*Physics for Game Developers*

# 游戏开发物理学

*David M. Bourg* 著
*O'Reilly Taiwan*公司 译
霍炬 审校

O'REILLY®
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Game Programming

O'REILLY®

# 游戏开发物理学

撞球、导弹弹道、高速赛车的转弯动力，等等——应用正确的物理定律可逼真地模拟游戏中任何弹跳、飞行、翻滚、滑行或非静止的物体，创建真实可信的游戏、动画内容。对于那些想增加物理真实度的游戏开发人员，本书乃绝佳的参考书。

第一部分是复习基本概念及讨论刚体动力学的机械力学初级课程；第二部分将这些概念应用到现实生活的问题上，诸如抛射体、船舰、飞机与汽车；第三部分介绍实时模拟器并示范如何将它们应用到电脑游戏中。许多特定游戏的元素都需要实际物理的模拟才能达到真实的效果，包括：

- 火箭与导弹弹道，包括燃料消耗的影响
- 物体的碰撞，如撞球
- 急转弯道的汽车稳定度
- 船舰与浮于水面的交通工具的动力学
- 棒球被球棒击打出去的飞行路径
- 飞机的飞行特性

阅读本书之前，你不必是物理专家，但作者假设非物理及非工程学系的读者需有大专程度的经典物理学知识。读者应熟悉三角函数、向量与矩阵运算（可参阅附录，其中有相关的参考公式与特性说明），并且需有大专程度的微积分（包括显函数的微分和积分）知识。

ISBN 7-121-00208-6

9 787121 002083 >

O'Reilly Media, Inc. 授权电子工业出版社出版

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-121-00208-6

定价：48.00 元

# 游戏开发物理学

*David M. Bourg* 著

*O'Reilly Taiwan*公司 译

霍炬 审校

O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo*

O'Reilly Media, Inc. 授权电子工业出版社出版

電子工業出版社

**Publishing House of Electronics Industry**

北京•BEIJING

**图书在版编目（CIP）数据**

游戏开发物理学 /（美）布兰革（Bourg, D. M.）著；O'Reilly Taiwan 公司译 .
－北京：电子工业出版社，2004.10
书名原文：Physics for Game Developers
ISBN 7-121-00208-6

I. 游 ... II. ①布 ... ② O... III. 游戏－软件开发－物理学 IV. TP311.5
中国版本图书馆 CIP 数据核字（2004）第 079170 号

版权贸易合同登记号
图字：01-2004-1768

©2002 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2004. Authorized translation of the English edition, 2002 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2002。

简体中文版由电子工业出版社出版 2004。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名 / 游戏开发物理学
书　　号 / ISBN 7-121-00208-6
责任编辑 / 朱沐红　高洪霞
封面设计 / Ellie Volckhausen，张健
出版发行 / 电子工业出版社（http://www.phei.com.cn）
地　　址 / 北京市海淀区万寿路 173 信箱（邮政编码 100036）
经　　销 / 各地新华书店
印　　刷 / 北京天竺颖华印刷厂
开　　本 / 787 × 980　1/16　印张：22.25　字数：410 千字
印　　次 / 2004 年 10 月第一次印刷
印　　数 / 0001-4000 册
定　　价 / 48.00 元（册）

# O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

# 目录


前言 ..... 1

第一章 基本概念 ..... 9

牛顿运动定律 ..... 9

单位与计量 ..... 10

坐标系 ..... 12

向量 ..... 13

质量、质心与转动惯量 ..... 13

牛顿第二运动定律 ..... 23

惯性张量 ..... 27

第二章 运动学 ..... 32

简介 ..... 32

速度与加速度 ..... 33

定加速度 ..... 35

不定加速度 ..... 37

2D 粒子运动学 ..... 38

3D 粒子运动学 ..... 40

粒子爆炸的运动学 ..... 50

刚体运动学 ..... 55

局部坐标轴 ..... 56
角速度及加速度 ..... 57

第三章 作用力 ..... 63
简介 ..... 63
力场 ..... 64
摩擦力 ..... 65
流体动阻力 ..... 66
压强 ..... 68
浮力 ..... 68
弹簧和阻尼 ..... 70
力与力矩 ..... 71

第四章 动力学 ..... 75
2D 的粒子动力学 ..... 76
3D 粒子动力学 ..... 81
刚体动力学 ..... 88

第五章 碰撞 ..... 93
冲量 / 动量定律 ..... 94
撞击 ..... 95
线性及角冲量 ..... 100
摩擦力 ..... 103

第六章 抛体 ..... 106
基本的抛体轨道 ..... 107
阻力 ..... 111
马格纳斯效应 ..... 118
变动质量 ..... 123

第七章 飞机 ..... 125

几何形状 ..... 127

升力与阻力 ..... 128

其他的作用力 ..... 133

控制 ..... 134

飞行模拟 ..... 135

第八章 船舰 ..... 149

漂浮 ..... 150

体积 ..... 152

阻力 ..... 162

虚质量 ..... 164

第九章 气垫船 ..... 166

运作原理 ..... 166

阻力 ..... 168

第十章 汽车运动 ..... 171

阻力 ..... 171

功率 ..... 172

刹车距离 ..... 173

道路边坡 ..... 174

第十一章 实时模拟 ..... 176

运动方程的积分 ..... 177

欧拉法 ..... 178

其他的方法 ..... 184

第十二章 2D 刚体模拟器 ..... 188

模型 ..... 189

积分函数 ..... 195

飞行控制 ..... 198

绘图 ..... 202

第十三章 碰撞反应实现 ..... 209

线性碰撞反应 ..... 210

角运动 ..... 215

第十四章 刚体的转动 ..... 227

旋转矩阵法 ..... 228

四元数法 ..... 231

第十五章 3D 刚体模拟器 ..... 233

模型 ..... 234

积分函数 ..... 238

飞行控制 ..... 241

绘图 ..... 245

第十六章 3D 多重物体模拟 ..... 252

模型 ..... 253

积分函数 ..... 267

碰撞反应 ..... 269

参数调整 ..... 272

第十七章 粒子系统 ..... 274

模型 ..... 274

积分函数 ..... 284

碰撞反应 ..... 285
参数调整 ..... 286

附录一 向量的运算 ..... 289

附录二 矩阵的运算 ..... 299

附录三 四元数的运算 ..... 308

参考文献 ..... 319

索引 ..... 325

# 前言

## 谁来读“我”

简单地说，本书是专为没有扎实的力学和物理背景，却要在游戏中负责加入物理模拟的电脑游戏设计师而设计的。

身为游戏设计师或者电脑玩家，你也许曾经见过号称“超真实”或“写实”的产品。同时，你或公司的营销部门，也许正在考虑如何在产品中添加物理的真实度。或许你想要做些全新的尝试，去探索真实的物理学。但问题是你早在期末考试之后就将物理的教材抛入湖中，而且之后从未再接触此领域。也许你买了一套相当酷的物理引擎，但是却不了解其中运作的原理或执行模拟时此引擎的影响。或者也许你要负责调整其他人所写的物理学程序代码，但是你却不清楚其运作的原理。这表明你该阅读本书了。

当然你可以搜寻网络、专业期刊和杂志获得相关资料与指引，让游戏能加入物理的真实性。你也可以由旧物理教材中重新开始。然而你也许会发现，这些资料或者由于太过广泛而不适于直接应用，或者由于难度太大而需要搜寻其他资料以先打好基础。本书将搜集所有需要的资料，作为你（游戏设计师）加入物理真实性以丰富游戏内容的出发点。

本书并不只是针对各式各样的问题提供不同的范例程序而已。网络中已经充斥太多这类的范例程序（当然也包括一些非常好的程序）。除了对某些问题提供解答之外，本书的目的是要让读者对于相关领域能有彻底而基础性的了解，进而对于其他的问题开发出自己的解决方案。本书将详细地解说运用在游戏开发领域的物理学原理，除了范例程序外，还将用手写的计算范例进行补充说明。

# 读者应有的基础

虽然读者并不一定都要是物理专家，不过非物理或工程学系的读者至少要有大专程度的基础物理学知识。你不必将物理学背景谨记在心，因为本书前几章会讲解与游戏物理相关的主题。

读者必须精通三角函数、向量和矩阵数学，在附录中也会提供这些领域的参考资料。此外读者需要有大专程度的基本微积分知识，包括对显函数（explicit function）的积分与微分。数值积分与微分又是另一个领域，在本书的稍后几章将会详细地介绍这些技巧。

# 力学

当我构思本书的概念时，与人交谈当中若提及“真实物理学”或“实时模拟”这些名词，这些人通常立刻会想到飞行模拟程序。顶尖的飞行模拟当然与本书的内容有关；然而许多不同类型的游戏和特定游戏的元素，也能从物理拟真增加几分可玩性。

想想这个例子：你正在做一个即将轰动的射击游戏，游戏中有第一人称的3D画面、华丽的贴图和令人震撼的音效，但是却缺少某个元素。这个元素就是真实度。具体地说，你希望通过挑战玩家的射击技术让游戏感觉更真实，要实现这个目标，就要加入某些考量，如与目标之间的距离、风速和方向，枪口速度，等等。你不必虚构这些元素，可以根据物理原理来模拟这些元素的物理特性。任职于MathEngine Plc的Gary Powell这么说：“虚拟世界的图像和令人沉迷的体验、以精密的多边形精心建立的模型、复杂的贴图和先进的光源投影，当物体开始移动和互动时通常是令人震惊的。”（注1）“这完全是互动性和吸引力”，Havok.com的CEO Steven Collins博士这么说（注2）。我认为这些人都找对了方向。为什么在增加游戏世界的真实性上耗费这么多时间和精力，而不再多走几步让游戏中的动作更真实呢？

以下是可利用物理学增加真实度的游戏元素的例子：

- 火箭和导弹的飞行轨迹，包含燃料消耗的影响。
- 物体（例如撞球）之间的碰撞。

---

注1：Gary Powell任职于MathEngine Plc。其产品包括Dynamics Toolkit 2和Collision Toolkit 1，可以处理单一或多重物体的动力学。如果读者对于他们的产品有兴趣可以去逛逛网站*www.mathengine.com*。

注2：Collins博士是Havok.com的CEO。这家公司的技术可以处理刚体、柔体、布料、流体和粒子的动力学。你可以在*www.havok.com*找到相关资料。

- 大型物体（例如星球和太空堡垒）之间的万有引力作用。
- 赛车急转弯时的稳定性。
- 船或浮于水面的交通工具的动力学。
- 棒球被球棒击打出去的飞行路径。
- 被丢进帽子的纸牌的飞行路径。

这绝不是一个完整的清单，而仅仅是一些提供正确概念的例子。游戏中有许多弹跳、飞行、翻滚、滑翔或非静止的物体，可被真实地模拟，以创造吸引人的、可信的游戏内容。

如何实现这样的真实性呢？当然，说到物理学原理，就要回到本节的标题：力学。物理学是一门领域极广的科学，包含许多不同却相关的领域。在真实游戏内容中最常使用的主题是力学，这也就是“真实的物理学”所代表的意义。

从定义上讲，力学研究的是在静止或运动时的物体，以及作用在物体上的力的影响。因此力学可分为静力学与动力学，前者特别注重对物体处于静止状态的研究，而后者则研究物体的运动状态。物理学中最古老且最常被研究的领域 —— 力学的起源，可追溯到两千年前亚里斯多德的时代。此领域另一个更早的论述为“Problem of Mechanics（力学的问题）”，但是这个作品的起源无法考证。虽然某些早期的研究让物理现象变成魔术领域，但是伟大的物理学家如伽利略、开普勒、欧拉、拉格朗日、达朗贝尔、牛顿、爱因斯坦等的不懈努力，使人们对物理学的了解能发展到今天这种程度，让我们目前所见技术先进到惊人程度。

为了使游戏能生动活泼并栩栩如生，我将把重点放在运动中的物体上，并钻研此领域的力学。在力学的领域中，还有许多特定的领域需要研究。例如，运动学将重点放在物体的运动而不是作用在物体上的力上。而动力学同时注重物体的运动和作用在物体上并影响物体运动的力。本书中将详细探讨这两个主题。

## 章节编排

物理拟真在游戏界中并不是个新事物；事实上，正在销售中的许多游戏都宣称具有物理引擎。许多 3D 模型和动画软件也有内建的物理引擎，能写实地模拟出特定的动作。自然而然地，杂志上的文章时常出现对各种物理式游戏的讨论。同样，在另一个层次，对于即时刚体（注 3）模拟的研究已经有好几年了，而技术期刊也满是关于这一主题不同

注 3：刚体正式的定义为粒子系统组成的物体，其粒子能彼此保持固定的距离而不会有相对的缩放或旋转。尽管力学的范畴也包括处理柔体甚至是像水一样的流体，但这里只讨论刚体。

方面的论文。这些论文讲述范围从多重、相连刚体的模拟到布料的模拟。然而，正如之前所述，虽然这些都是吸引人的主题和有价值的资源，对于游戏开发人员而言要直接使用却是有限制的。因为游戏开发人员必须对这些领域的力学有完整的认知，对于有些资源，还要从基础学起。况且，其中大多的重点主要放在解运动方程的数学上，并未真正处理作用在模拟中的物体或系统上的力。我问过在 Animats 工作的 John Nagle，在开发以物理为基础的模拟游戏时，对他而言哪个部分是最困难的。他的回答是开发数值稳定且健全的程序代码（注 4）。Gary Powell 对此发表回应，他告诉我说：最小化对产品稳定度与真实行为的参数调整所需的成本是最困难的挑战之一。我同意关于处理物体动作的数学，要使其速度和稳定兼具是模拟器最困难的部分。保持模拟程序在初始及进行中作用力的完整性和正确性也是如此。稍后在本书中就可以看到，作用力会控制模拟程序中物体的行为。如果想让物体有真实的行为，就要正确地将物体建模。

对于力学，以及作用于特定物体或系统上的作用力的实质的必要认知，都会影响本书的编排方式。

第一章至第五章基本上是力学的基础课程，并且由复习基本概念开始，再慢慢地建立刚体力学更具挑战性的概念。这几章的目的是提供充足的力学复习课程，让你可以阅读更多进阶的文章。如果你对于力学已经相当熟悉，可以略过这些章节直接阅读第六章。

第一章：基本概念

本章是热身的章节，内容包含一些最基础的定理，可作为本书其他章节的参考。将会提到的主题包括质量、质心、牛顿运动定律、惯性、单位与度量，以及向量。

第二章：运动学

本章的主题有线性速度与角速度、加速度、动量，以及 2D 及 3D 环境中粒子与刚体的一般性运动。

第三章：作用力

本章包括力与力矩的原理，可以作为由运动学跨入动力学的桥梁，会讨论一般种类的作用力，包括阻力、力场和压力。

第四章：动力学

本章结合第二章与第三章的元素，介绍运动学的主题，解释动力学与运动学之间的差异。关于粒子与刚体的动力学，在 2D 及 3D 环境中将进行更进一步的讨论。

---

注 4：John Nagle 是 Falling Bodies 的开发人员，这是在 Softimage 3D 软件上的一种动力学嵌入模块。你可以在 *www.animats.com* 上找到他的专利技术。

第五章：碰撞

本章包含粒子与刚体的碰撞反应，也就是两物体在相互碰撞后会发生什么事情。

第六章至第十章开始介绍真实世界中的一些问题。这些章节的重点在于模型化某些实体系统、尤其是其中涉及的作用力，并给予你健全的知识，因此系统能正确地在实时模拟器中模型化。这个部分的范例并没有完全包括所有可能在游戏中会模拟的系统。选择这些系统是来解释某些和广泛的问题有关的物理现象和概念。

第六章：抛体

由第六章开始的一系列章节，介绍可能在游戏模拟时遇到的问题，并提供实际上可行的内容。本章的主题是抛体，讨论在抛体飞行中所受的力，以及影响速度与轨迹的因素。

第七章：飞机

本章的重点是飞行的要素，包括推进力、阻力、几何形状、质量以及最重要的升力。本章同时也是实现3D实时模拟器的起点，这个模拟器的实现将在第十五章进行介绍。

第八章：船舰

本章讨论排水型船舰的基本元素，包括浮力、稳定度、体积、阻力和速度。

第九章：气垫船

气垫船同时有飞机和船的特性。本章讨论独特的交通工具 —— 气垫船 —— 的特性。包含的主题有盘旋飞行、空气静升力和方向控制。

第十章：汽车

本章将讨论汽车某些方面的性能，包括空气阻力、滚转阻力、滑行距离和道路边坡等。

第十一章至第十七章和3个附录，将介绍实时模拟程序。这些章节会介绍实时模拟器，并讨论在这个领域中可以应用在电脑游戏上的不同方面。实时模拟器的主题是非常广泛的，必须用一本书的篇幅才可能介绍，所以本书只介绍其中的基础。我将带领你实现一个有两艘气垫船的2D模拟器、3D的飞行模拟器、有碰撞反应的3D多重物体模拟器，以及使用粒子和弹簧的柔体模拟器。

第十一章：实时模拟

本章介绍的主题是实时模拟，包含一些用来解运动微分方程的数值积分法。

第十二章：2D刚体模拟器

本章介绍实现简单的2D粒子和刚体的模拟器的实用面。本章还将实现有两艘气垫船的简单实时模拟器。

第十三章：实现碰撞反应

本章将示范如何在实时模拟器中实现第五章所讨论的碰撞反应。具体地说，就是在第十二章开发的气垫船模拟程序中加入碰撞反应。

第十四章：刚体旋转

在进入3D模拟器之前，要先介绍在3D环境中表现刚体旋转方位会遇到的问题。这里会考虑欧拉角、旋转矩阵和四元数。

第十五章：3D 刚体模拟器

本章结合了第十一章至第十四章中所讨论的内容，并介绍在实现简单的3D刚体模拟器时的实用面。这里将介绍如何根据第七章介绍的空气动力模型来开发简单的飞行模拟器。

第十六章：3D 多重物体模拟

本章拓展在第十五章所介绍的范例程序，并处理在3D环境中多重物体在碰撞侦测和反应方面的能力。这里介绍的范例由一辆汽车和一对测试碰撞方块组成。

第十七章：粒子系统

本章示范可以用简单的粒子模拟系统来做什么。具体而言，本章介绍使用系统化的粒子和弹簧组成仿真的布质。这个模拟程序模拟一个旗子挂在旗杆上时迎风飘扬的样子。

附录一：向量运算

本附录介绍如何实现 C++ 的类，此类包含所有在 2D 和 3D 模拟程序中会用到的向量运算。

附录二：矩阵运算

本附录实现一个类，包含所有需要用来处理 3 × 3 矩阵的运算。

附录三：四元数运算

本附录实现一个类，此类包含所有在 3D 模拟程序中处理四元数时会用到的运算。

除了与实时模拟有关的资源外，在书后的参考文献中也提供关于力学、数学，以及与其他某些技术主题（例如空气动力学的书籍）有关的资料。

# 排版约定

以下是本书使用的排版体裁：

等宽字体（Constant width）

用来表示命令模式的电脑输出、范例程序代码。

等宽斜体（*Constant width italic*）

用来表示在范例程序代码中的变量。

*斜体字（italic）*

用来介绍新名词和表示URL、变量、文件名和目录、命令，以及扩展文件名。

***粗斜体字（Bold italic）***

用来表示向量、矩阵等。

# 建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

O'Reilly的每一本书都有专属网页，你可以在此找到关于该本书籍的相关信息，包括范例程序的下载、勘误表与相关资源的链接。

*http://www.oreilly.com/catalog/physicsgame/*

要评价本书或询问有关技术问题，请发电子邮件到：

*bookquestions@oreilly.com*
*info@mail.oreilly.com.cn*

要了解更多有关书籍、会议、资源中心和O'Reilly Network的信息，请参见O'Reilly的Web站点：

*http://www.oreilly.com*
*http://www.oreilly.com.cn*

# 致谢

我要感谢本书编辑 Robert Denn，因为他对我的著作做了技术检查，并给予了有深刻见解的评析和建议，更不用提他的耐心了。我也要感谢 O'Reilly 同意我这个计划，给我机会能实行这个长久以来的点子。此外还要特别感谢所有 O'Reilly 的产品与技术人员。还要感谢 MathEngine Plc 的 Gary Powell、Havok.com 的 Steven Collins 博士和 Animats 的 John Nagle。因为他们专业的评论和我许多对于游戏物理和实时模拟器的问题。我不能忘记技术审校 Sam Kalat、Mike Keith 和 Michelle McDonald，以及他们完整的回答卓越的评析和建议。同样要特别感谢我的商业伙伴兼长久的朋友 Glenn Seemann，他带领我开始电脑游戏的研发。最后我想要感谢我亲爱的妻子，也是最好的朋友 Helena，因为她无尽的支持和鼓励，以及我们的小女儿 Natalia，谢谢她让我的每一天都特别。

# 第一章

# 基本概念

本章是热身的章节，将介绍其他几章所使用或提到的基本原理。首先介绍对于力学研究非常重要的牛顿运动定律。接下来会讨论单位与计量，并解释在运算过程中保持单位一致的重要性。同时在本章也会看到不同的物理量所对应的单位。在讨论完单位后，将定义常用的坐标系，以及用来作为在参考物体时的标准体系结构。接下来解释质量、质心及转动惯量（moment of inertia）的概念，并且介绍在质量的组合下，如何计算这些物理量。本章最后会对牛顿第二运动定律做进一步的讨论，同时简单地介绍向量的概念。

## 牛顿运动定律

在17世纪末期（约1687年左右），以艾萨克·牛顿（Isaac Newton）在《Philosophiae Naturalis Principia Mathematia》期刊上发表一套力学原理。牛顿在这篇文章中说明了目前广为人知的运动定律。

定律一

物体在无外力作用的情况下，会维持静止不动状态，或继续做匀速直线运动。这就是“惯性”的概念。

定律二

物体的加速度与作用于该物体上的合力成正比，同时加速度的方向与作用力的方向相同。

定律三

对于所有作用在物体上的力（作用力），都会有一个大小相同但方向相反的反作用

力，而且作用力与反作用力处在同一条直线上（但此反作用力并非作用于该物体上，故两力不能互相抵消）。

这些定律构成了在力学领域中进行分析的基础。而与动力学研究关系最大的应该就是第二运动定律了，表示如下：

$$F = ma$$

式中，$F$是作用在物体上的合力，$m$是物体的质量，$a$是作用于物体重（质）心（译注1）上的线性加速度。在本章中，对第二运动定律会有更详尽的讨论。但在这之前，先介绍一些较基本的术语。

# 单位与计量

在多年以来所教授的工程课程中，我发现学生最容易犯的错误之一就是在运算时使用错误的单位。常常由于单位的不一致而产生一些奇奇怪怪的答案。例如，在船只性能的评估上，最常误用的单位就是速度，很多人常常忘了要将速度的单位由节（knot）转换成ft/s（英尺/秒）或m/s（米/秒）；一节相当于1.69 ft/s。同时在这个领域中，很多所求的量值与速度的平方成正比，这样一来，错误的速度单位会使计算结果与正确答案产生高达185%的差距！所以当你觉得运算结果很奇怪时，第一件事就是回头检查公式中的因次是否一致。

要检查因次的一致性，就必须考虑计量单位及其组成部分的因次。在这里我们讨论的并不是2D或3D空间的维度，而是指组成各种物理度量单位的基本因次（译注2）。这些基本的因次包含质量、长度与时间。

了解这些因次并能导出其他单位的因次组合，是极为重要的。因为这样才能保证在计算过程中因次的一致。例如，物体的重量是以力为测量的单位，而力可以分解成如下的因次式：

$$F = (M)(L/T^2)$$

式中，$M$是质量，$L$是长度，而$T$是时间。这个公式看起来很熟悉吗？如果考虑到加速度的组成单位为$(L/T^2)$，令$a$代表加速度的符号，而$m$代表物体的质量，会得到：

$$F = ma$$

---

译注1：在物体质量分布的范围不大的情况下（即物体位于均匀的重力场中），重心与质心可视为同一点（重合）；若物体为同一材质且密度均匀，其形心、重心与质心重合。

译注2：因次与维度英文都叫“dimension”；另外因次也称为量纲。

这就是著名的牛顿第二运动定律公式。稍后会对这个公式做更进一步的介绍。

这里并非是要推导出这个著名的公式，而是要检查其因次的一致性。即在任何表示物体上的作用力的公式中，单位最好都是由一致的因次式$(M)(L/T^2)$所组成的。现在看起来好像很无聊，但是，当物体上的作用力公式越来越复杂时，你就会想将这些公式分解成组成的因次，这样就能检查其因次的一致性。稍后的内容中，将使用实际的单位（英制或国际单位制都有）来表示物理量，除非你要显示这些数值给玩家看，不然游戏中使用英制或是国际单位都无所谓，重要的是保持因次的一致性。

为了弄清这一点，请看以下这个计算物体在流体（如水）中移动时所受摩擦力的公式：

$$R_f = 1/2\rho V^2 S C_f$$

式中，$R_f$表示因为摩擦而产生的阻力，$\rho$是水的密度，$V$是物体移动的速度，$S$是物体沉在水中的面积，而$C_f$是这物体的实验（由实验决定）阻力系数。现在，重新以基本的因次取代各变量来改写这个公式，会发现等号两边的因次是相同的。因为$R_f$是一种力，其基本因次式应为：

$$(M)(L/T^2)$$

也就表示等式右边的所有基本因次合并后，应该也产生同样的因次式。密度、速度跟面积单位的因次式如下：

密度

$$(M)/(L^3)$$

速度

$$(L)/(T)$$

面积

$$(L^2)$$

将$\rho V^2 S$的因次式合并会产生：

$$[(M)/(L^3)][(L)/(T)]^2[L^2]$$

将分子与分母部分的因次整理过后得到：

$$(ML^2L^2)/(L^3T^2)$$

在消去同时出现在分子与分母中的因次后，会得到：

$$M(L/T^2)$$

结果跟阻力 $R_f$ 的因次式是一样的。通过这个练习可以发现，用来代表摩擦系数的 $C_f$ 这个实验值是没有因次的，也就是说它是没有单位的常数值。

接下来，我们会介绍一些较常用的物理量所使用的符号、组成因次，以及在英制与国际单位制中所使用的单位。这些信息摘要如表 1-1 所示。

**表 1-1：常用物理量及其单位**

| 物理量 | 符号 | 因次 | 英制单位 | 国际单位 |
|---|---|---|---|---|
| 长度 | $L$ | $L$ | 英尺（ft） | 米（m） |
| 质量 | $m$ | $M$ | 斯勒格（Slug） | 千克（kg） |
| 时间 | $T$ | $T$ | 秒（s） | 秒（s） |
| 力 | $F$ | $M(L/T^2)$ | 英磅（lb） | 牛顿（N） |
| 线性加速度 | $a$ | $L/T^2$ | $ft/s^2$ | $m/s^2$ |
| 角加速度 | $\alpha$ | $rad/T^2$ | $rad/s^2$ | $rad/s^2$ |
| 线速度 | $v$ | $L/T$ | ft/s | m/s |
| 角速度 | $w$ | $rad/T$ | rad/s | rad/s |
| 密度 | $\rho$ | $M/L^3$ | $Slug/ft^3$ | $kg/m^3$ |
| 力矩 | $M$[a] | $M(L^2/T^2)$ | ft-lb | N-m |
| 压力 | $P$ | $M/(LT^2)$ | $lb/ft^2$ | $N/m^2$ |
| 转动惯量 | $I$ | $ML^2$ | $lb\text{-}ft\text{-}s^2$ | $kg\text{-}m^2$ |
| 动黏度 | $u$ | $L^2/T$ | $ft^2/s$ | $m^2/s$ |
| 黏度 | $\mu$[a] | $M/(LT)$ | $lb\ s/ft^2$ | $N\ s/m^2$ |

a. 一般来说，大写的 $M$ 用来表示作用在物体上的力矩，而小写的 $m$ 用来表示物体的质量。在大部分状况下如果要表示质量的因次，也就是说要表示度量单位的因次部分，使用大写的 $M$ 来表示。这些所使用的符号，通常都以简单易懂为原则。然而在容易发生混淆的状况下，会再特别说明。黏度的符号是希腊字母 $\mu$ 的大写而不是全形的大写 $M$。rad 为 radian（弧度）的简写。

# 坐标系

本书中依据笛卡尔右手坐标系，来指明 2D 或 3D 空间中的位置。2D 空间的坐标系如图 1-1a 所示，其中以逆时针方向旋转为正方向。

3D 空间的坐标系如图 1-1b 所示，其中 $x$ 轴的旋转方向以 $y$ 轴正向到 $z$ 轴正向为正，$y$ 轴的旋转方向以 $z$ 轴正向到 $x$ 轴正向为正，$z$ 轴的旋转方向以 $x$ 轴正向到 $y$ 轴正向为正。

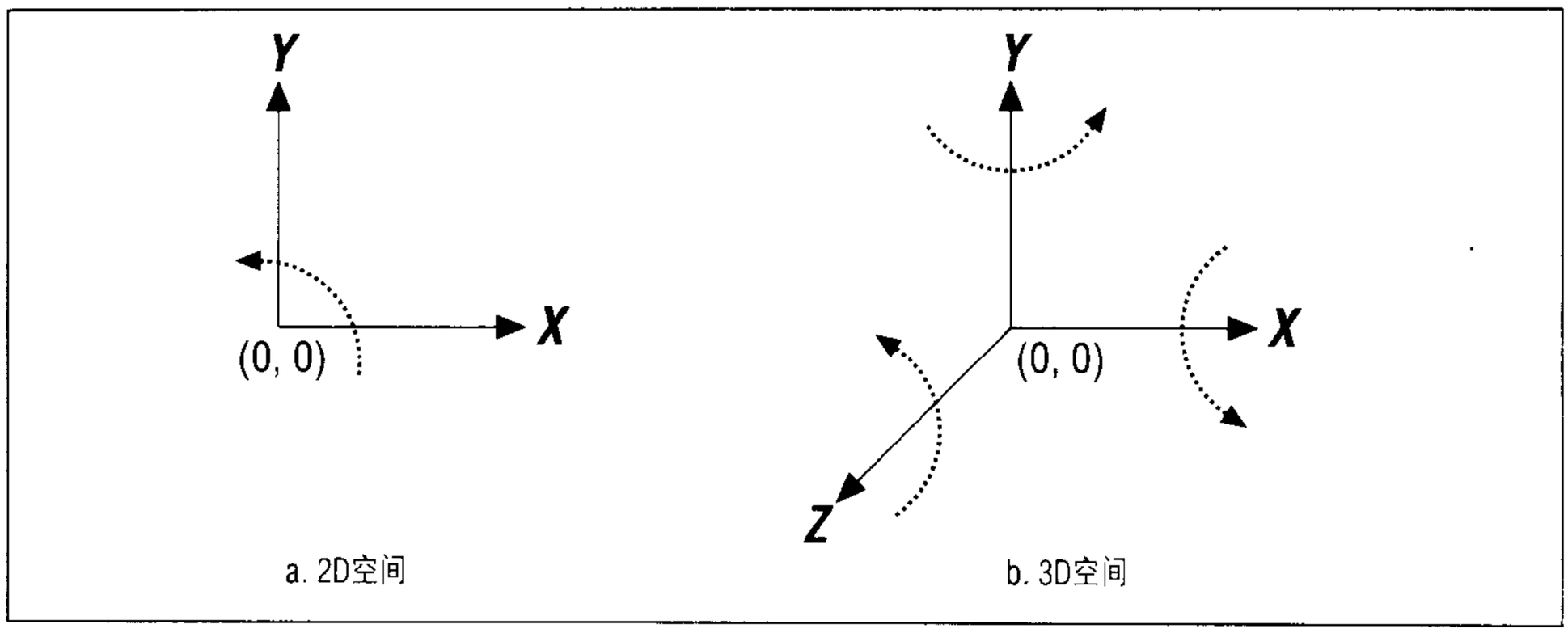



图 1-1：右手坐标系

# 向量

接下来，回到高中的数学课程并复习一些有关向量的概念。基本上，向量就是具备量值与方向的量；而标量就是只有量值但没有方向的量。在力学中，力、速度、加速度及动量就是向量，都必须同时考虑其量值与方向。而距离、密度跟黏度这些量，就是标量。

在标记时，会使用粗斜体字来表示向量。例如，力为 $\boldsymbol{F}$。当只考虑向量中的量值时，会用斜体字来表示。例如，力 $\boldsymbol{F}$ 这个向量的量值为 $F$，其在各个坐标轴上分力的大小则为 $F_x$、$F_y$，及 $F_z$。在这本书中的范例程序部分，根据前后文的关系，我会用符号“·”来表示向量内积（dot product）或是标量乘积（scalar product）的运算，并用符号“×”来表示向量的外积（cross product）运算。

因为这本书中会使用到很多向量，所以建议你最好先复习一下向量的基本运算，比如加法、内积、外积等。为了让读者能更快地进入状况，本书的附录一整理一些基本的向量运算，同时提供一个向量类的程序，里面已有所有重要的向量运算功能。此外，将解释如何利用特定的向量运算（如内积跟外积）来执行常用的计算。例如在动力学中，常常需要找出垂直（正交）于一个平面的向量，这时可利用外积运算来完成。另一个较常使用到的运算是找出空间中的一个点到一个平面的最短距离，这时就可以用内积运算来计算。这些例子在附录一中都有描述，在研究书中的范例程序之前，最好先复习一下这部分内容。

# 质量、质心与转动惯量

物体的质量特性，包括质量、质心、转动惯量（moment of inertia）等，在研究动力学

时是相当重要的。因为不论是物体的线性运动或角运动（注1），还是物体在受力之后的运动，都是这些质量特性的函数。因此要精确地模拟运动中的物体，必须先知道或求出这些质量特性。下面先看一些定义。

一般而言，质量大都被视为物体中所含物质的总量。但是就力学的观点来看，也可以把质量视为物体抵抗运动的阻力或改变其运动状态的力。因此，物体的质量越大，就表示越难使它运动或改变其运动状态。

以非专业的说法，物体的质心（或重心）即物体质量以其为中心均匀分布的点。以力学上的说法，质心是任何经过它作用在物体上的力都不会使物体旋转的点。

许多人虽然对于质心或重心这两个名词已经很熟悉，但是却不了解什么是转动惯量。在力学上，转动惯量也是非常重要的。物体的转动惯量是物体绕轴旋转，其质量呈辐射分布的度量。正如将物体的质量视为抵抗线性运动的物理量一样，也可以将转动惯量视为物体抵抗转动的物理量。

现在你已了解了这些特性的意义，接下来就来看如何计算这些值。

对某个由许多粒子所组成的物体而言，该物体的总质量就是所有组成粒子的质量的总和。每个粒子的质量为其质量密度与体积的乘积。假设该物体密度分布均匀，则该物体的总质量就是密度与总体积的乘积。可以表示为以下的方程：

$$m = \int \rho \mathrm{d}V = \rho \int \mathrm{d}V$$

实际上，很少需要对体积做积分来求出物体的质量，尤其是本书将要模拟的物体，例如汽车与飞机等，它们的密度都不是均匀分布的。可以将这些复杂的物体分解成已知（或容易求得）质量的零件，再将这些零件的质量加和后求得物体的总质量。

计算物体的重心也有一点复杂。首先要将该物体分解成无限多的质量元素，而每个元素的质心都是相对于参考坐标系的坐标轴指定的。接下来，计算质量元素对各参考坐标轴的一次矩（first moment），再将结果加合。一次矩就是质量与质心（沿给定坐标轴）到原点距离的乘积。最后再将这些一次矩的总和，除以该物体的总质量，就可以得到此物体在坐标系中相对于各参考坐标轴的质心坐标。每个坐标轴都必须做这样的计算，也就是说在2D空间需计算两次，在3D空间需计算三次。以下是在3D坐标系统中，计算某个物体质心的方程：

---

注1：线性运动指的是在空间中无关旋转的运动；角运动特别指的是物体绕着某个轴旋转的运动（物体也可能同时有线性运动）。

$$x_c = \left\{\int x_o dm\right\} \Big/ m$$

$$y_c = \left\{\int y_o dm\right\} \Big/ m$$

$$z_c = \left\{\int z_o dm\right\} \Big/ m$$

式中，$(x, y, z)_c$ 是该物体的质心坐标，而 $(x, y, z)_o$ 是各个质量元素的质心坐标。这些量值 $x_o dm$，$y_o dm$，$z_o dm$，则表示质量元素 $dm$ 对各个坐标轴的一次矩差。

再强调一次，不用担心这些方程中的积分问题。实际上只需将有限的物质加起来。这些公式可以表示成以下这个易懂的形式：

$$x_c = \left\{\sum x_o m_i\right\} \Big/ \left\{\sum m_i\right\}$$

$$y_c = \left\{\sum y_o m_i\right\} \Big/ \left\{\sum m_i\right\}$$

$$z_c = \left\{\sum z_o m_i\right\} \Big/ \left\{\sum m_i\right\}$$

在这些公式中，可以轻易地将质量元素的重量代入，由于分子跟分母的重力加速度都是常数因此可以消掉。回想一下，物体的重量是质量与重力加速度的乘积。在海平面上重力加速度的大小为 32.174 ft/s$^2$ (9.8 m/s$^2$)。

以下向量公式可用来计算离散质点系统的总质量与重心：

$$m_t = \sum m_i$$

$$\boldsymbol{CG} = \left[\sum (\boldsymbol{cg}_i)(m_i)\right] \Big/ m_t$$

式中，$m_t$ 是总质量，$m_i$ 是系统里每个质点的质量，$\boldsymbol{CG}$ 是合并后的重心，而 $\boldsymbol{cg}_i$ 是每个质量点以设计坐标（或参考坐标）计算的重心位置。请注意 $\boldsymbol{CG}$ 和 $\boldsymbol{cg}_i$ 都是向量，因为它们代表笛卡尔系统上的坐标。这是为了方便性而设计的，因为只需计算一次就可以处理 $x$，$y$，$z$ 分量（在 2D 空间中只需处理 $x$，$y$ 分量）。

在以下的范例程序代码中，假设构成物体的质点由结构（structure）阵列表示，每个结构中包含质点的设计坐标与质量。这个结构中的一个变量存储此质点相对于刚体重心的坐标。

```
Typedef struct _PointMass
{
    float mass;
    Vector designPosition;
    Vector correctedPosition;
```

```
} PointMass;

// 假设 _NUMELEMENTS 已经被定义
PointMass Elements[_NUMELEMENTS];
```

下面的程序代码说明如何计算这些质点的总质量及其合并后的重心：

```
int      i;
float     TotalMass;
Vector  CombinedCG;
Vector  FirstMoment;

TotalMass = 0;
for(i=0; i <_NUMELEMENTS; i++)
    TotalMass+= Elements[i].mass;

FirstMoment = Vector(0, 0, 0);
for(i=0; i<_NUMELEMENTS; i++)
{
    FirstMoment += Element[i].mass * Element[i].designPosition;
}
CombinedCG = FirstMoment / TotalMass;
```

现在已经算出合并后的重心位置，下面就可以求出每个质点的相对位置：

```
for(i=0; i<_NUMELEMENTS; i++)
{
    Element[i].correctedPosition = Element[i].designPosition - CombinedCG;
}
```

在计算转动惯量之前，要先取每个组成物体的质量元素对每个坐标轴的二次矩（second moment）。这里的杠杆（lever）并不是每个质量元素的形心（centroid，几何中心）沿某个坐标轴到原点的距离（像在计算质心时一样），而是质量元素的形心到坐标轴（欲求转动惯量的轴）的垂直距离。二次矩是质量元素的质量与此垂直距离平方的乘积。

参看图 1-2 中 3D 空间中任意的物体，当计算对 $x$ 轴的转动惯量 $I_{xx}$ 时，距离 $r$ 位于 $yz$ 平面上，而 $r_x^2 = y^2 + z^2$；同理，对于 $y$ 轴的转动惯量 $I_{yy}$，$r_y^2 = z^2 + x^2$；对于 $z$ 轴的转动惯量 $I_{zz}$，$r_z^2 = x^2 + y^2$。

在 3D 空间中，计算对某个坐标轴的转动惯量公式如下所示：

$$I_{xx} = \int r_x^2 \mathrm{d}m = \int (y^2 + z^2)\mathrm{d}m$$

$$I_{yy} = \int r_y^2 \mathrm{d}m = \int (z^2 + x^2)\mathrm{d}m$$

$$I_{zz} = \int r_z^2 \mathrm{d}m = \int (x^2 + y^2)\mathrm{d}m$$

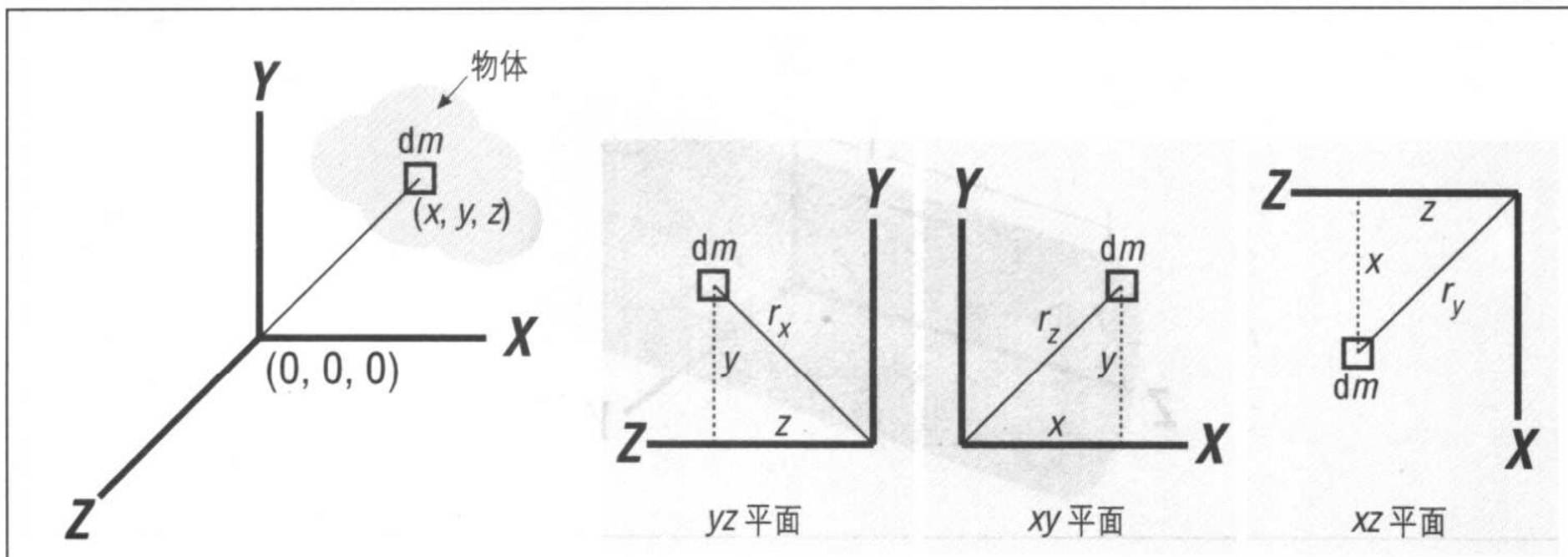



图 1-2：3D 空间中的任意物体

让我们花一点时间看看在练习中发生的状况。假设已知物体对中心轴（此轴通过物体的质心）的转动惯量 $I_o$，要计算物体对平行中心轴（但不重叠）的另一个轴的转动惯量 $I$。这个时候，可以使用坐标轴转换的方法（或称为平行轴定理），计算对新轴的转动惯量。以下是这个公式：

$$I = I_o + md^2$$

公式中的 $m$ 是物体的质量，而 $d$ 是平行的两个轴之间的垂直距离。

从这里可以观察出一些重要的现象：新的转动惯量是两轴之间距离平方的函数。这也就表示当 $I_o$ 相对很小而 $d$ 相对很大时，可以完全地忽略 $I_o$，因为 $md^2$ 的影响大于 $I_o$，这里当然要靠自己判断。坐标轴转换的公式同时也指出，当对某个通过物体质心的轴计算转动惯量时，所得到的转动惯量是最小值。当对某个不通过物体质心的轴计算平行轴的转动惯量时，会增加 $md^2$。

实际上，即使是计算形状简单而且密度均匀物体的转动惯量也是件复杂的工作。所以在计算通过物体质心轴的转动惯量时，通常使用对近似物体的基本形状的简单公式来估计。另外，可以将复杂的形状分解成较简单的形状，而且某些部分的 $I_o$ 小到可以被忽略，只需考虑其 $md^2$ 对整个物体的转动惯量的影响。

图 1-3 到图 1-7 显示一些容易计算转动惯量的简单立体几何图形。图中的标题中给出了这些密度均匀的立体几何图形各坐标轴的转动惯量公式。其他类似的基本几何图形的公式，可在高等院校的动力学教材中找到（在本书后面的参考文献中可以找到一些相关的资料）。

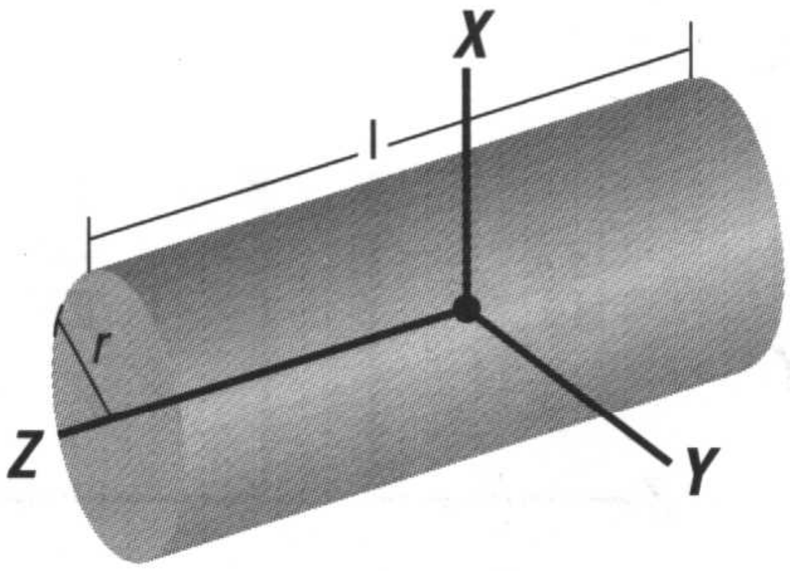



图 1-3：圆柱体；$I_{xx} = I_{yy} = (1/4)mr^2 + (1/12)ml^2$；$I_{zz} = (1/2)mr^2$

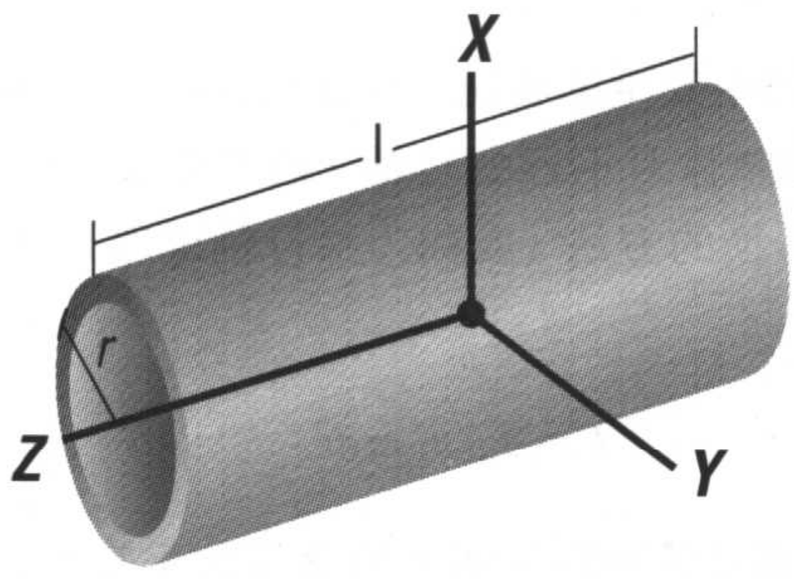



图 1-4：空心圆柱体；$I_{xx} = I_{yy} = (1/2)mr^2 + (1/12)ml^2$；$I_{zz} = mr^2$

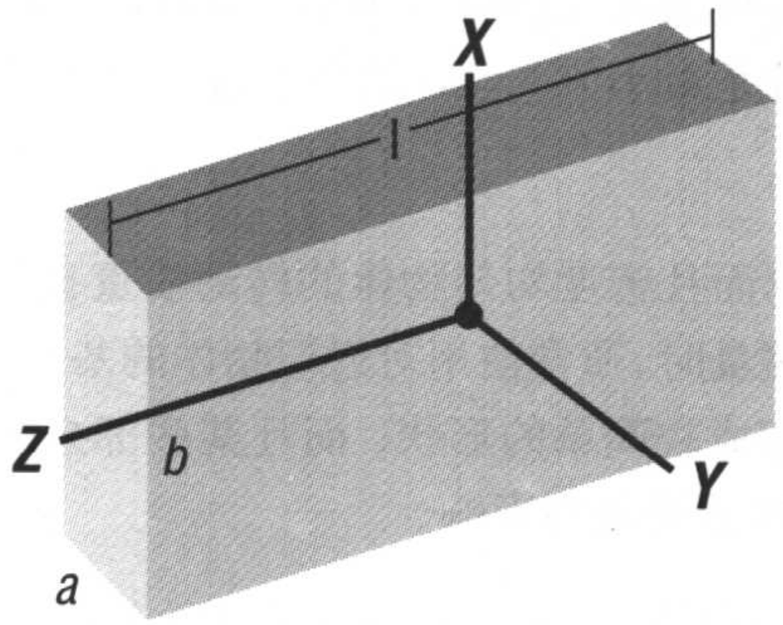



图 1-5：长方体；$I_{xx} = (1/12)m(a^2 + l^2)$；$I_{yy} = (1/12)m(b^2 + l^2)$；$I_{zz} = (1/12)m(a^2 + b^2)$

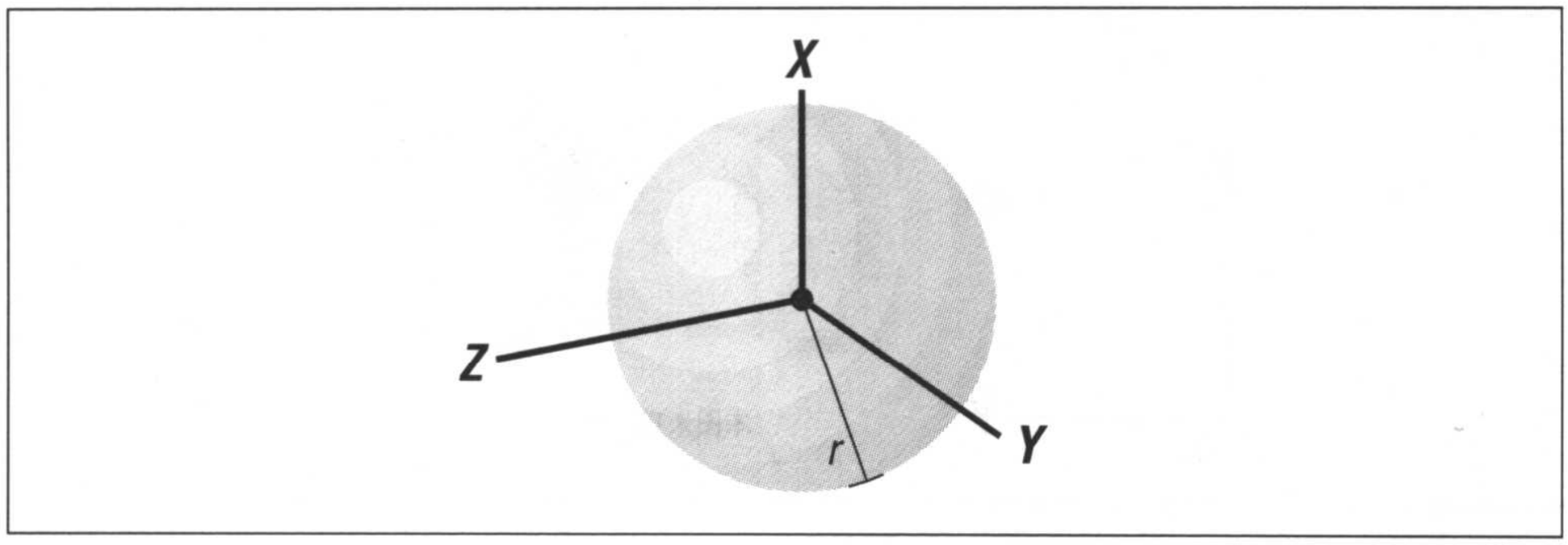



图 1-6：球体；$I_{xx} = I_{yy} = I_{zz} = (2/5)mr^2$

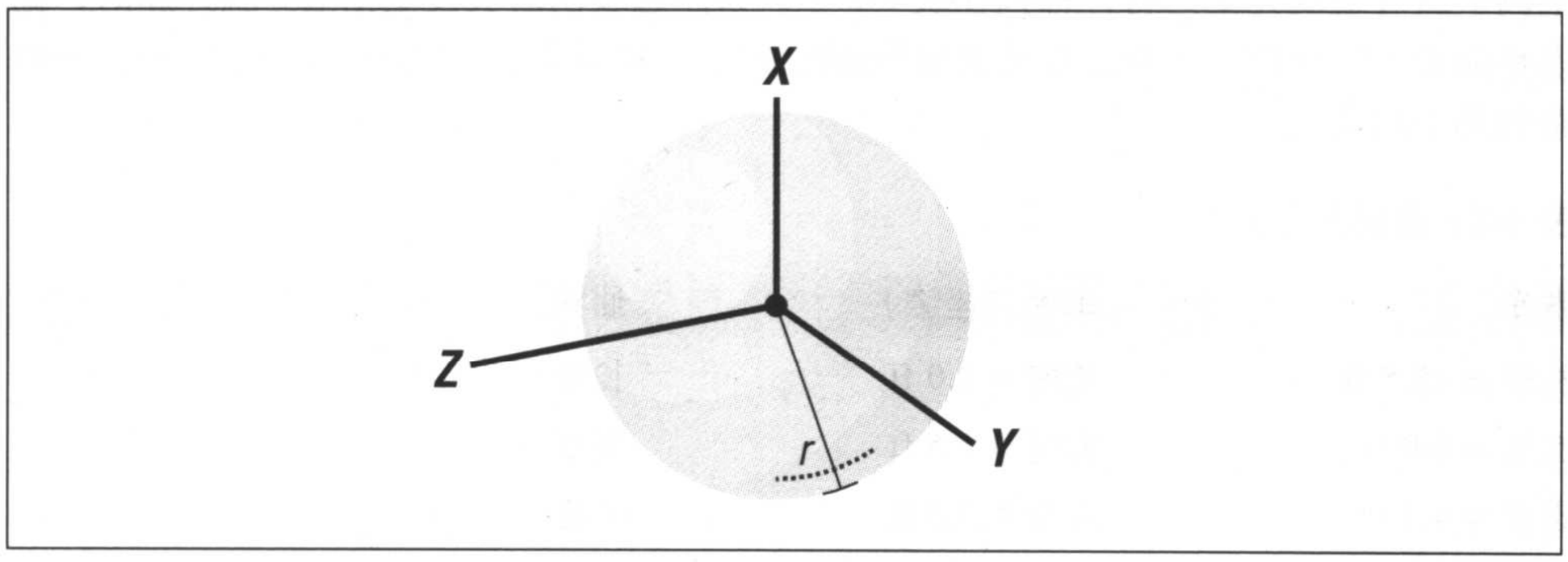



图 1-7：空心球体；$I_{xx} = I_{yy} = I_{zz} = (2/3)mr^2$

你可以发现，这些图形的公式比较容易推导。技巧是将复杂的物体分解成较小、较简单的典型几何形状，合并之后可以用来估计复杂物体的惯性特性。这个练习主要考虑精确度上的判断问题。

我们来看一个简单的 2D 范例，介绍如何使用本节介绍的方程。假设你正在做一个视角是由上往下的赛车游戏，模拟游戏中的赛车是根据 2D 刚体动力学所设定的。在游戏开始时，玩家的赛车停在起跑线上，装满着汽油正准备起跑。在开始模拟前，必须先计算赛车、车手和油料的质量特性。因此在这个状况下，整个物体是由三个元素所组成的：赛车、车手和满载的油料。然而之后在游戏进行时，质量特性会由于油料的损耗或车手因为撞击被弹出车外而改变。目前，我们先把焦点放在初始条件上，如图 1-8 所示。

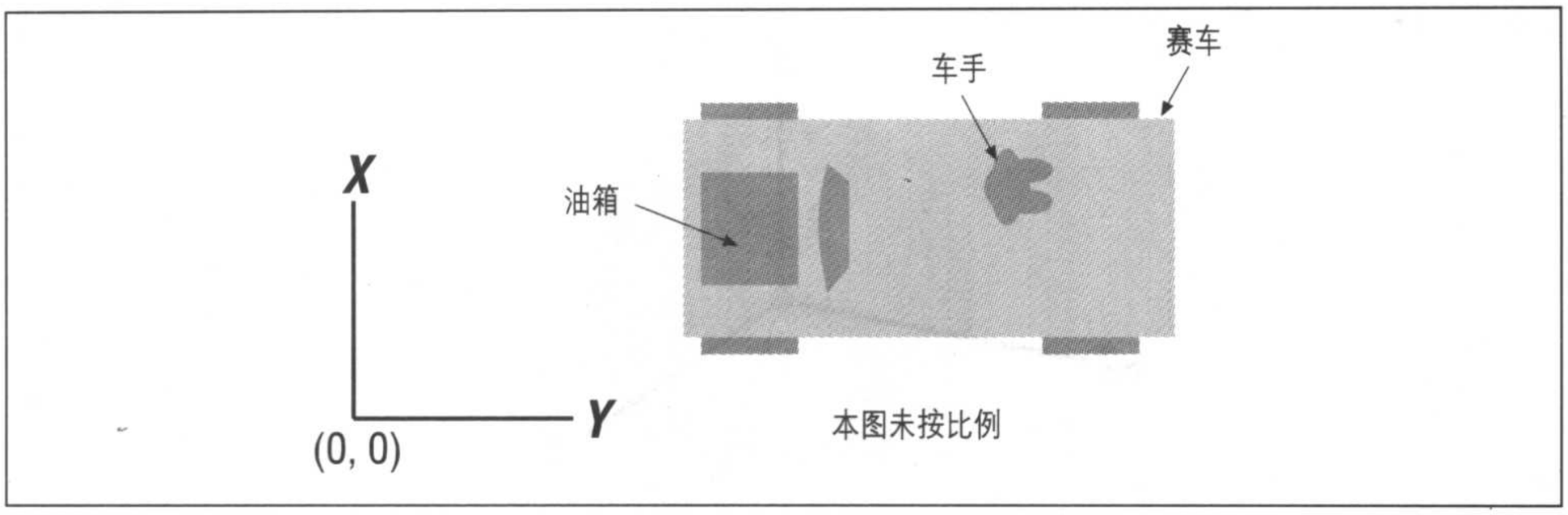



图 1-8：范例中的物体，包含赛车、车手和油料

范例中所有组成元素的特性都列在表 1-2 中。请注意表内的长度是沿着 $x$ 轴计算的，宽度是沿着 $y$ 轴计算的，而高度是突出纸面的距离，表示质心坐标的格式是$(x, y)$，所参考的是全局原点。

表 1-2：范例的特性

| 赛车 | 车手（坐姿） | 油料 |
|---|---|---|
| 长度 = 15.5 ft | 长度 = 3.0 ft | 长度 = 1.5 ft |
| 宽度 = 6.0 ft | 宽度 = 1.5 ft | 宽度 = 3.0 ft |
| 高度 = 4.1 ft | 高度 = 3.5 ft | 高度 = 1.0 ft |
| 重量 = 3913.0 lb | 重量 = 190.0 lb | 油料密度 = 1.45 slug/ft$^3$ |
| 质心 = (100 100) ft | 质心 = (103 105) ft | 质心 = (93 100) ft |

首先要计算的质量特性是物体的质量。这个计算很简单，因为我们已知车子和车手的重量。所以惟一要计算的是油料的重量。同时也知道了油料的密度和油箱的长宽高，因此可以将油箱的体积乘以油料的密度，再乘以重力加速度就得到油箱中油料的重量。根据这里的数据计算出的结果是 210 lb：

$$W_{\text{fuel}} = \rho v g = (1.45 \text{ slug/ft}^3)(1.5 \text{ ft})(3 \text{ ft})(1 \text{ ft})(32.174 \text{ ft/s}^2) = 210 \text{ lb}$$

现在物体的总重量是：

$$W_{\text{total}} = W_{\text{car}} + W_{\text{driver}} + W_{\text{fuel}}$$

$$W_{\text{total}} = 3913 \text{ lb} + 1901 \text{ lb} + 210 \text{ lb} = 4317 \text{ lb}$$

要求得物体的质量，只要将重量除以重力加速度即可：

$$M_{\text{total}} = W_{\text{total}}/g = 4317 \text{ lb}/(32.174 \text{ ft/s}^2) = 134.2 \text{ slugs}$$

斯勒格（slug）是个听起来很奇怪的单位，而使用起来也很奇怪。所以最好将质量换成国际单位，即 1958.2 kg，接近两吨重。

接着要计算的质量特性是物体的重心位置。在这个范例中，要先找出相对于全局原点的质心坐标，并且套用一次矩公式两次，分别求出 $x$ 轴坐标与 $y$ 轴坐标。计算过程如下：

$$X_{\text{cg body}} = \{(x_{\text{cg car}})(W_{\text{car}}) + (x_{\text{cg driver}})(W_{\text{driver}}) + (x_{\text{cg fuel}})(W_{\text{fuel}})\}/W_{\text{total}}$$

$$X_{\text{cg body}} = \{(100\ \text{ft})(3913\ \text{lb}) + (103\ \text{ft})(190\ \text{lb}) + (93\ \text{ft})(210\ \text{lb})\}/\ 4317\ \text{lb}$$

$$X_{\text{cg body}} = 99.7\ \text{ft}$$

$$Y_{\text{cg body}} = \{(y_{\text{cg car}})(W_{\text{car}}) + (y_{\text{cg driver}})(W_{\text{driver}}) + (y_{\text{cg fuel}})(W_{\text{fuel}})\}/W_{\text{total}}$$

$$Y_{\text{cg body}} = \{(100\ \text{ft})(3913\ \text{lb}) + (105\ \text{ft})(190\ \text{lb}) + (100\ \text{ft})(210\ \text{lb})\}/\ 4317\ \text{lb}$$

$$Y_{\text{cg body}} = 100.1\ \text{ft}$$

请注意在这些等式中使用重量代替质量。这么做的原因是重量中的重力加速度是个常数，并且同时在公式的分子和分母中出现，所以可以进行约分。

接着要计算物体的转动惯量。对于2D的范例这就很简单了，因为只会有一个旋转轴（指出纸面的轴），所以只需计算一次。第一步是计算每个元素对自身中心轴的转动惯量。由于每个元素的几何形状和质量分布的信息有限，因此这里简单地假设每个元素都是长方体，而使用同一个转动惯量公式。为了避免混淆，在等式中，小写 $w$ 表示的是物体的宽度，大写 $W$ 表示的是物体的重量。

$$I_{\text{o car}} = (m/12)(w^2 + L^2)$$

$$I_{\text{o car}} = ((3913\ \text{lb}/32.174\ \text{ft/s}^2)/12)((6.0\ \text{ft})^2 + (15.5\ \text{ft})^2) = 2800\ \text{lb-s}^2\text{-ft}$$

$$I_{\text{o driver}} = (m/12)(w^2 + L^2)$$

$$I_{\text{o driver}} = ((190\ \text{lb}/32.174\ \text{ft/s}^2)/12)((1.5\ \text{ft})^2 + (3.0\ \text{ft})^2) = 5.5\ \text{lb-s}^2\text{-ft}$$

$$I_{\text{o fuel}} = (m/12)(w^2 + L^2)$$

$$I_{\text{o fuel}} = ((210\ \text{lb}/32.174\ \text{ft/s}^2)/12)((3.0\ \text{ft})^2 + (1.5\ \text{ft})^2) = 6.1\ \text{lb-s}^2\text{-ft}$$

因为之前计算出的是每个元素对自身中心轴的转动惯量，所以现在要使用平行轴定理将转动惯量转移到整体的中心轴上（位于先前求得的物体重心）。要使用平行轴定理，必须先知道每个元素的重心到整体重心的距离。每个元素距离的平方是这么计算的：

$$d_{\text{car}}^2 = (x_{\text{cg car}} - X_{\text{cg}})^2 + (y_{\text{cg car}} - Y_{\text{cg}})^2$$

$$d_{\text{car}}^2 = (100\ \text{ft} - 99.7\ \text{ft})^2 + (100\ \text{ft} - 100.1\ \text{ft})^2 = 0.1\ \text{ft}^2$$

$$d_{\text{driver}}^2 = (x_{\text{cg driver}} - X_{\text{cg}})^2 + (y_{\text{cg driver}} - Y_{\text{cg}})^2$$

$$d_{\text{driver}}^2 = (103\ \text{ft} - 99.7\ \text{ft})^2 + (105\ \text{ft} - 100.1\ \text{ft})^2 = 34.9\ \text{ft}^2$$

$$d_{\text{fuel}}^2 = (x_{\text{cg fuel}} - X_{\text{cg}})^2 + (y_{\text{cg fuel}} - Y_{\text{cg}})^2$$

$$d_{\text{fuel}}^2 = (93\text{ ft} - 99.7\text{ ft})^2 + (100\text{ ft} - 100.1\text{ ft})^2 = 44.9\text{ ft}^2$$

现在就可以套用平行轴原理如下：

$$I_{\text{cg car}} = I_{\text{o}} + md^2$$

$$I_{\text{cg car}} = 2800\text{ lb-s}^2\text{-ft} + (3913\text{ lb}/32.174\text{ ft/s}^2)(0.1\text{ ft}^2) = 2812\text{ lb-s}^2\text{-ft}$$

$$I_{\text{cg driver}} = I_{\text{o}} + md^2$$

$$I_{\text{cg driver}} = 5.5\text{ lb-s}^2\text{-ft} + (190\text{ lb}/32.174\text{ ft/s}^2)(34.9\text{ ft}^2) = 211.6\text{ lb-s}^2\text{-ft}$$

$$I_{\text{cg fuel}} = I_{\text{o}} + md^2$$

$$I_{\text{cg fuel}} = 6.1\text{ lb-s}^2\text{-ft} + (210\text{ lb}/32.174\text{ ft/s}^2)(44.9\text{ ft}^2) = 299.2\text{ lb-s}^2\text{-ft}$$

计算后可以发现，在车手和油料的$I_{\text{cg}}$中，$md^2$占较大的部分。在本范例中，车手和油料自身的转动惯量$I_{\text{o}}$分别只有$md^2$的2.7%和2.1%。

最后只要将每个元素的$I_{\text{cg}}$加起来，就可以得到物体对其中心轴的总转动惯量：

$$I_{\text{cg total}} = I_{\text{cg car}} + I_{\text{cg driver}} + I_{\text{cg fuel}}$$

$$I_{\text{cg total}} = 2812\text{ lb-s}^2\text{-ft} + 211.6\text{ lb-s}^2\text{-ft} + 299.2\text{ lb-s}^2\text{-ft} = 3322.8\text{ lb-s}^2\text{-ft}$$

上面所求出的物体（汽车、车手，及油箱所组成）的质量特性如表1-3所示。

表1-3：范例质量特性一览

| 特性 | 求出的值 |
|---|---|
| 总质量（重量） | 134.2 slug（4317 lb） |
| 合并后的质心位置 | $(x, y) = (99.7\text{ ft}, 100.1\text{ ft})$ |
| 转动惯量 | 3322.8 lb-s$^2$-ft |

有一点非常重要：你需要彻底了解本范例说明的基本概念。因为将来要介绍更复杂的系统尤其是3D运动，其中的计算会越来越复杂。而且，要模拟的物体运动是这些质量特性的函数，这些特性中，质量决定物体被作用力影响的程度，质心用来追踪物体的位置，而转动惯量决定物体在不通过重心的力的作用下如何转动。

到目前为止，已经介绍过在3D空间中绕着三个坐标轴的转动惯量。然而在一般的3D刚体动力学中，即使物体的区域坐标轴通过质心，物体也可能会绕着任意轴，而未必是三个坐标轴之一旋转。这种复杂性表示必须在转动惯量$I$的公式中加入新的项目，才能处

理这类一般化的旋转。本章的最后一节将讨论这个主题，不过在这之前，要先详细介绍牛顿第二运动定律。

# 牛顿第二运动定律

如同本章第一节中所述，牛顿第二运动定律是学习力学时的一项特别重要的主题。牛顿第二运动定律的公式如下：

$$F = ma$$

式中，$F$ 是作用在物体上的合力，$m$ 是物体的质量，而 $a$ 是物体重心的线性加速度。

如果重新排列公式，变成：

$$F/m = a$$

由上面的公式可以了解物体的质量如何抵抗运动。假设物体所受的力固定不变，当位于分母的质量增加时，所产生的加速度会减小。因此可以说物体的质量越大则在运动时所受的阻力也越大。同样，在固定的作用力下，当物体的质量减小时，其加速度便会增加。因此可以说物体的质量越小，在运动时所受的阻力也会越小。

牛顿第二运动定律同时也说明物体加速度的方向与受力的方向一致。因此，力和加速度必须被视为向量。在特定时间下，通常会有一个以上的力作用于物体上。也就是说，最后的合力 $\boldsymbol{F}$ 是所有作用在该物体上的力的向量总和。而公式可以写成：

$$\sum \boldsymbol{F} = m\boldsymbol{a}$$

式中，$\boldsymbol{a}$ 是代表加速度的向量。

在3D空间中，力与加速度的向量在笛卡尔系统中会有 $x$，$y$，$z$ 三个分量。这种情况下，运动的分量公式可写成：

$$\sum F_x = ma_x$$
$$\sum F_y = ma_y$$
$$\sum F_z = ma_z$$

另一种对牛顿第二运动定律的解释是，所有作用在物体上的力的总和，等于该物体在时间上的动量变化率，也就是动量对时间的导数（derivative）。动量等于质量乘以速度，而且因为速度是向量，所以动量也是向量。因此：

$$G = mv$$

式中，$G$ 是物体的线性动量，$m$ 是物体的质量，且 $v$ 是物体重心的速度。动量对时间的变动率就是动量对于时间的导数：

$$\mathrm{d}G/\mathrm{d}t = \mathrm{d}/\mathrm{d}t(mv)$$

假设物体的质量是常数，等式能改写成：

$$\mathrm{d}G/\mathrm{d}t = m\mathrm{d}v/\mathrm{d}t$$

速度对于时间的变动率 $\mathrm{d}v/\mathrm{d}t$ 即为加速度，可导出：

$$\mathrm{d}G/\mathrm{d}t = ma$$

而

$$\sum F = \mathrm{d}G/\mathrm{d}t = ma$$

到目前为止，我们只考虑物体的移动，而没有考虑转动。在一般的 3D 运动中，必须考虑到物体的转动。因此也需要一些额外的公式来描述该物体的运动。考虑转动时，便需要用到与前面所述相类似的公式 —— 作用于物体上的力矩（转矩）总和与角动量对时间的变化率（或角动量对于时间的导数）的等式。即：

$$\sum M_{\mathrm{cg}} = \mathrm{d}/\mathrm{d}t(H_{\mathrm{cg}})$$

式中，$\Sigma M_{\mathrm{cg}}$ 是对于该物体重心的力矩的总和，而 $H$ 是该物体的角动量，$M_{\mathrm{cg}}$ 可以表示成：

$$M_{\mathrm{cg}} = r \times F$$

式中，$F$ 是作用在物体上的力，$r$ 是从 $F$ 到物体重心的距离向量（垂直于 $F$ 的作用线）。而 × 是向量的外积操作符。

物体的角动量是物体中所有粒子绕旋转轴的动量矩之和。在这里假设旋转轴会通过该物体的重心：

$$H_{\mathrm{cg}} = \sum r_i \times m_i(\omega \times r_i)$$

式中，$i$ 表示组成该物体的第 $i$ 个粒子，$\omega$ 是该物体在轴上的角速度。$(\omega \times r_i)$ 是第 $i$ 个粒子的角动量，其大小则为 $\omega r_i$。对于绕给定轴旋转的公式可以写成：

$$H_{\mathrm{cg}} = \int \omega r^2 \mathrm{d}m$$

如果组成该刚体的所有粒子的角速度都相同的话，可以得到：

$$H_{cg} = \omega \int r^2 dm$$

并且由于转动惯量 $I$ 等于 $\int r^2 dm$，因此可以得到

$$H_{cg} = I\omega$$

再取其相对于时间的导数，可得

$$dH_{cg}/dt = d/dt(I\omega) = I d\omega/dt = I\alpha$$

式中，$\alpha$ 是该物体对该轴的角加速度。

最后可写成：

$$\sum M_{cg} = I\alpha$$

如同在转动惯量中的讨论一样，我们必须进一步地将转动惯量与角动量的公式一般化，才能应用到绕任意轴的转动。通常 $M$ 和 $\alpha$ 是向量，而 $I$ 是张量（tensor）（注 2）。因为物体转动惯量的大小可能会因为旋转轴的不同而有变化。

在此我必须提出一些考虑到坐标轴的时候要注意的事项。这在写实际的模拟程序时是很重要的。到目前为止，运动方程都写成全局坐标而不是物体固定坐标的形式。这对追踪物体在全局坐标系中的位置与速度的运动线性方程而言是没问题的。然而从计算的观点来看，不会这样计算在 3D 的坐标空间中旋转的物体的角运动方程（注 3）。原因在于当以全局坐标来计算时，转动惯量的大小会因物体的位置与方位而改变。这表示在模拟的过程中，必须不断地重新计算惯性矩阵（与反矩阵）。这在计算上是很没有效率的。较好的方式是改写运动方程，使其中的项目为局部坐标。这样就只需在模拟开始时，计算一次所需的惯性矩阵（与反矩阵）即可。

一般而言，在固定（不旋转）坐标系中，向量 $V$ 对时间的导数与在转动坐标系中的时间导数的关系如下：

$$(dV/dt)_{fixed} = (dV/dt)_{rot} + (\omega \times V)$$

---

注 2：在这种状况下，$I$ 是二阶张量，基本上也就是 $3 \times 3$ 矩阵。向量是一阶张量，而标量是零阶张量。

注 3：在 2D 空间中，像这里一样不处理角运动方程是无所谓的，因为转动惯量项目只是常数标量。

## 张量

张量是具有大小与方向的数学式，不过其大小可能会因为方向的不同而改变。张量通常用来表示这些会因方向不同而有不同大小的物质特性。对于这种具有因方向而性质相异的物质，称它具有各向异性（anisotropic），而等方向性（isotropic）则表示各个方向的大小都相等。举例来说，考虑两个普通物质（分别是一张普通的纸和一块编织而成的布）的弹性（强度）。先将这张纸保持平放，轻轻地由相反的方向拉，并试着由纵向、横向和对角线的方向拉。可以由这个实验中观察到：这张纸几乎在所有的方向都有着相同的强度。这就是等方向性，所以只要用一个标量常数就能表示所有方向的强度。

现在找出一块简单而编织宽松的布，上面某个方向的纹路垂直于另一个方向的纹路。大部分的领带会采用这种织法。接着做如同对刚才那张纸的拉力测试，将这块布沿着每个纹路的方向和对角方向拉开。你会发现，当沿着对角方向拉开时，可以比沿着纹路方向拉长更多。布的各向异性是指依据拉力的方向，会存在不同的弹性（或强度）。于是，需要用向量（张量）的集合来表示它沿着各方向伸展的强度。

在本书内容所涵盖的范围内，需要列入考虑的是物体的转动惯量，在3D空间中需要用9个分量才能表示任意轴的转动。转动惯量并不是如纸张与布所具有的强度特性，而是随旋转轴变化的特性。因为需要用到9个分量，因此本书稍后几章会使用3 × 3矩阵（二秩张量）来表示转动惯量。

（$\omega \times V$）项表示$V$的时间导数在固定坐标系与转动坐标系间的差。可利用这个关系式来将此角运动方程改写成局部坐标系中的方程。另外，要考虑的向量是角动量$\boldsymbol{H}_{cg}$。已知$\boldsymbol{H}_{cg} = \boldsymbol{I}\omega$，而其时间导数即该物体重心所有的力矩总和。这些是在计算角运动方程时所必需的，并且可以在导数转换中将$\boldsymbol{V}$换成$\boldsymbol{H}_{cg}$，而得到以下公式：

$$\sum \boldsymbol{M}_{cg} = d\boldsymbol{H}_{cg} / dt = \boldsymbol{I}(d\omega / dt) + (\omega \times (\boldsymbol{I}\omega))$$

这里的力矩、惯性张量和角速度都是以局部坐标计算的。虽然这个公式比之前所介绍的公式还要复杂，但是用起来却很简单，因为$I$在模拟过程中将会是常数（除非在模拟过程中物体的质量或几何形状会改变），而且在局部坐标中计算力矩也比较简单。在第十五章介绍如何实现一个3D刚体模拟器时将会再次使用这个公式。

# 惯性张量

另外再看看角运动方程，请注意用粗斜体来表示转动惯量，表示它是向量。在2D空间中，因为转动惯量只有一个旋转轴，所以$I$只是标量。然而3D空间中，物体却能绕着三个坐标轴旋转。甚至，物体能绕着任意轴旋转。所以在3D中，$I$是3x3的矩阵，或称为二阶张量。

要了解惯性矩阵是如何推导出来的，就要再看看角动量方程：

$$\boldsymbol{H}_{\mathrm{cg}} = \int (\boldsymbol{r} \times (\omega \times \boldsymbol{r}))\mathrm{d}m$$

式中，$\omega$是物体的角速度，$\boldsymbol{r}$（请参考图1-9）是从物体的重心到每个质量元素d$m$之间的距离，而$\boldsymbol{r} \times (\omega \times \boldsymbol{r})$是每个质量元素的角动量。括号中表示的项称为向量三重积(triple product)，可取向量的外积将项目展开。$\boldsymbol{r}$和$\omega$可以写成以下的向量：

$$\boldsymbol{r} = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$
$$\omega = \omega_x\boldsymbol{i} + \omega_y\boldsymbol{j} + \omega_z\boldsymbol{k}$$

将三重积展开后可以得到：

$$\boldsymbol{H}_{\mathrm{cg}} = \int \{[(y^2 + z^2)\omega_x - xy\omega_y - xz\omega_z]\boldsymbol{i} + [-yx\omega_x + (z^2 + x^2)\omega_y - yz\omega_z]\boldsymbol{j} + [-zx\omega_x - zy\omega_y + (x^2 + y^2)\omega_z]\boldsymbol{k}\}\mathrm{d}m$$

要简化这个公式，可用以下各式取代某些项：

$$I_{xx} = \int (y^2 + z^2)\mathrm{d}m$$
$$I_{yy} = \int (z^2 + x^2)\mathrm{d}m$$
$$I_{zz} = \int (x^2 + y^2)\mathrm{d}m$$
$$I_{xy} = I_{yx} = \int (xy)\mathrm{d}m$$
$$I_{xz} = I_{zx} = \int (xz)\mathrm{d}m$$
$$I_{yz} = I_{zy} = \int (yz)\mathrm{d}m$$

用以上各式取代你可能很熟悉的$I$变量，回到展开的方程则产生以下等式：

$$\boldsymbol{H}_{\mathrm{cg}} = [I_{xx}\omega_x - I_{xy}\omega_y - I_{xz}\omega_z]\boldsymbol{i} + [-I_{yx}\omega_x + I_{yy}\omega_y - I_{yz}\omega_z]\boldsymbol{j} + [-I_{zx}\omega_x - I_{zy}\omega_y + I_{zz}\omega_z]\boldsymbol{k}$$

假设 $\boldsymbol{I}$ 是个矩阵还可以简化此步骤：

$$\boldsymbol{I} = \begin{matrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{matrix}$$

产生以下的等式：

$$\boldsymbol{H}_{\mathrm{cg}} = \boldsymbol{I}\omega$$

我们已经知道 $\boldsymbol{I}$ 代表转动惯量，三个熟悉的项 $I_{xx}$，$I_{yy}$，$I_{zz}$ 是对三个坐标轴的转动惯量。剩下的项称为惯性积（product of inertia）：

$$I_{xy} = I_{yx} = \int (xy)\mathrm{d}m$$

$$I_{xz} = I_{zx} = \int (xz)\mathrm{d}m$$

$$I_{yz} = I_{zy} = \int (yz)\mathrm{d}m$$

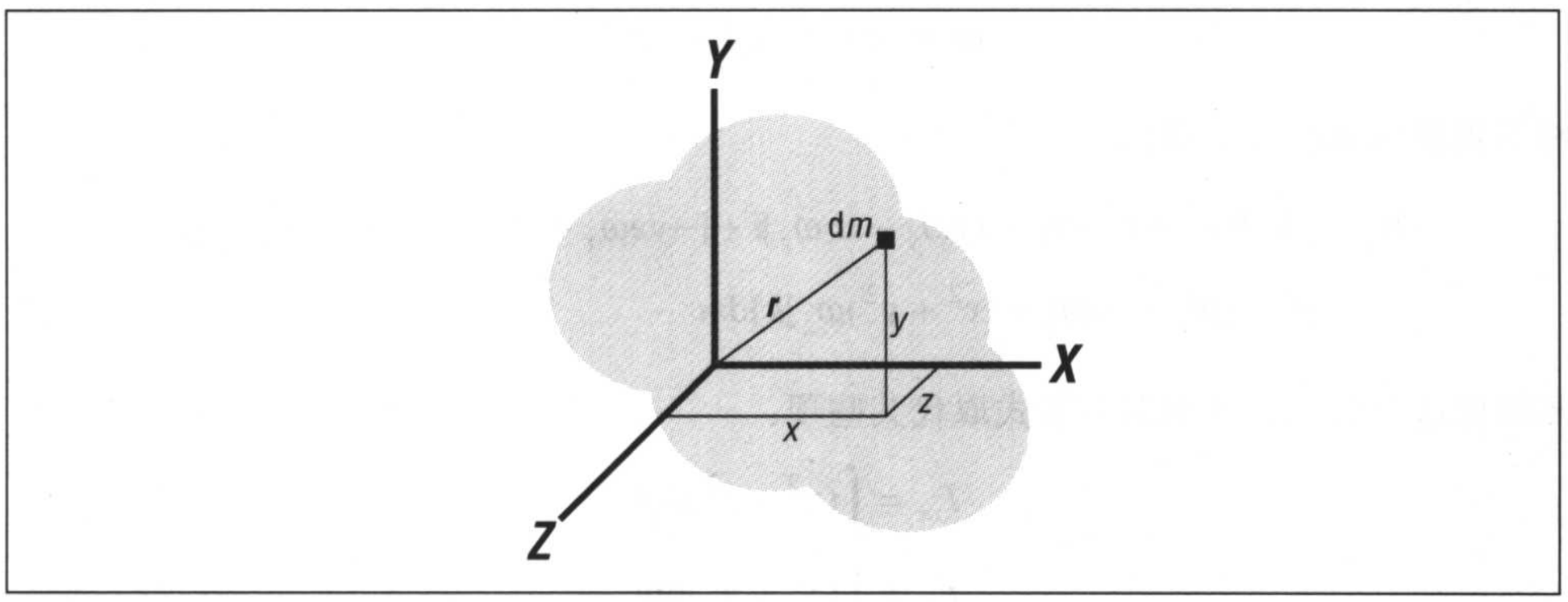



图 1-9：惯性积

惯性积也有类似平行轴定理的坐标轴转换公式：

$$I_{xy} = I_{\mathrm{o}(xy)} + md_xd_y$$

$$I_{xz} = I_{\mathrm{o}(xz)} + md_xd_z$$

$$I_{yz} = I_{\mathrm{o}(yz)} + md_yd_z$$

这里的 $I_o$ 表示本体的惯性积，也就是物体对通过自身重心的轴的惯性积，$m$ 是物体的质量，而 $d$ 是通过物体重心的坐标轴与某组平行坐标轴之间的距离（参考图 1-10）。

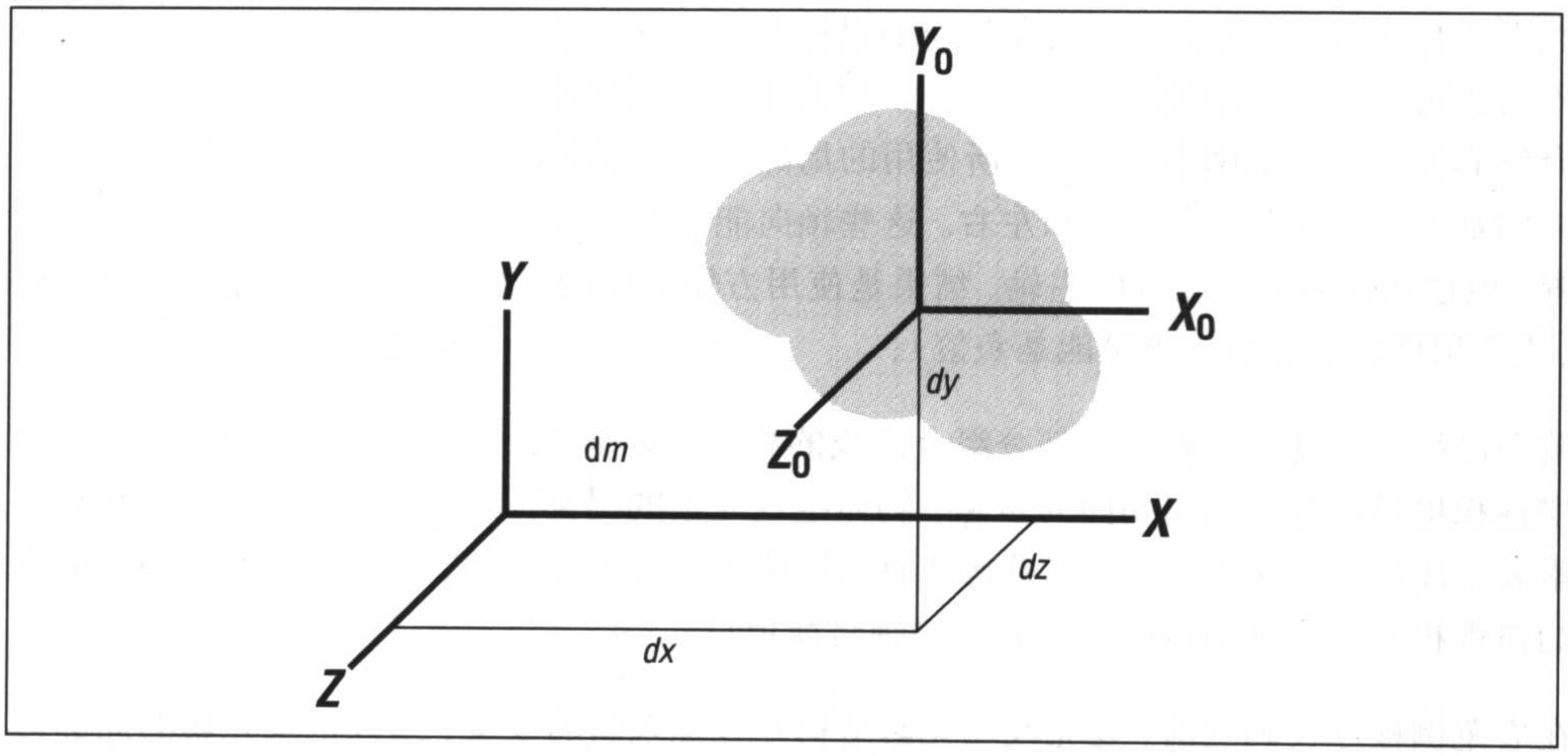



图 1-10：轴转换

你会注意到这里并没有任何简单图形的惯性积公式。因为转动惯量都是对这些图形的主轴来计算的。任何物体都有一组相对的坐标轴，会使惯性张量中的惯性积皆为零。

对于之前所介绍的简单几何图形，每个坐标轴都象征一个对称的平面，而对这些轴的惯性积都为零，这项结论可通过检查惯性积的公式得出。举例来说，如果如图 1-11 中所示的物体对称于 $y$ 轴，则所有积分中的 $(xy)$ 项都会被每个相应的 $-(xy)$ 项抵消掉。

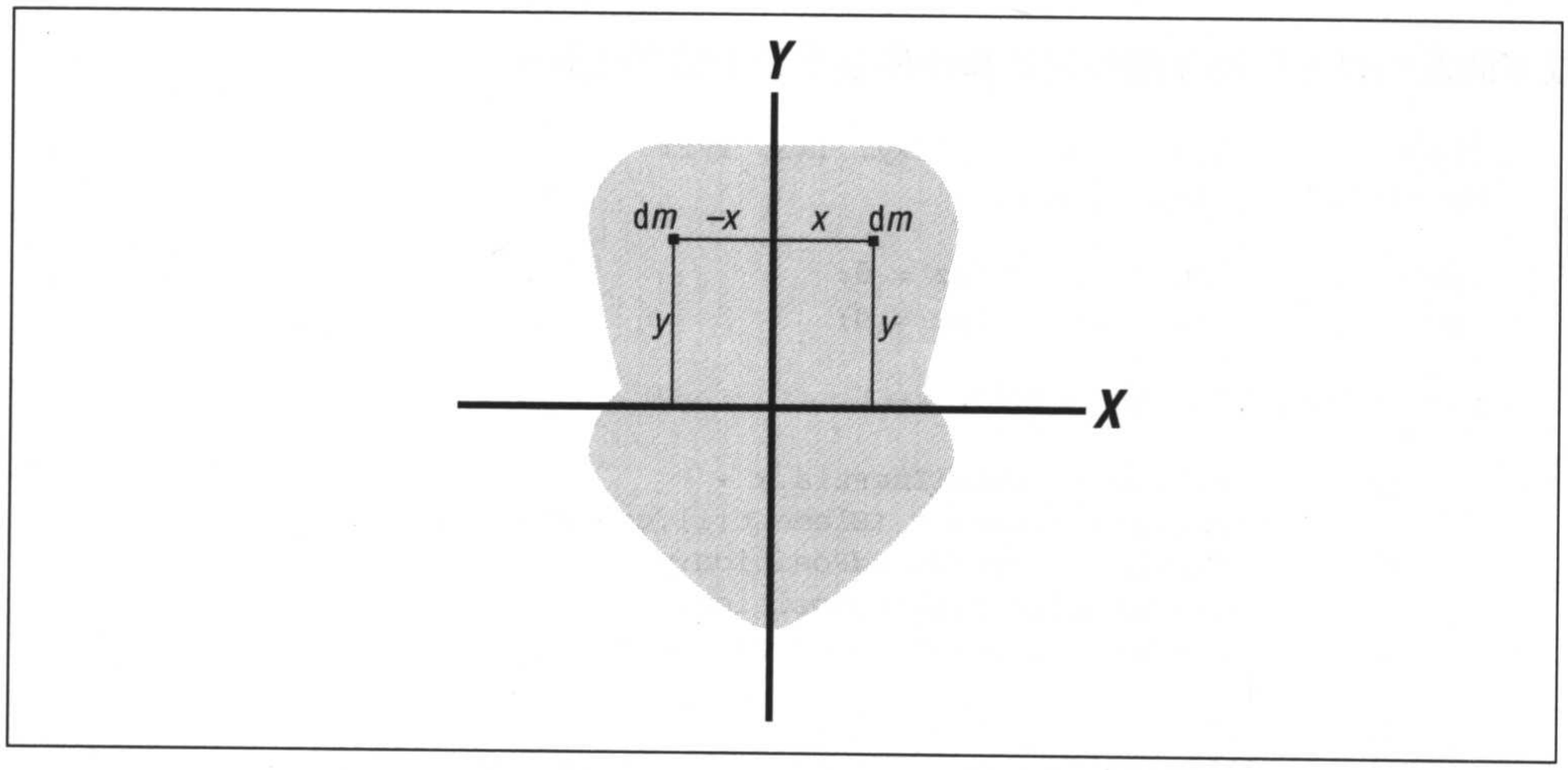



图 1-11：对称性

然而对合成物体而言，可能没有任何的对称平面。而且它的主轴方向并不明显。甚至你不会想用主轴当做刚体的局部坐标轴，因为这样会很难处理。举例来说，在第七章将会介绍的飞行模拟范例中的飞机，所使用的局部设计坐标轴以飞行员为原点，而三个轴向分别是飞行员的前后、上下和左右。这些轴向能方便地定出机翼、机尾、升降舵等的位置，但这些轴未必是飞机的主轴。结果是使用方便定位的坐标轴但需处理非零的惯性积（惯性积可能是正值，也可能是负值）。

前面已经介绍过对于由许多部分组合而成的物体，要如何计算合并后的转动惯量。考虑惯性积项目时也会使用相同的步骤，物体组成元素的局部惯性积是0除外，这只有将元素表示成质点、球体、长方体或是类似的简单图形时才会发生。而在这些状况下，刚体的惯性积主要来自刚体中每个元素的轴转换项目。

在看范例程序代码之前，要先改写元素结构并且加入新的变量，用以记录元素的局部转动惯量：

```
typedef struct _PointMass
{
      float mass;
      Vector designPosition;
      Vector correctedPosition;
      Vector localInertia;
} PointMass;
```

这里使用向量来表示三个局部转动惯量，并且假设每个元素的局部惯性积皆为0。

以下的范例程序代码介绍如何计算物体某个元素的惯性张量：

```
float         Ixx, Iyy, Izz, Ixy, Ixz, Iyz;
Matrix3x3     InertiaTensor;

Ixx = 0;      Iyy = 0;     Izz = 0;
Ixy = 0;      Ixz = 0;     Iyz = 0;

for (i = 0; i<_NUMELEMENTS; i++)
{
    Ixx +=  Element[i].LocalInertia.x +
            Element[i].mass * (Element[i].correctedPosition.y *
            Element[i].correctedPosition.y +
            Element[i].correctedPosition.z *
            Element[i].correctedPosition.z);

    Iyy +=  Element[i].LocalInertia.y +
            Element[i].mass * (Element[i].correctedPosition.z *
            Element[i].correctedPosition.z +
            Element[i].correctedPosition.x *
            Element[i].correctedPosition.x);
```

```
        Izz +=  Element[i].LocalInertia.z +
                Element[i].mass * (Element[i].correctedPosition.x *
                Element[i].correctedPosition.x +
                Element[i].correctedPosition.y *
                Element[i].correctedPosition.y);

        Ixy +=  Element[i].mass * (Element[i].correctedPosition.x *
                Element[i].correctedPosition.y);

        Ixz +=  Element[i].mass * (Element[i].correctedPosition.x *
                Element[i].correctedPosition.z);

        Iyz +=  Element[i].mass * (Element[i].correctedPosition.y *
                Element[i].correctedPosition.z);
    }

    // e11 表示在第 1 列第 1 行的元素，e12 表示第 1 列第 2 行的元素 ...

     InertiaTensor.e11 = Ixx;
     InertiaTensor.e12 = -Ixy;
     InertiaTensor.e13 = -Ixz;

     InertiaTensor.e21 = -Ixy;
     InertiaTensor.e22 = Iyy;
     InertiaTensor.e23 = -Iyz;

     InertiaTensor.e31 = -Ixz;
     InertiaTensor.e32 = -Iyz;
     InertiaTensor.e33 = Izz;
```

请注意，这里的惯性张量是对通过刚体合并后重心的轴而言的，因此当使用转换轴公式时，物体的每个组成元素相对于合并后的重心都要使用更正过的坐标。

同时也要注意，这里对于惯性张量的计算都是以物体本体坐标定位的。本章前面几节也说过，最好是以局部坐标改写角运动方程，并且使用局部惯性张量来存储在实时模拟程序中会使用的数字。

# 第二章

# 运动学

这一章将介绍运动学的基本观念，尤其是线性和角的位移、速度及加速度的概念。本章的范例程序示范如何实现粒子运动的运动学方程；在讨论粒子运动之后，接着将介绍刚体运动的特点。本章及下一章“力学”是学习第四章“动力学”之前必修的课程。

## 简介

在前言中曾提到，运动学是对物体运动的研究，并不考虑物体上的作用力。因此，运动学将专注于物体的位置、速度及加速度、这些特性的关联，以及它们如何随时间改变。

这里，你会看到两种类型的物体：粒子和刚体。前言中便提过刚体是一种粒子系统，粒子之间保持固定距离且无相对的移动与转动。换句话说，刚体移动时其外形不变，或者外形的变化小到（不重要）可以忽视不计。就刚体而言，其大小与方位是很重要的，且必须同时考虑物体的线性运动及角运动。

另一方面，粒子是具有质量的物体，而其大小在研究的问题中是可忽略或不重要的。例如，对于抛射体或长程火箭的路径，分析其轨道时大可忽略其大小。当讨论粒子时，其线运动是重要的，而粒子本身的角运动则相反。就好像在看一个问题时，你要将距离拉远才能看到宏观的全貌，反之，在看刚体的旋转时，要拉近距离才能看清微观的物体。

不管是粒子还是刚体的问题，均有共同的运动学特性，就是物体的位置、速度和加速度。下面会详细讨论这些特性。

# 速度与加速度

一般而言，速度是一个具有量值及方向的向量。速度的量值就是速率，这是常见的名词 —— 就是在你在公路上开车，车速表上显示的。正式地说，速率是移动率，或者称为行进距离与花费时间的比率。数学式为：

$$v = \Delta s/\Delta t$$

式中，$v$ 是速率，即速度 $\boldsymbol{v}$ 的大小，而 $\Delta s$ 是经过时间 $\Delta t$ 后的行进距离。注意，此关系式显示速率的单位是长度除以时间，因次是 $L/T$。一些常见的速率单位如：尺 / 每秒，ft/s；英里 / 每小时，mi/h；以及米 / 每秒，m/s。

以下是简单的例子：车子沿着直线行驶；它于时间 $t_1$ 经过标记 1 而在时间 $t_2$ 经过标记 2，其中 $t_1$ 等于 0 秒、$t_2$ 等于 1.136 秒，两个标记间的距离 $s$ 长 100 ft（如图 2-1 所示）。试计算车子的速率。

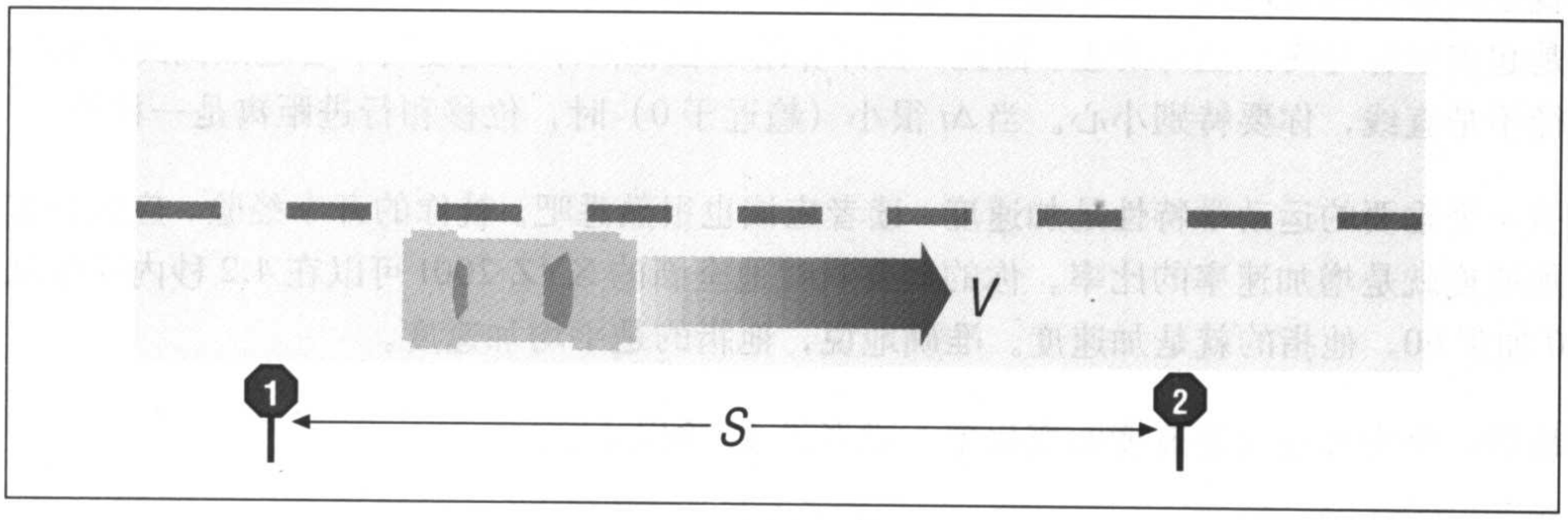



图 2-1：范例：行车速率

假设 $s$ 等于 100 ft；因此 $\Delta s$ 等于 100 ft，而 $\Delta t$ 等于 $t_2 - t_1$，即 1.136 s。通过这段距离的车速为：

$$v = \Delta s/\Delta t = 100 \text{ ft } /1.136 \text{ s} = 88.03 \text{ ft/s}$$

近似于 60 mi/h。这是简单的一维空间范例，但它谈到一个重点：刚才计算的速率是该段距离的平均车速。无法得知车子的加速度或它是否以 60 mi/h 的固定速率行进。车子在 100 ft 的距离间有加速（或减速）会使问题更恰当。

要想更精确地分析本例中车子的移动，需要了解即时速度的概念。即时速度是在给定时间点的特定速度，而非经过一段如以上范例的时间间隔。即你需考虑很小的 $\Delta t$。在数学上即考虑极限 $\Delta t$ 趋近于 0，即 $\Delta t$ 极小时的情形。数学式如下：

$$v = \lim_{\Delta t \to 0}(\Delta s/\Delta t)$$

以微分学的术语，速度是位移（位置的改变）对时间的导数：

$$v = \mathrm{d}s \,/\, \mathrm{d}t$$

你可以重新排列关系式并从区间 $s_1$ 到 $s_2$ 对 $s$ 积分、从 $t_1$ 到 $t_2$ 对 $t$ 积分，如下所示：

$$v\,\mathrm{d}t = \mathrm{d}s$$

$$\int_{s_1}^{s_2} \mathrm{d}s = \int_{t_1}^{t_2} v\,\mathrm{d}t$$

$$s_2 - s_1 = \Delta s = \int_{t_1}^{t_2} v\,\mathrm{d}t$$

此关系式显示位移是速度对时间的积分。这就是位移和速度之间前后运算的方式。

在运动学上，位移与行进距离两者要做区分。在一维空间中，位移等于行进距离；然而，就空间中的向量而言，位移实际上是起点至终点的向量，而不考虑行进的路径，即位移是起点坐标与终点坐标的差。因此，在计算给定位移的平均速度时，若起点到终点的路径不是直线，你要特别小心。当 $\Delta t$ 很小（趋近于 0）时，位移和行进距离是一样的。

另一个重要的运动学特性是加速度，读者应该也很熟悉吧。就你的开车经验，应该知道加速度就是增加速率的比率。你的朋友吹嘘他全新的 XYZ 200I 可以在 4.2 秒内速度从 0 加到 60，他指的就是加速度。准确地说，他指的是平均加速度。

最后，平均加速度是速度的变化率，或者说 $\Delta v$ 除以 $\Delta t$：

$$a = \Delta v/\Delta t$$

取极限 $\Delta t$ 趋近于 0 便是即时加速度：

$$a = \lim_{\Delta t \to 0}\Delta v/\Delta t$$

$$\mathrm{a} = \mathrm{d}v \,/\mathrm{d}t$$

因此，加速度是速度的时间变化率，或者说速度对时间的导数。

重新排列及合并得到：

$$\mathrm{d}v = a\,\mathrm{d}t$$

$$\int_{v_1}^{v_2} \mathrm{d}v = \int_{t_1}^{t_2} a\,\mathrm{d}t$$

$$v_2 - v_1 = \Delta v = \int_{t_1}^{t_2} a\,\mathrm{d}t$$

此关系式提供速度与加速度间前后运算的方式。

因此，位移、速度及加速度的关系式如下：

$$a = \mathrm{d}v/\mathrm{d}t = \mathrm{d}^2s/\mathrm{d}t^2$$

以及

$$v\,\mathrm{d}v = a\,\mathrm{d}s$$

这是移动的运动学微分方程，接下来的章节会有这些方程的范例，可应用于一些常见的运动学题型。

## 定加速度

运动学上最简单的题型便是定加速度。这类问题最好的例子就是重力加速度 g，当物体相当靠近地表移动时，其重力加速度是一个常数 32.174 $\mathrm{ft/s^2}$（9.8 $\mathrm{m/s^2}$）。定加速度使得对时间的积分相对地简单了，因为你可以把加速度常数移到被积函数外，只留下 d$t$。

当加速度是常数时，将前面讨论的速度与加速度的关系式积分，得到下面的即时速度方程：

$$\int_{v_1}^{v_2} \mathrm{d}v = \int_{t_1}^{t_2} a\,\mathrm{d}t$$

$$\int_{v_1}^{v_2} \mathrm{d}v = a\int_{t_1}^{t_2} \mathrm{d}t$$

$$v_2 - v_1 = a\int_{t_1}^{t_2} \mathrm{d}t$$

$$v_2 - v_1 = a(t_2 - t_1)$$

$$v_2 = at_2 - at_1 + v_1$$

当 $t_1$ 等于 0 时，可将方程改写成下列形式：

$$v_2 = at_2 + v_1$$

$$v_2 = v_1 + at_2$$

在已知经过时间、初速度及定加速度时，可求出给定时间的即时速度。

也能由位移的微分方程导出另一个方程，其中速度为位移的函数而非时间的函数：

$$v\,\mathrm{d}v = a\,\mathrm{d}s$$

方程两边同时积分，得到另一个即时速度的函数：

$$\int_{v_1}^{v_2} v\,\mathrm{d}v = \mathrm{a}\int_{s_1}^{s_2}\mathrm{d}s$$

$$\left(v_2^2 - v_1^2\right)/2 = a(s_2 - s_1)$$

$$v_2^2 = 2a(s_2 - s_1) + v_1^2$$

将下列微分方程积分，可以导出类似的公式，其中位移为速度、加速度及时间的函数

$$v\,\mathrm{d}t = \mathrm{d}s$$

利用前面导出的即时速度公式：

$$v_2 = v_1 + at$$

代入 $v$，得到以下公式：

$$s_2 = s_1 + v_1 t + (at^2)/2$$

总结以上导出的三个运动学方程如下：

$$v_2 = v_1 + at$$

$$v_2^2 = 2a(s_2 - s_1) + v_1^2$$

$$s_2 = s_1 + v_1 t + (at^2)/2$$

请记住，这些方程只用在加速度为常数时。当然，加速度可以是0 甚至是负数（在物体减速的情况下）。

你可以代入不同的变量重新排列这些方程，也可以用上述的方法导出其他方程。为了方便读者，表 2-1 列出了其他关于定加速度的运动学方程。

**表 2-1：定加速度运动学公式**

| 待求量 | 已知变量 | 代入公式 |
|---|---|---|
| $a$ | $\Delta t$, $v_1$, $v_2$ | $a = (v_2 - v_1)/\Delta t$ |
| $a$ | $\Delta t$, $v_1$, $\Delta s$ | $a = (2\Delta s - 2v_1\Delta t)/(\Delta t)^2$ |
| $a$ | $v_1$, $v_2$, $\Delta s$ | $a = \left(v_2^2 - v_1^2\right)/(2\Delta s)$ |
| $\Delta s$ | $a$, $v_1$, $v_2$ | $\Delta s = \left(v_2^2 - v_1^2\right)/(2a)$ |
| $\Delta s$ | $\Delta t$, $v_1$, $v_2$ | $\Delta s = (\Delta t/2)(v_1 + v_2)$ |

表 2-1：定加速度运动学公式（续）

| 待求量 | 已知变量 | 代入公式 |
|---|---|---|
| $\Delta t$ | $a, v_1, v_2$ | $\Delta t = (v_2 - v_1)/a$ |
| $\Delta t$ | $a, v_1, \Delta s$ | $\Delta t = \left(\sqrt{v_1^2 + 2a\Delta s} - v_1\right)/a$ |
| $\Delta t$ | $v_1, v_2, \Delta s$ | $\Delta t = (2\Delta s)/(v_1 + v_2)$ |
| $v_1$ | $\Delta t, a, v_2$ | $v_1 = v_2 - a\Delta t$ |
| $v_1$ | $\Delta t, a, \Delta s$ | $v_1 = \Delta s/\Delta t - (a\Delta t)/2$ |
| $v_1$ | $a, v_2, \Delta s$ | $v_1 = \sqrt{v_2^2 - 2a\Delta s}$ |

在加速度不是常数而是时间、速度或位置的函数时，可以将加速度函数代入先前所示的微分方程，以导出新的即时速度及位移的方程。下一节开始讨论这类问题。

# 不定加速度

现实世界中有一种常见的情况，就是阻力作用于移动中的物体之上。一般而言，阻力与速度的平方成正比。回想一下牛顿第二运动定律，$F = ma$，可以得出推论，由这些阻力引起的加速度与速度的平方也成正比。

稍后将介绍计算这类阻力的技巧，现在令阻力引起的加速度的函数形式为：

$$a = -kv^2$$

式中，$k$ 是常数而负号表示加速度方向与物体速度方向相反。接着将加速度公式代入上面的方程，重新排列得到：

$$a = \mathrm{d}v/\mathrm{d}t$$

$$-kv^2 = \mathrm{d}v/\mathrm{d}t$$

$$-k\,\mathrm{d}t = \mathrm{d}v/v^2$$

若将此方程右边从 $v_1$ 到 $v_2$ 积分而左边从 0 到 $t$ 积分，然后求出 $v_2$，会得到即时速度为初速度及时间的函数，即：

$$-k\int_0^t \mathrm{d}t = \int_{v1}^{v_2} (1/v^2)\,\mathrm{d}v$$

$$-kt = 1/v_1 - 1/v_2$$

$$v_2 = v_1/(1 + v_1 kt)$$

如果将关系式 $v = ds/dt$ 中的 $v$ 以此方程取代并再积分，会得到新的方程，其中位移为初速度及时间的函数。过程如下：

$$v\,dt = ds，\text{其中 } v = v_1/(1 + v_1kt)$$

$$\int_0^t v\,dt = \int_{s1}^{s_2} ds$$

$$\int_0^t [v_1/(1+v_1kt)]\,dt = \int_{s_1}^{s_2} ds$$

$$\ln(1+v_1kt)/k = s_2 - s_1$$

若 $s_1$ 等于 0，则

$$s = \ln(1 + v_1kt)/k$$

注意，方程中 ln 是自然对数运算符。

这个范例证明了不定加速度相对于定加速度的复杂度。这是相当简单的范例，你可以从中导出速度及位移的封闭式解答。然而实际上，作用于移动物体上的力有许多种，由此引起的加速度运算式相当复杂。这种复杂度使上面的封闭式解答达无法求得，除非简化问题的限制，并用其他解答技巧，如数值积分。关于这类问题将在第十一章深入探讨。

## 2D 粒子运动学

在讨论一维的移动（即移动限制于一条直线之上）时，问题相当简单，可以直接套用先前导出的公式求出即时速度、加速度及位移。然而在 2D 空间中，移动可以是平面上的任意方向，必须将运动学特性（速度、加速度及位移）当成向量。

使用标准的笛卡尔坐标系统，必须给出位移、速度及加速度的 $x$，$y$ 分量。基本上，可以分开处理 $x$ 分量及 $y$ 分量，然后再合并这些分量以定义对应向量的大小。

为了协助记录 $x$，$y$ 分量，令 $i$ 和 $j$ 分别为 $x$ 与 $y$ 方向的单位向量，便能以下列形式表示代表运动学属性的向量：

$$\boldsymbol{v} = v_x\boldsymbol{i} + v_y\boldsymbol{j}$$

$$\boldsymbol{a} = a_x\boldsymbol{i} + a_y\boldsymbol{j}$$

若 $x$ 是 $x$ 方向的位移，而 $y$ 是 $y$ 方向的位移，则位移向量为：

$$\boldsymbol{s} = x\boldsymbol{i} + y\boldsymbol{j}$$

接着分别将 $s$ 做一次微分得到 $v$，二次微分得到 $a$

$$\boldsymbol{v} = \mathrm{d}\boldsymbol{s}/dt = \mathrm{d}x/\mathrm{d}t\boldsymbol{i} + \mathrm{d}y/\mathrm{d}t\boldsymbol{j}$$

$$\boldsymbol{a} = \mathrm{d}\boldsymbol{v}/\mathrm{d}t = \mathrm{d}^2\boldsymbol{s}/\mathrm{d}t^2 = \mathrm{d}^2x/\mathrm{d}t^2\,\boldsymbol{i} + \mathrm{d}^2y/\mathrm{d}t^2\boldsymbol{j}$$

思考一下编写狩猎游戏的简单范例，需计算射出的子弹从瞄准点到真正击中目标的点之间的垂直落差。此例中，假设子弹飞过空气中时无风、无阻力（第六章将处理有风及空气阻力的抛体运动）。这些假设将问题简化成定加速度的问题，本例中加速度是由重力所引起的，因此就是重力加速度，它使子弹从步枪到目标行进时产生落差。如图 2-2 所示。

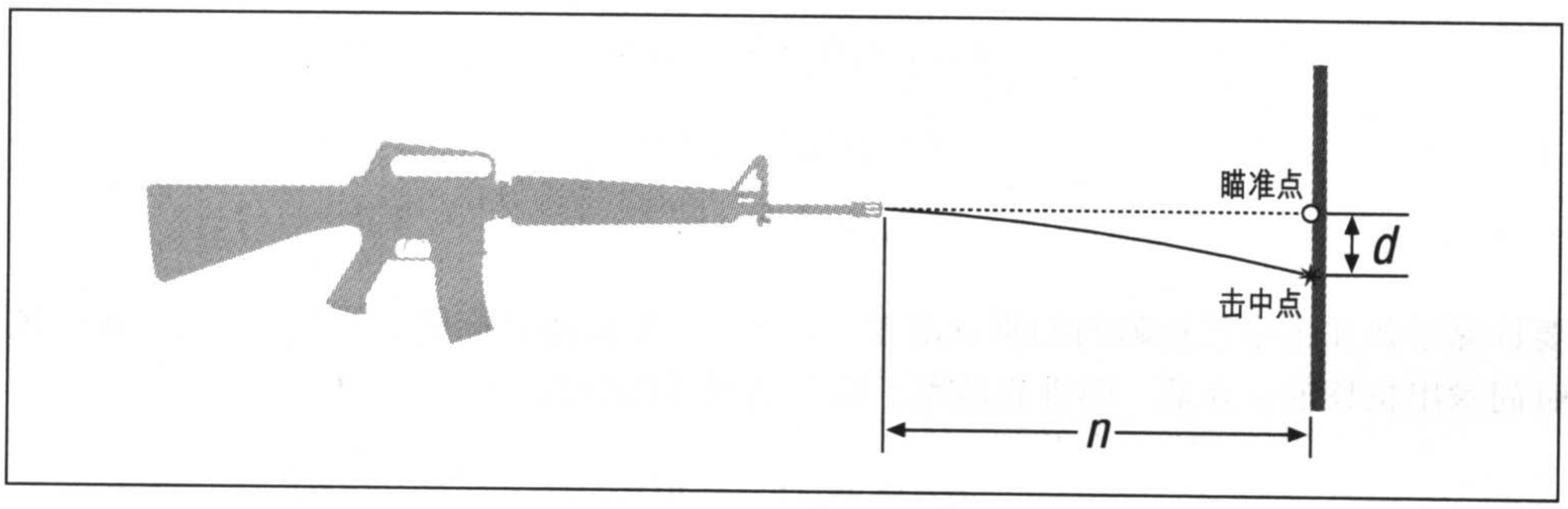



图 2-2：2D 运动学范例

令 2D 坐标系的原点在步枪的末端（枪口），$x$ 轴指向目标而 $y$ 轴向上。沿 $x$ 轴的正位移是朝向目标的，而沿 $y$ 轴的正位移便是往上的。这意味着重力加速度作用于 $y$ 轴的负方向。

分开处理 $x$ 分量与 $y$ 分量可将问题分解为小的、易处理的部分。先来看 $x$ 分量，子弹会以 $x$ 方向的枪口初速度 $v_{\mathrm{m}}$ 离开步枪，由于忽略空气阻力，此速率是常数。因此：

$$a_x = 0$$

$$v_x = v_{\mathrm{m}}$$

$$x = v_x t = v_{\mathrm{m}} t$$

再来看 $y$ 分量，当子弹脱离步枪时，$y$ 方向的初速率是 0，但 $y$ 轴加速度是 $-\mathrm{g}$（重力所引起）。因此：

$$a_y = -\mathrm{g} = \mathrm{d}v_y/\mathrm{d}t$$

$$v_y = a_y t = -\mathrm{g}t$$

$$y = (1/2)a_y t^2 = -(1/2)\mathrm{g}t^2$$

位移、速度及加速度向量的表示如下：

$$\boldsymbol{s} = (v_\mathrm{m}t)\ \boldsymbol{i} - (1/2\mathrm{g}t^2)\boldsymbol{j}$$

$$\boldsymbol{v} = (v_\mathrm{m})\ \boldsymbol{i} - (\mathrm{g}t)\boldsymbol{j}$$

$$\mathbf{a} = -(\mathrm{g})\boldsymbol{j}$$

这些方程在子弹离开步枪到击中目标的时间内，可求得给定时间点的即时位移、速度及加速度。而这些向量的大小即为给定时间的总位移、速度和加速度。例如：

$$s = \sqrt{(v_\mathrm{m}t)^2 + (1/2\mathrm{g}t^2)^2}$$

$$v = \sqrt{(v_\mathrm{m})^2 - (\mathrm{g}t)^2}$$

$$a = \sqrt{\mathrm{g}^2} = \mathrm{g}$$

要计算子弹在击中目标瞬间的垂直落差，必须先计算到达目标所需的时间，然后再以该时间求出位移的 $y$ 分量，即垂直落差。以下是所用的公式：

$$t_\mathrm{hit} = x_\mathrm{hit}/v_\mathrm{m} = n/v_\mathrm{m}$$

$$d = y_\mathrm{hit} = -(1/2)\mathrm{g}(t_\mathrm{hit})^2$$

式中，$n$ 是步枪到目标间的距离，而 $d$ 是子弹在目标处的垂直落差。

若到目标的距离 $n$ 等于 500 m 而枪口速度 $v_\mathrm{m}$ 等于 800 m/s，则得到 $t_\mathrm{hit}$ 和 $d$

$$t_\mathrm{hit} = 0.625\ \mathrm{s}$$

$$d = 1.9\ \mathrm{m}$$

这些结果显示，在该射程下欲击中预期的目标，需要瞄准目标上方约 2 米的点。

## 3D 粒子运动学

将运动学特性向量扩大到3D空间并不难，只需要在上一节2D运动学描述的向量表示法上再增加一个分量。以 $\boldsymbol{k}$ 作为 $z$ 方向的单位向量，现在可将位移、速度及加速度写成：

$$\boldsymbol{s} = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

$$\boldsymbol{v} = \mathrm{d}s/\mathrm{d}t = \mathrm{d}x/\mathrm{d}t\ \boldsymbol{i} + \mathrm{d}y/\mathrm{d}t\ \boldsymbol{j} + \mathrm{d}z/\mathrm{d}t\ \boldsymbol{k}$$

$$\boldsymbol{a} = \mathrm{d}^2\boldsymbol{s}/\mathrm{d}t^2 = \mathrm{d}^2x/\mathrm{d}t^2\,\boldsymbol{i} + \mathrm{d}^2y/\mathrm{d}t^2\,\boldsymbol{j} + \mathrm{d}^2z/\mathrm{d}t^2\,\boldsymbol{k}$$

现在不再是分开处理两个分量再合并，而是得分开处理三个分量再合并它们。下面实例说明。

假定不再是狩猎游戏，你现在要写一个游戏，让玩家从战舰发射炮弹攻击某段距离的目标（例如另一艘船或建筑物之类的陆地目标）。为了给玩家增加此行动的复杂度，你可能想让他能控制许多影响炮弹轨道的因素，即大炮（cannon）的发射角度（有水平及垂直角度），以及炮弹的枪口速度，此速度是炮弹填入大炮时由填充在后面的火药量所控制的。如图 2-3 所示。

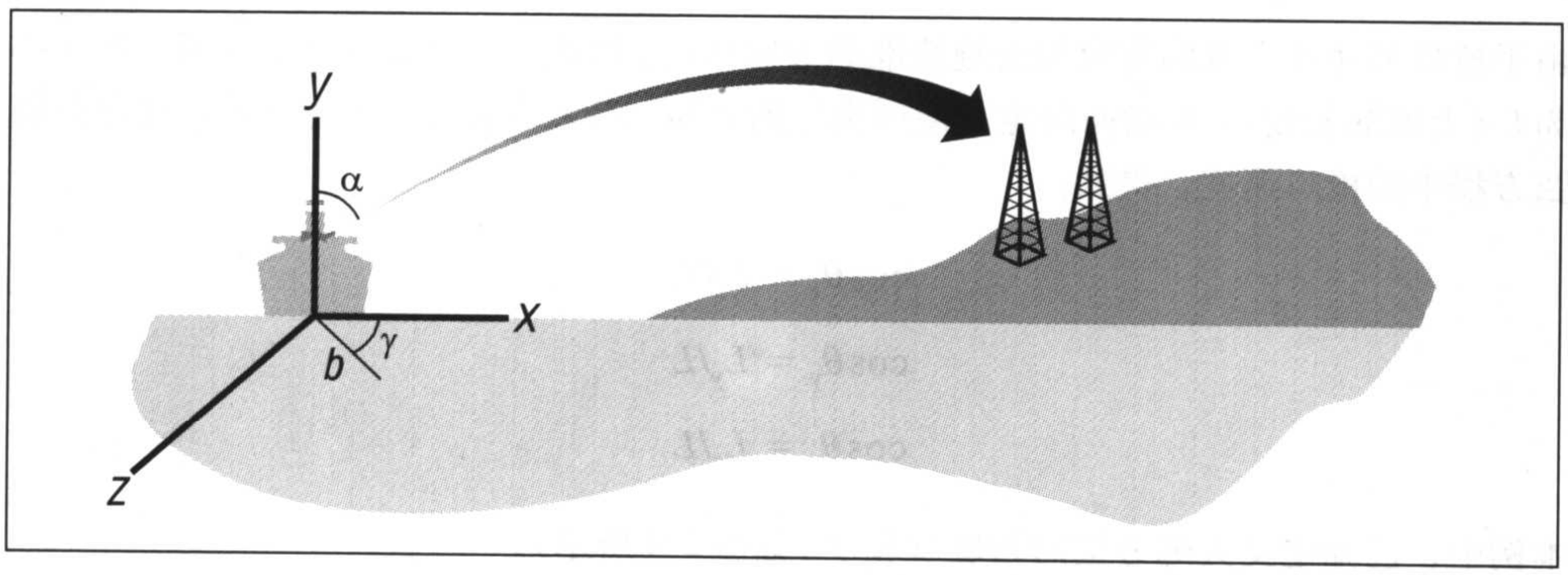



图 2-3：3D 运动学范例

下面将告诉你，如何通过个别处理向量分量再合并这些分量，来建立此问题的运动学方程。

## x 分量

这里的 $x$ 分量与上一节的步枪范例中的相似，没有阻力作用于炮弹之上。因此，加速度的 $x$ 分量是 0，即表示速度的 $x$ 分量是固定的，且等于炮弹离开大炮时的枪口速度的 $x$ 分量。注意，因为炮管可能不是水平的，因此需计算枪口速度的 $x$ 分量，它是大炮瞄准方向的函数。

枪口速度向量为：

$$\boldsymbol{v}_{\mathrm{m}} = v_{\mathrm{m}x}\boldsymbol{i} + v_{\mathrm{m}y}\boldsymbol{j} + v_{\mathrm{m}z}\boldsymbol{k}$$

$\boldsymbol{v}_{\mathrm{m}}$ 的方向由玩家把大炮对准的方向所决定，其大小由玩家选择填入大炮的火药量所决

定。要想求得枪口速度的分量，需要展开这些分量的方程，并以大炮的方向角及枪口速度大小表示。

可利用速度的方向余弦（cosine）决定速度分量，如下：

$$\cos\theta_x = v_{mx}/v_m$$

$$\cos\theta_y = v_{my}/v_m$$

$$\cos\theta_z = v_{mz}/v_m$$

可参阅附录一有关速度方向余弦的叙述和说明。

由于枪口初速度向量的方向与大炮瞄准的方向相同，因此可以将大炮当成向量，其大小为$L$（大炮的长度），并指向给定角度所决定的方向。以大炮长度$L$及其分量取代方向余弦方程中的枪口速度，得到：

$$\cos\theta_x = L_x/L$$

$$\cos\theta_y = L_y/L$$

$$\cos\theta_z = L_z/L$$

本例中，已知定义大炮方位的角度$\alpha$和$\gamma$（如图2-4所示）。

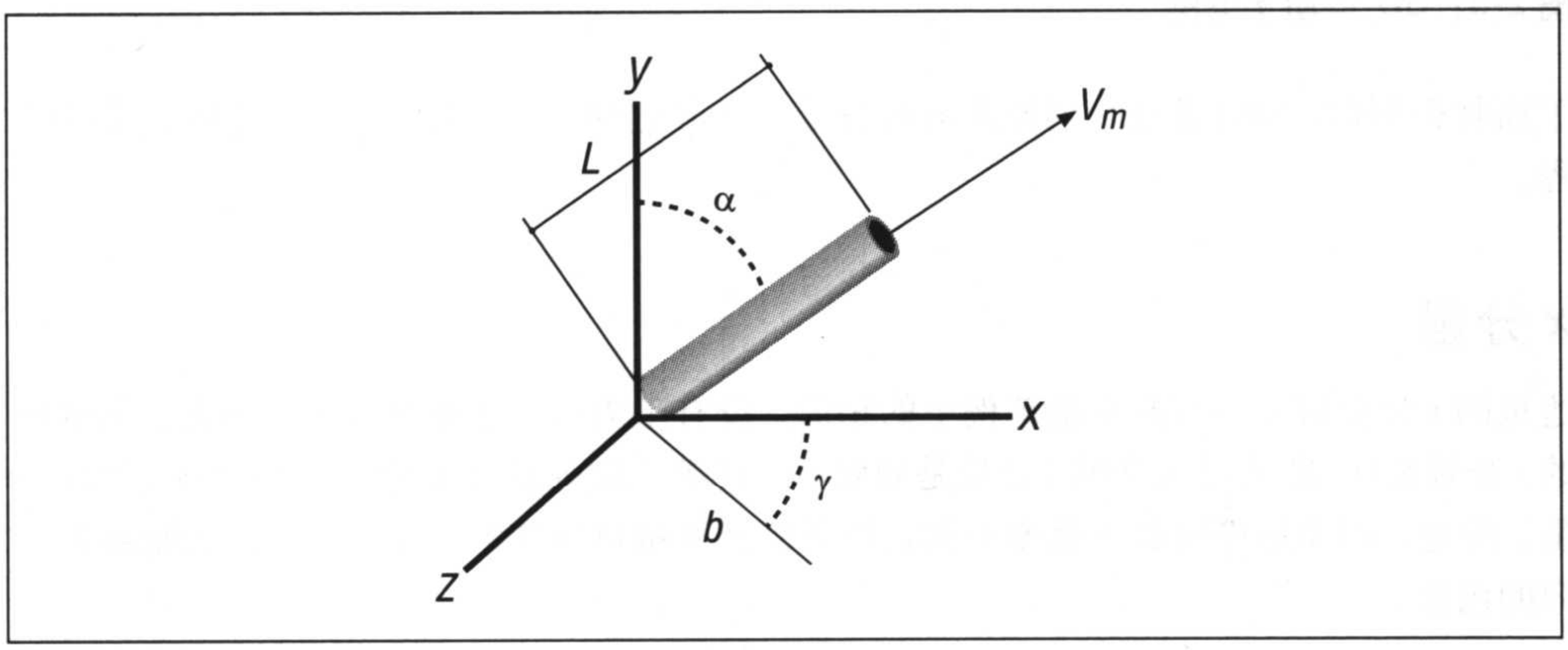



图2-4：大炮方位

用这些角度，可求出大炮长度$L$在$xz$平面上的投影$b$：

$$b = L\cos(90^\circ - \alpha)$$

而大炮长度 $L$ 在每个坐标轴上的分量等于：

$$L_x = b\cos\gamma$$

$$L_y = L\cos\alpha$$

$$L_z = b\sin\gamma$$

现在有了计算方向余弦所需的信息，枪口初速度分量的方程可写成：

$$v_{mx} = v_m\cos\theta_x$$

$$v_{my} = v_m\cos\theta_y$$

$$v_{mz} = v_m\cos\theta_z$$

最后，得到位移、速度和加速度的 $x$ 分量：

$$a_x = 0$$

$$v_x = v_{mx} = v_m\cos\theta_x$$

$$x = v_x t = (v_m\cos\theta_x)t$$

## y 分量

在这里，除了 $y$ 方向的初速度以外，$y$ 分量恰如之前的步枪范例：

$$v_{my} = v_m\cos\theta_y$$

因此，

$$a_y = -g$$

$$v_y = v_{my} + at = (v_m\cos\theta_y) - gt$$

在计算 $y$ 分量位移的方程之前，需考虑炮座的海拔高度，加上炮管尾端（炮口）的高度，才能求出当炮弹离开大炮时位移的初始 $y$ 分量。设 $y_b$ 为炮座的海拔高度，而 $L$ 为炮管的长度，则位移的初始 $y$ 分量 $y_o$ 等于：

$$y_o = y_b + L\cos\alpha$$

得到 $y$ 的方程：

$$y = y_o + v_{my}t + (1/2)at^2$$

$$y = (y_b + L\cos\alpha) + (v_m\cos\theta_y)t - (1/2)gt^2$$

## z 分量

z 分量与 x 分量相似，方程如下：

$$a_z = 0$$

$$v_z = v_{mz} = v_m \cos\theta_z$$

$$z = v_z t = (v_m \cos\theta_z)t$$

## 向量

处理好所有的分量后，便可以将它们组合成运动学特性的向量形式了。本例所求出的位移、速度和加速度向量如下所示：

$$\boldsymbol{s} = [(v_m \cos\theta_x)t]\boldsymbol{i} + [(v_b + L\cos\alpha) + (v_m \cos\theta_y)t - (1/2)gt^2]\boldsymbol{j} + [(v_m \cos\theta_z)t]\boldsymbol{k}$$

$$\boldsymbol{v} = [v_m \cos\theta_x]\boldsymbol{i} + [v_m \cos\theta_y - gt]\boldsymbol{j} + [v_m \cos\theta_z]\boldsymbol{k}$$

$$a = -gj$$

注意到位移向量基本上是指出在任一瞬时间，炮弹质心（质量中心）的位置，因此，可利用此向量测定从大炮到目标的弹道。

## 命中目标

现在，有了完整的描述弹道的方程，接着需考虑目标的位置，才能决定何时能直接命中。为了告诉你如何办到，以下有一个范例程序，实现这些运动学方程，以及检查炮弹是否击中目标的简易方块边界的碰撞侦测方法。基本上，在每个计算炮弹在离开大炮后的位置的时间点上，都要检查此位置是否落于以立方体表示的目标对象的边界范围内。

此范例程序设定为能改变弹道模拟中所有的变量，并能检测变更的效果。此程序是简易的对话式应用程序，以 C 语言并利用 Windows API 函数写成。可执行文件名为 *cannon.exe*，而本例只有一个源代码文件 *cannon.c*，和一个头文件 *cannon.h*。此应用程序是用 Microsoft Developer Studio 编译和建构而成的。

图 2-5 显示了大炮范例程序的主窗口，而控制的变量显示在左边。上面的图解是往下俯看大炮与目标的鸟瞰视野，而下面的图解是侧面视野。

你可以改变出现在主窗口的任一变量，并单击 Fire 按钮查看产生的炮弹飞行路径。当炮弹击中目标或地面时，会出现消息方块。程序设计成能一再变更变量并再次单击 Fire 按

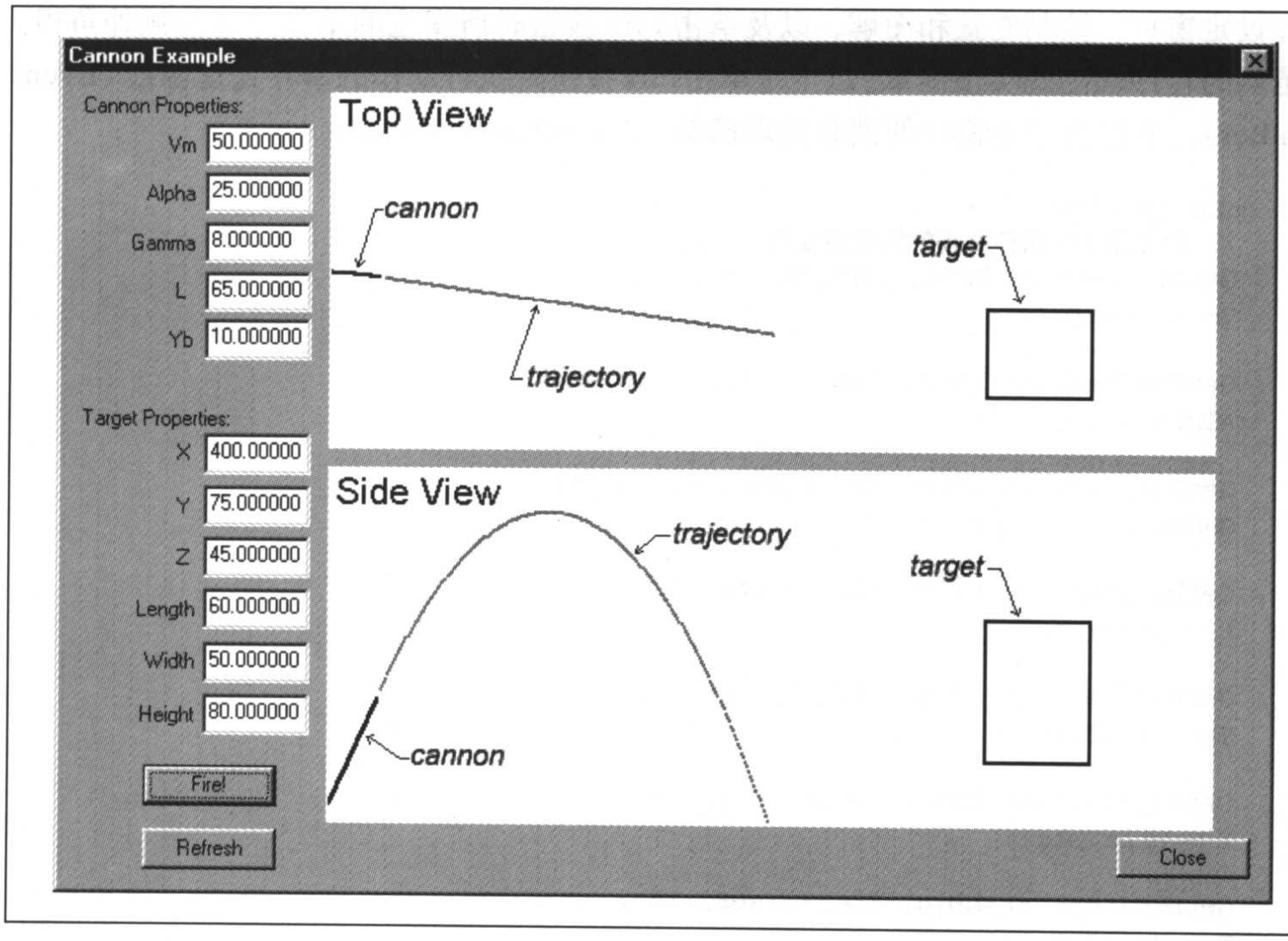



图 2-5：大炮范例程序的主窗口

钮查看结果，而无须清除上次的试射。这样你能估计需调整每个变量多少才能击中目标。当视野太乱时，可单击 Refresh 按钮重绘每个视野。

图 2-6 显示出击中目标之前所做的几次试射。

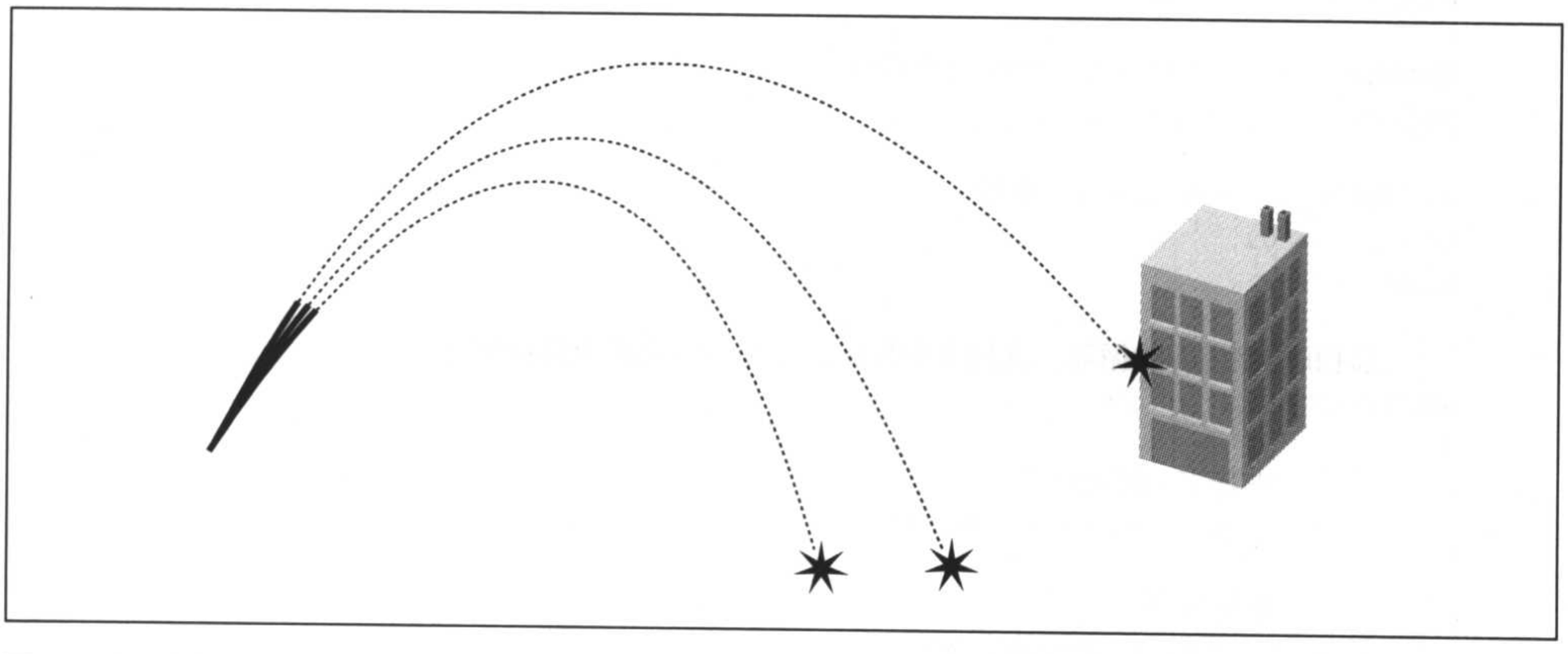


图 2-6：试射（侧面视野）

撇开设定窗口、控制变量和图解，以及单击Fire按钮时所发生的动作等这些额外负担。此范例的程序代码确实很简单。以下是单击Fire按钮时执行动作的事件处理函数（Event Handler），它包含于主窗口的消息处理函数（DemoDlgProc）中：

```
case IDC_FIRE:
 // 以显示于编辑控制项的值更新变量
 GetDlgItemText(hDlg, IDC_VM, str, 15);
 Vm = atof(str);

 GetDlgItemText(hDlg, IDC_ALPHA, str, 15);
 Alpha = atof(str);

 GetDlgItemText(hDlg, IDC_GAMMA, str, 15);
 Gamma = atof(str);

 GetDlgItemText(hDlg, IDC_L, str, 15);
 L = atof(str);

 GetDlgItemText(hDlg, IDC_YB, str, 15);
 Yb = atof(str);

 GetDlgItemText(hDlg, IDC_X, str, 15);
 X = atof(str);

 GetDlgItemText(hDlg, IDC_Y, str, 15);
 Y = atof(str);

 GetDlgItemText(hDlg, IDC_Z, str, 15);
 Z = atof(str);

 GetDlgItemText(hDlg, IDC_LENGTH, str, 15);
 Length = atof(str);

 GetDlgItemText(hDlg, IDC_WIDTH, str, 15);
 Width = atof(str);

 GetDlgItemText(hDlg, IDC_HEIGHT, str, 15);
 Height = atof(str);

 // 初始化time和status变量
 status = 0;
 time = 0;

 // 以时间控制模拟的循环。直到命中目标、炮弹击中地面或模拟超时
 while(status == 0)
 {
       // 进行下一时间点
       status = DoSimulation();

       // 更新视野
       hdc = GetDC(hTopView);
       GetClientRect(hTopView, &r);
       DrawTopView(hdc, &r);
```

```
        ReleaseDC(hTopView, hdc);

        hdc = GetDC(hSideView);
        GetClientRect(hSideView, &r);
        DrawSideView(hdc, &r);
        ReleaseDC(hSideView, hdc);
}

// 回报结果
if (status == 1)
      MessageBox(NULL, "Direct Hit", "Score!", MB_OK);

if (status == 2)
      MessageBox(NULL, "Missed Target", "No Score.", MB_OK);

if (status == 3)
      MessageBox(NULL, "Timed Out", "Error", MB_OK);
break;
```

前几行只是取得主窗口上的变量值。程序进入while循环后，随时间的递增步进并以位移向量*s*的公式(如前所示)重新计算弹道的位置。当前时间的炮弹位置在DoSimulation中计算。调用DoSimulation之后，程序立即更新主窗口的图解以显示弹道。若未有碰撞发生或时间未到达预先设定的超时值，DoSimulation会返回0，循环继续执行。

一旦while循环终止(DoSimulation返回非零值)，检测此函数调用的返回值，查看炮弹是击中目标还是地面。DoSimulation会返回3，指出时间花太久了，让程序不会卡在while循环。

来看看函数DoSimulation中发生什么事?

```
//---------------------------------------------------------------------//
// 定义自定型态代表3D向量的三个分量；其中i代表x分量
// j代表y分量，k代表z分量
//---------------------------------------------------------------------//
typedef struct TVectorTag
{
    double i;
    double j;
    double k;
} TVector;

//---------------------------------------------------------------------//
// 定义此模拟所需的变量
//---------------------------------------------------------------------//
double        Vm;       // 枪口速度的大小，单位m/s
double        Alpha;    // y轴（向上）与大炮的夹角
                        // 当夹角为0时，大炮笔直向上；当夹角为90度时，大炮为水平
double        Gamma;    // 大炮在xz平面上与x轴的夹角
                        // 当夹角为0时，大炮指向x正方向；若夹角为正值，大炮朝向z轴正向
double        L;        // 炮管的长度，单位m
```

```
double      Yb;       // 炮座的海拔高度，单位m

double      X;        // 目标中心的x坐标，单位m
double      Y;        // 目标中心的y坐标，单位m
double      Z;        // 目标中心的z坐标，单位m
double      Length;   // 沿x轴测得的目标长度，单位m
double      Width;    // 沿y轴测得的目标宽度，单位m
double      Height;   // 沿z轴测得的目标高度，单位m

TVector     s;        // 炮弹位置（位移）向量

double      time;     // 炮弹离开大炮后的经过时间，单位s
double      tInc;     // 步进时使用的时间递增量，单位s

double      g;        // 重力加速度，单位m/s²

//-------------------------------------------------------------------------//
// 此函数以时间步进。运动学特性都在这里求出。当击中目标时，函数返回1
// 若炮弹击中目标之前先击中地面（xz平面）则返回2；否则函数返回0
//-------------------------------------------------------------------------//
int DoSimulation(void)
//-------------------------------------------------------------------------//
{
    double cosX;
    double cosY;
    double cosZ;
    double xe, ze;
    double b, Lx, Ly, Lz;
    double tx1, tx2, ty1, ty2, tz1, tz2;

    // 往下一个时间推进
    time+=tInc;

    // 先求出大炮方位的方向余弦。真正的游戏中，不要将此计算放到此函数中
    // 因为模拟期间这些值不会改变，因此计算它们是浪费CPU时间的
    // 把它们放在这里只是为了在一个函数中能看到所有的计算步骤
    b = L * cos((90-Alpha) *3.14/180);      // 炮管在xz平面的投影
    Lx = b * cos(Gamma * 3.14/180);         // 炮管长度的x分量
    Ly = L * cos(Alpha * 3.14/180);         // 炮管长度的y分量
    Lz = b * sin(Gamma * 3.14/180);         // 炮管长度的z分量

    cosX = Lx/L;
    cosY = Ly/L;
    cosZ = Lz/L;

    // 这些是炮管末端的x和z坐标，作为x和z初始位移
    xe = L * cos((90-Alpha) *3.14/180) * cos(Gamma * 3.14/180);
    ze = L * cos((90-Alpha) *3.14/180) * sin(Gamma * 3.14/180);

    // 求出在此时间的位置向量
    s.i = Vm * cosX * time + xe;
    s.j = (Yb + L * cos(Alpha*3.14/180)) + (Vm * cosY * time) -
          (0.5 * g * time * time);
    s.k = Vm * cosZ * time + ze;
```

```
// 取得目标的范围（边界坐标）
tx1 = X - Length/2;
tx2 = X + Length/2;
ty1 = Y - Height/2;
ty2 = Y + Height/2;
tz1 = Z - Width/2;
tz2 = Z + Width/2;

// 检测炮弹是否通过目标，这里使用基本的碰撞侦测方案
// 只检查炮弹的坐标是否在目标边界方块之内。这只是示范用
// 但如果在给定的时间点炮弹位置的改变大到足以越过目标，可能会错失碰撞
// 较好的方式是看上个时间点的位置资料，并检测从上个位置到目前位置的线
// 查看该线是否与目标边界方块相交
if( (s.i >= tx1 && s.i <= tx2) &&
        (s.j >= ty1 && s.j <= ty2) &&
        (s.k >= tz1 && s.k <= tz2) )
        return 1;

// 检测与地面（xz 平面）的碰撞
if(s.j <= 0)
        return 2;

// 若时间过长，中断模拟，这样程序才不会卡在 while 循环
if(time>3600)
        return 3;

    return 0;
}
```

程序代码的注释让你能立刻了解什么事正在进行。此函数基本上做 4 件事：

- 以指定的时间增量递增时间变量；
- 求出初始枪口速度在 $x$，$y$，$z$ 方向的分量；
- 求出炮弹的新位置；
- 利用边界方块系统或地面，检查与目标的碰撞。

下面是计算炮弹位置的程序代码：

```
// 现在能求出此刻的位置向量
s.i = Vm * cosX * time + xe;
s.j = (Yb + L * cos(Alpha*3.14/180)) + (Vm * cosY * time) -
      (0.5 * g * time * time);
s.k = Vm * cosZ * time + ze;
```

这段程序代码使用先前的公式求出位移向量 $s$ 的三个分量。如果也要计算速度及加速度向量（只是为了看它们的值），应该将程序代码加在这部分。可设定一对新的全局变量作为速度和加速度向量（如同对位移向量所做的），并且套用之前的速度与加速度公式。

到这里该做的都做了。很明显，此范例程序的炮弹轨迹是抛物线，这通常是抛体运动的轨迹。关于此类运动第六章将有更详细的介绍。

即使原始程序代码都有注释，还是要重述有关本例使用的碰撞侦测方案的警告。因为只检查目前位置的坐标是否落在目标立方体的边界范围之内，所以若在给定时间点上位置的变化太大，就必须承担炮弹越过目标的风险。较好的方式是记录炮弹的上个位置，并检查上个位置到目前位置的线是否与目标立方体相交。

## 粒子爆炸的运动学

此刻你或许会对粒子运动学如何建立逼真的游戏内容感到好奇，除非你编写枪炮射击游戏。本章先提供一些概念，再展示一个例子。假设编写足球模拟程序，可利用粒子运动学模拟足球丢出或踢出之后的轨迹。当计算接球员是否能接住丢出的球时，也可以将他们当做粒子。在此情况下便有了接球员和球两个粒子，它们的运动是独立的；而你必须求出这两个粒子何时发生碰撞，便表示接到了（当然，除非玩家在球碰到手之后还漏接了）。你也可以找到类似的运动游戏的应用程序。

那么关于3D枪战游戏呢？撇开子弹、大炮、手榴弹等不讲，你如何将粒子运动学运用在此类问题上？是的，当主角奔跑或立定跳跃时，可以用粒子运动学来模拟这些动作。例如，当主角到达狭小通道的中央时，却发现有段缺口，想要越过这个缺口，你便让他后退几步再助跑起跳。真正需要做的便是定义主角的初速度（包含速率及起跳的角度），然后套用位移的速度公式求出他是否跳跃成功。也可以利用位移公式求出玩家的轨迹，让你可以对应地移动主角的视点，营造出跳跃的感觉。事实上，你也许已经利用过这些原理来模仿游戏中的动作或者至少你看过这种做法了（如果你玩过这类的游戏）。若主角跳跃距离太短而掉下去，可以使用速度的公式求出当他落到地面时的冲撞速度。根据冲撞速度便能决定减少主角生命点数(HP)的适当的损伤值，或若速度超过某个门槛时，便要对你的第一人称冒险说“再见”了！

简易粒子运动学的另一个用途是某些特殊效果，诸如粒子爆炸。这种效果相当容易实现并且确实能增加爆炸效果真实的感觉。粒子不会恰好随机飞出且成直线轨迹。相反，它们的上升与落下都受到本身的初速度、角度及重力加速度的影响，这是因为粒子具有质量，会受重力的影响。

接着示范运动学的粒子爆炸的范例。本例的程序代码是由先前的大炮范例改写的，因此有许多地方你应该很熟悉。此程序的主窗口如图2-7所示。

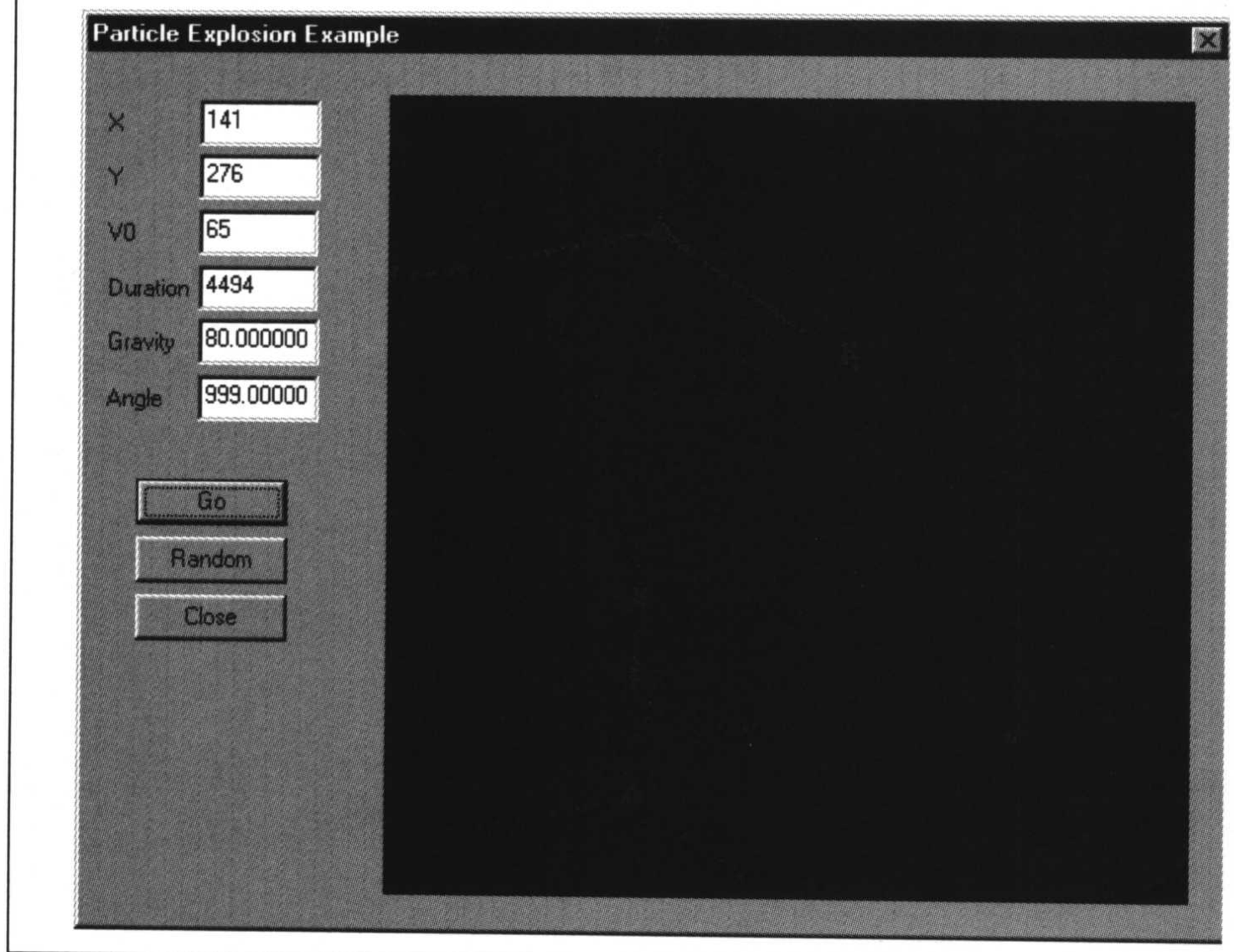



图 2-7：粒子爆炸程序

爆炸效果发生于右边的大方形区域。在这个截图中并没有体现出爆炸效果，因为在图中你只能看到一些点，这并不能真实地反映出实际效果；实际的爆炸效果是由这些点的运动形成的。

在左边的编辑控制项，可指定爆炸效果的 $x$ 位置和 $y$ 位置，以及粒子的初速度（为爆炸的强度），以毫秒为单位的持续时间（粒子的生命周期）和重力因子，最后还有角度。angle 参数可为 0～360° 间的任意数或 999。如果指定角度在 0～360° 的范围内，所有爆炸的粒子通常会向该方向发射出去。如果指定值为 999，则所有粒子会以随机方向散射出去。life 参数基本上是爆炸效果的寿命，当粒子接近该寿命时会慢慢淡出。

本例需要做的第一件事就是设定一些数据结构和全局变量，以表示粒子效果和制造效果的个别粒子，并描述上一段讨论的效果作用的初始参数。程序代码如下：

```
//-----------------------------------------------------------------------//
// 定义自定义类型表示效果中的每个粒子
//-----------------------------------------------------------------------//
typedef       struct _TParticle
```

```
{
 float        x;              // 粒子的x坐标
 float        y;              // 粒子的y坐标
 float        vi;             // 初速度
 float        angle;          // 初始轨道（方向）
 int             life;        // 持续时间，单位ms
 int             r;           // 粒子色彩的红色成分
 int             g;           // 粒子色彩的绿色成分
 int             b;           // 粒子色彩的蓝色成分
 int             time;        // 记录效果的时间
 float       gravity;         // 重力因子
 BOOL        Active;          // 指出粒子是“活的”还是“死的”

} TParticle;

#define      _MAXPARTICLES 50

typedef struct _TParticleExplosion
{
 TParticle   p[_MAXPARTICLES]; // 制造效果的粒子

 int         V0;               // 效果的初速度或强度
 int         x;                // 初始x位置
 int         y;                // 初始y位置
 BOOL        Active;           // 指出效果还在作用还是消失了

} TParticleExplosion;

//-------------------------------------------------------------------//
// 现在定义爆炸模拟所需的变量
//-------------------------------------------------------------------//
TParticle    Explosion;

int            xc;             // 效果的x坐标
int            yc;             // 效果的y坐标
int            V0;             // 初速度
int            Duration;       // 效果的寿命，单位ms
float          Gravity;        // 重力因子（加速度）
float          Angle;          // 指出粒子的方向
```

从程序中可以看出，粒子的爆炸效果是由大量粒子组成的。每个粒子的作用由运动学和每个粒子所设定的初始参数所决定。

无论何时单击GO按钮，指定的初始参数都会被用来将粒子的爆炸初始化（如果单击Random按钮，程序会随机选择这些初始值）。这都是在CreateParticleExplosion函数中执行的：

```
////////////////////////////////////////////////////////////////////
/*              本函数建立新的粒子爆炸效果

    x,y:        效果的起始点
    Vinit:      粒子飞出去的速度有多快（实际上指粒子的初速度）
```

```
    life:       粒子寿命；当粒子接近特定的寿命时它们便会淡化并消逝
    gravity:    重力加速度，控制粒子飞行时落下的速率
    angle:      粒子轨道的初始角度，指定为999会产生全方向散射粒子的爆炸；
                否则指定为0即向右，90即向上，180即向左，等等。
*/
void CreateParticleExplosion(int x, int y, int Vinit, int life,
                             float gravity, float angle)
{
    int     i;
    int     m;
    float   f;

    Explosion.Active = TRUE;
    Explosion.x = x;
    Explosion.y = y;
    Explosion.V0 = Vinit;

    for(i=0; i<_MAXPARTICLES; i++)
    {

        Explosion.p[i].x = 0;
        Explosion.p[i].y = 0;
        Explosion.p[i].vi = tb_Rnd(Vinit/2, Vinit);

        if(angle < 999)
        {
            if(tb_Rnd(0,1) == 0)
                m = -1;
            else
                m = 1;
            Explosion.p[i].angle = -angle + m * tb_Rnd(0,10);
        } else
            Explosion.p[i].angle = tb_Rnd(0,360);

        f = (float) tb_Rnd(80, 100) / 100.0f;
        Explosion.p[i].life = tb_Round(life * f);
        Explosion.p[i].r = 255;//tb_Rnd(225, 255);
        Explosion.p[i].g = 255;//tb_Rnd(85, 115);
        Explosion.p[i].b = 255;//tb_Rnd(15, 45);

        Explosion.p[i].time = 0;
        Explosion.p[i].Active = TRUE;
        Explosion.p[i].gravity = gravity;
    }
}
```

你会发现所有的粒子都设定于$x$及$y$坐标指定的相同位置散射出去；然而你会注意到每个粒子的初速度竟是从Vinit/2～Vinit随机选取的。这样做是为了让粒子作用有些变化。而每个粒子的life参数也是随机的，让它们不会在同一时间都淡化消逝。

粒子爆炸产生之后，程序进入循环开始扩散并描绘爆炸效果。此while循环如下：

```
while(status)
{
    DrawRectangle(hBufferDC, &r, 1, RGB(0,0,0));
    status = DrawParticleExplosion(hBufferDC);
    hdc = GetDC(hSideView);
    if(!BitBlt(hdc, 0, 0, r.right, r.bottom, hBufferDC, 0, 0, SRCCOPY))
    {
         MessageBox(NULL, "BitBlt failed", "Error", MB_OK);
         status = FALSE;
    }
    ReleaseDC(hSideView, hdc);
}
```

只要 status 维持 true，while 循环会继续跑下去，这表示粒子效果仍存在。所有粒子到达设定的 life 之后，效果便会消失而 status 将设为 false。所有粒子作用的计算都在 DrawParticleExplosion 函数中执行；while 循环剩下的程序代码将清除幕后缓冲区（offscreen buffer）然后将画面复制到主窗口。

DrawParticleExplosion会调用UpdateParticleState函数，以更新效果中每个粒子的状态，然后将效果绘制到作为参数传入的缓冲区。以下是这两个函数：

```
//-----------------------------------------------------------------//
// 绘制粒子系统并更新每个粒子的状态。当所有粒子消失时返回 false
//-----------------------------------------------------------------//
BOOL     DrawParticleExplosion(HDC hdc)
{
    int        i;
    BOOL       finished = TRUE;
    float      r;
    COLORREF   clr;

    if(Explosion.Active)
      for(i=0; i<_MAXPARTICLES; i++)
      {
          if(Explosion.p[i].Active)
          {
            finished = FALSE;
          r = ((float)(Explosion.p[i].life-
                Explosion.p[i].time)/(float)(Explosion.p[i].life));
          clr = RGB(tb_Round(r*Explosion.p[i].r),
                tb_Round(r*Explosion.p[i].g),
                tb_Round(r*Explosion.p[i].b));
          DrawCircle(      hdc,
                        Explosion.x+tb_Round(Explosion.p[i].x),
                        Explosion.y+tb_Round(Explosion.p[i].y),
                        2,
                        clr);
          Explosion.p[i].Active = UpdateParticleState(&(Explosion.p[i]), 10);

          }
    }
```

```
        if(finished)
            Explosion.Active = FALSE;

        return !finished;
}

//-----------------------------------------------------------------------//
/* 这是更新给定粒子的状态的一般函数
   p:          粒子结构的指针
   dtime:      推进粒子状态的时间增量，单位ms

   若此粒子的总消耗时间超过粒子设定的寿命
   则此函数返回 false，指出粒子应该死亡了
*/
BOOL    UpdateParticleState(TParticle* p, int dtime)
{
   BOOL retval;
   float     t;

   p->time+=dtime;
   t = (float)p->time/1000.0f;
   p->x = p->vi * cos(p->angle*PI/180.0f) * t;
   p->y = p->vi * sin(p->angle*PI/180.0f) * t + (p->gravity*t*t)/2.0f;

   if (p->time >= p->life)
       retval = FALSE;
   else
       retval = TRUE;
   return retval;
}
```

UpdateParticleState利用先前的运动学公式，来更新粒子的位置（为初速度、时间及重力加速度的函数）。调用UpdateParticleState之后，DrawParticleExplosion根据每个粒子的寿命及耗时，慢慢地降低粒子的色彩并让它褪成黑色。淡化效果是要显示粒子会随时间慢慢消逝，而非单纯地从屏幕消失不见。此效果类似于烟火在夜空中爆炸的效果。

# 刚体运动学

前面的章节讨论的粒子位移、速度与加速度的公式也适用于刚体。其差异是当考虑刚体的线性移动时，所追踪的刚体的点为其质心（重心）。

当刚体未旋转而移动时，所有组成刚体的粒子会平行移动，因为刚体不会改变其外形。再者，当刚体转动时，它通常会绕着通过质心的轴旋转，除非刚体被链住而迫使它要绕着某一点旋转。这些事实使质心适合作为追踪刚体线性运动的点。这是个好消息，因为你可以运用所有学过的粒子运动学公式，来处理刚体运动学的问题。

处理刚体运动的程序包含两个不同方面：

- 追踪物体质心的移动
- 追踪物体的转动

第一个方面是旧的概念 —— 只要将物体视为粒子；然而第二个方面需要考虑更多因素，即坐标系、角位移、角速度及角加速度。

本章剩余的内容主要讨论刚体的平面运动学。平面运动简单地说就是物体运动限制在空间的平面上，物体转动的轴线是垂直于平面的。平面运动本质上是 2D 的运动。这让我们能专注于角位移、角速度及角加速度的新概念上，而不会迷失于描述任意 3D 空间转动的数学式之中。

你或许对可用平面运动学来解答的问题的数量感到惊讶。例如，某些热门的 3D 枪战游戏，你的人物可以在地板上推动诸如箱子或桶子等物体。虽然游戏世界是 3D 的，但这些特定的物体只能在地板（平面）上滑动，因此可视为 2D 的问题。即使人物推动这些物体是以某个角度而非直线的，也可以用 2D 运动学（动力学）来模拟这些物体的滑动及转动。

## 局部坐标轴

以前，定义笛卡儿坐标系作为固定的全局参考坐标或全局坐标。当处理粒子运动时，便需要这种全局坐标系。然而，就刚体而言，也要有一组固定在物体上的局部坐标。此局部坐标系将固定于物体质心的位置上。利用此坐标系，便可在物体转动时记录其方位。

而平面运动只需要一个标量来描述物体的方位，如图 2-8 所示。

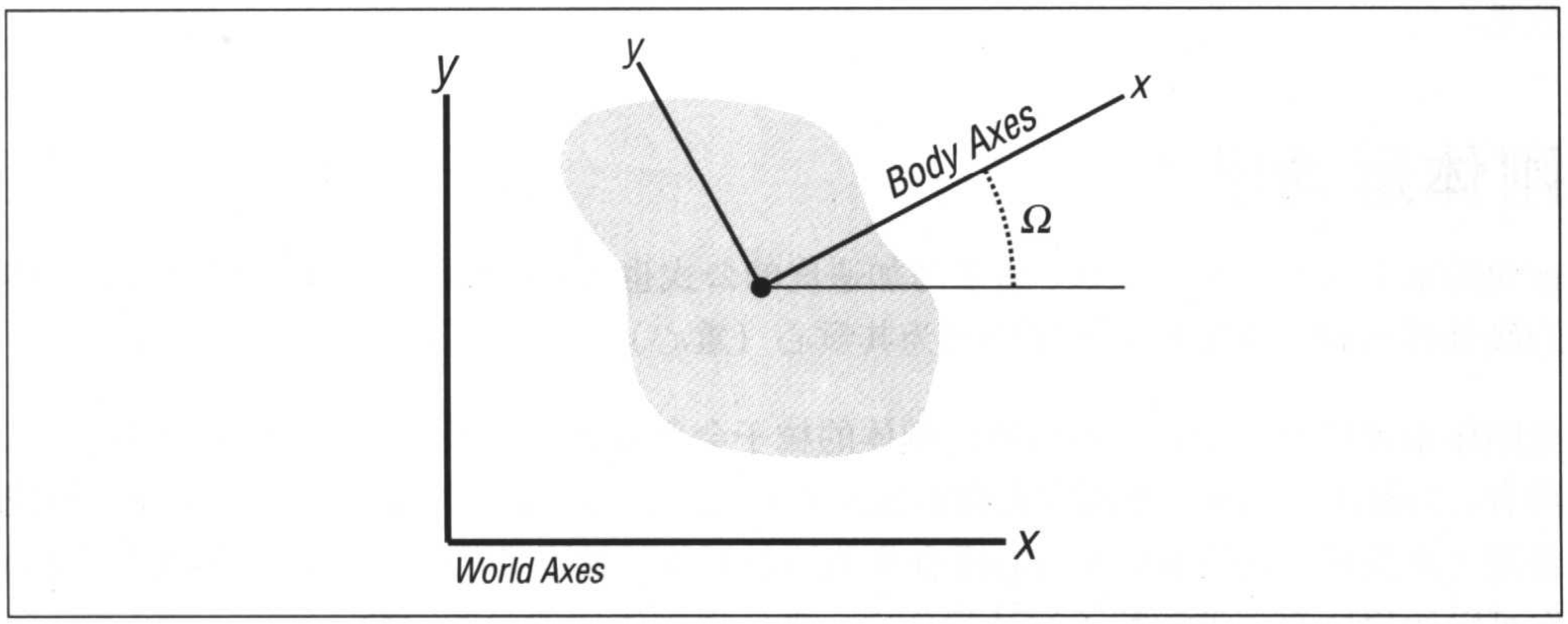



图 2-8：局部坐标轴

这里，方位$\Omega$定义为两组坐标轴的角度差：固定的全局坐标轴与局部物体坐标轴。这就是所谓的欧拉角（Euler angle）。一般的3D运动会有三个欧拉角，以气体动力学及流体动力学的术语来说，即偏转角（yaw）、俯仰角（pitch）和滚转角（roll）。从这些角的表示法很容易看出它们的物理意义，但从数字的观点在表示特殊角度时却会让人混淆，于是在编写3D实时模拟时便需要用另一种表示法。这些内容将在第十四章讨论。

## 角速度及加速度

在2D平面运动中，当物体转动时$\Omega$会改变，而$\Omega$的改变率就是角速度$\omega$。同样$\omega$的改变率就是角加速度$\alpha$。这些角的特性类似于线性的位移、速度和加速度。角位移、速度及加速度的单位分别是rad（弧度或径度）、rad/s、rad/s$^2$。

数学上，可以将角位移、角速度及角加速度之间的关系式写成：

$$\omega = d\Omega / dt$$
$$\alpha = d\omega / dt = d^2\Omega / dt^2$$
$$\omega = \int \alpha \, dt$$
$$\Omega = \int \omega \, dt$$
$$\omega \, d\omega = \alpha \, d\Omega$$

事实上，这些角的特性$\Omega$，$\omega$，$\alpha$可取代之前导出的粒子运动学方程中的$s$，$v$，$a$，得出类似的转动运动学方程。就常数角加速度而言，最后的方程如下：

$$\omega_2 = \omega_1 + \alpha t$$
$$\omega_2^2 = \omega_1^2 + 2\alpha(\Omega_2 - \Omega_1)$$
$$\Omega_2 = \Omega_1 + \omega_1 t + (1/2)\alpha t^2$$

当刚体绕一主轴转动时，刚体上的每一点会扫出绕着转动轴的圆形路径。可以将物体的转动假想成导致物体上粒子的额外线性运动。此线性运动与物体质心的线性运动不同。要想求出任一粒子或刚体上的点的全体线性运动，需找出物体的角运动与粒子（点）绕转动轴的线性运动的关系式。

在告诉你如何做到之前，先解释为什么还要做此类的计算。在动力学的范畴中，基本得知两物体碰撞是不够的，通常还需要知道碰撞有多么猛烈。当处理刚体间的交互作用时，有些点可能会与其他的点或与其他固体碰触，你不只需要判断接触点的位置，还要判断接触点间的相对速度或加速度。这些信息能求出两碰撞物体间的相互作用力。

刚体上粒子所扫过的弧长为转动轴到粒子的距离及角位移 $\Omega$ 的函数。用 $c$ 表示弧长，$r$ 表示转动轴到粒子的距离，如图 2-9 所示。弧长与角位的关系式为：

$$c = r\Omega$$

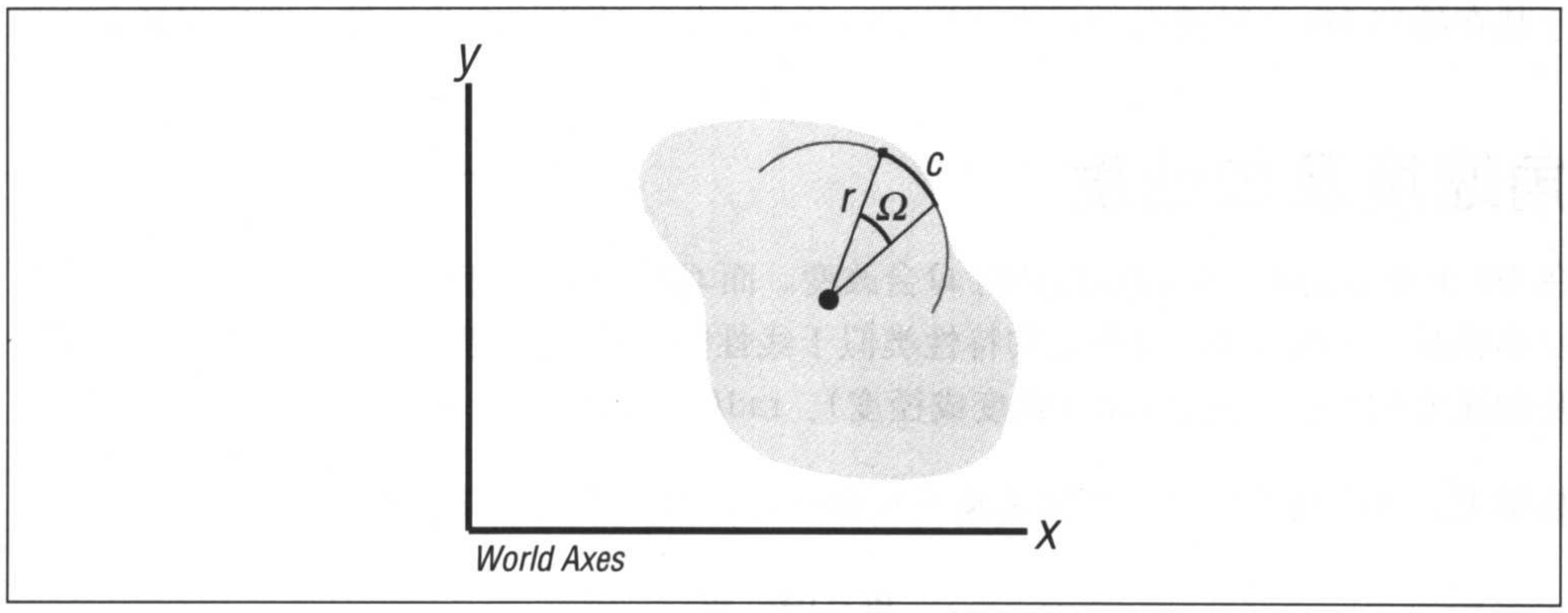



图 2-9：组成刚体的粒子的圆形路径

式中，$\Omega$ 的单位为弧度而非角度。若将此公式对时间微分：

$$\mathrm{d}c/\mathrm{d}t = r\ \mathrm{d}\Omega/\mathrm{d}t$$

得到当粒子沿其圆形路径移动时的线速度与刚体角速度相关的方程。该方程如下：

$$v = r\omega$$

此速度的向量与粒子扫过的圆形路径相切。你可以把此粒子想像为一颗球系于杆子的一端，而另一端固定为转动轴，在球转动时若将它从杆子的一端松掉，球将以与其系于杆子时所绕的圆形路径相切的方向飞出。这与上列等式给定的切线速度的方向相同。图 2-10 说明了该切线速度。

将方程 $v = r\omega$ 微分：

$$\mathrm{d}v/\mathrm{d}t = r\ \mathrm{d}\omega/\mathrm{d}t$$

得到切线加速度为角加速度的函数的公式：

$$a_t = r\alpha$$

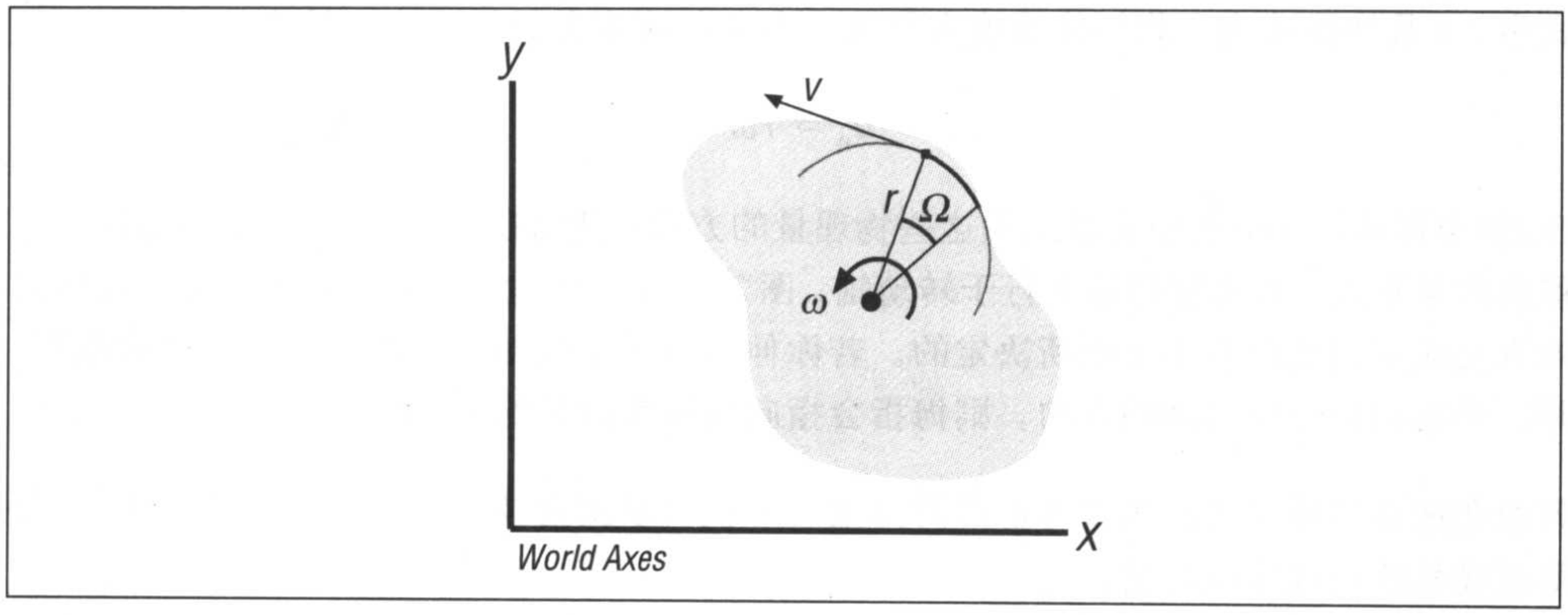



图 2-10：由角速度产生的切线速度

注意，另有一个刚体转动所导致的粒子加速度分量。此分量垂直于粒子的圆形路径，称为向心加速度，其方向永远指向转动轴（如图 2-11 所示）。记住，速度是一个向量且因为加速度是速度向量的变化率，所以产生加速度的方式有两种。其一，速度向量大小的变化，即速率的变化；其二，速度向量方向的变化。由速率的变化可求得切线加速度分量，而由方向的变化可求得向心加速度的分量。合成的加速度向量为切线和向心加速度的向量总和。当开车急转弯时所感觉到的（即使车速是常数），即向心加速度。

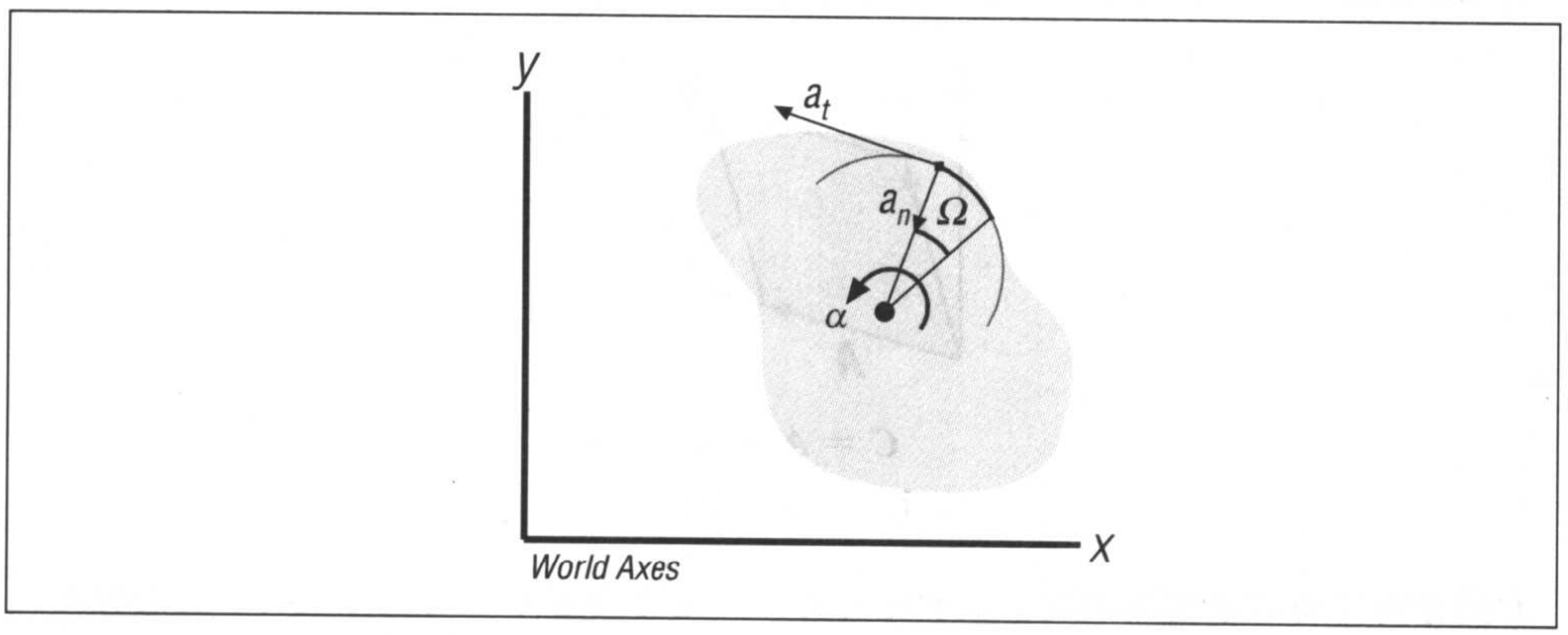



图 2-11：切线和向心加速度

描述向心加速度 $a_n$ 大小的公式为：

$$a_n = v^2/r$$

式中，$v$是切线速度。将切线速度方程代入向心加速度方程得到下列替代的形式：

$$a_n = r\omega^2$$

在2D空间中，可以很轻易地运用这些物理量的方程；然而在3D空间中，需利用这些方程的向量形式。角速度向量平行于转动轴，图2-10中，角速度的方向指出页面。其转动的意义或方向是由右手定则所决定的。若你伸出右手并卷曲手指成环绕转动轴的弧形，并让手指指向物体转动的方向，则拇指会指向角速度向量的方向。

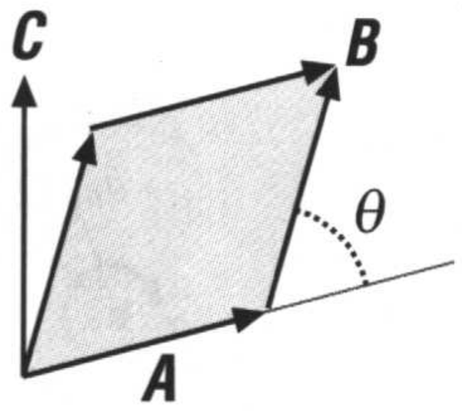
若取角速度向量与转动轴至考虑点的向量的外积（参阅附录一复习一下向量数学），会得到线性的切线速度向量：

$$\mathbf{v} = \omega \times \boldsymbol{r}$$

注意，此方程可求出切线速度的大小与方向。并且当取外积时需维持向量的顺序，即$\omega \times \boldsymbol{r}$，而不能是其他顺序，否则会得到错误的$\boldsymbol{v}$方向。

## 向量外积

对于给定的任意两个向量$\boldsymbol{A}$和$\boldsymbol{B}$，$\boldsymbol{A} \times \boldsymbol{B}$的外积由第三个向量$\boldsymbol{C}$来定义，其大小等于$AB\sin\theta$，其中$\theta$是$\boldsymbol{A}$、$\boldsymbol{B}$两向量的夹角，如下图所示。




$$\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$$

$$\boldsymbol{C} = AB\sin\theta$$

$\boldsymbol{C}$的方向可由右手定则判定。右手定则是协助追踪向量方向的简易技巧。假设将$\boldsymbol{A}$和$\boldsymbol{B}$置于平面，并使转动轴垂直于平面且通过$\boldsymbol{A}$尾端的点。再以右手由$\boldsymbol{A}$至$\boldsymbol{B}$沿着转动轴卷曲手指。当保持手指卷曲时，现在伸出拇指，如同你翘起拇指表示赞许。拇指所指的方向便是向量$\boldsymbol{C}$的方向。

上图中，$\boldsymbol{A}$和$\boldsymbol{B}$构成平行四边形（阴影区域）。此平行四边形的面积即为$\boldsymbol{C}$的大小，等于$AB\sin\theta$。

为求得切线及向心加速度向量，需要两个方程：

$$\boldsymbol{a}_{\mathrm{n}} = \omega \times (\omega \times \boldsymbol{r})$$

$$\boldsymbol{a}_{\mathrm{t}} = \alpha \times \boldsymbol{r}$$

以另一种方式来看$\boldsymbol{v}$，$\boldsymbol{a}_{\mathrm{n}}$，$\boldsymbol{a}_{\mathrm{t}}$，它们是粒子在刚体上相对于其转动所绕的点（例如物体质心的位置）的速度及加速度。这样是非常方便的，因为正如之前提过的，要将运动中的刚体宏观地视为粒子，不需一直顾虑那些组成刚体的粒子在做什么。因此，要分别处理刚体的线性运动与角运动。当需要微观地近看刚体上特定的粒子或点时，同样可以把刚体的运动视为粒子，然后再加入考虑点的相对运动。

图 2-12 显示的是以速率$\boldsymbol{v}_{\mathrm{cg}}$行进的刚体，其中$\boldsymbol{v}_{\mathrm{cg}}$是刚体质心（或重心）的速率。记住，质心是将刚体视为粒子时所追踪的点。此刚体也绕着通过质心的轴以角速度$\omega$转动。$\boldsymbol{r}$是刚体质心到刚体上某一点$P$的向量。

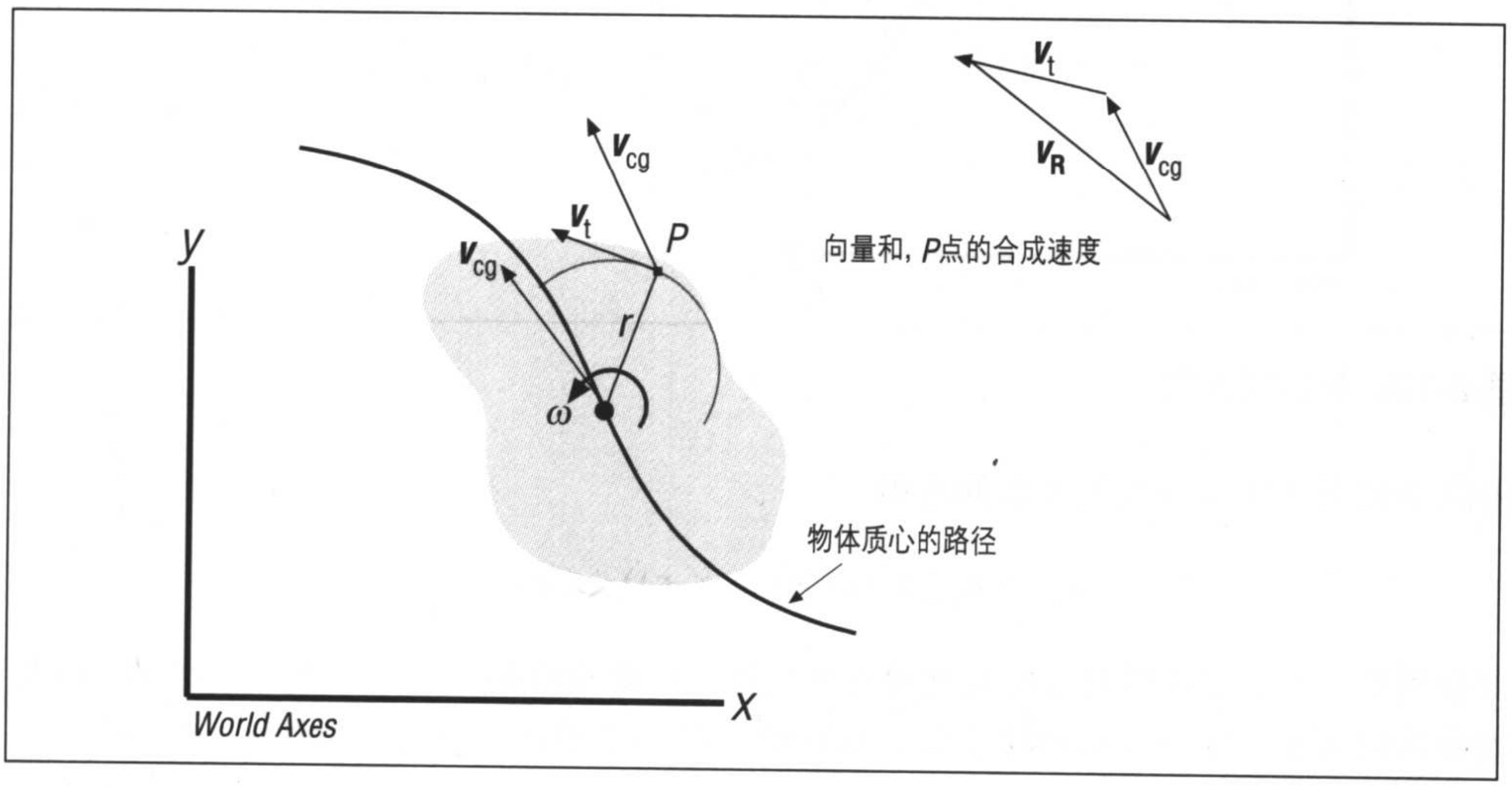



图 2-12：相对速度

在这个例子中，$P$点的合成速度可由质心速度与$P$点的切线速度（由物体的角速度$\omega$产生）的向量总和求得。此向量方程如下：

$$\boldsymbol{v}_{\mathrm{R}} = \boldsymbol{v}_{\mathrm{cg}} + \boldsymbol{v}_{\mathrm{t}}$$

或者

$$\boldsymbol{v}_{\mathrm{R}} = \boldsymbol{v}_{\mathrm{cg}} + (\omega \times \boldsymbol{r})$$

可用相同的方法求出$P$点的合成加速度，如图2-13所示。这里，需取刚体质心加速度与切线加速度（由物体角加速度产生），以及向心加速度（切线速度方向的改变所导致）三者的向量总和。其方程如下：

$$\boldsymbol{a}_{\mathrm{R}} = \boldsymbol{a}_{\mathrm{cg}} + \boldsymbol{a}_{\mathrm{n}} + \boldsymbol{a}_{\mathrm{t}}$$

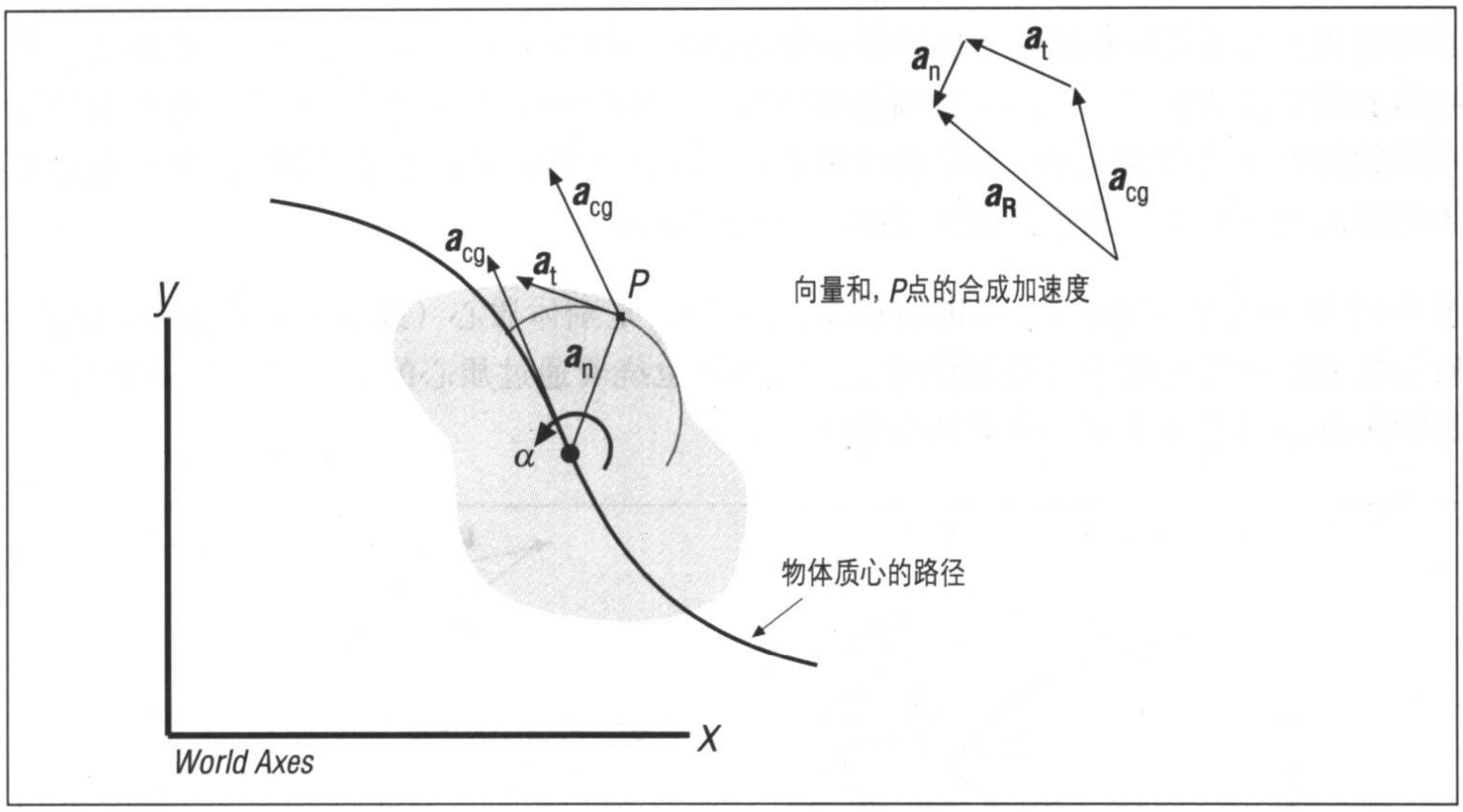



图2-13：相对加速度

能以下列形式重写合成加速度的方程：

$$\boldsymbol{a}_{\mathrm{R}} = \boldsymbol{a}_{\mathrm{cg}} + (\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{r})) + (\boldsymbol{\alpha} \times \boldsymbol{r})$$

如你所见，利用这些相对速度及加速度的定理，只要知道物体质心在做什么运动，以及物体如何转动，便可求得刚体上任意点在给定时间下的合成运动学特性。

# 第三章

# 作用力

本章是在学习第四章〈动力学〉之前的必修课题，旨在为读者提供足够的力学背景知识，让读者能轻易地进入动力学的领域。本章并不是最后一个讨论力学的章节。事实上，在实际模拟的过程中，你将发现力学的观念是十分重要的。本书后续章节的讨论中，也常会再提到本章所说明的主题。本章将讨论两种基本的作用力，并简单地介绍一些特殊类型的力。同时也会解释力与力矩之间的关系。

## 简介

如同第二章结尾所提到的，在了解动力学之前必须先明白力的概念，运动学仅是动力学的一部分。在日常的经验中，你已经非常熟悉力的概念了：在阅读本书时，你的手施力在这本书上，以抵抗重力；在移动鼠标时，则必须施力于鼠标；在踢足球时，必须通过脚施力于足球上。一般来说，力可以让物体移动，或者更精确地说，力会改变物体的加速度。拿着这本书时，虽然书并没有在动，但事实上你已经改变它的加速度使之为零了。踢足球时，从原本静止的球到球离开脚飞出去期间，其加速度已从一开始的零，变成最后的某个正值。这些是外部施力的“接触力”（contact force）的例子。

除了接触力之外，还有另一种常见的力“场力”（field force），或称为“非接触力”（force-at-a-distance force）。这种力不需实际接触物体，便能在该物体上产生作用力。物体之间的万有引力便是一个很好的例子。另外存在于电荷粒子之间的电磁吸引力也属于这一类。力场这个概念在很久以前就已存在，用来解释有距离的物体间所产生的作用力。可以说一个物体是受到另一个物体的重力场所作用的。用力场的概念可以帮助你了解物体为什么可以不通过实际的接触，就能够对另一物体施力的现象。

在这两大范畴中，有一些特定的力会与各种自然现象有关，例如摩擦所产生的力、浮力，以及压力等。本章会讨论在理想状况下这些力的一些特性，在这本书接下来的部分，将会从更实际的观点来探讨这些力。

在继续介绍之前，必须再解释第一章提及的牛顿第三运动定律的含义：作用在一个物体上的所有的力，都会有一个量值相等，而方向相反的反作用力同时产生。这表示力的存在是成对出现的，力本身并不会单独地存在。

想想在地球与你之间的重力（万有引力）。地球会施力在你身上（于是你有了体重），使你的加速度朝向地心。同样地，你也会施力于地球上，使其加速度朝向你自己。由于地球跟你的质量差异极大，于是地球产生的加速度非常小，小到可以忽略。之前提到当你拿着一本书时，必须施一个力在该书上；同样地，这本书也会施一个大小相等方向相反的力在你的手上，这个反作用力就是你感觉到的书本重量。

这个作用与反作用的现象，就是火箭推进的基本原理。火箭引擎会使燃料的分子受到力的作用产生加速度而从排气口喷出。而施于这些分子使之加速的力，也会同时产生作用于火箭上的反作用力，即推力（thrust）。在这本书接下来的部分，会有许多作用与反作用的例子，这在刚体动力学上是很重要的现象，尤其在稍后讨论物体的接触与碰撞的时候尤为重要。

# 力场

要解释力场或非接触力，最好的例子就是物体之间的万有引力。牛顿万有引力定律为：两物体之间的引力与这两个物体的质量成正比，且与两个物体质心距离的平方成反比，同时，该引力的作用线就是两物体质心的连线。万有引力的公式如下：

$$F_a=(Gm_1m_2)/r^2$$

其中G是重力常数即牛顿所称的万有引力常数。Henry Cavendish于1798年在实验室中首度测量出它的值，以国际单位表示为$6.673 \times 10^{-11}(N\text{-}m^2)/kg^2$，或以英制单位表示为$3.436 \times 10^{-8}\ ft^4/(lb\text{-}s^4)$。

到目前为止，所用重力加速度g的值是常数9.8 $m/s^2$（32.174 $ft/s^2$），这是物体靠近地表（如海平面）的重力加速度。事实上，g的值会随着海拔高度而变。考虑靠近地表的物体，受地心引力作用而产生了加速度（牛顿第二运动定律），将万有引力公式代入牛顿第二运动定律，得到：

$$ma=(GM_em)/(R_e+h)^2$$

其中 $m$ 是物体的质量，$a$ 是因物体与地球间的地心引力而产生的加速度（重力加速度），$M_e$ 是地球的质量，$R_e$ 是地球的半径，而 $h$ 是物体的海拔高度。解出 $a$ 的方程，可以得到重力加速度公式：

$$a = g' = (GM_e)/(R_e + h)^2$$

地球的半径大约是 $6.38 \times 10^6$ m，质量约为 $5.98 \times 10^{24}$ kg。将这些值代入以上的公式中，并假设物体的海拔高度为0（海平面上），就可以计算出至今一直使用的常数g的值。即海平面上的 g 值为 9.8 $m/s^2$。

# 摩擦力

当物体在运动时，会与接触面彼此交互作用而产生摩擦力。摩擦力也是接触力的一种。摩擦力的方向为接触点上平行于接触面，也就是正切于接触面的方向。而摩擦力的大小则是接触面间的正压力（normal force）与表面粗糙程度（摩擦系数）的函数。

参考如图 3-1 中一个水平表面上的木块，能比较容易想像各个力之间的关系。

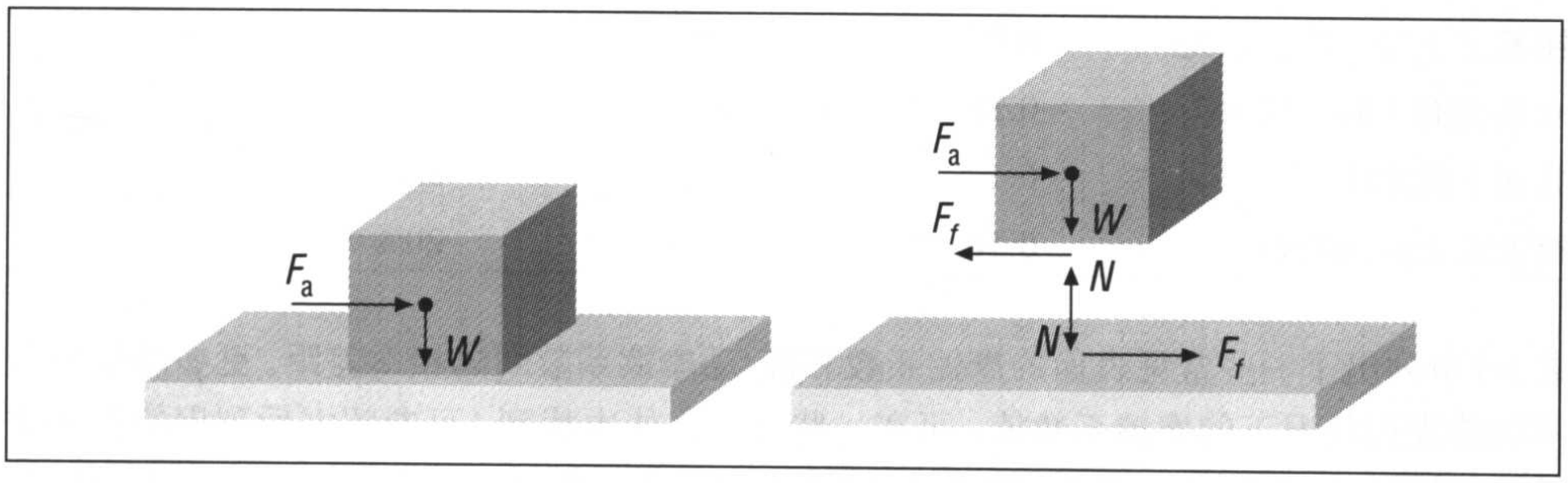



**图 3-1：摩擦力，与水平表面接触的木块**

图中的木块静置在一个水平的表面上，木块本身受一个很小的力 $F_a$，其作用线通过木块的质心。当作用力增加时，木块跟水平表面之间就会产生摩擦力来阻止木块的移动。这个摩擦力最大值为：

$$F_{f\max} = \mu_s N$$

其中的 $\mu_s$ 是由实验所得的静摩擦系数（coefficient of static friction）（注 1）；而 $N$ 是木块与平面之间的正压力，在目前的状况下 $N$ 的量值等于木块的重量。当作用力增加但

---

注 1：　静摩擦系数的“静”表示当时并没有运动发生；木块所受的所有力使其处于平衡状态。

仍然小于$F_{f\max}$时，木块依然静止不动，而此时的$F_f$等于作用力的大小（木块处于静平衡中）。当作用力大于$F_{f\max}$时，摩擦力就不能再妨碍木块的运动，因此木块在作用力的影响下就产生加速度。就在木块产生运动之后，摩擦力会由$F_{f\max}$减小至$F_{fk}$，其中的$F_{fk}$为：

$$F_{fk} = \mu_k N$$

这里的k表示“动的”（kinetic），因为此时木块正在运动中，而$\mu_k$就是动摩擦系数（coefficient of kinetic friction）（注2），它会小于$\mu_s$。与静摩擦系数一样，动摩擦系数也是由实验决定的。表3-1列出了在某些接触面上典型的摩擦系数。

表3-1：一般表面的摩擦系数

| 表面状况 | $\mu_s$ | $\mu_k$ | 差距% |
|---|---|---|---|
| 玻璃上的干燥玻璃 | 0.94 | 0.4 | 54% |
| 铁表面上的干燥铁 | 1.1 | 0.15 | 86% |
| 路面上的干燥橡胶 | 0.55 | 0.4 | 27% |
| 钢表面上的干燥钢 | 0.78 | 0.42 | 46% |
| 铁氟龙上的干燥铁氟龙 | 0.04 | 0.04 | — |
| 木头表面上的干燥木头 | 0.38 | 0.2 | 47% |
| 冰面上的冰块 | 0.1 | 0.03 | 70% |
| 钢表面上涂油的钢 | 0.10 | 0.08 | 20% |

表3-1中列出了一些常见表面的摩擦系数和静/动摩擦系数间的相对差距。更多的资料与不同的表面状况下的摩擦系数值，可在一些技术文件中找到（可参阅书后列出的参考文献）。另外要注意的是，实验所得的摩擦系数即使是在相同的表面下，其值也不一定会完全相同，还要考虑表面的材质及实验的技巧等特定条件。

## 流体动阻力

流体动阻力与摩擦力一样会阻挡物体的运动。事实上，流体动阻力的主要分力就是摩擦力，是由流体流经（并接触）物体表面的相对分流而产生的。但是摩擦力并不是流体动阻力惟一的分力。根据物体的外形、速度和流体的性质，当流体流经物体表面时，会因为不同的压力分布而产生不同的额外分力（压差阻力）。举例来说，当物体位于两种流

注2：　有时候我们也会用*dynamic*代替*kinetic*，两者同义。

体的交界面上时（例如在水面上的船，处在两种流体 —— 水和空气之间），会因为水波的产生而导致额外的阻力分力。

一般而言，流体动阻力是一种复杂的现象，是由许多变因所构成的函数。在本节中不会详细探讨所有的变因，因为后面几章将会再次讨论这个主题。然而，本节会介绍如何计算阻力中的黏滞（摩擦）阻力。

理想的黏滞阻力是物体速度与由实验得出的阻力系数（drag coefficient）的函数，其中的阻力系数取决于物体表面的状况、流体属性（密度和黏滞力）和流动状态。基本上计算黏滞阻力的公式可以写成这样：

$$F_v = -C_f v$$

其中的$C_f$是阻力系数，$v$是物体的速率，而负号则表示此力阻碍物体的运动。这个公式适用于在粘性流体中缓慢运动的物体。这里的缓慢运动表示物体周围的流动状态是属于层流（laminar flow），即表示流体的流线是不受扰动并且互相平行的。

对于快速移动的物体，可利用以下这个速度平方的函数来计算$F_v$：

$$F_v = -C_f v^2$$

快速移动表示物体附近流体的流动产生了湍流（turbulent flow）（译注 1），即表示流体的流线不再是平行的而是对于物体附近的流动产生混合效应。请注意这两个公式的$C_f$的值并不会相同。除了之前提到的因素外，值得注意的是$C_f$会依据流体的流动是层流还是湍流而定。

这些方程都是简化过的，并不适用于实际流体问题的分析。然而对于电脑游戏的模拟却有一定的好处。最明显的优点是，这些公式都很容易实现，只需要知道目前计算物体的速度（可以由运动学方程求得）和阻力系数的假设值即可。模拟程序所要模拟的物体通常都会有不同的大小和外型，使严谨的阻力特性分析变得不实际，此时使用这个公式就很方便。如果只是要模拟而不是要确实的精确度，使用这些公式将会是正确的选择。

使用这些理想化公式的另一个好处是，在计算运动方程时可以调整阻力系数来防止数值不稳定的情况发生，还能维持一定的真实度。如果想要有真实世界的精确度，惟一的选择是考虑使用更周全（也就是更复杂）的方法来计算流体动阻力。在第六章到第十章，将会更多地讨论关于阻力的议题。

---

译注 1： 湍流为流体随机的旋涡运动，又称紊流。

# 压强

很多人会把压强和力混为一谈。经常听见有人在说明一个现象时会用“以每平方英寸100磅的力推开”这种说法。虽然可以知道他要表达的意思，但学术上的说法应该是压强，而不是力。压强是每单位面积所受的力，因此其单位为磅每平方英寸（psi）或磅每平方英尺（psf）等。如果已知压强，则必须知道受到该压强的总面积，才能得到受力的大小。力等于压强乘以面积：

$$F = PA$$

由此公式得知，当压强不变时，作用的接触面积越大，所产生的力也越大。如果将此方程改写成计算压强，则可发现压强跟接触面积成反比；也就是说施固定的力于物体上时，接触面积越大，则所受的压强越小，反之亦然：

$$P = F/A$$

压强的重要特点为，其作用力都垂直于该物体的表面。这一点暗示了合力向量的方向。

这里会提及压强的原因是，当本书有些章节讨论小船、船舰或气垫船的力学时，会通过压强计算其所受的力。这里所使用的压力为流体静压力（即浮力）和空气静升力（aerostatic lift）。接下来简单地介绍一下浮力。

# 浮力

当你浸在浴缸中时，可以感觉到浮力作用在你身上。这就是为什么在水中时会觉得身体比在空气中较轻，以及为什么有些人在游泳池中可以浮起来的原因。

浮力是物体浸在流体中所产生的力，为物体体积跟流体密度的函数，也就是物体顶端的流体跟物体底部的流体所产生的压力差。压力越大，表示物体在水中的深度越深。所以对固定高度的物体而言，其底部的压力会比其顶端的压力大。请参考图3-2中的立方体。

假设$s$代表立方体的长、宽、高（对立方体而言这三者相等）。$h_t$表示立方体顶端的水深，而$h_b$是立方体底部的水深。立方体顶端的压强是$P_t = \rho g h_t$，作用在整个立方体顶端平面上，与平面垂直且方向向下。而立方体底部的压强是$P_b = \rho g h_b$，作用在整个立方体底部平面上，与平面垂直且方向向上。请注意立方体侧面平面所受的压强与沉入水中的深度呈线性递增，范围由$P_t$到$P_b$。另外也要注意，因为侧压强是互相对称，其量值相同，而且方向相反，所以侧面净压强为0，这就表示侧面的净力（压力产生的力）为0。但是对于立方体上下的压强就不是如此了，因为它们虽然方向相反，但是量值却不同。

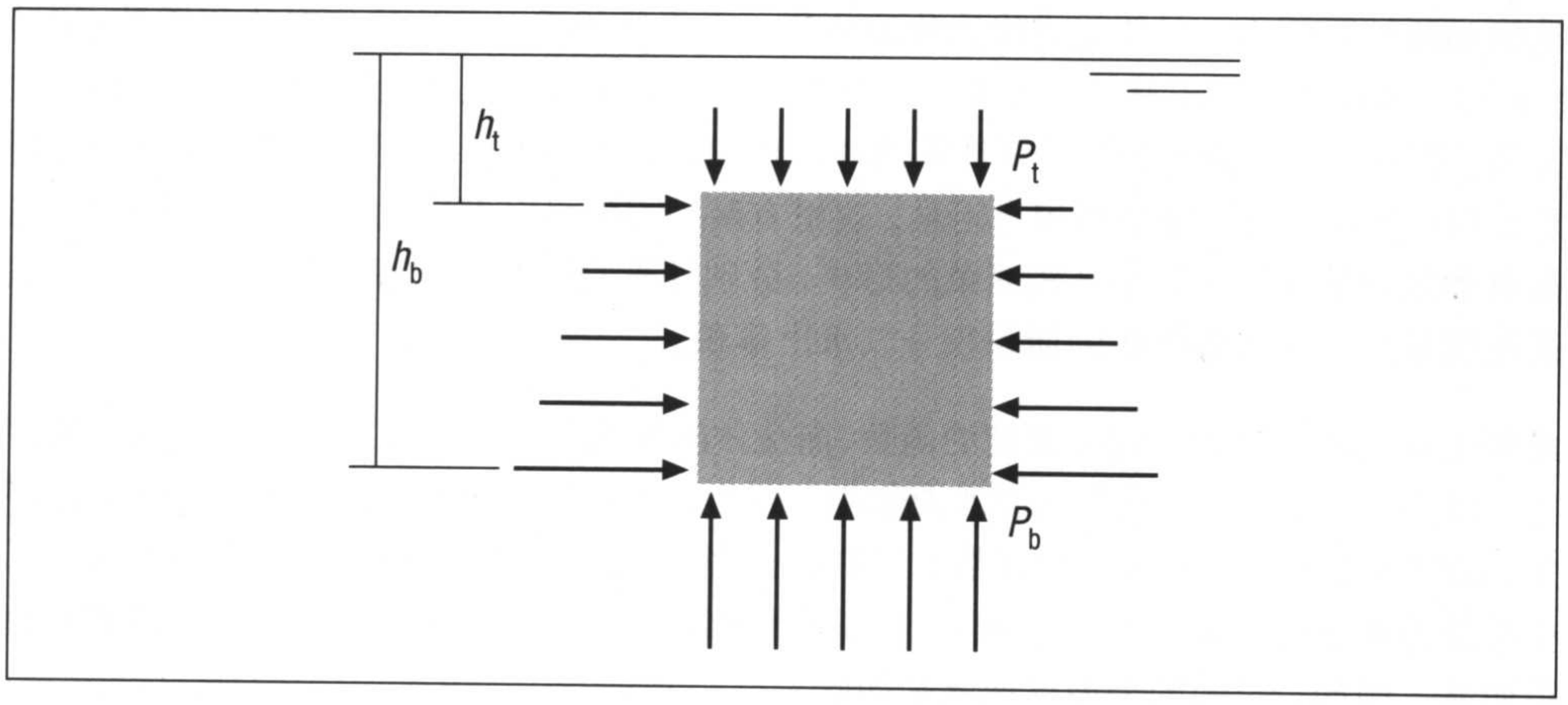



图 3-2：沉入流体的立方体

往下作用在立方体顶端平面的力，等于该立方体顶端所受的压力与其顶端面积的乘积，可表示如下：

$$F_t = P_t A_t$$

$$F_t = (\rho g h_t)(s^2)$$

同样，作用在立方体底部的力，就相当于底部所受的压力与其底部面积的乘积：

$$F_b = P_b A_b$$

$$F_b = (\rho g h_b)(s^2)$$

所以垂直净力（浮力）就是该立方体顶端与底部受力的差：

$$F_B = F_b - F_t$$

$$F_B = (\rho g h_b)(s^2) - (\rho g h_t)(s^2)$$

$$F_B = (\rho g)(s^2)(h_b - h_t)$$

通过此公式能得到浮力的值，其方向为向上，抵消该物体的重量。

在此能观察出一个很重要的现象。我们可以发现（$h_b - h_t$）就是立方体的高度，本范例中即为 $s$。将$(h_b - h_t)$代换为 $s$ 便可得出浮力是立方体体积的函数：

$$F_B = (\rho g)(s^3)$$

这表示要计算浮力，只要先求出物体的体积，再将体积乘以流体的相对体积质量（$\rho g$）（注 3）。事实上，除了简单的几何形状外，大部分状况都是说比做容易。对于球体、立方体、圆柱体或类似的形状，要计算物体的体积是很容易的。然而，遇到不规则的几何形状时，体积的计算就变得相当困难。这里有两种方法可以解决这个问题。第一种方法是将物体分解成许多较小且较简单的形状，分别计算它们的体积再加起来。第二种方法是使用数值积分法将物体的面积积分以求出体积。

此外还必须注意浮力是流体密度的函数，而流体的密度并不需要像水那样大才能提供浮力。事实上，你的身上现在也有浮力作用。因为你正被空气包围，所以会由空气产生浮力，虽然它非常小。而水的密度是空气的好几倍，这也是在水中感受到的浮力要比空气中的浮力明显的原因。然而请记住，对于体积相对较大的轻物体，在空气中的浮力就会很明显。例如，空中的大型气球就是如此。

## 弹簧和阻尼

弹簧是一种结构上的单元，当它连接两个物体时，会有量值相同而方向相反的两个力作用在两个物体上。根据虎克定律，弹力为弹簧由原先静止时的长度、因受力而伸长或缩短的长度与弹性系数的函数。弹性系数是与弹簧受力而变形有关的值：

$$F_s = k_s(L - r)$$

其中的$F_s$是弹力，$k_s$是弹性系数，$L$是弹簧伸长或压缩后的长度，而$r$是弹簧静止时的长度。$F_s$的国际单位是 N（1 N = 1 kg-m/s$^2$），而$L$，$r$的单位是 m，而$k$的单位则是 kg/s$^2$。如果一个弹簧连接两个物体，它会在一端施力$F_s$，而在另一端上则施力$-F_s$。两者的量值相同，但方向则不同。

在数值模拟中，阻尼（damper）常常与弹簧合并使用。阻尼的作用与黏滞阻力相同，都抵消速度。在这种状况下，当阻尼连接两个正在靠近或远离的物体时，阻尼会试着减慢两物体之间的相对速度。由阻尼所发出的力与两连接物体之间的相对速度和阻尼常数$k_d$成正比，相对速度和阻尼力（damping force）的关系式如下：

$$F_d = k_d(v_1 - v_2)$$

这个公式显示阻尼力$F_d$是阻尼常数和两连接物体上连接点的相对速度的函数。阻尼力的国际单位是 N，而速度的单位是 m/s，$k_d$的单位是 kg/s。

---

注 3：　相对体积质量（specific weight）是密度乘以重力加速度。单位一般是 lb/ft$^3$ 或 N/m$^3$。

基本上，弹簧与阻尼通常组合为单一的弹簧/阻尼单元，因此用一个公式就能计算合并后的力。用向量表示时，连接两物体的弹簧/阻尼单元的公式看起来像这样：

$$\boldsymbol{F}_1 = -\{k_s(L - r) + k_d[(\boldsymbol{v}_1 - \boldsymbol{v}_2)\cdot \boldsymbol{L}]/L\}\boldsymbol{L}/L$$

这里的$\boldsymbol{F}_1$是物体1所受的力，而物体2所受的力$\boldsymbol{F}_2$为：

$$\boldsymbol{F}_2 = -\boldsymbol{F}_1$$

$\boldsymbol{L}$是弹簧/阻尼单元的长度向量（非粗体字的$L$表示向量$\boldsymbol{L}$的量值），即物体1和物体2间连接点的坐标差向量。如果所连接的两物体为粒子，则$\boldsymbol{L}$等于物体1的位置减去物体2的位置。同理，$\boldsymbol{v}_1$和$\boldsymbol{v}_2$是物体1和物体2上连接点的速度。而$(\boldsymbol{v}_1 - \boldsymbol{v}_2)$的量值表示两连接物体之间的相对速度。

在模拟许多连接在一起的粒子或刚体时，使用弹簧和阻尼是很方便的。弹力可以提供结构力，或是附着力，让物体间可紧密结合（或是可以保持一定的距离）。而阻尼可以让连接的物体之间柔顺，使其不至于看起来太紧绷或太有弹性。以数值稳定的观点来看，阻尼也是很重要的，可以避免数值爆炸的情形。在第十七章会介绍在柔体（相对于刚体而言）的实时模拟程序中如何应用弹簧/阻尼单元。

## 力与力矩

在此先说明力与力矩（torque）的区别（注4）。力会产生线性加速度，而力矩则会产生旋转加速度。力矩是力乘以距离。更具体地说，要计算作用在物体上的力矩，必须先求得旋转轴到施力方向的垂直距离，再乘以施力的大小。力的单位一般是磅、牛顿。而由于力矩是力跟距离的乘积，因此其单位是长度单位乘以力的单位，如ft-lb，N-m。

力和力矩都属于向量，所以也必须判断力矩的方向。力的向量很容易想像：力的作用线会通过作用点，而其方向就是施力的方向。而力矩的向量，其作用线会沿着旋转轴，而它的方向可由旋转方向及右手定则（righthand rule）判定（如图3-3所示）。右手定则是用来帮助判定向量方向的小技巧——在这里可决定力矩的方向。方法为：伸出右手并假装要合起手掌，将指尖顺着旋转方向使指尖指向的方向跟掌心垂直，然后伸出大拇指使其向上指。这时大拇指所指的方向就是力矩向量的方向。注意力矩的向量和施力的向量互相垂直，如图3-3所示。

---

注4：　力矩（torque或moment）也有人称为转矩或扭力。

前面曾提到，力矩的大小是施力的大小乘以力臂的长度，力臂指的就是旋转轴与施力方向的垂直距离。在2D空间下的计算很简单，垂直距离（图3-3中的$d$）可以立即算出来。

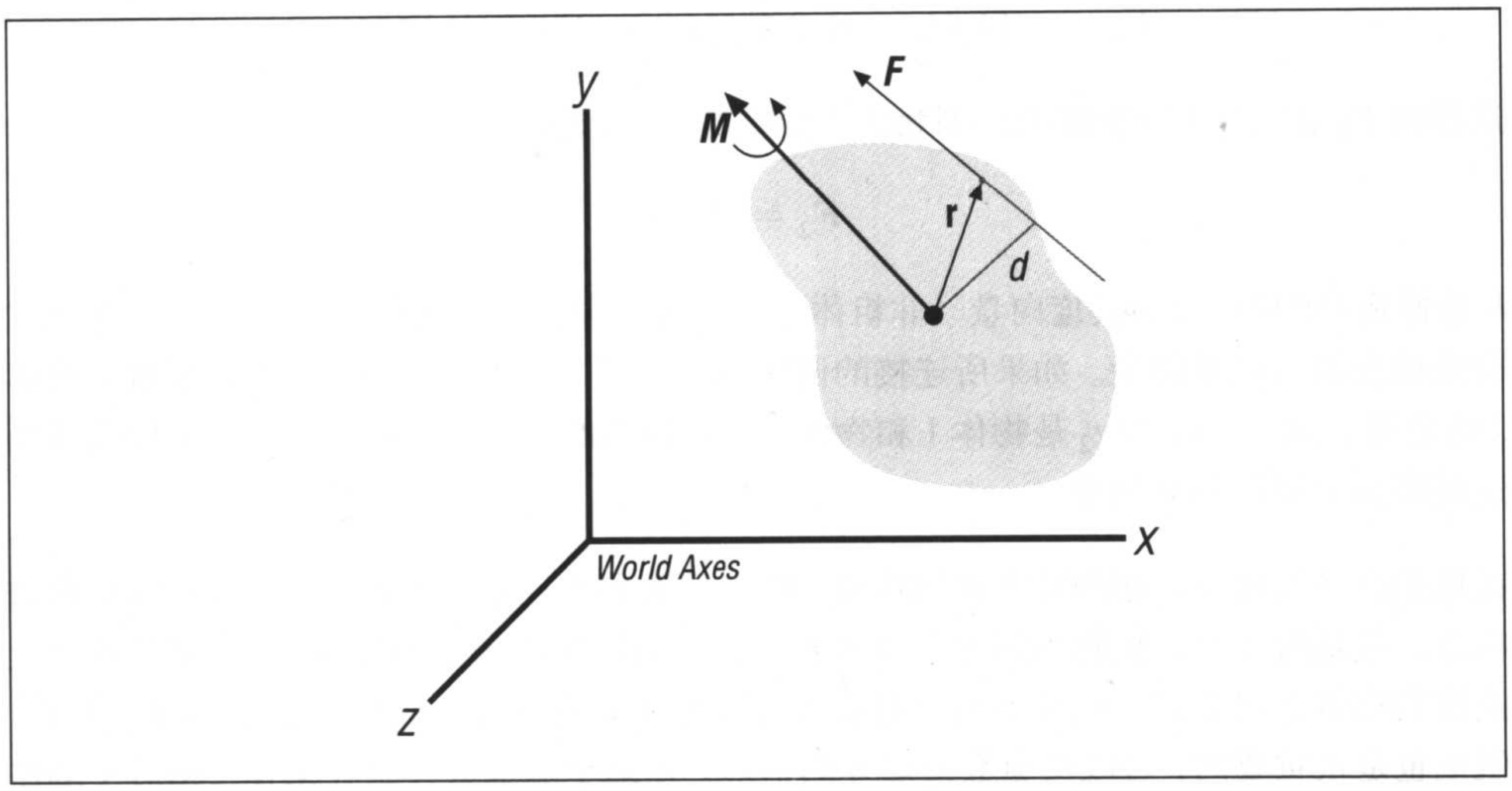



图3-3：力与力矩

但是，在3D空间下计算力矩时，不仅要知道力的向量，同时还要知道物体相对于旋转轴的受力点的坐标，可利用以下的公式来求得：

$$\boldsymbol{M} = \boldsymbol{r} \times \boldsymbol{F}$$

力矩$\boldsymbol{M}$是位置向量$\boldsymbol{r}$与施力向量$\boldsymbol{F}$的向量外积。

在直角坐标系统中，距离、施力与力矩的向量可以写成：

$$\boldsymbol{r} = x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

$$\boldsymbol{F} = F_x\boldsymbol{i} + F_y\boldsymbol{j} + F_z\boldsymbol{k}$$

$$\boldsymbol{M} = M_x\boldsymbol{i} + M_y\boldsymbol{j} + M_z\boldsymbol{k}$$

$\boldsymbol{r}$的标量部分（$x$，$y$，$z$）是旋转轴到施力$\mathbf{F}$的作用点的坐标距离。而力矩向量$\boldsymbol{M}$的标量部分则定义如下：

$$M_x = yF_z - zF_y$$

$$M_y = zF_x - xF_z$$

$$M_z = xF_y - yF_x$$

考虑如图3-4所示的刚体，受力$\boldsymbol{F}$的作用，但该力不是直接作用在该刚体的质心的情形。

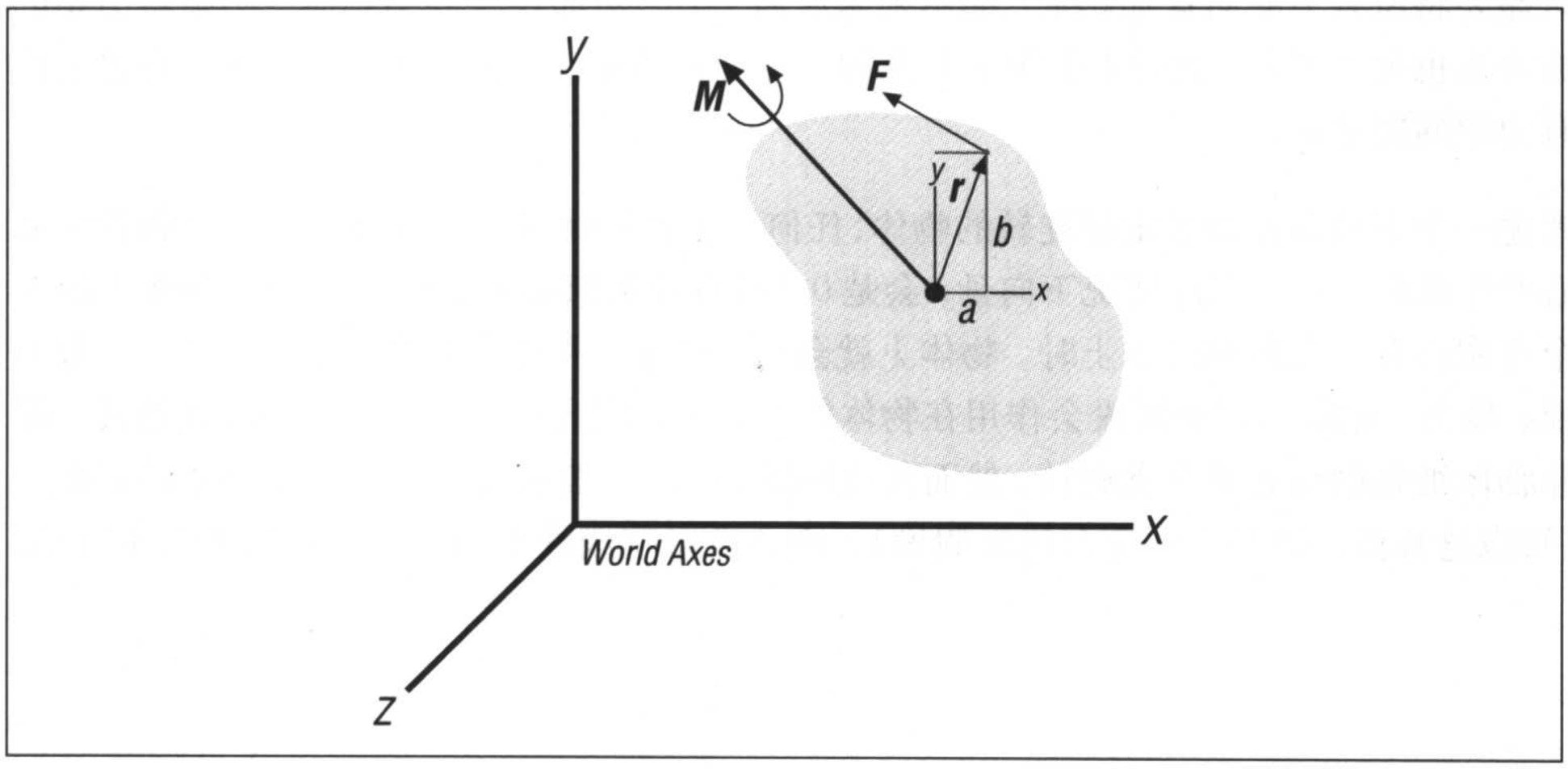



图3-4：力矩范例

在这个范例中，$\boldsymbol{F}$，$a$和$b$给定如下：

$$\boldsymbol{F} = (-90\ \text{lb})\boldsymbol{i} + (156\ \text{lb})\boldsymbol{j} + (0)\boldsymbol{k}$$

$$a = 0.66\ \text{ft}$$

$$b = 0.525\ \text{ft}$$

计算此力$\boldsymbol{F}$对于该物体质心所产生的力矩。

第一步是将$\boldsymbol{F}$对于物体质心的施力作用点的距离向量放在一起。因为已知局部坐标$a$和$b$，$\boldsymbol{r}$便是：

$$\boldsymbol{r} = (0.66\ \text{ft})\boldsymbol{i} + (0.525\ \text{ft})\boldsymbol{j} + (0)\boldsymbol{k}$$

现在使用公式$\boldsymbol{M} = \boldsymbol{r} \times \boldsymbol{F}$（或者是之前介绍的力矩分量公式），可得：

$$\boldsymbol{M} = [(0.66\ \text{ft})\boldsymbol{i} + (0.525\ \text{ft})\boldsymbol{j} + (0)\boldsymbol{k}] \times [(-90\ \text{lb})\boldsymbol{i} + (156\ \text{lb})\boldsymbol{j} + (0)\boldsymbol{k}]$$

$$\boldsymbol{M} = [(0.66\ \text{ft})(156\ \text{lb}) - (0.525\ \text{ft})(-90\ \text{lb})]\boldsymbol{k}$$

$$\boldsymbol{M} = (150.2\ \text{ft-lb})\boldsymbol{k}$$

请注意力矩向量的$x$分量和$y$分量皆为0，就表示力矩直接通过$z$轴。在本例中，力矩向量的方向为指向本书的上方。

在力学中，必须将作用于物体上所有力的合力与物体上的合力矩分开讨论。在合并力时，只需要将所有力的向量加起来，而不须考虑施力点。然而在合并力矩时，必须将力矩的施力点也列入考虑，然后才能像上个范例一样计算力矩。现在可以计算作用于物体上所有力矩的向量和。

考虑一个不被强制绕固定轴旋转的物体，任何作用在物体质心上的力并不会对物体的质心产生任何力矩，而此情况下向量$r$会是0（所有分量都是0）。当力作用在物体上的某个与质心有一段距离的点上时，物体上就会产生力矩，并且影响物体的角运动。一般来说，场力、超距力都被假设会作用在物体的质心上，因此只会影响物体的线性速度，除非物体被强制绕着某个点旋转。然而其他的接触力，一般来说不会作用在物体的质心上（可以这样做，但是不一定会这么假设），而会像影响线性运动一样影响物体的角运动。

# 第四章

# 动力学

回想一下动力学是对物体运动的研究，包含作用于物体上的力。现在结合前几章的题材即运动学、力学，来研究动力学的主题。和“运动学”这一章一样，我们将先讨论粒子动力学，然后再讨论刚体动力学。

在动力学上，最重要的方程就是牛顿第二运动定律：

$$\boldsymbol{F} = m\boldsymbol{a}$$

当牵涉到刚体时，必须要考虑到作用于物体的力除了会造成物体的移动，还会导致转动。其基本关系式如下：

$$\boldsymbol{M}_{\mathrm{cg}} = \boldsymbol{I}\alpha$$

其中 $\boldsymbol{M}_{\mathrm{cg}}$ 是作用于物体的力矩的向量总和，$\boldsymbol{I}$ 是物体的转动惯量，而 $\alpha$ 是角加速度。

这两个方程都同样归为运动方程。

你将碰到动力学的两类问题，一类问题是物体的加速度为已知或可用运动学轻易地判断，而必须求出作用于物体的力。另一类问题是作用于物体上的力为已知或可判断出来，而必须求出物体的合成加速度（接着求出其速度及位移）。显然，第二类问题更适用于游戏的物理，因此接下来主要讨论此类问题。

先在此强调，当解答动力学问题时，必须考虑所有作用于物体的力的总和。这包括所有的作用力及反作用力。除解答运动方程有其计算上的难度之外，动力学上的一个大挑战

就是辨别并适当地计算出所有的力。接下来的几章将探究涉及特殊作用力的特定问题。而现在就一般目的，先停留在上一章介绍的理想化的力。

解答动力学问题的一般过程如下：

1. 求出物体的质量特性（质量、质心和转动惯量）。
2. 分析并量化所有作用于物体的力与力矩。
3. 求出所有力与力矩的向量总和。
4. 解出线性及角加速度的运动方程。
5. 对时间积分，求出线性及角速度。
6. 再对时间积分，求出线性及角位移。

这些步骤使解动力学问题看起来比实际中有许多需克服的复杂因素的问题简单多了。例如，作用于物体的力大多是位移、速度或加速度的函数。即你需要用重复的技巧来解运动方程。再者，因为很可能无法导出加速度的封闭式解答，所以必须用数值积分以估计在每个考虑时间点的速度与位移。这些计算的方面将在第十一章到第十七章中说明。

## 2D 的粒子动力学

如同粒子运动学一样，在粒子动力学中，只需考虑粒子的线性运动。因此，运动方程将由方程$F = ma$构成，而每个坐标方向的运动都有自己的方程。2D粒子的运动方程如下：

$$\sum F_x = ma_x$$
$$\sum F_y = ma_y$$

其中$\Sigma F_x$是$x$方向上所有力的总和，$\Sigma F_y$则是$y$方向上所有力的总和，而$a_x$是$x$方向的加速度，$a_y$是$y$方向的加速度。

合力及合成加速度为：

$$a = a_x \boldsymbol{i} + a_y \boldsymbol{j}$$
$$a = \sqrt{a_x^2 + a_y^2}$$
$$\sum \boldsymbol{F} = \sum F_x \boldsymbol{i} + \sum F_y \boldsymbol{j}$$
$$\sum F = \sqrt{\left(\sum F_x\right)^2 + \left(\sum F_y\right)^2}$$

下面来看一个简单的说明范例。浮在水上的船，起初是静止的。启动其推进器，产生推力 T 驱动船身向前。假设船的前进速率很慢且阻力趋近于：

$$R = -Cv$$

其中 $R$ 是总阻力，$C$ 是阻力系数，$v$ 是船的速率，而负号表示此阻力方向与船前进方向相反。找出船速、加速度及行进距离为时间函数的公式，假设推进器的推力与阻力向量作用于同一通过船身重心的直线上（此假设将船视为粒子而不是刚体）。

解此问题的第一步是分析所有作用于船身的力。图 4-1 显示了所有作用于船身的力的自由体图（free-body diagram），即推力 $T$、阻力 $R$、船重 $W$ 和浮力 $B$。

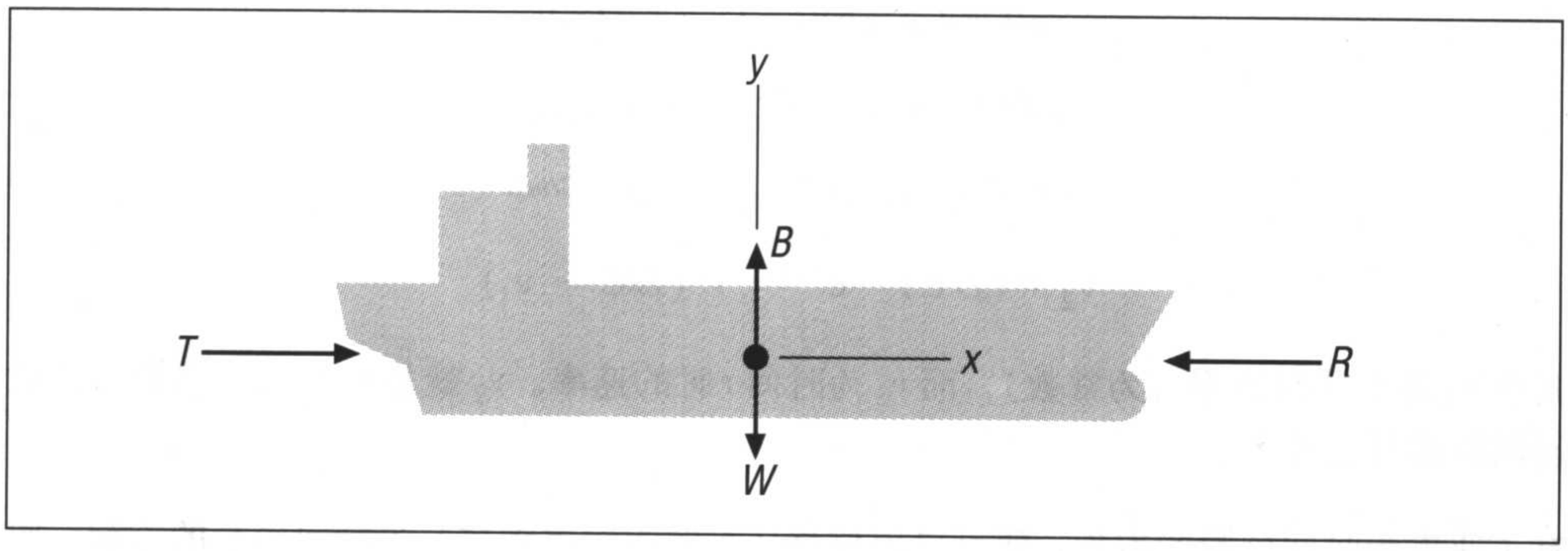



图 4-1：船的自由体图

注意到浮力大小恰好等于船重而两者的方向相反，因此，两力互相抵消，而 $y$ 方向便无任何运动，只要船继续浮着就会如此。这个结论有效地将问题简化为一维的 $x$ 方向运动，而作用于 $x$ 方向的力只有推进器推力及阻力。

现在用牛顿第二定律将 $x$ 方向的运动方程改写如下：

$$\Sigma F = ma$$

$$T - R = ma$$

$$T - (Cv) = ma$$

其中 $a$ 是 $x$ 方向的加速度，而 $v$ 是 $x$ 方向的速率。

下一步，将此运动方程积分，导出船速为时间函数的公式。需要将 $a = \mathrm{d}v/\mathrm{d}t$ 代入，重新排列、积分再解速率，过程如下：

$$T-(Cv)=m(dv/dt)$$

$$dt=[m/(T-Cv)]dv$$

$$\int_0^t dt=\int_{v_1}^{v_2}[m/(T-Cv)]\,dv$$

$$t-0=-(-(m/C)\ln(T-Cv))\Big|_{v_1}^{v_2}$$

$$t=-(m/C)\ln(T-Cv_2)+(m/C)\ln(T-Cv_1)$$

$$t=(m/C)[\ln(T-Cv_1)-\ln(T-Cv_2)]$$

$$(C/m)t=\ln[(T-Cv_1)/(T-Cv_2)]$$

$$e^{(C/m)t}=e^{\ln[(T-Cv_1)/(T-Cv_2)]}$$

$$e^{(C/m)t}=(T-Cv_1)/(T-Cv_2)$$

$$(T-Cv_2)=(T-Cv_1)e^{-(C/m)t}$$

$$v_2=(T/C)-e^{-(C/m)t}(T/C-v_1)$$

其中 $v_1$ 是船的初速率（为常数），而 $v_2$ 为时间 $t$ 时的速率。$v_2$ 就是所求的，它指出任意瞬间船的行进速率。

有了速率为时间函数的方程，便可导出位移（本例为行进距离）为时间函数的方程。回想一下公式 $v\,dt=ds$，将上面速率的公式代入，然后积分、重新排列，再解出行进距离。步骤如下：

$$v\,dt=ds$$

$$v_2\,dt=ds$$

$$((T/C)-e^{-(C/m)t}(T/C-v_1))dt=ds$$

$$\int_0^t(T/C)-e^{-(C/m)t}(T/C-v_1)dt=\int_{s_1}^{s_2}ds$$

$$(T/C)\int_0^t dt-(T/C-v_1)\int_0^t e^{-(C/m)t}dt=s_2-s_1$$

$$\left\{(T/C)t+[(T/C)-v_1](m/C)e^{-(C/m)t}\right\}_0^t=s_2-s_1$$

$$\{(T/C)t+[(T/C)-v_1](m/C)e^{-(C/m)t}\}-\{0+[(T/C)-v_1](m/C)\}=s_2-s_1$$

$$(T/C)t+[(T/C)-v_1](m/C)e^{-(C/m)t}-(T/C-v_1)(m/C)=s_2-s_1$$

$$s_2=s_1+(T/C)t+[(T/C)-v_1](m/C)e^{-(C/m)t}-(T/C-v_1)(m/C)$$

最后，回到原来的运动方程并解出加速度，可将方程写成：

$$T-(Cv)=ma$$

$$a=(T-(Cv))/m$$

$$\text{其中 } v=v_2=(T/C)-\mathrm{e}^{-(C/m)t}(T/C-v_1)$$

总结以上各式，速度、行进距离及加速度的方程如下：

$$v_2=(T/C)-\mathrm{e}^{-(C/m)t}(T/C-v_1)$$

$$s_2=s_1+(T/C)t+[(T/C)-v_1](m/C)\mathrm{e}^{-(C/m)t}-(T/C-v_1)(m/C)$$

$$a=[T-(Cv)]/m$$

为更进一步描述船的运动，图 4-2、图 4-3、图 4-4 绘制出了船的速率、行进距离与加速度对时间的关系图。为简化这些图解做了以下假设：

- 船的初速率与位移在时间 0 时皆为 0。
- 推进器推力是 20 000 推力单位。
- 船的质量为 10 000 质量单位。
- 阻力系数为 1000。

假设推力固定不变，将发现船的速率趋近稳定状态，20 速率单位。这与加速度的减少一致（从时间 0 的最大加速度到稳定速率时加速度为 0）。

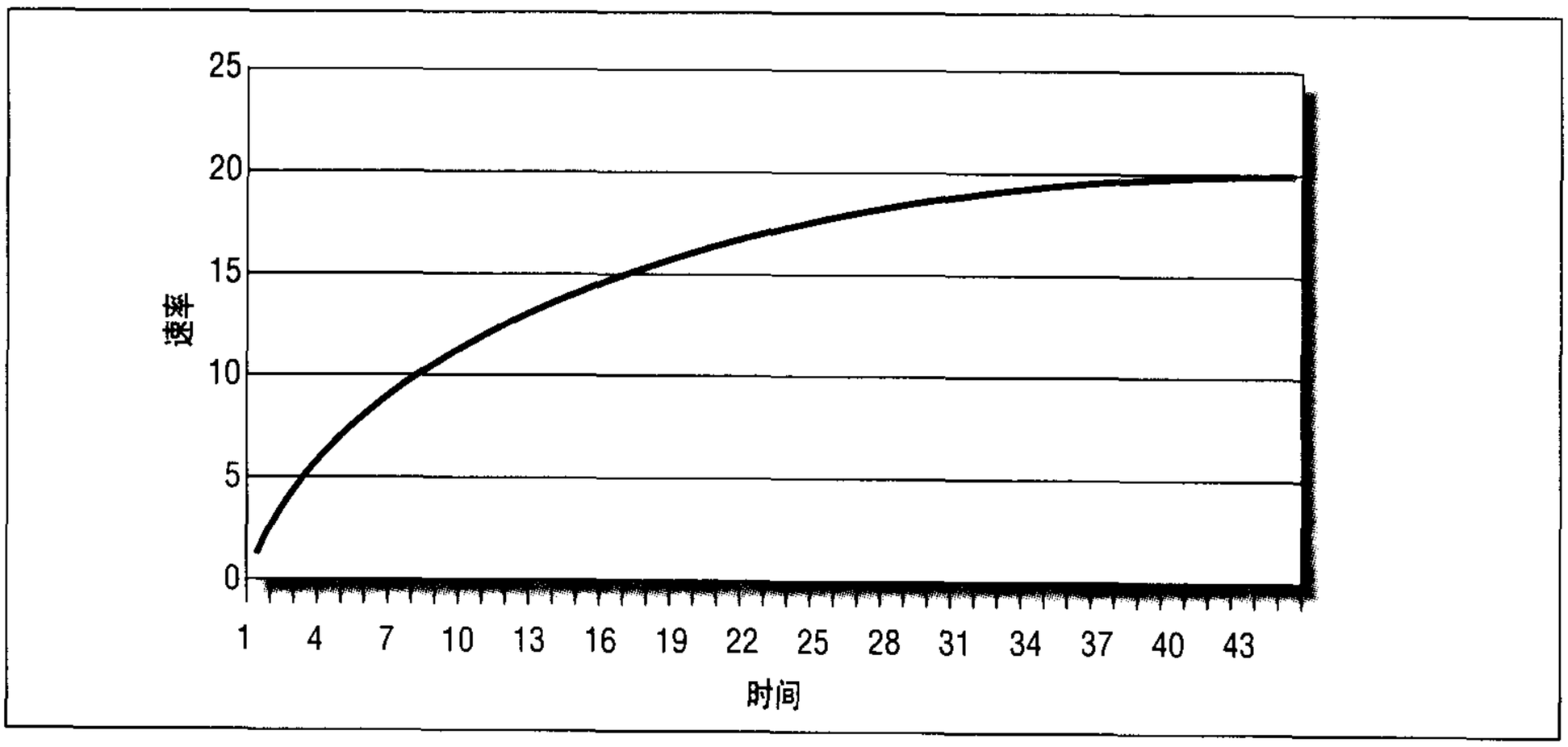



图 4-2：速率－时间关系图

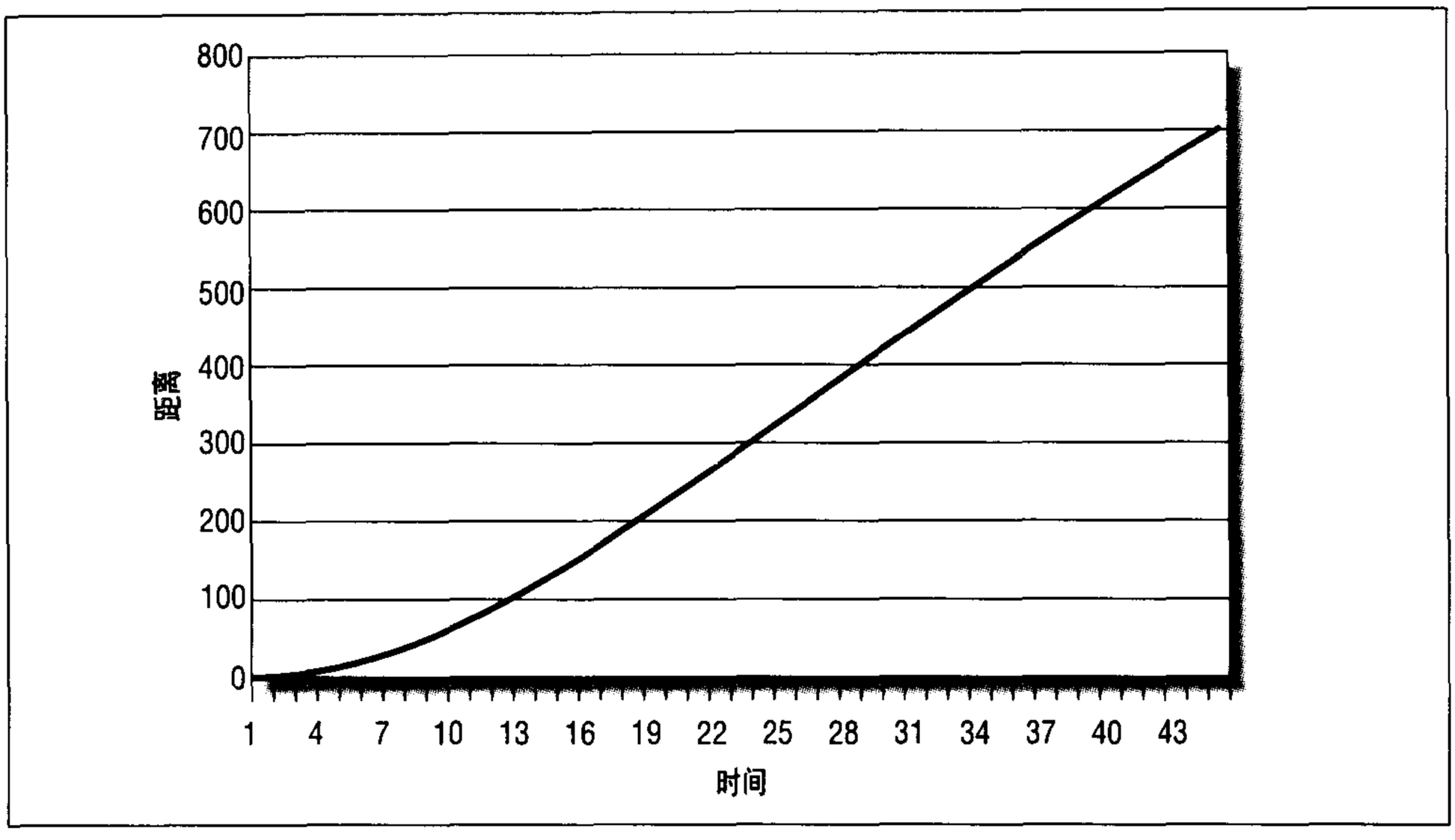



图 4-3：距离－时间关系图

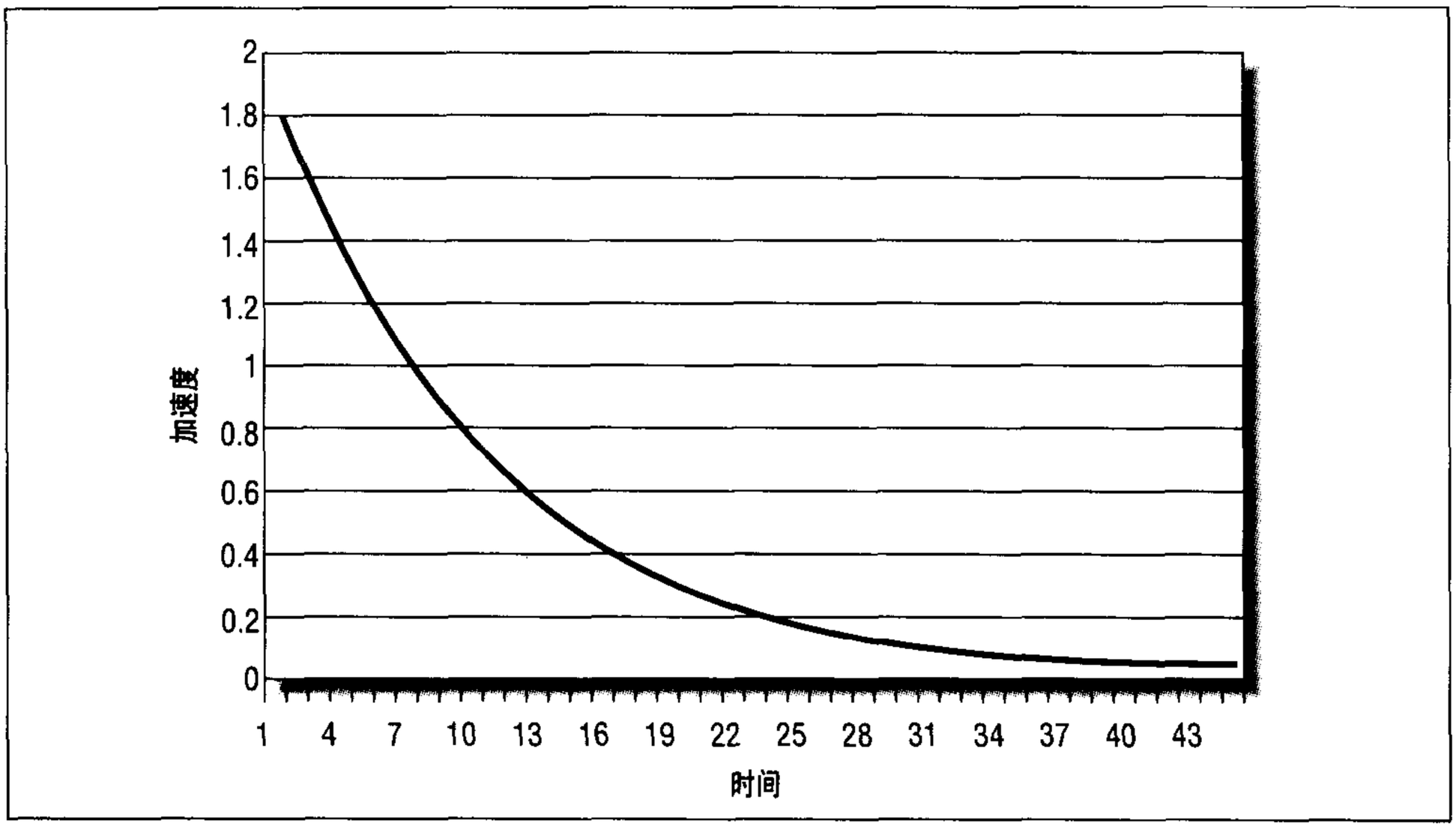



图 4-4：加速度－时间关系图

这个范例说明如何建立运动的微分方程，并将它们积分以求出速度、位移及加速度。此例中可求得封闭式解答，即可将方程以符号积分（symbolical integrate）并导出新的方程。因为加了足够的限制使问题易于处理，所以可以这样做。但你很快会发现，若有更

多力作用于船身上，或者推力并不固定且为速率的函数，或者若阻力为速率平方的函数等，问题就越来越复杂了，求出封闭式解答就更难了（如果可以求得）。

## 3D 粒子动力学

如同运动学一样，要将粒子运动方程延伸至3D空间很简单，只需多加一个分量便会导出三个方程，如下：

$$\sum F_x = ma_x$$
$$\sum F_y = ma_y$$
$$\sum F_z = ma_z$$

合力及加速度向量现在变成：

$$\boldsymbol{a} = a_x\mathbf{i} + a_y\mathbf{j} + a_z\mathbf{k}$$
$$a = \sqrt{a_x^2 + a_y^2 + a_z^2}$$
$$\sum \boldsymbol{F} = \sum F_x\boldsymbol{i} + \sum F_y\boldsymbol{j} + \sum F_z\boldsymbol{k}$$
$$\sum F = \sqrt{\left(\sum F_x\right)^2 + \left(\sum F_y\right)^2 + \left(\sum F_z\right)^2}$$

为加深对这些概念的理解，下面有另一个例子。

回到第二章讨论的大炮程序。该例子中有些简化的假设，我们才能将焦点集中在问题的运动学上，而不会将它弄得太复杂。其中主要的假设是炮弹在飞行过程中并没有空气阻力。实际上，只有在真空中这种假设才是有效的，在地球上当然是不可能的。另一个重要的假设是没有风的作用进而影响其路线。这两个需考虑的因素（风和空气阻力）在真实世界的抛体运动问题中是重要的，所以为了让本例更有趣些、更具挑战性（如果这是真实的游戏），接着将加入这两个考虑因素。

首先，假设炮弹为一个圆球，且作用于其上的空气阻力为某阻力系数和炮弹速率的函数。此空气阻力如下：

$$\boldsymbol{F}_{\mathrm{d}} = -C_{\mathrm{d}}\boldsymbol{v}$$
$$\boldsymbol{F}_{\mathrm{d}} = -C_{\mathrm{d}}v_x\boldsymbol{i} - C_{\mathrm{d}}v_y\boldsymbol{j} - C_{\mathrm{d}}v_z\boldsymbol{k}$$
$$\boldsymbol{F}_{\mathrm{dx}} = -C_{\mathrm{d}}v_{\mathrm{x}}$$
$$\mathbf{F}_{\mathrm{dy}} = -C_{\mathrm{d}}v_{\mathrm{y}}$$

其中 $C_d$ 是阻力系数，$v$ 是炮弹速度（$v_x$，$v_y$ 和 $v_z$ 为其分量），而减号表示空气阻力阻碍炮弹的运动。事实上这里稍微作弊一下，因为真实世界中流体的动阻力为速率平方的函数。为容易求出封闭式解答，所以将它简化了。

其次，假设物体受到刮风的影响且风力是某阻力系数和风速的函数。此力如下：

$$\boldsymbol{F}_w = -C_w v_w$$

$$\mathbf{F}_w = -C_w \mathbf{v}_{wx} \boldsymbol{i} - C_w \boldsymbol{v}_{wz} \boldsymbol{k}$$

其中 $C_w$ 是阻力系数，$v_w$ 是风的速率，而减号表示风力阻碍炮弹的运动（当风向与炮弹运动方向相反时）。当风顺着炮弹方向吹时（即从后面），则风会帮助炮弹向前而不是阻碍它的运动。通常 $C_w$ 不一定等于空气阻力公式的 $C_d$。参考图 2-3，将风向夹角定义为 $\gamma$。风力的 $x$ 和 $z$ 分量可用风向 $\gamma$ 表示如下：

$$F_{wx} = F_w \cos\gamma = -(C_w v_w)\cos\gamma$$

$$F_{wz} = F_w \sin\gamma = -(C_w v_w)\sin\gamma$$

最后，考虑炮弹所受的重力，而不是指定重力的效果为定加速度（如第二章所做的）。这让你能将重力纳入运动方程中。假设炮弹相当接近海平面，重力可写成：

$$\boldsymbol{F}_g = -mg\boldsymbol{j}$$

其中减号表示重力作用于负 $y$ 方向（将炮弹拉向地表），而方程右边的 g 是海平面的重力加速度。

现在，所有的力都确定了，可以写出每个坐标方向的运动方程：

$$\sum F_x = -F_{wx} - F_{dx} = m(\mathrm{d}v_x / \mathrm{d}t)$$
$$\sum F_y = -F_{dy} - F_{gy} = m(\mathrm{d}v_y / \mathrm{d}t)$$
$$\sum F_z = -F_{wz} - F_{dz} = m(\mathrm{d}v_z / \mathrm{d}t)$$

请注意，各方程中已经将加速度代换成 $\mathrm{d}v/\mathrm{d}t$。接下来的过程与上一节所示的相同，将运动方程做两次积分：第一次找出速度为时间函数的方程，第二次找出位移为时间函数的方程。如往常一样，后面将分别示范各分量的运动方程。

或许你会问，“大炮射出炮弹的推力到哪去了？”本例中所看到的是炮弹在离开炮管之后的运动，不再有推力作用在炮弹上（它不是自力推进的）。要解释大炮推力（当炮弹在炮管中，其作用时间很短）的影响，必须考虑炮弹最初离开大炮时的枪口速度。枪口

速度在坐标方向的分量便是各方向的初速度，而它们将会包括在积分后的运动方程。初速度会出现在速度及位移方程中，正如第二章中的范例一样。这些将在下一节讨论到。

## x分量

将运动方程中的力做适当的代换，然后将它们积分，求出速度的方程：

$$-F_{wx} - F_{dx} = m(dv_x/dt)$$

$$-(-C_w v_w)\cos\gamma - (-C_d v_x) = m(dv_x/dt)$$

$$t(C_w v_w)\cos\gamma + C_d v_x = m(dv_x/dt)$$

$$dt = m dv_x/[-(C_w v_w \cos\gamma) - C_d v_x]$$

$$\int_0^t dt = \int_{v_{x1}}^{v_{x2}} -m/[(C_w v_w \cos\gamma) + C_d v_x]dv_x$$

$$t = -(m/C_d)\ln[(C_w v_w \cos\gamma) + C_d v_x]\Big|_{v_{x1}}^{v_{x2}}$$

$$t = -(m/C_d)\ln[(C_w v_w \cos\gamma) + C_d v_{x_2}] + (m/C_d)\ln[(C_w v_w \cos\gamma) + C_d v_{x_1}]$$

$$(C_d/m)t = \ln\{[(C_w v_w \cos\gamma) + C_d v_{x_1}]/[(C_w v_w \cos\gamma) + C_d v_{x_2}]\}$$

$$e^{(C_d/m)t} = e^{\ln\{[(C_w v_w \cos\gamma) + C_d v_{x_1}]/[(C_w v_w \cos\gamma) + C_d v_{x_2}]\}}$$

$$e^{(C_d/m)t} = [(C_w v_w \cos\gamma) + C_d v_{x_1}]/[(C_w v_w \cos\gamma) + C_d v_{x_2}]$$

$$[(C_w v_w \cos\gamma) + C_d v_{x_2}] = [(C_w v_w \cos\gamma) + C_d v_{x_1}]e^{-(C_d/m)t}$$

$$v_{x_2} = (1/C_d)[e^{(-C_d/m)t}(C_w v_w \cos\gamma + C_d v_{x_1}) - (C_w v_w \cos\gamma)]$$

为求出位移为时间函数的方程，回忆一下 $v\,dt = ds$，用上面的方程将 $v$ 代换掉，然后再积分一次：

$$v_{x_2} dt = ds_x$$

$$(1/C_d)[e^{(-C_d/m)t}(C_w v_w \cos\gamma + C_d v_{x_1}) - (C_w v_w \cos\gamma)]dt = ds_x$$

$$\int_0^t (1/C_d)[e^{(-C_d/m)t}(C_w v_w \cos\gamma + C_d v_{x_1}) - (C_w v_w \cos\gamma)]dt = \int_{s_{x_1}}^{s_{x_2}} ds_x$$

$$s_{x_2} = \{(m/C_d)e^{(-C_d/m)t}[-(C_w v_w \cos\gamma)/C_d - v_{x_1}] - [(C_w v_w \cos\gamma)/C_d]t\}$$
$$-\{(m/C_d)[-(C_w v_w \cos\gamma)/C_d - v_{x_1}]\} + s_{x_1}$$

这些方程的确不好看。想像一下，如果没有假设空气阻力与速率（而非速率的平方）成正比，会求得有一个或两个反正切（arcus tangent）项的方程。

## y分量

就y分量而言，需遵循与上面的x分量示范的相同的程序，只是以y方向上的力来计算。即：

$$-F_{dy} - F_{gy} = m(dv_y/dt)$$

$$-(-C_d v_y) - (-mg) = m(dv_y/dt)$$

$$C_d v_y + mg = m(dv_y/dt)$$

$$\int_0^t dt = -m\int_{v_{y1}}^{v_{y2}} 1/(C_d v_y + mg)dv_y$$

$$v_{y2} = (1/C_d)\ e^{(-Cd/m)t}(C_d v_{y1} + mg) - (mg)/C_d$$

现在有了速度的方程，可如之前一样接着求出位移的方程：

$$v_{y2}dt = ds_y$$

$$[(1/Cd)e^{(-Cd/m)t}(C_d v_{y1} + mg) - (mg)/C_d]\ dt = ds_y$$

$$\int_0^t [(1/C_d)e^{(-C_d/m)t}(C_d v_{y1} + mg) - (mg)/C_d]dt = \int_{s_{y1}}^{s_{y2}} ds_y$$

$$s_{y2} = s_{y1} + \{-[v_{y1} + (mg)/C_d](m/C_d)e^{(-Cd/m)t} - t(mg)/C_d\} + \{(m/C_d)[v_{y1} + (mg)/C_d]\}$$

现在已经解决两个，只剩一个分量了。

## z分量

到了z分量就可以喘口气了。你会发现x分量与z分量的运动方程除了x，z下标符号及正弦对余弦之外，看起来几乎一样。利用这个事实，只要将x分量的方程复制过来，再将x下标符号代换成z，而余弦换成正弦，便可以了：

$$v_{z2} = (1/C_d)[e^{(-Cd/m)t}(C_w v_w \sin\gamma + C_d v_{z1}) - (C_w v_w \sin\gamma)]$$

$$s_{z2} = \{(m/C_d)e^{(-Cd/m)t}[-(C_w v_w \sin\gamma)/C_d - v_{z1}] - [(C_w v_w \sin\gamma)/C_d]t\} - \{(m/C_d)[-(C_w v_w \sin\gamma)/C_d - v_{z1}]\} + s_{z1}$$

## 大炮范例修正版

有了各坐标方向的炮弹位移新方程，再回到大炮范例程序代码，并把旧的位移计算公式换成新的。使DoSimulation中的改变如下：

```
//-----------------------------------------------------------
//
int          DoSimulation(void)
//-----------------------------------------------------------
//
{

      .
      .
      .

// 新的局部变量:
double              sx1, vx1;
double              sy1, vy1;
double              sz1, vz1;

.
.
.

// 计算此时间的位置向量

// 将旧的位置向量注释掉:
//s.i = Vm * cosX * time + xe;
//s.j = (Yb + L * cos(Alpha * 3.14/180)) + (Vm * cosY * time) -
//     (0.5 * g * time * time);
//s.k = Vm * cosZ * time + ze;

// 新的位置向量计算:
sx1 = xe;
vx1 = Vm * cosX;
sy1 = Yb + L * cos(Alpha * 3.14/180);
vy1 = Vm * cosY;

sz1 = ze;
vz1 = Vm * cosZ;

s.i = ( (m/Cd) * exp(-(Cd * time)/m) * ((-Cw * Vw *
       cos(GammaW * 3.14/180))/Cd - vx1) - (Cw * Vw *
       cos(GammaW * 3.14/180) * time)/Cd ) - ( (m/Cd) *
       ((-Cw * Vw * cos(GammaW * 3.14/180))/Cd - vx1) ) +sx1;

      s.j = sy1 + ( -(vy1 + (m * g)/Cd) * (m/Cd) * exp(-(Cd*time)/m) -
            (m * g * time) / Cd ) + ( (m/Cd) * (vy1 + (m * g)/Cd) );

      s.k = ( (m/Cd) * exp(-(Cd * time)/m) * ((-Cw * Vw *
             sin(GammaW *3.14/180))/Cd - vz1) - (Cw * Vw *
             sin(GammaW * 3.14/180) * time) / Cd ) - ( (m/Cd) *
             ((-Cw * Vw * sin(GammaW * 3.14/180))/Cd - vz1) ) + sz1;

      .
      .
      .
}
```

要想考虑到相反的风力和空气阻力，需加入一些新的全局变量，存储风速及风向、炮弹质量，以及阻力系数。还需在对话窗口上加入一些控制项，这样当执行此程序时才能改变这些变量。图 4-5 显示这些在主窗口右上角新增的界面控制项。

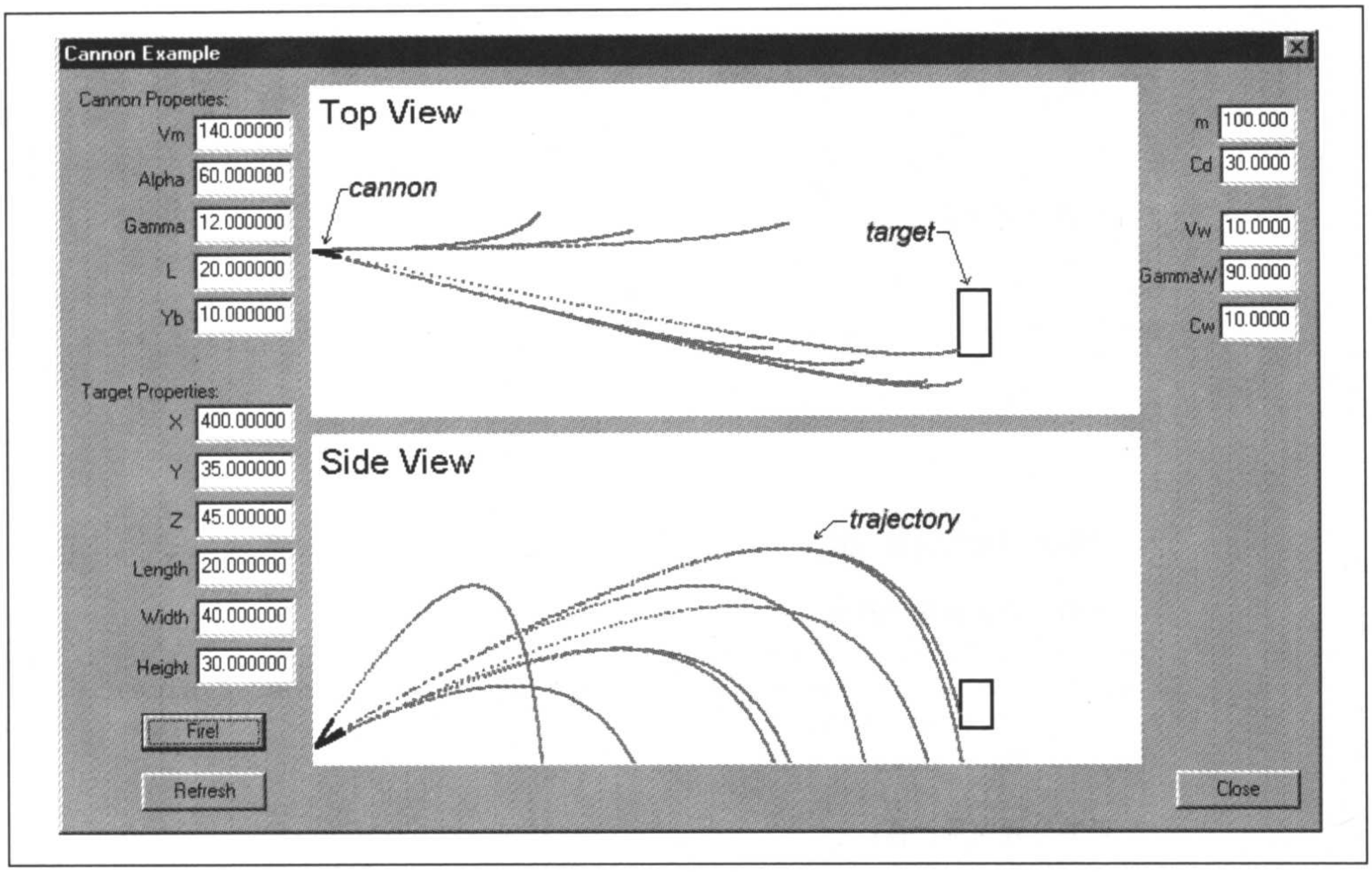



图 4-5：炮弹范例修订版的屏幕快照

在 DemoDlgProc 函数中也加了以下几行，来处理风速和风向的值：

```
//-------------------------------------------------------
//
LRESULT CALLBACK DemoDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
//-------------------------------------------------------
//
{
    .
    .

    case WM_INITDIALOG:

    .
    .
    .

            // 新变量:
            sprintf( str, "%f", m );
            SetDlgItemText(hDlg, IDC_M, str);
```

```
sprintf( str, "%f", Cd );
SetDlgItemText(hDlg, IDC_CD, str);

sprintf( str, "%f", Vw );
SetDlgItemText(hDlg, IDC_VW, str);

sprintf( str, "%f", GammaW );
SetDlgItemText(hDlg, IDC_GAMMAW, str);

sprintf( str, "%f", Cw );
SetDlgItemText(hDlg, IDC_CW, str);

.
.
.

    case IDC_REFRESH:

.
.
.

    // 新变量:
    GetDlgItemText(hDlg, IDC_M, str, 15);
    m = atof(str);

    GetDlgItemText(hDlg, IDC_CD, str, 15);
    Cd = atof(str);

    GetDlgItemText(hDlg, IDC_VW, str, 15);
    Vw = atof(str);

    GetDlgItemText(hDlg, IDC_GAMMAW, str, 15);
    GammaW = atof(str);

    GetDlgItemText(hDlg, IDC_CW, str, 15);
    Cw = atof(str);

.
.
.

    case IDC_FIRE:

.
.
.

    // 新变量:
    GetDlgItemText(hDlg, IDC_M, str, 15);
    m = atof(str);

    GetDlgItemText(hDlg, IDC_CD, str, 15);
    Cd = atof(str);
```

```
        GetDlgItemText(hDlg, IDC_VW, str, 15);
        Vw = atof(str);

        GetDlgItemText(hDlg, IDC_GAMMAW, str, 15);
        GammaW = atof(str);

        GetDlgItemText(hDlg, IDC_CW, str, 15);
        Cw = atof(str);

            .
            .
            .
    }
```

执行范例程序之后，你应该很快看到炮弹的轨迹明显地与原先范例中的不同。借助于调整风速、风向及摩擦系数，可以大大地影响炮弹的轨迹。若设定风速为0而阻力系数为1，轨迹看起来会跟原先范例（即不考虑风及空气阻力）的相似。但是要小心，别将阻力系数设为0，因为这会导致“除以0”的错误。程序中并没有做异常处理，但透过位移向量公式就能知道结果会如何了，因为在许多项的分母中出现了阻力系数。

如果这是电子游戏，从使用者的观点来看，当考虑风及空气阻力时，命中目标的问题变得更有挑战了。风的要素相当有趣，因为在玩游戏期间能改变风速及风向，促使玩家为精确命中目标而更加注意风的影响。

# 刚体动力学

你已经在第二章学到了处理加入转动或角运动的刚体运动学。如前面所述，运动方程由一组描述力与线性加速度关系的方程和另一组力矩与角加速度关系的方程构成。你可以将运动方程想像成力与线性动量变化率的关系，或者力矩与角动量变化率的关系（如第一章所讨论的）。

在运动学中，处理刚体动力学问题的程序涉及两个方面：记录物体质心的移动 —— 将物体视为粒子；记录物体的转动 —— 利用本体坐标的原理，和第二章讨论的相对角速度及角加速度。实际上，刚体运动学和动力学问题之间惟一的差异是动力学问题需考虑力（包含合成力矩）。

为方便再把向量方程列出：

$$\boldsymbol{F} = m\boldsymbol{a}$$

$$\boldsymbol{M}_{\mathrm{cg}} = \boldsymbol{I}\alpha$$

在2D空间中，

$$\sum F = \sum F_x i + \sum F_y j$$

$$\sum F = \sqrt{\left(\sum F_x\right)^2 + \left(\sum F_y\right)^2}$$

从2D空间粒子问题到刚体问题只需多增加一个方程。当然此方程为全部作用于物体的力矩总和与物体的转动惯量及其角加速度的关系式。在平面运动中，刚体的转动轴永远是垂直于坐标平面的。因为只有一个转动轴，所以只需考虑一个转动惯量和一个角加速度。因此可写成：

$$M_{cg} = I\alpha$$

其中 $M_{cg}$ 为总力矩且利用第三章“力与力矩”一节中提到的公式求出，而 $I$ 是对转动轴的转动惯量，用第一章“质量、质心和转动惯量”一节讨论的技巧求出。

2D动力学问题的运动方程以其分量形式为：

$$\sum F_x = ma_x$$
$$\sum F_y = ma_y$$
$$\sum M_{cg} = I\alpha$$

因为这些方程表示 $xy$ 平面上的线性运动，所以角加速度会绕着垂直 $xy$ 平面的 $z$ 轴。同样，转动惯量 $I$ 也会绕着 $z$ 轴。

回忆第三章中，力矩是取该力的位置向量与该力向量的外积。这表示，不同于粒子动力学，现在需记录每个作用在物体上的力的精确位置。这最好用例子来说明。

如图4-6所示，考虑一个密度均匀的箱子。密度均匀即箱子的重心位于其几何中心（形心）。求出作用于箱子上边使箱子倾斜所需的力 $F_p$ 的最小值。

图中，$F_p$ 是作用力，$R_1$ 和 $R_2$ 分别为支脚1和2的反作用力，$F_{f1}$ 和 $F_{f2}$ 分别是点1和点2的摩擦力，而 $mg$ 是箱子的重量。

这是这类问题的范例，已知关于物体运动的一些条件，而必须求出作用于其上的一个或更多的力的值。要求出足以使箱子开始倾斜的力，需看支脚2上反作用力为0的瞬间。这表示箱子所有的重量全由点1支撑，且箱子开始倾斜。在开始转动前的一瞬间，箱子的角加速度为0。注意，箱子的线性加速度不一定为0。也就是可推动箱子且它会滑动却未倾斜。

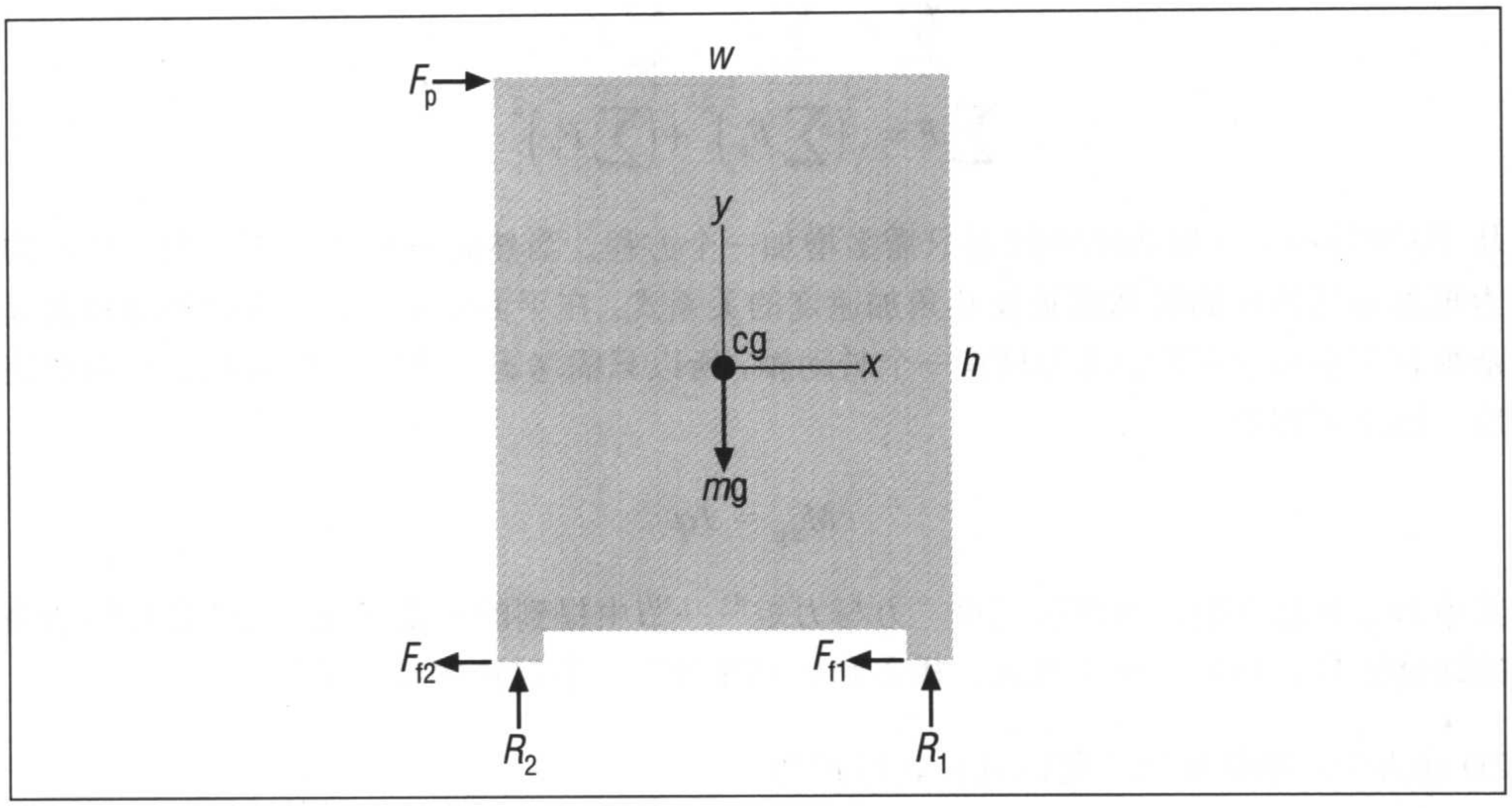



图 4-6：箱子的自由体图

这个问题的运动方程如下：

$$\sum F_x = F_p - F_{f_2} - F_{f_1} = ma_x$$

$$\sum F_y = R_1 + R_2 - mg = ma_y = 0$$

$$\sum M_{cg} = F_p(h/2) + R_2(w/2) - R_1(w/2) + F_{f_2}(h/2) + F_{f_1}(h/2) = I\alpha = 0$$

当 $R_2$ 为0时，重写（上面）第二个方程得出 $R_1$ 等于箱子的重量。当 $R_2$ 为0时，$R_2(w/2)$ 这项便可从力矩方程去掉，解 $F_p$ 与 $R_1$ 的关系。当 $R_2$ 为0时，$F_{f_2}$ 也会为0。几个代数运算之后，此方程如下：

$$F_p = mg(w/h) - F_{f_1}$$

从方程得知作用于箱子上边的倾斜力与箱子的重量和尺寸成比例（实际上，为其宽与长的比），从物理的观点可以很容易地领会。摩擦力在这里是很重要的，正是由于它的存在才能使箱子倾斜。若箱子处在无摩擦力的表面，那么它只会滑动而不会倾斜。

再来看另一个例子。一圆柱体位于倾斜平面上，如图4-7所示。若圆柱体置于平面顶端且放开，它会开始滚下平面。下面开发圆柱体滚下时的线性加速度及角速度的方程。注意，圆柱体会滚动是因为它与平面间的摩擦力产生的力矩。若这是无摩擦力的问题，圆柱体便不会滚下平面，而是滑下平面，而其角速度为0。

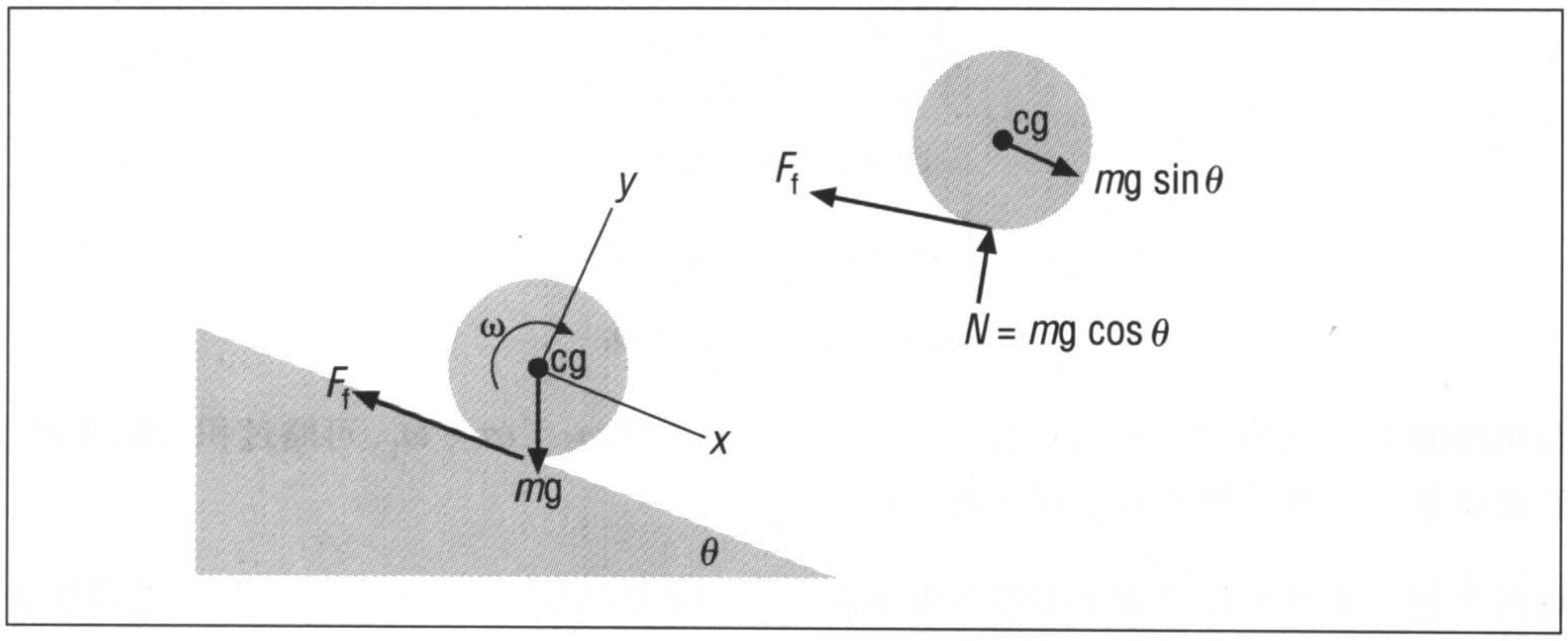



图 4-7：位于倾斜平面的圆柱体

这个问题中，设定坐标系使 $x$ 轴平行于倾斜平面。这让方程更简洁且可以有效地排除 $y$ 分量，因为圆柱体并未（垂直地）移进或离开平面。$y$ 方向的运动方程指出 $y$ 方向上的两个力（此例中为圆柱体重量的 $y$ 分量，以及垂直于平面的反作用力）相等且方向相反，因而互相抵消：

$$\sum F_y = mg\cos\theta - mg\cos\theta = 0$$

这就容易多了，现在看 $x$ 方向上的力。运动方程为：

$$\sum F_x = (mg\sin\theta) - F_f = ma_x$$

其中 $F_f$ 是摩擦力，而 $a_x$ 为圆柱体重心的线性加速度（在 $x$ 方向上）。假设圆柱体滚动而不滑动，则摩擦力等于 $\mu_s N$，其中 $\mu_s$ 为静摩擦力系数，而 $N$ 是圆柱体与平面间的垂直反作用力（注 1）。将 $F_f$ 代换并解 $a_x$，得到

$$(mg\sin\theta) - \mu_s N = ma_x$$

$$a_x = g(\sin\theta - \mu_s\cos\theta)$$

注意，在指定平面角度与摩擦系数的情况下此加速度是固定的。

要想求出角速度，需求出重心所有的力矩总和，并以 $d\omega/dt$ 代入 $\alpha$，然后积分并解角速度 $\omega$：

---

注 1：　若圆柱体滚动并滑动，则用动摩擦力系数而非静摩擦力系数。

$$\sum M_{cg} = F_f r = I_{cg}\alpha$$
$$F_f r = I_{cg}\mathrm{d}\omega / \mathrm{d}t$$
$$\mathrm{d}t = I_{cg} /(F_f r)\mathrm{d}\omega$$
$$\int_0^t \mathrm{d}t = I_{cg} /(F_f r)\int_{\omega_1}^{\omega_2} \mathrm{d}\omega$$
$$\omega_2 = [(F_f r) / I_{cg}]t + \omega_1$$

如果你察觉到此为定加速度问题并回想第二章的方程 $\omega_2 = \alpha t + \omega_1$，可略过积分的步骤，直接由第一行公式解出 $\alpha$ 并代入此方程。

这两个例子说明刚体动力学中非常重要的一点：除考虑力的大小和方向之外，还必须考虑力的作用点，才能正确地计算角运动。

在这里讨论的刚体平面运动或2D运动的例子中，能轻易地设定运动方程并研究物体的线性及角运动。在广义的3D运动中，刚体的线性运动与粒子的运动没有区别，只需追踪刚体的重心运动。然而在3D空间中，转动却有点麻烦，因为它不再是简单地将单一轴线的转动视为平面运动，而需考虑任一轴线的转动，这导致描述任意的转动有点困难（欧拉角不再有用），并且使决定任一轴线的转动惯量有点复杂。第十一章到第十五章将讨论这些问题。

# 第五章

# 碰撞

现在你已了解了粒子和刚体的运动，接着要看当它们撞在一起时会发生什么事情。这就是本章讨论的主题，特别是告诉你如何处理粒子和刚体的碰撞反应。

在开始之前，先将碰撞“侦测”和碰撞“反应”做个区分。碰撞侦测是计算的几何问题，这牵扯到判断两物体（或更多）是否碰撞及在何处碰撞。而碰撞反应是物理问题，牵涉到两物体（或更多）在碰撞之后的运动。虽然这两类问题密切相关，而本章将着重讨论碰撞反应方面的问题。

然而我必须说，碰撞侦测并没有被忽略；它是假定物体不能互相穿透的实时模拟中要考虑的重要方面。碰撞反应演算法依据碰撞侦测演算法的结果，以准确地判断任何碰撞的适当反应；因此，应确定你的碰撞侦测法则是准确而可靠的。也就是说，碰撞侦测不是件简单的差事，我发现要完整地实现它比刚体模拟的物理层面还难上许多。就游戏应用而言，执行速度也是主要的议题，相信你是知道的，十分精确的碰撞侦测会很慢。正是由于速度和简单的缘故，我们将利用边界球体（以及边界方块）与边和面的碰撞侦测法则。在第十三章、第十六章中的范例模拟中将谈到更多关于此主题的内容。

本章中对刚体碰撞反应的处理是根据标准的（牛顿）碰撞定律。碰撞的物体无论其结构与材质皆视为刚体。如同前几章一样，这里讨论的刚体即使在碰撞时也不改变其外形。当然，这只是理想化的情况。从日常经验中可知，当物体相撞时它们会凹陷、弯曲、压缩或起皱。例如，当棒球击中球棒时，棒球在撞击的毫秒间会压缩 3/4 英寸。尽管这是事实，我们仍依赖完整的分析及经验方法来估算刚体碰撞。

这种标准的方法广泛地应用于工程机械设计、分析和模拟；然而，就刚体模拟而言，有另一套方法可使用，就是“惩罚法”（penalty method）（注 1）。

惩罚法中，撞击力就是使撞击点上的物体间压缩的暂时弹力。此弹力压缩经过很短暂的时间，并将大小相等且方向相反的力作用于碰撞物体上以模拟碰撞反应。此方法的提议者说它有容易实现的好处。然而，实现上碰到的困难之一是数值的不稳定。惩罚法的使用还有其他争议，在这里不加以讨论。我将许多参考资料加入参考文献中，若你有兴趣，可以看看。

## 冲量 / 动量定律

冲力的定义是作用时间非常短暂的力。例如，开枪时施加在子弹上的力称为冲力。两碰撞物体间的碰撞力也称为冲力，当你踢足球或以球棒击中棒球时也是冲力。

冲量是一个向量，其大小等于动量的变化量。所谓的“冲量 / 动量”定律，就是力矩的变化等于所作用的冲量。关于定质量及转动惯量的问题，可写成

$$\text{线性冲量} = \int_{t-}^{t+} \boldsymbol{F}\,\mathrm{d}t = m(\boldsymbol{v}_+ - \boldsymbol{v}_-)$$

$$\text{角冲量} = \int_{t-}^{t+} \boldsymbol{M}\,\mathrm{d}t = \boldsymbol{I}(\boldsymbol{\omega}_+ - \boldsymbol{\omega}_-)$$

这些等式中，**F** 是冲力，**M** 是冲力的力矩，$t$ 是时间，**v** 是速度，下标符号 − 表示冲撞前的瞬间，上标符号 + 表示冲撞后的瞬间。用下列等式便可求出平均冲力及力矩：

$$\boldsymbol{F} = m(\boldsymbol{v}_+ - \boldsymbol{v}_-)/(t_+ - t_-)$$

$$\boldsymbol{M} = \boldsymbol{I}(\boldsymbol{\omega}_+ - \boldsymbol{\omega}_-)/(t_+ - t_-)$$

考虑下面这个简单的例子：150 g（0.01028 slug）的子弹以枪口速度 2480 ft/s 发射出去，通过 24 in. 长的枪管共花了 0.0008 s；求子弹受到的冲量及平均冲力。本例中，子弹的质量固定为 150 g，而其初速度为 0；因此，其初动量为 0。在开枪后一瞬间，子弹的动量为其质量乘以枪口速度 2480 ft/s，得到动量等于 25.5 slug.ft/s。冲量等于动量的变化量，所以是 25.5 slug.ft/s。平均冲力等于冲量除以力作用的时间，本例为：

---

注 1：本书用此标准方法并提到惩罚法，只是要让你知道将示范的方法不只一个。大致来说，“惩罚法”中的“惩罚数”是指反弹常数，它通常很大，用来表示反弹的僵硬程度从而表示碰撞物体的硬度（或软度）。这些常数会在描述物体碰撞前后的运动等式中用到。

$$\text{平均冲力} = (25.5\ \text{slug-ft/s})/(0.0008\ \text{s})$$

$$\text{平均冲力} = 3187\ \text{lb}$$

这是对冲量概念简单而重要的说明，而在你处理刚体碰撞时也会用到相同的定律。撞击期间，冲撞力通常很大，而撞击时间很短。当两物体相撞时，两者均施冲力于对方；这些力大小相等而方向相反。在步枪范例中，施于子弹上的冲量，也以反方向施于步枪上而产生后坐力。这就是牛顿第三运动定律。

## 撞击

除了上一节讨论的冲量/动量定律，标准的撞击或碰撞反应分析还依据另一个基本定律：牛顿的动量守恒定律 —— 当刚体系统碰撞时，动量守恒。这表示固定质量的物体，其质量与速度之积的总和在撞击前后是相等的：

$$m_1 v_{1-} + m_2 v_{2-} = m_1 v_{1+} + m_2 v_{2+}$$

这里，$m$ 代表质量，$v$ 代表速度，下标符号 1 表示物体 1、下标符号 2 表示物体 2，下标符号 − 表示撞击前的瞬间，而下标符号 + 表示撞击后的瞬间。

本方法假设撞击瞬间主要的力是冲撞力，其他的力都假设为在短时间内可忽略。记住此假设，因为稍后的第十三章实现碰撞反应 2D 实时模拟范例时，将用到它。

先前提到刚体在碰撞时并不会改变外形，而你自身的经验中真实的物体在碰撞时确实改变了外形。真实世界中，动能会转成应变能（strain energy），使物体变形。当物体的变形是永久的时，能量消失，因此动能不会转换。

### 动能

动能是关于移动物体的能量形式。动能等于物体从静止加速所需的能量，也等于使移动物体静止所需的能量。动能是物体速率或速度与其质量的函数。线性动能的公式如下：

$$KE_{\text{linear}} = (1/2)mv^2$$

角或转动动能是物体的转动惯量及角速度的函数：

$$KE_{\text{angular}} = (1/2)I\omega^2$$

两碰撞物体间的动能守恒是指，两物体碰撞前的能量总和等于碰撞后的能量总和：

$$m_1 v_{1-}^2 + m_2 v_{2-}^2 = m_1 v_{1+}^2 + m_2 v_{2+}^2$$

牵涉到动量散失的碰撞称为非弹性碰撞或塑性碰撞。例如，若以相反方向丢出两个泥球，它们的动能转化成使泥球变形的应变能，而它们的碰撞反应（即撞击之后的运动）就没那么引人注目了。若为完全非弹性碰撞，则两泥球会粘在一块且在撞击后以相同的速度一起移动。动能守恒的碰撞称为完全弹性碰撞。在这些碰撞中，所有物体动能的总和在撞击前后是相等的。弹性碰撞（虽非完全弹性）的最佳范例是两颗撞球间的碰撞，其中球的变形是可忽略的，而且在正常情况下是非永久的。

当然事实上，碰撞大多介于完全弹性和完全非弹性之间。这表示就刚体而言（其外形不会改变），将利用由经验得出的关系式为要模拟的碰撞的弹性程度定量。此关系式是碰撞物体的相对分离速度与相对接近速度的比例：

$$e = -(v_{1+} - v_{2+})/(v_{1-} - v_{2-})$$

这里，$e$ 是恢复系数（coefficient of restitution）且为物体材质、结构、几何形状的函数。这个系数可由特殊的碰撞实验测得，例如，棒球与球棒或高尔夫球棍与球之间的碰撞。对于完全非弹性碰撞，$e$ 等于 0；而对于完全弹性碰撞，$e$ 等于 1。对于既不是完全弹性也不是完全非弹性碰撞的情况，$e$ 是介于 0 和 1 之间的任意值。在这方面，所考虑的速度沿着碰撞的作用线。

在无摩擦力的碰撞中，撞击的作用线垂直（或正交）于碰撞的接触面。当物体速度沿着作用线时，这种碰撞称为“直接碰撞”（direct impact）；当作用线通过物体的质心时，这种碰撞称为“中心碰撞”（central impact）。粒子与质量分布均匀的球体遭受的撞击都是中心碰撞。而直接中心碰撞发生在作用线通过碰撞物体质心且速度沿着作用线时。当物体速度不沿着作用线时，这种撞击称为“倾斜碰撞”（oblique impact）。你可以利用分量坐标来分析倾斜碰撞，但其中平行于作用线的分量才与撞击有关，而垂直于作用线的分量则无。图 5-1 显示了这些撞击。

来看一个例子，考虑图 5-2 中两颗撞球之间的碰撞。

两颗球直径都是标准的 2.25 in.，重量都为 5.5 oz.，假设碰撞几乎是完全弹性的且恢复系数是 0.9。若当球 1 撞倒球 2 时其 $x$ 方向的速度为 20 ft/s，如图 5-2 所示，求两球碰撞后的速度（不考虑摩擦力）。

所要做的第一件事就是确认撞击的作用线是沿着两球重心连线，因为两物体皆为球体，所以其作用线亦垂直于球体表面。则其单位垂直向量可写成：

$$\boldsymbol{n} = \left(\sqrt{\left((2r)^2 - r^2\right)}\boldsymbol{i} - r\boldsymbol{j}\right)/|\boldsymbol{n}|$$

$$\boldsymbol{n} = (0.866)\boldsymbol{i} - (0.5)\boldsymbol{j}$$

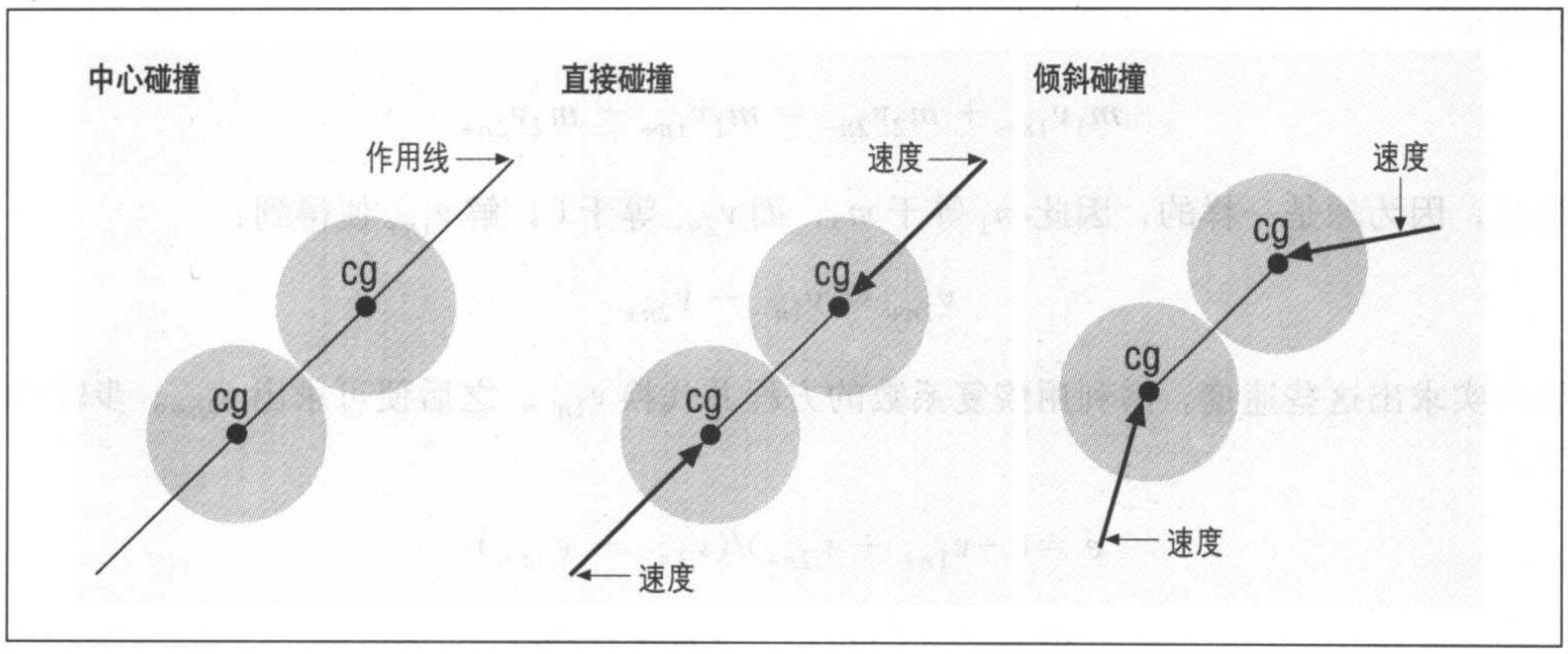



图 5-1：撞击的种类

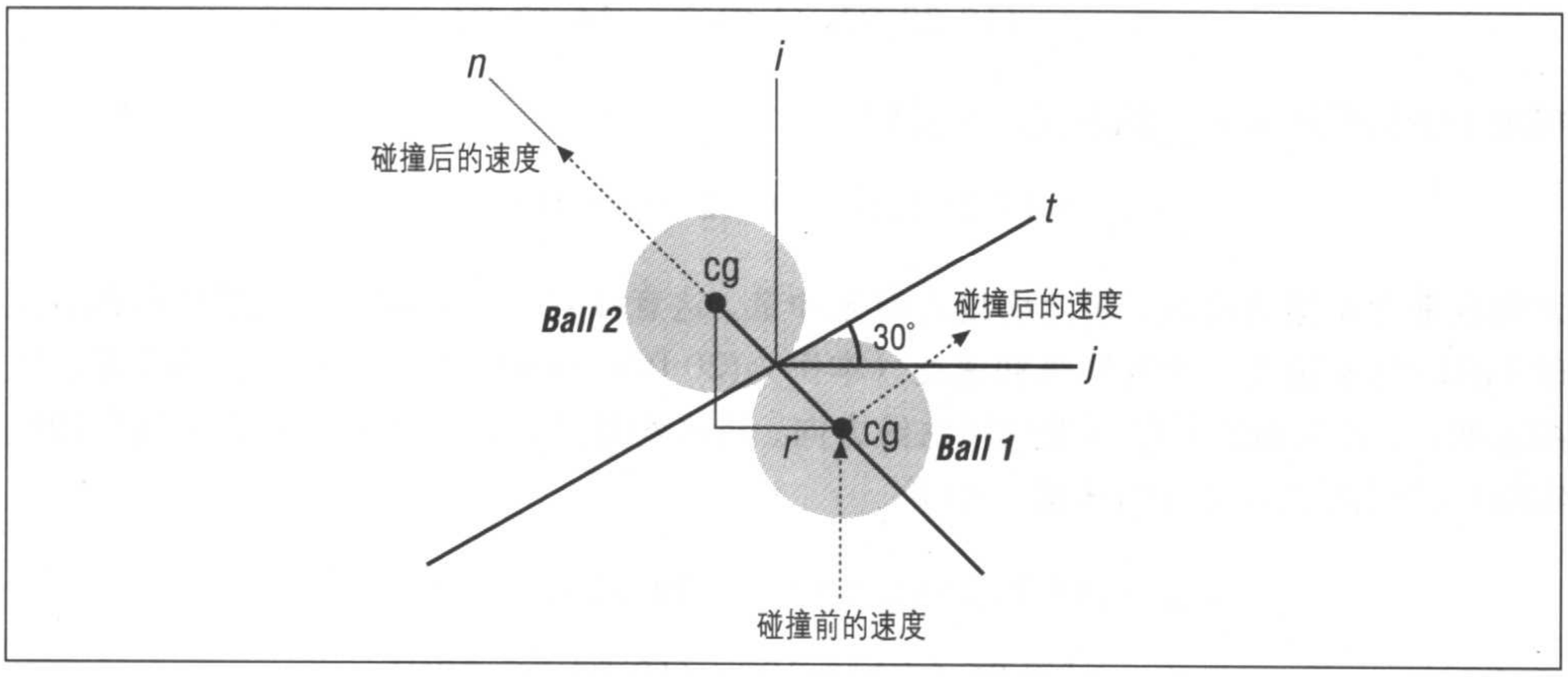



图 5-2：撞球碰撞范例

其中 $\boldsymbol{n}$ 是单位法线向量，$r$ 是球半径，而 $\boldsymbol{i}$ 和 $\boldsymbol{j}$ 分别表示 $x$ 与 $y$ 方向的单位向量。

现在有了碰撞的作用线，或单位法线向量，就可以求出于碰撞瞬间两球间的相对法线速度：

$$v_{rn} = [\boldsymbol{v}_{1-} - \boldsymbol{v}_{2-}] \cdot \boldsymbol{n}$$

$$v_{rn} = [(20\ \text{ft/s})\boldsymbol{i} + (0\ \text{ft/s})\boldsymbol{j}] \cdot [(0.866)\boldsymbol{i} - (0.5)\boldsymbol{j}]$$

$$v_{rn} = 17.28\ \text{ft/s}$$

请注意，因为球 2 起初为静止的，所以 $v_{2-}$ 等于 0。

接着可套用法线方向的动量守恒定律，如下：

$$m_1 v_{1n-} + m_2 v_{2n-} = m_1 v_{1n+} + m_2 v_{2n+}$$

注意，因为球是一样的，因此 $m_1$ 等于 $m_2$，而 $v_{2n-}$ 等于 0，解 $v_{1n+}$ 便得到：

$$v_{1n+} = v_{1n-} - v_{2n+}$$

要确实求出这些速度，需利用恢复系数的方程并代换 $v_{1n+}$。之后便可求出 $v_{2n+}$。步骤如下：

$$e = (-v_{1n+} + v_{2n+})/(v_{1n-} - v_{2n-})$$

$$ev_{1n-} = -(v_{1n-} - v_{2n+}) + v_{2n+}$$

$$v_{2n+} = v_{1n-}(e + 1)/2$$

$$v_{2n+} = (17.28 \text{ ft/s})(1.9)/2 = 16.43 \text{ ft/s}$$

利用上述结果代入 $v_{1n+}$ 的公式，可得到

$$v_{1n+} = 17.28 \text{ ft/s} - 16.42 \text{ ft/s} = 0.86 \text{ ft/s}$$

因为这是无摩擦力碰撞，因此切线方向无冲量。这意味着在该方向动量也是守恒的，且球 1 的切线末速度等于其切线初速度（本例为(20 ft/s)sin30° 即 10 ft/s）。而球 2 无切线初速度，因此其撞击后的速度仅沿法线方向。将这些结果转为 $xy$ 坐标（而非法线与切线坐标）得到两球撞击后的速度，如下：

$$\boldsymbol{v}_{2+} = (16.42 \text{ ft/s})\sin 60°\boldsymbol{i} - (16.42 \text{ ft/s})\cos 60°\boldsymbol{j}$$

$$\boldsymbol{v}_{1+} = [(0.86 \text{ ft/s})\cos 30° + (10 \text{ ft/s})\sin 30°]\boldsymbol{i} + [(-0.86 \text{ ft/s})\sin 30° + (10 \text{ ft/s})\cos 30°]\boldsymbol{j}$$

$$\boldsymbol{v}_{1+} = (5.43 \text{ ft/s})\boldsymbol{i} + (8.23 \text{ ft/s})\boldsymbol{j}$$

为更进一步说明这些碰撞反应定律的应用，来看另一个例子，这次是棒球棍与棒球间的碰撞，如图 5-3 所示。

为了有合理的正确度，球棒的动量在碰撞瞬间可视为与击打者分开。即可以假设球棒是自由移动的，且以靠近球棒尾端的点为中心旋转。假设球击中球棒上的击球甜点（sweet point），即打击中心附近的点（注 2）。进一步假设，球棒是水平挥动的且当棒球被击中

---

注 2：击打中心是位于自然振动的节点附近的点，且当球击中球棒时不会有力传到握柄处。若你曾不正确地击中棒球且你的手体验到振动的疼痛感，便会了解未击中击打中心的感觉了。

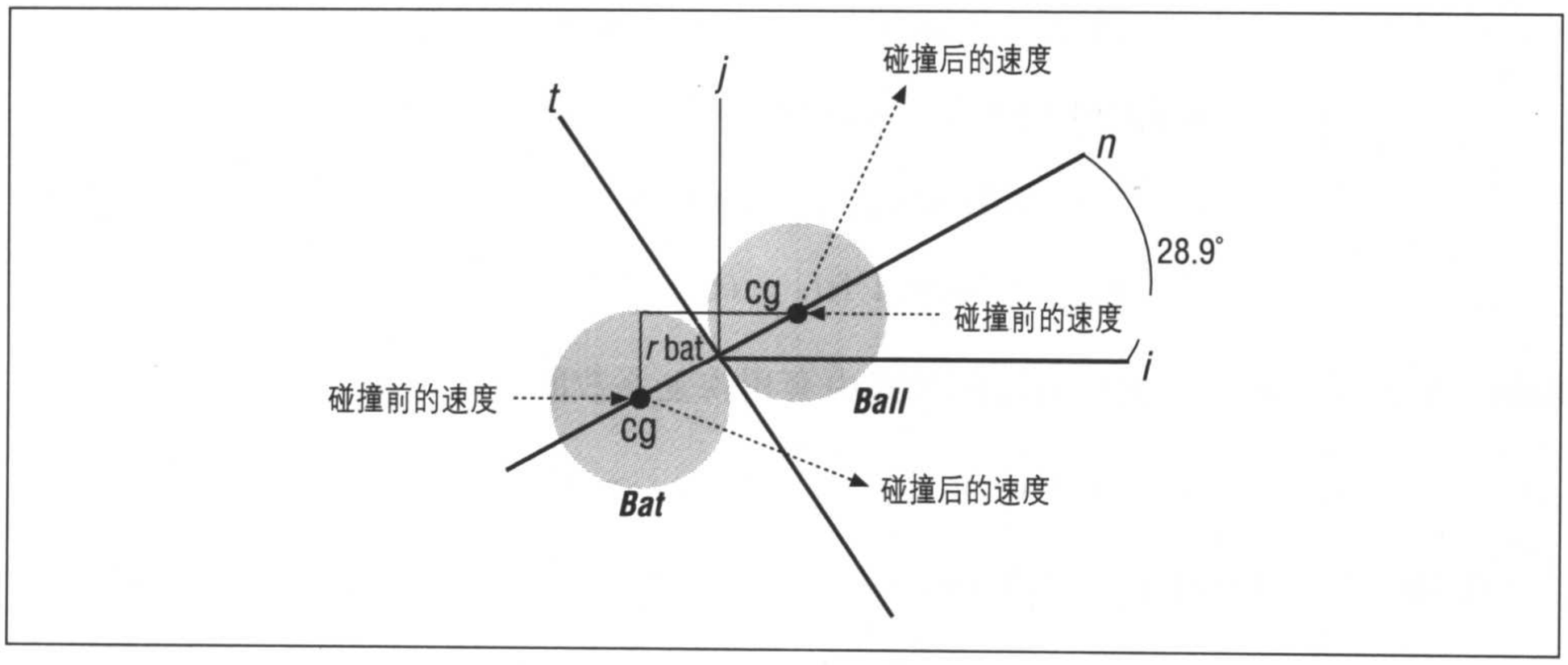



图 5-3：棒球与球棒的碰撞范例

时是水平行进的。球棒是最大直径2.75 in.的标准尺寸，且重量为36 oz（质量等于0.07 slug）。球也是半径1.47 in.的标准尺寸，且重量为5.125 oz（质量等于$9.96 \times 10^{-3}$ slug）。球被击中的瞬间，球速为132 ft/s（90 mph），而球棒在撞击点的速度为103 ft/s（70.2 mph）。碰撞的恢复系数等于0.46。在发生撞击的瞬间，棒球确实会压缩一小块，然而，此分析中假设球与球棒皆为刚体。最后，假设此撞击是无摩擦力的。

如前一个例子一样，撞击作用线顺着球与球棒重心的连线；因此单位法向量为：

$$\boldsymbol{n} = \left(\sqrt{\left((r_1 + r_2)^2 - r_1^2\right)}\boldsymbol{i} - r_1\boldsymbol{j}\right)/|\boldsymbol{n}|$$

$$\boldsymbol{n} = (0.875)\boldsymbol{i} - (0.484)\boldsymbol{j}$$

其中下标1和2分别代表球棒与球。

球棒与球间的相对法线速度为：

$$v_{rn} = [\boldsymbol{v}_{1-} - \boldsymbol{v}_{2-}] \cdot \boldsymbol{n}$$

$$v_{rn} = [(235\ \text{ft/s})\boldsymbol{i} + (0\ \text{ft/s})\boldsymbol{j}] \cdot [(0.875)\boldsymbol{i} - (0.484)\boldsymbol{j}]$$

$$v_{rn} = 205.6\ \text{ft/s}$$

球棒与球于法线方向的速度分量为：

$$v_{1n-} = \boldsymbol{v}_{1-} \cdot \boldsymbol{n} = 90.125\ \text{ft/s}$$

$$v_{2n-} = \boldsymbol{v}_{2-} \cdot \boldsymbol{n} = -115.5\ \text{ft/s}$$

套用法线方向的动量守恒定律并解$v_{1n+}$，得到：

$$m_1 v_{1n-} + m_2 v_{2n-} = m_1 v_{1n+} + m_2 v_{2n+}$$

$$(0.07\ \text{slug})(90.125\ \text{ft/s}) + (9.96 \times 10^{-3}\ \text{slug})(-115.5\ \text{ft/s})$$

$$= (0.07\ \text{slug}) v_{1n+} + (9.96 \times 10^{-3}\ \text{slug}) v_{2n+}$$

$$v_{1n+} = 73.691\ \text{ft/s} - (0.142\ \text{ft/s}) v_{2n+}$$

如前一个例子一样，在上列 $v_{1n+}$ 的公式中套用恢复系数的公式，得到：

$$e = (-v_{1n+} + v_{2n+})/(v_{1n-} - v_{2n-})$$

$$0.46 = [-73.691\ \text{ft/s} + (0.142\ \text{ft/s})\ v_{2n+} + v_{2n+}]/(90.125\ \text{ft/s} + 115.5\ \text{ft/s})$$

$$v_{2n+} = 147.34\ \text{ft/s}\ \text{而}\ v_{1n+} = 52.77\ \text{ft/s}$$

再一次强调，因为此撞击无摩擦力，所以每个物体仍保留原本的切线速度。就球棒而言，此分量为49.78 ft/s；而球的为−63.8 ft/s。取这些法线及切线分量并将它们转换成 $xy$ 坐标，得到球棒与球于碰撞之后瞬间的速度：

$$\boldsymbol{v}_{1+} = (70.25\text{ft/s})\boldsymbol{i} + (18\text{ft/s})\boldsymbol{j}$$

$$\boldsymbol{v}_{2+} = (98.2\text{ft/s})\boldsymbol{i} + (127\text{ft/s})\boldsymbol{j}$$

这两个例子都是利用正统的方法来做基础的碰撞分析。对它们也做了一个重要的假设：该碰撞无摩擦力。事实上，你知道撞球、棒球及球棒碰撞是会有摩擦力的，否则，你无法打出下塞球（English shot；又称旋转球）或投出上飘球。本章稍后将讨论如何分析含有摩擦力的碰撞。

# 线性及角冲量

上一节中，你可以利用动量守恒定律与恢复系数，动手解决特定的例子。如果编写的游戏其碰撞事件是妥善定义并可预期的，此方法就够用了。然而，如果编写的是物体（尤其是形状不规则的刚体）的碰撞侦测实时模拟，则要用到更全面的方法。这种方法会用到计算碰撞物体间的实际冲量公式，才能将此冲量套用于每个物体，且立即改变其速度。本节将导出线性及角冲量的方程，且在第十三章将示范如何以程序代码实现这些方程。

在处理粒子或球体时，惟一需要的冲量公式是线性冲量，可让你求出撞击后物体新的线性速度。所以本节要导出的第一个公式是两碰撞物体（如图5-4所示）间的线性冲量公式。

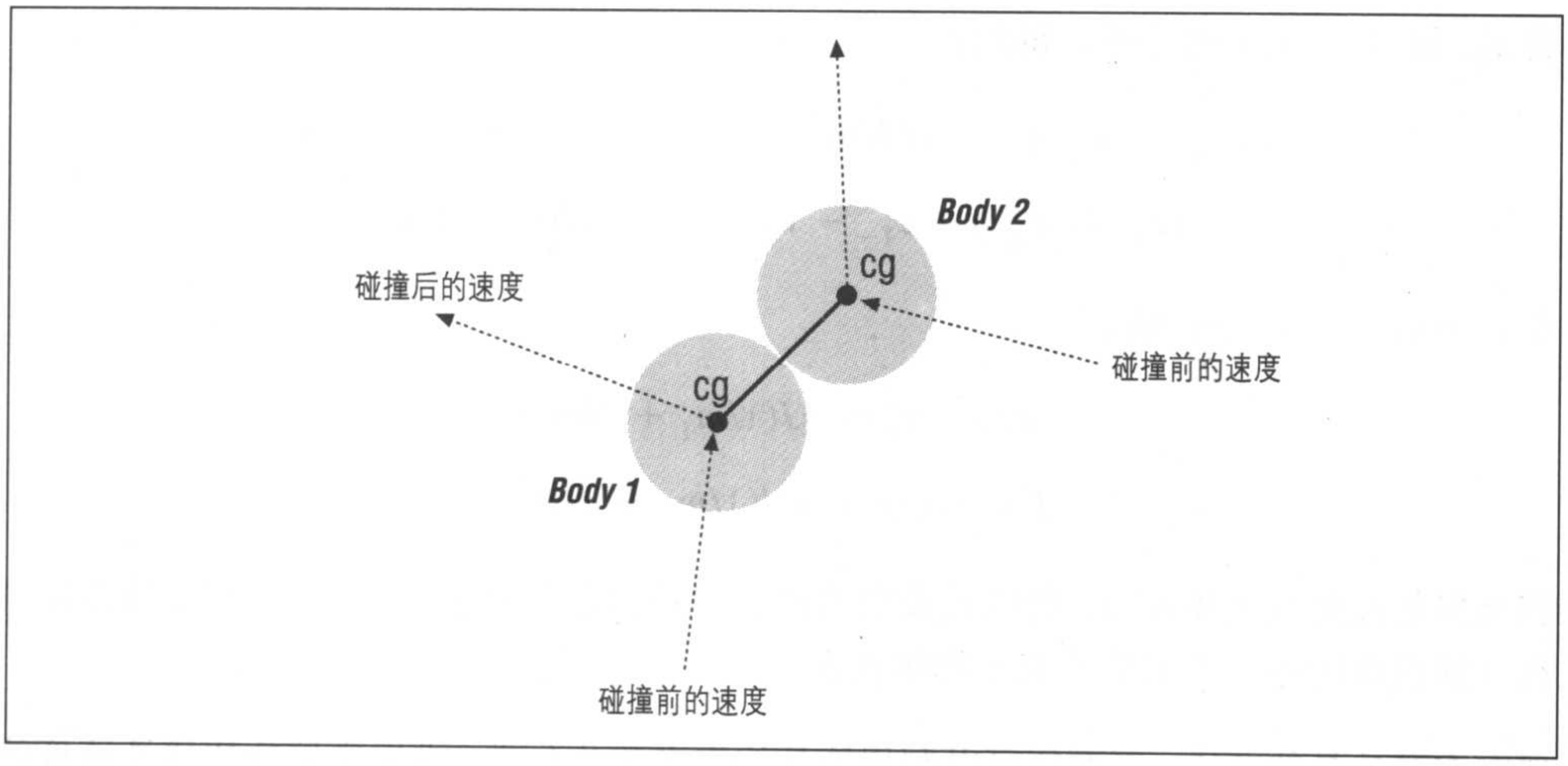



图 5-4：两碰撞粒子（或球体）

现在，假设碰撞无摩擦力且冲量的作用线沿着两物体质心连线。此线垂直于两物体的表面（接触面）。

要想导出线性冲量的公式，需考虑从冲量的定义而来的公式，以及恢复系数的公式。这里，以 $J$ 代表冲量：

$$J = m(v_+ - v_-)$$

$$e = -(v_{1+} - v_{2+})/(v_{1-} - v_{2-})$$

这些方程中，速度是指那些沿着撞击的作用线（本例是两物体的质心连线）的速度。因为相同的冲量（方向相反）作用于两物体，所以实际上有三个方程：

$$J = m_1(v_{1+} - v_{1-})$$

$$-J = m_2(v_{2+} - v_{2-})$$

$$e = -(v_{1+} - v_{2+})/(v_{1-} - v_{2-})$$

请注意，其中假设 $J$ 正方向作用于物体 1，而其反方向的 $-J$ 作用于物体 2。另外请注意到这些方程中有三个未知数：冲量及撞击后两物体的速度。因为有三个方程及三个未知数，可先重新排列冲量的方程，再将它们代入 $e$ 的方程，便能求出每个未知数。经过一些代数运算后，将导出 $J$ 的公式，然后可利用它来判断撞击后两物体的速度。步骤如下：

$$\text{物体 1：} v_{1+} = J/m_1 + v_{1-}$$

$$\text{物体 2：} v_{2+} = -J/m_2 + v_{2-}$$

将 $v_{1+}$ 和 $v_{2+}$ 代入 $e$ 的方程，得到：

$$e(v_{1-} - v_{2-}) = -[(J/m_1 + v_{1-}) - (-J/m_2 + v_{2-})]$$

$$e(v_{1-} - v_{2-}) + v_{1-} - v_{2-} = -J(1/m_1 + 1/m_2)$$

令 $v_r = (v_{1-} - v_{2-})$；则：

$$ev_r + v_r = -J(1/m_1 + 1/m_2)$$

$$J = -v_r(e + 1)/(1/m_1 + 1/m_2)$$

因为作用线垂直碰撞表面，所以 $v_r$ 是沿着撞击作用线的相对速度，而 $J$ 作用于撞击作用线（如前面所述，本例是垂直于物体表面上）。

现在有了冲量的公式，下面便可以利用冲量的定义及此公式，求出碰撞物体线性速度的变化。以下是两物体碰撞的情形下其计算过程：

$$\boldsymbol{v}_{1+} = \boldsymbol{v}_{1-} + (J\boldsymbol{n})/m_1$$

$$\boldsymbol{v}_{2+} = \boldsymbol{v}_{2-} + (-J\boldsymbol{n})/m_2$$

请注意，对第二个物体而言，负的冲量作用于其上，因为冲量等量作用于两物体但其方向相反。

当处理转动的刚体时，需导出包含角运动的冲量方程。你将利用此冲量求出物体碰撞后其线速度及角速度。考虑两物体在 $P$ 点碰撞，如图 5-5 所示。

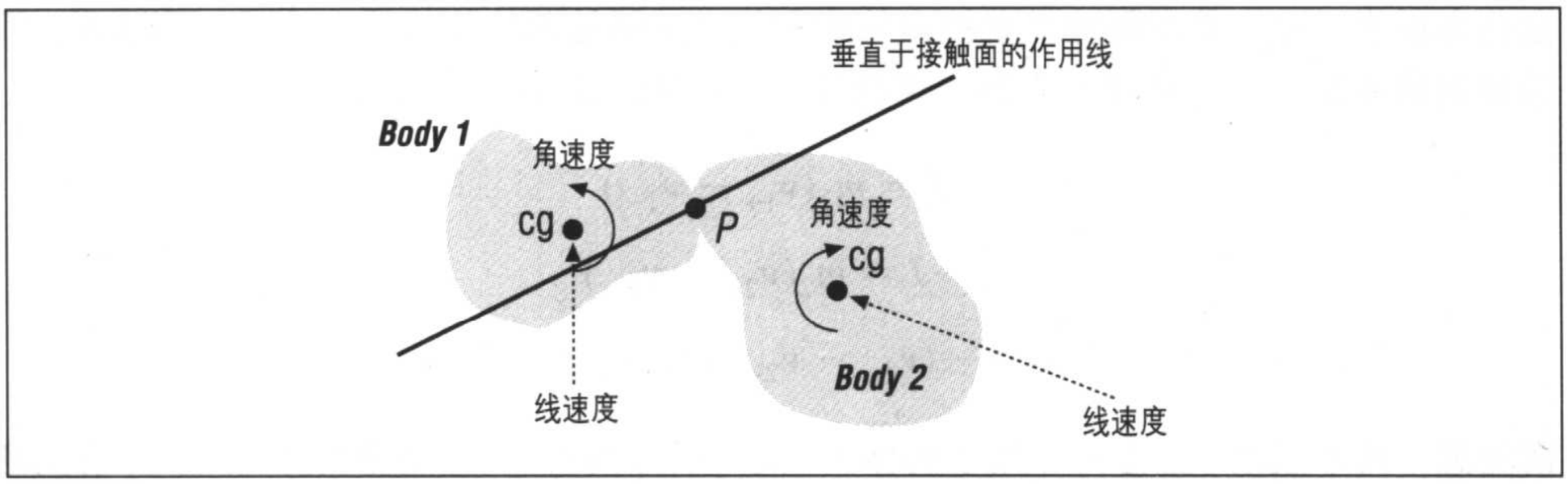



图 5-5：两碰撞刚体

此碰撞与先前讨论的有一点显著的不同，本例中在两物体接触点的速度是物体线速度和角速度的函数，而你必须利用下列公式，才能求出在两物体撞击点的速度：

$$\boldsymbol{v}_{\mathrm{p}} = \boldsymbol{v}_{\mathrm{g}} + (\boldsymbol{\omega} \times \boldsymbol{r})$$

此关系式中，$r$ 是物体重心到接触点 $P$ 的向量。

利用此公式，可改写两个撞击后的线速度与冲量及初速度的关系式，如下：

$$\text{物体 1：}\ v_{1g+} + (\omega_{1+} \times r_1) = J/m_1 + v_{1g-} + (\omega_{1-} \times r_1)$$

$$\text{物体 2：}\ v_{2g+} + (\omega_{2+} \times r_2) = -J/m_2 + v_{2g-} + (\omega_{2-} \times r_2)$$

这里多了两个未知数（两物体撞击后的角速度），这表示需要多两个方程。从角冲量的定义可得到这两个方程：

$$\text{物体 1}:(r_1 \times J) = I_1(\omega_{1+} - \omega_{1-})$$

$$\text{物体 2}:(r_2 \times -J) = I_2(\omega_{2+} - \omega_{2-})$$

而冲量引起的力矩是通过取冲量与距离（物体重心到撞击点）的外积求出的。

通过将这些方程与 $e$ 的方程结合，并依照导出线性冲量公式的相同程序，会得到包含线性与角运动的 $J$ 公式，可用来找出撞击之后物体的线速度及角速度。结果如下：

$$J = -v_r(e + 1)/\{1/m_1 + 1/m_2 + n \cdot [(r_1 \times n)/I_1] \times r_1 + n \cdot [(r_2 \times n)/I_2] \times r_2\}$$

其中 $v_r$ 是沿着撞击点 $P$ 的作用线的相对速度，而 $n$ 是沿着作用线且从撞击点指向物体 1 的单位向量。

有了 $J$ 的这个新公式，可利用这些公式求出碰撞物体的线速度和角速度的变化：

$$v_{1+} = v_{1-} + (Jn)/m_1$$

$$v_{2+} = v_{2-} + (-Jn)/m_2$$

$$\omega_{1+} = \omega_{1-} + (r_1 \times Jn)/I_{cg}$$

$$\omega_{2+} = \omega_{2-} + (r_2 \times -Jn)/I_{cg}$$

如前所述，到了第十三章将教你如何将冲量的公式实现为程序代码。

# 摩擦力

摩擦力作用于接触面并阻碍运动。当物体以任何碰撞类型（直接碰撞除外）相撞时，在接触的短暂片刻，它们便会遭受作用于接触面切线方向的摩擦力。此切线方向的力不只会改变碰撞物体的切线速度，也会产生改变物体角速度的力矩（转矩）。此切线冲量与法线冲量结合成总碰撞冲量，其实际的作用线不再垂直于接触面。

实际上我们很难为此碰撞摩擦力定量，因为若碰撞时的摩擦力未超过最大静摩擦力，则摩擦力不会是一个常数。另一个复杂的问题是当物体碰撞时会变形，这会产生额外的阻力来源。因为摩擦力是接触面之间的法线作用力（正向力）的函数，可知法线作用力与摩擦力的比例即等于摩擦系数。如果假设碰撞可套用动摩擦系数，则此比例是固定的：

$$\mu = F_{\mathrm{f}}/F_n$$

这里，$F_{\mathrm{f}}$是切线摩擦力而$F_n$是法线冲力（正向力）。你可以扩大解释：切线冲量与法线冲量的比例等于摩擦系数。

考虑高尔夫球与杆头的碰撞，如图 5-6 所示。

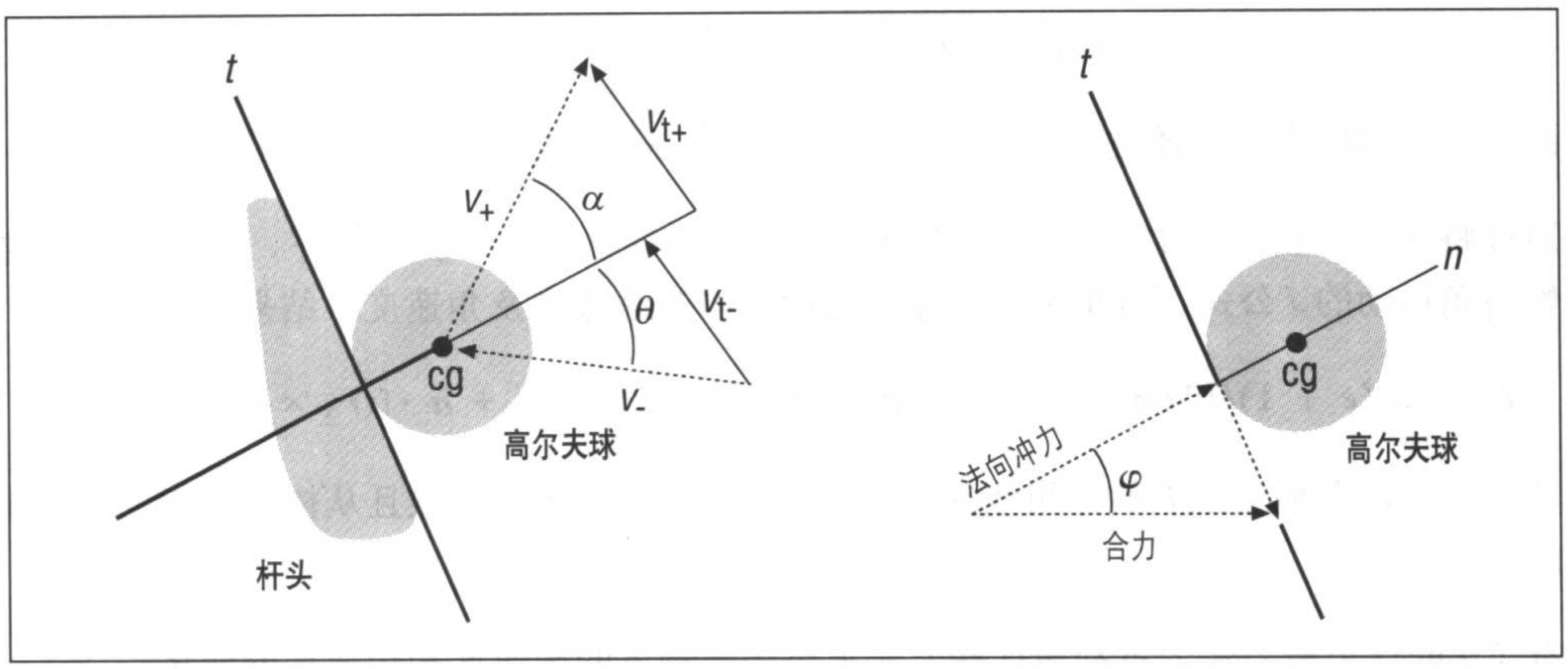



图 5-6：高尔夫球碰撞

在左边的速度分解图中，$v_-$代表撞击瞬间球与杆头的相对速度，$v_+$代表撞击后的球速，而$v_{t-}$与$v_{t+}$分别代表在撞击时及撞击后瞬间球速的切线分量。

若此为无摩擦力的碰撞，则$v_{t-}$和$v_{t+}$会相等，角$\alpha$与$\theta$也会相等。然而，若有摩擦力，球的切线速度会减少，使$v_{t+}$小于$v_{t-}$，这也表示$\alpha$会小于$\theta$。

图 5-6 右边的力分解图说明了与此有摩擦力碰撞相关的力。因为切线摩擦力与正向力之比例等于摩擦系数，由此可得出角$\varphi$与摩擦系数的关系式：

$$\tan\varphi = F_{\mathrm{f}}/F_n = \mu$$

此摩擦力除了改变球在切线方向的线速度外，也会改变球的角速度。因为摩擦力作用于球的表面，距其重心有一点距离，所以产生绕球重心的力矩（转矩），造成球的自旋。若用与第四章滚动圆柱范例类似的方法，可得出球的角速度与正向冲力或冲量相关的方程：

$$\sum M_{cg} = F_f r = I_{cg} d\omega / dt$$
$$\mu F_n r = I_{cg} d\omega / dt$$
$$\mu F_n r \, dt = I_{cg} d\omega$$
$$\int_{t-}^{t+} F_n \, dt = I_{cg} / (\mu r) \int_{\omega-}^{\omega+} d\omega$$

请注意等式左边的积分就是法线冲量；因此，

$$\text{冲量} = I_{cg}/(\mu r)(\omega_+ - \omega_-)$$

$$\omega_+ = (\text{冲量})(\mu r)/I_{cg} + \omega_-$$

此关系式看起来与本章前面的角冲量方程很类似，并且可以利用它来估算特定碰撞问题中摩擦产生的自旋。

回到前一节中包含线性及角运动的冲量 $J$ 方程，为了方便这里再列出来：

$$J = -\boldsymbol{v}_r (e + 1)/[1/m_1 + 1/m_2 + \boldsymbol{n} \cdot (\boldsymbol{r}_1 \times \boldsymbol{n})/\boldsymbol{I}_1 + \boldsymbol{n} \cdot (\boldsymbol{r}_2 \times \boldsymbol{n})/\boldsymbol{I}_2]$$

此公式可求得法线方向的碰撞冲量。要想了解如何将摩擦力代到此公式，需记住摩擦力作用与接触面相切，将摩擦力与正向冲力结合产生碰撞新的实际作用线，以及摩擦力（切线冲量）为正向力（正向冲量）与摩擦系数的函数。考虑这些因素，计算两碰撞物体线速度及角速度变化的新方程如下：

$$\boldsymbol{v}_{1+} = \boldsymbol{v}_{1-} + [J\boldsymbol{n} + (\mu J)\, \boldsymbol{t}]/m_1$$

$$\boldsymbol{v}_{2+} = \boldsymbol{v}_{2-} + [-J\boldsymbol{n} + (\mu J)\, \boldsymbol{t}]/m_2$$

$$\omega_{1+} = \omega_{1-} + \{\boldsymbol{r}_1 \times [J\boldsymbol{n} + (\mu J)\, \boldsymbol{t}]\}/I_{cg}$$

$$\omega_{2+} = \omega_{2-} + \{\boldsymbol{r}_2 \times [-J\boldsymbol{n} + (\mu J)\, \boldsymbol{t}]\}/I_{cg}$$

在这些方程中，$\boldsymbol{t}$ 是单位切线向量，相切于碰撞表面且与单位法线向量垂直。若已知单位法线向量及相同平面的相对速度向量，便可求出切线向量：

$$\boldsymbol{t} = (\boldsymbol{n} \times \boldsymbol{v}_r) \times \boldsymbol{n}$$

$$\boldsymbol{t} = \boldsymbol{t} / |\boldsymbol{t}|$$

对于许多将面对的问题，你也许可以在碰撞反应中适度地忽略摩擦力，因为其影响或许比法线冲量的影响小很多。然而，对于某些题型，摩擦力是必要的。例如，高尔夫球的飞行轨迹得看碰撞结果造成的自旋。下一章将涵盖抛体运动，再讨论自旋是如何影响轨迹的。

# 第六章

# 抛体

从本章起，会用一系列的章节来探讨现实世界中的特定现象与系统，例如抛体运动与飞机的飞行等。让读者了解其实际的行为如何运作后，才能在游戏中精确地模拟这些系统或类似的系统。本章会介绍许多较实际的公式与资料，而不是那些纯理想化的公式。接下来的几个章节中所选用的范例，会介绍一些存在于许多系统中的常用力学原理与现象。但切记，文中所讨论的原理并不是仅限于文中所提到的范例。例如第八章会详细讨论船的浮力问题，但是浮力并非只限于船，任何沉浸在流体中的物体都会受到浮力的影响。第七、九、十章所讨论的主题也是如此。读者在读过这些章节后，可以进一步地运用在其他类似的系统上。

一旦了解这些系统会面临怎样的问题后，在解释模拟结果时就会有较好的立足点，并能进一步判断这些结果是否具有意义，或是否接近实际行为。同时，在了解这些系统如何运作后，才能知道影响系统最重要的因素，并以适当的假设来简化问题。基本上，在设计与最佳化程序代码时，必须要知道哪些因素可以除掉而不会牺牲正确性。同时这也牵涉到参数调整的主题。

接下来的几章将详细介绍一些物理现象，让你能够根据所要达到的行为来调整模型。如果在模拟程序中，建立一些相似物体的模型，但又希望每个模型都有些微的不同，就必须微调每个物体上所受的力，使每个物体能有不同的行为。因为在这样的模拟中，力控制了整个物体的行为，所以我们会着重于力的运算，知道这些力是如何运作的，以及为什么会这样运作。而不像第三章只是用这些理想化的公式。另外，参数调整并不是只局限于调整模组的行为，还包含处理数值的问题，例如在积分演算法中的数值稳定性。在第十二章到第十七章中，会介绍一些模拟的范例，届时针对这些问题会有更进一步的讨论。

本章全部的篇幅都是在介绍抛体运动，因为在设计的游戏中，许多问题其实都属于此类。另外，控制着抛体运动的力也会影响其他未必是抛体的系统。例如抛体所受的空气阻力，在飞机、汽车等其他物体在流体（空气或水）中运动也会受到类似的阻力。

抛体就是短时间受力（也就是第五章中所讲的冲量）而运动的物体。抛体在发射的阶段中，会受一开始的冲量作用而进入运动阶段，之后就不再有其他推力作用在抛体上。正如在第二章与第四章的范例中所介绍的，会有其他的力作用在抛体上（目前暂不讨论自力推进的抛体，如火箭，因为它们本身具有燃料所产生的推力，所以在燃料耗尽之前，它们的行为并不遵循“典型的”抛体运动）。

最简单的例子是，如果忽略空气动力的影响，作用在抛体上的力除了一开始的冲力之外，还有地球的重力。如果抛体接近地球表面，这类的问题可以简化成定加速度的问题。假设地球的表面是水平的，也就是说相对于地球的整个表面而言，抛体的抛射范围不至于大到需考虑地球的弯曲度，在这样的条件下可对抛体运动做以下的描述：

- 运动轨道呈抛物线状。
- 在给定的抛射速度下，抛射角为45°时，抛射距离可达最远。
- 当着地点与抛射点位于同一水平高度时，抛体在着地时的速度等于抛射速度。
- 在运动轨道的最高点，抛体速度的垂直分量为0。
- 当着地点与抛射点位于同一水平高度时，抛体从抛射点到达最高点所需时间，与抛体从最高点掉落到着地点所需的时间相同。
- 抛体从最高点掉落到着地点所需的时间，与物体从最高点自由（垂直）落到地面所需时间相同。

## 基本的抛体轨道

在这里将抛体运动的问题归纳为以下4类：

- 目标与抛射点等高。
- 目标高于抛射点。
- 目标低于抛射点。
- 抛射体由高于目标的移动系统（如飞机）落下。

在第一类问题中，目的点与抛射点位于同样的水平面上。如图6-1所示，$v_0$是抛体在发射时的初速度，$\varphi$是初射角，$R$是抛射距离，$h$则是抛体在运动轨道中最高点的高度。

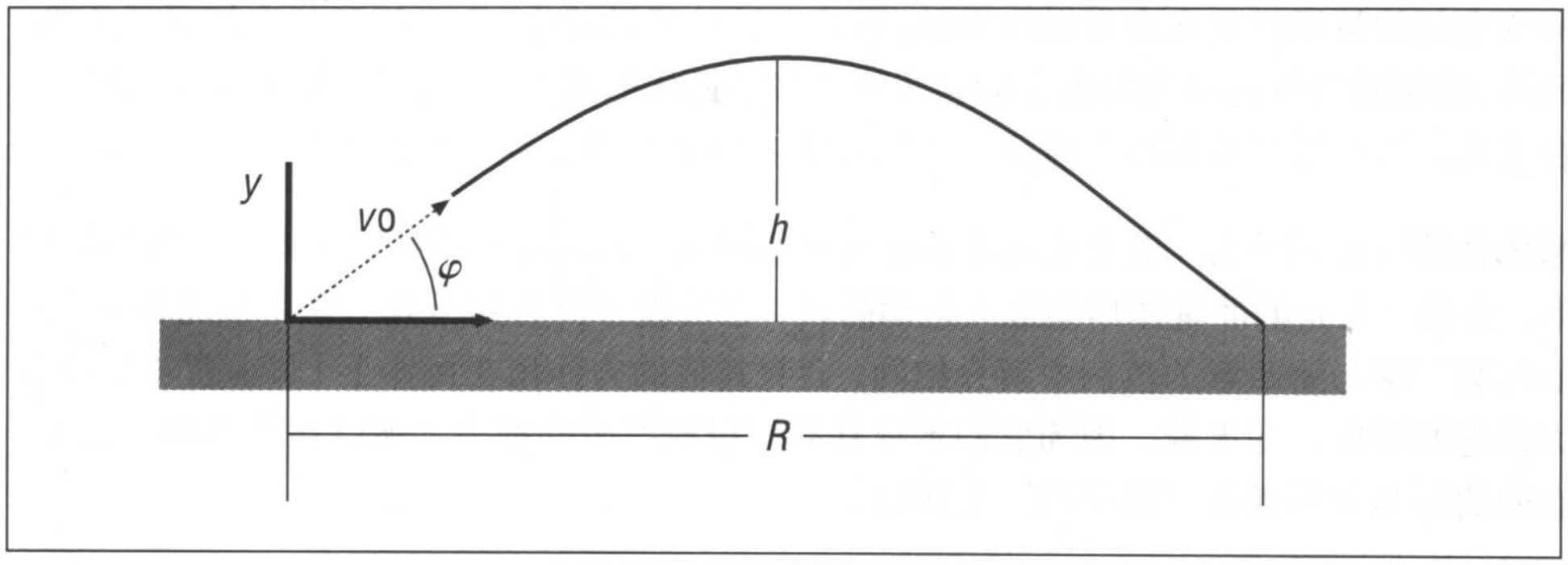



图 6-1：目标与发射点等高

要解决这类问题，可以利用表 6-1 中所列的公式。其中 $t$ 表示在抛射后的任何一个时间点，$T$ 则表示从抛射到落地的总时间。

表 6-1：公式 —— 目标与抛射点等高度

| 欲求值 | 运用公式 |
|---|---|
| $x(t)$ | $(v_0\cos\varphi)t$ |
| $y(t)$ | $(v_0\sin\varphi)t - (gt^2)/2$ |
| $v_x(t)$ | $v_0\cos\varphi$ |
| $v_y(t)$ | $v_0\sin\varphi - gt$ |
| $v(t)$ | $\sqrt{v_0^2 - 2gtv_0\sin\varphi + g^2t^2}$ |
| $h$ | $(v_0^2\sin^2\varphi)/(2g)$ |
| $R$ | $v_0T\cos\varphi$ |
| $T$ | $(2v_0\sin\varphi)/g$ |

在套用这些公式时，要注意单位是否一致。如果使用英制系统，所有关于长度与距离的值都必须以 ft 为单位，时间的单位是 s，速度的单位是 ft/s，加速度则是 $ft/s^2$。如果是用国际单位制系统，则所有关于长度与距离的值都必须以 m 为单位，时间的单位是 s，速度的单位是 m/s，加速度则是 $m/s^2$。因此在英制系统中，g 值为 32.2 $ft/s^2$，在国际单位制系统中，g 值则为 9.8 $m/s^2$。

在第二类问题中，抛射点所位于的水平面比目标低。如图 6-2 所示，抛射点的 $y$ 轴坐标值比目标的 $y$ 轴坐标值小。

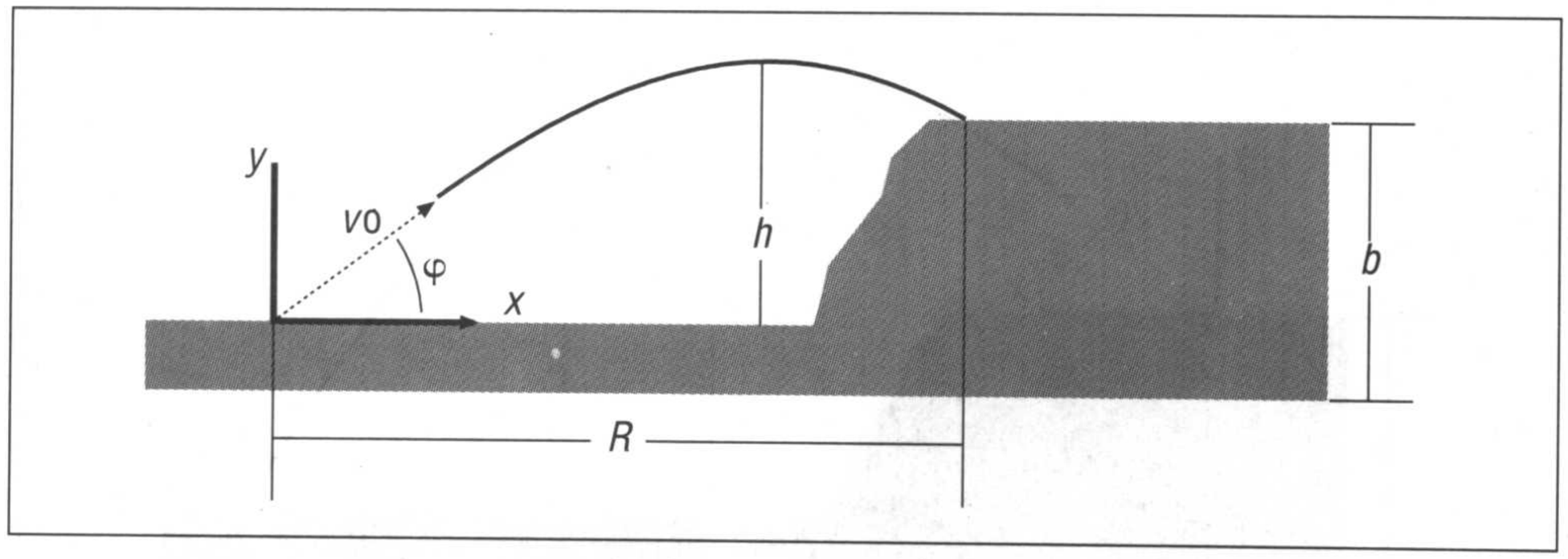



图 6-2：目标高于抛射点

对这一类问题，可利用表 6-2 中所列公式来解决。请注意这些公式大部分与表 6-1 中的公式相同。

表 6-2：公式 —— 目标高于抛射点

| 欲求值 | 运用公式 |
|---|---|
| $x(t)$ | $(v_0\cos\varphi)t$ |
| $y(t)$ | $(v_0\sin\varphi)t - (gt^2)/2$ |
| $v_x(t)$ | $v_0\cos\varphi$ |
| $v_y(t)$ | $v_0\sin\varphi - gt$ |
| $v(t)$ | $\sqrt{v_0^2 - 2gtv_0\sin\varphi + g^2t^2}$ |
| $h$ | $(v_0^2\sin^2\varphi)/(2g)$ |
| $R$ | $v_0T\cos\varphi$ |
| $T$ | $(v_0\sin\varphi)/g + \sqrt{[2(h-b)]/g}$ |

事实上，惟一的不同在于 $T$ 的公式，因为目标与抛射点之间的高度有所不同。

在第三类问题中，目标所位于的水平面比抛射点低，也就是目标的 $y$ 轴坐标值比抛射点的 $y$ 轴坐标值小（如图 6-3 所示）。

表 6-3 中列出了解决这一类问题的公式。同样地，几乎所有的公式都与表 6-1 中所列的相同。

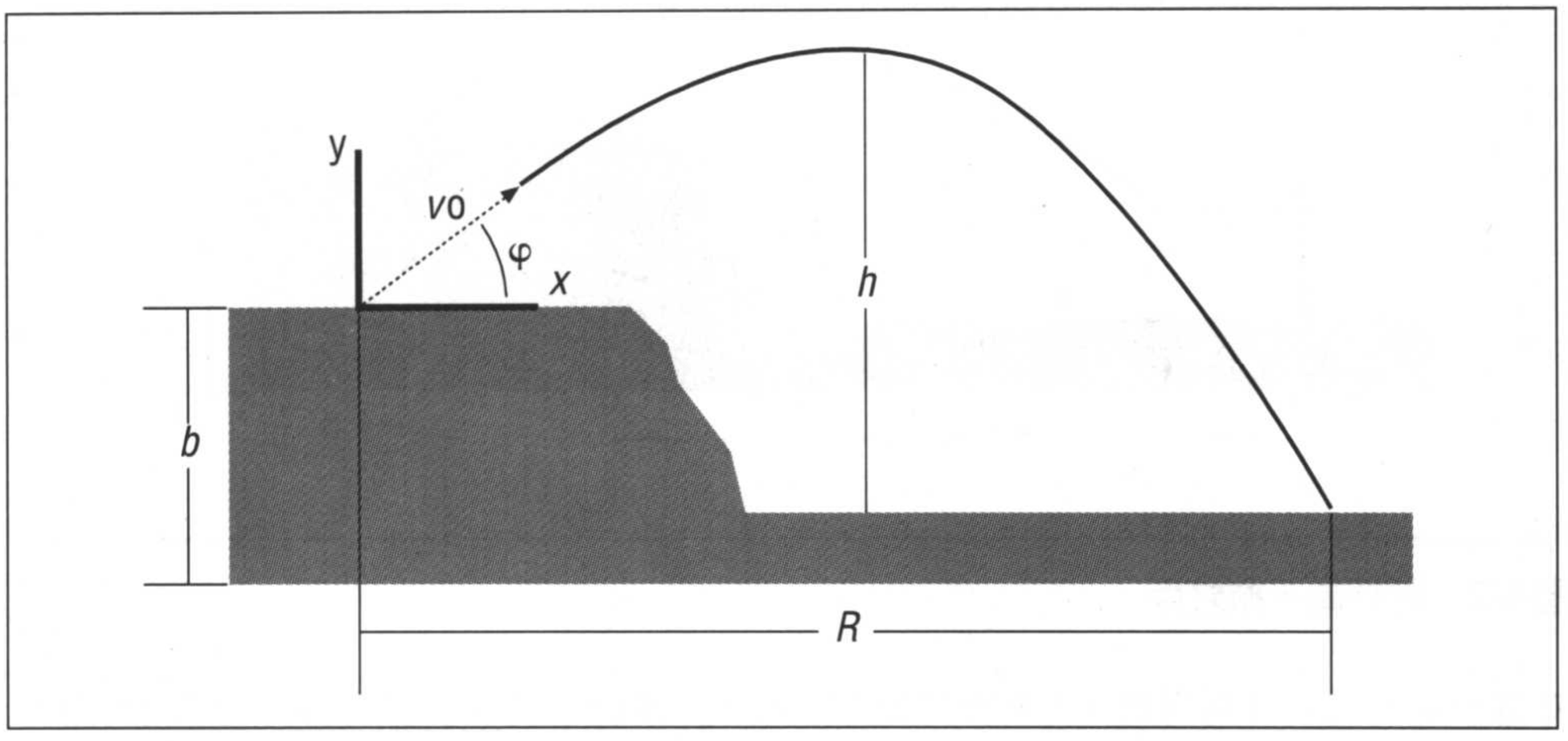



图 6-3：目标低于抛射点

表 6-3：公式 —— 目标低于抛射点

| 欲求值 | 运用公式 |
| --- | --- |
| $x(t)$ | $(v_0\cos\varphi)t$ |
| $y(t)$ | $(v_0\sin\varphi)t - (gt^2)/2$ |
| $v_x(t)$ | $v_0\cos\varphi$ |
| $v_y(t)$ | $v_0\sin\varphi - gt$ |
| $v(t)$ | $\sqrt{v_0^2 - 2gtv_0\sin\varphi + g^2t^2}$ |
| $h$ | $(v_0^2\sin^2\varphi)/(2g) + b$ |
| $R$ | $v_0T\cos\varphi$ |
| $T$ | $(v_0\sin\varphi)/g + \sqrt{(2h)/g}$ |

与第二类问题相比，惟一不同的还是 $T$ 和 $h$ 的公式，因为目标与抛射点之间的高度有所不同（这次目标比抛射点低）。

最后，第四类问题为抛体由高于目标的移动系统（像飞机）上落下。在这种情形下，抛体的初速度为水平方向，而且大小与其原本所在的移动系统的速度相同。图 6-4 说明了此类问题。

表 6-4 中列出了解答此类问题的公式。请注意当 $v$ 为 0 时，可以将此类问题简化成普通的自由落体问题。在这种状况下，抛体只会往下直落。

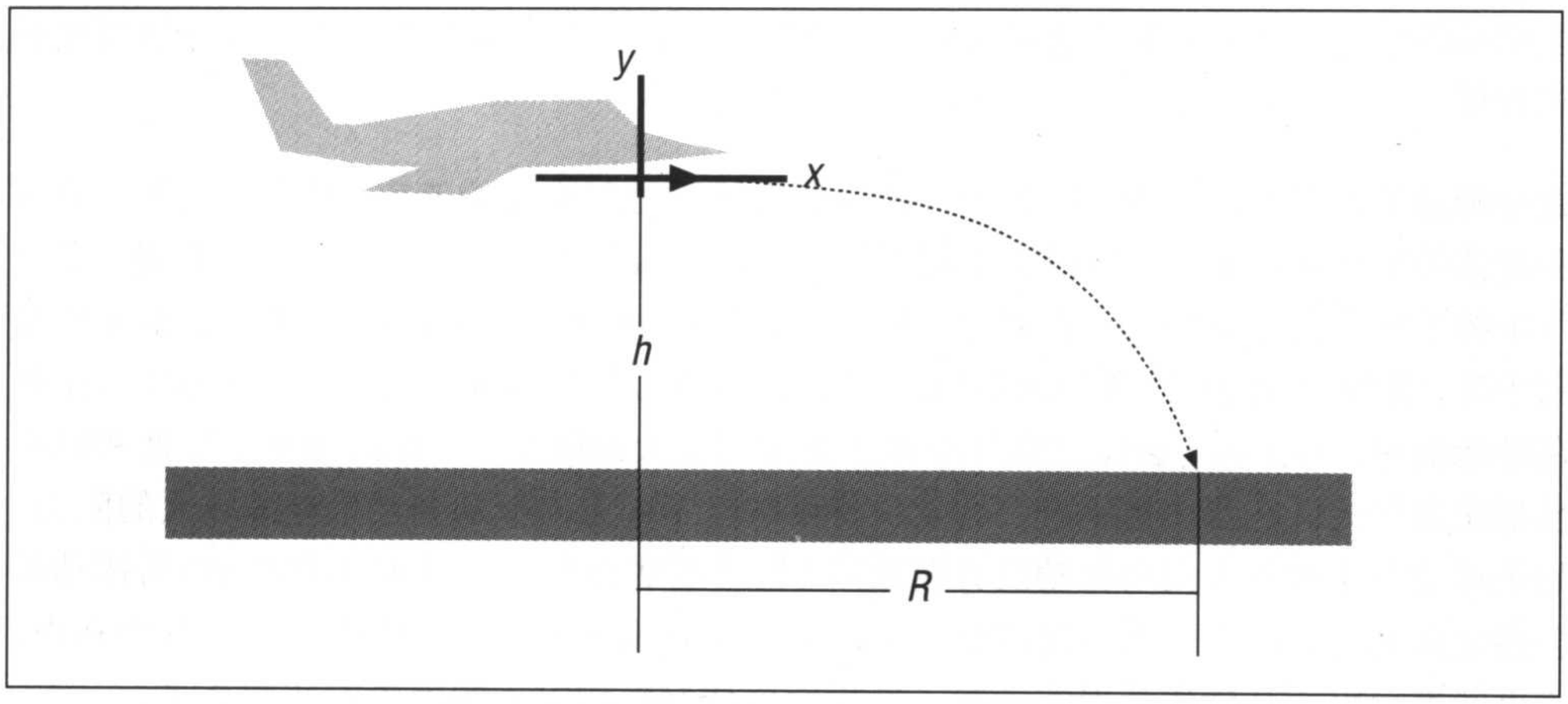



图 6-4：抛体从移动系统中落下

表 6-4：公式—— 由移动系统中落下的抛体

| 欲求值 | 运用公式 |
|---|---|
| $x(t)$ | $v_0 t$ |
| $y(t)$ | $h - (gt^2)/2$ |
| $v_x(t)$ | $v_0$ |
| $v_y(t)$ | $-gt$ |
| $v(t)$ | $\sqrt{v_0^2 + g^2 t^2}$ |
| $h$ | $(gt^2)/2$ |
| $R$ | $v_0 T$ |
| $T$ | $\sqrt{(2h)/g}$ |

如果在你写的游戏中对抛体运动的计算不需要很精确，例如不用去考虑运动中的抛体也会受其他力作用的话，这些公式就已经很够用了。但是如果需要更加精确的计算，就必须利用第四章中的范例，考虑其他力的作用。

## 阻力

在第三章及第四章中曾经介绍黏滞流体动阻力的理想方程，也介绍了如何在抛体运动方程中实现阻力。这些在第四章的范例程序中已经讨论过了。阻力如同其他力一样也是个向量，同样作用在速度向量的作用线上，只是方向相反。如前所述，在游戏的模拟程序中运用这些方程时，不可能做到十全十美。虽然不可能在整本书中讨论所有与流体动力

学相关的主题，但还是要让读者对阻力有更进一步的认识，而不仅限于之前介绍的理想化方程。

由分析法的结果可以得知，物体通过流体所受的阻力与本身的速率、大小、外形、密度和流体的黏滞度成正比。从现实生活中的经验也可以得到这样的结论。举例来说，在空气中挥手时，会感觉到一点点阻力。然而从时速 60 mph（每小时 60 英里）的车子中伸出手时，你的手会感受到较大的阻力。这是因为阻力是与速率成正比的。在水中（比如在游泳池中）挥手时，会比在空气中挥手感觉到更大的阻力。这是因为水的密度和黏滞性都比空气大。在水中挥手时，会发现将手掌以不同方向挥动会产生差异极大的阻力。如果挥动时手掌的方向与挥动的方向相同（也就是掌心向前），比将手掌转90°后挥动时会感觉到更大的阻力。在后者的状况下，就好像对水做出空手道中以手刀砍劈的动作。这也就表示阻力是物体外形的函数。这样你应该对影响阻力的因素有些概念了。

为了辅助关于流体动阻力的讨论，来看看一个球体经过流体（如水或空气）时周围的流动情形。如果球体慢慢地在流体中移动，则周围流体的流动模式看起来如图 6-5 所示。

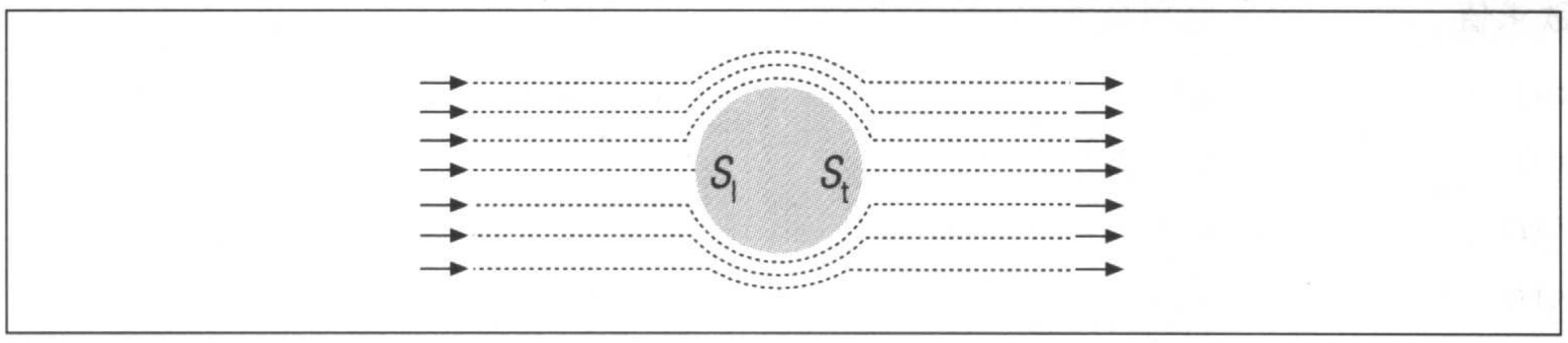



**图 6-5：球体缓慢移动时周围的流动模式**

贝努利定律（压力与流体速度的关系）指出：当流体流经球体而加速时，流体（本身的）的压力会下降。这个方程（由 Daniel Bernoulli 在西元 1738 年发表）可以应用在无摩擦且不能被压缩的流体上，这个方程如下所示（注 1）：

$$P/\gamma + z + V^2/(2g) = 常数$$

其中的$P$是流体容积中某个考虑点的压力，$\gamma$是流体的相对体积质量，$z$是某个考虑点的高度，$V$是某个考虑点上流体的速度，而 g 是重力加速度。正如你能看到的，如果方程左方维持固定，并假设$z$是常数，则当速度加快时，压力就会减小。同样地，如果压力增加，则速度就会减慢。

---

注 1：　对于会产生摩擦的真实流体，此方程会有额外的项目，用来处理因为摩擦力而产生的能量损耗。

请参考图 6-5，压力在停滞点 $S_1$ 会达到最大值，从球体的前端开始向外递减，到球体的后端才又慢慢增加。在无摩擦的理想流体中，压力在球体后方会完全恢复，因此也存在一个后端停滞点 $S_t$，此点上的压力等于球体前端停滞点 $S_1$ 的压力。因为球体前后的压力量值相等且方向相反，因此球体上没有净阻力。

在球体上下的压力会小于停滞点的压力，因为流体的速度在顶端及底部为最大值。由于在球体的状况下，流体的流动是对称的，因此在球体的上下端也没有净压力差。

在真实的流体中存在的摩擦力，会影响球体附近的流动让压力在球体的后方不能完全地恢复。当流体流经球体时，会有一层薄薄的流体因为摩擦力而粘在球体的表面上。在此边界层（boundary layer）上，流体的速度会由球体表面的 0 变化到理想的自由流动速度，如图 6-6 所示。

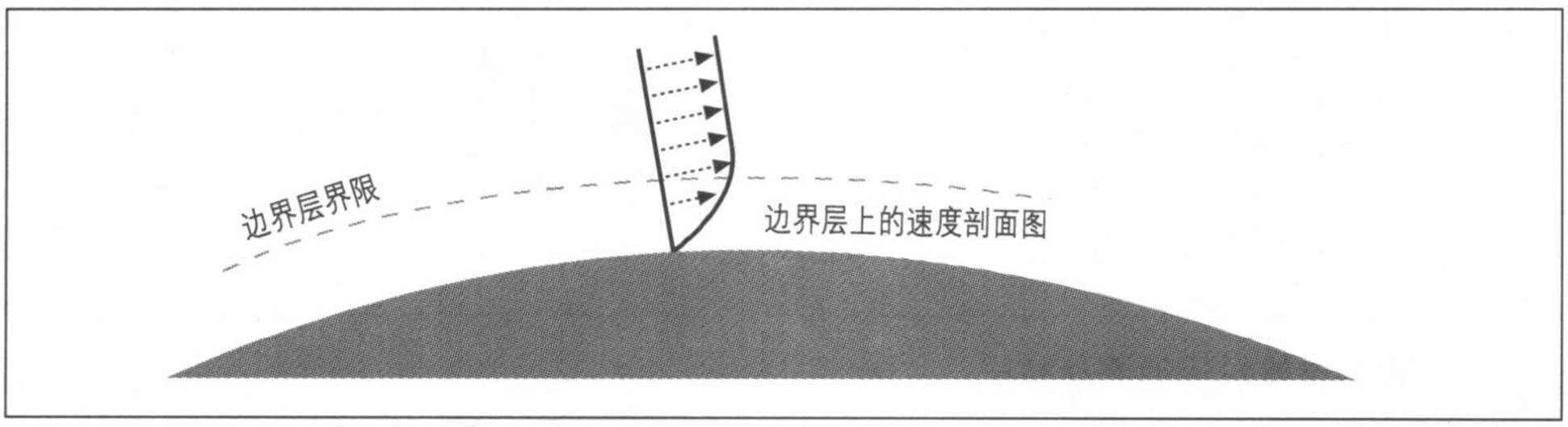



图 6-6：边界层内的速度梯度

速度梯度代表动量从球体转移到流体，并产生阻力的摩擦分力。因为有一些流体“粘”在球体的表面上，你可以把这些能量看做是为了加速这些流体，让它们能随着球体移动而转移（如果边界层上的流动是层流，流“层”间的黏滞剪应力（译注 1）会产生摩擦阻力。当流动为紊流或有速度梯度时，动量的转移也会产生摩擦阻力）。

当流体往后移动到球体后端时，边界层的厚度会增加并且无法再维持其对球体表面的附着，因此在某个点上就会被分开。在此分离点之后，流体会产生紊流，称之为紊流尾流（turbulent wake）。在此区域内的压力比球体前端的压力小。这种压力差产生阻力的压力分力。图 6-7 画出了这些流动模式。

对于移动缓慢的球体，分离点约在前端的 80° 左右。

---

译注 1：剪应力（shear stress）：当力作用于物体的一平面上，并使物体产生形变；此作用力方向与作用的面平行，用这个力除以其作用面积，就称剪（切）应力。

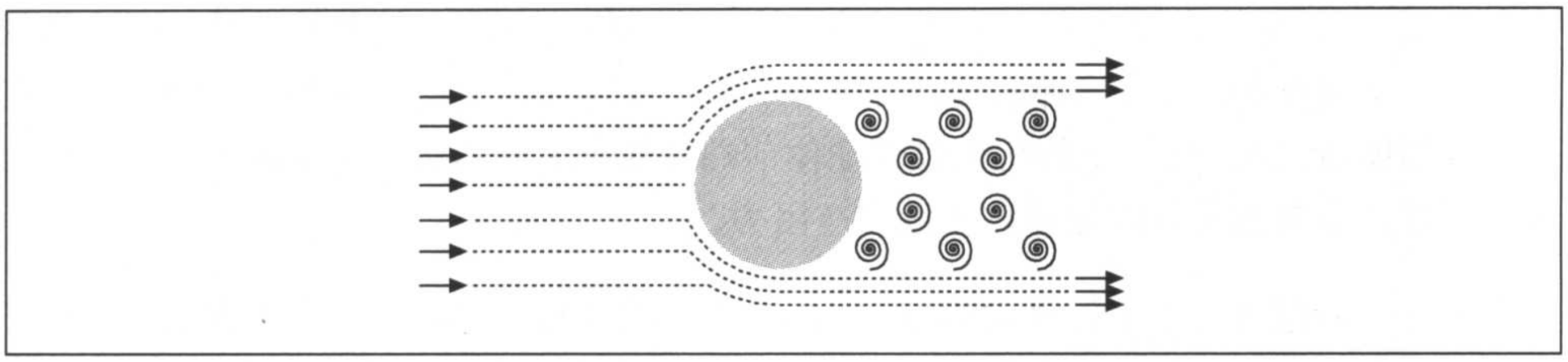


图 6-7：球体附近产生分离的流动模式

现在如果改用粗糙表面的球体，也会影响周围的流动。和预期的一样，粗糙的球体会有更大的摩擦阻力分力。然而更重要的是，流体附着在球体上的时间更长，而分离点会推向更后方，约在 115° 的位置，如图 6-8 所示。

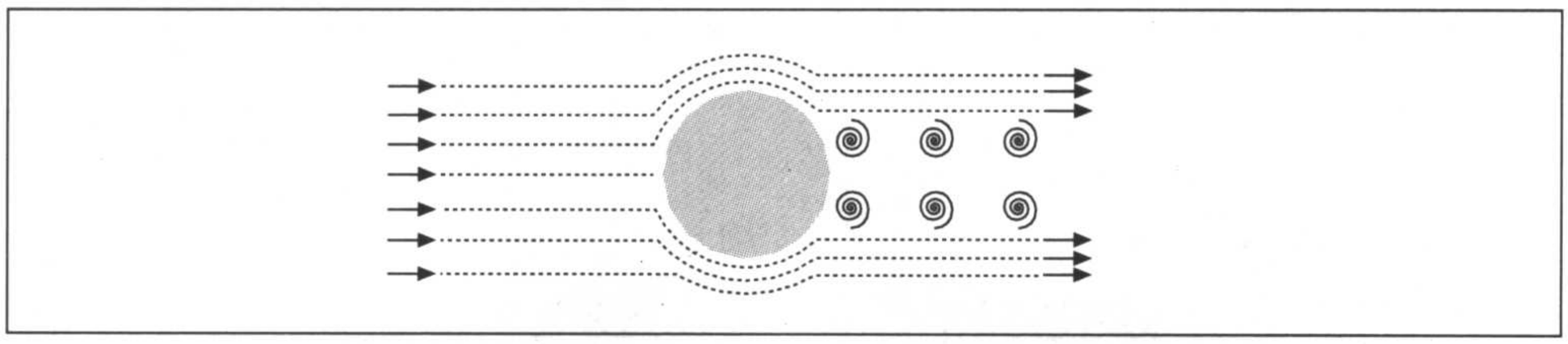


图 6-8：粗糙球体附近的流动模式

这样一来会减少紊流尾流以及压力差的大小，因此会减小压差阻力。听起来有点矛盾，但这是事实，而且所有的物体都一样，有点粗糙的球体比光滑的球体会有较小的总阻力。你可曾想过为什么在高尔夫球上会有小凹洞？我想这就是答案了。

球体的总阻力大多根据球体附近的流动状况（层流或紊流）而定。这从实验数据中很容易看出来。图 6-9 是典型的球体总阻力系数曲线图，是以雷诺数（Reynold's number）的函数画出来的。

雷诺数（通常以 $N_r$ 或 $R_n$ 表示）是无因次的数，表示物体周围流体流动的速率。它不仅仅是速率的测量值，因为雷诺数包含了物体的特征长度和流体的黏度与密度。雷诺数的公式是：

$$R_n = (vL)/\upsilon$$

或是

$$R_n = (vL\rho)/\mu$$

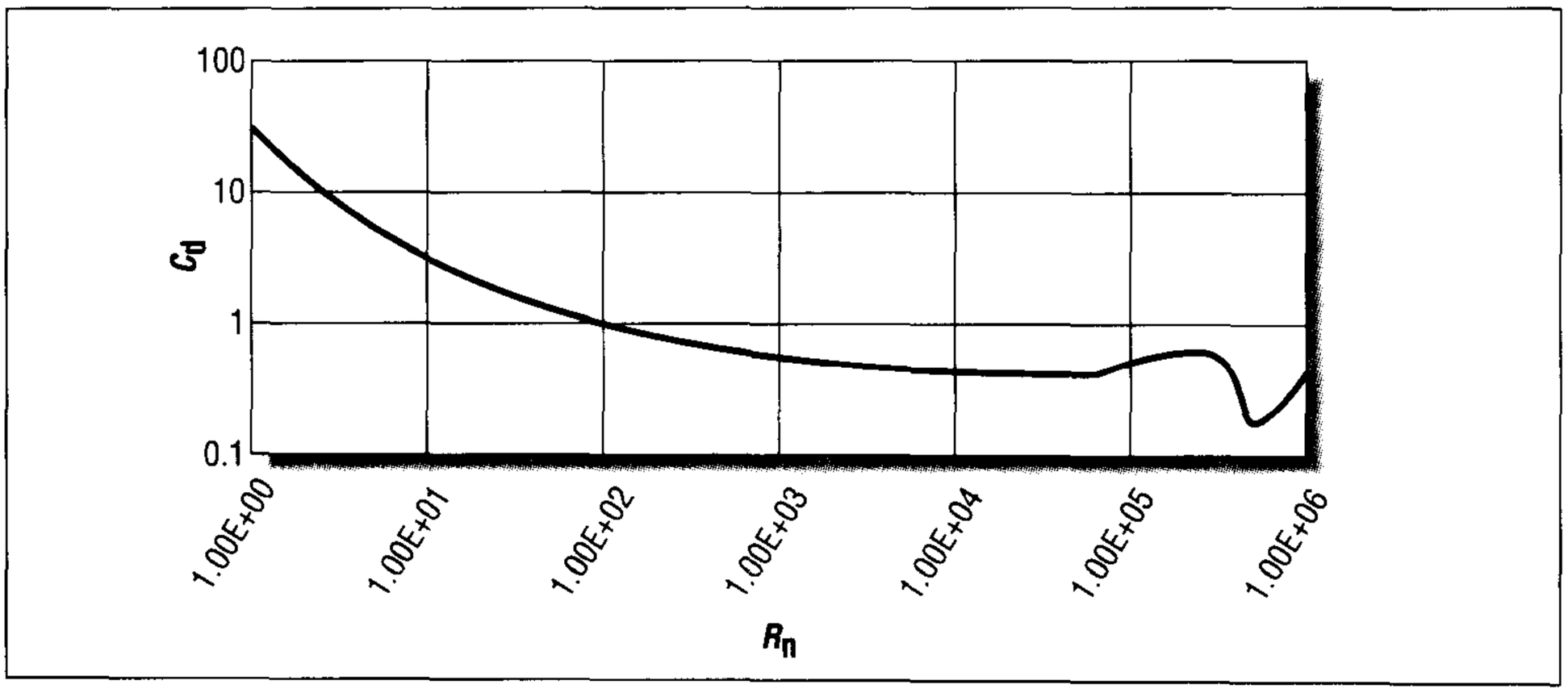



图 6-9：光滑球体的总阻力系数对雷诺数的曲线图（注 2）

其中 $v$ 是速率，$L$ 是物体特征长度（对球体而言是直径），$\upsilon$ 是流体的动粘度，$\rho$ 是流体的质量密度，而 $\mu$ 则是流体的绝对粘度。为了让雷诺数成为无因次的数，速度、长度和动粘度的英制单位必须分别是 ft/s，ft，$ft^2/s$。而国际单位则分别是 m/s，m，$m^2/s$。

雷诺数对于无因次化给定大小的物体（例如缩小模型）上测得的数据是很有用的，因此可以按比例缩放这些数据，来估计形状类似却大小不同的物体的数据。这里的“类似”是指物体的几何形状类似（只是大小不同），而物体周围的流动也会是类似的。球体的特征长度就是它的直径，因此可以用从已知直径的球体模型所获得的阻力资料，来估计另一个不同直径的较大球体上的阻力。这种比例缩放技巧更有用的应用，是根据由风洞或拖曳水槽实验中获得的模型测试资料，可以估计真正的船或飞机附属物上的黏滞阻力。

雷诺数可以作为流体流动性质的指针。低雷诺数通常代表层流，而高雷诺数通常代表紊流。两者之间存在着由层流转变成紊流的过渡范围。通过控制良好的实验，可以得知这个过渡的临界雷诺数。然而一般来说，物体周围的流场不管是低紊流还是高紊流，都会影响转变发生的时间。也就是说，临界雷诺数会根据所调查的问题类型而改变。例如，管子里的水流、船附近的水流、飞机附近的气流等。

总阻力系数 $C_d$，是由测试和使用以下公式所得到的总阻力 $R_t$ 所计算出来的：

$$C_d = R_t/(0.5\rho v^2 A)$$

---

注 2：　图中的曲线显示光滑球体的 $C_d$ 对 $R_n$ 的趋向。想要取得球体及其他形体更精确的阻力系数数据，请参考大学用的流体力学书籍，诸如 Daugherty，Franzini 与 Finemore 所合著的“Fluid Mechanics with Engineering Applications”。

其中的$A$是物体的特征面积，对一个球体来说，$A$基本上就是球体的正投影面积，等于一个直径与球体相同的圆形的面积。而船壳的面积$A$通常是船壳在水面下的表面积。如果找出公式右边的单位，你会发现阻力系数是无因次的，也就是它没有单位。

如果已知总阻力系数，就可以用以下公式估计总阻力：

$$R_t = (0.5\rho v^2 A)C_d$$

假如已经有了足够的信息，即总阻力系数、密度、速率和面积，这个公式就会比第三章中的公式好用多了。请注意总阻力与速率平方成正比。要得到以lb为单位的$R_t$值，速率的单位就必须是ft/s，面积的单位就应是$ft^2$，而密度的单位就应是$slug/ft^3$（请记得$C_d$是无因次的）。在国际单位制系统中，要想得到单位以N为单位的$R_t$值，速率的单位就必须是m/s，面积的单位就应是$m^2$，而密度的单位就应是$kg/m^3$。

回到图6-9，可以做一些观察。首先可以发现当雷诺数增加时总阻力系数会减小。这是因为当雷诺数增加时，分离点及尾流的形成会往球体后面移动，并使压差阻力减小。当雷诺数约在250 000时，阻力会大幅地减小。这是因为流动完全变成紊流，而使压力阻力相对地减小。

在第四章的Cannon2范例中，曾实现抛体上空气阻力的理想公式。该例子使用的阻力系数是个随便定义的常数。如前所述，使用本章所介绍的公式计算总阻力，以及使用图6-9中的总阻力系数数据来估计抛体的阻力是较好的。虽然更精确，但是也更复杂。具体地说，阻力系数现在是雷诺数的函数，而雷诺数是速度的函数。你必须先建立阻力系数对雷诺数的表格，并且改写在每个时间点计算出的雷诺数。另一个方法，可以将阻力系数画成曲线图，并推导出可以代用的公式。然而阻力系数数据可能要用片段法才能推导出适用于每个阻力系数区域的曲线。这里要用的球体数据就是一个例子。这个数据并不适用于全范围雷诺数的单一多项式曲线。在这种状况下，要对阻力系数使用可掌握的方程。这些方程只对有限范围的雷诺数有意义。

虽然Cannon2范例有其缺陷，但观察阻力对抛体轨迹的影响是很有用的。最明显的影响是抛体的轨迹不再是抛物线。从图6-10中你可以看到，当轨迹到达最高点后，会有更明显的下降。

阻力对轨迹的另一个重大影响（也适用于自由落体），就是阻力会限制可达到的最大垂直速度。这个极限称为极限速度。拿一个物体做自由落体，当物体因为重力加速度的作用而向地球加速时，它的速度会增加。因为阻力是速度的函数，所以当速度增加时，阻力也会增加。在某个速度时，使物体运动减缓的阻力会增加到与将物体拉向地球的重力相等。如果没有其他会影响物体运动的力，此时的净加速度为0，物体将以固定的极限速度继续下降。

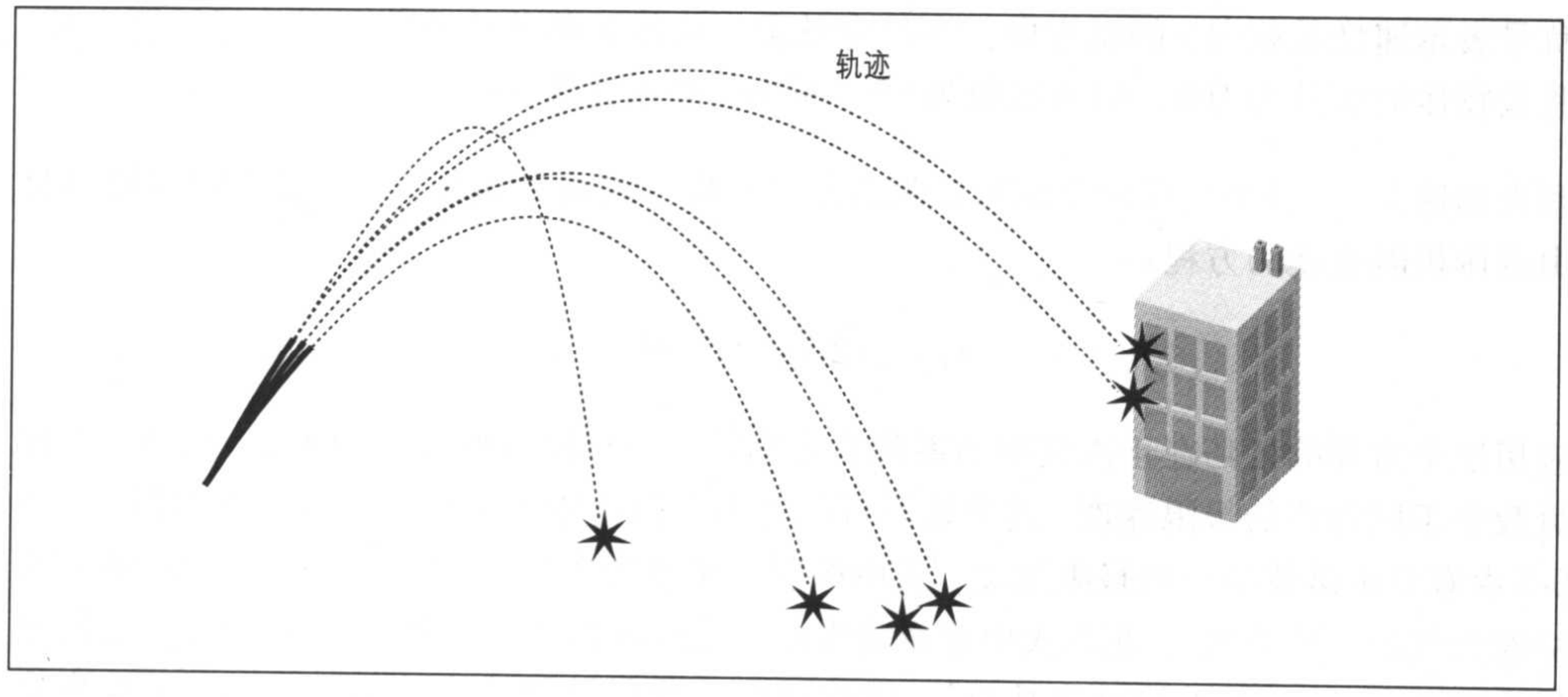



图 6-10：Cannon2 范例的轨迹

下面再进一步说明。回到 Cannon2 范例中所推导出的抛体速度 $y$ 分量（垂直分量）的公式。这里再重写一次，可不用再翻回第四章：

$$v_{y_2} = (1/C_d)e^{(-C_d/m)t}(C_d v_{y_1} + mg) - (mg)/C_d$$

从公式中看得并不明显，但是当时间增加时，速度的分量 $v_{y_2}$ 趋近于某个常数值。为了显示此现象，将这个公式绘制成如图 6-11 所示的样子。

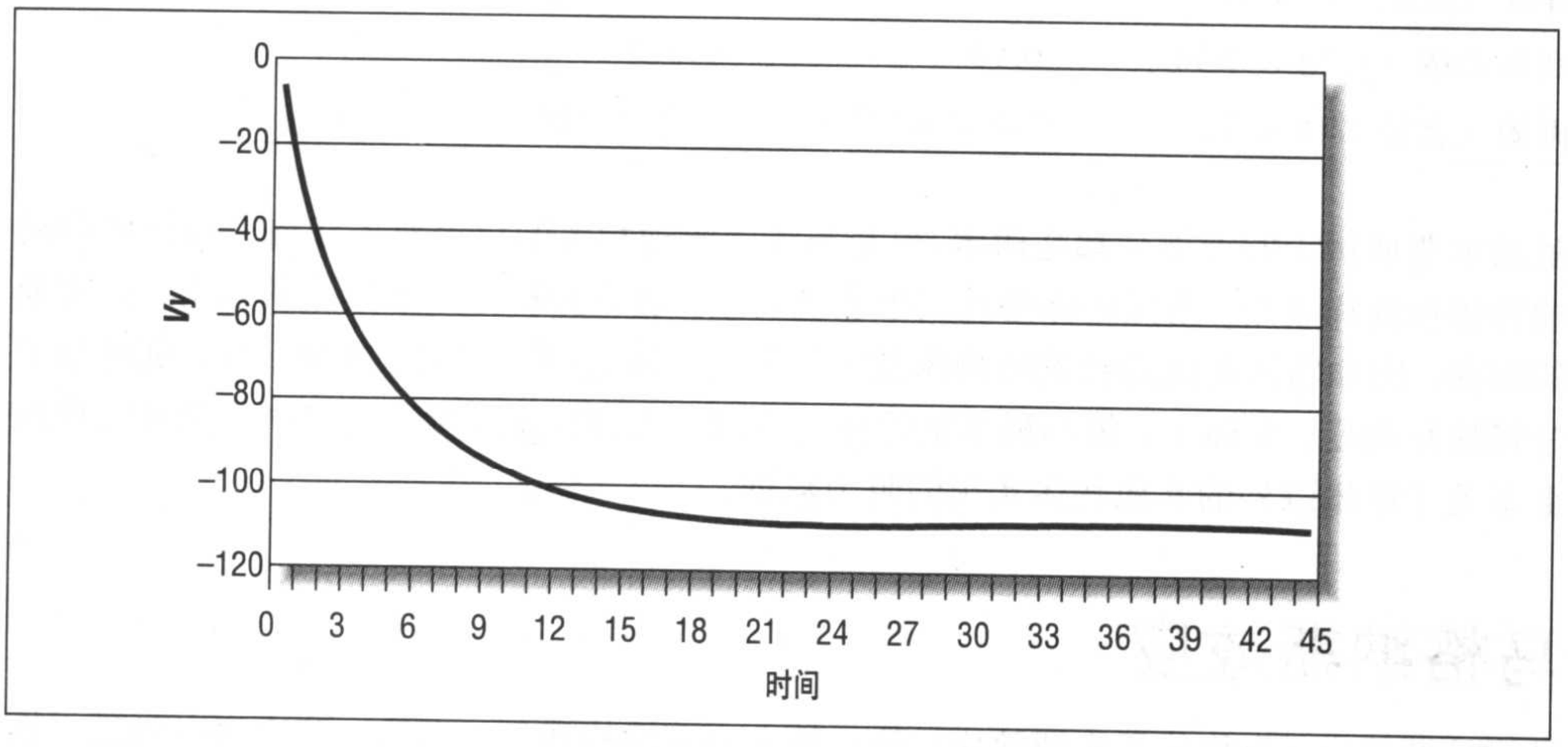



图 6-11：极限速度

正如你所看到的，随着时间的增加，速度达到最大值约 −107.25 速率单位。速度的值为

负号表示速度是朝向 $y$ 的反方向。本例中就是指物体正朝着地球落下（为了计算方便，假设物体的质量为 100，阻力系数为 30，而物体的初速为 0）。

假设初速为 0，并令之前所介绍的总阻力方程与物体的重量相等，可以导出以下计算自由落体极限速度的方程：

$$v_t = \sqrt{(2mg)/(C_d \rho A)}$$

应用这个方程的技巧在于决定阻力系数右方的值。可以试着假设阻力系数为 0.5，来计算数个不同物体的极限速度。这个练习可以让你看出物体大小对于极限速度的影响。表 6-5 中有自由落体的一些极限速度，其中的空气密度为 $2.37 \times 10^{-3}$ slug/ft$^3$（在 60°F 及标准大气压下的空气）。在公式中使用密度单位为 slug/ft$^3$，$m$ 的单位必须是 slug，g 的单位必须为 ft/s$^2$，而 $A$ 的单位是 ft$^2$，由此可以计算出极限速度的单位为 ft/s。我先将 ft/s 转换成 mph，来表示表 6-5 中的资料。表中所有物体的重量只是将它的质量 $m$ 乘以 $g$ 而已。

表 6-5：不同物体的极限速度

| 物体 | 重量（lb） | 面积（ft2） | 极限速度（mph） |
|---|---|---|---|
| 自由落下的跳伞员 | 180 | 9 | 125 |
| 张开降落伞的跳伞员 | 180 | 226 | 25 |
| 棒球（直径 2.88 in.） | 0.32 | 0.045 | 75 |
| 高尔夫球（直径 1.65 in.） | 0.10 | 0.015 | 72 |
| 雨滴（直径 0.16 in.） | $7.5 \times 10^{-5}$ | $1.39 \times 10^{-4}$ | 20 |

虽然本节所讨论的大部分都是球体，但是与流体流动相关的讨论对于任何在流体中移动的物体也同样适用。当然物体的几何形状越复杂，越难分析作用于其上的阻力。而其他的因素，比如物体表面的状况和物体是否位于两流体之间（如海上的船）等，更增加了分析的复杂度。实际上，缩小模型试验相当有用。在书后面的参考文献中，你可以找到更多关于球体以外的其他物体实用的阻力数据。

## 马格纳斯效应

马格纳斯效应（也称为罗宾斯效应）是一种十分有趣的现象。由前一节中可以知道，物体在流体中移动时会受到阻力的影响。物体在流体中移动并旋转时，会发生什么事呢？举例来说，假设之前所讨论的球在流体（例如水或空气）中移动时，会绕着通过其质心的轴旋转。当球旋转时会有什么影响是值得注意的部分：事实上会产生升力！没错，就

是升力！在多数人的经验中，通常会认为升力与类似飞机机翼或是水翼的形状有关。很少有人知道圆柱体或球体旋转时也能产生升力。本节将使用移动的球体来解释这种现象的成因。

通过前面〈阻力〉一节，我们已经知道球体在快速移动时，会产生一个分离点并在球体后方产生紊流。前面提到作用在紊流尾流中的压力低于作用在球体前端的压力，而此压力差会产生阻力分力。当球旋转时（假设球是绕着通过中心的水平轴顺时针旋转的，如图 6-12 所示），通过球体上方的流速会加快，而通过球体下方的流速则会减慢。

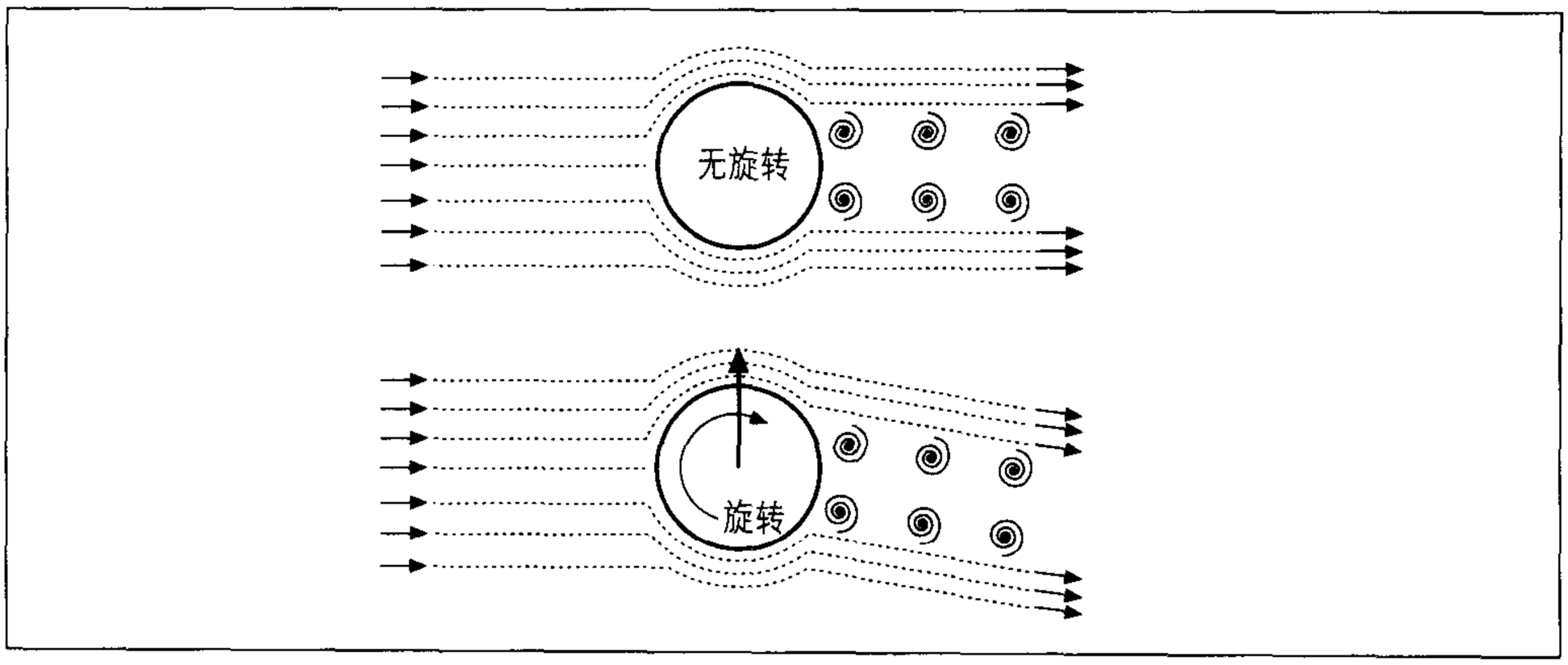



图 6-12：旋转的球体

由于摩擦力的作用，球体表面上有一层薄薄的流体边界层。在球体的表面上，边界层里的流体对球体的相对速度为 0。在边界层中，离球体表面的距离越远则流速越快。在球体旋转的状况下，因为球体上方的流速增大而下方的流速减小，所以球体上下方的流体就产生压力差。此外球体上方的分离点会被推向更后方。结果在球体四周产生不对称的流动模式和垂直于流动方向的净升力（因压力差产生的）。如果球体的表面有点粗糙，不只会使摩擦力增加，也会使升力增加。

不要因为升力这个名词而认为这个力总是会使球体上升。升力对球体运动轨迹的影响与球体旋转的轴线和角速度的方向息息相关。

马格纳斯升力的量值与物体运动的速度、旋转率、流体密度、物体的大小，以及流体流动的性质成正比。这个力并不容易分析计算，而且由于流体动力学中的许多问题，你必须依赖实验数据来正确地估计特定状况下物体的马格纳斯升力。然而有一些分析上的技巧可以用来估计马格纳斯升力。不必理会理论上的细节，只需使用Kutta-Joukouski理论来估计旋转物体（像是圆柱体或球体）的升力。Kutta-Joukouski理论是基于无摩擦的理想流体与物体的环流概念（例如物体周围的涡流）而来的。你可以在任何的流体力学的

教材中找到这个理论（在书后的参考文献中也提供了），所以这里就不再赘述，只是会介绍一些结论。

对于在流体中移动并且旋转的圆柱体，可用以下的公式估计马格纳斯升力：

$$F_L = 2\pi\rho Lvr^2\omega$$

其中的$v$是移动的速度，$L$是圆柱体的长度，$r$是它的半径，而$\omega$是它的角速度，单位是rad/s。如果已知旋转率$n$的单位是rps（每秒几转），则$\omega = 2\pi n$。如果已知旋转率$n$的单位是rpm（每分钟几转），则$\omega = (2\pi n)/60$。

对于在流体中移动并旋转的球体，可用以下公式：

$$F_L = \pi\rho vr^3\omega$$

其中$r$是球体的半径。通过这个公式中相应的单位会算出升力的单位为lb（英制）或N（国际单位制）。在英制系统中，密度、速度、长度和半径的单位分别是slug/ft$^3$，ft/s和ft。在公制系统中，对于这些量合适的单位分别是kg/m$^3$，m/s和m。

这些公式只适用于估计马格纳斯升力；所求得的值接近实际数字但不够精确，而误差在不同的情况下可能高达50%。这些公式假设在流体与旋转的物体表面之间没有滑动，也就是没有摩擦力，也不考虑表面的粗糙程度，而且也没有边界层。

无论如何，这些公式可以在游戏中模拟飞行物体的马格纳斯升力，因此可以模拟不同大小的物体，在不同的速度及不同的旋转率中的相对差异。可以得到看起来差不多的结果。如果要求数值的正确性，就必须改采用对于不同问题的实验资料。

类似于前一节的阻力资料，实验得出的升力资料一般也是以升力系数表示的。使用类似于阻力公式的公式，可用以下公式计算升力：

$$F_L = (0.5\rho v^2 A)C_L$$

如往常一样，这个公式并不像看起来那样简单。技巧在于决定升力系数$C_L$，它是物体表面现况、雷诺数、速度和旋转率的函数。并且实验数据显示阻力系数也会受旋转影响。

举例来说，假设高尔夫球被完美地击出去，也就是飞行时这个球会绕着垂直于运动方向的水平线旋转。在这种状况下，马格纳斯升力会使球在空气中飞得更高，因而增加飞行时间和距离。若高尔夫球以190 ft/s的初速度和10°的发射角被挥击出去，正常的状况下由马格纳斯升力所增加的距离约为65码，很明显升力的影响非常惊人。事实上，在高尔夫球运动悠久的历史中，人们试着要增加此力的影响。在18世纪末（当高尔夫球的表面仍然光滑的时候），人们观察到粗糙表面的球比光滑表面的球飞得更远。这个观察结

果促使人们开始制造粗糙表面的球，以增加马格纳斯升力的影响。现今高尔夫球上的凹洞便是数十年的实验、研究结果，并且公认为最佳的效果。

一般来说，高尔夫球被球杆击出去时的初速度为250 ft/s，并以60 rps的速度后旋(backspin)。在这种初始条件下，相对的马格纳斯升力系数的范围约在0.1到0.35之间。根据不同的旋转率，升力系数可以高到0.45，而球所受的升力可以达到球重的50%。

如果高尔夫球并不是被完美地击出去，马格纳斯升力反而会妨碍飞行。举例来说，当球离开杆头时，并不是绕着水平轴旋转时，球的轨迹就会偏左或偏右弯曲。如果挥击到球的上面，会使球的顶部以远离你的方向旋转，而且球的轨迹会更快地向下弯，明显地减少球飞行的距离。

再举一个例子，假设将棒球投出，并让它绕着与运动方向垂直的水平轴上旋（topspin)。这时马格纳斯升力会使球的轨迹往下弯曲，使球比没有旋转时更快地落下。如果投手投球时让球绕着非水平的轴旋转，就会产生偏左或偏右的曲球。投手会使用的另一个技巧是让球下旋（后旋），让球看起来会往上飘（从打击者的角度)。这种上飘快速球并不是真正地往上飘，而是因为马格纳斯升力的影响，让它会比没有旋转时更慢地落下。

对于一般的投球速度148 ft/s和旋转率30 rps，升力可以达到球重的33%。而对于典型的曲球而言，升力系数约在0.1到0.2之间，对于高飞球则可以高达0.4。

这里虽然只举出两个范例，但是你不需要花太多时间去找其他有关马格纳斯升力作用的例子。看看板球、足球、网球和乒乓球在飞行中旋转时的表现。一把有膛线的枪所发射出来的子弹在飞行中会旋转，因此也受马格纳斯升力的影响。甚至有些装设着高垂直旋转圆筒风帆的帆船，也是使用马格纳斯升力来提供推进力的。我曾经看过某些技术性文章，介绍使用旋转圆筒状的螺旋桨来代替翼型螺旋桨的推进器。

为了更多地介绍马格纳斯升力的影响，以下的范例程序模拟球在不同的上旋（或下旋）转速所产升的轨迹。这个范例程序是由cannon（大炮）程序改写过来的，所以你应该会熟悉大部分的程序码。这个范例将会忽略阻力的影响，所以会影响球的只有重力和马格纳斯效应。这么做是为了突出旋转升力所产生的影响，并且让运动方程能更简洁。

因为这个范例中大部分的程序码和之前的cannon范例相同（或相似），所以就不再重复了。以下会介绍在此范例中所用的全局变量，以及负责计算运动方程的DoSimulation函数修订版。

```
//-------------------------------------------------------------------------//
// 模拟程序中所需的全局变量
//-------------------------------------------------------------------------//
TVector          V1;      // 初速度（给定值），m/s
```

```
TVector         V2;     // 时间 t 时的速度向量，m/s
double          m;      // 抛体质量（给定值），kg
TVector         s1;     // 初始位置（给定值），m
TVector         s2;     // 抛体的位置（位移）向量，m
double          time;   // 抛体刚发射时的时间，s
double          tInc;   // 循环中使用的时间增量，s
double          g;      // 重力加速度（给定值），m/s2
double          spin;   // 旋转率（给定值），rpm
double          omega;  // 旋转率（角速度），rad/s
double          radius; // 抛体的半径（给定值），m
#define         PI      3.14159f
#define         RHO      1.225f      // kg/m3
//-------------------------------------------------------------------------//
int          DoSimulation(void)
//-------------------------------------------------------------------------//
{
    double          C = PI * RHO * RHO * radius * radius * radius * omega;
    double          t;

    // 跳至下个时间点
    time+=tInc;
    t = time;

    // 计算V2：
    V2.i = 1.0f/(1.0f-(t/m)*(t/m)*C*C) * (V1.i + C * V1.j * (t/m) -
          C * g * (t*t)/m);
    V2.j = V1.j + (t/m)*C*V2.i - g*t;

    // 计算S2：
    s2.i = s1.i + V1.i * t + (1.0f/2.0f) * (C/m * V2.j) * (t*t);
    s2.j = s1.j + V1.j * t + (1.0f/2.0f) * ( ((C*V2.i) - m*g)/m) * (t*t);

    // 检查与地面（xz平面）的碰撞
    if(s2.j <- 0 )
        return 2;

    // 如果模拟时间过长，就结束模拟程序
    // 让模拟程序不会卡在循环中
    if(time>60)
        return 3;

    return 0;
}
```

这个模拟程序的重点在于，分别计算即时速度 v2 和抛体位置 s2 的那几行程序码。这里使用的运动方程来自 2D 运动学的运动方程，包括第四章所讨论的重力，结合以下的公式，来估计旋转球体的马格纳斯升力：

$$F_L = (2\pi^2 \rho v r^4 \omega)/(2r)$$

你可以通过给予这个程序不同的旋转率（rpm），来观察旋转对于抛体轨道的影响。程序会将旋转率的单位换算成rad/s，并将这个值存在变量omega中。正的旋转率值表示下旋，这时球的底部会朝着远离你的方向旋转；负的旋转率值表示上旋，这时球的顶部会朝着远离你的方向旋转。上旋会产生正的升力，会增加抛体飞行的距离；下旋会产生负的升力，这个力会让球往地面落下，减少球飞行的距离（这个范例假设旋转轴是水平的，并且垂直于屏幕所在平面）。图6-13显示球的轨道。

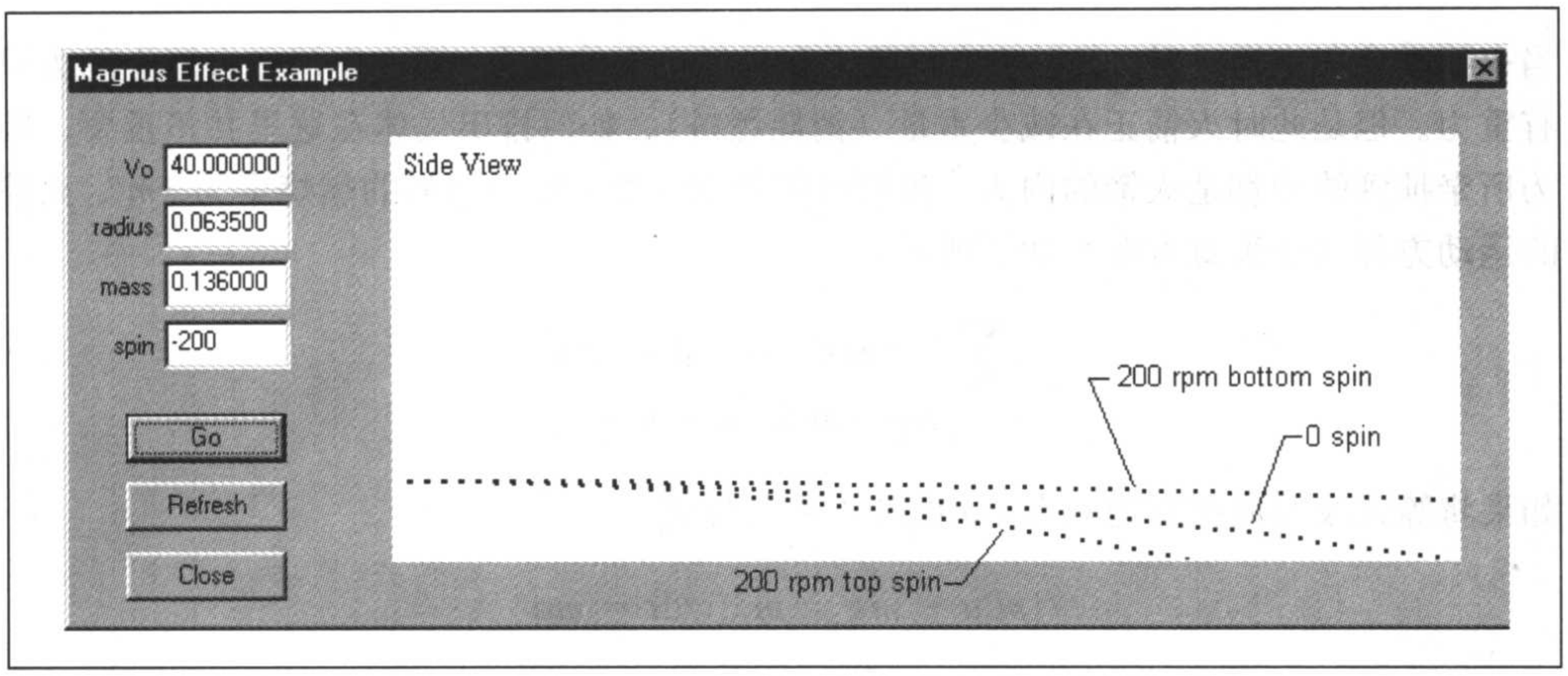



图6-13：马格纳斯效应的范例程序

## 变动质量

在本书的前面几章，曾经提到过关于变动质量的问题。这里会再次看到变动质量是因为它与自力推进的抛体（如火箭）有关。当火箭使用推力加速时，会以某种速率排出气体（燃烧燃料所产生的）。当所有的燃料用尽时，火箭不再产生推力而到达最高速度。在燃料用尽后，可以将火箭的轨迹视为之前所讨论的非自力推进的抛体。然而因为火箭会产生推力，所以需要考虑质量的改变，因为这会影响它的运动。

当某个物体的质量改变是以绝对速度0失去或获得质量时（例如船耗油的情况），你可以如往常般设定运动方程，其中所有力的总和等于动量改变率。然而在此情况下，质量是时间的函数，运动方程如下：

$$\sum F = ma = \mathrm{d}/\mathrm{d}t(mv) = m(\mathrm{d}v/\mathrm{d}t) + (\mathrm{d}m/\mathrm{d}t)/v$$

你可以如往常般去解这个方程，但是要注意质量与时间的关系。

另一方面，火箭是以非零的速度排出质量（气体），而不能用以上的方法正确地处理质量改变量。在这种状况下，需要考虑排出的质量与火箭本身的相对速度。而线性运动方程现在看起来像这样：

$$\sum F = m\,\mathrm{d}v/\mathrm{d}t + \mathrm{d}m/\mathrm{d}t\,u$$

其中 $u$ 是排出的质量（气体）与物体（这里是火箭）之间的相对速度。

当火箭垂直向上运动时，若忽略空气阻力和排气口上的压力，惟一作用在火箭上的力只有重力。但是此时火箭正在减少重量（消耗燃料）。如何排出气体在这里并不重要，因为所牵扯到的力都是火箭的内力，我们只需考虑外力。假设燃料的燃烧率为 $-m'$，火箭的运动方程（于垂直方向）如下所示：

$$\sum F = m\,\mathrm{d}v/\mathrm{d}t + \mathrm{d}m/\mathrm{d}t\,u$$
$$-mg = m\,\mathrm{d}v/\mathrm{d}t - m'\,u$$

如果将等式改写成在右边只有 $ma$ 项时，可以得到：

$$m'u - mg = m\,\mathrm{d}v/\mathrm{d}t = ma$$

这里可以看到，火箭在空气中的推力为 $m'u$。因为燃料的燃烧率为常数，因此在任何时间点上火箭的质量等于：

$$m = m_0 - m't$$

其中 $m_0$ 是初始质量，而燃烧率 $m'$ 的形式是质量 / 单位时间。

# 第七章

# 飞机

在制作关于飞行模拟的游戏时，在游戏引擎中最重要的部分之一就是飞行模型。当然其他部分如立体影像、用户界面、故事情节、航电系统模拟和程序代码等也是很重要的，但是真正用来定义所模拟飞机的行为属于飞行模型这部分。这基本上是飞机飞行物理模型的简化版，其中包括假设、估计值和所有用来计算质量、惯性、升力及阻力与力矩的方程。

飞机在飞行时会受到4个主要的力：重力、推力、升力和阻力。理所当然地，重力就是将飞机拉向地面的力。而升力是由飞机的主翼（或升力面）所产生的力，以抵消重力使飞机能悬浮在空中。推力是由飞机的推进器（喷射引擎或螺旋桨）所产生的力，能使飞机的速度增加并且使升力面能产生升力。最后，阻力是会抵消推力的力，使飞机的运动减慢。图7-1描绘出了这些力。

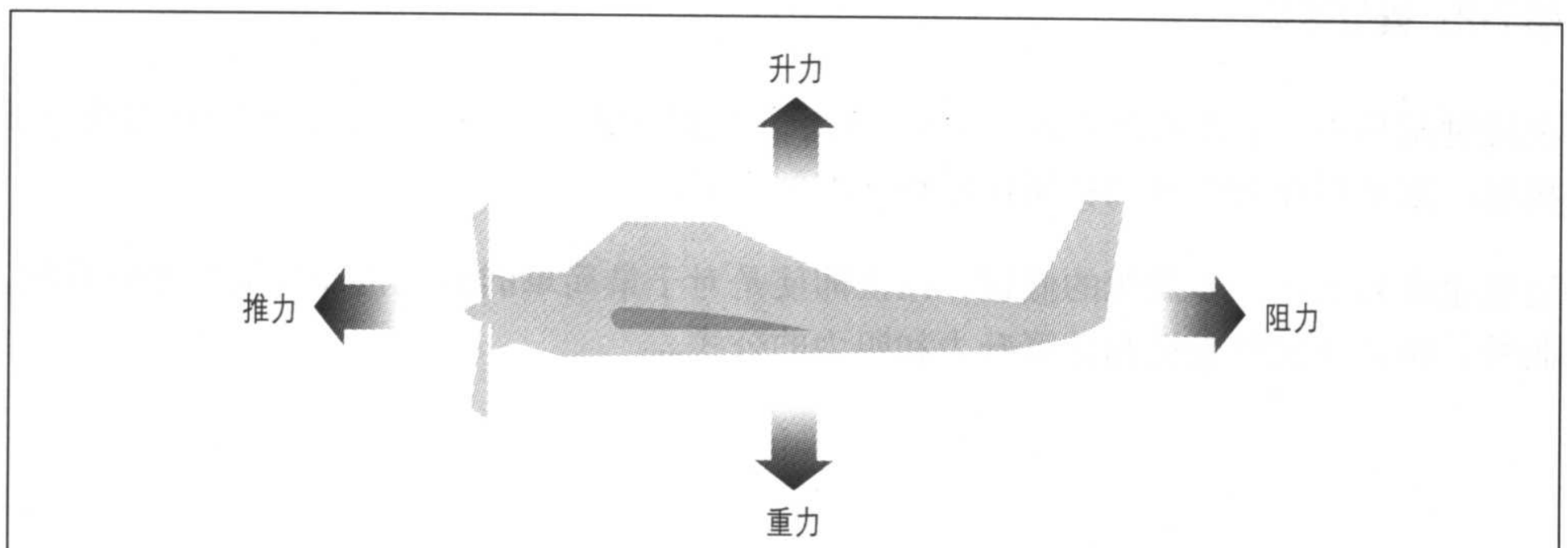



图7-1：飞机飞行时所受的力

因为在前几章已经介绍过由重力所产生的力，所以在本章中就不再赘述了，不过在讨论飞机上合力的影响时还会再提到。如果要让飞机保持飞行状态，那么总升力必须要大于或等于重力所产生的力。

在介绍其他三种作用于飞机上的力时，将提到一个精简且普通的飞机模型，并将它作为范例来说明。因为有太多种飞机的类型和设定，所以要在短短的一章中介绍所有的类型是不可能的。况且关于空气动力学主题的范围实在是太广泛且太复杂。因此将使用一种典型的次音速结构的模型来介绍本章的主题，如图 7-2 所示。

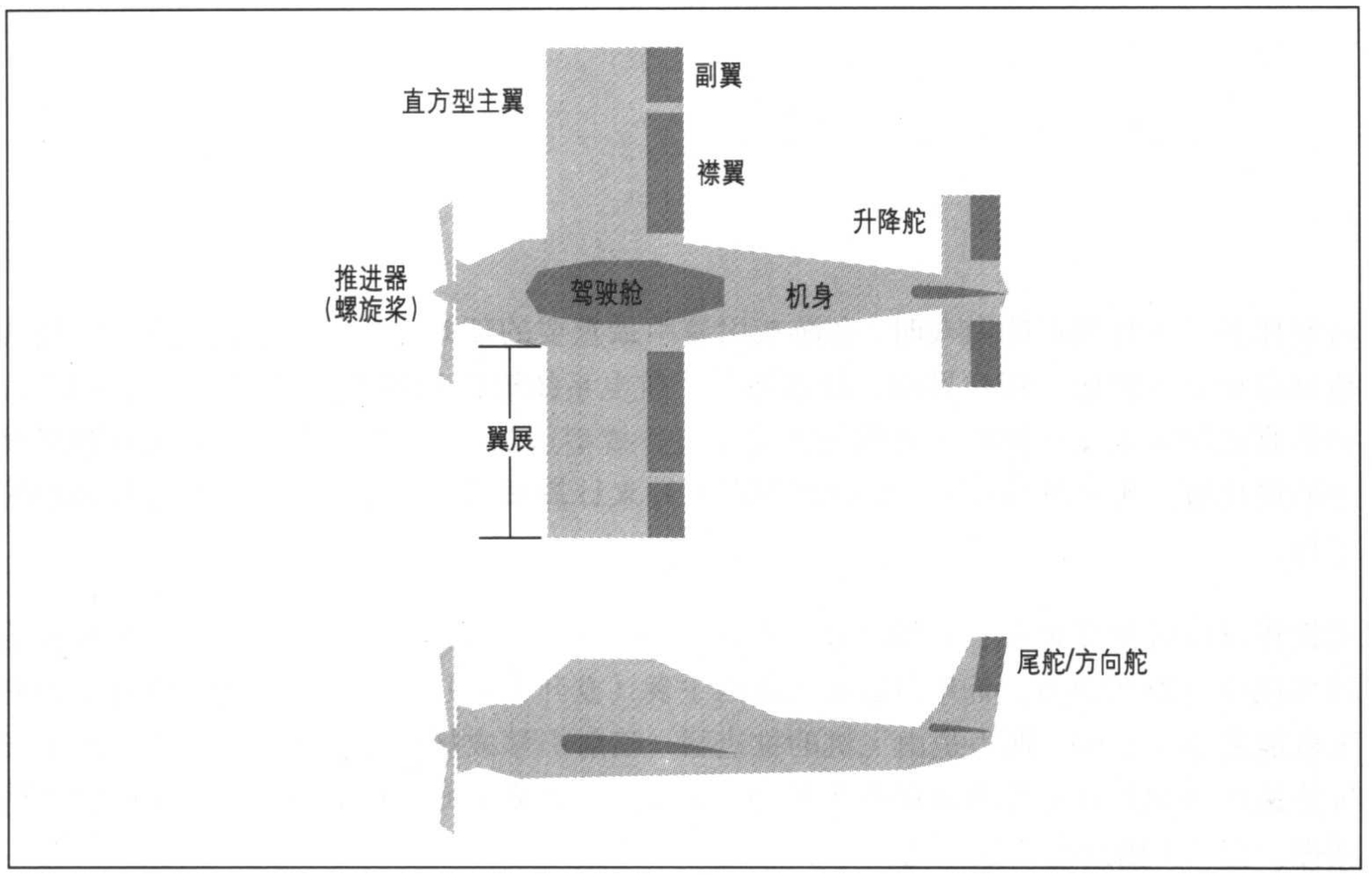



图 7-2：模型结构

在这种结构中，主要的升力面（主翼）位于飞机的前方，另一个相对较小的升力面位于机尾。这是现在大部分飞机所使用的基本配置方式。

这里也必须做进一步简化的假设，以使即使是对于最简单的模型也能容易地进行管理。此外，将依据实验数据和计算升力和阻力的公式。

# 几何形状

在开始谈升力、阻力和推力之前，我想先介绍一些基本的几何学和术语作为讨论的基础。熟悉了这些术语，在稍后提供的参考文献中，将能很快地找到所需的资料。

首先看看图 7-2 中所介绍的模型飞机的配置图。飞机的主要部分（通常用来装载乘客或货物)被称为机身（fuselage)。主翼（wing）是大块矩形的升力面，装设在机身两侧靠近前方的部位。主翼上较长的边称为翼展（span)，较短的边称为翼弦长（chord length)，或简称翼弦。翼展的平方与主翼面积的比称为展弦比，对于矩形主翼的飞机来说，可以将展弦比简化为翼展对翼弦的比。

在我们的模型中，副翼（aileron）装设在主翼后方的外侧。而襟翼（flap）则装设在主翼后方的内侧（在副翼的旁边)。装设在尾翼上类似飞行翼的小平面称为升降舵(elevator)。而装设在尾翼上垂直的襟翼称为方向舵（rudder)。稍后会介绍这些控制翼的作用。

仔细看看图 7-3 中机翼的剖面图可以更多地了解概念。

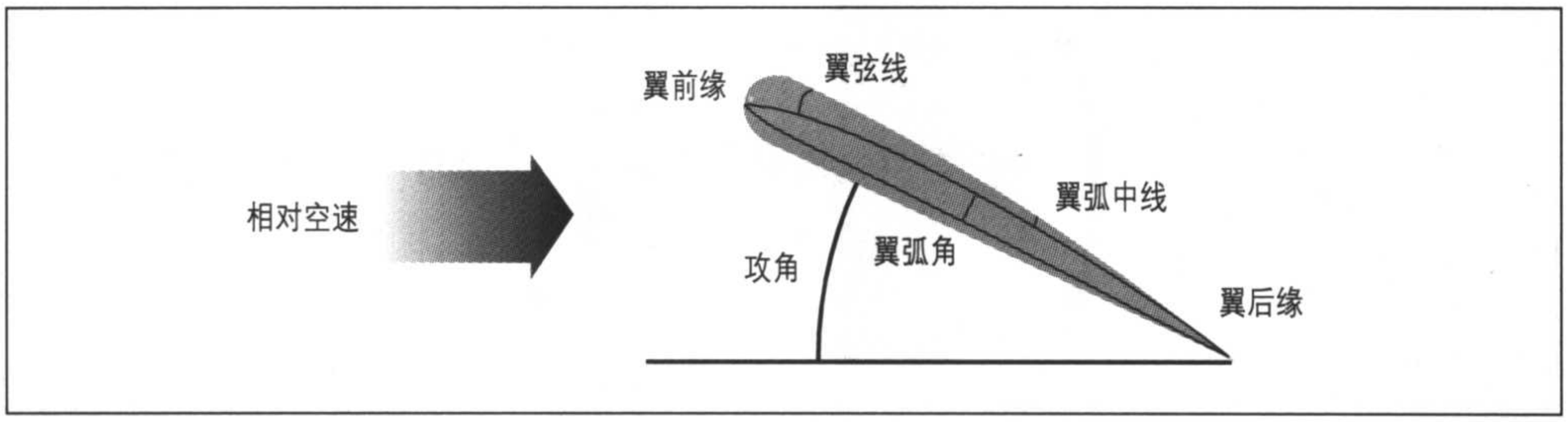



图 7-3：机翼的剖面图

图 7-3 中所示的是一个典型的弧形机翼剖面图（称为翼切形)。翼弧度（camber）指的是机翼剖面的曲率。如果从机翼的前缘至后缘画一条直线，那么所得到的线称为翼弦线(chord line)。如果将机翼从前缘至后缘分割成好几个剖面（像是面包的切片一样)，那么由这些切片厚度的中点所连接成的线称为翼弧中线（mean camber line)。翼弦线与翼弧中线最大的差值即为翼面弧度的大小。机翼行进方向（机翼通过空气时的相对速度向量）与翼弦线间的夹角称为绝对攻角（attack angle)。

当飞机在飞行时，可能会绕着某个轴转动。通常习惯将飞机的旋转视为绕着相对于飞行员的三个轴线运动。因此，这三个轴线是固定在飞机上的，也可以说不必理会飞机在3D空间中真正的方位。这三个轴线分别为俯仰轴、滚转轴和偏转轴。

俯仰轴横向地穿过飞机，即飞机左/右舷方向（注 1）。以飞行员的角度来看，当机鼻上抬或下降时就形成所谓的俯仰轴旋转。而当飞机主翼的翼尖相对于飞行员而上抬或下降时，就形成所谓的滚转运动。偏转轴是一个垂直于机身的轴线，让飞机能做以飞行员角度看来的由左向右（或由右向左）的旋转。这些旋转的方式在图 7-4 说明。

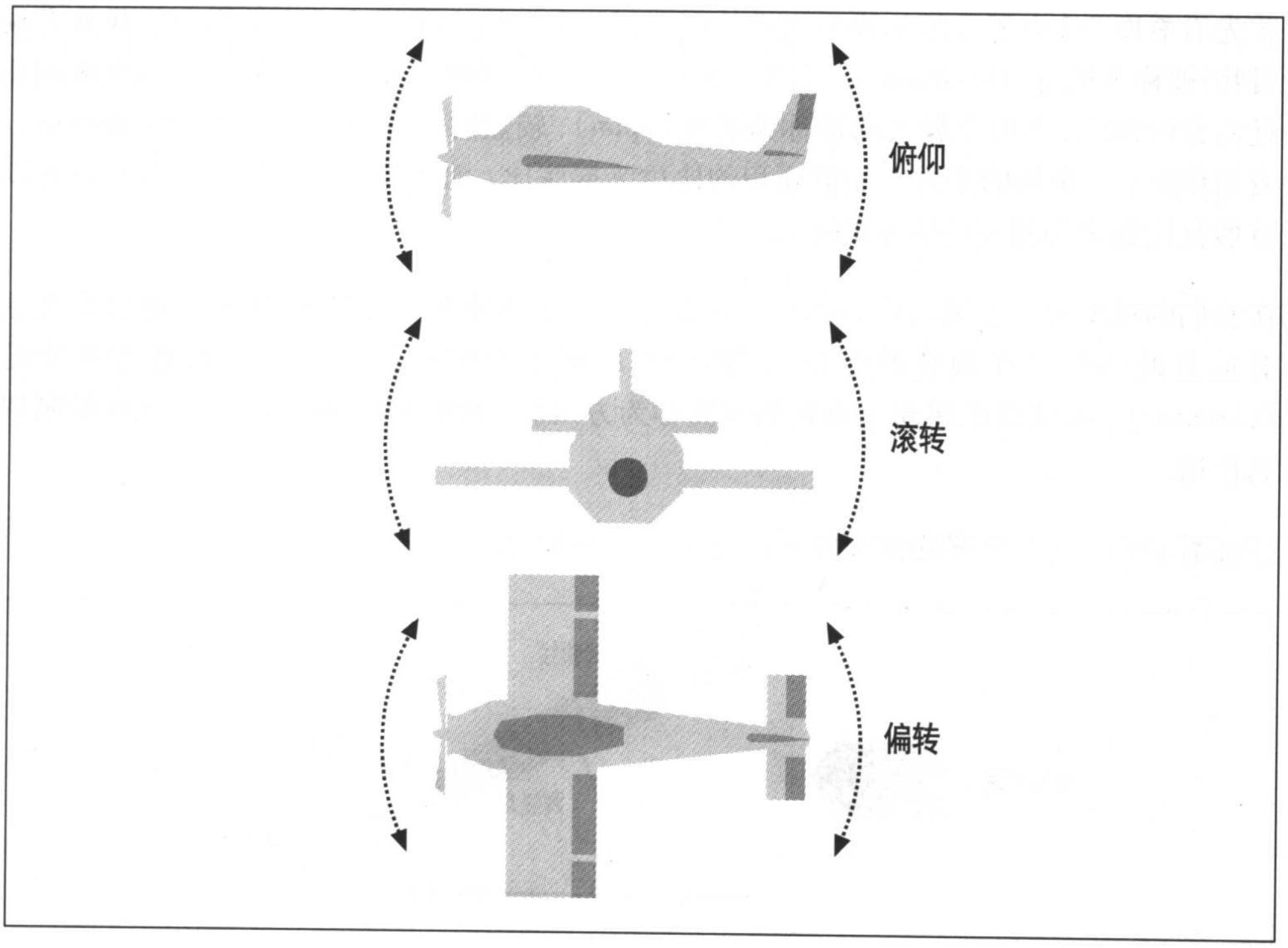



图 7-4：飞机的旋转

# 升力与阻力

当机翼在流体（例如空气）中移动时会产生升力。这种升力的机制与前一章所讨论的马格纳斯升力类似，都是贝努利定律的效应。然而不同的是，这种升力并不是由旋转产生的，而是由机翼的形状与攻角使流经机翼的空气产生升力。

注 1：以飞行员面向飞机前方（机头）为准，飞行员的左方称为左舷（port），右方称为右舷（starboard）。

图 7-5 显示机翼在空速为 $V$ 的空气中的移动情形的剖面图。$V$ 是翼面与翼面前未被扰动的气流之间的相对速度。当空气撞击到翼面时，会被翼面前缘附近的停滞点分成两股气流，一股气流由翼面上方流过，另一股则由下方流过。翼面下方的气流偏折下降，而当翼面上方的气流流经机翼前缘和翼面上方时则被加速。当这两股气流在机翼后缘合流时，就产生了所谓的 Kutta 条件。在理想状况下，边界层仍然会保持“附着”在翼面上，而不会像前一章中讨论的球体一样会被分开。

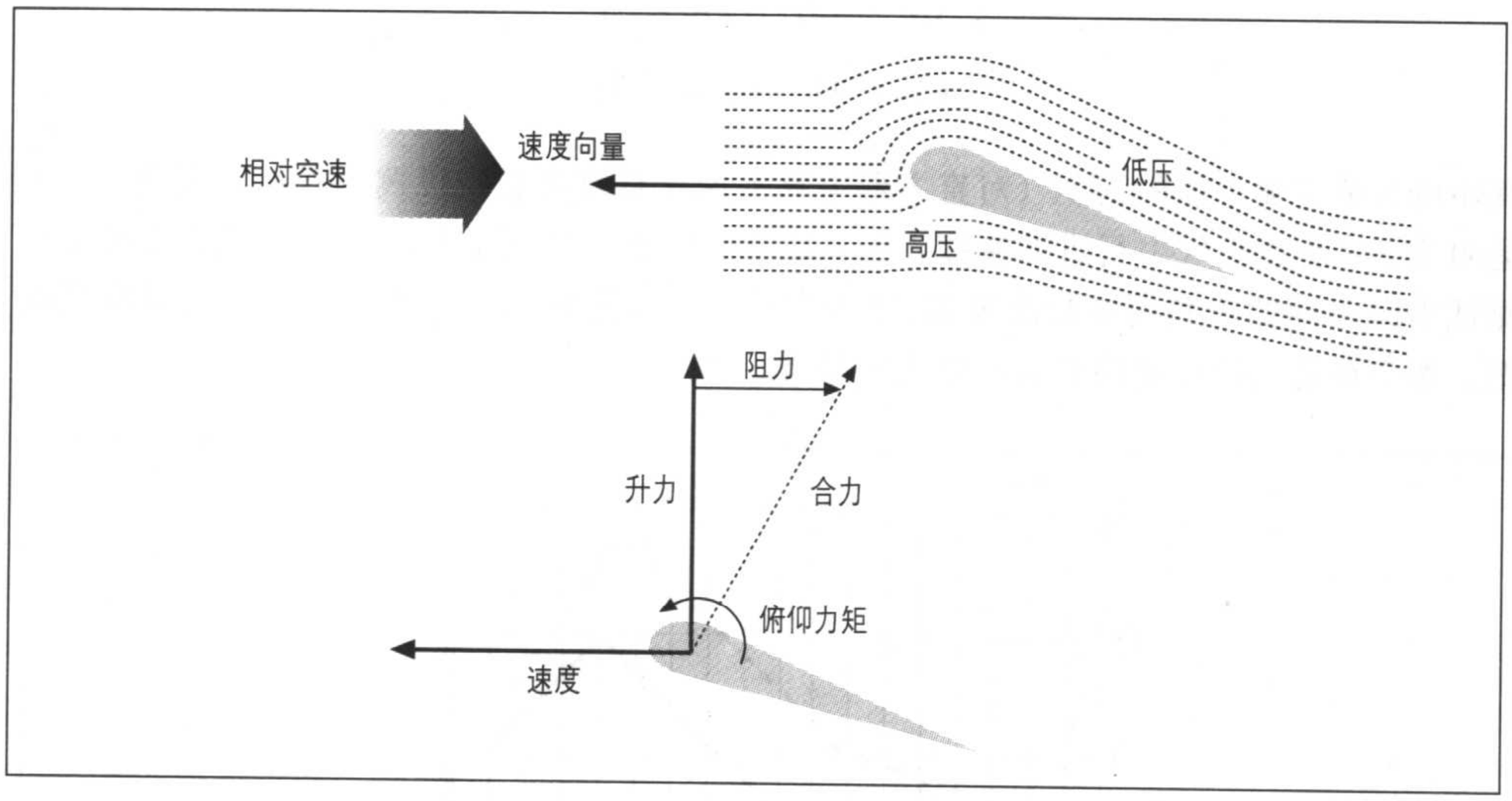



图 7-5：机翼在空气中移动的剖面图

相对速度较快的上层气流会在翼面上方产生低压区（请记住贝努利方程中提到，流体流动中的压力与速度成反比）。撞击到机翼并且流到翼面下方的气流会产生高压区。这些流动模式的组合效应，会使翼面的上下方分别产生低压区与高压区。翼面上下压力的差就会产生升力。定义上，升力与飞行方向（速度向量）垂直。

但是翼面不一定是弧形的才能产生升力，平板的机翼如果与气流方向形成一个攻角也会产生升力。同样，弧形的翼面也不需要有攻角就会产生升力。弧形的机翼在零度甚至是负的攻角下也能产生升力。因此，机翼所产生的总升力取决于两个条件：由弧形机翼产生的和由攻角产生的。

理论上，机翼的厚度与升力的产生无关。因此可以采用薄的弧形机翼，就像是滑翔机上用的布质机翼一样。但是在实现上，由于某些结构上的因素，机翼必须要有足够的厚度，比如机翼前缘的厚度可延迟失速。

机翼上下的压力差也会产生作用在速度向量反方向的阻力。升力与阻力是互相垂直的，并落在速度向量和垂直于翼弦线的向量所定义的平面上。这两种分力——升力与阻力，在合并后会产生合力作用在飞行中的机翼上。图 7-5 描绘出了这些现象。

升力与阻力都是空气密度、速度、粘性、表面积、展弦比和攻角的函数。传统上某种翼型设计的升力与阻力特性可由以下的无因次系数表示：

$$C_L = L/[(1/2)\rho V^2 S]$$

$$C_D = D/[(1/2)\rho V^2 S]$$

其中的 $S$ 是主翼平面的面积（对直方形主翼来说就是翼展长乘以翼弦长）。$L$ 是升力，$D$ 是阻力，$V$ 是在空气中行进的速度，而 $\rho$ 是空气密度。这些系数是用机翼的模型在风洞测试中以不同的攻角所实验出来的。这些实验的结果通常画成升力与阻力系数对攻角的图。图 7-6 是某种机翼的升力与阻力系数图。

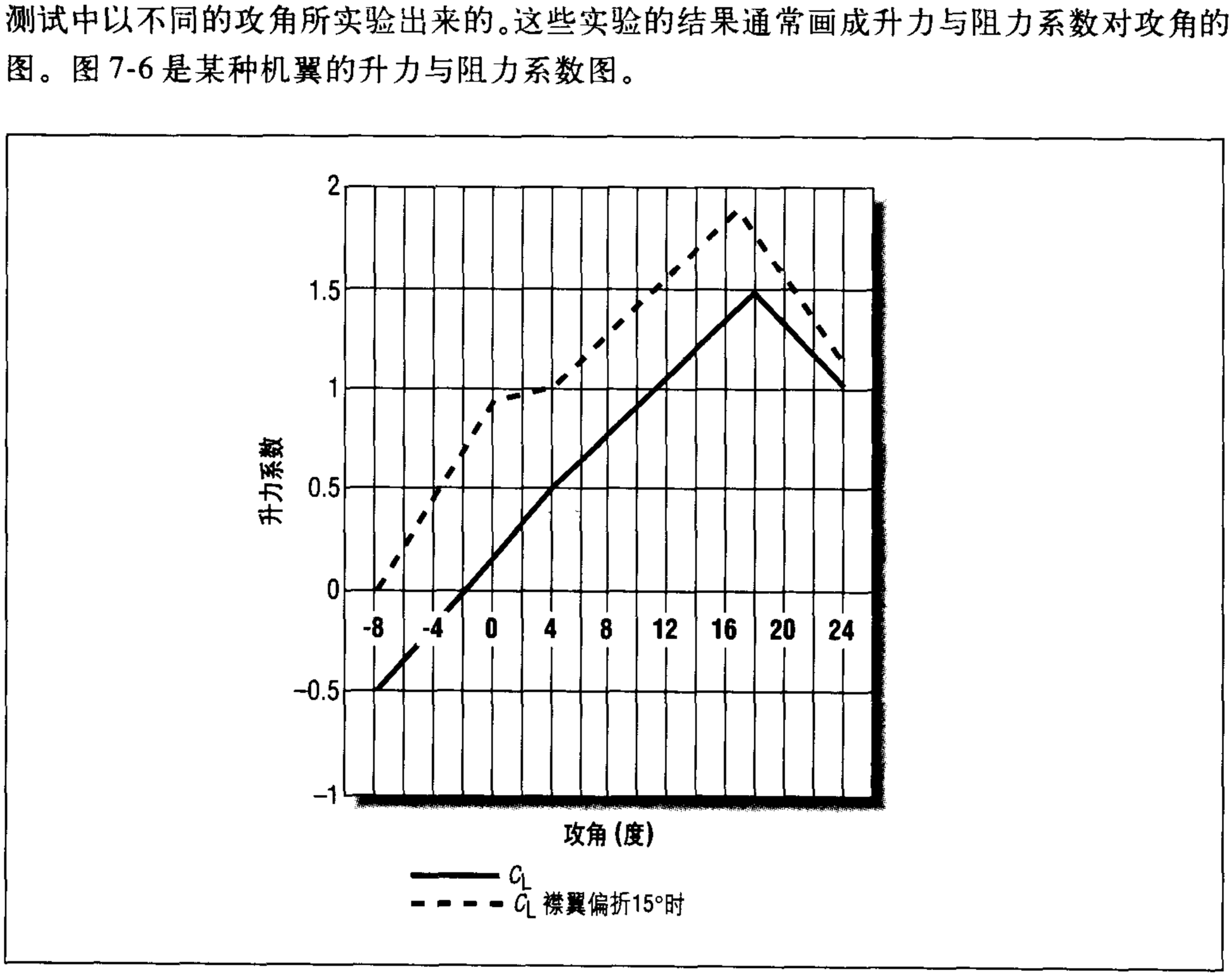



图 7-6：典型的 $C_L$、$C_D$ 和 $C_M$ 对攻角图

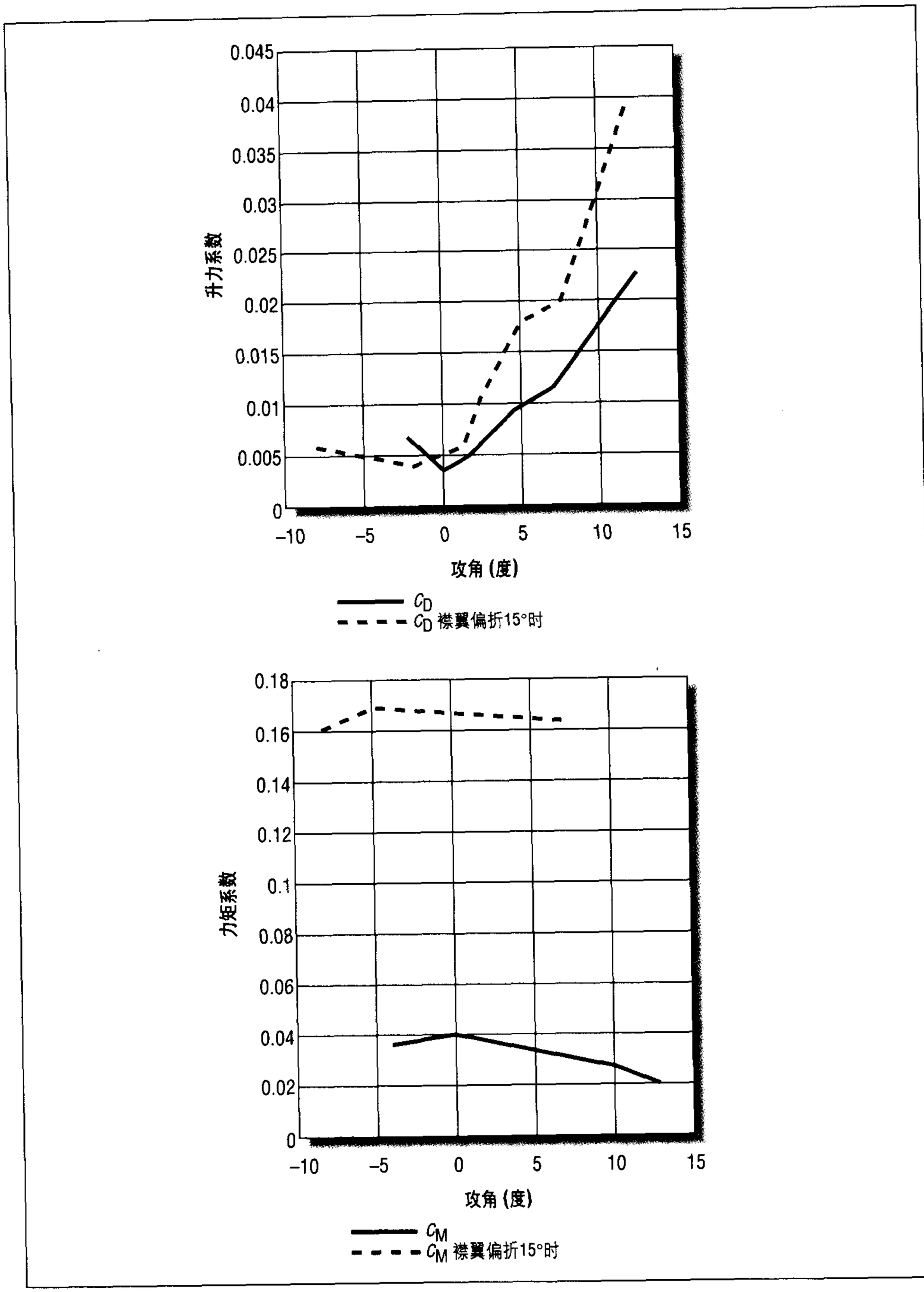



图 7-6：典型的 $C_L$、$C_D$ 和 $C_M$ 对攻角图（续）

最广为人知的机翼截面设计与测试资料是NACA机翼截面系列。由Ira H. Abbott与Albert E. Von Doenhoff合著的“Theory of Wing Sections”一书，包含了许多机翼设计的升力与阻力数据（可由参考文献中得到完整的书目）（注2）。

实际上，机翼四周的气流并不只是在2D空间活动而已，也就是说，气流除了一致地流过每个平行截面外，也有某些气流在主翼上跨区域地流动。这些气流就是在3D空间中活动的。在3D空间中活动的气流越多，主翼所能提供的效率越低（注3）。这些影响在较长或高展弦比的机翼（有装设端板的主翼会使有效展弦比增加）上会减少。也就是说，比较起来高展弦比的机翼效率较高。

为了要处理展弦比的影响，通常会用同一种机翼截面设计搭配不同的展弦比做测试，得到一系列升力与阻力对攻角的曲线。另外还有其他几何方面的因素，会影响机翼附近的气流。如果想要对这方面有严谨的认识，可以参考“Theory of Wing Sections”与“Fluid-Dynamic Lift”这两本书（注4）。

回到图7-6，你会发现阻力系数随着攻角而明显地增加。这是合理的，当机翼平面正对气流或者与气流垂直时，理所当然地会增加飞行时的阻力。

让我们观察升力系数曲线，刚开始时会随着攻角线性地增加，而在某个攻角值时，升力系数会达到最大值。这个时候的角度称为临界攻角。当角度大于临界攻角时，升力系数会下降得非常快，此时机翼会失速并停止产生升力。这是很糟的状况。当飞机在空中失速时，会很快地降低高度直到飞行员有正确的处置，例如减低俯仰角或是增加推力等等。当发生失速时，机翼后的气流不再平顺地合流，而相对的高攻角因此产生乱流。这种状况如图7-7所示。在失去升力的同时往往会伴随着阻力的增加。

理论上，机翼所受的合力会作用在距离前缘四分之一翼弦长的点上。这个点称为四分弦点（quarter-chord point）。事实上，合力所作用的地点会根据攻角、压力分布和速度等因素而不同。然而在实现时，在典型的环境中假设合力的作用线通过四分弦点是合理

---

注2：“Theory of Wing Sections”一书中收录了标准的机翼截面的几何图形与实现的数据，包含著名的NACA系列的机翼截面。此书的附录中也收录了许多机翼截面设计的升力与阻力数据，当然也有使用襟翼的。

注3：升力效率可以用升力对阻力比的方式表示。升力与阻力比越高，主翼或翼截面的效率越高。

注4：“Fluid-Dynamic Lift”是由Sighard F. Hoerner与Henry V. Borst所著的，而“Fluid-Dynamic Drag”是由Sighard F. Hoerner所著的。书中包含了关于飞机的空气动力学各方面的数据、图表和方程。此外也有高速汽船与汽车的数据。

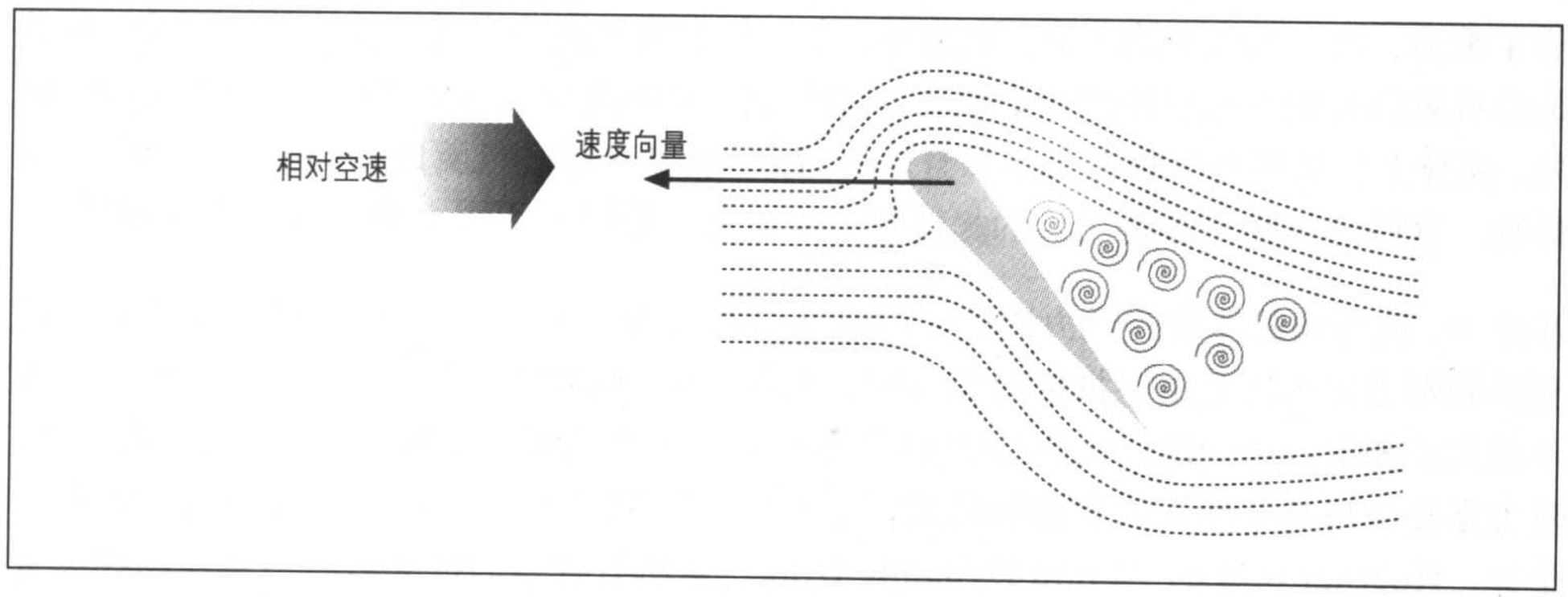



图 7-7：失速的机翼剖面图

的。如果要计算合力作用在四分弦点与实际点之间的差异，就必须考虑四分弦点上的俯仰力矩。俯仰力矩常常会使机翼前缘向下倾。在某些状况下，此力矩比飞机上其他力矩小而可以被忽略（注 5），而当飞机放下襟翼时可能就不能忽略俯仰力矩了。

襟翼是一个控制装置，可以用来变更机翼的外形而改变升力特征。图 7-6 也显示了机翼放下襟翼 15° 时，典型的升力、阻力与力矩系数的值。在襟翼放下后，升力、阻力和俯仰力矩都有很明显的增加。“Theory of Wing Sections”一书中也提供当机翼放下襟翼 −15°～60° 时所产生的数据。

## 其他的作用力

在其他尚未提到的作用力中，最值得注意的就属推力（推进力）了。推力会使飞机向前运动，如果没有推力，飞机的机翼就没有办法提供升力，飞机也就无法飞行了。不管推力是由螺旋桨产生的还是由喷射引擎所产生的，通常都以磅为单位。而用来比较飞机动力的性能时，通常使用推力重量比（简称推重比）来衡量。推重比是推进器所能送出的最大推力除以飞机的总重量。当推重比大于 1 时，飞机在垂直爬升时就能克服重力。在这种状况下，虽然将飞机拉离垂直航道的升力仍会产生，但是机翼产生的升力对维持（或增加）飞机的高度毫无帮助。

不管推力是由螺旋桨产生的还是由喷射引擎所产生的，它都是空气密度的函数。在高空中空气密度（或含氧量）会降低，所以推力也会减小。直到某个高度，引擎会因失速而不再将飞机推高。如果你曾经看过特技飞行秀，也许会看过以这种方式表演的特技飞行。

---

注 5：飞机设计人员在设计飞机结构时必须将俯仰力矩也列入计算，因为此力矩可能会将机翼从机身上扭下。

除了重力、推力和机翼的升力、阻力外，在飞行中的飞机还会受到其他的作用力。这些是由机翼以外的其他组件产生的阻力（或升力）。例如机身会增加飞机上整体的阻力。此外，机身上有某些突出物也会增加整体的阻力。机身上除了机翼以外的突出物总称为悬挂物。这些悬挂物可能是飞机的起落架、座舱罩、炸弹、飞弹、副油箱和进气道等。

基本上，机身和悬挂物的阻力资料可写成类似第六章以阻力系数表示的形式，而由实验测得的阻力其因次是由正投影面积（$S$）、密度（$\rho$）和速度平方（$V^2$）所组成的。这表示用实验测得的阻力除以大小$(1/2)\rho V^2 S$可求得无因次的阻力系数。根据所考虑的物体，阻力系数资料是某些几何参数的函数，这些参数包括机翼的攻角，或者是座舱罩的长高比等。Hoerner 所著的“Fluid Dynamic Drag”是所有种类的机身形状与悬挂物相关资料的绝佳参考。

举例来说，当飞机放下起落架准备降落时，轮子（或是其他的机械装置）会增加整体的阻力。Hoerner 的书中指出某些小飞机的起落架设计，会使以正投影面积计算的阻力系数大约在 0.25～0.55 之间。这里也提供另一个比较数据，一般流线型的外挂舱（像是副油箱）只会使阻力系数维持在 0.06～0.26 之间。

飞机在飞行中会受到的另一个阻力就是表面摩擦力。飞机机翼、机身和悬挂物并不是完全光滑的。焊接点、铆钉，甚至是涂料常常会导致飞机表面的瑕疵而增加摩擦阻力。如第六章中球体的例子中，摩擦阻力是由围绕在飞机部位周围的气流本质所决定的，也就是说不用管气流是层流或是湍流。这表明表面的摩擦阻力系数通常是雷诺数的函数。

对某架飞行中的飞机做严谨的分析时，自然而然地会考虑所有的阻力分力。如果读者有兴趣了解这些计算的核心细节，建议你阅读 Hoerner 的“Fluid Dynamic Drag”，该书的第十四章有计算战斗机上总阻力的详细例子。

## 控制

在我们使用的模型中，装设在主翼后缘内侧的襟翼用来改变弧形主翼截面的弦长与弧度，以在给定的速度下增加升力。襟翼主要在低速飞行时增加升力，例如起飞或降落时。当降落时，襟翼通常会放在向下的高角度上（襟翼往下偏折的角度视为正值），约在 30°～60°。这样会同时增加机翼的升力与阻力。落地时，阻力的增加正好帮助飞机的速度降至适合降落的速度。起飞时，阻力的增加会迫使飞机必须加大推力才能达到预定的速度。所以此时襟翼不能放在像降落时那样大的角度。

控制副翼或产生滚转动作的原理，都是在左右主翼上产生不同升力所导致的。基本上副翼只是一对像襟翼一样的控制翼，只不过是装设在主翼外侧而已。一对副翼向不同的方

向偏折（一个向下而另一个向上），就能产生左右主翼的升力差。副翼之间距离的升力差，会产生使飞机滚转的力矩。如果要让飞机向左滚转（飞行员的左方），那么要将右副翼偏下而左副翼偏上。同样，相反的偏折方向能使飞机向右滚转。真实的飞机中，左右移动操纵杆便可控制副翼的偏折。

升降舵（或称尾翼）用来控制飞机的俯仰（升降舵可以算是襟翼，如图 7-2 所示，也可以算是整片可以旋转的尾翼，例如洛克希德马丁（Lockheed Martin）公司制造的 F-16 战斗机）。当升降舵偏折使后缘向下（相对于飞行员）时，因反作用力而使飞机的机尾上抬、机鼻朝下，如此一来飞机会向下俯冲。真实的飞机中，飞行员将操纵杆向前推即可做俯冲的动作。当升降舵偏折使后缘向上时，飞机会做向上爬升的动作。

在调整飞机的姿态（俯仰）时，升降舵是非常重要的。一般来说，飞机的重心大约在机翼的平均四分弦线的上方，因此重心位于主要升力作用的线上。然而如同之前解释的一样，升力不一定会作用在四分弦点上。此外，飞机的重心在飞行中也会改变，例如油料的损耗和飞弹的发射而引起的改变等。因此飞行员要通过控制升降舵来调整飞机的姿态并平衡作用在飞机上所有的作用力，让飞机能朝着固定的航向（或俯仰角）飞行。

最后，方向舵用来控制飞机的偏转。飞行员可以使用脚踏板来控制方向舵。踩左边的踏板可以让飞机向左（左舷）偏转，而踩右边的踏板可以让飞机向右（右舷）偏转。当在降落或瞄准目标时，使用方向舵来调整飞机的航向是很方便的。基本上，较大的方向舵在偏折时通常也会导致飞机滚转，必须适当地用副翼来抵消方向舵的作用。

某些状况下，方向舵的后缘会装设垂直的襟翼。而有些状况下，并不装设方向舵襟翼，而是让整个垂直尾翼都能旋转。这两种能提供方向稳定性的垂直尾翼，通常会有个对称的翼切形，也就是说翼弧中线与翼弦线相等。当飞机水平地直线飞行时，垂直的尾翼不会产生升力，因为它是对称的而且攻角为零。然而当飞机以侧向飞行（也就是飞机机首与飞行方向有夹角）时，尾翼就会有攻角并产生升力，让飞机改变原始航向。

## 飞行模拟

虽然尚未完全介绍在实现即时飞行模拟时所有必需的知识，仍要进一步介绍计算飞机模型的升力与阻力的几个必需的步骤。以下是这些步骤：

1. 将升力面分成一些更小的机翼零件。
2. 收集几何造型与机翼的性能资料。
3. 计算每个机翼零件的相对空速。

4. 计算每个机翼零件的攻角。

5. 确定正确的升力与阻力系数，并计算出升力与阻力。

第一步必然是要将整架飞机分解成较小的零件，每个零件大约都有相同的特性。要对图7-2中的模型实施这个步骤，必须要将机翼分成四个零件 —— 两个装设副翼的零件（左右各一），以及两个装设襟翼的零件（左右各一）。同样，要将升降舵分成两个零件 —— 左舷和右舷，而尾翼/方向舵只有一个零件。最后，可以将全部的机身看成同一个零件，或者是将它分成几个更小的零件，由你想要实现的真实度而定。

如果要模拟飞机的刚体运动，就必须计算飞机在飞行时所受的力与力矩。因为飞机是由许多不同的零件构成的，每个零件都会影响总升力与总阻力。我们必须分成好几个小零件来计算，再相加才能得到总升力与总阻力。当然也可以加入其他的组件，如驾驶舱的座舱罩、起落架、副油箱和炸弹等，使你的模型更加真实。模型的真实度取决于要模拟的精确程度。如果要模拟特定飞机的飞行姿态，则必须在模型上多花点时间。

定义了每个零件后，接着要准备正确的几何造型与性能资料。举例来说，对于主翼和其他的升力面，必须决定每个零件的初始入射角（相对于飞机参考坐标的固定俯仰角或攻角）、翼展、翼弦长、展弦比、平面面积，以及相对于飞机重心的四分弦点。此外还要为考虑的零件准备适当的升力与阻力系数对攻角的资料表。由于这些资料大多是图形表格，因此你必须从图形中取出适当的资料建立查询表，以在游戏程序中使用。最后，还要计算垂直于每个机翼零件的单位法向量（稍后计算攻角时会需要这个数据）。

以上两个步骤只需要在游戏或模拟程序的开头做一次就好，因为这些资料通常只是常数而已（除非你的飞机会改变形状，或飞机的重心位置在模拟过程中会改变）。

第三个步骤包含了计算空气与飞机每个零件的相对速度，这样才能计算升力与阻力。虽然刚开始看可能会觉得很繁琐，但是飞机在空气中行进的速度对模拟程序是非常重要的。然而，既然把飞机视为刚体，它的重心除了有线性速度外，还必须处理角速度。

回到第二章的公式，可以用来计算任一刚体上的点在线性及角运动时的相对速度：

$$v_R = v_{cg} + (\omega \times r)$$

这个公式将用来计算飞机模型每个零件的相对速度。其中，$v_{cg}$代表空气速度及飞机飞行的方向，$\omega$代表飞机的角速度向量，而$r$是从飞机的重心到零件的距离向量。

当处理机翼时，既然已经知道相对速度向量，就可以计算每个机翼零件的攻角。阻力向量会与相对速度向量平行，而升力向量会与速度向量垂直。攻角是升力速度向量与机翼的法向量之间的夹角。这牵扯到两向量的内积。

求得攻角之后就可以从升力与阻力系数对攻角的表格中，找到此刻需用的升力与阻力系数。有了这些系数，可以用下列的公式估计作用在机翼某个部分升力与阻力的量值：

$$升力 = C_L(1/2)\rho V^2 S$$

$$阻力 = C_D(1/2)\rho V^2 S$$

这里提到的方法是经过简化的，它只能估计升力与阻力的特征。这个方法并没有考虑跨区域的气流效应，或邻近区域间的气流效应。此方法当然也没有考虑空气的乱流（如下冲气流），这可能会影响每个机翼零件的相对攻角。也就是说，我们将每个机翼零件的气流假设为稳定且一致的。

从简单的范例开始，先看主翼的1号板即右舷副翼部分。假设主翼的初始入射角设定为3.5°，而飞机正以75 kt（节；海里/小时）低空直线飞行，俯仰角是4.5°。这个主翼部分的翼弦长为5.2 ft，而翼展为6 ft。使用在图7-6中的升力与阻力资料，计算这个主翼零件的升力与阻力，并假设副翼没有偏折而空气密度为$2.37 \times 10^{-3}$ slug/ft$^3$。

第一步是根据目前的资料计算攻角（8°）。接着查看表7-6，可以找出机翼的升力与阻力系数分别为0.75和0.013。

接着计算这个区域的面积，只需要将翼弦长乘以翼展长即可。计算出的值是31.2 ft$^2$。现在有足够的资料可计算升力与阻力了（别忘了将速度单位由kt转换成ft/s；1 kt = 1.688 ft/s）。计算方式如下：

$$升力 = C_L(1/2)\rho V^2 S$$

$$升力 = 0.75(1/2)(2.37 \times 10^{-3}\ \text{slug/ft}^3)[(75\ \text{kt})(1.688\ \text{ft/s/kt})]^2(31.2\ \text{ft}^2)$$

$$升力 = 442.0\ \text{lb}$$

$$阻力 = C_D(1/2)\rho V^2 S$$

$$阻力 = 0.013(1/2)(2.37 \times 10^{-3}\ \text{slug/ft}^3)[(75\ \text{kt})(1.688\ \text{ft/s/kt})]^2(31.2\ \text{ft}^2)$$

$$阻力 = 8.0\ \text{lb}$$

在你的模拟程序中，必须为每个定义的零件都做类似的运算。你可以看到，虽然这类实验数据与公式只能用来估计，但是却让计算变得很简单。较难的工作在于决定飞机模型和找出正确的资料，之后计算升力与阻力就相当简单了。

下面的范例程序教你如何使用这里介绍的方法模拟简单的飞机飞行。这个程序名称是FlightSim.exe，它是即时的3D飞行模拟程序（注6）。这架模拟的飞机与图7-2中介绍的飞机模型相似。

这个程序包含以下的原始档和一个文字档*Instructions.txt*来说明飞行控制的方法：

- *Physics.cpp*与*Physics.h*
- *D3dstuff.cpp*与*D3dstuff.h*
- *Mymath.h*
- *Winmain.cpp*

此程序是实时模拟程序，并将飞机视为刚体。本书还没谈到实时模拟，因此有很多程序代码在此时可能不好懂。别担心，本书后面几章将会提供所有关于实时模拟的知识。然而目前将焦点放在实现飞行模型的函数上。这些函数都在原始档*Physics.cpp*中。

第一个要介绍的函数是CalcAirplaneMassProperties：

```
//-----------------------------------------------------------------//
// 这个模型使用8个不同的零件来表示一架飞机
// 这些是：
//
//          零件1：左舷外侧主翼装设副翼的部分
//          零件2：左舷内侧主翼装设襟翼的部分
//          零件3：右舷内侧主翼装设襟翼的部分
//          零件4：右舷外侧主翼装设副翼的部分
//          零件5：左舷升降舵装设襟翼的部分
//          零件6：右舷升降舵装设襟翼的部分
//          零件7：垂直尾翼/方向舵（无襟翼，整片控制翼都能转动）
//          零件8：机身
//
//  这个函数先设定每个零件的参数。接着计算总重量、重心，以及飞机的惯性张量。
//  当计算飞机的升力与阻力时，每个零件所需的其他特性也一并计算。
//-----------------------------------------------------------------//
void     CalcAirplaneMassProperties(void)
{
     float     mass;
     Vector    vMoment;
     Vector    CG;
     int       i;
     float     Ixx, Iyy, Izz, Ixy, Ixz, Iyz;
     float     in, di;
```

注6：在这个程序中使用了微软的Direct3D函数库。因此在执行这个程序之前，必须确定电脑中已经安装了Direct3D函数库。

```
// 在这里初始化每个零件
// 刚开始时，零件的坐标是参考设计的坐标系而来的
// 原点位于飞机的最末端、基线和中线。稍后，这些坐标会转换
// 为以参考飞机重心的坐标系而定的坐标。
Element[0].fMass = 6.56f;
Element[0].vDCoords = Vector(14.5f, 12.0f, 2.5f);
Element[0].vLocalInertia = Vector(13.92f, 10.50f, 24.00f);
Element[0].fIncidence = -3.5f;
Element[0].fDihedral = 0.0f;
Element[0].fArea = 31.2f;
Element[0].iFlap = 0;

Element[1].fMass = 7.31f;
Element[1].vDCoords = Vector(14.5f, 5.5f, 2.5f);
Element[1].vLocalInertia = Vector(21.95f, 12.22f, 33.67f);
Element[1].fIncidence = -3.5f;
Element[1].fDihedral = 0.0f;
Element[1].fArea = 36.4f;
Element[1].iFlap = 0;

Element[2].fMass = 7.31f;
Element[2].vDCoords = Vector(14.5f, -5.5f, 2.5f);
Element[2].vLocalInertia = Vector(21.95f, 12.22f, 33.67f);
Element[2].fIncidence = -3.5f;
Element[2].fDihedral = 0.0f;
Element[2].fArea = 36.4f;
Element[2].iFlap = 0;

Element[3].fMass = 6.56f;
Element[3].vDCoords = Vector(14.5f, -12.0f, 2.5f);
Element[3].vLocalInertia = Vector(13.92f, 10.50f, 24.00f);
Element[3].fIncidence = -3.5f;
Element[3].fDihedral = 0.0f;
Element[3].fArea = 31.2f;
Element[3].iFlap = 0;

Element[4].fMass = 2.62f;
Element[4].vDCoords = Vector(3.03f, 2.5f, 3.0f);
Element[4].vLocalInertia = Vector(0.837f, 0.385f, 1.206f);
Element[4].fIncidence = 0.0f;
Element[4].fDihedral = 0.0f;
Element[4].fArea = 10.8f;
Element[4].iFlap = 0;

Element[5].fMass = 2.62f;
Element[5].vDCoords = Vector(3.03f, -2.5f, 3.0f);
Element[5].vLocalInertia = Vector(0.837f, 0.385f, 1.206f);
Element[5].fIncidence = 0.0f;
Element[5].fDihedral = 0.0f;
Element[5].fArea = 10.8f;
Element[5].iFlap = 0;

Element[6].fMass = 2.93f;
Element[6].vDCoords = Vector(2.25f, 0.0f, 5.0f);
```

```
Element[6].vLocalInertia = Vector(1.262f, 1.942f, 0.718f);
Element[6].fIncidence = 0.0f;
Element[6].fDihedral = 90.0f;
Element[6].fArea = 12.0f;
Element[6].iFlap = 0;

Element[7].fMass = 31.8f;
Element[7].vDCoords = Vector(15.25f, 0.0f, 1.5f);
Element[7].vLocalInertia = Vector(66.30f, 861.9f, 861.9f);
Element[7].fIncidence = 0.0f;
Element[7].fDihedral = 0.0f;
Element[7].fArea = 84.0f;
Element[7].iFlap = 0;

// 计算垂直于每个升力面的法向量
// 在为升力与阻力计算相对空速时需要这个值
for (i = 0; i< 8; i++)
{
     in = DegreesToRadians(Element[i].fIncidence);
     di = DegreesToRadians(Element[i].fDihedral);
     Element[i].vNormal = Vector((float)sin(in),(float)(cos(in)*sin(di)),
                                 (float)(cos(in)*cos(di)));
     Element[i].vNormal.Normalize();
}

// 计算总质量
mass = 0;
for (i = 0; i< 8; i++)
     mass += Element[i].fMass;

// 计算合并所有零件后的重心位置
vMoment = Vector(0.0f, 0.0f, 0.0f);
for (i = 0; i< 8; i++)
{
     vMoment += Element[i].fMass*Element[i].vDCoords;
}
CG = vMoment/mass;

// 计算每个零件相对于合并后重心的坐标
for (i = 0; i< 8; i++)
{
     Element[i].vCGCoords = Element[i].vDCoords - CG;
}

// 现在计算每个零件合并后的力矩与惯性积
// 惯性矩阵（张量）是以局部坐标计算的
Ixx = 0;    Iyy = 0;    Izz = 0;
Ixy = 0;    Ixz = 0;    Iyz = 0;
for (i = 0; i< 8; i++)
{
     Ixx += Element[i].vLocalInertia.x + Element[i].fMass *
            (Element[i].vCGCoords.y*Element[i].vCGCoords.y +
            Element[i].vCGCoords.z*Element[i].vCGCoords.z);
```

```
            Iyy += Element[i].vLocalInertia.y + Element[i].fMass *
                (Element[i].vCGCoords.z*Element[i].vCGCoords.z +
                Element[i].vCGCoords.x*Element[i].vCGCoords.x);
            Izz += Element[i].vLocalInertia.z + Element[i].fMass *
                (Element[i].vCGCoords.x*Element[i].vCGCoords.x +
                Element[i].vCGCoords.y*Element[i].vCGCoords.y);
            Ixy += Element[i].fMass * (Element[i].vCGCoords.x *
                Element[i].vCGCoords.y);
            Ixz += Element[i].fMass * (Element[i].vCGCoords.x *
                Element[i].vCGCoords.z);
            Iyz += Element[i].fMass * (Element[i].vCGCoords.y *
                Element[i].vCGCoords.z);
        }

        // 最后，设定飞机的质量和惯性矩阵，并且求这个矩阵的反矩阵
        Airplane.fMass = mass;
        Airplane.mInertia.e11 = Ixx;
        Airplane.mInertia.e12 = -Ixy;
        Airplane.mInertia.e13 = -Ixz;
        Airplane.mInertia.e21 = -Ixy;
        Airplane.mInertia.e22 = Iyy;
        Airplane.mInertia.e23 = -Iyz;
        Airplane.mInertia.e31 = -Ixz;
        Airplane.mInertia.e32 = -Iyz;
        Airplane.mInertia.e33 = Izz;

        Airplane.mInertiaInverse = Airplane.mInertia.Inverse();
    }
```

这个函数基本上完成了模拟方法中的第1个步骤（和部分的第2个步骤）：将飞机分成几个小零件，每个零件都有自己的质量和升力与阻力参数。这个范例将飞机分成8个零件；函数最前面的注解解释这些零件所代表的部位。

这个函数刚开始先初始化飞机所有零件模拟飞机特性的参数。每个零件都有质量、一组质心的设计坐标、一组对个别质心的转动惯量、初始入射角、平面面积和上反角(dihedral angle)。

设计坐标的原点位于机尾末端、飞机中线和基线上（中线是飞机俯视图的对称线，而基线则坐落在飞机底部）。坐标系的$x$轴指向飞机的机鼻，$y$轴指向飞机的左舷，$z$轴指向上方。刚开始必须要将每个零件的坐标设定成设计坐标，因为这时候并不知道飞机整体的质心（所有零件合并后的质心）。最后，要将零件的坐标转换成参考飞机总质心的坐标，因为它是在模拟中追踪的质心（请复习第二章、第四章所讨论的内容）。

上反角是每个零件初始设定的绕$x$轴与水平面所夹的角（有些机翼会有向上翘的角度，这个角就是上反角）。在这里的模型中，除了尾翼方向舵以外，飞机所有零件的上反角都是0°。因为其他零件都在水平面上，而只有尾翼方向舵是垂直的，所以有90°的上反角。

在设定完这些零件的初始值后，此函数执行的第一个计算是根据每个零件的入射角和上反角找出单位法向量。以后会需要此方向向量，用来协助计算气流方向与零件之间的攻角。

接着要计算总质量，只要将所有零件的质量累加即可。紧接着使用第一章介绍的方法计算合并之后的总质心。合并后的质心是以设计坐标系计算的。必须要将这个坐标减去每个零件的设计坐标，求出每个零件相对于总质心的坐标。这些都设定好之后，就剩下合并后的转动惯量了，不过这留在第十一章才讨论。

飞行模拟方法的第2步是收集所有飞机性能的数据。在这个范例程序中，使用弧面翼切形和平面襟翼做为主翼和升降舵，并且用无襟翼的对称翼切形作为方向舵。不在尾翼方向舵装设襟翼的原因是要让整片方向舵绕垂直轴旋转以调整方向。

模拟程序中为主翼设计两个函数，来处理升力与阻力系数：

```
//-------------------------------------------------------------------//
// 此函数可由传入的攻角和襟翼的状态为装设平面襟翼的弧形机翼
// 计算并返回正确的升力系数（襟翼最大偏折角度为正负15°）
//-------------------------------------------------------------------//
float       LiftCoefficient(float angle, int flaps)
{
      float clf0[9] = {-0.54f, -0.2f, 0.2f, 0.57f, 0.92f, 1.21f, 1.43f,
                       1.4f, 1.0f};
      float clfd[9] = {0.0f, 0.45f, 0.85f, 1.02f, 1.39f, 1.65f, 1.75f,
                       1.38f, 1.17f};
      float clfu[9] = {-0.74f, -0.4f, 0.0f, 0.27f, 0.63f, 0.92f, 1.03f,
                       1.1f, 0.78f};
      float a[9]    = {-8.0f, -4.0f, 0.0f, 4.0f, 8.0f, 12.0f, 16.0f,
                       20.0f, 24.0f};
      float  cl;
      int    i;
      cl = 0;
      for (i=0; i<8; i++)
      {
          if( (a[i] <= angle) && (a[i+1] > angle) )
          {
              switch(flaps)
              {
                case 0:// 襟翼无偏折
                     cl = clf0[i] - (a[i] - angle) * (clf0[i] - clf0[i+1]) /
                         (a[i] - a[i+1]);
                     break;
                case -1: // 襟翼向下
                     cl = clfd[i] - (a[i] - angle) * (clfd[i] - clfd[i+1]) /
                         (a[i] - a[i+1]);
                     break;
                case 1: // 襟翼向上
                     cl = clfu[i] - (a[i] - angle) * (clfu[i] -clfu[i+1]) /
```

```
                                    (a[i] - a[i+1]);
                            break;
                        }
                        break;
                    }
            }

            return cl;
    }

//-------------------------------------------------------------//
// 此函数可由传入的攻角和襟翼的状态，为装设平面襟翼的弧形机翼
// 计算并返回正确的升力系数（襟翼最大偏折角度为正负 15°）
//-------------------------------------------------------------//
float    DragCoefficient(float angle, int flaps)
{
    float cdf0[9] = {0.01f, 0.0074f, 0.004f, 0.009f, 0.013f, 0.023f,
                     0.05f, 0.12f, 0.21f};
    float cdfd[9] = {0.0065f, 0.0043f, 0.0055f, 0.0153f, 0.0221f,
                     0.0391f, 0.1f, 0.195f, 0.3f};
    float cdfu[9] = {0.005f, 0.0043f, 0.0055f, 0.02601f, 0.03757f,
                     0.06647f, 0.13f, 0.18f, 0.25f};
    float a[9]    = {-8.0f, -4.0f, 0.0f, 4.0f, 8.0f, 12.0f, 16.0f,
                     20.0f, 24.0f};
    float cd;
    int  i;

    cd = 0.5;
    for (i=0; i<8; i++)
    {
        if( (a[i] <= angle) && (a[i+1] > angle) )
        {
            switch(flaps)
            {
              case 0:// 襟翼无偏折
                   cd = cdf0[i] - (a[i] - angle) * (cdf0[i] - cdf0[i+1]) /
                        (a[i] - a[i+1]);
                   break;
              case -1: // 襟翼向下
                   cd = cdfd[i] - (a[i] - angle) * (cdfd[i] - cdfd[i+1]) /
                        (a[i] - a[i+1]);
                   break;
                case 1: // 襟翼向上
                   cd = cdfu[i] - (a[i] - angle) * (cdfu[i] - cdfu[i+1]) /
                        (a[i] - a[i+1]);
                   break;
            }
            break;
        }
    }

    return cd;
}
```

这两个函数都以攻角作为参数，另一个参数是 flap，用来表示襟翼的状态是中间位置、偏折向上还是偏折向下。升力与阻力系数的数据与离散的攻角相关，也就是说可以用线性内插法来决定两个离散的攻角之间的系数。

用来计算尾翼方向舵的升力与阻力系数的函数类似于计算主翼的函数，惟一不同点在于系数本身，以及尾翼并没有装设襟翼。以下是函数的程序代码：

```
//-------------------------------------------------------------//
// 此函数可由传入的攻角值，为对称翼切形而无襟翼的尾翼
// 计算并返回正确的升力系数。
//-------------------------------------------------------------//
float   RudderLiftCoefficient(float angle)
{
   float clf0[7] = {0.16f, 0.456f, 0.736f, 0.968f, 1.144f, 1.12f, 0.8f};
   float a[7] = {0.0f, 4.0f, 8.0f, 12.0f, 16.0f, 20.0f, 24.0f};
   float cl;
   int       i;
   float     aa = (float) fabs(angle);

   cl = 0;
   for (i=0; i<8; i++)
   {
         if( (a[i] <= aa) && (a[i+1] > aa) )
         {
         cl = clf0[i] - (a[i] - aa) * (clf0[i] - clf0[i+1]) /
                        (a[i] - a[i+1]);
               if (angle < 0) cl = -cl;
               break;
         }
   }
   return cl;
}

//-------------------------------------------------------------//
// 此函数可由传入的攻角，为对称翼切形而无襟翼的尾翼
// 计算并返回正确的阻力系数。
//-------------------------------------------------------------//
float       RudderDragCoefficient(float angle)
{
      float cdf0[7] = {0.0032f, 0.0072f, 0.0104f, 0.0184f, 0.04f,
                       0.096f, 0.168f};
      float a[7]  = {0.0f, 4.0f, 8.0f, 12.0f, 16.0f, 20.0f, 24.0f};
      float cd;
      int   i;
      float aa = (float) fabs(angle);

      cd = 0.5;
      for (i=0; i<8; i++)
      {
            if( (a[i] <= aa) && (a[i+1] > aa) )
            {
```

```
                    cd = cdf0[i] - (a[i] - aa) * (cdf0[i] - cdf0[i+1]) /
                        (a[i] - a[i+1]);
                    break;
                }
            }
            return cd;
    }
```

结束第1、2步之后，第3、4、5步是由CalcAirplaneLoads函数来负责的：

```
//-----------------------------------------------------//
// 此函数计算在某个时间，作用在飞机上的合力和合力矩
//-----------------------------------------------------//
void     CalcAirplaneLoads(void)
{
      Vector     Fb, Mb;

      // reset forces and moments:
      Airplane.vForces.x = 0.0f;
      Airplane.vForces.y = 0.0f;
      Airplane.vForces.z = 0.0f;

      Airplane.vMoments.x = 0.0f;
      Airplane.vMoments.y = 0.0f;
      Airplane.vMoments.z = 0.0f;

      Fb.x = 0.0f; Mb.x = 0.0f;
      Fb.y = 0.0f; Mb.y = 0.0f;
      Fb.z = 0.0f; Mb.z = 0.0f;

      // 定义作用在飞机的重心上的推力向量
      Thrust.x = 1.0f;
      Thrust.y = 0.0f;
      Thrust.z = 0.0f;
      Thrust *= ThrustForce;

      // 计算局部坐标空间中的力与力矩
      Vector    vLocalVelocity;
      float    fLocalSpeed;
      Vector    vDragVector;
      Vector    vLiftVector;
      float    fAttackAngle;
      float    tmp;
      Vector          vResultant;
      int      i;
      Vector          vtmp;

      Stalling = false;

      for(i=0; i<7; i++) // 用循环计算每个零件，但略过机身
      {
            if (i == 6) // 尾翼方向舵是特殊情况，因为它能旋转
            {           // 因此要再次计算法向量
```

```
            float in, di;
            in = DegreesToRadians(Element[i].fIncidence); // 入射角
            di = DegreesToRadians(Element[i].fDihedral);  // 上反角
            Element[i].vNormal = Vector(  (float)sin(in),
                                          (float)(cos(in)*sin(di)),
                                          (float)(cos(in)*cos(di)));
            Element[i].vNormal.Normalize();
        }

        // 计算每个零件本身的速度，包括飞机线性运动的速度
        // 加上飞机旋动的速度

        // 以下是旋转的部分
        vtmp = Airplane.vAngularVelocity^Element[i].vCGCoords;

        vLocalVelocity = Airplane.vVelocityBody + vtmp;

        // 计算本身的空速
        fLocalSpeed = vLocalVelocity.Magnitude();

        // 求出阻力作用的方向
        // 阻力通常与相对速度位于同一直线上，但方向相反
        if(fLocalSpeed > 1.)
              vDragVector = -vLocalVelocity/fLocalSpeed;

        // 求出升力作用的方向
        // 升力向量通常垂直于阻力向量
        vLiftVector = (vDragVector^Element[i].vNormal)^vDragVector;
        tmp = vLiftVector.Magnitude();
        vLiftVector.Normalize();

        // 求出攻角的大小
        // 攻角是升力向量与零件的法向量夹角
        // 攻角的正弦值等于阻力向量和法向量夹角的余弦值
        tmp = vDragVector*Element[i].vNormal;
        if(tmp > 1.) tmp = 1;
        if(tmp < -1) tmp = -1;
        fAttackAngle = RadiansToDegrees((float) asin(tmp));

        // 计算作用在特定零件的合力（升力和阻力）
        tmp = 0.5f * rho * fLocalSpeed*fLocalSpeed * Element[i].fArea;
        if (i == 6) // 尾翼方向舵
        {
          vResultant = (vLiftVector*RudderLiftCoefficient(fAttackAngle) +
              vDragVector*RudderDragCoefficient(fAttackAngle)) * tmp;
        } else
              vResultant = (vLiftVector*LiftCoefficient(fAttackAngle,
                           Element[i].iFlap) +
                           vDragVector*DragCoefficient(fAttackAngle,
                           Element[i].iFlap) ) * tmp;
        // 检查是否发生失速
        // 可用以下简单的方法检查升力系数是否为0
        // 在真实的状况下，失速警告器会在升力系数到达0
        // 之前就会发出警告，让飞行员有时间能改正错误
```

```
if (i<=0)
{
      if (LiftCoefficient(fAttackAngle, Element[i].iFlap) == 0)
            Stalling = true;
}

// 加合全部的合力
Fb += vResultant;

      // 计算零件的合力对于重心所产生的力矩，并且加合全部的合力矩
      vtmp = Element[i].vCGCoords^vResultant;
      Mb += vtmp;
   }

   // 现在加上推力
   Fb += Thrust;

   // 将力从模型的局部坐标转换成全局坐标
   Airplane.vForces = QVRotate(Airplane.qOrientation, Fb);

   // 考虑重力的影响 (1 g = - 32.174 ft/s^2)
   Airplane.vForces.z += g * Airplane.fMass;

   Airplane.vMoments += Mb;
}
```

这个函数首先重设储存飞机上合力和合力矩的变量。接着设定推力向量。这个步骤在此范例中其实是不重要的，因为假设推力向量总是指向$x$轴的正方向，并通过飞机的重心（因此不会产生力矩）。

计算推力向量后，函数接着用循环逐一计算每个零件的升力与阻力，但是省略机身部分；然而，如果要将机身的阻力也列入计算，就需要在这里加入。

进入循环后，函数会先检查目前计算的飞机部分是否为第7个零件，也就是尾翼方向舵的部分。如果是，就要以目前的入射角为基础，重新计算方向舵的法向量。因为方向舵的入射角会因为玩家按下X键或C键而改变。

接着要计算空气与飞机零件的相对速度。正如之前提过的，相对速度是飞机在空气中移动所产生的线性速度，加上每个零件因为飞机转动而产生的角速度。一旦取得此向量，就可以由相对速度向量的量值得到相对空速。

下一步是找出阻力作用的方向。因为阻力与运动方向相反，所以阻力与相对速度向量共线而反方向。因此所要做的是取相对速度向量的负值，再将结果正规化（就是除以本身的量值）可得到阻力的方向向量。因为此向量已被正规化，所以其长度为1。稍后的计算就是将它乘以阻力，便得到阻力向量。

在得到阻力方向向量之后，函数使用它来决定升力方向向量。因为升力与阻力向量垂直，所以计算升力的方向时，先求阻力方向向量与零件的法向量的外积，接着再求此外积与阻力方向向量的外积。最后将结果正规化就得到升力方向向量。

现在已经得到升力与阻力方向向量，函数接着为目前的零件计算攻角。攻角是升力向量与法向量的夹角。所以先求升力方向向量与法向量的外积，再求此外积值的反余弦（inverse cosine）就能得到攻角的大小。因为阻力向量与升力向量垂直，所以先求阻力方向向量与法向量的外积值，再求此外积值的反正弦（inverse sine）也会得到相同的结果。

到目前为止，已经完成了升力与阻力向量相关的计算，函数继续计算作用在零件上的合力。合力向量只是升力与阻力向量的向量和。请注意这里就是调用计算升力与阻力系数函数的地方，之前所讨论的升力与阻力公式也是用在这里。

在计算完合力之后，函数会检查升力系数是否为0。如果升力系数为0，就设定失速标记，以警告我们飞机即将失速。

最后，将合力累计到总合力向量变量，并取零件的坐标向量与合力的外积以求得合力矩。合力矩随后被累计到总合力矩变量。结束循环后，函数也会将推力向量加到总合力。

到目前为止，所有的力和力矩都是参考物体固定坐标系。惟一还没有计算的是重力的影响，但是重力作用在全局坐标的$y$轴负方向上。所以在计算重力的影响之前，函数必须先将物体受力的向量由局部坐标转换成全局坐标。在范例程序中，用到四元数（quaternion）旋转的技巧（稍后介绍）。

在飞行模型中要做的事情很多。范例程序中剩下的程序代码，会在本书适当的章节中介绍。我惟一未详细介绍的是如何实现Direct3D的部分。在参考文献中列出了一些好的参考资料。

我鼓励你尽量玩玩本范例中的飞行模型。可以改变某些飞机零件的参数，看看会发生什么事。虽然这是个粗糙的模型，不过飞行模拟的结果还是颇真实的。

# 第八章

# 船舰

本章的目的并不是要教导你如何设计船舰，而是要用范例来解释船舰的物理原理，包括浮力、稳定性、虚质量和阻力等。在实现以物理为基础的游戏或模拟程序时，都必须将这些因素考虑在内。典型排水型（displacement-type）的船舰正好都能证明这些理论，而这些理论对于沉入或半沉入流体的物体大部分也都适用，例如潜水艇和飞行船等。请记住，当讲到浮力时，空气也被视为流体。

虽然水面船舰（就是指能在水面上航行的船舰，也可以说是在水和空气的交界面行进的船舰）类似完全沉入流体的物体，例如潜水艇或飞行船等 —— 都受到浮力的影响。可是这两种船的物理性质还是有很大的不同，本章将特别强调这些差异。这些差异会影响船舰的行为，如果想正确地模拟这类船舰，就必须特别注意这些差别。

因为本章的范例中谈到了船舰，所以必须先介绍船舰的几何造型和术语，以免你无法了解本章所述内容。而本章要介绍的是典型的排水型船舰。“排水型”的意思是指，支撑船舰的只是浮力，而不是在高速快艇或气垫船中所具有的动升力或空气静升力。另外，排水量指的是船舰浮于水面时所排开水的重量（在下一节中将继续讨论这个主题）。

船舰的船壳指的是船排开水的不透水部分。船上的每个部分都被部分地沉入水中的船壳所包覆。船的长度是船首（bow）到船尾（stern）之间的距离。习惯上，有许多的方式可以用来表示船的长度，但是这里用的是船壳的总长度。船首是指船舰最前面的部分，而船尾指的是船舰最后面的部分。当你站在船上并面向船首时，左边称为左舷，而右边称为右舷。船壳的总高度称为船深（depth），而船壳的宽度称为船幅（beam）。当船浮在水面上时，由水平面到船壳底部的距离称为吃水深度（draft）。图8-1中说明了这些术语所指的部分。

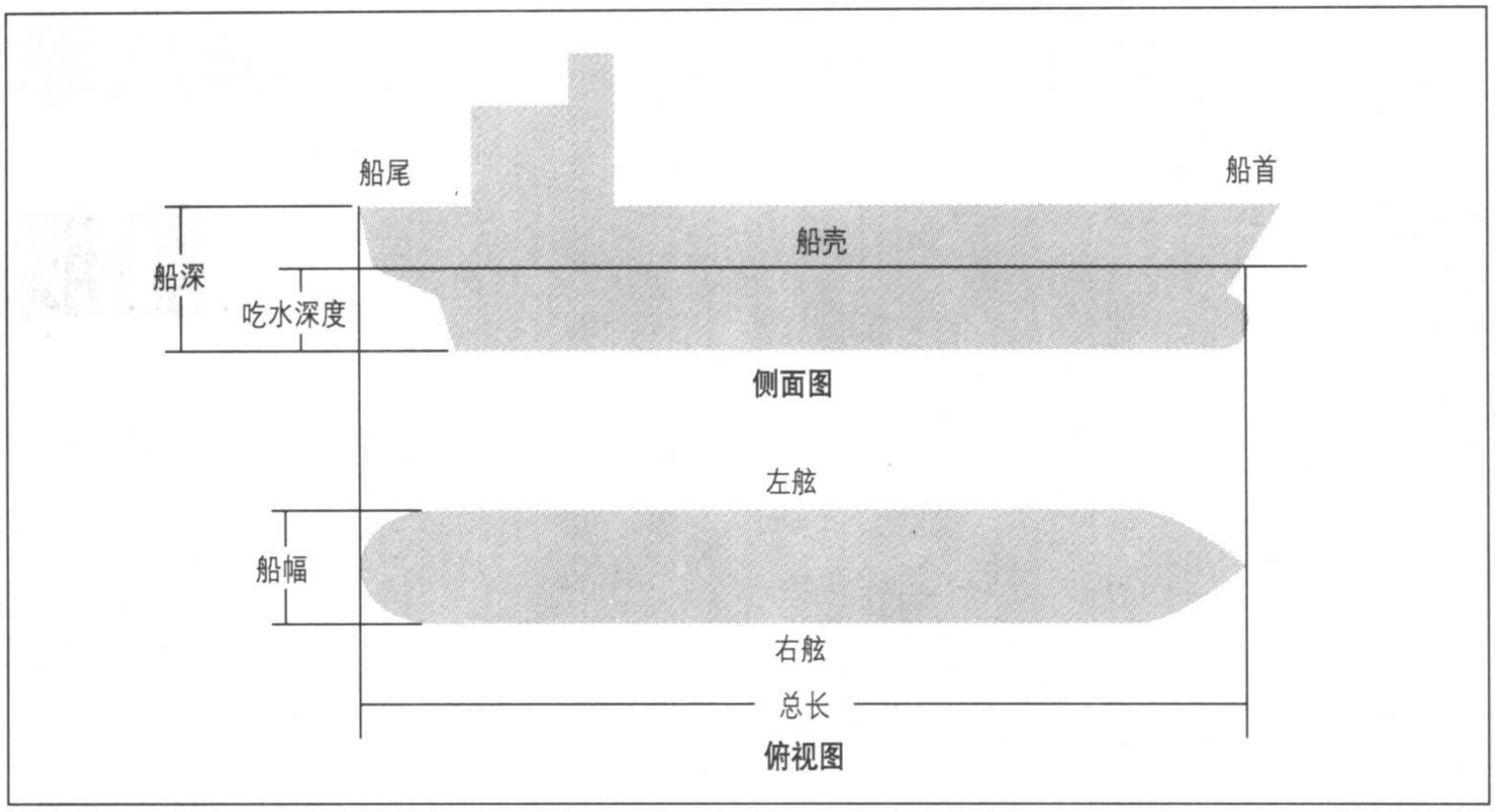



图 8-1：船舰的几何构造

## 漂浮

对于船来说最重要的是能浮起来，而且要能笔直地浮起来。

第三章已经介绍过浮力的概念，并且说明了物体沉入水中所受的浮力是物体沉入体积的函数。根据阿基米德定理，当一个物体浮在流体中时，其重量等于物体所排开的流体体积的重量。这是一个很重要的定理，它说明了给定重量的船舰，必须要有足够的体积排开等于本身重量的水，才能浮在水上。另外此定理提供一个简单的方法来计算船的重量：只需要计算船所排开的水重，就可以得到船的重量。在海事领域中，排水量是船重量的同义词。

任何物体的浮力都可以用下列公式计算：

$$F_B = \rho g \blacktriangledown$$

这里的▼是船沉入的体积，$\rho$是流体的密度。而g是重力加速度。因为浮力是某种作用力，所以它也有量值与方向，而它的方向通常是向上通过浮力中心的。浮力中心指的是物体沉入部分的几何中心。

当一艘船浮在水面上并保持平衡时，它的浮力中心一定在船舰重心的正下方。船的重量

是一个通过重心垂直往下的作用力，会与浮力相抵消。当船在平衡状态时，两力（重力和浮力）的大小相同，但方向相反。

当船翻转或俯仰时，船壳在水中部分的体积改变，而浮力中心也会移到船壳在水中部分的新的几何中心。例如，当船向右舷翻转时，它的浮力中心也朝向右舷移动。此时船的重力和浮力并不会作用在同一条直线上，并且对于船身产生了力矩。这个力矩等于重力与浮力作用线间的垂直距离乘以船重。

接下来要讨论之前所提到，船必须要笔直地浮起来的部分。举例来说，当船开始翻转时，你不会希望它一直倾斜到翻覆，而是当船翻转的力（例如是风力）消失后，船能慢慢地回到原先直立的姿态，简而言之 —— 希望船能稳定。要使一艘船维持稳定，浮力的作用线必须与船舰的中心线交会于重心上方的某个点，此点称为定倾中心（metacenter）。当船开始翻转时，就会产生恢复力矩使船回到原先直立的姿势。如果定倾中心在重心下方，它所产生的力矩会使船更容易翻覆。图 8-2 画出了这两种状况。

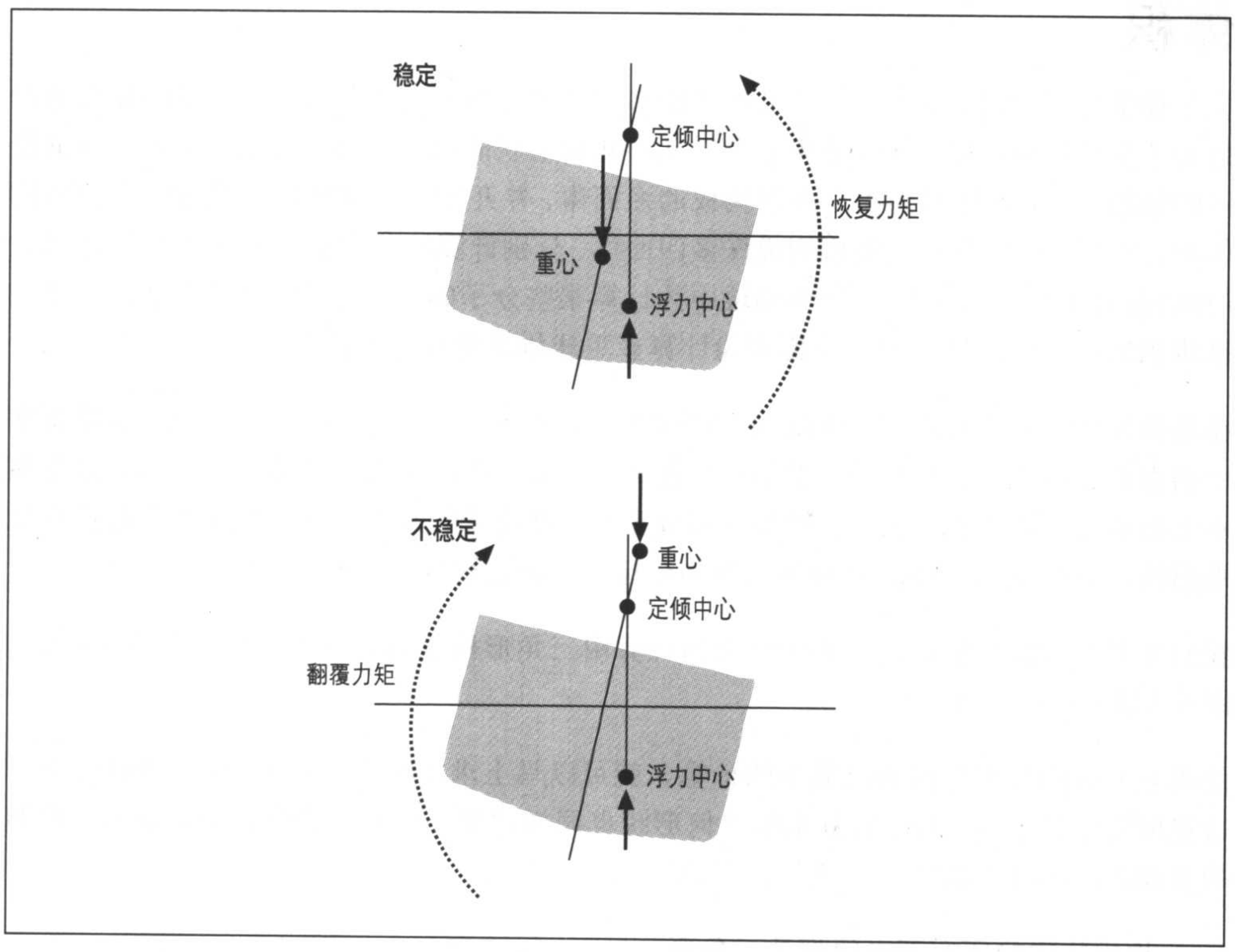



图 8-2：船舰的稳定性

如果你是船员，便会知道尽量降低船重心的重要性。因为这样会增加定倾中心与重心之间的距离，有助于增加船的稳定性。

在完全沉浸的物体（如潜水艇或飞行船）的例子中，情况就不同了。虽然浮力依旧作用于物体的几何中心，但是浮力中心必须在重心的上方才能保持船的稳定性。这样一来，当船开始翻转时，重力和浮力分别作用的直线会被分开，并且产生力矩使船恢复直立的姿势。如果发生其他的状况，就会使船体产生不稳定状况，就像要让一个保龄球在另一个保龄球上保持平衡一样。在这种状况下，一点小小的乱流就会打破这个平衡，而会让物体上下颠倒，使重心跑到浮力中心的下方。

计算中比较困难的部分在于，要用简单的几何学算出沉浸体积和船体的几何中心。船壳通常有许多弯曲的复杂几何形状，而且会有凹陷处和突出物，所以计算船舰的排水体积时需要用到数值积分法。下一节会介绍这个方法。

# 体积

在工程学与科学的领域中，已经发展了许多的技术和演算法来计算体积。这些技巧通常是为了某些特殊的工作，或是为了求出某种几何形状的体积而最佳化的。例如在电脑绘图的领域中，将物体视为由三角形构成的多面体，并开发许多演算法计算这些多面体的体积：先由这些表面的三角形构成许多四面体，分别计算这些四面体的体积再加合起来（稍后会介绍这个方法）。在化学领域中为计算某些分子的体积，同样也开发了另一种计算体积的方法。此领域的方法特别为计算多重球体的体积而最佳化。

在船体设计（正式地说是船体结构）的领域也不例外。计算船舰体积的传统方法是将整个船壳长度的横切面积积分。然而要注意的是，尽管实现的技巧可能不尽相同，但是基本上都是使用数值积分法，将船体分解成更小、更简单的几何形状，而其体积是较容易求出的。然后求出这些部分的体积再加合起来，就能得到总体积。

我们先看看较简单的范例，来解释如何计算由三角形构成的立方体的体积与体积中心。图 8-3 显示此立方体。

使用立方体的原因是因为以简单的计算，就可以马上求出它的体积和形心。但请记住，这里所介绍的方法，对于更复杂的几何形状也同样适用，只要它是简单且可分成三角形的多面体。即这些物体必须满足以下条件：

- 构成物体的所有面必须都是三角形。

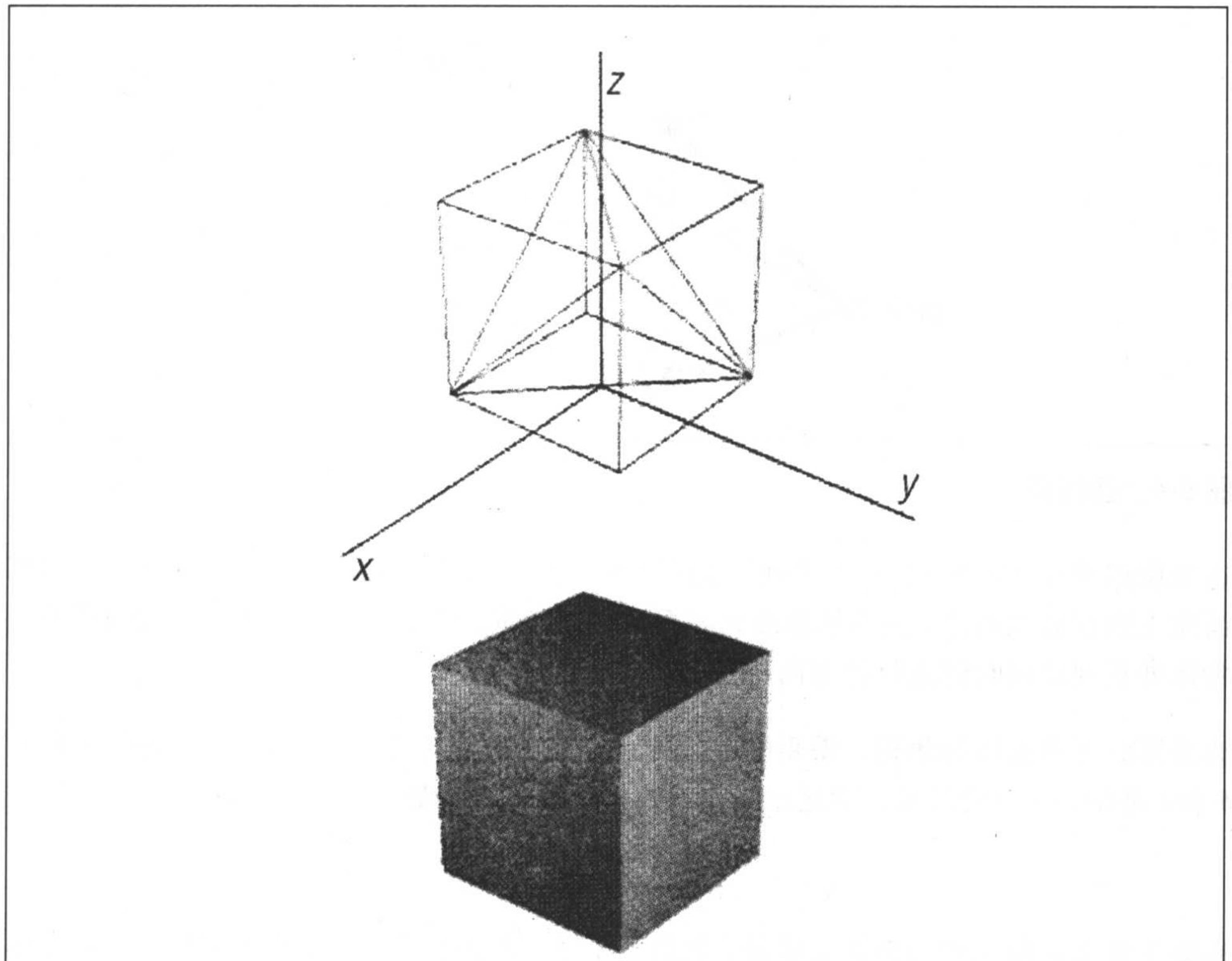



图 8-3：以三角形构成的立方体

- 物体上不能有任何凹洞。
- 物体的体积必须是封闭的 —— 不能有任何不连接的面或边，每边连接两个顶点，并且刚好被两个面共用。
- 这个物体必须满足欧拉方程，也就是顶点的数目减去边的数目再加上面的数目，必须等于 2：顶点的数目 − 边的数目 + 面的数目 = 2。

正如前面说过的，此方法背后的原理是将物体切割成数个四面体，计算每个四面体的体积，再将所有四面体的体积加起来就能得到物体的总体积。当然也能使用这些四面体来计算物体的几何中心（体积的中心），使用的技巧类似于找出质点集合的质心（在第一章已经讨论过），在这种情况下只要把质量换成四面体的体积就好了。图 8-4 说明了如何用三角面构成一个四面体。

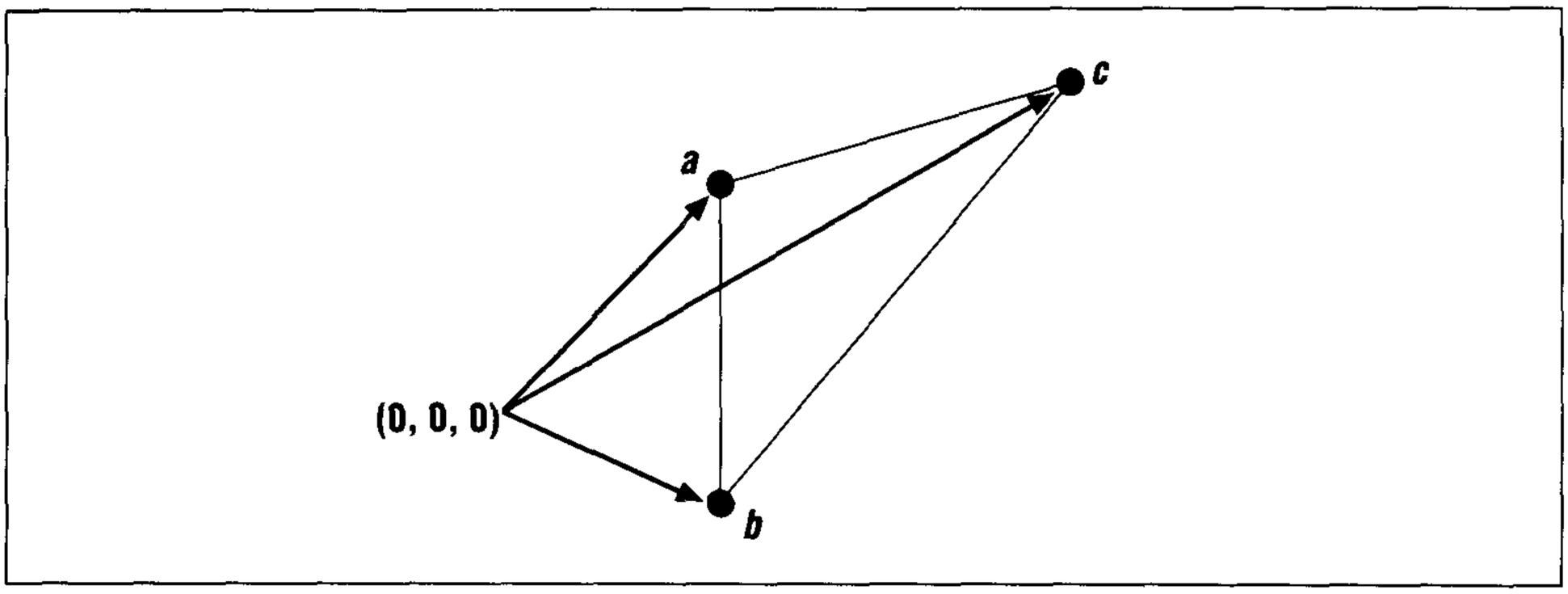



图 8-4：四面体

这里使用原点（0, 0, 0）和三个在三角面上的顶点，来表示四面体的四个顶点。可将四面体上连接原点的边，视为从原点到各面顶点的向量。请注意一个面上顶点的顺序是从物体外正视此面时的逆时针方向排列的。

要计算此类四面体的体积，需要使用向量的标量三重积（请参照附录一）。假设向量 $a$，$b$ 和 $c$ 是图 8-4 中的向量，则这三个向量的标量三重积就是：

$$a \cdot (b \times c)$$

标量三重积最简单的物理意义就是，其值等于图 8-5 中由三个向量所构成的平行六面体的体积（注 1）。

而这三个向量（图 8-4 所示）所构成的四面体体积，是平行六面体积的六分之一。所以计算四面体体积的公式就变成：

$$[a \cdot (b \times c)]/6$$

找出四面体几何中心相当简单：只要计算四个顶点坐标的平均即可。请注意，即使其中的一个顶点位于原点上，求平均时还是要将它列入计算。参考图 8-4 并使用向量表示法，四面体的形心 $d$ 是：

$$d = (a + b + c)/4$$

---

注 1：　平行六面体具有三组平行的边。长方体是各边垂直的平行六面体。立方体（正方体）也是平行六面体的一种，只是各边都等长。

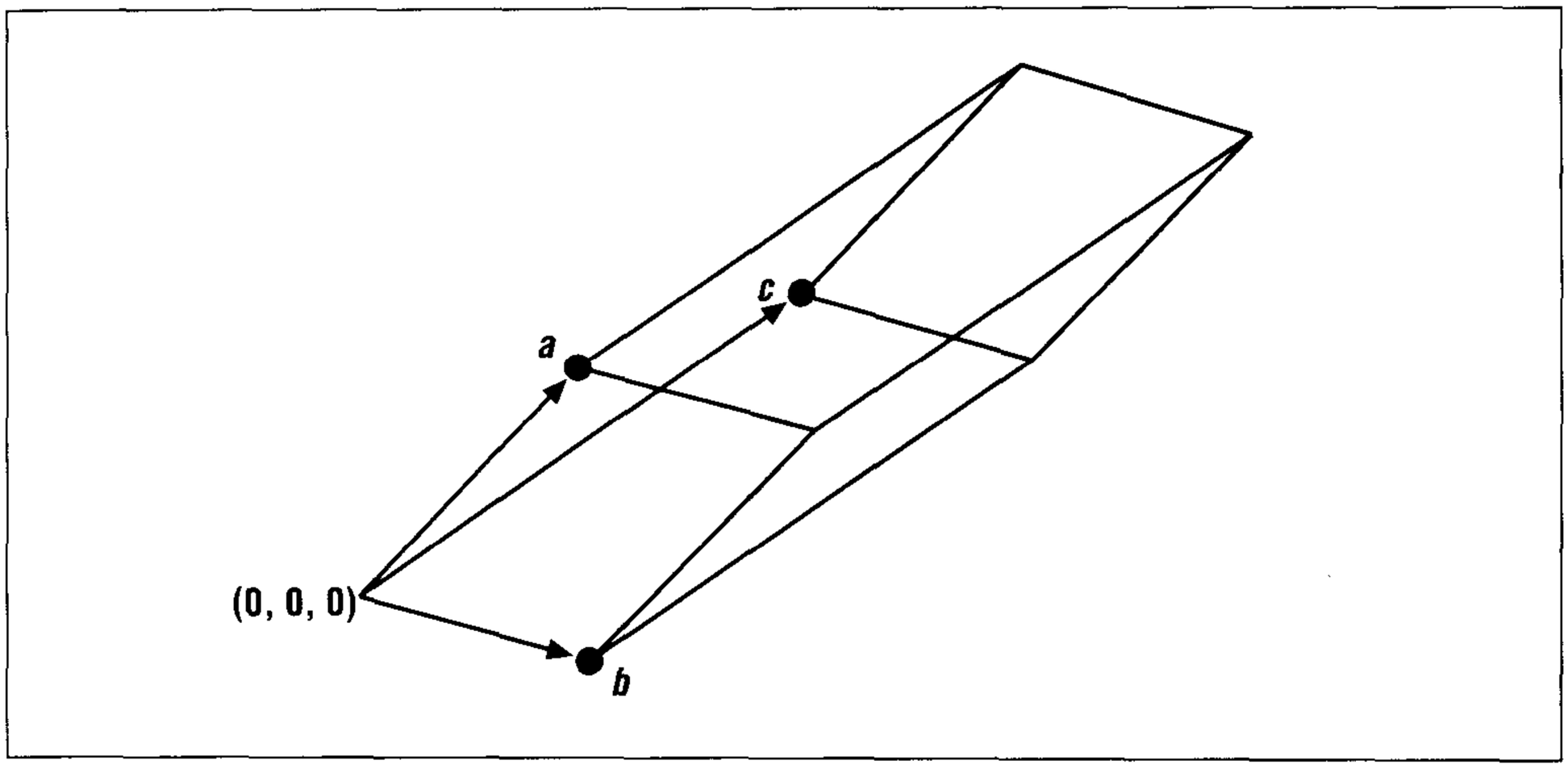



图 8-5：平行六面体

这个公式假设第四个顶点位于原点，坐标为（0, 0, 0）。这就是为什么公式中只有三个向量却除以 4 的原因。

接下来的范例有个简单的Body3D类，存储物体顶点和面的数据。并实现两个成员函数，其中一个可以从文件读入物体的数据，而另一个负责计算物体的体积和形心：

```
#define     MAX_NUM_VERTICES  100
#define     MAX_NUM_FACES     100

typedef struct VertexTag      {
      float x;     // 顶点的x坐标
      float y;     // 顶点的y坐标
      float z;     // 顶点的z坐标

} TVertex;

typedef struct FaceTag {
      // 定义面的顶点顺序是从物体外正视此面时的逆时针方向
      int a; // 第1个顶点（顶点串列中的索引）
      int b; // 第2个顶点（顶点串列中的索引）
      int c; // 第3个顶点（顶点串列中的索引）

} TFace;

//-------------------------------------------------------------------------//
// Body3D类，用来表示由三角形构成的多面体
//-------------------------------------------------------------------------//
class Body3D {
```

```
public:
      int          nFaces;                          // 三角面的数目
      int          nVertices;                       // 顶点的数目
      TVertex      Vertex[MAX_NUM_VERTICES];        // 顶点串列
      TFace        Face[MAX_NUM_VERTICES];          // 面的串列
      float        Volume;                          // 总体积
      Vector       Centroid;                        // 形心

      Body3D(void);                                 // 构造函数
      void  ReadData(char *filename);               // 读入顶点和面的数据
      void  CalculateProperties(void);              // 计算体积和形心

};
```

对于此类中每个成员，由程序代码的注解就能轻易地了解其意义，所以这里就不赘述了。接下来介绍其中两个成员函数（构造函数较不重要，它只是将所有值设定为0而已）。

ReadData()只是从文本文件读入物体的数据：

```
void  Body3D::ReadData(char *filename)
{
      FILE  *fptr;
      int   i;

      fptr = fopen(filename, "r");

      fscanf(fptr, "%d\n", &nVertices);
      for(i=0; i< nVertices; i++)
      {
            fscanf(     fptr,
                  "%f %f %f\n",
                  &(Vertex[i].x),
                  &(Vertex[i].y),
                  &(Vertex[i].z));
      }

      fscanf(fptr, "%d\n", &nFaces);
      for(i=0; i< nFaces; i++)
      {
            fscanf( fptr,
                  "%d %d %d\n",
                  &(Face[i].a),
                  &(Face[i].b),
                  &(Face[i].c));     // 逆时针顺序
      }
      fclose(fptr);
}
```

文件的第一行是一个整数，用来表示以下几行会包含多少个顶点。接着几行是所有顶点的$x$，$y$和$z$坐标（浮点数），每一行代表一个顶点。在所有的顶点数据后，紧接着一个

代表面数目的整数，接下来的几行就是面的数据，每一行有三个数字分别代表顶点的索引值（逆时针方向）。以下是物体的范本文件，定义一个2个单位高、2个单位宽和2个单位深的立方体，而且其底面中心位于原点上并与 *xy* 平面切齐，如图8-3所示。

```
8
-1.000000 -1.000000 0.000000
-1.000000 -1.000000 2.000000
1.000000 -1.000000 0.000000
1.000000 -1.000000 2.000000
-1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
1.000000 1.000000 2.000000
-1.000000 1.000000 2.000000
12
2 3 1
2 1 0
4 5 2
4 2 0
6 3 2
6 2 5
6 7 1
6 1 3
6 5 4
6 4 7
1 7 4
1 4 0
```

下一个成员函数CalculateProperties是本范例的重点，它会将物体切割成一系列的四面体再计算物体的总体积和形心。下面列出程序代码并详加解释：

```
void        Body3D::CalculateProperties(void)
{
      Vector      a;
      Vector      b;
      Vector      c;
      int         i;
      float       dv = 0;
      float       vol = 0;
      Vector      d;
      Vector      dmom;

      for(i=0; i i++)
      {
            a.x = Vertex[Face[i].a].x;
            a.y = Vertex[Face[i].a].y;
            a.z = Vertex[Face[i].a].z;

            b.x = Vertex[Face[i].b].x;
            b.y = Vertex[Face[i].b].y;
            b.z = Vertex[Face[i].b].z;
```

```
            c.x = Vertex[Face[i].c].x;
            c.y = Vertex[Face[i].c].y;
            c.z = Vertex[Face[i].c].z;

            dv = (TripleScalarProduct(a, b, c)) / 6.0f;
            vol += dv;

            d = ((a + b + c) / 4);
            dmom += (d * dv);
        }

        Volume = vol;
        Centroid = dmom / vol;
    }
```

请注意此函数定义一些区域变量*a*，*b*和*c*，这些变量的类型是Vector，用来表示四面体中由原点指向各面顶点的向量。Vector型态定义在附录一中。整数变量*i*只是一个计数器。浮点数变量dv和vol分别代表单一四面体的体积和目前物体的总体积。Vector型态的变量*d*和dom分别是单一四面体的坐标，以及目前所有四面体体积的一次矩总和。

在初始化所有区域变量后，函数会反复做完构成物体的平面阵列，为原点和目前平面Face[i]组成的四面体，建立向量*a*，*b*和*c*。接着，计算这三个向量的标量三重积，并且将结果除以6，此结果表示四面体的体积dv，接着加到目前的总体积vol。此函数接着计算四面体的形心*d*，并将它乘以四面体的体积dv，再将结果加到dmom（体积的一次矩）。依次计算完所有的面之后，总体积就是vol，而形心就是一次矩的总和除以总体积，dmom/vol。

这就是此类全部的内容。为了测试此类，以下是简单的应用程序用来说明Body3D——读入立方体的数据，并计算物体的体积特性。main函数如下：

```
int    main(int argc, char* argv[])
{
       Body3D       body = Body3D();
       float        volume = 0;
       int          i;
       Vector       centroid;

       // 读入物体的数据
       body.ReadData("cube.txt");

       // 将数据输出在屏幕上
       printf("Number of vertices = %d\n", body.nVertices);
       for(i=0; i<body.nVertices; i++)
              printf( "Vertex %d: x=%f y=%f z=%f\n",
                     i,
                     body.Vertex[i].x,
```

```
                body.Vertex[i].y,
                body.Vertex[i].z);

    printf("Number of faces = %d\n", body.nFaces);
    for(i=0; i<body.nFaces; i++)
        printf( "Face %d: a=%d b=%d c=%d\n",
                i,
                body.Face[i].a,
                body.Face[i].b,
                body.Face[i].c);

        // 计算体积和形心
        body.CalculateProperties();

        // 将结果输出在屏幕上
        printf("\n");
        printf("Volume = %f\n", body.Volume);
        printf("\n");
        printf("Centroid:\n");
        printf("x=%f y=%f z=%f\n", body.Centroid.x, body.Centroid.y, body.Centroid.z);
        printf("\n");

        printf("Done.\n");

        return 0;
}
```

如果你编译并执行此应用程序，就可以看到立方体的体积是8.0立方单位，而形心在（0, 0, 1）上。

以下有个更有趣的测试 —— 利用此应用程序计算一个类似船壳的物体（也许并不是很像）体积。

图 8-6 显示这个船壳物体，而其相关的数据文档如下：

```
36
-5.500000 -0.693775 0.281525
-5.500000 -0.693775 2.000000
3.888562 0.000000 1.700000
3.888562 -0.100000 1.991344
-5.500000 0.693775 0.281525
3.888562 0.000000 1.700000
3.888562 0.100000 1.991344
-5.500000 0.693775 2.000000
1.000000 0.900000 0.105572
1.000000 0.950000 2.000000
1.000000 -0.950000 2.000000
1.000000 -0.900000 0.105572
1.500000 0.794000 0.219941
1.500000 0.900000 2.000000
1.500000 -0.900000 2.000000
1.500000 -0.793988 0.219941
```

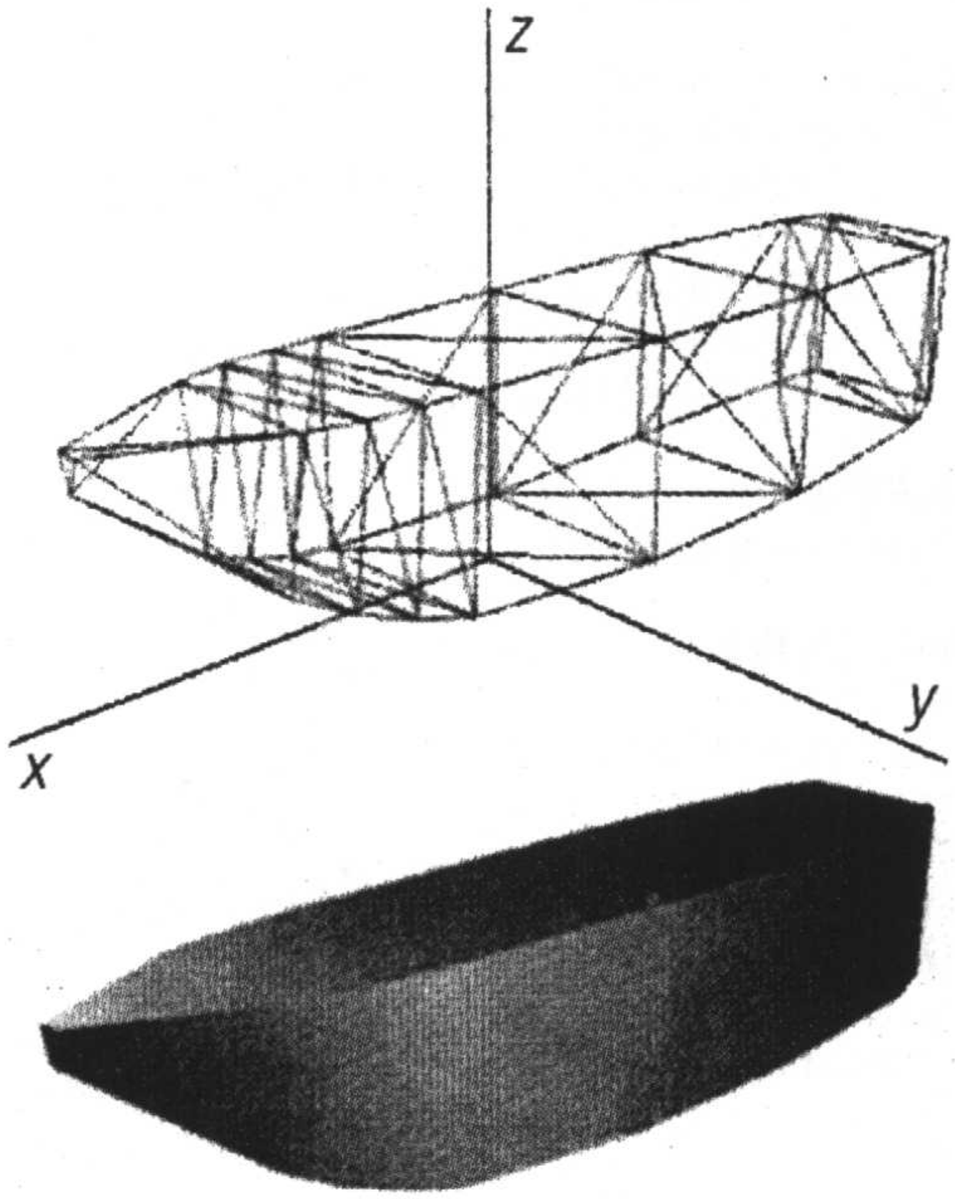



图 8-6：船壳

```
2.000000 0.680000 0.387096
2.000000 0.870000 2.000000
2.000000 -0.870381 2.000000
2.000000 -0.680000 0.387096
2.500000 0.538000 0.633431
2.500000 0.747000 2.000000
2.500000 -0.747214 2.000000
2.500000 -0.538416 0.633431
-1.000000 -1.000000 0.000000
-1.000000 -1.000000 2.000000
-1.000000 1.000000 2.000000
-1.000000 1.000000 0.000000
-3.000000 -1.000000 0.070381
-3.000000 -1.000000 2.000000
-3.000000 1.000000 2.000000
-3.000000 1.000000 0.070381
-5.000000 -0.864029 0.211143
-5.000000 -0.864029 2.000000
-5.000000 0.864029 2.000000
-5.000000 0.864029 0.211143
68

2 3 22
2 22 23
```

```
20 5 23
5 2 23
6 3 2
6 2 5
6 21 3
3 21 22
6 5 20
6 20 21
1 7 4
1 4 0
34 35 4
34 4 7
34 7 1
34 1 33
32 33 1
32 1 0
4 35 0
0 35 32
13 12 8
13 8 9
13 9 14
14 9 10
15 14 10
15 10 11
8 12 15
8 15 11
17 16 12
17 12 13
17 13 18
18 13 14
19 18 14
19 14 15
12 16 19
12 19 15
21 20 16
21 16 17
21 17 22
22 17 18
23 22 18
23 18 19
16 20 23
16 23 19
11 10 25
11 25 24
9 26 10
10 26 25
9 8 27
9 27 26
27 8 11
27 11 24
24 25 29
24 29 28
26 30 29
26 29 25
```

```
26 27 31
26 31 30
31 27 24
31 24 28
28 29 33
28 33 32
30 34 33
30 33 29
30 31 35
30 35 34
35 31 32
32 31 28
```

这个测试的结果显示物体体积是28.67个立方单位，而其形心位于(−1.43, 0.00, 1.08)。

这些范例程序代码中并没有做任何错误的检查，这使示范的成员函数能简洁而易懂。当然，在正式的完成品中还是要加入错误检查的程序代码。所要做的事包括：检查输入资料以确保读入的物体是简单的三角多面体，以及防止在CalculateProperties函数中有任何除以零的错误。

# 阻力

在第六章中已经介绍过物体在流体中移动时会受到摩擦力和压力所产生的阻力。当船在水面上移动时也会受到这些阻力的影响；然而，在空气和水界面上，还有其他阻力分量需列入考虑。如果要将作用于船上的阻力分成三个主要分力，方程看起来就是：

$$R_{\text{total}} = R_{\text{friction}} + R_{\text{preassure}} + R_{\text{waves}}$$

接下来将逐一地介绍这些分力，并介绍一些由实验数据所推导出来的公式。可是在此之前要先阐明：以下公式终究是非常一般的，只有知道船体几何形状的小细节，才能正确地应用这些公式。所以在实际设计船时，只在较初期的阶段中才会使用这些公式来估计船的阻力。也就是说，这在粗略估计阶段是很实用的，在做参数研究时也可以用来观察主要参数改变引起的影响。

第一个阻力的分力是当船在水中行进时作用在水面下船壳表面的摩擦力。这和第六章所讨论的摩擦力是相同的。然而对于船来说，有一些比较方便的实验数据公式可以用来计算这些力：

$$R_{\text{f}} = (1/2)\rho V^2 S C_{\text{f}}$$

在这个方程中，$\rho$是水的密度，$V$是船的速度，$S$是船壳在水面下的表面积，而$C_{\text{f}}$是摩擦力的阻力系数。你可以用这个实验公式来计算$C_{\text{f}}$：

$$C_f = 0.075/(\log 10 R_n - 2)^2$$

这里的$R_n$是雷诺数（第六章中提过），是根据船壳的长度而定的。这个公式在1957年被国际拖船协会（ITTC）所采用并且广泛应用于船体结构的领域，用来估计船舰的摩擦力阻力系数。

要想应用这个公式来计算$R_f$，还要知道船壳在水面下的表面积$S$。可用数值积分法直接求出这个表面积，就像计算体积一样。不然可用另一个公式：

$$S = C_{ws}\sqrt{\nabla L}$$

在这个方程中，$\nabla$是排水量的体积，$L$是船的长度。而$C_{ws}$是湿表面系数；此系数是船体外形的函数——船幅与吃水深度的比例。对排水型船舰的船壳外形而言，此系数的范围约在2.6～2.9之间。

船体上所受到的压差阻力与第六章提及的抛体上所受的阻力相同。请记住，这个阻力是由黏滞效应（会在船的后方产生相对的低压力区）所致。因为船体上有许多不规则的几何形状，因此我们难以为此压差阻力定量。计算流体动力学的演算法可以用来估计此力，但是这需要船壳几何形状的详细信息和许多费时的计算。另一个选择是用缩小模型的测试资料，来推断原尺寸的船舰所受的阻力。

如同压差阻力一样，水波阻力也很难去计算，所以也必须依赖缩小模型的测试。水波阻力起因于船到流体之间的能量转移或动量转移。换句话说，水波阻力就是船在周围流体产生水波所做的功的函数。我们可以从一些现象观察水波阻力的影响：在船首会因船前进而产生波浪，而在船尾也会产生波浪。波浪会影响船四周的压力分布，并且会影响压差阻力。所以要个别分析压差阻力和水波阻力的影响并不容易。

当进行缩小模型的测试时，压差阻力和水波阻力通常组成所谓的残余阻力（residual resistance）。类似于摩擦力的系数，也可以定义残余阻力的系数，如下：

$$R_r = R_{pressure} + R_{wave} = (1/2)\rho V^2 S C_r$$

这里的$R_r$是总残余阻力，而$C_r$是残余阻力系数。

有许多可用的阻力估计法可以用来估计船的残余阻力系数；然而它们通常是为特定的船舰类型而定的。例如某种方法的公式是用来计算驱逐舰的$C_r$值，而另一种方法可能会有用来计算大型油轮$C_r$值的公式。所以要为所分析的船舰类型选择适当的方法（注2）。一

注2：　这些方法大多都很复杂，也偏离本书的主题。所以本书仅在后面列出相关的参考文献。

般来说，$C_r$ 值会随着船速和排水量的增加而增加。排水型船壳的 $C_r$ 值的范围约在 $1.0e^{-3} \sim 3.0e^{-3}$ 之间。

虽然这三种阻力 —— 摩擦力、压差和水波阻力 —— 对于排水型船体是非常重要的，但并不是只有这些阻力而已。因为当船行驶在水与空气表面时，船体有很大的部分是在水面上，暴露在空气中的。也就是说船也会受到空气的阻力。可用以下公式计算空气的阻力：

$$R_{air} = (1/2)\rho V^2 A_p C_{air}$$

这里的 $C_{air}$ 是空气阻力系数，$\rho$ 是空气密度，$V$ 是船的速度，而 $A_p$ 是船横切面的投影面积。根据船的种类而不同，$C_{air}$ 通常在 0.6～1.1 之间。油轮和大型货轮可能会达到范围的最大值，而战舰可能会达到此范围的最小值。如果知道船的横切面投影面积，可用以下公式计算空气阻力：

$$A_p = B^2/2$$

这里的 $B$ 是船幅（船宽）。

船所受到的阻力也和其他因素有关，如船龄、海相、和船的用途等。举例来说，如果船在海上航行一段时间而没有清理它的外壳，在外壳会产生一层海洋滋生物并增加船的摩擦力。当船在浅海或狭窄的海峡行驶时，船的阻力也会增加。因为狭窄的水流会使船的吃水深度增加。如果海相不佳时，大风大浪会使船体承受更大的阻力。如果船在水面下的船壳上有许多突出物的话，会比没有这些突出物时增加10%～15%的阻力。以上这些因素会因情况的不同而改变，所以要视实际情况而定。

# 虚质量

虚质量（virtual mass）的概念在实时模拟器计算船的加速度时是非常重要的。虚质量等于船的质量加上和船一起加速前进的水的质量。

回到第六章，曾经提过黏滞边界层，也说明靠近移动物体表面的流体粒子其相对速度（相对于移动物体）是0，而且离物体表面越远自由水流的速度就越快。实际上，物体表面的流体会随着黏附在物体上随之移动和加速。因为在不同边界层的流体速度和加速度是不同的，所以附加的质量（也就是一同加速的水之质量）是所有被物体加速所影响的流体总质量的加权积分。

一艘船的黏滞边界层可能很薄，也可能长达船后数英尺；在这种情况下，随着船一起加速的水的质量就很可观了。所以在对船的加速度作任何分析时，也要一并考虑附加质量。

虽然附加质量的计算方式已经超出本书范围；但还是有一点要指出——附加质量是一个张量，也就是说它会根据加速度的方向而改变。此外，附加质量也可应用于线性运动和角运动。

附加质量通常以附加质量系数表示，此系数的量值等于附加质量除以船的质量。计算附加质量的方法已经超出本书范围。有些方法会对于整个船壳做积分，而有的方法则以类似船壳的椭圆体来估计附加质量。

使用估计法时，椭圆体的长度等于船的长度，而它的宽度等于船幅。当船纵向行进时，也就是以平行于船长轴线的方向行进时，附加质量系数几乎直线地由0（当船的宽长比近乎0时，这时的船体非常的窄）到1/2（当船的宽长比为1时，船体是个球型）。

当附加质量系数以船质量的百分比表示时，虚质量可以用以下公式计算：$m_v = m(1 + x_a)$。这里的$m$是质量，而$x_a$是附加质量系数，用0.2表示20%。对典型的排水型船舰的比例来说，纵向附加质量的范围约是船质量的4%～15%。保守的估计通常用20%。

# 第九章

# 气垫船

气垫船（或称为气垫式运输工具）最近已经成为一两个电脑游戏中的主角。具有未来感的外观、高速度，以及无往不利的漂浮能力，使它们具有极大的吸引力。如果能再加上一对大枪，并且放在充满坏蛋的环境中，就会让人仿佛处在刺激的射击游戏中。在真实世界里，约在1950年左右时，气垫船开始被赋予不同的任务：军事作战、搜索救援、运送货物、渡船和娱乐。它们有不同的外型和尺寸，不过运作原理大多是相同的，简单地说就是让船身在地面或海面上漂浮以减小摩擦力。本章将介绍气垫船运作的基本原理，以及当在游戏中模拟气垫船的运作时需考虑的作用力。

## 运作原理

当我在达信海洋装备公司（Textron Marine Systems）当基础造船设计师时，曾经很幸运地参与几艘气垫船的设计工作（注1）。虽然某些建造的气垫船有着非常复杂的系统，但由于军事上的需要，这些气垫船运作的基本原理仍然十分简单。

初期气垫船的设计靠围绕船体四周的环状喷嘴来喷出空气（如图9-1所示）。这些船体下的喷嘴所需的空气由一个大型风扇来提供。空气的喷射流在船体下制造了一个相对气压非常高的区域，并产生了净升力。如果想要让气垫船能上升，这个升力至少要等于气垫船的重量。这种升力被称为空气静升力（aerostatic lift）。盘旋高度受限于气垫船所能提

注1：　达信公司位于路易西安那州的新奥尔良。我在那里工作时，曾经参与几艘气垫船的建造工作。其中一艘是美国海军的气垫登陆艇（LCAC），陆战队用它来进行两栖作战。

供的动力，以及风扇由喷嘴喷出足够空气的能力，也就是说，盘旋高度越高所需的动力越大。

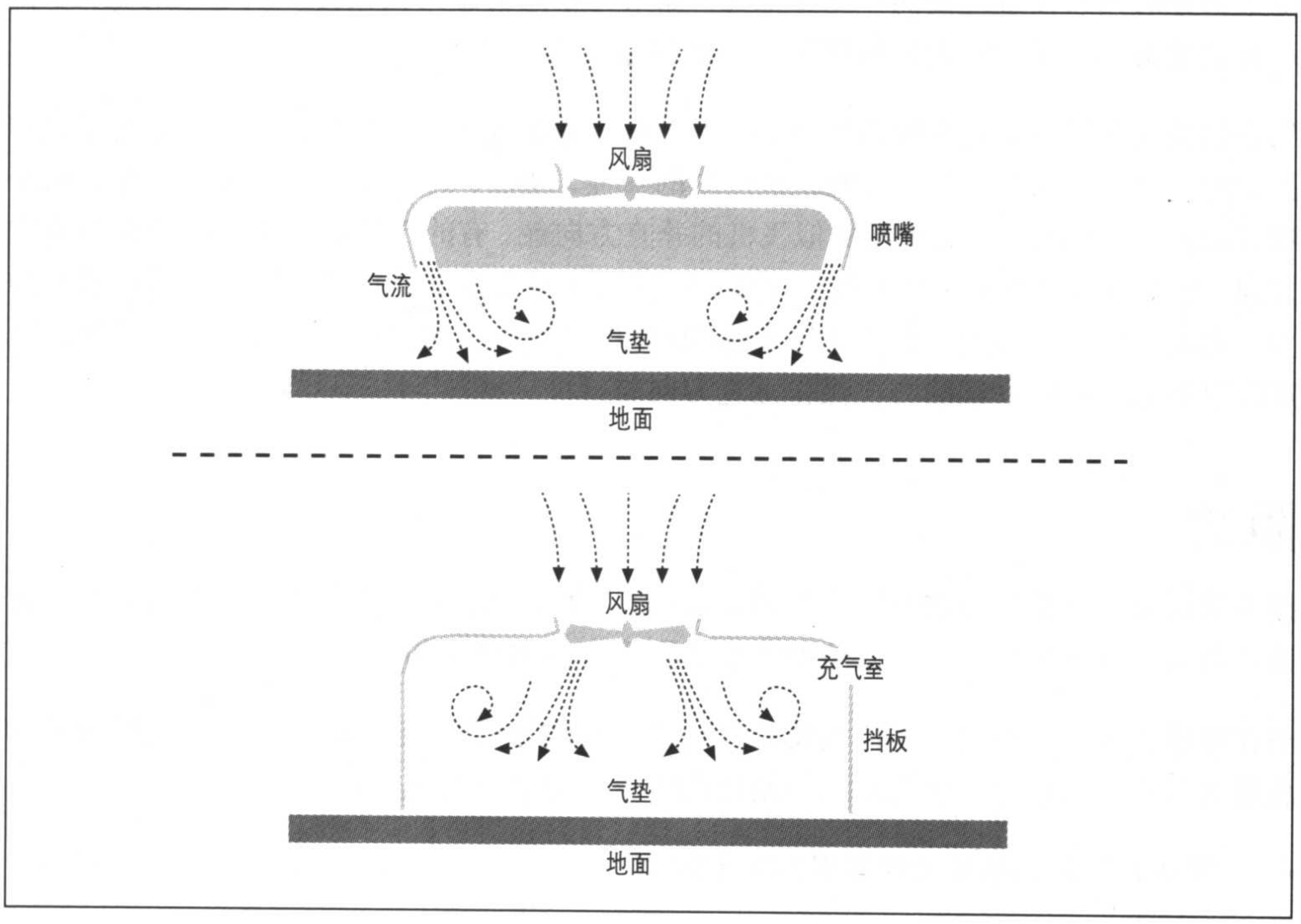



图 9-1：气垫船的结构

这个方法很不切实际，因为盘旋高度非常有限，并且使船体坚硬的构造与地面的空隙很小，以至于遇到小障碍物也不能越过。为了解决这样的问题，我们在船体四周装设柔韧的“挡板”，这个挡板包覆气垫形成充气室（plenum chamber）。尽管挡板底部与地面之间的空隙仍然很小，但是这个方法明显地加大了船体坚硬结构与地面间的空间。虽然挡板有多种不同的设计，但这仍是大部分气垫船运转的基本结构。有些挡板的设计像是简单的帘子，而有的使用纵向并列的精密加压袋。这样一来，装设挡板的气垫船能清除相当大的障碍物而不至于伤害船的硬件结构。而且在船前进时，挡板会顺应地面的形状而扭曲。

真正对于空气静升力的计算十分复杂，因为气垫内的压力分布并不是一致的，而且也要考虑风扇系统的性能。虽然现在已经有计算环状喷射和充气室的理论，但是这些已经超出了本书的范围。此外对游戏中的模拟来说，比较重要的是：要维持气垫船在盘旋飞行中的平衡，必须要使升力等于气垫船的重量。

在理想状况下，气垫船在航行中若能减少与地面（或水面）的接触，就能加快行进速度，因为能减少接触的摩擦阻力。请注意这是理想状况下。现实中气垫船常常会俯仰和滚转，使某部分的挡板接触地面而产生摩擦力，如果有障碍物碰到挡板会产生更大的阻力。在一般的情况下，减少对地面的接触对于航行速度是有利的，可是对于操控性就不利了。

气垫船为人所诟病的是它的操控性。因为气垫船是在地面上滑行的，当你想要改变它的方向时，它仍然会有沿着原先的轨道前进的惯性。现在，有许多方法想要以改变结构来增强操控性。有的气垫船使用类似飞机的垂直方向舵，有的气垫船会直接改变推进力的方向。这些方法都能简单地在程序中模拟，因为这些方法都是提供非作用在船体重心的力，进而产生转向力矩。在第十二章的2D模拟程序中，将介绍如何操纵侧推进器，也可以以第七章中所介绍的方式操作垂直方向舵。

# 阻力

接下来讨论当气垫船行进时所受到的阻力。我们将地面上的阻力与水面上的阻力分开讨论，因为在这两种状况下，气垫船所受的阻力有显著的不同。

当在平滑的地面上行进时（假设忽略气垫的挡板和障碍物撞击的阻力），气垫船所受的总阻力只有空气阻力。空气动阻力是由以下三种分力所构成的：

- 气垫船外表的摩擦力和黏滞的压差阻力
- 当船体俯仰时产生的诱导阻力
- 动量阻力

以方程表示总阻力，如下：

$$R_{\text{total}} = R_{\text{viscous}} + R_{\text{induced}} + R_{\text{momentum}}$$

第一种所谈到的阻力是船体上的黏滞阻力，与第六章所讲的抛体在空气中飞行所受的阻力相同。这个阻力可以用以下这个熟悉的方程来计算：

$$R_{\text{viscous}} = (1/2)\rho V^2 S_{\text{p}} C_{\text{d}}$$

这里的$\rho$是指空气的质量密度。$V$是气垫船的速度，$S_{\text{p}}$是气垫船正面投影的面积，通常与$V$的方向垂直，而$C_{\text{d}}$是阻力系数。如今气垫船的阻力系数$C_{\text{d}}$值通常位于0.25～0.4之间。

第二种阻力分量称为诱导阻力（induced drag），是由气垫船在移动时产生俯仰角所导致

的。当船头上仰$\tau$度时，会产生一个空气静升力向量的分量，作用在速度$V$的反方向。此分量约等于船重乘以俯仰角的正切值：

$$R_{\text{induced}} \approx W\,(\tan\,\tau)$$

最后，动量阻力（momentum drag）是由空气相对于船体的水平动量，在进入升力风扇的进气口时被破坏而产生的。要计算这个分力相当困难，除非能知道整个升力系统的特性，例如空气进入风扇的质量流率（mass flow rate）。若已知质量流率，则$R_{\text{momentum}}$等于质量流率乘以船体的速度：

$$R_{\text{momentum}} = (\mathrm{d}m_{\text{fan}}/\mathrm{d}t)V$$

质量流率的单位是slugs/s，而速度的单位为ft/s，相乘后得到以lb为单位的阻力。

阻力除了由这三种分力所构成外，气垫船在水面上行驶时还会受到其他的阻力——水波阻力和湿阻力。在水面行驶的情况下，阻力方程被改写成以下形式：

$$R_{\text{total}} = R_{\text{viscous}} + R_{\text{induced}} + R_{\text{momentum}} + R_{\text{wave}} + R_{\text{wetted}}$$

当气垫船在水面行驶时，会因为充气室的压力而使水面下降（如图9-2所示）。在静止或低速行驶时，这些被排开的水重等于气垫船的重量，就像船体被浮力支撑而浮在水面上一样。当气垫船向前移动时，船头会向上仰。这个时候，在被压缩区域的水面大约与船底平行。当船速增加时，水被推挤开的情况会较舒缓，而俯仰角也会减小。

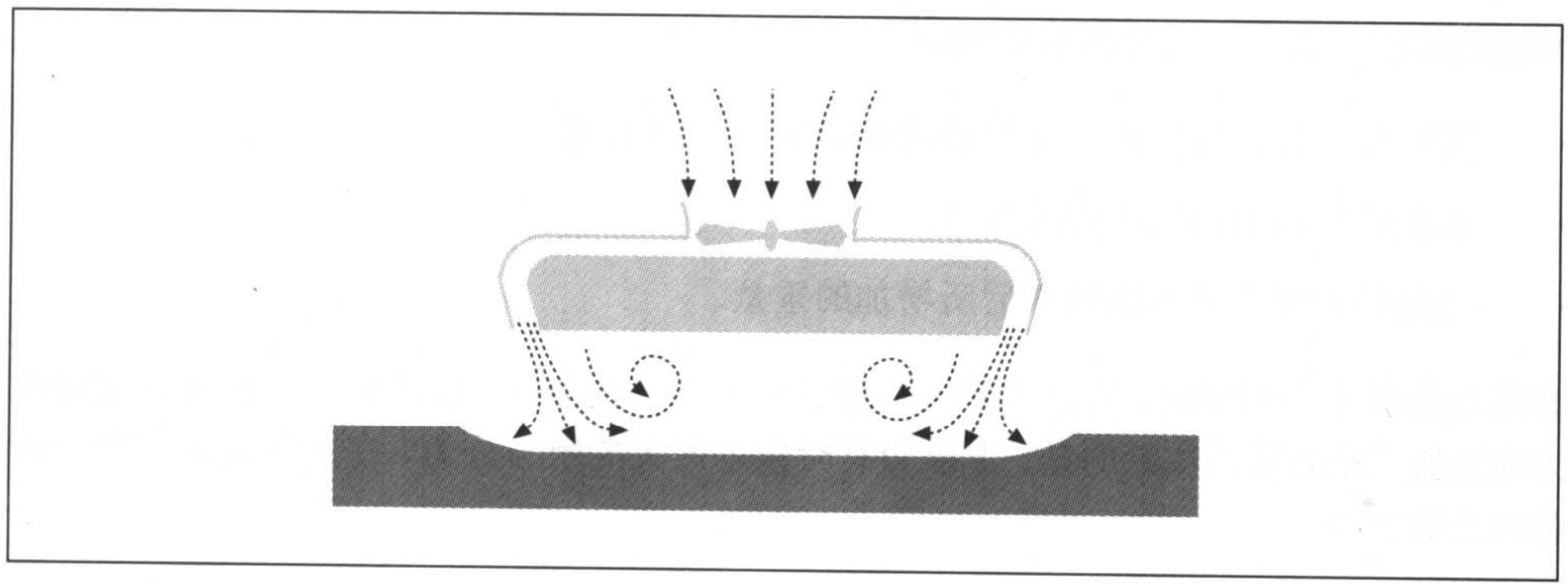


图9-2：水面上的气垫船

水波阻力是由这种水被推挤的情况所产生的，它的大小等于压缩区域水面上压力的水平分力。结果是当船以低速与小俯仰角前进时，水波阻力约等于诱导阻力的量值：

$$R_{\text{wave}} \approx W\,(\tan\,\tau)$$

因为水波阻力与水被压缩区域的大小成比例，它在低速时会最大，并且在加速时会逐渐减小。当你绘制气垫船的水波阻力（速度的函数）曲线图时，会发现它并不是直线也不是抛物线，而是如图 9-3 中所示，在低速范围会有隆起的曲线。

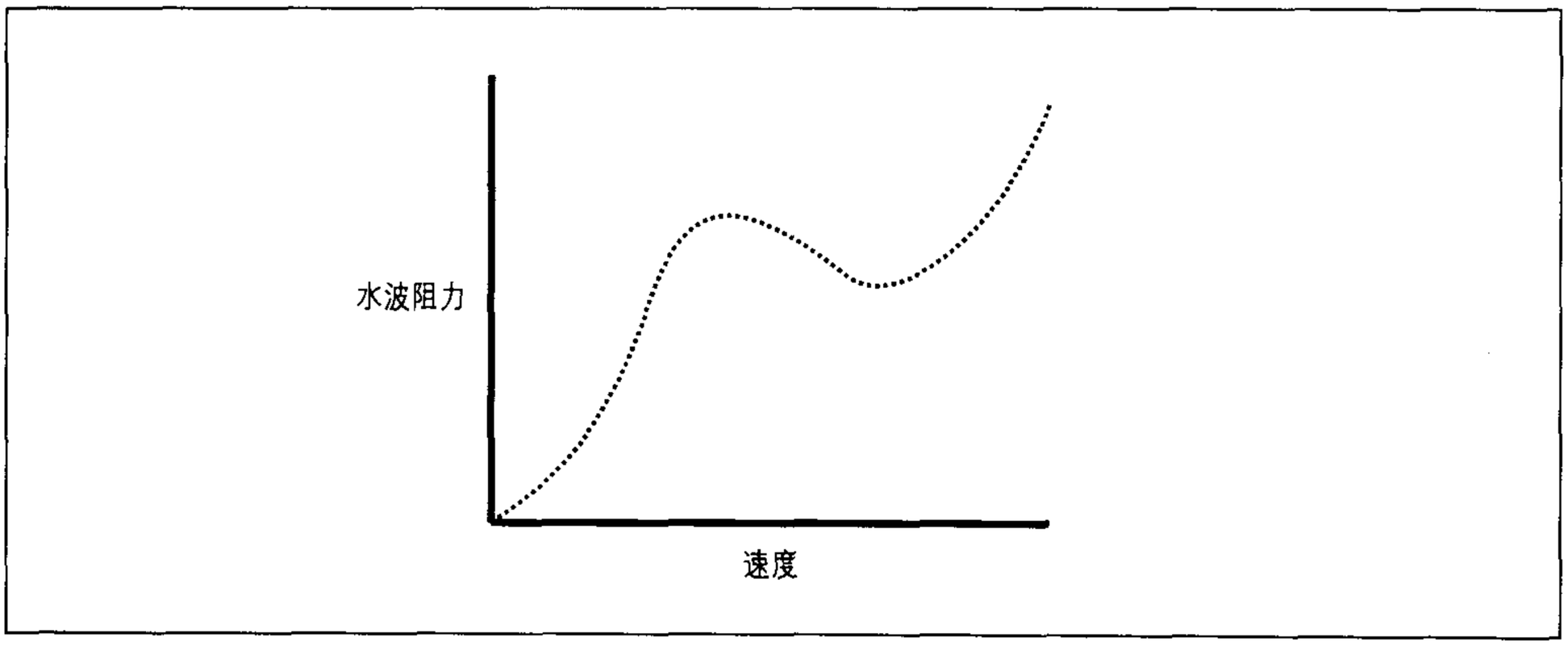



图 9-3：水波阻力

在文献中有几种理论可以计算水波阻力，并预测在什么速度下会产生隆起及隆起的大小。这些理论指出，隆起是根据气垫船的平面几何形状所决定的，通常会发生在速度为 $\sqrt{gL/2}$ 到 $\sqrt{gL}$ 的范围内。这里的 g 是重力加速度，而 $L$ 是气垫的长度。实际上，特定气垫船的水波阻力的特征，最好是由缩小模型的测试来决定。

所谓的湿阻力是以下几种因素的函数：

- 气垫船在行进时，船壳和气垫档板会碰到水面的部分。
- 水花碰到船壳和档板的冲击力。
- 气垫船弄湿或者是碰到水后所增加的重量。

湿阻力是很难去预测的，所以实际上在设计特殊的气垫船时，会用模型测试来决定湿阻力的量值。然而值得注意的是，湿阻力有时候是最大的阻力分力，在某些状况下会达到总阻力的 30%。

# 第十章

# 汽车运动

本章将讨论汽车运动所涉及的物理概念。如同前四章一样，本章会借助例子来解释特定的物理现象，对于那些你会在游戏中模拟的汽车运动，我也会解释相关的运作机制；当然，我所说的“机制”，并不是指汽车的内燃机运作原理或传动系统如何将动力传输到车轮上，这些机械原理离本书主题太远，本章所要探讨的是“刚体运动”，也就是说，我们将整部汽车视为一个“刚体”，然后将焦点集中在外力对刚体的作用。然而，我将会讨论作用到传动轮上的扭力（力矩）是如何转换成推动汽车前进的推力的。

## 阻力

当汽车在路上行驶时，主要有两种力会拖慢它的速度，第一种是空气动阻力(aerodynamic drag)，第二种是滚动阻力（rolling resistance)。汽车承受到的总阻力是这两种分量的合力：

$$R_{\text{total}} = R_{\text{air}} + R_{\text{rolling}}$$

空气阻力主要来自汽车外表与空气的摩擦，以及压差阻力 —— 类似抛射体（第六章）与飞机、船和气垫船（第七章至第九章）的状况。因此，可使用类似形式的阻力公式：

$$R_{\text{air}} = (1/2)\rho V^2 S_{\text{p}} C_{\text{d}}$$

这里的$\rho$是气体密度，$V$是汽车的速率，$S_{\text{p}}$是汽车的受风面积（垂直于行进方向的车面的投影面积)，而$C_{\text{d}}$是阻力系数。阻力系数随车种而异，对跑车而言，其阻力系数大约介于0.29～0.4之间，小卡车为0.43～0.5，联结车约在0.6～0.9，而一般的轿车大约在

0.4～0.5之间。阻力系数是汽车外形的函数，也就是箱形或流线形的程度。流线形车体的阻力系数比较低，以雪佛兰的Corvette为例，其阻力系数相当低，只有0.29而已，而一般没有整流罩的联结车其阻力系数高达0.9。在游戏中，可以用不同的阻力系数来调节不同车种的运动模式。

当轮胎在道路上滚动时，它会受到“滚动阻力”的影响，这种阻力也会逐渐拖慢汽车的运动。滚动阻力不是来自摩擦，而是由轮胎滚动时的形变所造成的。我们很难构思出一套完善的理论来精确计算滚动阻力的值，因为它是具有许多复杂因素的函数，例如轮胎和道路的形变、轮胎接触面积上的压力、轮胎和道路材质的伸缩弹性、轮胎和道路表面的粗糙程度、胎压（轮胎内部的气压）等。因此，你只能依赖靠经验推导出来的公式。我们所用的公式如下：

$$R_{\text{rolling}} = C_{\text{r}} w$$

此公式可推导出各个轮胎的滚动阻力。其中的$w$是指轮胎所承受的重量，而$C_{\text{r}}$是“滚动阻力系数”（coefficient of rolling resistance），$C_{\text{r}}$事实上只是“滚动阻力”和“轮胎所承受的重量”的比例而已。还好，轮胎制造商通常会提供他们的轮胎在特定条件下的滚动阻力系数。一般轮胎的$C_{\text{r}}$值约为0.015，而卡车轮胎的$C_{\text{r}}$值约在0.006～0.01之间。假设一辆汽车有4个完全相同的轮胎，那么，只要将汽车的总重量代入上述公式中的$w$，就能算出该汽车的总滚动阻力。

## 功率

知道了如何计算汽车所承受的总阻力，就能轻易算出在特定速度下要抵消这些阻力所需的“功率”（power）。功率是“力”或“扭力”在一段时间内所做的功（work）之量。力所做的“机械功”（mechanical work），等于“力所造成的物体位移距离”与“力”的乘积，功的英制单位是ft-lb；而功率是单位时间内所做的功，所以其计量单位是ft-lb/s。在描述汽车引擎的输出功率时，我们通常使用“马力”（horsepower）为功率的单位，1马力等于550 ft-lb/s。

要克服某速度下的总阻力，所需的功率（马力）如下：

$$P = (R_{\text{total}} V)/550$$

在此，$P$是功率（以马力为单位），而$R_{\text{total}}$是车速到达$V$时所受的总阻力。请注意，此公式的$R_{\text{total}}$必须以lb为单位，而$V$的单位是ft/s。

请注意，这不是让汽车速度达到$V$所需的引擎输出功率，而是要使速度达到$V$时，需要

传到驱动轮的功率。实际上，引擎输出的功率必须高于理论值，因为当传动系统将引擎输出功率传递到驱动轴、再传递到轮胎时，必定有一些机械功的损失。功率会以“扭力”（torque）的形式抵达轮胎，也就是说，对于半径为 $r$ 的轮胎，扭力 $T_w$ 所产生的前进力为 $F_w$：

$$F_w = T_w / r$$

这里的 $F_w$ 是轮胎推动汽车前进所发出的力量，而 $T_w$ 是施加到轮胎上的扭力，$r$ 是轮胎的半径。另一个需要引擎输出较高功率的原因是汽车里的其他系统也需要引擎输出的功率，例如，充电电池与空调压缩机都需要靠引擎运转来支撑。

## 刹车距离

在正常情况下，“刹车距离”（stopping distance）是一个由“刹车系统”和“驾驶者施加于刹车系统的力量”（踩刹车的力量）的函数，驾驶者越用力踩刹车，刹车距离就越短；当然，如果刹车过猛，导致轮胎开始打滑，那就是另外一回事了。当轮胎打滑时，轮胎和路面之间的摩擦力，以及道路本身的坡度，也会影响到刹车距离。如果汽车正在上坡，因为地心引力（重力）有助于减慢汽车的速度，所以滑行距离会比较短；而汽车正在下坡时，重力对汽车施加的额外加速度，会拉长汽车的刹车滑行距离。

计算滑行距离的公式，必须将这些因素都考虑进去：

$$d_s = V^2 / [2g(\mu \cos\varphi + \sin\varphi)]$$

这里的 $d_s$ 是滑行距离，g 是重力加速度，$\mu$ 是轮胎和道路之间的摩擦系数，$V$ 是汽车的初速度，而 $\varphi$ 是路面坡度（正角度代表上坡，负角度代表下坡）。请注意，此公式并没有将有助于刹车的“空气阻力”也考虑在内，因为空气阻力的影响通常很小，小到几乎可以忽略。

摩擦系数因轮胎和路面的状况而异，对于橡胶轮胎和平整的路面，动摩擦系数通常在0.4左右，而静摩擦系数大约在 0.55 左右。

比如在一个实时模拟系统中，若要计算轮胎和路面之间的实际摩擦力，你可以使用第四章曾介绍过的公式：

$$F_f = \mu W$$

$F_f$ 是各个轮胎在没有滚动时所受的摩擦力，而 $W$ 是各个轮胎所承受的重量。如果假设所有轮胎都完全相同，你可以用汽车的总重量来代入公式中的 $W$，进而算出所有轮胎所受的总摩擦力。

# 道路边坡

当你转动汽车方向盘时，施加于前轮的“侧力”（side force）会导致汽车开始转向。以“欧拉角”（Euler angle）的术语来说，称此为偏转角（yaw），不过，当我们在讨论汽车转向时，很少用欧拉角度来表示。当汽车转向时，即使汽车速度是固定的，但由于速度向量的方向改变了，也间接造成加速的效果。不要忘了，加速度是速度在单位时间内的变化率，而速度本身包含了大小和方向。

对于弯曲行驶中的车辆，必定有一定程度的“向心力”（或“离心力”）作用于该车辆。这种力所造成的效果，相当于轮胎和路面之间的“侧摩擦力”（side friction），或“道路边坡”（roadway banking）（译注 1）所造成的效应，或是这两者的综合效应。在驾驶一辆正在转向的汽车时，可以感受到一股明显的“向心加速度”（centrifugal acceleration），或一股源自回转中心的力量（离心力）。这种加速度其实是惯性所造成的效应，也就是说，驾驶者的躯体与车体倾向于维持原本的运动路径，而非真的有一股外力作用于汽车或驾驶人的身体。真正的力是向心力，少了此力，汽车将继续沿着直线前进，而不是沿着曲线转弯。

如果汽车转向过快，轮胎和地面之间的侧摩擦力就可能不足以支撑转向中的车辆，这就是为什么要在道路转弯处垫高外侧车道的原因；当汽车行经转弯处时，边坡有助于将汽车维持在原跑道，因为边坡的高度有助于抵消汽车转弯时的离心力（惯性），因为当汽车倾斜时，会向回旋中心的方向产生一股分力（如图 10-1 所示）。

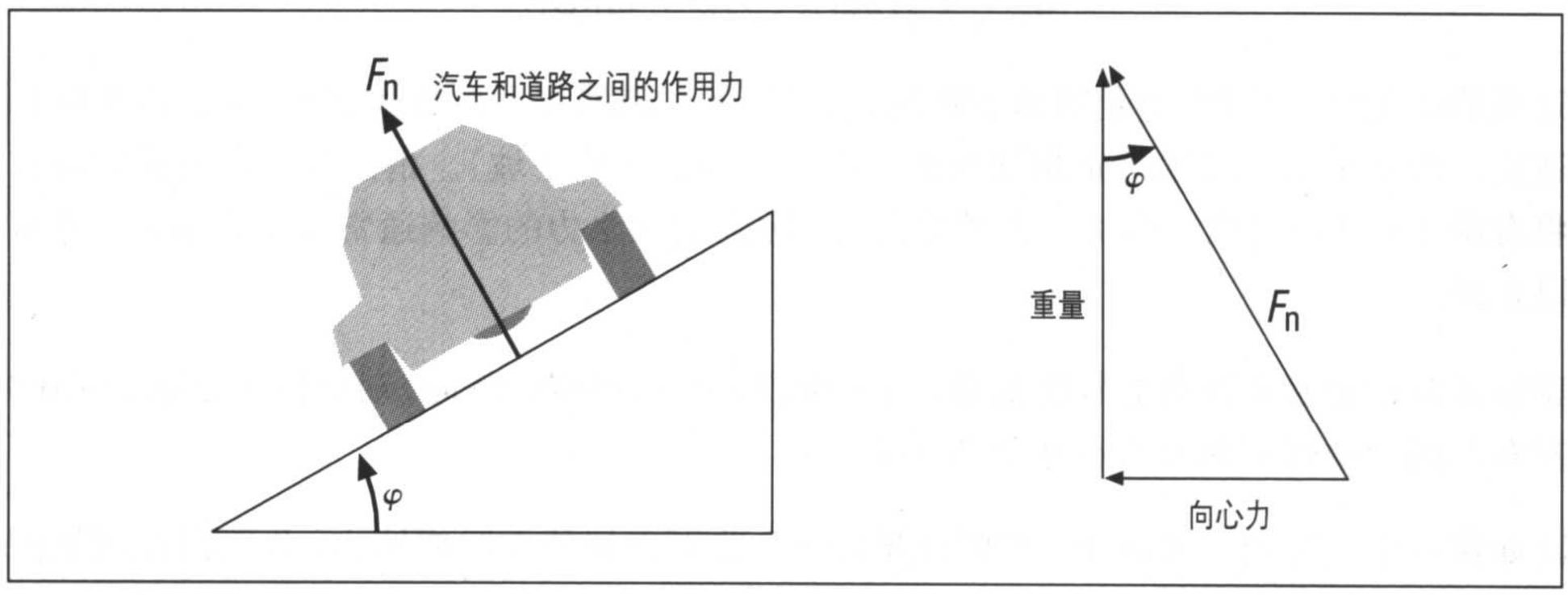



图 10-1：边坡效应

---

译注 1：在土木工程学中，道路边坡（roadway banking）又称为超高（superelevation），也就是在道路转弯处，外侧高于内侧的程度。

边坡垫高的角度、汽车的速度，以及轮胎和道路之间摩擦力，这三者的关系可用下列公式来表示：

$$\tan\varphi = V_t^2 /(gr) - \mu$$

这里的$\varphi$就是边坡垫高的角度（参考图10-1），$V_t$是汽车在转弯时的切线速度，g是重力加速度，$r$是转弯处的曲线路径的半径，$\mu$是轮胎和道路之间的摩擦系数。如果知道$\varphi$，$r$和$\mu$的值，就可以计算出汽车会开始打滑而被抛出路面的速度。

# 第十一章

# 实时模拟

接下来的几章开始针对“实时模拟”这个主题做彻底的介绍。这里之所以用“介绍”这个字眼，是因为这主题实在是太广泛且太复杂，就算用好几章的篇幅也不能涵盖所有必要的细节。另一方面，用“彻底”来形容介绍，是因为本章对实时模拟的演示要比理论探讨还要多。后面的章节将讨论 2D 与 3D 实时模拟的开发过程。

这几章的目标是提供足够的知识让你有能力顺利学习这个主题。因为你必须有扎实的基础知识，才能顺利地使用别人写好的物理引擎，或为自己设计专用的物理引擎。

本书中所谓的“实时模拟”（real-time simulation），是指一边计算物体的状态、一边呈现该物体的过程。你无法用事先编制好顺序的动画来呈现模拟对象，而必须在模拟过程中，利用物理模型、运动方程或微分方程来计算模拟对象的动作或状态。

这种模拟可用来塑造“刚体”（如 FlightSim 范例中的飞机）或“柔体”（如布料、人体外形）的运动模型。在模拟刚体运动的程序中，最基本的概念或许是使用数值积分技术去解运动方程。有鉴于此，我打算用一章的篇幅来解释数值积分技术。在后续的章节中，会利用本章的技术来开发 2D 与 3D 的模拟器。

在阅读本章内容之前，希望你花点时间复习第四章，尤其是那个用来解决运动学问题的通用步骤，因为本章将以此为基础。在前面的章节中，已经示范了如何估计物体的质量特性、如何推导运动方程，以及如何精确地模拟作用力和扭力。本章将示范如何解出运动方程，求得加速度、速度和位移。

# 运动方程的积分

你应该已经对动态粒子和动态刚体的运动方程有了整体认识，如果没有，请先回头复习第一章到第四章的内容。运动方程的用途之一，是在模拟过程中实际求出它们的解。我们已讨论过的运动方程，由于只涉及单纯的加速度、速度和位移而已，因此它们皆可被归类为“常微分方程”（ordinary differential equations，译注 1），在第二章与第四章，已详细说明这类微分方程的实际解法，但这并非我们的模拟所需。正如前几章所看到的，要计算你的系统的“力”和“力矩”，可能需要相当复杂的数据，而且这些数据有时只能靠经验值得来，这使你无法写出可以轻易算出积分值的数学函数。这表示你必须使用数值积分技术来估算运动方程的近似积分值。之所以用“近似”来形容，是因为数值积分法求出的解，不会是绝对精确的，会有一定程度的误差（误差程度依所用的方法而定）。

先用非正式的方法来解释如何使用数值积分法，因为它较容易理解。之后再用正式的数学式来解释。先看以下粒子（或刚体的质心）的线性运动方程：

$$F = m\ \mathrm{d}v/\mathrm{d}t$$

本书前几章已经提过这个简单的例子，将此方程重写成下面的形式，让它能被积分：

$$\mathrm{d}v/\mathrm{d}t = F/m$$
$$\mathrm{d}v = (F/m)\ \mathrm{d}t$$

这个方程可以这么解释：速度上无穷小的变化量 $\mathrm{d}v$ 等于 $(F/m)$ 乘以时间无穷小的变化量。先前的范例中，将方程的左边对速度求定积分，而右边是对时间求定积分。但是在数值积分中则必须取时间的有限间隔，因此无限小的 $dt$ 变成离散的时间增量 $\Delta t$，并得到离散的速度改变量 $\Delta v$ 为：

$$\Delta v = (F/m)\ \Delta t$$

请注意这个公式计算的不是瞬间速度，而是速度改变量的估计值。所以要估计粒子（或刚体）的实际速度，必须要知道时间变量 $\Delta t$ 之前的速度。也就是在模拟开始时（当时间为0），需先知道粒子的初速度，这被称为初始状态，当使用以下的方程计算粒子在不同时间的速度时，此初始状态是必需的（注 1）：

$$v_{t+\Delta t} = v_t + (F/m)\Delta t$$

---

译注 1：“常微分方程”（ordinary differential equations）：通常缩写成 ode，它们是不含任何“偏导数”（partial derivation）的微分方程。

注 1：在数学中，这种问题称为“初值问题”。

其中的初始状态是：

$$v_{t=0} = v_0$$

$v_t$是在时间$t$时的速度，$v_{t+\Delta t}$是$t$时加上某个时间间隔时的速度，$\Delta t$代表时间间隔，而$v_0$时间为0时的初速度。

这里可再次利用线性运动方程积分来估计粒子的位移（或位置）。当算出新的速度后，可用以下公式估计在时间$t + \Delta t$时的位移：

$$s_{t+\Delta t} = s_t + \Delta t(v_{t+\Delta t})$$

其中位移的初始状态是：

$$s_{t=0} = s_0$$

到目前为止所讨论的积分法被称为欧拉法，而这是一种最基本的积分方法。此方法虽然容易学习而且容易实现，但是它却不是一种精确的方法。

理所当然的，时间间隔越小时（也就是说$\Delta t$越接近$dt$），求出来的值越精确。然而，使用越小的时间间隔则计算越为困难。也就是说，使用更小的$\Delta t$会算出小数点以下更多位数的数字。因为计算方法不是太精确（为了不让数字太复杂，必须要舍去某些值而截掉数字的位数），最后会得到一个不精确的数字。这也就是说所选择的时间间隔不能小于某个极限值。不过很幸运，我们利用数值积分的某些技巧，能使用合理的时间间隔大小来求出较正确的值。

虽然刚才的范例中使用了粒子的线性运动方程，但是此积分技巧（以及稍后将介绍的技巧），同样可以应用于角运动方程。

## 欧拉法

在前一节中，对于欧拉法的解释是非正式的。要想以数学上更严谨的方式探讨欧拉法，先来看一个普通的函数$y(x)$的泰勒展开式（Taylor series expansion）。泰勒的理论可用来估计函数在某一点的值。这个估计值可以用以下的无穷多项式数列来表示：

$$y(x + \Delta x) = y(x) + (\Delta x)y'(x) + [(\Delta x)^2/2!]y''(x) + [(\Delta x)^3/3!]y'''(x) + \ldots$$

这里的$y$是$x$的函数，$(x + \Delta x)$是在估计$y$时$x$的新值，$y'$是$y$的一次导数，$y''$是$y$的二次导数，依此类推。

在上一节所讨论的运动方程中，所估计的函数是速度对于时间的函数。所以可用$v(t)$代替$y(x)$，而得出下列的泰勒展开式：

$$v(t + \Delta t) = v(t) + (\Delta t)v'(t) + [(\Delta t)^2/2!]v''(t) + [(\Delta t)^3/3!]v'''(t) + \ldots$$

请注意$v'(t)$与$dv/dt$相等，也等于上一节的运动方程中的$F/m$。也请注意在时间$t$时的速度$v$值也是已知的。由已知在时间$t$时的速度$v$和时间$t$时的导数，可求得在时间$t + \Delta t$的新速度$v$值。在第一次估计时，因为我们不知道$v$的第二阶、第三阶及以上的导数，所以舍去在$(\Delta t)v'(t)$之后的多项式数列，而得到：

$$v(t + \Delta t) = v(t) + (\Delta t)v'(t)$$

这就是在前一节所看到的欧拉积分方程。由于欧拉方程只包含一次导数，所以数列中有一部分值被舍去了，被舍去的部分称为截断误差（truncation error），又称为高阶项。而把它们舍去产生了第一阶近似值。此估计值的原理就是数列越长，各项的值越小而对估计值的影响也越小。因为$\Delta t$是一个很小的数目，所以$\Delta t^2$的值就更小了，依此类推。也因为$\Delta t$出现在分子，所以数列越高阶，值也就越小。在这个例子中，被舍去的部分$[(\Delta t)^2/2!]v''(t)$是截断误差，而我们称此为$(\Delta t)^2$误差。

用几何图形表示时可以看出，欧拉法在目前的时间间隔，以函数的导数值为斜率外推下一个时间间隔时的新值。这个方法如图 11-1 所示。

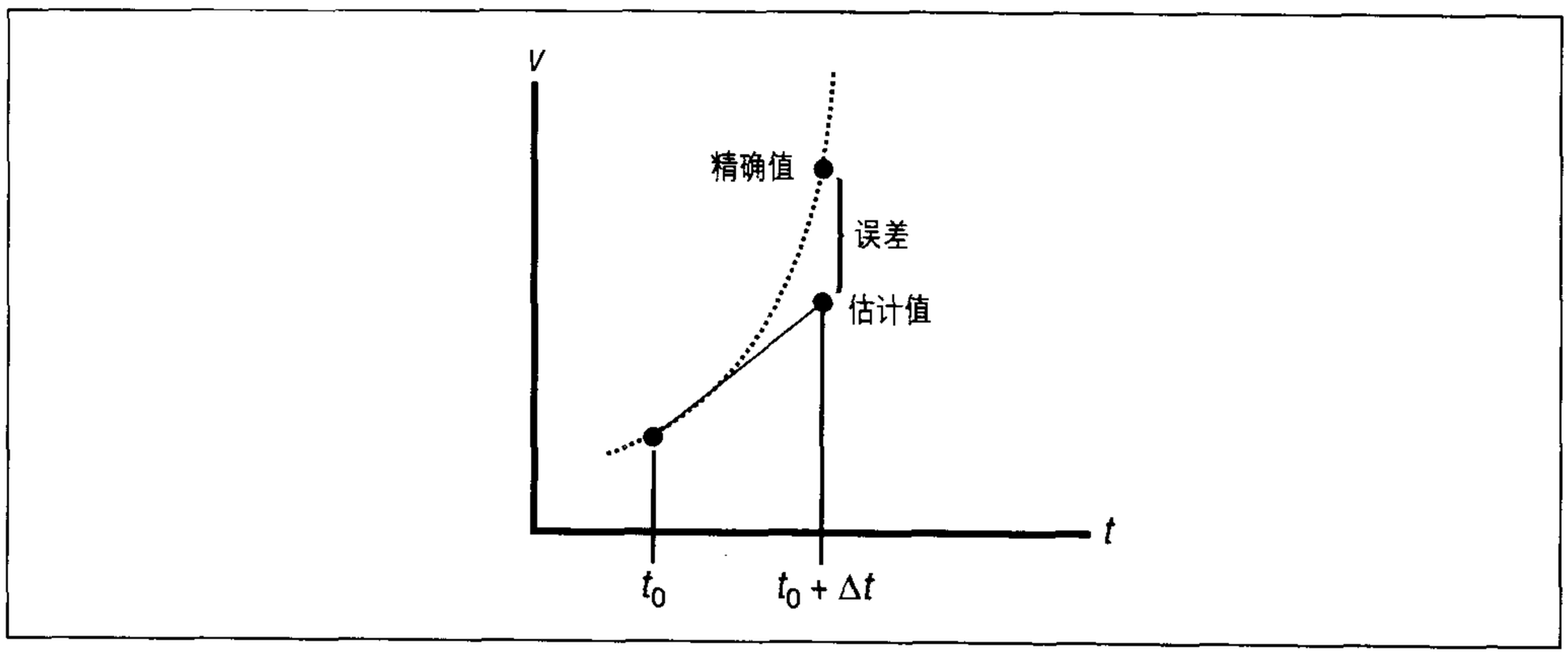



图 11-1：欧拉积分法

图11-1说明了截断误差是怎么产生的，并且展示了欧拉法在估计平滑函数时会产生多边形的近似值。更清楚地说，如果减小时间间隔的大小，就能够增加多边形边的数目，从而求出更精确的函数值。然而之前也提到，当模拟程序中计算的次数增加时，截断误差会累积得更快。

为了实际地说明欧拉法，这里再拿第四章中的范例（船舰的线性运动方程）来说明：

$$T - (Cv) = ma$$

这里的$T$是螺旋桨的推力，$C$是阻力系数，$v$是船的速度，$m$是船的质量，而$a$是它的加速度。

图11-2显示了欧拉积分法所求出的结果，与在第四章中船的速度在时间上的重叠比较。

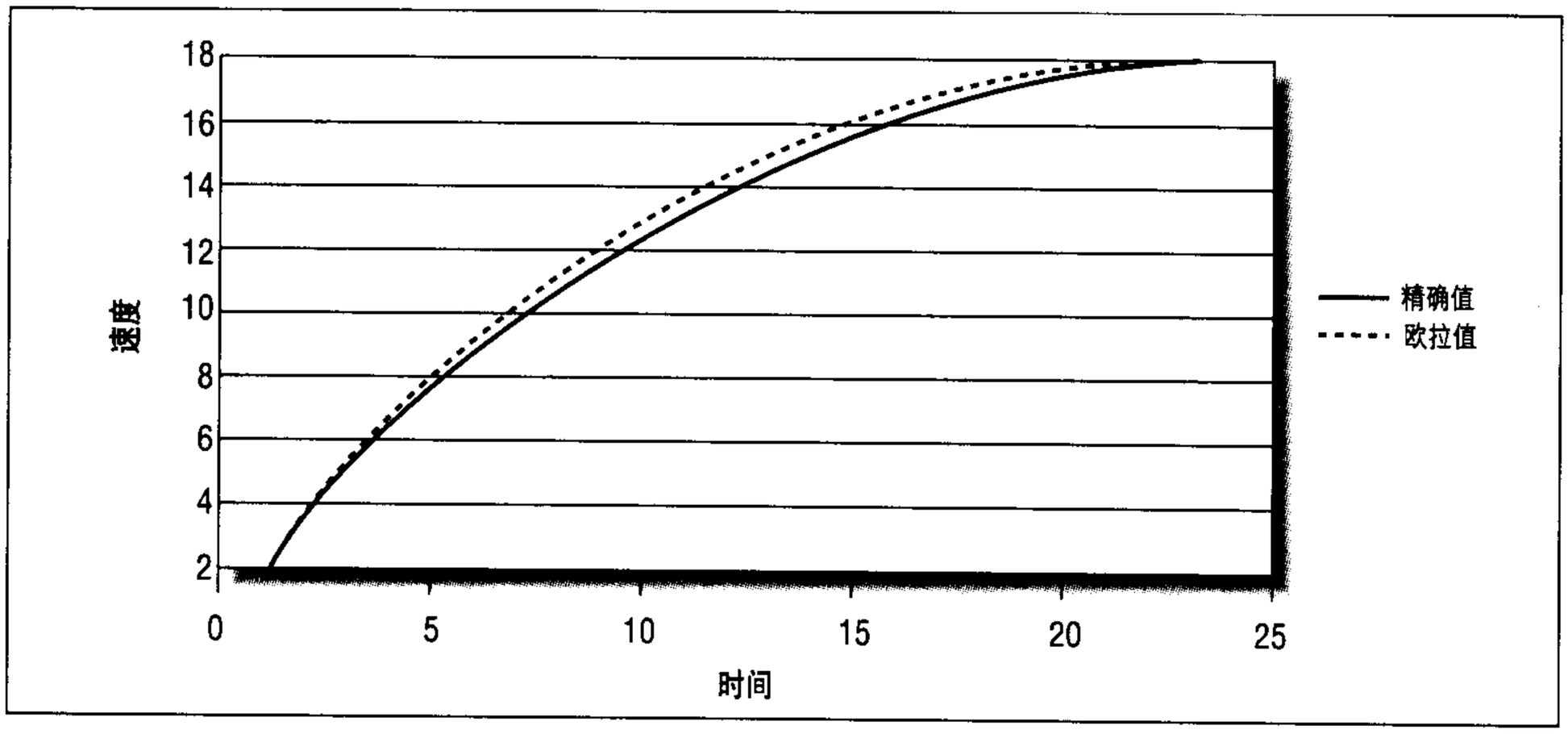



图11-2：欧拉积分法的比较

放大图11-2可以看出欧拉近似值的误差。如图11-3所示。

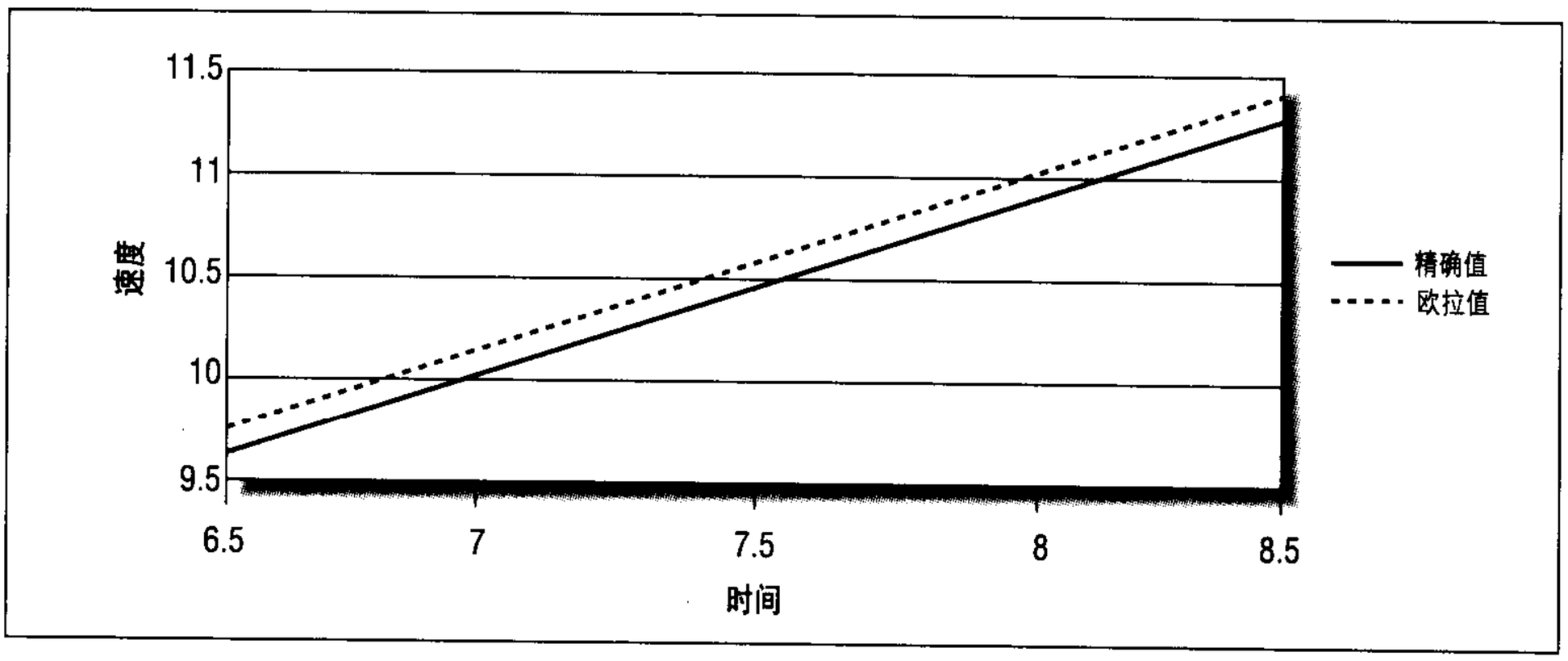



图11-3：欧拉值误差

表11-1指出了速度对时间在图11-3范围中的数值。从表11-1中也可以看到在精确值及欧拉估计值之间误差的百分比。

从这里可以看到，这个范例中的截断误差并不太严重，不过稍后所介绍的方法可以更精确。在那之前，你也可以发现在这个例子中欧拉法是稳定的 —— 在某个时间值之后误差就开始收敛了。我们将时间轴放大之后画在图 11-4 中。

表 11-1：精确值与欧拉值

| 时间（s） | 速度：精确值（ft/s） | 速度：欧拉值（ft/s） | 误差 |
|---|---|---|---|
| 6.5 | 9.559084 | 9.733158 | 1.82% |
| 7 | 10.06829 | 10.2465 | 1.77% |
| 7.5 | 10.55267 | 10.73418 | 1.72% |
| 8 | 11.01342 | 11.19747 | 1.67% |
| 8.5 | 11.4517 | 11.63759 | 1.62% |

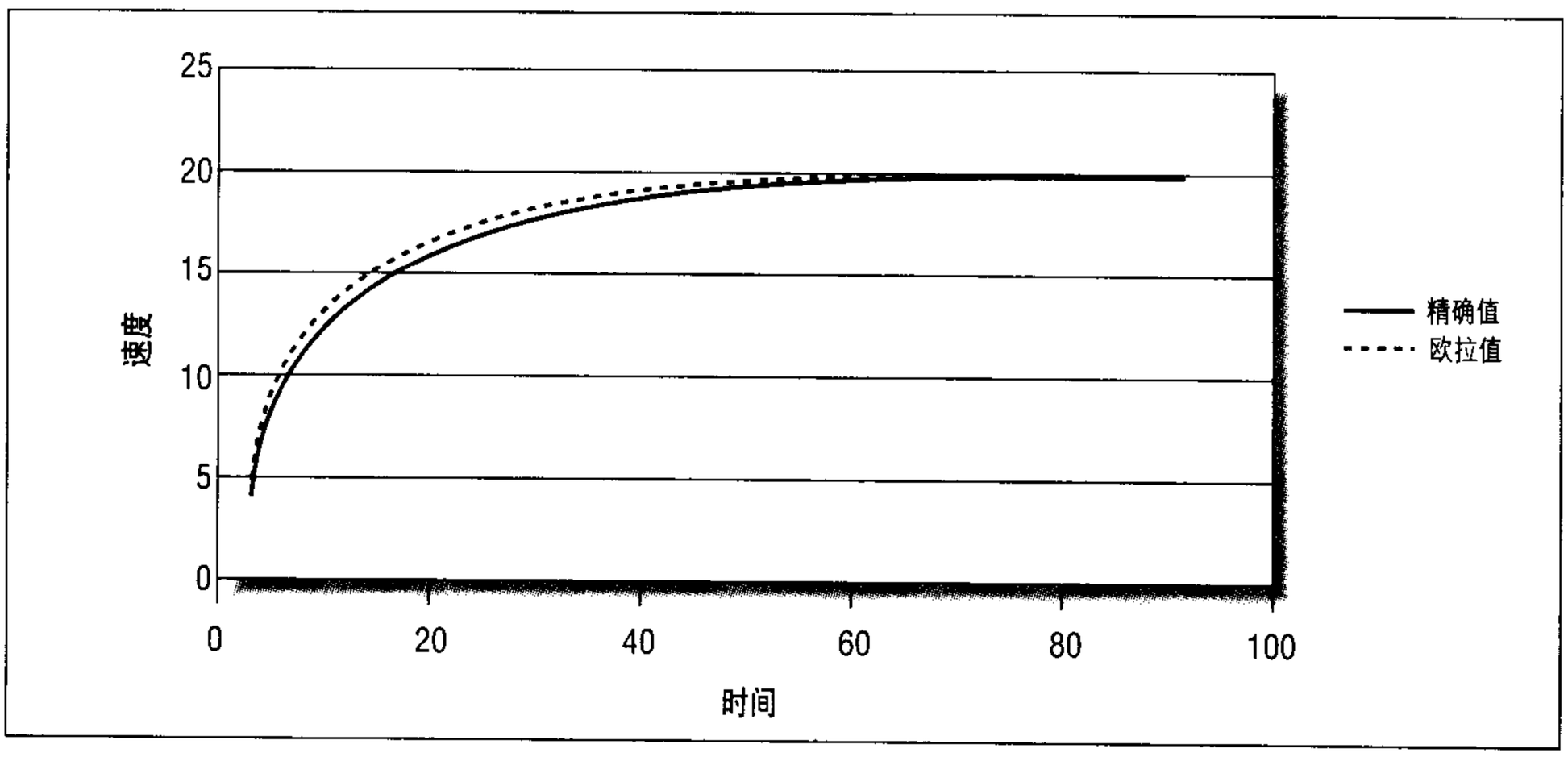



图 11-4：收敛

以下这段程序代码使用欧拉法实现了这个范例：

```
// 全局变量
float    T;   // 推力
float    C;   // 阻力系数
float    V;   // 速度
float    M;   // 质量
float    S;   // 位移
.
.
.
// 此函数使用欧拉法估计 dt 秒的新速度及位移
void StepSimulation(float dt)
{
```

```
    float   F;     // 总合力
    float   A;     // 加速度
    float   Vnew; // 在时间 t + dt 时的新速度
    float   Snew; // 在时间 t + dt 时的新位置

    // 计算总合力
    F = (T - (C * V));

    // 计算加速度
    A = F / M;

    // 计算在时间 t + dt 时的新速度，其中 V 是时间 t 时的速度
    Vnew = V + A * dt;

    // 计算在时间 t + dt 时的新位移，其中 S 是时间 t 时的位置
    Snew = S + Vnew * dt;

    // 更新速度和位移值
    V = Vnew;
    S = Snew;
}
```

虽然欧拉法在本范例中是稳定的，但是在碰到其他的问题时却常常不是这样，所以在实现任何数值积分法时都要随时注意估计值是否稳定。这里所说的稳定是指欧拉值会收敛而近似于精确值。如果不稳定，则欧拉值对于精确值会呈现发散的状况，而使误差随着时间的增加而扩大。在另一种情况中，估计值会在精确值的上下来回摆荡，也被视为不稳定。

选择间隔的大小常常会影响稳定与否：越小的间隔越减小不稳定性而越大的间隔则越增大不稳定性。当处理一个庞大的函数时，减小间隔可以增加稳定性，然而却增加计算量。解决这种问题时可以使用称为“可变间隔”的方法。这种方法是根据截断误差的大小，决定是否立刻改变下次间隔的大小。如果截断误差太大，可以回到上一个间隔，减少间隔大小再计算一次。

另一个实现欧拉法的方式是先选定一个间隔 $\Delta t$，并算出在时间 $t + \Delta t$ 时的估计值。接着再从时间 $t$ 开始，间隔选定为 $\Delta t/2$，并连续做两次计算，最后算出在时间 $t + \Delta t$ 时的估计值。因为已经在这个例子中讨论过速度，所以这里用 $v_1$ 代表第一个估计值，用 $v_2$ 代表第二个估计值（注 2）。对于截断误差的计算是

$$e_t = |v_1 - v_2|$$

如果需要将截断误差控制在一个值 $e_{to}$ 以下，以下的公式可算出要维持某个精确度所要使用的间隔大小：

---

注 2：虽然这里只讨论速度对时间的函数，不过此方法可用在其他函数上，例如位移对时间的函数。

$$h_{new} = h_{old}(e_{to}/e_t)^{(1/2)}$$

这里的$h_{old}$是旧的时间间隔，而$h_{new}$是为了要维持精确度。在每次间隔的计算中都必须检查误差值，如果发现必须要用更小的时间间隔才能确保误差，就要回到上一个时间间隔，使用新的时间间隔再重复计算一次。

以下是修改过的StepSimulation函数，实现了可变间隔的技巧。它在对速度积分时会检查截断误差：

```
// 新全局变量
float   eto;   // 截断误差的许可值

// 此函数使用欧拉法估计 dt 秒的新速度及位移
void StepSimulation(float dt)
{
     float      F;       // 总合力
     float      A;       // 加速度
     float      Vnew;    // 在时间 t + dt 时的新速度
     float      Snew;    // 在时间 t + dt 时的新位置
     float      V1, V2;  // 暂时的速度变量
     float      dtnew;   // 新的时间间隔
     float      et;      // 截断误差

     // 以间隔 dt 求新的速度
     F = (T - (C * V));
     A = F / M;
     V1 = V + A * dt;

     // 以间隔 dt/2 求新的速度（计算两遍）
     F = (T - (C * V));
     A = F / M;
     V2 = V + A * (dt/2);

     F = (T - (C * V2));
     A = F / M;
     V2 = V2 + A * (dt/2);

     // 估计截断误差
     et = absf(V1 - V2);

     // 估计新的间隔大小
     dtnew = dt * SQRT(eto/et);

     if (dtnew < dt)
     { // 使用更小的间隔
          F = (T - (C * V));
          A = F / M;
          Vnew = V + A * dtnew;
          Snew = S + Vnew * dtnew;
     } else
     { // 原先的间隔没问题
          Vnew = V1;
```

```
            Snew = S + Vnew * dt;
        }

        // 更新速度和位移值
        V = Vnew;
        S = Snew;
    }
```

## 其他的方法

这里你可能会有个疑问，为什么不用更多项的泰勒展开式来减小欧拉法的截断误差呢？事实上，这是比欧拉法更能精确估计的其他数值积分方法的基础。在取出泰勒展开式更多项时，部分的困难是如何决定函数的二次、三次或更高次的导数。解决这个问题的方式，是做额外的泰勒展开式来估计函数的导数，然后替换原先展开式中的值。

采用此方法将超出基本欧拉法更多的泰勒展开项列入，即演变成所谓的“改良型欧拉法”，并能减小截断误差，使之由$(\Delta t)^2$阶进步到$(\Delta t)^3$阶。新的欧拉法公式如下：

$$k_1 = (\Delta x)y'(x, y)$$
$$k_2 = (\Delta x)y'(x + \Delta x, y + k_1)$$
$$y(x + \Delta x) = y(x) + 1/2(k_1 + k_2)$$

其中，$y$是$x$的函数，$y'$是导数（且为$x$或$y$的函数），而$\Delta x$是间隔大小。

为了要解释得更清楚（和前一节一样），再以第四章中船的运动方程为例。在这个例子中，速度的值由以下公式来估计：

$$k_1 = \Delta t[1/m(T - Cv_t)]$$
$$k_2 = \Delta t\{1/m[T - C(v_t + k_1)]\}$$
$$v_{t+\Delta t} = v_t + 1/2(k_1 + k_2)$$

其中$v_t$是在时间$t$时的速度，而$v_{t+\Delta t}$是在时间为$t + \Delta t$时的新速度。

以下是修改过的StepSimulation函数，实现了这个方法：

```
// 此函数使用改进的欧拉法估计 dt 秒的新速度及位移
void StepSimulation(float dt)
{
    float   F;      // 总合力
    float   A;      // 加速度
    float   Vnew;   // 在时间 t + dt 时的新速度
    float   Snew;   // 在时间 t + dt 时的新位置
    float   k1, k2;
```

```
    F = (T - (C * V));
    A = F/M;
    k1 = dt * A;

    F = (T - (C * (V + k1)));
    A = F/M;
    k2 = dt * A;

    // 计算在时间 t + dt 时的新速度，其中 V 是在时间 t 时的速度
    Vnew = V + (k1 + k2) / 2;

    // 计算在时间 t + dt 时的新位移，其中 S 是在时间 t 时的位移
    Snew = S + Vnew * dt;

    // 更新速度和位移值
    V = Vnew;
    S = Snew;
}
```

这种使用更多泰勒展开式项的方法可再被改良。著名的Runge-Kutta法使用了相同的方法来将截断误差减小到$(\Delta t)^5$。这个方法使用的积分方程如下：

$$k_1 = (\Delta x)y'(x, y)$$
$$k_2 = (\Delta x)y'(x + \Delta x/2, y + k_1/2)$$
$$k_3 = (\Delta x)y'(x + \Delta x/2, y + k_2/2)$$
$$k_4 = (\Delta x)y'(x + \Delta x, y + k_3)$$
$$y(x + \Delta x) = y(x) + 1/6[k_1 + 2(k_2) + 2(k_3) + k_4]$$

将这些方程用在船的例子中得到：

$$k_1 = \Delta t[1/m(T - Cv_t)]$$
$$k_2 = \Delta t\{1/m[T - C(v_t + k_1/2)]\}$$
$$k_3 = \Delta t\{1/m[T - C(v_t + k_2/2)]\}$$
$$k_4 = \Delta t\{1/m[T - C(v_t + k_3)]\}$$
$$v_{t+\Delta t} = v_t + 1/6[k_1 + 2(k_2) + 2(k_3) + k_4]$$

在这个例子中，Runge-Kutta法可用以下方法实现：

```
// 此函数使用 Runge-Kutta 法估计时间 dt 秒的速度和位移
void StepSimulation(float dt)
{
    float   F;    // 总合力
    float   A;    // 加速度
    float   Vnew; // 在时间 t + dt 时的新速度
    float   Snew; // 在时间 t + dt 时的新位置
    float   k1, k2, k3, k4;

    F = (T - (C * V));
```

```
    A = F/M;
    k1 = dt * A;

    F = (T - (C * (V + k1/2)));
    A = F/M;
    k2 = dt * A;

    F = (T - (C * (V + k2/2)));
    A = F/M;
    k3 = dt * A;

    F = (T - (C * (V + k3)));
    A = F/M;
    k4 = dt * A;

    // 计算在时间 t + dt 时的新速度，其中 V 是在时间 t 时的速度
    Vnew = V + (k1 + 2*k2 + 2*k3 + k4) / 6;

    // 计算在时间 t + dt 时的新位移，其中 S 是在时间 t 时的位移
    Snew = S + Vnew * dt;

    // 更新速度和位移值
    V = Vnew;
    S = Snew;
}
```

为了显示与基本欧拉法相比精确度的改进，以下将这两种方法的积分结果和图11-2及图11-3重叠画在图11-5及图11-6中，其中图11-6是将图11-5的局部放大。

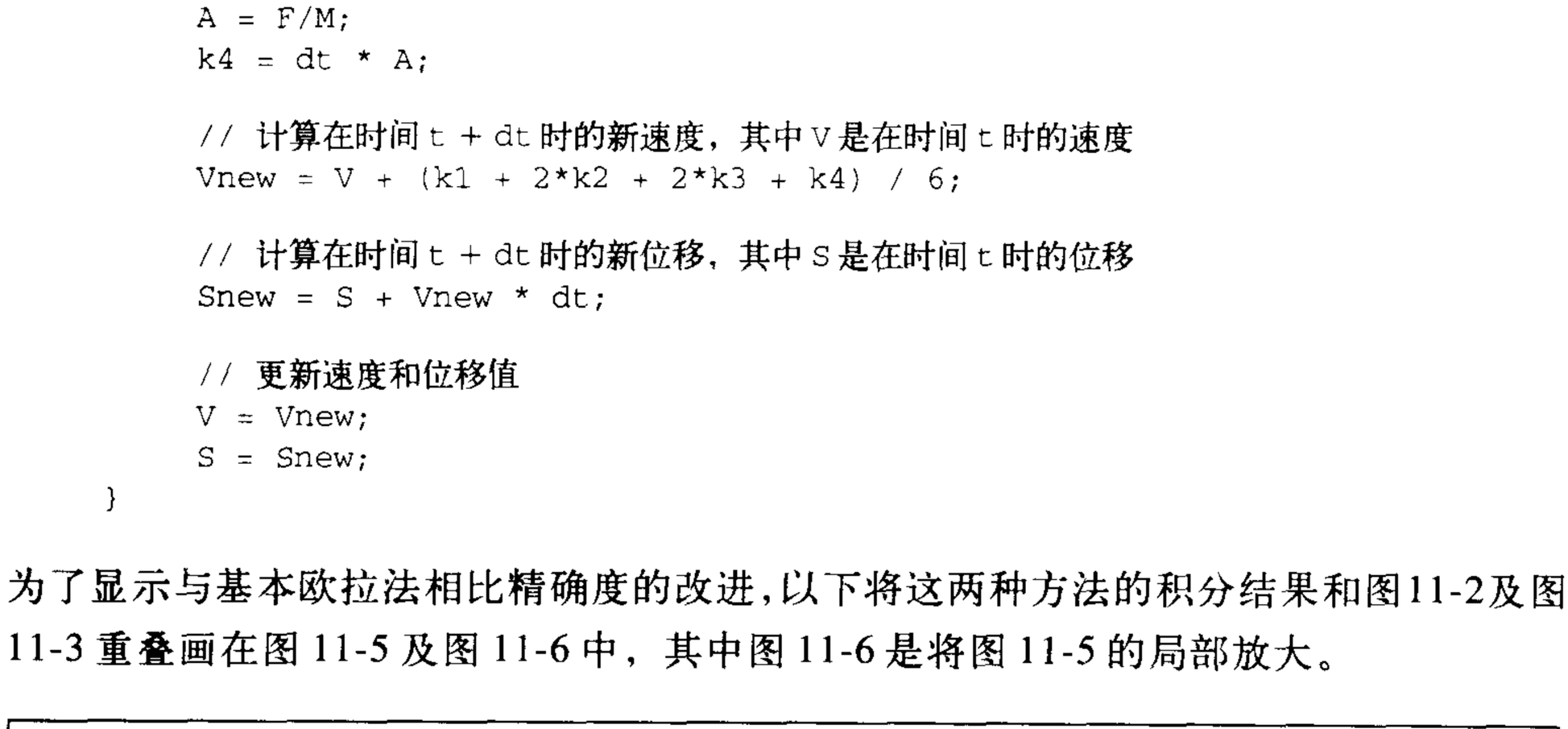



图 11-5：不同方法间的比较（译注 2）

译注 2：图 11-5 与图 11-6 中，有两条曲线看不出来是因为误差太小，且以图中的比例无法清楚地与精确值曲线区别。

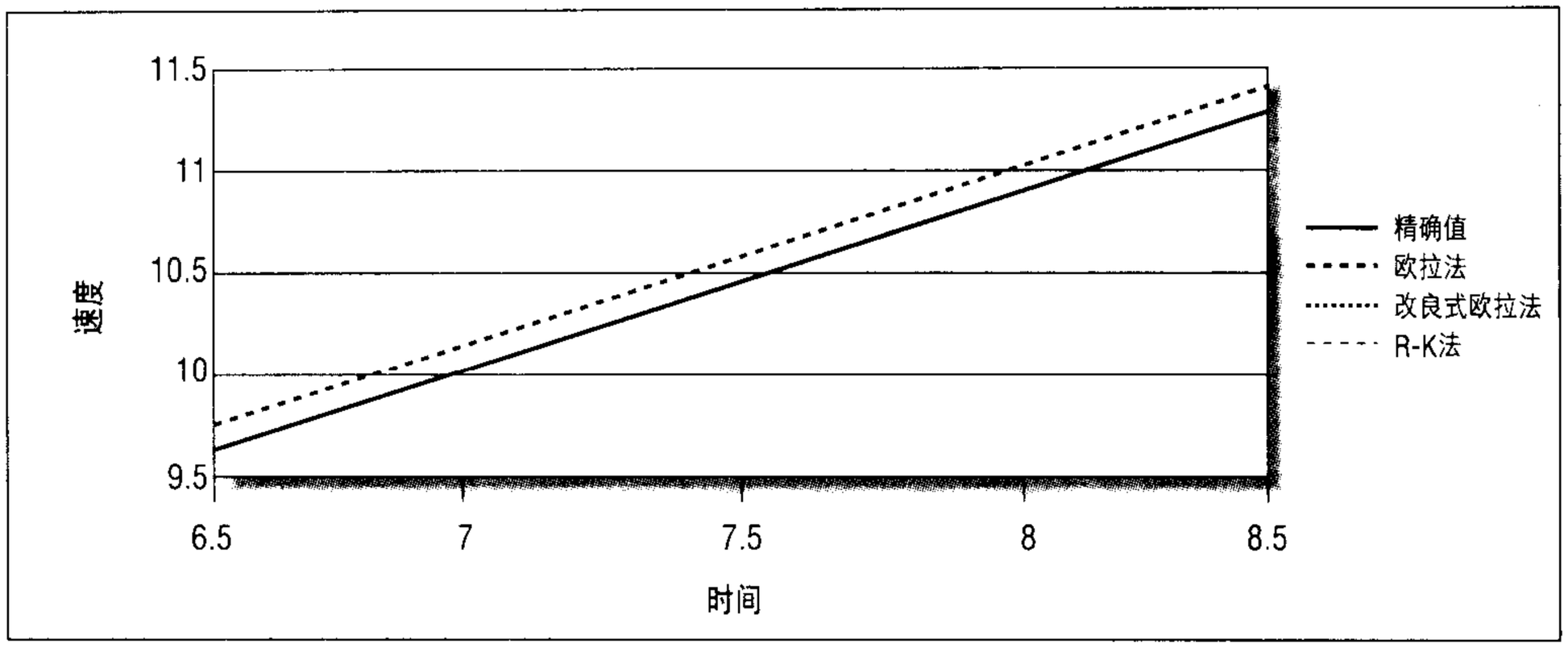



图 11-6：比较图的局部放大

从图中可以看到，很难分辨改良型欧拉法及 Runge-Kutta 法与精确值所画的线，因为它们几乎是互相重叠的。这些结果清楚地显示了其精确度优于基本欧拉法的改进 —— 基本欧拉法的曲线与其他三条是分开的。在 6.5～8.5 秒间的间隔中，欧拉法的平均截断误差是 1.72%，改良型欧拉法是 0.03%，而 Runge-Kutta 法是 $3.6 \times 10^{-6}$%。从这个结果能很明显地看出，Runge-Kutta 法比其他只是改变时间间隔大小的方法要好。要有这样的精确度必须付出代价，在 Runge-Kutta 法中每个时间间隔所要做的运算比其他的方法更多。

这些方法并不是处理这类问题的惟一方法，然而，它们却是最常见的。其他方法努力想要使每一时间间隔所做的计算更少 —— 想要减小截断误差并且仍能使用较大的时间间隔，使在做积分时不用增加计算次数。在参考文献中列举了一些相当好的参考资料可供你更进一步学习。

# 第十二章

# 2D 刚体模拟器

现在可以着手整合目前所学的内容并实现实际的实时模拟程序了。本章将实现一个 2D 刚体模拟器，这个模拟程序选用两艘气垫船作为范例，以便在下一章继续沿用这个范例，示范如何处理气垫船之间的碰撞反应。但是现在，我先简单而清楚地整理前面几章所介绍的内容。

虽然这个范例的所有原始代码在 O'Reilly 网站上（*http://www.oreilly.com/catalog/physicsgame/*）都可以找到，但是这里仍然会收录大部分。书中的程序代码着重在模拟器中实现物理学的部分，其他未收录的部分是使用Microsoft的DirectX函数库实现的绘图部分，你可以在 O'Reilly 的网站上找到它们。

这个模拟器包含了 4 个主要部分：

模型

模型指的是将物体的规格数值化的方法，在这个范例中使用气垫船当做模拟对象。

积分函数

这部分指的是用来对运动的微分方程积分的函数。

用户输入

用户输入及用户和模拟程序互动的部分。

绘图

最后，绘图为用户在模拟程序中所看到的图像。

在这个模拟器中，全局坐标系统的 $x$ 轴方向指向屏幕中，$y$ 轴的方向指向屏幕左方，$z$ 轴的方向指向屏幕上方。虽然本章使用的是 2D 的模拟程序，所有的运动只能在 $xy$ 平面上

作用，但仍然需要 $z$ 轴来标识气垫船旋转的方向。此外，物体也有局部的坐标系（或称固定坐标系），其中 $x$ 轴的方向指向气垫船前方，$y$ 轴的方向指向气垫船的左舷，而 $z$ 轴的方向是指向气垫船的上方。

# 模型

这个模拟程序所模拟的是两艘在平地上操作的气垫船。这两艘船是完全相同的，其规格如表 12-1 所示。

表 12-1：气垫船的规格

| 各项特性 | 数值 |
|---|---|
| 长度 | 70 ft |
| 宽度 | 50 ft |
| 全车的平均投影面积 | 1500 $ft^2$ |
| 阻力的中心点 | 由重心向船尾 2.5 ft 处 |
| 重量 | 10 吨（20000 lb） |
| 质量 | 621.6 slug |
| 重心（CG）位置 | 船首沿中线向船尾 35 ft 处（假设在气垫船中心处） |
| 转动惯量[a] | 383320 lb-ft-$s^2$ |
| 最大推力（空气推进器） | 2000 lb |
| 推进器位置 | 重心沿中线向后 30 ft 处 |
| 侧推进器 | 左右舷各一，位于重心前 30 ft 距中线两侧 25 ft 处 |
| 侧推进器的推力 | 500 lb |
| 极速 | 40 kt（67.51 ft/s） |

a. 请注意，在2D空间中转动惯量是标量。所以本范例中，转动惯量是绕着通过重心的局部坐标 $z$ 轴旋转的。

每一艘气垫船都在船尾装设了一个提供前进（或后退）推力的单螺旋桨空气推进器。为了操控方便，在船的左右舷各装设了一个侧推进器，各提供了500 lb的推力。船侧推进器可用来改变气垫船的方向。

这里使用一个简化的阻力模型，只考虑全船的空气动阻力。计算时假设平均投影面积是 1500 $ft^2$ 而阻力系数是0.25。更严格的模型应该将真正的投影面积设定为相对速度方向的函数，如同第七章的飞行模拟器一样，船的挡板下缘与地面间的摩擦阻力也应该如此

设定。这里也假设阻力中心在重心向船尾 2.5 ft 处。如此一来，可以增加些方向上的稳定性用来抵消转动。此功能与飞机上的垂直尾翼相同。

在程序代码中，先将所有描述气垫船运动状态及计算气垫船上的作用力所需要的数据，定义成一个刚体的数据结构。以下是我所定义的数据结构：

```
typedef struct _RigidBody {

    float     fMass;                // 总质量（常数）
    float     fInertia;             // 以局部坐标定义的转动惯量
    float     fInertiaInverse;      // 反转动惯量
    Vector    vPosition;            // 全局坐标中的位置
    Vector    vVelocity;            // 全局坐标中的速度
    Vector    vVelocityBody;        // 局部坐标中的速度
    Vector    vAngularVelocity;     // 局部坐标中的角速度

    float     fSpeed;               // 速率（速度的大小）
    float     fOrientation;         // 方位
    Vector    vForces;              // 物体上的总合力
    Vector    vMoment;              // 物体上的总力矩（2D空间：只能绕 z 轴转动）

    Vector    CD;                   // 阻力中心相对于重心的位置
    Vector    CT;                   // 推力中心相对于重心的位置
    Vector    CPT;                  // 左舷推进器相对于重心的位置
    Vector    CST;                  // 右舷推进器相对于重心的位置
    float     ProjectedArea;        // 平均投影面积（计算阻力用）
    float     ThrustForce;          // 推力的方向
    Vector    PThrust, SThrust;     // 侧推进器的向量

    float     fWidth;               // 气垫船的大小尺寸
    float     fLength;

} RigidBody2D, *pRigidBody2D;
```

这个结构包含了描述气垫船状态的所有数据。

下一步是完成当程序初次启动时所调用的初始化函数，将气垫船的状态初始化。此函数的程序代码如下：

```
void     InitializeHovercraft(pRigidBody2D body)
{
    // 设定初始位置
    body->vPosition.x = 0.0f;
    body->vPosition.y = 0.0f;
    body->vPosition.z = 0.0f;          // 2D空间中所有的 z 值皆为 0

    // 设定初始速度
    body->vVelocity.x = 0.0f;
    body->vVelocity.y = 0.0f;
    body->vVelocity.z = 0.0f;          // 2D空间中所有的 z 值皆为 0
    body->fSpeed = 0.0f;
```

```
    // 设定初始角速度
    body->vAngularVelocity.x = 0.0f;        // 2D空间中为0
    body->vAngularVelocity.y = 0.0f;        // 2D空间中为0
    body->vAngularVelocity.z = 0.0f;        // 2D空间中只有这个值有用

    // 设定初始推力、力和力矩
    body->vForces.x = 0.0f;
    body->vForces.y = 0.0f;
    body->vForces.z = 0.0f;                 // z值总为0
    body->vMoment.x = 0.0f;                 // 2D空间中为0
    body->vMoment.y = 0.0f;                 // 2D空间中为0
    body->vMoment.z = 0.0f;                 // 2D空间中只有这个值有用

    // 将局部坐标中的速度设为0
    body->vVelocityBody.x = 0.0f;
    body->vVelocityBody.y = 0.0f;
    body->vVelocityBody.z = 0.0f;

    // 设定初始方向
    body->fOrientation = 0.0;

    // 定义质量特性
    body->fMass = 621.6;
    body->fInertia = 383320;
    body->fInertiaInverse = 1.0f / body->fInertia;

    // 物体阻力中心的坐标
    body->CD.x = -2.5f;         body->CD.y = 0.0f;

    // 推进器推力向量的坐标
    body->CT.x = -30.0f;    body->CT.y = 0.0f;

    // 左舷推进器的坐标
    body->CPT.x = 30.0f;    body->CPT.y = 25.0f;

    // 右舷推进器的坐标
    body->CST.x = 30.0f;    body->CST.y = -25.0f;

    body->ProjectedArea = 1500.0f;   // 平均投影面积
    body->ThrustForce = 0;           // 初始化推力

    body->fWidth = 50;    // 船体的宽度（沿着y轴计算）
    body->fLength = 70;   // 船体的长度（沿着x轴计算）
}
```

这里使用的Vector类有三个成员变量，分别是：$x$，$y$和$z$。因为这是2D的范例，所以$z$值皆为0，但是在计算角速度时，只有$z$值会被用到（因为船是绕着$z$轴旋转的）。这个类可以在附录一中找到，所以这里并没有附上原始代码。这里不另外定义一个只包含$x$，$y$数据的2D Vector类，是因为本范例不久将会被沿用并扩充成3D的范例。此外，你只要将3D的Vector类中的成员变量$z$忽略就成了2D的Vector类了。

请注意，这个函数使用一个指向RigidBody结构的指针作为参数。这样一来，可以调用

相同的初始化函数来初始化这两艘气垫船。如果之后要改变某一艘船的初始状态，只要改变相关的属性即可。以下是初始化两艘气垫船的范例：

```
void      Initialize(void)
{
     InitializeHovercraft(&Hovercraft1);
     InitializeHovercraft(&Hovercraft2);

     Hovercraft2.vPosition.y = -50;
     Hovercraft2.vPosition.x = 1500;
     Hovercraft2.fOrientation = 180;
}
```

Hovercraft1 和 Hovercraft2 定义成两个全局变量：

```
RigidBody2D      Hovercraft1, Hovercraft2;   // 两个气垫船物体
```

这段程序代码并不是使两艘气垫船出现在同一个地点，而是使第二艘船离第一艘船有一段距离并旋转 180°，让它面向第一艘船。

初始函数在程序的前面就会被调用，因此在主窗口建立并显示后立即调用初始化函数是不错的选择。这个范例把它放在 Windows API 的 InitInstance 函数之中：

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{

     hInst = hInstance;
     nShowCmd = nCmdShow;

     hTheMainWindow = CreateWindow(szAppName,
                                   szTitle,
                                   WS_OVERLAPPEDWINDOW |
                                   WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                                   0, 0, 640, 480,
                                   NULL, NULL, hInst, NULL);

     if (!CreateD3DRMObject())
          return (FALSE);

     if (!CreateD3DRMClipperObject(hTheMainWindow))
          return (FALSE);

     if (!CreateViewPort(hTheMainWindow))
          return (FALSE);

     ShowWindow(hTheMainWindow, nCmdShow);
     UpdateWindow(hTheMainWindow);

     Initialize();

     return (TRUE);
}
```

在初始化气垫船之后，还需要一个函数来计算模拟程序中作用在气垫船上的力和力矩。如果没有此函数，气垫船就只能静止不动。所以很明显，这个函数十分重要。虽然本章中的其他程序代码通常不需要修改就能用在别的刚体模拟程序中，但这个函数却不能这样。因为每个模拟程序都需要一个函数来计算力和力矩，而这个函数会因为不同的物理模型而大不相同。以下是为了气垫船所写的函数：

```
void      CalcLoads(pRigidBody2D body)
{
     Vector        Fb;              // 存储总合力的值
     Vector        Mb;              // 存储总力矩的值
     Vector        Thrust;          // 推力向量

     // 重设力和力矩
     body->vForces.x = 0.0f;
     body->vForces.y = 0.0f;
     body->vForces.z = 0.0f;     // 2D空间中为0

     body->vMoment.x = 0.0f;     // 2D空间中为0
     body->vMoment.y = 0.0f;     // 2D空间中为0
     body->vMoment.z = 0.0f;

     Fb.x = 0.0f;
     Fb.y = 0.0f;
     Fb.z = 0.0f;

     Mb.x = 0.0f;
     Mb.y = 0.0f;
     Mb.z = 0.0f;

     // 定义作用于气垫船重心的推力向量
     Thrust.x = 1.0f;
     Thrust.y = 0.0f;
     Thrust.z = 0.0f;              // 2D空间中为0
     Thrust *= body->ThrustForce;

     // 计算作用于船体的力与力矩
     Vector     vLocalVelocity;
     float      fLocalSpeed;
     Vector     vDragVector;
     float      tmp;
     Vector     vResultant;
     Vector     vtmp;

     // 计算空气阻力：
     // 计算本身的速度：
     // 本身的速度包括船的线性速度加上船体每部分旋转的速度
     vtmp = body->vAngularVelocity^body->CD; // 旋转的部分
     vLocalVelocity = body->vVelocityBody + vtmp;

     // 计算空气速率
     fLocalSpeed = vLocalVelocity.Magnitude();
```

```
// 找出阻力作用的方向：
// 阻力通常与相对速度共线但方向相反
if(fLocalSpeed > tol)
{
      vLocalVelocity.Normalize();
      vDragVector = -vLocalVelocity;
      // 计算出每个部分的合力
      tmp = 0.5f * rho * fLocalSpeed*fLocalSpeed *
            body->ProjectedArea;
      vResultant = vDragVector * LINEARDRAGCOEFFICIENT * tmp;
      // 计算出总合力
      Fb += vResultant;

      // 计算每个部分相对重心的力矩并算出总力矩
      vtmp = body->CDvResultant;
      Mb += vtmp;
}

// 计算左右舷的侧推力，并累计到总合力
Fb += body->PThrust;

// 计算左弦相对重心的力矩，并累计到总力矩
vtmp = body->CPT^body->PThrust;
Mb += vtmp;

      // 加上右弦推力
      Fb += body->SThrust;

      // 计算右弦相对重心的力矩并加合到总力矩中
      vtmp = body->CST^body->SThrust;
      Mb += vtmp;

      // 加上主推力
      Fb += Thrust; // 主推力因为作用线通过重心，所以不会产生力矩

      // 将力的局部坐标转换成全局坐标
      body->vForces = VRotate2D(body->fOrientation, Fb);

      body->vMoment += Mb;
}
```

因为两艘气垫船是完全相同的，所以只要将所求气垫船的RigidBody数据结构的指针，传给相同的函数就可以计算出各自的量值。

CalcLoads函数先初始化力和力矩的变量。这些变量会用来存储在某个时间，作用于船上的合力和合力矩的量值。

接着函数会定义代表推进器推力的向量。这个范例中所用的推力向量会作用在$x$轴正值（局部坐标）的方向，且其量值由ThrustForce所定义（使用者透过键盘设定）。当ThrustForce为负值时，会产生反推力（向后）而不是正推力（向前）。

在定义推力向量后，此函数接着计算作用于气垫船上的空气动阻力。计算方式和第七章所用的很类似。先找出在阻力中心的相对速度，包含线性速度和角速度。当计算阻力大小时需要相对速度的大小，要判断阻力的方向则需要相对速度向量的方向，因为它们通常是相反的。计算完后，函数会将它加到总合力中。同样，函数计算对重心的阻力力矩并将它加入总力矩中。阻力系数 LINEARDRAGCOEFFICIENT 定义如下：

```
#define LINEARDRAGCOEFFICIENT          0.25f
```

完成阻力的计算后，接着计算侧推进器（可在任何时间被开启或关闭）的力和力矩。

下一步，将主推进器的推力加入总合力中。因为主推进器的推力通过船的重心，所以不必考虑力矩。

最后，通过给定气垫船方位的旋转向量将总合力从局部坐标转换成全局坐标，再将总合力和总力矩值存储在个别的RigidBody数据结构中。这些数值可以在模拟程序的任何时间间隔，对运动方程积分时使用。

# 积分函数

现在已经完成了用来定义模型、初始化和计算刚体上受力的程序代码，接着要继续开发真正能对运动方程积分的程序，让模拟程序能计算时间进行中的变化。先选择在第十一章提过的所要使用的积分方法。在这个范例中将使用“改良型欧拉法”。UpdateBody函数负责这方面的工作，此函数使用指向RigidBody数据结构的指针和时间间隔（以秒为单位）作为参数。

```
void    UpdateBody(pRigidBody2D craft, float dtime)
{
        Vector Ae;
        float Aa;
        RigidBody2D     body;
        Vector          k1, k2;
        float           k1a, k2a;
        float           dt = dtime;

// 复制气垫船的状态
memcpy(&body, craft, sizeof(RigidBody2D));

// 计算线性速度和角速度的 k1 值
CalcLoads(&body);
Ae = body.vForces / body.fMass;
k1 = Ae * dt;

Aa = body.vMoment.z / body.fInertia;
k1a = Aa * dt;
```

```
// 将k1值加到初始速度和角速度中
body.vVelocity += k1;
body.vAngularVelocity.z += k1a;

// 计算新的受力和k2值
CalcLoads(&body);
Ae = body.vForces / body.fMass;
k2 = Ae * dt;

Aa = body.vMoment.z / body.fInertia;
k2a = Aa * dt;

// 计算在时间t + dt时的新速度
craft->vVelocity += (k1 + k2) / 2.0f;
craft->vAngularVelocity.z += (k1a + k2a) / 2.0f;

// 计算新位置
craft->vPosition += craft->vVelocity * dt;
craft->fSpeed = craft->vVelocity.Magnitude();

// 计算新方位
craft->fOrientation +=
            RadiansToDegrees(craft->vAngularVelocity.z * dt);

craft->vVelocityBody =
            VRotate2D(-craft->fOrientation, craft->vVelocity);
}
```

因为函数以RigidBody的指针为参数，所以不同物体可重复使用同一个函数。另外，使用时间间隔为参数可以任意改变适合的时间间隔大小。下一章谈到处理碰撞反应时，将会使用此技巧。

UpdateBody函数先暂时复制一份刚体目前的状态。这么做的原因是在改良型欧拉法的积分步骤中需要将初始速度加上$k_1$值。为了不改变在步骤中仍然需要的初始速度值，所以先做一个备份。

接下来，用之前备份的指针作为参数调用CalcLoads函数计算刚体上的力与力矩。接着计算线性速度和角速度的$k_1$值。这个$k_1$值被加到初始速度中，然后再调用CalcLoads函数。$k_2$值就是在第二次调用CalcLoads后求出的。

现在已求出$k_1$和$k_2$的值了，可以用改良型欧拉法算出新的速度。接着，函数使用欧拉法将新速度积分，得到此刚体新的位置和方位。

最后UpdateBody根据物体新的方位套用全局坐标的向量旋转，计算出在局部坐标的速度。以后在CalcLoads函数中计算阻力时，将会需要这个局部坐标的速度，而在这里求出此速度是个合适的选择。

因为在模拟程序中有两个刚体——两艘气垫船，所以这两个物体必须各调用UpdateBody一次。本例在 StepSimulation 函数中实现此步骤：

```
void StepSimulation(float dt)
{
    UpdateBody(&Hovercraft1, dt);
    UpdateBody(&Hovercraft2, dt);
}
```

StepSimulation在这个模拟程序中是不重要的，因为它只有两个刚体而且并没有碰撞反应的机制。如果这里在模拟程序中有许多刚体，就必须设定一个RigidBody的阵列来存储这些刚体，并且在 StepSimulation 中用循环的方式来更新每个刚体的状态。

StepSimulation函数应该在每一次的游戏循环中被调用。对于这个范例我使用另一个函数 NullEvent，会在每次的主窗口消息循环中被调用：

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow)
{
.
.
.
.

    OldTime = timeGetTime();
    NewTime = OldTime;
    // 主要消息循环:
    while (1) {

        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
            if (msg.message == WM_QUIT) {
                return msg.wParam;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        NullEvent();

    }
.
.
.
}
```

当NullEvent 函数调用 StepSimulation函数时，使用时间间隔作为*dt*参数传入，但并不是必须如此。我这样做是因为要试验用即时求出的时间间隔，当做上次调用 StepSimulation 与目前时间的时间差。程序代码如下：

```
void    NullEvent(void)
{
.
.
.
      NewTime = timeGetTime();
      dt = (float) (NewTime - OldTime)/1000;
      OldTime = NewTime;
      if (dt > 0.016) dt = 0.016;
      if (dt < 0.001f) dt = 0.001f;

      StepSimulation(dt);
.
.
.
}
```

这样做使用了真实比例的时间来处理模拟。问题是如果程序花太多时间执行某一个消息循环，下一个时间间隔会因此而太大，就会使刚体的运动不顺畅，更不用说太大的时间间隔会导致积分的不正确和不稳定。所以这里用一点小技巧让时间间隔不至于太大，也避免时间间隔小于1毫秒（ms）。timeGetTime函数返回值的精确度虽然只有1 ms，却常常返回小于1 ms的值。所以这里加上程序代码以确认最小时间为1 ms并保持一致，以维持timeGetTime函数的精确度。

另一个选择是固定时间间隔使每一个间隔都相同，而不论游戏循环中是否有延迟。好的时间间隔大小需要自行实验来决定。如果选定一个太小的值，模拟程序看起来像是慢动作。相反，如果选定太大的值，会使程序看起来像是在快转，并且会增加计算上的困难。

# 飞行控制

执行目前完成的模型和积分函数的程序代码时，气垫船仍然不会动。这是因为还没有完成控制的机制。接下来介绍的程序代码是关于处理使用者的输入，会与气垫船互动并控制它们的动作。更具体地说，是将键盘上的某些按键与作用在气垫船的某些力结合起来——这些力之前已经提过了，就是主推力和两个侧推力。如此，你不能直接推动或转动气垫船；只能对气垫船施力，并用积分函数计算气垫船在这些力作用下的行为。

这个例子中的控制系统非常简单。对一号气垫船而言，方向键的上键每次会增加主推力100 lb直到上限2000 lb；下键每次会减少主推力100 lb直到下限−2000 lb；方向键的左键使用右舷推进器让船身向左转；方向键的右键使用左舷推进器让船身向右转。对于二号气垫船而言，分别使用W，Z，A，S键执行相同的动作。

当使用者按下飞行控制键时，程序会调用一些函数来处理主推进器和侧推进器的作用。以下两个函数处理主推进器：

```
void        IncThrust(int craft)
{
      if(craft == 1)
      {
          Hovercraft1.ThrustForce += _DTHRUST;
          if(Hovercraft1.ThrustForce > _MAXTHRUST)
              Hovercraft1.ThrustForce = _MAXTHRUST;
      } else {
          Hovercraft2.ThrustForce += _DTHRUST;
          if(Hovercraft2.ThrustForce > _MAXTHRUST)
              Hovercraft2.ThrustForce = _MAXTHRUST;
      }
}

void        DecThrust(int craft)
{

      if(craft == 1)
      {
          Hovercraft1.ThrustForce -= _DTHRUST;
          if(Hovercraft1.ThrustForce < -_MAXTHRUST)
              Hovercraft1.ThrustForce = -_MAXTHRUST;
      } else {
          Hovercraft2.ThrustForce -= _DTHRUST;
          if(Hovercraft2.ThrustForce < -_MAXTHRUST)
              Hovercraft2.ThrustForce = -_MAXTHRUST;
      }
}
```

IncThrust函数只是增加主推力_DTHRUST单位，并检查主推力的值是否超过_MAXTHRUST单位。_DTHRUST和_MAXTHRUST的值定义如下：

```
#define       _DTHRUST      100.0f
#define       _MAXTHRUST    2000.0f
```

另一方面，DecThrust函数减少主推力_DTHRUST单位，并且检查主推力的值是否低于-_MAXTHRUST单位。这两个函数都使用一个代表气垫船（一号或二号）的整数作为参数，以识别推力的改变该套用到哪艘气垫船。

接下来的函数负责侧推进器：

```
void        PortThruster(int craft)
{
      if(craft == 1)
          Hovercraft1.PThrust.y = -500.0f;
      else
          Hovercraft2.PThrust.y = -500.0f;
}
```

```
void        STBDThruster(int craft)
{
      if(craft == 1)
           Hovercraft1.SThrust.y = 500.0f;
      else
           Hovercraft2.SThrust.y = 500.0f;
}

void        ZeroThrusters(int craft)
{
      if(craft == 1)
      {
           Hovercraft1.PThrust.x = 0.0f;
           Hovercraft1.PThrust.y = 0.0f;
           Hovercraft1.PThrust.z = 0.0f;

           Hovercraft1.SThrust.x = 0.0f;
           Hovercraft1.SThrust.y = 0.0f;
           Hovercraft1.SThrust.z = 0.0f;
      } else {
           Hovercraft2.PThrust.x = 0.0f;
           Hovercraft2.PThrust.y = 0.0f;
           Hovercraft2.PThrust.z = 0.0f;

           Hovercraft2.SThrust.x = 0.0f;
           Hovercraft2.SThrust.y = 0.0f;
           Hovercraft2.SThrust.z = 0.0f;
      }
}
```

PortThruster函数将左舷推进器的推力设定在−500，也就是朝着右舷方向500 lb的力，让船身向右舷方向转。负值是指左舷推进力向量指向局部坐标*y*轴的负方向。与之类似，STBDThruster函数将右舷推进器的推力设定在500 lb，可让船身向左舷方向转。这里的右舷推力向量指向局部坐标*y*轴的正方向。ZeroThrusters函数只是将两侧推进力归零。以上三个函数都使用一个整数作为参数，来识别测推力所要作用的船。

如前所述，这些函数当使用者按下控制键时被调用。再者，为了在目前的时间间隔中计算力和力矩的改变，这些函数要在StepSimulation函数之前被调用。因为StepSimulation放在NullEvent中，所以这些函数也要放在同一个函数中。以下是程序代码：

```
void        NullEvent(void)
{
.
.
.

      // 处理被按下的控制键
      ZeroThrusters(1);

      if (IsKeyDown(VK_UP))
           IncThrust(1);
```

```
if (IsKeyDown(VK_DOWN))
     DecThrust(1);

if (IsKeyDown(VK_RIGHT))
{
     ZeroThrusters(1);
     PortThruster(1);
}

if (IsKeyDown(VK_LEFT))
{
     ZeroThrusters(1);
     STBDThruster(1);
}

ZeroThrusters(2);
if (IsKeyDown(0x57)) // W键
     IncThrust(2);

if (IsKeyDown(0x5A)) // Z键
     DecThrust(2);

if (IsKeyDown(0x53)) // S键
{
     ZeroThrusters(2);
     PortThruster(2);
}

if (IsKeyDown(0x41)) // A键
{
     ZeroThrusters(2);
     STBDThruster(2);
}

NewTime = timeGetTime();
dt = (float) (NewTime - OldTime)/1000;
OldTime = NewTime;

if (dt > 0.016) dt = 0.016;
if (dt < 0.001f) dt = 0.001f;
StepSimulation(dt);

.
.
.
}
```

在调用StepSimulation函数前，要检查每个控制键是否被按下，并根据按下的键调用适当的函数去调整推进力。

检查某个键是否被按下的 IsKeyDown 函数如下所示：

```
BOOL IsKeyDown(short KeyCode)
{
```

```
    SHORT    retval;

    retval = GetAsyncKeyState(KeyCode);
    if (HIBYTE(retval))
        return TRUE;

    return FALSE;
}
```

因为同时间可能有数个控制键被按下，所以利用此函数能同时处理数个按键，而不像标准的窗口消息处理函数一次只处理一个。

加上了处理飞行控制的程序代码后，模拟程序的物理部分已经相当完整了。目前已完成了模型、积分函数和用户控制的处理。接着要做的是产生应用程序的主窗口并画出要模拟的物体。

# 绘图

设定主窗口和画出可视物体事实上和物理无关。然而为了完整起见，我将简短地介绍在范例中用来设定主窗口，以及利用 Direct3D 绘制模拟物体的程序代码（注 1）。

程序中使用标准的 Windows API 来初始化应用程序：建立并更新主窗口，接着处理窗口消息和使用者输入。由于读者已经熟悉 Windows API 的程序设计，所以这里并不深入解释这些程序代码。

以下是完整的 WinMain 函数的程序代码：

```
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
    MSG msg;
    HANDLE hAccelTable;

    if (!hPrevInstance) {
        // 初始化应用程序
        if (!InitApplication(hInstance)) {
            return (FALSE);
        }
    }

    // 初始化实体代号
    if (!InitInstance(hInstance, nCmdShow)) {
        return (FALSE);
```

注 1：如果读者不熟悉 Direct3D，可以参考 Peter J. Kovach 所著的“The Awesome Power of Direct3D/DirectX”。这本书很实用。

```
}

hAccelTable = LoadAccelerators (hInstance, szAppName);

OldTime = timeGetTime();
NewTime = OldTime;

// 主要的消息循环:
while (1) {

     while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          if (msg.message == WM_QUIT) {
               return msg.wParam;
          }
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     NullEvent();
}

return (msg.wParam);
}
```

WinMain会调用InitInstance和InitApplication函数。之前已介绍过InitInstance，以下仅列出 InitApplication：

```
BOOL InitApplication(HINSTANCE hInstance)
{
     WNDCLASS   wc;
     HWND       hwnd;

      hwnd = FindWindow (szAppName, NULL);
      if (hwnd) {
          if (IsIconic(hwnd)) {
               ShowWindow(hwnd, SW_RESTORE);
          }
          SetForegroundWindow (hwnd);

          return FALSE;
          }

wc.style         = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
wc.lpfnWndProc   = (WNDPROC)WndProc;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance     = hInstance;
wc.hIcon         = NULL;
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);

wc.lpszMenuName = NULL;
wc.lpszClassName = szAppName;
```

```
        return RegisterClass(&wc);
}
```

到目前为止，程序为主窗口建立了一个窗口类并注册了这个类，接着设定并画出了一个640×480的窗口，并建立了一对需要在Direct3D窗口中画出来的对象（这些调用是在InitInstance中），最后执行每一次都会调用NullEvent的主程序循环。

另一个需要实现的API函数是窗口消息处理函数WndProc：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{

    int             wmId, wmEvent;
    BOOL            validmenu = FALSE;
    int             selection =0;
    PAINTSTRUCT     ps;
    HDC             pDC;
    WPARAM          key;

    switch (message) {
        case WM_ACTIVATE:
        if (SUCCEEDED(D3D.Device->QueryInterface(
                    IID_IDirect3DRMWinDevice,
                    (void **) &WinDev)))
        {
                if (FAILED(WinDev->HandleActivate(wParam)))
                    WinDev->Release();
        }

        break;

    case WM_DESTROY:
        CleanUp();
        PostQuitMessage(0);
        break;

    case WM_KEYDOWN:
        key = (int) wParam;

    if (key == 0x31) // 1
        SetCamera1();

    if (key == 0x32) // 2
        SetCamera2();

    if (key == 0x33) // 3
        SetCamera3();

    if (key == 0x34) // 4
        SetCamera4();
```

```
            if (key == 0x35) // 5
                SetCamera5();

            if (key == 0x36) // 6
                SetCamera6();

            break;

        case WM_PAINT:
                pDC = BeginPaint(hTheMainWindow, (LPPAINTSTRUCT) &ps);

                if (SUCCEEDED(D3D.Device->QueryInterface(
                            IID_IDirect3DRMWinDevice,
                            (void **) &WinDev)))
                {
                    if (FAILED(WinDev->HandlePaint(ps.hdc)))
                        WinDev->Release();
                }

                EndPaint(hTheMainWindow, (LPPAINTSTRUCT) &ps);
                return (0);
            break;
            default:
                return (DefWindowProc(hWnd, message, wParam, lParam));
        }
        return (0);
    }
```

当收到 WM_ACTIVATE 消息时，这个函数会取得使用 Direct3D Retained Mode（保留模式）所需的 IDirect3DRMWinDevice 对象。当收到 WM_KEYDOW 消息时，程序在预先设定的 6 个不同角度的摄影机间切换。一号摄影机放置在一号气垫船的驾驶座上，二号摄影机放置在一号气垫船后方外，三号摄影机放置在一号船的正上方并对直向下照。四、五、六号摄影机被放置在相对于二号气垫船上，与一、二、三号摄影机相同的位置。

当收到 WM_PAINT 消息时，程序会将场景画在主窗口中。最后，收到 WM_DESTROY 消息时，程序清除掉所有在 Direct3D 中用到的对象并结束程序。

在介绍所用的 Direct3D 程序代码之前，先介绍另一个版本的 NullEvent 函数：

```
void      NullEvent(void)
{
    Vector  vz, vx;
    char    buf[256];
    char    s[256];
    // 处理被按下的控制键
    ZeroThrusters(1);

    if (IsKeyDown(VK_UP))
        IncThrust(1);
```

```
if (IsKeyDown(VK_DOWN))
     DecThrust(1);

if (IsKeyDown(VK_RIGHT))
{
     ZeroThrusters(1);
     PortThruster(1);
}

if (IsKeyDown(VK_LEFT))
{
     ZeroThrusters(1);
     STBDThruster(1);
}

ZeroThrusters(2);

if (IsKeyDown(0x57)) // W键
     IncThrust(2);

if (IsKeyDown(0x5A)) // Z键
     DecThrust(2);

if (IsKeyDown(0x53)) // S键
{
     ZeroThrusters(2);
     PortThruster(2);
}

if (IsKeyDown(0x41)) // A键
{
     ZeroThrusters(2);
     STBDThruster(2);
}

NewTime = timeGetTime();
dt = (float) (NewTime - OldTime)/1000;
OldTime = NewTime;

if (dt > 0.016) dt = 0.016;
if (dt < 0.001f) dt = 0.001f;

StepSimulation(dt);

if(FrameCounter >= RENDER_FRAME_COUNT)
{
     SetCameraPosition(-Hovercraft1.vPosition.y,
                       Hovercraft1.vPosition.z,
                       Hovercraft1.vPosition.x);

     vz = GetBodyZAxisVector();   // 指向坐标系的上方

     vx = GetBodyXAxisVector(1); // 指向坐标系的前方

     SetCameraOrientation(-vx.y, vx.z, vx.x,
                          -vz.y, vz.z, vz.x);
```

```
            SetCameraPosition2(-Hovercraft2.vPosition.y,
                               Hovercraft2.vPosition.z,
                               Hovercraft2.vPosition.x);

            vz = GetBodyZAxisVector();   // 指向坐标系的上方

            vx = GetBodyXAxisVector(2); // 指向坐标系的前方

            SetCameraOrientation2(-vx.y, vx.z, vx.x,
                                  -vz.y, vz.z, vz.x);

            Render();

            sprintf( buf, "Craft 1 (blue): T= %.0f ; ",
                    Hovercraft1.ThrustForce);
            strcpy(s, buf);
            sprintf( buf, "S= %.0f ",
                    Hovercraft1.fSpeed/1.688); // 除以1.688
                                               // 将单位从ft/s转换成节
            strcat(s, buf);

            sprintf( buf,
                    "      Craft 2 (red): T= %.0f ; ",
                   Hovercraft2.ThrustForce);
            strcat(s, buf);
            sprintf( buf,
                     "S= %.0f ",
                     Hovercraft2.fSpeed/1.688);
                     // 除以1.688，将单位从ft/s转换成节

                strcat(s, buf);

                SetWindowText(hTheMainWindow, s);
            } else
                FrameCounter++;
    }
```

这些出现在StepSimulation函数调用后的程序代码是从未介绍过的，下面看看这些程序代码做了哪些事。

首先，设置画面计数器使负责绘图程序代码的执行次数不会像物理程序代码那样频繁。这个技巧可以在极小时间间隔的物理模拟上使用，让程序不必浪费太多时间去更新窗口的内容。这个程序将 RENDER_FRAME_COUNT 值设定为 300，如下：

```
#define       RENDER_FRAME_COUNT              300
```

这意味此物理模拟程序在执行300 个时间间隔的计算后，才会更新窗口一次。这里使用的300 次，并不表示在其他的模拟程序中也必须使用这个设定值。因为在这个程序中只有两个对象，也没有碰撞测试，所以物理方面的计算是比较少的。以后必须根据不同模拟程序的状况，调整不同的“画面更新率”（frame rate）或“物理更新与窗口更新比率”。

接下来，必须根据气垫船的新位置来更新摄影机的位置。这非常简单，不过要注意在Direct3D中的坐标系与程序中所用的不同。Direct3D使用“左手坐标系”：$x$轴指向右方，$y$轴指向上方，而$z$轴指向屏幕内。所以，Direct3D的$x$轴是程序坐标系$y$轴负的方向，$y$轴是程序坐标的$z$轴，而$z$轴是程序坐标的$x$轴。

除了将摄影机设定在正确的位置之外，还要确定它的方位也是对的。Direct3D使用两个向量：一个定义画面新的$z$轴，另一个定义新的$y$轴。为简化起见，当调整摄影机的方位对准每艘气垫船的方位时，使用两个函数分别取得气垫船的$x$轴和$z$轴向量，用以代表Direct3D中$z$轴和$y$轴的向量。例如，从一号摄影机（一号船的驾驶座位置）的角度看去，当一号船旋转时，我们会希望看到视角如同真正坐在船里面一样地转动。

```
Vector      GetBodyZAxisVector(void)
{
     Vector       v;

     v.x = 0.0f;
     v.y = 0.0f;
     v.z = 1.0f;

     return v;
}

Vector      GetBodyXAxisVector(int craft)
{
     Vector v;

     v.x = 1.0f;
     v.y = 0.0f;
     v.z = 0.0f;

     if(craft == 1)
          return VRotate2D(Hovercraft1.fOrientation, v);
     else
          return VRotate2D(Hovercraft2.fOrientation, v);
}
```

回到NullEvent函数，摄影机定位后，程序调用Render函数开始绘制主窗口的场景。一旦绘制完成，窗口的标题列会显示出各气垫船的统计数据：就是以磅为单位的主推力值，以及以节为单位的速度值。

模拟程序其他所需的程序代码是利用Direct3D绘制气垫船及贴图，与物理并无直接关联，所以这里并没有列出来。读者可以从O'Reilly的网站上下载本范例完整的程序代码。

# 第十三章

# 碰撞反应实现

现在是为范例程序加入些令人惊喜的成分的时候了，本章将介绍如何在第十二章的气垫船范例中加入碰撞反应的实现，让气垫船可以像碰碰车一样互相碰撞并反弹。这在各种游戏中是非常重要的部分，所以了解本章的程序代码也是非常重要的。你可以先翻回第五章复习刚体碰撞反应的基础知识，因为我们将使用第五章讨论的原理和公式，来发展气垫船模拟程序的碰撞反应演算法。

首先将问题简化，本章将气垫船视为粒子（正确地说应该是球体）来实现碰撞反应，并且只考虑线性冲量而忽略转动的效应。这样一来，模拟的结果并不像真正气垫船的反应。然而，这个方法可以应用在其他你可能会感兴趣的问题上，例如撞球的碰撞。

在将线性的运动完成之后，接着会实现角运动的影响。这会让模拟程序更加真实。当一艘气垫船撞上另一艘船时，不只会让它们互相反弹，而且会让它们根据碰撞时的不同情况而旋转。

程序中将加入两个新的函数，并大幅改写前一章所用的StepSimulation函数来实现这个新的功能。虽然不是全新的程序代码，却更复杂，所以本章将一一介绍这些程序代码并解释它们的作用。

这里要再次强调本章的目的是介绍如何实现基本的碰撞“反应”，而不是另一个不同的主题：碰撞“侦测”。虽然碰撞侦测对于任何一个碰撞反应演算法都是必需的，它却是一个比物理问题还要复杂的几何计算问题。因此本章将重点放在物理上的碰撞反应，只会视情况实现必需的碰撞侦测。如果读者有兴趣深入了解碰撞侦测，可以参考某些介绍几何计算技术的文献来获得更多的资料。

# 线性碰撞反应

本节示范在将气垫船视为粒子（或球体）的情况下，如何实现简单的碰撞反应。模拟程序中只会实现最精简的碰撞侦测。然而，不论碰撞侦测程序有多精密，它至少要收集某些特定的数据，才能让物理的碰撞反应程序正确运作。

要在前一章的气垫船范例中加入简单的碰撞反应，必须修改StepSimulation并加入两个新函数：CheckForCollision和ApplyImpulse。看看这两个函数。

在介绍CheckForCollision函数之前，先解释碰撞侦测函数该做些什么。第一，必须判断两艘船之间是否已经产生碰撞；第二，判断某一艘船是否正在穿透另一艘船；第三，如果两艘船正在发生碰撞，则要决定两艘船之间碰撞的法向量及相对速度。

要测定两艘船是否碰撞，需考察两个因素：

- 对象间的距离是否小于允许值，以判断是否正在碰撞接触中。
- 相对于法线，速度的方向是否向着物体。

如果对象未相互靠近，就意味着它们并不会碰撞。如果两物体距离小于接触允许值，就视为正在碰撞中，而如果它们因接触重叠而移动到对方内部，就视为正在穿透中。以上介绍的情形请参考图13-1。当碰撞侦测程序发现对象之间已经够靠近而构成碰撞的条件时，必须再检查物体间的法向速度，以判断两物体是靠近中或是远离中。总而言之，碰撞发生的条件是物体正在接触且接触点朝对方移动。

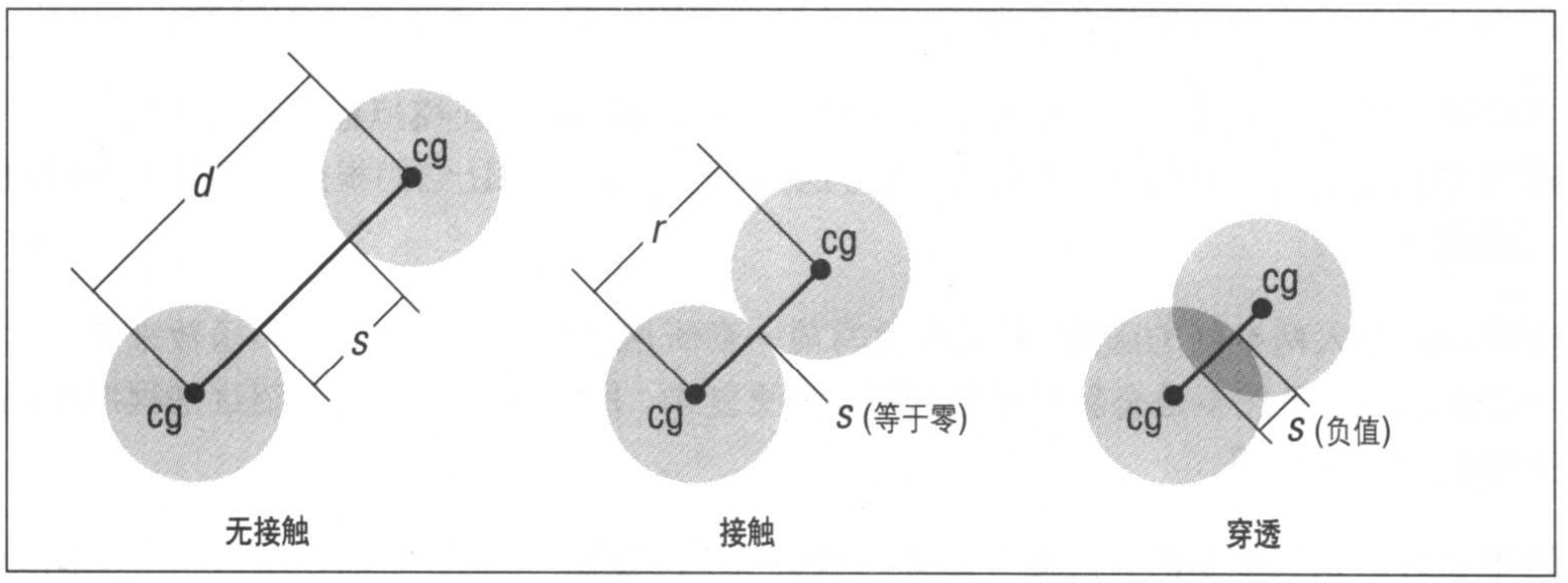



图 13-1：碰撞的用语

检查是否穿透是很重要的，如果模拟程序中的对象相互重叠，结果会看起来很不真实 —— 就像前一章一样，会有一艘气垫船开进另一艘的情形发生。如果发生穿透的情

形，要将模拟程序倒退一个时间间隔，并减少时间间隔再计算一次，直到没有穿透的情形发生或刚好要碰撞的情形发生为止。

另外还需要用法线速度向量测定碰撞时的冲量，以便用来计算碰撞时的反应。本范例要判断法向量是很简单的。因为我们使用粒子或球体来代表要模拟的物体，所以碰撞法线沿着两碰撞物体重心连线，这就是第五章提过的中心撞击（central impact）。这是将气垫船视为粒子或球体会发生的状况。

以下是模拟程序中检查碰撞的函数：

```
int     CheckForCollision (pRigidBody2D body1, pRigidBody2D body2)
{
     Vector    d;
     float     r;
     int       retval = 0;
     float     s;
     Vector    v1, v2;
     float     Vrn;

     r = body1->fLength/2 + body2->fLength/2;
     d = body1->vPosition - body2->vPosition;
     s = d.Magnitude() - r;

     d.Normalize();
     vCollisionNormal = d;

     v1 = body1->vVelocity;
     v2 = body2->vVelocity;
     vRelativeVelocity = v1 - v2;

     Vrn = vRelativeVelocity * vCollisionNormal;
     if((fabs(s) <= ctol) && (Vrn < 0.0))
     {
          retval = 1;  // 发生碰撞
     } else if(s < -ctol)
     {
          retval = -1; // 互相穿透
     } else
          retval = 0;  // 无碰撞
     return retval;
}
```

这个函数使用简单的圆作为边界来测定气垫船是否发生碰撞。程序一开始先计算距离r，它代表气垫船之间接触时的绝对最小距离。并假设这个圆形边界的直径等于气垫船的长度。

接着，计算当此函数被调用时气垫船之间的距离，并存在变量$d$中。因为假设气垫船是粒子，所以要求出它们之间的距离$d$只需要计算两船的重心的距离即可。当使用向量表示时，这个值就是其中一艘船的位置向量减去另一艘船的位置向量。

知道了$d$和$r$之后，还要计算两艘船的圆形边界分隔距离$s$。求出此距离后，函数将向量$d$正规化。因为向量d是沿着两艘船重心的连线，正规化后会得到碰撞反应计算所需要的法向量。这个碰撞的法向量存储在全局变量vCollisionNormal中。

在计算法向量之后，函数接着计算气垫船之间的相对速度。以向量形式表示，只要计算两艘船速度向量的差；但请注意，速度向量需使用全局坐标而不是局部坐标。因为真正用来判定物体是否碰撞的是相对法线速度，所以函数中求的是相对速度与碰撞法向量的内积，并存在变量Vrn中。

到目前为止，所需的计算已经完成。剩下的只是要判断对象是否已经发生碰撞，或是互相穿透。

先检查对象是否碰撞。可由比较气垫船之间距离$s$的绝对值和距离的允许值ctol得知。如果距离$s$的绝对值小于ctol，则表示碰撞正在发生。第二个条件是检查相对法线速度是否为负值，若是则表示碰撞点正朝着气垫船靠近。如果确定碰撞发生，函数返回1代表必须对碰撞做出反应。

如果气垫船没有碰撞发生，再检查两艘船是否太靠近而发生互相穿透的情况。这种情况就是$s$小于-ctol，表示船正在穿透中而函数返回-1。如果船不碰撞也不穿透，函数返回0而并不需要做任何事。

再来看看另一个新的函数ApplyImpulse：

```
void      ApplyImpulse(pRigidBody2D body1, pRigidBody2D body2)
{
     float j;

     j =   (-(1+fCr) * (vRelativeVelocity*vCollisionNormal)) /
           (1/body1->fMass + 1/body2->fMass) ;
           body1->vVelocity += (j * vCollisionNormal) / body1->fMass;
           body2->vVelocity -= (j * vCollisionNormal) / body2->fMass;
}
```

这是一个简单却重要的碰撞反应函数。此函数使用第五章提到的公式计算线性碰撞冲量(气垫船的相对法线速度、质量和恢复系数的函数)。接着，将此冲量套用在两艘气垫船上，计算碰撞回应并改变船的速度。请注意，冲量作用于一艘船时，该冲量的负值会作用在另一艘船上。

完成两个新函数后，接着要修改StepSimulation函数，来处理模拟进行时的碰撞侦测和反应。以下是新的StepSimulation函数：

```
void      StepSimulation(float dt)
{
```

```
float     dtime = dt;
bool      tryAgain = true;
int           check=0;
RigidBody2D     craft1Copy, craft2Copy;
bool      didPen = false;
int           count = 0;

while(tryAgain && dtime > tol)
{
     tryAgain = false;
     memcpy(&craft1Copy, &Hovercraft1, sizeof(RigidBody2D));
     memcpy(&craft2Copy, &Hovercraft2, sizeof(RigidBody2D));

     UpdateBody(&craft1Copy, dtime);
     UpdateBody(&craft2Copy, dtime);

     CollisionBody1 = 0;
     CollisionBody2 = 0;
     check = CheckForCollision(&craft1Copy, &craft2Copy);

     if(check == PENETRATING)
     {
          dtime = dtime/2;
          tryAgain = true;
          didPen = true;
     } else if(check == COLLISION)
          {
               if(CollisionBody1 != 0 && CollisionBody2 != 0)
                    ApplyImpulse(CollisionBody1, CollisionBody2);
          }
     }

     if(!didPen)
     {
          memcpy(&Hovercraft1, &craft1Copy, sizeof(RigidBody2D));
          memcpy(&Hovercraft2, &craft2Copy, sizeof(RigidBody2D));
     }
}
```

显然，这个版本比前一个版本要复杂多了。原因是：在一个时间间隔中，一艘气垫船可能会移动得太多而导致与另一艘船重叠。结果看起来会不真实而且不吸引人，所以要想办法避免这种情况发生。

这个函数一开始先进入 while 循环：

```
while(tryAgain && dtime > tol)
{
     .
     .
     .
}
```

当穿透发生在初始的时间间隔时，此循环用来倒转模拟程序。情况是：函数先试着去更新气垫船的状态并检查是否发生了碰撞。如果有碰撞发生，则计算作用于船上的冲量。但是如果发生了穿透，就表示使用的时间间隔太大，必须重新计算一次。所以当这种情况发生时，变量tryAgain会被设为true，并将时间间隔减半再计算一次。只要仍有穿透发生，函数会一直执行循环直到时间间隔够小为止。循环的目的是找出小于或等于dt时间间隔的最大值，以达到避免穿透发生。我们想要的只是发生碰撞或不发生碰撞。

接着看看while循环里面发生了什么事。函数先将变量tryAgain设成false，乐观地假设没有穿透发生，再将气垫船的状态备份，保存上次成功调用StepSimulation的结果。

接着，如往常一样的调用UpdateBody函数。然后再调用碰撞侦测函数CheckForCollision，以判断气垫船间是否发生了碰撞或穿透。如果发生了穿透，就会将变量tryAgain设为true，并将dtime减半，didPen设为true，再运行一次while循环。didPen是记录穿透是否发生的变量。

如果发生了碰撞，函数会计算适当的冲量来处理碰撞反应：

```
if(CollisionBody1 != 0 && CollisionBody2 != 0)
    ApplyImpulse(CollisionBody1, CollisionBody2);
```

在结束while循环后，会将气垫船更新后的状态存储起来，而StepSimulation也会结束。

最后一点需要加上的程序代码是一些全局变量和定义：

```
#define     LINEARDRAGCOEFFICIENT 0.25f
#define     COEFFICIENTOFRESTITUTION    0.5f
#define     COLLISIONTOLERANCE          2.0f

Vector      vCollisionNormal;    // 碰撞法向量
Vector      vRelativeVelocity;   // 全局坐标中，物体碰撞时的相对速度
float       fCr = COEFFICIENTOFRESTITUTION;    // 恢复系数
float       const  ctol = COLLISIONTOLERANCE; // 碰撞（距离）的允许值
```

到目前为止惟一还没有介绍的是在函数ApplyImpulse中的fCr，称为恢复系数。这里将它设为0.5，意思是指物体的碰撞介于完全弹性与完全非弹性中间（如果读者对此处仍不熟悉，可参考第五章）。恢复系数也是可以用来调整气垫船行为的参数之一。

当谈到调整时，就必须顺便提到可以用来计算气垫船上阻力的线性阻力系数。虽然这个系数是用来模拟流体动阻力，但在数字的稳定度上此系数也很重要。模拟程序中需要某些限制，否则积分器里的计算将会过于庞大，也就是计算运动方程的结果会远离理论值。

这会使模拟程序变得不真实而且不可预测。在第十七章介绍柔体的模拟时，这种限制的重要性会相当明显。

在实现基本的碰撞反应方面已经介绍得够多了。执行这个程序时，可以驾驶一艘气垫船撞向另一艘并且反弹。也可试着调整气垫船的质量和恢复系数，看看在不同的质量或不同的恢复系数下会发生什么事。

也许这个范例中的碰撞反应有一点奇怪。这是因为到目前为止的碰撞反应演算法，都将气垫船假设为圆形而不是多边形。这个方法对于圆形的物体如撞球来说是没什么问题，不过对于非圆形的刚体就显得很不真实了。对于非圆形的刚体必须考虑角运动，这在下一节会介绍。

# 角运动

加入角运动将产生更真实的刚体（气垫船）碰撞反应。为了达到此目标，必须对ApplyImpulse与CheckForCollision做若干修正，而StepSimulation函数则不必更改。其中改变最大的是CheckForCollision函数，所以这里将优先讨论。

新版的CheckForCollision将不只是做简单的圆形边界检查。这里的气垫船将会以有4个边和4个顶点的多边形代表，而对象接触的类型则会有“顶点对顶点”与“顶点对边”两种（参考图13-2）（注1）。

除了前一节所讨论的工作外，新版的CheckForCollision还必须找出气垫船之间正确的接触点。这是与前一版之间重要的不同点。因为要计算角速度的影响并且套用在接触点上的冲量，所以必须要知道接触点的位置。在前一节中，因为将物体视为球体，所以接触点的法向量总是通过对象的重心，但是在本节中就不同了。

在这种情况下要寻找法向量就有点挑战性了。这里有两种可能发生的情况。在顶点对边的碰撞中，法向量总是与被碰撞的边垂直。然而在顶点对顶点的碰撞中，法向量却有一种以上的定义。所以这里将这种情况下的法向量设定为平行于两艘船重心的连线。

以上这些考虑让CheckForCollisions函数比前一节更复杂。以下就是新的程序代码：

```
int     CheckForCollision(pRigidBody2D body1, pRigidBody2D body2)
{
     Vector    d;
     float     r;
     int       retval = 0;
```

注1：　请注意此函数无法处理多重的接触点。

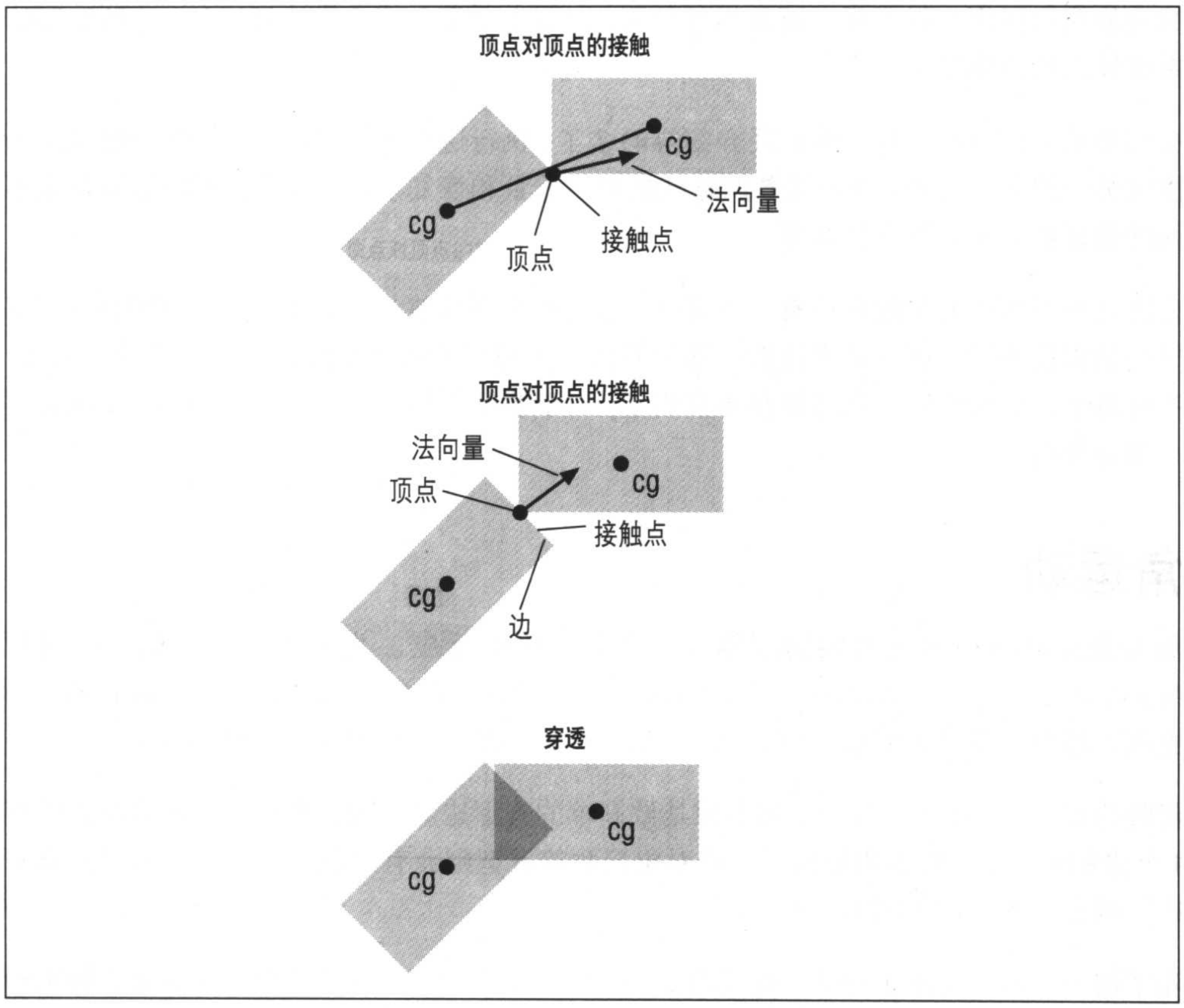



图 13-2：碰撞的类型

```
float     s;
Vector    vList1[4], vList2[4];
float     wd, lg;
int       i, j;
bool      haveNodeNode = false;
bool      interpenetrating = false;
bool      haveNodeEdge = false;
Vector    v1, v2, u;
Vector    edge, p, proj;
float     dist, dot;
float     Vrn;

// 先检查圆形边界是否产生碰撞
r = body1->fLength/2 + body2->fLength/2;
d = body1->vPosition - body2->vPosition;
s = d.Magnitude() - r;

if(s <= ctol)
{  // 也许产生了碰撞，再进一步检查
```

```
// 为两艘船建立顶点串列
wd = body1->fWidth;
lg = body1->fLength;
vList1[0].y = wd/2;  vList1[0].x = lg/2;
vList1[1].y = -wd/2; vList1[1].x = lg/2;
vList1[2].y = -wd/2; vList1[2].x = -lg/2;
vList1[3].y = wd/2;  vList1[3].x = -lg/2;

for(i=0; i<4; i++)
{
     VRotate2D(body1->fOrientation, vList1[i]);
     vList1[i] = vList1[i] + body1->vPosition;
}
wd = body2->fWidth;
lg = body2->fLength;
vList2[0].y = wd/2;  vList2[0].x = lg/2;
vList2[1].y = -wd/2; vList2[1].x = lg/2;
vList2[2].y = -wd/2; vList2[2].x = -lg/2;
vList2[3].y = wd/2;  vList2[3].x = -lg/2;

for(i=0; i<4; i++)
{
     VRotate2D(body2->fOrientation, vList2[i]);
     vList2[i] = vList2[i] + body2->vPosition;
}

// 检查是否为顶点对顶点的碰撞
for(i=0; i<4 && !haveNodeNode; i++)
{
    for(j=0; j<4 && !haveNodeNode; j++)
    {
         vCollisionPoint = vList1[i];
         body1->vCollisionPoint = vCollisionPoint -
                                  body1->vPosition;

         body2->vCollisionPoint = vCollisionPoint -
                                  body2->vPosition;

         vCollisionNormal = body1->vPosition -
                            body2->vPosition;

         vCollisionNormal.Normalize();

         v1 = body1->vVelocityBody +
              (body1->vAngularVelocity^body1->vCollisionPoint);

         v2 = body2->vVelocityBody +
              (body2->vAngularVelocity^body2->vCollisionPoint);

         v1 = VRotate2D(body1->fOrientation, v1);
         v2 = VRotate2D(body2->fOrientation, v2);

         vRelativeVelocity = v1 - v2;
         Vrn = vRelativeVelocity * vCollisionNormal;
```

```
        if( ArePointsEqual(vList1[i], vList2[j]) && (Vrn < 0.0))
            haveNodeNode = true;

        }
}

// 检查是否为顶点对边的碰撞
if(!haveNodeNode)
{
    for(i=0; i<4 && !haveNodeEdge; i++)
    {
        for(j=0; j<3 && !haveNodeEdge; j++)
        {
            if(j==3)
                edge = vList2[0] - vList2[j];
            else
                edge = vList2[j+1] - vList2[j];
            u = edge;
            u.Normalize();

            p = vList1[i] - vList2[j];
            proj = (p * u) * u;

            d = p^u;
            dist = d.Magnitude();

            vCollisionPoint = vList1[i];
            body1->vCollisionPoint = vCollisionPoint -
                                     body1->vPosition;

            body2->vCollisionPoint = vCollisionPoint -
                                     body2->vPosition;

            vCollisionNormal = ((u^p)^u);
            vCollisionNormal.Normalize();

            v1 = body1->vVelocityBody +
                 (body1->vAngularVelocity ^
                 body1->vCollisionPoint);

            v2 = body2->vVelocityBody +
                 (body2->vAngularVelocity
                 body2->vCollisionPoint);

            v1 = VRotate2D(body1->fOrientation, v1);
            v2 = VRotate2D(body2->fOrientation, v2);

            vRelativeVelocity = (v1 - v2);
            Vrn = vRelativeVelocity * vCollisionNormal;

            if( (proj.Magnitude() > 0.0f) &&
                 (proj.Magnitude() <= edge.Magnitude()) &&
                 (dist <= ctol) &&
                 (Vrn < 0.0) )
                  haveNodeEdge = true;
```

```
                    }
               }
          }

          // 检查是否为穿透的情况
          if(!haveNodeNode && !haveNodeEdge)
          {
               for(i=0; i<4 && !interpenetrating; i++)
               {
                    for(j=0; j<4 && !interpenetrating; j++)
                    {
                         if(j==3)
                              edge = vList2[0] - vList2[j];
                         else
                              edge = vList2[j+1] - vList2[j];
                         p = vList1[i] - vList2[j];
                         dot = p * edge;
                         if(dot < 0)
                         {
                              interpenetrating = true;
                         }
                    }
               }
          }

          if(interpenetrating)
                    {
                         retval = -1;
                    } else if(haveNodeNode || haveNodeEdge)
                    {
                         retval = 1;
               } else
                         retval = 0;

          } else
          {
                    retval = 0;
          }

          return retval;
     }
```

CheckForCollision先使用圆形边界做快速的检查，以决定是否会产生碰撞。如果没有，函数就返回0。这里实现中所使用的圆形边界检查与前一版的相同：

```
r = body1->fLength/2 + body2->fLength/2;
d = body1->vPosition - body2->vPosition;
s = d.Magnitude() - r;

if(s <= ctol)
{
     .
     .
     .
```

```
} else
      retval = 0;
}
```

如果圆形边界检查发现有碰撞的可能，CheckForCollision就为两艘船设定两个多边形，并以顶点阵列表示。

```
wd = body1->fWidth;
lg = body1->fLength;
vList1[0].y = wd/2;          vList1[0].x = lg/2;
vList1[1].y = -wd/2;         vList1[1].x = lg/2;
vList1[2].y = -wd/2;         vList1[2].x = -lg/2;
vList1[3].y = wd/2;          vList1[3].x = -lg/2;

for(i=0; i<4; i++)
{
     VRotate2D(body1->fOrientation, vList1[i]);
     vList1[i] = vList1[i] + body1->vPosition;
}

wd = body2->fWidth;
lg = body2->fLength;
vList2[0].y = wd/2;          vList2[0].x = lg/2;
vList2[1].y = -wd/2;         vList2[1].x = lg/2;
vList2[2].y = -wd/2;         vList2[2].x = -lg/2;
vList2[3].y = wd/2;          vList2[3].x = -lg/2;
for(i=0; i<4; i++)
{
     VRotate2D(body2->fOrientation, vList2[i]);
     vList2[i] = vList2[i] + body2->vPosition;
}
```

顶点阵列是根据气垫船的长与宽而初始化的，并且设定在不能旋转的局部坐标系中。然后这些顶点的坐标依照船首的方位而被旋转至正确值。之后船的位置被加到顶点阵列中将局部坐标转换成全局坐标。

首先检查顶点对顶点的碰撞时，函数会逐一检查阵列中所有的顶点，并与其他阵列中的顶点进行比较以判定这些顶点是否发生碰撞：

```
// 检查是否为顶点对边的碰撞
for(i=0; i<4 && !haveNodeNode; i++)
{
     for(j=0; j<4 && !haveNodeNode; j++)
     {
          vCollisionPoint = vList1[i];
          body1->vCollisionPoint = vCollisionPoint -
                                   body1->vPosition;

     body2->vCollisionPoint = vCollisionPoint -
                              body2->vPosition;
```

```
vCollisionNormal = body1->vPosition -
                   body2->vPosition;

vCollisionNormal.Normalize();

v1 = body1->vVelocityBody +
     (body1->vAngularVelocity^body1->vCollisionPoint);

v2 = body2->vVelocityBody +
     (body2->vAngularVelocity^body2->vCollisionPoint);

v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);

vRelativeVelocity = v1 - v2;
Vrn = vRelativeVelocity * vCollisionNormal;

if( ArePointsEqual(vList1[i], vList2[j]) && (Vrn < 0.0) )
     haveNodeNode = true;
}
}
```

程序调用新的函数ArePointsEqual，来比较顶点：

```
if( ArePointsEqual(vList1[i], vList2[j]) && (Vrn < 0.0) )
     haveNodeNode = true;
```

ArePointsEqual函数只是检查顶点与其他点是否在特定的距离内。

```
bool    ArePointsEqual(Vector p1, Vector p2)
{
     // 如果每个顶点的坐标差都小于ctol，则两点为同一点
      if( (fabs(p1.x - p2.x) <= 0.1) &&
          (fabs(p1.y - p2.y) <= 0.1) &&
          (fabs(p1.z - p2.z) <= 0.1) )
          return true;
      else
          return false;
}
```

在顶点对顶点检查的嵌套for循环结构中，执行许多重要的计算，以决定碰撞反应所需要碰撞法向量与相对速度。

第一步先计算发生碰撞的顶点，以得到发生碰撞顶点的坐标。这些顶点是全局坐标，所以必须先转换成局部坐标才能用来处理碰撞反应。以下是这部分的程序代码：

```
vCollisionPoint = vList1[i];
body1->vCollisionPoint = vCollisionPoint -
                         body1->vPosition;

body2->vCollisionPoint = vCollisionPoint -
                         body2->vPosition;
```

第二步的计算决定碰撞法向量。在顶点对顶点的碰撞中，假设碰撞法向量沿着连接两艘船重心的连线。这里的计算与CheckForCollision前一版本相同：

```
vCollisionNormal = body1->vPosition -
                   body2->vPosition;

vCollisionNormal.Normalize();
```

第三步也就是最后一步，用来计算两个碰撞顶点之间的相对速度。这与前一版的函数有很大的不同，因为每艘船碰撞顶点的速度是船的线性速度与角速度的函数：

```
v1 = body1->vVelocityBody +
     (body1->vAngularVelocity^body1->vCollisionPoint);

v2 = body2->vVelocityBody +
     (body2->vAngularVelocity^body2->vCollisionPoint);

v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);

vRelativeVelocity = v1 - v2;
Vrn = vRelativeVelocity * vCollisionNormal;
```

这里的v1和v2是两艘船之间碰撞点的相对速度，虽然是以局部坐标表示的但随后将被转换成全局坐标。知道相对速度vRelativeVelocity后，就可以由相对速度与碰撞法向量的内积求得相对角速度Vrn。

如果没有顶点对顶点的碰撞发生，CheckForCollision将继续检查顶点对边的碰撞是否发生。

```
// 检查顶点对边的碰撞
if(!haveNodeNode)
{
      for(i=0; i<4 && !haveNodeEdge; i++)
      {
            for(j=0; j<3 && !haveNodeEdge; j++)
            {
                  if(j==3)
                        edge = vList2[0] - vList2[j];
                  else
                        edge = vList2[j+1] - vList2[j];
                  u = edge;
                  u.Normalize();

                  p = vList1[i] - vList2[j];
                  proj = (p * u) * u;

                  d = p^u;
                  dist = d.Magnitude();
```

```
                vCollisionPoint = vList1[i];
                body1->vCollisionPoint = vCollisionPoint -
                                         body1->vPosition;

                body2->vCollisionPoint = vCollisionPoint -
                                         body2->vPosition;

                vCollisionNormal = ((u^p)^u);
                vCollisionNormal.Normalize();

                v1 = body1->vVelocityBody +
                     (body1->vAngularVelocity ^
                      body1->vCollisionPoint);

                v2 = body2->vVelocityBody +
                     (body2->vAngularVelocity ^
                      body2->vCollisionPoint);

                v1 = VRotate2D(body1->fOrientation, v1);
                v2 = VRotate2D(body2->fOrientation, v2);

                vRelativeVelocity = (v1 - v2);
                Vrn = vRelativeVelocity * vCollisionNormal;

                if( (proj.Magnitude() > 0.0f) &&
                     (proj.Magnitude() <= edge.Magnitude()) &&
                     (dist <= ctol) &&
                     (Vrn < 0.0) )
                      haveNodeEdge = true;
                }
            }
        }
```

这里的嵌套for循环检查阵列中的每个点是否与另一个阵列中顶点连成的边接触。找出发生碰撞的边，将它存储到另一个变量并正规化该变量后，就会得到这个边的单位向量。

```
if(j==3)
     edge = vList2[0] - vList2[j];
else
     edge = vList2[j+1] - vList2[j];
u = edge;
u.Normalize();
```

变量u代表该单位向量，在之后的计算中会用到它。接着找出发生碰撞的顶点到边的投影点，以及顶点与边的最小距离。

```
p = vList1[i] - vList2[j];
proj = (p * u) * u;

d = p^u;
dist = d.Magnitude();
```

变量 p 是由边上第一个顶点到检查的顶点的向量，而 proj 是边上第一个顶点沿着边指到投影点的向量。dist 是从顶点到边的最小距离。图 13-3 说明了其几何关系。

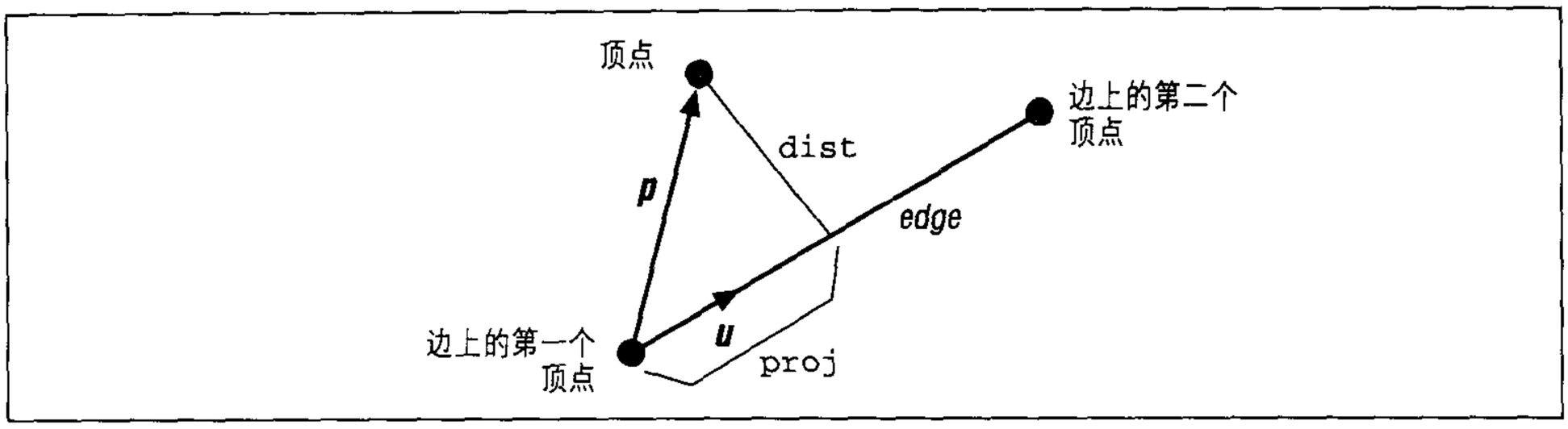



图 13-3：顶点与边的碰撞检查

如果发生碰撞，则碰撞点的全局坐标位置与检查的顶点坐标相同。顶点位置必须先从局部坐标转换成全局坐标：

```
vCollisionPoint = vList1[i];
body1->vCollisionPoint = vCollisionPoint -
                         body1->vPosition;

body2->vCollisionPoint = vCollisionPoint -
                         body2->vPosition;
```

因为在这种碰撞之中，碰撞法向量与被碰撞的边垂直，所以可先求 u 和 p 的外积，再取该结果与 u 的外积求得法向量。

```
vCollisionNormal = ((u^p)^u);
vCollisionNormal.Normalize();
```

这些计算得到垂直于边并且与向量 u、p 共平面的单位向量。

接着，每艘船与撞击点的相对速度就可以知道了，如同顶点对顶点撞击的检查一样：

```
v1 = body1->vVelocityBody +
     (body1->vAngularVelocity ^
      body1->vCollisionPoint);

v2 = body2->vVelocityBody +
     (body2->vAngularVelocity ^
      body2->vCollisionPoint);

v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);

vRelativeVelocity = (v1 - v2);
Vrn = vRelativeVelocity * vCollisionNormal;
```

在决定一个顶点是否与边撞击时，还要检查从顶点到边的距离是否在撞击允许值内。还要确定这个顶点是否真正投影在这个边上（也就是说，不能投影在边的两个顶点之外）。此外，还要确定指向撞击点的相对法线速度是否向对方移动。以下是检查这些条件的程序代码：

```
if( (proj.Magnitude() > 0.0f) &&
    (proj.Magnitude() <= edge.Magnitude()) &&
    (dist <= ctol) &&
    (Vrn < 0.0) )
    haveNodeEdge = true;
```

在CheckForCollision检查完顶点对顶点及顶点对边的碰撞后，继续检查是否穿透：

```
if(!haveNodeNode && !haveNodeEdge)
{
     for(i=0; i<4 && !interpenetrating; i++)
     {
          for(j=0; j<4 && !interpenetrating; j++)
          {
               if(j==3)
                    edge = vList2[0] - vList2[j];
               else
                    edge = vList2[j+1] - vList2[j];

               p = vList1[i] - vList2[j];
               dot = p * edge;
               if(dot < 0)
               {
                    interpenetrating = true;
               }
          }
     }
}
```

这是个典型的点与多边形的检查，并使用外积来决定一个多边形的所有点是否在另一个多边形内。在检查过后，函数会返回适当的值。0表示没有碰撞或穿透，1表示发生碰撞，-1表示发生穿透。

在CheckForCollision结束后，请注意我们也应该修改ApplyImpulse来处理角运动，即也要在冲量公式中加入处理角运动的程序（参考第五章），而且除了将冲量套用到线性速度外，也得套用到角速度。以下是新的ApplyImpulse函数：

```
void      ApplyImpulse(pRigidBody2D body1, pRigidBody2D body2)
{
     float j;

     j = (-(1+fCr) * (vRelativeVelocity*vCollisionNormal)) /
          ( (1/body1->fMass + 1/body2->fMass) +
          (vCollisionNormal * (((body1->vCollisionPoint ^
```

```
            vCollisionNormal)/body1->fInertia)^body1->vCollisionPoint)) +
           (vCollisionNormal * (((body2->vCollisionPoint ^
            vCollisionNormal)/body2->fInertia)^body2->vCollisionPoint))
           );

    body1->vVelocity += (j * vCollisionNormal) / body1->fMass;
    body1->vAngularVelocity += (body1->vCollisionPoint ^
                                (j * vCollisionNormal)) /
                               body1->fInertia;

            body2->vVelocity -= (j * vCollisionNormal) / body2->fMass;
            body2->vAngularVelocity -= (body2->vCollisionPoint ^
                                        (j * vCollisionNormal)) /
                                       body2->fInertia;
    }
```

请注意，发生碰撞时气垫船所受的冲量同样也作用在另一艘船上，只是方向相反而已。

至此完成了新版的气垫船模拟程序。执行这个程序时，你会发现当碰撞产生时，气垫船会反弹并且根据撞击时情况的不同而发生不同的旋转。新版的程序比前一个只实现线性反应的模拟程序要真实多了。程序中也可以改变质量与恢复系数的大小来试验这些参数对气垫船碰撞反应的影响。

# 第十四章

# 刚体的转动

在下一章介绍如何实现3D的模拟器之前，必须先讨论在3D的环境中如何表示方位或旋转的问题。在2D空间中，要表示刚体的方位很简单，只需要用一个标量就可以表示对单一轴的旋转。然而在3D的世界中，刚体可能会绕着三个主要的坐标轴旋转。此外，刚体也可能会绕着任意轴旋转，未必是这三个轴之一。

在2D的环境中，刚体只有一个旋转自由度，但是在3D空间中，刚体却有三个旋转自由度。这表示在3D的环境中，必须用三个标量来表达物体的旋转。事实上，这是最小的需求。之前也介绍过如何在3D环境中，使用角度的集合来表达物体的方位，也就是在第七章讨论过的三个欧拉角：滚转、俯仰和偏转。

这三个角度——滚转、俯仰和偏转——是很直观而容易想像的。例如，飞机的机鼻上下俯仰，机身向左右翻滚（或倾斜），机首向左右偏转。然而，在刚体模拟中使用欧拉角会发生一些困难。问题发生在当俯角或仰角达到正/负90°（π/2）时，翻滚和偏转的角度会含糊不清。更严重的是，以欧拉角表示的角运动方程其中有俯仰角的余弦作为分母，意思是说当俯仰角为正/负90°时，方程就变得不寻常了（因为有除以0的项）。如果这种情形发生在模拟程序中会使结果不可预测。为了应用欧拉角的问题，必须使用其他方法来记录模拟程序的方位变化。本章会讨论两种方法：旋转矩阵法与四元数法。

事实上几乎每一本介绍计算机图形学的书都会有讨论旋转矩阵法的章节。虽然很少有书会讨论四元数法，但是四元数法与旋转矩阵法有着相同的背景，如旋转3D的点、对象、场景、视点等。然而在模拟程序中，必须将比四元数法与旋转矩阵法更多的技巧用在想要呈现的结果上。更具体地说，必须记录物体在空间中的方位和方位随时间的改变量。所以本章剩下的篇幅将讨论四元数法与旋转矩阵法。这里尽可能简明地介绍而不去讨论

证明和推导的过程。如果你对于推导和证明的过程有兴趣，可以在参考文献中找到相关的资料。

## 旋转矩阵法

旋转矩阵是一个 3 × 3 的矩阵，将这个矩阵与点或向量相乘，会得到这个点绕着某个坐标轴旋转后的结果，也就是一组新的坐标。可以使用旋转矩阵计算绕着坐标系的轴线旋转的结果，也可以将某个坐标系上的点转换至另一个坐标系（该坐标系转动后的结果）。

用来旋转向量的旋转矩阵典型的写法是：如果 $v$ 是向量而 $R$ 是旋转矩阵，$v'$ 就是 $v$ 根据以下公式旋转 $R$ 后的向量：

$$v' = Rv$$

多重旋转矩阵是指多个连续的旋转可用矩阵相乘合并成一个旋转矩阵。如果旋转矩阵以固定的全局坐标表示，则可以这样合并：

$$R_c = R_2 R_1$$

这里的 $R_c$ 是合并后的旋转矩阵，先由矩阵 $R_1$ 旋转后再由矩阵 $R_2$ 旋转。如果旋转矩阵是以旋转的局部坐标表示的，则它们是以相反的顺序合并的：

$$R_c = R_1 R_2$$

这里并不会证明此关系式。但是根据旋转矩阵的定义而得到不同结果的原因，是由于坐标轴仍然固定，所以由固定坐标定义的旋转矩阵并不会被旋转影响。相反，如果旋转矩阵是相对于会随着旋转矩阵而旋转的坐标轴而定义的，则在第一个之后的旋转矩阵都会被影响。因为这些矩阵起初的定义是相对于原始的坐标系（与第一个旋转矩阵相乘之前）。这表示在与后续的旋转矩阵相乘前，必须先计算新坐标系对旋转的影响。换句话说，在相乘前要先以 $R_1$ 旋转 $R_2$ 以得到新的 $R_2$。这就是当定义会旋转的坐标系时，为何要倒转矩阵相乘顺序的原因。

以上说明了如何合并旋转矩阵以反映绕着三个坐标轴旋转的结果。以下是这些矩阵。

图 14-1 是一个右手坐标系并指明绕着坐标轴旋转的正方向。

接着考虑图 14-2 中的点绕着 $z$ 轴旋转 $\theta$ 度。

在旋转前这个点的坐标是 $(x, y, z)$，而旋转后坐标是 $(x_r, x_r, z_r)$。而旋转后的坐标与原始坐标和旋转的角度有以下的关系：

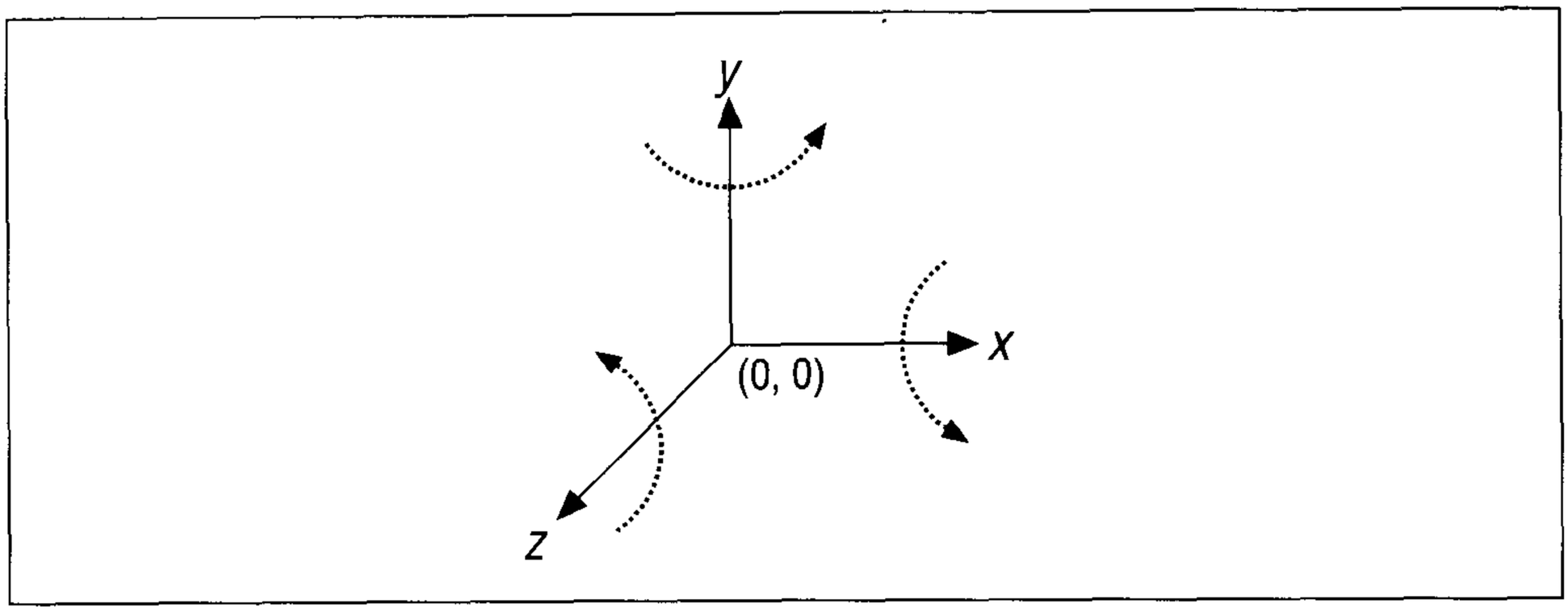



图 14-1：右手坐标系

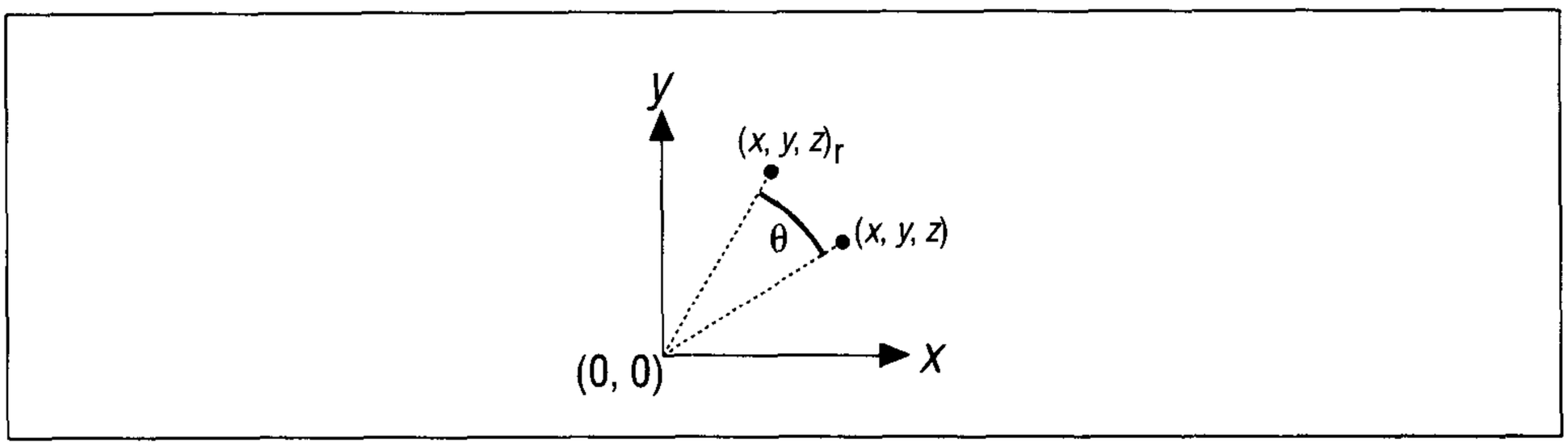



图 14-2：绕 z 轴的旋转

$$x_r = x\cos\theta + x\sin\theta$$
$$x_r = -x\sin\theta + x\cos\theta$$
$$z_r = z$$

请注意因为点绕着 z 轴旋转，所以 z 坐标不会改变。以向量矩阵表示是 $\boldsymbol{v}' = \boldsymbol{R}\boldsymbol{v}$，令 $\boldsymbol{v} = [x\ y\ z]$，而 $\boldsymbol{R}$ 是以下的矩阵：

$$\begin{vmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

这里的 $\boldsymbol{v}'$ 是新的旋转后的向量 $\boldsymbol{v}' = [x_r\ x_r\ z_r]$。

绕着 x 轴与 y 轴的旋转与绕着 z 轴的旋转类似。然而在那些情况下，绕着 x 轴或 y 轴旋转后的 x 坐标或 y 坐标是不会变的。如果将绕着三个坐标轴旋转的矩阵分开看，会产生与绕着 z 轴的旋转矩阵类似的矩阵。

绕着 $x$ 轴旋转的矩阵为：

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{vmatrix}$$

而绕着 $y$ 轴旋转的矩阵为：

$$\begin{vmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{vmatrix}$$

这些旋转矩阵在计算机图形学的书中是很常见的，尤其是教导矩阵转换的章节，例如有平移、缩放、旋转等。可以用之前提过的矩阵相乘，将这三种转换矩阵合并成一个旋转矩阵来表示绕着三个坐标轴的旋转。

在刚体模拟中，可以用旋转矩阵来代表刚体的方位。另一个思考方向是，当旋转矩阵套用在固定全局坐标中未旋转的刚体上时，会旋转刚体的坐标，以改变刚体于任意时间的方位。这导致了当使用旋转矩阵记录刚体方位时需考虑的另一个重要事项：旋转矩阵是时间的函数。

当设定刚体初始的旋转矩阵时，不要直接从方位角计算。作用在刚体上的力和力矩会改变物体的角速度，同样也会在每个时间间隔改变物体的方位。因为旋转矩阵与角速度有关联，所以方位会相对地改变。以下是需要用到的公式：

$$d\boldsymbol{R}/dt = \boldsymbol{\Omega R}$$

这里的 $\boldsymbol{\Omega}$ 是角速度向量分量组成的斜对称矩阵（skew symmetric matrix）：

$$\boldsymbol{\Omega} = \begin{vmatrix} 0 & \omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{vmatrix}$$

尽管这是一个严格的证明，不过仍然可以看出其巧妙之处：只需由角速度做基本的矩阵相乘（以矩阵 $\boldsymbol{\Omega}$ 的方式）就能产生不同的旋转矩阵。在这个模拟程序中，可以知道初始的旋转矩阵，并在每一个时间间隔计算出角速度；所以能轻易地处理旋转矩阵或对它积分。

在这里可以发现，因为只要清楚地计算旋转矩阵一次，并且使用矩阵相乘来更新，就不必在每个时间间隔都使用这个需要大量计算的三角函数，所以就避免了在这一章的介绍中提到的奇怪问题。

很明显，取得这些好处还是要付出一些代价的。第一，必须处理旋转矩阵的9个元素（3 × 3 矩阵的元素）来代表三个角的自由度；第二，为了达到这个目的，必须对旋转矩阵有若干限制。也就是要强制这个矩阵要与1的行列式正交，并能符合以下条件（矩阵中每一行代表一个单位向量，而它们都互相垂直）（注1）：

$$\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R} = \boldsymbol{I}$$

其中$\boldsymbol{R}^{\mathrm{T}}$是$\boldsymbol{R}$的转置矩阵，而$\boldsymbol{I}$是单位矩阵。由于数字上舍去及截断的误差，常常要在模拟中加上此限制。否则，旋转矩阵将不只旋转物体，可能会缩放或平移物体。

除了要处理9个元素并限制6个自由度，只让所需的3个元素出现，还有另外一个方法可以保留旋转矩阵的优点，但却不需要付出那么大的代价。这个方法就是下一节要讨论的四元数法。

## 四元数法

四元数法是一种在数学上有点怪异的方法。四元数是在一百多年前由William Hamilton研究复数（虚数）数学时所发明的，但是这种方法却很有实用价值。四元数是一个量，像是向量却拥有4个元素。常以下列方式表示：

$$\boldsymbol{q} = q_0 + q_x\boldsymbol{i} + q_y\boldsymbol{j} + q_z\boldsymbol{k}$$

四元数事实上是在复数空间中一个4D的量，但它却不能被形象化。然而，这里将四元数当做3D空间中的方位，让它可以被赋予实际上的意义（稍后会看到）。

这里特别要注意的是，在单位四元数的定义中必须满足以下条件：

$$q_0^2 + q_x^2 + q_y^2 + q_z^2 = 1$$

这和正规化的向量（或单位向量）是很类似的。

这里将四元数写成以下形式：$\boldsymbol{q} = [q_0, \boldsymbol{v}]$。这里的$\boldsymbol{v}$是$q_x\boldsymbol{i} + q_y\boldsymbol{j} + q_z\boldsymbol{k}$的向量，而$q_0$是一个标量。在旋转中，$\boldsymbol{v}$代表转动轴的方向。绕着一个以单位向量$\boldsymbol{u}$表示的任意轴线旋转$\theta$角，可以用以下的四元数表示：

$$\boldsymbol{q} = [\cos(\theta/2), \sin(\theta/2)\boldsymbol{u}]$$

图14-3表示任一刚体绕着通过重心的轴线旋转。这里的单位向量$\boldsymbol{u}$是将向量$\boldsymbol{v}$正规化为一单位长所得到的。

---

注1：　若两个向量的内积为0，则称两向量正交（互相垂直）。

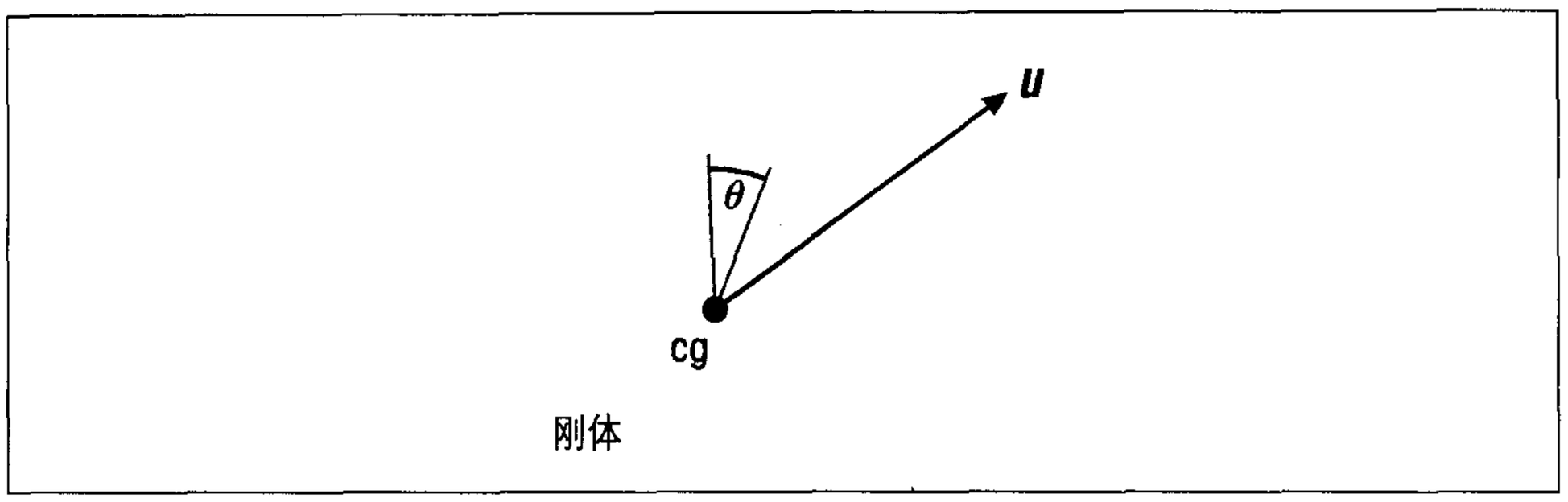



图 14-3：四元数的旋转

从这里可以发现，在旋转时使用四元数将只需要处理 4 个元素而不是 9 个，惟一的限制是要使用单位四元数。这样看来，我认为在处理刚体旋转的模拟程序中，使用四元数应该是不错的选择。事实上，我在下一章真的使用了这个方法。

以四元数代表方位的用法类似于使用旋转矩阵的方式。首先设定四元数代表时间0 时刚体的初始方位（这是惟一要计算四元数的地方）。然后用某个时间点所求出的角速度更新方位，以反映在该时间点新的方位。关于方位四元数对角速度的微分方程与旋转矩阵的微分方程相似，如下：

$$\mathrm{d}\boldsymbol{q}/\mathrm{d}t = (1/2)\boldsymbol{\omega q}$$

这里的角速度根据全局坐标以四元数的形式表示成[0, $\omega$]。如果$\omega$ 是以局部坐标表示的，则需用以下公式：

$$\mathrm{d}\boldsymbol{q}/\mathrm{d}t = (1/2)\boldsymbol{q\omega}$$

和旋转矩阵一样，四元数法也可以用来旋转点或向量。如果$\boldsymbol{v}$ 是一个向量，则$\boldsymbol{v}'$ 就是以四元数 $\boldsymbol{q}$ 旋转后的向量：

$$\boldsymbol{v}' = \boldsymbol{qvq}^*$$

$\boldsymbol{q}^*$ 是 $\boldsymbol{q}$ 的共轭四元数，定义成：

$$\boldsymbol{q}^* = q_0 - q_x\boldsymbol{i} - q_y\boldsymbol{j} - q_z\boldsymbol{k}$$

也可以用上列公式转换坐标系，至旋转后的坐标系。在某些状况可以运用这个技巧，例如在模拟程序中要计算作用于物体上的力，必须先将此力从全局坐标转换成局部坐标。或者在计算物体上的作用力时，要先将物体以全局坐标表示的速度转换成局部坐标。

如同向量和矩阵运算一样，四元数有许多自己的运算规则：如乘法、加法和减法。为了方便读者参考，在附录三中收录了实现四元数运算的程序代码。

# 第十五章

# 3D 刚体模拟器

本章将带领你由2D领域跨入到3D领域，并以飞机为模型实现刚体模拟程序。这个模拟程序使用在第七章中假设的飞机模型。这个飞机模型具有典型的结构：包括了前主翼、机尾升降舵、垂直尾翼和主翼上的襟翼。

再一次提醒你，这个范例的所有程序代码在O'Reilly的网站上都可以找到，但是本章仍然只会包含大部分。如同第十二章及第十三章中的2D模拟器一样，我只会把焦点放在模拟器的物理部分，使用Direct3D技术绘图的其余程序代码就得从网站上下载了。

虽然这是3D的模拟程序，但是却和之前所介绍的2D气垫船模拟器很类似。之前在介绍2D模拟器时，为了能容易地改写成3D的环境，就使用3D的向量类来代表2D的向量——除了模型的不同外——要将2D的程序改写成3D的程序就容易多了。所以如果你已经读过第十二章，应该能看懂本章大部分的程序代码。

和2D的程序一样，3D程序中也有4个主要元素：模型、积分函数、用户输入及绘图。模型是指想要模拟物体的理想化——本例是飞机，而积分函数是负责对运动微分方程积分的函数。这两个元素负责大部分的物理模拟。用户输入和绘图是负责让用户操作和观看模拟程序。本范例依然使用键盘控制，并用Direct3D来绘图。

模拟程序中的全局坐标系是：$x$轴指向屏幕内，$y$轴指向屏幕左方，而$z$轴指向上方。而局部坐标是：$x$轴指向飞机前方，$y$轴指向飞机左方，而$z$轴指向飞机上方。因为这是3D的飞机模拟程序，所以在执行时可以向着任何一个方向飞。因此你可以做出任何特技飞行的动作，例如翻筋斗、侧滚、俯冲、爬升等。

# 模型

在模拟程序中最重要的无非是飞行模型了。第七章整章都在讨论飞行模型的物理学，所以这里不再浪费篇幅介绍，只会在相关的程序代码中复习而已。如果你尚未读过第七章，建议你还是翻回第七章看看，至少得看“飞行模拟”一节。

实现飞行模型的第一步，就是要先定义刚体的结构，封装代表刚体在任何时间的状态所需的资料。以下是我所定义的 RigidBody 结构：

```
typedef struct _RigidBody {

    float           fMass;              // 总质量
    Matrix3x3       mInertia;           // 以局部坐标计算的转动惯量

    Matrix3x3       mInertiaInverse;    // 反转动惯量
    Vector          vPosition;          // 以全局坐标计算的位置
    Vector          vVelocity;          // 以全局坐标计算的速度
    Vector          vVelocityBody;      // 以局部坐标计算的速度
    Vector          vAngularVelocity;   // 以局部坐标计算的角速度
    Vector          vEulerAngles;       // 以局部坐标计算的欧拉角
    float           fSpeed;             // 速率（速度的量值）
    Quaternion      qOrientation;       // 以全局坐标计算的方位
    Vector          vForces;            // 作用于物体上的合力
    Vector          vMoments;           // 作用于物体上的合力矩
} RigidBody, *pRigidBody;
```

你会发现这个结构和2D 气垫船模拟程序中所用的RigidBody2D结构很类似。然而有一个很大的不同，在 2D 的模拟中，方位的资料形态是浮点数，但是在 3D 的模拟程序中，却是使用Quaternion类的四分位数。在前一章已经讨论过以四分位数来记录刚体方位的方式。附录三是 Quaternion 类的完整定义。

接下来是为飞行模型准备初始化函数，好让程序在执行前能先初始化某些值。以下是初始化函数 InitializeAirplane：

```
RigidBody  Airplane;        // 代表飞机的全局变量
.
.
.

void    InitializeAirplane(void)
{
     float iRoll, iPitch, iYaw;
     // 设定初始位置
     Airplane.vPosition.x = -5000.0f;
     Airplane.vPosition.y = 0.0f;
     Airplane.vPosition.z = 2000.0f;

     // 设定初始速度
     Airplane.vVelocity.x = 60.0f;
```

```
        Airplane.vVelocity.y = 0.0f;
        Airplane.vVelocity.z = 0.0f;
        Airplane.fSpeed = 60.0f;

        // 设定初始角速度
        Airplane.vAngularVelocity.x = 0.0f;
        Airplane.vAngularVelocity.y = 0.0f;
        Airplane.vAngularVelocity.z = 0.0f;

        // 设定初始推力、作用力及力矩
        Airplane.vForces.x = 500.0f;
        Airplane.vForces.y = 0.0f;
        Airplane.vForces.z = 0.0f;
        ThrustForce = 500.0;

        Airplane.vMoments.x = 0.0f;
        Airplane.vMoments.y = 0.0f;
        Airplane.vMoments.z = 0.0f;

        // 在局部坐标中将速度设为0
        Airplane.vVelocityBody.x = 0.0f;
        Airplane.vVelocityBody.y = 0.0f;
        Airplane.vVelocityBody.z = 0.0f;

        // 将以下值设为false，可以用键盘来控制这些值
        Stalling = false;
        Flaps = false;

        // 设定初始方位
        iRoll = 0.0f;
        iPitch = 0.0f;
        iYaw = 0.0f;
        Airplane.qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);

        // 现在开始计算飞机质量相关的性质
        CalcAirplaneMassProperties();
    }
```

这个函数设定飞机的初始位置、速度、势态、推力等。接着调用CalcAirplaneMass-Properties来计算与飞机质量特性。这些函数在第七章已经详细地介绍过了，所以这里就不再重复介绍。不过这里要指出和2D模拟不同的地方，就是转动惯量张量。

```
void    CalcAirplaneMassProperties(void)
{
        .
        .
        .

        // 计算飞机零件组合后的转动惯量和惯性积
        // 其中的惯性矩阵（张量）以局部坐标表示
        Ixx = 0;    Iyy = 0;    Izz = 0;
        Ixy = 0;    Ixz = 0;    Iyz = 0;
        for (i = 0; i< 8; i++)
```

```
{
        Ixx += Element[i].vLocalInertia.x + Element[i].fMass *
               (Element[i].vCGCoords.y*Element[i].vCGCoords.y +
               Element[i].vCGCoords.z*Element[i].vCGCoords.z);
        Iyy += Element[i].vLocalInertia.y + Element[i].fMass *
               (Element[i].vCGCoords.z*Element[i].vCGCoords.z +
               Element[i].vCGCoords.x*Element[i].vCGCoords.x);
        Izz += Element[i].vLocalInertia.z + Element[i].fMass *
               (Element[i].vCGCoords.x*Element[i].vCGCoords.x +
               Element[i].vCGCoords.y*Element[i].vCGCoords.y);
        Ixy += Element[i].fMass * (Element[i].vCGCoords.x *
               Element[i].vCGCoords.y);
        Ixz += Element[i].fMass * (Element[i].vCGCoords.x *
               Element[i].vCGCoords.z);
        Iyz += Element[i].fMass * (Element[i].vCGCoords.y *
               Element[i].vCGCoords.z);
}

// 最后，设定飞机的质量和惯性矩阵并求出反矩阵
Airplane.fMass = mass;
Airplane.mInertia.e11 = Ixx;
Airplane.mInertia.e12 = -Ixy;
Airplane.mInertia.e13 = -Ixz;
Airplane.mInertia.e21 = -Ixy;
Airplane.mInertia.e22 = Iyy;
Airplane.mInertia.e23 = -Iyz;
Airplane.mInertia.e31 = -Ixz;
Airplane.mInertia.e32 = -Iyz;
Airplane.mInertia.e33 = Izz;

Airplane.mInertiaInverse = Airplane.mInertia.Inverse();
}
```

这架飞机是由不同零件所组成的，每个零件代表飞机上不同的结构，例如方向舵、升降舵、机翼、机身等。这里标识出来的程序代码使用了第十一章所介绍的技巧，将每一个零件的质量特性加合，最后得到整架飞机的总惯性张量。2D模拟器与3D模拟器最大的不同是，在3D中转动惯量是一个张量，在2D中只是一个标量。

InitializeAirplane函数在程序开始前被调用。将它放在建立及显示窗口的程序代码后是不错的选择。在这个例子中，InitializeAirplane的调用是在InitInstance中：

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
        hInst = hInstance;
        nShowCmd = nCmdShow;
        hTheMainWindow = CreateWindow(szAppName,
                                      szTitle,
                                      WS_OVERLAPPEDWINDOW |
                                      WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                                      0, 0, 640, 480,
                                      NULL, NULL, hInst, NULL);
```

```
if (!CreateD3DRMObject())
     return (FALSE);

if (!CreateD3DRMClipperObject(hTheMainWindow))
     return (FALSE);

if (!CreateViewPort(hTheMainWindow))
     return (FALSE);

ShowWindow(hTheMainWindow, nCmdShow);
UpdateWindow(hTheMainWindow);

InitializeAirplane();

return (TRUE);
}
```

最后要做的是计算于任意时间作用在飞机上的合力及合力矩。如同2D的气垫船模拟程序一样，如果没有此类函数，飞机是不会有反应的。因此这里要设计一个CalcAirplaneLoads，并且在程序执行的每个时间间隔调用。在这个函数中还调用其他函数：LiftCoefficient，DragCoefficient、RudderLiftCoefficient和RudderDragCoefficient。这些函数在第七章的（飞行模拟）一节中已经详细地讨论过了。

CalcAirplaneLoads与CalcLoads函数要做的事大部分都一样。不过CalcAirplanLoads函数比较复杂。因为飞机模型是由许多零件组成的，这些零件都会影响飞机的总升力和总阻力。以下标识出其他不同的地方：

```
void    CalcAirplaneLoads(void)
{
     .
     .
     .

     // 将力转换到全局坐标
     Airplane.vForces = QVRotate(Airplane.qOrientation, Fb);

     // 考虑重力（定义成 -32.174 ft/s^2)
     Airplane.vForces.z += g * Airplane.fMass;

     .
     .
     .
}
```

先将作用于飞机上的力以局部坐标计算，在考虑重力的影响前先转换成全局坐标。这里使用QVRotate函数来转换坐标，而且这个函数根据以四元数表示的飞机方位来转换力的向量（注1）。

---

注1：　QVRotate函数定义在附录三中。

# 积分函数

到目前为止已经完成了定义、初始化、受力计算的程序代码，接着要实现的是负责将微分方程按照不同的时间来积分的程序。在这个范例中所选择的积分法是欧拉法。在第十一章中已经讨论过这个方法，并且在第十二章及第十三章中已经在模拟程序中实现过了。选用欧拉法的原因是它够简单，而我也不想让程序看起来太复杂，我想让读者能专注于重要的程序代码。所以这里提供一个函数StepSimulation负责程序里所有的积分动作：

```
void    StepSimulation(float dt)
{
      // 先处理飞机的移动：
      // （如果物体是粒子，只要考虑显性移动即可）

          Vector Ae;

          // 计算飞机上的合力和合力矩：
          CalcAirplaneLoads();

          // 计算飞机以全局坐标计算的加速度：
          Ae = Airplane.vForces / Airplane.fMass;

          // 计算飞机以全局坐标计算的速度：
          Airplane.vVelocity += Ae * dt;

          // 计算飞机以全局坐标计算的位置：
          Airplane.vPosition += Airplane.vVelocity * dt;

    // 处理飞机的转动：
          float                 mag;

          // 计算飞机以局部坐标计算的角速度：
          Airplane.vAngularVelocity += Airplane.mInertiaInverse *
                                       (Airplane.vMoments -
                                       (Airplane.vAngularVelocity^
                                       (Airplane.mInertia *
                                         Airplane.vAngularVelocity)))
                                         * dt;

          // 计算新的旋转四元数：
          Airplane.qOrientation += (Airplane.qOrientation *
                                    Airplane.vAngularVelocity) *
                                   (0.5f * dt);

          // 将代表方位的四元数正规化：
          mag = Airplane.qOrientation.Magnitude();
          if (mag != 0)
               Airplane.qOrientation /= mag;

          // 计算飞机以局部坐标计算的速度
          // （计算升力和阻力时会用到此值）
          Airplane.vVelocityBody = QVRotate(~Airplane.qOrientation,
                                            Airplane.vVelocity);
```

```
    // 计算空速：
    Airplane.fSpeed = Airplane.vVelocity.Magnitude();

    // 计算出欧拉角以备后来使用
    Vector u;

    u = MakeEulerAnglesFromQ(Airplane.qOrientation);
    Airplane.vEulerAngles.x = u.x;     // 滚转
    Airplane.vEulerAngles.y = u.y;     // 俯仰
    Airplane.vEulerAngles.z = u.z;     // 偏转
}
```

StepSimulation先调用CalcAirplaneLoads函数来计算目前飞机的总受力。然后根据这个受力值计算出飞机的线性加速度。接着，在计算飞机的线性速度及位置时，使用欧拉法来积分。如果模拟的只是粒子，那么只要做到这样就可以了。但是，因为现在模拟的飞机不是粒子，所以要处理角的运动。

处理角运动的第一步是计算在这个时间间隔的新角速度。使用欧拉积分法并根据之前求得的飞机上的力矩和质量特性来计算。使用之前讨论的运动方程计算出的位置是以局部坐标表示的：

$$\sum \boldsymbol{M}_{\mathrm{cg}} = \mathrm{d}\boldsymbol{H}_{\mathrm{cg}} / \mathrm{d}t = \boldsymbol{I}(\mathrm{d}\boldsymbol{\omega} / \mathrm{d}t) + [\boldsymbol{\omega} \times (\boldsymbol{I}\boldsymbol{\omega})]$$

下一步是再积分一次，并更新以四元数表示的飞机方位。这里要用到前一章所介绍的，方位四元数对角速度的微分方程。

$$\mathrm{d}\boldsymbol{q}/\mathrm{d}t = (1/2)\boldsymbol{\omega}\boldsymbol{q}$$

接着要检查代表方位的四元数是否为单位四元数，否则就要将方位四元数正规化。

因为先前的线性速度是以全局坐标计算的，而CalcAirplaneLoads需要的是以局部坐标计算的速度，所以要先将这个函数转换坐标系，并存储在RigidBody结构的成员vVelocityBody中。使用QVRotate可以轻易地根据飞机的方位来旋转。请注意这里用的是共轭四元数，因为现在要将坐标由全局坐标转换成局部坐标。

为了方便，我们会一并计算空速。这个值只是线性速度的量值，只是为了显示在窗口的标题列而已。最后，可以由方位四元数得到三个欧拉角——分别是俯仰、滚转及偏转。这些值也会在标题列中显示。这里所使用的函数的定义在附录三中可以找到。

StepSimulation应该在每个周期被调用一次。在这个范例中，使用另一个函数NullEvent，在每次的窗口消息循环中都会调用此函数：

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
```

```
                              int nCmdShow)
{
.
.
.
     OldTime = timeGetTime();
     NewTime = OldTime;
     // 主消息循环:
     while (1) {
          while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
               if (msg.message == WM_QUIT) {
                    return msg.wParam;
               }
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }

          NullEvent();

     }
.
.
.
}
```

当NullEvent函数调用StepSimulation时，将时间间隔的大小dt当做参数传入。和气垫船范例一样，大可不必这样做。这么做的原因是，要试验以两次调用StepSimulation之间的时间差作为即时算出的时间间隔：

```
void      NullEvent(void)
{
.
.
.
     NewTime = timeGetTime();
     dt = (float) (NewTime - OldTime)/1000;
     OldTime = NewTime;

     if (dt > (0.016f)) dt = (0.016f);
     if (dt < 0.001f) dt = 0.001f;

     StepSimulation(dt);
.
.
.
}
```

在这里我又将限制时间间隔的大小：上限是为了控制数值的稳定，下限是为了控制计时器的精确度。你可以根据积分法和所选择的计时器来调整这个上下限。

# 飞行控制

现在的模拟程序仍然不能正确运作，因为还没有实现飞行控制。飞行控制可以让你透过不同的介面来操作飞机 —— 这里用键盘来当做输入装置。请记住，在这种以物理为基础的模拟程序中，不能直接控制飞机的动作，你能控制的是作用在飞机上的各种作用力。积分函数会根据这些作用力来改变飞机的动作。

在程序中，飞机的操纵杆是以方向键来模拟的。下键是指将操纵杆向后拉，会拉起机鼻。上键是指将操纵杆向前推，使飞机俯冲。左键会使飞机向左翻滚，而右键会使飞机向右翻滚。X 键是使方向舵向左偏，会使飞机向左方偏转。C 键使方向舵右偏，使飞机向右方偏转。推力是由 A 键和 Z 键控制的。A 键使推力增加 100 lb，Z 键使推力减少 100 lb。最小推力是 0 lb，最大推力是 3000 lb。F 键会启动襟翼使飞机在低速时能增加升力，D 键会关掉襟翼。

俯仰的角度由机尾升降舵襟翼的偏折来控制。举例来说，升降舵的襟翼偏折向上（后缘上扬），可以让机鼻上扬。滚转的角度是将两个襟翼分别置于不同的方向来控制的；例如，要使飞机向右滚转，就要将右襟翼向上偏折，左襟翼向下偏折。偏转则是由垂直尾翼上的方向舵来控制的，要使飞机向左偏转，就要使方向舵后缘向左偏折。

当使用者按下控制键时程序会调用相对应的函数。以下是控制推力的两个函数：

```
void    IncThrust(void)
{
     ThrustForce += _DTHRUST;
     if(ThrustForce > _MAXTHRUST)
          ThrustForce = _MAXTHRUST;

}

void    DecThrust(void)
{
     ThrustForce -= _DTHRUST;
     if(ThrustForce < 0)
          ThrustForce = 0;
}
```

IncThrust 每次会增加推力 _DTHRUST，并确保推力不会超过 _MAXTHRUST。_DTHRUST 和 _MAXTHRUST 这两个值定义如下：

```
#define     _DTHRUST       100.0f
#define     _MAXTHRUST     3000.0f
```

而 DecThrust 函数每次会减少推力 _DTHRUST，并确保推力不会低于 0。

以下三个函数用来操纵方向舵来控制飞机的偏转：

```
void    LeftRudder(void)
{
     Element[6].fIncidence = 16;
}

void    RightRudder(void)
{
     Element[6].fIncidence = -16;
}

void    ZeroRudder(void)
{
     Element[6].fIncidence = 0;
}
```

LeftRudder将Element[6]（方向舵）的入射角改为16°；RightRudder将入射角改为−16°。ZeroRudder则将角度归零。

而副翼（或襟翼）的角度是由以下三个函数所控制的：

```
void    RollLeft(void)
{
     Element[0].iFlap = 1;
     Element[3].iFlap = -1;
}

void    RollRight(void)
{
     Element[0].iFlap = -1;
     Element[3].iFlap = 1;
}

void    ZeroAilerons(void)
{
     Element[0].iFlap = 0;
     Element[3].iFlap = 0;
}
```

RollLeft函数使左副翼（位于左舷主翼部分Element[0]）向上偏并使右副翼（位于右舷主翼部分Element[3]）向下偏。RollRight的动作恰好相反。ZeroAilerons则重设两个副翼至原先未偏折的位置。

另外有一些函数是用来控制后方升降舵来让飞机俯仰的：

```
void    PitchUp(void)
{
     Element[4].iFlap = 1;
     Element[5].iFlap = 1;
}
```

```
void    PitchDown(void)
{
     Element[4].iFlap = -1;
     Element[5].iFlap = -1;
}

void    ZeroElevators(void)
{
     Element[4].iFlap = 0;
     Element[5].iFlap = 0;
}
```

Element[4]和Element[5]都是升降舵。PitchUp函数使升降舵的襟翼向上偏折，而PitchDown则是使襟翼向下偏折。ZeroElevators是使两个襟翼回到未偏折的位置。

最后还有两个控制着陆襟翼的函数：

```
void    FlapsDown(void)
{
     Element[1].iFlap = -1;
     Element[2].iFlap = -1;
     Flaps = true;
}

void    ZeroFlaps(void)
{
     Element[1].iFlap = 0;
     Element[2].iFlap = 0;
     Flaps = false;
}
```

着陆襟翼装设在左右方的主翼（分别是Element[1]和Element[2]）内侧。FlapsDown使着陆襟翼向下偏折，ZeroFlaps函数则使着陆襟翼回到原先的位置。

当使用者按下控制键时，这些函数会被调用。而且要在调用StepSimulation之前调用，这样一来在目前时间间隔的力与力矩的计算中才能回应这些改变。因为把StepSimulation放在NullEvent中，所以控制飞行姿态的这些函数也要放在同一个地方。以下是实现方法：

```
void    NullEvent(void)
{
.
.
.

     ZeroRudder();
     ZeroAilerons();
     ZeroElevators();

     // 向下俯冲
```

```
if (IsKeyDown(VK_UP))
     PitchDown();
// 向上仰升
if (IsKeyDown(VK_DOWN))
     PitchUp();

// 向左滚转
if (IsKeyDown(VK_LEFT))
     RollLeft();

// 向右滚转
if (IsKeyDown(VK_RIGHT))
     RollRight();

// 增加推力
if (IsKeyDown(0x41)) // A键
     IncThrust();

// 减少推力
if (IsKeyDown(0x5A)) // Z键
     DecThrust();

// 向左偏转
if (IsKeyDown(0x58)) // X键
     LeftRudder();

// 向右偏转
if (IsKeyDown(0x43)) // C键
     RightRudder();

// 着陆襟翼下偏折
if (IsKeyDown(0x46)) // F键
     FlapsDown();

// 着陆襟翼上偏折
if (IsKeyDown(0x44)) // D 键
     ZeroFlaps();

NewTime = timeGetTime();
dt = (float) (NewTime - OldTime)/1000;
OldTime = NewTime;
if (dt > (0.016f)) dt = (0.016f);
if (dt < 0.001f) dt = 0.001f;

     StepSimulation(dt);

.
.
.
}
```

在调用StepSimulation之前，要先检查是否有控制键被按下。如果有，就要调用相对应的函数。

IsKeyDown 函数会检查是否有某个键被按下。程序看起来像这样：

```
BOOL IsKeyDown(short KeyCode)
{
      SHORT retval;

      retval = GetAsyncKeyState(KeyCode);

      if (HIBYTE(retval))
            return TRUE;

      return FALSE;
}
```

使用此函数是因为在同时间内可能有多个按键被按下，我想在同时一并处理，而不是像标准窗口消息处理函数一样一次只处理一个。

额外的飞行控制程序码使模拟程序的物理部分更加完整。到目前为止，已经完成了模型、积分函数和使用者输入（飞行控制）的程序代码。剩下要做的是如何显示窗口并画出模拟的东西。

# 绘图

设定窗口并画出有趣的东西与物理学并没有关系。不过为了完整起见，这里简短地介绍本范例建立窗口及实际绘图所用的程序代码（注 2）。

从主窗口开始，先调用标准的 Windows API 来初始化应用程序，建立并更新窗口，并处理窗口消息与使用者输入。由于已经假设你对于窗口的程序设计已经很了解了，所以不再深入地解说这些程序代码。

之前已经看过部分的 WinMain 函数，以下是完整的程序代码：

```
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
      MSG msg;
      HANDLE hAccelTable;

      if (!hPrevInstance) {
            // 进行应用程序的初始化：
            if (!InitApplication(hInstance)) {
                  return (FALSE);
```

---

注 2：　如果你对于 Direct3D 的程序设计不熟悉，可以参考由 Peter J. Kovack 所著的“The Awesome Power of Direct3D”。此书相当实用。

```
        }
    }

    // 进行实体代号的初始化:
    if (!InitInstance(hInstance, nCmdShow)) {
        return (FALSE);
    }

    hAccelTable = LoadAccelerators (hInstance, szAppName);

    OldTime = timeGetTime();
    NewTime = OldTime;
    // 主消息窗口:
    while (1) {

        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
            if (msg.message == WM_QUIT) {
                return msg.wParam;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        NullEvent();
    }

    return (msg.wParam);
}
```

WinMain函数调用InitInstance和InitApplication函数。前面已经看过InitInstance，以下是InitApplication函数：

```
BOOL InitApplication(HINSTANCE hInstance)
{
    WNDCLASS wc;
    HWND     hwnd;

    hwnd = FindWindow (szAppName, NULL);
    if (hwnd) {
        if (IsIconic(hwnd)) {
            ShowWindow(hwnd, SW_RESTORE);
        }
    SetForegroundWindow (hwnd);

    return FALSE;
    }
    wc.style         = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
    wc.lpfnWndProc   = (WNDPROC)WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = NULL;
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
```

```
        wc.lpszMenuName = NULL;
        wc.lpszClassName = szAppName;

        return RegisterClass(&wc);
}
```

这个 API 先为主窗口建立窗口类，接着注册此类，并建立一个 640 × 480 的窗口，以及建立在 Direct3D 的视埠上绘图所需要的 Direct3D 对象（在 InitInstance 中调用），最后进入主程序循环，并在每次循环调用 NullEvent。

另一个用来处理窗口消息的 API 函数是 WndProc：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    int             wmId, wmEvent;
    BOOL            validmenu = FALSE;
    int             selection = 0;
    PAINTSTRUCT     ps;
    HDC             pDC;
    WPARAM          key;

    switch (message) {
        case WM_ACTIVATE:
            if (SUCCEEDED(D3D.Device->QueryInterface(
                          IID_IDirect3DRMWinDevice,
                          (void **) &WinDev)))
            {
                    if (FAILED(WinDev->HandleActivate(wParam)))
                        WinDev->Release();
            }

            break;

        case WM_DESTROY:
            CleanUp();
            PostQuitMessage(0);
            break;

        case WM_KEYDOWN:
            key = (int) wParam;

            if (key == 0x31) // 1 键
                SetCamera1();

            if (key == 0x32) // 2 键
                SetCamera2();

            if (key == 0x33) // 3 键
                SetCamera3();

            break;
```

```
case WM_PAINT:
        pDC = BeginPaint(hTheMainWindow, (LPPAINTSTRUCT) &ps);

        if (SUCCEEDED(D3D.Device->QueryInterface(
                    IID_IDirect3DRMWinDevice,
                    (void **) &WinDev)))
        {
            if (FAILED(WinDev->HandlePaint(ps.hdc)))
                WinDev->Release();
            }

            EndPaint(hTheMainWindow, (LPPAINTSTRUCT) &ps);
            return (0);
        break;

    default:
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
    return (0);
}
```

为了处理WM_ACTIVATE消息，函数会取得使用Direct3D Retained Mode所需的IDirect3DRMWinDevice对象。

当收到WM_KEYDOWN消息时，函数会让视角在三个事先设定好的摄影机之间切换。一号摄影机放置在飞机的驾驶舱，二号摄影机放在机身的正后方，三号摄影机放置在全局坐标的原点上，并随着飞机的移动随时对准飞机。

当收到WM_PAINT消息时就表示必须在主窗口上画出场景。最后收到WM_DESTROY消息时，程序要清除所有的Direct3D 对象并结束程序。

以下是另一个版本的NullEvent函数：

```
void    NullEvent(void)
{
    Vector  vz, vx;
    char    buf[256];
    char    s[256];

    ZeroRudder();
    ZeroAilerons();
    ZeroElevators();

    // 向下俯冲
    if (IsKeyDown(VK_UP))
        PitchDown();

    // 向上仰升
    if (IsKeyDown(VK_DOWN))
        PitchUp();
```

```
// 向左滚转
if (IsKeyDown(VK_LEFT))
     RollLeft();

// 向右滚转
if (IsKeyDown(VK_RIGHT))
     RollRight();

// 增大推力
if (IsKeyDown(0x41)) // A
     IncThrust();

// 减小推力
if (IsKeyDown(0x5A)) // Z
     DecThrust();

// 向左偏转
if (IsKeyDown(0x58)) // x
     LeftRudder();

// 向右偏转
if (IsKeyDown(0x43)) // c
     RightRudder();

// 放下着陆襟翼
if (IsKeyDown(0x46)) // f
     FlapsDown();

// 升起着陆襟翼
if (IsKeyDown(0x44)) // d
     ZeroFlaps();

NewTime = timeGetTime();
dt = (float) (NewTime - OldTime)/1000;
OldTime = NewTime;

if (dt > (0.016f)) dt = (0.016f);
if (dt < 0.001f) dt = 0.001f;

StepSimulation(dt);

if(FrameCounter >= RENDER_FRAME_COUNT)
{
   // Direct3D的x轴 = 局部坐标的-y轴
   // Direct3D的y轴 = 局部坐标的z轴
   // Direct3D的z轴 = 局部坐标的x轴
   SetCameraPosition( -Airplane.vPosition.y,
                      Airplane.vPosition.z,
                      Airplane.vPosition.x);

   vz = GetBodyZAxisVector(); // 指向局部坐标的上方
   vx = GetBodyXAxisVector(); // 指向局部坐标的前方

   SetCameraOrientation(-vx.y, vx.z, vx.x, -vz.y, vz.z, vz.x);
   Render();
   OldTime = NewTime;
```

```
// 将状态显示在窗口的标题列上
sprintf( buf, "Roll= %.2f ; ", Airplane.vEulerAngles.x);
strcpy(s, buf);
//将此视为负数，因为飞行员将向上仰视为正值：
sprintf( buf, "Pitch= %.2f ; ", -Airplane.vEulerAngles.y);
strcat(s, buf);
sprintf( buf, "Yaw= %.2f ; ", Airplane.vEulerAngles.z);
strcat(s, buf);
sprintf( buf, "Alt= %.0f ; ", Airplane.vPosition.z);
strcat(s, buf);
sprintf( buf, "T= %.0f ; ", ThrustForce);
strcat(s, buf);
sprintf( buf, "S= %.0f ", Airplane.fSpeed/1.688); // 除以1.688
                                                  // 从ft/s换算成节
strcat(s, buf);
if(Flaps)
     strcat(s, "; Flaps");

if(Stalling)
{
     strcat(s, "; Stall!");
     Beep(10000, 250);
}

SetWindowText(hTheMainWindow, s);
} else
   FrameCounter++;
}
```

在调用 StepSimulation 以后是从未看过的程序代码，在这里执行了许多事。

首先，摄影机的位置必须随飞机的新位置而更新。虽然这很容易做到，但是要记得在前面气垫船范例中的注意事项，Direct3D 所用的坐标系与模拟程序所用的坐标系是不同的。Direct3D 所用的是左手坐标系：$x$ 轴指向右方、$y$ 轴指向上方、而 $z$ 轴指向屏幕内。所以，Direct3D 的 $x$ 轴是模拟程序 $y$ 轴的负方向，它的 $y$ 轴是模拟程序的 $z$ 轴，而 $z$ 轴是模拟程序的 $x$ 轴。

除了把摄影机放在正确的位置之外，还要注意其方位是否正确。所以必须为Direct3D准备一对向量，分别用来定义在画面中新的 $z$ 轴和新的 $y$ 轴。为了简单起见，在根据飞机的方位设定摄影机的方位时，我用两个新函数来计算飞机的 $x$ 轴与 $z$ 轴向量，分别代表 Direct3D 的 $z$ 轴与 $y$ 轴向量。这样一来，模拟的内容就会比较真实。举例来说，你坐在一号摄影机（也就是驾驶舱的位置），当飞机俯仰、滚转或偏转时，会看到摄影机也有相对应的动作，就好像真地坐在驾驶舱一样：

```
Vector      GetBodyZAxisVector(void)
{
     Vector      v;
```

```
        v.x = 0.0f;
        v.y = 0.0f;
        v.z = 1.0f;

        return QVRotate(Airplane.qOrientation, v);
}

Vector      GetBodyXAxisVector(void)
{
        Vector v;

        v.x = 1.0f;
        v.y = 0.0f;
        v.z = 0.0f;

        return QVRotate(Airplane.qOrientation, v);

}
```

再回到NullEvent，在放好摄影机之后，调用Render函数将场景画在主窗口上。之后，一些统计资料会在窗口标题显示，即三个欧拉角、推力及空速。当着陆襟翼被放下时，标题列会显示“flaps”这个字。同样地，当飞机失速时，标题列会出现“stall”。

请注意这里用了和气垫船程序一样的技巧，使物理模拟计算的频率高于绘图的频率。这个比率要根据模拟程序来调整。

剩下的程序代码是和使用Direct3D来绘图有关的，和物理学并没有直接的关系，所以我并没有把这些程序代码放进来。然而，你可以在O'Reilly的网站（*www.oreilly.com*）上得到全部的程序代码。

# 第十六章

# 3D 多重物体模拟

本章将介绍在3D 的空间中如何处理多个刚体之间的碰撞。这里所使用的范例是使一辆汽车撞击一对测试用的障碍物。图 16-1 是这个模拟程序中刚好在碰撞时的快照。

这个范例程序会自动地在三个不同的碰撞场景中循环播放，每个场景中障碍物的摆放方式都不同（注 1）。你可以按下数字键 1，2，3 来切换不同的摄影机角度。一号摄影机放在汽车的重心位置，二号摄影机放在汽车后方的外侧，而三号摄影机则放在汽车左方的外侧。

这个范例使用上一章的飞行模拟范例用到的许多程序代码，所以我不会再重复这些相同的程序代码。然而我会将程序中不同的地方标识出来。处理这个模拟程序的步骤与前面所用的步骤很类似：

- 设定刚体数据结构来存储每个物体状态的信息。
- 设定一个阵列来存储这些刚体。
- 将这些物体初始化。
- 计算在每个时间间隔作用在每个物体上的力。
- 用积分函数算出并更新每个物体上的速度及位置。
- 处理碰撞的情形。

---

注 1：　在此提醒大家，本书中所讨论的所有实例的源文件和可执行文件，都可以在 O'Reilly 的网站上找到。

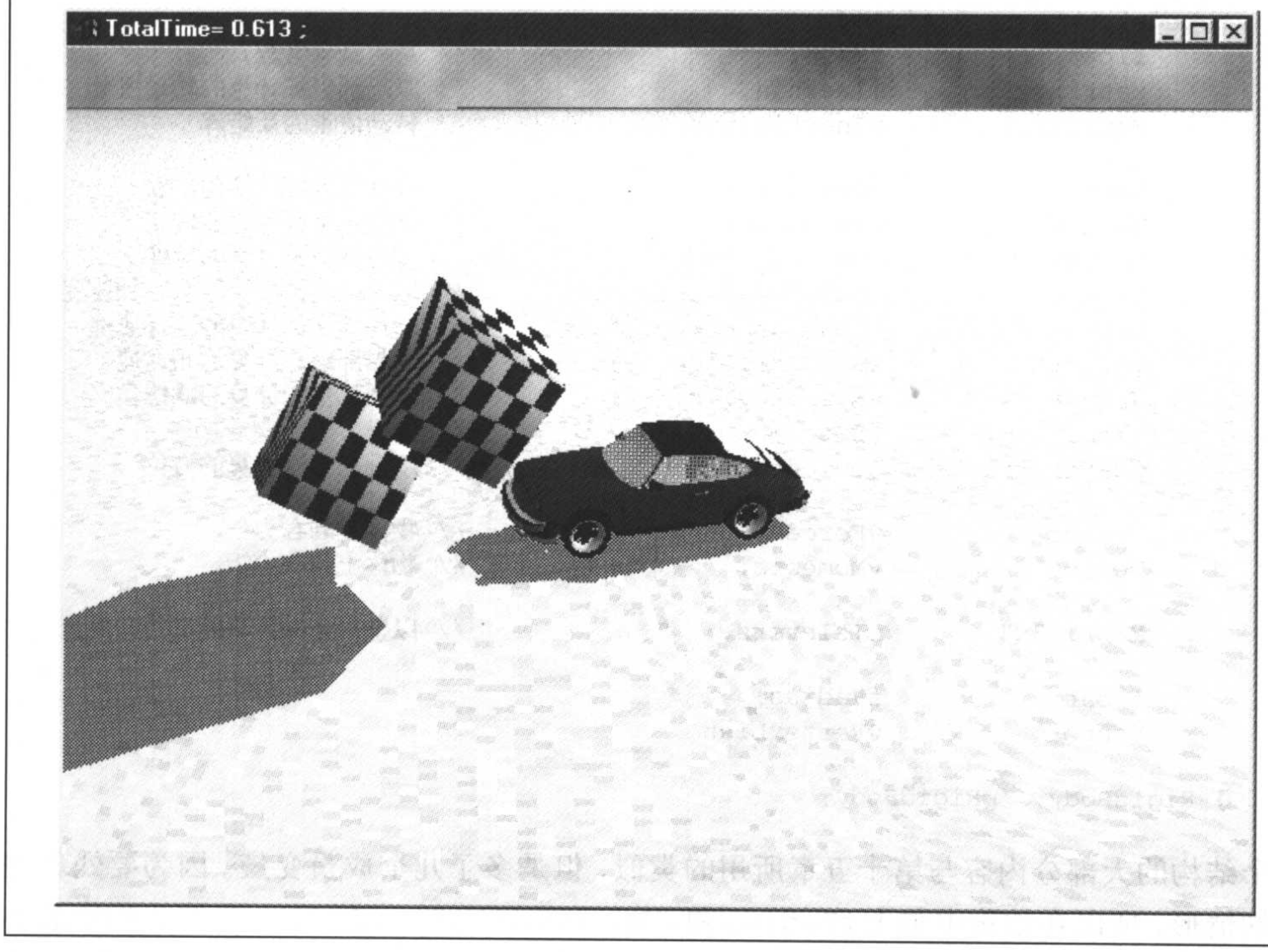



图 16-1：撞击模拟

在接下来的几节中，会陆续介绍这几个步骤。接着会讨论如何调整这个模拟程序的各项参数。最后也会讨论这个模拟程序的限制，并建议改进的方法。

# 模型

这个模拟程序像前一章的模拟飞行程序一样，并不是要教你如何做出精细的模型，所以程序中将会用简单的模型。这一章的重点是教你如何处理多个物体的模拟，并不会让复杂的初始化和公式计算占据整个程序代码。当然你可以在自己的模拟程序中，使用前几章示范的技巧加入精密的模型来模拟。

# 初始化

先从记录物体状态的 RigidBody 结构开始查看：

```
//-----------------------------------------------------------------------------//
// 刚体的结构
```

```
//-------------------------------------------------------------------------//
typedef struct _RigidBody {
     float          fMass;                  // 总质量（常数）
     Matrix3x3      mInertia;               // 以局部坐标计算的转动惯量
     Matrix3x3      mInertiaInverse;        // 转动惯量的反矩阵

     Vector         vPosition;              // 以全局坐标计算的位置
     Vector         vVelocity;              // 以全局坐标计算的速度
     Vector         vVelocityBody;          // 以局部坐标计算的速度
     Vector         vAcceleration;          // 以全局坐标计算的重心加速度
     Vector         vAngularAcceleration;   // 以局部坐标计算的角加速度
     Vector         vAngularVelocity;       // 以局部坐标计算的角速度
     Vector         vEulerAngles;           // 以局部坐标计算的欧拉角
     float          fSpeed;                 // 速率
     Quaternion     qOrientation;           // 以全局坐标计算的方位

     Vector         vForces;                // 物体上的合力
     Vector         vMoments;               // 物体上的合力矩

     Matrix3x3      mIeInverse;             // 以全局坐标计算的反转动惯量

     float          fRadius;
     Vector         vVertexList

} RigidBody, *pRigidBody;
```

这个结构的大部分内容与第十五章所用的类似，但是多了几个成员变量。因为要处理碰撞的情形，所以在结构中加上fRadius和vVertexList[8]。fRadius存储此物体最小球形边界的半径，用来做碰撞检查。vVertexList[8]是顶点的集合，这些顶点就是“硬点”，用来做碰撞侦测。

另外多了两个变量vAcceleration和vAngularAcceleration来存储物体的线性加速度和角加速度。我们需要这两个值来处理当物体放在地上而没有碰撞发生时物体与地面间的接触力。稍后会讨论这一部分。

在定义好结构后，可以把刚体存储在阵列中：

```
#define         NUMBODIES      3
RigidBody       Bodies[NUMBODIES];
```

在模拟程序开始前，要先调用初始化函数；以下是其程序代码：

```
//-------------------------------------------------------------------------//
// 此函数设定物体的初始值
//-------------------------------------------------------------------------//
void InitializeObjects(int          configuration)
{
     float    iRoll, iPitch, iYaw;
     int      i;
     float    Ixx, Iyy, Izz;
     float    s;
```

```
// 初始化汽车:
// 设定初始位置
Bodies[0].vPosition.x = -50.0f;
Bodies[0].vPosition.y = 0.0f;
Bodies[0].vPosition.z = CARHEIGHT/2.0f;

// 设定初始速度
switch(configuration)
{
     case 0: s = 110.0f; break; // ft/s
     case 1: s = 120.0f; break; // ft/s
     case 2: s = 115.0f; break; // ft/s
}
Bodies[0].vVelocity.x = s;
Bodies[0].vVelocity.y = 0.0f;
Bodies[0].vVelocity.z = 0.0f;
Bodies[0].fSpeed = s;

// 设定初始角速度
Bodies[0].vAngularVelocity.x = 0.0f;
Bodies[0].vAngularVelocity.y = 0.0f;
Bodies[0].vAngularVelocity.z = 0.0f;

Bodies[0].vAngularAcceleration.x = 0.0f;
Bodies[0].vAngularAcceleration.y = 0.0f;
Bodies[0].vAngularAcceleration.z = 0.0f;

Bodies[0].vAcceleration.x = 0.0f;
Bodies[0].vAcceleration.y = 0.0f;
Bodies[0].vAcceleration.z = 0.0f;

// 设定初始推力、作用力和力矩
Bodies[0].vForces.x = 0.0f;
Bodies[0].vForces.y = 0.0f;
Bodies[0].vForces.z = 0.0f;
ThrustForce = 0.0;

Bodies[0].vMoments.x = 0.0f;
Bodies[0].vMoments.y = 0.0f;
Bodies[0].vMoments.z = 0.0f;

// 将以局部坐标表示的速度置零
Bodies[0].vVelocityBody.x = 0.0f;
Bodies[0].vVelocityBody.y = 0.0f;
Bodies[0].vVelocityBody.z = 0.0f;

// 设定初始方位
iRoll = 0.0f;
iPitch = 0.0f;
iYaw = 0.0f;
Bodies[0].qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);


// 设定与质量相关的特性
Bodies[0].fMass = 2000.0f/(-g);
```

```
Ixx = Bodies[0].fMass/12.0f *
      (CARWIDTH*CARWIDTH + CARHEIGHT*CARHEIGHT);
.Iyy = Bodies[0].fMass/12.0f *
      (CARHEIGHT*CARHEIGHT + CARLENGTH*CARLENGTH);
Izz = Bodies[0].fMass/12.0f *
      (CARWIDTH*CARWIDTH + CARLENGTH*CARLENGTH);

Bodies[0].mInertia.e11 = Ixx;
Bodies[0].mInertia.e12 = 0;
Bodies[0].mInertia.e13 = 0;
Bodies[0].mInertia.e21 = 0;
Bodies[0].mInertia.e22 = Iyy;
Bodies[0].mInertia.e23 = 0;
Bodies[0].mInertia.e31 = 0;
Bodies[0].mInertia.e32 = 0;
Bodies[0].mInertia.e33 = Izz;

Bodies[0].mInertiaInverse = Bodies[0].mInertia.Inverse();

Bodies[0].fRadius = CARLENGTH/2; // 球形边界检查所需

// 相对于重心的边界顶点（假设重心居中）
Bodies[0].vVertexList[0].x = CARLENGTH/2.0f;
Bodies[0].vVertexList[0].y = CARWIDTH/2.0f;
Bodies[0].vVertexList[0].z = -CARHEIGHT/2.0f;

Bodies[0].vVertexList[1].x = CARLENGTH/2.0f;
Bodies[0].vVertexList[1].y = CARWIDTH/2.0f;
Bodies[0].vVertexList[1].z = CARHEIGHT/2.0f;

Bodies[0].vVertexList[2].x = CARLENGTH/2.0f;
Bodies[0].vVertexList[2].y = -CARWIDTH/2.0f;
Bodies[0].vVertexList[2].z = CARHEIGHT/2.0f;

Bodies[0].vVertexList[3].x = CARLENGTH/2.0f;
Bodies[0].vVertexList[3].y = -CARWIDTH/2.0f;
Bodies[0].vVertexList[3].z = -CARHEIGHT/2.0f;

Bodies[0].vVertexList[4].x = -CARLENGTH/2.0f;
Bodies[0].vVertexList[4].y = CARWIDTH/2.0f;
Bodies[0].vVertexList[4].z = -CARHEIGHT/2.0f;

Bodies[0].vVertexList[5].x = -CARLENGTH/2.0f;
Bodies[0].vVertexList[5].y = CARWIDTH/2.0f;
Bodies[0].vVertexList[5].z = CARHEIGHT/2.0f;
Bodies[0].vVertexList[6].x = -CARLENGTH/2.0f;
Bodies[0].vVertexList[6].y = -CARWIDTH/2.0f;
Bodies[0].vVertexList[6].z = CARHEIGHT/2.0f;

Bodies[0].vVertexList[7].x = -CARLENGTH/2.0f;
Bodies[0].vVertexList[7].y = -CARWIDTH/2.0f;
Bodies[0].vVertexList[7].z = -CARHEIGHT/2.0f;

ThrustForce = 0.0f;
```

```
// 初始化障碍物

for(i=1; I<NUMBODIES; i++)
{
   // 设定初始位置
   switch(configuration)
   {
        case 2:
             if(i==1)
             {
                  Bodies[i].vPosition.x = BLOCKSIZE*4;
                  Bodies[i].vPosition.y = -(BLOCKSIZE/2.0f+1.0f);
                  Bodies[i].vPosition.z = BLOCKSIZE/2.0f;
             } else {
                  Bodies[i].vPosition.x = 0.0f;
                  Bodies[i].vPosition.y = 0.0f;
                  Bodies[i].vPosition.z = BLOCKSIZE/2.0f;
             }
             break;

        case 1:
             if(i==1)
             {
                  Bodies[i].vPosition.x = BLOCKSIZE*4;
                  Bodies[i].vPosition.y = -(BLOCKSIZE/2.0f+1.0f);
                  Bodies[i].vPosition.z = BLOCKSIZE/2.0f;
             } else {
                  Bodies[i].vPosition.x = 0.0f;
                  Bodies[i].vPosition.y = BLOCKSIZE/2.0f+1.0f;
                   Bodies[i].vPosition.z = BLOCKSIZE/2.0f;
             }
             break;

        case 0:
             if(i==1)
             {
                  Bodies[i].vPosition.x = BLOCKSIZE+1.0f;
                  Bodies[i].vPosition.y = BLOCKSIZE/2.0f+1.0f;
                  Bodies[i].vPosition.z = BLOCKSIZE/2.0f;
             } else {
                  Bodies[i].vPosition.x = 0.0f;
                  Bodies[i].vPosition.y = BLOCKSIZE/2.0f+1.0f;
                  Bodies[i].vPosition.z = BLOCKSIZE/2.0f;
             }
             break;
   }

   // 设定初始速度
   Bodies[i].vVelocity.x = 0.0f;
   Bodies[i].vVelocity.y = 0.0f;
   Bodies[i].vVelocity.z = 0.0f;
   Bodies[i].fSpeed = 0.0f;

   // 设定初始角速度
```

```
Bodies[i].vAngularVelocity.x = 0.0f;
Bodies[i].vAngularVelocity.y = 0.0f;
Bodies[i].vAngularVelocity.z = 0.0f;

Bodies[i].vAngularAcceleration.x = 0.0f;
Bodies[i].vAngularAcceleration.y = 0.0f;
Bodies[i].vAngularAcceleration.z = 0.0f;

Bodies[i].vAcceleration.x = 0.0f;
Bodies[i].vAcceleration.y = 0.0f;
Bodies[i].vAcceleration.z = 0.0f;

// 设定初始推力、作用力和力矩
Bodies[i].vForces.x = 0.0f;
Bodies[i].vForces.y = 0.0f;
Bodies[i].vForces.z = 0.0f;

Bodies[i].vMoments.x = 0.0f;
Bodies[i].vMoments.y = 0.0f;
Bodies[i].vMoments.z = 0.0f;

// 将以局部坐标表示的速度置零
Bodies[i].vVelocityBody.x = 0.0f;
Bodies[i].vVelocityBody.y = 0.0f;
Bodies[i].vVelocityBody.z = 0.0f;

// 设定初始方向
iRoll = 0.0f;
iPitch = 0.0f;
if(configuration == 2)
     iYaw = 45.0f;
else
     iYaw = 0.0f;
Bodies[i].qOrientation = MakeQFromEulerAngles(iRoll,
                                              iPitch,
                                              iYaw);

// 设定与质量相关的性质
Bodies[i].fMass = 500.0f/(-g);
Ixx = Iyy = Izz = Bodies[i].fMass/12.0f *
                  (BLOCKSIZE*BLOCKSIZE + BLOCKSIZE*BLOCKSIZE);

Bodies[i].mInertia.e11 = Ixx;
Bodies[i].mInertia.e12 = 0;
Bodies[i].mInertia.e13 = 0;
Bodies[i].mInertia.e21 = 0;
Bodies[i].mInertia.e22 = Iyy;
Bodies[i].mInertia.e23 = 0;
Bodies[i].mInertia.e31 = 0;
Bodies[i].mInertia.e32 = 0;
Bodies[i].mInertia.e33 = Izz;

Bodies[i].mInertiaInverse = Bodies[i].mInertia.Inverse();

Bodies[i].fRadius = BLOCKSIZE/2; // 球形边界测试所需
```

```
            // 相对于重心的边界顶点（假设重心置中）
            Bodies[i].vVertexList[0].x = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[0].y = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[0].z = -BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[1].x = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[1].y = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[1].z = BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[2].x = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[2].y = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[2].z = BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[3].x = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[3].y = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[3].z = -BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[4].x = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[4].y = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[4].z = -BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[5].x = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[5].y = BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[5].z = BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[6].x = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[6].y = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[6].z = BLOCKSIZE/2.0f;

            Bodies[i].vVertexList[7].x = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[7].y = -BLOCKSIZE/2.0f;
            Bodies[i].vVertexList[7].z = -BLOCKSIZE/2.0f;
        }

    }
```

这个函数相当长，却很简单 —— 只是将RigidBody结构中所有的参数初始化，包括所有阵列中的物体。其中 Bodies[0]指的是模拟程序中的汽车。

传入函数的 configuration 参数用来指定要使用的场景。

这里要先说明，我假设所有物体的转动惯量（包括汽车的），大概是一个方柱体（方块）的转动惯量。所以可以使用第一章介绍的方柱体的惯性方程。

## 力与力矩

在前面已经提过，我会以简单的方式处理力的模拟。这里先列出所要考虑的4种主要的受力：

- 推力（只对汽车作用）

- 空气动阻力（线性的和角的）
- 重力
- 与地面的交互作用

所有力的影响都由 CalcObjectForces 函数处理：

```
//-----------------------------------------------------------------------//
// 此函数计算在任意的时间时作用在物体上的力和力矩
//-----------------------------------------------------------------------//
void      CalcObjectForces(void)
{
     Vector       Fb, Mb;
     Vector       vDragVector;
     Vector       vAngularDragVector;
     int          i, j;
     Vector       ContactForce;
     Vector       pt;
     int          check = NOCOLLISION;
     pCollision   pCollisionData; // 用在计算接触力
     Vector       FrictionForce;
     Vector       fDir;

     for(i=0; i<NUMBODIES; i++)
     {
          // 将力与力矩置零：
          Bodies[i].vForces.x = 0.0f;
          Bodies[i].vForces.y = 0.0f;
          Bodies[i].vForces.z = 0.0f;

          Bodies[i].vMoments.x = 0.0f;
          Bodies[i].vMoments.y = 0.0f;
          Bodies[i].vMoments.z = 0.0f;

          Fb.x = 0.0f;      Mb.x = 0.0f;
          Fb.y = 0.0f;      Mb.y = 0.0f;
          Fb.z = 0.0f;      Mb.z = 0.0f;

          // 定义推力向量，假设是作用在重心上
          if(i==0)
          {
               Thrust.x = 1.0f;
               Thrust.y = 0.0f;
               Thrust.z = 0.0f;
               Thrust *= ThrustForce;
               Fb += Thrust;
          }

          // 处理阻力
          vDragVector = -Bodies[i].vVelocityBody;
          vDragVector.Normalize();
          Fb += vDragVector * (Bodies[i].fSpeed * Bodies[i].fSpeed * rho *
```

```
            LINEARDRAGCOEFFICIENT * Bodies[i].fRadius *
            Bodies[i].fRadius);

        vAngularDragVector = -Bodies[i].vAngularVelocity;
        vAngularDragVector.Normalize();
        Mb += vAngularDragVector *
             (Bodies[i].vAngularVelocity.Magnitude() *
              Bodies[i].vAngularVelocity.Magnitude() * rho *
              ANGULARDRAGCOEFFICIENT * Bodies[i].fRadius *
              Bodies[i].fRadius);

        // 将力由局部坐标转换成全局坐标
        Bodies[i].vForces = QVRotate(Bodies[i].qOrientation, Fb);

        // 处理重力的影响
        Bodies[i].vForces.z += GRAVITY * Bodies[i].fMass;

        // 存储力矩
        Bodies[i].vMoments += Mb;

        // 处理与地面的接触
        Bodies[i].vAcceleration = Bodies[i].vForces / Bodies[i].fMass;
        Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
                                         (Bodies[i].vMoments -
                                         (Bodies[i].vAngularVelocity ^
                                         (Bodies[i].mInertia *
                                          Bodies[i].vAngularVelocity)));
        pCollisionData = Collisions;
        NumCollisions = 0;
        check = CheckGroundPlaneContacts(pCollisionData, i);
        if(check == CONTACT)
        {// 有接触...
            for(j=0; j<NumCollisions; j++)
            {
                ContactForce = (Bodies[i].fMass/NumCollisions * (
                               -Collisions[j].vRelativeAcceleration *
                                Collisions[j].vCollisionNormal)) *
                                Collisions[j].vCollisionNormal;
                FrictionForce = (ContactForce.Magnitude() *
                                 FRICTIONCOEFFICIENT) *
                                 Collisions[j].vCollisionTangent;
                Bodies[i].vForces += ContactForce;
                Bodies[i].vForces += FrictionForce;
                ContactForce = QVRotate(~Bodies[i].qOrientation,
                                        ContactForce);
                FrictionForce = QVRotate(~Bodies[i].qOrientation,
                                         FrictionForce);
               pt = Collisions[j].vCollisionPoint -
                    Bodies[i].vPosition;
               Bodies[i].vMoments += pt ^ ContactForce;
               Bodies[i].vMoments += pt ^ FrictionForce;
            }
        }
    }
}
```

此函数的大部分对你来说应该是非常熟悉的，除了接触力的部分以外。另外值得注意的，这个函数先进入循环计算阵列中所有的物体，所以调用一次CalcObjectForces就会求出所有物体的受力。

在进入循环时，先计算只对汽车作用的推力。我们假设此推力作用在汽车的重心上，并不会产生任何力矩。接着计算线性阻力和角阻力。得到这些值以后，接着将这些值加合得到以全局坐标表示的合力，并且考虑重力的影响。虽然这些计算已经被简化了，但是和前一个模拟程序所用的还是很像。

此函数最后计算物体与地面的接触力。

## 接触

接触力是指当物体实体接触但是尚未碰撞时之间的作用力。当两物体在接触中且接触点间的相对速度是朝对方移动的，则此两物体视为碰撞。在静接触中，接触点确实是刚好碰触到的，但两物体并没有朝对方移动。事实上，它们是朝着对方加速的，但是之间却存在一个力让物体不会穿过对方。这个力叫反作用力，此力作用在物体上虽然有同样的量值，方向却是相反的。

在模拟程序中，刚开始物体放置在地面上。所以在地面与物体之间会存在接触力，这个力会抵消物体所受的地心引力（重力）。当物体放在地面时，物体惟一受力的是重力，重力的大小是物体的质量乘以当地的重力加速度值。所以使物体不会穿过地面的力必须与重力的大小相等但方向相反。这样一来，这两个力互相抵消，物体就能静止在地面上。

事实上计算接触力并不容易，因为有许多复杂的因素：同时间会有多个接触力作用在相对于物体的重心不同的点上，而这些力不一定相同；此外物体也可能因重力以外的力而产生加速度，如角加速度。你可以在参考文献中找到一些用来处理接触力的方法。其中有些方法在接触点上加入暂时的弹力（惩罚法）；而有些方法（冲量法）使用冲量来处理，这和处理碰撞时用的方法一样，还有一些用分析法来处理接触力。这里我使用简化而粗略的方法来模拟，虽然这个方法并不是一个好方法。

以下是此方法的原理：先决定有哪些点发生接触；再判定是否为静接触；接着求出每个点的加速度；最后，假设各点的质量，而整个物体的质量是这些连接点的质量总和，便可求出所求的接触力。

决定两点是否够接近而产生接触和决定两个物体是否产生碰撞的方法很类似。我用一个新函数CheckGroundPlaneContacts来实现。此函数与之前用来检查碰撞的函数很相像。只不过它的用途是检查物体是否与地面接触。事实上这两个函数很类似，你也可以

将这两个函数写成同一个函数，并用一个输入参数来指定要做接触检查还是碰撞检查。为了不让你产生混淆，这里还是分成两个函数做个别的检查。以下是 CheckGround-PlaneContacts 的程序代码：

```
int       CheckGroundPlaneContacts(pCollision CollisionData, int body1)
{
     int          i;
Vector       v1[8];
Vector       tmp;
Vector       u, v;
float        d;
Vector       f[4];
Vector       vel1;
Vector       pt1;
Vector       Vr;
float        Vrn;
Vector       n;
int          status = NOCOLLISION;
Vector       Ar;
float        Arn;

// 旋转边界顶点并转换到全局坐标
for(i=0; i<8; i++)
{
     tmp = Bodies[body1].vVertexList[i];
     v1[i] = QVRotate(Bodies[body1].qOrientation, tmp);
     v1[i] += Bodies[body1].vPosition;
}

// 检查 body1 相对于地面的所有顶点
for(i=0; i<8; i++)
{
     u.x = 1.0f;
     u.y = 0.0f;
     u.z = 0.0f;
     v.x = 0.0f;
     v.y = 1.0f;
     v.z = 0.0f;
     tmp.x = 0.0f;
     tmp.y = 0.0f;
     tmp.z = 0.0f;
     d = CalcDistanceFromPointToPlane(v1[i], u, v, tmp);
     if(d < COLLISIONTOLERANCE)
     {
          // 计算相对速度
          pt1 = v1[i] - Bodies[body1].vPosition;

          vel1 = Bodies[body1].vVelocityBody +
                 (Bodies[body1].vAngularVelocity^pt1);

          vel1 = QVRotate(Bodies[body1].qOrientation, vel1);
```

```
n = u^v;
n.Normalize();

Vr = vel1;
Vrn = Vr * n;

if(fabs(Vrn) <= VELOCITYTOLERANCE) // 于静止状态
{
        // 检查相对加速度
        Ar = Bodies[body1].vAcceleration +
             (Bodies[body1].vAngularVelocity
              ^ (Bodies[body1].vAngularVelocity^pt1)) +
             (Bodies[body1].vAngularAcceleration^pt1);

        Arn = Ar * n;

        if(Arn <= 0.0f)
        {
             // 发生接触，填上数据结构并返回
             assert(NumCollisions < (NUMBODIES*8));
             if(NumCollisions < (NUMBODIES*8))
             {
                  CollisionData->body1 = body1;
                  CollisionData->body2 = -1;
                  CollisionData->vCollisionNormal = n;
                  CollisionData->vCollisionPoint = v1[i];
                  CollisionData->vRelativeVelocity = Vr;
                  CollisionData->vRelativeAcceleration = Ar;
                  CollisionData->vCollisionTangent = -(Vr-
                                                       ((Vr*n)*n));
                  // 注意切线向量的负值表示其方向与切线速度相反
                  // 稍后会用此向量处理摩擦力
                  CollisionData->vCollisionTangent.Normalize();
                  CollisionData++;
                  NumCollisions++;
                  status = CONTACT;
             }
        }
   }
}

        return status;
}
```

到目前为止，我用一系列的硬点来表示了碰撞或接触，而这些点是以局部坐标的形式存在的，所以此函数会先将表示物体的目前方位转换成全局坐标。

接着要检查每一个顶点（或硬点）是否与地面够接近而产生了接触。这是函数CalcDistanceFromPointToPlane的工作。***v***1是被检查的点，而向量***u***，***v***和tmp则表示地面。向量***u***和***v***是在平面中与坐标轴平行的向量，而tmp可以是平面上的任意点，

在这个范例中使用的是原点。CalcDistanceFromPointToPlane是一个简单的函数，负责计算点到此平面的垂直距离，也就是最短距离。函数如下：

```
float       CalcDistanceFromPointToPlane(Vector pt, Vector u, Vector v,
Vector ptOnPlane)
{
     Vector n = u^v;
     Vector PQ = pt - ptOnPlane;
     n.Normalize();

     return PQ*n;
}
```

如你所见，此函数先求出地面的法向量，也就是先求向量*u*，*v*的外积，再将它正规化。接着从任意点（这里用的是原点）到目标点构成一个向量。最后，计算此向量与法向量的内积即可求出平面的投影距离，也就是点到平面的距离。最后将这个值返回。

如果CalcDistanceFromPointToPlane返回的距离小于碰撞允许值，CheckGround-PlaneContacts接着会做两个重要的检查来决定此点是否发生静接触：

相对速度

相对速度为0或小于某个最小速度临界值。

相对加速度

相对加速度必须显示物体正往地面加速。

如果任何一个检查都是错的，那就表示没有静接触发生。如果相对速度显示物体正朝着地面移动，就表示发生碰撞，不过要等一下再处理。如果相对速度或相对加速度显示物体正移动或加速离开地面，就表示没有碰撞或静接触发生，则不需要处理。

之前已示范过如何计算刚体上一个点的相对速度和加速度，所以这里就不再浪费篇幅。但我要再次提醒：当为刚体上的点进行此类计算时，要记得计算线性速度和线性加速度，以及角速度和角加速度。

如果检查发现某个顶点正发生接触，必须要将发生接触的数据存在一个碰撞的结构中。我用同一个结构来存储碰撞数据和接触数据，所以不要被结构的名字搞混了。另外，也用阵列存储这些结构，因此这个程序可以处理多个物体间的碰撞。以下是这个结构和实体的阵列：

```
typedef struct      _Collision {
     int                  body1; // 代表body 1 (-1是指在地面上)
     int                  body2; // 代表body 2 (-1是指在地面上)
     Vector               vCollisionNormal; // body2平面向外的法向量
     Vector               vCollisionPoint;  // 以全局坐标计的接触点
```

```
    Vector                  vRelativeVelocity;// 相对速度
    Vector                  vRelativeAcceleration; // 相对加速度
    Vector                  vCollisionTangent; // 与相对速度切于接触面方向相反的
                                               // 切线向量
}       Collision, *pCollision;

Collision       Collisions[NUMBODIES*8];
int               NumCollisions = 0;
```

在Collision结构中每个成员都有注释解释。因为一个物体中定义8个碰撞点，所以一共有NUMBODIES*8个可能碰撞或接触的情形发生。NumCollisions是用来记录目前有多少的碰撞或接触发生的变量。

回到CalcObjectForces中，可以看到最底下有一个检查与计算接触的地方。这是CalcObjectForces在计算重力的影响和存储合力矩在物体结构中之后的部分。为了方便，这里只列出部分程序代码：

```
.
.
.
// 处理与地面的接触
Bodies[i].vAcceleration = Bodies[i].vForces / Bodies[i].fMass;
Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
                                 (Bodies[i].vMoments -
                                 (Bodies[i].vAngularVelocity ^
                                 (Bodies[i].mInertia *
                                  Bodies[i].vAngularVelocity)));

pCollisionData = Collisions;
NumCollisions = 0;
check = CheckGroundPlaneContacts(pCollisionData, i);
if(check == CONTACT)
{ // 发生接触....
     for(j=0; j<NumCollisions; j++)
     {
          ContactForce = (Bodies[i].fMass/NumCollisions * (
                         -Collisions[j].vRelativeAcceleration *
                          Collisions[j].vCollisionNormal)) *
                          Collisions[j].vCollisionNormal;
          FrictionForce = (ContactForce.Magnitude() *
                           FRICTIONCOEFFICIENT) *
                           Collisions[j].vCollisionTangent;
          Bodies[i].vForces += ContactForce;
          Bodies[i].vForces += FrictionForce;
          ContactForce = QVRotate(~Bodies[i].qOrientation,
                                  ContactForce);
          FrictionForce = QVRotate(~Bodies[i].qOrientation,
                                   FrictionForce);
          pt = Collisions[j].vCollisionPoint -
               Bodies[i].vPosition;
          Bodies[i].vMoments += pt^ContactForce;
```

```
                Bodies[i].vMoments += pt^FrictionForce;
        }
    }
    .
    .
    .
```

在做任何检查之前，物体上所有线性加速度和角加速度都会先被求出并存储。这个方法类似于飞行模拟中的StepSimulation，等一下你也能看到。如果存储CheckGround-PlaneContacts返回值的变量check显示发生了接触，所有的接触力会被计算并存储起来。

ContactForce是在平面与接触点之间的接触正向力，并等于接触粒子的质量乘以粒子的加速度。这里假设物体是由相等大小的粒子所组成的，这些粒子位于之前定义的硬点上。这里也必须考虑摩擦力的影响，主要是作用在接触平面的切线上。FrictionForce就是由摩擦产生的力，量值为正向力的大小乘以摩擦系数再乘以单位切线向量。这就得到接触点上与切线速度反方向的摩擦力。请注意CheckGroundPlaneContacts中的碰撞切线加上了负号，表示它与接触点的切线速度方向相反。也请注意因为我们用的是简单的摩擦力模型，所以只考虑动摩擦力而忽略静摩擦力。

当计算出这两个力之后，如往常一样将它们套用到物体上，也就是将这两个力累加到此物体的数据结构中，并计算合力矩。

# 积分函数

接着来处理运动方程的积分，首先介绍StepSimulation。此函数大部分都和前面范例的StepSimulation函数一样；但这里多加了一个循环逐一对物体的阵列做检查。以下是这个新的函数：

```
//-----------------------------------------------------------------------
//
// 使用欧拉法
//-----------------------------------------------------------------------
//
void    StepSimulation(float dtime)
{
     Vector  Ae;
     int             i;
     float           dt = dtime;

     // 计算所有物体上的合力和合力矩
     CalcObjectForces();

     // 积分
     for(i=0; i<NUMBODIES; i++)
```

```
{
        // 计算全局坐标上的加速度:
        Ae = Bodies[i].vForces / Bodies[i].fMass;
        Bodies[i].vAcceleration = Ae;

        // 计算全局坐标上的速度:
        Bodies[i].vVelocity += Ae * dt;

        // 计算全局坐标上的位置:
        Bodies[i].vPosition += Bodies[i].vVelocity * dt;

        // 处理旋转:
        float           mag;
        Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
                                         (Bodies[i].vMoments -
                                         (Bodies[i].vAngularVelocity^
                                         (Bodies[i].mInertia *
                                          Bodies[i].vAngularVelocity)));

        Bodies[i].vAngularVelocity += Bodies[i].vAngularAcceleration *
                                      dt;

        // 计算新的旋转四元数:
        Bodies[i].qOrientation += (Bodies[i].qOrientation *
                                   Bodies[i].vAngularVelocity) *
                                   (0.5f * dt);

        // 正规化方位四元数:
        mag = Bodies[i].qOrientation.Magnitude();
        if (mag != 0)
             Bodies[i].qOrientation /= mag;

        // 计算局部坐标上的速度:
        Bodies[i].vVelocityBody = QVRotate(~Bodies[i].qOrientation,
                                           Bodies[i].vVelocity);

        // 计算速度:
        Bodies[i].fSpeed = Bodies[i].vVelocity.Magnitude();

        // 计算出欧拉角
        Vector u;

        u = MakeEulerAnglesFromQ(Bodies[i].qOrientation);
        Bodies[i].vEulerAngles.x = u.x;   // 滚转
        Bodies[i].vEulerAngles.y = u.y;   // 俯仰
        Bodies[i].vEulerAngles.z = u.z;   // 偏转
    }

    // 处理碰撞
    if(CheckForCollisions() == COLLISION)
         ResolveCollisions();
}
```

此函数使用欧拉法是为了简化程序代码，并不是因为它是数值稳定上的最佳选择。

StepSimulation先调用CalcObjectForces更新物体上的力和力矩。接着进入一个循环，对所有的物体积分并计算出速度、位置和方位。所有执行积分的程序代码都来自飞行模拟的范例，所以你应该很熟悉了。

在所有物体上的力和积分都完成后，StepSimulation最后会处理碰撞，也就是调用CheckForCollisions，如果发生碰撞则再调用ResolveCollisions函数。请记住，这些碰撞已被考虑在内，而接触力的影响已经被计算在运动方程的积分中了。

## 碰撞反应

正如你所知道的，处理碰撞有两部分：碰撞侦测和碰撞反应。我也说过，虽然碰撞侦测并不是物理学的一部分，但它是模拟程序处理碰撞反应的重要环节。进行碰撞侦测时必须在速度与精确度的平衡之间取舍。你可以用很多的几何模型计算，来得到非常精确的碰撞侦测，但是这种方法却会使模拟程序执行速度变慢，而且这样的方法可能会产生不需要的结果。不管你用哪一种方法，都要得到所需要的数据。这些数据都在之前所讨论的Collision结构中。

虽然这个范例中用的碰撞侦测方法并不是最完美的——因为它不够正确也不能侦测穿透——但是却很简单且符合我们的需求，也就是能处理3D空间中的碰撞反应。本章不会详细介绍碰撞侦测的程序代码，因为这与之前第十三章所用的类似，而且可以从O'Reilly的网站下载。我只会概括地介绍碰撞侦测的程序，至少让你能了解碰撞数据的来源。

基本上，我用球形边界来检查物体间是否有碰撞发生。如果检查发现有可能发生碰撞，就对每个硬点和其他物体的方块边界平面做检查。如果发现碰撞点在碰撞允许值之内，接着检查可能发生的碰撞点之间的相对速度，以决定它们是否相互移动。如果是，就是发生了碰撞，并存储碰撞数据，包括发生碰撞物体的阵列索引、碰撞法向量和切线向量、以全局坐标表示的碰撞点坐标，以及碰撞点之间的相对速度。这些碰撞数据存储在和前一节一样的碰撞结构中，可用来处理碰撞反应。然而，碰撞资料会覆盖之前的接触数据，且不再需要接触数据了（至少在下次计算力之前不需要，因为新的接触数据会重新产生）。

现在来处理碰撞反应。因为我们以冲量为基础来处理碰撞反应，所以需要逐一计算碰撞数据结构并套用适当的冲量。ResolveCollisions是用来处理碰撞反应的函数：

```
void ResolveCollisions(void)
{
     int      i;
     Vector   pt1, pt2;
```

```
float    j;
float    fCr = COEFFICIENTOFRESTITUTION;
int      b1, b2;
float    Vrt;
float    mu = FRICTIONCOEFFICIENT;

for(i=0; i i++)
{
     b1 = Collisions[i].body1;
     b2 = Collisions[i].body2;

     if(b2 != -1) // 不在地面上
     {
          pt1 = Collisions[i].vCollisionPoint - Bodies[b1].vPosition;
          pt2 = Collisions[i].vCollisionPoint - Bodies[b2].vPosition;

          // 计算冲量
          j = (-(1+fCr) * (Collisions[i].vRelativeVelocity *
               Collisions[i].vCollisionNormal)) /
               ( (1/Bodies[b1].fMass + 1/Bodies[b2].fMass) +
               (Collisions[i].vCollisionNormal * ( ( (pt1^
                Collisions[i].vCollisionNormal) *
                Bodies[b1].mInertiaInverse )^pt1) ) +
               (Collisions[i].vCollisionNormal * ( ( (pt2^
                Collisions[i].vCollisionNormal) *
                Bodies[b2].mInertiaInverse )^pt2) ) );

          Vrt = Collisions[i].vRelativeVelocity *
                Collisions[i].vCollisionTangent;

          if(fabs(Vrt) > 0.0) {
               Bodies[b1].vVelocity += ( (j *
                         Collisions[i].vCollisionNormal) + ((mu * j)
                         * Collisions[i].vCollisionTangent) ) /
                         Bodies[b1].fMass;

               Bodies[b1].vAngularVelocity += (pt1^((j *
                         Collisions[i].vCollisionNormal) + ((mu * j)
                         * Collisions[i].vCollisionTangent))) *
                         Bodies[b1].mInertiaInverse;

               Bodies[b2].vVelocity -= ((j *
                         Collisions[i].vCollisionNormal) + ((mu * j)
                         * Collisions[i].vCollisionTangent)) /
                         Bodies[b2].fMass;

                    Bodies[b2].vAngularVelocity -= (pt2^((j *
                         Collisions[i].vCollisionNormal) + ((mu * j)
                         * Collisions[i].vCollisionTangent))) *
                         Bodies[b2].mInertiaInverse;

     } else {
          // 套用冲量
          Bodies[b1].vVelocity += (j *
```

```
                    Collisions[i].vCollisionNormal) /
                    Bodies[b1].fMass;

Bodies[b1].vAngularVelocity += (pt1^((j *
                    Collisions[i].vCollisionNormal)) *
                    Bodies[b1].mInertiaInverse;

Bodies[b2].vVelocity -= (j *
                    Collisions[i].vCollisionNormal) /
                    Bodies[b2].fMass;

            Bodies[b2].vAngularVelocity -= (pt2^(j *
                    Collisions[i].vCollisionNormal)) *
                    Bodies[b2].mInertiaInverse;
        }
    } else { // 如果在地面上
        fCr = COEFFICIENTOFRESTITUTIONGROUND;
        pt1 = Collisions[i].vCollisionPoint - Bodies[b1].vPosition;

        // 计算冲量
        j = (-(1+fCr) * (Collisions[i].vRelativeVelocity *
             Collisions[i].vCollisionNormal)) /
             ( (1/Bodies[b1].fMass) +
             (Collisions[i].vCollisionNormal * ( ( (pt1^
              Collisions[i].vCollisionNormal) *
              Bodies[b1].mInertiaInverse )^pt1)));
        Vrt = Collisions[i].vRelativeVelocity *
              Collisions[i].vCollisionTangent;
        if(fabs(Vrt) > 0.0) {
             Bodies[b1].vVelocity += ( (j *
                    Collisions[i].vCollisionNormal) + ((mu * j) *
                    Collisions[i].vCollisionTangent) ) /
                    Bodies[b1].fMass;

             Bodies[b1].vAngularVelocity += (pt1^((j *
                    Collisions[i].vCollisionNormal) + ((mu * j) *
                    Collisions[i].vCollisionTangent))) *
                    Bodies[b1].mInertiaInverse;

        } else {
             // 套用冲量
             Bodies[b1].vVelocity += (j *
                    Collisions[i].vCollisionNormal) /
                    Bodies[b1].fMass;

             Bodies[b1].vAngularVelocity += (pt1^((j *
                    Collisions[i].vCollisionNormal)) *
                    Bodies[b1].mInertiaInverse;
        }
    }
}
```

此函数看来好像很复杂，因为其中有两个主要的部分，分别用来处理物体之间和物体与地面之间的碰撞。

函数一开始就进入循环逐一检查碰撞结构阵列。因为每次碰撞的信息已经被计算出来，并存储在阵列中，所以函数惟一要做的就是计算冲量并且套用到物体上。

在两个物体间碰撞的情况下，函数先计算冲量并检查相对切线速度的大小是否大于0。如果是，则当更新物体的线性及角速度时，需使用有摩擦力的冲量公式；否则就使用不含摩擦力的公式。这两个公式都可以在第五章中找到，如果已经忘了，建议你先翻回去复习。

在物体与地面的碰撞中，也是先计算冲量再检查相对切线速度来更新物体。惟一显著的不同是计算冲量并更新碰撞地面的物体的方式。首先，计算冲量时，我假设地面有无限大的质量与无限大的转动惯量，于是冲量公式中，其分母为Bodies[b2]的质量或转动惯量的项目会趋近于0，可忽略不计。接着，因为地面是静止的，所以不需要更新速度，所以只需要更新Bodies[b1]。请注意我的碰撞侦测假设当物体撞到地面时，地面是body2，在Collisions阵列中索引值被设为-1。

对于存储在Collisions阵列中的每一组碰撞数据，都会执行这些计算，一直到阵列索引值NumCollisions-1（包含此物体）。等到所有的计算完毕，函数就将值返回并执行下个时间间隔。

## 参数调整

我必须承认，在所有的数值设定好之后，第一次执行这个程序竟然不能正常运作——应该说结果相当不真实。主要的原因是当初假设的参数，如碰撞反应的恢复系数、摩擦力系数、时间间隔的大小等都有问题。必须逐一对这些参数调整使这个程序能正确地运作。本书一开始说过，参数调整是模拟程序中重要的一环。你需要在正确性和真实性，也就是数值稳定与速度之间取得平衡。在不同的模拟程序中，当然要根据模拟程序的性质来决定哪一项比较重要。

这个模拟程序不考虑执行的速度，又因为在碰撞侦测中并没有检查物体的穿透，所以也忽略物体穿透的考虑。因此用较小的时间间隔和较大的碰撞允许值。同时我选择欧拉法，而不是改良型欧拉法或Runge-Kutta 法。我也发现要增加摩擦力系数以提供足够的阻力从而维持数值稳定。

为了方便，所有需调整的参数都写成全局的定义常数。如下：

```
#define     GRAVITY                         -32.174f
#define     LINEARDRAGCOEFFICIENT           5.0f
#define     ANGULARDRAGCOEFFICIENT          1200.0f
#define     COLLISIONTOLERANCE              0.9f
#define     COEFFICIENTOFRESTITUTION        0.5f
#define     COEFFICIENTOFRESTITUTIONGROUND  0.025f
#define     VELOCITYTOLERANCE               0.05f
#define     FRICTIONCOEFFICIENT             0.9f
```

你可以调整这些参数并看看会发生什么事。不过不能将某个值改变太多，否则会造成不可预期的结果。例如，如果把碰撞允许值改得太小，会发现物体间互相穿过（因为程序没有处理穿透的情况）。如果将恢复系数增加并且将阻力因素减小，会发现物体间就像失去控制般互相弹跳，这是由于数值不稳定的缘故。

这里有几项建议让你改进模拟程序。首先，实现较好的碰撞侦测系统。这里的“较好”并不是说要你检查每一点对多面体上每一面的碰撞，或是三角形与三角形的相交。我指的是可以加入穿透检查，在第十三章也介绍过，或者是在碰撞侦测时，物体（非方块形的物体）多加入一些硬点。在“参考文献”中有一些详细讨论这些主题的参考资料。

接着，建议你使用在第十一章所用的改良型欧拉法来代替欧拉法。这样可以增加数值的稳定，就不用再使用较大的时间间隔。

最后，我建议你可以调整力的计算公式以适合欲模拟的系统。可以用类似于前一章所用的技巧 —— 例如在飞行模拟程序中，示范过如何利用质点集合计算质量特性，并示范如何正确地处理升力和阻力。

# 第十七章

# 粒子系统

本章讨论的是与刚体模拟稍微有点不同的主题，我要介绍如何实现柔体模拟程序。更具体地说，本章将实现旗杆上的旗帜在风中飘扬的模拟。图 17-1 是这个模拟程序的快照。这个范例的目的不是要教你如何明确地处理布料，而是要教你如何利用粒子和弹簧的集合来代替刚体模拟。粒子系统可以用来模拟大范围的东西，例如布料、烟雾和火。使用粒子系统的好处在于比刚体容易处理，此外也不用考虑旋转和角运动方程。不只是简化力的计算和积分，连碰撞反应也很简单 —— 只需处理线性冲量。

当第一次执行这个程序时，你会看到一面旗子（如图 17-1 所示）在轻柔的风中摆动。你可用数字键 1 到 6 来控制风的强度，1 是最弱而 6 是最强。按下 0 会呈现无风状态，将看到旗子因为本身的重量而垂下。按下 R 键会使旗子脱离旗杆而掉在地面上。如果这时的风力是 0，旗子会垂直掉落，否则将会随着风被吹离旗杆。也可以用方向键来改变观看的角度，上下键会使摄影机前后移动，左右键会使摄影机向左右旋转。

这里再强调一次，你可以在 O'Reilly 网站上找到本范例的所有可执行文件和源代码。本范例的程序代码大部分与之前的范例相同或类似，所以不会再重复介绍，我只会介绍不同的部分。

## 模型

为了模拟范例中的旗子，我利用格状排列的粒子，并将这些粒子以弹簧相连。这些弹簧也是结构的要素，用来抵抗粒子之间的受力并使粒子能保持连接状态。图 17-2 是显示这些格状排列粒子及连接弹簧的网格图。

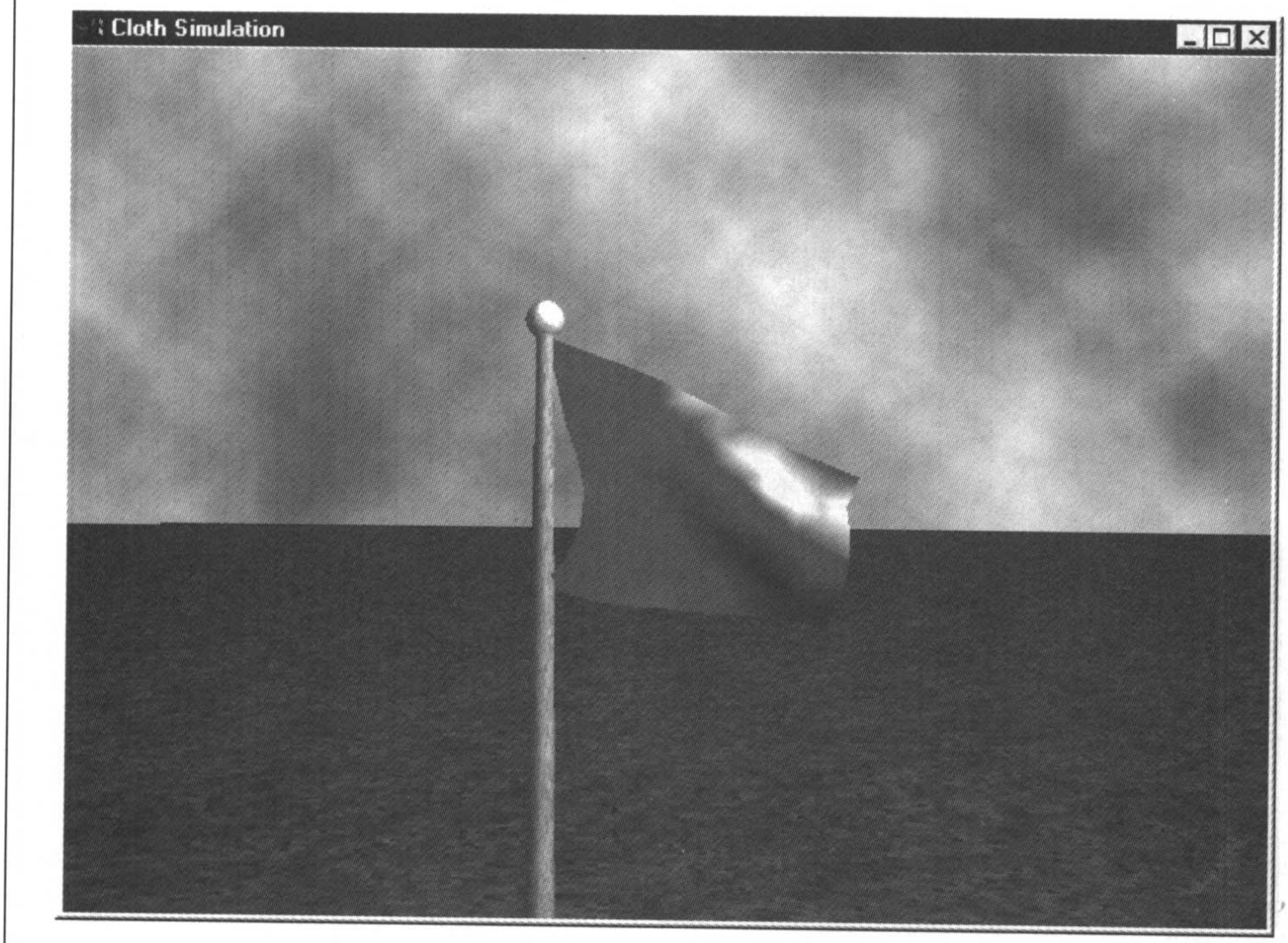



图 17-1：布料的模拟

每一个网格旗上的线段代表一个弹簧阻尼元件，而弹簧之间交叉的节点是一个粒子。这些线段是用第三章所教的弹簧阻尼方程来制作模型的。刚开始的水平及垂直弹簧构成旗子基本的结构，而斜的弹簧会抵抗剪力（shear force）并使布料能更强韧。如果没有这些剪力弹簧，旗子就会被风力拉长。请注意这些斜的弹簧交叉的节点并不是粒子。

为了要处理粒子的状态，在模拟程序中用 Particle 结构来代表粒子的状态，并用阵列来存储每个粒子。实际上我用的是一个多维的阵列，因为当设定连接弹簧时，这样比较容易理解粒子的格状位置。以下是 Particle 结构和全局阵列的程序代码：

```
typedef     struct _Particle {
 float      fMass;
 float      fInvMass;
 Vector     vPosition;
 Vector     vVelocity;
 Vector     vAcceleration;
 Vector     vForces;
 BOOL       bLocked;
} Particle, *pParticle;
```

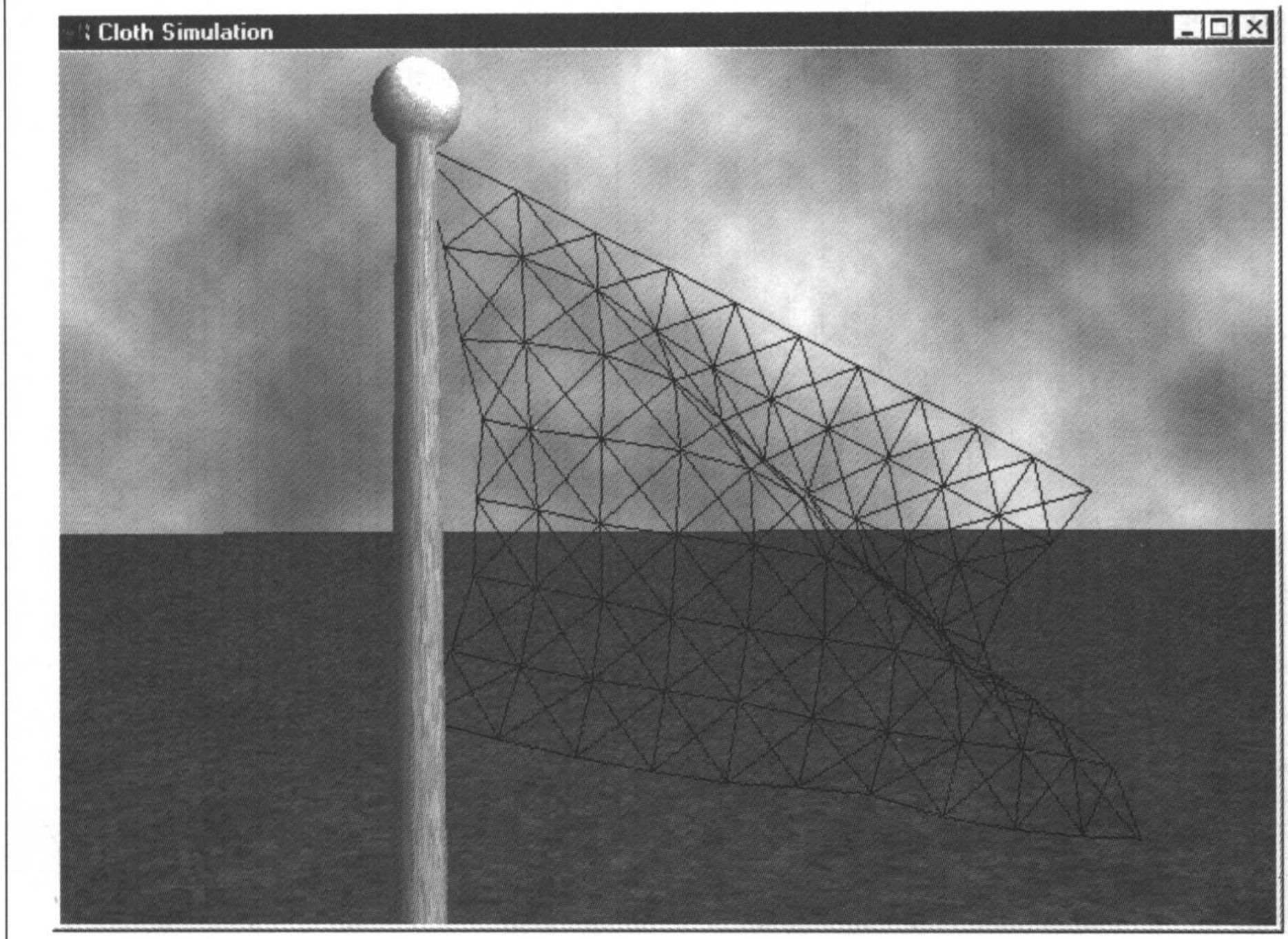



图 17-2：粒子 / 弹簧系统

```
// NUMROWS 是每一列粒子之间空间的数目
// NUMCOLUMNS 是每一行粒子之间空间的数目

Particle      Particles[NUMROWS+1][NUMCOLUMNS+1];
```

在 Particle结构中的每一个成员都是很容易了解而且你应该很熟悉的。基本上，这些参数包含粒子的质量特性，以及于给定瞬间的位置、速度、加速度、合力。

其中你可能不熟悉的应该是 bLocked。这个参数用来指定这个粒子是否固定。如果bLocked是true，就表示这个粒子是固定的，也就是对运动方程积分时可以忽略。在这个程序开始时，我将旗子左上及左下的粒子设定为不能移动，看起来像把旗子绑在旗杆上一样。当你按下R键时，bLocked值就被设为false，这时旗子就可能会被吹离旗杆。

模拟程序中用了另一个结构来存储弹簧的数据，并将所有的结构存在阵列中：

```
typedef struct _ParticleRef {
     int      r;          // 列索引
     int      c;          // 行索引
} ParticleRef;
```

```
typedef     struct      _Spring {
     ParticleRef     p1;          // 所连接的1号粒子
     ParticleRef     p2;          // 所连接的2号粒子
     float           k;           // 伸展的弹簧常数
     float           d;           // 阻尼系数
     float           L;           // 弹簧静止时的长度
} Spring, *pSpring;

#define NUMSTRUCTURALSPRINGS (NUMCOLUMNS*(NUMROWS+1)  +
                              NUMROWS*(NUMCOLUMNS+1)  +
                              NUMCOLUMNS*NUMROWS*2)

Spring                       StructuralSprings[NUMSTRUCTURALSPRINGS];
```

Spring 结构中的成员 p1 和 p2 是网格中粒子的参考指针。它们的资料类型是 ParticleRef，此类型存储了粒子在 Particle阵列中的行、列位置。其他三个参数 $k$，$d$，$L$，保存此弹簧的数据，以备将来用在计算每一对粒子之间所受的弹力。$k$，$d$，$L$分别是弹簧常数、阻尼系数和弹簧静止时的长度。

因为旗子的几何形状在模拟进行中会不断地改变，所以必须在每个时间间隔根据粒子的状态重新建立顶点和面的数据。为了记录旗子的几何形状，我用了两个全局的阵列来记录顶点和面的数据，以便让 Direct3D 来建立 3D 对象。以下是这些阵列：

```
unsigned int        ClothFaces[NUMFACES*3*2];
float               ClothVertices[NUMVERTICES*3*2];
```

每个面都是由三个节点构成的三角形。请注意这里用两倍的顶点和面来表示这个旗子（将阵列大小乘以 2）。这样做的原因是由于程序使用 Direct3D Retained Mode，在这种模式下会自动进行背面去除（back face culling）。所以我必须建立旗子的两面，这样才能看到旗子的两面。如果用的是立即模式（immediate mode）或 OpenGL 函数库，可以把背面去除的功能关闭。

欲设定模拟程序的开头，以下的 Initialize函数会将所有粒子、弹簧和布料几何结构的数据初始化：

```
#define     MASSPERFACE     (CLOTHMASS/(float) NUMFACES)
#define     CSTEP           ((float) CLOTHWIDTH / (float) NUMCOLUMNS)
#define     RSTEP           ((float) CLOTHHEIGHT / (float) NUMROWS)
void     Initialize(void)
{
     int             r, c;
     float           f;
     unsigned int    *faceVertex;
     float           *vertices;
     Vector          L;
     int             count;
     int             n;
```

```
for(r=0; r<=NUMROWS; r++)
{
     for(c=0; c<=NUMCOLUMNS; c++)
     {
          // 计算粒子的质量
          if((r == 0) && (c == 0))
               f = 1;
          else if((r == NUMROWS) && (c == 0))
               f = 2;
     else if((r == 0) && (c == NUMCOLUMNS))
          f = 2;
     else if((r == NUMROWS) && (c == NUMCOLUMNS))
          f = 1;
     else if(((r == 0) || (r == NUMROWS)) && ((c != 0) &&
               (c != NUMCOLUMNS)))
          f = 3;
     else
          f = 6;

     Particles[r][c].fMass = (f * MASSPERFACE) / 3;
     Particles[r][c].fInvMass = 1 / Particles[r][c].fMass;

     // 设定粒子的初始位置
     Particles[r][c].vPosition.x = c * CSTEP;
     Particles[r][c].vPosition.y = (CLOTHHEIGHT - (r * RSTEP)) +
                                   YOFFSET;
     Particles[r][c].vPosition.z = 0.0f;

     // 将初始速度和力设为0
     Particles[r][c].vVelocity.x = 0.0f;
     Particles[r][c].vVelocity.y = 0.0f;
     Particles[r][c].vVelocity.z = 0.0f;

     Particles[r][c].vAcceleration.x = 0.0f;
     Particles[r][c].vAcceleration.y = 0.0f;
     Particles[r][c].vAcceleration.z = 0.0f;

     Particles[r][c].vForces.x = 0.0f;
     Particles[r][c].vForces.y = 0.0f;
     Particles[r][c].vForces.z = 0.0f;
     if((c == 0) && (r == 0 || r == NUMROWS))
          Particles[r][c].bLocked = TRUE;
     else
          Particles[r][c].bLocked = FALSE;
     }
}

vertices = ClothVertices;
for(r=0; r<=NUMROWS; r++)
{
     for(c=0; c<=NUMCOLUMNS; c++)
     {
          // 设定顶点
          *vertices = Particles[r][c].vPosition.x; vertices++;
```

```
                *vertices = Particles[r][c].vPosition.y; vertices++;
                *vertices = Particles[r][c].vPosition.z; vertices++;
        }
}
for(r=0; r<=NUMROWS; r++)
{
        for(c=0; c<=NUMCOLUMNS; c++)
        {
                // 设定顶点
                *vertices = Particles[r][c].vPosition.x; vertices++;
                *vertices = Particles[r][c].vPosition.y; vertices++;
                *vertices = Particles[r][c].vPosition.z; vertices++;
        }
}

faceVertex = ClothFaces;
for(r=0; r<NUMROWS; r++)
{
        for(c=0; c<=NUMCOLUMNS; c++)
        {
                // 设定面
                if(c == 0)
                {
                        *faceVertex = ((NUMCOLUMNS+1)*r) + c;
                        faceVertex++; // 顶点1

                        *faceVertex = ((NUMCOLUMNS+1)*r) + (c+1);
                        faceVertex++; // 顶点2

                        *faceVertex = ((NUMCOLUMNS+1)*r) + (NUMCOLUMNS+1) + c;
                        faceVertex++; // 顶点3
                } else if(c == NUMCOLUMNS) {
                        *faceVertex = ((NUMCOLUMNS+1)*r) + c;
                        faceVertex++; // 顶点1

                        *faceVertex = ((NUMCOLUMNS+1)*r) + (NUMCOLUMNS+1) + c;
                        faceVertex++; // 顶点2

                        *faceVertex = ((NUMCOLUMNS+1)*r) + (NUMCOLUMNS+1) +
                                        (c-1);
                        faceVertex++; // 顶点3
                } else {
                        *faceVertex = ((NUMCOLUMNS+1)*r) + c;
                        faceVertex++; // 顶点1

                        *faceVertex = ((NUMCOLUMNS+1)*r) + (NUMCOLUMNS+1) + c;
                        faceVertex++; // 顶点2

                        *faceVertex = ((NUMCOLUMNS+1)*r) + (NUMCOLUMNS+1) +
                                        (c-1);
                        faceVertex++; // 顶点3

                        *faceVertex = ((NUMCOLUMNS+1)*r) + c;
                        faceVertex++; // 顶点1
```

```
                *faceVertex = ((NUMCOLUMNS+1)*r) + (c+1);
                faceVertex++; // 顶点 2

                *faceVertex = ((NUMCOLUMNS+1)*r) + (NUMCOLUMNS+1) + c;
                faceVertex++; // 顶点 3
            }
        }
    }

    for(r=0; r<NUMROWS; r++)
    {
        for(c=0; c<=NUMCOLUMNS; c++)
        {
            // 设定面
            if(c == 0)
            {
                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (NUMCOLUMNS+1) + c;
                faceVertex++; // 顶点 3

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (c+1);
                faceVertex++; // 顶点 2

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) + c;
                faceVertex++; // 顶点 1
            } else if(c == NUMCOLUMNS) {
                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (NUMCOLUMNS+1) + (c-1);
                faceVertex++; // 顶点 3

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (NUMCOLUMNS+1) + c;
                faceVertex++; // 顶点 2

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) + c;
                faceVertex++; // 顶点 1

            } else {
                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (NUMCOLUMNS+1) + (c-1);
                faceVertex++; // 顶点 3

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (NUMCOLUMNS+1) + c;
                faceVertex++; // 顶点 2

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) + c;
                faceVertex++; // 顶点 1

                *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                              (NUMCOLUMNS+1) + c;
                faceVertex++; // 顶点 3
```

```
                    *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) +
                                  (c+1);
                    faceVertex++; // 顶点2

                    *faceVertex = NUMVERTICES + ((NUMCOLUMNS+1)*r) + c;
                    faceVertex++; // 顶点1

               }
          }
     }

     // 建立代表布料的D3D对象
     CreateCloth("test.bmp", ClothFaces, NUMFACES*2, ClothVertices,
                 NUMVERTICES*2, FALSE);
     // 设定构造的弹簧，并连接邻近的粒子
     count = 0;
     n = NUMSTRUCTURALSPRINGS;
     for(r=0; r<=NUMROWS; r++)
     {
          for(c=0; c<=NUMCOLUMNS; c++)
          {
               if(c<NUMCOLUMNS)
               {
                    StructuralSprings[count].p1.r = r;
                    StructuralSprings[count].p1.c = c;
                    StructuralSprings[count].p2.r = r;
                    StructuralSprings[count].p2.c = c+1;
                    StructuralSprings[count].k = SPRINGTENSIONCONSTANT;
                    StructuralSprings[count].d = SPRINGDAMPINGCONSTANT;
                    L = Particles[r][c].vPosition -
                        Particles[r][c+1].vPosition;
                    StructuralSprings[count].L = L.Magnitude();
                    count++;
               }
               if(r<NUMROWS)
               {
                    StructuralSprings[count].p1.r = r;
                    StructuralSprings[count].p1.c = c;
                    StructuralSprings[count].p2.r = r+1;
                    StructuralSprings[count].p2.c = c;
                    StructuralSprings[count].k = SPRINGTENSIONCONSTANT;
                    StructuralSprings[count].d = SPRINGDAMPINGCONSTANT;
                    L = Particles[r][c].vPosition -
                        Particles[r+1][c].vPosition;
                    StructuralSprings[count].L = L.Magnitude();
                    count++;
               }
               if(r<NUMROWS && c<NUMCOLUMNS)
               {
                    StructuralSprings[count].p1.r = r;
                    StructuralSprings[count].p1.c = c;
                    StructuralSprings[count].p2.r = r+1;
                    StructuralSprings[count].p2.c = c+1;
```

```
            StructuralSprings[count].k = SPRINGSHEARCONSTANT;
            StructuralSprings[count].d = SPRINGDAMPINGCONSTANT;
            L = Particles[r][c].vPosition -
                Particles[r+1][c+1].vPosition;
            StructuralSprings[count].L = L.Magnitude();
            count++;
        }
        if(c>0 && r<NUMROWS)
                {
                    StructuralSprings[count].p1.r = r;
                    StructuralSprings[count].p1.c = c;
                    StructuralSprings[count].p2.r = r+1;
                    StructuralSprings[count].p2.c = c-1;
                    StructuralSprings[count].k = SPRINGSHEARCONSTANT;
                    StructuralSprings[count].d = SPRINGDAMPINGCONSTANT;
                    L = Particles[r][c].vPosition -
                        Particles[r+1][c-1].vPosition;
                    StructuralSprings[count].L = L.Magnitude();
                    count++;
                }
            }
        }

        WindVector.x = 10.0;
        WindVector.y = 0.0;
        WindVector.z = 1.0;

    }
```

在函数中第一个嵌套for循环是将阵列中粒子的数据逐一初始化。正如之前所说的，每个粒子的位置被计算成格子状排列。而每个粒子的质量是各面（由三个粒子组成）质量的三分之一。

循环接下来执行到调用CreateCloth函数之前的程序代码，设定将传入Direct3D对象的旗面和顶点的数据。CreateCloth使用Direcr3D建立旗子对象。本章未列出与Direct3D相关的程序代码，读者可从O'Reilly的网站上自行下载。

最后的嵌套循环设定组成旗子构造的弹簧。在本范例中这个程序很容易理解，因为大部分弹簧的数据是固定的，且由于Particle阵列是多维的，而粒子的索引值对应到它在网格中的行数和列数，因此利用此特点很容易决定粒子的位置。

最后将WindVector向量初始化，此向量用来表示风的方向。WindVector是一个全局变量，其声明如下：

```
Vector          WindVector;
float           WindForceFactor = WINDFACTOR;
```

对于其中的变量WindForceFactor，当确定作用在旗子上的风力时，要将WindVector乘以这个值。

模型中除了风力之外，还有其他的作用力作用在粒子上。已知弹簧会对粒子施力维持旗子的构造。此外还有重力及黏滞阻力。这些力都由CalcForces函数负责计算：

```
// 本范例使用Direct3D坐标系 —— z轴指向屏幕内，x轴指向右方，y轴指向上方

void   CalcForces(Particle particles[NUMROWS+1][NUMCOLUMNS+1])
{
     int          r, c, i, r1, c1, r2, c2;
     Vector       dragVector;
     Vector       f1, f2, d, v;
     float        L;

     // 将所有力置零
     for(r=0; r<=NUMROWS; r++)
     {
          for(c=0; c<=NUMCOLUMNS; c++)
          {
               particles[r][c].vForces.x = 0;
               particles[r][c].vForces.y = 0;
               particles[r][c].vForces.z = 0;
          }
     }

     // 处理重力和阻力
     for(r=0; r<=NUMROWS; r++)
     {
          for(c=0; c<=NUMCOLUMNS; c++)
          {
               if(particles[r][c].bLocked == FALSE)
               {
                    // 重力
                    particles[r][c].vForces.y += (float) (GRAVITY *
                                                 particles[r][c].fMass);

                    // 黏滞组力
                    dragVector = -particles[r][c].vVelocity;
                    dragVector.Normalize();
                    particles[r][c].vForces += dragVector *
                                   (particles[r][c].vVelocity.Magnitude() *
                                    particles[r][c].vVelocity.Magnitude())
                                    *DRAGCOEFFICIENT;
                    // 风力
                    SetWindVector(tb_Rnd(0, 10), 0, tb_Rnd(0, 1));
                    WindVector.Normalize();
                    particles[r][c].vForces += WindVector *
                                   tb_Rnd(0, WindForceFactor);
               }
          }
     }
```

```
// 计算弹力
for(i = 0; i<NUMSTRUCTURALSPRINGS; i++)
{
     r1 = StructuralSprings[i].p1.r;
     c1 = StructuralSprings[i].p1.c;
     r2 = StructuralSprings[i].p2.r;
     c2 = StructuralSprings[i].p2.c;

     d = particles[r1][c1].vPosition - particles[r2][c2].vPosition;
     v = particles[r1][c1].vVelocity - particles[r2][c2].vVelocity;
     L = StructuralSprings[i].L;

     f1 = -(StructuralSprings[i].k * (d.Magnitude() - L) +
          StructuralSprings[i].d * ( (v * d) / d.Magnitude() )) *
          ( d / d.Magnitude() );
          f2 = -f1;

     if(particles[r1][c1].bLocked == FALSE)
          particles[r1][c1].vForces += f1;

     if(particles[r2][c2].bLocked == FALSE)
          particles[r2][c2].vForces += f2;
   }
}
```

此函数先将每一个粒子上的所有作用力置零。接着计算作用在每个粒子上的重力、黏滞阻力和风力。这些计算和前一个范例所用的很类似。然而，在风力的计算上加入了一点随机性。这么做是为了要让旗子的垂直面更凌乱一点，使摆动能更为真实。

最后的循环是处理所有粒子上的弹力。因为所有弹簧的资料在初始化时都已设定了，所以取出每个弹簧的资料，并将弹簧阻尼力（利用第三章中的弹簧阻尼力公式）套用到连接的粒子上就变得很简单了。

请注意，在这些计算中都会检查每个粒子是否被锁住。如果是，作用力就不能套用到该粒子，表示粒子维持静止。

## 积分函数

本范例仍因简单的理由再选择欧拉法。如同前一个范例一样，StepSimulation函数用来处理运动方程的积分。此函数在本范例中是非常简单的：

```
void     StepSimulation(float dt)
{
     Vector     Ae;
     int        r, c;
     int        check = 0;
```

```
// 计算所有的力
CalcForces(Particles);

// 积分
for(r=0; r<=NUMROWS; r++)
{
     for(c=0; c<=NUMCOLUMNS; c++)
     {
          Ae = Particles[r][c].vForces * Particles[r][c].fInvMass;
          Particles[r][c].vAcceleration = Ae;
          Particles[r][c].vVelocity += Ae * dt;
          Particles[r][c].vPosition += Particles[r][c].vVelocity *
                                       dt;
     }
}
// 检查碰撞
check = CheckForCollisions(Particles);
if(check == COLLISION)
     ResolveCollisions(Particles);

// 更新D3D 布料对象的几何形状
UpdateClothGeometry();
}
```

因为处理的是粒子而不是刚体，因此运动方程只有线性运动。在调用CalcForces之后，函数进入循环对每一个粒子更新它们的位置、速度和加速度。

循环结束之后，调用CheckForCollisions检查粒子是否与旗杆或地面碰撞。如果是，就调用ResolveCollisions函数处理粒子的碰撞反应。

最后调用UpdateClothGeometry，根据更新后的粒子，来更新Direct3D中旗子对象新的顶点与面的数据。

## 碰撞反应

本节只需要检查粒子与旗杆和粒子与地面之间的碰撞。如果将风力设为0，旗子将沿着旗杆往下垂；如果按下R键释放旗子，旗子会往地面掉，并且因风力大小的不同，旗子可能会静止在地面上或被吹走。

因为旗子是由许多的粒子所组成的，所以程序要能计算多重的碰撞。在本范例中，最大的碰撞次数与粒子数目相等（假设粒子不能同时与旗杆和地面碰撞）。另外，并建立Collision结构以及全局的结构阵列来存储这些碰撞数据：

```
typedef struct      _Collision {
     ParticleRef           p1;
     Vector                n;
}      Collision, *pCollision;
```

```
Collision                    Collisions[NUMVERTICES];
```

Collision结构的第一个参数p1是碰撞粒子的参考指针。第二个参数n是用来计算线性冲量的碰撞法向量。

不管是否已经调用CheckForCollisions,Collision阵列里的所有元素都会填入碰撞数据。而未填入数据的阵列元素，其p1值会设为-1，表示没有碰撞数据。

如果确定在阵列中有任何碰撞数据，就调用ResolveCollisions。该函数如下：

```
void     ResolveCollisions(Particle p[NUMROWS+1][NUMCOLUMNS+1])
{
     int     i;
     int     r, c;
     Vector  Vn, Vt;

     for(i=0; i<NUMVERTICES; i++)
     {
          if(Collisions[i].p1.r != -1)
          {
               r = Collisions[i].p1.r;
               c = Collisions[i].p1.c;
               Vn = (Collisions[i].n * p[r][c].vVelocity) *
                    Collisions[i].n;
               Vt = p[r][c].vVelocity - Vn;
               p[r][c].vVelocity = (-(KRESTITUTION+1) * Vn) +
                                   (FRICTIONFACTOR*Vt);
          }
     }
}
```

正如你看到的，此函数十分简短，因为它只需要处理粒子与不会移动的物体间的碰撞，即假设旗杆和地面的质量相对于粒子的质量是无限大的。本范例惟一要做的是计算碰撞粒子速度的法线分量，求它的反矩阵，再乘以恢复系数，最后求得粒子碰撞后的法线速度。接着可以得到速度的切线分量，再乘以摩擦系数求得滑动摩擦力。最后将新的法线速度和切线速度加起来，可以得到粒子在碰撞之后的新速度。

# 参数调整

和前一章一样，将一些重要的控制参数设定成全局的定义常数，以方便调整模拟程序。以下是这些定义：

```
#define          GRAVITY                    -32.174
#define          SPRINGTENSIONCONSTANT      500.0f
#define          SPRINGSHEARCONSTANT        300.0f
#define          SPRINGDAMPINGCONSTANT      2.0f
#define          YOFFSET                    120.0f
```

```
#define          DRAGCOEFFICIENT          0.01f
#define          WINDFACTOR               100
#define          FLAGPOLEHEIGHT           200
#define          FLAGPOLERADIUS           10
#define          COLLISIONTOLERANCE       0.05f
#define          KRESTITUTION             0.25f
#define          FRICTIONFACTOR           0.5f
```

不管是用了欧拉法还是改良型欧拉法，我发现只要将弹簧调整得当，此模拟程序便能相当真实。在范例中，时间间隔大小固定为 16 ms，并将物理更新与画面更新的速度比例设为 10:1。

你可以发现，我设定了两种不同的弹力常数来表示主要的构造弹簧（垂直或水平连接相邻粒子的弹簧）和剪力弹簧（网格中连接对角粒子的弹簧）。这样可以单独调整剪力弹簧来测试它们对整面旗子的影响，而不会动到张力弹簧（构造弹簧）。如果对这些数字做试验，你会发现当剪力弹簧常数很小时，旗子看起来会很有弹性。相反，如果这个常数很大，会发现旗子非常没弹性，好像不能被风吹动一样。

但是在增加这些常数值的时候要非常小心。如果将弹簧常数设定得很高来减小旗子的弹性时，结果会不够自然，并导致数值不稳定。阻尼可以在这里帮上忙。事实上，你要加入一点阻尼，不管是黏滞阻尼还是弹簧阻尼，都可以减少数值的不稳定。

如果要观察阻尼的影响，可以试着把弹簧阻尼常数的值改小一点。你会发现组成布料的粒子会来回地跳动，模拟程序看起来会很假。尤其当释放旗子往地面掉落，让粒子碰撞到地面时，这种情形会很明显。

谈到碰撞时，你也许会想要实现第十三章所谈到的穿透检查。这里不谈穿透的原因是因为粒子只会和旗杆或地面碰撞。然而，如果想要在碰撞程序中加入其他会和粒子碰撞的物体，就要处理可能会发生穿透的情况了。

# 附录一

# 向量的运算

附录一 将介绍 Vector 类的实现方式，此类封装了编写 2D 或 3D 刚体模拟程序时所有需要用到的向量运算。虽然 Vector 类使用了 3D 的向量，但是你也可以把它当做 2D 向量来使用，只要在实现时忽略所有的 $z$ 值或将 $z$ 值都设为 0 即可。

## Vector 类

Vector 类定义三个成员变量 $x$、$y$、$z$，以及几个实现基本向量运算的成员函数和向量操作符。此类有两个构造函数，其中一个将所有的分量变量初始化为 0，另外一个则将分量变量设为传入的参数。

```
//-----------------------------------------------------------------------
// Vector 类和其中封装的函数
//-----------------------------------------------------------------------
class Vector {
public:
     float x;
     float y;
     float z;

     Vector(void);
     Vector(float xi, float yi, float zi);

     float Magnitude(void);
     void Normalize(void);
     void Reverse(void);

     Vector& operator+=(Vector u);
     Vector& operator-=(Vector u);
```

```
        Vector& operator*=(float s);
        Vector& operator/=(float s);

        Vector operator-(void);

};

// 构造函数
inline Vector::Vector(void)
{
        x = 0;
        y = 0;
        z = 0;
{

// 构造函数
inline Vector::Vector(float xi, float yi, float zi)
}
        x = xi;
        y = yi;
        z = zi;
}
```

## 量值

Magnitude成员函数只根据以下公式计算向量的大小：

$$|v| = \sqrt{x^2 + y^2 + z^2}$$

这是一个以零为基准的向量，也就是说其分量都是相对于原点的。向量的大小也就是向量的长度，如图 A-1 所示。

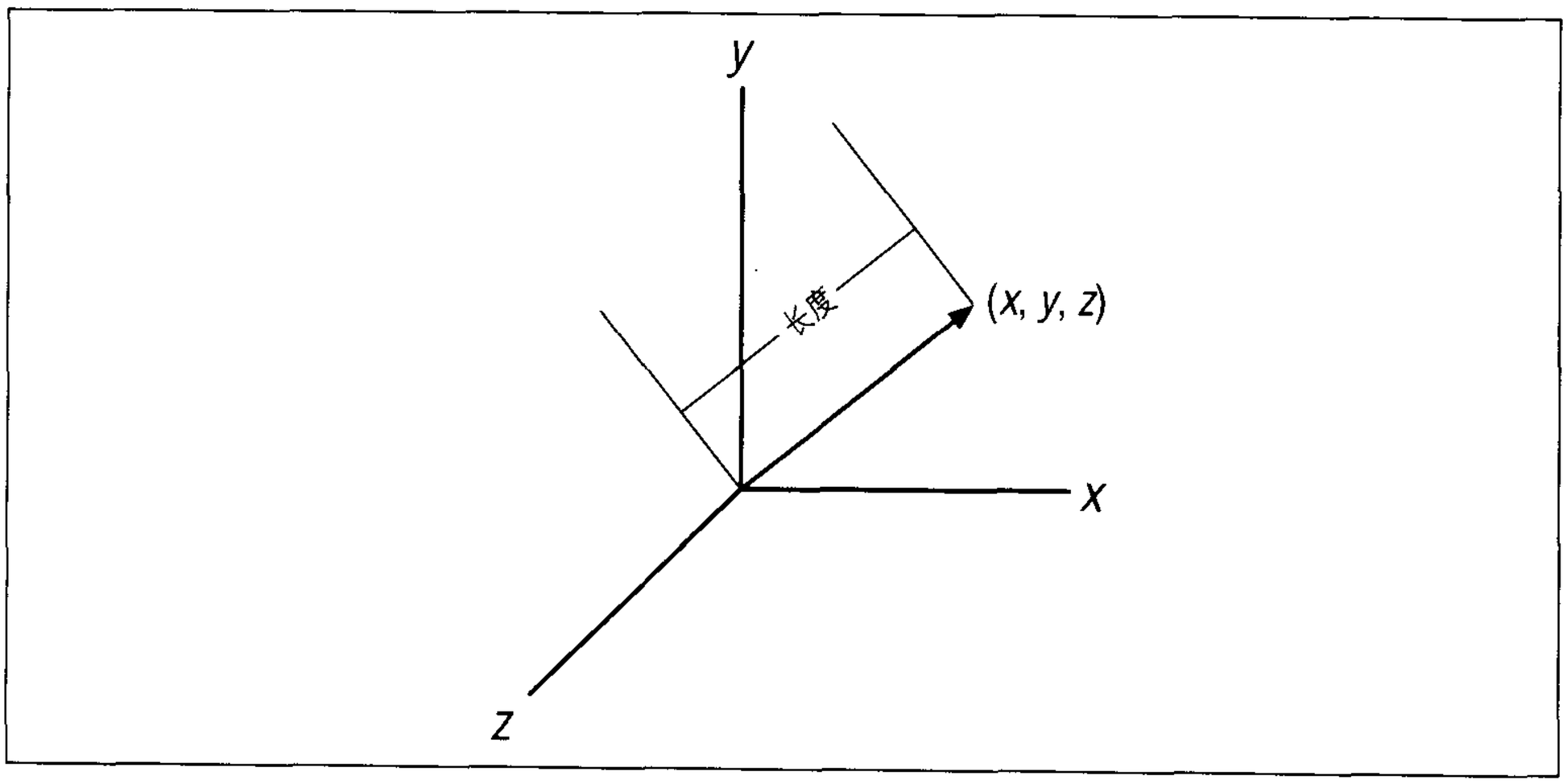



图 A-1：向量的长度（大小）

以下是Vector类中计算向量量值的程序代码：

```
inline        float Vector::Magnitude(void)
{
     return (float) sqrt(x*x + y*y + z*z);
}
```

如果知道一个向量的长度和所有的方向角（direction angle）就可以求所有的分量。方向角的定义是向量与坐标轴的夹角，如图 A-2 所示。

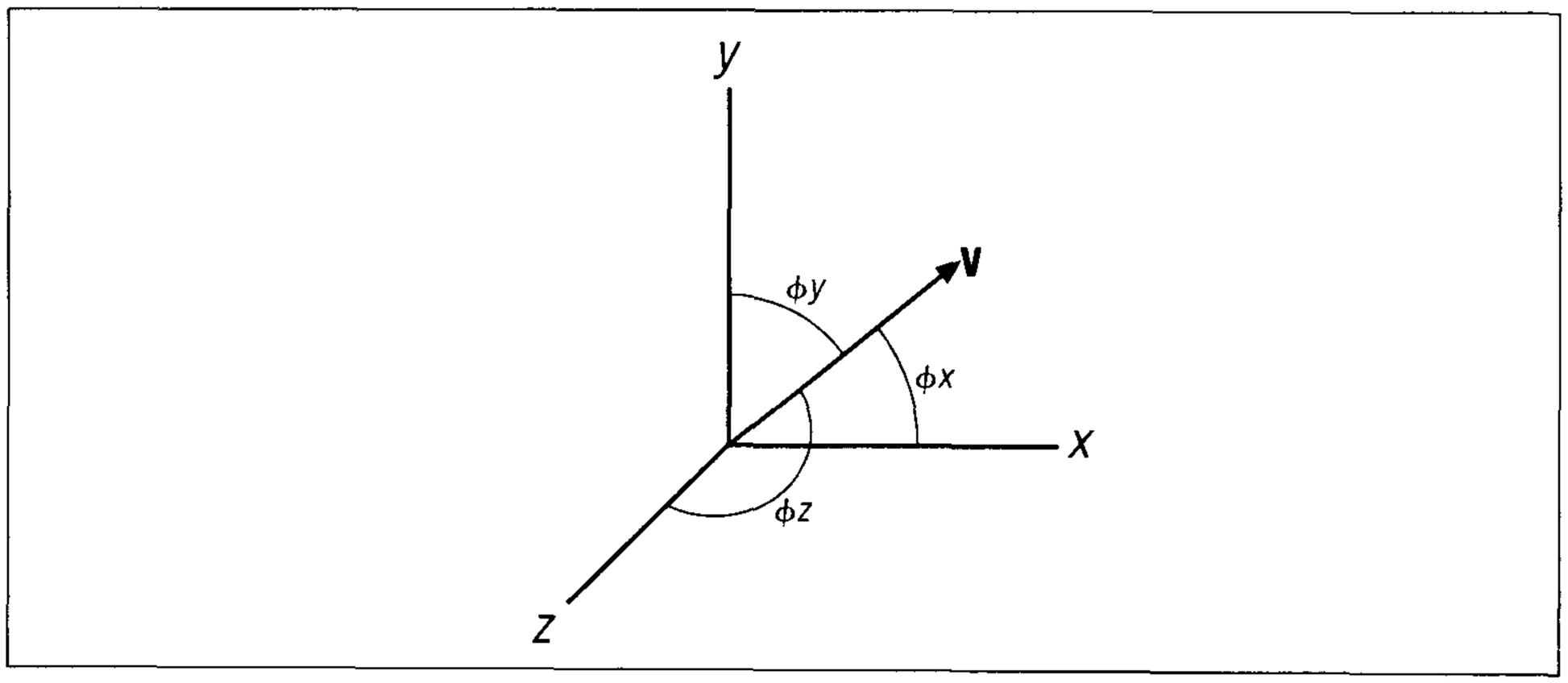



图 A-2：方向角

图中向量的分量可以这样表示，其中 $i$，$j$，$k$ 分别为 $x$，$y$，$z$ 轴上的单位向量：

$$v_x = |v|\cos\varphi_x i$$
$$v_y = |v|\cos\varphi_y j$$
$$v_z = |v|\cos\varphi_z k$$

公式中方向角的余弦称为方向余弦（direction cosine）。这些方向余弦的平方和一定等于 1：

$$\cos^2\varphi_x + \cos^2\varphi_y + \cos^2\varphi_z = 1$$

## 正规化

Normalize成员函数会将向量正规化，或者是说将向量转换成单位向量，以满足下列公式：

$$|v| = \sqrt{x^2 + y^2 + z^2} = 1$$

换句话说，被正规化后向量的长度为1单位长。假设$v$是一个非单位向量的向量，并有三个分量$x$，$y$，$z$，其单位向量$u$可用以下公式从$v$中求得：

$$u = v/|v| = (x/|v|)i + (y/|v|)j + (z/|v|)k$$

这里的$|v|$是前面所述向量$v$的大小，或称为长度。

以下程序代码可将Vector类的向量转换成单位向量：

```
inline        void Vector::Normalize(void)
{
     float m = (float) sqrt(x*x + y*y + z*z);
     if(m <= tol) m = 1;
     x /= m;
     y /= m;
     z /= m;
     if (fabs(x) < tol) x = 0.0f;
     if (fabs(y) < tol) y = 0.0f;
     if (fabs(z) < tol) z = 0.0f;
}
```

此函数中的tol是浮点数类型的允许值，例如：

```
float  const tol = 0.0001f;
```

## 反转向量

Reverse成员函数反转向量的方向，做法是取所有分量的负数。调用Reverse后，向量的方向会指向在调用Reverse之前的反方向。

```
inline     void Vector::Reverse(void)
{
     x = -x;
     y = -y;
     z = -z;
}
```

图A-3说明了此函数的运算结果。

## 向量的加法：+=操作符

这个加法操作符专用于向量的加法，将传入向量的每个分量分别加到目前向量的每个分量中。在图A-4中可以看到，向量是以“头连尾”的方式相加的。

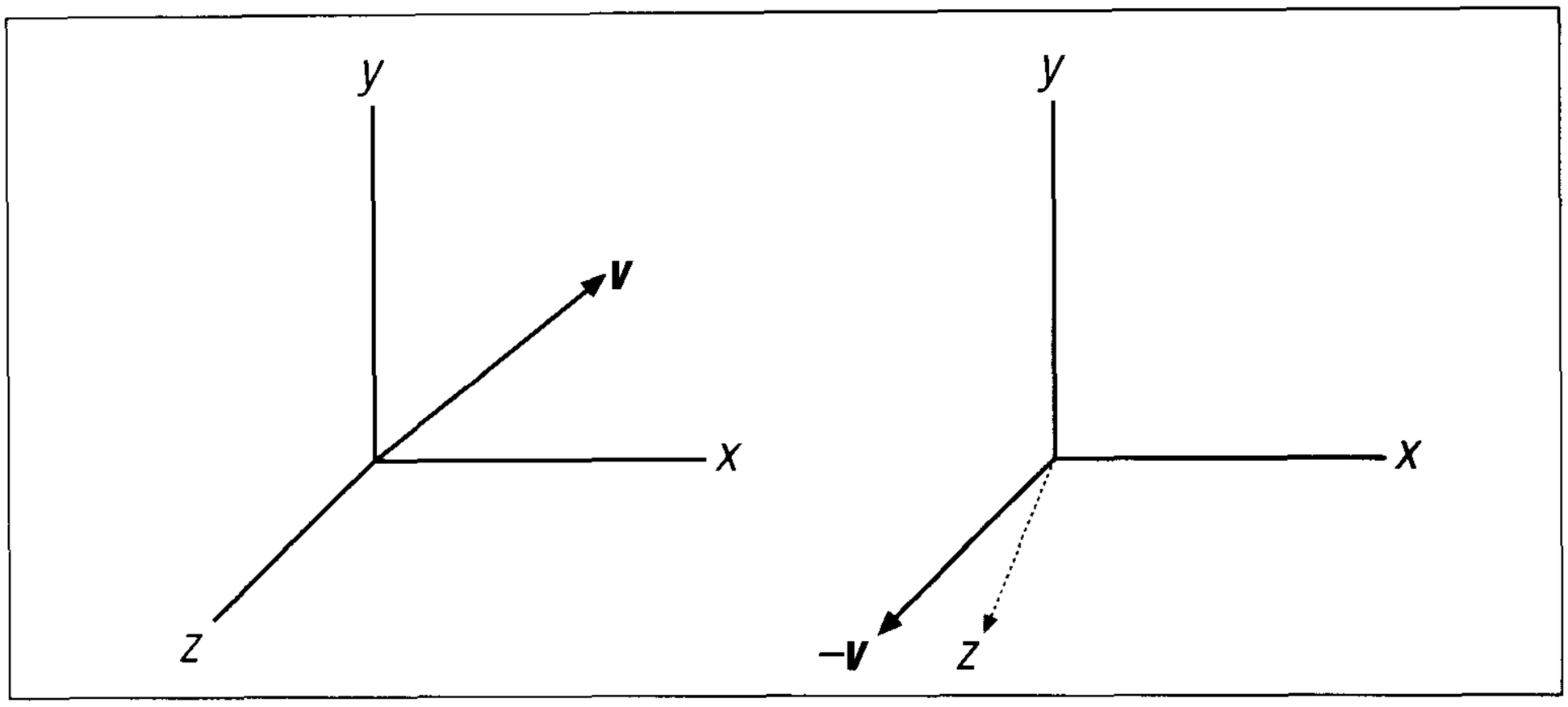



图 A-3：向量的反转

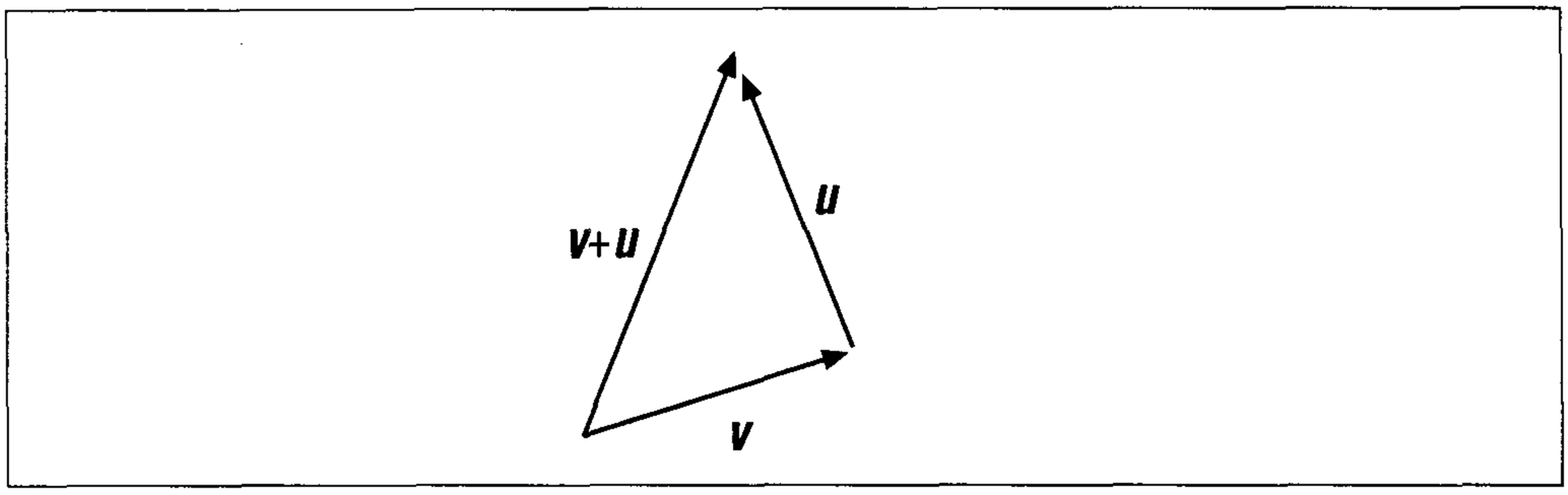



图 A-4：向量的加法

以下是将向量 *u* 加到 Vector 向量的程序代码：

```
inline Vector& Vector::operator+=(Vector u)
{
      x += u.x;
      y += u.y;
      z += u.z;
      return *this;
}
```

## 向量的减法：-= 操作符

这个减法操作符用来将目前的向量减去传入的向量，也是使用分量对分量的方式来计算的。向量的减法与向量的加法很类似，但向量减法会先取得第二个向量的反转向量再加到第一个向量中。向量的减法如图 A-5 所示。

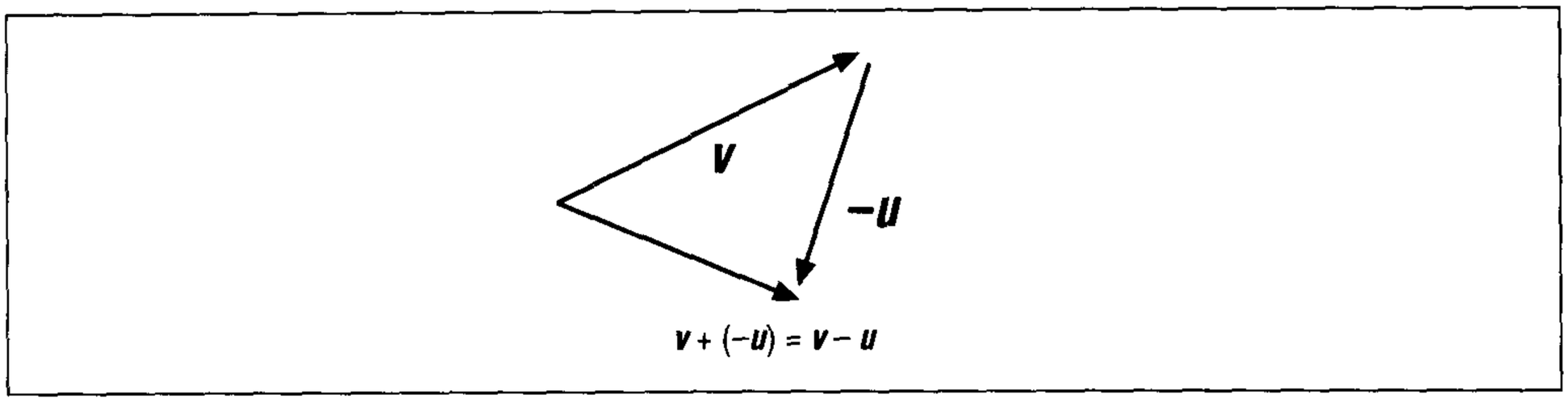



图 A-5：向量的减法

以下是从 Vector 类的向量减去向量 **u** 的程序代码：

```
inline     Vector& Vector::operator-=(Vector u)
{
     x -= u.x;
     y -= u.y;
     z -= u.z;
     return *this;
}
```

## 标量的乘法：*= 操作符

标量乘法的操作符将向量乘以传入的标量，会使向量的长度缩放。将向量乘以标量时，事实上是将每个分量分别乘以这个标量值。所得到的新向量的方向与旧向量的方向相同，但是长度就不同了（除非相乘的标量是 1）。图 A-6 说明了向量的变化。

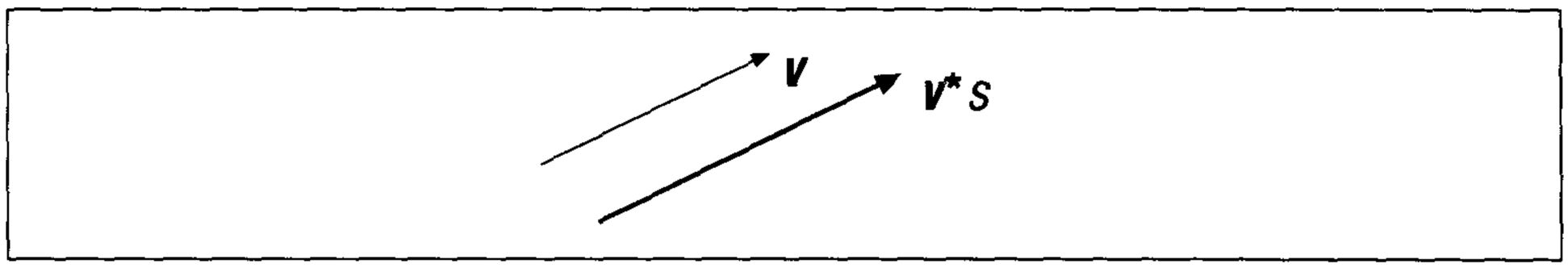



图 A-6：标量的乘法

以下是 Vector 类的向量乘以标量的程序代码：

```
inline     Vector& Vector::operator*=(float s)
{
     x *=s;
     y *=s;
     z *=s;
     return *this;
}
```

## 标量的除法：/= 操作符

标量的除法与标量的乘法很类似，只是标量的除法是将每个分量除以传入的标量值。

```
inline      Vector& Vector::operator/=(float s)
{
      x /=s;
      y /=s;
      z /=s;
      return *this;
}
```

## 向量的共轭：－操作符

共轭操作符只是取每个分量的负数，可用在向量的相减上或者是求向量的反转向量时。共轭操作符和之前所说的反向量操作符的作用是相同的。

```
inline      Vector Vector::operator-(void)
{
      return Vector(-x, -y, -z);
}
```

# 向量函数与操作符

以下的函数和多载操作符可以用在两个向量间，或者向量与标量间的运算，而其中的向量是 Vector 类实体。

## 向量的加法：+ 操作符

此加法操作符用以下的公式将向量 $v$ 加到向量 $u$ 中：

$$u + v = (u_x + v_x)i + (u_y + v_y)j + (u_z + v_z)k$$

以下是程序代码：

```
inline      Vector operator+(Vector u, Vector v)
{
      return Vector(u.x + v.x, u.y + v.y, u.z + v.z);
}
```

## 向量的减法：－操作符

这个减法操作符用以下的公式将向量 $v$ 由向量 $u$ 中减去：

$$u - v = (u_x - v_x)i + (u_y - v_y)j + (u_z - v_z)k$$

以下是程序码：

```
inline     Vector operator-(Vector u, Vector v)
{
     return Vector(u.x - v.x, u.y - v.y, u.z - v.z);
}
```

## 向量的外积：×操作符

此外积操作符计算向量 $u$ 和向量 $v$ 的外积 —— $u \times v$，并返回一个与两向量 $u$ 和 $v$ 垂直的向量。公式如下：

$$u \times v = (u_y * v_z - u_z * v_y)i + (-u_x * v_z + u_z * v_x)j + (u_x * v_y - u_y * v_x)k$$

所得到的向量会垂直于向量 $u$ 和向量 $v$ 所构成的平面。这个向量的方向可以由右手定则来决定。也就是你先将向量 $u$ 和向量 $v$ 如同图 A-7 一样尾端对尾端放好，再将右手的手指（除大拇指以外）从向量 $u$ 的方向弯曲到向量 $v$ 的方向，此时大拇指的方向就是所求外积向量的方向。

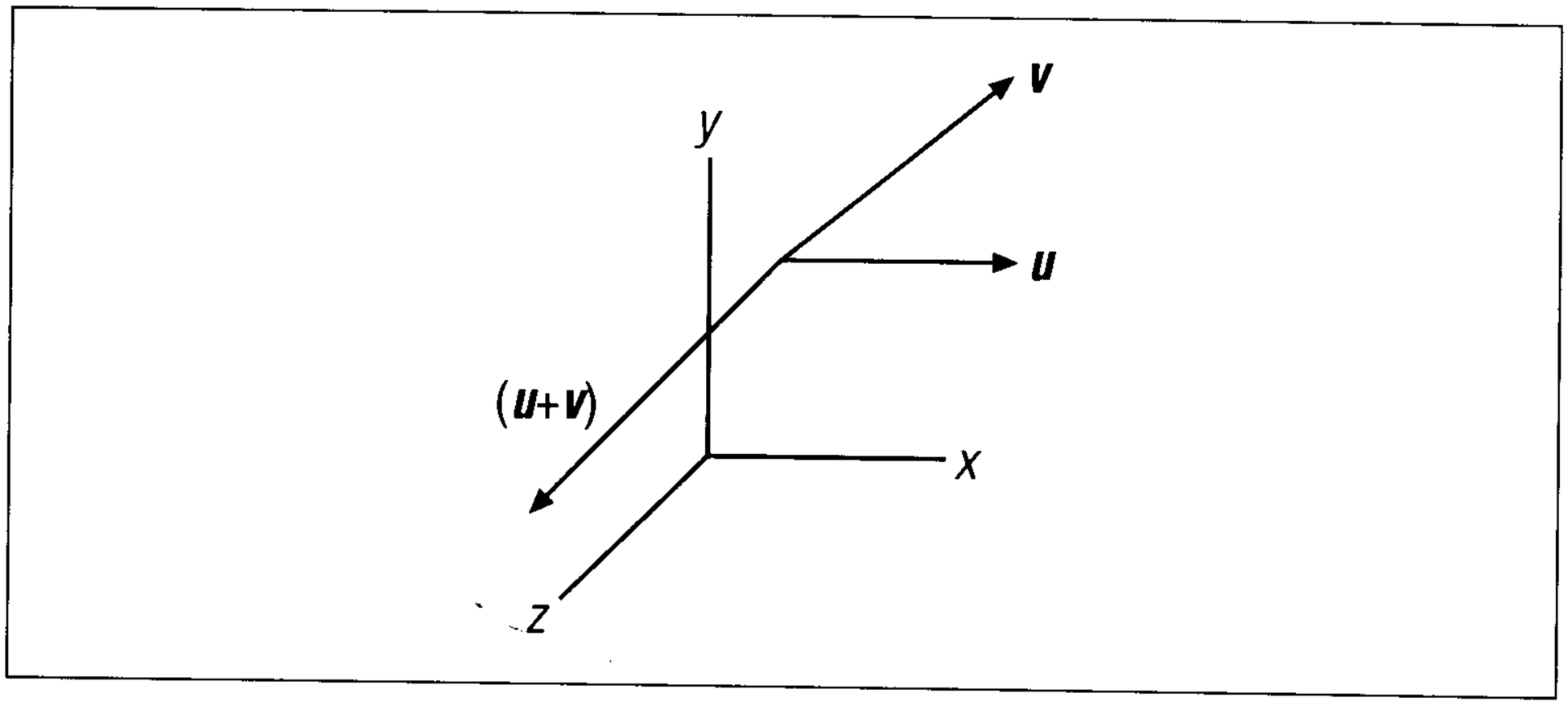



图 A-7：向量的外积

图 A-7 中所得到的向量指向 $z$ 轴的方向，因为向量 $u$ 和向量 $v$ 位于 $x$ 轴与 $y$ 轴所构成的平面上。

如果两向量平行，则其外积向量是 0。这在判断两个向量是否平行时是很有用的。

外积的运算符合分配律，然而却不符合交换律：

$$u \times v \neq v \times u$$
$$u \times v = -(v \times u)$$

$$s(\boldsymbol{u} \times \boldsymbol{v}) = (s)(\boldsymbol{u}) \times \boldsymbol{v} = \boldsymbol{u} \times (s)(\boldsymbol{v})$$
$$\boldsymbol{u} \times (\boldsymbol{v} + \boldsymbol{p}) = (\boldsymbol{u} \times \boldsymbol{v}) + (\boldsymbol{u} \times \boldsymbol{p})$$

以下是求向量 $\boldsymbol{u}$ 和向量 $\boldsymbol{v}$ 的外积的程序代码：

```
inline     Vector operator^(Vector u, Vector v)
{
     return Vector(       u.y*v.z - u.z*v.y,
                          -u.x*v.z + u.z*v.x,
                          u.x*v.y - u.y*v.x );
}
```

使用向量的外积来求法向量（垂直向量）是很方便的。例如进行碰撞侦测时，需求出多面体平面上的法向量。可利用多面体的顶点来得到构成平面的向量，再求这些向量的外积便得到平面上的法向量。

## 向量的内积：* 操作符

此操作符根据以下的公式求得向量 $\boldsymbol{u}$ 和向量 $\boldsymbol{v}$ 之间的内积：

$$\boldsymbol{u} \cdot \boldsymbol{v} = (u_x * v_x) + (u_y * v_y) + (u_z * v_z)$$

向量 $\boldsymbol{u}$，$\boldsymbol{v}$ 的内积表示向量 $\boldsymbol{u}$ 在向量 $\boldsymbol{v}$ 上的投影量乘上 $\boldsymbol{v}$ 的长度，如图 A-8 所示。

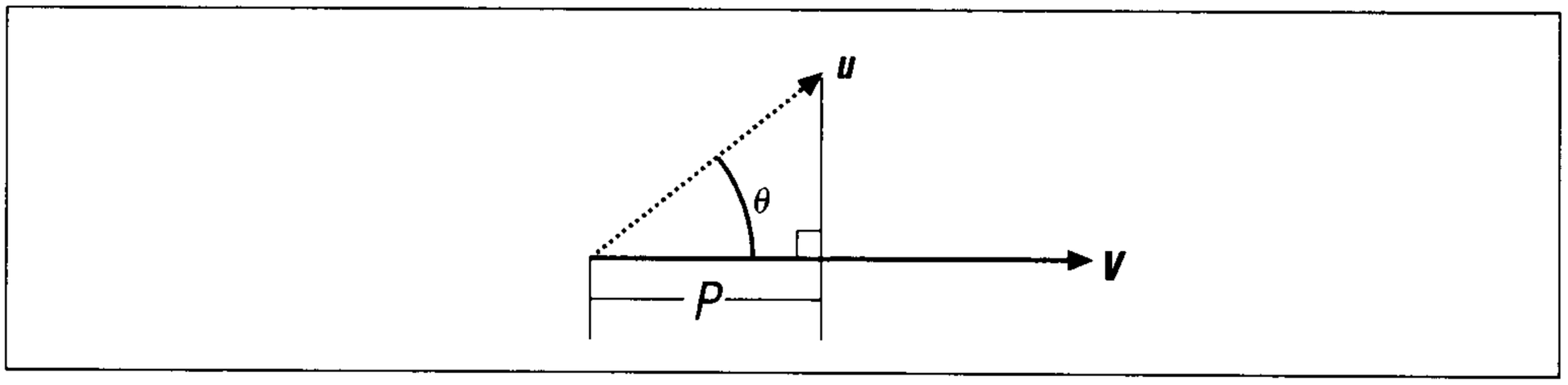



图 A-8：向量的内积

在图中，$P$ 是所求的内积（若 $\boldsymbol{v}$ 为单位向量，则 $P$ 即为 $\boldsymbol{u}$ 在 $\boldsymbol{v}$ 方向上的投影量），且是一个标量。你也可以由夹角求得两向量的内积：

$$P = \boldsymbol{u} \cdot \boldsymbol{v} = |\boldsymbol{u}||\boldsymbol{v}|\cos\theta$$

以下是求向量 $\boldsymbol{u}$ 和向量 $\boldsymbol{v}$ 之间内积的程序代码：

```
// 向量的内积
inline        float operator*(Vector u, Vector v)
{
     return (u.x*v.x + u.y*v.y + u.z*v.z);
}
```

利用向量的内积可求得一向量在另一个向量上的投影向量。回到碰撞侦测的范例，常常需要求多面体（物体1）上的顶点A到另一个多面体（物体2）上某一个面的最小距离。你可以这样做：先找出一个向量（从物体2某面的任一顶点指到A点），此向量与物体2平面法向量的内积就是A点到物体2上某个平面的距离（如果平面法向量不是单位向量，你必须要将结果除以这个法向量的量值）。

## 标量的乘法：* 操作符

这个操作符计算向量 $u$ 乘以标量 $s$ 的结果 —— 每个分量分别乘以标量 $s$。以下是依据标量与向量的顺序而衍生的两个不同版本，当然结果是相同的：

```
inline     Vector operator*(float s, Vector u)
{
     return Vector(u.x*s, u.y*s, u.z*s);
}

inline     Vector operator*(Vector u, float s)
{
     return Vector(u.x*s, u.y*s, u.z*s);
}
```

## 标量的除法：/ 操作符

这个操作符会求出向量 $u$ 除以标量 $s$ 的结果 —— 将每个分量分别除以标量 $s$：

```
inline     Vector operator/(Vector u, float s)
{
     return Vector(u.x/s, u.y/s, u.z/s);
}
```

## 标量三重积

此函数使用以下的公式来求向量 $u$，$v$，$w$ 之间的标量三重积：

$$s = u \cdot (v \times w)$$

这里求出来的值 $s$ 是标量。程序代码如下：

```
inline     float TripleScalarProduct(Vector u, Vector v, Vector w)
{
     return float(         (u.x * (v.y*w.z - v.z*w.y)) +
                           (u.y * (-v.x*w.z + v.z*w.x)) +
                           (u.z * (v.x*w.y - v.y*w.x)) );
}
```

# 附录二

# 矩阵的运算

附录二将介绍Matrix3x3类的实现方式，这个类封装了在编写3D刚体模拟程序时，处理3 × 3矩阵（9个元素）所需的运算。

## Matrix3x3类

Matrix3x3类中有9个元素，每个元素可以用$e_{ij}$来代表，表示这个元素的位置在第$i$列第$j$行。举例来说，$e_{21}$表示这个元素在第2列第1行。以下是所有元素的排列方式：

$$\boldsymbol{M} = \begin{vmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{vmatrix}$$

这个类有两个构造函数，其中一个在初始化时将所有元素设为0，另外一个则将所有元素设定为所传入的值：

```
class Matrix3x3 {
public:
    // 元素eij：第i列、第j行
    float     e11, e12, e13, e21, e22, e23, e31, e32, e33;
    Matrix3x3(void);
    Matrix3x3(float r1c1, float r1c2, float r1c3,
              float r2c1, float r2c2, float r2c3,
              float r3c1, float r3c2, float r3c3 );

    float          det(void);
    Matrix3x3      Transpose(void);
    Matrix3x3      Inverse(void);
```

```
        Matrix3x3& operator+=(Matrix3x3 m);
        Matrix3x3& operator-=(Matrix3x3 m);
        Matrix3x3& operator*=(float s);
        Matrix3x3& operator/=(float s);
};

// 构造函数
inline        Matrix3x3::Matrix3x3(void)
{
        e11 = 0;
        e12 = 0;
        e13 = 0;
        e21 = 0;
        e22 = 0;
        e23 = 0;
        e31 = 0;
        e32 = 0;
        e33 = 0;
}

//构造函数
inline     Matrix3x3::Matrix3x3(float r1c1, float r1c2, float r1c3,
                                float r2c1, float r2c2, float r2c3,
                                float r3c1, float r3c2, float r3c3 )
{
        e11 = r1c1;
        e12 = r1c2;
        e13 = r1c3;
        e21 = r2c1;
        e22 = r2c2;
        e23 = r2c3;
        e31 = r3c1;
        e32 = r3c2;
        e33 = r3c3;
}
```

# 行列式

det 成员函数返回矩阵的行列式值，一个 2 × 2 的矩阵：

$$m = \begin{vmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{vmatrix}$$

其行列式值如下：

$$det[m] = e_{11}e_{22} - e_{21}e_{12}$$

要求3×3矩阵的行列式值，可以先展开成2×2的子行列式（minor）（译注1），再解所有的子行列式值。以下是展开成子行列式的方法：

$$\boldsymbol{M} = e_{11}\begin{vmatrix} e_{22} & e_{23} \\ e_{32} & e_{33} \end{vmatrix} - e_{12}\begin{vmatrix} e_{21} & e_{23} \\ e_{31} & e_{33} \end{vmatrix} + e_{13}\begin{vmatrix} e_{21} & e_{22} \\ e_{31} & e_{32} \end{vmatrix}$$

以下是这部分的程序代码：

```
inline     float    Matrix3x3::det(void)
{
     return   e11*e22*e33 -
              e11*e32*e23 +
              e21*e32*e13 -
              e21*e12*e33 +
              e31*e12*e23 -
              e31*e22*e13;
}
```

## 转置矩阵

Transpose成员函数用来将矩阵的行与列对调，形成转置矩阵——第一列的所有元素会变成第一行，而第二列和第三列依此类推。以下的关系式是转置运算的特性：

$$(\boldsymbol{M}^t)^t = \boldsymbol{M}$$
$$(s\boldsymbol{M})^t = s(\boldsymbol{M}^t)$$
$$(\boldsymbol{MN})^t = \boldsymbol{N}^t\boldsymbol{M}^t$$
$$(\boldsymbol{M} + \boldsymbol{N})^t = \boldsymbol{M}^t + \boldsymbol{N}^t$$
$$\det[\boldsymbol{M}^t] = \det[\boldsymbol{M}]$$

这里的$\boldsymbol{M}$和$\boldsymbol{N}$是矩阵，$t$是转置操作符，$s$是一个标量。

以下是Matrix3x3类中的Transpose成员函数：

```
inline       Matrix3x3       Matrix3x3::Transpose(void)
{
     return  Matrix3x3(e11,e21,e31,e12,e22,e32,e13,e23,e33);
}
```

## 反矩阵

Inverse成员函数计算出矩阵的反矩阵。矩阵与反矩阵之间有以下关系：

---

译注1：子行列式：把$n \times n$矩阵的第$i$列与第$j$行去掉，剩下来的$n$-1 × $n$-1矩阵的行列式值就是$e_{ij}$的子行列式。

$$MM^{-1} = M^{-1}M = \mathbf{I}$$

这里的$M^{-1}$是矩阵$M$的反矩阵，而$\mathbf{I}$是单位矩阵。对于$3 \times 3$矩阵来说，可以用以下方法求出反矩阵：

$$M^{-1} = 1/\det[M] \begin{vmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{vmatrix}$$

这里的$E_{ij}$是$e_{ij}$的辅因子（cofactor），就是每个元素的子行列式值。只有方阵（也就是行数与列数相等的矩阵）才有反矩阵。但是并不是所有的方阵都有反矩阵。当矩阵的行列式值不是0时才有反矩阵。

反矩阵有以下的关系式：

$$(MN)^{-1} = N^{-1}M^{-1}$$

以下是在Matrix3x3类中计算反矩阵的程序代码：

```
inline      Matrix3x3       Matrix3x3::Inverse(void)
{
float       d = e11*e22*e33 -
                e11*e32*e23 +
                e21*e32*e13 -
                e21*e12*e33 +
                e31*e12*e23 -
                e31*e22*e13;

if (d == 0) d = 1;

return      Matrix3x3(          (e22*e33-e23*e32)/d,
                                -(e12*e33-e13*e32)/d,
                                (e12*e23-e13*e22)/d,
                                -(e21*e33-e23*e31)/d,
                                (e11*e33-e13*e31)/d,
                                -(e11*e23-e13*e21)/d,
                                (e21*e32-e22*e31)/d,
                                -(e11*e32-e12*e31)/d,
                                (e11*e22-e12*e21)/d );
}
```

## 矩阵的加法：+= 操作符

这个操作符将传入的矩阵中的元素逐一地与目前矩阵的元素相加。两个能相加的矩阵一定要有相同的阶数——有相同数目的行及列。

```
inline      Matrix3x3& Matrix3x3::operator+=(Matrix3x3 m)
{
```

```
        e11 += m.e11;
        e12 += m.e12;
        e13 += m.e13;
        e21 += m.e21;
        e22 += m.e22;
        e23 += m.e23;
        e31 += m.e31;
        e32 += m.e32;
        e33 += m.e33;
        return *this;
    }
```

矩阵的加法（减法亦同）符合交换律、结合律和分配律：

$$M + N = N + M$$
$$M + (N + P) = (M + N) + P$$
$$M(N + P) = MN + MP$$
$$(N + P)M = NM + PM$$

## 矩阵的减法：-= 操作符

这个操作符将传入的矩阵中的元素逐一地与目前矩阵的元素相减。两个能相减的矩阵一定要有相同的阶数 —— 有相同数目的行及列。

```
inline     Matrix3x3& Matrix3x3::operator-=(Matrix3x3 m)
{
     e11 -= m.e11;
     e12 -= m.e12;
     e13 -= m.e13;
     e21 -= m.e21;
     e22 -= m.e22;
     e23 -= m.e23;
     e31 -= m.e31;
     e32 -= m.e32;
     e33 -= m.e33;
     return *this;
}
```

## 标量的乘法：*= 操作符

此操作符将矩阵中的所有元素乘以标量 $s$：

```
inline     Matrix3x3& Matrix3x3::operator*=(float s)
{
     e11 *= s;
     e12 *= s;
     e13 *= s;
     e21 *= s;
     e22 *= s;
```

```
        e23 *= s;
        e31 *= s;
        e32 *= s;
        e33 *= s;
        return *this;
    }
```

标量的乘法（除法亦同）满足以下关系式：

$$s(\boldsymbol{MN}) = (s\boldsymbol{M})\boldsymbol{N} = \boldsymbol{M}(s\boldsymbol{N})$$

## 标量的除法：/= 操作符

此操作符将矩阵中所有元素除以标量 $s$：

```
inline     Matrix3x3& Matrix3x3::operator/=(float s)
{
     e11 /= s;
     e12 /= s;
     e13 /= s;
     e21 /= s;
     e22 /= s;
     e23 /= s;
     e31 /= s;
     e32 /= s;
     e33 /= s;
     return *this;
}
```

# 矩阵的函数和操作符

以下的函数和多载操作符可用来执行两个矩阵、矩阵和标量，或者矩阵与向量之间的运算。这里假设矩阵的类型是 Marix3x3 类，而向量的类型是附录一的 Vector 类。

## 矩阵的加法：+ 操作符

此操作符将两个矩阵中的每个元素逐一地相加：

```
inline     Matrix3x3 operator+(Matrix3x3 m1, Matrix3x3 m2)
{
     return     Matrix3x3(      m1.e11+m2.e11,
                                m1.e12+m2.e12,
                                m1.e13+m2.e13,
                                m1.e21+m2.e21,
                                m1.e22+m2.e22,
                                m1.e23+m2.e23,
                                m1.e31+m2.e31,
                                m1.e32+m2.e32,
```

```
                                    m1.e33+m2.e33);
}
```

## 矩阵的减法：－操作符

此操作符将矩阵 *m*1 中的元素逐一地减去矩阵 *m*2 中相对应的元素：

```
inline     Matrix3x3 operator-(Matrix3x3 m1, Matrix3x3 m2)
{
     return     Matrix3x3(     m1.e11-m2.e11,
                               m1.e12-m2.e12,
                               m1.e13-m2.e13,
                               m1.e21-m2.e21,
                               m1.e22-m2.e22,
                               m1.e23-m2.e23,
                               m1.e31-m2.e31,
                               m1.e32-m2.e32,
                               m1.e33-m2.e33);
}
```

## 标量的除法：/操作符

此操作符将矩阵 *m* 中的每个元素都除以标量 *s*：

```
inline     Matrix3x3 operator/(Matrix3x3 m, float s)
{
     return     Matrix3x3(     m.e11/s,
                               m.e12/s,
                               m.e13/s,
                               m.e21/s,
                               m.e22/s,
                               m.e23/s,
                               m.e31/s,
                               m.e32/s,
                               m.e33/s);
}
```

## 矩阵的乘法：*操作符

此操作符套用在两个矩阵上面时，称为矩阵相乘。在矩阵相乘得到的矩阵中，元素 $e_{ij}$ 是由第一个矩阵的 *i* 列与第二个矩阵的 *j* 列相乘求得的：

```
inline     Matrix3x3 operator*(Matrix3x3 m1, Matrix3x3 m2)
{
     return Matrix3x3(m1.e11*m2.e11 + m1.e12*m2.e21 + m1.e13*m2.e31,
                      m1.e11*m2.e12 + m1.e12*m2.e22 + m1.e13*m2.e32,
                      m1.e11*m2.e13 + m1.e12*m2.e23 + m1.e13*m2.e33,
                      m1.e21*m2.e11 + m1.e22*m2.e21 + m1.e23*m2.e31,
                      m1.e21*m2.e12 + m1.e22*m2.e22 + m1.e23*m2.e32,
```

```
                m1.e21*m2.e13 + m1.e22*m2.e23 + m1.e23*m2.e33,
                m1.e31*m2.e11 + m1.e32*m2.e21 + m1.e33*m2.e31,
                m1.e31*m2.e12 + m1.e32*m2.e22 + m1.e33*m2.e32,
                m1.e31*m2.e13 + m1.e32*m2.e23 + m1.e33*m2.e33 );
}
```

当矩阵的行数等于另一个矩阵的列数时，这两个矩阵才能相乘。矩阵的乘法不符合交换律，但是符合结合律：

$$MN \neq NM$$
$$(MN)P = M(NP)$$

## 标量的乘法：*操作符

此操作符套用在矩阵及标量上时，会将矩阵 $m$ 中每个元素乘以标量 $s$。以下根据传入矩阵与标量的顺序的不同，分别实现两个不同的函数（但结果是一样的）：

```
inline     Matrix3x3 operator*(Matrix3x3 m, float s)
{
     return     Matrix3x3(     m.e11*s,
                               m.e12*s,
                               m.e13*s,
                               m.e21*s,
                               m.e22*s,
                               m.e23*s,
                               m.e31*s,
                               m.e32*s,
                               m.e33*s);
}

inline     Matrix3x3 operator*(float s, Matrix3x3 m)
{
     return     Matrix3x3(     m.e11*s,
                               m.e12*s,
                               m.e13*s,
                               m.e21*s,
                               m.e22*s,
                               m.e23*s,
                               m.e31*s,
                               m.e32*s,
                               m.e33*s);
}
```

## 向量的乘法：*操作符

当矩阵与向量相乘时，此操作符会将矩阵的第 $i$ 行的元素乘以向量中的第 $i$ 个元素。以下根据传入矩阵与向量的顺序的不同，分别实现两个不同的函数：

```
inline     Vector operator*(Matrix3x3 m, Vector u)
{
     return Vector(      m.e11*u.x + m.e12*u.y + m.e13*u.z,
                         m.e21*u.x + m.e22*u.y + m.e23*u.z,
                         m.e31*u.x + m.e32*u.y + m.e33*u.z);
}

inline     Vector operator*(Vector u, Matrix3x3 m)
{
     return Vector(      u.x*m.e11 + u.y*m.e21 + u.z*m.e31,
                         u.x*m.e12 + u.y*m.e22 + u.z*m.e32,
                         u.x*m.e13 + u.y*m.e23 + u.z*m.e33);
}
```

# 附录三

# 四元数的运算

附录三将介绍Quaternion类的实现，此类封装3D刚体模拟程序中，处理四元数所需的运算。

## Quaternion类

Quaternion类由一个标量元素$n$，和一个向量元素$v$（$v=xi+yj+zk$）所组成。此类有两个构造函数，其中一个在初始化时将所有元素都设为0，另外一个根据所传入的值来设定元素的值：

```
class Quaternion    {
public:
     float    n;      // 标量部分
     Vector   v;      // 向量部分：v.x, v.y, v.z

     Quaternion(void);
     Quaternion(float e0, float e1, float e2, float e3);

     float    Magnitude(void);
     Vector   GetVector(void);
     float    GetScalar(void);
     Quaternion operator+=(Quaternion q);
     Quaternion operator-=(Quaternion q);
     Quaternion operator*=(float s);
     Quaternion operator/=(float s);
     Quaternion operator~(void) const { return Quaternion( n,
                                                           -v.x,
                                                           -v.y,
                                                           -v.z);}
```

```
};

// 构造函数
inline      Quaternion::Quaternion(void)
{
     n   = 0;
     v.x = 0;
     v.y = 0;
     v.z = 0;
}

// 构造函数
inline      Quaternion::Quaternion(float e0, float e1, float e2, float e3)
{
     n   = e0;
     v.x = e1;
     v.y = e2;
     v.z = e3;
}
```

## 量值

这个 Magnitude 成员函式用以下的公式计算四元数的量值：

$$|q| = \sqrt{n^2 + x^2 + y^2 + z^2}$$

此公式和计算向量量值的公式很类似，不同的是要将第 4 个元素 —— 标量 $n$ —— 也纳入计算。

以下是在 Quaternion 类中计算量值的程序代码：

```
inline        float       Quaternion::Magnitude(void)
{
     return (float) sqrt(n*n + v.x*v.x + v.y*v.y + v.z*v.z);
}
```

## GetVector

GetVector 成员函式使用在附录一中的 Vector 类，返回四元数中的向量部分：

```
inline        Vector      Quaternion::GetVector(void)
{
     return Vector(v.x, v.y, v.z);
}
```

## GetScalar

GetScalar 成员函式会返回四元数中的标量部分：

```
inline        float        Quaternion::GetScalar(void)
{
     return n;
}
```

## 四元数的加法：+= 操作符

此操作符会将四元数 $q$ 中的元素逐一地与目前四元数的元素相加。

假设 $q$ 和 $p$ 皆为四元数，则：

$$q + p = [n_q + n_p, (x_q + x_p)i + (y_q + y_p)j + (z_q + z_p)k]$$

这里的 $n_q + n_p$ 是四元数和的标量部分，而 $(x_q + x_p)i + (y_q + y_p)j + (z_q + z_p)k$ 是四元数和的向量部分。

四元数的加法符合结合律和交换律，因此，

$$q + (p + h) = (q + p) + h$$
$$q + p = p + q$$

以下是将四元数 $q$ 加到 Quaternion 类中的程序代码：

```
inline     Quaternion     Quaternion::operator+=(Quaternion q)
{
     n += q.n;
     v.x += q.v.x;
     v.y += q.v.y;
     v.z += q.v.z;
     return *this;
}
```

## 四元数的减法：-= 操作符

此操作符会将四元数 $q$ 中的元素逐一地与目前四元数中的元素相减。

假设 $q$ 和 $p$ 皆为四元数，则：

$$q - p = q + (-p) = [n_q - n_p, (x_q - x_p)i + (y_q - y_p)j + (z_q - z_p)k]$$

这里的 $n_q - n_p$ 是四元数和的标量部分，而 $(x_q - x_p)i + (y_q - y_p)j + (z_q - z_p)k$ 是四元数和的向量部分。

以下是从 Quaternion 类中减去四元数 $q$ 的程序代码：

```
inline     Quaternion     Quaternion::operator-=(Quaternion q)
{
```

```
        n -= q.n;
        v.x -= q.v.x;
        v.y -= q.v.y;
        v.z -= q.v.z;
        return *this;
    }
```

## 标量的乘法：*= 操作符

此操作符会逐一地将四元数的元素与标量 $s$ 相乘。这个操作符与附录一中向量的缩放类似。

```
inline     Quaternion Quaternion::operator*=(float s)
{
     n *= s;
     v.x *= s;
     v.y *= s;
     v.z *= s;
     return *this;
}
```

## 标量的除法：/= 操作符

此操作符会逐一地将四元数的元素除以标量 $s$：

```
inline     Quaternion Quaternion::operator/=(float s)
{
     n /= s;
     v.x /= s;
     v.y /= s;
     v.z /= s;
     return *this;
}
```

## 四元数的共轭：~ 操作符

此操作符会计算出共轭四元数 $\sim q$，计算的方式是将向量部分反向。如果 $q = [n, xi + yj + zk]$，则共轭四元数 $\sim q = [n, (-x)i + (-y)j + (-z)k]$。

两个四元数乘积的共轭等于这两个四元数个别的共轭四元数的乘积，不过前后顺序相反。如以下公式所述：

$$\sim(qp) = (\sim p)(\sim q)$$

以下是 Quaternion 类计算共轭四元数的程序代码：

```
Quaternion operator~(void) const { return Quaternion( n,
```

```
                -v.x,
                -v.y,
                -v.z);}
```

# 四元数函数与操作符

以下的函数和重载操作符可用来执行两个四元数、四元数和标量，甚至是四元数与向量之间的运算。这里假设四元数的类型是Quaternion类，而向量类型是附录一的Vector类。

## 四元数的加法：+操作符

此操作符将四元数 *q*1 中的每个元素逐一地与四元数 *q*2 中的元素相加并返回：

```
inline     Quaternion operator+(Quaternion q1, Quaternion q2)
{
     return      Quaternion(     q1.n + q2.n,
                                 q1.v.x + q2.v.x,
                                 q1.v.y + q2.v.y,
                                 q1.v.z + q2.v.z);
}
```

## 四元数的减法：－操作符

此操作符将四元数 *q*1 中的每个元素逐一地减去四元数 *q*2 中的元素并返回：

```
inline     Quaternion operator-(Quaternion q1, Quaternion q2)
{
     return      Quaternion(     q1.n - q2.n,
                                 q1.v.x - q2.v.x,
                                 q1.v.y - q2.v.y,
                                 q1.v.z - q2.v.z);
}
```

## 四元数的乘法：*操作符

此操作符使用以下的公式计算四元数的乘法：

$$\boldsymbol{qp} = n_q n_p - \boldsymbol{v}_q \cdot \boldsymbol{v}_p + n_q \boldsymbol{v}_p + n_p \boldsymbol{v}_q + (\boldsymbol{v}_q \times \boldsymbol{v}_p)$$

其中，$n_q n_p - \boldsymbol{v}_q \cdot \boldsymbol{v}_p$是四元数积的标量部分，而$n_q \boldsymbol{v}_p + n_p \boldsymbol{v}_q + (\boldsymbol{v}_q \times \boldsymbol{v}_p)$是向量部分。请注意$\boldsymbol{v}_q$和$\boldsymbol{v}_p$分别是$\boldsymbol{q}$和$\boldsymbol{p}$的向量部分，“·”是向量的内积操作符，而“×”是向量的外积操作符。

四元数的乘法符合结合律但不符合交换律；所以：

$$q(ph) = (qp)h$$

$$qp \neq pq$$

以下是实现两个四元数 $q1$，$q2$ 相乘的程序代码：

```
inline      Quaternion operator*(Quaternion q1, Quaternion q2)
     return      Quaternion(q1.n*q2.n - q1.v.x*q2.v.x
                          - q1.v.y*q2.v.y - q1.v.z*q2.v.z,
                g1.n*q2.v.x + q1.v.x*q2.n
                     + q1.v.y*q2.v.z - q1.v.z*q2.v.y,
                q1.n*q2.v.y + q1.v.y*q2.n
                     + q1.v.z*q2.v.x - q1.v.x*q2.v.z,
                q1.n*q2.v.z + q1.v.z*q2.n
                     + q1.v.x*q2.v.y - q1.v.y*q2.v.x);
}
```

## 标量的乘法：*操作符

此操作符逐一地将四元数中的元素乘以标量 $s$。以下根据传入四元数与标量的前后顺序实现两个不同的函数：

```
inline        Quaternion operator*(Quaternion q, float s)
{
     return        Quaternion(q.n*s, q.v.x*s, q.v.y*s, q.v.z*s);
}
inline     Quaternion operator*(float s, Quaternion q)
     return        Quaternion(q.n*s, q.v.x*s, q.v.y*s, q.v.z*s);
}
```

## 向量的乘法：*操作符

此操作符在计算四元数 $q$ 与向量 $v$ 的相乘时，可以将向量 $v$ 视为一个标量部分为 0 的四元数。根据四元数与向量的前后顺序不同，必须实现两个不同的函数。因为将向量 $v$ 视为一个标量部分为 0 的四元数，所以运算规则与之前介绍的四元数乘法相同。

```
inline     Quaternion operator*(Quaternion q, Vector v)
{
     return     Quaternion(     -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),
                                  q.n*v.x + q.v.y*v.z - q.v.z*v.y,
                                  q.n*v.y + q.v.z*v.x - q.v.x*v.z,
                                  q.n*v.z + q.v.x*v.y - q.v.y*v.x);
}
inline     Quaternion operator*(Vector v, Quaternion q)
{
     return     Quaternion(     -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),
                                  q.n*v.x + q.v.z*v.y - q.v.y*v.z,
                                  q.n*v.y + q.v.x*v.z - q.v.z*v.x,
                                  q.n*v.z + q.v.y*v.x - q.v.x*v.y);
}
```

## 标量的除法：/操作符

此操作符逐一地将四元数中的元素除以标量 $s$：

```
inline     Quaternion operator/(Quaternion q, float s)
{
     return     Quaternion(q.n/s, q.v.x/s, q.v.y/s, q.v.z/s);
}
```

## QGetAngle（注 1）

此函数会返回以四元数的向量部分为轴而旋转的角度：

```
inline     float QGetAngle(Quaternion q)
{
     return     (float) (2*acos(q.n));
}
```

## QGetAxis

此函数会返回以四元数 $q$ 的向量部分为旋转轴的单位向量（此四元数向量部分的单位向量）：

```
inline       Vector QGetAxis(Quaternion q)
{
     Vector v;
 float m;

     v = q.GetVector();
     m = v.Magnitude();

     if (m <= tol)
          return Vector();
     else
          return v/m;
}
```

## QRrotate

此函数用以下公式使四元数 $p$ 旋转 $q$：

$$p' = (q)(p)(\sim q)$$

这里的 $\sim q$ 是单位四元数 $q$ 的共轭四元数，以下是程序代码：

---

注 1：　在叙述四元数如何用来表示旋转时，请参考第十四章的“四元数”一节。

```
inline      Quaternion QRotate(Quaternion q1, Quaternion q2)
{
     return      q1*q2*(~q1);
}
```

## QVRotate

此函数根据以下公式将向量 $v$ 旋转了单位四元数 $q$。

$$p' = (q)(v)(\sim q)$$

这里的 $\sim q$ 是单位四元数 $q$ 的共轭四元数，以下是程序代码：

```
inline      Vector      QVRotate(Quaternion q, Vector v)
{
     Quaternion t;

     t = q*v*(~q);
     return      t.GetVector();
}
```

## MakeQFromEulerAngles

此函数会由一组欧拉角建立一个四元数。

由一组欧拉角 —— 分别是偏转角（$\psi$）定义对 $z$ 轴的旋转，俯仰角（$\tau$）定义对 $y$ 轴的旋转，滚转角（$\varphi$）定义对 $x$ 轴的旋转，可建立代表旋转角度的四元数。刚开始先分别建立各角度的四元数，接着将这三个四元数根据下列的公式相乘。以下是三个代表不同欧拉角的四元数：

$$q_{\text{roll}} = \{\cos(\varphi/2), [\sin(\varphi/2)]i + 0j + 0k\}$$
$$q_{\text{pitch}} = \{\cos(\tau/2), 0i + [\sin(\tau/2)]j + 0k\}$$
$$q_{\text{yaw}} = \{\cos(\psi/2), 0i + 0j + [\sin(\psi/2)]k\}$$

每个四元数都是单位四元数（注 2）。

现在将这些四元数相乘得到代表三个欧拉角定义的旋转角度（或方位）的四元数：

$$q = q_{\text{yaw}}q_{\text{pitch}}q_{\text{roll}}$$

相乘之后得到：

---

注 2：　由三角函数的关系 $\cos^2\theta + \sin^2\theta = 1$ 可验证此结论。

$$\begin{aligned}\boldsymbol{q} = \{&[\cos(\varphi/2)\cos(\tau/2)\cos(\psi/2) + \sin(\varphi/2)\sin(\tau/2)\sin(\psi/2)],\\ &[\sin(\varphi/2)\cos(\tau/2)\cos(\psi/2) - \cos(\varphi/2)\sin(\tau/2)\sin(\psi/2)]\mathrm{i} +\\ &[\cos(\varphi/2)\sin(\tau/2)\cos(\psi/2) + \sin(\varphi/2)\cos(\tau/2)\sin(\psi/2)]\mathrm{j} +\\ &[\cos(\varphi/2)\cos(\tau/2)\sin(\psi/2) - \sin(\varphi/2)\sin(\tau/2)\cos(\psi/2)]\mathrm{k}\}\end{aligned}$$

以下是传入三个欧拉角并计算出四元数的程序代码：

```
inline    Quaternion    MakeQFromEulerAngles(float x, float y, float z)
{
    Quaternion    q;
    double   roll = DegreesToRadians(x);
    double   pitch = DegreesToRadians(y);
    double   yaw = DegreesToRadians(z);

    double   cyaw, cpitch, croll, syaw, spitch, sroll;
    double   cyawcpitch, syawspitch, cyawspitch, syawcpitch;

    cyaw = cos(0.5f * yaw);
    cpitch = cos(0.5f * pitch);
    croll = cos(0.5f * roll);
    syaw = sin(0.5f * yaw);
    spitch = sin(0.5f * pitch);
    sroll = sin(0.5f * roll);

    cyawcpitch = cyaw*cpitch;
    syawspitch = syaw*spitch;
    cyawspitch = cyaw*spitch;
    syawcpitch = syaw*cpitch;

    q.n = (float) (cyawcpitch * croll + syawspitch * sroll);
    q.v.x = (float) (cyawcpitch * sroll - syawspitch * croll);
    q.v.y = (float) (cyawspitch * croll + syawcpitch * sroll);
    q.v.z = (float) (syawcpitch * croll - cyawspitch * sroll);

    return q;
}
```

## MakeEulerAnglesFromQ

此函数由输入的四元数中算出三个欧拉角。

由四元数算出三个欧拉角的步骤是，先将四元数转换成旋转矩阵，再由旋转矩阵中算出三个欧拉角。假设 $\boldsymbol{R}$ 是一个 3x3 的旋转矩阵：

$$\boldsymbol{R} = \begin{vmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{vmatrix}$$

而假设 $q$ 是一个四元数：

$$q = [n, xi + yj + zk]$$

$R$ 中的每个元素可以如下从 $q$ 中解出：

$$r_{11} = n^2 + x^2 - y^2 - z^2$$
$$r_{21} = 2xy + 2zn$$
$$r_{31} = 2zx - 2yn$$
$$r_{12} = 2xy - 2zn$$
$$r_{22} = n^2 - x^2 + y^2 - z^2$$
$$r_{32} = 2zy + 2xn$$
$$r_{13} = 2xz + 2yn$$
$$r_{23} = 2yz - 2xn$$
$$r_{33} = n^2 - x^2 - y^2 + z^2$$

要从 $R$ 中求得三个欧拉角：偏转角（$\psi$）、俯仰角（$\tau$）、滚转角（$\varphi$），可以使用以下的公式：

$$\sin\tau = -r_{31}$$
$$\tan\varphi = r_{32}/r_{33}$$
$$\tan\psi = r_{21}/r_{11}$$

以下是从输入的四元数中得到三个欧拉角的程序代码，返回值为 Vector 类：

```
inline     Vector     MakeEulerAnglesFromQ(Quaternion q)
{
     double     r11, r21, r31, r32, r33, r12, r13;
     double     q00, q11, q22, q33;
     double     tmp;
     Vector     u;
     q00 = q.n * q.n;
     q11 = q.v.x * q.v.x;
     q22 = q.v.y * q.v.y;
     q33 = q.v.z * q.v.z;

     r11 = q00 + q11 - q22 - q33;
     r21 = 2 * (q.v.x*q.v.y + q.n*q.v.z);
     r31 = 2 * (q.v.x*q.v.z - q.n*q.v.y);
     r32 = 2 * (q.v.y*q.v.z + q.n*q.v.x);
     r33 = q00 - q11 - q22 + q33;

     tmp = fabs(r31);
     if(tmp > 0.999999)
     {
          r12 = 2 * (q.v.x*q.v.y - q.n*q.v.z);
          r13 = 2 * (q.v.x*q.v.z + q.n*q.v.y);
```

```
        u.x = RadiansToDegrees(0.0f); // 滚转角
        u.y = RadiansToDegrees((float) (-(pi/2) * r31/tmp));    // 俯仰角
        u.z = RadiansToDegrees((float) atan2(-r12, -r31*r13)); // 偏转角
        return u;
}

u.x = RadiansToDegrees((float) atan2(r32, r33)); // 滚转角
u.y = RadiansToDegrees((float) asin(-r31));      // 俯仰角
u.z = RadiansToDegrees((float) atan2(r21, r11)); // 偏转角
        return u;

    }
```

## 单位转换函数

以下两个函数用来转换角度的单位。这两个函数并非四元数专用的，在之前某些范例程序中曾用到过。

```
inline     float     DegreesToRadians(float deg)
{
     return deg * pi / 180.0f;
}
inline     float     RadiansToDegrees(float rad)
{
     return rad * 180.0f / pi;
}
```

# 参考文献

一位睿智的老教授曾经对我说：知道所有问题的答案并不重要，只要在需要的时候知道要去哪里找答案即可。根据这种精神，我为本书编写参考资料的清单，其中包含书籍、文章和网络资源。当你为了本书所讨论过的各种主题，寻找额外的资料时，会发现这个清单很有用。我尽我所能将这份清单分类。然而请注意，某些参考资源可能涵盖超出其分类范围的资料。

## 普通物理学和动力学


D.K. Anand, P.F. Cunniff 著 . *Engineering Mechanics Dynamics*. Boston: Houghton Mifflin, 1973 年

Ferdinand P. Beer, E.Russell Johnston Jr.著. *Vector Mechanics for Engineers*. New York: McGraw-Hill, 1988 年

Rene Dugas 著 . *A History of Mechanics*. NewYork: Dover, 1988 年

Jerry H. Ginsberg著. *Advanced Engineering Dynamics*. New York: Cambridge University Press, 1995 年

Michael R. Lindeburg 著 . *Engineer-in-Training Reference Manual*. Belmont, Calif: Professional Publications, 1990 年

J.L. Meriam, L.G.Kraige著. Engineering Mechanics, Vol.2 ,Dynamics. New York: John Wiley & Sons, 1987 年

Harold A. Rothbart编辑. *Mechanical Design Handbook*. New York: McGraw-Hill, 1996 年

Raymond A. Serway 著 . *Physics for Scientists and Engineers*. New York: Saunders College Publishing, 1986 年

## 数学和数值方法

William E. Boyce, Richard C. DiPrima著. *Elementary Differential Equations*. New York: John Wiley & Sons, 1986 年

Erwin Kreyszig著. *Advanced Engineering Mathematics*. New York: John Wiley & Sons, 1988 年

Roland E. Larson, Robert P. Hostetler 著 . *Calculus with Analytic Geometry*. Lexington, Mass: D.C. Heath, 1986 年

William H. Press, Brian P.Flannery, Saul A.Teukolsky, William T. Vetterling著. *Numerical Recipes in Pascal:The Art of Scientific Computing*. New York: Cambridge University Press, 1989 年

## 计算几何

James Arvo 编辑 . *Graphics Gems II*. New York: Academic Press, 1991 年

Nick Bobic 著 . *Advanced Collision Detection Techniques*. 2000 年 3 月于 Gamasutra 网站上发表；URL：*http://www.gamasutra.com/features/20000330/bobic 01.htm*

Mark DaLoura 编辑 . *Game Programming Gems* 第四、五章 . Hingham, Mass: Charles River Media, 2000 年

James Foley, Andries van Dam, Steven Feiner, John Hughes 著 . *Computer Graphics: Principles and Practice*. Reading, Mass: Addison-Wesley, 1996 年

Andrew Glassner 编辑 . *Graphics Gems,* New York: Academic Press, 1990 年

J.E. Goodman , J.O 'Rourke 编辑 . *Handbook of Discrete and Computational Geometry*. Boca Raton, Fla.: CRC Press, 1997 年

Paul Heckbert 编辑 . *Graphics Gems IV*. New York: Academic Press, 1994 年

David Kirk 编辑 . *Graphics Gems III*. New York: Academic Press, 1992 年

Brian Mirtich著. *Fast and Accurate Computation of Polyhedral Mass Properties.* 收录于1996年 *Journal of Graphics Tools vol.1, no.2* 第31～50页. URL: *http://www. merl.com/people/mirtich/pubs.html*（译注1）

Brian Mirtich著. *Rigid Body Contact:Collision Detection to Force Computation.* MERL技术报告:TR 98-01. 发表于1998年5月IEEE International Conference on Robotics and Automation 举办的“接触分析与模拟研讨会”. URL: *http://www.merl.com/papers/TR98-01/*

Brian Mirtich著. *Efficient Algorithms for Two-Phase Collision Detection.* MERL技术报告. TR 97-23. 收录于1998年 *Practical Motion Planning in Robotics: Current Approaches and Future Directions*. K. Gupta, A. P. del Pobil 编辑. URL: *http://www.merl.com/papers/TR97-23/*

Brian Mirtich著. *V-Clip:Fast and Robust Polyhedral Collision Detection.* MERL技术报告: TR 97-05. 收录于1998年 *ACM Transactions on Graphics vol.17, no.3* 第177～208页. URL: *http://www.merl.com/papers/TR97-05/*

Copy right 2000 by Joseph O'Rourke. *comp.graphics.algorithms Frequently Asked Questions.* URL: *http://www.faqs.org/faqs/graphics/algorithms-faq/*

Joseph O'Rourke 著. *Computational Geometry in C.* New York: Cambridge University Press, 1998年

Alan Paeth 编辑. *Graphics Gems V.* New York: Academic Press, 1995年

## 抛体

H.L. Power, J. D. Iversen 著. *Magnus Effect on Spinning Bodies of Revolution.* 收录于1973年4月 *AIAA Journal vol.11, no.4*

## 球类运动的物理学

Robert K. Adair 著. *The Physics of Baseball.* New York: Harper Perennial, 1994年

John M. Davies 著. *The Aerodynamics of Golf Balls.* 收录于1949年9月 *Journal of Applied Physics vol.20, no.9*

---

译注1: 此网页已经不在了，但可从以下URL下载此文件: *http://www.cs.berkeley.edu /~jfc/mirtich/papers/volInt.ps*。

Theodore P. Jorgensen 著 . *The Physics of Golf.* New York: Springer, 1999 年

William M. MacDonald著. *The Physics of the Drive in Golf.* 收录于1991年3月*American Journal of Physics vol.59, no.3*

John J. McPhee, Gordon C.Andrews 著 . *Effect of Sidespin and Wind on Projectile Trajectory, with Particular Application to Golf.* 收录于 1988 年 10 月 *American Journal of Physics vol.56, no.10*

Rabindra D. Mehta 著 . *Aerodynamics of Sports Balls.* 收录于 1985 年 *Annual Review of Fluid Mechanics vol.17*, 第 151 ~ 189 页

Ron Shepard 著 . *Amateur Physics for the Amateur Pool Player.* 1997 年

Robert G. Watts, Steven Baroni著. *Baseball-Bat Collisions and the Resulting Trajectories of Spinning Balls.* 收录于 1989 年 1 月 *American Journal of Physics vol.57, no.1*

Robert G. Watts, Eric Sawyer 著 . *Aerodynamics of a Knuckleball.* 收录于 1975 年 11 月 *American Journal of Physics vol.43, no.11*

## 空气动力学

Ira H. Abbot, Albert E. Von Doenhoff 著 . *Theory of Wing Sections.* New York: Dover, 1959 年

Sighard F. Hoerner, Henry V. Borst著. *Fluid Dynamic Lift.* Bakersfield, Calif.: Hoerner Fluid Dynamics, 1985 年

Sighard F. Hoerner著. *Fluid Dynamic Drag.* Bakersfield, Calif.: Hoerner Fluid Dynamics, 1992 年

Bryan Thwaites 编辑 . *Incompressible Aerodynamics.* New York: Dover, 1960 年

## 流体静力学和流体动力学

B. R. Clayton, R. E. D. Bishop著. *Mechanics of Marine Vehicles.* Houston, Texas: Gulf, 1982 年

Robert L. Daugherty, Joseph B. Franzini, E. John Finnemore 著 . *Fluid Mechanics with Engineering Applications.* New York: McGraw-Hill, 1985 年

Thomas C. Gillmer, Bruce Johnson 著 . *Introduction to Naval Architecture.* Annapolis, Md.: Naval Institute Press, 1982 年

Edward V. Lewis编辑. *Principles of Naval Architecture, Second Revision, Vol.I, Stability and Strength.* Jersey City, N.J.: The Society of Naval Architects and Marine Engineers, 1988年

Edward V. Lewis 编辑 . *Principles of Naval Architecture, SecondRevision, Vol.II, Resistance, Propulsion and Vibration.* Jersey City, N.J.: The Society of Naval Architects and Marine Engineers, 1988年

John Nicholas Newman 著 . *Marine Hydrodynamics.* Cambridge, Mass.: The MIT Press, 1989年

Robert B. Zubaly著. *Applied Naval Architecture.* Jersey City, N.J.: The Society of Naval Architects and Marine Engineers, 1996年

## 汽车的物理学

Brian Beckman著. *Physics of Racing Series.* URL：*http://www.autopedia.com /stuttgart-west/StuttPhysics.html*

## 即时物理模拟

Mark DaLoura 编辑 . *Game Programming Gems Section 2.* Hingham, Mass.: Charles River Media, 2000年

Chris Hecker 著 . *Physics, The Next Frontier.* 收录于1996年10月、1996年11月 *Game Developer* 杂志

Chris Hecker 著 . *Physics, Part 2: Angular Effects.* 收录于1996年12月及1997年1月 *Game Developer* 杂志

Chris Hecker著. *Physics, Part 3: Collision Response.* 收录于1997年3月*Game Developer* 杂志

Chris Hecker 著 . *Physics, Part 4: The Third Dimension.* 收录于1997年6月 *Game Developer* 杂志

Amnon Katz著. *Computational Rigid Vehicle Dynamics.* Malabar, Fla.: Krieger, 1997年

Jeff Lander 著 . *Collision Response: Bouncy, Trouncy, Fun.* 发表于2000年2月8日 Gamasutra 网站上 . URL：*http://www.gamasutra.com/features/20000208 /lander_pfv.htm*

Jeff Lander 著 . *Crashing into the New Year.* 发表于2000年2月10日 Gamasutra 网站上 . URL：*http://www.gamasutra.com/features/20000210 /lander_pfv.htm*

Jeff Lander著. *Lone Game Developer Battles Physics Simulator.* 发表于2000年2月15日Gamasutra网站上. URL: *http://www.gamasutra.com/features/20000215/lander_pfv.htm*

Jeff Lander著. *Trials and Tribulations of Tribology.* 发表于2000年5月10日Gamasutra网站上. URL: *http://www.gamasutra.com/features/20000510/lander_pfv.htm*

Jeff Lander著. *Physics on the Back of a Cocktail Napkin.* 发表于2000年5月16日Gamasutra网站上. URL: *http://www.gamasutra.com/features/20000516/lander_pfv.htm*

Brian Mirtich著. *Impulse-Based Dynamic Simulation of Rigid Body Systems.* 发表于1996年12月加州柏克莱大学博士论文

Brian Mirtich, John Canny著. *Impulse-based Simulation of Rigid Bodies.* 于1995年4月"1995 Symposium on Interactive 3D Graphics"会议发表. URL: *http://www.cs.berkeley.edu/~jfc/mirtich/papers/ibsrb.ps*

Brian Mirtich, John Canny著. *Impulse-Based Dynamic Simulation.* 于1994年2月"Workshop on Algorithmic Foundations of Robotics"会议发表. URL: *http://www.cs.berkeley.edu/~jfc/mirtich/papers/ibds.ps*

Andrew Witkin, David Baraff著. *An Introduction to Physically Based Modeling.* URL: *http://www-2.cs.cmu.edu/afs/cs/user/baraff/www/pbm/pbm.html*

# 索引

本索引让读者方便查找特定词汇在本书里的页码，快速找到所需的内容。传统上，中文书没有编制索引的惯例，因为中文不像英文那样有一套公认而且大家都知道的排序规则（字母顺序）。然而，这是一本工具书，为了方便读者查阅，索引是不可或缺的。为了兼顾“查阅”与“中文化”，我们决定沿袭英文版的编排格式，也就是依照英文字母顺序排列所有条目，所以，我们保留所有原文，如此读者才能快速找到想找的条目，而中文是以辅助说明的方式出现的。

然而，使用英文字母顺序编排意味着读者必须先知道英文词汇原文才能顺利找到其所在的页码。例如，假设读者想知道本书哪几页有提到“域名服务”，那么，你必须先知道其原文是“Domain Name Service”，或是知道其缩写是“DNS”，然后才能推断此条目应该是编在“D”小节。如果读者觉得这样不方便，我们为此感到抱歉，因为我们还没有找到一套大家都公认的中文排序规则，如果我们像编字典那样使用首字笔划顺序来排列，除了不方便查找之外，还必须面临一词多译的问题。例如，有人习惯将“serial port”翻译成“串行端口”，但也有人将它翻译成“顺序端口”，如果你不知道本书采用哪一种译词，要第一次就顺利找到“serial port”的页码，惟一的办法是碰运气。如果运气不好，你必须同时知道 serial port 的每一种可能译法才有机会找到。

## 格式说明

所有条目都是依照英文字母顺序排列。举一个实例说明如何使用本索引，以及索引条目的编排格式。如果你想知道本书哪几页提到“performance tuning（性能调整）”，则必须先翻到“P”小节，然后你会看到：

performance tuning（性能调整）
basic steps（基本步骤），97
capacity planning（容量规划），107-111
External（外部的），102
Internal（内部的），103

如你所见，我们以缩排格式来表示各项信息。通常，从最左侧逐层往右读，可以得到一个符合文法的完整句子或词汇；不过，并非每次都能这样，有时候你得到的只是特定概念的关键词而已，例如：“performance tuning、基本步骤”。所以，这段索引是这样解读的：第97页提到了“性能调整的基本步骤”，在103页提到了“内部的性能调整”。

最后，在不妨碍查阅的前提下，对于第二层与第三层的条目我们会予以中文化。

希望本节的说明有助于读者使用本索引。如果读者对本公司书籍的索引方式有任何意见，请用 E-mail 告诉我们，好让我们知道如何改进。

## 符号

*operator（* 操作符）
in matrix operations（于矩阵运算）
matrix multiplication（矩阵的乘法），305
scalar multiplication（标量的乘法），305
vector multiplication（向量的乘法），307
in quaternion operations（于四元数运算）
quaternion multiplication（四元数的乘法），312
scalar multiplication（标量乘法），312-313
vector multiplication（向量乘法），313
in vector operations（于向量运算）
scalar multiplication（标量乘法），297
vector dot product（向量内积），296-297
*=operator（*= 操作符）
in matrix operations（于矩阵运算），303
in quaternion operations（于四元数运算），310
in vector operations（于向量运算），294
+operator（+ 操作符）
in matrix operations（于矩阵运算），304
in quaternion operations（于四元数运算），311
in vector operations（于向量运算），295
+=operator（+= 操作符）
in matrix operations（于矩阵运算），302
in quaternion operations（于四元数运算），309-310
in vector operations（于向量运算），293
-operator（- 操作符）
in matrix operations（于矩阵运算），304
in quaternion operations（于四元数运算），312
in vector operations（于向量运算）
conjugate（共轭），294-295
vector substraction（向量减法），295
-=operator（-= 操作符）
in matrix operations（于矩阵运算），303
in quaternion operations（于四元数运算），310
in vector operations（于向量运算），293-294

/operator （/^ 操作符）
in matrix operations（于矩阵运算），304
in quaternion operations（于四元数运算），313
in vector operations（于向量运算），298
/=operator（/= 操作符）
in matrix operations（于矩阵运算），303
in quaternion operations（于四元数运算），311
in vector operations（于向量运算），294
~operator in quaternion operations（~ 操作符于四元数运算），311
^operator in vector operations （^ 操作符于向量运算），295-296

A

acceleration（加速度）
angular（角加速度），12，57-62
centripetal（向心加速度），59-62
concepts（加速度概念），34-35
constant（定加速度），35-37
equations for（加速度方程式），78-79
linear units and symbol for（线性加速度、单位和符号），12
nonconstant（不定加速度），37-38
relative（相对加速度），61-62
tangential（切线加速度），59-62
velocity and（速度与加速度），32-35
acceleration vector in law of motion（加速度向量，于运动定律），24
aerodynamic drag（空气动阻力），168-170
in cars（汽车的空气动阻力），171
components of（空气动阻力的分力），168
induced（诱导阻力），168-169
momentum（动量阻力），169
viscous（黏滞阻力），168
wetted（湿阻力），169-170
aerostatic lift（空气静升力），166
ailerons in aircraft（副翼，于飞机上），135
aircraft（飞机），125-148
airfoil（翼切形），127
angle of attack（攻角），127
chord line（翼弦线），127
control（飞机的控制），134-135
ailerons（副翼），135
elevators（升降舵），135
flaps（襟翼），134-135
rudders（方向舵），135
fluid dynamic drag（流体动阻力），129，132
forces on（飞机上的力），125-126
geometry of（飞机的几何形状），126-128
lift（升力），128-129
mean camber line（翼弧中线），127
modeling（模型化），136-148，234-238
fluid dynamic drag（流体动阻力），137-138
lift（升力），137-138
sample code（范例程序代码），139-147
sample program（范例程序），138
steps in（步骤），136
parts of（飞机零件），126，127
pitch axis（俯仰轴），128
roll axis（滚转轴），128
thrust（推力），133，147
yaw axis（偏转轴），128
airfoil（翼切形），127
moving through air（在空气中移动），128-129
stalled（失速），132，133
angle of attack（攻角），127
critical（临界攻角），132
in lift and drag（于升力和阻力），130-131
stalls and（失速和攻角），132，133
angular acceleration（角加速度），57-62，89
angular impulse（角冲量），94
in collisions（于碰撞中），102-103

angular momentum（角动量）
equation（方程式），27-28
in law of motion（于运动定律），25
angular motion（角运动）
defined（定义），13
in rigid body kinetics（于刚体运动学），88-91
angular velocity（角速度），57-62，92
aspect ratio（展弦比），127

## B

banking in cars（边坡于汽车运动学），174
baseball（棒球）
as collision example（碰撞范例），99-101
as Magnus effect example（马格纳斯效应范例），121
Bernoulli 's equation（贝努利方程式），112，129
billiard ball game as collision example（撞球游戏碰撞范例），97-99
boat（小艇）
boundary layer in fluid dynamic drag（边界层于流体动阻力），113
buoyancy force（浮力），68-70，77
in ship flotation（船的漂浮），150-151

## C

cannon ball game fluid dynamic drag example（炮弹游戏 流体动阻力范例），116-117
cars（汽车），171-174
power（功率），172-173
resistance（阻力），171-172
roadway banking（道路边坡），174
stopping distance（刹车距离），173
Cartesian coordinate system（笛卡尔坐标系），12，56-57
center of gravity（重心）
in ship flotation（船的漂浮），150-151
centripetal acceleration（向心加速度），59-62
equation for（方程式），60
centripetal force in car banking（向心力于汽车的倾斜），174
cloth simulation（柔体仿真），274-287
coefficient of restitution（恢复系数），96
collision detection（碰撞侦测），93，209-210
collision response（碰撞反应），93
angular effects（角的影响），215-226
check for collision（碰撞检查），215-225
collision impulse（碰撞冲力），225-226
penetration（穿透），219，225
vertex-edge（顶点对边）214-215，222-225
vertex lists（顶点数组），220
vertex-vertex（顶点对顶点）213-214，216-218
defined（定义），209
in 3D multiple body simulation（于 3D 多重物体仿真中），269-272
implementing（实现），209-226
linear（线性），210-215
check for collision（检查碰撞），211
collision impulse（碰撞冲量），213-214
determination of collision（碰撞的判定），210-212
in particle systems（于粒子系统），285-286
collisions（碰撞），93-105
angular impulse in（角冲量），102-103
ball and bat example（球与球棒的范例），99-101
billiard ball example（撞球的范例），97-99
friction and（摩擦力和碰撞），103-105
golf example（高尔夫球范例），104-105
impact（碰撞），95-101
impulse-momentum principle（冲量－动量定律），94-95
linear impulse in（线性冲量），101-102，213-214

line of action of（作用线），96
of particles（粒子的碰撞），101-102
of rigid bodies（刚体的碰撞），102-103
types of（碰撞类型）
inelastic（非弹性碰撞），96
penetration （穿透）206-207，212
plastic（弹性碰撞），96
vertex-edge（顶点对边），215-216
vertex-vertex（顶点对顶点），215-216
conjugate（共轭）
in quaternion operations（于四元数运算），311
in vector operations（于向量运算），294-295
in vector operations class（于向量运算类别），294-295
constant acceleration（定加速度），35-37
conversion functions in quaternion operations（转换函数 于四元数运算），317
coordinate system（坐标系），12-13
right-handed Cartesian（笛卡尔右手坐标）4，49-50

D

dampers（阻尼）
defined（定义），70
equation for（方程式），70-71
uses for（用法），70
density units and symbol for（密度、单位、和符号），12
determinant in matrix operations（行列式 于矩阵运算），300-301
dihedral angle（上反角），142
displacement （位移）
in angular velocity and acceleration（于角速度和加速度），57
concepts（概念），34
ships and （船与位移），149
vs.distance traveled（对行进距离），34
distance（距离）
skidding（滑行距离），173
stopping（刹车距离），173
distance traveled（行进距离）
equations for（方程式），78-79
vs.displacement（对位移），34
drag（See aerodynamic drag;fluid dynamic drag）（阻力）
drag coefficient（阻力系数），67，116，134

E

elevators in aircraft（升降舵 于飞机），135
equations of motion（运动方程式），75
in real-time simulations（于实时仿真），177-178
for two-dimensional kinetics（对于 2D 动力学），89
error（误差）
Euler（欧拉），180，181
of order（误差的次数），179
truncation（截断误差），179
Euler 's angles in banking of cars（欧拉角于倾斜的汽车），174
Euler 's method（欧拉法）
in 3D rigid body simulation（于 3D 刚体仿真），238-239
improved（改良型欧拉法）
hovercraft example（气垫船范例），195-196
for real-time simulations（实时仿真），184-187
with multiple bodies in 3D（3D 多重物体），267-268
in real-time simulations（实时仿真），178-184
Euler error（欧拉误差），180，181
integration comparison（积分比较），180
integration step（积分间隔），179
sample code（范例程序代码），182-184

explosion（爆炸），50-55
sample code（范例程序代码），52-55
sample program（范例程序），51-52

F

flight controls（飞行控制）
2D rigid body simulator（2D 刚体仿真器），198-202
3D rigid body simulator（3D 刚体仿真器），241-245
flotation in ships（漂浮于船），150-152
fluid dynamic drag（流体动阻力），66-67
inaircraft（于飞机），129，132
around a sphere（球体周围），112-114
boundary layer（边界层），113
cannon ball example（炮弹范例），116-117
drag coefficient（阻力系数），67，116
fast-moving（快速移动），67
in projectiles（于抛射体），111-119
Reynold 's number（雷诺数），114-115
separation point in（分离点），113-114
slow-moving（低速移动），67
of spinning sphere（旋转球体的流体阻动力），119-122
turbulent wake（紊流尾流），113-114
force-at-a-distance（超距力），63
force(s)(作用力），63-74
on aircraft in flight（飞行中的飞机上），125-126
buoyancy and（浮力和作用力），68-70
concepts（概念），63-64
contact（接触），63
defined（定义），71
field（力场），63，64-65
fluid dynamic drag（流体动阻力），66-67
friction（摩擦力），65-66
impulse（冲量），94
pressure and（压力和作用力），68
springs and dampers（弹簧和阻尼），70-71
torque and（力矩和作用力），71-74
units and symbol for（单位和符号），12
friction（摩擦力），65-66
calculation of（计算方法），65-66
coefficients of for common surfaces（一般表面的摩擦系数），66
collisions and（碰撞和摩擦力），103-105
friction drag on body through fluid（在流体中物体的摩擦阻力），11

G

GetScalar（于四元数运算），309
GetVector（于四元数运算），309
golf（高尔夫球）
as collision example（碰撞范例），104-105
as Magnus effect example（马格纳斯效应范例），121

H

Hamilton William，231
Hook 's law（虎克定律），70
horsepower in cars（马力汽车），172-173
hovercraft（气垫船），166-170
aerodynamic drag（空气动阻力），168-170
components of（阻力分力），168
induced（诱导阻力），168-169
momentum（动量阻力），169
viscous（黏滞阻力），168
wetted（湿阻力），169-170
concepts（概念），166-168
aerostatic lift（空气静升力），166-168
skirts for（气垫船的档板），167
2D rigid body simulation（2D 刚体仿真），188-208
linear collision response in（线性碰撞反应），210-215
over water（在水面上），168-170
resistance（阻力），168-170

I

impact of collision（碰撞冲力），95-101，97
impulse（冲量）
  angular（角冲量），94
  collision（碰撞冲量）
    angular（角碰撞冲量），102-103，225-226
    linear（线性碰撞冲量），101-102，213-214
  force（作用力），94
  linear（线性），94
  torque（力矩），94
impulse-momentum principle（冲量－动量定律），94-95
inelastic collisions（非弹性碰撞），96
inertia products of（惯性乘积），28-29
inertia tensors（惯性张量），27-31
  angular momentum equation（角动量方程式），27-28
  products of inertia（惯性积），28-29
  sample code（范例程序代码），30-31
  symmetry（对称），29-30
  transferofaxes（旋转轴），29
instantaneous velocity（瞬间速度），34
  calculation for（瞬间速度的计算），35-36
inverse in matrix operations（转置矩阵 于矩阵运算），301-302

K

kinematics（运动学），32-62
  angular velocity and acceleration（角速度和加速度），57-62
  constant acceleration（定加速度），35-37
  2D particle（2D 粒子），38-40
  3D particle（3D 粒子），40-50
  local coordinate axes（局部坐标轴），56-57
  nonconstant acceleration（不定加速度），37-38
  particle explosion（粒子爆炸），50-55
  rigid body（刚体），56
  velocity and acceleration（速度和加速度），32-35
kinematic viscosity units and symbol for（动黏度，单位和符号），12
kinetic energy（动能）
  collision impact and（碰撞），95
  concepts（概念），95
kinetics（动力学），75-92
  2D particle（2D 粒子），76-81
  3D particle（3D 粒子），81-88
  problem solving guidelines（问题解答指引），76
  rigid body（刚体），88-92
Kutta-Joukouiski theorem Kutta-Joukouiski（理论），120，129

L

length units and symbol for（长度单位和符号），12
lift force（升力），119-122
  in aircraft（飞机上），128-129
  equation for（方程式），120
linear acceleration（线性加速度），12
linear collision response simulation（线性碰撞反应 仿真），210-215
  check for collision（碰撞检查），211
  collision impulse（碰撞冲量），213-214
  determination of collision（碰撞的判定），210-212
linear impulse（线性冲量），94
  in collisions（于碰撞），101-102，213-214
linear momentum in law of motion（线性动量于运动定律），24
linear motion defined（线性运动 定义），13
linear velocity（线性速度），12

## M

magnitude（量值）
in quaternion operations（于四元数运算），309
in vector operations（于向量运算），290-291
Magnus effect（马格纳斯效应），119-123
MakeEulerAnglesFromQ（于四元数运算），316-317
MakeQFromEulerAngles（于四元数运算），314-315
mass（质量）
calculation of（质量的计算），14-16
defined（定义），14
2D example（2D 范例），20-21
units and symbol for（单位和符号），12
variable projectiles of（变动质量抛射体），122-124
virtual of a ship（船舰虚质量），164-165
mass moment of inertia（转动贯量）
calculation of（转动贯量的计算），16-19
defined（定义），14
2D example（2D 范例），21-23
units and symbol for（单位和符号），12
mass properties（质量特性），13-22
matrix addition（矩阵的加法），302-304
matrix functions and operators（矩阵函数和操作符），304-307
matrix multiplication（矩阵的乘法），305
matrix operations（矩阵运算），299-307
matrix functions and operators（矩阵函数和操作符），304-307
matrix addition（矩阵相加），304
matrix multiplication（矩阵相乘），305
matrix substraction（矩阵相减），304
scalar division（与标量相除），304
scalar multiplication（与标量相乘），305
vector multiplication（与向量相乘），307
Matrix3x3（类别），228，299-303
determinant（行列式），300-301
inverse（反矩阵），301-302
matrix addition（矩阵相加），302-303
matrix subtraction（矩阵相减），303
scalar division（与标量相除），303
scalar multiplication（与标量相乘），303
transpose（转置矩阵），301
matrix subtraction（矩阵的减法），303-304
Matrix3x3（类别），228，299-303
measures and units（度量和单位），10-12
metacenter in ship flotation（定倾中心船的漂浮），151
modeling（模型化）
aircraft flight（飞机飞行），136-148，234-238
fluid dynamic drag（流体动阻力），137-138
lift（升力），137-138
sample code（范例程序代码），139-147
sample program（范例程序），138
steps in（步骤），136
steps in（步骤），136
model(s)(模型)
aircraft flight（飞机飞行），136-148，234-238
2D rigid body simulation（2D 刚体仿真），189-195
3D rigid body simulation（3D 刚体仿真），234-238
of multiple bodies in 3D（3D 多重物体），253-267
of particle systems（粒子系统），275-284
moment（See torque）（力矩）
multiple bodies in 3D simulation（3D 多重物体仿真），252-273
collision response（碰撞反应），269-272
integration（积分），267-268
model（模型），253-267

contact（接触力），261-267
forces and moments（力和力矩），259-261
initialization（初始化），253-259
steps in（步骤），252
tuning of（参数调整），272-273

## N

Newton's Law（牛顿运动定律）
of conservation of momentum（动量守衡定律），95
first（第一运动定律），9
of gravitation（于万有引力），64-65
second（motion）（第二运动定律），9-10，23-27，75
third（第三运动定律），9
nonconstant acceleration（不定加速度），37-38
normalize in vector operations（正规化 于向量运算），291-292

## P

parallelepiped volume of（平行六面体 体积），154-156
particle explosion kinematic（粒子爆炸 运动学），50-55
particles（粒子）
collisions between（粒子之间的碰撞），101-102
concepts（概念），32
particle systems（粒子系统），274-287
collision response（碰撞反应），285-286
integration（积分），284-285
model（模型），275-284
initialization（初始化），277-284
particle-spring system（粒子－弹簧系统），275-277
tuning（参数调整），286-287
penalty methods（惩罚法），94
pitch angles（俯仰角），128，227
plastic collisions（弹性碰撞），96
plenum chamber（充气室），166
pressure（压力）
force and（力和压力），68
units and symbol for（单位和符号），12
products of inertia（惯性积），28-29
projectiles（抛射体），106-124
characteristics of（特征），107
drag and（阻力），111-119
Magnus effect（马格纳斯效应），119-123
simple trajectories（基本轨迹），107-111
launch point lower（发射点较低），109
projectile is dropped（抛射体落下），110-111
target and launch at same level（目标点与发射点高度相同），108-109
target point lowe（目标点较高），109-110
terminal velocity（极限速度），117-118
variable mass（变动质量），122-124

## Q

QGetAngle（于四元数运算），313
QGetAxis（于四元数运算），314
QRotate（于四元数运算），314
quaternion addition（四元数的加法），309-311
Quaternion（类别），308-311
quaternion functions and operators（四元数函数和操作符），311-317
quaternion multiplication（四元数的乘法），312
quaternion operations（四元数运算），308-317
Quaternion（类别），308-311
conjugate（共轭），311
GetScalar，309
GetVector，309
Magnitude，309
quaternion addition（与四元数相加），309-310

quaternion substraction（与四元数相减），310
scalar division（与标量相除），311
scalar multiplication（与标量相乘），310
quaternion functions and operators（四元数函数和操作符），311-317
conversion functions（转换函数），317
MakeEulerAnglesFromQ，316-317
MakeQFromEuler ngles，314-315
QgetAngle，313
QgetAxis，314
Qrotate，314
quaternion addition（与四元数相加），311
quaternion multiplication（与四元数相乘），312
quaternion substraction（与四元数相减），312
QVRotate，314
scalar division（与标量相除），313
scalar multiplication（与标量相乘），312-313
vector multiplication（与向量相乘），313
for rigid body rotation（于刚体旋转），231-232
quaternion substraction（四元数的减法），310，312
QVRotate（于四元数运算），314

## R

real-time simulations（实时仿真），176-187
concepts（概念），176
equations of motion and（的运动方程式），177-178
Euler 's method（欧拉法），178-184
improved Euler method for（改良型欧拉法），184-187
Runge-Kutta method for（Runge-Kutta 法），185-186
Taylor 's theorem in（泰勒定理于实时仿真），178-179，184-186
relative acceleration（相对加速度），61-62
relative velocity（相对速度），61
rendering（绘图）
for 2D rigid body simulation（2D 刚体仿真），202-208
for 3D rigid body simulation（3D 刚体仿真），245-251
resistance（阻力）
in cars（汽车），171-172
equation for（阻力方程式），77
in hovercraft（气垫船阻力），168-170
rolling（滚动阻力），171-172
in ships（对于船），162-164
restitution coefficient of（恢复系数），96
reverse in vector operations（反转向量 于向量运算），292
Reynold 's number（雷诺数），114-115
rigid body(ies)(刚体)
circular path of particles making up a（由粒子所组成的圆形路径），58
collisions of（刚体的碰撞），102-103
concepts（概念），32
kinematics of（运动学的），56
kinetics of（动力学的），88-92
multiple in 3D（3D 多重物体），252-273
rigid body rotation（刚体旋转），227-232
in 2D（2D 旋转），227
in 3D（3D 旋转），227
quaternions（四元数），231-232
rotation matrices（旋转矩阵），228-231
Robins effect（罗宾效应），119-122
roll angles（滚转角），128，227
rolling resistance（滚动阻力），171-172
coefficient of（阻力系数），172
rotation matrices（旋转矩阵），228-231
rudders in aircraft（方向舵飞机），135
Runge-Kutta method for real-time simulations (Runge-Kutta 法），185-186

## S

scalar division（标量的除法）
in matrix operations（于矩阵运算），303-304
in quaternion operations（于四元数运算），311，313
in vector operations（于向量运算），294，298
scalar multiplication（标量的乘法）
in matrix operations（于矩阵运算），303，305
in quaternion operations（于四元数运算），310，312-313
in vector operations（于向量运算），294，297
scalar product triple in vector operations（标量三重积于向量运算），298
scalars（标量），13
ships（船），149-165
displacement and（位移），149
2D particle kinetics example（2D 粒子动力学范例），77
flotation（漂浮），150-152
geometry of（几何形状），149，150
resistance（阻力），162-164
virtual mass（虚质量），164-165
volume（体积），152-161
shooting game（射击游戏）
2D particle kinematics example（2D 粒子运动学范例），39-40
3D particle kinematics example（3D 粒子运动学范例），40-50
sample code（范例程序代码），46-50
sample program（范例程序），44-45
3D particle kinetics example（3D 粒子动力学范例），77-88
program screen（程序窗口），86
sample code（范例程序代码），85-88
simulation(s)（仿真）
cloth（布料），274-287
2D rigid body（2D 刚体），188-208
3D rigid body（3D 刚体），233-251
linear collision response（线性碰撞反应），210-215
multiple bodies in 3D（3D 多重物体），252-273
particle systems（粒子系统），274-287
real-time（实时），176-187
rigid body rotation（刚体旋转），227-232
tuning（参数调整），272-273
skidding distance（滑行距离），173
speed（速度）
calculation of（计算方式），33
defined（定义），33
springs（弹簧）
defined（定义），70
equation for（方程式），70
uses for（作用），71
stopping distance in cars（汽车的刹车距离），173

## T

tangential acceleration（切线加速度），59-62
equation for（方程式），60
Taylor 's theorem in real-time simulations（泰勒定理于实时仿真系统），178-179，184-186
tensors（张量）
concepts（概念），26
inertia（惯性），27-31
terminal velocity（极限速度），117-118
tetrahedron volume of（四面体 体积），153-156
3D multiple bodies in（3D 多重物体），252-273
3D particle kinematics（3D 粒子运动学），40-50

vectors（向量），44
x-components（2 分量），41-43
y-components（y 分量），43
z-components（z 分量），44
3D particle kinetics（3D 粒子动力学），81-88
x-components（2 分量），83-84
y-components（y 分量），84
z-components（z 分量），84-85
3D rigid body simulator（3D 刚体仿真器），233-251
flight controls（飞行控制），241-245
integration（积分），238-241
model（模型），234-238
rendering（绘图），245-251
thrust in aircraft（推力飞机），133，147
time units and symbol for（时间单位和符号），12
torque（力矩）
calculation of（力矩的计算），71-73
defined（定义），71
in 3D rigid body kinetics（3D 刚体动力学），89
force and（力和力矩），71-74
impulse（冲量），94
units and symbol for（单位和符号），12
transpose in matrix operations（转置矩阵于矩阵运算），301
triangulated polyhedron simple（三角多面体），153
triple scalar product in vector operations（标量三重积于向量运算），298
truncation error（截断误差），179
sample code for checking（用来检查的范例程序代码），183-184
turbulent wake（紊流尾流），113-114
2D particle kinematics（2D 粒子运动学），38-40
2D particle kinetics（2D 粒子动力学），76-81
2D rigid body simulator（2D 刚体仿真器），188-208
flight controls（飞行控制），198-202
bow thrusters（侧推进器），199-200
propeller（螺旋桨），199
integration（积分），195-198
main elements of（主要部分），188
model（模型），189-195
calculation of forces on vehicle（作用力的计算），192-195
define vehicle structure（定义船体结构），189-190
initialization（初始化），190-192
rendering（绘图），202-208

## U

uniform density defined（密度均匀定义），89
units（单位）
derived（衍生），10
and measures（度量），10-12
universal constant（万有引力常数），64

## V

vector addition（向量的加法），293，295
vector class（向量类别），289-295
vector cross product（向量的外积），60，295-296
vector direction cosines（向量的方向余弦），42
vector dot product（向量的内积），296-297
vector functions and operators（向量的函数和操作符），295-298
vector multiplication（向量的乘法）
in matrix operations（于矩阵运算），307
in quaternion operations（于四元数运算），313
vector operations（向量的操作符），289-298
vector class（向量类别），289-295
conjugate（共轭），294-295
magnitude（量值），290-291
normalize（正规化），291-292

reverse（反转向量），292
scalar division（与标量相除），294
scalar multiplication（与标量相乘），294
vector addition（与向量相加），293
vector substraction（与向量相减），293-294
vector functions and operators（向量函数和操作符），295-298
scalar division（与标量相除），298
scalar multiplication（与标量相乘），297
triple scalar product（标量三重积），298
vector addition（与向量相加），295
vector cross product（与向量的外积），295-296
vector dot product（与向量的内积），296-297
vector substraction（与向量相减），295
vectors（向量），13
vector substraction（向量的减法），293-295
velocity（速度）
acceleration and（加速度与速度），32-35
angular（角速度），57-62，92
equations for（速度方程式），79
instantaneous（瞬间速度），34，35
magnitude of（速度的量值），33
relative（相对速度），61
terminal（极限速度），117-118
units and symbol for（单位和符号）
angular（角速度），12
linear（线性速度），12
virtual mass of a ship（船的虚质量），164-165
viscosity units and symbol for（黏度单位和符号），12
volume（体积）
of a cube（方块的体积），152-153
of parallelepiped（平行六面体的体积），153-156
sample code for finding（求体积的范例程序代码），156-161
of a ship（船的体积），152-161
of a tetrahedron（四面体的体积），153-156

## Y

yaw angles（偏转角），128，227

[General Information]
书名=游戏开发物理学
作者=
页数=337
SS号=11424432
出版日期=

封面
书名
版权
前言
目录
正文