜入元素，保持不被发现的状态以完成主要任务。它不仅是一个军事游戏中广泛流行的子类型，甚至也可以在几乎所有游戏中看到它的应用。深入地看，任何游戏中，如果敌人会被噪声触发、或者因看到玩家而触发，都是应用了潜入元素。这也意味着，在AI中实现感知或者战术会非常有用，不论我们在开发哪一种游戏类型。

10.2 关于战术

战术是角色或小队用来达成特定目标的过程。它通常意味着角色可以使用所有能力、根据情景选择其中最好的来击败敌人。电子游戏中战术的概念是让AI具有决策的能力，让它能够聪明地行动以达到主要目标。我们可以拿游戏中的战术与真实的士兵或警察抓住坏人的战术做比较。

真实世界中，他们有各种各样的技术与资源来抓住强盗，但是为了成功完成任务，他们需要聪明地选择具体要做什么，一步一步制定计划。同样的原理可以用于AI角色，我们要让AI选择最佳可用方案去达成目标。

要实现这点，我们可以利用本书中前面介绍的所有方法，依靠它们可以开发出能够选择最佳战术的AI，它能够打败玩家或者达成自己的目标。

10.3 关于感知

与战术相关的非常重要的一点是角色的感知。一些常见的因素组成了感知，例如声音、视野和知觉。这些因素基于人类都有的基本功能，依靠视觉、听觉、触觉和知觉来感受周围发生的事情。

因此，我们正在寻找的创造人工智能角色的方法，能够让他在处理所有信息的同时也能感知周围，以做出当时更好的决策。

10.4 实现视觉感知

在开始实现战术之前，我们先看看在角色身上怎样实现感知系统。

我们从实现视觉感知开始。基本想法是模拟人类的视觉，我们在近处看得很清楚，很远的地方就不那么清晰了。很多游戏实现了这种系统但又都有不同，某些游戏具有复杂的系统而某些游戏就只有比较基本的系统。基本系统的例子在面向低年龄的游戏中更容易找到，比如《塞尔达：时之笛》，其中敌人只会在你到达特定的触发区域时才会做出反应，

如下图所示：

举个例子，在玩家后退并退出敌人触发区域的时候，敌人会在原地停住，即便它显然能看见玩家。这就是感知系统的基本形式，我们可以把它算作感知的视觉部分。

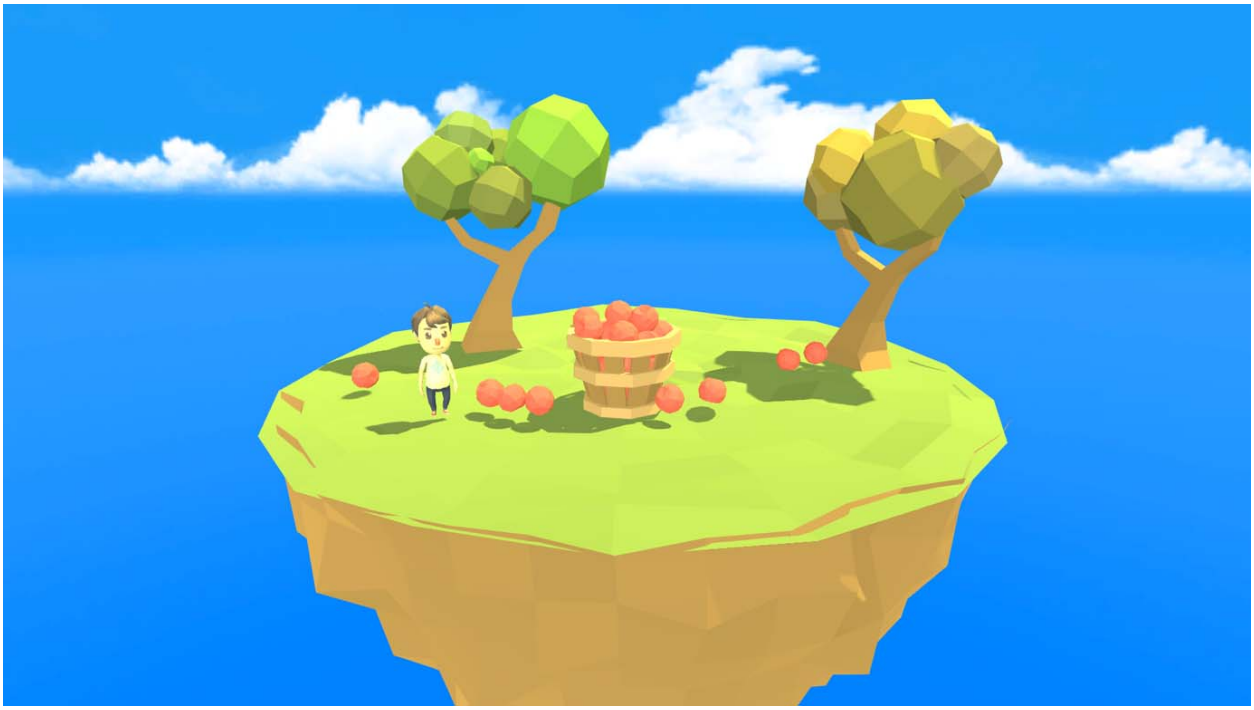
同时，也有其他游戏围绕着这个主题（视觉感知）来开发整个游戏玩法，在游戏玩法中视野范围极其重要。其中的一个例子是育碧的招牌作品——《细胞分裂》。

在这个游戏中，所有类型的感知系统都被用到了，声音、视野、触碰和知觉。如果玩家在阴影里保持安静，就会有较小的几率被敌人发现，而在照明良好的区域就算保持安静也会更容易被发现。所以，在上图的例子中，玩家可以非常近地接近正在看其他方向的敌人。

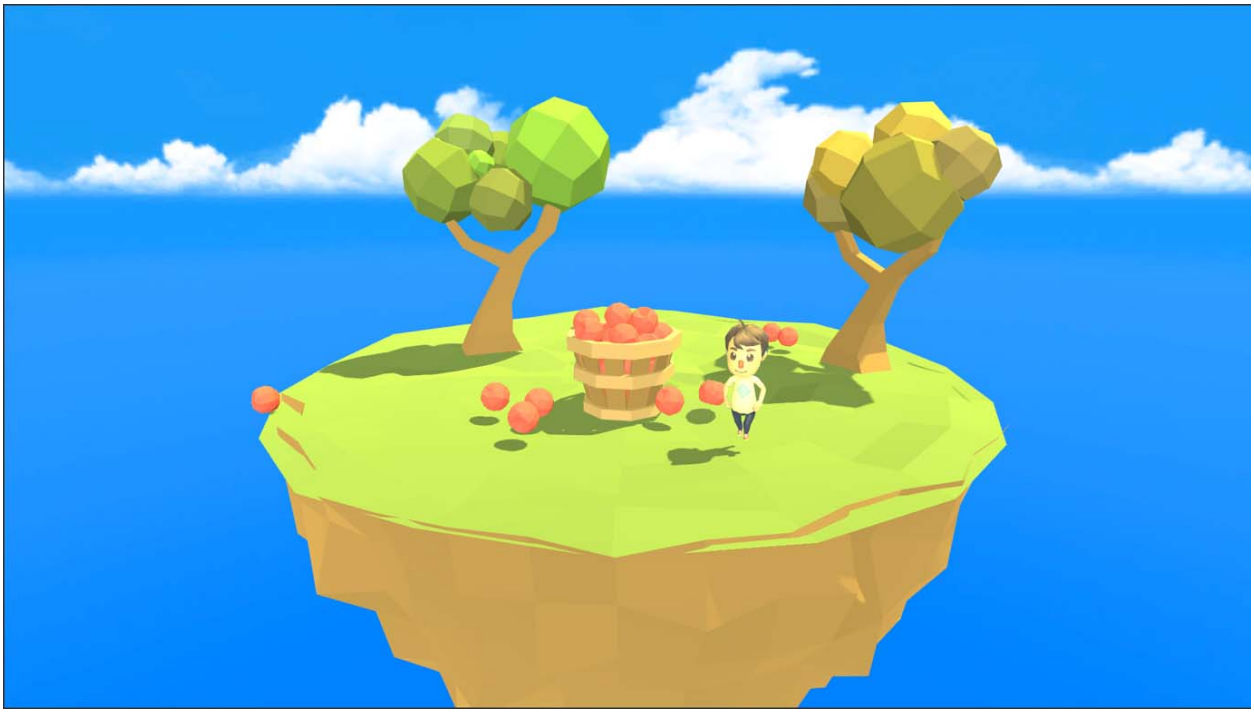
玩家为了能如此接近而不被察觉，必须要非常轻地在阴影中移动。如果玩家制造出一些噪声，或者直接走到有照明的区域，敌人就会发现他。这相比塞尔达来说是一个更复杂的系统，但是又一次，系统的好坏还是取决于我们想创造什么样的游戏和想要怎样的系统。下面会先展示基本的例子，然后再到更高级的例子。

基本视线检测

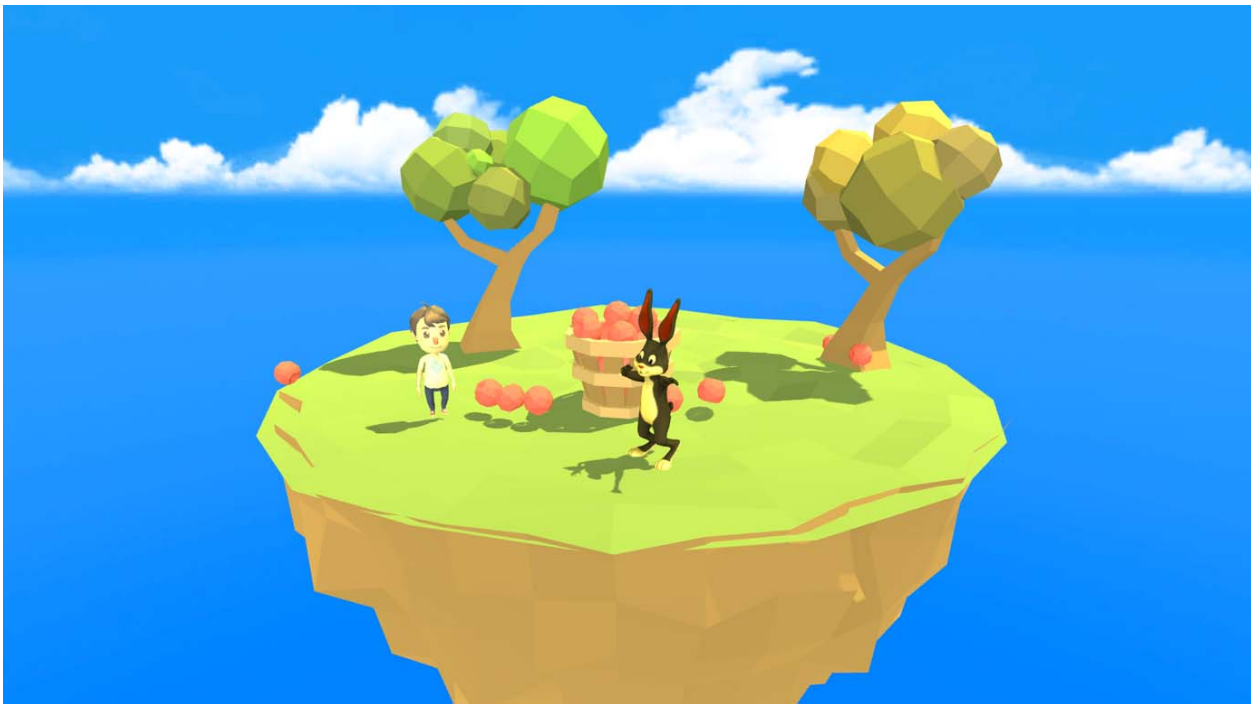
首先，我们创建一个场景添加到游戏中，之后添加玩家角色。



我们把必要的代码放到玩家身上，以便于我们移动角色和测试游戏。这里已经快速地在玩家身上定义了一些基本的移动信息，因为这是玩家和AI角色之间唯一会发生的交互的部分。



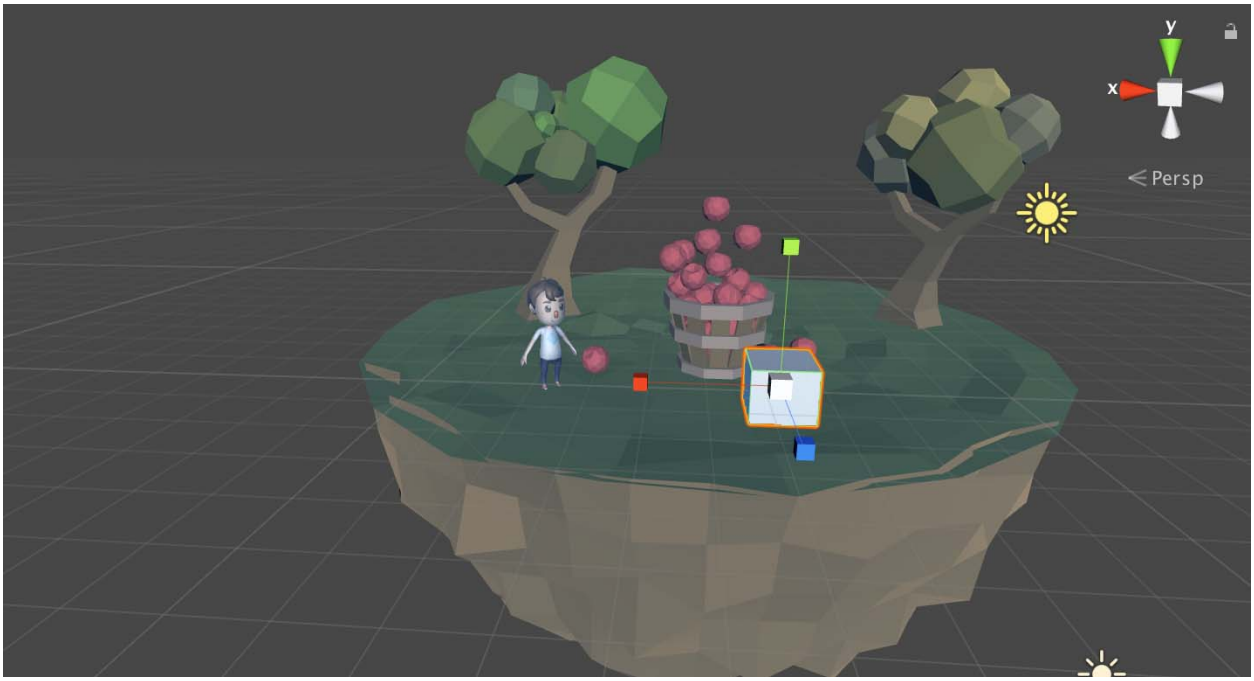
现在，我们的角色可以自由地在场景里移动了，准备就绪可以开始制作敌人角色。我们希望重现塞尔达游戏中的经典场景，当玩家靠近敌人的位置时敌人会走过来；当玩家远离这个位置，敌人回到原来的位置。



截图中的兔子就是我们刚才导入的AI角色，现在我们需要定义它附近的区域作为感知范围。这样，如果玩家靠近，兔子就会检测到玩家并从洞里出来。

我们希望，兔子应当能从它的洞里看到图中虚线标示的区域。现在如何实现呢？这里有两种方法，一种是在洞的对象上添加碰撞检测触发器，它可以检测到玩家并在洞的位置实例化一个兔子；另一种方法是把碰撞检测触发器直接加在兔子身上，兔子一开始是不可见的（假设它还在洞里），代码中有一个状态来表示兔子还在洞里的情况，另一个状态用在兔子出来的一刻。

这个例子里我们决定使用洞作为主要对象，来处理兔子隐藏的情况和某个时刻玩家进

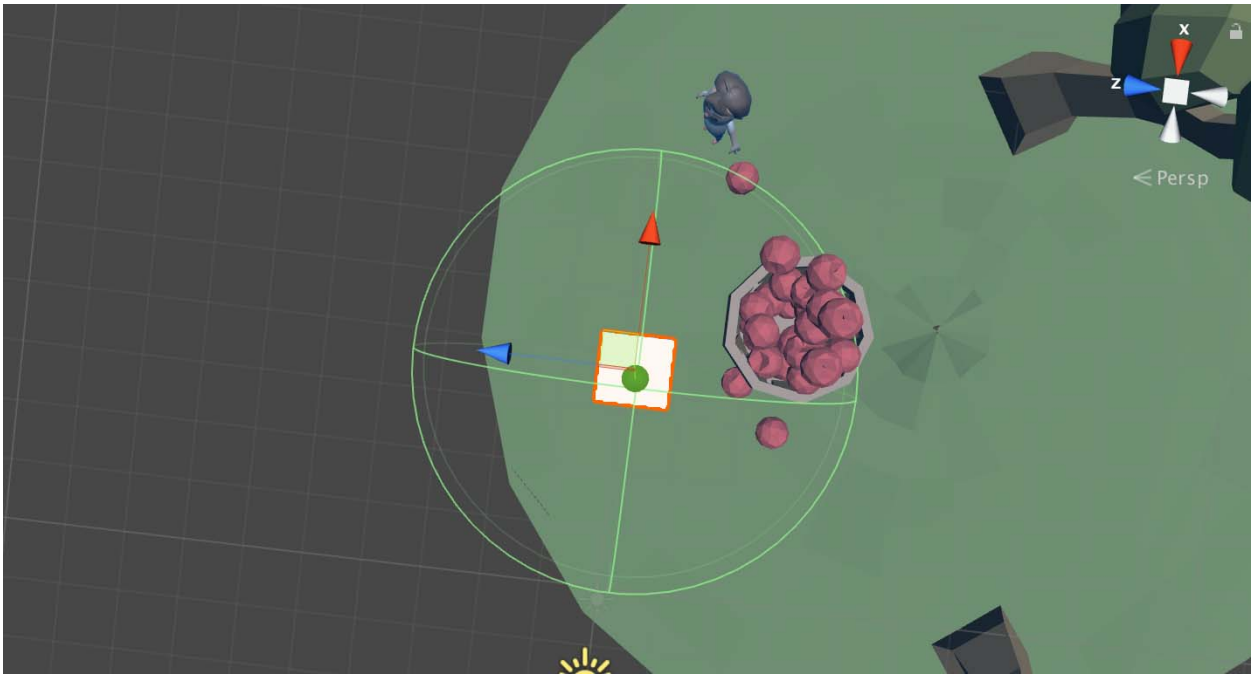


入触发区域的情况，洞的对象会实例化AI角色。

我们把兔子做成了一个预置体（prefab）并把它从场景中删除，这样我们就可以稍后再实例化它了。之后我们创建了一个立方体放在洞的位置。因为我们不需要洞的对象可见，所以只要把洞的对象的网格（mesh）组件关闭就可以了。

创建一个立方体而不是空的对象有助于我们更好地在编辑器中观察游戏里面的物体，特别是在我们需要修改参数或仅仅是为了了解这些对象位置的时候。

这时我们需要让这个对象来检测玩家，所以我们添加一个触发器，大小和之前计划中



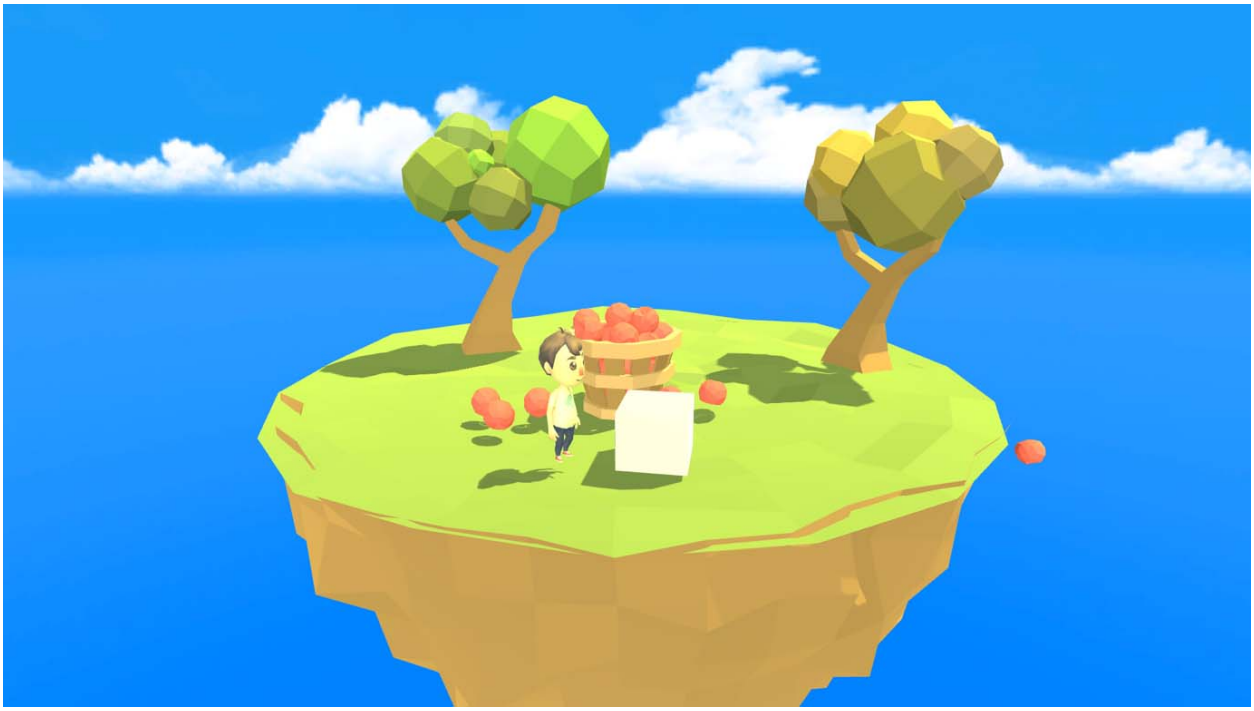
的一样。

我们删除了默认的自动创建的立方体触发器，然后设置一个新的球体触发器。为什么不用立方体触发器呢？其实也可以用立方体触发器，技术上它也能正常工作，但是那样的话隐藏的区域就会和计划中的环形区域完全不同，所以我们删除了默认的触发器并添加了新的以符合设计目的。

现在已经有了球形触发器挡在我们想要的区域上面，还需要让它检测到玩家。因此需要创建脚本挂在立方体（洞）上。

```
void OnTriggerEnter (Collider other) {

 if(other.gameObject.tag == "Player")
 {
 Debug.Log("Player Detected");
 } }
```

在脚本中，我们添加了这行代码。这是个简单的触发检测，用来判断何时物体到达了触发区域中（本书之前的例子里也有用到）。现在先简单地检查一下玩家过来时能否触发，使用Debug.Log（"Player Detected"）；语句即可。我们把这个脚本挂在立方体（洞）的对象上就可以测试了。



如果移动玩家到刚才创建的触发区域，就可以看到打印信息“Player Detected”。

这只是基本例子的第一部分，现在玩家已经能够在地图上移动，并且洞也可以检测到玩家靠近了。


使用触发器本身和任何感知系统并没有直接关系，因为这只是具体技术实现的一部分，它是不是属于AI角色感知系统的一部分取决于我们使用触发器的方式。

现在来继续制作兔子，我们的AI角色。现在它已经被创建并做成了一个预置体，可以随时出现在游戏里了。所以下一步是让洞的对象来实例化兔子，让玩家感觉到兔子已经发现了他而且还从洞里钻了出来。在洞的对象的脚本中，我们把“Player Detected”信息修改为instantiate函数。

```
public GameObject rabbit;
public Transform startPosition;
public bool isOut;

void Start ()
{
isOut = false;
}
void OnTriggerEnter (Collider other)
{

if(other.gameObject.tag == "Player" && isOut == false)
{
isOut = true;
Instantiate(rabbit, startPosition.position,
startPosition.rotation);
}
}
```

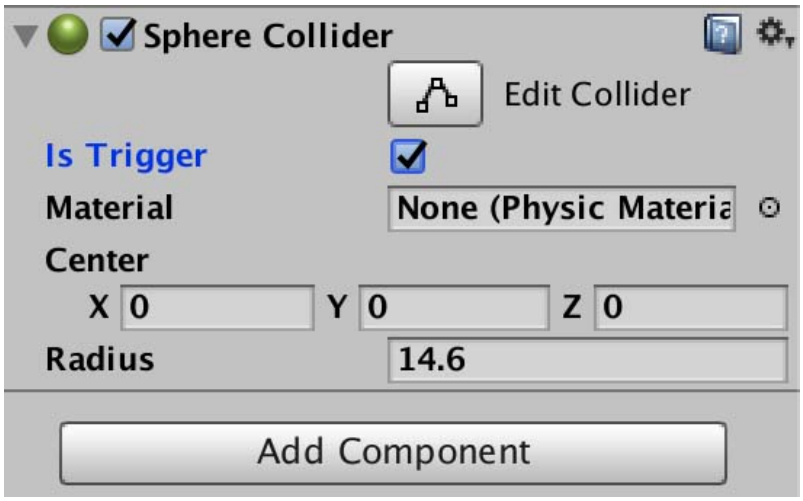
这样我们就完成了实例化对象的部分，在本例中也就是AI角色rabbit。之后添加了startPosition变量来设置角色出现的位置，我们也可以用洞的位置来代替，不过两种方式在本例中都可以正常工作。最后，添加了简单的布尔变量isOut来防止洞同时创建多于一只兔子。

当玩家进入了触发区域，兔子被实例化并跳出洞口。

现在，兔子看见玩家并跳出了洞口。下一步是为兔子自身添加相同的视觉，但这次我们希望兔子持续检测玩家是否在触发区域中，这将代表它是否能看到玩家，如果玩家远离它的视野范围，兔子就不会再看到他并回到洞里。

为这个AI角色我们可以创建一个比之前更大的区域。

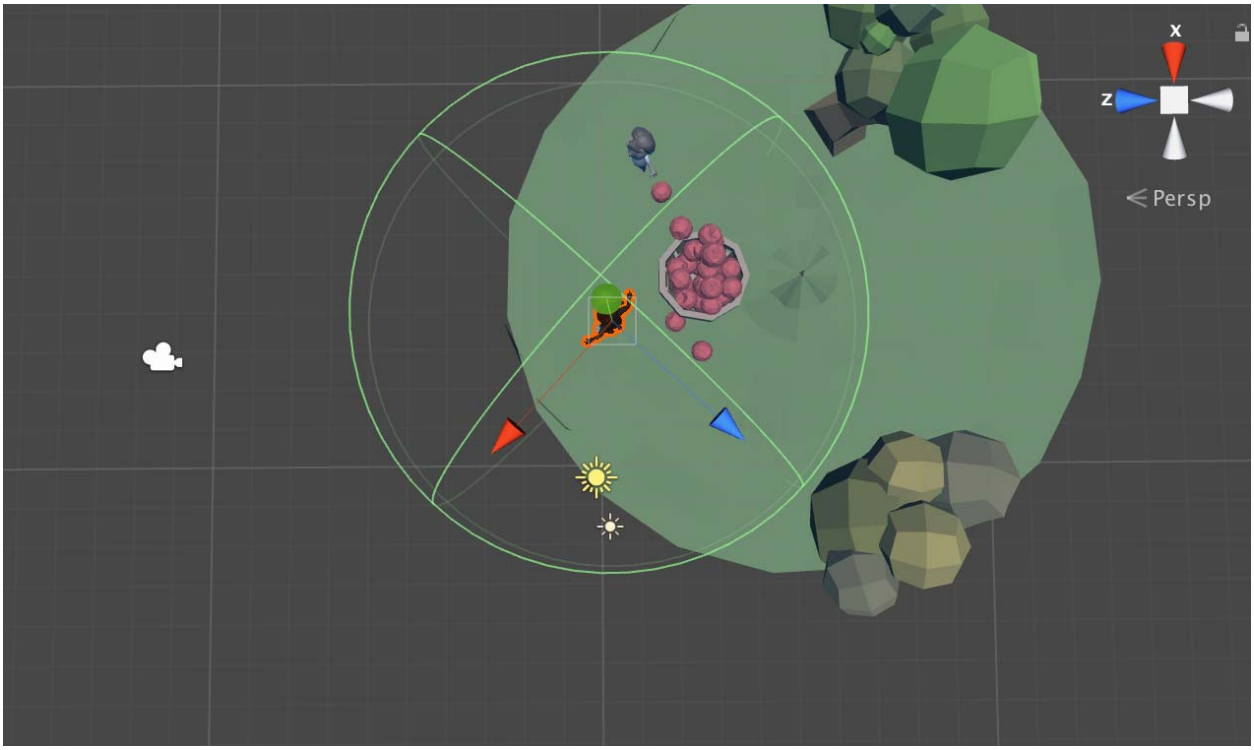
如我们所见，这就是兔子能看到玩家的区域，如果玩家离开这个区域，兔子就看不到



玩家了。

又一次，我们添加一个球形碰撞检测，但是这次是给兔子的。


打开“Is Trigger”选项来让碰撞体变成触发区域，否则它无法正常工作。



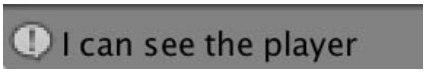
现在，球形的碰撞检测具有了之前设计的大小，并且准备好接收玩家的位置信息，它被用来作为AI角色的视野。

现在，需要做的是把处理触发区域的代码添加到兔子的脚本中去。

```
void OnTriggerStay (Collider other) {

 if(other.gameObject.tag == "Player")
 {
 Debug.Log("I can see the player");
 }
 }
```

这里是一个触发检测，用来查看玩家是否一直待在触发区域中，为做到这一点我们简单地定义了OnTriggerStay函数，对这个例子来说它能完美地工作。

我们使用了Debug.Log（"I can see the player"）；语句，只是简答地用来测试它能否按照预期工作。



测试一下这个游戏，会看到当玩家进入兔子的区域，控制台里能看到上面的信息，这代表它能正常工作。

现在，我们添加兔子视野的第二部分，玩家离开了触发区域，兔子不再看得见他了。为实现这一点我们需要添加另一个触发检查，用来检查玩家是否离开了区域。

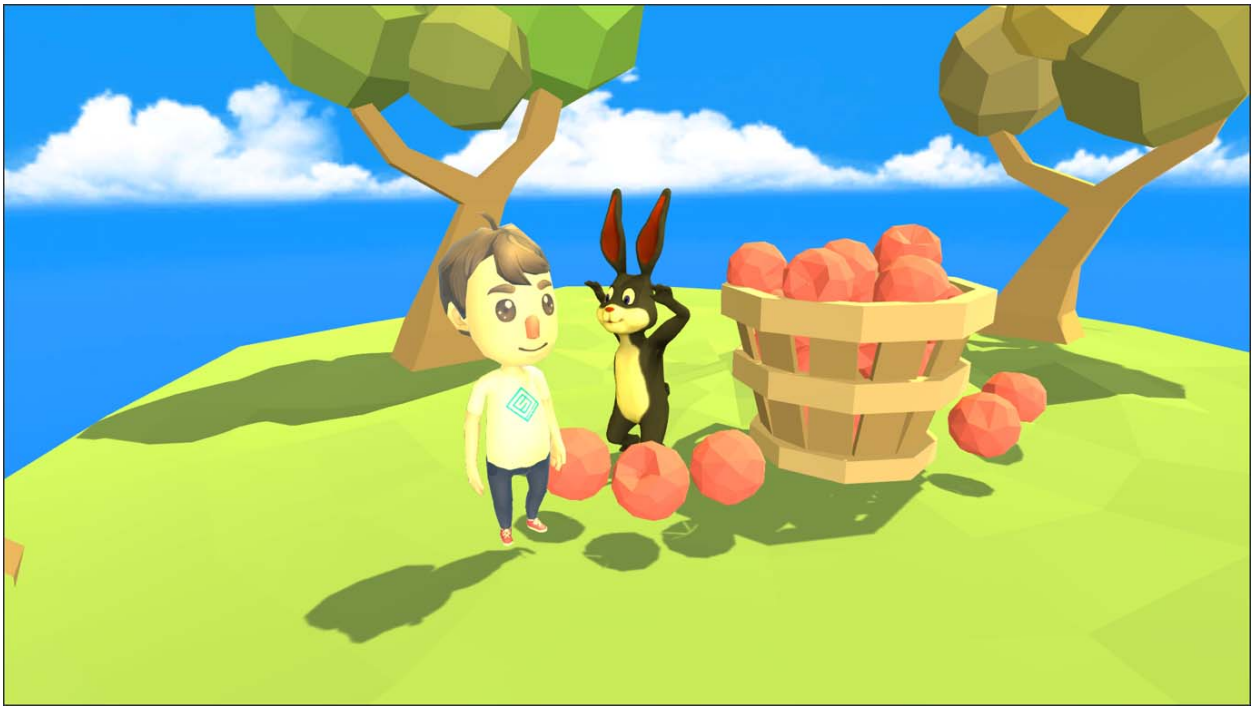
```
void OnTriggerStay (Collider other) {

 if(other.gameObject.tag == "Player")
 {
 Debug.Log("I can see the player");
 }
 }

 void OnTriggerExit (Collider other){

 if(other.gameObject.tag == "Player")
 {
 Debug.Log("I've lost the player");
 }
 }
```

紧接着是OnTriggerStay函数，我们添加了新的代码来检查玩家是否离开了触发区域。为实现这个，要使用OnTriggerExit函数，如它的字面意思，检查对象从触发区域离开的事件。但是要让它工作我们需要先定义OnTriggerEnter，否则它就无法判断玩家是否离开了触发区域，只知道玩家是否在区域里面。

```
void OnTriggerEnter (Collider other) {

 if(other.gameObject.tag == "Player")
 {
```

```
 Debug.Log("I can see the player");
 }
 }

 void OnTriggerStay (Collider other){

 if(other.gameObject.tag == "Player")
{
Debug.Log("I can see the player");
}
}

void OnTriggerExit (Collider other){

if(other.gameObject.tag == "Player")
{
Debug.Log("I've lost the player");
}
}
```

现在已经能够知道玩家何时进入区域、是否一直待在区域里以及何时离开了区域。这代表着兔子开始看到玩家的时刻、能持续看到玩家的时间以及看不到玩家的时刻。



到这里，我们测试一下游戏看看之前完成的部分能否正确工作。当开始运行游戏的时

候，我们可以通过查看控制台信息来确认一切是否按计划工作。

看到OnTriggerStay函数的信息数量一直在增长是正常的，因为它是每一帧持续检测玩家的，所以如上图所见，AI角色所具有的基本视野功能正常工作了。

高级视线检测

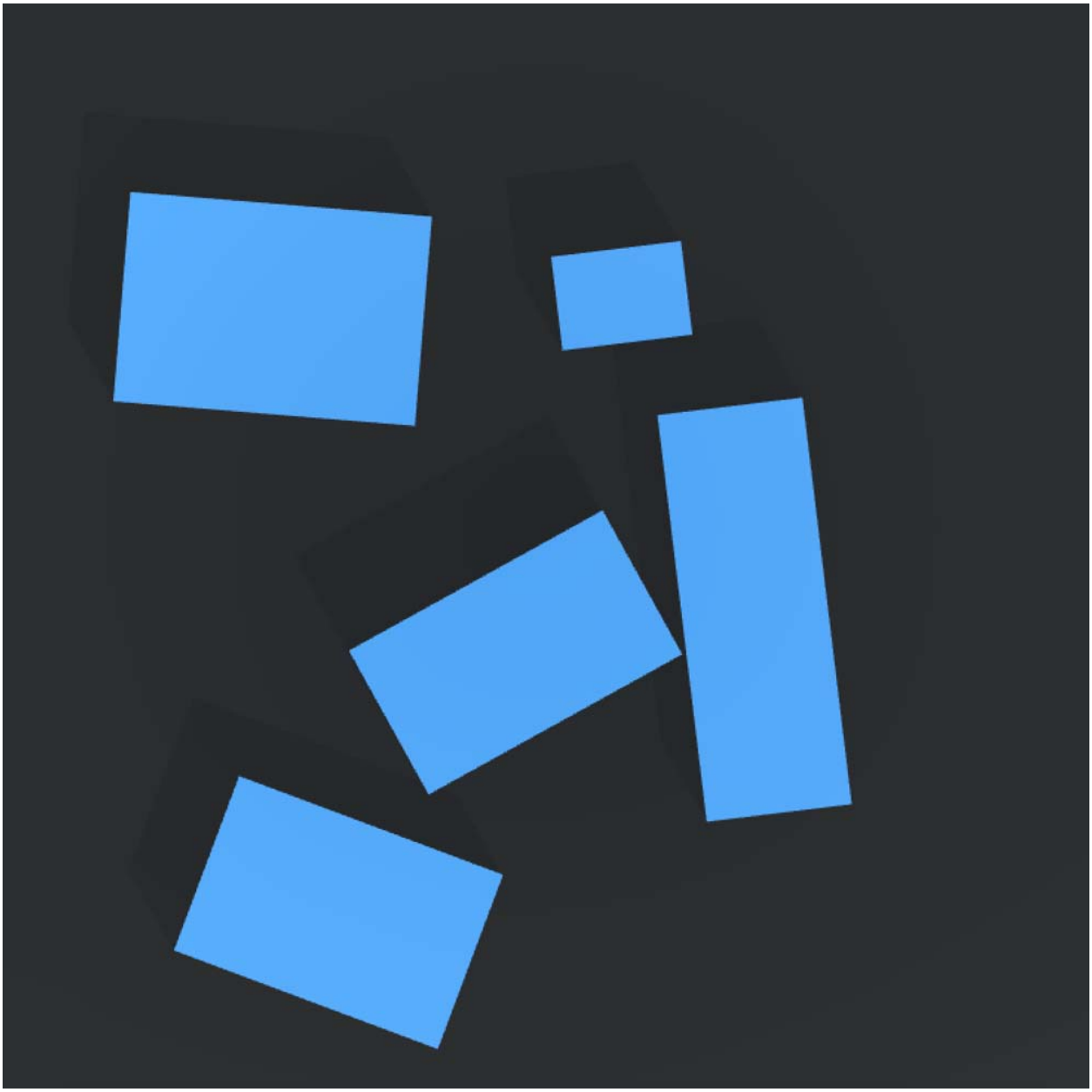
现在我们理解了很多动作游戏或冒险游戏中的基本视线检测是如何工作的，接下来看看潜入类游戏中用到的高级视线检测。让我们深入研究一下《合金装备》，并看看AI角色的视觉是如何被开发出来的。

如果仔细看上面的截图，会发现敌人AI无法看到玩家。但是玩家在场景中敌人理论上应该能够看到他。因此为什么AI角色不转向玩家并开始攻击他呢？其实仅仅是因为触发区域被设置在敌人眼睛的前方。

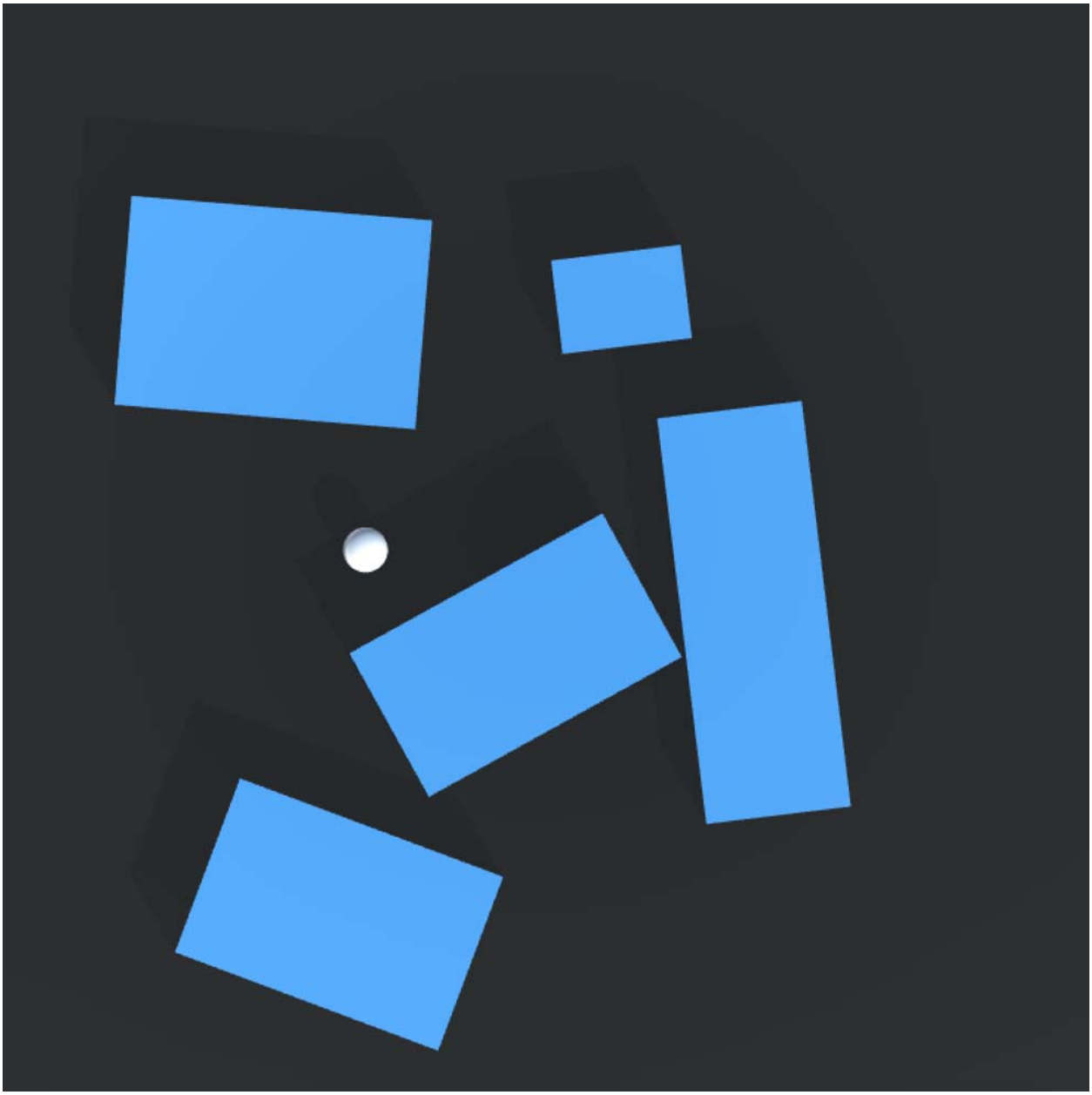
出于这个原因，如果玩家在敌人后面，它就无法发现玩家。

像我们在第二张截图中看到的，所有东西都处于黑暗的环境中，敌人没有任何途径获取玩家存在的信息，上图中高亮的区域代表敌人的视野，在视野范围内敌人能看到任何事物。现在我们看看如何实现一个类似的机制用在我们自己的AI角色上面。

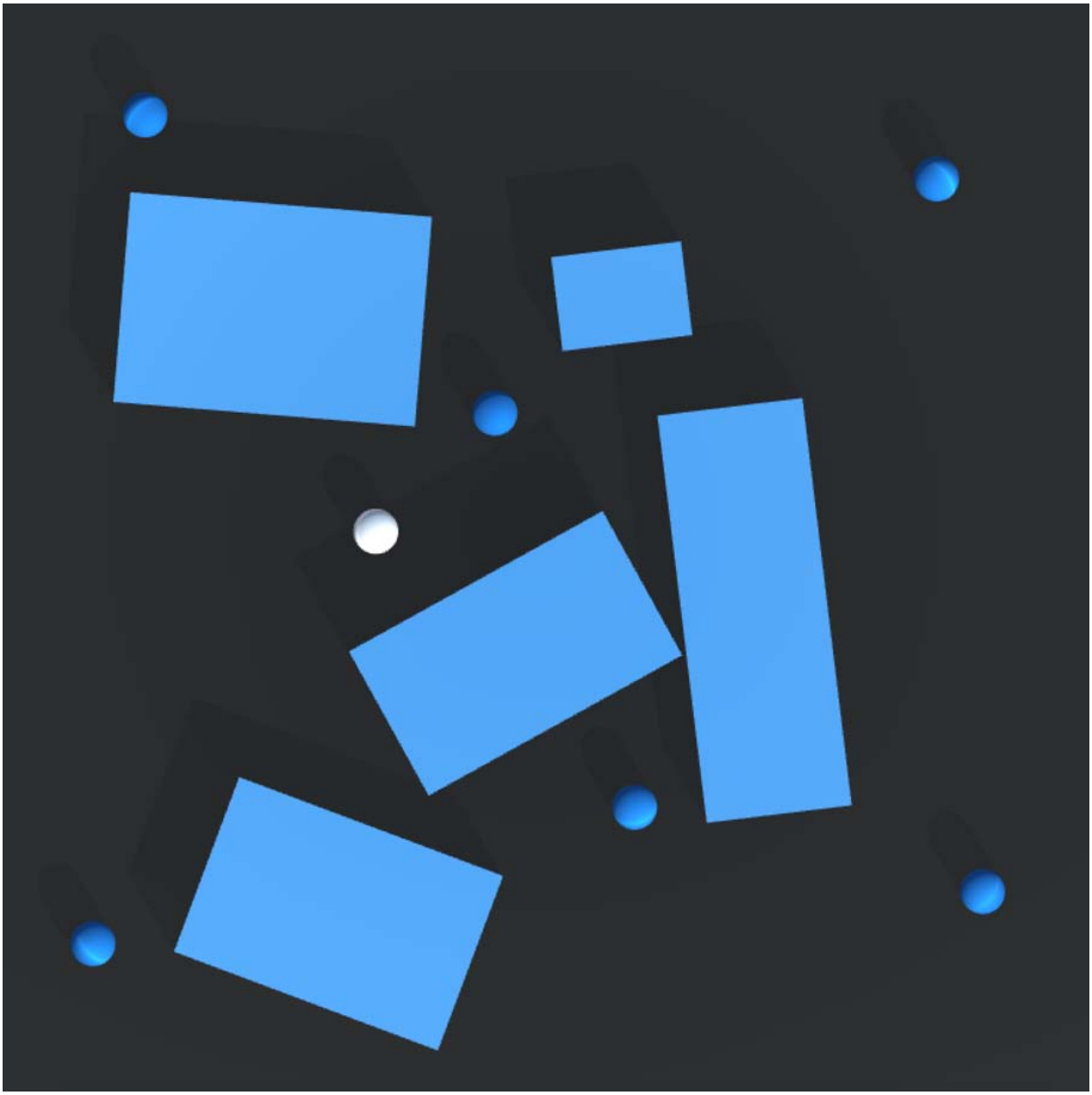
我们从创建一个场景开始。现在只是一个简单的立方体模型，之后再把外观改得美观一些。



我们创建了一些立方体模型，把它们随机地放在平面上（平面作为地面）。下一步是创建角色，我们使用一个胶囊模型代表角色。

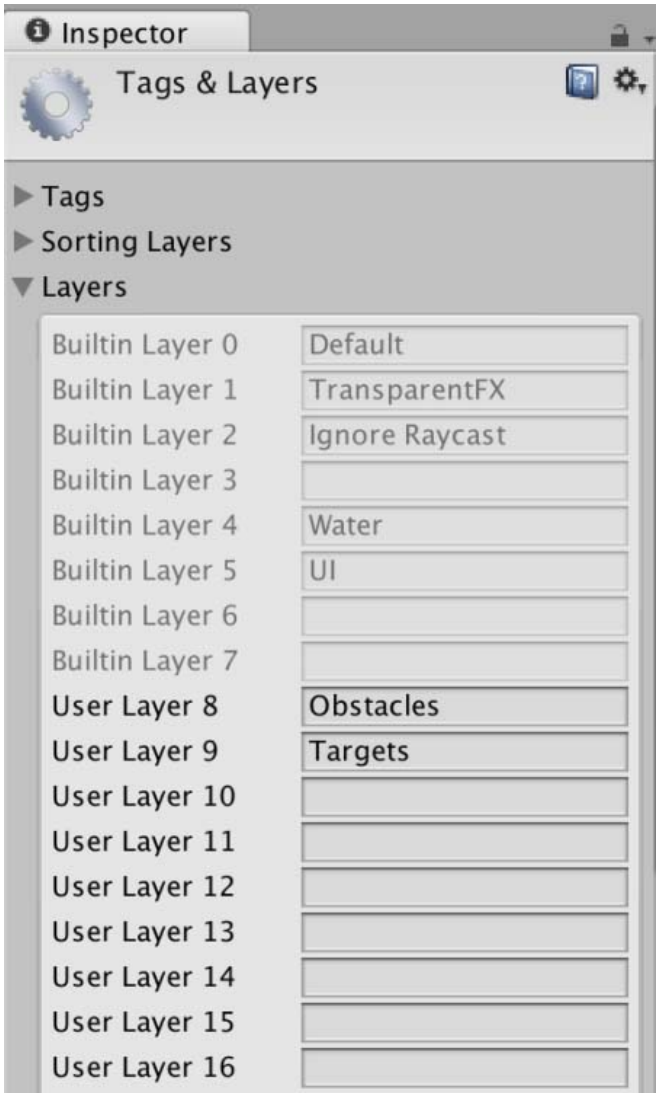


我们可以把新创建的胶囊放在地图上的任何位置。现在，需要创建一些会被AI角色发现的目标。

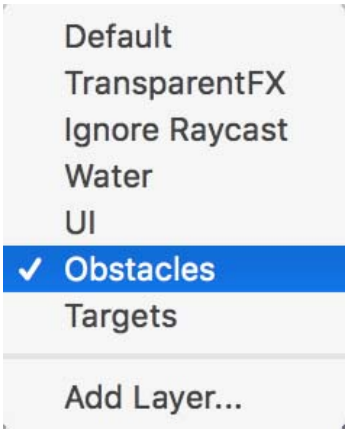


我们可以把目标物体散布在地图上的任何地方。现在，需要定义两个不同的层，一个用于障碍物，另一个用于目标物体。

在Unity里点击Layers按钮，展开更多选项，然后点击Edit Layers...那里。

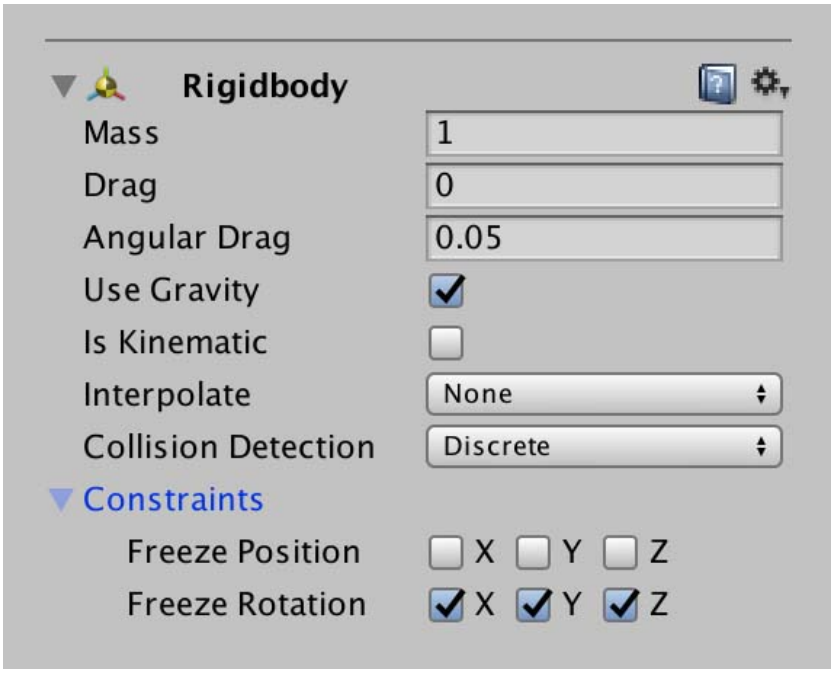


这一列输入框会出现，可以写上需要创建的层。如上图所示，已经有两个需要的层了，一个是Obstacles，另一个是Targets。这之后，我们需要将它们分配到对象上。



要做到这一点只需要简单地选择所有障碍物并点击Layers按钮然后选择Obstacles层。我们还需要为目标对象做同样的操作，但选择Targets层。

下一件事是开始添加必要的代码到角色中。还需要给角色添加一个刚体（rigidbody）组件并冻结所有旋转轴，如下图所示：



之后我们就可以为角色创建新的脚本了[1]：

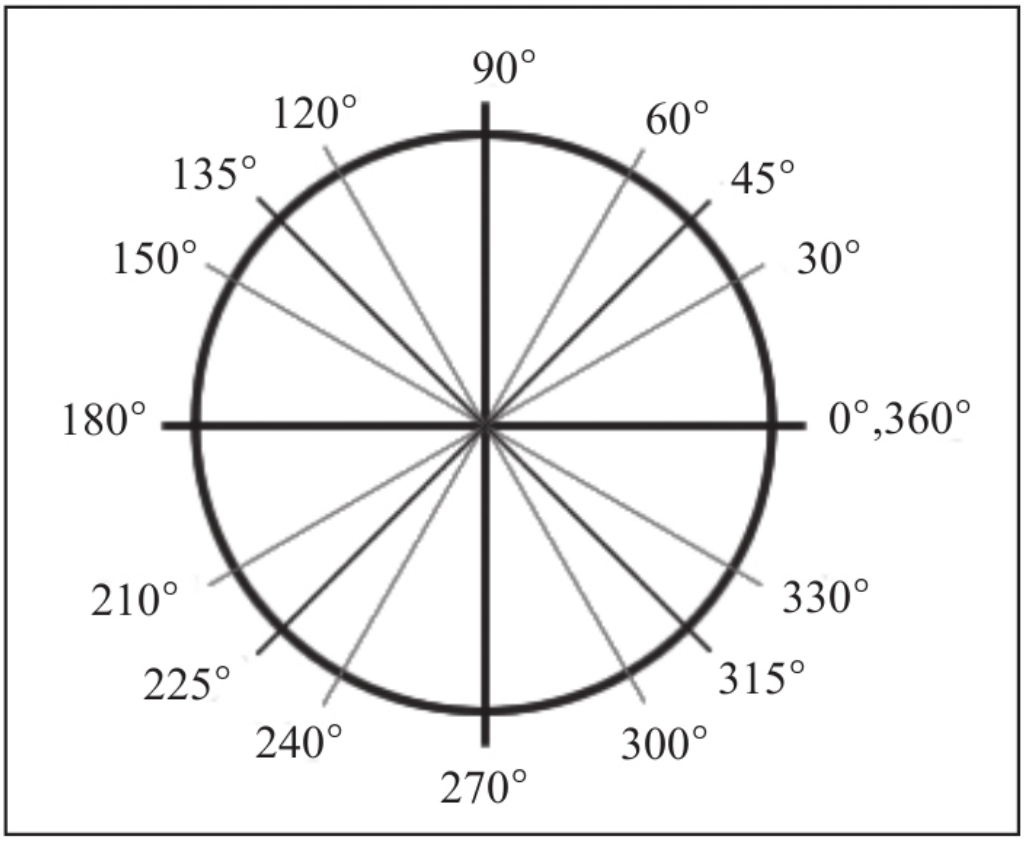
```
public float moveSpeed = 6;

Rigidbody myRigidbody;
Camera viewCamera;
Vector3 velocity;

void Start ()
{
myRigidbody = GetComponent<Rigidbody> ();
viewCamera = Camera.main;
}

void Update ()
{
Vector3 mousePos = viewCamera.ScreenToWorldPoint(new
    Vector3(Input.mousePosition.x, Input.mousePosition.y,
    viewCamera.transform.position.y));
transform.LookAt (mousePos + Vector3.up * transform.position.y);
velocity = new Vector3 (Input.GetAxisRaw ("Horizontal"), 0,
    Input.GetAxisRaw ("Vertical")).normalized * moveSpeed;
}

void FixedUpdate()
{
myRigidbody.MovePosition (myRigidbody.position + velocity *
Time.fixedDeltaTime);
}
```

这样就得到了基本的角色移动，可以自行测试，移动角色到任何地方。这个做好之后，我们可以到处移动角色，并且我们可以用鼠标模拟角色查看的方向。

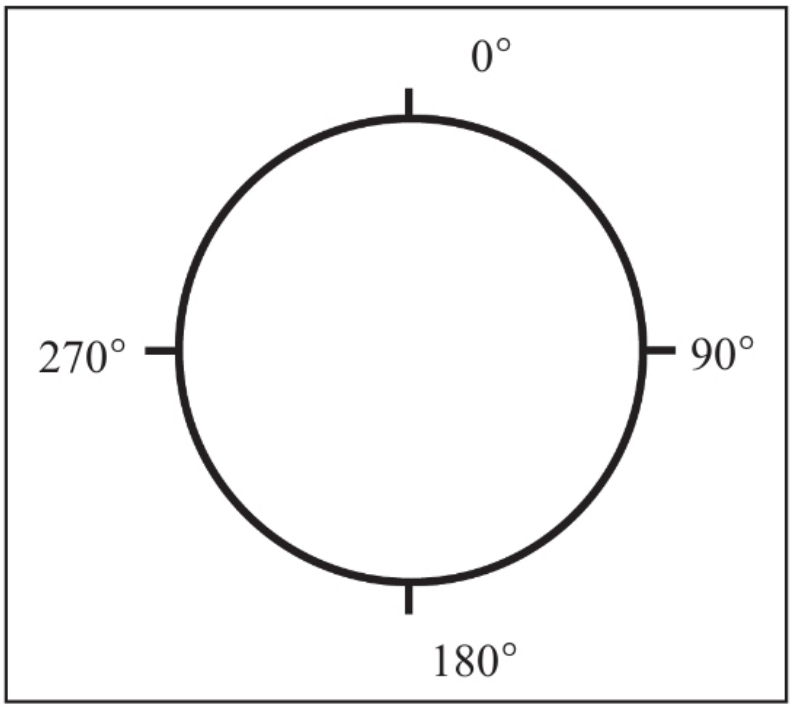
现在，让我们来继续写脚本模拟角色的视野：

```
public float viewRadius;
 public float viewAngle; public Vector3 DirFromAngle(float
    angleInDegrees)
{
 }
```

我们从两个公开的浮点变量开始，一个是viewRadius，另一个是viewAngle。之后我们创建了公开的Vector3类型的变量DirFromAngle，并且我们想要结果以角度形式表示，这里要用到三角学解决这个问题。



上面的图代表了三角学中默认的角度值，从右边开始是0度，逆时针角度增大。

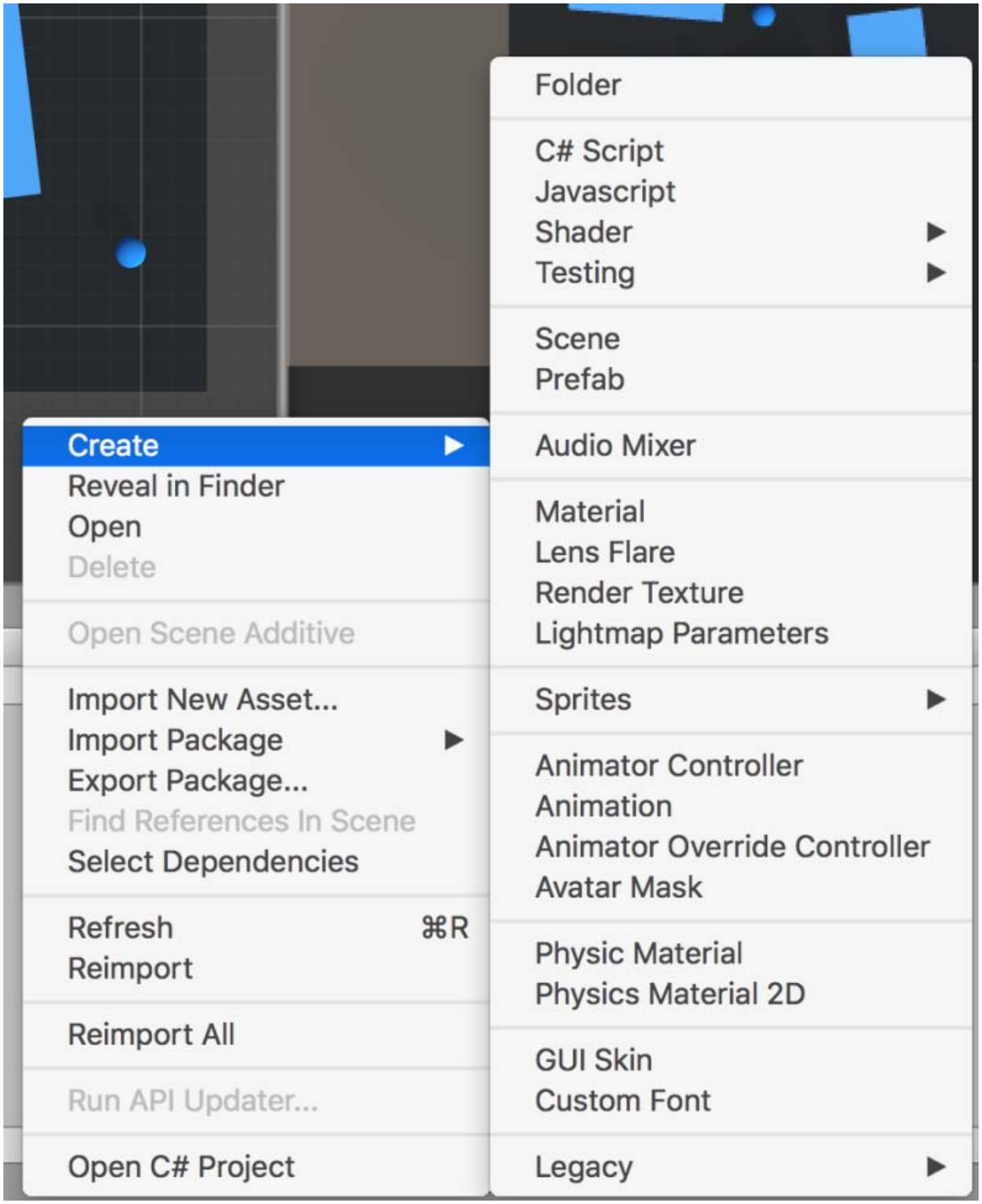


由于我们是用Unity来开发这个例子，需要记住角度的值顺序不太一样，如上图所示。这里，0度从顶部开始，角度值顺时针增大。

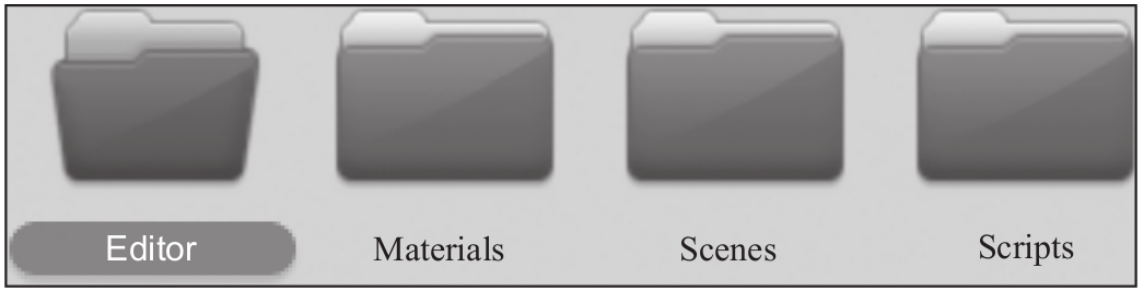
记住了这一点，我们就可以继续写角色看到的角度的代码了：

```
public float viewRadius;
public float viewAngle; public Vector3 DirFromAngle(float
   angleInDegrees)
{
  return new Vector3(Mathf.Sin(angleInDegrees *
  Mathf.Deg2Rad), 0,
  Mathf.Cos(angleInDegrees * Mathf.Deg2Rad));
}
```

现在，我们已经有了这个练习的基本结构，但是为了能在编辑器里直观地看到效果，需要创建一个新的脚本，显示角色视野的半径：



为了实现这一点，我们继续，再在工程里创建一个新的文件夹：



为了让引擎将这些内容呈现在编辑器中，要把这个新建的文件夹起名为Editor。这里面的东西可以在游戏编辑器中用到，而不需要点击开始游戏的按钮，这个功能在很多情况下都非常好用。比如我们现在的情况。

然后在刚创建的Editor文件夹中，我们再创建一个新的脚本用来显示角色的视野：

```
using UnityEngine;
using System.Collections;
using UnityEditor;
```

因为我们想要在编辑模式中用这个脚本，需要在脚本最前面指定这一点。要做到这个，在代码一开始就要写上using UnityEditor。

之后添加一行，用来让脚本可以和编辑模式联系在一起：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor (typeof (FieldOfView))]
```

现在开始编写屏幕上将会出现的表示视野范围的部分：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor (typeof (FieldOfView))]
public class FieldOfViewEditor : Editor{

void OnSceneGUI(){
FieldOfView fow = (FieldOfView)target; } }
```

我们创建了OnSceneGUI（）函数，它将包含所有在游戏编辑器中可见的东西。我们开

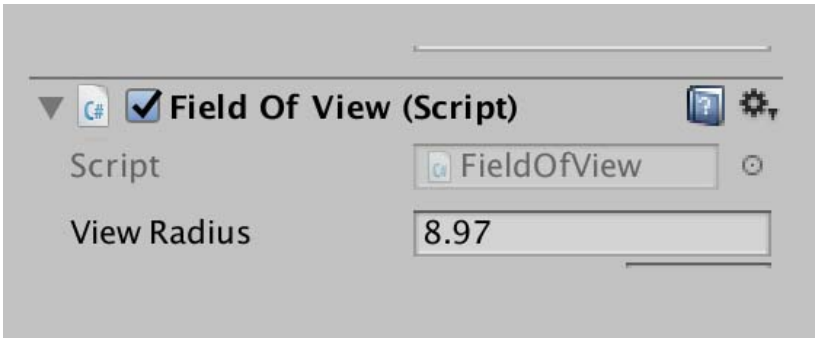
```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor (typeof (FieldOfView))]
public class FieldOfViewEditor : Editor{

void OnSceneGUI(){
FieldOfView fow = (FieldOfView)target; Handles.color = color.white; } }
```

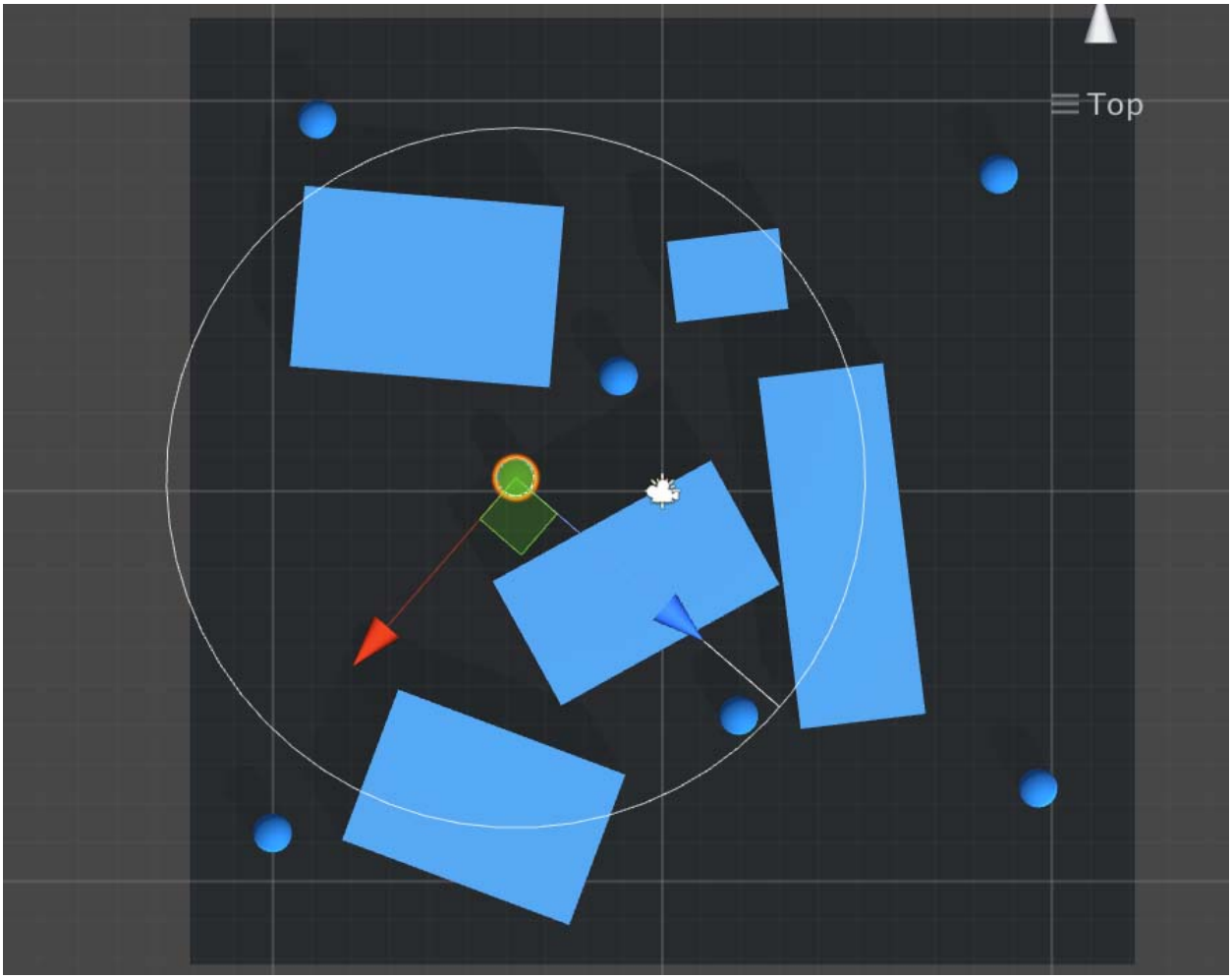
始添加视野的目标，这样就能获得视野对象的引用了。

接下来定义用来代表角色视野的颜色，要做到这一点，添加Handles.color，这里选择了白色。这在游戏的导出版本中不可见，所以可以选择一个在编辑器里容易看见的颜色。

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor (typeof (FieldOfView))]
public class FieldOfViewEditor : Editor{

void OnSceneGUI(){
FieldOfView fow = (FieldOfView)target; Handles.color = color.white;
Handles.DrawWireArc (fow.transform.position, Vector3.up,
Vector3.forward, 360, fow.viewRadius); } }
```

上面完成的功能是给视野显示范围一个框。这个框是扇形的，这就是为什么使用了DrawWireArc函数的原因。现在，让我们看看已经完成了哪些东西：





在我们创建并分配给玩家的脚本中，需要修改View Radius的值为想要的值。

当增大值时，需要注意到角色周围的环形在扩大，这说明脚本目前为止是正常工作的。圆环代表角色的视野，现在我们修改一点来让它看起来和《合金装备》截图里的一样。

再次打开角色的FieldOfView脚本，做一些改动：

```
public float viewRadius;
[Range(0,360)]
public float viewAngle;
public Vector3 DirFromAngle(float angleInDegrees, bool angleIsGlobal)
 {
  if(!angleIsGlobal)
 {
  angleInDegrees += transform.eulerAngles.y;
}
 return new Vector3(Mathf.Sin(angleInDegrees * Mathf.Deg2Rad), 0,
 Mathf.Cos(angleInDegrees * Mathf.Deg2Rad));
}
```

我们添加了ViewRadius的范围，这样我们就可以确保圆环不会超过360度。之后添加了布尔变量public Vector3DirFromAngle来检查是否角度值被设置为世界坐标，这样做就可以控制它指向角色面朝的方向了。

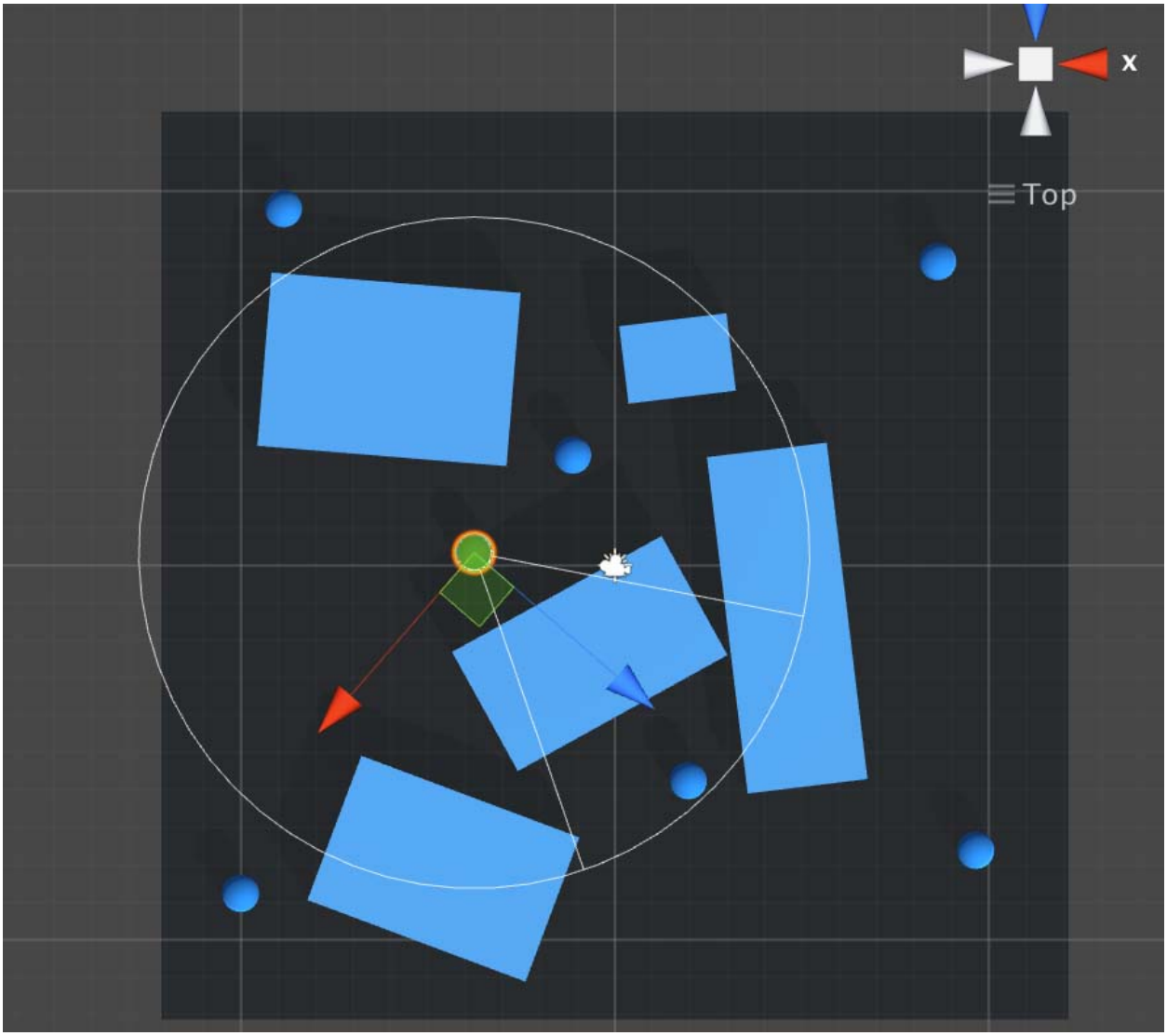
然后我们将再次打开FieldOfViewEditor脚本添加viewAngle的信息。

```
  using UnityEngine;
  using System.Collections;
  using UnityEditor;

  [CustomEditor (typeof (FieldOfView))]
  public class FieldOfViewEditor : Editor
 {
  void OnSceneGUI()
  {
   FieldOfView fow = (FieldOfView)target;
   Handles.color = color.white;
   Handles.DrawWireArc (fow.transform.position, Vector3.up,
   Vector3.forward, 360, fow.viewRadius);
   Vector3 viewAngleA =
   fow.DirFromAngle(-fow.viewAngle/2, false);
   Handles.DrawLine(fow.transform.position, fow.transform.position +
```

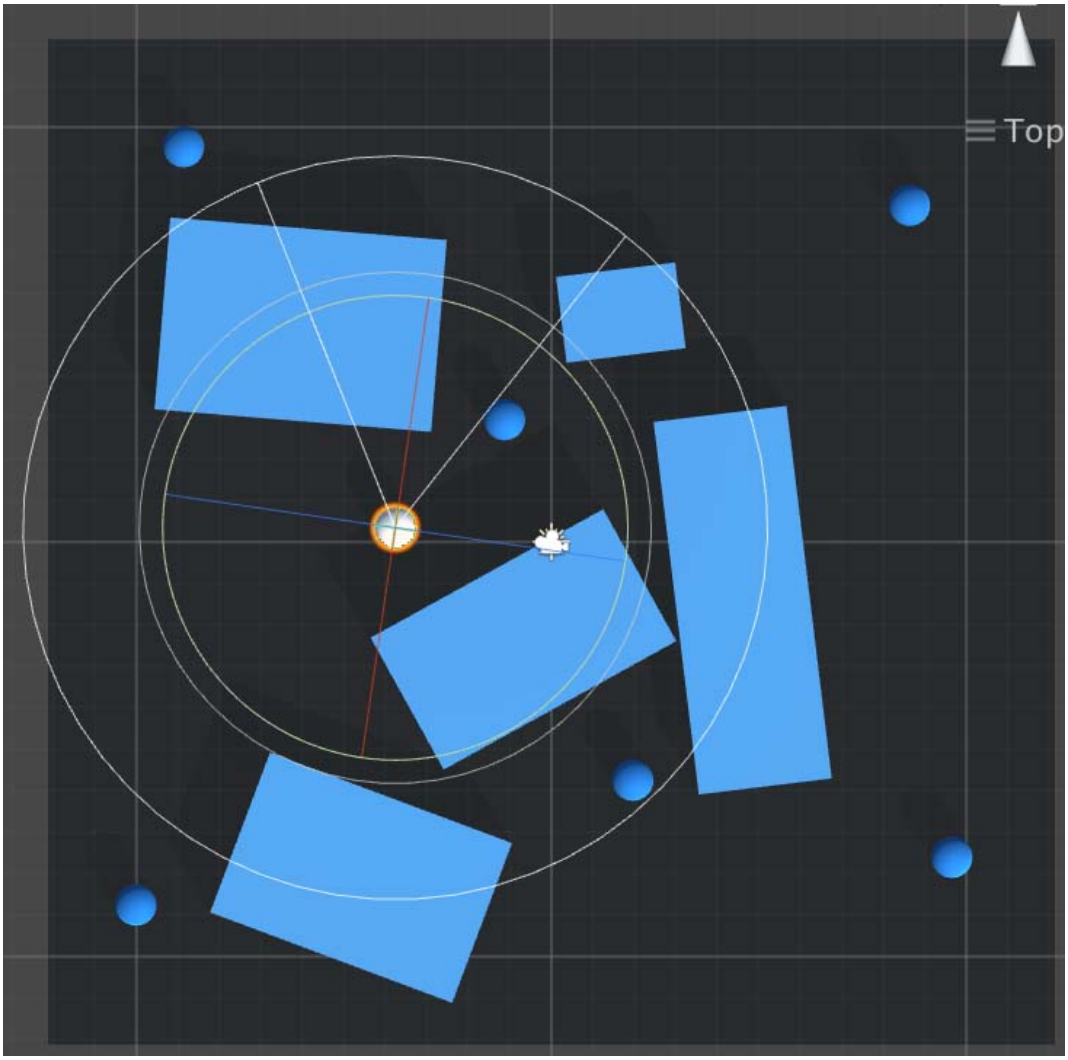
```
 viewAngleA * fow.viewRadius);
 Handles.DrawLine(fow.transform.position,
 fow.transform.position +
 viewAngleB * fow.viewRadius);
}
}
```

现在，测试一遍看看新修改的部分：

在View Angle选项里我们将修改它的值从0到任意值来查看它会如何工作：



现在，看看角色周围的环形，我们会注意到它的内部有一个三角形。三角形的大小可以通过View Angle选项来控制，三角形代表着我们角色的视野，所以现在我们会注意到角色正在看着偏右下方的位置。



由于我们把角度设置为世界坐标，我们可以旋转角色，而视野角度也会跟着旋转。

现在继续做视野射线，它的功能是检测什么东西位于角色正在看的方向。再一次修改之前创建的FieldOfView脚本：

```
public float viewRadius;
[Range(0,360)]
public float viewAngle;
public LayerMask targetMask;
public LayerMask obstacleMask;
public List<Transform> visibleTargets = new List<Transform>();
void FindVisibleTargets ()
 {
  visibleTargets.Clear ();
  Collider[] targetInViewRadius =
  Physics.OverlapSphere(transform.position, viewRadius, targetMask);
   for (int i = 0; i < targetsInViewRadius.Length; i++)
   {
   Transform target = targetInViewRadius [i].transform; Vector3
    dirToTarget = (target.position - transform.position).normalized;
   if (Vector3.Angle (transform.forward, dirToTarget) < viewAngle / 2)
    {
      float dstToTarget = Vector3.Distance (transform.position,
       target.position);
      if (!Physics.Raycast(transform.position,
       dirToTarget, dstToTarget, obstacleMask))
           {
             visibleTargets.Add (target);
            }
   }
 }
 public Vector3 DirFromAngle(float angleInDegrees, bool angleIsGlobal) {
  if(!angleIsGlobal)
     {
       angleInDegrees += transform.eulerAngles.y;
     }
  return new Vector3(Mathf.Sin(angleInDegrees * Mathf.Deg2Rad), 0,
     Mathf.Cos(angleInDegrees * Mathf.Deg2Rad));
}
```

这里完成的功能是添加Physics物理信息到脚本中，检测只在View Angle范围内出现的物体。要检测某件东西是否在角色视野内，我们使用了Raycast射线来检查是否存在带有obstacleMask的物体。现在我们创建一个IEnumrator协程函数来给发现障碍物加上一个小的延迟：

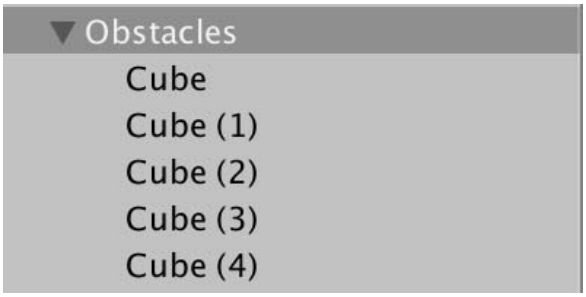
```
public float viewRadius; [Range(0,360)]
public float viewAngle; public LayerMask targetMask;
public LayerMask obstacleMask;
[HideInInspector] public List<Transform> visibleTargets = new
List<Transform>();
void Start ()
    {
     StartCoroutine("FindTargetsWithDelay", .2f);
    }
IEnumerator FindTargetsWithDelay(float delay)
{
```

```
while (true) {
 yield return new WaitForSeconds (delay);
 FindVisibleTargets ();
 }
}
void FindVisibleTargets ()
 {
 visibleTargets.Clear ();
 Collider[] targetInViewRadius
 =Physics.OverlapSphere(transform.position,viewRadius, targetMask);
for (int i = 0; i < targetsInViewRadius.Length; i++)
{
 Transform target = targetInViewRadius [i].transform; Vector3 dirToTarget =
(target.position - transform.position).normalized;
 if (Vector3.Angle (transform.forward, dirToTarget) < viewAngle / 2) {
float dstToTarget = Vector3.Distance (transform.position, target.position);
 if (!Physics.Raycast(transform.position, dirToTarget, dstToTarget,
     obstacleMask))
 {
  visibleTargets.Add (target);
 }
 }
 }
 public Vector3 DirFromAngle(float angleInDegrees, bool angleIsGlobal)
{
if(!angleIsGlobal)
 {
 angleInDegrees += transform.eulerAngles.y;
 }
 return new Vector3(Mathf.Sin(angleInDegrees * Mathf.Deg2Rad), 0,
 Mathf.Cos(angleInDegrees * Mathf.Deg2Rad)); }
```

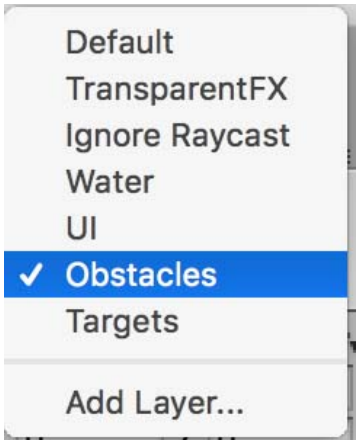
现在我们创建了一个协程，角色会有一个小的反应时间，这个例子里反应时间被设置为0.2秒，之后发现视野中的物体。为了测试方便，我们要对FieldOfViewEditor脚本做一些修改。我们打开脚本并添加一行代码：

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor (typeof (FieldOfView))]
public class FieldOfViewEditor : Editor{

void OnSceneGUI(){
FieldOfView fow = (FieldOfView)target;
Handles.color = color.white; Handles.DrawWireArc
(fow.transform.position, Vector3.up,
Vector3.forward, 360, fow.viewRadius); Vector3 viewAngleA =
fow.DirFromAngle(-fow.viewAngle/2, false);

Handles.DrawLine(fow.transform.position, fow.transform.position +
viewAngleA * fow.viewRadius);
Handles.DrawLine(fow.transform.position,fow.transform.position +
viewAngleB * fow.viewRadius); Handles.color = Color.red;
Foreach (Transform visibleTarget in fow.visibleTargets)
{


Handles.DrawLine(fow.transform.position, visibleTarget.position);
}
}
}
```

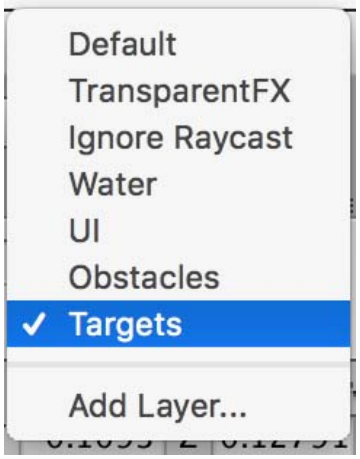
有了这个修改以后，我们可以直观地看到什么时候角色检测到了哪些障碍物、什么时候障碍物从视野中消失。



为了测试，我们选择所有场景里所有的障碍物：

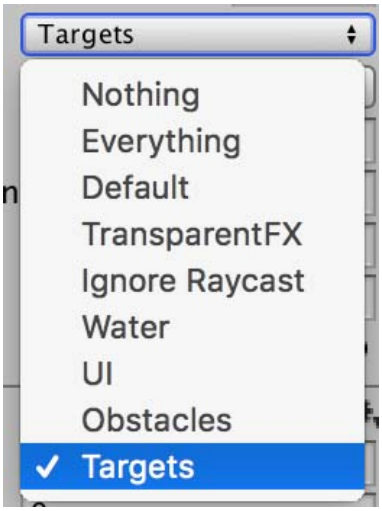


将它们的层设置为Obstacles：



同样，选择游戏中所有Targets物体：

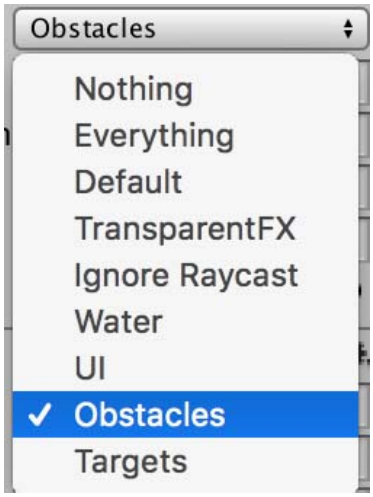
将它们的层设置为Targets。这一步非常重要，这样一来我们的Raycast射线检测就可以识别出视野中的物体是什么了。现在，我们点击角色对象并定义哪一层代表Targets以及哪一层代表Obstacles：



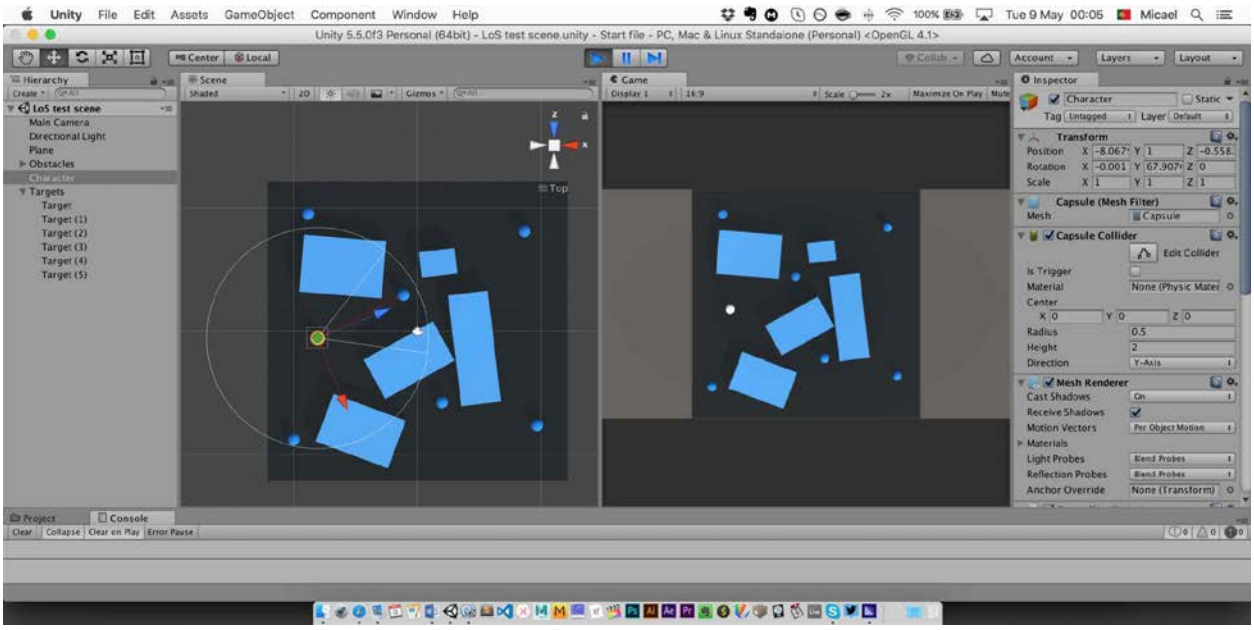
在Field Of View脚本中找到Layer Mask选项：

选择Targets层：

之后同样找到Obstacles选项：

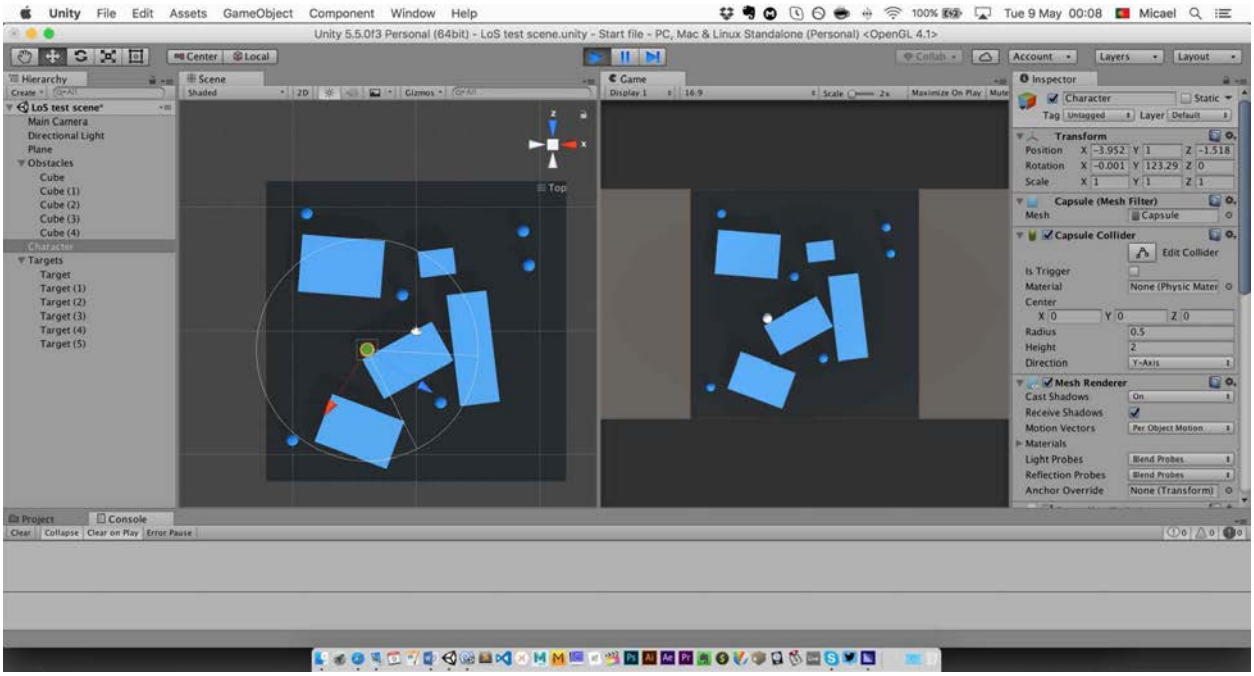


选择Obstacles层。

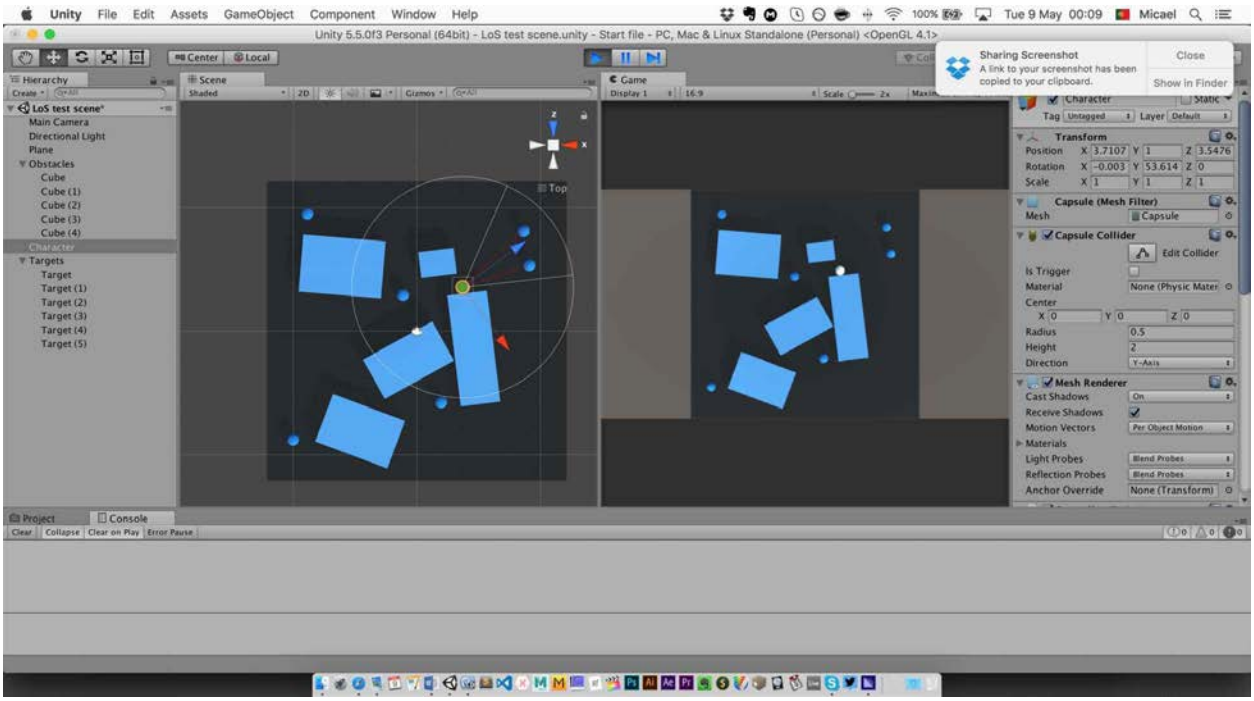


这一步完成后，我们可以最终测试这个例子了，看看角色发现目标时会发生什么。

开始播放这个例子后，我们可以看见当一个目标到了视野范围内的时候，红色的线出现在角色和目标之间。这可以用来代表角色已经发现了敌人。



但是，当我们移动角色到障碍物前面的时候，即便目标在视野范围内，角色依然无法发现它因为障碍物挡住了角色的视线。这就是为什么我们把Obstacles层设置给每一个障碍物，用这种方式我们就不会得到一个像X光一样的视线了。



我们还可以让角色同时发现两个目标，这两个目标都会用射线标记出来，这代表我们的角色具有同时探测到一个以上目标的能力，这对我们定义更好的策略和战术很有帮助。

拟真视野效果

现在我们已经有了可以正常工作的视线检测系统，接下来我们要添加一个拟真视野效果。这会让角色拥有视线的余光，让视线外侧的物体具有较少的细节，视线正前方的物体具有较多的细节。这是一种对人类视觉的模拟。我们会更多地注意前方的东西，如果需要查看旁边的东西，我们需要转到那个方向来更清晰地看到它。

还是从打开FieldOfView脚本开始。然后我们添加一个新的浮点变量叫做meshResolut

```
public float viewRadius; [Range(0,360)]
public float viewAngle; public LayerMask targetMask; public LayerMask
obstacleMask; [HideInInspector] public List<Transform> visibleTargets = new
List<Transform>(); public float meshResolution;
```

ion：

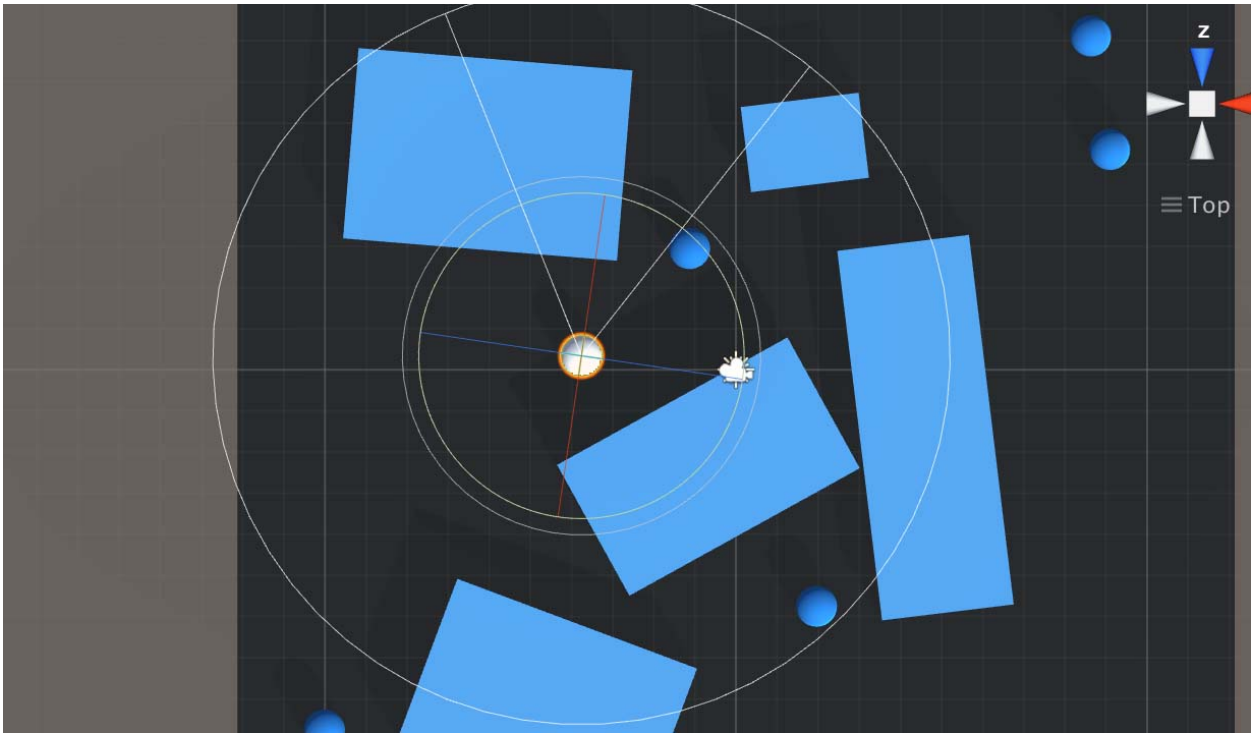
现在我们需要创建一个新的方法叫做DrawFieldOfView。在这个方法中，我们定义了一系列Raycast射线用于角色的视野。同样我们还要定义画出的每一条射线的角度。

```
void DrawFieldOfView() {
 int stepCount = Mathf.RoundToInt(viewAngle * meshResolution);
 float stepAngleSize = viewAngle / stepCount;
 for (int i = 0; i <= stepCount; i++) {
 float angle = transform.eulerAngles.y - viewAngle / 2 + stepAngleSize *
i;Debug.DrawLine (transform.position, transform.position + DirFromAngle
(angle, true) * viewRadius, Color.red);
 }
 }
```

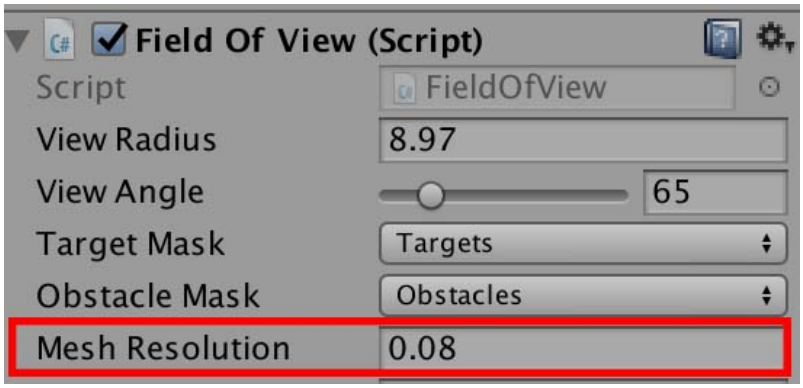
```
void LateUpdate() {
DrawFieldOfView ();
}
```

在创建这个新的方法以后，我们简单地在更新函数中调用它：

到这里，我们可以打开游戏编辑器测试一下，看看代码创建的视觉效果：

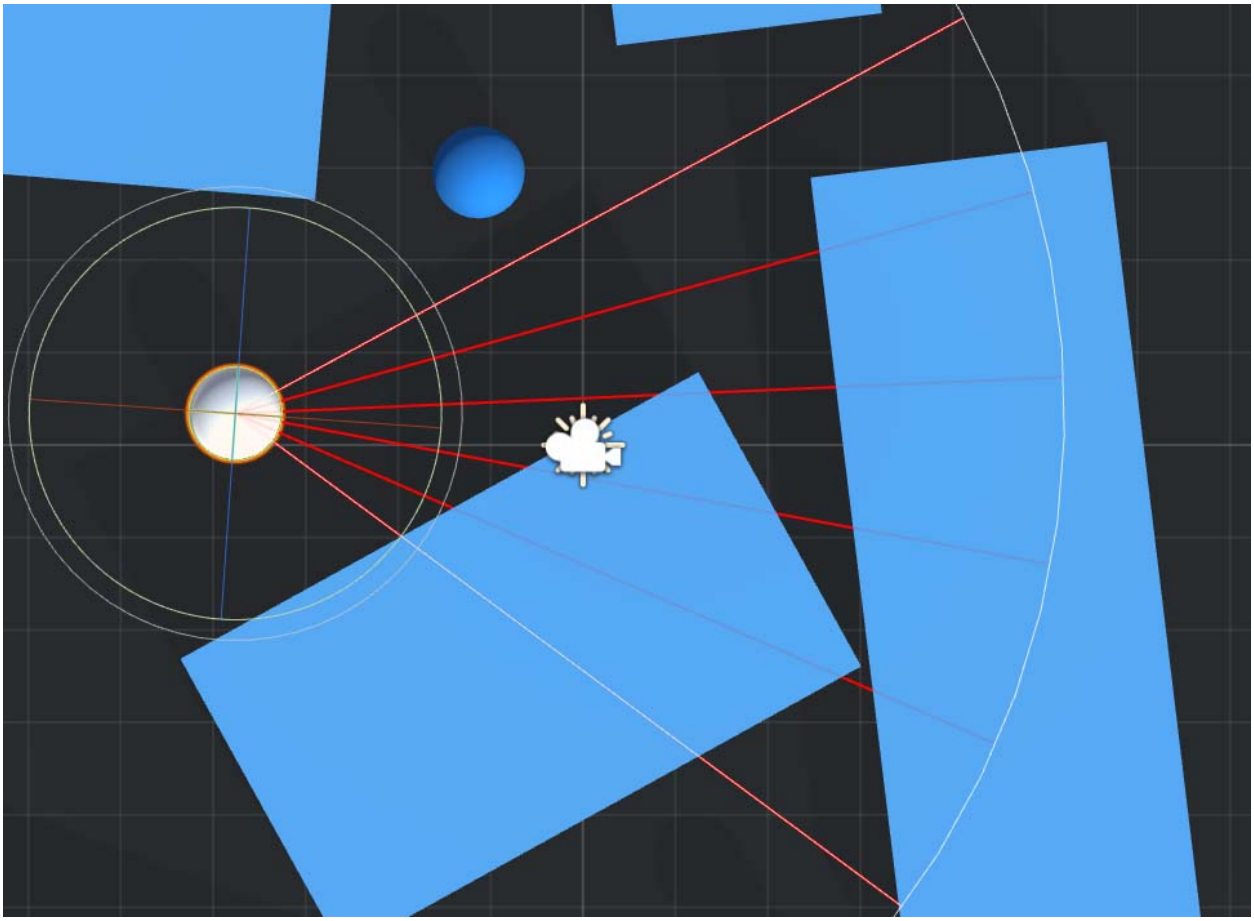


我们播放游戏后马上测试，并不会发现新脚本的任何不同。这是正常的，因为我们需



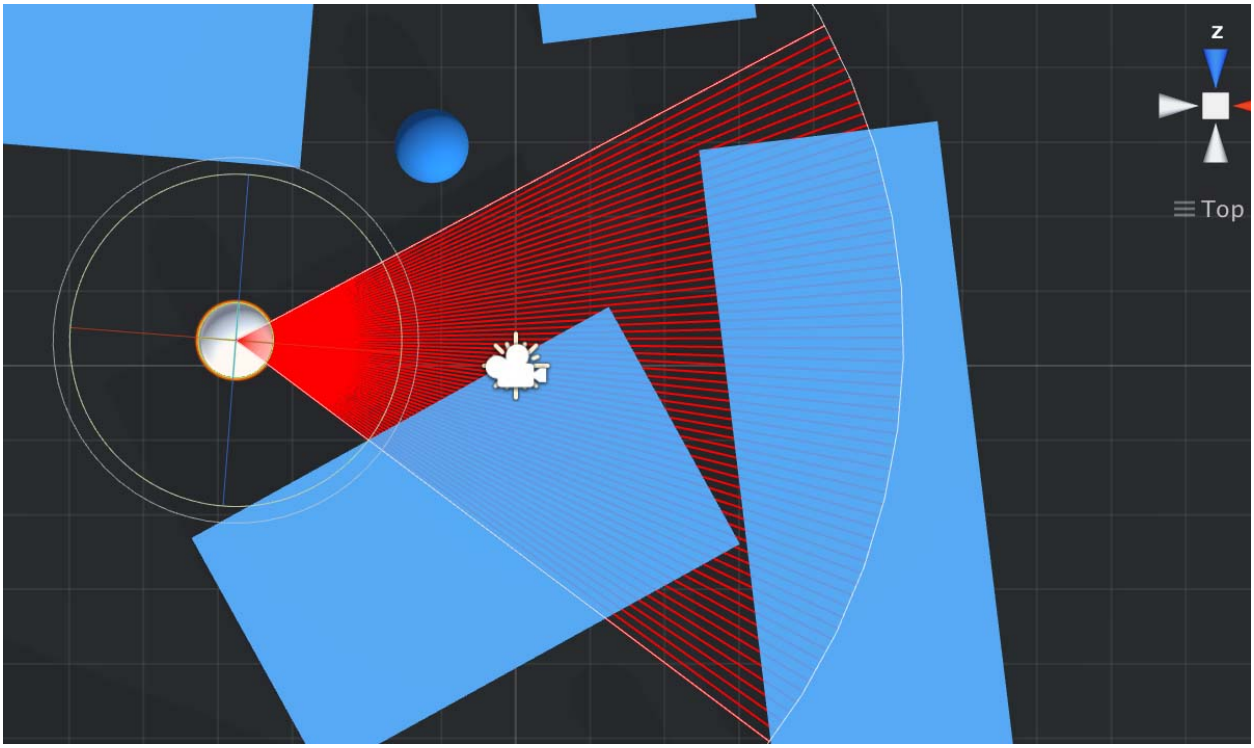
要增大角色的Mesh Resolution。

如上图所示，我们需要添加一个值到Mesh Resolution变量中，这样才能看到想要的效果。



将Mesh Resolution改为0.08，我们会发现游戏编辑器中已经出现了几条红色的线，这正是我们想要的。

如果我们继续增大这个值，会产生更多的线，意味着视野有更多的细节，典型的效果如下图：



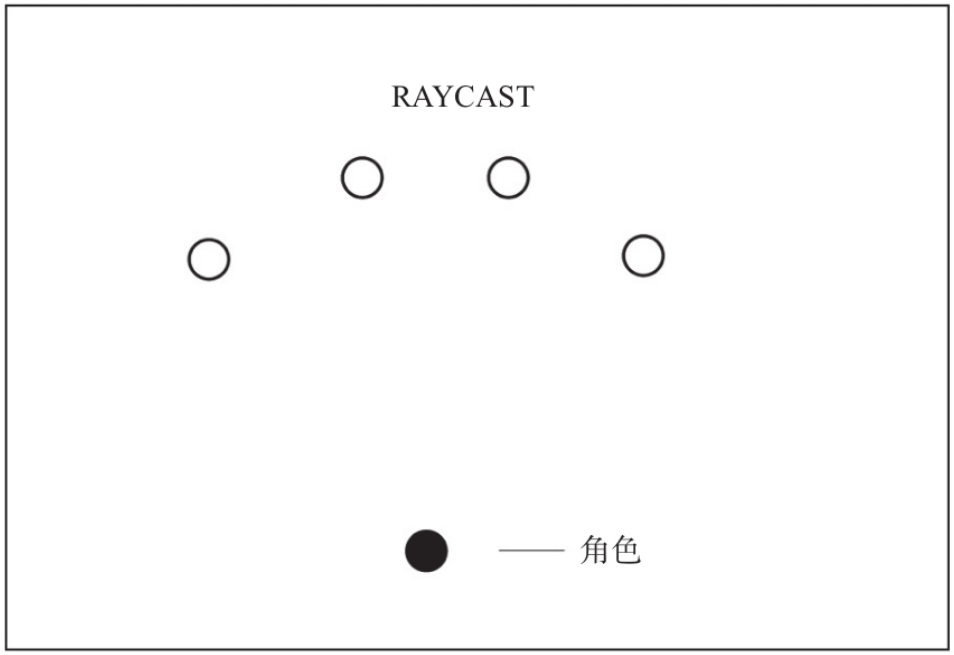
但是，我们要知道增加这个值也会增加CPU的使用率，需要考虑到这一点，特别是在场景中有众多角色的情况下。

现在我们回到脚本，为每一条线增加一个碰撞检测，允许角色同时接收多条线的信息。我们从创建一个新的结构开始，它用于保存所有将要创建的射线的信息：

```
public struct ViewCastInfo {
public bool hit;
public Vector3 point;
public float dst;
public float angle;

public ViewCastInfo(bool _hit, Vector3 _point, float _dst, float
_angle) {
hit = _hit;
point = _point;
dst = _dst;
angle = _angle;
} }
```

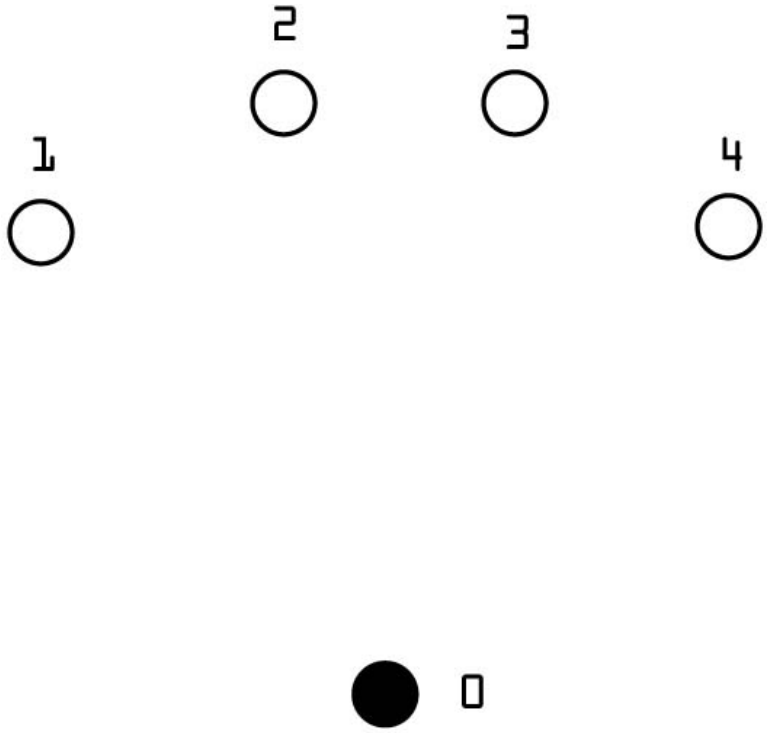
这个新的结构体创建好了，我们回到DrawFieldOfView（）方法并为射线添加功能，让它们能够检测碰撞：

```
void DrawFieldOfView() {
int stepCount = Mathf.RoundToInt(viewAngle * meshResolution);
float stepAngleSize = viewAngle / stepCount;
List<Vector3> viewPoints = new List<Vector3>();
for (int i = 0; i <= stepCount; i++)
{
 float angle = transform.eulerAngles.y - viewAngle / 2 + stepAngleSize
 * i;
 ViewCastInfo newViewCast = ViewCast(angle);
 Debug.DrawLine(transform.position, transform.position +
 DirFromAngle(angle, true) *
 viewRadius, Color.red);
 viewPoints.Add(newViewCast.point);
}
}
```

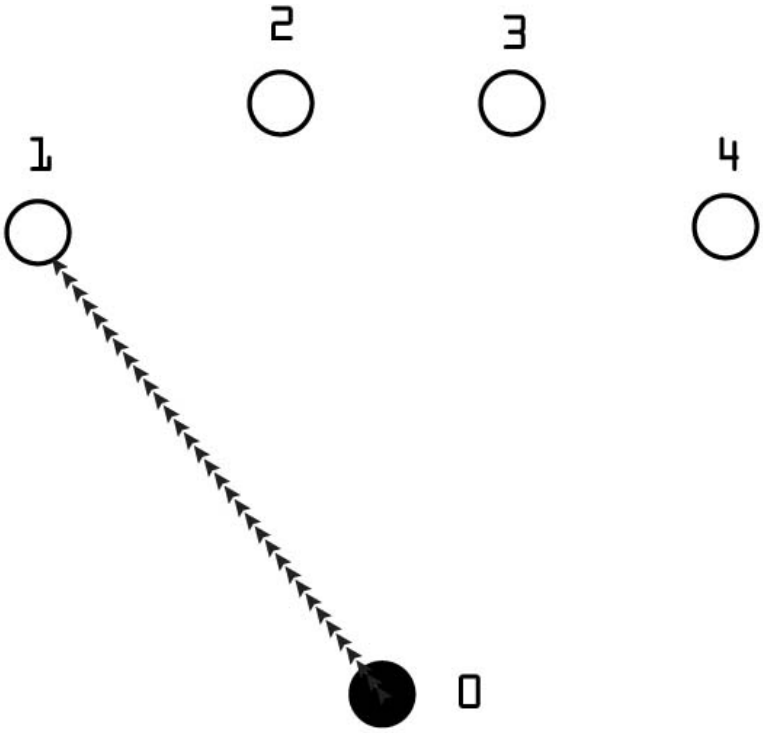


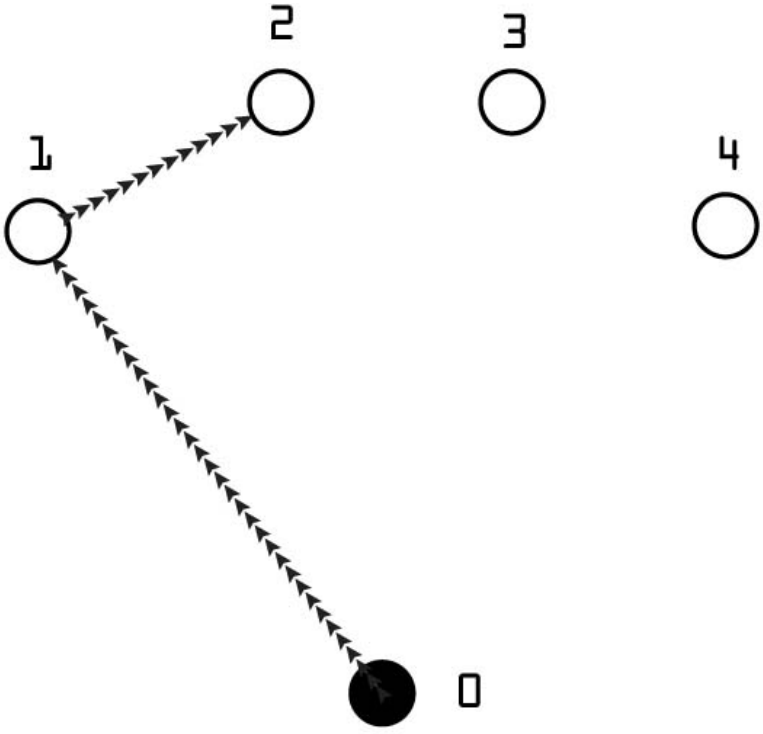
为了理解下一步，我们仔细看看如何用脚本创建3D网格：

在上图中，可以看到一个黑点代表角色，四个圆圈代表射线的终点。

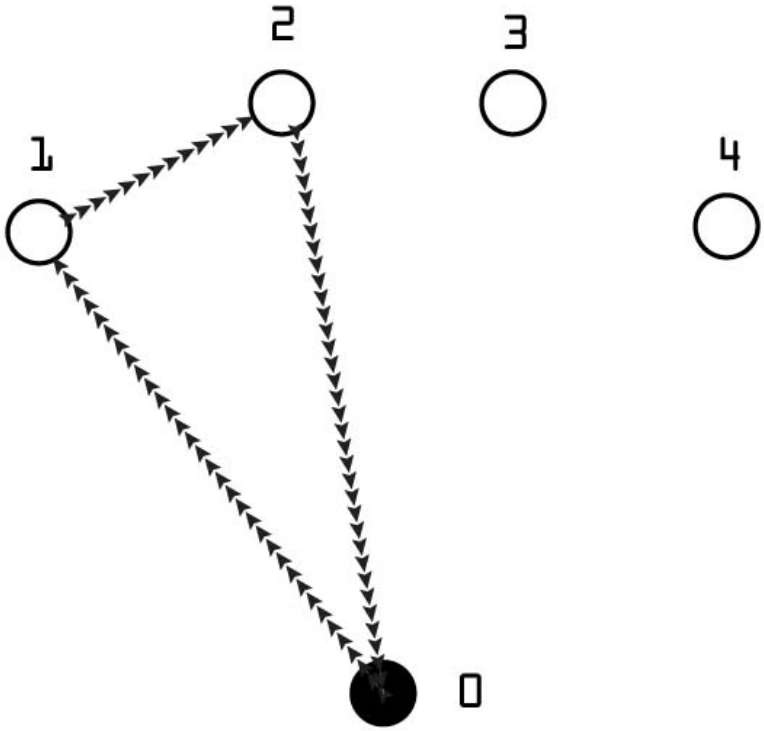


每个顶点都被赋予了一个编号，第一个顶点是角色，编号是0。然后顺时针旋转，下一个顶点从左边开始，往右数。



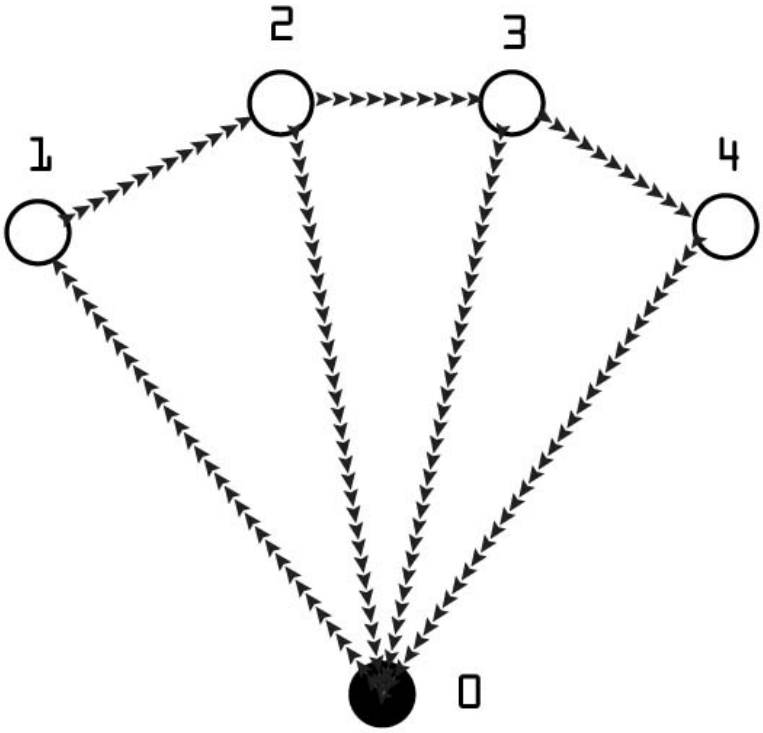


顶点0连接顶点1。



顶点1连接顶点2。

顶点2连回顶点0，形成一个三角形网格。



当第一个三角形创建好后，继续连接其他顶点，顺序是0>2>3>0，这样第二个三角形也创建好了。最后一个，0>3>4>0。现在我们想要在代码中描述这个过程，这样一来视野形成的数组可以表示为：

顶点总数是5：

```
[0,1,2,0,2,3,0,3,4]
```

```
v = 5
```

三角形总数是3：

三角形数量等于：

$$t = 3$$

$$t = v-2$$

这意味着数组长度为：

```
void DrawFieldOfView() {
int stepCount = Mathf.RoundToInt(viewAngle * meshResolution);
```

现在我们回到脚本，把刚刚解决的问题添加进去：

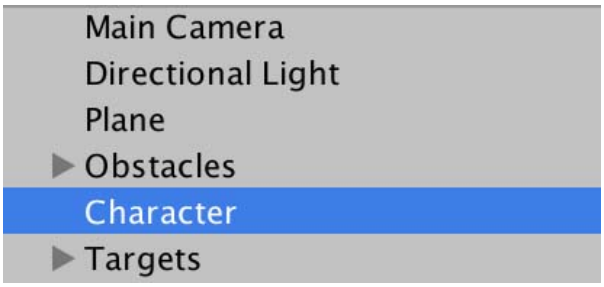
```
float stepAngleSize = viewAngle / stepCount;
 List<Vector3> viewPoints = new List<Vector3> ();
 ViewCastInfo oldViewCast = new ViewCastInfo ();
for (int i = 0; i <= stepCount; i++) {
float angle = transform.eulerAngles.y - viewAngle / 2 + stepAngleSize * i;
 ViewCastInfo newViewCast = ViewCast (angle);
 Debug.DrawLine(transform.position, transform.position +
DirFromAngle(angle, true) * viewRadius, Color.red);
 viewPoints.Add (newViewCast.point);
 }

int vertexCount = viewPoints.Count + 1;
 Vector3[] vertices = new Vector3[vertexCount];
int[] triangles = newint[(vertexCount-2) * 3];

 vertices [0] = Vector3.zero;
for (int i = 0; i < vertexCount - 1; i++) {
 vertices [i + 1] = viewPoints [i];

if (i < vertexCount - 2) {
 triangles [i * 3] = 0;
 triangles [i * 3 + 1] = i + 1;
 triangles [i * 3 + 2] = i + 2;
 }
 } }
```

现在，我们到脚本的顶部添加两个新的变量，public MeshFilter viewMeshFilter以及Mesh viewMesh：

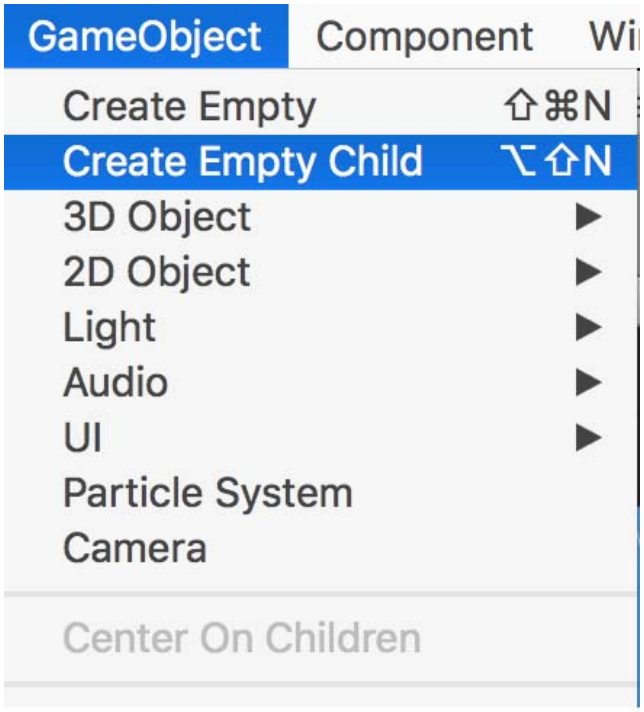
```
publicfloat viewRadius;
 [Range(0,360)]
publicfloat viewAngle;

public LayerMask targetMask;
public LayerMask obstacleMask;

 [HideInInspector]
public List<Transform> visibleTargets = new List<Transform>();

publicfloat meshResolution;

public MeshFilter viewMeshFilter;
 Mesh viewMesh;
```

```
void Start() {
viewMesh = new Mesh ();
viewMesh.name = "View Mesh";
viewMeshFilter.mesh = viewMesh;
StartCoroutine ("FindTargetsWithDelay", .2f);
}
```

接下来，我们需要在Start函数中初始化这些变量：

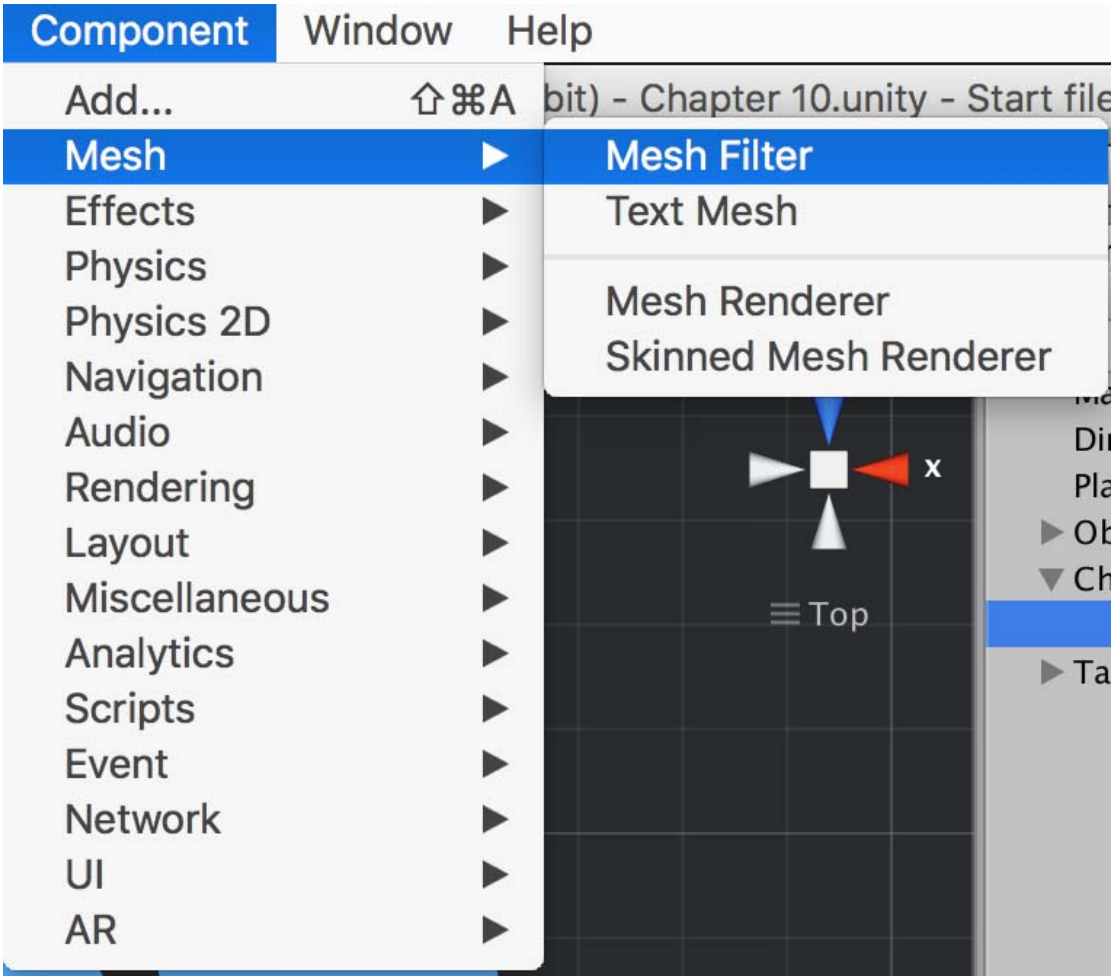


下一步是在编辑器里选择角色对象：

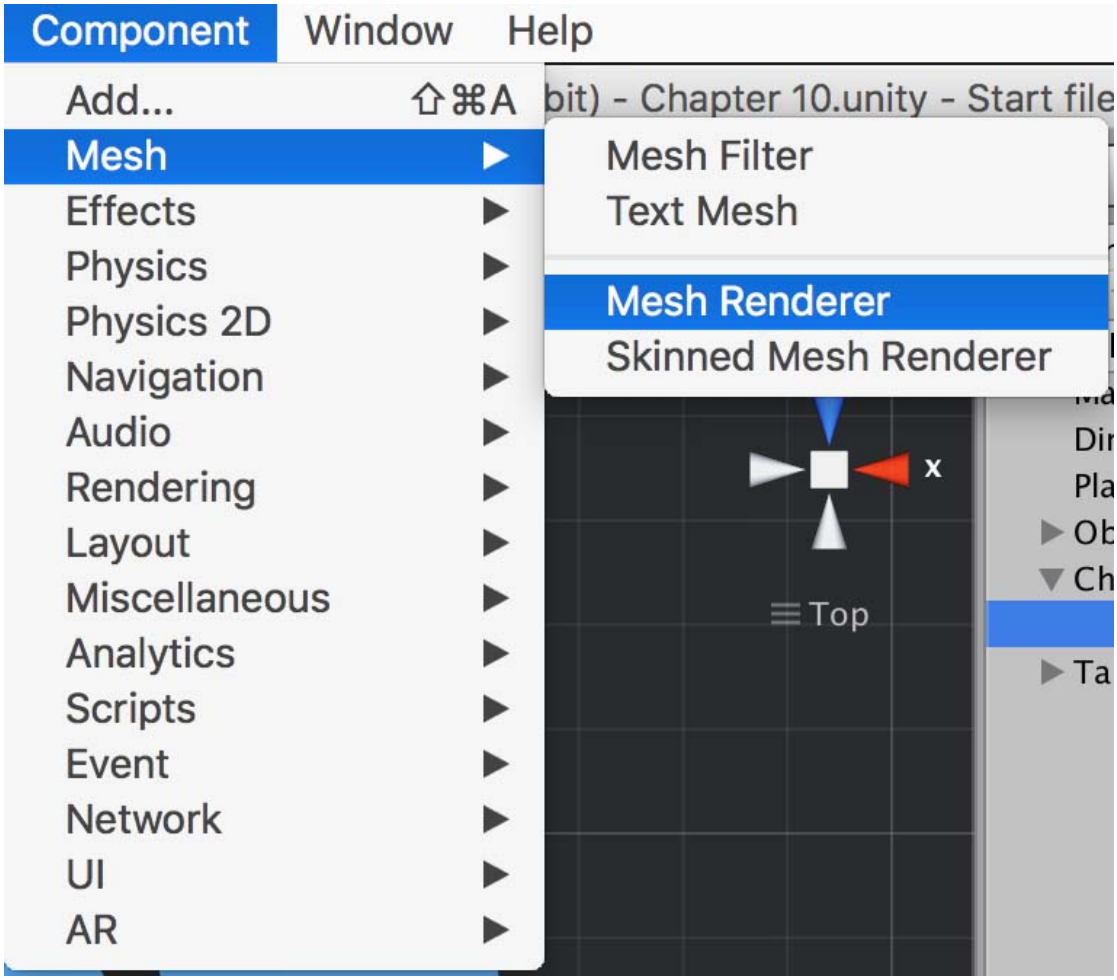


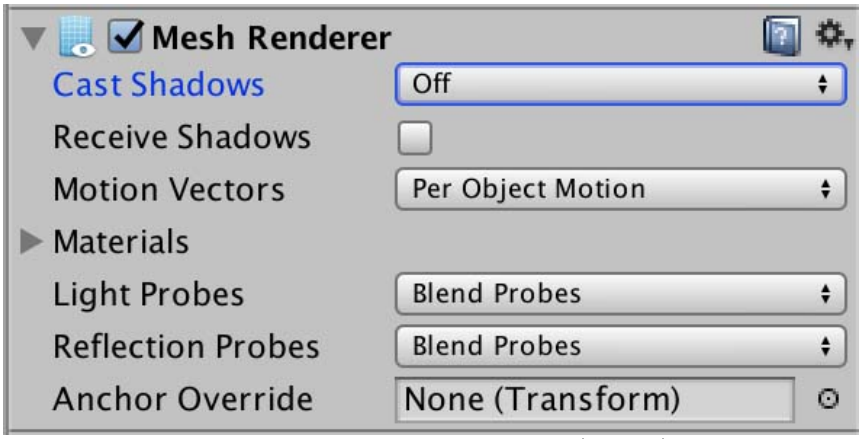
点击GameObject菜单，选择创建空的子物体：

将刚才创建的空对象改名为View Visualization。

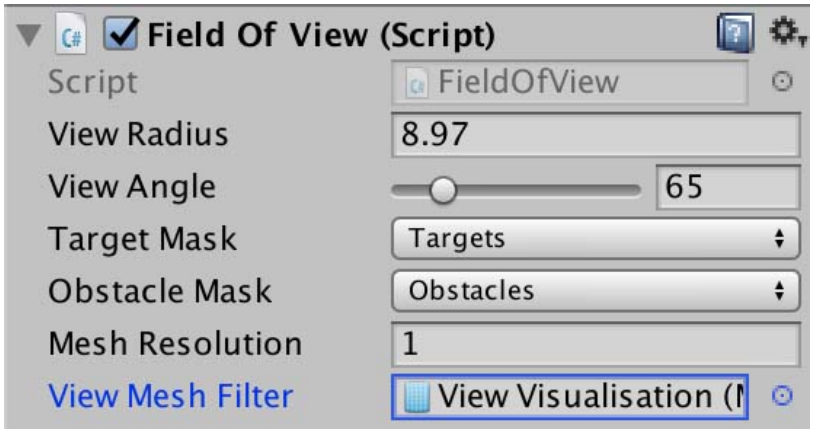


保持选择同样的物体，点击Component|Mesh|Mesh Filter，来添加一个网格过滤器给这个物体。





然后同样需要添加Mesh Renderer组件，Component|Mesh|Mesh Renderer。



这里可以关闭Cast Shadows和Receive Shadows，即关闭所有阴影。

最后，把刚才创建的物体拖到脚本变量View Mesh Filter中去，并修改Mesh Resolution为任何想要的值，这里我们选择1。

现在，我们回到脚本，再一次修改DrawFieldOfView方法。

```
void DrawFieldOfView() {
int stepCount = Mathf.RoundToInt(viewAngle * meshResolution);
float stepAngleSize = viewAngle / stepCount;
 List<Vector3> viewPoints = new List<Vector3> ();
 ViewCastInfo oldViewCast = new ViewCastInfo ();
for (int i = 0; i <= stepCount; i++) {
float angle = transform.eulerAngles.y - viewAngle / 2 + stepAngleSize * i;
 ViewCastInfo newViewCast = ViewCast (angle);
 viewPoints.Add (newViewCast.point);
 }

int vertexCount = viewPoints.Count + 1;
 Vector3[] vertices = new Vector3[vertexCount];
int[] triangles = newint[(vertexCount-2) * 3];

 vertices [0] = Vector3.zero;
for (int i = 0; i < vertexCount - 1; i++) {
 vertices [i + 1] = viewPoints [i];

if (i < vertexCount - 2) {
 triangles [i * 3] = 0;
 triangles [i * 3 + 1] = i + 1;
 triangles [i * 3 + 2] = i + 2;
 }
 }

 viewMesh.Clear ();

 viewMesh.vertices = vertices;
```

```
viewMesh.triangles = triangles;
viewMesh.RecalculateNormals ();
}
```

测试游戏，来看看我们都完成了什么：

当我们测试游戏时，会注意到游戏中渲染出的网格，这正是这一阶段的目标。

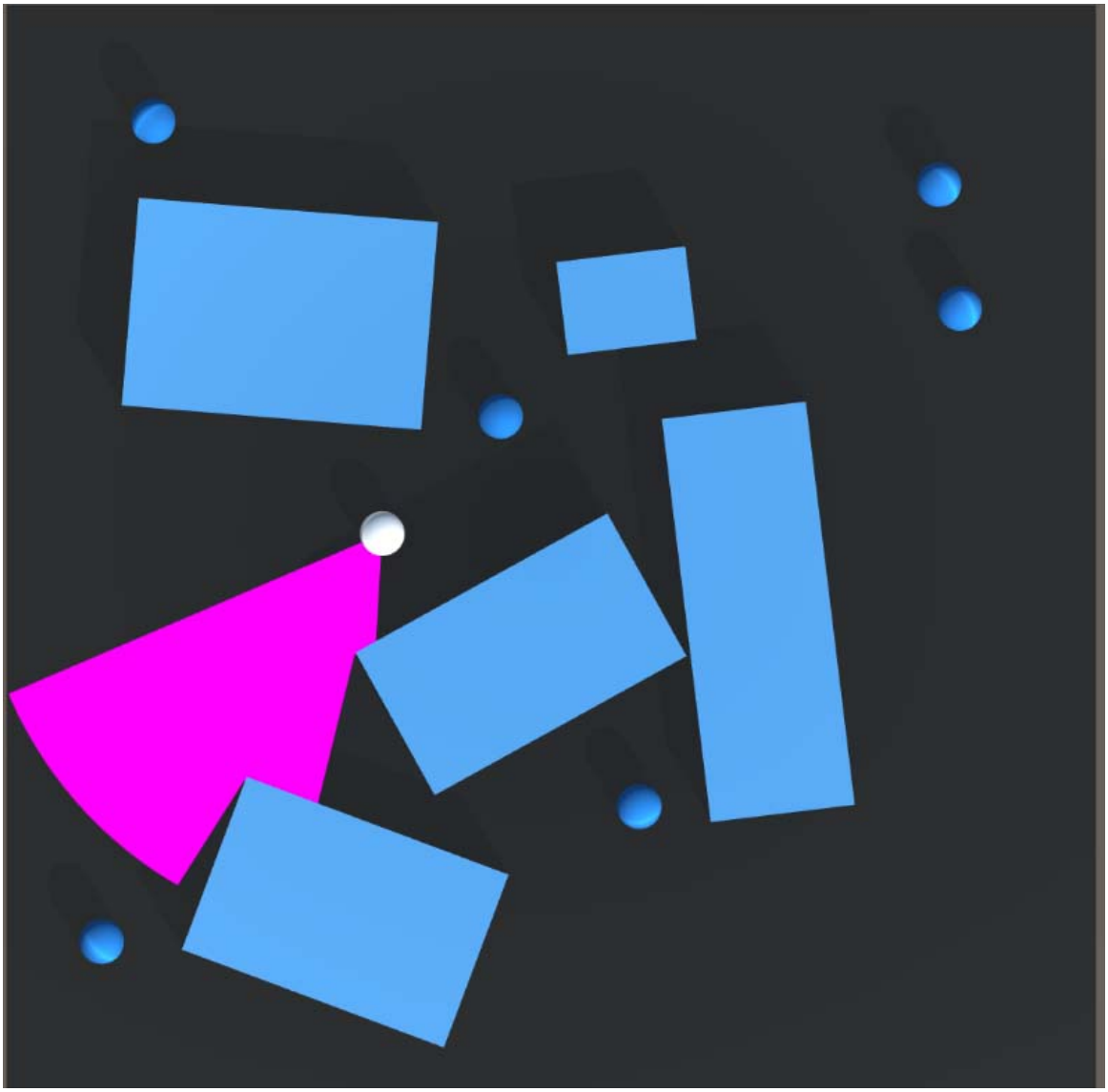

记住删除Debug.DrawLine那行代码，否则网格就不会在编辑器中出现。

为了优化视觉效果，我们需要修改viewPoints从世界坐标到本地坐标系。要做到这一点，需要使用InverseTransformPoint函数。

```
void DrawFieldOfView() {
int stepCount = Mathf.RoundToInt(viewAngle * meshResolution);
float stepAngleSize = viewAngle / stepCount;
 List<Vector3> viewPoints = new List<Vector3> ();
 ViewCastInfo oldViewCast = new ViewCastInfo ();
for (int i = 0; i <= stepCount; i++) {
float angle = transform.eulerAngles.y - viewAngle / 2 + stepAngleSize * i;
 ViewCastInfo newViewCast = ViewCast (angle);
 viewPoints.Add (newViewCast.point);
 }

int vertexCount = viewPoints.Count + 1;
 Vector3[] vertices = new Vector3[vertexCount];
int[] triangles = newint[(vertexCount-2) * 3];

 vertices [0] = Vector3.zero;
for (int i = 0; i < vertexCount - 1; i++) {
 vertices [i + 1] = transform.InverseTransformPoint(viewPoints [i]) +
Vector3.forward * maskCutawayDst;

if (i < vertexCount - 2) {
```

```
  triangles [i * 3] = 0;
  triangles [i * 3 + 1] = i + 1;
  triangles [i * 3 + 2] = i + 2;
  }
  }

  viewMesh.Clear ();

  viewMesh.vertices = vertices;
  viewMesh.triangles = triangles;
  viewMesh.RecalculateNormals (); }
```

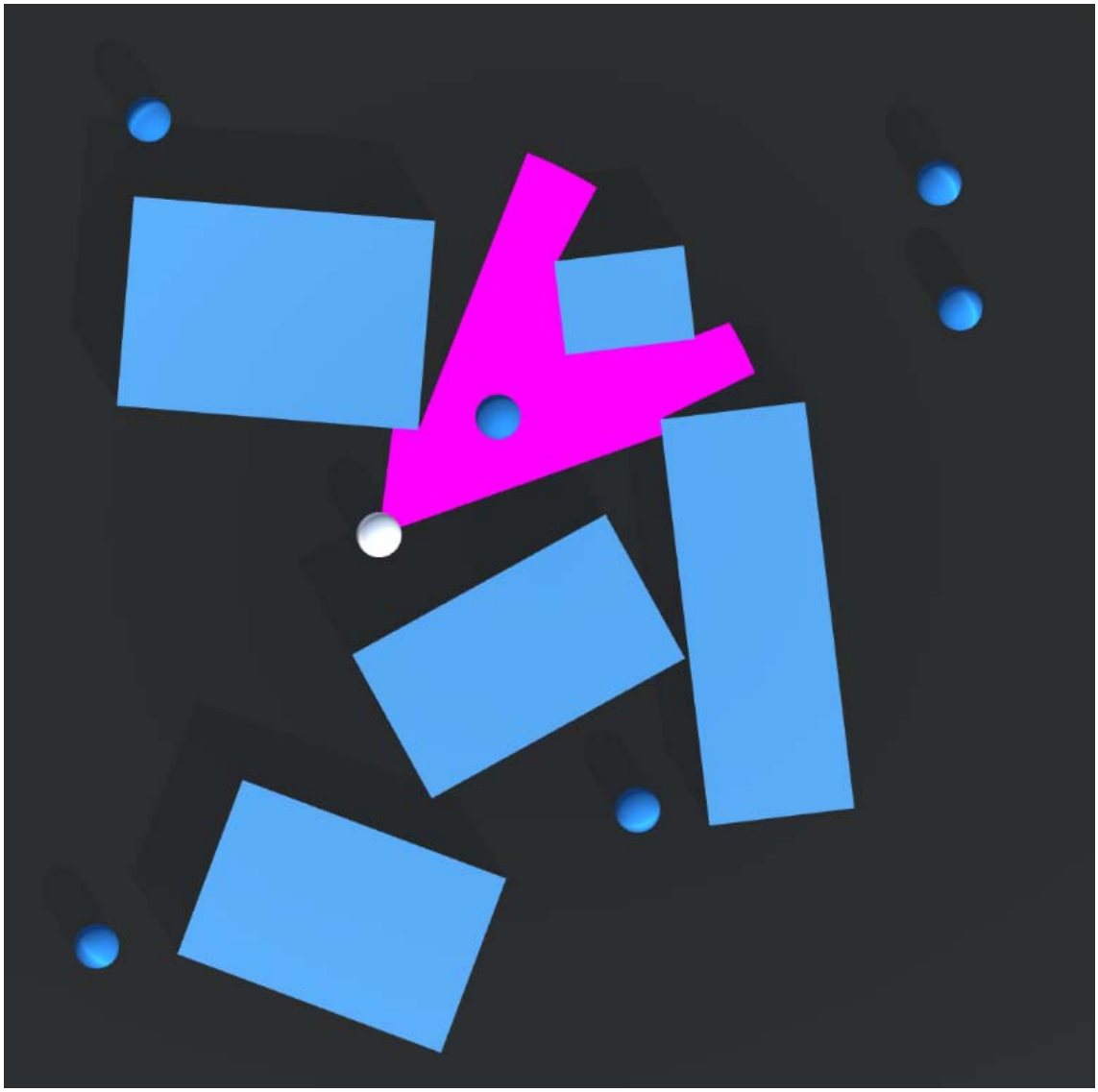
现在再测试一下，比刚才精确多了。



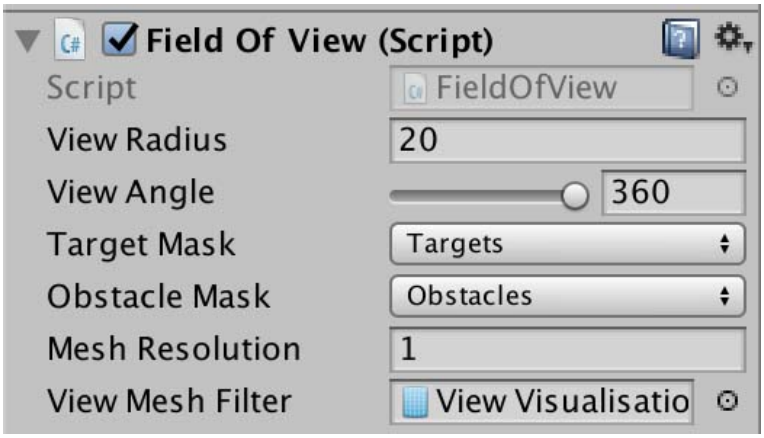
```
void LateUpdate() {
DrawFieldOfView ();
}
```

目前看起来已经不错了，但是我们需要改进代码，把Update函数修改为LateUpdate。

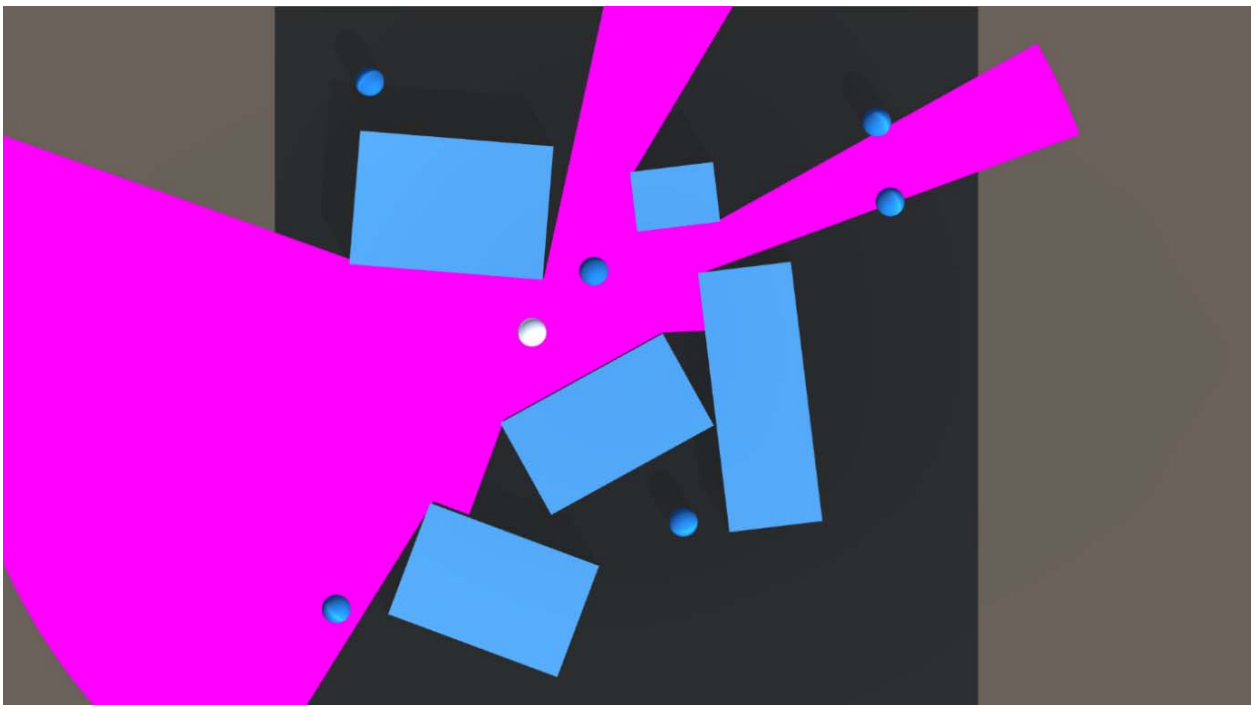
这样做网格的移动会更平滑。



修改完了这部分代码，我们就结束了这个例子，为角色实现了视觉系统的真实视野。



我们要做的仅仅是修改参数，让角色对环境的感知力增强或减弱，从而来调整到想要



的结果。

例如，如果我们设置View Angle到360，这会让角色能够感知到周围所有事情，如果我们减小这个值，就会得到一个更真实的视野，就像《合金装备》游戏里用到的。

到这里，我们已经可以拿出一个潜入类游戏并重现它们最标志性的功能，比如真实视野或者声音感知。前面已经学习到了整个框架，现在我们可以以此为出发点开发属于自己的游戏了。

[1] 脚本名称为FieldOfView，类名也是FieldOfView。——译者注

10.5 总结

本章中，我们揭示了潜入类游戏的实现原理，以及我们怎样重新创建同样的系统用在自己的游戏里。我们从简单的例子开始做到了一个复杂的例子，这让我们能够根据游戏是依赖于潜入的玩法还是简单地让角色能够感知到玩家即可，来决定在自己创作的游戏中哪种方法更合适。本章中学到的功能还可以继续扩展，用在之前的任何例子里，用这种方法提升碰撞检测、寻路、决策和动画等其他功能，让它们从简单功能性的实现进化为能够处理更加真实的情况。

我们创作游戏的方式一直在进步，每一个新发布的游戏都会带来全新的方法来创造某些东西，要做到这一点，我们必须愿意去试验、去组合所有已知的东西，甚至打碎重建我们的知识来达到想要的结果，就算那看起来十分复杂。有时，这只是一个探索基本概念并扩展它们的过程，把一个简单的想法变成一个复杂的系统。