

FUNDAMENTALS OF MATH AND
PHYSICS FOR GAME PROGRAMMERS

游戏编程
数学和物理基础

(美) Wendy Stahler 著

徐明亮 郭红 王琬 胡婷婷 等译



机械工业出版社
China Machine Press

游戏编程数学和物理基础

FUNDAMENTALS OF MATH AND PHYSICS FOR GAME PROGRAMMERS

掌握开发游戏所必需的数学和物理知识

如果你想踏入游戏行业，或者想对代数、几何和三角函数等知识有所了解的话，本书非常适合您！本书向读者介绍了成功开发一款游戏所必需的数学物理概念、定律和公式等。本书讨论了三角几何知识、向量和矩阵运算、变换、碰撞和能量，以及1D/2D/3D空间内的运动等内容，循循善诱地指导读者如何利用数学与物理知识达到提升自身游戏开发水平的目的。

本书包括以下内容：

- 进入3D实时游戏编程高级领域的必备知识。
- 以简易的形式提供开发3D游戏所必需掌握的重要公式。
- 书中程序都采用C++语言编写，教导读者如何将相关概念在具体编程中实现。

本书除包含读者应该掌握的基础知识之外，还包含一些有用的工具，引导读者在实际项目中运用所学到的知识：

- 可视化体验：这些演示程序在交互性环境下，用图形直观地阐述每章中的重要概念。
- 自我测试：在进一步学习下一章之前，对您所学的知识进行检验。
- 练习与程序设计：通过这些训练，提升在实践中运用所学知识的能力。
- 问题讨论：这些问题会对每章中的一些重要概念进行思维拓展。
- 光盘资源：在光盘上可以找到“可视化体验”中用到的程序，使您能够直接感受到所涉及的每个概念。

在线网站 (www.prenhall.com/gamedesdev) 包含以下内容：

- 一些测试题目和项目，可帮助您巩固本书所学内容。
- 一些有用的链接，能够帮助您在游戏开发领域内进一步发展。



www.PearsonEd.com



本书程序范例

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzsj@hzbook.com



华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

封面设计：杨宇梅

游戏开发技术系列丛书

上架指导：计算机/游戏开发

ISBN 978-7-111-23696-2



9 787111 236962

定价：46.00元

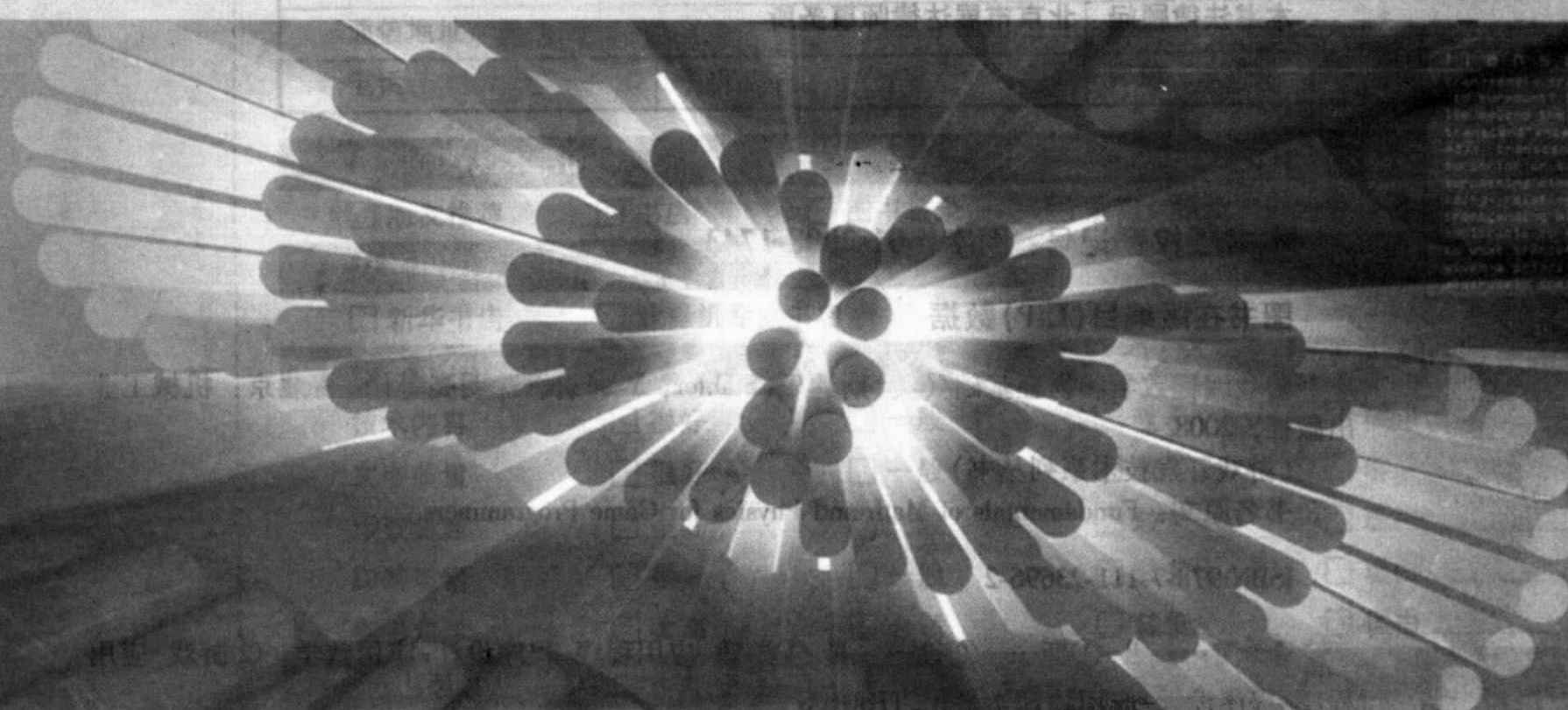
TP319/37D

2008

G 游戏开发技术系列丛书

**FUNDAMENTALS OF MATH AND
PHYSICS FOR GAME PROGRAMMERS**

游戏编程数学和物理基础



(美) Wendy Stahler 著

徐明亮 郭红 王琬 胡婷婷 等译



机械工业出版社
China Machine Press

本书介绍了成功开发一款游戏所必需的数学物理概念、定律和公式等。主要包括三角几何知识、向量和矩阵运算、变换、碰撞和能量,以及1D/2D/3D空间内的运动等专题,循循善诱地教导读者如何利用数学与物理知识提升自身游戏开发水平。

本书适合于所有对游戏开发与编程感兴趣的读者,如果你想踏入游戏行业,或者想对代数、几何和三角函数知识有所了解的话,本书非常适合您!

Simplified Chinese edition copyright © 2008 by Pearson Education Asia Limited and China Machine Press.

Original English language title: Fundamentals of Math and Physics for Game Programmers (ISBN 0-13-168742-5) by Wendy Stahler, Copyright © 2006.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。

版权所有,侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2008-1743

图书在版编目(CIP)数据

游戏编程数学和物理基础/(美)斯达乐(Stahler, W.)著;徐明亮等译. —北京:机械工业出版社, 2008. 4

(游戏开发技术系列丛书)

书名原文: Fundamentals of Math and Physics for Game Programmers

ISBN 978-7-111-23696-2

I. 游… II. ①斯… ②徐… III. ①游戏-应用程序-程序设计-应用数学 ②游戏-应用程序-程序设计-应用物理学 IV. TP301. 6

中国版本图书馆 CIP 数据核字(2008)第 039483 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑:杨庆燕

北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行

2008 年 5 月第 1 版第 1 次印刷

186mm × 240mm · 20 印张

标准书号: ISBN 978-7-111-23696-2

光盘号: ISBN 978-7-89482-595-7

定价: 46.00 元(附光盘)

凡购本书,如有倒页、脱页、缺页,由本社发行部调换

本社购书热线:(010)68326294

译者序

应朋友邀请，为明亮这本《游戏编程数学和物理基础》译著作序，因为书稿要马上付印，所以成文仓促，请作者和读者海涵。

电脑游戏技术日新月异，游戏画面从早期的平面到立体，从抽象到拟真。使得玩游戏享受的不再只是娱乐性，而是更具欣赏性。现在许多游戏的画面看起来如同电影大片，使得某些时候看别人玩游戏也是一种享受，这是因为三维图形技术的不断进步。三维技术的发展也是促使游戏产业市场高速发展的因素之一，我认为其重要性仅次于网络。画面技术的进步，改变了许多人对游戏的印象，不再认为游戏只是给小孩玩的，更多不同年龄层的人成为游戏玩家，如今，成年人游戏玩家已经比未成年的多得多。

为什么这些游戏能让人乐此不疲的沉浸其中？除了它的剧情刺激有趣、引人入胜外，其场景的高度逼真使得玩家们仿佛身临其境，也是一个重要原因。这些均要归功于本书的主题——游戏编程中的数学和物理知识，这也是一个合格的游戏程序员所必须具备的基本数理基础。试想，即便对一个编程水平很高的程序员来说，如果不懂一点牛顿动力学原理，无论如何他也不可能开发出能在凹凸不平的地面上剧烈颠簸的极品飞车；如果不了解一些基本的空间几何知识，恐怕他也很难编写出正确的三维图形处理程序。因此，我建议有志投身于游戏产业的读者，在学习之初，一定要好好复习一下那些曾经在中学或大学期间学习过，但现在已经淡忘的数学和物理知识，并好好体会一下在游戏开发中应该如何运用这些知识。

国内三维图形技术这几年发展的很快，和欧美的差距在不断缩小，但毕竟这门学问起源于欧美，要发展中国的游戏产业，除了国人自身努力外，引进国外的新知也是必须的。身为中国游戏制作者的一份子，我很感谢作者的努力，也感谢译者的辛苦，希望对国内三维技术多添一份力量。在国内游戏研发产业发展迅猛，技术人才供不应求的今天，希望有更多相关书籍出现，为民族游戏产业培育更多人才。

仙剑之父

大宇资讯软星科技(北京)有限公司总经理

姚壮宪

2008年4月13日于北京

前 言

8年前，有人曾问我有哪关于游戏编程中的数学和物理知识的书籍，但当时基本上没有该方面的参考书籍。虽然在一些游戏开发书籍或产业杂志上有一些零零碎碎的相关文章，但是没有一本书是集中讨论游戏编程中的数学物理知识的。于是，我便满怀热情地搜集和整理该方面的资料，并查阅了经典数学、物理教科书和相关文献，结合游戏编程的相关书籍，最终完成了本书的核心部分。

当我在 Full Sail 公司研究游戏设计与开发时，又进一步整理和提炼了相关资料。此时，市面上也出版了一些相关书籍，然而由于这些书籍非常抽象难懂，看起来让人头疼，所以我便打算根据已有的资料为编程人员写一些通俗易懂的东西。起初，编写该书是想作为一本能够提供查阅的工具书而已，后来发现将其作为一本教科书会更合适一些。本书不仅可以作为学习游戏编程中数学物理知识的参考书籍，我更希望它能成为一座里程碑，使你能够在游戏开发的世界中展翅翱翔，获得无穷的乐趣！

适合读者

本书适合于所有对游戏开发与编程感兴趣的读者，书中的大部分实例源代码以及演示程序的代码都放在随书光盘上了。如果你已经对 C++ 比较熟悉，可以对本书所讲的内容进行深一步的研究。不过，在理解具体概念之前，不应该直接进行编程练习，建议首先做一下自我测试中的问题。若要完成问题讨论和项目中的问题，则需要读者对 C++ 有一个全面的理解。请记住，本书不是一本关于 3D 游戏编程的书籍，而只是介绍 3D 游戏编程中所涉及的数学物理知识，此外本书更不是介绍 3D 高等数学的书籍，它只是一本入门书籍，讲解模拟真实 3D 场景的基础知识。如果你对代数学知识掌握扎实的话，本书你将会收获更多的知识。另外你也需对几何学和三角学知识有一定的掌握，这样才能获得更大的成功。

本书概览

本书的前一部分介绍数学基础知识，后一部分介绍物理相关知识，其中前面内容是学习后面物理知识的基础。

第 1 章从点和直线开始，对 2D 和 3D 中的点与直线进行了定义，然后讨论了它们在相交检测中的应用。

第 2 章介绍了游戏编程中的几何学知识，复习了勾股定理、两点间距离公式以及中点公式；然后分别介绍了几种常见几何体的方程，如圆、抛物线和球等，它们会在碰撞检测中进

一步用到。

第3章介绍了游戏编程中用到的三角形知识，定义了6种基本的三角函数，然后讨论了在角度与弧度之间的转换过程，并用程序进行了实现。

第4章介绍了向量的概念，它可用来描述一个物理量，然后介绍了向量的不同形式，并讨论了向量相关的运算，如向量相加、相减、标量与向量相乘、点乘、叉乘和单位化等。

第5章介绍了矩阵以及相关的数学运算，如矩阵相加、相减、标量与矩阵相乘、矩阵相乘、转置等。

第6章介绍了矩阵最好的应用——变换，矩阵变换包括平移、缩放和旋转控制了游戏中所有物体的运动，最后给出了一些关于游戏编程的优化技巧。

第7章介绍物理单位之间的转换，在编程中一定要将所有单位转换成一致的。

第8章开始介绍物理知识，定义了速度、加速度和位移的概念，并给出了它们在一维空间中与时间的关系。

第9章介绍速度和加速度的导数形式，并简单论述了微积分的基础知识。

第10章将前面两章的内容扩展到2D和3D空间中，本章侧重介绍了抛物运动。

第11章介绍作用力与物体运动之间的关系，由牛顿三大运动定律可以确定现实中的所有运动，本章的运动定律与以前章节内容具有很重要的联系。

第12章介绍了两种不同类型的能量，在游戏中可以利用另外一种方法——能量守恒去模拟物体的运动。

第13章介绍游戏中的一个有趣环节——碰撞，游戏若没有碰撞就没有物体间的交互性，该章描述了物体发生碰撞后的响应处理，即物体应该怎样运动。

第14章讨论物体的旋转运动，它对前面介绍的物理概念进行了扩展，以适应于旋转运动。

附录A列出了本书中的一些重要公式。

附录B给出了一些参考书籍和资料，你可以通过它们获得更多知识。

附录C给出了每章后面的自我测试答案。

附录D列出了随书光盘上的内容。

最后，给出了本书中用到的术语表。

每章的开始都提出了每章包含的主题，然后举一些例子来说明这些主题，每节后的“自我测试”部分给出与之相应的练习题。自我测试的答案可以从附录C查阅到，在做完练习后你可以对照答案检验一下。许多章节还包含有“可视化体验”部分，随书光盘上有演示程序与之相对应，有些演示是为了说明书中的一些概念，有些则是具有交互性的演示。这些演示的C++代码都可从光盘上获得，你可以仔细分析一下，弄清楚它们到底是如何实现的。每章都还包括“能力测试”部分，它包括选择题、练习题、程序设计和问题讨论，它们也和每章开头提出的主题是相对应的。

在学习完每章内容后，你将能够对书中的数学原理有一个深入的理解，并能够构造和实现一般的3D物理问题。从此时起，你掌握的知识将会有有一个巨大的飞跃，就可以开始研究

如何将这些知识运用到游戏引擎中，并且在以后的学习中处理更加高级、复杂的问题。

随书光盘内容

本书中包含一些可视化体验例子，它们能够以图像的方式说明书中的相关概念，而且这些例子的源程序也可以在随书光盘上找到，有一些是演示程序，另一些则具有交互性，它们的源代码都可从光盘上找到。通过这些演示程序，你可以明白在 C++ 中是如何处理这些数学、物理运算的。

Circles 演示：在第 2 章用到，它演示了两圆移动直到相交的全过程。

Matrix Vortex 演示：在第 4 章用到，它是一个交互性演示程序，直观地演示了向量以及矩阵的不同运算规则。

Transformations 演示：在第 6 章用到，它可以帮助读者理解对物体进行 2D 变换的原则，同时它也具有交互性，读者可以输入一些变换信息，然后即可观看变换的结果。

1D Racer 演示：在第 8 章用到，演示了具有恒定加速度的物体在一维空间中的运动，同时该演示也回顾了物体的 5 个运动方程。

Cannon Physics 演示：在第 10 章用到，它是一个简单炮弹发射游戏，用户可以控制大炮的初始参数以及墙的位置，然后开始向对手发出攻击，在发射之后可以同时观察到轨道的位置信息以及移动路径等。

Basement Billiards 演示：在第 13 章用到，演示了 2D 球体之间的碰撞，同时也说明了许多关于碰撞响应的相关概念。

上述演示程序都具有一定的趣味性，演示了本书中所讨论的一些数学、物理概念，同时也让读者亲身体会到了这些概念的应用，毕竟它们都是与程序相关的。

教学资源介绍

在本书的随书光盘上，提供了一些教学资源，包括：

教学手册：提供教学中的一些小提示、各章内容概要、教学目标、教学建议、章末问题及解答等。

幻灯片：提供每章内容的相关幻灯片，可以用于教学。

测试题：该测试文件可以使用软件 Prentice Hall's TestGen，它可以从 www.prenhall.com/testgen 免费下载。TestGen 软件是一个强大的测试生成程序，你可以容易地编辑测试题目，将它们转换成测试题，然后以合适的形式输出这些题目。该程序也提供许多选项用于组织和显示测试题，利用内置的随机数以及文本生成器，可以创建多个版本的测试题。此外，该软件强大的搜索和排序功能使你更容易地查找问题并随意地组织它们。

在线网站：www.prenhall.com/gamedesdev

www.prenhall.com/gamedesdev 是一个为学习和教学提供在线支持的工具网站，这里有以下内容：

- 在线学习指导，具有交互性的自我测试网站，为学生提供便利的在线测试机制，以

检查学生对课本知识的理解程度。

- 提供额外的项目资源，可直接用于每章内容的实践中。

作者简介

Wendy Stahler 是位于美国佛罗里达州冬季公园市 (Winter Park) 的 Full Sail Real World Education 学院游戏设计与开发专业的首席教员。在 Full Sail 学院的 6 年里，她致力于游戏编程中数学、物理知识的教学研究。同时，Wendy 也是罗林斯学院 (Rollins College) IT 工程系的副教授，并最近刚开始负责 IT 培训的任务。Wendy 毕业于罗林斯学院，在那里她获得了数学和计算机科学的学士学位，并获得企业传播与技术专业硕士学位。

致谢

爸爸和妈妈——感谢你们的鼓励和支持！

Hap Aziz——感谢您将我带入游戏开发的世界！

Dr. Douglas Child 和 Dr. Donald Griffin——感谢你们提出的宝贵建议！

Gene Woolcock 和 Tony Judice——感谢你们在大学里对我数学与计算机科学方面知识的教导！

Stephanie Wall——感谢您的耐心和支持！

Marc Mencher——感谢您给我提供编写本书的动力！

感谢 Cody Kahrizi 和 Dustin Clingman 为本书提供编程的练习题，同样要感谢程序的编写者 Dustin Henry、Mike Wiggand 和 Mike Fawcett。

特别要感谢我的所有学生，感谢你们提供的有益启发！

质量保证

非常感谢质量保证小组的努力，是你们的认真工作和努力保证了程序的正确性。

技术编辑

Robert Smolenski

Delaware County Community College

Shai Neumann

Brevard Community College

校对

Philip Tavel

DeVry University



目 录

译者序

前言

第 1 章 点和直线	1
1.1 点的定义	1
1.2 线的定义	6
1.3 线的属性	7
1.4 在相交检测中的应用	13
本章总结	20
能力测试	20
第 2 章 相关几何知识	24
2.1 两点间的距离	24
2.2 抛物线	30
2.3 圆和球	32
2.4 在碰撞检测中的应用	36
2.5 碰撞检测的视觉体验	40
本章总结	41
能力测试	41
第 3 章 三角函数	44
3.1 角度和弧度	44
3.2 三角函数	47
3.3 三角函数的性质	55
3.4 使用 C++ 中的数学库	58
本章总结	60
能力测试	60
第 4 章 向量运算	63
4.1 向量与标量	63
4.2 极坐标与笛卡儿坐标	67
4.3 向量加法和减法	71
4.4 标量与向量相乘	75
4.5 点乘	77
4.6 叉乘	80

4.7 可视化体验	85
本章总结	86
能力测试	86
第 5 章 矩阵运算	90
5.1 相等矩阵	90
5.2 矩阵的加减运算	94
5.3 矩阵与标量相乘	97
5.4 矩阵相乘	99
5.5 矩阵转置	104
5.6 可视化体验	107
本章总结	108
能力测试	108
第 6 章 变换	111
6.1 平移	111
6.2 缩放	118
6.3 旋转	124
6.4 组合变换	131
6.5 可视化体验	139
本章总结	141
能力测试	141
第 7 章 单位换算	144
7.1 公制	144
7.2 不同单位制间的转换	146
7.3 计算机转换	150
本章总结	155
能力测试	155
第 8 章 一维空间运动	158
8.1 速度和速率	158
8.2 加速度	162
8.3 运动方程	165

8.4 可视化体验	171	12.2 势能和能量守恒定律	227
本章总结	172	本章总结	231
能力测试	172	能力测试	232
第 9 章 一维空间的运动导数	174	第 13 章 动量和碰撞	234
9.1 速度的图像和导数	174	13.1 有关静止物体的碰撞	234
9.2 加速度图像和二阶导数	179	13.2 向量和冲量	240
本章总结	184	13.3 碰撞建模	244
能力测试	184	13.4 可视化检测	249
第 10 章 二维和三维空间运动	186	本章总结	253
10.1 使用向量	186	能力测试	254
10.2 抛物运动	192	第 14 章 旋转运动	256
10.3 可视化检测	200	14.1 圆周运动	256
本章总结	202	14.2 旋转力学	262
能力测试	202	本章总结	266
第 11 章 牛顿定律	205	能力测试	266
11.1 力	205	附录 A 一些数学和物理公式	268
11.2 牛顿定律	213	附录 B 推荐资源	286
本章总结	217	附录 C 自我测试答案	288
能力测试	217	附录 D 光盘内容	302
第 12 章 能量	220	术语表	304
12.1 功和动能	220		

第1章 点和直线

本章目标

阅读本章内容并完成后面的练习之后，你将能够：

- 在屏幕上用数值表示一个位置
- 在3D空间内用数值表示一个位置
- 计算直线的斜率
- 使用数学方程表示直线
- 使用数学方程表示垂线
- 在3D空间内定义直线
- 检测两条直线是否相交，以及在何处相交

介绍

你是否曾经想过要编写一个游戏，但又不知道该从哪里入手？电脑是怎么知道将一个物体放到何处的？它又是怎样让物体移动的？这一章的内容将会帮你回答这些问题，而且还会为本书的后续章节打下基础。

本章首先定义了点，从而就可以把物体放置在2D平面或者复杂3D场景里的具体位置上；然后建立了坐标系，它将在后续章节里一直沿用。至此，你就可以开始研究2D和3D场景中的直线了。点给出了物体的具体位置，而线能够确定物体的移动轨迹甚至物体的边缘。在讨论完有关直线的所有属性（比如斜率）之后，本章的最后将会讨论一下游戏环境中直线与直线的相交检测问题。

1.1 点的定义

在游戏编程中，需要处理许多不同类型的函数。一般而言，一个函数就相当于一条规则，它接受一些信息（输入信息）并返回由该规则产生的新信息（输出信息）。在编程过程中，函数可以接受不同类型的输入信息，比如变量、文本或者数字等，而返回不同类型的输出信息，甚至输出就是完成一定的功能。在一个编写优秀的程序中，你应该将重复执行的部分编写为函数，从而使书写的代码保持简单性和清晰性。下面让我们来看一个有用而且具有良好注释的例子：

```

//功能:判断一个数字是否是 2 的 n 次幂
//输入:待检测的数字
//输出:如果该数字为 2 的 n 次幂则返回 true,否则返回 false
bool powOfTwo(int num)
{
    return !(num & (num - 1));
}

```

该函数的功能就是, 根据判断传入的整数值是否是 2 的 n 次幂(比如, 16, 32, 64, ...) 返回 true 或者 false。

然而, 在数学中函数被仅仅局限于数值, 即函数接受一个数值, 然后按照指定的规则对它进行运算, 最后输出一个新的数值。在数学中定义函数有两种方法, 其一是把输入值和其对应的输出值构成的有序对——列举出来, 比如下面的一组有序对就定义了一个函数:

(0, 0), (1, 2), (2, 4), (3, 6) 和 (4, 8)

同样也可以把它们用表格的形式表示出来, 如表 1-1 所示。

表 1-1 表格中的有序对

x	y
0	0
1	2
2	4
3	6
4	8

尽管按照上述方法建立一个查找表, 会显得非常简单和快速, 但是该方法存在一些问题。假如你输入的数值在表格的输入栏中找不到, 那么将会得到错误信息。对于表 1-1, 如果你输入数值 5 将会发生什么? 因为在表格的输入栏中找不到输入值 5, 因此计算机将不知道该如何处理这个数值。使用查找表的另一个问题是, 对于输入值的所有可能取值必须全部列举出来, 这就需要一个巨大的表格来保存相关信息, 当然就需要占用较大的存储空间。鉴于此, 我们想到了一个较好的办法。

定义数学函数的第二种方法是, 使用一个能够表达两个数值之间关系的公式或方程。比如上面讨论的函数就可以写成下面的方程, $y = 2x$, 其中 x 为输入值, y 为输出值。试图对每一个有序对都应用该方程, 它们都会适合吗? 我们首先应该清楚地认识到, 这两种方法实际上定义的是同一个函数。然而使用第二种方法会更好, 因为它适用于任何输入值, 而不仅仅是表格中列出的 5 个数值。用这个函数可以把任何实数作为输入, 因此与表格相比, 它需要较少的存储空间。

下面让我们来看一个使用函数的问题: 把物体放置到屏幕的指定位置。当在编写游戏程序时, 你希望计算机把物体放置到屏幕上, 但是计算机又是怎么知道应该把物体放到什么地方的呢? 一般来说, 编程人员是通过使用笛卡儿坐标系来确定物体具体位置的。笛卡儿坐标系由一个水平轴 x 轴和一个垂直轴 y 轴组成, 每个点都可以被定义为一个有序对, 写成 (x, y) , 其中 x 和 y 分别为该点在 x 轴和 y 轴上的坐标值。坐标系的原点是两个坐标轴相交的地方, 其坐标为 $(0, 0)$ 。从原点出发, 向右是 x 轴的正方向, 向左是 x 轴的负方向; 同样, y 轴的正方向向上, y 轴的负方向向下。

【例 1-1】画出 2D 位置

在屏幕上放置 6 个物体 A ~ F, 并在笛卡儿坐标系中画出它们的位置。

A(0, 0)、B(1, 2)、C(4, 3)、D(-1, 2)、E(-2, -1) 和 F(3, -2)

解答:

图 1-1 中画出了这些点的位置。

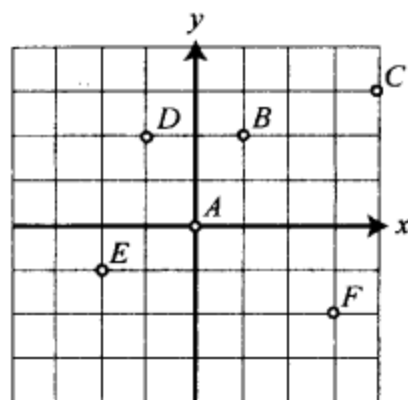


图 1-1 点 A ~ F 在笛卡儿坐标系中的位置

现在我们就可以根据 x 坐标和 y 坐标将物体放置到屏幕上了。

注意

当心不要混淆笛卡儿坐标系和屏幕坐标系。在笛卡儿坐标系中 y 轴正方向是向上的，然而显示器则是被设置为从上往下读，因此屏幕坐标系使用向下作为 y 轴的正方向，见图 1-2。

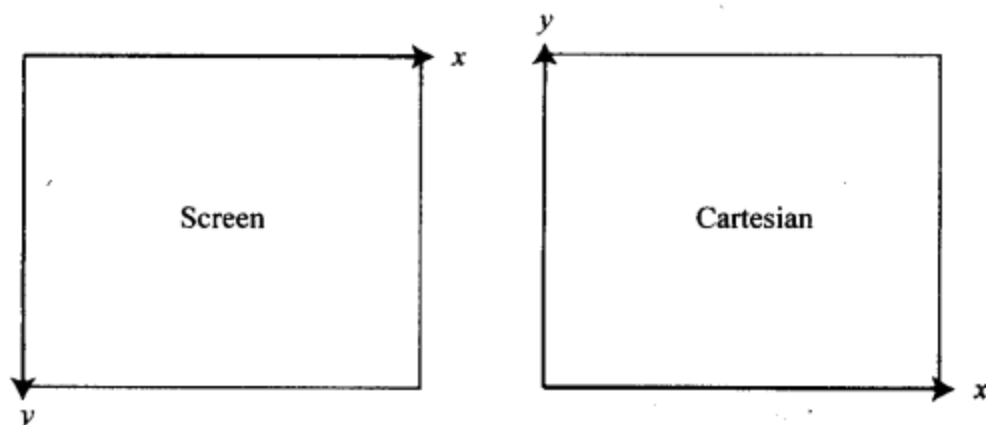


图 1-2 屏幕坐标系

在整个图形渲染管道中，最后一个步骤是把笛卡儿坐标系转换为屏幕坐标系，第 6 章将会介绍该步骤将如何去做，你只需记住，现在所做的任何工作都是在笛卡儿坐标系内完成的，即意味着 y 轴的正方向是向上的。

同样地，可以将坐标系推广到 3D 空间，所要做的仅是添加第三个坐标轴—— z 轴而已。不幸的是，在这里并没有一个统一的行业标准，有些开发包把 y 轴作为竖直方向，而其他开发包则把 z 轴作为竖直方向。在实际项目中，有些引擎使用右手系，而另外一些使用左手系，这是为什么呢？主要原因是人们在定义每个轴时非常任意，因此对于一个具体项目，你必须要清楚它使用的是哪个手系。本书采用 y 轴向上的右手坐标系，如图 1-3 所示。

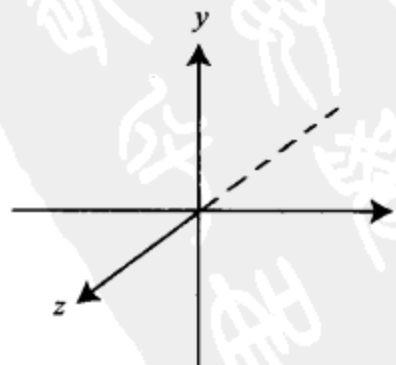


图 1-3 y 轴向上的右手系

在右手系中， x 轴的正方向向右， y 轴的正方向向上， z 轴的正方向由屏幕里指向观察

者。与右手系的名称相比,也许把该坐标系称之为 y 轴向上的坐标系会更加直观。伸出你的右手,弯曲你的无名指和小拇指,让大拇指指向右方(x 轴的正方向),并且让食指指向上方(y 轴的正方向),那么你将会发现中指指向屏幕外部(z 轴的正方向)。如果用左手来做同样的事情,让左大拇指指向右方(x 轴的正方向),食指向上(y 轴的正方向),那么你的中指将会指向屏幕内部(左手系中 z 轴的正方向)。选择 y 轴向上的右手系有下面几个原因:

- 它是传统数学中采用的坐标系。
- 它是大多数编程人员采用的坐标系(但不是全部)。
- 它是 OpenGL 中采用的坐标系。

在这种坐标系下,可以使用一个有序三元组(x, y, z)来确定 3D 空间内的任意一个位置。

【例 1-2】 画出 3D 位置

给出图 1-4 中 P 点的坐标。

解答:

为了得到 P 点坐标,需要首先从原点向右移动 2 个单位,然后向上移动 4 个单位,再沿 z 轴正方向移动 5 个单位,因此点 P 的坐标可以表示为有序三元组(2, 4, 5)。

传统数学采用圆括号符号表示一个点的坐标,然而有些编程人员则采用其他符号来表示同一个坐标,形如 $\langle 2, 4, 5 \rangle$ 或者 $[2, 4, 5]$,我们将在第 4 章和第 5 章中详细介绍这些符号。在程序代码中,也有许多方法存储一个坐标。许多程序员习惯使用简单的数据类型,而另一些人则使用面向对象的方法将点封装成类或者结构体。

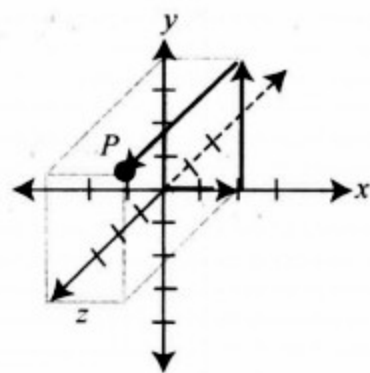


图 1-4 3D 坐标系内一点 P

```
//用来存储点的一些数据类型
//1. 浮点数数组,优点是速度快、简单和内存开销低
float 3dPoint[3];

//2. 包含 3 个浮点数的结构体,优点是可以重载运算符和自定义函数
struct 3dPoint
{
    float x, y, z;

    //重载加号运算符
    3dPoint operator + (const 3dPoint &P2 ) const
    {
        3dPoint temp = { this->x + P2.x;
                        this->y + P2.y;
                        this->z + P2.z;
        };
        return temp;
    }
};
```

使用该结构体可以简单地实现两个点的相加，如下所示：

```
3dPoint A, B, C;  
C = A + B;
```

另外，该方法也可以让你养成利用坐标系考虑物体在游戏中出现位置的习惯。在大多数 2D 游戏中，场景的坐标系原点(0, 0)在屏幕的左上角， x 和 y 值分别向右和向下递增。通常屏幕坐标系是以像素为单位，假如游戏的运行模式为 800×600 ，那么屏幕坐标系的 x 轴就向右延伸 800 个坐标单位， y 轴向下延伸 600 个坐标单位。需要注意的是，大部分 3D 游戏是采用朝向 z 轴负方向的方式开发的，每个坐标单位的长度通常是由你正在使用的 API 预先决定的。

2D 点可以表示为一个二元组(x, y)。

3D 点可以表示为一个三元组(x, y, z)。

自我测试

1. 给出图 1-5 中的 2D 点 $A \sim E$ 的坐标。

2. 在图 1-6 中画出下面的点：

$A(3, 1)$ 、 $B(-3, 1)$ 、 $C(0, 2)$ 、 $D(-2, 0)$ 和 $E(-1, -2)$

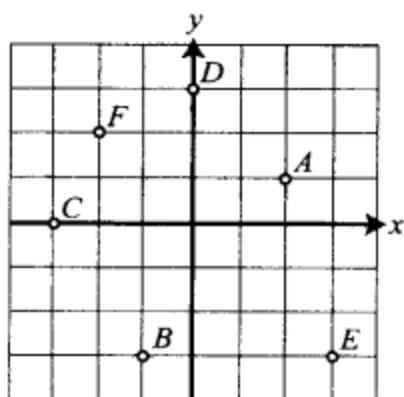


图 1-5 2D 点 $A \sim E$

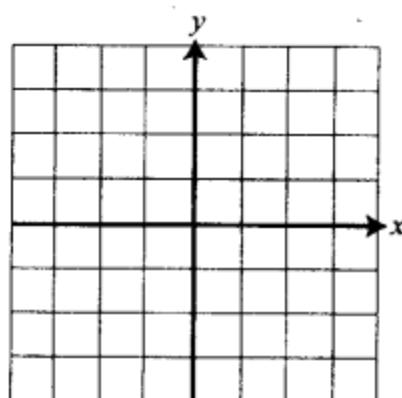


图 1-6 空白网格

3. 给出图 1-7 中 3D 点 P 的坐标。

4. 给出图 1-8 中 3D 点 Q 的坐标。

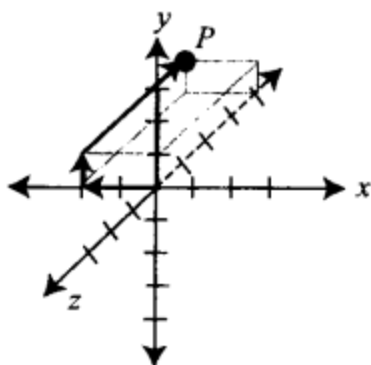


图 1-7 3D 点 P

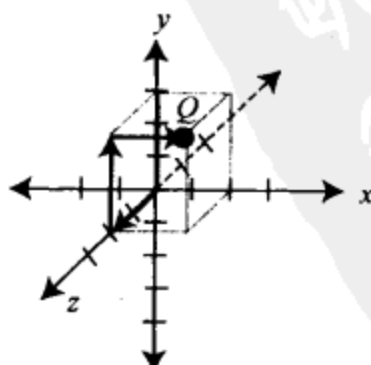


图 1-8 3D 点 Q

1.2 线的定义

线性方程是一类特殊的函数，还记得上一节中我们是如何使用一组有序对来定义函数的吗？所有这些有序对都是直线方程 $y = 2x$ 的一个特殊解，而方程的全解则是所有满足该方程的点。如果把直线方程的所有解都一一画出来，那么将会出现一个有趣的图形。

现在让我们再来回顾一下表 1-1，它列出了满足线性方程 $y = 2x$ 的五个特殊解。在看图 1-9 之前，请将这 5 个点都画出来，看看会出现什么图形。如果不明显你可以再画出更多的解。当你画出的解越来越多时，一条直线就开始慢慢出现了，如图 1-9 所示，这条直线就是方程 $y = 2x$ 的全部解。

到底它是什么图形呢？没错，是一条直线，这就是为什么把它称为直线方程的原因。

如果直线方程的形式为 $Ax + By = C$ ，而且系数 A 和 B 不同时为 0，那么它代表的图形就是一条直线。反之，每条直线的方程形式都是 $Ax + By = C$ ，其中 A 和 B 不同时为 0。

一般情况下，画直线最简单的方法是利用直线方程求出一系列有序对。首先对方程进行变换，使方程的一边只有 y ，然后选择一个 x 值，并代入方程计算出一个 y 值。虽然两个有序对就能够完全确定一条直线，但是我们经常会计算出第三个有序对，以便对结果进行检验。

小技巧

在选择 x 值时要巧妙一些，比如选择较小和易于处理的值；另外对 x 值取 0 值也是非常有用的。

【例 1-3】画线

画出方程 $3x - 2y = 8$ 表示的直线。

解答：

1. 变换方程将 y 移到方程的一边。

$$\begin{aligned} 3x - 2y &= 8 \\ -2y &= -3x + 8 \\ y &= (3/2)x - 4 \end{aligned}$$

2. 在画点时使用整数坐标值会比较容易一些，因此 x 取值为 0, 2 和 4。将这 3 个值代入方程后，将会得到下面三个有序对作为特殊解：

(0, -4)、(2, -1) 和 (4, 2)

3. 画出这 3 个点，并用线将它们连起来（从技术上来说，只需两个点就可以确定一条直线，而第三个点将具有检测的双重作用）。

直线方程 $3x - 2y = 8$ 表示的图形如图 1-10 所示。

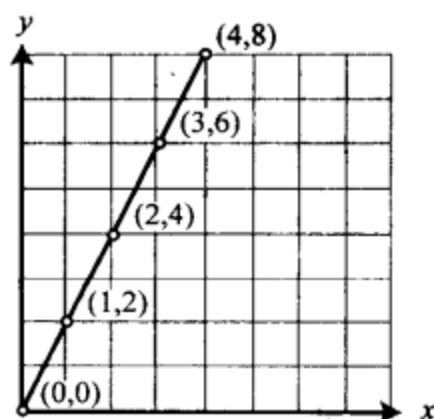


图 1-9 直线方程 $y = 2x$ 的全部解

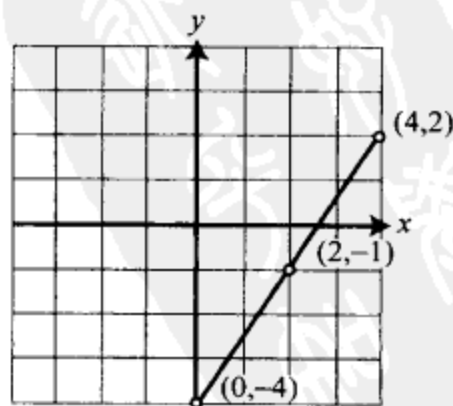


图 1-10 直线方程 $3x - 2y = 8$ 的全部解

【例 1-4】 画水平直线

画出方程为 $y = 3$ 的直线。

解答：

初看这个方程，好像它不是一个直线方程，可能会让你没有一点头绪。然而，把它写成下面的形式：

$$0x + 1y = 3$$

它还是符合形式 $Ax + By = C$ 的。

1. 找到满足该方程的三组特殊解。

2. 选择 3 个 x 值 0、4 和 -4，将它们代入方程，从而得到三个有序对 $(0, 3)$ 、 $(4, 3)$ 和 $(-4, 3)$ 。实际上，无论 x 取什么值，其对应的 y 值总是 3。

3. 画出这三个点，并用线连接它们。

直线 $y = 3$ 表示的图形如图 1-11 所示。

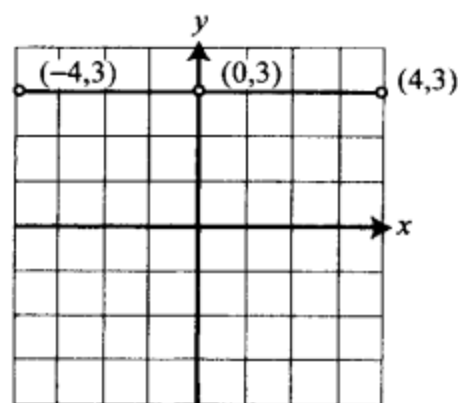


图 1-11 直线 $y = 3$ 的全部解

注意

在这里我们定义了平面直线方程： $Ax + By = C$ ，在下一节我们将会介绍该直线方程的更多细节，并且还会将之推广到 3D 空间。

自我测试

1. 确定下述方程是否是直线方程(看它的全部解是否是一条直线)：

(1) $2x - y = 5$

(2) $-x + 5y = 0$

(3) $x = -1$

(4) $y + x^2 = 5$

2. 画出下面直线方程的图形：

(5) $x - 2y = 0$

(6) $-3x + y = 4$

(7) $x = 1$

1.3 线的属性

在前一节中，我们定义了直线的方程，下面就进一步来看看直线的相关属性。斜率是直线最重要的属性之一，图 1-12 显示的图形是一斜面(直线)，它以速度 50m/s 沿垂直方向上升、以速度 100m/s 沿水平方向运动。

该斜面的斜率是通过垂直上升的速度与水平运动的速度的比率来确定的，在该图中此比率就是 50/100，或 50%。该斜面可以用数学上的直线方程 $1/2x - y = c$ 来表示，如图 1-13 所示。

让我们来看坐标系中的 P 点和 Q 点，该直线每次沿水平方向运动两个单位长度，相应地沿垂直方向上升一个单位长度，那么该直线的斜率将是：

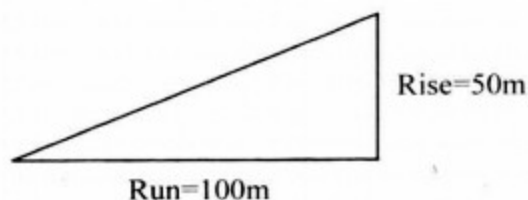
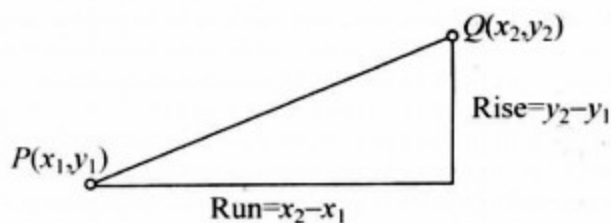


图 1-12 用直线表示的斜面

图 1-13 直线 $1/2x - y = c$ 的斜率为 $1/2$

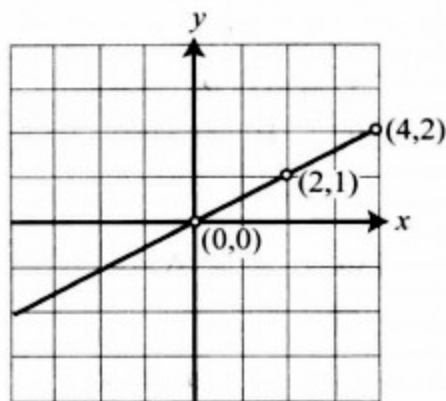
垂直上升速度 / 水平运动速度 = $1/2$

通常我们使用符号 m 来表示斜率，对于由两个点 $P(x_1, y_1)$ 和 $Q(x_2, y_2)$ 确定的非垂直直线来说，其斜率的计算公式如下：

$$\text{斜率} = m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

无论选择直线上的哪两个点，得出的斜率都将是一样的，因为一条直线只有一个斜率。下面来看方程 $1/2x - y = 0$ 的图形，如图 1-14 所示。

从右图可以看出，点 P 、 Q 和原点都位于同一条直线上，如果你使用点 P 和 Q 来计算斜率，结果是 $m = (2 - 1)/(4 - 2) = 1/2$ ；如果使用点 P 和原点，得到的结果为： $m = (1 - 0)/(2 - 0) = 1/2$ ，任何一种方法得到的斜率都是 $1/2$ 。

图 1-14 直线 $1/2x - y = 0$ 的斜率为 $1/2$

【例 1-5】 两点之间的斜率

计算点 $(1, 5)$ 和 $(-2, 0)$ 之间的斜率。

解答：

$$\text{斜率} = m = (y_2 - y_1)/(x_2 - x_1) = (0 - 5)/(-2 - 1) = (-5)/(-3) = 5/3$$

让我们来看一下如何在程序中解决该问题，下面定义的函数功能就是计算两个 2D 点之间的斜率。

```
//功能:计算给定两点之间的斜率
//输入:P1—第一个点(具有两个浮点值的数组)
      P2—第二个点(具有两个浮点值的数组)
//输出:两点之间的斜率
float slopeBetweenPoints ( float * P1, float * P2)
{
    return (P2[1] - P1[1]) / (P2[0] - P1[0]);
}
```

【例 1-6】 直线的斜率

现在让我们来看一下前面所提到的直线的斜率。计算下面两条直线的斜率，并且将它们画到同一个坐标系中。

a. $y = 1/2x + 1$

b. $-3x + 6y = -12$

解答：

在两条直线上分别找到两个点，然后利用斜率公式计算出直线斜率。

a. 点(0, 1)和(4, 3)都满足方程 $y = 1/2x + 1$ 。

利用斜率公式，求出直线斜率为： $m = (3 - 1)/(4 - 0) = 2/4 = 1/2$ 。

b. 点(0, -2)和(3, -1/2)都满足方程 $-3x + 6y = -12$ 。

利用斜率公式，求出直线斜率为： $m = [(-1/2) - (-2)]/(3 - 0) = (3/2)/3 = 1/2$ 。

在知道直线上的两个点之后，就可以画出这两点，并且连接它们形成一条直线，如图 1-15 所示。

观察这两个方程所对应的直线，它们具有同样的斜率 $1/2$ ，而且它们互相平行，该现象非常有意思，难道不是吗？如果它们都以同样的速率延伸，那么它们必然永远保持平行。下面再来看一下直线的斜率，通过它我们可以获得关于直线的更多信息。如果斜率为负值，那么直线就沿左上一右下方向延伸；如果斜率为正值（如图 1-16），那么它就向左下一右上方向延伸；如果斜率为

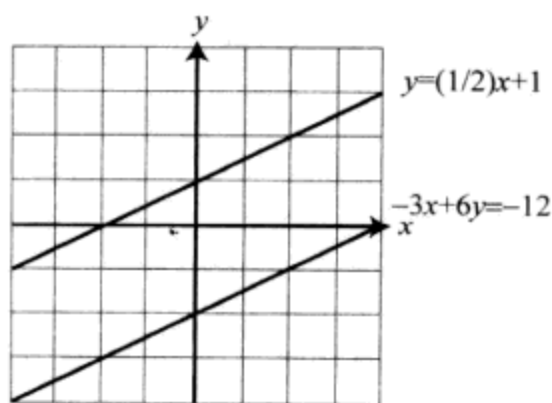


图 1-15 直线 a 和 b

0，该直线将会是一条水平线；最后，如果斜率公式中的分母为 0，那么斜率就不存在，此时的直线将是一条垂直线。通过这几条规律，你可以根据直线的斜率值想象出直线的大致走向。

图 1-16 中的两条直线是平行的，那是因为它们的斜率值是相等的。如果两条直线相互垂直，或者它们相交成直角，那么它们斜率之间的关系会比较有趣。对于相互垂直的任意两条直线，将它们的斜率相乘，结果总是 -1 ，也就是说这两个斜率值互为负倒数。

如果两条直线相互垂直，那么 $m_1 m_2 = -1$ ，或者 $m_1 = -1/m_2$ ，或者 $m_2 = -1/m_1$ 。

下面这个函数包含了上述公式：

```
//功能:计算与给定直线相垂直直线的斜率
//输入:slope——给定直线的斜率
//输出:与给定直线相垂直直线的斜率
float perpSlope ( float slope )
{
    return -1 /slope;
}
```

让我们来看一个该规律的实际应用吧，比如你正在开发一款游戏，确定在某个角度是否会发生碰撞是非常重要的，既然知道两条直线相互垂直，那么用 -1 去除以一条直线的斜率即可得到另一条直线的斜率。因此，如果两条直线相互垂直，那么它们斜率相乘的结果必定是 -1 。

```
//功能:判断两条直线是否相互垂直
//输入:slope1——第一条直线的斜率
      slope2——第二条直线的斜率
//输出:如果两直线相互垂直则返回 true,否则 false
bool arePerp( float slope1, float slope2 )
{
    if( slope1 * slope2 == -1)
        return true;
    return false;
}
```

回顾一下例子 1-6, 我们首先从直线上选择两个点, 然后利用斜率公式计算出直线的斜率。如果直线的方程是标准形式(即形如 $Ax + By = C$), 这里有一种计算斜率的简捷方法。

对于标准形式的直线方程 $Ax + By = C$, 其斜率为: $m = -A/B$ 。

例 1-6 中的第二个直线方程 $-3x + 6y = -12$, 它即是一个标准形式, 因此我们可以使用上述简捷方法来计算其斜率, 斜率 $= -A/B = -(-3)/6 = 1/2$, 其结果和前面采用两点法计算出的结果是一样的, 不过大家一定要注意别忘记带上公式前边的负号哟!

【例 1-7】 计算直线的斜率

计算直线 $2x + y = 5$ 的斜率。

解答:

细心的你可能会发现该直线方程中 y 前没有数字, 不用惊慌! 这意味着 $B = 1$ 。如果方程中没有 y 项, 则 B 一定为 0。

斜率 $= m = -A/B = -2/1 = -2$

这种求斜率的方法适用于采用标准形式的直线方程, 下面还有两种有用的直线方程形式:

斜截式: $y = mx + b$

点斜式: $(y - y_1) = m(x - x_1)$, 其中点 (x_1, y_1) 是直线上任一点。

下面让我们来仔细看一下这两种形式。

斜截式在这三种形式中可能是最经常使用的, 尤其是在程序中更加常见, 这是因为它的 y 值(作为输出值)等于方程另一边的运算结果, 更易于在程序中表述公式。在斜截式中, m 值代表直线斜率, 但是你可能不知道 b 值的含义, 其实它代表直线在 y 轴上的截距, 也就是直线与 y 轴交点的纵坐标。我们可以从直线的斜截式直接地知道它与 y 轴的交点位置, 通过斜率可以知道它的倾斜程度, 能够让人更加直观地想象出直线的形状来。

注意

直线在 y 轴上的截距, 就是横坐标 $x = 0$ 时的 y 值; 同样地, 直线在 x 轴上的截距是纵坐标 $y = 0$ 时的 x 值。

更有趣的是, 直线方程的点斜式比其他两种形式有更多的优势, 如果你想从两个点产生直线的方程, 该怎么做呢? 举个例子, 假如你知道你的当前位置和想要到达的目标位置, 那

么怎样计算出到达目的的直线方程呢？直线的点斜式提供了一种简捷的计算方式，只需知道直线的斜率和直线上的一点即可确定直线方程。在由两点计算出斜率之后，就可以获得直线的相关信息了。

【例 1-8】 计算直线方程

假如人物角色在游戏中的位置为(50, 200)，当玩家在点(150, 400)处点击了鼠标，这说明它想要去此位置，那么就需要找到一条到达目的地的直线路径，请计算出该直线方程。

解答：

要想从两个点产生直线方程，可以使用点斜式计算，它所需要的就是直线的斜率和直线上一点。

$$\text{斜率} = m = (400 - 200) / (150 - 50) = 200 / 100 = 2$$

当然了，你可以使用两个点中的任何一个，因为它们都在一条直线上。在这里我们采用起始点(50, 200)，然后把它代入到点斜式方程中：

$$(y - y_1) = m(x - x_1)$$

$$(y - 200) = 2(x - 50)$$

如果你想把该直线方程变换成斜截式的话，只需经过两步代数运算即可。

$$(y - 200) = 2(x - 50)$$

$$y - 200 = 2x - 100$$

$$y = 2x + 100$$

有了直线方程，现在就可以让人物角色沿着直线运动了。对于角色每一步要移动的 x 值，使用斜截式计算出对应的 y 值，然后将角色放置到该位置即可。从本例可以看出，使用点斜式可以方便地计算出直线的方程，使用斜截式可以容易地使用该方程。

现在让我们把这种处理方法与前面讨论的垂直直线知识结合起来。

【例 1-9】 计算垂直直线的方程

想象一下，在你的游戏中角色正在沿着直线 $y = 2/3x + 20$ 移动，当它到达位置(30, 40)时玩家按下了方向按钮，即命令它向左转 90° ，然后继续沿直线前进，请计算出新路径的直线方程。

解答：

和上面最后一个例子类似，只要找到直线上的一点和直线的斜率就可以很快地写出该直线的点斜式。现在我们已经知道了直线上的一点，因为角色在位置(30, 40)处发生了转弯，所以现在只需找到与原来路径相垂直直线的斜率即可。

如果原先路径的直线斜率为 $2/3$ ，它的负倒数为 $-3/2$ ，因而新路径直线的斜率就等于 $-3/2$ 。

现在只需把这两项代入直线的点斜式即可。

$$(y - y_1) = m(x - x_1)$$

$$(y - 40) = -3/2(x - 30)$$

如果你喜欢的话，可以把该答案改成斜截式：

$$(y - 40) = -3/2(x - 30)$$

$$(y - 40) = (-3/2)x + 45$$

$$y = (-3/2)x + 85$$

例子 1-9 如图 1-16 所示。

注意

在第 4 章中，你将知道应该怎样使用向量运算去完成该过程，同样可以将其容易地扩展到 3D 空间中。

除了能够快速计算出直线方程之外，点斜式的另一个优势是可以容易地将其扩展到 3D 空间。在 2D 空间中，给定一个点和一个斜率就可以确定一条直线。回顾一下例子 1-9，新的起始点 (30, 40) 和斜率 $-3/2$ 就确定了新路径的直线方程。在第 4 章“向量”中，我们将会使用向量符号来表示直线。在向量形式中，点的表示形式为 $\langle x, y \rangle$ 或 $\langle 30, 40 \rangle$ ，斜率的表示形式为 $\langle \Delta x, \Delta y \rangle$ 或 $\langle 2, -3 \rangle$ 。采用这种表示形式我们也能容易地定义 3D 直线，其中 3D 点的形式为 $\langle x, y, z \rangle$ ，斜率的形式为 $\langle \Delta x, \Delta y, \Delta z \rangle$ 。

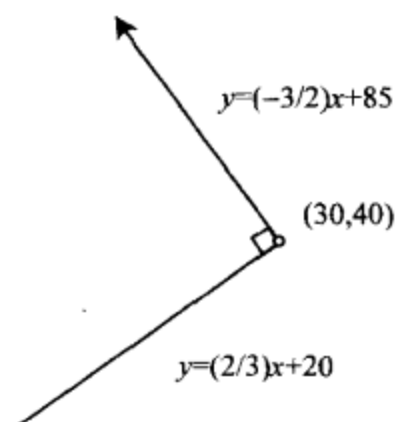


图 1-16 例 1-9 的示意图

【例 1-10】 确定 3D 直线

在 3D 游戏场景中，假如角色当前所处的位置为 (50, 200, 75)，你希望它移动到位置 (100, 50, 225)，确定该角色到达目标位置的直线。

解答：

已知直线上的一点 $\langle 50, 200, 75 \rangle$ ，现在只需计算出 Δx ， Δy 和 Δz 即可。

$$\Delta x = 100 - 50 = 50$$

$$\Delta y = 50 - 200 = -150$$

$$\Delta z = 225 - 75 = 150$$

因而，该直线可以通过点 $\langle 50, 200, 75 \rangle$ 和移动方向 $\langle 50, -150, 150 \rangle$ 来确定。

自我测试

计算出由下列每组点所确定的直线斜率：

1. (0, 10) 和 (5, 0)
2. (3, 5) 和 (1, 9)
3. (2, -1) 和 (6, 1)
4. (-2, -5) 和 (1, 4)
5. (-3, 5) 和 (-4, 7/2)
6. (9, 8) 和 (9, -7)
7. (4, 2) 和 (-2, 1)
8. (3, 7) 和 (-8, 7)
9. 第 1 题和第 2 题中的两条直线具有什么样的关系？
10. 描述出第 6 题中直线的形状。

11. 描述出第8题中直线的形状。

计算下列直线的斜率：

12. $2x + 3y = 10$

13. $x - 5y = 0$

14. $2y = 8$

15. $x + y = -7$

计算出下列经过每组两个点的直线方程：

16. (0, 10)和(5, 0)

17. (3, 5)和(1, 9)

18. (2, -1)和(6, 1)

19. (-2, -5)和(1, 4)

20. (2, 0, -1)和(3, 4, 5)

21. (-3, 1, 5)和(0, 8, -2)

1.4 在相交检测中的应用

在游戏编程中，我们经常要判断两条直线是否相交，如果相交的话那么交点在哪里？直线在游戏中可以代表建筑物的边界、地面或者物体的移动路径等，因此需要编写一个判断两条直线是否相交的程序。通过前面的学习，我们已经能够计算出直线的方程，那么下一步就是将这两条直线的方程联立组成一个线性方程组，然后就可以利用数学方法解决该问题了。在本节中我们首先讨论方程组解的三种情况，然后求出其精确解。

求解由两个线性方程组成的方程组，实际上相当于在求两条直线的交点，方程组的解是所有同时满足两个方程的点的集合。图 1-17、图 1-18 和图 1-19 分别画出了方程组解的三种情况，在图 1-17 中直线：

$$\begin{cases} 2x + 3y = 3 \\ -x + 3y = -6 \end{cases}$$

仅相交于一点，用 P 来表示，它在图中的大致位置是(3, -1)。该结果很容易验证，将其分别代入两个方程检查是否满足即可，在大多数情况下图形并不能精确地描述交点位置，在后面我们会介绍如何去计算交点的精确坐标位置。

$$2(3) + 3(-1) = 3$$

$$6 - 3 = 3\checkmark$$

$$-(3) + 3(-1) = -6$$

$$-3 - 3 = -6\checkmark$$

从图中可以发现，两条直线的斜率是不相等的。

在图 1-18 中，直线：

$$\begin{cases} -3x + 6y = 6 \\ -x + 2y = 2 \end{cases}$$

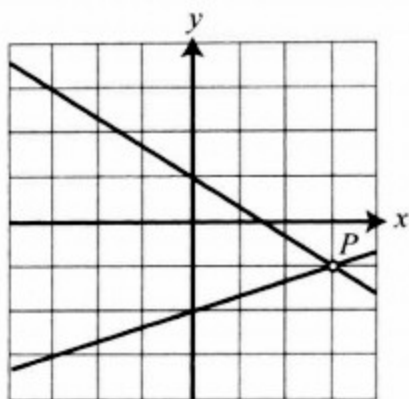


图 1-17 两条直线相交于一点

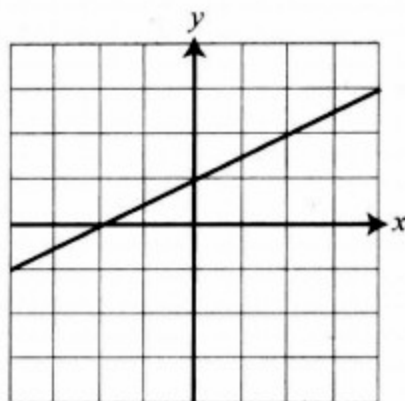


图 1-18 两条直线重合

重叠在一起了,此时方程组的解不仅仅是一个点了,而是直线上的所有点。从图中我们可以发现这两条直线具有相同的斜率和 y 轴上的截距。

最后,在图 1-19 中,直线:

$$\begin{cases} -x + 2y = 2 \\ -x + 2y = -2 \end{cases}$$

相互平行,方程组的解为空集,这是因为两条直线没有公共点,即它们永远不会相交。从图中可以发现,两条直线具有相同的斜率而在 y 轴上的截距不相等。

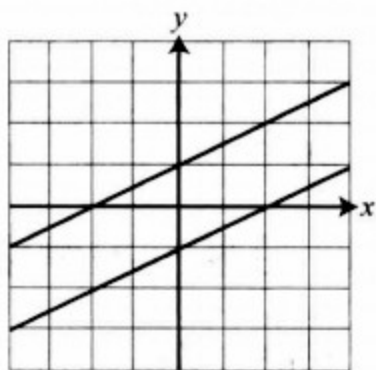


图 1-19 两条直线平行

正如你所看到的,线性方程组解的数目与直线的斜率以及它们在 y 轴上的截距有关,让我们总结一下这三种情况。

结论

由在同一平面内的两条直线组成的线性方程组,其解的情况如下:

- 如果两条直线的斜率不相等,则仅有一组解。
- 如果两条直线的斜率以及在 y 轴上的截距分别相等,则有无穷组解。
- 如果两条直线的斜率相等,而 y 轴上的截距不相等,则方程组无解。

下面的处理过程可以容易地用程序来实现,其算法如下:

分别计算出直线的斜率 m_1 和 m_2 。

1. 如果 $m_1 \neq m_2$, 则输出只有一组解。
2. 如果 $m_1 = m_2$, 分别计算直线在 y 轴上的截距 b_1 和 b_2 ,
如果 $b_1 \neq b_2$, 则输出 0 组解;
如果 $b_1 = b_2$, 则输出无穷组解。

【例 1-11】 线性方程组

画出下面的直线方程组(两条直线),并指出方程组解的数目(即存在的交点数目)。

$$\begin{cases} x + y = 2 \\ -x + 2y = -2 \end{cases}$$

解答:

在图 1-20 中画出了这两条直线,从图中可以粗略看出它们有一个交点,坐标大致是

(2, 0), 用图得出的结果不是那么准确, 下面让我们用算法来验证一下:

1. 计算直线的斜率 m_1 和 m_2 , 由于两条直线都是标准型, 因此可以利用公式 $-A/B$ 来计算:

$$m_1 = -1/1 = -1$$

$$m_2 = -(-1)/2 = 1/2$$

2. 显然两条直线的斜率不相等 ($m_1 \neq m_2$), 因此它们必有一个交点。

【例 1-12】 线线相交检测

屏幕上一小球沿着直线 $-x + 2y = -2$ 运动, 现沿着直线 $3x - 6y = -6$ 放置一堵墙, 问该小球是否会撞到墙上?

解答:

你可以采取例 1-11 的处理方法, 即首先画出它们的图形, 然后再进行判断。但是图形往往会误导人们的视线, 因此这里直接利用算法来处理:

1. 首先将这两条直线写成斜截式, 此时很容易就能得到直线的斜率和 y 轴截距:

$$-x + 2y = -2 \rightarrow y = \frac{1}{2}x - 1$$

$$3x - 6y = -6 \rightarrow y = \frac{1}{2}x + 1$$

2. 找到直线的斜率: $m_1 = \frac{1}{2}$ 和 $m_2 = \frac{1}{2}$ 。

3. 显然 $m_1 = m_2$, 找到 y 轴的截距: $b_1 = -1$ 和 $b_2 = 1$ 。

4. 显然 $b_1 \neq b_2$, 输出 0 组解。因为它们互相平行, 那么永远不会相交, 这也就意味着小球不会撞到墙上。

当要检测两条直线是否相交时, 该算法非常有用。在上述例子中, 如果你在等待小球撞到墙上, 那么将会发生什么? 很显然, 你将陷入一个无穷循环——等待一个永远不会发生的事情, 因此该算法可以帮助你避免陷入困境, 通过程序检测直线是否相交在游戏中的确非常重要, 而且该算法又是那么的快速, 为什么不使用呢?

当检测到两条直线相交于一点后, 下一步就是要求找出该位置, 计算该位置的坐标有两种方法: 联合法和代入法。你也许已经听说过二元一次方程组, 也就是说在两个线性方程中具有两个未知、待求的变量, 在该情形下为 x 和 y , 这两种方法中的任何一种都可求出未知变量。

首先让我们来看联合法, 首先利用等式的性质将方程组转换成容易求解的形式, 这可以通过将方程两边同时乘上一个非 0 的数值来完成, 最终将方程组转化成 x 的系数相同或者 y 的系数相同 (记住系数就是 x 或 y 前面的数字)。假如你有下面的线性方程组:

$$3x + 2y = 10$$

$$4x + 3y = 6$$

首先对方程组进行变换, 使变量 y 的系数相同。将第一个方程的两边同时乘上 3, 从而

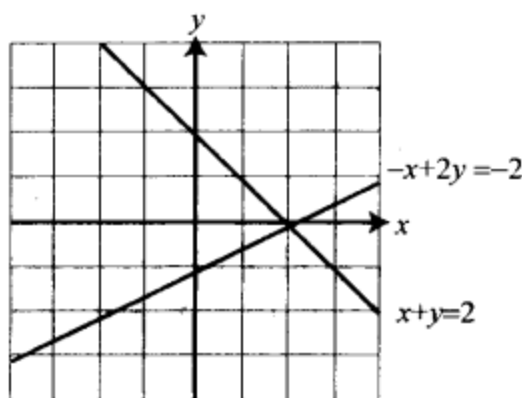


图 1-20 例 1-11 中的直线

得到与方程 $3x + 2y = 10$ 等价的方程 $9x + 6y = 30$ ；然后将第二个方程的两边同时乘上 2，从而得到方程 $8x + 6y = 12$ 。它们构成的方程组与前面的方程组是等价的，不过此时它们 y 变量的系数是相等的。

$$9x + 6y = 30$$

$$8x + 6y = 12$$

下一步就是將这两个方程合并成一个，该步骤通过用一个方程减去另一个方程即可实现。在这里我们用上面方程减去下面方程：

$$9x + 6y = 30$$

$$-(8x + 6y = 12)$$

$$x + 0y = 18$$

$$x = 18$$

方程 $x = 18$ 被称作原始方程组的联合方程，此时已经得到了 x 的具体值，下面所要做的就是將 18 代入原始方程得到 y 的值。如果將 18 代入上面方程，你将得到：

$$3(18) + 2y = 10$$

因此 $y = -22$ 。

于是该线性方程组的解就是 $(18, -22)$ ，这有什么几何意义呢？请记住，该解就是两条直线的交点坐标。下面总结一下利用联合法求解方程组的步骤：

1. 选择你要消去的变量(x 或 y)。
2. 将两个方程分别乘上一个非 0 值，使你想要消去的变量前的系数相同。(请记住利用小数值会使计算方便一些。)
3. 用一个方程减去另一个方程，得到方程组的联合方程。
4. 求出方程组中的第一个变量值。
5. 将得出的变量值代入原始方程，求出另外一个变量的值。

下面这个例子采用的就是上述方法，也正好说明了方法的具体步骤。

【例 1-13】 用联合法求交点

假如在你的游戏中，一辆汽车沿着直线 $3x + 5y = 8$ 行驶，而一堵墙被放置在直线 $x + 3y = 4$ 处，如果汽车仍沿着原来的路线前进，它是否会撞到墙上？如果发生相撞，那么相撞点是多少？

解答：

1. 检测汽车是否会撞到墙上，第一条直线的斜率是 $-3/5$ ，第二条直线的斜率为 $-1/3$ ，因此它们必相交。

2. 利用联合法求出下列直线的交点。

$$3x + 5y = 8$$

$$x + 3y = 4$$

3. 选择你将要消去的变量，假如在这里我们将来消去变量 x 。

4. 在方程两边同时乘上一个非 0 数值，使 x 前的系数相同。在本题中，在下面方程的

两边同时乘上 3, 从而得到下面的方程组:

$$3x + 5y = 8$$

$$3x + 9y = 12$$

5. 用上面方程减去下面方程, 可得到 $0x - 4y = -4$ 。(同样地, 你也可以用下面方程减去上面方程。)

6. 从第 5 步中可得到等式 $-4y = -4$, 从而可解出 $y = 1$ 。

7. 将 $y = 1$ 代入原始方程组中的任何一个等式, 解出变量 x 的值, 看起来将其代入下面的方程会比较容易求解一些。

8. 代入 y 的值, 得到 $x + 3(1) = 4$, 即 $x = 1$, 因此原方程组的解就是 $(1, 1)$ 。

下面让我们来看求解线性方程组的另一种方法: 代入法。该方法需要你具有一定的代数知识, 而且也是每次求解出一个未知数。首先选择一个原始方程, 将其变换成一个未知数的形式, 也就是说等式的一边是该未知数, 余下的元素都位于等式的另一边。假如方程组中的一个方程为:

$$x + 2y = 5$$

为了求解出未知数 x , 需要将方程的两边同时减去 $2y$, 因而可得到

$$x = -2y + 5$$

然后将上述等式代入另一个方程消去 x , 假如另一个方程为:

$$3x - 2y = -1$$

从而得到 $3(-2y + 5) - 2y = -1$ 。

下面所要进行的的就是代数运算了, 此时我们想求出未知数 y , 进行相应的代数运算后你会得到:

$$3(-2y + 5) - 2y = -1$$

$$-6y + 15 - 2y = -1$$

$$-8y + 15 = -1$$

$$-8y = -16$$

$$y = 2$$

最后一个步骤与联合法一样, 将 $y = 2$ 代入原始方程从而求出未知数 x 的值, 如果将其代入第一个方程, 你将得到下列等式:

$$x + 2(2) = 5$$

因此 $x = 1$ 。

这样, 原方程组的解就是 $(1, 2)$ 。

和联合法一样, 下面我们对代入法求解方程组的步骤进行一下总结:

1. 从原始方程组中选择一个等式, 对其进行变换使一个未知数用另外一个未知数表示出来, 即一个未知数位于方程等式的一边, 而余下的元素全部位于等式另一边。

2. 将在上步中得到的等式带入原始方程组中的另一个方程, 此时就可以消去一个未知数。

3. 求解出一个未知数的值。
 4. 将在步骤 3 中求得的数值代入原始方程等式，从而求出另外一个未知数。
- 好，下面让我们来看另外一个例子。

【例 1-14】 用代入法求交点

利用代入法求解下列线性方程组：

$$-3x + y = 8$$

$$5x - 2y = 9$$

解答：

1. 观察原始方程组中的两个方程，利用第一个等式计算 y 会比较容易，因为在该方程中它前面的系数为 1。在方程两边同时加上 $3x$ ，可以得到等式：

$$y = 3x + 8$$

2. 将 $y = 3x + 8$ 代入方程组中下面那个方程，结果是

$$5x - 2(3x + 8) = 9$$

你将会发现，此时已经把未知数 y 消去了。

3. 求解在步骤 2 中得到的等式，得出 x 的值。

$$5x - 2(3x + 8) = 9$$

$$5x - 6x - 16 = 9$$

$$-x = 25$$

$$x = -25$$

4. 将 $x = -25$ 代入原始方程，求出未知数 y 。在这里将其代入上面那个方程，得到：

$$-3(-25) + y = 8$$

因此， $y = -67$ 。

即原方程组的解就是 $(-25, -67)$ 。

上面所讨论的两种方法都可用于求解具有一组解的线性方程组。如果方程中的一个未知数的系数为 1，此时利用代入法计算会容易一些；否则就应该利用联合法。对于一个特定的方程组，采用合适的方法往往会节省大量时间，因此你需要熟练地掌握这两种方法。

现在面临的问题是，如何用程序来定义直线，以及如何去检测它们是否相交。假如我们利用直线的斜率和直线上任一点来定义直线，那么直线方程将是下列形式：

$$y - y_1 = m(x - x_1)$$

其中， m 为直线的斜率，点 (x_1, y_1) 是直线上任一点坐标。假如直线写成 y 用 x 表示的形式，假如有如下两个方程：

$$y = m_1(x - x_1) + y_1$$

$$y = m_2(x - x_2) + y_2$$

用代入法便可求出未知数 x ：

$$x = (m_1x_1 - m_2x_2 + y_2 - y_1) / (m_1 - m_2)$$

好，下面让我们来写一个函数，当给定两条直线时返回它们的交点位置，该函数假定两

条直线不是互相平行的。

```
//功能:找到两条直线的交点
//输入:L1Point——位于第一条直线上的一个2D点
//      L1Slope——第一条直线的斜率
//      L2Point——位于第二条直线上的一个2D点
//      L2Slope——第二条直线的斜率
//输出:保存直线交点的浮点数组
float * lineIntersect ( float * L1Point, float L1Slope,
                        float * L2Point, float L2Slope )
{
    //临时数组,用于保存结果
    float * temp = new float[2];
    //求解 x 的值
    temp[0] = ( L1Slope * L1Point[0] - L2Slope * L2Point[0] +
                L2Point[1] - L1Point[1] ) / ( L1Slope - L2Slope );
    //利用刚求出的 x 计算 y 的值
    temp[1] = L1Slope * ( temp[0] - L1Point[0] ) + L1Point[1];

    return temp;
}
```

自我测试

计算出下列方程的斜率和 y 轴上的截距, 对线性方程组请给出解的数目:

1. $x + y = 7$
2. $x - 3y = -6$
3. $x + 4y = 8$
 $x - 3y = -2$
 $-x + 3y = -6$
 $x/4 + y = 2$
4. $x - y = 5$
5. $x + 3y = -1$
6. $3x - 5y = -8$
 $5x + y = 0$
 $2x + 6y = -5$
 $3x + 5y = 8$

利用联合法求解下列线性方程组:

7. $6x - 5y = 7$
8. $1/2x + 2y = 3$
 $x - 7y = -1$

- A. $-3x + 4y = 25$ B. $3x + 4y = 10$ C. $4x + y = 23$ D. $y = -4/3x + 10$
11. 下列哪条直线与直线 $y = 1/2x + 10$ 平行, 而且经过点 $(12, 5)$?
- A. $y = 3/4x + 10$ B. $y = 1/2x - 1$ C. $y = -2x + 10$ D. $y = 2x - 10$
12. 下列哪组定义的直线包含了点 $(20, 100, -50)$ 和点 $(100, 60, -20)$?
- A. 点 $\langle 20, 100, -50 \rangle$, 并且沿着方向 $\langle 80, 40, -70 \rangle$ 变换
B. 点 $\langle 100, 60, -20 \rangle$, 并且沿着方向 $\langle 80, 40, -70 \rangle$ 变换
C. 点 $\langle 20, 100, -50 \rangle$, 并且沿着方向 $\langle -80, 160, -70 \rangle$ 变换
D. 点 $\langle 100, 60, -20 \rangle$, 并且沿着方向 $\langle -80, 40, -30 \rangle$ 变换
13. 下列两个方程有多少个共同点?
- $3x + 4y = 10$
 $4x + y = -20$
- A. 0 B. 1 C. 2 D. 无穷多个
14. 下列两个方程有多少个共同点?
- $y = 2x - 10$
 $4x - 2y = 15$
- A. 0 B. 1 C. 2 D. 无穷多个
15. 下面两条直线的交点位置是多少?
- $y = 2x + 10$
 $y = 1/4x + 45$
- A. $(20, 50)$ B. $(-20, 50)$ C. $(150, 300)$ D. $(-150, 300)$

练习

练习 1: 确定并画出线性方程

1. 指出下列哪个方程是线性的(也就是说其全部解是一条直线):

- (1) $2x - y = 5$
(2) $-x + 5y = 0$
(3) $x = -1$
(4) $y + x^2 = 5$

2. 画出下列线性方程:

- (1) $x - 2y = 0$
(2) $-3x + y = 4$
(3) $x = 1$

练习 2: 计算斜率

1. 计算下列点确定的直线斜率:

- (1) $(0, 8)$ 和 $(3, 0)$
(2) $(4, 6)$ 和 $(1, 7)$

(3) $(3, -1)$ 和 $(4, 2)$

(4) $(-3, -6)$ 和 $(1, 5)$

2. 计算下列直线的斜率:

(1) $2x + 5y = 10$

(2) $x - 4y = 0$

(3) $2y = 10$

(4) $x + y = -9$

练习 3: 计算直线方程

计算下列点确定的直线方程:

(1) $(0, 10)$ 和 $(5, 0)$

(2) $(3, 5)$ 和 $(1, 9)$

(3) $(2, -1)$ 和 $(6, 1)$

(4) $(-2, -5)$ 和 $(1, 4)$

(5) $(2, 0, -1)$ 和 $(3, 4, 5)$

(6) $(-3, 1, 5)$ 和 $(0, 8, -2)$

练习 4: 利用联合法和代入法求解线性方程组

1. 利用联合法求解下列线性方程组:

(1) $6x - 5y = 9$

$1/2x + 2y = 6$

(2) $x - 7y = -6$

$4x - 5y = 28$

2. 利用代入法求解下列线性方程组:

(1) $3x - 2y = -8$

$4x + 3y = 12$

(2) $x + 5y = 17$

$2x + y = 8$

练习 5: 相交检测

假如一个物体的当前位置为 $(300, 100)$, 它沿着斜率为 $1/2$ 的直线向左下方运动。一堵墙被放置到直线 $y = -4x + 400$ 处, 问该物体是否会撞上墙? 如果发生相撞, 那么相撞点在哪里?

1. 画一个笛卡儿坐标系。

2. 画出墙的位置。

3. 计算物体此时运动路径的直线方程。

4. 画出物体路径方程的图形。

5. 计算这两条直线的斜率, 并确定它们是否会发生碰撞。

6. 如果发生碰撞, 利用联合法或代入法计算出相撞点的具体坐标。

程序设计

程序设计 1：请求物体的当前位置

编写一个程序，当请求物体此时的位置和移动方向后，返回由此确定的直线路径的方程。

程序设计 2：请求物体的当前位置和斜率

编写一个程序，当请求物体此时的位置和直线斜率后，在屏幕上绘制出当前位置，然后画出下面它将要移动的直线路径。

程序设计 3：请求两个直线方程，确定它们是否相交

编写一个程序，请求输入两个线性方程，然后确定它们是否会相交。如果它们确实相交，返回它们的交点位置；否则，返回一个提示信息。

问题讨论

1. 在传统的笛卡儿坐标系和屏幕坐标系之间进行转换，采用哪种方法会更好一些？
2. 对于 3D 坐标系，没有一个统一的行业标准会带来哪些问题？
3. 在程序中除了当描述物体沿直线运动时采用直线方程之外，哪里还会用到线性方程？



第2章 相关几何知识

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 计算两点之间距离
- 计算中间点
- 写出合适的抛物线轨迹方程
- 写出2D空间中圆的轨迹方程
- 写出3D空间中球的轨迹方程
- 判断两个圆边界是否相交
- 判断两个球边界是否相交

介绍

本章将介绍在本书中所用到的几何知识，在以后的游戏编程中，这些几何知识都将是非常有用的。比如，每当你用向量来控制物体运动或用直角三角形模拟游戏场景时，都会用到勾股定理。而两点间的距离公式在检测物体碰撞和研究物体运动方面也是非常有用的。抛物线可以表示2D和3D场景中自由物体的运动轨迹。圆和球体可以用于模拟环状物体和它们的运动轨迹，在物体碰撞检测中，它们也都有很重要的应用。此外，在随书附带的光盘中向大家展示了一个如何利用圆来进行碰撞检测的例子。

2.1 两点间的距离

在编程中，我们经常需要计算屏幕上两点之间的距离。它可以用于判断两物体是否碰撞，或者两智能体是否相遇的问题。另外，在人工智能程序中，智能体可以通过判断敌人与自己的距离来决定发起攻击的时机。无论在什么样的情况下，能够快速计算出两点之间距离都是非常重要的。而完成这种计算最简单的方法就是利用勾股定理。

勾股定理

$$a^2 + b^2 = c^2$$

a 、 b 为直角三角形的两条直角边， c 为斜边。

注意

勾股定理只适用于直角三角形。图2-1中的小方块表明 a 、 b 之间的夹角为 90° ，即说明这是一个直角三角形。

勾股定理的逆定理也是正确的。

勾股定理的逆定理

如果 a 、 b 、 c 分别为一个三角形的三边，并且满足 $a^2 + b^2 = c^2$ ，那么这个三角形为直角三角形，斜边为 c 。

在第4章中我们将把勾股定理扩展到向量操作，这样我们就可以计算屏幕上两点之间的距离。如图2-2所示，假设两点 $P_1(x_1, y_1)$ ， $P_2(x_2, y_2)$ 。

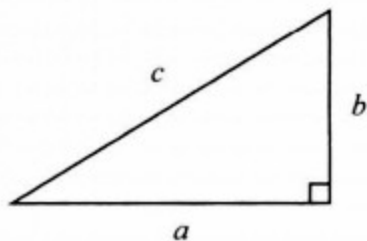


图 2-1 $a^2 + b^2 = c^2$

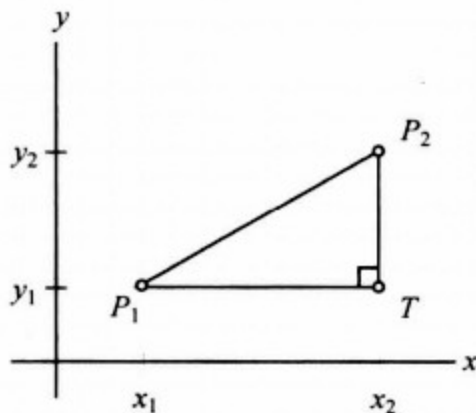


图 2-2 P_1, P_2 两点之间距离

连接 P_1 、 P_2 以 P_1P_2 为斜边作一个直角三角形，这个三角形的第三个节点设为 $T(x_2, y_1)$ 。正如我们所看到的那样， P_1T 的长度为 $(x_2 - x_1)$ ， P_2T 的长度为 $(y_2 - y_1)$ 。现在利用勾股定理计算：

$$(P_1P_2)^2 = (P_1T)^2 + (P_2T)^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

我们可以将这个公式总结为一个计算两点之间距离的简单公式，距离公式。

2D 场景中的距离公式

$$P_1P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

其中 $P_1(x_1, y_1)$ ， $P_2(x_2, y_2)$ 分别为线上的点。

在下面函数中用两个双精度的数组代表 2D 空间中的点，并返回它们的距离值。注意，在头文件 `<cmath>` 中的函数 `sqrt()` 返回输入参数的平方根。另外，函数 `pow()` 带有两个参数，并返回以第一个参数为底，第二个参数为幂的乘方的值。虽然函数 `pow()` 比乘法运算稍慢一些，但它还是很有用的。例如，虽然 `pow(x, 2)` 比直接计算 $x * x$ 运算的要慢一些，但如果使用微软编译器中 `#pragma intrinsic` 预编译指令，那么程序的速度将会大大提高。

```
#pragma intrinsic(sqrt, pow)
```

这一命令不再经过堆栈，而直接将数学运算送入运算器中。（更多关于 `intrinsic` 命令的信息，可以参阅 MSDN。）注意，为了避免编译器在双精度数据转化为单精度时因数据丢失而发出警告，这一指令在返回值前首先将结果转换成单精度的数值。同样，由于 `sqrtrf()` 和 `powf()` 的输入输出都是单精度的，也可以使用这两个函数来避免编译器警告。

```
//计算两点之间的距离
```

```
//输出:P1 - 表示点1的实型数组
```

```
      P2 - 表示点2的实型数组
```

```
//输出:两点之间距离
```

```
float distance2D(float *P1, float *P2)
```



```
{
    //计算并返回所求距离
    return (float)sqrt(pow(P2[0] - P1[0],2) + pow(P2[1] - P1[1],2));
}
```

【例2-1】 屏幕上两点之间的距离

如果两物体的中心点分别在(25, 80)和(55, 40), 那么这两个点的距离是多少呢?

解答:

$$\begin{aligned}
 P_1P_2 &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \\
 &= \sqrt{(55 - 25)^2 + (40 - 80)^2} \\
 &= \sqrt{(30)^2 + (-40)^2} \\
 &= \sqrt{900 + 1600} \\
 &= \sqrt{2500} \\
 &= 50
 \end{aligned}$$

下面, 我们在另一个例子中将距离公式和勾股定理结合在一起应用。

【例2-2】 判断直角三角形

一个三角形的三个顶点分别为 $A(20, 50)$, $B(100, 90)$ 和 $C(70, 150)$, 判断此三角形是否是直角三角形。

解答:

如图2-3所示, 仅仅通过图示无法判断这个三角形的形状, 我们需要进行数学证明。

根据勾股定理的逆定理, 假如三个边的长度满足 $a^2 + b^2 = c^2$, 那么它就是一个直角三角形。首先, 运用三次距离公式分别计算出每条边的距离。

对于边 AB :

$$\begin{aligned}
 AB &= \sqrt{(100 - 20)^2 + (90 - 50)^2} \\
 &= \sqrt{80^2 + 40^2} \\
 &= \sqrt{6400 + 1600} \\
 &= \sqrt{8000}
 \end{aligned}$$

对于边 BC :

$$\begin{aligned}
 BC &= \sqrt{(70 - 100)^2 + (150 - 90)^2} \\
 &= \sqrt{(-30)^2 + (60)^2} \\
 &= \sqrt{900 + 3600} \\
 &= \sqrt{4500}
 \end{aligned}$$

对于边 CA :

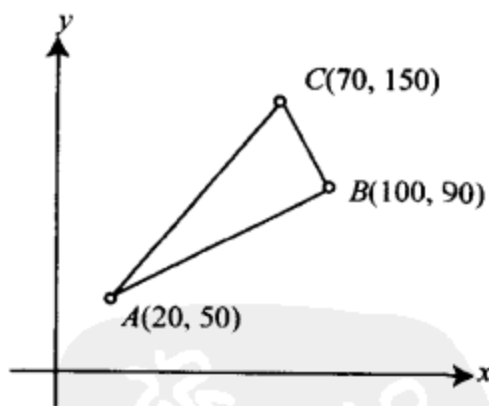


图2-3 三角形ABC

$$\begin{aligned}
 CA &= \sqrt{(70 - 20)^2 + (150 - 50)^2} \\
 &= \sqrt{50^2 + 100^2} \\
 &= \sqrt{2500 + 10000} \\
 &= \sqrt{12500}
 \end{aligned}$$

现在，将三个边的长度代入到勾股定理中，看其是否满足勾股定理：

$$\begin{aligned}
 a^2 + b^2 &= c^2 \\
 (\sqrt{8000})^2 + (\sqrt{4500})^2 &= (\sqrt{12500})^2 \\
 8000 + 4500 &= 12500 \\
 12500 &= 12500 \checkmark
 \end{aligned}$$

所以，它是直角三角形。

距离公式还可以扩展到三维空间中，只需增加关于 z 轴的计算即可。

3D 空间中的距离公式

$$P_1P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

$P_1(x_1, y_1, z_1)$, $P_2(x_2, y_2, z_2)$ 分别是线上的点。

【例 2-3】 3D 空间中两点之间的距离

如果两物体中心点分别为(25, 80, 30)和(55, 40, 100)，那么它们之间的距离为多少呢？

解答：

$$\begin{aligned}
 P_1P_2 &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \\
 &= \sqrt{(55 - 25)^2 + (40 - 80)^2 + (100 - 30)^2} \\
 &= \sqrt{(30)^2 + (-40)^2 + (70)^2} \\
 &= \sqrt{900 + 1600 + 4900} \\
 &= \sqrt{7400} \\
 &\approx 86
 \end{aligned}$$

与距离公式相关的另一个公式就是中点公式，有时，我们可能需要知道屏幕上两物体之间中点坐标的确切位置，此时我们就需要用到下面的公式。

2D 空间中的中点公式

$$M\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

为点 $P_1(x_1, y_1)$ 和 $P_2(x_2, y_2)$ 的中点。

下面这个函数带有两个参数 P_1 、 P_2 ，并返回它们的中点。在这个函数中，用关键字 `new` 为变量 `temp` 分配存储空间，当不再需要这些内存时，可以使用关键字 `delete` 清除内存。由于我们在分配内存时使用了括号，所以在清理时同样需要加上括号：

```

delete[] temp;
//计算中点坐标
//输入:P1 - 表示点1 的实型数组
        P2 - 表示点2 的实型数组
//输出:两点的中点坐标
float * find2DMidPoint(float * P1,float * P2)
{
    //为指针分配足够的空间
    float * temp = new float [2];
    //计算中点坐标
    temp[0] = (P1[0] + P2[0])/2.0f;
    temp[1] = (P1[1] + P2[1])/2.0f;
    //返回结果
    return temp;
}

```

【例2-4】 屏幕上两点的中点坐标

如果两物体的中心分别在(25, 80)和(55, 40), 那么它们的中点坐标是多少呢?

解答:

中点坐标实际上就是原来两点的 x 的平均值和 y 的平均值:

$$M\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

$$M\left(\frac{25 + 55}{2}, \frac{80 + 40}{2}\right)$$

$$M\left(\frac{80}{2}, \frac{120}{2}\right)$$

$$M(40, 60)$$

注意

在计算时要注意加减号。距离公式中是对横坐标和纵坐标作减法运算, 但在中点公式中是作加法运算。

像距离公式一样, 3D 空间的中点公式也可以通过增加 z 坐标轴的计算来完成。

3D 空间中的中点公式

$$M\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}, \frac{z_1 + z_2}{2}\right)$$

为点 $P_1(x_1, y_1, z_1)$ 和 $P_2(x_2, y_2, z_2)$ 的中点。

3D 空间的中点公式与 2D 空间的中点公式差别不大, 仅仅就是增加了一个变量。同时也要注意, 在程序结束前用 delete 清理内存:

```

//计算3 维空间的中点坐标
//输入:P1 - 表示点1 的实型3 维数组
        P2 - 表示点2 的实型3 维数组
//输出:两点的中点坐标

```

```

float *find3DMidPoint(float *P1,float *P2)
{
    //为指针分配足够的空间
    float *temp=new float[3];
    //计算中点坐标
    temp[0]=(P1[0]+P2[0])/2.0f;
    temp[1]=(P1[1]+P2[1])/2.0f;
    temp[2]=(P1[2]+P2[2])/2.0f;
    //返回结果
    return temp;
}

```

【例 2-5】 3D 空间的中点坐标

如果两个物体的中心分别在(25, 80, 30)和(55, 40, 100), 那么它们的中点坐标为多少呢?

解答:

利用扩展的中点公式计算:

$$\begin{aligned}
 &M\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}, \frac{z_1 + z_2}{2}\right) \\
 &M\left(\frac{25 + 55}{2}, \frac{80 + 40}{2}, \frac{30 + 100}{2}\right) \\
 &M\left(\frac{80}{2}, \frac{120}{2}, \frac{130}{2}\right) \\
 &M(40, 60, 65)
 \end{aligned}$$

学习到这里, 你可以做到以下几点:

- 已知直角三角形的两边, 求出第三边的长。
- 运用勾股定理验证一个三角形是否为直角三角形。
- 在 2D 和 3D 空间中计算两点之间距离。
- 在 2D 和 3D 空间中计算两点的中点坐标。

我们将在以后的几章中用到这些知识, 所以请牢记它们。

自我测试

1. 如图 2-4 所示, 计算边 b 的长。
2. 如果一个三角形的三个顶点坐标分别为(30, 75)、(25, 0)和(-50, 45), 那么这个三角形是直角三角形吗?
3. 计算点(30, 80)与点(150, 130)之间的距离。
4. 计算点(20, 50, 10)与点(100, 120, 40)之间的距离。
5. 计算点(30, 80)、(150, 130)的中点坐标。
6. 计算点(20, 50, 10)、(100, 120, 40)的中点坐标。

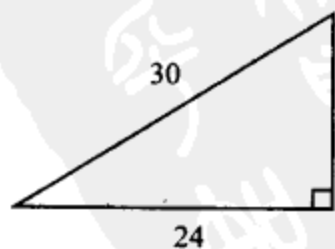


图 2-4 计算边 b 的长

2.2 抛物线

回忆一下你最近一次打棒球或者踢足球时的情景。球在空中是沿直线运动的吗？不，它们更像是沿曲线或弧线运动的。事实上，任何抛出物体都是沿着弧线轨迹运动的。抛物线仅仅就是这种弧线的一个形象的名称。如图2-5所示，它可以是上下方向的，也可以是左右方向的，但抛物线总是轴对称的。

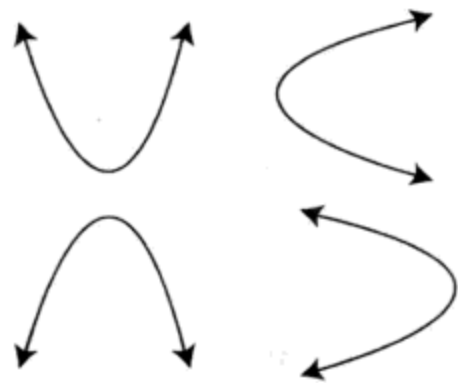


图2-5 四种不同类型的抛物线

有两个因素决定了抛物线的形状，第一就是**顶点**，它是抛物线的顶端。第二个就是**对称轴**，它经过顶点，并且将抛物线从中间分开，所以，抛物线的一边是另一边的映像。抛物线方程有两种形式，一种形式表达对称轴竖直且开口朝上或朝下的抛物线，另一种形式表达对称轴水平且开口朝向侧面的抛物线。

对称轴竖直的抛物线方程

$y = a(x - h)^2 + k$ ，顶点为 (h, k) ，对称轴为 $x = h$ 。

对称轴水平的抛物线方程

$x = a(y - k)^2 + h$ ，顶点为 (h, k) ，对称轴为 $y = k$ 。

注意

在多边形定义中也用到顶点这一概念，但不要把那个顶点与这里的曲线顶点混淆。

另外，抛物线的顶点坐标的形式同样是 (h, k) ，但要注意在两个坐标方程变换中 h 和 k 的位置是不同的。

最后，无论坐标轴是水平还是竖直的，它都经过顶点。

在画抛物线草图时，要考虑抛物线两个方面的特征。第一，假如抛物线方程形式为 $y = a(x - h)^2 + k$ ，则它的开口是朝上或朝下的，假如抛物线方程形式为 $x = a(y - k)^2 + h$ ，则它的开口是朝向侧面的。注意如果是竖直形式的抛物线，可以通过代入 x 的值求 y 值。如果是水平方向的抛物线，可以通过代入 y 的值求 x 值。第二，常数 a 代表了抛物线的开口方向和开口大小。如果 a 是正数，对于形如 $y = a(x - h)^2 + k$ 的抛物线来说开口朝上，对于形如 $x = a(y - k)^2 + h$ 的抛物线来说开口朝右。如果 a 是负数，相应的抛物线开口朝下或朝左。同样， a 的绝对值越大，抛物线开口越小。这些技巧可以帮助我们估计抛物线的走向，并根据我们的需要调整抛物线的形状。

【例2-6】画抛物线草图

粗略地画出抛物线 $y = \left(-\frac{1}{3}\right)(x - 2)^2 + 3$ 的草图。

解答：

1. 首先确定抛物线是竖直型还是水平型的。由于这一方程是对 x 而不是对 y 进行平方

运算, 所以此抛物线一定是竖直的。

2. 在知道了它的形式后, 我们就知道从哪里找 h 和 k , 在这里, 顶点坐标为 $(2, 3)$ 。

3. 最后, 根据 a 的值确定它的开口方向, 对于这个抛物线来说, $a = -1/3$, 所以它一定是开口朝下, 且开口相对较宽。

4. 抛物线的草图如图 2-6 所示, 为了说明起见, 标注出对称轴为 $x = 2$, 但其并不是抛物线的一部分。

【例 2-7】画另一个抛物线草图

粗略地画出抛物线 $x = y^2 + 5$ 的草图。

解答:

1. 首先确定抛物线是竖直型还是水平型的。由于是对 y 而不是对 x 进行平方运算, 所以抛物线一定是水平的。

2. 在知道了它的形式后, 我们就知道从哪里找 h 和 k , 在这里, 顶点坐标为 $(5, 0)$, 注意, 虽然只看到一个 y , 但它其实就是 $(y - 0)$, 所以 k 一定是 0。

3. 最后, 根据 a 的值确定它的开口方向, 假如在 a 的位置没有数字, 那么 $a = 1$, 对于这个抛物线来说 $a = 1$, 所以它一定是开口朝右, 且开口相对较宽, 但没有上一个抛物线的开口大。

4. 抛物线的草图如图 2-7 所示, 对称轴为 x 轴 ($y = 0$)。

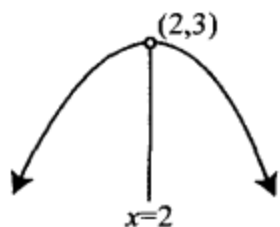


图 2-6 抛物线 $y = \left(-\frac{1}{3}\right)(x-2)^2 + 3$ 的草图

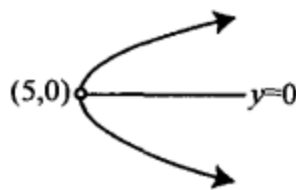


图 2-7 抛物线 $x = y^2 + 5$ 的草图

现在我们已经熟悉了抛物线的轨迹方程, 可以利用它做一个弹道轨迹模型。比起精确的作图, 更重要的是知道如何调整方程以满足自己的需要。如果想看精确的图形, 可以从网络上搜索到很多有关精确作图的共享程序。更重要的是知道如何调整函数方程以满足自己的需要。考虑一下如何在方程中加入一个值以模拟在棒球比赛中激起的高飞球的轨迹。在第 10 章中, 将会对如何利用抛物线方程来建立物体运动模型进行进一步讨论。当你已知物体的起始点和它的初速度, 抛物线方程可以帮你求得它运动的最高点。

自我测试

写出下列抛物线的顶点:

1. $y = 10(x-4)^2 + 7$
2. $x = -2(y-5)^2 + 1$
3. $y = (x+3)^2 + 2$
4. $x = -3y^2 - 12$

$$5. y = (x + 1)^2$$

画出下列方程的图：

$$6. y = 10(x - 1)^2 + 3$$

$$7. x = y^2 - 2$$

2.3 圆和球

熟悉圆的方程是很重要的。你有可能希望在屏幕上画一个圆，这就意味着你需要先写出圆的方程。你还有可能需要描绘出一个物体的圆形运动轨迹，或者利用圆边界来进行碰撞检测，在这种情况下，无论怎样，你都需要能够熟练掌握圆的方程。在平面（比如电脑屏幕）上，圆即是所有到定点距离等于定长的点的集合，这个定长称为半径，定点称为圆心。所以，只需要半径和圆心两个元素即可确定圆方程。那么什么公式可以计算出圆心到圆上一点的距离呢？答案就是勾股定理。注意勾股定理与圆方程之间的相似之处。

在程序代码中，我们可以通过相同的数学方法来定义圆和球：存储圆心和半径。下面两个例子即是在程序中圆和球的结构体定义。

```
struct circle
{
    float center[2];
    float radius;
};

struct sphere
{
    float center[3];
    float radius;
};
```

圆的方程

$$(x - h)^2 + (y - k)^2 = r^2$$

其中圆心坐标为 (h, k) ，半径为 r 。

如图 2-8 所示的圆。

其圆心为 $(1, 2)$ ，半径为 2。根据本章最初所讲的距离公式，我们可以发现，对于圆上每一点 (x, y) ，都有

$$(x - 1)^2 + (y - 2)^2 = 2^2$$

或

$$(x - 1)^2 + (y - 2)^2 = 4$$

你可能已经意识到当圆心在原点时，我们可以更容易地理解勾股定理。下面我们来看一下，当圆心为 $(0, 0)$ 时，圆方程将发生什么样的变化。

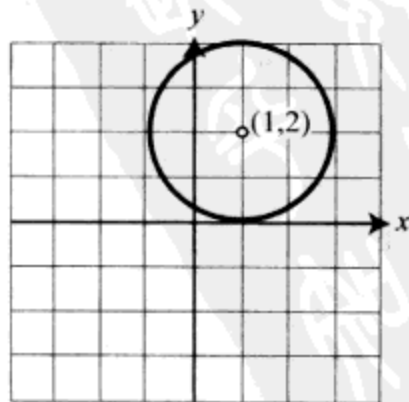


图 2-8 圆 $(x - 1)^2 + (y - 2)^2 = 4$

圆点在坐标系原点的圆方程

$$x^2 + y^2 = r^2$$

其中圆心为(0, 0), 半径为 r 。

【例 2-8】 画出圆的草图

粗略地画出圆 $x^2 + (y+1)^2 = 9$ 的草图。

解答:

要画出圆的草图, 需要知道两件事情: 圆心和半径。

1. 首先为了找到圆心, 需要把方程转化成标准形式, 即 $(x-h)^2 + (y-k)^2 = r^2$, 点 (h, k) 为圆心, r 为半径。在这个方程 $x^2 + (y+1)^2 = 9$ 中, $h=0$, $k=-1$ 。所以, 它的圆心为 $(0, -1)$ 。

2. 其次, 需要知道半径, 这一步很简单; 直接观察等式右边的数字, 即为 r^2 。所以直接开方即可。在这里 $r^2 = 9$, 所以 $r=3$ 。

3. 现在知道了圆心为 $(0, -1)$, 半径为 3, 就可以画出圆的草图, 如图 2-9 所示。

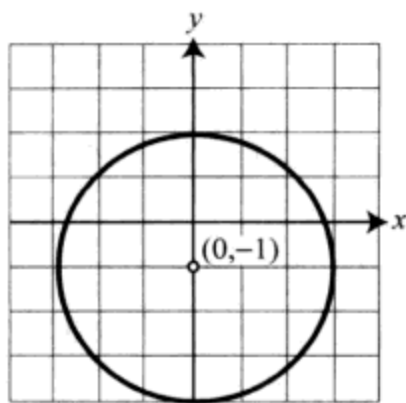


图 2-9 圆 $x^2 + (y+1)^2 = 9$

注意

在求圆心时, 要注意前面的正负号。我们发现在标准形式括号里 h 和 k 前是负号, 这就意味着如果出现了正号, 其实就是减去了一个负值, 即实际所求圆的圆心 h 或 k 为负值。正如例 2-8 中所示。

【例 2-9】 画出另一个圆的草图

画出圆 $x^2 + y^2 = 16$ 的草图。

解答:

1. 首先为了找到圆心, 我们需要把方程转化成标准形式, 即 $(x-h)^2 + (y-k)^2 = r^2$, 点 (h, k) 为圆心, r 为半径。在这个方程 $x^2 + y^2 = 16$ 中, $h=0$, $k=0$ 。所以, 它的圆心为 $(0, 0)$ 。

2. 其次, 需要知道半径, 这一步很简单; 直接观察等式右边的数字, 即为 r^2 。所以直接开方即可。在这里 $r^2 = 16$, 所以 $r=4$ 。

3. 现在知道了圆心为 $(0, 0)$ 半径为 4, 就可以画出圆的草图, 如图 2-10 所示。

现在我们可以基于一个圆方程画出圆的图像了。下面从另一个方面进行学习, 即基于游戏场景为一个圆构建圆方程。

【例 2-10】 写出圆方程

在游戏中当一辆汽车驶上斜坡时, 我们需要利用圆的边界对它进行碰撞检测。假设这辆汽车的中心点为 $(20, 50)$, 车上距离中心点最远的点为 $(60, 80)$, 如图 2-11 所示。找出这个圆边界对应的圆方程。

解答:

为了确定圆方程, 需要先确定圆心和半径。

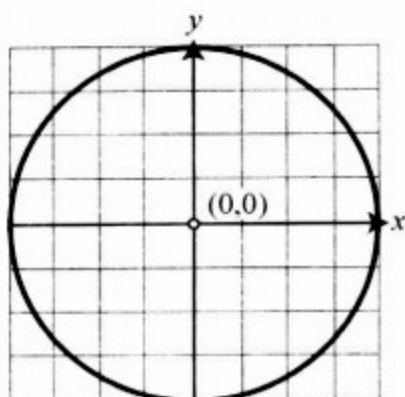
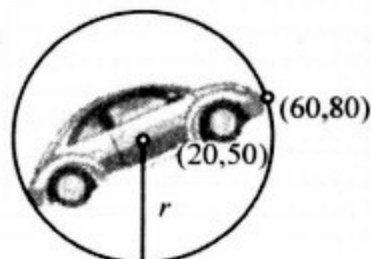
图 2-10 圆 $x^2 + y^2 = 16$ 

图 2-11 一辆汽车的圆边界

1. 已知物体的圆心为(20, 50)，这就意味着 $h = 20$, $k = 50$ 。代入圆的标准方程得：

$$(x - 20)^2 + (y - 50)^2 = r^2$$

2. 下面确定半径，由于半径的定义为圆上任意一点到圆心的距离，所以，我们只需计算出圆上一点(60, 80)到圆心(20, 50)的距离即可：

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$r = \sqrt{(60 - 20)^2 + (80 - 50)^2}$$

$$r = \sqrt{(40)^2 + (30)^2}$$

$$r = \sqrt{1600 + 900}$$

$$r = \sqrt{2500}$$

$$r = 50$$

3. 确定圆半径为 50 后，直接代入圆方程即可得：

$$(x - 20)^2 + (y - 50)^2 = 2500$$

虽然圆方程可以表示在 2D 空间的圆，但我们还需要在三维空间中讨论球。已知球是一个圆绕着圆心旋转所得到的。篮球、网球都是球体的例子。球体可以表示类似球形的物体形状，也可以用来表示复杂物体所在的几何区域，比如汽车、飞船等。游戏开发人员经常在不需计算精确碰撞的情况下，利用球来简化碰撞检测。和圆相似，球体也是由球心和半径定义的。唯一不同的是这里的球心由三个量决定，而不是两个。

球体的方程

$$(x - h)^2 + (y - k)^2 + (z - l)^2 = r^2$$

其中球心是 (h, k, l) ，半径是 r 。

注意：当球心在原点 $(0, 0, 0)$ 的情况。

球心在原点的球体方程

$$x^2 + y^2 + z^2 = r^2$$

其中球心坐标是 $(0, 0, 0)$ ，半径是 r 。

【例 2-11】 球的球心坐标和半径

在游戏中，需要利用球的边界进行碰撞检测。物体的球心为(20, 50, -30)，物体上距离球心最远的点为(60, 80, -20)。写出代表这个球体边界的方程。

解答：

为了确定球体方程，我们需要确定球心和半径。

1. 物体的中心已知为(20, 50, -30)，这意味着 $h=20$, $k=50$, $l=-30$ 。将它带入球的标准方程得

$$(x-20)^2 + (y-50)^2 + (z+30)^2 = r^2$$

2. 其次，需要知道球的半径，由于球的半径定义为球体表面任意一点到球心的距离，所以，我们只需计算出球上一点(20, 50, -30)到球心(60, 80, -20)的距离即可：

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

$$r = \sqrt{(60-20)^2 + (80-50)^2 + (-20+30)^2}$$

$$r = \sqrt{(40)^2 + (30)^2 + (10)^2}$$

$$r = \sqrt{1600 + 900 + 100}$$

$$r = \sqrt{2600}$$

3. 现在已经求得半径为 $\sqrt{2600}$ ，代入球方程得

$$(x-20)^2 + (y-50)^2 + (z+30)^2 = 2600$$

注意

和圆一样，要注意圆心坐标的正负号。

我们已经讨论了圆的方程和球的方程。现在，你可以在游戏中定义圆形物体，例如一个球，可用它来定义物体的圆形运动轨迹，你还可以利用它作为几何边界进行碰撞检测，关于这一点我们将在下一节中进行讨论。

自我测试

写出下列圆的圆心和半径：

1. $(x-30)^2 + (y-10)^2 = 400$

2. $(x+20)^2 + (y-90)^2 = 100$

3. $(x+50)^2 + y^2 = 625$

根据所给条件写出圆方程：

4. center = (40, -25), radius = 30

5. center = (0, 0), radius = 15

6. center = (10, 40), circle goes through the point(20, 50)

计算下列球体的球心和半径：

7. $(x-10)^2 + (y-30)^2 + (z-50)^2 = 1600$

8. $x^2 + y^2 + z^2 = 100$

9. $(x-50)^2 + y^2 + (z+40)^2 = 1$

根据所给条件写出球体方程:

10. center = (40, -25, 30), radius = 10

11. center = (0, 0, 0), radius = 22

12. center = (10, 0, -60), circle goes through the point(10, 50, -30)

2.4 在碰撞检测中的应用

在前面的章节中提到可以在游戏中利用圆或球的边界进行碰撞检测。当然也可以利用其他图形进行检测, 不过由于圆和球都可以方便地进行数字计算, 它们在检测的速度上也优于其他图形。这种检测虽然精确度不高, 但速度非常快, 所以很适合应用于外围检测。下面我们来进一步讨论如何利用这些曲线方程来检测两物体是否相撞。

我们先从2D空间圆开始讨论, 随后再扩展到3D空间中。在游戏中, 经常会遇到像汽车这样的不规则图形, 对其很难进行精确的碰撞检测。经常采用的方法是将会车放在一个圆中, 用圆的边界代替汽车进行碰撞检测。正如前面的例子所示, 将汽车放在一个方程为 $(x-h)^2 + (y-k)^2 = r^2$ 的圆中, 如图2-12所示。

现在可以将两辆汽车内接于两个圆中进行碰撞检测了。两个圆都是由圆心和半径决定的。如图2-13所示。

注意图2-13中的两个圆是相互外切的。当出现这种情况时, 两个圆心之间的距离 D 正好等于两个圆半径之和 $(r_1 + r_2)$ 。这意味着如果两个圆重叠(即相撞), 两圆圆心的距离要小于等于两半径之和 $(r_1 + r_2)$, 如图2-14所示。

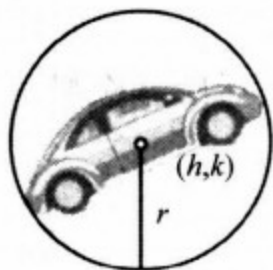


图 2-12 汽车的外接圆

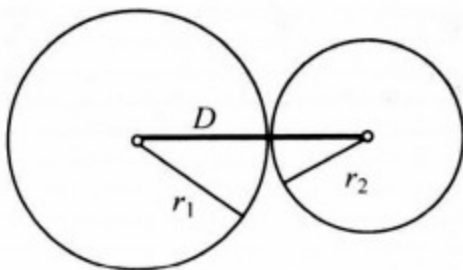


图 2-13 相互外切的两个圆

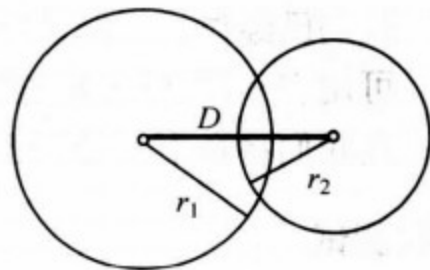


图 2-14 两相交圆

这样, 通过判断两圆心之间距离是否小于等于两半径长度之和, 即可很容易的检测到碰撞。我们可以利用在本章开始时复习的距离公式计算两圆的圆心距离。然后与两半径之和 $(r_1 + r_2)$ 相比较即可。

两圆之间的碰撞检测

设两圆方程分别为 $(x-h_1)^2 + (y-k_1)^2 = r_1^2$ 和 $(x-h_2)^2 + (y-k_2)^2 = r_2^2$, 如果 $\sqrt{(h_2-h_1)^2 + (k_2-k_1)^2} \leq (r_1 + r_2)$, 则两个圆发生碰撞。

注意

用这种方法进行碰撞检测时圆的大小不是相同的, 可以根据物体的大小来调整圆的半径。

在游戏中，物体都是一帧一帧的移动，为了对每一帧图像进行检测，在程序里需要把这一过程放入循环体中，对每一帧图像进行重复检测。由于物体的移动并不是平滑的，物体从一帧跳入另一帧时移动了很小的距离。这就意味着虽然在上一帧时物体之间还有距离，但到达下一帧时有可能就重叠了，即物体可能会跳过恰好接触的过程而直接重叠。这就是为什么要判断圆心距离是否小于等于半径长度之和，而不是仅检测它们是否相等。

【例 2-12】 圆与圆的碰撞检测

假设在一个 2D 空间的赛车游戏中运用圆边界来代替真实的汽车轮廓。在当前帧中，两汽车的圆边界分别为 $(x-50)^2 + (y-20)^2 = 900$ 和 $(x+10)^2 + (y-10)^2 = 400$ ，检测它们是否已经碰撞。

解答：

1. 为了进行碰撞检测，首先需要确定圆心和半径，第一个圆的圆心为 $(50, 20)$ ，半径为 30，第二个圆的圆心为 $(-10, 10)$ ，半径为 20。

2. 计算两圆心之间的距离 D ：

$$\begin{aligned} D &= \sqrt{(h_2 - h_1)^2 + (k_2 - k_1)^2} \\ D &= \sqrt{(-10 - 50)^2 + (10 - 20)^2} \\ D &= \sqrt{(-60)^2 + (-10)^2} \\ D &= \sqrt{3600 + 100} \\ D &= \sqrt{3700} \\ D &\approx 60.83 \end{aligned}$$

3. 最后需要判断 $D \leq (r_1 + r_2)$ 是否成立。在本题中 $(r_1 + r_2) = 30 + 20 = 50$ ， $D > (r_1 + r_2)$ ，所以没有发生碰撞。

在编程之初我们就会发现，开方运算需要花费大量的时间，即占用大量处理器资源。下面介绍一种既快捷又简便的方法来消除开方运算以大大加速碰撞检测的时间。如果不等式两边都有平方，那么开方后不等式的符号不变。在例 2-12 中，如果 $60.83 > 50$ ，那么， $(60.83)^2 > (50)^2$ 依然成立。

两圆碰撞检测方法的优化

给定两圆方程分别为 $(x-h_1)^2 + (y-k_1)^2 = r_1^2$ 和 $(x-h_2)^2 + (y-k_2)^2 = r_2^2$ ，如果 $(h_2 - h_1)^2 + (k_2 - k_1)^2 \leq (r_1 + r_2)^2$ 成立，则发生碰撞。

此时，你应该已经知道如何将这种方法扩展到 3D 空间中了。在 3D 空间中，不再用圆的边界，而是用球的边界进行检测。将物体的中心定义为球的球心，物体上距离中心最远的距离定义为球的半径，然后对每一帧进行检测，观察两球是否重叠。现在我们用前面提到的快速方法进行检测。

球与球的碰撞检测的优化

给定两球方程分别为 $(x-h_1)^2 + (y-k_1)^2 + (z-l_1)^2 = r_1^2$ 和 $(x-h_2)^2 + (y-k_2)^2 + (z-l_2)^2 = r_2^2$ ，如果

$(h_2 - h_1)^2 + (k_2 - k_1)^2 + (l_2 - l_1)^2 \leq (r_1 + r_2)^2$ 成立, 则发生碰撞。

下面的示例函数用于检测两球是否相撞。在这个例子中, 我们可以很简便地使用前面定义的球结构体。注意, 在计算球心距离时, 我们不再比较平方根, 而是直接与半径的平方和比较。如果在每一帧中都需要对大量的球体进行碰撞检测, 那么这个简便而又有效的方法将会大大加速程序的运行。

```
struct sphere
{
    float center[3];
    float radius;
};
//检测两球是否发生碰撞
//输入: S1 - 第一个球
        S2 - 第二个球
//输出: 假如发生碰撞, 返回真, 否则为假
bool ColBetweenSpheres(sphere &S1, sphere &S2)
{
    return (pow(S2.center[0] - S1.center[0], 2) +
            pow(S2.center[1] - S1.center[1], 2) +
            pow(S2.center[2] - S1.center[2], 2) <
            pow(S1.radius + S2.radius, 2));
}
```

注意, 在这个函数中并没有用到 if-else 条件分支结构。这是因为 return 后面的不等式如果成立, 程序会自动输出 true, 如果不成立, 也会自动输出 false。

【例 2-13】球与球的碰撞检测

假设在一个 3D 空间的赛车游戏中运用球边界来代替真实的汽车轮廓。在当前帧中, 两汽车的球边界分别为 $(x - 30)^2 + (y - 20)^2 + (z + 10)^2 = 1600$ 和 $x^2 + (y - 40)^2 + (z + 50)^2 = 2500$, 检测它们是否已经碰撞。

解答:

1. 为了进行碰撞检测, 首先需要确定球心和半径, 第一个圆的圆心为 (30, 20, -10) 半径为 40, 第二个球的球心为 (0, 40, -50), 半径为 50。

2. 计算两球心之间的距离 D :

$$\begin{aligned}
 D &= \sqrt{(h_2 - h_1)^2 + (k_2 - k_1)^2 + (l_2 - l_1)^2} \\
 D &= \sqrt{(0 - 30)^2 + (40 - 20)^2 + (-50 + 10)^2} \\
 D &= \sqrt{(-30)^2 + (20)^2 + (-40)^2} \\
 D &= \sqrt{900 + 400 + 1600} \\
 D &= \sqrt{2900} \\
 D &\approx 53.85
 \end{aligned}$$

3. 最后需要判断 $D \leq (r_1 + r_2)$ 是否成立。在本题中 $(r_1 + r_2) = 40 + 50 = 90$, $D \leq (r_1 + r_2)$, 所以没有发生碰撞。

利用圆边界和球边界是判断两物体是否相撞的简单方法。它们仅仅是相对而言较快的方法，但不是最准确的方法。在图 2-11 中，我们首次用圆边界代替汽车边界。哪些空白区域堆与碰撞检测来说是危险的，因为它们会产生错误的碰撞检测结果。如图 2-15 中所示。

避免碰撞检测错误的一种方法就是使用其他更适合物体轮廓的图形来代替圆。只需要保证我们所选的图形可以用方程来表示，这样就能进行数学运算。另一种方法是建立多层次的圆。首先检测外围的圆边界，看其是否出现重叠现象，如果返回值显示没有重叠，则表明没有发生碰撞。但如果返回值显示出现重叠，这也有可能是一个错误的警告，我们需要对其进行进一步的检测。换句话说，如果在检测大圆边界没有发生重叠，则可以进行下一帧的检测，如果发生了重叠，我们还需要进行进一步的检测。在进一步的检测中，我们需要使用一系列更能精确表示物体形状的小圆。这样做虽然提高了精度，但也消耗了大量处理器的资源。如果两大圆之间没有发生重叠，那么就没有必要对小圆再进行检测。图 2-16 展示了我们有可能用到的一系列更符合汽车形状的小圆。

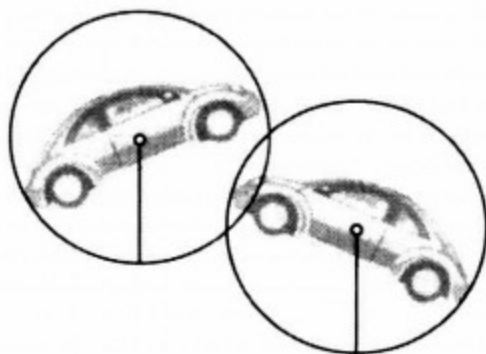


图 2-15 错误的碰撞检测

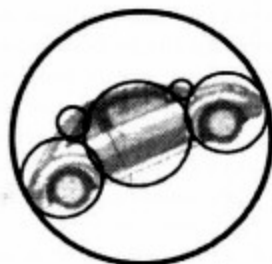


图 2-16 表示汽车中的多层次圆

注意

无论是利用更小的圆进行检测还是在其他更加精确的检测中，对大圆或球的检测都是快速测试的第一步。

在上一章中我们学习了相交直线，本章我们又讨论了有关相交圆和球的问题。碰撞检测在任何游戏中都是很重要的一部分，因为它可以让我们轻松掌握游戏中所有物体是否相遇。当然还会有一些更快更精确的检测方法，但本章讨论的内容给大家提供了相关研究方法的基础。

自我测试

1. 假设在一个 2D 空间的棒球比赛游戏中运用圆边界来代替真实的物体轮廓。在当前帧中，其中一个棒球手的圆边界为 $(x - 70)^2 + (y - 20)^2 = 1600$ ，棒球的圆边界分别为 $(x - 50)^2 + (y - 60)^2 = 256$ ，检测它们有无发生碰撞。
2. 假设在一个 3D 空间的足球比赛游戏中运用球边界来代替真实的物体轮廓。在当前帧中，一个运动员的球边界为 $(x - 50)^2 + y^2 + (z + 20)^2 = 1600$ ，足球边界分别为 $(x - 60)^2 + (y - 70)^2 + (z + 50)^2 = 400$ ，检测它们有无发生碰撞。
3. 在运用球边界进行碰撞检测时我们还需要哪些步骤来避免错误的检测结果？

2.5 碰撞检测的视觉体验

在光盘中，有一个名为 circles 的演示程序。下面是编程人员的一个简短的说明。

本演示程序用来说明两圆以屏幕边沿为边界进行运动并发生碰撞的过程，两个圆在屏幕上的位置和移动方向都是随机的，为简单起见，在两球碰撞过程中没有使用真实的向量反射模型，而是利用一种基本反射算法对其进行模拟。

双击 circles.exe 文件运行该演示程序，你将看到两个球随机移动并会发生碰撞。在该演示画面的背后，演示程序实际上在利用前面提到的圆与圆碰撞检测的方法，一帧一帧地检测两球是否会发生碰撞。

双击点开 circles_code.txt 可以看到程序的源代码，将滚动条拖到相应的位置可以看到有关两圆重叠检测的代码部分。清单 2-1 列出了这部分代码。

清单 2-1 圆与圆的碰撞检测

```

////////////////////////////////////
//checkBallCollision:check for a collision with another ball
//输入:ball2 - 用于检测碰撞的球
//
//返回:如果与其他球发生碰撞则返回真
////////////////////////////////////
//bool Ball::checkBallCollision(Ball &ball2)
{
    //球的半径
    int radius1 = diameter / 2;
    int radius2 = ball2.diameter / 2;

    //第一个球的球心
    POINT center1;
    center1.x = radius1 + bounds.left;
    center1.y = radius1 + bounds.top;

    //第二个球的球心
    POINT center2;
    center2.x = radius2 + ball2.bounds.left;
    center2.y = radius2 + ball2.bounds.top;

    //两球之间的距离
    double distance = sqrt(SQUARE(center2.x - center1.x) + SQUARE(center2.y - center1.y));

    //检测是否发生碰撞
    if (distance <= radius1 + radius2)

```

你可能已经发现在程序中包含两个功能，不仅仅检测两个圆是否相撞，而且还检测球是否撞到了屏幕边缘。这里利用了碰撞盒的方法，因为屏幕其实可以看作是由四条直线组成的

盒子。

在第 14 章介绍完碰撞响应后，我们有可能还要回头来看这个实验。

本章总结

在本章中，我们继续回顾了几何中一些重要内容。这些知识可以帮助我们在以后的章节中对物体运动和碰撞检测进行编码。我们首先介绍了距离公式和中点公式，以及著名的勾股定理。这些公式除了在本章中应用于碰撞检测之外，还广泛应用于向量数学和物理中。随后我们又介绍了抛物线方程，它们可以用来模拟抛物运动轨迹。最后，本章针对圆和球讨论了如何利用它们的边界进行碰撞检测。

能力测试

多项选择

- 在 2D 空间中，点(40, -150)与点(280, -50)之间距离是多少？
A. 260 B. $\sqrt{260}$ C. $\sqrt{142400}$ D. $\sqrt{377}$
- 在 3D 空间中，点(400, 25, -100)与点(200, -50, -150)之间距离是多少？
A. 31875 B. $\sqrt{31875}$ C. 48125 D. $\sqrt{48125}$
- 在 2D 空间中，点(40, -150)、点(280, -50)的中点坐标是什么？
A. (320, -200) B. (-240, -100)
C. (240, 100) D. (160, -100)
- 在 3D 空间中，点(400, 25, -100)、点(200, -50, -150)的中点坐标是什么？
A. (600, -25, -250) B. (200, 75, 50)
C. (300, -12.5, -125) D. (100, 37.5, 25)
- 下列哪个方程最适合表示足球被踢出后的运动轨迹？
A. $y = 3x - 15$ B. $y = 2(x - 10)^2 + 10$
C. $x = y^2 - 20$ D. $y = -5x^2 + 15$
- 下列哪个方程表示圆心在(45, 200)且过点(45, 209)的圆？
A. $(x + 45)^2 + (y + 200)^2 = 9$ B. $(x + 45)^2 + (y + 200)^2 = 81$
C. $(x - 45)^2 + (y - 200)^2 = 9$ D. $(x - 45)^2 + (y - 200)^2 = 81$
- 下列哪个方程表示球心在(200, 0, -30)且半径为 10 的球体？
A. $(x + 200)^2 + (y + 0)^2 + (z - 30)^2 = 10$
B. $(x + 200)^2 + (y + 0)^2 + (z - 30)^2 = 100$
C. $(x - 200)^2 + y^2 + (z + 30)^2 = 10$
D. $(x - 200)^2 + y^2 + (z + 30)^2 = 100$
- 圆 $(x - 45)^2 + (y - 200)^2 = 81$ 与圆 $x^2 + y^2 = 400$ 有重叠吗？
A. Yes B. No

9. 球 $(x-10)^2 + y^2 + (z+25)^2 = 64$ 与球 $x^2 + y^2 + (z-25)^2 = 100$ 有重叠吗?

A. Yes

B. No

练习

练习1：两点间的距离

1. 一个三角形三个顶点分别为 $(40, 85)$, $(15, 0)$ 和 $(-40, 35)$, 判断此三角形是否是直角三角形。
2. 计算点 $(20, 70)$ 与点 $(150, 120)$ 之间的距离。
3. 计算点 $(25, 55, 15)$ 与点 $(105, 125, 45)$ 之间的距离。
4. 计算点 $(40, 90)$ 、点 $(160, 140)$ 的中点坐标。
5. 计算点 $(30, 60, 20)$ 、点 $(110, 130, 50)$ 的中点坐标。

练习2：抛物线

1. 给出下列抛物线的顶点:

a. $y = 5(x-4)^2 + 4$

b. $x = -3(y-6)^2 + 1$

c. $y = (x+4)^2 + 4$

d. $x = -5y^2 - 12$

e. $y = (x+2)^2$

2. 画出下列抛物线的草图:

a. $y = 5(x-1)^2 + 7$

b. $x = y^2 - 4$

练习3：圆和球

1. 给出圆 $(x-30)^2 + (y-10)^2 = 400$ 的圆心和半径。
2. 写出满足圆心在 $(40, -25)$, 半径为 30 的圆方程。
3. 给出球 $x^2 + y^2 + z^2 = 200$ 的球心和半径。
4. 写出满足球心在 $(40, -25, 30)$, 半径为 10 的圆方程。

练习4：在碰撞检测中的应用

1. 假设在一个 2D 空间的棒球比赛游戏中运用圆边界来代替真实的物体轮廓。在当前帧中, 其中一个棒球手的圆边界为 $(x-50)^2 + (y-10)^2 = 1200$, 棒球的圆边界分别为 $(x-50)^2 + (y-60)^2 = 25$, 检测它们有无发生碰撞。
2. 在运用球边界进行碰撞检测时我们还需要哪些步骤来避免错误的检测结果?

程序设计

程序设计1：为 2D 和 3D 空间中的点编写程序

1. 编写一段简单的程序求 2D 空间中点两点坐标并返回两点间的距离以及它们的中点的坐标。

2. 编写一段简单的程序求 3D 空间中点两点坐标并返回两点间的距离以及它们的中点的坐标。

程序设计 2:

编写一段简单的程序求物体的中心和距离中心最远的点并返回它的外接圆方程。

程序设计 3:

编写一段简单的程序分别求两个球的球心和半径，并判断它们是否会相撞。

程序设计 4:

检查本章中所列出的程序，并写出自己的程序以用于对圆边界进行碰撞检测。

问题讨论

1. 在编写视频游戏时需要在哪里运用到距离公式？
2. 勾股定理除了派生出距离公式，还在哪些地方有所应用？
3. 哪些运动还可以利用抛物线来模拟？
4. 使用圆边界或球边界进行碰撞检测有哪些优点？
5. 使用圆边界或球边界进行碰撞检测有哪些缺点？
6. 你可以将圆边界或球边界与其他形式的碰撞检测方法相结合以提高执行的性能吗？



第3章 三角函数

本章目标

认真阅读本章内容并完成练习之后，你能够掌握以下内容：

- 在角度与弧度之间进行转换
- 在直角三角形中计算三角函数值
- 能够绘制正弦和余弦的波形图
- 用正弦或余弦波形图模拟圆周运动
- 掌握三角函数的相关性质
- 利用 C++ 中的数学运算库进行编程

介绍

在这一章中，我们将首先介绍角度的相关知识，它不仅能够运用于旋转，还可用来表示向量的方向；然后介绍了三角函数的相关知识，利用三角函数可以模拟游戏中的波形运动，或者绘制一个直角三角形。当我们在第 4 章中讨论向量时，这些性质将会得到进一步的应用。本章的后一部分重点介绍如何将三角函数运用于方程和 C++ 编程中，本章知识可以说是本书以后章节的基础。

3.1 角度和弧度

在我们开始学习三角函数之前，有必要介绍一下角度及其相关的知识。从表面上看来，角度是用来进行旋转的，其实它还可以用来表示方向，我们将会在第 6 章和第 14 章中进一步对其进行讨论。另外，第 4 章以及与物理知识相关的章节都将会谈到这个问题。在这里，我们首先来研究一下角度的表示方法，方便以后的使用。

每个角都由相交于一点（称作顶点）的两条射线组成，我们把其中的一条射线叫做始边，另一条叫作终边。为了定义标准位置的角，我们决定采用直角坐标系，把角的顶点放在原点 $(0, 0)$ 上，而角的始边总是沿着 x 轴正方向，如图 3-1 所示。

从 x 轴正方向开始，沿逆时针方向进行旋转的角称作正角，而沿顺时针方向旋转的角都是负角，注意该旋转同时也决定了终边的位置。请记住，一个周角是 360° ；如果一个角度正好是 90° 或称之为直角，那么它的终边将会落在 y 轴的正方向上；若

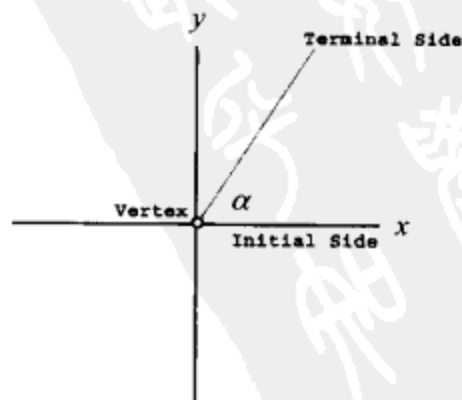


图 3-1 角的标准位置

一个角是 180° ，那么它的终边将落在 x 轴的反方向上。注意图 3-1 中的符号 α ，希腊字母通常被用来表示角，比如 α 、 β 、 θ 等，因此在讨论角的时候，不要忘记这些字母的含义。

【例 3-1】 标准位置上的正角

在标准位置画一个 60° 的角。

解答：

首先，将角的顶点放在原点 $(0, 0)$ 处，将其始边放在 x 轴的正方向上，然后相对 x 轴的正方向逆时针旋转 60° ，并画出终边，如图 3-2 所示。

【例 3-2】 标准位置上的负角

在标准位置画一个 -100° 的角。

解答：

首先，把角的顶点放在原点 $(0, 0)$ 处，并使其始边放在 x 轴正方向上，然后相对 x 轴正方向顺时针旋转 100° ，并画出终边，如图 3-3 所示。

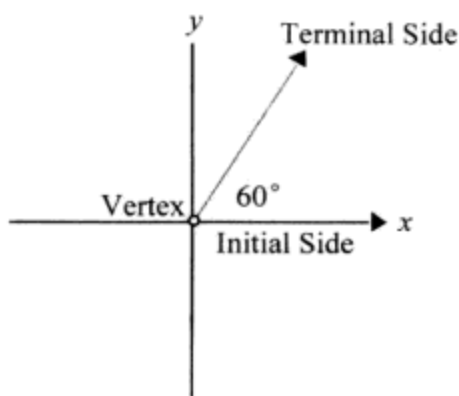


图 3-2 位于标准位置的 60° 角

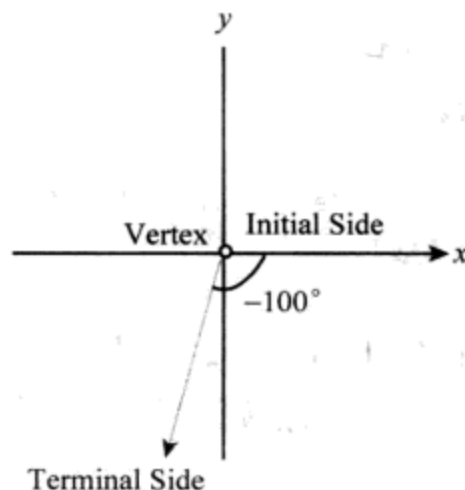


图 3-3 位于标准位置的 -100° 角

到目前为止，我们都是在用角度来衡量一个角，这也是表示角的习惯做法。其实，角还可以用弧度来表示，就像距离可以同时用英尺、米来表示一样，角也可以用角度和弧度来表示，它们只不过是两个不同的单位而已。在一般情况下，我们会利用角度来表示角，因为此形式比较直观；但是在编程的时候，那就不得不用弧度表示了，因此需要能够在角度和弧度之间进行相互转换。

在前面我们曾提到，一个周角是 360° ，它也可以表示成 $2\pi^R$ ，那么半周角就是 180° 或 π^R ，这是在角度和弧度之间进行转换的基础。

角度转换成弧度公式

$$\text{角度} \times \left(\frac{\pi^R}{180^\circ} \right) = \text{弧度}$$

注意

不要把 R 看成指数 R ，它只不过是弧度的单位符号，就和度的单位符号 $(^\circ)$ 一样。建议大家尽量养成一种习惯，即在角的后面标上单位符号 (R 或 $^\circ$)。否则，小组内的其他同伴就不清楚你表示的是角度还是弧度。

【例 3-3】 把角度转换成弧度

将角度 120° 转换成弧度。

解答：

把角度转换成弧度，只要将角度乘上分数 $\frac{\pi^R}{180^\circ}$ 就行了：

$$120^\circ * \left(\frac{\pi^R}{180^\circ} \right) = \left(\frac{120^\circ \pi^R}{180^\circ} \right) = \left(\frac{2\pi^R}{3} \right)$$

注意

用弧度表示角的标准形式，通常把它写成 π^R 倍数的形式，如例 3-3 所示，但是如果要想把它表示成十进制数字形式，只需把 $\pi \approx 3.141592654$ 代入式子就行了。

请注意，对于 C++ 中所有的三角函数，比如 $\sin()$ 、 $\cos()$ 、 $\tan()$ 它们都是以弧度值作为输入的；与它们对应的反三角函数，比如 $\text{asin}()$ 、 $\text{acos}()$ 、 $\text{atan}()$ 等，其返回值同样是弧度的形式。因此在处理角度的时候，我们可以在函数的开头用 `#define` 定义一个宏，完成角度与弧度的转换。

```
#define RadToDeg 57.29577951f
#define DegToRad 0.017453293f
```

弧度乘以宏 `RadToDeg` 可以把该数值转换成角度，而角度乘以宏 `DegToRad` 可以把它转换成弧度。如果想把弧度转换成角度，只需将弧度乘以上面公式中的那个数值的倒数即可。

弧度转换成角度公式

$$\text{弧度} \cdot \left(\frac{180^\circ}{\pi^R} \right) = \text{角度}$$

【例 3-4】 把弧度转换成角度

将 $\left(\frac{6\pi^R}{5} \right)$ 转换成角度。

解答：

把弧度转换成角度，只需利用上面公式将弧度乘上分数 $\left(\frac{180^\circ}{\pi^R} \right)$ 就行了：

$$\left(\frac{6\pi^R}{5} \right) \left(\frac{180^\circ}{\pi^R} \right) = \left(\frac{6\pi^R (180^\circ)}{5\pi^R} \right) = \left(\frac{6(180^\circ)}{5} \right) = 216^\circ$$

注意

例 3-4 中，分子中的 π^R 和分母中的 π^R 相互抵消了。前面给出的两个转换公式很容易混淆，所以一定要记清楚，只要保证将与期望的形式无关的元素消去即可。

现在我们已经确立了表示角的标准方法，一般地，希腊字母通常用来表示一个角，相对坐标系的标准位置可以建立角的方向。此外，我们还可以在角度和弧度之间相互转换。

自我测试

画出下列角的标准位置：

1. 120°
2. 270°
3. -45°

将下列角度转换成弧度：

4. 60°
5. 270°
6. 45°

将下列弧度转化为角度：

7. $\left(\frac{3\pi^R}{4}\right)$
8. $\left(\frac{\pi^R}{3}\right)$
9. $\left(\frac{2\pi^R}{5}\right)$

3.2 三角函数

三角函数在游戏编程中经常出现，比如当你用直角三角形模拟一个网格状的场景时，很容易会想到使用三角函数来完成。同样，我们也可以把它应用于向量（第4章）以及2D和3D的物理知识中。此外我们还将学习正弦和余弦函数的图像，它们能够表示各种波状的移动路径。好了，还是开始学习这些函数吧！

所有的三角函数都是在直角三角形中定义的，它们实际上是直角三角形中有关边的关系，看图3-4中的三角形，它是一个直角三角形，因此可以用它来定义正弦、余弦和正切。

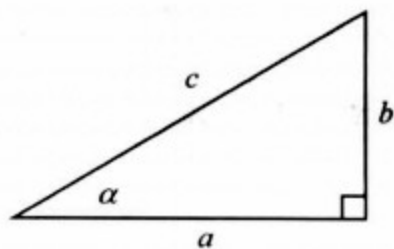


图 3-4 直角三角形

从上图可以发现，角 α 已经标出，这表明下面将要使用角 α 作为研究对象，如果没有把某一个角作为研究对象，那么研究三角形将是毫无意义的。在这里，我们来研究 α 的正弦（sin）、余弦（cos）和正切（tan）。

三角函数

$$\sin\alpha = \frac{opp}{hyp} = \frac{b}{c}$$

$$\cos\alpha = \frac{adj}{hyp} = \frac{a}{c}$$

$$\tan\alpha = \frac{opp}{adj} = \frac{b}{a}$$

其中， opp 是角 α 的对边， adj 是角 α 的邻边， hyp 是角 α 的斜边。

注意

这些三角函数只能运用于直角三角形，不能应用于其他三角形。

正如你看到的，正弦、余弦、正切只涉及直角三角形三边中两条边，结果是它们二者的比率。注意，研究哪个角将决定使用哪些边进行运算，另外当我们研究的不是直角三角形时，当然就无法确定哪条边是斜边。

【例3-5】定义正弦、余弦和正切

如果 α 是一个标准位置的角，(12, 5) 是它终边上的一个点，求出 $\sin\alpha$ 、 $\cos\alpha$ 、 $\tan\alpha$ 。

解答：

1. 为了讨论三角函数，首先需要画一个直角三角形，图3-5即给出了一个直角三角形，其中点(12, 5)在其终边上。

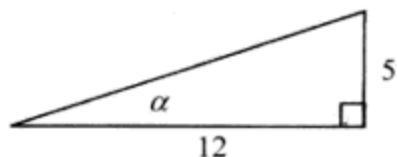


图3-5 例3-5的三角形

2. 由点(12, 5)的坐标，我们已经知道直角三角形其中两条边的长度，然后就可以利用勾股定理计算出第三边的长度。

$$c = \sqrt{12^2 + 5^2} = \sqrt{144 + 25} = \sqrt{169} = 13$$

3. 下面利用三角函数的定义，分别求 α 角的正弦、余弦和正切值。

$$\sin\alpha = \frac{\text{opp}}{\text{hyp}} = \frac{b}{c} = \frac{5}{13}$$

$$\cos\alpha = \frac{\text{adj}}{\text{hyp}} = \frac{a}{c} = \frac{12}{13}$$

$$\tan\alpha = \frac{\text{opp}}{\text{adj}} = \frac{b}{a} = \frac{5}{12}$$

其实，三角函数共有6个，在图3-4中我们已经定义了正弦、余弦和正切，另外3个三角函数分别是余割、正割和余切，它们刚好分别是前面3个函数的倒数。

其他3个三角函数

$$\csc\alpha = \frac{1}{\sin\alpha} = \frac{\text{hyp}}{\text{opp}}$$

$$\sec\alpha = \frac{1}{\cos\alpha} = \frac{\text{hyp}}{\text{adj}}$$

$$\cot\alpha = \frac{1}{\tan\alpha} = \frac{\text{adj}}{\text{opp}}$$

其中，*opp* 是角 α 的对边，*adj* 是角 α 的邻边，*hyp* 是角 α 的斜边。

由于后3个三角函数仅仅是前3个的倒数，所以在通常情况下我们只需要前3个三角函数——正弦、余弦和正切就可以了。但是在化简公式时，如果最后结果中的分母中含有正弦、余弦或正切，此时就应该对程序代码进行优化，即使用后3个三角函数——余割、正割或余切进行代替。

从表面上看这些三角函数不太好理解，其实它们已经被作为函数而预先编写好、并植入计算器里了，你可以像使用乘法和平方一样使用它们。如果你有科学计算器的话，请利用它去熟悉这些三角函数。如果你的计算机是 Windows 操作系统的话，在开始菜单中就有计算器，把它的属性改为科学型，那么三角函数将在左边出现，此时若输入 30，然后按下 $\sin$

键,那么结果将显示 0.5。同样的,你也可以将其转换成弧度计算模式,大多数计算器都有此转换功能,使用你的计算器计算表 3-1 中的常用角度。

表 3-1 常用角度的三角函数值

α (Degrees)	α (Radians)	$\sin\alpha$	$\cos\alpha$	$\tan\alpha$
0	0	0	1	0
30	$\pi/6$	0.5	0.8660	0.5774
45	$\pi/4$	0.7071	0.7071	1
60	$\pi/3$	0.8660	0.5	1.7321
90	$\pi/2$	1	0	—
120	$2\pi/3$	0.8660	-0.5	-1.7321
180	π	0	-1	0
270	$3\pi/2$	-1	0	—
360	0	0	1	0

在这里给大家介绍一个技巧,在真正进入游戏主循环之前,通常我们可以建立一个三角函数查找表,这样在游戏中当需要三角函数值的时候就不用再重新计算了,只需查找对应的表格就行了,该技巧可以大大加快程序的运行速度。下面就是一个创建三角函数查找表的例子。

```
//该数组将保存三角函数值
float sin_table[360];
//填充表格
for (int i=0 ; i < 360; ++i)
{
    //请记住我们在前面定义的宏 DegToRad
    sin_table[i] = sin ( i * DegToRad );
}
```

建立了这个正弦函数表,在程序中就不需要计算角度的正弦值了,即可以通过下面的方式进行查找:

```
//计算任意角度的正弦值
float value = sin_table[ abs ( (int) angle ) % 360 ];
```

首先将角度 angle 强制转换成一个整形值,然后取其绝对值,再对其进行模 360 运算,以此为索引查找正弦表即可。通过取模运算就不用限制角度的大小了,同时也就不会发生数组下标越界的错误。另外,还有一点要注意,如果你在程序中使用预编译命令 `#pragma intrinsic` 指示编译器将三角函数编译成内联函数的话,那么利用查找表而进行的优化也就会变得不再那么明显了。

```
#pragma intrinsic(sin, cos, tan)
```

注意

请查询 MSDN,找到将函数预编译成内联函数的命令选项;另外那些被编译成内联函数的命令不能够在调试(debug)模式下使用,也就是说当你最终要生成程序的发行(release)版本时,再使用这些编译选项。

下面让我们看两个例子，弄明白这些三角函数应该怎样使用。

【例3-6】 利用余弦

假如在游戏中，你需要对空中的目标射击，以 60° 的角度向上方发射，并且箭沿直线运动了 50 个像素单位的长度。如果太阳在其正上方，那么这个箭在地面上的影子走了多远？

解答：

1. 可以使用直角三角形模拟该场景，将该角度放在标准位置，并且让其斜边的长度等于 50，如图 3-6 所示。

2. 由图可以看出，本题实际上是在求直角三角形底边的长度，我们将其标记为 a 。

3. 利用余弦函数建立邻边 a 和斜边的关系，即

$$\cos 60^\circ = \frac{\text{adj}}{\text{hyp}} = \frac{a}{50}$$

$$50(\cos 60^\circ) = a$$

4. 利用 Windows 自带的计算器，输入 60，然后按 $\cos$ 按钮，其显示结果是 0.5，最后再乘以 50 得到结果 25。

这意味着影子沿着地面走了 25 个像素单位的长度。

请注意，当我们已知角度值时，可以利用计算器计算出其余弦函数值，实际上计算器算出的就是该角的邻边与斜边的比率。反过来，我们应该怎样进行逆运算呢？也就是说，如果已知该比率，怎样去得到对应的角度值？这就需要用到反三角函数，这些函数的右上角都有一个 -1 作为上标，字母“arc”放在原函数名的前面。例如，反余弦函数写成 $\cos^{-1}$ ，读作 arccosine，许多计算器用第二个功能或者用 Shift 键在两者之间转换，检查你的计算器并找到它们。在 Windows 自带的计算器中，你将发现它有一个转换功能的复选框。

让我们再次看上面那个射箭的例子，学习和使用反函数。

【例3-7】 使用反正切

假如游戏中的角色玩家对着空中的目标射箭，他与目标的距离是 100 个像素单位，而它们之间的距离在地面上的长度是 400 个像素单位，如果射出的箭将沿直线运动，那么他瞄准的角度应该是多少？

解答：

1. 建立一个直角三角形模拟该场景，在该题中我们已经知道了其中两边的长度，如图 3-7 所示。

2. 本题是在求一个位于标准位置的角，我们将其记为 α 。

3. 正切函数建立了对边与邻边之间的关系，因此可得到：

$$\tan \alpha = \frac{\text{opp}}{\text{adj}} = \frac{400}{100} = 4$$

$$\tan \alpha = 4$$

$$\alpha = \tan^{-1}(4)$$

4. 利用 Windows 自带的计算器，输入对边与邻边的比值 4，选

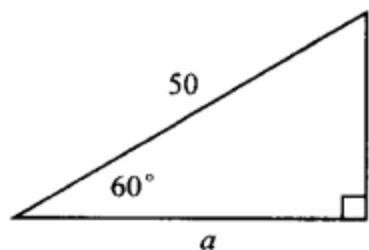


图 3-6 模拟射箭的直角三角形

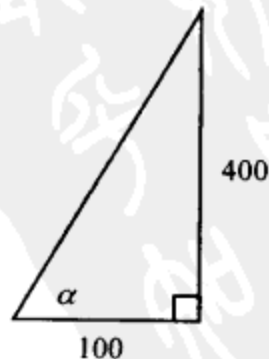


图 3-7 模拟射箭的另一个直角三角形

择上“Inv”复选框，然后点击“tan”按钮，可以得到近似值 76° 。

这就意味着玩家必须在标准位置以 76° 的角度射击目标。

下面让我们看看在编程中应该如何解决此类问题，假如给出两个物体的位置，计算它们之间的角度：

```
//功能:计算 2D 空间内两个物体之间的角度
//输入: p1 - 第一个物体的位置
//      p2 - 第二个物体的位置
//输出: 两个物体之间的角度
float calcAngle2D ( float *p1, float *p2 )
{
    //计算标准角
    float ang = (float)atan( (p2[1] - p1[1]) / (p2[0] - p1[0]) ) * RadToDeg;
    //如果该角度在第一象限
    if ( p2[1] < p1[1] && p2[0] > p1[0] )
        return ang;
    //如果该角度位于第二、三象限
    else if ( (p2[1] < p1[1] && p2[0] < p1[0]) ||
              (p2[1] > p1[1] && p2[0] < p1[0]) )
        return ang + 180;
    //否则该角度一定位于第四象限
    else
        return ang + 360;
}
```

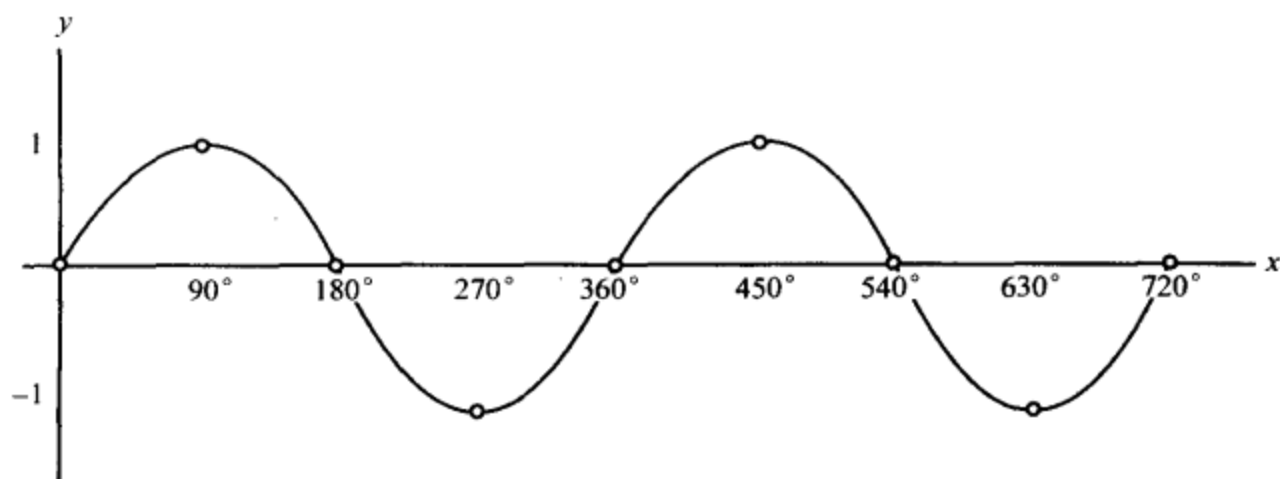
该函数有一点非常重要，由它计算出的实际角度是函数 $\text{atan}()$ 的返回值，还要将其与 RadToDeg 相乘，转换成角度形式。此外要记住，角的正弦值在第一、第二象限是正值，角的余弦值在第一、第四象限是正值，正切值在第一、三象限为正值。对于所有的反三角函数，如果传入的参数是一个正值，那么它们的返回值都为正，即意味该角位于第一象限；如果传入的参数为负数，那么反正弦 $\text{asin}()$ 和反正切 $\text{atan}()$ 的返回角度将位于第四象限，而反余弦 $\text{acos}()$ 的返回角度位于第二象限。因而在实例中，我们必须弄清楚物体的实际位置，确定角度所处的正确象限，然后通过加上 180° 或 360° 计算出它在标准位置的角度值。

注意

当已知角度时，就用正弦、余弦或正切函数得到对应的三角函数值；反之，如果要想得到角度，就应该利用反三角函数计算。

在本节的最后，我们将要画出正弦和余弦函数的图像。以前你可能看到过声波的图像，其实正弦和余弦函数的图像看起来很像声波图像。

首先，让我们来看正弦函数 $y = \sin(x)$ 的图像，然后与余弦函数 $y = \cos(x)$ 的图像进行比较。你可以从表 3-1 中的角度开始，画出离散的一组点，其中用角度值来代表 x 坐标值，与其对应 $\sin\alpha$ 列的值来代表 y 坐标值。如果你用计算器不断计算它们中间角度的正弦值，你将会发现它们都回落在如图 3-8 的曲线上。

图 3-8 正弦函数 $y = \sin(x)$ 的图像

从上图可以发现，它有一个特点就是每隔 360° (或 $2\pi^R$) 就会重复之前的图像，这被称作周期，即上述函数的周期为 $2\pi^R$ 。根据需要，这里有相应的方法来改变函数的周期，比如可以在 x 前面加一个系数。

【例 3-8】水平拉伸正弦曲线

画出方程 $y = \sin(1/2x)$ 的图像。

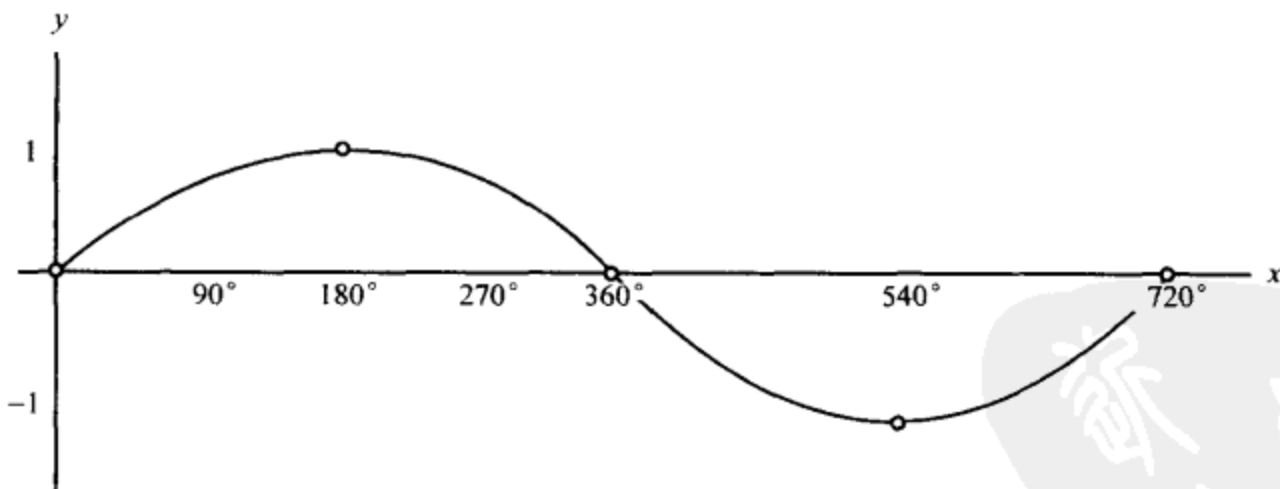
解答：

1. 最简单的方法是代入一系列 x 值，并求出对应的 y 值，表 3-2 列出了一些关键点。

2. 画出这些点之后，尝试画出更多的点，你会得到如图 3-9 所示的曲线。

表 3-2 函数 $y = \sin(1/2x)$ 上的关键点

x	y
0°	0
180°	1
360°	0
540°	-1
720°	0

图 3-9 正弦函数 $y = \sin(1/2x)$ 的图像

【例 3-9】水平压缩正弦曲线

画出函数 $y = \sin(2x)$ 的曲线图。

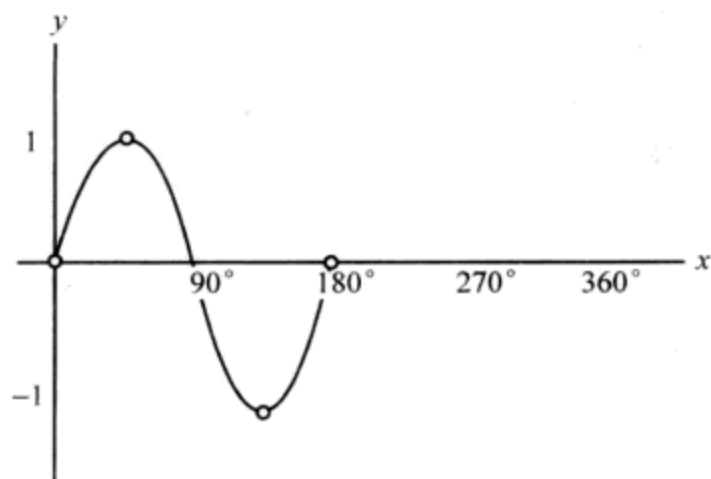
解答：

1. 最简单的方法是代入一系列 x 值，并求出对应的 y 值，表 3-3 列出了一些关键点。

2. 画出这些点之后，尝试画出更多的点，你将会得到如图 3-10 所示的曲线。

表 3-3 函数 $y = \sin(2x)$ 上的关键点

x	y
0°	0
45°	1
90°	0
135°	-1
180°	0

图 3-10 正弦函数 $y = \sin 2x$ 的图像

回头观察最后两个例子，当在 x 前加上系数 $1/2$ 时，函数周期由原来的 360° 变为 720° ；如果在 x 前加上系数 2 时，周期变为 180° ，这是正弦曲线的第一个特点。

正弦函数的周期

对于正弦函数 $y = \sin(Bx)$ ，其周期 $= \frac{360^\circ}{|B|}$ 。

注意

函数周期表示正弦曲线隔多长时间会重复之前的图像，我们在公式中对 B 取了绝对值，这是因为数值前的负号不会影响函数的周期。

如果 B 的位置没有数值，那意味着 $B = 1$ ，那么该函数的周期就等于基本周期 360° 。

同样的，你也可以在正弦函数前增加一个系数，从而实现正弦曲线在竖直方向上缩放或拉伸，此时就改变了曲线的振幅。如果该系数小于 1 ，则缩小正弦图像；如果大于 1 ，则拉伸正弦图像。必须要注意，此性质与周期正好相反。

【例 3-10】垂直拉伸正弦曲线

画出函数 $y = 2\sin x$ 的图像。

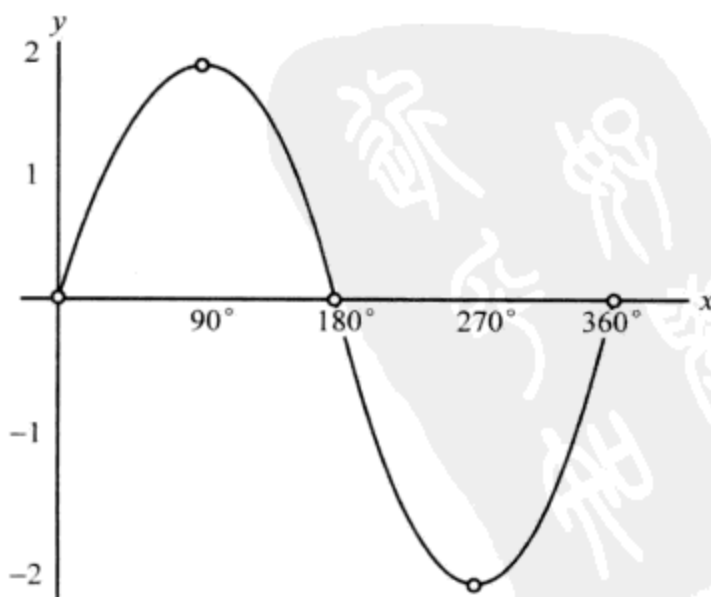
解答：

1. 最简单的方法是代入一系列 x 值，并求出对应的 y 值，表 3-4 列出了一些关键点。

2. 画出这些点之后，尝试画出更多的点，你将会得到如图 3-11 所示的曲线。

表 3-4 函数 $y = 2\sin x$ 上的关键点

x	y
0°	0
90°	2
180°	0
270°	-2
360°	0

图 3-11 正弦函数 $y = 2\sin x$ 的图像

【例 3-11】 垂直压缩正弦曲线

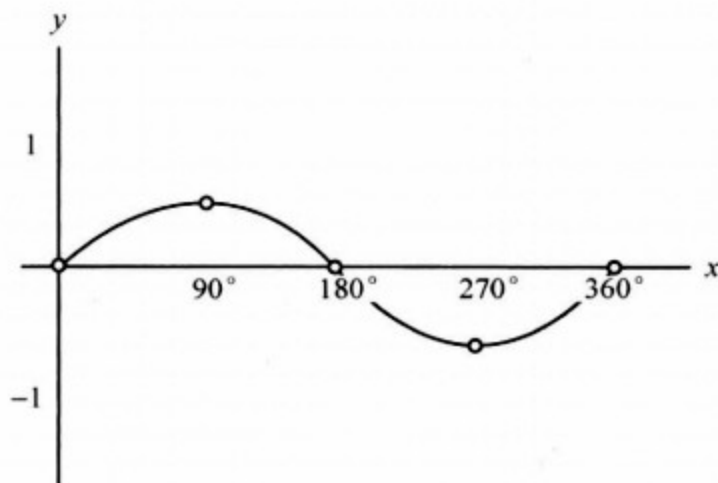
画出函数 $y = 1/2 \sin x$ 的图像。

解答：

1. 最简单的方法是代入一系列 x 值，并求出对应的 y 值，表 3-5 列出了一些关键点。
2. 画出这些点之后，尝试画出更多的点，你将会得到如图 3-12 所示的曲线。

表 3-5 函数 $y = 1/2 \sin x$ 上的关键点

x	y
0°	0
90°	$1/2$
180°	0
270°	$-1/2$
360°	0

图 3-12 正弦函数 $y = 1/2 \sin x$ 的图像

从上面的例子可以看出，基本正弦函数的取值在 1 和 -1 之间来回波动，当在正弦函数前增加系数 2 时，图像的最高点变为 2，而最低点变为 -2；当在正弦函数前加系数 $1/2$ 时，图像在 $1/2$ 和 $-1/2$ 之间波动，这就是正弦函数波形图的第二个特点。

正弦函数波形图的振幅

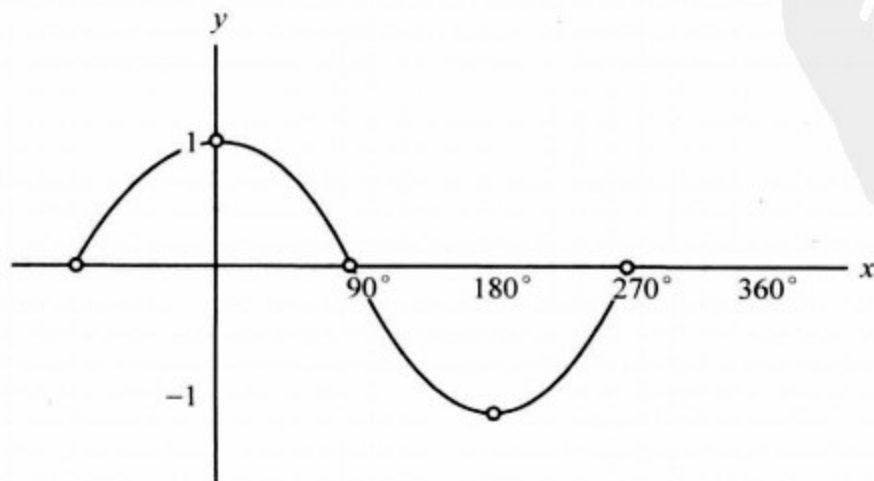
对于正弦函数 $y = A \sin(x)$ ，其振幅 = $|A|$ 。

注意

振幅表示正弦函数上下波动的最大值，计算 A 的绝对值是因为负号不影响振幅。

你可以使用小振幅的正弦函数来模拟船在水中的运动。

对于上面得出的两个关于正弦曲线的特点，我们同样可得到余弦函数也具有类似的特点，唯一不同的是余弦函数曲线是正弦函数曲线向左平移 90° 而得到的图像。图 3-13 显示了

图 3-13 余弦函数 $y = \cos(x)$ 的图像

函数 $y = \cos x$ 的图像，观察其最高点位于 y 轴上，而不在 90° 的位置，对于其他的点同样相对正弦图像向左平移 90° 即可。

在本节中我们定义了 6 种三角函数，这些函数都能够运用到下一章向量的学习中去，另外它们还可用于直角三角形模拟的场景，也可以给出计算物体的波动路径。

自我测试

使用计算器，计算下列三角函数的值：

1. $\sin 45^\circ$
2. $\cos 125^\circ$
3. $\tan -35^\circ$
4. 计算图 3-14 中所示角度 α 的 $\sin \alpha$ 、 $\cos \alpha$ 、 $\tan \alpha$ 。
5. 计算出图 3-14 中的 α 值。
6. 计算 $\cot -35^\circ$
7. 计算 $\csc 20^\circ$
8. 计算 $\sec 45^\circ$

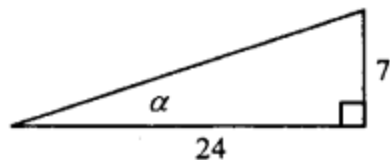


图 3-14 第 4 题对应的直角三角形

求出下列函数的周期和振幅：

9. $y = 5\sin 3x$
10. $y = 3\cos x$
11. $y = \cos 4x$
12. $y = 5\sin x$
13. $y = -2\cos(1/2x)$
14. $y = 1/2\sin(-2x)$

3.3 三角函数的性质

在上一节中，我们定义了 6 种三角函数，使用三角函数的性质(或规律)可以让我们方便地对三角函数方程进行转化，你不可能知道什么时候这些性质将会起作用，因此首先要记住它们。在编写的程序中，每一步操作都会花费一定的时间，因此一定要尽量简化程序中的计算，当然这需要使用三角函数的相关性质。

第一个性质是单位圆，即是以原点为圆心、1 为半径的圆。它的方程是 $x^2 + y^2 = 1$ ，对于单位圆上的任一点，可以利用标准位置的角画一个直角三角形，如图 3-15 所示。

从上图我们可以发现，直角三角形的斜边长等于圆的半径，即 1 个单位长度。现在我们就可以在此三角形中定义正弦和余弦：

$$\sin \alpha = \frac{y}{1} = y \quad \text{和} \quad \cos \alpha = \frac{x}{1} = x$$

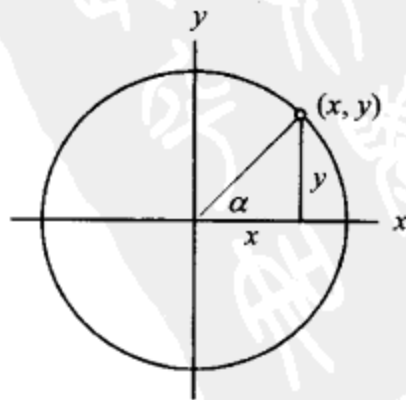


图 3-15 单位圆

因此, $y = \sin\alpha$

$$x = \cos\alpha$$

该结论对于单位圆上的任何一点都适用。

注意

记住 90° 倍数角的正弦或余弦值是非常有用的, 比如 90° 、 180° 、 270° 和 360° 等。 360° 是单位圆与坐标轴相交的位置, 所以我们计算出坐标轴上的正弦、余弦值, 例如角度 90° 在单位圆上的坐标是 $(0, 1)$, 因此, $\sin 90^\circ = 1$ 、 $\cos 90^\circ = 0$ 。

由上已知 $y = \sin\alpha$ 和 $x = \cos\alpha$, 现在将它们带入单位圆的方程, 可得到三角函数的第一个性质:

单位圆性质

$$\cos^2\alpha + \sin^2\alpha = 1$$

在正切、正弦和余弦之间也有一个关系, 如果你回头看看这三个函数的定义, 将会看到:

$$\sin\alpha = \frac{\text{opp}}{\text{hyp}}$$

$$\cos\alpha = \frac{\text{adj}}{\text{hyp}}$$

那么将两式相比, 可得

$$\frac{\sin\alpha}{\cos\alpha} = \frac{\text{opp/hyp}}{\text{adj/hyp}} = \frac{\text{opp}}{\text{adj}}, \quad \text{即为 } \tan\alpha \text{ 的定义。}$$

因此可得到下面两条性质:

正切和余切

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha}$$

$$\cot\alpha = \frac{\cos\alpha}{\sin\alpha}$$

下面给出关于负角的一些性质, 你可以代入一些值, 使用计算器验证它们的正确性。

负角的性质

$$\sin(-\alpha) = -\sin\alpha$$

$$\cos(-\alpha) = \cos\alpha$$

$$\tan(-\alpha) = -\tan\alpha$$

【例 3-12】 验证负角性质

利用角度 $\alpha = 30^\circ$, 验证负角的性质 $\sin(-\alpha) = -\sin\alpha$ 。

解答:

1. 使用计算器计算出 $\sin(-30^\circ)$, 可得到 $\sin(-30^\circ) = -0.5$ 。
 2. 使用计算器算出 $-\sin(30^\circ)$, 由于 $\sin(30^\circ) = 0.5$, 因此 $-\sin(30^\circ) = -0.5$ 。
- 即所证负角性质是正确的。

下面让我们来看正弦函数的和与差:

正弦函数的和与差

$$\sin(\alpha_1 + \alpha_2) = \sin\alpha_1 \cos\alpha_2 + \cos\alpha_1 \sin\alpha_2$$

$$\sin(\alpha_1 - \alpha_2) = \sin\alpha_1 \cos\alpha_2 - \cos\alpha_1 \sin\alpha_2$$

【例 3-13】 $\sin(90^\circ + \alpha)$

化简 $\sin(90^\circ + \alpha)$ 。

解答:

1. 利用正弦函数和的公式:

$$\sin(\alpha_1 + \alpha_2) = \sin\alpha_1 \cos\alpha_2 + \cos\alpha_1 \sin\alpha_2$$

$$\sin(90^\circ + \alpha) = \sin 90^\circ \cos\alpha + \cos 90^\circ \sin\alpha$$

2. 代入 $\sin 90^\circ$ 和 $\cos 90^\circ$ 的值化简:

$$\sin(90^\circ + \alpha) = (1)\cos\alpha + (0)\sin\alpha$$

$$\sin(90^\circ + \alpha) = \cos\alpha$$

【例 3-14】 $\sin(180^\circ - \alpha)$

化简 $\sin(180^\circ - \alpha)$ 。

解答:

1. 利用正弦函数差的公式:

$$\sin(\alpha_1 - \alpha_2) = \sin\alpha_1 \cos\alpha_2 - \cos\alpha_1 \sin\alpha_2$$

$$\sin(180^\circ - \alpha) = \sin 180^\circ \cos\alpha - \cos 180^\circ \sin\alpha$$

2. 代入 $\sin 180^\circ$ 和 $\cos 180^\circ$ 的值化简:

$$\sin(180^\circ - \alpha) = (0)\cos\alpha - (-1)\sin\alpha$$

$$\sin(180^\circ - \alpha) = \sin\alpha$$

余弦函数的和与差也有类似但不同的性质:

余弦函数的和与差

$$\cos(\alpha_1 + \alpha_2) = \cos\alpha_1 \cos\alpha_2 - \sin\alpha_1 \sin\alpha_2$$

$$\cos(\alpha_1 - \alpha_2) = \cos\alpha_1 \cos\alpha_2 + \sin\alpha_1 \sin\alpha_2$$

【例 3-15】 $\cos(180^\circ + \alpha)$

化简 $\cos(180^\circ + \alpha)$ 。

解答:

1. 利用余弦函数的和公式:

$$\cos(\alpha_1 + \alpha_2) = \cos\alpha_1 \cos\alpha_2 - \sin\alpha_1 \sin\alpha_2$$

$$\cos(180^\circ + \alpha) = \cos 180^\circ \cos\alpha - \sin 180^\circ \sin\alpha$$

2. 代入 $\sin 180^\circ$ 和 $\cos 180^\circ$ 的值, 化简

$$\cos(180^\circ + \alpha) = (-1)\cos\alpha - (0)\sin\alpha$$

$$\cos(180^\circ + \alpha) = -\cos\alpha$$

【例 3-16】 $\cos(90^\circ - \alpha)$

化简 $\cos(90^\circ - \alpha)$ 。

解答:

1. 利用余弦函数的差公式:

$$\cos(\alpha_1 - \alpha_2) = \cos\alpha_1 \cos\alpha_2 + \sin\alpha_1 \sin\alpha_2$$

$$\cos(90^\circ - \alpha) = \cos 90^\circ \cos\alpha + \sin 90^\circ \sin\alpha$$

2. 代入 $\sin 90^\circ$ 和 $\cos 90^\circ$ 的值, 化简:

$$\cos(90^\circ - \alpha) = (0)\cos\alpha + (1)\sin\alpha$$

$$\cos(90^\circ - \alpha) = \sin\alpha$$

这一节介绍了许多三角函数的相关性质, 即从单位圆性质到三角函数的和与差的性质。作为一个程序员, 你应该知道使用三角函数的代价相当高, 相比简单的加法和乘法, 它们要花费更多的处理时间, 因此在编程中, 应该尽量减少使用三角函数的个数。在下一节, 我们将介绍如何使用 C++ 函数库中的三角函数。

自我测试

1. 利用单位圆计算 $\sin 180^\circ$ 和 $\cos 180^\circ$ 。
2. 利用第 1 题中得到的数值, 验证单位圆性质在角 180° 时成立。
3. 不使用计算器的正切函数, 计算出 $\tan 30^\circ$ 。
4. 利用三角函数和的性质计算 $\sin(2\alpha)$ 。
5. 计算 $\cos(2\alpha)$ 。
6. 如果你建立的正弦函数值查找表只能储存到 $\sin(90^\circ)$, 那么怎样计算出 $\sin(120^\circ)$ 的值?

3.4 使用 C++ 中的数学库

当你开始编程时, 大可不必闭门造车, 什么函数都尝试自己去写一遍, 因为许多函数可能已经被别人写过了, 何不把它直接拿来使用呢? 我们已经掌握了三角函数的相关知识, 现在该是使用它们的时候了。如果你用 C++ 编程, 可以直接包含头文件 `math.h`, 从而使用编译器自带的数学库, 在它之中就包括了正弦、余弦、正切以及它们的反函数。

注意

无论什么时候想用这些三角函数, 只要在你的程序代码开头包含头文件 `math.h` 就行了, 方法如下:

```
#include <math.h>
```

对于数学库中的正弦、余弦、正切函数，它们的输入和输出值都是双精度 double 型，输入的参数是一个弧度形式的角，输出的数值是以十进制形式，比如正弦、余弦值在 -1 和 1 之间。一定要记住，输入角度必须以弧度形式表示，因此你可能需要将角度转换为弧度（参看本章第一节）。编程人员常常定义一个常数 π ：

$$\pi = 3.14159265$$

然后利用它就可将角度值转换为弧度形式。下面给出一个调用正弦函数的典型形式：

```
result = sin(angle * pi / 180);
```

对于一个经常要用到的常量，最好在头文件中用#define 将其定义成一个宏。

```
#define PI 3.14159265f
#define RadToDeg  57.29577951f
#define DegToRad  0.017453293f
```

对于函数库中的余弦和正切函数，它们也具有相同的调用形式。

math.h 中的三角函数

```
sine: double sin(double x)
cosine: double cos(double x)
tangent: double tan(double x)
```

其中，当输入的角度用弧度表示时，正弦和余弦的返回值在 -1 和 1 之间。

反三角函数也在 math.h 中定义，请记住反三角函数用来得到角度（看本章 3.2 节）。这些三角函数输入值和输出值也是 double 双精度型，当然如果你需要，可以把弧度值转换成角度值。这些三角函数被写成 asin、acos、atan。

math.h 中的反三角函数

```
arcsine: double asin(double x)
arccosine: double acos(double x)
arctangent: double atan(double x)
```

其中，当输入一个双精度值时函数将返回一个弧度值。

调用反三角函数可以采用如下的形式：

```
angle = asin(input) * 180 / pi;
```

注意

不要忘记在使用数学函数之前包含数学库头文件 math.h。

```
#include <math.h>
```

同样你也可以调用更加常用的 <cmath> 库，它会更加兼容一些。

注意反正弦和反余弦输入值的范围必须在 1 和 -1 之间。

提示

在计算机中，除法运算会比乘法运算更加耗时，因此可以去定义另一个常量 $1/\pi$ (0.31830989)，然后在程序中乘以它，从而代替除法运算。

注意

现在我们已经为下一章学习向量打下了基础，这一节也介绍了在 C++ 中调用三角函数和反三角函数的方法，这也是前几节知识在编程中的总结和应用，恭喜你！现在你已经在编程中迈出了一大步。

本章总结

在这一章，我们复习了三角函数的一些概念，为后面几章的学习奠定了坚实的基础。然后深层次讨论了角度，这对于向量在模拟路径中的应用是很重要的。由于数学库中的函数都用弧度值作为输入，因此必须要掌握在弧度和角度之间互相转换。随后，我们定义了3种主要的三角函数，在下一章向量中我们将会应用它们，接着又讨论了三角函数的相关性质。最后，介绍了在 C++ 编程中如何运用这些函数，以节省时间。

能力测试

选择题

1. 以下哪个值等于 60° ?
A. $(1/3)\pi^R$ B. $(2/3)\pi^R$ C. $3\pi^R$ D. $(1/6)\pi^R$
2. 下面哪个值等于 $(3/4)\pi^R$?
A. 45° B. 60° C. 135° D. 180°
3. 判断正误，如果知道一个直角三角形其中两条边的长度，你能否计算出其中一个非直角的余弦值?
A. 正确 B. 错误
4. 如果你知道一个直角三角形的斜边长度是 18，角 α 的对边长度是 15，下列哪个值最接近角 α ?
A. 42° B. 56° C. 63° D. 83°
5. 方程 $y = 3\sin 2x$ 的周期是多少?
A. 60° B. 90° C. 120° D. 180°
6. $\cos(90^\circ + \alpha)$ 等于多少?
A. $\sin(\alpha)$ B. $\sin(-\alpha)$ C. $\cos(\alpha)$ D. $\cos(-\alpha)$

练习

练习 1：角度与弧度互换

1. 画出以下标准位置的角：

- 180°
 - 45°
 - -270°
2. 将下列角度转换成弧度:
- 60°
 - 270°
 - 45°
3. 将下列弧度转换成角度:
- $\pi/4$
 - $\pi/2$

练习 2: 三角函数

1. 使用计算器, 计算下列三角函数的值:
- $\sin 45^\circ$
 - $\cos 55^\circ$
 - $\tan -25^\circ$
2. 求出以下三角函数的周期和振幅:
- $y = 4\sin 2x$
 - $y = 3\cos x$
 - $y = \cos 2x$

练习 3: 三角函数的性质

1. 运用单位圆性质, 计算 $\sin(360^\circ)$ 和 $\cos(360^\circ)$ 。
2. 利用第 1 题中得到的数值, 验证在 360° 时单位圆性质是正确的。
3. 不使用计算器中的正切函数, 计算 $\tan(60^\circ)$ 。
4. 利用三角函数和的性质化简 $\sin(2\alpha)$ 。
5. 化简 $\cos(2\alpha)$ 。
6. 如果你使用查找表来储存正弦函数值, 但是只能储存到 $\sin(60^\circ)$, 那么如何计算出 $\sin(90^\circ)$ 的值?

程序设计

程序设计 1: 编写一个程序, 完成在角度与弧度之间的互相转换
编写一个简单的程序, 能够在角度与弧度之间相互转换。

程序设计 2: 已知直角三角形的两边, 编写一个程序
编写一个程序, 如果已知直角三角形的其中两边, 计算出三角形的另外两个角和第三边的长度。

程序设计 3: 编写程序显示一个图形
编写一个程序, 在屏幕上显示一艘上下来回波动的航船(或者任何一艘简单的船)。

程序设计4：总结使用数学运算库的好处和缺点

使用数学运算库有什么好处？另外，有哪些缺点？

问题讨论

1. 对于角度，规定和建立一个标准位置的好处是什么？
2. 当在程序中使用 π 时，有多少地方会使用该变量？
3. 在程序中的哪些地方会用到三角函数？
4. 利用正弦、余弦波形函数能够模拟什么类型的运动？
5. 在程序中的哪些地方会用到三角函数的性质？
6. 在编程中，应该在哪些时候使用查找表，哪些时候使用数学库？



第4章 向量运算

本章目标

认真阅读本章内容并完成练习之后，你能够掌握以下内容：

- 能够区分向量和标量
- 计算物体的位移
- 完成向量在极坐标系和笛卡儿坐标系的相互转换
- 用图形法计算向量的加法和减法
- 标量与向量相乘
- 单位化向量
- 计算两个向量的点乘
- 计算两个向量的叉乘
- 计算两个向量之间的夹角
- 计算平面法线

介绍

本章我们将要介绍向量相关的知识，比如如何去定义一个向量，以及怎样使用向量等。向量对我们来说，是一个相对较新的事物，它在 19 世纪才踏入数学的殿堂，那为什么要引入它呢？主要是由于当时的数学家和物理家们希望能够描述运动的物体，而不仅仅是静态的位置。我们过去一直使用的数字被称作标量，它可以用来描述一个数量或者一个静态位置等。不幸的是，由于标量不具有方向，因此它不能描述物体的运动。在过去的这么多年里，人们一直在使用标量进行着加法或者乘法运算，本章将会回顾这些运算，不过操作对象不是标量而是向量，它具有方向，能够控制游戏中所有物体的运动，本章还将建立向量运算的一些规则，以便在后续章节中使用。

4.1 向量与标量

当在游戏中使用一个量的时候，一定要区分它是标量还是向量，它们二者的最大差别就是是否具有方向。

当从你开始查数起，就已经开始使用标量了。标量就是一个数值，有时也可以称为无方向的量，假如你朋友的汽车发生了故障，此时他给你打电话向你请求援助，并且告诉你他在

离你 2 英里的地方，你肯定会立即答应并开上你的车前去帮忙，但是怎样才能找到他呢？假如他说在东大街离你 2 英里的地方，那么很快就可以找到他，这就是向量和标量的区别。2 英里是一个标量，它不能提供充分的信息；然而位于向东 2 英里处则是一个向量，它提供了你朋友所在的方向，因而能够指导你很快找到你朋友。正是有了方向，才使得世界变得不同，因此标量就是一个数值，它只具有大小；而向量则是既包含大小又包含方向的量，你可能会觉得该差别微不足道，但是当你真正开始进行物体模拟时才知道它们的差别是那么大。

向量与标量

标量 = 只有大小的量

向量 = 大小 + 方向

在程序中，标量可以存储到任意类型的变量中，其取值将由变量能够表示的范围所决定。对于一个字符类型的变量空间来说，它能够存储 -128 到 127 之间的任何一个值；而对于类型为无符号长整型变量，它能够存储的数据范围为 0 ~ 18 466 744 073 709 551 616 (即 2 的 64 次方)；对于类型为 double 双精度的变量，它能存储的数据范围为 $\pm 1.7E \pm 308$ (即 15 个数量级)。然而，并不存在专门用来存放向量的类型，不过像一些 3D 图形 API 如 OpenGL 或 Direct3D，它们都有自己的一套标准，因此向量可以用浮点数数组或者用户指定的特定类型来存储。

```
//使用具有 3 个浮点数的数组,来存储向量
float 3Dvector[3] = {0,0,0};
//使用用户指定的数据类型
struct 3Dvector
{
    float x, y, z;
};
```

通过使用合适的数据类型，从而能够创建功能强大的结构体或者类，然后再将需要进行的操作全部包含进来，以便于以后再次使用，这些内容在以后章节中会涉及到。

无论何时，当我们需要在编程中处理物体的运动时，一定要记住将其方向考虑进来。有些量，比如时间、点等不具有方向，因此利用标量表示它们就行了；然而对于位移、速度和力等这些量，常用来描述物体的运动，因此它们总包含方向。

你可能对位移和速度这两个概念不太清楚，其实位移就是距离的向量形式，而速度就是速率的向量形式。比如 55mph 只是个标量，所以叫作速率；如果将 55mph 再加上向东的方向属性，那么它就是向量，即叫作速度了，这两个概念会在第 8 章中详细介绍。

利用向量最有用的一点就是能够处理方向，对在一维空间内的运动来说，只有两个可能方向，可以用正负号来区分它们。例如物体在屏幕上只做左右方向上的运动，正值则意味着它向右运动，而负值则说明向左运动。同样的，当在处理垂直方向上的运动时，常常会用正值代表向上，而负值代表向下。

注意

在第1章中,我们曾讨论了3D空间内的不同坐标系。在本书中,通常把在屏幕上向上、向右和向外的方向叫作正方向,而把向下、向左和向里的方向叫作负方向。请注意你使用的引擎所采用的坐标系。

【例4-1】 正向位移

假如在游戏编程中,一个挡板只能在竖直(上下)方向上运动,现在该挡板的中心位置为(20, 50),然后它移动到位置(20, 400),那么它的位移是多少?

解答:

观察其 y 坐标,可以发现挡板从50像素的位置移动到400像素的位置,因此它的位移就是 $400 - 50 = 350$ 像素。

【例4-2】 负向位移

同样的,对于上例中的游戏,挡板只能在竖直(上下)方向上移动,它当前的中心位置在(20, 400),然后移动到位置(20, 50),那么它的位移是多少?

解答:

观察其 y 坐标,可以发现这次挡板从400像素位置移动到50像素位置,因此它的位移是 $50 - 400 = -350$ 像素。

注意

在一维空间中,利用正负即可完全确定物体的运动方向,上例4-1具有正的位移,现在翻转其位置后,例4-2中的位移就变成了负值。

当处理位移时,方向是非常重要的,距离(标量)和位移(向量)具有较大的差别。当计算物体的位移时,只需关注物体的起始位置和最终位置即可,至于在该移动过程中发生了什么则是不重要的。橄榄球就是一个区别距离和位移的很好例子,假如接球者在20码的位置接到了球,然后他就开始奔跑,但是途中有一个对方队员,于是他绕过对方队员,最终到达50码的位置。

图4-1画出了该队员在这个过程中的运动路线,很显然他经过的距离远大于30码,然而对于游戏来说,他只给球队赢得了30码。此时的30码就是他的位移,而他实际经过的距离大于30码。如果完全忽略方向,那么它就会变成一个标量——距离,这就是距离和位移的主要区别。

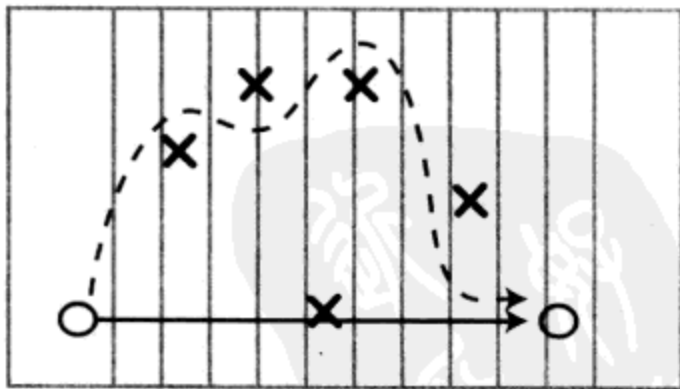


图4-1 队员的位移

位移公式

位移 = 最后位置 - 原始位置;

$$\Delta x = x_f - x_i。$$

注意

请记住，当计算物体的位移时，方向将要起作用，所要求的就是物体的最终位置和原始位置；如果要是计算标量——距离的话，那就要考虑中间过程了。

【例4-3】 距离与位移

假如你正在玩玛丽兄弟游戏，他只能向左或向右运动(而不会跳跃)，如果他的原始水平位置为200像素，然后他开始向右运动，但当到达250像素处时他发现错过了一个蘑菇，于是他返回到100像素位置去采蘑菇，然后他奔跑到450像素位置，与公主相遇。问他经过的总位移是多少？他实际经过的距离是多少？

解答：

1. 首先计算位移。他的初始位置为200像素处，结束位置在450像素处。因此，他总共经过的位移是：

$$\text{最终位置} - \text{原始位置} = 450 \text{ 像素} - 200 \text{ 像素} = 250 \text{ 像素}$$

2. 当计算距离时，可以忽略方向。因此将他每段走过的距离进行相加，从200像素到250像素距离是50像素，然后他又走了150像素到达蘑菇处，此时总的距离为200像素；最后他又跑了350像素到达公主处，因此他总共走过的距离为550像素。

从上例可以看出在大多数情况下，距离和位移的值会相差很大。同样可以发现，当计算位移时忽略了所有的中间过程，也许你会想把过程分解为更小的时间间隔，以不丢失相关信息。在上例玛丽游戏中，可以将该过程划分为3个阶段，请记住当在编程时，一定要保持位移的方向，因此上述3个阶段就是：+50像素、-150像素到达蘑菇位置和+350像素到达公主的位置。

注意

如果你正在编写一个实时游戏，尽量要将时间间隔取为接近于1/30秒，从而尽量在两帧之间不丢失太多信息。

对标量的处理比较容易，因为你在生活中一直使用着它。然而对于向量，它是一个新的概念，而且很容易忘记它与方向相关。对于一维空间内的运动，其处理也比较容易，因为可以使用正负号来代表向量的方向，本章的后面部分将会处理2D和3D空间内的情形，不过它会比较复杂一些。

自我测试

1. 向量和标量之间的差别是什么？
2. -65feet 代表的是标量还是向量？
3. 35秒是标量还是向量？
4. 在一个直线跑道上，一个运动员的起始位置为5ft处，最后他停到65ft处，那么他的位移是多少？
5. 同样在问题4中，另一个运动员的起始位置为65ft，最终停到位置5ft处，那么他的位移又是多少？

6. 假如你正在玩挡板游戏，挡板的起始 y 坐标为 250 像素，现在你将其向下移动到 100 像素处去挡小球，然后你又将其向上移动到 300 像素处以便再次去挡小球，当球快要接触挡板时，你发现挡板的位置偏了一些，于是你又将其向下移动了 20 像素，那么挡板的总共位移是多少？
7. 在问题 6 中，挡板经过的实际距离是多少？
8. 在棒球游戏中，一个运动员正在跑垒，他想盗垒上二垒，经过激烈的跑垒，最后他终于跑完全垒，那么他的位移是多少？

4.2 极坐标与笛卡儿坐标

通过上一节的学习，我们知道对于一维空间内的运动，使用正负号就能确定向量的方向。然而，在 2D 和 3D 空间中仅使用正负还不够，2D 空间内的向量具有两种表示形式：极坐标和笛卡儿坐标。极坐标可能会更直观一点，因此首先来看极坐标。

极坐标

向量 $A = \|A\| @ \theta$

其中， $\|A\|$ 为向量 A 的模， θ 为方向角。

图 4-2 显示了向量 A 的极坐标表示。

注意

向量 A 是大写字母，本书使用大写字母来表示向量。有些书籍使用带箭头的形式来表示，如 $\vec{A}$ ，它们代表的意义是一样的。此外，@ 读作“at”，比如 $20\text{m} @ 30^\circ$ ，读作“长度为 20m，且位于标准位置 30 度角的向量”。

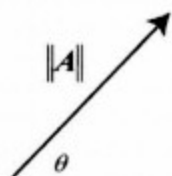


图 4-2 向量 A 的极坐标表示

从极坐标可以直观的看出向量的外观，这也就是我们在第 3 章中建立标准位置角的原因，现在就要想法如何确定向量的大小与方向，显然方向就是位于标准位置的角，它可以代表 2D 空间内的任何方向。

利用笛卡儿坐标表示向量不太直观，但是在编程中都将使用该形式。它不是利用长度和方向来表示向量，而是用它在水平和竖直方向上的位移分量来表示，这两个部分就叫作水平分量和竖直分量，并将它们写成“ i 和 j 的形式”。

笛卡儿坐标(分量)

向量 $B = b_1 \hat{i} + b_2 \hat{j}$

其中， $\hat{i}$ 和 $\hat{j}$ 分别为 x 和 y 方向上的单位向量。

图 4-3 显示了向量 B 的笛卡儿坐标系形式。

注意

上述向量的 i 和 j 形式，看起来会觉得有些奇怪，主要是由于 i 和 j 上面那个符号。

它们分别读作：“在 x 方向上”和“在 y 方向上”，其实 $\hat{i}$ 和 $\hat{j}$ 本身就是一个向量，它们的模（大小）都是 1，而且其方向分别沿 x 轴和 y 轴的正方向。

此外还要注意一点，当某个分量为负值时，这将意味着方向是向左或向下，而不是向右或向上。

由于电脑屏幕是通过栅格的形状制造的，因此在编程中会经常使用笛卡儿坐标系。通常的做法是：首先用极坐标的形式分析问题，然后在具体编程中再将其转换成笛卡儿坐标系。现在，就让我们来看将极坐标转换成笛卡儿坐标的处理过程。观察图 4-4 中的向量 A ，可以借助于直角三角形利用三角函数完成该转换。

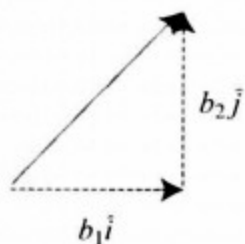


图 4-3 向量 B 的笛卡儿坐标形式

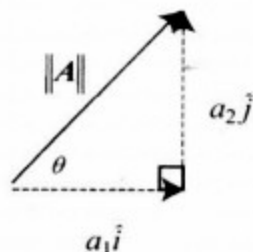


图 4-4 极坐标转换成笛卡儿坐标

极坐标转换成笛卡儿坐标

对于向量 $A = \|A\| @ \theta$,

$$A = a_1 \hat{i} + a_2 \hat{j}$$

其中， $a_1 = \|A\| \cos \theta$, $a_2 = \|A\| \sin \theta$ 。

【例 4-4】 将极坐标转换成笛卡儿坐标

向量 A 是一个位移向量，将 $A = 20\text{m} @ 30^\circ$ 转换成笛卡儿坐标。

解答：

利用正弦函数和余弦函数将向量 A 分解成两个分量：

$$a_1 = \|A\| \cos \theta = 20 \cos 30^\circ \approx 20(0.8660) \approx 17.32$$

$$a_2 = \|A\| \sin \theta = 20 \sin 30^\circ = 20(0.5) = 10$$

因此， $A = 17.32 \hat{i} + 10 \hat{j}$ 即是向量 A 的 i 、 j 分量形式。

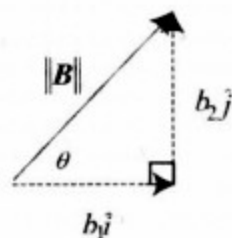


图 4-5 笛卡儿坐标转换成极坐标

笛卡儿坐标转换成极坐标

对于向量 $B = b_1 \hat{i} + b_2 \hat{j}$,

$$\|B\| = \sqrt{(b_1)^2 + (b_2)^2}, \quad \theta = \tan^{-1} \left(\frac{b_2}{b_1} \right)$$

图 4-5 说明了该转换过程。

【例 4-5】 笛卡儿坐标转换成极坐标

将向量 $B = 3 \hat{i} + 4 \hat{j}$ 转换成极坐标。

解答:

1. 计算向量 B 的模:

$$\|B\| = \sqrt{(b_1)^2 + (b_2)^2} = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

2. 计算方向:

$$\theta = \tan^{-1}\left(\frac{b_2}{b_1}\right) = \tan^{-1}\left(\frac{4}{3}\right) \approx 53.1^\circ$$

因此向量 B 的极坐标形式为: $B=5$ 单位@ 53.1° 。

下面让我们来看一些函数, 它们对应于前面问题的解, 首先定义将要使用到的数据类型:

```
//保存向量分量形式的结构体
struct 2Dvector_comp
{
    float x, y;
};
//保存向量极坐标形式的结构体
struct 2Dvector_polar
{
    float mag, dir;
};
```

下面让我们介绍两个函数, 完成极坐标与笛卡儿坐标下向量的转换。

```
//功能:将一个向量由极坐标形式转换为分量形式
//输入:vec——待转换的向量,极坐标形式
//输出:转换后的向量
2Dvector_comp PolarToCompConversion(2Dvector_polar vec)
{
    //临时变量,保存转化后的结果
    2Dvector_comp temp;
    //填入数值
    temp.x = mag * cos(dir * PI / 180);
    temp.y = mag * sin(dir * PI / 180);
    //返回结果
    return temp;
}
```

```
//功能:将一个向量由分量形式转换为极坐标形式
//输入:vec——待转换的向量,分量形式
//输出:转换后的向量
2Dvector_polar CompToPolarConversion(2Dvector_comp vec)
{

```

```

//临时变量,保存转化后的结果
2Dvector_polar temp = {0,0};
//计算向量的模
temp.mag = sqrtf( vec.x * vec.x + vec.y * vec.y );
//错误检测,判断除数是否为0
if(temp.mag == 0)
return temp;

//计算角度,我们使用 asin()函数来完成
temp.dir = (180 /PI) * asin(vec.y /temp.mag);

//当角度位于第二或三象限时,调整角度值,
if(vec.x < 0)
temp.dir += 180;

//当角度位于第四象限时,调整角度值,
else if( vec.x > 0 && vec.y < 0)
temp.dir += 360;

//返回新的向量
return temp;
}

```

前面已经提到过,在编程中通常使用笛卡儿坐标来表示向量。此外,该向量表示形式还可以很容易地扩展到3D空间中。

3D 中的笛卡儿坐标(分量)

$$\text{向量 } \mathbf{B} = b_1 \hat{i} + b_2 \hat{j} + b_3 \hat{k}$$

其中, $\hat{i}$ 、 $\hat{j}$ 和 $\hat{k}$ 分别为 x 、 y 和 z 方向上的单位向量。

你是否已经注意到,当我们在程序中实现某个数学概念之后,它们的表示形式往往会发生改变。比如对于笛卡儿坐标形式的向量,它常常会被写作具有单行或单列的矩阵形式,而不是前面所说的 i 、 j 形式。例如2D 矩阵 $\mathbf{A} = 5\hat{i} + 6\hat{j}$, 常常会写成如下形式: $[5 \ 6]$ 或 $\begin{bmatrix} 5 \\ 6 \end{bmatrix}$,

而对于3D 向量 $\mathbf{B} = 7\hat{i} + 8\hat{j} + 9\hat{k}$, 就会写成: $[7 \ 8 \ 9]$ 或 $\begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}$ 。

至于对矩阵进行单位化的操作,我们打算把它放到本章“向量与标量相乘”那一节再介绍。此外,对于笛卡儿坐标下的向量,请使用单行的矩阵来表示。

自我测试

1. 已知向量 A 是一个速度向量, 请将 $A = 25\text{m/s} @ 45^\circ$ 转换成笛卡儿坐标下的形式。
2. 将向量 $B = 12\hat{i} + 5\hat{j}$ 转换成极坐标形式。
3. 向量 C 是一个力向量, 将 $C = 200\text{N} @ 60^\circ$ 转换成笛卡儿坐标下的形式。
4. 将向量 $D = 8\hat{i} - 6\hat{j}$ 转换成极坐标形式。

4.3 向量加法和减法

当一个向量代表位移、速度或力时, 我们就可对其进行加法或减法运算, 这些运算将在第 10 章、第 11 章、第 12 章和第 14 章中还会被用到。下面就让我们首先利用图形法处理这些运算, 然后再介绍其精确形式的处理方法。

当用图形法求解向量的加法或减法运算时, 我们通常会在向量的一端加上箭头, 然后使用箭头的长度代表向量的模, 而箭头所指的方向代表向量的方向。由于这些向量都是根据比例绘制的, 因此一定要保证它们的比例是一样的(即同是米或者英尺等), 那么经过该方式的处理, 长度为 5m 的向量看起来是长度 10m 向量长度的一半, 如图 4-6 所示。

我们把带箭头的一端叫作向量的头端, 而另一端叫作向量的尾端, 如图 4-7 所示。

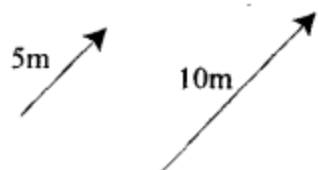


图 4-6 用箭头表示的两个向量

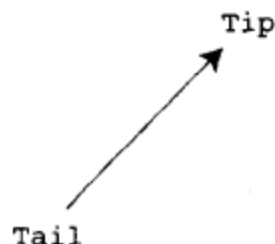
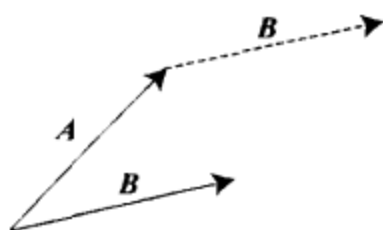
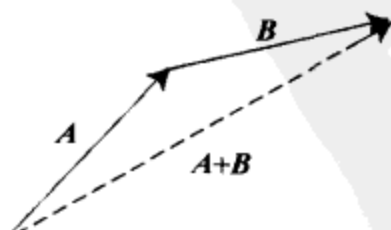


图 4-7 向量的头端和尾端

下面就可以利用首尾相连的方法进行向量的加法运算, 向量具有一个特点, 即它可以处于任何位置, 只要保持其长度和方向不变即可, 也就是说可以对其进行平移处理。如果使用图形法来完成两个向量的相加, 可以平移其中的一个向量, 使之与另一向量首尾相连, 如图 4-8 所示。

此时, 从第一个向量的尾端向第二个向量的头端, 从而构造一个新的向量。

观察图 4-9, 可以将其看作是地图上的位移问题。如果物体沿向量 A 移动, 而后沿向量 B 运动, 那么物体的起始位置为向量 A 的尾端, 最终位置为向量 B 的头端。

图 4-8 向量 A 和 B 进行首尾相连图 4-9 向量 $A+B$

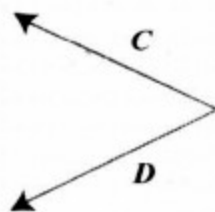
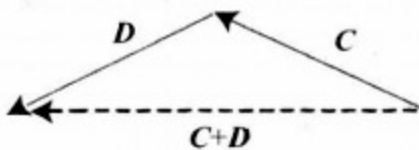
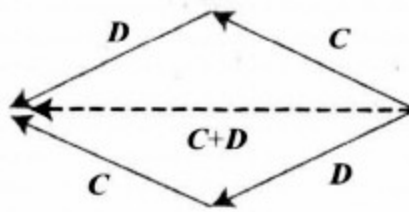
【例 4-6】 画图法实现向量相加

对于图 4-10 中的向量 C 和向量 D ，找出向量 $C+D$ 。

解答：

1. 平移向量 D ，使它与向量 C 首尾相连。确保向量 D 在平移过程中保持长度和方向不变。
2. 从向量 C 的尾部到向量 D 的头部，画出一个新向量 $C+D$ 。
3. 最终向量如图 4-11 所示。

观察图 4-11，我们同样可以对向量 C 进行平移，使之与向量 D 首尾相连，这样我们将得到同一个结果，如图 4-12 所示。

图 4-10 向量 C 和 D 图 4-11 向量 $C+D$ 图 4-12 $C+D=D+C$

向量相加的交换率

$$A+B=B+A$$

其中， A 、 B 为任意向量。

下面尝试给向量赋上一些特定值，你可能会从上面例子中发现，向量 $C+D$ 的长度会比向量 C 的长度加上向量 D 的长度短一些，这是因为向量运算将方向也考虑在内了，也就是说：

$$\|A+B\| \neq \|A\| + \|B\|$$

正是由于该原因，对于极坐标形式的两个向量，如果它们的方向不一致，就不能直接进行相加。因此在这里建议大家，当要进行相加运算时，总要将它们转换成笛卡儿坐标的形式，然后只对对应部分进行相加，也就是说分别相加其 $\hat{i}$ 、 $\hat{j}$ 分量。

精确相加两个向量

$$A+B=(a_1+b_1)\hat{i}+(a_2+b_2)\hat{j}$$

其中，向量 $A=a_1\hat{i}+a_2\hat{j}$ ，向量 $B=b_1\hat{i}+b_2\hat{j}$ 。

【例 4-7】 精确相加两个向量

已知向量 $C=8\hat{i}+3\hat{j}$ ， $D=5\hat{i}+12\hat{j}$ ，求向量 $C+D$ 。

解答：

1. 首先，利用画图法完成向量的相加。平移向量 D ，使之与向量 C 首尾相连，确保在平移过程中保持长度和方向不变。

2. 现在, 从向量 C 的尾端到 D 的头端画出一个新的向量 $C+D$, 如图 4-13 所示。

3. 精确计算向量相加的方法是, 分别将其在 x 和 y 方向上的分量相加, 即对应的 $\hat{i}$, $\hat{j}$ 分量。在本题中, 相加过程如下:

$$C + D = (8 + 5)\hat{i} + (3 + 12)\hat{j} = 13\hat{i} + 15\hat{j}$$

如图 4-14 所示。

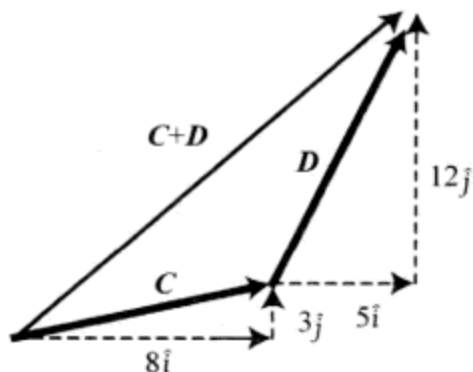


图 4-13 向量 $C+D$

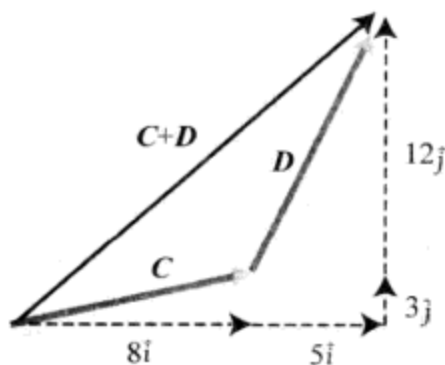


图 4-14 将向量的对应分量相加

注意

在上例 4-7 中, 向量一开始就是笛卡儿坐标的形式, 如果它们是极坐标形式, 则需要首先进行转换, 然后再执行相加运算。你若对转换不太熟悉, 可以复习前面介绍如何转换的相关知识。

向量相加法则同样适用于 3D 空间。但是, 在进行相加之前一定要将其转换成笛卡儿坐标的形式, 公式如下:

3D 向量相加

$$A + B = (a_1 + b_1)\hat{i} + (a_2 + b_2)\hat{j} + (a_3 + b_3)\hat{k}$$

其中, 向量 $A = a_1\hat{i} + a_2\hat{j} + a_3\hat{k}$, 向量 $B = b_1\hat{i} + b_2\hat{j} + b_3\hat{k}$ 。

【例 4-8】 3D 向量相加

已知向量 $C = 9\hat{i} + 5\hat{j} - 2\hat{k}$, $D = -2\hat{i} + \hat{j} + 4\hat{k}$, 求向量 $C+D$ 。

解答:

$$\begin{aligned} C + D &= (a_1 + b_1)\hat{i} + (a_2 + b_2)\hat{j} + (a_3 + b_3)\hat{k} \\ &= (9 - 2)\hat{i} + (5 + 1)\hat{j} + (-2 + 4)\hat{k} \\ &= 7\hat{i} + 6\hat{j} + 2\hat{k} \end{aligned}$$

从前面的例子可以看出, 将笛卡儿坐标形式的两个向量进行相加非常简单。同样的, 对于向量相减也一样比较容易, 即将对应向量相减即可。

2D 向量相减

$$\mathbf{A} - \mathbf{B} = (a_1 - b_1)\hat{i} + (a_2 - b_2)\hat{j}$$

其中, 向量 $\mathbf{A} = a_1\hat{i} + a_2\hat{j}$, 向量 $\mathbf{B} = b_1\hat{i} + b_2\hat{j}$ 。

3D 向量相加

$$\mathbf{A} - \mathbf{B} = (a_1 - b_1)\hat{i} + (a_2 - b_2)\hat{j} + (a_3 - b_3)\hat{k}$$

其中, 向量 $\mathbf{A} = a_1\hat{i} + a_2\hat{j} + a_3\hat{k}$, 向量 $\mathbf{B} = b_1\hat{i} + b_2\hat{j} + b_3\hat{k}$ 。

实际上, 向量相减就相当于加上第二个向量的负向量, 也就是说将第二个向量的各个分量取反, 然后再进行向量相加即可。如果要想用画图的方法演示该运算, 同样可以反转第二个向量的方向, 然后利用首尾相连的方法即可, 如图 4-15 所示。

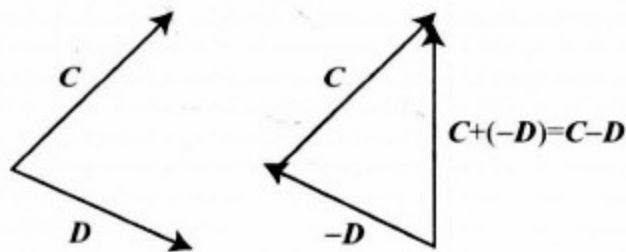


图 4-15 向量相减相当于加上一个负向量

【例 4-9】 3D 向量相减

已知向量 $\mathbf{C} = 9\hat{i} + 5\hat{j} - 2\hat{k}$, $\mathbf{D} = -2\hat{i} + \hat{j} + 4\hat{k}$, 求向量 $\mathbf{C} - \mathbf{D}$ 。

解答:

$$\begin{aligned}\mathbf{C} - \mathbf{D} &= (a_1 - b_1)\hat{i} + (a_2 - b_2)\hat{j} + (a_3 - b_3)\hat{k} \\ &= (9 + 2)\hat{i} + (5 - 1)\hat{j} + (-2 - 4)\hat{k} \\ &= 11\hat{i} + 4\hat{j} - 6\hat{k}\end{aligned}$$

到目前为止, 我们已经介绍了向量的两种形式: 笛卡儿坐标形式(分量形式)和极坐标形式(模和方向), 而后利用直角三角形的运算法则(勾股定理和三角函数)实现了这两种形式的互相转换; 最后又分别介绍了 2D 和 3D 向量进行相加、相减的运算法则, 可以发现对于采用笛卡儿坐标形式的向量, 它们的运算比较简单。在下面几节中, 我们将介绍向量相乘的运算法则。

自我测试

1. 将向量 $\mathbf{A} = 20\text{ft} @ 80^\circ$ 转换成笛卡儿坐标形式。
2. 将向量 $\mathbf{B} = 6\hat{i} + 8\hat{j}$ 转换成极坐标形式。
3. 利用前面两个问题中的向量, 计算向量 $\mathbf{A} + \mathbf{B}$ 。
4. 利用前面两个问题中的向量, 计算向量 $\mathbf{A} - \mathbf{B}$ 。
5. 利用图 4-16 中的向量 $\mathbf{C}$ 和 $\mathbf{D}$, 计算向量 $\mathbf{C} + \mathbf{D}$ 。
6. 已知 3D 向量 $\mathbf{F} = 2\hat{i} - 4\hat{j} - \hat{k}$, $\mathbf{G} = -5\hat{i} + 3\hat{j} + 6\hat{k}$, 计算向量 $\mathbf{F} + \mathbf{G}$ 。
7. 已知 3D 向量 $\mathbf{F} = 2\hat{i} - 4\hat{j} - \hat{k}$, $\mathbf{G} = -5\hat{i} + 3\hat{j} + 6\hat{k}$, 计算向量 $\mathbf{F} - \mathbf{G}$ 。

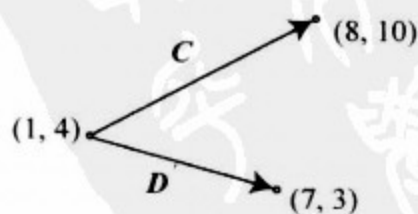


图 4-16 向量 $\mathbf{C}$ 和 $\mathbf{D}$

4.4 标量与向量相乘

标量之间的相乘，肯定大家都应该很熟悉吧？其实，标量相乘就等于两个数字相乘，现在让我们来看标量与向量相乘。对于采用极坐标形式的向量，它只包含两个部分——模和方向，当用标量与之相乘时，只需将标量与其模相乘即可，而方向不用改变。如果相乘标量是一个整数，那么向量的模将会变大；否则，若它为一个小于1的小数，向量的模将变小。

标量与极坐标向量相乘

$$cA = c \|A\| @ \theta$$

其中， c 为任意标量，向量 $A = \|A\| @ \theta$ 。

【例 4-10】 标量与向量(极坐标形式)相乘

已知向量 $A = 3\text{ft} @ 22^\circ$ ，计算向量 $5A$ 。

解答：

$$5A = 5(3\text{ft} @ 22^\circ) = 15\text{ft} @ 22^\circ$$

如图 4-17 所示，它以图形的方式演示了该运算。



图 4-17 向量 $2A$

请记住，以箭头方式显示的向量，它的长度和向量的模相对应，而箭头的指向与向量的方向相对应。正如你所看到的，向量 $2A$ 的长度是向量 A 的 2 倍，但方向并不发生改变。

假若现在向量采用的是笛卡尔坐标形式，那么标量与之相乘的运算怎么进行呢？提示你一下，大可不必再将其转换成极坐标形式，而只需简单地将标量与向量各分量分别相乘即可，公式如下。

标量与笛卡尔坐标向量相乘

$$cA = ca_1 \hat{i} + ca_2 \hat{j}$$

其中， c 为任意标量，向量 $A = a_1 \hat{i} + a_2 \hat{j}$ 。

【例 4-11】 标量与向量(笛卡尔坐标形式)相乘

已知向量 $A = 12\hat{i} + 4\hat{j}$ ，计算向量 $\frac{1}{2}A$ 。

解答：

$$\frac{1}{2}A = \frac{1}{2}(12\hat{i}) + \frac{1}{2}(4\hat{j}) = 6\hat{i} + 2\hat{j}$$

在游戏编程中，你会经常听到“单位化”这个术语，其实单位化就是将向量的模缩放到 1 的过程。通常我们会用向量来表示一个方向，也就是采用 $\hat{i}$ 、 $\hat{j}$ 分量的形式。其中， $\hat{i}$ 分量代表沿 x 轴正方向的单位向量(模为 1)， $\hat{j}$ 分量代表沿 y 轴正方向的单位向量。在后面章节中，我们会介绍如何使用单位向量来表示方向。

对于极坐标形式的向量，单位化过程非常简单，只需将其模变为 1，而方向保持不变。

然而在编程中，通常采用向量的笛卡儿坐标形式，下面介绍一下该向量形式的单位化过程。首先计算向量的模，然后将每个分量除以该数值，也相当于用模的倒数去乘以该向量。

2D 向量的单位化

$$\hat{A} = \frac{1}{\|A\|} A = \left[\frac{a_1}{\|A\|} \quad \frac{a_2}{\|A\|} \right]$$

其中向量 $A = [a_1 \quad a_2]$ 。

3D 向量的单位化过程与之类似。

3D 向量的单位化

$$\hat{A} = \frac{1}{\|A\|} A = \left[\frac{a_1}{\|A\|} \quad \frac{a_2}{\|A\|} \quad \frac{a_3}{\|A\|} \right]$$

其中向量 $A = [a_1 \quad a_2 \quad a_3]$ 。

注意

对于单位化向量，我们在其顶部作一个标记。比如，向量 A 的模为 1，则将其写成 $\hat{A}$ 。

【例 4-12】 单位化向量

将向量 $A = [5 \quad 0 \quad -12]$ 进行单位化。

解答：

1. 首先计算向量的模。（可能需要你复习一下本章介绍极坐标与笛卡儿坐标的那部分内容。）

$$\|A\| = \sqrt{5^2 + 0^2 + (-12)^2} = \sqrt{25 + 0 + 144} = \sqrt{169} = 13$$

2. 现在已经计算出了向量的模，将其各分量除以该数值，于是可得到：

$$\hat{A} = \left[\frac{a_1}{\|A\|} \quad \frac{a_2}{\|A\|} \quad \frac{a_3}{\|A\|} \right] = \left[\frac{5}{13} \quad \frac{0}{13} \quad \frac{-12}{13} \right]$$

注意

如果要想检查上面得到的结果是否正确，可以计算一下向量 $\hat{A}$ 的模 $\|\hat{A}\|$ ，看它是否等于 1。

由本节的内容可知，标量与向量进行相乘的结果就是将向量的模进行了扩大或者缩小，即向量的模发生了改变，而方向则保持不变。向量单位化的过程其实就是标量与向量相乘的一种特殊情形，此时标量等于向量模的倒数。此时，你可能会自然地问到，两个向量进行相乘又是怎样呢？回答该问题也很容易，我们会在接下来的两节中介绍这些知识。

自我测试

1. 标量与向量相乘，是否会影响向量的模和方向？
2. 向量 $A = 12\text{m/s} @ 43^\circ$ ，计算向量 $(-1/3)A$ 。
3. 已知向量 $B = 3\hat{i} - \hat{j}$ ，计算向量 $6B$ 。

4. 单位化向量 $C = [24 \ 10]$ 。

5. 单位化向量 $D = [0 \ 7 \ 24]$ 。

4.5 点乘

还记得在小学时学过的乘法口诀吧，那时老师可能会教你：2 乘以 3 可以写成 2×3 或者 $2 \cdot 3$ ，它们所代表的意思是一样的。其实该情形只对于标量是正确的，而对于复杂结构，比如向量，它就不正确了。我们常把类似 $2 \cdot 3$ 的形式叫作点乘，而把 2×3 的形式叫作叉乘，本节将介绍点乘，在下一节再介绍叉乘。

值得注意的是，两个向量点乘的结果总是一个标量。首先让我们来看点乘是如何运算的，然后再讨论它具体有什么含义。

注意

有些数学课本上，把向量的点乘叫作向量的数量积，这是因为向量的点乘运算返回的是一个数值。

2D 向量的点乘

$$A \cdot B = a_1 b_1 + a_2 b_2$$

其中，向量 $A = [a_1 \ a_2]$ ， $B = [b_1 \ b_2]$ 。

由上面公式可知，向量的点乘运算就是将对应 x 分量相乘，然后再加上对应 y 分量的乘积，那么该运算得到的结果将是一个数值。此时，你也许能够猜到如何将其扩展到 3D 向量中。

3D 向量的点乘

$$A \cdot B = a_1 b_1 + a_2 b_2 + a_3 b_3$$

其中，向量 $A = [a_1 \ a_2 \ a_3]$ ， $B = [b_1 \ b_2 \ b_3]$ 。

注意

向量的点乘运算不仅适用于 2D 和 3D 向量，它还适用于任意维的向量。在第 5 章中，我们会看到一个高维的例子，其实已经把它作为矩阵看待了。对于 4D 或者 10D 向量，即使它们没有具体的物理意义，但是只要满足参加运算向量的维数相同，就能够进行点乘运算。

向量的点乘是一个非常有用的运算，它的运算结果蕴涵着许多有用信息。首先，由它可以获得两向量之间角度的信息（对于 2D 和 3D 向量而言）。

判断是否垂直

如果 $A \cdot B = 0$ ，那么 $A \perp B$ 。

注意

你可能会对符号 $\perp$ 不太熟悉，它代表的含义是垂直。如果向量 A 、 B 的点乘等于 0，那么向量 A 垂直于 B 。

这是一种检测直线是否垂直的有效方法。对于2D向量，只需进行两次乘法和一次加法运算即可；对于3D向量，则需要3次乘法与2次加法运算，然后再判断结果是否等于0。如果点乘的结果不等于0，结果数值的正负号也能够提供有用的信息。

点乘结果的正负

如果 $A \cdot B < 0$ (负值)，那么 $\theta > 90^\circ$

如果 $A \cdot B > 0$ (正值)，那么 $\theta < 90^\circ$

其中， θ 为向量 A 和 B 之间的夹角。

【例4-13】检测物体是否可见

假设在一款游戏中，摄像机的当前位置是(1, 4)，向量 $C = [5 \ 3]$ 代表摄像机的视线。现在假设一个物体的位置为(7, 2)，且摄像机只能看到某方向上 90° 范围内的物体(即最大视角为 90°)，问该物体是否可见？

解答：

1. 首先，找到由摄像机的当前位置到物体位置所构成的向量，如图4-18所示。

计算上述向量，可以用物体的位置减去摄像机位置，把它称作向量 D ，那么 $D = [(7-1) \ (2-4)] = [6 \ -2]$ 。

2. 计算向量 C 和 D 的点乘 $C \cdot D$ ；

$$C \cdot D = 5(6) + 3(-2) = 30 - 6 = 24$$

3. 由于 $C \cdot D > 0$ ，那么摄像机视线与物体之间的夹角一定小于 90° ，这也就意味着物体是可见的。

当要判断两个向量之间的夹角是否大于 90° ，或者小于 90° 时，该方法非常简单和有效。若想进一步知道该角度的具体信息，利用点乘仍能够计算出该数值。

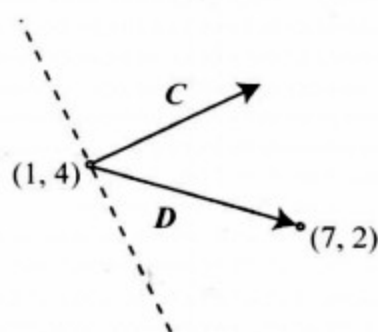


图4-18 摄像机方向

两向量之间的夹角

$$A \cdot B = \|A\| \|B\| \cos\theta$$

其中， θ 为两向量 A 和 B 之间的夹角。

【例4-14】两向量之间的角度

假设你当前的前进方向(速度)由向量 $C = [5 \ 2 \ -3]$ 确定，现在你又想改变前进方向，新方向有向量 $D = [8 \ 1 \ -4]$ 确定，那么当前向量 C 和目标向量 D 之间的夹角是多少？

解答：

1. 首先，利用公式计算向量 C 和 D 的点乘 $C \cdot D$ ；

$$C \cdot D = 5(8) + 2(1) - 3(-4) = 40 + 2 + 12 = 54$$

2. 然后，计算各个向量的模。此时，你可能需要复习一下计算向量模的相关知识。

$$\|C\| = \sqrt{5^2 + 2^2 + (-3)^2} = \sqrt{25 + 4 + 9} = \sqrt{38}$$

$$\|D\| = \sqrt{8^2 + 1^2 + (-4)^2} = \sqrt{64 + 1 + 16} = \sqrt{81}$$

3. 将得到的数值代入到上面的新公式, 计算角度 θ 。

$$C \cdot D = \|C\| \|D\| \cos\theta$$

$$54 = \sqrt{38}(\sqrt{81})\cos\theta$$

$$\frac{54}{\sqrt{38}(\sqrt{81})} = \cos\theta$$

$$0.9733 \approx \cos\theta$$

$$\theta \approx \cos^{-1}(0.9733)$$

$$\theta \approx 13.3^\circ$$

也就是说, 你必须旋转 13.3° 才能转到新的前进方向上。

向量的点乘运算还有一个重要含义, 它代表了向量在另一个向量方向上的投影长度。为了更直观一些, 如图 4-19 所示, 向量 A 将要投影到向量 B 上, 假设在向量 A 的正上方有一个光源, 它向下照射到向量 B 上, 那么其投影长度就等于向量 A 在 B 上的影子长度。但是, 将要投影到 B 上的向量必须是一个单位向量, 即它的模等于 1。在后面章节中, 我们还会提到投影这个概念。

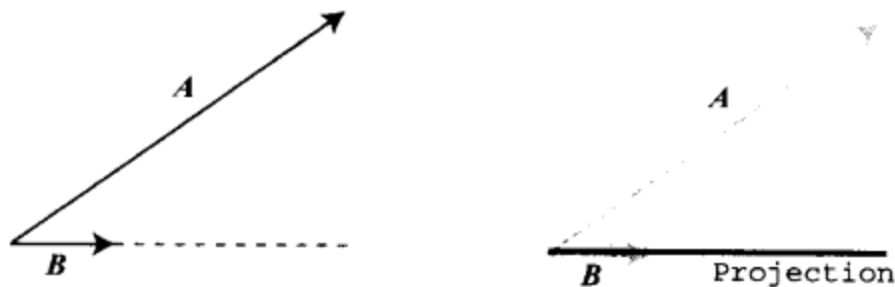


图 4-19 向量 A 在 B 上的投影

向量点乘是一个非常有用的运算, 在后面章节还会被大量使用。在许多应用程序中, 该运算的计算代价也是比较小的, 因此在编程中它会被广泛使用。在下一节中, 我们会介绍另外一种向量相乘运算——叉乘, 通过学习你会发现, 点乘运算会比较简单一些, 然而它们的用途是一样的。

自我测试

1. 已知向量 $A = [-2 \ 8]$, $B = [4 \ 1]$, 计算 $A \cdot B$ 。
2. 已知向量 $C = [-1 \ 4 \ 2]$, $D = [3 \ 0 \ 5]$, 计算 $C \cdot D$ 。
3. 判断第 1 题中的两个向量 A 和 B 是否垂直。
4. 计算第 2 题中向量 C 和 D 之间的夹角。
5. 在游戏中, 假设摄像机的当前位置为 $(0, 0)$, 它的视线方向为向量 $F = [4 \ 9]$, 如果一个物体的位置为 $(3, -2)$, 假设摄像机在某个方向上的最大视角为 90° , 请问该物体是否可见?

4.6 叉乘

在上一节中，我们介绍了向量相乘的一种形式——点乘，本章将介绍另外一种形式——叉乘。这两种运算的主要区别是，点乘的运算结果为一个数值，而叉乘的运算结果为一个向量，让我们来看一下叉乘的计算过程。

注意

在有的数学教材上，两个向量的叉乘被称作向量积，这是因为叉乘运算返回的是一个向量。

叉乘

$$\mathbf{A} \times \mathbf{B} = [(a_2b_3 - a_3b_2) \ (a_3b_1 - a_1b_3) \ (a_1b_2 - a_2b_1)]$$

其中，向量 $\mathbf{A} = [a_1 \ a_2 \ a_3]$ ， $\mathbf{B} = [b_1 \ b_2 \ b_3]$ 。

【例 4-15】 叉乘

计算向量 $\mathbf{A} = [5 \ -6 \ 0]$ 和 $\mathbf{B} = [1 \ 2 \ 3]$ 的叉乘。

解答：

叉乘运算返回另外一个向量，因此可以利用上面公式进行计算。

$$\begin{aligned} \mathbf{A} \times \mathbf{B} &= [(a_2b_3 - a_3b_2) \ (a_3b_1 - a_1b_3) \ (a_1b_2 - a_2b_1)] \\ &= [(-6(3) - 0(2)) \ (0(1) - 5(3)) \ (5(2) - (-6)(1))] \\ &= [(-18 - 0) \ (0 - 15) \ (10 + 6)] \\ &= [-18 \ -15 \ 16] \end{aligned}$$

注意

当计算一些叉乘运算以后，你就会发现它为什么叫作“叉乘”了，一定要注意运算时的正负号，很容易弄错。

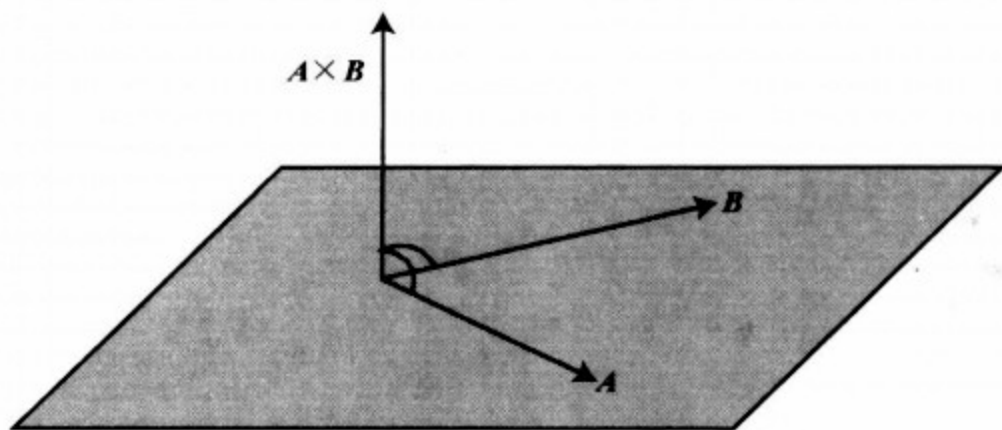
向量叉乘的一个重要性质是，它得到的结果向量分别垂直于两个原始向量，也正是由于这个原因，它仅适用于 3D 向量。

垂直向量

向量 $\mathbf{A} \times \mathbf{B}$ 同时垂直于向量 $\mathbf{A}$ 和 $\mathbf{B}$ 。

下图 4-20 显示说明了该运算，你会发现叉乘运算的结果向量好像有两个方向。

由下图可知，与向量 $\mathbf{A}$ 和 $\mathbf{B}$ 所确定的平面垂直的向量有两个方向：“向上”和“向下”。你可以使用右手法则确定叉乘结果向量的方向，将你的右手腕放到向量 $\mathbf{A}$ 、 $\mathbf{B}$ 的交点处，使手指指向沿着向量 $\mathbf{A}$ 的方向，然后弯曲手指使之转向 $\mathbf{B}$ 的方向，这时大拇指所指的方向即为 $\mathbf{A} \times \mathbf{B}$ 的方向。此时如果交换 $\mathbf{A}$ 、 $\mathbf{B}$ 的次序，那么我们会得到 $\mathbf{B} \times \mathbf{A}$ ，它与 $\mathbf{A} \times \mathbf{B}$ 的方向刚好相反。

图 4-20 $A \times B$ **叉乘不满足交换率**

$$A \times B \neq B \times A$$

实际上, $A \times B = -(B \times A)$ 。

由于叉乘运算可以确定一个向量, 而且该向量还同时垂直于 2 个原始向量, 因此我们可以利用叉乘来计算平面法线。任何两个 3D 向量都可以确定一个平面, 而平面法线就是一个垂直于平面的向量, 而且其长度等于 1。

到目前为止, 我们已经讨论了一些重要运算, 是该建立一个完整 3D 向量类的时候了。注意, 在给向量类取名时尽量不要简单地取作“vector”, 这是因为在 STL 标准模板库中已经存在一个叫作 vector 的类了。下面这段代码可以直接写入一个头文件 (*.h) 中, 它完成了一个 3D 向量类的定义。

```
#ifndef _3DVECTOR_H_
#define _3DVECTOR_H_

class 3Dvector
{
private:
    float x, y, z;
public:
    //功能:构造函数
    //输入:ex——向量的 i 分量
    //      why——向量的 j 分量
    //      zee——向量的 k 分量
    //输出:没有显式输出
    3Dvector(float ex=0, float why=0, float zee=0)
    {
        x=ex; y=why; z=zee;
    }
}
```



```

//功能:析构函数
//输入:无
//输出:无
~3Dvector(){}

//功能:计算向量的模
//输入:没有显式输入
//输出:向量的模
float getMagnitude()
{
    return sqrtf(x*x+y*y+z*z);
}

//功能:操作符重载,向量乘以一个标量
//输入:num——相乘的标量值
//输出:构造的新向量
3Dvector operator* (float num) const
{
    return 3Dvector(x*num, y*num, z*num);
}

//功能:操作符重载,向量乘以一个标量
//输入:num——相乘的标量值
//      vec——相乘向量
//输出:构造的新向量
friend 3Dvector operator* (float num, const 3Dvector& vec)
{
    return 3Dvector(vec.x*num, vec.y*num, vec.z*num);
}

//功能:操作符重载,向量相加
//输入:vec——相加向量
//输出:相加后的结果向量
3Dvector operator+ (const 3Dvector &vec) const
{
    return 3Dvector(x+vec.x, y+vec.y, z+vec.z);
}

//功能:操作符重载,向量相减
//输入:vec——相减向量
//输出:相减后的结果向量
3Dvector operator- (const 3Dvector &vec) const
{
    return 3Dvector(x-vec.x, y-vec.y, z-vec.z);
}

//功能:单位化向量
//输入:无显式输入

```



```

//输出:无
void normalize3Dvector (void)
{
    float mag=sqrtf(x*x+y*y+z*z);
    x /=mag; y /=mag; z /=mag;
}

//功能:向量点乘
//输入:vec——参加点乘运算的向量
//输出:向量的点乘结果
float dot3Dvector (const 3Dvector& vec) const
{
    return x * vec.x+y * vec.y+z * vec.z;
}

//功能:向量叉乘
//输入:vec——参加叉乘运算的向量
//输出:向量的叉乘结果
3Dvector cross3Dvector (const 3Dvector& vec) const
{
    return 3Dvector (y * vec.z - z * vec.y,
                     z * vec.x - x * vec.z,
                     x * vec.y - y * vec.x);
}

};

#endif

```

平面法线

$$\text{平面法线} = (\mathbf{A} \times \mathbf{B}) = \frac{\mathbf{A} \times \mathbf{B}}{\|\mathbf{A} \times \mathbf{B}\|}$$

其中, $\mathbf{A}$ 、 $\mathbf{B}$ 为任意 3D 向量。

【例 4-16】 平面法线

一个由向量 $\mathbf{A} = [5 \quad -2 \quad 0]$ 和 $\mathbf{B} = [1 \quad 2 \quad 3]$ 所确定的平面, 计算该平面的法线以便于确定发生碰撞后的运动方向。

解答:

1. 计算向量 $\mathbf{A}$ 和 $\mathbf{B}$ 的叉乘:

$$\begin{aligned}
 \mathbf{A} \times \mathbf{B} &= [(a_2b_3 - a_3b_2) \quad (a_3b_1 - a_1b_3) \quad (a_1b_2 - a_2b_1)] \\
 &= [(-2(3) - 0(2)) \quad (0(1) - 5(3)) \quad (5(2) - (-2)(1))] \\
 &= [(-6 - 0) \quad (0 - 15) \quad (10 + 2)] \\
 &= [-6 \quad -15 \quad 12]
 \end{aligned}$$

2. 计算向量 $A \times B$ 的模:

$$\|A \times B\| = \sqrt{(-6)^2 + (-15)^2 + 12^2} = \sqrt{36 + 225 + 144} = \sqrt{405} \approx 20.125$$

3. 最后, 单位化向量 $A \times B$:

$$\left[\frac{-6}{20.125} \quad \frac{-15}{20.125} \quad \frac{12}{20.125} \right] \approx [-0.298 \quad 0.745 \quad 0.596]$$

现在, 让我们利用建立的 3Dvector 类来解决该问题:

```
#include <iostream>
using namespace std;
#include "3Dvector.h"

int main()
{
    //定义两个向量
    3Dvector A(5, -2, 0);
    3Dvector B(1, 2, 3);
    //计算叉乘
    3Dvector C=A.cross3Dvector(B);
    //单位化
    C.normalize3Dvector();
    //将结果输出到屏幕上
    cout << C.x << "i " << C.y << "j " << C.z << "k \n";

    return 0;
}
```

在上一节中, 我们知道可以利用点乘计算两个向量间的夹角, 同样也可利用叉乘来计算。如果要对这两种方法进行选择的话, 建议你使用点乘运算, 因为它的运算速度较快一些。但是, 如果已知向量叉乘的话, 那么此时使用叉乘会更容易一些。

两向量间的夹角

$$\|A \times B\| = \|A\| \|B\| \sin\theta$$

其中, A 、 B 为任意的 3D 向量, θ 为它们间的夹角。

【例 4-17】 向量间的夹角

计算向量 $A = [5 \quad -2 \quad 0]$ 和 $B = [1 \quad 2 \quad 3]$ 之间的夹角。

解答:

1. 计算向量 A 和 B 的叉乘 $A \times B$:

$$\begin{aligned} A \times B &= [(a_2b_3 - a_3b_2) \quad (a_3b_1 - a_1b_3) \quad (a_1b_2 - a_2b_1)] \\ &= [(-2(3) - 0(2)) \quad (0(1) - 5(3)) \quad (5(2) - (-2)(1))] \\ &= [(-6 - 0) \quad (0 - 15) \quad (10 + 2)] \\ &= [-6 \quad -15 \quad 12] \end{aligned}$$

2. 计算向量 $A \times B$ 的模:

$$\|A \times B\| = \sqrt{(-6)^2 + (-15)^2 + (12)^2} = \sqrt{36 + 225 + 144} = \sqrt{405}$$

3. 分别计算向量 A 和 B 的模:

$$\|A\| = \sqrt{(5)^2 + (-2)^2 + (0)^2} = \sqrt{25 + 4 + 0} = \sqrt{29}$$

$$\|B\| = \sqrt{(1)^2 + (2)^2 + (3)^2} = \sqrt{1 + 4 + 9} = \sqrt{14}$$

4. 最后, 将上述值代入公式, 求解角度 θ :

$$\|A \times B\| = \|A\| \|B\| \sin\theta$$

$$\sqrt{405} = \sqrt{29} \sqrt{14} \sin\theta$$

$$\frac{\sqrt{405}}{\sqrt{29(14)}} = \sin\theta$$

$$0.99876 \approx \sin\theta$$

$$\theta \approx \sin^{-1}(0.99876)$$

$$\theta = 87.2^\circ$$

现在, 让我们把计算两向量间夹角作为一个函数, 加入先前所写的类 `3Dvector` 中。

```
float angleBetween3Dvectors(const 3Dvector & vec) const
{
    return (acos (dot3Dvector(vec) / (getMagnitude() * vec.getMagnitude()))
            * (180 / PI));
}
```

在本节中, 我们首先定义了向量的叉乘, 请记住——叉乘运算返回的是一个 3D 向量, 而点乘运算返回的仅是一个标量; 然后介绍了向量的单位化运算, 它可用于计算平面的法线; 同样, 叉乘也可用于计算两向量间的夹角, 求解过程与利用点乘求解的过程类似。目前, 我们已经都介绍了向量相乘的两种形式, 在下一节中你会看到这些运算的 C++ 代码实现。

自我测试

1. 已知向量 $A = [2 \ -3 \ 5]$ 和 $B = [-1 \ 0 \ 4]$, 计算 $A \times B$ 。
2. 计算由问题 1 中的向量 A 和 B 所确定平面的法线。
3. 利用叉乘运算, 计算问题 1 中两向量之间的夹角。

4.7 可视化体验

在本书的随书光盘上, 有一个名叫 `Matrix Vortex` 的演示程序, 下面是开发者 Kaveh Kahrizi 对该演示的介绍:

演示程序 `Matrix Vortex` 能够让你从视觉上感受并熟悉矩阵以及向量的相关运算, 所有的数学运算代码包含在类 `MatCalc` 中, 该类分别由文件 `MatCalc.cpp` 和 `MatCalc.h` 进行定义和实现。该演示程序运行的最低配置要求是: 700MHz 以上的处理器, 至少

8MB 内存和 64MB RAM，程序的接口非常直观，即可以通过鼠标左键来导航；输入数据需要使用键盘，其中退格键 Backspace 用于纠正错误的输入。当你输入一个矩阵或者向量，使用 Tab 键或者 Shift-Tab 键选择相应的正方形输入具体数值，按下 Esc 键退出演示程序。

演示程序 Matrix Vortex 能够执行本章介绍的所有向量运算，通过双击文件 Mat_Calc.exe 运行该程序，变量和矩阵的大小会出现在屏幕的左侧。现在不要选择矩阵相关的选项，你将会在下一节中用到它们。

首先，让我们进行向量加法运算。点击“ 3×1 vector”按钮，并输入数值 12, 3 和 -5，

注意使用 Tab 键在各元素间切换。此时点击按钮“Enter”，那么向量 $\begin{bmatrix} 12 \\ 3 \\ -5 \end{bmatrix}$ 将会显示到屏幕的

上方。选择你想要进行的运算操作，此时选择加法“Addition”，然后选择将要与前面向量进行相加的向量大小，请注意只有“ 3×1 Choice”选项是可选择的，这是因为只有同样大小的向量才能相加。点击“ 3×1 vector”按钮，并输入数值 -2, 0 和 4，同样使用 Tab 键在元素间切换，当点击“Enter”按钮后，该向量也将会显示在屏幕上方。请再次检查一下输入的数值

是否正确，确保无误后点击按钮“Solve”，随后向量 $\begin{bmatrix} 10 \\ 3 \\ -1 \end{bmatrix}$ 将会显示在屏幕下方。此时如果要

想进行其他运算，请点击“Clear”按钮以重新开始。

对于自我测试中的所有问题，利用该演示程序 Matrix Vortex 检验你的结果是否正确。若你自己想再构造一些问题，首先在纸上手动计算一下结果，然后再用演示程序去验证。

本章总结

在本章中，我们学习了向量运算的相关知识，它将会在后面的物理知识章节中被用到，而且为物理运动的模拟建立了坚实的基础。本章首先定义了标量和向量，然后介绍了关于向量的几种运算——加法、减法、标量与向量相乘、点乘和叉乘等，并讨论了如何在编程中使用它们，比如可以利用点乘或叉乘计算两向量间的夹角、利用点乘运算判断两向量是否垂直等；随后我们又讨论了向量单位化运算，以及如何去使用单位向量；最后，我们介绍了如何利用叉乘计算平面的法线，在后面物理知识章节中我们还会用到这些概念。

能力测试

选择题

1. 下面哪一个向量？

A. 10 秒

B. 50

C. 20 英里

D. $8\text{m} @ -30^\circ$

2. 如果一个物体从 400 像素的位置移动到 150 像素的位置, 那么它的位移是多少?
A. 250 像素 B. -250 像素 C. -150 像素 D. 550 像素
3. 下面哪个向量与 $30\text{m}@40^\circ$ 最接近?
A. $19\hat{i} + 23\hat{j}$ B. $30\hat{i} + 40\hat{j}$ C. $30\hat{i} - 40\hat{j}$ D. $23\hat{i} + 19\hat{j}$
4. 下面哪个向量与 $150\hat{i} + 200\hat{j}$ 最接近?
A. 250 单位@ 53° B. 250 单位@ 37° C. 350 单位@ 90° D. 350 单位@ 53°
5. 下面哪个向量与 $30\text{m}@40^\circ + 25\text{m}@37^\circ$ 最接近?
A. $43\hat{i} + 34\hat{j}$ B. $23\hat{i} + 19\hat{j}$ C. $55\text{m}@77^\circ$ D. 55m
6. 下面哪个向量与 $20(30\hat{i} - 40\hat{j})$ 最接近?
A. $60\hat{i} - 40\hat{j}$ B. $60\hat{i} + 80\hat{j}$ C. $600\hat{i} - 400\hat{j}$ D. $600\hat{i} + 400\hat{j}$
7. 将向量 $40\hat{i} + 10\hat{j} - 25\hat{k}$ 进行单位化, 结果是下面哪一个?
A. $[0.83 \quad 0.21 \quad -0.52]$ B. $[40 \quad 10 \quad -25]$
C. $[4 \quad 1 \quad -2.5]$ D. $[0.45 \quad 0.72 \quad -0.18]$
8. 计算下列点乘: $[25 \quad 0 \quad -15] \cdot [-1 \quad 23 \quad 4]$, 结果是哪一个?
A. $[-25 \quad 0 \quad -60]$ B. $[24 \quad 23 \quad -11]$
C. -85 D. 36
9. 计算下列叉乘: $[25 \quad 0 \quad -15] \times [-1 \quad 23 \quad 4]$, 结果是哪一个?
A. -85 B. 36 C. $[-25 \quad 0 \quad -60]$ D. $[345 \quad -85 \quad 575]$
10. 计算向量 $[25 \quad 0 \quad -15]$ 和 $[-1 \quad 23 \quad 4]$ 之间的夹角。
A. -15° B. 36° C. 60° D. 97°

练习

练习 1: 向量与标量

1. 向量与标量之间的差别是什么?
2. -65 英尺是一个标量还是一个向量?
3. 25 秒是一个标量还是一个向量?
4. 如果一个运动员在直线跑道上的初始位置在 15ft 标记处, 他跑了一段距离后停在 55ft 标记处, 那么他的位移是多少?
5. 和问题 4 一样, 一个运动员的初始位置为 25ft 标记处, 他跑了一段距离后停在 5ft 标记处, 那么他的位移是多少?
6. 在棒球游戏中, 一个运动员正在跑垒, 他想盗垒上二垒, 经过激烈的跑垒, 最后他终于跑完全垒, 那么他的位移是多少?

练习 2: 极坐标与笛卡儿坐标

1. 向量 A 是一个速度向量, 将 $A = 35\text{m/s}@75^\circ$ 转换成笛卡儿坐标形式。

2. 将向量 $B = 13\hat{i} + 6\hat{j}$ 转换成极坐标形式。
3. 向量 C 是一个力向量, 将 $C = 100\text{N}@40^\circ$ 转换成笛卡儿坐标形式。
4. 将向量 $D = 10\hat{i} - 8\hat{j}$ 转换成极坐标形式。

练习 3: 向量相加和相减

1. 将向量 $A = 30\text{ft}@70^\circ$ 转换成笛卡儿坐标形式。
2. 将向量 $B = 9\hat{i} + 11\hat{j}$ 转换成极坐标形式。
3. 利用问题 1 和 2 中的向量 A 、 B , 计算向量 $A + B$ 。
4. 利用问题 1 和 2 中的向量 A 、 B , 计算向量 $A - B$ 。
5. 已知 3D 向量 $F = 3\hat{i} - 4\hat{j} - \hat{k}$, $G = -7\hat{i} + 5\hat{j} + 8\hat{k}$, 计算向量 $F + G$ 。
6. 已知 3D 向量 $F = 3\hat{i} - 5\hat{j} - \hat{k}$, $G = -6\hat{i} + 3\hat{j} + 6\hat{k}$, 计算向量 $F - G$ 。

练习 4: 标量与向量相乘

1. 标量与向量相乘, 是否会影响向量的模和方向? 请举例说明。
2. 已知向量 $A = 10\text{m/s}@73^\circ$, 计算向量 $(-2/3)A$ 。
3. 已知向量 $B = 5\hat{i} - \hat{j}$, 计算向量 $6B$ 。
4. 单位化向量 $C = [34 \ 20]$ 。
5. 单位化向量 $D = [0 \ 9 \ 27]$ 。

练习 5: 向量点乘

1. 已知向量 $A = [-3 \ 8]$, $B = [5 \ 1]$, 计算 $A \cdot B$ 。
2. 已知向量 $C = [-1 \ 6 \ 3]$, $D = [5 \ 0 \ 7]$, 计算 $C \cdot D$ 。
3. 判断第 1 题中的两个向量 A 和 B 是否垂直。
4. 计算第 2 题中向量 C 和 D 之间的夹角。
5. 在游戏中, 假设摄像机的当前位置为 $(0, 0)$, 它的视线方向为向量 $F = [6 \ 12]$, 如果一个物体的位置为 $(5, -3)$, 假设摄像机在某个方向上的最大视角为 120° , 请问该物体是否可见?

练习 6: 向量叉乘

1. 已知向量 $A = [3 \ -5 \ 7]$ 和 $B = [-1 \ 0 \ 6]$, 计算 $A \times B$ 。
2. 计算由问题 1 中的向量 A 和 B 所确定平面的法线。
3. 利用叉乘运算, 计算问题 1 中两向量之间的夹角。

程序设计

程序设计 1: 编程实现向量在笛卡儿坐标与极坐标形式之间的转换

编写一个程序, 能够在向量的笛卡儿坐标形式与极坐标形式之间进行相互转换。

程序设计 2: 构造一些向量相关的问题

构造一些向量相关的问题, 应该包括向量加法、减法, 标量与向量相乘, 向量点乘和向量叉

乘等，然后利用演示程序 Matrix Vortex 求解这些问题。

程序设计 3：构造向量运算库

建立自己的向量运算库，它应该包括向量加法、减法，标量与向量相乘，向量点乘和向量叉乘等运算。

程序设计 4：编写程序以单位化一个向量

编写一个程序，能够实现向量的单位化。

程序设计 5：编写程序，请求两个向量

1. 编写一个程序，请求输入两个向量，然后计算由它们所确定的平面法线。
2. 编写一个程序，请求输入两个向量，计算它们之间的夹角。

问题讨论

1. 在 C++ 程序中，分别是如何处理标量与向量的？
2. 对于 3D 向量，是否存在与之对应的极坐标形式？（这需要进行一些研究。）
3. 将向量 $25\text{pix}@30^\circ$ 和 $75\text{pix}@45^\circ$ 进行相加之后，结果向量的模为什么不等于 100？
4. 在精确计算向量相加或相减的结果之前，首先采用画图法计算一下有哪些好处？
5. 计算向量在另一个向量上的投影，在哪些地方会更有用？
6. 在游戏中，为什么需要计算两个向量之间的夹角？
7. 当计算两个向量之间的夹角时，哪些时候采用点乘运算会更有效？哪些时候采用叉乘运算会更简便一些？



第5章 矩阵运算

本章目标

认真阅读本章内容并完成练习之后，你能够掌握以下内容：

- 判断两个矩阵是否全等
- 对两个矩阵进行加减运算
- 实现矩阵与常数相乘
- 实现两个矩阵相乘
- 计算转置矩阵

介绍

上一章介绍了关于向量的数学运算。这一章将定义矩阵并介绍矩阵的相关运算。在第6章中你将发现，利用矩阵可以实现2D物体在屏幕上移动，对世界坐标系内的3D物体进行仿射变换。在使用矩阵之前，必须要先弄明白它是如何工作的，如果你对向量操作比较熟悉的话，将对本章矩阵的学习具有很大的帮助。

5.1 相等矩阵

在游戏中，矩阵可以用来以行和列的形式组织一系列数据，你可能会对数组比较熟悉，它就像许多格子一样，用来组织存放任意类型的信息（比如数字、变量、文本以及数组等）。不过，在矩阵中只能储存数字，在上一章我们曾介绍过一个矩阵例子。有时我们会用单行矩阵代表一个向量，实际上矩阵能够具有任意多列和任意多行。在游戏中可以用具有多行多列的矩阵表示物体的所有信息，在该情况下，每行都代表一个单独顶点。作为一个程序员，你应该根据你的需要赋予矩阵每行每列不同的意义，第6章将讨论矩阵的更多应用，在本章首先学习一下关于矩阵的数学运算。

在课本上，我们经常使用中括号[]来表示矩阵，如果你为了省事而用直线代替，那么意义就完全不同了，所以一定要用中括号。在括号内，我们把数字以行和列的形式写入对应位置，每个数字称为矩阵的元素，每个元素可用它所在的行号和列号来确定，来看图5-1所示的矩阵A。

当需要在程序中访问元素3时，取行号0、列号2位置的元素即可。注意，在C++编程中，行和列的索引号都是从0开始的，而不是从1开始。图5-2显示了每个元素在矩阵中的

位置。

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

图 5-1 矩阵 A

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

图 5-2 元素的位置

当在编程中使用矩阵时，需要处理许多问题，比如应用程序是需要固定大小的矩阵，还是要求它会动态改变。如果属于前面的情形，这里具有较多的方法来组织数据，从而完成矩阵的定义，当然此时仅仅是满足使用而已，也就不必再去考虑其扩展性了。下面就是一个具有固定大小的矩阵例子：

```
typedef struct
{
    float index[3][3];
} Matrix3x3;
```

此时若想获得矩阵 A 中的元素 3，就可以使用下面的方法：

```
Matrix3 x3 A;
A.index[0][2];
```

注意

在引用元素时总是先写行号再写列号。

对于另外一种选择，我们可以通过编程将矩阵定义成一个类，该类可以具有任意多的行和列，利用它可以创建具有任意大小的矩阵。很显然，该处理方式具有更好的结构，而且便于扩展和复用，但是它会造成之前的大量计算开销。

如果你决定创建一个矩阵的类，那么就要确定矩阵类的表示形式。它是否包括指向浮点型数组的指针？还是使用 STL 标准模板库中的向量容器来实现？还是使用链表？其实，使用这些数据结构的任何一个都可完成矩阵类的构造，但是每个方法都有其各自的特点。对于一个完整定义的矩阵类来说，它通常会对许多运算符进行重载，从而能够方便地完成各种数学运算，此外还应该提供方便访问矩阵元素的方法。

对于游戏开发而言，大多数图形开发 API 如 DirectX 和 OpenGL，已经封装好了自己的矩阵类，我们可以很方便地拿来使用，从而完成相应的图形编程，但这并不意味着一个优秀程序员可以不用理解矩阵的数学运算规则。由于这本书的重点在于数学，所以我们将使用前面的矩阵讨论矩阵的建立。当我们求解一个问题时，首先应该寻找是否存在捷径能够快速解决问题，然而本章中的示例代码将侧重于理解，而不是效率，因此在实际编程中不要都从本章的实例出发。本章和第 6 章中，所有矩阵的例子都具有固定大小，请参考随书光盘中的程序代码，从而得到关于矩阵类的完整定义，它们大多可以从其类型名字进行区分。

【例 5-1】 矩阵的定义

对于图 5-3，矩阵 B 的维数是多少？应该怎样访问值为 5 的元素？

解答：

1. 观察图 5-3 中的矩阵 B ，它有两行、三列，所以它的维数是 2×3 。
2. 找到元素 5，它位于第二行、第一列，别忘了行、列的索引都是以 0 开始，这就意味着元素 5 的行号是 1，列号是 0。

$$B = \begin{bmatrix} -1 & 4 & -3 \\ 5 & 0 & -2 \end{bmatrix}$$

图 5-3 矩阵 B

对于两个具有相同维数的矩阵，我们把在同行同列的两个元素叫作其对应元素。两矩阵相等的充要条件是：它们具有相同的维数，并且所有的对应元素相同。显然，如果两个矩阵的维数不同，那么它们不可能相等；如果两个矩阵维数相等，但是其对应元素有一个不相等，那么这两个矩阵就不是相等矩阵。

矩阵相等

如果两个矩阵相等，那么：

- 两矩阵具有相等的维数。
- 两矩阵所有的对应元素都相等。

【例 5-2】 判断矩阵是否相等

如图 5-4 所示，矩阵 C 和矩阵 D 是否相等？

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

图 5-4 矩阵 C 和 D

解答：

1. 首先，检查每个矩阵的维数。矩阵 C 是 2×3 矩阵， D 是 3×3 矩阵。
2. 由于两个矩阵的维数不相同，因此两个矩阵不相等。尽管它们中一些对应元素是相等的，但是对于相等矩阵，矩阵的大小必须相等。

【例 5-3】 判断矩阵是否相等

图 5-5 中，矩阵 F 和矩阵 G 是否相等？

$$F = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 8 & 9 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

图 5-5 矩阵 F 和 G

解答：

1. 首先，检查每个矩阵的维数。矩阵 F 是一个 3×3 矩阵， G 也是一个 3×3 矩阵，这是一个好兆头。
2. 现在，检查两矩阵的对应元素，可以发现它们中的大部分是相等的。然而，在 2 行、0 列的对应元素不相等，所以矩阵 F 和矩阵 G 不是相等矩阵。

那么，我们应该怎样在程序中判断两矩阵是否相等呢？大多数对矩阵元素的存取都是以 for 循环的形式进行的，for 循环可以通过索引值依次处理每个元素，这种方法是处理所有矩

阵数学运算的基本方法，请牢记。

下面函数是用来判断两个 3×3 矩阵是否相等：

```
bool areMatricesEqual (Matrix3 x3 a, Matrix3 x3 b)
{
    int errorFlag = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if ((a.index[i][j]) != (b.index[i][j]))
                errorFlag = 1;
        }
    }
    //检测错误标记
    if (errorFlag == 1)
        return false;
    else
        return true;
}
```

上面这个函数，通过循环比较两个矩阵处于相同位置上的元素，一旦检测到其值不相等，就将错误标记 errorFlag 赋值为 1；最后判断该标记，如果为 1 则输出不相等 (false)，否则输出相等 (true)。请参阅一下光盘上本章的编程示例代码，仔细理解该检测过程。你能想出一个更快的检测方法呢？（提示：如果发现任意一个对应元素不相等，那么这两个矩阵就不相等。）

在本节中，我们介绍了什么是矩阵，以及如何去描述矩阵；随后建立了一些处理矩阵的标准和方法，并介绍了如何检测两个矩阵是否是相等的。恭喜你，现在你已经掌握了矩阵的相关名词，下一节我们将介绍矩阵的加减运算。

自我测试

1. 图 5-6 所示矩阵 H 的维数是多少？

$$H = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

图 5-6 矩阵 H

2. 对于图 5-6 中的矩阵 H ，应该怎样引用元素 2？
3. 图 5-7 所示矩阵 J 的维数是多少？

$$J = \begin{bmatrix} 3 & 5 & 15 & -2 & 0 \\ 0 & -3 & 1 & 8 & 2 \\ -1 & 2 & 12 & 4 & 7 \\ 10 & 8 & -5 & 0 & -5 \end{bmatrix}$$

图 5-7 矩阵 J

4. 在图 5-7 中, 矩阵 J 中位于第 2 行、第 3 列的元素是什么?
 5. 如图 5-8 所示, 矩阵 K 和矩阵 L 是否相等?

$$K = \begin{bmatrix} 1 & 2 & 0 & 2 \\ 0 & 1 & -5 & 4 \\ 9 & 0 & 3 & -6 \\ 2 & -3 & 1 & 4 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & -5 \\ 9 & 0 & 3 \end{bmatrix}$$

图 5-8 矩阵 K 和 L

6. 图 5-9 中矩阵 M 和矩阵 N 是否相等?

$$M = \begin{bmatrix} 2 & -3 \\ 0 & 1 \\ -1 & 5 \end{bmatrix} \quad N = \begin{bmatrix} 2 & -3 \\ 0 & 6 \\ -1 & 5 \end{bmatrix}$$

图 5-9 矩阵 M 和 N

5.2 矩阵的加减运算

在上一节中, 我们已经讨论了怎样检测两个矩阵是否相等, 接下来就要讨论矩阵的加减运算。上一节曾提到, 两个矩阵的对应元素位于相同的行和列上, 如果两个矩阵大小相同, 而对应元素也相等, 那么它们肯定相等。其实, 对于两矩阵的相加, 也要求具有相同的维数, 然后将对应元素相加即可。

矩阵加法

对两个大小相同的矩阵相加, 只需将其对应元素值相加即可。

注意

请务必记住, 两个相加矩阵必须要有相同的维数(就像相等矩阵一样)。

一些数学课本把矩阵的维数视为它的行列数, 其实它们是一样的, 当看到该说法时不要困惑。

【例 5-4】 矩阵相加

如图 5-10 所示, 将矩阵 A 和矩阵 B 相加, 即计算 $A + B$ 的值。

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \quad B = \begin{pmatrix} -3 & 9 & 4 \\ 0 & 1 & -1 \\ 5 & -2 & 0 \end{pmatrix}$$

图 5-10 矩阵 A 和 B 相加

解答:

1. 在进行矩阵相加之前, 要确保这两个矩阵的大小相同。在本题中, 矩阵 A 和 B 都是 3×3 矩阵, 因此可以进行相加。

2. 将所有的对应元素相加。比如, 将 0 行、0 列的元素相加, 即 $1 + (-3) = -2$, 其他元素的计算也是如此。

图 5-11 显示了将所有对应元素相加后的最终结果。

下面这段程序代码，说明了如何实现两个矩阵相加：

```
Matrix3 x3 addMatrices (Matrix3 x3 a, Matrix3 x3 b)
{
    Matrix3 x3 temp;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            temp.index[i][j] = (a.index[i][j] + b.index[i][j]);
        }
    }
    return temp;
}
```

上面函数返回矩阵相加后的 3×3 结果矩阵，它分别计算两个矩阵每个位置的元素之和，然后将它们存入一个独立的矩阵，最后将该矩阵返回。查看第 5 章的编程示例，弄明白程序是如何利用该函数实现矩阵相加的。

在这里介绍一个技巧！我们知道，一个 4×4 矩阵有 16 个数据位置，如果进行两个矩阵相加的话，就需要一共输入 32 个数据，很显然这将花费不少功夫。因此，根据用户输入的种子，创建一个随机矩阵是十分有用和快速的，通过随机矩阵就可以验证你写的程序是否正确，下面是一个例子：

```
//根据种子生成一个随机矩阵
Matrix3 x3 createRandom3 x3Matrix(int seed)
{
    Matrix3 x3 temp;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            temp.index[i][j] = rand() % seed;
        }
    }
    return temp;
}
```

利用随机函数并不会生成真正意义上的随机数，这个问题留在以后再仔细讲解。

$$A + B = \begin{bmatrix} -2 & 13 & 11 \\ 2 & 6 & 7 \\ 8 & 4 & 9 \end{bmatrix}$$

图 5-11 矩阵相加 $A + B$ 结果

同理，矩阵减法的运算过程与矩阵加法相似，即把对应元素相减。

矩阵减法

对两个大小相同的矩阵相减，只需将其对应元素值相减即可。

注意

当手动进行矩阵减法运算时，很容易丢掉正负号。因此一个较好的方法是，加上第二个矩阵对应元素的相反数，也就是说首先将第二个矩阵的所有值添上一个负号，然后再与第一个矩阵进行相加。

【例 5-5】 矩阵相减

对于图 5-10，计算矩阵 A 和矩阵 B 相减的结果即 $A - B$ 。

解答：

1. 在进行矩阵相减运算之前，首先确保这两个矩阵大小相同。在本题中，矩阵 A 和 B 都是 3×3 矩阵，因此可以进行减法运算。
2. 将所有的对应元素相减。比如将两个矩阵中的 0 行、0 列的元素相减，也就是 $1 - (-3) = 4$ ，其他元素的计算也是如此。
3. 图 5-12 显示了两个矩阵所有对应元素相减后的结果。

$$A - B = \begin{bmatrix} 4 & -5 & 3 \\ 2 & 4 & 9 \\ -2 & 8 & 9 \end{bmatrix}$$

图 5-12 矩阵相减 $A - B$

在程序代码中，实现矩阵减法的过程与加法一样，下面是两个 4×4 矩阵相减的程序：

```
Matrix4x4 subtractMatrices(Matrix4x4 a, Matrix4x4 b)
{
    Matrix4x4 temp;
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<4; j++)
        {
            temp.index[i][j] = (a.index[i][j] - b.index[i][j]);
        }
    }
    return temp;
}
```

在上面程序中，其运算法则与矩阵加法相同。首先获得两矩阵对应元素的值，然后进行相减运算即可，并将该结果保存到结果矩阵的相应位置。在第 5 章的程序示例代码中，有一个使用该函数的例子，请你自行查看。

再次提醒一下，在进行矩阵相加或相减运算之前，一定要保证参加运算的两个矩阵具有同样的维数。如果在程序中对两个维数不同的矩阵进行运算，那么肯定会出错。另外一个需要注意的地方是，当在进行矩阵减法运算时，减去一个负值很容易出现错误。因此在进行矩阵减法运算时一定要细心，否则出现了错误就不容易进行错误的定位，很有可能会花费你大量的时间。

自我测试

1. 在图 5-13 中, 计算矩阵 C 和 D 相加($C+D$)的结果。

$$C = \begin{bmatrix} 2 & 3 \\ -1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} -5 & 1 \\ 0 & 6 \end{bmatrix}$$

图 5-13 矩阵 C 和 D

2. 在图 5-13 中, 计算矩阵 C 和 D 相减($C-D$)的结果。

3. 在图 5-14 中, 计算矩阵 F 和 G 相加($F+G$)的结果。

$$F = \begin{bmatrix} -2 & 5 \\ 1 & 4 \\ 7 & 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 & 3 \\ 8 & -1 \\ -2 & 6 \end{bmatrix}$$

图 5-14 矩阵 F 和 G

4. 在图 5-14 中, 计算矩阵 F 和 G 相减($F-G$)的结果。

5.3 矩阵与标量相乘

在上一章中, 我们介绍了向量和标量的乘法运算。其实, 向量可以看成是具有单行或单列的矩阵, 因此标量与矩阵相乘和与向量相乘是类似的。事实上, 在上一节矩阵减法运算中, 我们已经使用了矩阵与标量相乘的方法, 还记得当时, 我曾建议大家把第二个矩阵的所有元素取其相反数, 然后再与第一个矩阵相加, 从而完成矩阵相减运算, 其实改变每个元素的符号就相当于让矩阵乘以 -1 。

矩阵与标量相乘

$$cA = \begin{bmatrix} ca_{00} & ca_{01} & \cdot & \cdot & \cdot \\ ca_{10} & \cdot & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ \cdot & & & & ca_{nn} \end{bmatrix}$$

其中, c 为任意常量值, A 为任意大小的矩阵。

从上述公式可以看出, 矩阵与标量相乘就像前面提到的反转每个元素的符号一样, 只需简单的让矩阵每个元素乘以标量值就行了。请注意, 该过程适用于任意类型的矩阵(不仅仅是方形矩阵)和任意标量值(甚至是分数和负数)。

【例 5-6】 标量 · 矩阵

计算 $-5B$, 其中 $B = \begin{bmatrix} 3 & 6 & -4 \\ 0 & -1 & 2 \end{bmatrix}$ 。

解答:

将矩阵的每个元素都乘以 -5 , 可以得到:

$$-5B = \begin{bmatrix} -15 & -30 & 20 \\ 0 & 5 & -10 \end{bmatrix}$$

进行矩阵与标量的相乘，这只是使用矩阵的第一步，以后会进行更好地运算。下面这个函数说明了标量与矩阵是如何实现相乘的：

```
Matrix3x3 scalarMultiply(Matrix3x3 a, float scale)
```

```
{
    Matrix3x3 temp;
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            temp.index[i][j] = (a.index[i][j]*scale);
        }
    }
    return temp;
}
```

由上述函数可知，首先通过索引获得矩阵每个元素值，然后用指定的标量乘以该元素。到目前为止，你已经明白了如何处理矩阵运算，在本章的示例程序中，有关于标量和矩阵相乘的程序例子，请自行分析查看。

在前面几节中，我们已经学习了矩阵的加减运算，以及矩阵乘上或除以一个标量，而且也能够进行这些运算的混合运算，其实这些运算规则与单一运算是一样的，唯一不同的是，进行矩阵相加或相乘的对象不是单一的数字而已。

【例 5-7】 矩阵方程

给出矩阵 $A = \begin{bmatrix} 4 & 5 \\ 1 & -2 \end{bmatrix}$ 、 $B = \begin{bmatrix} 0 & -1 \\ -3 & 2 \end{bmatrix}$ ，计算矩阵 X ，使得 $2X = 3A - B$ 。

解答：

1. 观察方程 $2X = 3A - B$ ，要想求出矩阵 X ，那么就需要使用代数方法，也就是将方程的两边同乘以 $1/2$ (或除以 2)。

$$2X = 3A - B$$

$$\frac{1}{2}(2X) = \frac{1}{2}(3A - B)$$

$$X = \frac{1}{2}(3A - B)$$

2. 和数字运算的顺序一样，首先计算括号中的 $(3A - B)$ ：

$$\begin{bmatrix} 12 & 15 \\ 3 & -6 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 3 & -2 \end{bmatrix} = \begin{bmatrix} 12 & 16 \\ 6 & -8 \end{bmatrix}$$

3. 最后，将矩阵 $(3A - B)$ 中的每一元素都乘以 $1/2$ 就可以了。

$$\frac{1}{2}(3A - B)$$

$$\begin{bmatrix} 6 & 8 \\ 3 & -4 \end{bmatrix} = X$$

至此，我们已经讨论了矩阵的加减运算以及矩阵和标量的乘法运算，这使我们可以像处理数字一样用代数的方法解决方程中的矩阵问题。下一个问题就是应该怎样进行矩阵间的乘法运算呢？别急，我们将在下一节中详细介绍。

自我测试

1. 已知 $A = \begin{bmatrix} -2 & 3 & 5 \\ 0 & 1 & -4 \\ 10 & 0 & -1 \end{bmatrix}$ ，计算 $-3A$ 。

2. 已知 $B = \begin{bmatrix} 6 & -8 & -2 \\ 0 & 10 & -4 \\ 12 & 0 & -16 \end{bmatrix}$ ，计算 $\frac{1}{2}B$ 。

3. 利用问题 1 和 2 中的矩阵 A 和矩阵 B ，计算矩阵 X 使得 $-\frac{1}{4}X = \frac{1}{2}B - 3A$ 。

5.4 矩阵相乘

在上一节中，我们介绍了标量和矩阵的乘法运算，本节将研究两个矩阵之间的相乘，该过程将会涉及到点乘的知识，因此在学习本节内容之前，你应该复习一下第 4 章“向量运算”中的点乘知识。

你可以把矩阵之间相乘看成一系列的点乘，请记住点乘的两个方面：

- 参加点乘运算的两个向量，必须具有相同数目的元素。
- 点乘的结果将输出一个标量（一个数字）。

让我们以 2×2 矩阵为例，进行相乘的矩阵不必是方阵，其维数也不一定非要相同，但是它也是有限制的，这等一会儿再介绍。现在介绍两个 2×2 矩阵相乘，得到的是另一个 2×2 矩阵。

两个 2×2 矩阵相乘

$$AB = \begin{bmatrix} (a_{00}b_{00} + a_{01}b_{10}) & (a_{00}b_{01} + a_{01}b_{11}) \\ (a_{10}b_{00} + a_{11}b_{10}) & (a_{10}b_{01} + a_{11}b_{11}) \end{bmatrix}$$

其中，矩阵 $A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ ， $B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$ 。

对于上述公式，初看你可能会摸不着头脑，但是仔细观察第 0 行、0 列的元素，是不是看起来很熟悉？是的，它是 A 的第 0 行元素与 B 的第 0 列元素点乘的结果。

现在来看第0行、第1列的元素，它是 A 的第0行元素与 B 的第1列元素点乘的结果，对于其他两个元素，具有类似的规律。

每个元素都可由它的位置来决定，也就是说结果矩阵中的某个元素值是第一个矩阵的对应行上的元素与第二个矩阵对应列上的元素点乘的结果。

【例5-8】两个 2×2 矩阵相乘

已知矩阵 $A = \begin{bmatrix} 4 & 5 \\ 1 & -2 \end{bmatrix}$ ，矩阵 $B = \begin{bmatrix} 0 & -1 \\ -3 & 2 \end{bmatrix}$ ，求 AB 。

解答：

$$\begin{aligned} AB &= \begin{bmatrix} (a_{00}b_{00} + a_{01}b_{10}) & (a_{00}b_{01} + a_{01}b_{11}) \\ (a_{10}b_{00} + a_{11}b_{10}) & (a_{10}b_{01} + a_{11}b_{11}) \end{bmatrix} \\ &= \begin{bmatrix} 4(0) + 5(-3) & 4(-1) + 5(2) \\ 1(0) - 2(-3) & 1(-1) - 2(2) \end{bmatrix} \\ &= \begin{bmatrix} (0 - 15) & (-4 + 10) \\ (0 + 6) & (-1 - 4) \end{bmatrix} \\ &= \begin{bmatrix} -15 & 6 \\ 6 & -5 \end{bmatrix} \end{aligned}$$

上面例子进行了4次点乘运算——对每一个元素都进行一次。还记得点乘的第一个条件吗？即进行点乘的两个向量必须有相同数量的元素，这就意味着矩阵的行列数已经确定，即第一个矩阵的行数必须等于第二个矩阵的列数，这是确保第一个矩阵行中的元素数目与第二个矩阵的列中的元素数目相等的唯一方法。

矩阵相乘是否有意义

如果矩阵 A 和 B 相乘有意义，那么 A 的列数必须等于 B 的行数。

如果通过上面结论检测出矩阵相乘有意义，那么下一步就要找出结果矩阵的维数大小，并计算出结果。如果两个矩阵相乘有意义，那么其结果矩阵的维数大小可以由下面结论得到：

结果矩阵的维数大小

如果矩阵乘积 AB 有意义，那么 AB 的行数等于 A 的行数， AB 的列数等于 B 的列数。

综合应用这两条规则也比较容易，下面用一种直观的方法说明该问题，称作可视化法。如果写出两个矩阵的维数，如图5-15所示，那么里面的两个数字必须相等。

$$AB = \begin{bmatrix} 4 & 3 & -2 \\ -1 & 0 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$2 \times 3 \quad 3 \times 1$

图5-15 判断矩阵相乘是否有意义

如果里面的两个数字不相等，计算就应该停止，其相乘结果是没有意义的；如果它们相等，下面就可以对这两个矩阵进行乘法运算了。下一步要找出结果矩阵的维数大小，这可以方便地从外面的两个数得到，它们就是结果矩阵的维数。在图 5-15 中，你会发现里面的两个数都是 3，因此相乘结果是有意义的，进而结果矩阵的维数大小是 2×1 ，这是通过外面的两个数得到的。一旦知道了结果矩阵的大小，余下的工作就是做一系列点乘运算了。在这个例子中，结果矩阵仅有两个元素，所以我们仅需进行两次点乘运算就行了，图 5-16 显示了最终的结果。

$$AB = \begin{bmatrix} 4 & 3 & -2 \\ -1 & 0 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 4(1) + 3(2) - 2(3) \\ -1(1) + 0(2) + 5(3) \end{bmatrix} = \begin{bmatrix} 4 + 6 - 6 \\ -1 + 0 + 15 \end{bmatrix} = \begin{bmatrix} 4 \\ 14 \end{bmatrix}$$

图 5-16 矩阵 A 和 B 相乘的结果

【例 5-9】 两个不同大小的矩阵相乘

用图 5-15 中的两个矩阵计算结果矩阵 BA。

解答：

1. 在进行两个矩阵相乘之前，检查相乘结果是否有意义，在这里用可视化法写出每个矩阵的维数。如图 5-17 所示。

$$AB = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 4 & 3 & -2 \\ -1 & 0 & 5 \end{bmatrix}$$

$3 \times 1 \quad 2 \times 3$

图 5-17 矩阵相乘 BA

2. 检查里面的两个数字，在这个例子中 $2 \neq 1$ ，这就意味着 B 的列数 $\neq$ A 的行数，因此相乘是没有意义的，故就不能计算 BA。

记得前面我们计算 AB 时，它是存在的，结果是一个 2×1 矩阵。然而当你改变相乘顺序，尝试用 B 乘以 A 时，它是没有结果的。这就说明矩阵间的乘法不满足交换律。

矩阵相乘不满足交换率

$$AB \neq BA$$

其中 A、B 为任意大小的矩阵。

在有些情况下，当你改变矩阵的相乘顺序，其相乘结果都是存在的（并且还有可能会相等）。然而在大多数情况下，你将会得到一个完全不同的结果。例如，再次看一下例 5-8，当 A 乘以 B 时，得到如下结果：

$$AB = \begin{bmatrix} -15 & 6 \\ 6 & -5 \end{bmatrix}$$

现在计算 B 乘以 A，你会发现将会得到不同的结果。

$$\begin{aligned}
 BA &= \begin{bmatrix} 0 & -1 \\ -3 & 2 \end{bmatrix} \begin{bmatrix} 4 & 5 \\ 1 & -2 \end{bmatrix} \\
 &= \begin{bmatrix} 0(4) - 1(1) & 0(5) - 1(-2) \\ -3(4) + 2(1) & -3(5) + 2(-2) \end{bmatrix} \\
 &= \begin{bmatrix} 0 - 1 & 0 + 2 \\ -12 + 2 & -15 - 4 \end{bmatrix} \\
 &= \begin{bmatrix} -1 & 2 \\ -10 & -19 \end{bmatrix}
 \end{aligned}$$

在该例中， AB 和 BA 都是有意义的，但是它们的结果却是完全不同的。

矩阵间的乘法比矩阵与标量间的乘法复杂一些，请记住下面三个计算步骤：

1. 首先检查相乘结果是否存在(两个内项数是否相等)。
2. 如果有结果，找出它的维数(两个外项数)。
3. 逐次计算各个元素值。根据元素的位置，将第一个元素对应行与第二个元素对应列做点乘运算。

在程序代码中进行矩阵相乘，同样也要遵循这些原则。相乘结果必须要有意义，在进行乘法之前确保相乘结果有意义是程序员的责任。

下面是两个 3×3 矩阵相乘的函数：

```

Matrix3 x3 multiply3x3Matrices(Matrix3 x3 a, Matrix3 x3 b)
{
    Matrix3 x3 temp = createFixed3x3Matrix(0);
    for (int i=0; i<3; i++)
    {
        for (int j=0; j<3; j++)
        {
            for (int k=0; k<3; k++)
            {
                temp.index[i][j] += (a.index[i][k] * b.index[k][j]);
            }
        }
    }
    return temp;
}

```

在使用这个函数之前，你应该明白它的程序逻辑，也就是能够逐步跟踪矩阵值的计算过程。函数中的三重循环可以确保完成点乘运算，在程序中正确应用该函数可以减少计算出错的机会。

在 3D 游戏中，通常使用矩阵乘法来完成物体的旋转等变换。假设该相乘过程发生在游戏中，为了适应 3D 游戏的实时需要，你能否想到另外一种更快的处理方法呢？

对于两个不同大小的矩阵相乘，其函数过程与上面函数稍有不同：


```

Matrix3 x1 multiplyMatrixN x M(Matrix3 x3 a, Matrix3 x1 b)
{
    Matrix3 x1 temp;
    temp.index[0] = 0.0f;
    temp.index[1] = 0.0f;
    temp.index[2] = 0.0f;
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            temp.index[i] += (a.index[i][j] * b.index[j]);
        }
    }
    return temp;
}

```

注意在上述函数中，我们只乘以一个 3×1 矩阵。请注意在开始计算之前，要将矩阵初始化为 0，该初始化步骤并没有在以前的代码中列出，在这里只是为了节省空间，在你的程序中一定不要忘了这一步。当进行矩阵乘法时，一定要记住将临时矩阵进行初始化，本章的示例代码包含了上面两个有用的函数。

这里还有另外一个有用的函数 `printMatrix()`，它用来输出矩阵中包含的信息，该主意非常不错，当在进行程序调试的时候，它会显得非常有用。下面列出了一个输出矩阵的示例，同时在本章的示例代码中也使用了它。

```

void printMatrix(Matrix4 x4 a)
{
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<4; j++)
        {
            float temp = a.index[i][j];
            cout << temp << " ";
        }
        cout << "\n";
    }
}

```

注意本章所讲的例子使用的都是普通矩阵。其实，我们可以给矩阵的相应行或列附上有用的信息，就像向量一样。下一章我们将会给矩阵的行或列附上特殊信息，从而完成对物体的移动或旋转等。

当你使用上述方法编写一些矩阵运算程序之后，一定要检验它们能否正确工作，还有当它们被用于新的数据时还能否保持正确，尝试在纸上手动计算一下结果，与程序结果进行对比，看它们是否一致。如果不相等，要找出到底是哪个错了。当你花了很长时间而没有找到程序的 Bug 时，此时可以手动演算一下，确定出错的具体位置，请不要完全依赖于在机器上验证。

一个重要的概念是，计算机能够通过执行算法帮我们解决类似的问题，因此在具体编程之前一定要理解算法的思想。

在进入第6章学习之前，请一定要仔细分析本章所附带的示例代码，理解它包含的矩阵运算库，这些知识将还会在下一章中广泛应用。

自我测试

对于下面的几组矩阵，判断其结果是否存在。如果存在，请计算出结果矩阵。

$$1. \begin{bmatrix} 3 & 7 \\ -2 & 0 \end{bmatrix} \begin{bmatrix} 0 & -5 \\ 1 & 4 \end{bmatrix}$$

$$2. \begin{bmatrix} 0 & -5 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 3 & 7 \\ -2 & 0 \end{bmatrix}$$

$$3. \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \begin{bmatrix} 4 & 0 \\ -1 & 3 \end{bmatrix}$$

$$4. \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \\ -4 & 5 & 10 \end{bmatrix}$$

$$5. \begin{bmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \\ -4 & 5 & 10 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$6. \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 1 & 5 & 1 \\ -1 & 3 & -2 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 5 \\ 3 & -1 \\ 4 & 0 \end{bmatrix}$$

5.5 矩阵转置

最后要介绍的一个矩阵运算是矩阵转置，该操作将在第6章讨论 OpenGL 和 DirectX 的不同时会应用到。转置是一个简单而又有用的操作。到目前为止，我们都使用行和列的位置来引用一个元素，转置操作其实很简单，即只需交换每个元素的行和列，转置可适用于任意大小的矩阵中。我们首先来看 3×3 矩阵的转置，以便你能够快速清楚该运算。

3×3 矩阵的转置

$$\text{如果 } A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}, \text{ 那么 } A^T = \begin{bmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{bmatrix}。$$

注意

上标 T 代表转置，即矩阵 A 的转置可写成 A^T 。

【例 5-10】 计算 3×3 矩阵的转置

已知矩阵 $B = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ ，求 B^T 。

解答：

1. 矩阵转置就是将每个元素的行和列互换。我们每次处理一个元素，元素 1 在第 0 行、第 0 列，所以当你转换行列时，它仍然在第 0 行、第 0 列；再如元素 2 在第 1 行、第 0 列，转换后变成第 0 行、第 1 列；同理，元素 3 由第 2 行、第 0 列变成第 0 行、第 2 列。

2. 如果你将所有的元素进行转置之后，你会发现所有的元素沿着对角线进行了翻转，而对角线上的行数等于列数，不发生转换。转置后的结果矩阵是这样的：

$$B^T = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

从上例注意到，一个 3×3 矩阵转置后得到的仍是一个 3×3 矩阵。如果原始矩阵不是方阵，那么转置后矩阵的大小就会改变，这是因为矩阵的行列互换了。这就意味着一个 3×1 矩阵转置后变为一个 1×3 矩阵，其实转置任意一个向量时的情形都是一样的。

【例 5-11】 向量的转置

已知 $C = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$ ，求 C^T 。

解答：

1. 转置就是将每个元素的行和列互换。让我们每次处理一个，元素 4 在 0 行、0 列，因此当你转换行列时，它仍然在 0 行、0 列；元素 5 在 1 行、0 列，转换后变成 0 行、1 列；同理，元素 6 由 2 行、0 列变成 0 行、2 列。

2. 转置结果是一个单行向量：

$$C^T = [4 \ 5 \ 6]$$

再次说明，转置可以应用于任意大小的矩阵。只需要记住，矩阵转置后会改变矩阵的维数和每个元素的行列位置。

矩阵的转置

对于任意大小的矩阵，元素 a_{mn} 在矩阵 A^T 中变为 a_{nm} 。

【例 5-12】 计算转置矩阵

已知 $D = \begin{bmatrix} 2 & -1 & 3 & 8 \\ 0 & 5 & 1 & 4 \end{bmatrix}$ ，求 D^T 。

解答:

1. 转置就是将每个元素的行和列互换。让我们每次处理一个, 元素 2 在 0 行、0 列, 所以当你转换行列时, 它仍然在 0 行、0 列; 元素 0 在 1 行、0 列, 转换后变成 0 行、1 列, 注意 0 和 -1 互换了位置。

2. 继续每次转置一个元素。你可能注意到, 第一行变成第一列, 第二行变成第二列。

3. 转置后的结果如下:

$$D^T = \begin{bmatrix} 2 & 0 \\ -1 & 5 \\ 3 & 1 \\ 8 & 4 \end{bmatrix}$$

矩阵转置的运算非常简单, 让我们看看程序中如何实现一个 4×4 矩阵的转置:

```
Matrix4x4 transpose4x4Matrix(Matrix4x4 a)
{
    Matrix4x4 temp;
    for(int i=0;i<4;i++)
    {
        for(int j=0;j<4;j++)
        {
            temp.index[i][j] = a.index[j][i];
        }
    }
    return temp;
}
```

只要找到一种交换行列的方法, 相应地就能很快地完成矩阵的转置运算。此时你可能会想去编写一个转置程序, 你所需要的就是一种转换行列的系统化方法而已。在下一章中我们将再次用到该运算, 到时你将明白它在程序中是多么有用。

自我测试

求下列每个矩阵的转置矩阵。

1. $\begin{bmatrix} 7 & 13 \\ 0 & 10 \end{bmatrix}$

2. $\begin{bmatrix} 0 & -5 \\ 1 & 4 \end{bmatrix}$

3. $\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$

$$4. \begin{bmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \\ -4 & 5 & 10 \end{bmatrix}$$

$$5. \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 1 & 5 & 1 \\ -1 & 3 & -2 & 0 \end{bmatrix}$$

$$6. \begin{bmatrix} 1 & 2 \\ 2 & 5 \\ 3 & -1 \\ 4 & 0 \end{bmatrix}$$

5.6 可视化体验

在本书的随书光盘上，有一个名叫 Matrix Vortex 的演示程序，下面是开发者 Kaveh Kahrizi 对该演示的介绍：

演示程序 Matrix Vortex 能够让你从视觉上感受并熟悉矩阵以及向量的相关运算。所有的数学运算代码包含在类 MatCalc 中，该类分别由文件 MatCalc.h 和 MatCalc.cpp 进行定义和实现。该演示程序运行的最低配置要求是：700MHz 以上的处理器，至少 8MB 内存和 64MB RAM，程序的界面非常直观，即可以通过鼠标左键来导航；输入数据需要使用键盘，其中退格键 Backspace 用于纠正错误的输入。当你输入一个矩阵或者向量，使用 Tab 键或者 Shift-Tab 键选择相应的正方形输入具体数值，按下 Esc 键退出演示程序。

演示程序 Matrix Vortex 能够执行本章介绍的所有矩阵运算。通过双击文件 Mat_Calc.exe 运行该程序，然后变量和矩阵的大小会出现在屏幕的左侧。

假如现在进行矩阵乘法，即 3×3 矩阵乘上一个 3×1 矩阵，请记住矩阵相乘的次序是非常重要的。在该情形下， 3×3 矩阵必须要首先指定，通过使用 Tab 键在元素之间切换，并输入下面的元素值：

$$\begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 6 \\ 0 & 0 & 1 \end{bmatrix}$$

当你单击“Enter”按钮，该矩阵将会显示到屏幕上方。

在屏幕上单击你所看到的操作按钮，在这里就是矩阵乘法“Multiplication”；然后点击要与前一矩阵相乘的矩阵，我们选择 3×1 向量“ 3×1 vector”。要注意不是所有的选项都能被选，你只能选择能够与当前矩阵相乘的选项。输入如下元素值，同样地使用 Tab 键在各元素间切换：

$$\begin{bmatrix} 12 \\ 3 \\ 1 \end{bmatrix}$$

当点击“Enter”按钮，第二个矩阵将会显示到屏幕上方，请再检查一下你输入的元素值，如果都正确的话，就点击“Solve”按钮，随后向量：

$$\begin{bmatrix} 17 \\ 9 \\ -1 \end{bmatrix}$$

将会显示到屏幕的下方，此时你若还想进行另外的矩阵运算，就请点击“Clear”按钮。

对于自我测试中的问题，利用 Matrix Vortex 演示程序来检查一下你的答案。你可能会想自己设计一些问题，首先手动计算一下结果，然后再用 Matrix Vortex 程序来验证一下。

本章总结

在这一章中，我们由向量运算扩展到矩阵运算。首先对矩阵进行了定义，然后设计了一些代码。然后我们学习了以下矩阵运算：相等矩阵、矩阵的加减运算、矩阵与标量的乘法运算、矩阵间的乘法运算和转置，其中也讨论了矩阵中的四则运算顺序。最后介绍了这些运算操作在 C++ 中的具体应用。在下一章中，我们将会使用这些操作对物体进行变换操作，从而实现物体在屏幕上的移动、缩放和旋转等。

能力测试

选择题

1. 判断正确还是错误。下面两个矩阵是相等的。

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \text{ 和 } \begin{bmatrix} 1 & 3 & 0 \\ 2 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

A. 正确

B. 错误

2. 对于矩阵 $A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ ，矩阵 $A[2][0]$ 所对应的元素值是多少？

A. 1

B. 2

C. 3

D. 4

3. 判断正确还是错误。矩阵乘法满足交换率。

A. 正确

B. 错误

4. 下列两个矩阵能否相加？

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \text{ 和 } \begin{bmatrix} 5 & 3 & 0 \\ 6 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

A. 能

B. 不能

5. 对于下列矩阵 A , 矩阵 $-5A$ 对应于哪个选项?

$$A = \begin{bmatrix} -3 & 1 \\ 0 & -5 \\ 6 & 20 \end{bmatrix}$$

A. $\begin{bmatrix} -15 & 5 \\ 0 & -25 \\ 30 & 100 \end{bmatrix}$

B. $\begin{bmatrix} -8 & -4 \\ -5 & -10 \\ 1 & 15 \end{bmatrix}$

C. $\begin{bmatrix} -4 & 15 \\ -2 & -1 \\ 0 & 12 \end{bmatrix}$

D. $\begin{bmatrix} 15 & -5 \\ 0 & 25 \\ -30 & -100 \end{bmatrix}$

6. 下面两个矩阵相乘后的结果矩阵的维数是多少?

$$\begin{bmatrix} 15 & -5 \\ 0 & 25 \\ -30 & -100 \end{bmatrix} \times \begin{bmatrix} -20 \\ 3 \end{bmatrix}$$

A. 2×1

B. 3×1

C. 2×3

D. 没有意义

7. 判断正确还是错误。只有方形矩阵(其行数等于列数)时才能进行转置运算。

A. 正确

B. 错误

练习

练习 1: 矩阵的加减运算

1. 写出矩阵相加的步骤。
2. 写出矩阵相减的步骤。

练习 2: 矩阵间的乘法运算

1. 写出两个不同大小矩阵的相乘步骤。
2. 写出矩阵间相乘的三个步骤。

练习 3: 矩阵和向量的转置

1. 写出矩阵转置的步骤。
2. 写出向量转置的步骤。

程序设计

程序设计 1: 构造一些矩阵运算问题

构造一系列矩阵运算问题, 包括矩阵的加法/减法运算、标量乘以矩阵、矩阵间相乘和矩阵转置, 然后利用 Matrix Vortex 演示程序计算这些问题的解。

程序设计 2: 判断两个矩阵能否相乘

编写一个程序, 输入两个任意大小的矩阵, 然后返回它们能否进行相乘运算。

程序设计 3：建立一个矩阵运算函数库

建立你自己的矩阵运算函数库，其中包括矩阵加法/减法运算、标量与矩阵相乘、矩阵间的相乘和矩阵转置等运算。

问题讨论

1. 建立一个标准位置有哪些好处？矩阵和数组的主要区别是什么？
2. 为什么矩阵的乘法不满足交换率？
3. 对于本章中的示例代码，能够进行哪些优化？
4. 既然目前大部分图形 API 都已经包含了各自的矩阵类型，那为什么还要学习矩阵的数学运算呢？



第6章 变 换

本章目标

学习完本章，并完成后面的练习之后，你将能够完成以下工作：

- 利用矩阵加法对物体进行平移变换
- 利用矩阵乘法对物体进行平移变换
- 利用矩阵乘法对物体进行缩放变换
- 利用矩阵乘法对物体进行旋转变换
- 利用矩阵乘法对一系列变换进行组合

介绍

在上一章中我们介绍了矩阵的相关运算，接下来就来看一下它在游戏编程中的具体应用：仿射变换。变换是指在空间中能让物体发生运动的操作的总称，包括前后移动、上下移动、缩小放大和旋转等；仿射是指物体在变换之后应该保持先前的形状。本章首先介绍二维空间中的变换，然后将其扩展到三维空间中，最后你将能够仅利用一个矩阵即可完成任何形式的变换。如果在此学习过程中，你对有些矩阵运算知识不太清楚，请回顾一下第5章矩阵运算中的相关知识。

6.1 平移

让我们以能在2D屏幕上进行上、下、左、右运动的简单例子开始学习吧，如果物体是沿上述方向进行运动，那么该运动就叫作平移。物体的平移运动既可以由矩阵的加法来完成，也可以利用矩阵的乘法来实现，如果对物体进行的仅是平移变换的话，建议你采用矩阵加法来完成，因为它是目前最快、最容易的方法。然而，如果你还想在同一帧中对物体进行缩放和旋转变换的话，就应该用矩阵乘法来实现。下面让我们首先来学习采用矩阵加法的方法。

对于在笛卡儿坐标系下工作的朋友来说，大家都会有一种使用矩阵加法实现平移运动的直观感觉。假设一个物体的位置在点 $P(1, 2)$ 处，现在要想让它向右平移3个单位、向上平移1个单位，该如何实现呢？如果将其 x 坐标值加上3、 y 坐标加上1，那么将确定一个新位置 $(1+3, 2+1) = (4, 3)$ ，使用 P' 表示该位置，那么物体在经过3个单位右移、1个单位

上移之后, 它将位于 $P'(4, 3)$ 处。

对于上述方法而言, 如果仅仅对一个点而不是一系列点进行平移的话, 它是很简便的。但是对于游戏中的具体模型, 它们可能包含了几百甚至几千个顶点, 这该怎么办呢? 因此我们需要一种系统的方法来对所有顶点进行处理, 如将其 x 坐标加上 3、 y 坐标加上 1, 你可能会使用循环, 对每一个顶点执行如下操作:

$$x' = x + 3$$

$$y' = y + 1$$

该操作可以写成矩阵的形式:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

请记住, 当对矩阵进行加法运算时仅仅是将对应元素相加, 因此参加运算的两个矩阵的结构必须是相同的。在程序中使用矩阵来组织数据是非常高效的, 你可以建立一个循环结构, 其输入为顶点的原始位置, 然后返回顶点的新位置。

在实际的编程过程中, 你可能不会每次都知知道物体在 x 和 y 方向上平移的具体数值, 这时你可能会在等待用户的输入来决定将物体移动到何处。这种方式很好, 我们首先建立一个通用的矩阵方程, 然后在读取用户输入后尽快加上该数值, 即可解决该问题。

以前在本书中使用符号 Δ 代表平移量, 但是键盘上没有对应的按键, 因此很多编程者都使用 dx 和 dy 分别代表在 x 、 y 方向上的变化量, 由此可以得出矩阵加法的通用形式:

利用矩阵加法实现 2D 平移

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix}$$

注意

dx 和 dy 的取值并没有被限制为正数。如果 dx 的值为负, 那么意味着物体向左而不是向右移动; 同样地, dy 取负值意味着物体向下而不是向上移动。

【例 6-1】 矩阵加法实现 2D 平移

建立一个矩阵方程, 能将一个 2D 物体在屏幕上向右移动 50 个像素, 向下移动 100 个像素, 然后利用该方程平移一个三角形, 其顶点分别为: $A(20, 30)$ 、 $B(0, 200)$ 和 $C(300, 400)$ 。

解答:

1. 建立矩阵方程。为了能够让物体向右移动 50 个像素, 向下移动 100 个像素, 因此 dx 取值为 50, dy 取值为 -100 (负值意味着向下移动):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 50 \\ -100 \end{bmatrix}$$

2. 将原始顶点值代入, 并与矩阵方程相加, 得出平移后的新位置。首先, 将顶点 A 的原始值 (20, 30) 代入:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 20 \\ 30 \end{bmatrix} + \begin{bmatrix} 50 \\ -100 \end{bmatrix} = \begin{bmatrix} 70 \\ -70 \end{bmatrix}$$

得出顶点 A 的新位置 $A'(70, -70)$ 。

3. 对顶点 B 和 C 重复步骤 2, 分别得出其新位置 $B'(50, 100)$ 和 $C'(350, 300)$ 。通过移动这三个顶点, 就移动了整个三角形。

4. 图 6-1 显示了该平移过程, 其中三角形的原始位置用点画线标出, 新位置用实线标出。

在前面例 6-1 中, 我们平移了一个简单的三角形。但是别忘了, 游戏中的模型大都包含许多顶点, 很多多边形模型包括了几百个(或几千个)三角形, 但是只要我们知道怎样移动一个三角形, 那么就可以移动许多个。

同样地, 也可以通过矩阵加法对 3D 空间中的物体进行平移, 只不过要增加一个 z 轴坐标。

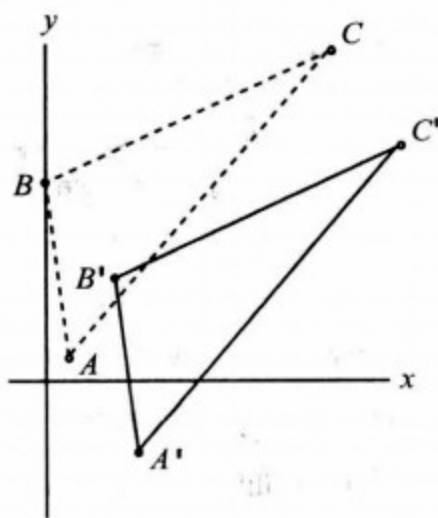


图 6-1 平移后的三角形 ABC

利用矩阵加法实现 3D 平移

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$$

【例 6-2】 矩阵加法实现 3D 平移

建立一个矩阵方程, 能将一个 3D 物体在屏幕上向左移动 100 个像素, 向上移动 200 个像素, 向后(屏幕里面)移动 50 个像素。然后利用该方程平移一个三角形, 其顶点分别为: $A(40, 0, 100)$ 、 $B(0, 350, 200)$ 和 $C(-100, 200, -10)$ 。

解答:

1. 建立矩阵方程。为了能够让物体向左移动 100 个像素, 向上移动 200 个像素, 向后移动 50 个像素, 因此 $dx = -100$, $dy = 200$, $dz = -50$ (如果你记不清楚哪个方向是取正值还是负值, 就去复习一下第 1 章):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} -100 \\ 200 \\ -50 \end{bmatrix}$$

2. 将原始顶点值代入, 并与矩阵方程相加, 得出平移后的新位置。首先, 将顶点 A 的原始值 $(40, 0, 100)$ 代入:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 40 \\ 0 \\ 100 \end{bmatrix} + \begin{bmatrix} -100 \\ 200 \\ -50 \end{bmatrix} = \begin{bmatrix} -60 \\ 200 \\ 50 \end{bmatrix}$$

得出顶点 A 的新位置 $A'(-60, 200, 50)$ 。

3. 对顶点 B 和 C 重复步骤 2, 分别得出其新位置 $B'(-100, 550, 150)$ 和 $C'(-200, 400, -60)$ 。通过移动这三个顶点, 就移动了整个三角形。

利用矩阵加法实现平移相当直观, 下面的函数说明了它是如何移动一个 3D 点的。

```
Matrix3x1 translate3DByAddition(Matrix3x1 start, Matrix3x1 trans)
{
    Matrix3x1 temp;
    temp = addMatrices(start, trans);
    return temp;
}
```

在游戏中, 利用矩阵平移一个 3D 位置会比平移一个 2D 位置更为常见。一般来说, 游戏中的人物角色都包含一个代表其位置的 x 值和 y 值, 如果要想让角色向右移动 10 个单位, 可以使用下面的方法来实现:

```
Player.setX ( Player.getX() + 10 );
```

如果你觉得使用矩阵比较方便的话, 那就记下它吧! 当它对一个 3D 物体和 2D 物体进行旋转操作时, 它会显得更加有用。

再说一下, 如果对物体仅仅进行平移变换的话, 那就使用矩阵加法; 但是如果还想同时对物体进行缩放和旋转的话, 那就应该使用矩阵乘法。同样地, 你可以使用相同的方式构造一个矩阵方程, 然后每次代入一个原始位置, 并与矩阵相乘, 从而得出相应的新位置。

利用矩阵乘法实现 2D 平移

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中, dx 是 x 方向上的平移量, dy 是 y 方向上的平移量。

细心的你可能已经发现, 在原始位置和新位置的坐标后面都多了一个 1, 其实它并不是坐标的组成部分, 在这里使用它可以保证矩阵运算的简便和统一, 对于此你可以进一步研究一下齐次坐标。

另外, 一定要注意矩阵相乘的次序, 如果将相乘的次序调换成: 原始点 * 变换矩阵, 那么计算出来的结果将不再正确, 程序也将失败。还记得第 5 章中, 我们曾提到矩阵相乘的次序非常重要, 第一个矩阵的列数必须等于第二个矩阵的行数。在这里, 就必须将相乘的形式设置为 3×3 矩阵乘上 3×1 矩阵, 其他的形式都不正确。

请务必记住矩阵相乘的次序: 变换矩阵 $\times$ 原始位置。

让我们利用矩阵乘法重新计算例 6-1, 看它们得出的结果是否一致。

【例 6-3】 矩阵乘法实现 2D 平移

建立一个矩阵方程(使用矩阵乘法), 将一个 2D 物体在屏幕上向右移动 50 个像素, 向下移动 100 个像素, 然后利用该方程平移一个三角形, 其顶点分别为: $A(20, 30)$ 、

$B(0, 200)$ 和 $C(300, 400)$ 。

解答：

1. 建立矩阵方程。为了能够让物体向右移动 50 个像素，向下移动 100 个像素，因此 dx 取值为 50， dy 取值为 -100 (负值意味着向下移动)：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & -100 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. 将原始顶点值代入，并与矩阵方程相乘，得出平移后的新位置。首先，将顶点 A 的原始值 (20, 30) 代入：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 50 \\ 0 & 1 & -100 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 30 \\ 1 \end{bmatrix} = \begin{bmatrix} 1(20) + 0(30) + 50(1) \\ 0(20) + 1(30) - 100(1) \\ 0(20) + 0(30) + 1(1) \end{bmatrix} = \begin{bmatrix} 70 \\ -70 \\ 1 \end{bmatrix}$$

得出顶点 A 的新位置 $A'(70, -70)$ ，这与例 6-1 的结果完全一致。

3. 对顶点 B 和 C 重复步骤 2，分别得出其新位置 $B'(50, 100)$ 和 $C'(350, 300)$ ，和例 6-1 得出的结果也完全一致。通过移动这三个顶点，就移动了整个三角形，注意到这里计算出的最终位置和例 6-1 的结果是一样的。

如果对矩阵乘法进一步研究的话，你将会发现坐标中最后的那个 1 的重要作用了。下面就来仔细看一下计算 x' 的具体过程。首先，它保持原有的 x 值 (x 乘以 1)，然后忽略 y 值 (y 乘以 0)，最后加上 dx (dx 乘以 1)；同样在计算 y' 的时候，它忽略了 x 值，保留了 y 的原始值，并加上了 dy 。很明显，在这里使用 1 完成了与 dx 和 dy 相加的操作。

当我们使用矩阵乘法完成 3D 物体的平移时，情形与 2D 平移相似。

让我们来看一下在编程中是如何解决该问题的，下面这个函数的功能就是利用矩阵乘法实现 2D 点的平移。

```
Matrix3 x1 translate2DByMultiplication(Matrix3 x1 start, float dx, float dy)
```

```
{
```

```
    Matrix3 x3 temp;
```

```
    Matrix3 x1 result;
```

```
    // 创建一个初始值为零的矩阵
```

```
    temp = createFixed3X3Matrix(0);
```

```
    // 建立 3 x 3 乘法变换矩阵
```

```
    temp.index[0][0] = 1;
```

```
    temp.index[1][1] = 1;
```

```
    temp.index[2][2] = 1;
```

```
    // 代入平移量
```

```
    temp.index[0][2] = dx;
```

```
    temp.index[1][2] = dy;
```



```
result = multiplyMatrixNxM(temp, start);
return result;
```

在这里有一些需要注意的问题, 首先我们依次将位置[0][0]、[1][1]和[2][2]处的值设置为1, 之所以这样做是为了保证点乘结果的正确性; 然后设置物体在各个方向上的平移量。一旦在设置好该矩阵之后, 下面就可以利用该 3×3 矩阵乘以原始位置矩阵了。在建立变换矩阵的过程中, 如果要是进行3D变换的话, 为矩阵对应元素赋上正确的数值会变得更加重要。

利用矩阵乘法实现3D平移

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

此处 dx 为 x 的变化量, dy 为 y 的变化量, dz 为 z 的变化量。

在3D空间中进行平移变换与2D平移的处理方法一样, 唯一不同的地方是要使用一个 4×4 的变换矩阵; 同时还要注意, 为了保证计算结果的正确性, 还需要那个额外的数值1。

【例6-4】矩阵乘法实现3D平移

建立一个矩阵方程(使用矩阵乘法), 将一个3D物体在屏幕上向左移动100个像素, 向上移动200个像素, 向后(屏幕里面)移动50个像素。然后利用该方程平移一个三角形, 其顶点分别为: $A(40, 0, 100)$ 、 $B(0, 350, 200)$ 和 $C(-100, 200, -10)$ 。

解答:

1. 建立矩阵方程。为了能够让物体向左移动100个像素, 向上移动200个像素, 向后移动50个像素, 因此 $dx = -100$, $dy = 200$, $dz = -50$ (如果你记不清楚哪个方向是取正值还是负值, 就去复习一下第1章):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -100 \\ 0 & 1 & 0 & 200 \\ 0 & 0 & 1 & -50 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. 将原始顶点值代入, 并与矩阵方程相乘, 得出平移后的新位置。首先, 将顶点A的原始值(40, 0, 100)代入:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -100 \\ 0 & 1 & 0 & 200 \\ 0 & 0 & 1 & -50 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 40 \\ 0 \\ 100 \\ 1 \end{bmatrix} = \begin{bmatrix} 1(40) + 0(0) + 0(100) - 100(1) \\ 0(40) + 1(0) + 0(100) + 200(1) \\ 0(40) + 0(0) + 1(100) - 50(1) \\ 0(40) + 0(0) + 0(100) + 1(1) \end{bmatrix} = \begin{bmatrix} -60 \\ 200 \\ 50 \\ 1 \end{bmatrix}$$

得出顶点A的新位置 $A'(-60, 200, 50)$, 与前面得出的结果完全一致。

3. 对顶点 B 和 C 重复步骤 2, 分别得出其新位置 $B'(-100, 550, 150)$ 和 $C'(-200, 400, -60)$ 。通过移动这三个顶点, 就移动了整个三角形。注意到这里计算出的结果与前面例子 6-2 的结果完全一样。

利用矩阵乘法实现 3D 空间内的平移与 2D 空间的平移非常类似, 它们最大的不同之处是, 在 3D 空间中是使用 4×4 矩阵和 4×1 矩阵相乘来实现的, 其余的过程就非常简单了。下面函数的功能是利用矩阵乘法实现 3D 平移。

```
Matrix4 x1 translate3DByMultiply( Matrix4 x1 start, float dx, float dy, float dz)
{
    Matrix4 x4temp;
    Matrix4 x1result;

    //创建一个初始值为零的矩阵
    temp = createFixed4X4Matrix(0);

    //建立 4 x 4 乘法变换矩阵
    temp.index[0][0] = 1;
    temp.index[1][1] = 1;
    temp.index[2][2] = 1;
    temp.index[3][3] = 1;

    //代入平移量
    temp.index[0][3] = dx;
    temp.index[1][3] = dy;
    temp.index[2][3] = dz;

    result = multiplyMatrixNxM( temp, start );
    return result;
}
```

同样, 该函数首先对变换矩阵的相关元素进行赋值, 然后将顶点的原始值构成的矩阵作为右乘矩阵, 经过相乘操作最终得出一个 4×1 矩阵, 它就包含了顶点的新位置。

注意

每次在计算新位置时, 它的最后一个元素总等于 1。在实际编程中, 如果对每个顶点都计算这个额外的数值 1, 那将浪费宝贵的 CPU 时间。注意, 我们的目的是计算出新的坐标值 x' 、 y' 、 z' , 因此没必要每次都去计算最后一个元素。你能否实现该设想, 对平移函数进行一些优化呢?

在本章的开始我们就说矩阵加法是比较简单和快速的方法, 经过前面对两种方法的实践, 我想你是应该会同意该说法的。但是, 唯一能够与旋转、缩放变换进行组合的方法就是矩阵乘法, 但是不管怎样只要将物体移动到正确位置, 那么我们的目的也就达到了。

自我测试

利用提供的矩阵方程，分别计算下列各顶点的新位置：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 40 \\ -20 \end{bmatrix}$$

1. $D(30, 80)$
2. $E(-50, 200)$
3. $F(100, 0)$
4. 建立一个矩阵方程(使用矩阵加法)，使一个 3D 物体向右移动 50 个像素，向下移动 300 个像素，向后(屏幕后面)移动 0 个像素，然后利用该矩阵方程去移动一个三角形，其顶点分别为： $G(200, -30, -50)$ 、 $H(90, 0, -40)$ 和 $J(-400, 50, -100)$ 。
5. 建立一个矩阵方程(使用矩阵乘法)，使一个 2D 物体在屏幕上向左移动 200 个像素，向上移动 20 个像素，然后利用该矩阵方程去移动一个三角形，其顶点分别为：
 $L(-100, 30)$ 、 $M(50, -80)$ 和 $N(70, 0)$ 。
6. 利用矩阵乘法重做第 4 题，并比较一下得到的 3 个新顶点值与第 4 题得出的结果是否一致。

6.2 缩放

在游戏中，我们同样可以使用矩阵乘法对一个物体进行缩放。与平移变换相似，如果对物体的所有顶点进行了缩放，那么变换的结果便是缩放了整个物体。让我们建立另外一个矩阵方程用来实现缩放变换，同样仍以 2D 空间内的缩放开始，然后再将其推广到 3D 空间。

2D 缩放变换

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中， S_x 为 x 方向上的缩放因子， S_y 为 y 方向上的缩放因子。

只要我们将缩放因子 S_x 和 S_y 代入矩阵方程，下面的处理过程就和平移一样了：每次代入一个顶点，并与变换矩阵相乘，从而得出一个新的坐标值。如果要进行均匀缩放的话，即保持物体原先的形状，那么就要保证 $S_x = S_y$ ；然而在实际中你不一定要进行均匀缩放，即 S_x 和 S_y 的值不相等，此时进行的变换就是非均匀缩放。

当为缩放因子 S_x 和 S_y 赋值时，如果取值位于 0 和 1 之间，那么变换的结果将缩小物体；反之，如果取值大于 1，那么变换将扩大物体(注意：当缩放因子为负值时，变换会使物体反射到相反方向上)。下面就来看一段演示缩放一个矩形的程序。

【例 6-5】 2D 均匀缩放

建立一个矩阵方程，将一个 2D 物体均匀放大 3 倍，然后利用该方程对图 6-2 中的矩形

进行缩放变换。

解答：

1. 建立矩阵方程。为了能够让物体均匀缩放到原来的3倍，因此缩放因子等于3：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. 依次将原始顶点值代入，并与矩阵方程相乘，得出变换后的新位置。首先，将顶点A的原始值(10, 10)代入：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 3(10) + 0(10) + 0(1) \\ 0(10) + 3(10) + 0(1) \\ 0(10) + 0(10) + 1(1) \end{bmatrix} = \begin{bmatrix} 30 \\ 30 \\ 1 \end{bmatrix}$$

得出顶点A的新位置A'(30, 30)。

3. 对顶点B、C和D重复步骤2，分别得出其新位置B'(150, 30)、C'(150, 120)和D'(30, 120)。通过缩放这四个顶点，就将整个矩形扩大了3倍。

4. 矩形的原始位置和缩放后的新位置如图6-3所示。

将上述过程写成程序并不困难，下面给出了一个函数，其功能就是根据传进来的缩放因子数值，利用矩阵乘法来实现缩放变换：

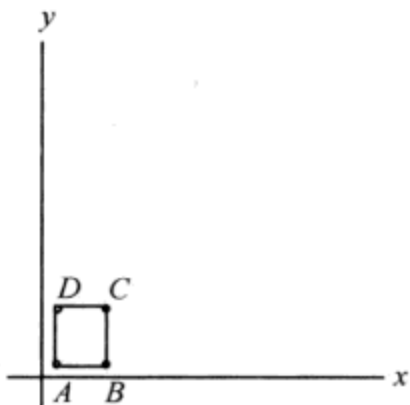


图6-2 将被缩放的矩形

```
Matrix3 x1 scale2DBuMultiplication(Matrix3 x1 start, float dx, float dy)
{
    Matrix3 x3 temp;
    Matrix3 x1 result;

    // 创建一个初始值为零的矩阵
    temp = createFixed3 x3Matrix(0);

    // 建立 3 x 3 乘法变换矩阵
    temp.index[0][0] = dx;
    temp.index[1][1] = dy;
    temp.index[2][2] = 1;

    result = multiplyMatrixNxM(temp, start);
    return result;
}
```

在上述函数中，传入参数 dx、dy 决定了该函数实现的是均匀缩放还是非均匀缩放，如果你仅仅想要进行均匀缩放的话，那就让函数只接受一个参数，然后在程序中将两个缩放因子设置为相等的数值即可。

能够对一个单独点进行缩放是非常有意义的，但是对于大多数物体而言，它们往往具有一定的几何形状。下面就让我们来看一个均匀缩放矩形的演示程序。


```

void scale2D()
{
    Matrix3x1 start, temp;
    float dx, dy, height, width;
    cout << "Please enter the coordinates and dimensions of a rectangle. \n";
    cout << "X coordinates \n";
    cin >> start.index[0];
    cout << "Now the Y coordinates \n";
    cin >> start.index[1];
    cout << "Enter the rectangle's height. \n";
    cin >> height;
    cout << "Enter the rectangle's width. \n";
    cin >> width;

    //将矩阵最后一个元素设为1
    start.index[2] = 1;
    cout << endl;
    cout << "Now enter the amount to scale by. \n";
    cin >> dx;
    dy = dx;

    temp = scale2DByMultiplication(start, dx, dy);
    width = temp.index[0] + width;
    height = temp.index[1] + height;

    cout << "The new position is
    ↵" << temp.index[0] << "," << temp.index[1] << "\n";
    cout << "The right coord is
    ↵" << width << "," << temp.index[1] << "\n";
    cout << "The bottom coord is
    ↵" << height << "," << temp.index[0] << "\n";
}

```

该函数给出了矩阵缩放后的所有顶点信息，它比仅缩放一个点更加有用，在后面章节中我们会看到另外一个关于旋转变换的类似例子。

请仔细看一下图6-3，你可能注意到矩形确实被扩大了3倍，但是它好像相对原点发生了偏移，为什么会这样呢？原因是我们建立的变换矩阵是相对于原点的，因此当放大矩形时它会远离原点，当缩小矩形时它会移近原点。如果要想让物体相对于其自身位置进行缩放变换的话，这就需要一系列操作，在本章的后面章节会介绍该问题。下面让我们再来看一个相对原点进行非均匀缩放的例子。

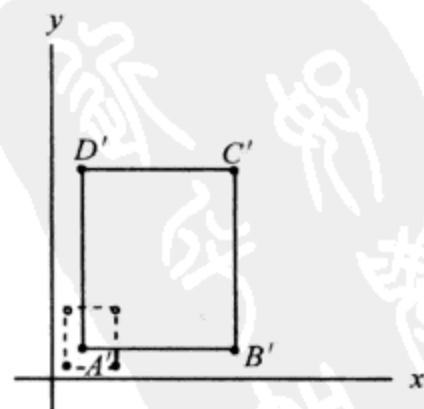


图6-3 均匀缩放前后矩形的位置

提示

解决该问题的一个方法是相对于物体原点建立一个局部坐标系。比如,如果我们想要对站在地面上的一个人进行缩放的话,就可以将其脚中心作为局部坐标系的原点,然后利用平移矩阵将其移动到全局坐标系的原点,并使用一个单一的缩放矩阵对其进行缩放变换,最后再重新将其移回到原始位置。

【例 6-6】 2D 非均匀缩放

假设有一个矩形形状的物体,如图 6-4 所示,其四个顶点坐标分别为 $A(20, 0)$ 、 $B(50, 0)$ 、 $C(50, 100)$ 和 $D(20, 100)$,现在有一个巨大的石头砸在它上面,为了让物体看起来确实是被砸扁了,请你建立一个矩阵方程,将物体沿 x 轴放大 1.5 倍、沿 y 轴方向缩小 0.1 倍,然后利用该方程对这个矩形进行缩放变换。

解答:

1. 建立矩阵方程。在该问题中, x 轴方向的缩放因子为 1.5, y 轴方向的缩放因子为 0.1:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. 依次将原始顶点值代入,并与矩阵方程相乘,得出变换后的新位置。首先,将顶点 A 的原始值 $(20, 0)$ 代入:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.5(20) + 0(0) + 0(1) \\ 0(20) + 0.1(0) + 0(1) \\ 0(20) + 0(0) + 1(1) \end{bmatrix} = \begin{bmatrix} 30 \\ 0 \\ 1 \end{bmatrix}$$

得出顶点 A 的新位置 $A'(30, 0)$ 。

3. 对顶点 B 、 C 和 D 重复步骤 2, 分别得出其新位置 $B'(75, 0)$ 、 $C'(75, 10)$ 和 $D'(30, 10)$ 。通过缩放这四个顶点,就缩放了整个矩形。

4. 矩形的原始位置和缩放后的新位置如图 6-5 所示,可以看出矩形变得宽、扁了。

其实,在 3D 空间内的缩放操作和 2D 空间相似,只需在矩阵方程中增加一维即可。

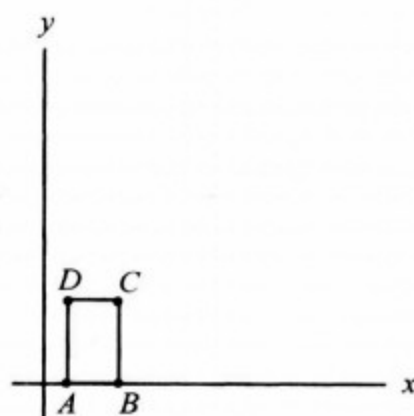


图 6-4 非均匀缩放的矩形

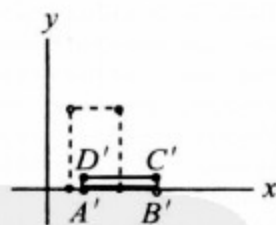


图 6-5 非均匀缩放矩形前后的位置

3D 缩放变换

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中, S_x 、 S_y 和 S_z 分别为在 x 轴、 y 轴和 z 轴方向上的缩放因子。

【例6-7】 3D 均匀缩放

建立一个矩阵方程，将一个3D物体均匀放大5倍，然后利用此方程对一个三角形进行缩放变换，其顶点分别为： $A(50, 0, -10)$ 、 $B(0, 20, -100)$ 和 $C(200, 150, -50)$ 。

解答：

1. 建立矩阵方程。为了能够让物体均匀缩放5倍，因此三个方向上缩放因子都应该等于5：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. 依次将原始顶点值代入，并与矩阵方程相乘，得出变换后的新位置。首先，将顶点A的原始值(50, 0, -10)代入：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 0 \\ -10 \\ 1 \end{bmatrix} = \begin{bmatrix} 5(50) + 0(0) + 0(-10) + 0(1) \\ 0(50) + 5(0) + 0(-10) + 0(1) \\ 0(50) + 0(0) + 5(-10) + 0(1) \\ 0(50) + 0(0) + 0(-10) + 1(1) \end{bmatrix} = \begin{bmatrix} 250 \\ 0 \\ -50 \\ 1 \end{bmatrix}$$

得出顶点A的新位置 $A'(250, 0, -50)$ 。

3. 对顶点B、C重复步骤2，分别得出其新位置 $B'(0, 100, -500)$ 和 $C'(1000, 750, -250)$ 。通过放大这三个顶点，就将整个三角形放大了5倍。

同样的，我们也可以完成3D空间内的非均匀缩放，让我们来看下面这个例子：

【例6-8】 3D 非均匀缩放

建立一个矩阵方程，将一个3D物体的高度扩大到原来的2倍，深度(z方向)变为原来的一半。然后利用它对一个三角形进行缩放，其顶点分别为： $A(50, 0, -10)$ 、 $B(0, 20, -100)$ 和 $C(200, 150, -50)$ 。

解答：

1. 建立矩阵方程。为了能够让物体的高度放大2倍，深度变为原来的一半，因此它在x方向上的缩放因子为1(保持不变)，y方向的缩放因子为2，z方向上的缩放因子为0.5：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. 依次将原始顶点值代入，并与矩阵方程相乘，得出变换后的新位置。首先，将顶点A的原始值(50, 0, -10)代入：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 0 \\ -10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1(50) + 0(0) + 0(-10) + 0(1) \\ 0(50) + 2(0) + 0(-10) + 0(1) \\ 0(50) + 0(0) + 0.5(-10) + 0(1) \\ 0(50) + 0(0) + 0(-10) + 1(1) \end{bmatrix} = \begin{bmatrix} 50 \\ 0 \\ -5 \\ 1 \end{bmatrix}$$

得出顶点 A 的新位置 $A'(50, 0, -5)$ 。

3. 对顶点 B 、 C 重复步骤 2，分别得出其新位置 $B'(0, 40, -50)$ 和 $C'(200, 300, -25)$ 。通过缩放这三个顶点，就缩放了整个三角形。

你应该还记得，在前面我们定义了一个函数，它实现了 2D 空间内的均匀缩放和非均匀缩放。下面就也来定义一个相似的函数，不过它要完成 3D 空间内的缩放。

```
Matrix4x1 scale3DByMultiply (Matrix4x1 start, float dx, float dy, float dz)
{
    Matrix4x4 temp;
    Matrix4x1 result;

    temp = createFixed4x4Matrix(0);

    // 建立一个 3x3 的变换矩阵
    temp.index[0][0] = dx;
    temp.index[1][1] = dy;
    temp.index[2][2] = dz;
    temp.index[3][3] = 1;

    result = multiplyMatrixNxm(temp, start);
    return result;
}
```

该函数的实现与前面 2D 缩放函数的过程是类似的，只不过在这里增加了一个新的变量 dz 。同样，如果你想对一个点或者一组点仅进行均匀缩放的话，那么就把函数设计为只接受一个参数，然后你在程序中将三个缩放因子设置为相同的数值即可，在本章的后面会有关于这样函数的例子，你可以留神注意一下。

注意缩放变换与利用矩阵乘法进行平移变换的过程是类似的，这很好，因为有利于对它们进行组合，形成复杂的变换，这将在后面再作介绍。同时我们还介绍了相对原点进行 2D、3D 缩放变换，这将能够使相对任何一个位置进行缩放的实现变得更加容易。

自我测试

1. 建立一个矩阵方程，将一个 2D 物体均匀缩放到原来的一半，然后利用此方程缩放一个三角形，其顶点分别为： $A(-50, 20)$ 、 $B(-10, 20)$ 和 $C(-30, 40)$ 。
2. 建立一个矩阵方程，将一个 2D 物体变得比原来看起来更高、更瘦，在 x 方向上缩放到原来的 $1/4$ ， y 方向上缩放到原来的 3 倍。然后利用此方程缩放一个三角形，其顶点分别为： $A(-50, 20)$ 、 $B(-10, 20)$ 和 $C(-30, 40)$ 。
3. 建立一个矩阵方程，将一个 3D 物体均匀缩放到原来的 10 倍，然后利用此方程缩放一个三角形，其顶点分别为： $D(0, 30, -100)$ 、 $E(-50, 100, -20)$ 和 $F(-20, 0, -300)$ 。

4. 建立一个矩阵方程，将一个3D物体在 x 方向上放大2倍，在 y 方向上保持不变，在 z 方向上变为原来的一半，然后利用此方程缩放一个三角形，其顶点分别为：
 $D(0, 30, -100)$ 、 $E(-50, 100, -20)$ 和 $F(-20, 0, -300)$ 。

6.3 旋转

要想对一个物体进行旋转变换，这里有两种方法，第一种是利用四元数，但是这种方法需要的数学知识比较复杂；另外一种是利用矩阵方程，它与平移矩阵、缩放矩阵非常类似，被称作欧拉旋转。处理旋转的过程仍然是这样的，首先建立一个2D或者3D空间内的变换矩阵，然后依次将顶点的原始值代入，与矩阵相乘得出各顶点对应的新位置。

2D 旋转变换

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中， θ 为旋转角度。

只要知道了旋转角度，就可以利用三角函数如正弦、余弦等求出其对应的数值，然后就可以利用它们进行矩阵相乘了。关于角度的相关知识，你可以回顾一下第3章的相关章节。请记住，正的旋转角度意味着沿逆时针方向进行旋转，而负的旋转角度意味着沿顺时针方向进行旋转。同样的，如果你已经知道了一个旋转角度，而且正要计算它们的正弦、余弦值，你此时可以在结果中保留角度的形式，而不必计算出具体的数值；但是，如果要想用程序来计算角度的正弦或余弦值，那么就应该首先将其转换成弧度的形式。

利用矩阵进行关于变换的编程

在程序中，我们会经常用到 `math.h` 头文件，它包含了计算角度的正弦、余弦值等关键函数。需要注意的是，这些函数接受的参数都是弧度值，因此在调用它们的时候必须首先将参数转换为正确的形式，下面定义的这两个宏将非常有用。

```
#define RadsToDegrees( radian ) ( (radian) * (180.0f / M_PI) )
```

应该很容易地看出来，上面这个宏将传入的弧度值转换成对应的角度值，关于常量 `M_PI` 的定义我们一会儿再说。

```
#define DegreesToRads( degrees ) ( (degrees) * (M_PI / 180.0f) )
```

很明显，该宏的功能刚好与上面的相反，即将一个角度值转换为对应的弧度值。

常量 `M_PI` 的定义来自源头文件 `math.h`，当你在编写程序时，不仅要保证你包含了头文件 `math.h`，还要设置你将要使用数学头文件的定义，比如 `PI` 等。同样的，常量 `M_PI` 来源于文件 `math.h`，在本书中它代表 `pi`。如果你的计算机支持头文件的定义，那么就应该使用下面的形式来使用它们：

```
#define _USE_MATH_DEFINES
#include <math.h>
```

在包含头文件 `math.h` 之前, 定义 `USE_MATH_DEFINES` 非常重要, 只有这样你才能在程序中使用那些常量。

【例 6-9】 2D 旋转

建立一个矩阵方程, 将一个 2D 物体旋转 90° , 然后利用它旋转图 6-6 中的三角形, 其顶点分别为: $A(50, 40)$ 、 $B(100, 40)$ 和 $C(75, 200)$ 。

解答:

1. 建立矩阵方程, 其旋转角度为 90° :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ & 0 \\ \sin 90^\circ & \cos 90^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. 依次将原始顶点值代入, 并与矩阵方程相乘, 得出变换后的新位置。首先, 将顶点 A 的原始值 $(50, 40)$ 代入:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 50 \\ 40 \\ 1 \end{bmatrix} = \begin{bmatrix} 0(50) - 1(40) + 0(1) \\ 1(50) + 0(40) + 0(1) \\ 0(50) + 0(40) + 1(1) \end{bmatrix} = \begin{bmatrix} -40 \\ 50 \\ 1 \end{bmatrix}$$

得出顶点 A 的新位置 $A'(-40, 50)$ 。

3. 对顶点 B 、 C 重复步骤 2, 分别得出其新位置 $B'(-40, 100)$ 和 $C'(-200, 75)$ 。通过旋转这三个顶点, 就旋转了整个三角形。该三角形被旋转前后的位置如图 6-7 所示。

在 2D 空间中旋转物体非常有用, 而且其计算过程也非常直观, 下面就让我们来看一个旋转 2D 点的函数:

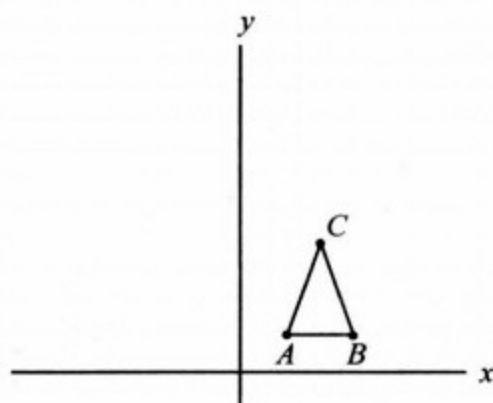


图 6-6 旋转一个 2D 三角形

```
Matrix3 x1 rotate2D(Matrix3 x1 start, float theta)
{
    Matrix3 x3 temp;
    Matrix3 x1 result;

    //初始化一个零矩阵
    temp = createFixed3x3Matrix(0);

    //设置变换矩阵的对应元素
    temp.index[0][0] = cos(DegreesToRads(theta));
    temp.index[1][1] = cos(DegreesToRads(theta));
    temp.index[2][2] = 1;
    temp.index[0][1] = -1 * (sin(DegreesToRads(theta)));
    temp.index[1][0] = sin(DegreesToRads(theta));
    temp.index[2][2] = 1;
    result = multiplyMatrixNxM(temp, start);
    return result;
}
```

你可能已经看到了,在该函数中我们使用了前面定义的宏 DegreesToRads(),让我们来看一下该函数的过程。首先,根据用户指定的旋转角度初始化旋转矩阵,在建立好矩阵之后,就让它与顶点的原始位置相乘,从而得出变换后的新位置。

下面这段程序演示了如何绕某一点旋转一个矩形。

```
void rotate2D()
{
    Matrix3x1 start, temp;
    float height, width, theta;
    cout << "Let's rotate a 2D! \n";
    cout << "Please enter the coordinate and dimensions. \n";
    cout << "X coordinates \n";
    cin >> start.index[0];
    cout << "Now the Y coordinate. \n";
    cin >> start.index[1];
    cout << "Enter the rectangle's height. \n";
    cin >> height;
    cout << "Enter the rectangle's width. \n";
    cin >> width;

    //将矩阵最后一个元素设为1
    start.index[2] = 1;
    cout << endl;
    cout << "Now enter the amount to rotate by in degrees. \n";
    cin >> theta;

    //现在我们有了自己的 info,开始旋转
    temp = rotate2D(start, theta);

    width = temp.index[0] + width;
    height = temp.index[1] + height;

    //This gives the new locations calced from the temp matrix.
    cout << "The right coordinate of the rectangle
    << width << ", " << temp.index[1] << " \n";
    cout << "The bottom coordiante of the rectangle
    << height << ", " << temp.index[0] << " \n";
}
```

在游戏中能够对一个矩形进行旋转是非常有意义的,这是因为大多数的 2D 图形都是矩形形状的,游戏中对图形的显示大多数都采用下面的方法,即根据视点的位置只显示图形的特定部分。在上面的旋转函数中,它就允许用户自己指定需要显示的部分,而有的游戏则直接根据用户的输入进行显示,并且使用类似的图形更新过程。

在图 6-7 中,你可能发现该三角形发生了旋转和移动,你或许希望它只是待在原地进行旋转。遗憾的是该操作的变换也将会是一种组合变换。

提示

该问题仍然可以通过为模型建立一个局部坐标系来解决。

利用旋转矩阵对物体进行旋转变换时，它与缩放变换类似，默认都是相对于原点进行变换的，那么这样变换的结果会使物体看起来好像在关于原点运动。但是如果你要想让物体绕其自身中心或某一特定点进行旋转的话，那么就必须进行一系列变换，称作组合变换，我们将在下一节中讨论这个问题。

与此同时，让我们来看一下 3D 空间中的旋转问题，大家都知道在 2D 空间中进行旋转时它仅在一个平面内进行，就像电脑的显示器屏幕一样。然而在 3D 空间中包含了 3 个不同的旋转平面—— xy 平面、 yz 平面和 xz 平面，这三个平面如图 6-8 所示。

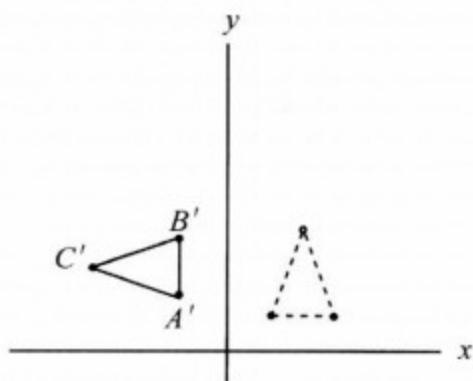


图 6-7 旋转前后的 2D 三角形

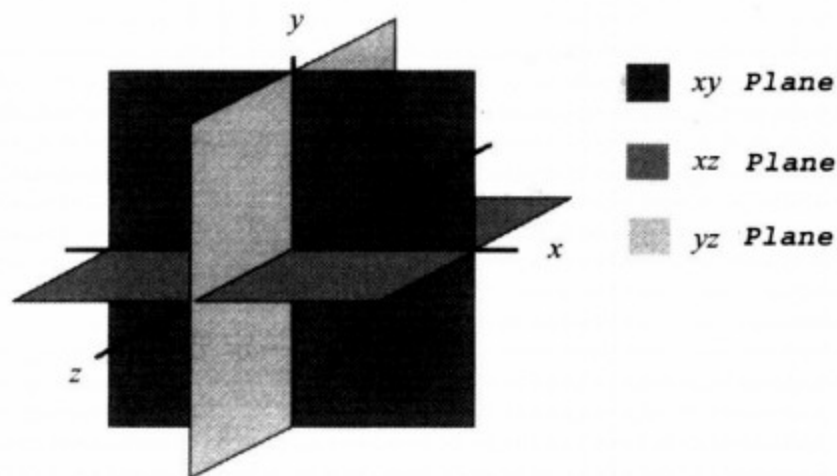


图 6-8 3D 空间中的 3 个平面

注意， xy 平面其实就是计算机屏幕所在的 2D 平面。由于在 3D 空间中存在 3 个不同的旋转平面，因此相应的就有三个各自的旋转矩阵，下面让我们依次介绍它们。

绕 Z 轴 3D 旋转

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中， θ 为旋转角度。

请记住， z 轴的正方向由屏幕里面指向外部，因此绕 z 轴的旋转其实就是在屏幕空间 (xy 平面) 内进行旋转，这也就是此时的旋转矩阵与 2D 旋转矩阵相似的原因。但与 2D 矩阵相比，它多了一行一列，因此构成了一个 4×4 矩阵，这种绕 z 轴的旋转被称作滚动，该名称来源于模拟飞行领域的专业术语。如果你站在地平面上，伸展开胳膊，并进行来回的摆动，那么该动作就非常类似于飞机的摆动。

绕 z 轴的旋转可以使用下列函数：

```
Matrix4 x1 rotate3DZ(Matrix4 x1 start, float theta)
{
    Matrix4 x4 temp;
    Matrix4 x1 result;
```



```

//创建一个零矩阵
temp = createFixed4x4Matrix(0);

//设置变换矩阵的对应元素
temp.index[0][0] = cos(DegreesToRads(theta));
temp.index[1][1] = cos(DegreesToRads(theta));
temp.index[2][2] = 1;
temp.index[3][3] = 1;

temp.index[0][1] = -1 * (sin(DegreesToRads(theta)));
temp.index[1][0] = sin(DegreesToRads(theta));

result = multiplyMatrixNxm(temp, start);
return result;
}

```

当在建立旋转变换矩阵时，一定要注意角度的计算，同时还要将它们的正弦、余弦值放到正确的位置。

绕 X 轴 3D 旋转

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中， θ 为旋转角度。

x 轴的朝向是左右方向的，因此绕 x 轴的旋转与站在地面上进行弯腰鞠躬的动作是相似的。

在模拟飞行领域中，飞机的前后转动与此转动形式非常类似，因此很多编程者都将绕 x 轴的旋转称作倾斜转动，注意角度的正弦、余弦值在矩阵中位置的变化。矩阵的第一行与第一列和 x 值有关，因此在此种情况下它们具有单位化的形式。绕 x 轴旋转，其 x 坐标不会发生变化，这个规律非常有用，因为它可以帮助你记清楚 3D 旋转矩阵的形式，同样绕 y 轴和 z 轴的旋转都具有类似的规律。

下面的函数完成了绕 x 轴的旋转：

```

Matrix4x1 rotate3DX(Matrix4x1 start, float theta)
{
    Matrix4x4 temp;
    Matrix4x1 result;

    //创建一个零矩阵
    temp = createFixed4x4Matrix(0);

```

```

//设置变换矩阵的对应元素
temp.index[0][0] = 1;
temp.index[1][1] = cos ( DegreesToRads ( theta ) );
temp.index[2][2] = cos ( DegreesToRads ( theta ) );
temp.index[3][3] = 1;

temp.index[1][2] = -1 * ( sin ( DegreesToRads ( theta ) ) );
temp.index[2][1] = sin ( DegreesToRads ( theta ) );

result = multiplyMatrixN×M( temp , start );
return result;
}

```

在该函数中，最大的不同是旋转角度的正弦、余弦值在矩阵的位置不一样，本章中的示例程序使用的就是这个例子。

绕 Y 轴 3D 旋转

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中， θ 为旋转角度。

注意在该情形下，变换矩阵中的第二行与第二列(代表 y 坐标)遇有单位化的形式，绕 y 轴的旋转与模拟飞行领域中的偏航类似。如果你站在地面上像芭蕾舞演员那样进行旋转，该动作就称作偏转，这也很容易想象，比如将某人的照片绕竖直方向的轴进行画圈等。一旦我们建立了旋转矩阵，下面所要做的就是依次将每个顶点代入，并与矩阵相乘，找到其对应的新位置。需要说明的是，这三种不同形式的旋转必须要单独进行，不过在下一节中我们可看到它们其实是可以合并到一起的。

下面的函数实现了绕 y 轴的旋转：

```

Matrix4×1 rotate3DY(Matrix4×1 start, float theta)
{
    Matrix4×4 temp;
    Matrix4×1 result;

    //创建一个零矩阵
    temp = createFixed4×4Matrix(0);

    //设置变换矩阵的对应元素
    temp.index[0][0] = cos ( DegreesToRads ( theta ) );
    temp.index[1][1] = 1;
    temp.index[2][2] = cos ( DegreesToRads ( theta ) );
}

```

```

temp.index[3][3] = 1;

temp.index[2][0] = -1 * (sin(DegreesToRads(theta)));
temp.index[0][2] = sin(DegreesToRads(theta));

result = multiplyMatrixN×M(temp, start);
return result;
}

```

请记住再次检查一下角度的正弦、余弦值在矩阵中的位置。

提示

这三种形式的旋转矩阵都是相对于原点建立的，因此它们与 2D 旋转具有相同的效果。再次说明一下，如果你想让物体绕其他某个点进行旋转，就必须建立一个组合变换矩阵。

【例 6-10】 绕 y 轴 3D 旋转

建立一个矩阵方程，使一个 3D 物体绕 y 轴旋转 180° ，然后利用它旋转一个三角形，其顶点分别为：A(100, 0, -50)、B(40, -30, 0)和 C(-20, 100, 50)。

解答：

1. 建立矩阵方程，其旋转角度为 180° ：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos 180^\circ & 0 & \sin 180^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 180^\circ & 0 & \cos 180^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2. 依次将原始顶点值代入，并与矩阵方程相乘，得出变换后的新位置。首先，将顶点 A 的原始值(100, 0, -50)代入：

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 100 \\ 0 \\ -50 \\ 1 \end{bmatrix} = \begin{bmatrix} -1(100) + 0(0) + 0(-50) + 0(1) \\ 0(100) + 1(0) + 0(-50) + 0(1) \\ 0(100) + 0(0) - 1(-50) + 0(1) \\ 0(100) + 0(0) + 0(-50) + 1(1) \end{bmatrix} = \begin{bmatrix} -100 \\ 0 \\ 50 \\ 1 \end{bmatrix}$$

得出顶点 A 的新位置 A'(-100, 0, 50)。

3. 对顶点 B、C 重复步骤 2，分别得出其新位置 B'(-40, -30, 0)和 C'(20, 100, -50)，通过旋转这三个顶点，就旋转了整个三角形。

如果你能够完成绕 y 轴的旋转变换，那么你也能够以同样的方式实现绕另外两个轴的旋转，它们的处理方式都是一样的。注意到在例子中我们使用的角度度数是 90° 的倍数，因此它们的正弦和余弦值等于 0 或者 1。然而在实际中，对于其他的角度，其正弦、余弦值都为小数，也就是说变换后新位置的坐标值不一定总是整数。另外，仍要注意所有的旋转矩阵都是相对于原点而建立的，下一节我们将会讨论如何绕其他点进行旋转。

在上一节的函数中，我们创建了一个新矩阵，这对于检验矩阵乘法非常有用，另外在程

序中最好能够让用户输入旋转角度值。最后请认真看一下本章的示例代码，并细心观察一下绕各个轴到底是怎样完成旋转的。

自我测试

1. 建立一个矩阵方程，将一个 2D 物体旋转 30° ，然后利用它旋转一个三角形，其顶点分别为： $A(-10, 50)$ 、 $B(80, 0)$ 和 $C(-50, 100)$ 。
2. 建立一个矩阵方程，让一个 3D 物体保持在原地，并且绕 y 轴每次旋转 1° ，然后利用它旋转一个三角形，其顶点分别为： $D(0, 20, -50)$ 、 $E(-10, 0, 20)$ 和 $F(30, 20, 0)$ 。
3. 建立一个矩阵方程，将一个 3D 物体绕 x 轴旋转 45° ，然后利用它旋转一个三角形，其顶点分别为： $G(0, 30, -100)$ 、 $H(-50, 100, -20)$ 和 $J(-20, 0, -300)$ 。
4. 建立一个矩阵方程，将一个物体绕 z 轴旋转 90° ，然后利用它旋转一个三角形，其顶点分别为： $L(0, 30, -100)$ 、 $M(-50, 100, -20)$ 和 $N(-20, 0, -300)$ 。

6.4 组合变换

截止到目前，本章前面的所有章节都是在为本节打基础。组合变换其实就是将一系列变换矩阵组合成一个矩阵的过程。其实在本书的第 1 章我们已经提到过组合变换的例子，比如：相对物体自身的中心进行缩放变换、相对于中心点(或一个顶点)进行旋转变换以及组合 3D 旋转变换等，下面我们将以它们作为例子讲解组合变换的过程。

对变换矩阵进行组合的操作，不仅仅局限于上述情形，它可以对任何的平移变换、缩放变换和旋转变换进行组合。在同一帧中利用组合变换可以节省大量的处理时间，因为这样就不需要让处理器再进行组合的操作了，不过它会让你在编程过程中花费较多的时间，但是当游戏在运行时其性能将会得到较大的提升。

让我们以绕物体自身中心旋转为例，详细介绍一下组合变换的处理过程。在前面的例 6-9 中，我们完成了对一个 2D 物体进行旋转，注意当时是相对于原点对其旋转 90° 的，现在要让它绕其自身中心(75, 93)旋转 90° ，该组合变换需要三个步骤，如图 6-9 所示。

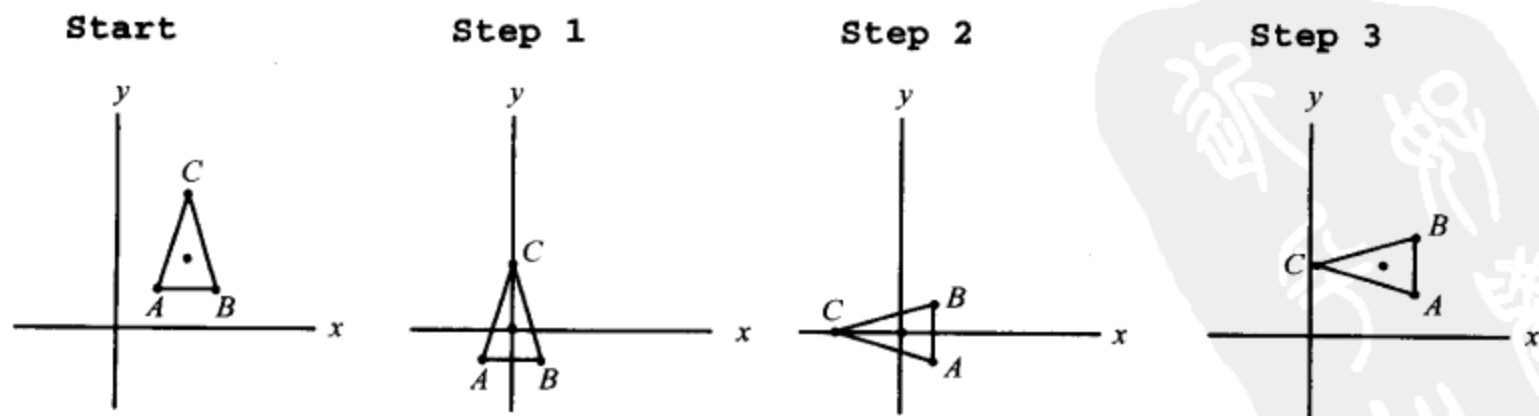


图 6-9 三个操作步骤

1. 将物体中心平移到坐标系原点(即向左移 75 个单位，向下移 93 个单位)；
2. 旋转 90° ；

3. 将物体移回到原处(即向右移 75 个单位、向上移动 93 个单位)。

在上面步骤中, 我们首先将物体的中心移到坐标系的原点, 这样再进行旋转其实上就是相对于其自身中心旋转了; 同样的, 你也可以使用上面的三个步骤来完成相对任何一点进行缩放变换的处理。只要知道了组合变换的步骤, 以及它们的相对次序, 下面就可以将这几个变换矩阵组合成一个, 首先将物体的原始位置放到最右边, 然后按照从右到左的顺序依次将变换矩阵组合进来。在上面的例子中, 物体的新位置其实就等于下面等式的计算结果。

$$\begin{bmatrix} 1 & 0 & 75 \\ 0 & 1 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ & 0 \\ \sin 90^\circ & \cos 90^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

注意

在上面的例子中, 组合变换只有三个步骤, 其实在实际的变换中你想组合多少个变换都可以。如果你想在一帧中组合多个变换, 那么就需要事先将其变换矩阵组合到一块。

在对一系列变换矩阵进行组合的时候一定要特别小心, 还记得在第 5 章中我们介绍过矩阵的乘法是不满足交换率的, 这也就意味着矩阵相乘的次序非常重要。如果你将两个矩阵相乘的次序反转, 那么肯定得不出所期望的结果, 因此一定要保证它们顺序的正确性。

一旦将所有的变换矩阵都建立完毕, 最后一个步骤就是将它们合并成一个矩阵, 只要将它们相乘在一起, 即可完成该步骤。在上面的例子中, 最终的变换矩阵如下所示:

$$\begin{aligned} & \begin{bmatrix} 1 & 0 & 75 \\ 0 & 1 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ & 0 \\ \sin 90^\circ & \cos 90^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 75 \\ 0 & 1 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 & 75 \\ 1 & 0 & 93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -75 \\ 0 & 1 & -93 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 & 168 \\ 1 & 0 & 18 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \end{aligned}$$

在上面的过程中, 一直与右边的矩阵进行相乘, 直到最终得到一个矩阵为止。此时最终的矩阵已经完成了前面的三个步骤, 在该例中玩家将看到物体的初始位置和最终位置, 如图 6-9 所示, 而在它们之间发生的中间变换将不会被看到, 这是因为所有的变换是在同一帧中完成的。现在来看一下如何在程序中完成这些步骤, 其实大可不必将这三个矩阵都写到程序中, 只需将最终的组合矩阵写入即可。

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 168 \\ 1 & 0 & 18 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

你也许感觉在编程的前期完成了许多工作，的确是这样的。然而这样做会大大提升程序的性能，这是因为你完成了矩阵相乘的操作，就不必让计算机对每个顶点都再重复该过程，试想一个复杂的物体常常包含数千个顶点，这样显然能够节省大量的处理时间，那么程序的性能当然就会提升许多。

到现在，后面的处理步骤就和以前的一样了，即依次将每个顶点的原始值代入，并与矩阵相乘得出其对应的新位置。同样的，你通过变换物体所有的顶点，便变换了整个物体。在该例中，三角形三个顶点的原始位置分别为：A(50, 40)、B(100, 40)和C(75, 200)，变换后的新位置分别为：A'(128, 68)、B'(128, 118)和C'(-32, 93)。下图 6-10 显示了该三角形变换前后的位置。

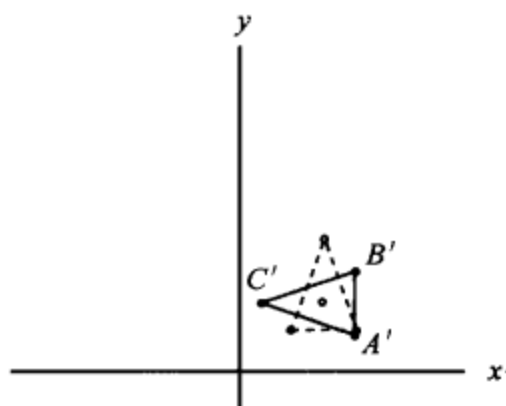


图 6-10 相对物体中心进行 2D 旋转

上述过程仍然可以应用于 3D 空间中，下面就让我们来看一个 3D 空间中关于组合变换的例子。

【例 6-11】 相对中心对 3D 物体进行缩放

假设在你的游戏中有一个 3D 模型，你想让它保持在原地并将其缩小到原来的一半。建立一个矩阵方程，使用单个组合变换矩阵将任意一个 3D 物体相对其中心 (x_c, y_c, z_c) 缩小一半。

解答：

1. 首先组织一下组合变换的三个步骤：

- 将物体的中心平移到原点。
- 将物体均匀缩放到原来的一半。
- 将物体的中心移回到原始位置。

2. 利用这三个变换矩阵按照从右到左的顺序建立组合矩阵方程。

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_c \\ 0 & 1 & 0 & y_c \\ 0 & 0 & 1 & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3. 最后一个步骤，也是最难的一步，将所有的变换矩阵相乘到一块，一定要注意相乘的次序，即从左到右相乘。

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_c \\ 0 & 1 & 0 & y_c \\ 0 & 0 & 1 & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & x_c \\ 0 & 0.5 & 0 & y_c \\ 0 & 0 & 0.5 & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_c \\ 0 & 1 & 0 & -y_c \\ 0 & 0 & 1 & -z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & (0.5)x_c \\ 0 & 0.5 & 0 & (0.5)y_c \\ 0 & 0 & 0.5 & (0.5)z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4. 最后一行就是最终的组合变换矩阵，它将任意一个 3D 物体相对其中心缩放到原来的一半。

在前面一节中，我们提到对于 3D 空间中旋转必须将其分解为三个独立的过程，即分别绕 x 、 y 和 z 轴旋转。下面这个例子可以完成将这三个过程组合成一个，不过在研究这个例子之前你可能首先需要回顾一下上一节的相关内容。

【例 6-12】 3D 组合旋转

建立一个组合变换矩阵方程，将一个 3D 物体绕 z 轴旋转 30° ，绕 x 轴旋转 180° ，绕 y 轴旋转 90° ，然后利用它旋转一个三角形，其顶点分别为： $A(200, 0, -30)$ 、 $B(0, 50, -150)$ 和 $C(40, 20, -100)$ 。

解答：

1. 首先组织一下组合变换的三个步骤：

- 绕 z 轴旋转 30° ；
- 绕 x 轴旋转 180° ；
- 绕 y 轴旋转 90° 。

2. 利用这三个单独的变换矩阵按照从右到左的顺序建立组合矩阵方程。

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos 90^\circ & 0 & \sin 90^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90^\circ & 0 & \cos 90^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 180^\circ & -\sin 180^\circ & 0 \\ 0 & \sin 180^\circ & \cos 180^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30^\circ & -\sin 30^\circ & 0 & 0 \\ \sin 30^\circ & \cos 30^\circ & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3. 最后一个步骤，也是最难的一步，将所有的变换矩阵相乘到一块，一定要注意相乘的次序，即从左到右相乘。

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.8660 & -0.5 & 0 & 0 \\ 0.5 & 0.8660 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.8660 & -0.5 & 0 & 0 \\ 0.5 & 0.8660 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -0.5 & -0.8660 & 0 & 0 \\ -0.8660 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4. 现在我们已经得到了最终的组合变换矩阵, 下面所要做的就和前面的例子一样了, 即依次将每个顶点的原始值代入, 与变换矩阵相乘, 从而得到新的位置值。比如将顶点 A 的原始位置(200, 0, -30)代入, 可以得到下面的等式:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -0.5 & -0.8660 & 0 & 0 \\ -0.8660 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 200 \\ 0 \\ -30 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0(200) + 0(0) - 1(-30) + 0(1) \\ -0.5(200) - 0.8660(0) + 0(-30) + 0(1) \\ -0.8660(200) + 0.5(0) + 0(-30) + 0(1) \\ 0(200) + 0(0) + 0(-30) + 1(1) \end{bmatrix} = \begin{bmatrix} 30 \\ -100 \\ 173.2 \\ 1 \end{bmatrix}$$

因此顶点 A 的新位置为 A'(30, -100, 173.2)。

5. 对顶点 B 和 C 重复步骤 4, 得到对应的新位置 B'(150, -43.3, 25)、C'(100, -37.32, -24.64)。通过旋转这三个顶点, 便旋转了整个三角形。

对组合变换进行编程是本章最后一个重要内容, 它需要你对前面介绍的知识有一个完整的认识。

所要做的第一步就是建立组合变换矩阵, 幸运的是我们已经具备了建立它的所有知识, 这里有一个函数, 它可以用来建立一个 3D 组合变换矩阵。

```
Matrix4x4 createRotationCombo(float thetax, float thetay, float thetaz)
{
    Matrix4x4 X,Y,Z,temp,result;

    X = createFixed4x4Matrix(0.0f);
    Y = createFixed4x4Matrix(0.0f);
    Z = createFixed4x4Matrix(0.0f);
    temp = createFixed4x4Matrix(0.0f);
    result = createFixed4x4Matrix(0.0f);

    //将绕 X 轴旋转的值放置到矩阵中
    X.index[0][0] = 1;
```



```

X.index[1][1] = cos( DegreesToRads( thetax ) );
X.index[2][2] = cos( DegreesToRads( thetax ) );
X.index[3][3] = 1;

X.index[2][1] = -1 * ( sin( DegreesToRads( thetax ) ) );
X.index[1][2] = sin( DegreesToRads( thetax ) );

//将绕 Y 轴旋转的值放置到矩阵中
Y.index[0][0] = cos( DegreesToRads( thetay ) );
Y.index[1][1] = 1;
Y.index[2][2] = cos( DegreesToRads( thetay ) );
Y.index[3][3] = 1;

Y.index[2][0] = -1 * ( sin( DegreesToRads( thetay ) ) );
Y.index[0][2] = 1 * sin( DegreesToRads( thetay ) );

//将绕 Z 轴旋转的值放置到矩阵中
Z.index[0][0] = cos( DegreesToRads( thetaz ) );
Z.index[1][1] = cos( DegreesToRads( thetaz ) );
Z.index[2][2] = 1;
Z.index[3][3] = 1;

Z.index[0][1] = -1 * ( sin( DegreesToRads( thetaz ) ) );
Z.index[1][0] = sin( DegreesToRads( thetaz ) );

//创建组合矩阵
temp = multiply4x4Matrices( Y, X );
result = multiply4x4Matrices( temp, Z );
return result;

```

对于上面这个函数，其中的大部分内容看起来都很熟悉，因为在前面的章节中介绍了如何建立绕各个轴的旋转矩阵，在这里我们只是将其拿过来并加入到该函数中而已。当初初始化好各个矩阵之后，便利用矩阵乘法将其按照从右到左的顺序依次相乘，在这里我们使用了一个局部变量作为桥梁，其实你大可不必将值赋给一个中间矩阵，从而加快程序的处理速度，不过请你一定要注意它们相乘的次序。

该函数的输出是最终的组合变换矩阵，当然为了完成对物体的旋转，下面还要进行一个转换过程，既然我们已经完成了最困难的步骤，那就接着看一下最后的转换过程吧。

```

Matrix4x1 rotate3DWithCombo( Matrix4x4 combo, Matrix4x1 vertex )
{
    Matrix4x1 temp;
    temp = multiplyMatrixNxm( combo, vertex );
    return temp;
}

```

让我们接着往下处理，将每个顶点与组合矩阵相乘，请不要忘了绕各个轴的旋转值已经包含在组合矩阵中了，接下来就可以得到各顶点的新位置，再往下面就与前面例子的处理一样了。

当然，编程者们一直都在寻找是否存在数学上的捷径。不幸的是，对于组合变换矩阵来说，所有的数值都纠缠在一起，可以说没有捷径可走，得到正确结果的唯一方法是，分别建立单个矩阵，然后按照次序将其相乘。但是这里也有一些好消息，当你对其进行调试的时候一个有趣的形式将会出现，让我们还以前面的例子进行讨论，要完成绕三角形的中心旋转 90° ，它的组合变换矩阵为：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 168 \\ 1 & 0 & 18 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

如果仔细观察这个矩阵，你将会发现关于旋转的数值都保存在左上角 2×2 区域中，而平移数值则保存在最后一列中。

尝试去建立更多的组合矩阵，你将会发现下面的规律：

2D 组合变换矩阵

$$\begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

其中， r 存放着关于缩放和旋转的信息， t 则存放着关于平移的信息。

注意变换矩阵的最后一行总是 $(0, 0, 1)$ ，这也就是说当你在进行调试的时候，应该首先检查最下面一行，因为这比较简单。如果最后一行没有问题，再去检查上面两行，如果物体进行了平移，那就去检查最后一列；如果物体发生了缩放或旋转，那就需要去检查前两列。这种处理方法可以避免对矩阵进行多余的检查，从而节省更多的时间。

对于 3D 组合变换矩阵具有同样的形式。

3D 组合变换矩阵

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

其中， r 存放了关于缩放和旋转的信息； t 保存了关于平移的信息。

同样的，关于缩放和旋转的信息都保存在左上角 3×3 的区域中，这可以由例 6-11 和例 6-12 得到验证；同样的，最后一列保存了关于平移的信息。你也许注意到了另外一个规律，矩阵的第一行与 x 值对应，第二行与 y 值对应，第三行与 z 值对应，而最后一行总是 $(0, 0, 0, 1)$ ，这些规律可以帮助你进行程序的调试。

在本节的最后我们想介绍一下矩阵的转置，你也许需要回顾一下第 5 章的相关内容。首先让我们来看 2D 矩阵方程，一个典型的组合矩阵应该具有下面的形式：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

你也许也注意到了，在本章中我们总是使用一个单列的矩阵代表顶点的坐标，使用这种形式非常不错，因为像 3D 图形开发包 OpenGL 以及传统数学都采用单列的形式代表顶点。然而，对于另外一个 3D 图形开发包 DirectX 而言，它则使用单行的形式来表示顶点，因而在实际中就要根据情况进行必要的形式转换。

最后进行矩阵转置的时候，第一步就是要将一切的数值进行转置，即将矩阵行列的值互换。注意，矩阵相乘的次序非常重要，当顶点坐标采用单列的形式时，我们将一个 3×3 的矩阵与一个 3×1 的矩阵相乘，然而当对矩阵进行转置后就变成一个 3×3 矩阵乘上一个 1×3 矩阵了，显然这种矩阵相乘形式是没有定义的。因此在进行矩阵转置时，还要进行第二步工作，即将矩阵相乘的次序颠倒一下，经过这两步就可以得到下面的矩阵方程，它适合 DirectX 的顶点定义形式。

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} r_{00} & r_{10} & 0 \\ r_{01} & r_{11} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

在 OpenGL 和 DirectX 形式之间进行转换

$AB = B^T A^T$ ，其中 A 、 B 都为矩阵

$$\begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} r_{00} & r_{10} & 0 \\ r_{01} & r_{11} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

在 3D 空间中进行矩阵转置的方法与上面类似，需要注意的是，当把矩阵转置以后一定要将其相乘顺序颠倒一下，否则相乘的形式是没有定义的。

注意

即使你不打算使用 OpenGL，你也会发现在建立组合矩阵时采用单列的形式是比较容易的，只不过需要在最后将其转换为单行形式，毕竟从左到右进行矩阵相乘会比较容易一些。

在本章的最后，你将发现本章前面的内容是在为最后建立组合变换矩阵打基础。本章首先利用矩阵加法实现了平移变换，再次说明一下，如果你只进行平移变换的话就使用矩阵加法；如果想要进行缩放或者旋转变换的话就应该使用矩阵乘法。再一个就是，记住矩阵相乘的次序，将顶点的原始位置放到最右边，然后依次将变换矩阵放到左边，最后相乘的时候按照从左到右的顺序进行。最后，如果你采用 DirectX 的形式一定要将顶点坐标转换成单行形

式, 最终你将发现不管使用 OpenGL 还是 DirectX, 对顶点进行变换是所有 3D 游戏中最重要的操作。

自我测试

1. 建立一个组合变换矩阵, 将一个 2D 物体绕其自身中心(10, 50)旋转 -90° 。
2. 使用第 1 题中的变换矩阵, 旋转一个三角形, 其顶点分别为: $A(-10, 50)$ 、 $B(90, 0)$ 和 $C(-50, 100)$ 。
3. 建立一个组合变换矩阵, 使一个 2D 物体绕其自身中心(x_c, y_c)旋转 -90° 。
4. 把第 3 题中的矩阵转换成 DirectX 形式。
5. 建立一个组合变换矩阵, 使一个 3D 物体绕 Z 轴旋转 45° , 绕 X 轴旋转 90° , 绕 Y 轴旋转 -90° ; 然后利用它旋转一个三角形, 其顶点分别为: $D(200, 0, -30)$ 、 $E(0, 50, -150)$ 和 $F(40, 20, -100)$ 。
6. 把在第 3 题中建立的变换矩阵转换成 DirectX 形式。

6.5 可视化体验

在随书光盘中有一个名叫 Transformations 的演示程序, 下面是对它的一个简单介绍:

(1) 设计该演示程序的目的是帮助读者理解 2D 物体变换的原理, 它通过矩阵运算对网格中的方块进行旋转和平移变换, 读者可以通过该演示程序掌握组合变换的过程。

(2) 用户可以在网格中增加、选择、移动和跟踪方块, 同时还可对它进行取消和重做等操作。在体验的过程中, 你可以随时观察方块的当前位置信息。

——迈克尔·韦格特 与 迈克尔·弗斯特

该演示程序也具有交互性, 你可以在变换矩阵上加上一些值, 然后看看物体到底移动到了哪个位置。你也可以手动设置一些例子, 然后看看演示程序是否按照你所期望的变换发生了运动。

你可以通过双击文件 Transformations.exe 来运行该演示程序, 一旦程序被成功启动, 你将看到一个窗口, 它包含一些红色网格线, 而且在中央会有一个彩色盒子。你可以通过矩阵变换将该盒子移动到窗口的其他位置, 与此同时该演示中还有一个工具条, 如图 6-11 所示。

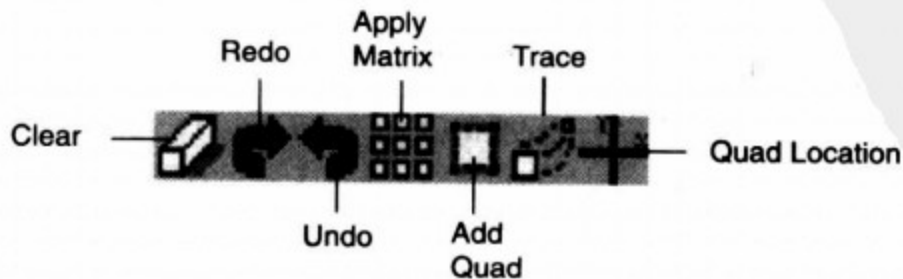


图 6-11 演示程序的工具条

首先单击 Apply matrix(应用矩阵)按钮, 随后会弹出一个 3×3 的矩阵, 并输入如下值:

$$\begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

那么该盒子将会向右移动5个单位，并向上移动8个单位。

单击工具条上 Trace(跟踪)按钮，这时将会看到盒子移动的轨迹。

单击工具条上 Add(添加)按钮，这时将会在坐标原点的左下角增加一个盒子。

这次试着将该盒子向左移动5个单位，向上移动3个单位。进行此操作之后，让盒子保持被选中状态(轮廓线为白色)，跟踪其位置，并实施下面的矩阵变换：

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

这将发生什么样的变换呢？当然了，它将相对于原点被缩放到原来的3倍，这也就意味着它在放大的同时并远离了原点。单击 Undo(撤销)按钮，可以观察其先前的原始位置。为了能够在进行该缩放变换的同时而不会远离原点，这将会是一个具有3步的组合变换过程，其矩阵为：

$$\begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}$$

将上述矩阵按照正确的顺序相乘，你将得到一个最终的组合变换矩阵，如下：

$$\begin{bmatrix} 3 & 0 & 10 \\ 0 & 3 & -6 \\ 0 & 0 & 1 \end{bmatrix}$$

这时再对方块进行缩放变换，就是相对于方块自身中心而不是坐标系原点了。如果你的屏幕窗口中的图形变得十分杂乱，可以单击 Clear(擦除)按钮将屏幕擦干净；否则就新建一个盒子，为下一次进行变换做准备。

这次利用下面的矩阵，将盒子向右移动9个单位，并向上移动4个单位：

$$\begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

跟踪盒子的位置，然后利用下面矩阵将其旋转90°：

$$\begin{bmatrix} \cos 90^\circ & -\sin 90^\circ & 0 \\ \sin 90^\circ & \cos 90^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

这又会发生怎样的运动呢？盒子看起来向上和向左发生了平移，其实这并不是变换的全部，仔细观察一下盒子的蓝色角，并分别单击 Undo(撤销)和 Redo(重做)按钮，你将会发现它确实旋转了90°。另外，你可以单击工具条上的 Quad Position(方块位置)按钮查看方块四

个顶点的坐标值，记下坐标的新位置，然后单击 Undo(撤销)按钮重新观察坐标的旧位置，你将发现盒子确实按照该方式发生了旋转。因此，盒子确实相对于坐标系原点发生了旋转。

尝试让盒子绕红色的角进行旋转，在该情形下，它会看起来保持在原地并发生了翻转。如果目前不在其旧位置，就单击 Undo(撤销)按钮返回。现在你可以看一下红色角的坐标值，应该是(9, 4)。下面就让我们来建立一个组合变换矩阵，让其绕红色顶点旋转 90° 。

$$\begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -9 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 13 \\ 1 & 0 & -5 \\ 0 & 0 & 1 \end{bmatrix}$$

你得到预期的结果了吗？建议你尝试进行多次组合变换，单个例子并不能说明问题。同样你可以首先手动实施一些组合变换，然后利用该演示对结果进行验证。首先记下四个顶点的原始位置，并计算出其新位置，然后验证一下该演示程序是否将其移动到了正确的位置。希望它们得到的结果是一致的。

本章总结

在本章中，我们利用前面学习的知识实现对屏幕上的物体进行变换。首先使用矩阵加法完成了简单的平移变换，然后利用矩阵乘法再次实现了该操作，因而能够与其他的变换进行组合。接下来我们讨论了如何对屏幕上的物体进行缩放和旋转，最后介绍如何通过矩阵的级联完成对一系列变换的组合，并且讲述了在行形式和列形式之间的转换。

能力测试

选择题

- 如果一个物体的中心坐标为(50, 25)，现将其向左移动 25 个像素，并向上移动 200 个像素，那么物体新位置的坐标是多少？
A. (50, 25) B. (25, 200) C. (-25, 200) D. (25, 225)
- 利用矩阵加法进行 3D 空间内的平移变换，需要分别进行多少次乘法和加法运算？
A. 0 次乘法，1 次加法 B. 1 次乘法，3 次加法
C. 0 次乘法，3 次加法 D. 3 次乘法，3 次加法
- 利用矩阵乘法进行 3D 空间内的平移变换，需要分别进行多少次乘法和加法运算？
A. 3 次乘法，3 次加法 B. 3 次乘法，2 次加法
C. 9 次乘法，6 次加法 D. 9 次乘法，9 次加法
- 如果一个物体的中心坐标为(50, 25)，现在利用一个单一的缩放矩阵将其均匀缩放 2 倍，那么物体新位置的坐标是多少？
A. (100, 50) B. (50, 25) C. (200, 100) D. (-50, -25)
- 现在要对一个 3D 物体绕任一坐标轴进行旋转变换，要求该变换相对于原点进行，那么需要多少个变换矩阵？

A. 1 B. 3 C. 5 D. 7

6. 现在要对一个3D物体绕任一坐标轴进行旋转变换, 要求该变换相对于另外一点进行, 那么需要多少个变换矩阵?

A. 1 B. 3 C. 5 D. 7

练习

练习1: 平移变换

1. 建立一个矩阵方程(利用矩阵加法), 将一个3D物体向右平移20个像素、向下移动100个像素、向后(屏幕后面)移动0个像素。然后利用该矩阵平移一个三角形, 其顶点分别为: $G(150, -20, -40)$ 、 $H(80, 0, -30)$ 和 $J(-200, 20, -50)$ 。
2. 建立一个矩阵方程(利用矩阵乘法), 将一个2D物体向左平移300个像素、向上移动10个像素。然后利用该矩阵平移一个三角形, 其顶点分别为: $L(-200, 20)$ 、 $M(60, -90)$ 和 $N(80, 0)$ 。
3. 利用矩阵乘法重做第1题, 看它们得到的结果是否一致?

练习2: 缩放变换

1. 建立一个矩阵方程, 将一个2D物体均匀缩放到其一半大小, 然后利用该矩阵缩放一个三角形, 其顶点分别为: $A(-70, 30)$ 、 $B(-20, 40)$ 和 $C(-35, 45)$ 。
2. 建立一个矩阵方程, 使一个2D物体看起来变得既高又扁, 其中在 x 轴方向上的缩放因子为 $1/4$, y 轴方向上的缩放因子为3。然后利用该矩阵缩放一个三角形, 其顶点分别为: $A(-40, 15)$ 、 $B(-20, 40)$ 和 $C(-40, 50)$ 。
3. 建立一个矩阵方程, 将一个3D物体缩放到原来的20倍。然后利用该矩阵缩放一个三角形, 其顶点分别为: $D(0, 40, -100)$ 、 $E(-60, 110, -30)$ 和 $F(-30, 0, -400)$ 。
4. 建立一个矩阵方程, 将一个3D物体在 x 轴方向上缩放2倍, 在 y 轴方向上无变化, 在 z 轴方向上缩小到原来的一半。然后利用该矩阵缩放一个三角形, 其顶点分别为: $D(0, 35, -100)$ 、 $E(-55, 100, -20)$ 和 $F(-25, 0, -300)$ 。

练习3: 旋转变换

1. 建立一个矩阵方程, 将一个2D物体旋转 30° , 然后利用该矩阵旋转一个三角形, 其顶点分别为: $A(-15, 55)$ 、 $B(85, 0)$ 和 $C(-55, 110)$ 。
2. 建立一个矩阵方程, 使一个3D物体保持在原地, 并每次绕 y 轴旋转 1° 。然后利用该矩阵旋转一个三角形, 其顶点分别为: $D(0, 10, -40)$ 、 $E(-10, 0, 20)$ 和 $F(35, 25, 0)$ 。
3. 建立一个矩阵方程, 将一个3D物体绕 x 轴旋转 45° , 然后利用该矩阵旋转一个三角形, 其顶点分别为: $G(0, 25, -150)$ 、 $H(-40, 100, -20)$ 和 $J(-20, 0, -200)$ 。

练习4: 组合变换

1. 建立一个组合变换矩阵, 使一个2D物体相对其自身中心 $(10, 50)$ 旋转 -90° 。
2. 利用在第1题中建立的矩阵, 旋转一个三角形, 其顶点分别为 $A(-10, 50)$ 、 $B(90, 0)$ 和 $C(-50, 100)$ 。

3. 建立一个组合变换矩阵, 能够使任何一个 2D 物体相对其自身中心(x_c, y_c)旋转 -90° 。
4. 将你在第 3 题中得到的结果转换成能够在 DirectX 中使用的形式。
5. 建立一个组合变换矩阵, 让一个 3D 物体分别绕 z 轴旋转 45° 、绕 x 轴旋转 90° 、绕 y 轴旋转 -90° 。然后使用该矩阵对一个三角行实施变换, 其顶点分别为: $D(300, 0, -60)$ 、 $E(0, 40, -100)$ 和 $F(40, 20, -80)$ 。
6. 将你在第 3 题中得到的结果转换成能够在 DirectX 中实现的形式。

程序设计

程序设计 1: 获得物体的顶点信息

1. 请写一个简单程序, 请求物体的顶点信息, 然后以用户期望的平移方式, 返回物体所有顶点的新位置。
2. 请写一个程序, 请求一个 2D 三角形的顶点信息, 并绘制出该三角形, 然后根据用户期望的平移方式绘制出该三角形的新位置。
3. 请写一个程序, 请求一个 2D 三角形的顶点信息, 并绘制出该三角形; 然后根据用户输入的相对于原点进行缩放或旋转的具体信息, 并绘制出三角形的新位置。
4. 请写一个程序, 请求一个 2D 三角形的顶点信息, 并绘制出该三角形; 然后根据用户输入的相对其自身中心进行缩放或旋转的具体信息, 并绘制出三角形的新位置。

程序设计 2: 建立变换矩阵

1. 建立一系列变换矩阵, 利用在视觉体验中的变换演示, 将结果用图形显示出来。
2. 请写一个程序, 请求一系列变换, 然后按照给定顺序将其转换成一个单个组合变换矩阵。

程序设计 3: 建立一个进行平移、缩放和旋转的函数库

使用矩阵乘法, 为平移、缩放和旋转变换建立一个函数库。

程序设计 4: 转换矩阵形式

请写一个程序, 在矩阵方程的行形式与列形式之间进行转换。

问题讨论

1. 在进行平移变换中, 为什么使用向量加法会比矩阵乘法速度更快?
2. 当我们在进行缩放或旋转变换时, 说“相对于”另外一点是什么意思?
3. 为了使建立的矩阵方程进行乘法运算少于 9 次、加法运算少于 6 次, 我们应该怎样对其进行优化?
4. 为什么在组合一系列变换时次序非常重要?
5. 对于矩阵方程的行形式与列形式, 它们之间有什么不同?
6. 检查一下本章的示例程序代码, 看能不能对其进行一些优化。

第7章 单位换算

本章目标

认真学习这一章并完成课后练习，你将掌握以下内容：

- 在公米制之间进行转换
- 在不同单位之间进行转换
- 在十进制和二进制之间进行转换
- 在计算机存储容量间进行转换

介绍

当我们准备进入这本书的物理知识部分之前，还需要学习最后一个数学知识：单位换算。在物理中，我们常常会处理到一些物理量，比如速度和位移，它们都有特定的单位，如果在程序中忽视了这些单位，那么程序就会出现错误。你可能还记得，1999 年秋美国国家航空和宇宙航行局发射火星气象卫星失败了，这次失误其实是由于在对系统进行编程时，一个工程小组使用英国单位制，而另一个小组则使用了世界公制，最后导致卫星发射升空后，在太空中失去了方向。现在可以假想一下，如果在游戏中出现了单位不一致的情况，后果会有哪些呢？如果开发小组中的某个成员使用米作为单位，而另一成员则采用英里，那么当链接程序时就会发生错误。假如在编程中都不考虑单位，那么大部分时间都会花费在检查错误上，因此本章我们就来学习一下这个很小而又非常重要的单位换算问题。

7.1 公米制

如果你不是居住在美国，那么你可能经常会使用公米制单位，而不是英制单位。本节将首先介绍如何使用公米制，此后你将会经常用到它们，而且在学习该部分内容之后，你将发现公米制使用起来非常方便，因为它的运算是以十进制为基础的。

在编写的程序中，计算机是不能区别单位的，比如米和厘米，也就是说它识别的只是数字，因此对于一个编程人员来说，必须要在所写的程序中采用统一的物理量单位。在游戏中无论你使用的单位是米、千米、英里还是码，只要它们是统一、一致的，那么玩家就不会受到影响。让我们从学习公米制开始，这是因为它比较简单和容易处理。在游戏中如果采用公米制单位，就会减少许多麻烦，而且许多游戏引擎的插件也都是采用公米制作为输入的，比如 Havok 图像引擎，因此我们要熟练掌握公米制。

假如你要跟踪游戏中人物角色的移动，当角色在奔跑时你可能会使用米作为它移动位移的单位，而当他移动缓慢时使用厘米作为位移的单位，这样处理起来会比较容易。假如角色跑了 10 米，然后缓慢移动了 75 厘米，现在直接将这两个位移加在一起是没有意义的，因此在相加之前必须要将其中一个单位换算成另一个单位，至于选择转换成哪个单位关系不大，只要保证它们前后统一即可。

在公米制之间进行转换是比较容易的，因为它们之间的转换是采用十进制进行的，完成它们之间的转换只需移动小数点在数值中的位置即可。下面列出了一些公米制的前缀，这些前缀的后面跟着米或克。

注意

1 米和 1 码的长度差不多，它们比 3 英尺长一些。

1 克大约是一个图钉的质量，一个硬币的质量差不多是 5 克。

公米制

kilo-	hecto-	deca-	meters, grams	deci-	centi-	milli-
1000	100	10	1	0.1	0.01	0.001

注意

这些前缀在重量换算时也一样。

当我们沿着这个表向右边的单位换算时，就乘以 10 或者将小数点向右移一位。比如，若想将 10 米转换成厘米，就向右移动两次，即将小数点向右移两位，就会得到 1000 厘米；同样如果想将 75 厘米转换成米，就将小数点向左移两位，即得到 0.75 米。这就是公米制的妙处所在，由于公米制采用十进制运算，当每次向左或向右移动小数点时，就等于将数值乘以或除以 10，这很像我们所熟悉的十进制运算。

【例 7-1】 把米转换成千米

假设游戏中的角色在跑了 50 米之后，跳上一辆汽车，然后车又行驶了 25 千米，那么该角色总共前进了多少千米？

解答：

1. 若想要对两个位移量相加，那么它们必须要有相同的单位。此时，我们可以将 50 米转换成千米，查看前面的前缀表，千米是米的左边第三个，所以将小数点向左移动 3 个位置，即 50 米等于 0.05 千米。

2. 现在就可以将这两个数相加了，因此总位移是 25.05 千米。

【例 7-2】 把千米转换成米

如果在你的编程小组中，其余的人使用米作单位，那么例 7-1 中的游戏角色总共前进了多少米？

解答：

1. 可以首先将 25 千米转换成米，然后再将两个数相加。也可以直接将结果 25.05 千米转换成米。

2. 在这里我们采用第二种方法。由于米是千米的右边第三个，因此将小数点向右移3个位置，即得到25050米。

通过上面两个例子的介绍，就可以知道公米制的精妙之处了。在公米制之间进行转换其实就是小数点移动的过程，或者只需将值数乘以或除以10的乘方即可。后面我们将介绍英制单位间的转换，以及公米制与英制之间的转换，与前面相比它们会比较复杂一些。

自我测试

1. 将35米转换成厘米。
2. 将35米转换成千米。
3. 将4567毫米转换成千米。
4. 将56克转换成千克。
5. 将3千克转换成克。

7.2 不同单位制间的转换

不同单位制间的转换比公米制间的转换复杂一些，因此建议大家采用公米制，即用米作为距离的单位，用千米每小时作为速度的单位。假如你在美国长大，就会倾向于使用英制单位，比如采用英里每小时作为速度的单位。

在游戏编程中，我们经常会处理关于长度和时间的问题，因此在这里将着重介绍这些常用单位的转换。在游戏中你经常会在英制单位中进行转换，也可能在英制和公米制间进行转换，其实无论哪种转换，它们的过程都是相同的。

在进行单位转换时，具有正确的工作秩序是非常重要的。我们首先应该明白，如果将一个数乘以1，这个数是不变的。因此在单位转换时，将一个数值多次乘以1，那么它的大小是不会改变的。请记住，如果一个分数的分子和分母相等，那么这个分数就等于1，比如分数 $3/3$ 等于1，分数 $(60\text{秒}/1\text{分钟})$ 等于1。在它们之间的转换非常容易，只需将原数值乘上一个转换因子就可以了。比如60分钟/1小时，我们都知道60分钟等于1小时，所以这个分数就等于1，其实它就是一个转换因子，代表了两个不同单位间的转换关系，你可以利用这个特殊的转换因子完成小时和分钟的转换。例如把3小时转换成分钟，只需要将其乘上60分钟/小时：

$$3\text{hrs}\left(\frac{60\text{min}}{1\text{hr}}\right) = 180\text{min}$$

进一步来看转换因子(60分钟/小时)，如果将它的分子和分母颠倒，它仍是一个转换因子，此时它可以把分钟转换成小时。假如你不知道应该使用哪种方法进行转换，那么就首先看它的单位，当作乘法运算时总是希望旧的单位被新的单位取代。因此可以将原始的数值乘上一个值为1的转换因子，只要这个值等于1并且能够消去旧单位即可。表7-1列出了一些可能会用到的转换因子。

表 7-1 转换因子

长 度	时 间
1 米 = 39.37 英寸 = 3.281 英尺	1 秒 = 1000 毫秒
1 英寸 = 2.54 厘米	1 分钟 = 60 秒
1 千米 = 0.621 英里	1 小时 = 3600 秒
1 英里 = 5280 英尺 = 1.609 千米	1 年 = 365.242 天

当我们在编程中需要处理这些转换时,最好的做法是定义一些常量宏,以用于在代码中频繁调用。例如:

```
#define KM_ TO_ MTLES 1.609
#define KM_ TO_ METERS 1000
#define DAYS_ TO_ WEEK 7
#define HOURS_ TO_ DAY 24
#define MIN_ TO_ HRS 60
#define SEC_ TO_ MIN 60
```

上面定义的这些常量可以由表 7-1 得到,当然也可通过其他的资料得到. 将这些常量定义成宏,可以方便于在游戏中进行计算,当然我们也可以将游戏中所有需要使用的值定义成常量. 这时需要注意一个问题,当游戏是以地球为背景时,这些转换因子将是正确的,假如游戏不是以地球为背景比如土星,那么此时的一天要比地球上的一天多 10.5 个小时。

理解这些概念最好的方法是多练习一些例子,首先来看一个关于时间转换的简单的例子。

【例 7-3】 时间转换

2 周有多少秒?

解答:

1. 从 2 周这个初始量开始。

2. 把所有与周和秒转换有关的转换因子列出来,将其写成一行,这样做可以减少相乘时出现错误。

$$\frac{2\text{wks}}{1} \left(\frac{7\text{days}}{1\text{wk}} \right) \left(\frac{24\text{hrs}}{1\text{day}} \right) \left(\frac{60\text{min}}{1\text{hr}} \right) \left(\frac{60\text{s}}{1\text{min}} \right)$$

3. 保证这些分数相乘后能够如你所愿消去相应的单位。

4. 将所有的分数相乘:

$$\frac{2\text{wks}}{1} \left(\frac{7\text{days}}{1\text{wk}} \right) \left(\frac{24\text{hrs}}{1\text{day}} \right) \left(\frac{60\text{min}}{1\text{hr}} \right) \left(\frac{60\text{s}}{1\text{min}} \right) = 1209600\text{s}$$

在前面我们曾提到,如果使用英制单位,那么在编程中就要进行英制和公米制单位之间的转换,让我们来看这样一个例子。

下面是一个将天转换成秒的函数. 看看你能否根据这个函数导出其他类似的时间转换函数:


```
float convertDaysToSeconds(float days)
{
    float seconds;
    seconds = days * HOURS_TO_DAY * HOURS_TO_MIN * MIN_TO_SEC;
    return seconds;
}
```

在程序代码中使用常量能够使程序变得整洁，这是一个提高程序可读性的好方法。

【例 7-4】 距离转换

假设在游戏中，你使用英里作为距离单位，而小组内的其他人员采用公米制单位。现在游戏中的角色跑了 5 英里，那么他跑了多少米？

解答：

1. 从 5 英里这个初始量开始。

2. 把所有与英里转换成米有关的转换因子列出来，将它们写成一行，这样做可以减少相乘时出现错误。

$$\frac{5\text{mi}}{1} \left(\frac{1.609\text{km}}{1\text{mi}} \right) \left(\frac{1000\text{m}}{1\text{km}} \right)$$

3. 确保这些分数相乘时能够如你所愿消去相应的单位。

4. 将所有的分数相乘：

$$\frac{5\text{mi}}{1} \left(\frac{1.609\text{km}}{1\text{mi}} \right) \left(\frac{1000\text{m}}{1\text{km}} \right) = 8045\text{m}$$

下面是一个将英里转换成米的函数：

```
float convertMilesToMeters(float miles)
{
    float meters;
    meters = miles * KM_TO_MILES * KM_TO_METERS;
    return meters;
}
```

注意

本书和其他书一样，在标记单位时，用“mi”表示英里，用“m”表示米。

当所要处理的单位是含有分式的量时，过程会变得复杂起来。例如，速度可以用英里每小时表示，许多错误都是在对分母中的小时进行转换时发生的，避免这类错误的一种方法就是将 55mi/hr 写成 55mi/1hr，这可能会有助于你列出所需要的转换因子，例 7-5 说明了这个问题。

【例 7-5】 速度转换

假设在赛车游戏中，汽车的运行速度是 180 英里/小时，但是游戏小组决定采用公米制来编程，那么该车的速度是多少米/秒？

解答:

1. 从 180 英里/小时这个初始量开始。
2. 把所有与这两个转换有关的转换因子列出来。将分子中的英里转换成米, 分母中的小时转换成秒。将这些分数写成一行, 这样做可以减少相乘时出现错误。

$$\frac{180\text{mi}}{1\text{hr}} \left(\frac{1.609\text{km}}{1\text{mi}} \right) \left(\frac{1000\text{m}}{1\text{km}} \right) \left(\frac{1\text{hr}}{60\text{min}} \right) \left(\frac{1\text{min}}{60\text{s}} \right)$$

3. 确保这些分数相乘时能如你所愿消去相应的单位, 注意要保持分母是 1 小时, 这样就很容易列出最后两个转换因子。

4. 将所有分数相乘:

$$\frac{180\text{mi}}{1\text{hr}} \left(\frac{1.609\text{km}}{1\text{mi}} \right) \left(\frac{1000\text{m}}{1\text{km}} \right) \left(\frac{1\text{hr}}{60\text{min}} \right) \left(\frac{1\text{min}}{60\text{s}} \right) = 80.45\text{m/s}$$

该转换也很容易用程序实现, 见下面函数:

```
float convertSpeed(float miles)
{
    float meters;
    meters =
        (miles * KM_TO_MILES * KM_TO_METERS) / (HOURS_TO_MIN * MIN_TO_SEC);
    return meters;
}
```

在单位转换中, 还有最后一个麻烦, 即如果单位中出现平方如加速度单位 m/s^2 , 记住 s^2 就是 $\text{s} \times \text{s}$ 而不仅仅是秒, 这就意味着在进行单位换算时, 对这两个秒都要进行转换。下面来看一个关于这个的例子。

【例 7-6】 加速度转换

假设在赛车游戏中, 一辆汽车以 20000mi/hr^2 的加速度前进, 但是编程小组决定使用公制单位, 那么该汽车的加速度用 m/s^2 表示是多少?

解答:

1. 从 20000mi/hr^2 这个初始量开始, 将它写成 $20000\text{mi}/1\text{hr}^2$, 时间单位的平方仍在分母中。
2. 列出所有相关的转换因子, 这些转换因子会将英里转换成米, 小时平方转换成米平方。保证将小时进行两次转换成秒, 这是由于单位是 hr^2 , 不仅仅是 hours。并确保这些分数在相乘时能如你所愿消去单位。保证将这些分数写成一行, 以减少相乘时出现错误。

$$\frac{20000\text{mi}}{1\text{hr}^2} \left(\frac{1.609\text{km}}{1\text{mi}} \right) \left(\frac{1000\text{m}}{1\text{km}} \right) \left(\frac{1\text{hr}}{60\text{min}} \right) \left(\frac{1\text{min}}{60\text{s}} \right) \left(\frac{1\text{hr}}{60\text{min}} \right) \left(\frac{1\text{min}}{60\text{s}} \right)$$

3. 将所有分数相乘。

$$\frac{20000\text{mi}}{1\text{hr}^2} \left(\frac{1.609\text{km}}{1\text{mi}} \right) \left(\frac{1000\text{m}}{1\text{km}} \right) \left(\frac{1\text{hr}}{60\text{min}} \right) \left(\frac{1\text{min}}{60\text{s}} \right) \left(\frac{1\text{hr}}{60\text{min}} \right) \left(\frac{1\text{min}}{60\text{s}} \right) = 2.48\text{m/s}^2$$

无论你需要转换什么类型的单位，处理过程都是相同的。将初始量乘以一系列值为 1 的分式，总会得到你所希望的单位。当在手动计算时要注意，将这些转换因子写成一行，这样做当你进行乘法时就会减少错误。另外，记住最后想要把它转换成什么单位，这一点在转换时很容易忘记。更重要的是，当你在一个游戏开发小组工作时，在开始编程之前一定要记住统一单位，单位不一致会引起严重的后果，就像美国航空航天局一样！

让我们看一下加速度转换是如何使用程序实现的，下面是一个将 MPH^2 转换成 m/s^2 的函数：

```
float convertAccelMilesToMeters(float miles){  
    float accelMeters;  
    accelMeters = (miles * KM_ TO_ MILES * KM_ TO_ METERS) /  
    (HOURS_ TO_ MIN * MIN_ TO_ SEC * HOURS_ TO_ MIN * MIN_ TO_ SEC);  
    return accelMeters;  
}
```

再次观察一下该公式，当我们知道更简单的转换时，应该优先使用它。例如，在这里我们可以直接将小时转换成秒，而不是将小时转换成分钟，再将分钟转换成秒。实例给了我们一种解决问题的简单方法，当我们在游戏编程时，尽量减少计算机的运算量，从而使程序运算速度加快，这是一个很大的挑战，请在你的代码中寻找这种机会。

自我测试

1. 3 天有多少分钟？
2. 10 英寸是多少厘米？
3. 假设赛车游戏中汽车行驶了 12 英里，但是编程小组决定用米作单位来编程，那么该汽车行驶了多少米？
4. 假设赛车游戏中汽车行驶速度为 225 英里/小时，但是编程小组决定用公制单位来编程，那么汽车的行驶速度是多少千米/小时？
5. 将问题 4 中速度转换成千米/小时之后，又发现游戏编程小组中的其他成员用米和秒作单位，那么 225 英里/小时是多少米/秒？
6. 9.8m/s^2 是多少 mi/hr^2 ？

7.3 计算机转换

我们常常听到人们说，这个文件的大小是 50 千字节，该硬盘的容量是 2 兆字节，这些数字听起来好像没有什么规律，这主要是因为我们不理解这些量之间的关系。其实它们是用来描述电脑储存或读取信息大小的单位。请记住计算机能够识别的只是二进制代码。

当你采用某种编程语言如 C++ 进行编程时，编程器就会将你编写的源代码编译成二进制代码，然后计算机根据生成的代码进行一系列的逻辑运算。二进制代码是由一系列 0 或 1

组成，它们决定了计算机内部电子元件的开关状态，每个开关都存放 1 比特的信息——0 或者 1，然后每 8 个二进制位形成一组，构成 1 字节。因此，1 个字节可以用来表示 0 和 255 之间的一个十进制数字。

在本章第一节中，我们知道十进制之间的换算是比较容易的，这是因为十进制运算的基值是 10，每个数字可用 10 的乘方来表示。你也许还记得在学校时学的数位，如个位、十位、百位等。现在假设每个数字用 2 的乘方来表示，而不是 10 的乘方，那么便可建立二进制系统。表 7-2 对十进制数和二进制数进行了比较。

表 7-2 十进制与二进制的比较

十进制							
10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
10 000 000	1 000 000	100 000	10 000	1 000	100	10	1
二进制							
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

在十进制中数字 702 可以表示为： $7 \times 100 + 0 \times 10 + 2 \times 1$ ，即 7 个 100 加上 0 个 10，再加上 2 个 1 得到。在二进制中，每个数字是 2 的乘方而不是 10 的乘方，这就意味着 10011010 在二进制中是 1 个 128 加上 0 个 64 加上 0 个 32 加上 1 个 16 加上 1 个 8 加上 0 个 4 加上 1 个 2 再加上 0 个 1 得到，该数值换算成十进制就是： $128 + 16 + 8 + 2 = 154$ 。表 7-3 列出了 10011010 的二进制数位。

表 7-3 二进制数 10011010 对应的数位

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	0
128	0	0	16	8	0	2	0

【例 7-7】 二进制转换成十进制

将二进制数 01001101 转换成十进制。

解答：

1. 最简单的方法是用下面的表来表示二进制各数位。

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	1	0	0	1	1	0	1
0	64	0	0	8	4	0	1

2. 将表最下面的一行数相加： $64 + 8 + 4 + 1 = 77$ 。

因此，二进制数字 01001101 转换成十进制数字就是 77。

从上表可以计算出, 如果一个二进制数的 8 位全都是 0, 那么该数转换成十进制也是 0; 如果所有的数位全是 1, 那么转换成的十进制数就是 255。由此可知, 每个二进制数可以代表 0 和 255 之间的一个十进制数。同理, 也可以将十进制数字转换成二进制, 下面的例子演示了该转换过程。

【例 7-8】十进制转换成二进制

将十进制数 185 转换成二进制。

解答:

1. 找到小于 185 的最大的 2 的乘方数, 即上表的第一列对应的数 128。
2. 185 减去 128 得到 57。
3. 用 57 重复上面的过程, 小于或等于 57 的最大的 2 的乘方数是 32。
4. 57 减去 32 得到 25。
5. 重复前面的过程, 直到最后得到 0。
6. 小于或等于 25 的最大的 2 的乘方数是 16。
7. 将 25 减去 16 得到 9。
8. 小于或等于 9 的最大的 2 的乘方数是 8。
9. 将 9 减去 8 得到 1。
10. 小于或等于 1 的最大的 2 的乘方数是 1。
11. 1 减去 1 等于 0, 此时计算停止。

12. 下面列出一个表, 将前面出现过的被减数的对应位置上写上 1, 其他跳过去的 2 的乘方数位置写上 0, 于是可得到下表:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	1	1	1	0	0	1

因此, 十进制数 185 的二进制形式是 10111001。

十进制与二进制间的转换程序

将二进制转换成十进制及其逆转换过程, 在程序代码上存在微小的差别。虽然它们在算法上是相似的, 但是必须要保证它们能够在计算机上运行, 并且能得到用户所期望的转换值。首先让我们看一下将十进制转换成二进制的程序代码:

```
void decimaltoBinary()
{
    int tobeconverted;
    int binaryplaceholder[32];

    cout << "Please enter a binary number in ones and zeros \n";
    cin >> tobeconverted;

    for(int i=0; i<32; i++)
```

```
{
    //将数值进行模 2 运算,得到 0 或 1

    //将数值除以 2,并赋值给自身
    tobeconverted = tobeconverted / 2;
}

cout << "Your decimal number converted to binary is:\n";

//确保以正确的顺序输出各个二进制位,
//即从后往前逆序输出数组各元素

for(int j = 31; j >= 0; j --)
{
    cout << binaryplaceholder[j];
}
}
```

首先该程序要求用户输入一个待转换的数值,然后对其进行模 2 运算,判断运算结果是偶数还是奇数,即对应二进制位为 0 或 1,并将得到的结果储存在变量 `binaryplaceholder` 数组中,以备在后面输出时使用。接下来将上一步得到的结果除以 2,再进行模 2 运算,然后再除以 2,重复该过程直到结果为 0。该函数的最后部分是从左向右输出二进制数的各位,为了保证二进制数的正确顺序,要求从后到前逆序输出数组各元素,该处理过程非常巧妙吧?

下面让我们来看看将二进制数转换成十进制数的过程,其过程与前面过程是相似的,但是它要使用到一些附加函数:

```
void binarytoDecimal ()
{
    char tobeconverted[9];
    int finalvalue = 0;
    int powerval = 0;

    cout << "Please enter a binary value (8 bits max) MSB first \n";
    cin >> tobeconverted;

    //循环处理第 0 - 7 位,共 8 位
    for(int j = 0; j <= 7; j ++ )
    {
        //由于使用 MSB 顺序,因此需要反转得到其幂
        powerval = pow(2, (7 - j));

        //检测该位是否为 1
        if (tobeconverted[j] == '1')
            finalvalue = finalvalue + powerval;
    }
}
```

```
cout << "the number from binary to decimal is << finalvalue << "\n";
}
```

该函数首先要求用户输出一个具有8位的二进制数，这里选择8位是为了使该转换更容易说明，当然你可以随意增加位数，其运算法则都是相同的。一旦将这个二进制数储存在数组中，就可以通过索引逐个访问各数位。然后依次将数字代入函数 `power()` 中进行运算，求得其2的乘方，然后判断该二进制位如果为1，就将 `powerval` 累加到最终值 `finalvalue` 上；如果该二进制位是0，则不必相加，因为 `finalvalue` 加上0仍等于原值。由于二进制数采用MSB顺序输入，即最先输入的数位对应数值的最高位，也就是说它是整个二进制数的幂指数最大的数位，这一点是很容易忽视的，如果将顺序弄反将会得到错误的结果。

在该函数中，转换顺序是一个隐含的错误陷阱，该问题在很多编程论坛上都提及过，在你的程序中一定要注意该问题，并进行正确的处理。

我们可以将比特单位转换成其他的二进制单位，让我们从最小的计量单位开始。在本章的前面曾提到，1字节的信息等于8个比特位，然而我们平常所用到的文档大都是由成千上万个字节的信息组成，处理如此大的数量是很麻烦的。因此我们常常将它们转换成千字节，即1千字节(KB)相当于1024字节。许多文件尤其是那些包含图像的文件容量都很大，因此又要将它转换成更大的单位兆字节，1兆字节(MB)等于1024千字节(KB)。再往更深层次看，硬盘驱动器能够储存大量的信息，描述它的最好的单位是千兆，1千兆字节(GB)等于1024兆字节(MB)。总结所有的信息，其转换如下：

- 1 字节 = 8 比特
- 1 千字节 = 1024 字节
- 1 兆字节 = 1024 千字节
- 1 千兆字节 = 1024 兆字节

注意

这些转换因子看起来很奇怪，别忘了现在处理的是二进制，它的基础值是2，这也就是转换因子必须为2的乘方的原因了。

在上面这些单位之间进行转换，其过程和前面讨论的过程是相同的，只不过此处的转换因子不一样而已，下面来看一个例子。

【例7-9】 计算机存储容量转换

如果你的计算机的硬盘驱动器能储存20GB的信息，那么转换成比特是多少？

解答：

1. 从原始量20GB开始。
2. 把所有与千兆字节转换成比特有关的转换因子列出来，保持这些分数写成一行，这样可以减少相乘时出现错误。

$$\frac{20\text{GB}}{1} \left(\frac{1024\text{MB}}{1\text{GB}} \right) \left(\frac{1024\text{KB}}{1\text{MB}} \right) \left(\frac{1024\text{B}}{1\text{KB}} \right) \left(\frac{8\text{b}}{1\text{B}} \right)$$

3. 保证这些分数相乘后，能够如你所愿消去相应的单位。
4. 所有分数相乘：

$$\frac{20\text{GB}}{1} \left(\frac{1024\text{MB}}{1\text{GB}} \right) \left(\frac{1024\text{KB}}{1\text{MB}} \right) \left(\frac{1024\text{B}}{1\text{KB}} \right) \left(\frac{8\text{b}}{1\text{B}} \right) = 1.718 \times 10^{11} \text{bits}$$

C++ 与众不同的特征之一是，程序运行时变量需要的存储空间必须由程序员自己分配和回收，因此你必须清楚地知道程序需要的空间是多少。如果只是粗略估计，可以转换因子 1000 即可，因为它比较简单；但是要在程序中分配内存，或者处理存储问题，就必须使用 1024 作为转换因子。如果程序中仅一个字节发生错误，那么程序就会出错，这也就是计算机转换非常重要的原因。

自我测试

1. 将二进制数字 10101010 转换成十进制数。
2. 将二进制数字 01101101 转换成十进制数。
3. 将十进制数 176 转换成二进制。
4. 将十进制数 205 转换成二进制。
5. 100MB 的压缩硬盘能储存多少比特的信息？
6. 如果你分配 4KB 的存储空间，它能分配多少字节？
7. 如果你有一个 12345678 比特大小的文件，那么它是多少兆字节？
8. 23753891429 字节有多少千兆字节？

本章总结

在这一章，我们讨论了单位转换的重要性，先后讨论了公制转换、不同单位间转换和计算机转换的问题。请务必记住，计算机不认识单位，它只知道数字而已，因此在程序中所有的量转换成相同的单位是编程人员的责任，至于你选择什么单位关系不大，只要保证它们一致就行了，在本书后面物理知识的相关章节，我们会再次使用到单位转换。

能力测试

选择题

1. 下列哪个选项与 8.5m 相等？
A. 8500mm B. 85cm C. 850hm D. 8500km
2. 如果游戏角色走了 12m，然后跑了 2km，他一共移动了多少距离？
A. 14m B. 14km C. 2.012m D. 2.012km
3. 下列与 750 英寸最接近的数是？
A. 0.02km B. 52.5km C. 3.54ft D. 0.02ft
4. 如果游戏角色步行了 12m，然后跑了 12mi，那么他总共移动了多少距离？
A. 28435ft B. 193.2m C. 19320m D. 193.2km
5. 下列哪个最接近 5m/s^2 ？

- A. 11.2mi/hr^2 B. 40273.5mi/hr^2 C. 0.0006mi/hr^2 D. 0.6mi/hr^2
6. 下列哪个十进制数等于 10011101?
- A. 164 B. 157 C. 128 D. 64
7. 下列哪个二进制数等于 96?
- A. 0110000 B. 0011000 C. 01100000 D. 01010000
8. 下列哪个最接近于 4.5GB?
- A. 4718592MB B. 4718592KB C. 4718592B D. 4718592b

练习

练习 1: 公制

1. 将 345 米转换成厘米。
2. 将 525 米转换成千米。
3. 将 9182 毫米转换成千米。
4. 将 3.25 克转换成千克。
5. 将 0.0058 千克转换成克。

练习 2: 不同单位间的转换

1. 23 天是多少分钟?
2. 63 英寸是多少厘米?
3. 假设在你的赛车游戏中, 一辆汽车行驶了 15 英里, 但是游戏小组决定用公制单位来编程, 那么汽车行驶了多少米?
4. 假设在你的赛车游戏中, 一辆汽车以 230mi/hr 的速度行驶, 但是游戏小组决定用公制单位来编程, 那么汽车的速度是多少 km/hr ?
5. 将问题 4 中的速度转换成 km/hr 之后, 又发现游戏小组其他成员都用米和秒来编程, 那么 230mi/hr 用 m/s 表示是多少?
6. 37m/s^2 是多少 mi/hr^2 ?

练习 3: 计算机转换

1. 将二进制数 00101110 转换成十进制。
2. 将二进制数 11001011 转换成十进制。
3. 将十进制数 135 转换成二进制。
4. 将十进制数 26 转换成二进制。
5. 如果你分配 10KB 的存储空间, 它能分配多少字节?
6. 如果你有一个 10478541 比特大小的文件, 那么它是多少兆字节?
7. 190530212385 字节有多少千兆字节?

程序设计

程序设计 1: 编写一个程序, 求出公制转换公式

编写一个程序, 要求用户输入一个公制单位, 然后输出所有相关的转换等式(从 km 到 mm)。

程序设计 2：编写一个程序，完成公米制单位的转换

编写一个程序，给定任何一个公米制或英制数值，然后请求用户指定要将其转换成哪种形式，然后输出转换后的数值。

程序设计 3：转换程序

1. 创建一个函数库，分别完成时间、距离、速度以及加速度的转换。
2. 仔细观察本章十进制和二进制之间转换的程序代码示例，对于该过程请找出一个不同的编程方法。

问题讨论

1. 为什么公米制单位容易处理？
2. 对于英制单位和公米制单位，你会选择使用哪一个进行编程？为什么？
3. 二进制数会大于 256 吗？
4. 比千兆字节大的单位是什么？



第8章 一维空间运动

本章目标

认真阅读本章内容并完成练习之后，你能够掌握以下内容：

- 计算速率和速度
- 计算加速度
- 利用运动方程描述一个物体的位移、速率和加速度与时间的关系

介绍

在视频游戏中，画面的更新就像一页一页地翻书一样，当程序处于主循环时，物体的移动是通过一帧一帧地播放画面来实现的，如果每帧的显示速度足够快，那么运动看起来就很真实。但是计算机又是怎么知道物体在一帧图像中的位置的呢？实际上，这可以由物理定律精确地确定。然而在现实世界中，有很多因素影响物体的运动，在程序中不可能把它们全部考虑在内，也就是说必须丢弃一些非重要因素，从而完成物理运动的模拟。本章我们开始介绍游戏中的物理知识，首先定义了一些基本物理量，然后讨论如何用数学方程模拟和控制运动。

8.1 速度和速率

物体只要移动就会有速率，速率用来表示物体运动的快慢。相应地，如果一个物体有速率，那么它就会有速度，速度是速率的向量形式，即速度是有方向的速率。此时若有疑问，可以翻开第4章复习一下矢量和标量的不同。在一维空间中，速度的方向可用正负号表示，第10章将会介绍怎样处理2D、3D中速度的方向，现在我们仅仅考虑物体在直线上的运动。当物体做直线运动时，速度为正值表示前进，而速度为负值则表示后退。

注意

我们上面提到的“速度”和“速率”，它们很容易混淆，不过你可以记住一点，那就是当使用速率时，其方向是可以忽略的；相反，使用速度时，它一定具有方向。

首先来看一个匀速运动的例子，假若我正在驾驶一辆汽车以60mi/hr的速度行驶，现在保持该速度行驶一小时，那么汽车将行使60英里；同样地，如果继续保持这个速率，行驶2个小时就会行驶120英里，3小时行驶180英里等。由此可推导出下列公式：

匀速运动的位移公式

$$\text{位移} = \text{速度} \times \text{时间} (D = v \times t)$$

其中, v 是匀速运动的速度

注意

如果在上述公式中忽略速度的方向, 对于标量仍有下面公式:

距离 = 速率 $\times$ 时间。

【例 8-1】 计算匀速运动的位移

假设游戏中的角色跳上了一辆汽车, 该车的速度恒定是 150 像素/秒, 那么 1 秒、2 秒、3 秒后它分别移动了多远?

解答:

1. 通过上面公式计算位移:

$$D = v \times t = (150\text{px/s}) \times t$$

2. 现在将时间 t 分别代入上述公式:

$$1 \text{ 秒后: } D = (150\text{px/s}) \times t = (150\text{px/s}) \times (1\text{s}) = 150\text{px}$$

$$2 \text{ 秒后: } D = (150\text{px/s}) \times t = (150\text{px/s}) \times (2\text{s}) = 300\text{px}$$

$$3 \text{ 秒后: } D = (150\text{px/s}) \times t = (150\text{px/s}) \times (3\text{s}) = 450\text{px}$$

下面, 让我们把该计算过程用程序来实现。当游戏在一帧一帧显示画面时, 我们必须要知道在每帧之间汽车移动了多远。请记住, 位移就是物体位置的改变, 即新位置和旧位置。如果将这些代入先前的公式, 就可以很容易地写出如下的程序代码:

```
//传入一个 float 速度值和一个 float 时间值,该函数将返回位移值  
float calcDisplacement(float vel, float time)
```

```
{  
    return vel * time;  
}
```

有些游戏利用另外一种方式计算位移, 即通过相对上次位移值的一个变化值来确定, 这对于网络游戏来说比较适合, 它可以减少网络上传输的数据量。根据物体在游戏中的绝对位置, 服务器发给它一个相对位移, 然后就可以在本地计算最终的位移值并移动, 随后只要再将该新值通告给服务器即可, 该方法显然大大减轻了服务器的计算负担, 以及在网络上传送的数据量。下面来扩展上面的例子, 根据原始位移值计算经过一段时间后的新位移, 计算公式如下:

计算两帧之间的位移

$$\text{新位置} = \text{原始位置} + \text{速度} \times \text{时间}$$

其中, 这里的时间是一帧的时间(通常是 1/30 秒)。

注意

在游戏中，画面更新的帧速并不都是30帧/秒，大多数编程人员通常采用如下方法确定该时间间隔，即自从上次更新到本次更新所逝去的系统时间。通过这种方法，当加入新的AI规则或者粒子系统时，帧更新才不会发生错误。

【例8-2】 计算匀速运动物体的新位置

重新来看例8-1，如果你不想计算汽车1秒钟后移动了多远，而是要计算两帧之间汽车移动了多远（这里假设帧速为30fps）。现在假设汽车在某一帧中的位置是50像素，那么下一帧中汽车的位置在哪里？

解答：

1. 根据上述公式，建立计算新位置的公式：

$$\text{新位置} = \text{原始位置} + \text{速度} \times \text{时间} = 50\text{px} + (150\text{px/s}) \times \text{时间}$$

2. 由于帧速为30fps，因此两帧之间的时间间隔是1/30秒，用1/30秒作为时间来计算新位置：

$$\text{新位置} = 50\text{px} + (150\text{px/s}) \times (1/30\text{s}) = 55\text{px}$$

下面函数完成该过程的计算：

```
//传入物体的原始位置、速度和间隔时间
//返回物体的新位置
float calcPosition (float oldPosition, float vel, float time)
{
    return oldPosition + (vel * time);
}
```

在前面的例子中，我们都假设物体的运动速度是保持不变的，即做匀速直线运动。但是，在实际中比如你驾驶一辆汽车，它真能够一直保持匀速吗？当然了，在游戏中建立一个匀速运动的模型会比较容易；但是对于赛车游戏呢，其速度若再保持匀速那就肯定不行了。下次当你坐在汽车上时，仔细观察一下速度计，你会注意到速度总是会发生改变的。因此，这就需要讨论两种不同的速度：平均速度和瞬时速度。

首先来看平均速度，假设你在一次驾车旅行中，用里程表记下本次旅行所行驶的距离，比如4小时行驶了200英里，那么这次旅行的平均速度是200英里/4时，即50英里/小时。但是，在这4个小时中你不可能一直保持50英里/小时的速度不变，更可能的情况是有时加速有时减速，比如遇见红灯时必须减速停下，因此在旅行时汽车的速度是不断改变的，但是其平均速度是50英里/小时。平均速度的计算公式如下所述，其中 $\bar{v}$ 上面的横线表示使用的是平均速度。

平均速度

$$\bar{v} = \frac{\Delta x}{t} = \frac{x_f - x_i}{t}$$

其中， Δx 表示位移， t 表示时间间隔。

【例 8-3】 计算平均速度

假设交警在一个繁忙的街道中对汽车的行驶速度进行了限速，如图 8-1 所示，在这条路上标记了 1/4 英里的标志。现在使用精确秒表计时，汽车 8 秒钟经过了该段距离，如果此路段限速 60 英里/小时，那么该车会不会收到罚单？

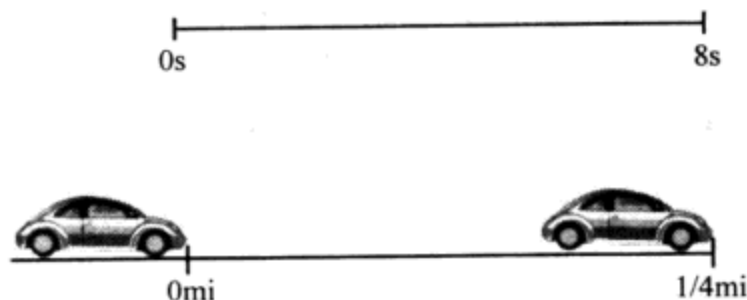


图 8-1 速度监控

解答：

1. 根据上述公式，计算平均速度：

$$\bar{v} = \frac{\Delta x}{t} = \frac{x_f - x_i}{t} = \frac{0.25\text{mi}}{8\text{s}} = 0.03125\text{mi/s}$$

2. 即汽车行驶的平均速度是 0.03125mi/s，现在将其单位转换成 mi/hr，可以复习一下第 7 章的相关知识。

$$\left(\frac{0.03125\text{mi}}{1\text{s}}\right)\left(\frac{60\text{s}}{1\text{min}}\right)\left(\frac{60\text{min}}{1\text{hr}}\right) = 112.5\text{mi/hr}$$

很显然，该汽车已经超速行驶了，因此会收到一张罚单。

瞬时速度在概念上与平均速度不同，假如你现在正驾驶着汽车下坡，其行驶速度会不断发生变化，请在任意时刻观察速度表，它会显示当前的行驶速度是多少，那么该速度表上所显示的速度就叫作瞬时速度。下一节我们将介绍如何计算该瞬时速度，先在这里提出其概念。假设在行驶过程中，汽车的平均速度是 60mi/hr，而警察测定汽车的瞬时速度却是 90mi/hr，也就是说他不关注平均速度，而只关注瞬间速度，这就是平均速度和瞬时速度的区别。

很明显，对于做匀速运动的物体来说，其平均速度等于任何时刻的瞬时速度。

然而，如果物体的速度经常发生改变，那么平均速度与在大多数时刻的瞬时速度就可能不相等。对游戏程序员来说这里有个好消息，很多游戏都是基于帧更新并显示的，而且很多编程人员都采用 30fps 的帧速。这也就是说，如果要想计算两帧间的平均速度，那么其时间间隔就是 1/30 秒，由于该数值很小，因此平均速度就很接近于瞬时速度，于是就可以将该平均速度作为瞬时速度看待。

【例 8-4】 计算两帧间平均速度

对于游戏中的某个物体，假如它在某一帧中的位置是 50 像素，而在下一帧中的位置是 52 像素，再假设当前的帧速为 30fps，那么物体的瞬时速度是多少？

解答:

1. 根据平均速度公式, 计算两帧间的平均速度:

$$\bar{v} = \frac{\Delta x}{t} = \frac{x_f - x_i}{t} = \frac{2\text{px}}{(1/30)\text{s}} = 60\text{px/s}$$

2. 由于该时间间隔(1/30s)很小, 因此可以将它看成瞬时速度。

用程序来处理该问题比较简单, 物体的初始位置和最终位置可以同是像素值, 也可以同是具体的坐标值。由于该方法处理的是在一定时间间隔内物体的位移改变量, 因此与所采用的单位关系不大, 只要保证它们一致即可; 如果它们的单位不一致, 就在利用公式之前将它们转换一致。

```
float calcAvgVel (float start , float end , float time )
{
    return (end - start)/time;
}
```

在游戏编程中, 速度是一个比较重要的物理概念, 它能够控制物体的运动方式。在这里说明一下, 如果程序只关心物体移动的有多快, 而不关心其方向, 那么就用速度的模(大小)来代替, 也就是它的速率。

然而, 假如你要控制物体在屏幕上运动, 此时就要使用速度, 否则物体就会不知道该向哪个方向移动了。作为一个程序员, 你应该尽量采用简单、快速的方法来移动物体, 比如匀速运动就比较简单, 因为其数学模型比较简单; 但是, 要想模拟出更现实、逼真的运动, 速度就不可能一直保持不变。在这种情况下, 对于较长的时间间隔来说, 就应该使用瞬时速度而不是平均速度, 在下一节中, 我们将对此做更详细的介绍。

自我测试

1. 在高速公路上, 汽车的行驶速度是 65mi/hr, 那么 15 分钟后汽车行驶了多远?
2. 假设你编出一个可以移动的游戏平台, 玩家可以跳到该平台上, 且该平台仅能向左(-)和向右(+)移动, 假设平台以 -5px/s 的速度向左做匀速运动, 它的当前位置是 200 像素, 那么 3 秒之后它的位置在哪里(假设该平台还没有到达转弯的位置)?
3. 假如在游戏中, 角色在 20 秒内跑了 25 米, 那么在 20 秒内他的平均速度是多少?
4. 假如问题 3 中的那个人忘记了一些东西, 他又返回到起点, 那么整个进程中他的平均速度是多少?
5. 假设在游戏中, 某物体在一帧中的位置是 300 像素, 在下一帧的位置是 295 像素, 并假设游戏当前的帧速为 30fps, 那么它的瞬时速度是多少?

8.2 加速度

在上一节中, 我们讨论了匀速运动和变速运动的相关知识, 实际上如果一个物体的速度发生了改变, 或加速或减速, 那么它就必然有一个加速度。在这里使用加速度来衡量速率的

变化快慢程度，如果物体加速越快，那么其加速度就越大；如果速度保持不变，那么加速度就等于零。让我们回头看前面汽车的例子，踩下刹车汽车就会减速，对应的加速度是负值；相应地，踩下油门就会加速，此时加速度为正值。让我们利用汽车的例子建立加速度的数学定义。

请看图 8-2，在初始时刻 t_i ，汽车的瞬时速度为 v_i ，经过几秒钟，在时刻 t_f 汽车的瞬时速度为 v_f ，那么加速度就是速度变化的快慢，计算公式如下：



图 8-2 汽车加速的例子

加速度

$$a = \frac{\Delta v}{\Delta t} = \frac{v_f - v_i}{t_f - t_i}$$

注意

在上述公式中，请注意单位要一致，即在进行除法运算之前确保速度单位和时间单位一致，比如 mi/hr 是不能直接除以秒的，要么把秒转换成小时，要么把小时转换成秒。

加速度的单位是长度单位除以时间单位的平方。

【例 8-5】 计算加速度

假设图 8-2 中的汽车以 40mi/hr 的速度开始行驶，5 秒后其速度是 50mi/hr，那么它的加速度用 mi/hr² 表示是多少？用 m/s² 表示呢？

解答：

1. 利用上述加速度公式计算：

$$a = \frac{\Delta v}{\Delta t} = \frac{v_f - v_i}{t_f - t_i} = \frac{50 - 40 \text{ mi/hr}}{5 \text{ s}} = \frac{10 \text{ mi/hr}}{5 \text{ s}}$$

2. 请注意，速度 and 时间的单位不一致。在相除之前需要将 5 秒转换成小时，你可以复习一下第 7 章的相关知识。转换之后，可以得到 $5 \text{ s} = 0.00139 \text{ hr}$ ，现在就可以继续计算了：

$$a = \frac{\Delta v}{\Delta t} = \frac{v_f - v_i}{t_f - t_i} = \frac{50 - 40 \text{ mi/hr}}{5 \text{ s}} = \frac{10 \text{ mi/hr}}{5 \text{ s}} = \frac{10 \text{ mi/hr}}{0.00139 \text{ hr}} = 7200 \text{ mi/hr}^2$$

3. 根据题意，我们还需要将其转换成 m/s²。从长远来看，采用公制是比较容易计算的，所以将速度单位 mi/hr 转换成 m/s，而不转换时间单位。转换之后，可以得到 $10 \text{ mi/hr} = 4.47 \text{ m/s}$ ，现在接着计算：

$$a = \frac{\Delta v}{\Delta t} = \frac{4.47 \text{ m/s}}{5 \text{ s}} = 0.894 \text{ m/s}^2$$

现在仔细观察在上面例子中得到的两个结果，首先看加速度 7200 mi/hr^2 ，它意味着如果汽车以恒定的加速度继续加速，那么每一小时其速度会增加 7200 mi/hr 。不过很显然，如果汽车真要这样加速 1 个小时，情形可能很难想象。因此，加速度采用公制单位会更容易理解些，比如在这个例子中 7200 mi/hr^2 与 0.894 m/s^2 相等，这也就意味着如果均匀加速的话，

汽车每秒钟差不多加速 1 米/秒。下面让我们来看一个例子：

【例 8-6】 计算加速度

现在假设一辆汽车撞上护栏而被迫刹车，在 3 秒内速度由 50 英里/小时变为 10 英里/小时，那么加速度是多少 m/s^2 ？

解答：

1. 利用加速度公式计算：

$$a = \frac{\Delta v}{\Delta t} = \frac{v_f - v_i}{t_f - t_i} = \frac{10 - 50 \text{mi/hr}}{3 \text{s}} = \frac{-40 \text{mi/hr}}{3 \text{s}}$$

2. 显然单位不一致，需要将 -40mi/hr 转换成 m/s 。转换之后，可得到 $-40 \text{mi/hr} = -17.88 \text{m/s}$ 。现在继续计算：

$$a = \frac{\Delta v}{\Delta t} = \frac{-17.88 \text{m/s}}{3 \text{s}} = -5.96 \text{m/s}^2$$

注意，在上例中得到的是负的加速度，这意味着物体是减速而不是加速。此处一定要注意，加速度为负数并不表示物体向后移动。在汽车例子中，踩油门可产生正的加速度，而踩刹车则产生负的加速度。然而加速度为负值，并不意味着正在倒车，仅仅说明汽车正在减速，这个概念经常会让人误解。

对于加速运动或减速运动，在程序中计算加速度的代码是相同的，其含义已经在前面解释过了。下面就是一个计算加速度的函数，它采用的速度单位为单位/秒。

```
//该函数计算以秒为单位的加速度。
float calcAccelerationSeconds (float startVel, float finalVel, float time)
{
    return ( finalVel - startVel ) / time;
}
```

该函数可以很容易地转换到以其他时间为单位的加速度计算。

在本节中，我们已经学习了速度和加速度，下一节将会把它们运用到运动方程中，请务必记清这两个量。此外，下一章我们还会以图形的方式介绍它们，并讨论它们之间的关系，随后在第 10 章还会将其扩展到 2D、3D 空间中。

自我测试

1. 假设图 8-2 中的汽车以 80mi/hr 的速度开始行驶，8 秒后速度变成 100mi/hr ，那么它的加速度是多少 m/s^2 ？
2. 假设汽车撞上护栏被迫刹车，原始速度是 40 英里/小时，5 秒后汽车停在路上，那么汽车的加速度是多少 m/s^2 ？
3. 一辆汽车以 0.6m/s^2 的加速度加速，那么从 55mi/hr 加速到 60mi/hr 需要多长时间？
4. 一辆汽车的初始速度是 7m/s ，它开始以 0.8m/s^2 的加速度加速，那么 2 秒后这辆车将走多远？

8.3 运动方程

在前两节中，我们已经学习了相关的变量，本节将要学习一维空间内的运动，并给出其相应的运动方程。不过请注意一点，这些方程只有在加速度恒定的条件下才有效，还记得前面曾提到，如果物体的运动速度不变，那么其平均速度和瞬时速度就相等，这个原理同样适用于加速度。所幸的是，自然界的大部分物体大多是在恒定的加速度下运动的，比如自由落体运动。假如在游戏中遇到加速度改变的场景，就尝试把时间间隔划分成许多小时间段，在每段时间内可以把加速度看成是恒定的，这在编程中很容易处理，因为游戏画面的更新都是基于帧的，因此在一帧内就能保证在足够小的时间间隔内，其加速度是恒定的。

注意

为了简化方程，我们用 t 来代替 $(t_f - t_i)$ 表示整个时间间隔。

由前面所知，加速度的定义如下：

$$a = \frac{\Delta v}{t} = \frac{v_f - v_i}{t}$$

对上面公式进行变换，左右两边同乘上 t ，然后再加上 v_i ，从而得到第一个运动方程：

运动方程 1

$$v_f = v_i + at$$

即：末速度 = 初速度 + 加速度 × 时间。

本章我们只讨论匀加速运动，由于速度在整个时间内均匀地增加或减小，因此对于一定时间间隔内的平均速度就等于初速度 v_i 和末速度 v_f 的平均值，于是可以得到第二个方程：

运动方程 2

$$\bar{v} = \frac{v_i + v_f}{2}$$

即：平均速度 = (初速度 + 末速度) / 2；

该公式只适合于匀加速运动。

在本章的第一节中，我们曾介绍了平均速度等于位移除以时间：

$$\bar{v} = \frac{\Delta x}{t}$$

而上面运动方程 2 给出了匀加速运动条件下平均速度的公式，将这两个公式等同起来就可以得到：

$$\frac{\Delta x}{t} = \frac{v_i + v_f}{2}$$

进行代数变换，于是得到方程 3：

$$\frac{\Delta x}{t} = \frac{v_i + v_f}{2}$$

$$\frac{\Delta x}{t} = \frac{1}{2}(v_i + v_f)$$

$$\Delta x = \frac{1}{2}(v_i + v_f)t$$

运动方程 3

$$\Delta x = \frac{1}{2}(v_i + v_f)t$$

即：位移 = $\frac{1}{2}$ (初速度 + 末速度) × 时间。

现在，将方程 1 代入到方程 3，消去 v_f ：

$$\Delta x = \frac{1}{2}(v_i + v_i + at)t$$

$$\Delta x = \frac{1}{2}(2v_i + at)t$$

$$\Delta x = (v_i + \frac{1}{2}at)t$$

$$\Delta x = v_i t + \frac{1}{2}at^2$$

运动方程 4

$$\Delta x = v_i t + \frac{1}{2}at^2$$

即：位移 = 初速度 × 时间 + $\frac{1}{2}$ × 加速度 × 时间²。

上面方程 4 是由方程 1 和方程 3 联合得到的。如果将方程 1 转换成用时间 t 表示的形式 $t = \frac{v_f - v_i}{a}$ ，然后将它代入方程 3，将得到：

$$\Delta x = \frac{1}{2}(v_i + v_f) \left(\frac{v_f - v_i}{a} \right)$$

$$\Delta x = \frac{(v_i + v_f)(v_f - v_i)}{2a}$$

$$\Delta x = \frac{v_f^2 - v_i^2}{2a}$$

$$2a\Delta x = v_f^2 - v_i^2$$

$$v_f^2 = v_i^2 + 2a\Delta x$$

运动方程 5

$$v_f^2 = v_i^2 + 2a\Delta x$$

即：末速度² = 初速度² + 2 × 加速度 × 位移。

到目前为止，我们已经介绍了5个运动方程，使用它们可以解决匀加速条件下的一维空间运动问题。由这些公式的推导可知，这些方程都源于速度和加速度的定义公式，熟悉它们最好的方法就是多做练习。

处理一维空间中的匀加速运动，可以采用下面的步骤：

1. 画出一个表，列出已知哪些量，待求哪些量。
2. 检查一下单位，确保它们是一致的。
3. 根据前面列出的表，从表8-1中选出合适的运动方程进行计算。

如果已知某些量的具体数值，可以将它们代入方程，计算出待求量的值。如果要在编程中处理该问题，这些值可能会从游戏中实时获得，只需利用代数法对运动方程进行必要的转化，从而求出你所需要变量的值。如表8-1所示：

表8-1 五个运动方程

运动方程1	$v_f = v_i + at$
运动方程2	$\bar{v} = \frac{v_i + v_f}{2}$
运动方程3	$\Delta x = \frac{1}{2}(v_i + v_f)t$
运动方程4	$\Delta x = v_i t + \frac{1}{2}at^2$
运动方程5	$v_f^2 = v_i^2 + 2a\Delta x$

注意

在上述方程中，方程2可能不会被经常使用，不过它可以提醒我们：平均速度不等于瞬时速度（除非 $a=0$ ）。

【例8-7】赛车例子

假设一辆汽车撞上护栏而被迫刹车，此时汽车的速度是50mi/hr，而且已知护栏会使汽车以 -6.5m/s^2 的加速度进行减速，那么经过多长时间汽车才能停下？

解答：

1. 画一个表，列出已知条件和待求变量值。

已知条件	待求值
$v_i = 50\text{mi/hr}$	$t = ?$
$v_f = 0\text{mi/hr}$	
$a = -6.5\text{m/s}^2$	

2. 由于单位不统一，因此需要进行转换，在这里可以转换成英里和小时或者转换成米和秒，在选择时建议大家采用公制单位，因此将所有的单位都转换成米和秒。

已知条件	待求值
$v_i = 50\text{mi/hr} = 22.35\text{m/s}$	$t = ?$
$v_f = 0\text{mi/hr} = 0\text{m/s}$	
$a = -6.5\text{m/s}^2$	

3. 现在可以选择一个方程进行计算, 由于方程 1 给出了题中 4 个量之间的关系, 因此利用方程 1 求解:

$$\begin{aligned}v_f &= v_i + at \\0\text{m/s} &= 22.35\text{m/s} + (-6.5\text{m/s}^2)t \\-22.35\text{m/s} &= (-6.5\text{m/s}^2)t \\t &= 3.44\text{s}\end{aligned}$$

【例 8-8】再看赛车例子

假设现在情况与例 8-7 一样, 再假设离你 10 米远处有一辆汽车, 问该车能否及时停下, 而不发生相撞?

解答:

1. 列表中的已知条件是相同的, 唯一不同的是现在求的是位移而不是时间。

已知条件	待求值
$v_i = 50\text{mi/hr}$	$\Delta x = ?$
$v_f = 0\text{mi/hr}$	
$a = -6.5\text{m/s}^2$	

2. 单位换算过程, 与上例相同。

已知条件	待求值
$v_i = 50\text{mi/hr} = 22.35\text{m/s}$	$\Delta x = ?$
$v_f = 0\text{mi/hr} = 0\text{m/s}$	
$a = -6.5\text{m/s}^2$	

3. 现在, 从表 8-1 中选择一个方程进行计算, 显然方程 5 最适合。

$$\begin{aligned}v_f^2 &= v_i^2 + 2a\Delta x \\(0\text{m/s})^2 &= (22.35\text{m/s})^2 + 2(-6.5\text{m/s}^2)\Delta x \\0 &= 499.5235\text{m}^2/\text{s}^2 + (-13\text{m/s}^2)\Delta x \\-499.5235\text{m}^2/\text{s}^2 &= (-13\text{m/s}^2)\Delta x \\\Delta x &= 38.425\text{m}\end{aligned}$$

显然, $38.425\text{m} > 10\text{m}$, 则本题的答案就是不能。

从上面两个例子可以看出, 其初始条件完全相同, 而结果却不一样。在解题之前, 一定要弄清楚已知哪些条件, 要求哪些变量的值, 据此选择合适的方程进行求解。需要注意的是, 在将数据代入方程之前一定要保证其单位是一致的, 你甚至可以把单位代入方程作为另外一种检验方法, 如果得出结果的单位具有物理意义, 那么就说明代数运算过程是正确的。在最后一个例子中, 需要计算出位移, 因此用米作单位是有意义的。让我们来看一个较复杂的例子。

【例 8-9】利用多个方程求解

假设你开车在公路上行驶, 在踩油门之前看了一下速度计, 当前速度是 45 英里/小时。如果汽车以 2m/s^2 的加速度匀加速前进, 那么需要多长时间它能够行驶 0.25 英里?

解答:

1. 列一个表, 列出已知条件和待求变量。

已知条件	待求值
$v_i = 45 \text{ mi/hr}$	$t = ?$
$\Delta x = 0.25 \text{ mi}$	
$a = 2 \text{ m/s}^2$	

2. 检查单位, 发现它们不一致, 因此需要进行转换, 将其都转换成公制。

条件	求值
$v_i = 45 \text{ mi/hr} = 20.11 \text{ m/s}$	$t = ?$
$\Delta x = 0.25 \text{ mi} = 402.25 \text{ m}$	
$a = 2 \text{ m/s}^2$	

3. 现在可以利用上表选择一个方程进行计算, 选择方程 4。

$$\Delta x = v_i t + \frac{1}{2} a t^2$$

$$402.25 \text{ m} = (20.11 \text{ m/s})t + \frac{1}{2}(2 \text{ m/s}^2)t^2$$

在上面式子中的时间 t 即是待求的, 该问题怎么解决呢? 当然, 可以通过求解二次方程式, 但是有点麻烦。解决该问题的另一种方法是, 不使用方程 4 而用其他两个方程。注意, 在列表中没有出现 v_f , 虽然题中并没有求它的值, 但我们可以利用关于 v_f 的方程来求时间。

4. 用方程 5 算出 v_f :

$$v_f^2 = v_i^2 + 2a\Delta x$$

$$v_f^2 = (20.11 \text{ m/s})^2 + 2(2 \text{ m/s}^2)(402.25 \text{ m})$$

$$v_f^2 = 404.41 \text{ m}^2/\text{s}^2 + 1609 \text{ m}^2/\text{s}^2$$

$$v_f^2 = 2013.41 \text{ m}^2/\text{s}^2$$

$$v_f = 44.87 \text{ m/s}$$

5. 利用方程 1 计算时间:

$$v_f = v_i + at$$

$$44.87 \text{ m/s} = 20.11 \text{ m/s} + (2 \text{ m/s}^2)t$$

$$24.76 \text{ m/s} = (2 \text{ m/s}^2)t$$

$$t = 12.38 \text{ s}$$

求解同一个问题, 通常会有不止一种方法, 因此尽量尝试你自己的方法, 当试图利用某个方程计算比较困难时, 可以尝试使用其他多个方程进行求解。

至此, 前面仅仅介绍了水平方向上的一维空间运动, 其实对于竖直方向上的一维运动, 前面的运动方程同样适用, 不过要将方程中的 Δx 换成 Δy , 用于表示 y 方向上坐标。在竖直方向上的运动通常会考虑重力, 对于地球上所有的物体, 都由于重力而具有相同的加速度

(如果我们忽视空气阻力), 叫作重力加速度, 其值为 $g = 9.8\text{m/s}^2 = 32.1\text{ft/s}^2$ 。在上述五个方程中, 可以用 $-g$ 或 -9.8m/s^2 代替 a 来计算竖直方向上的运动, 注意它前面的负号表示方向向下, 也就是说当扔下一个物体时它将向下落; 如果将一个物体抛向空中, 它将减速至 0 然后开始下落。下面来看一个例子。

【例 8-10】 竖直运动

假若在游戏中, 一个怪物在平台上丢下一块石头, 该平台距离地面的高度是 50 米, 那么石头落到地面上需要多长时间?

解答:

1. 列一个表, 列出已知条件和待求变量值。

已知条件	待求值
$v_i = 0\text{m/s}$	$t = ?$
$\Delta x = -50\text{m}$	
$a = -9.8\text{m/s}^2$	

由于石头是掉落下去的, 而不是被扔出去的, 因此其初速度为 0; 另外, 由于石头是向下落, 故其位移量是负值; 还有由于是竖直方向上的运动, 故加速度已知为 g 。

2. 检查单位, 所有的单位都是米和秒, 所以不需要进行转换。
3. 利用方程 4 来计算时间:

$$\begin{aligned}\Delta x &= v_i t + \frac{1}{2} a t^2 \\ -50\text{m} &= (0\text{m/s})t + 1/2(-9.8\text{m/s}^2)t^2 \\ -50\text{m} &= 0 + (-4.9\text{m/s}^2)t^2 \\ t^2 &= 10.20\text{s}^2 \\ t &= 3.19\text{s}\end{aligned}$$

由上例可知, 一维空间内的竖直运动和水平运动的计算方法是一样的。本章介绍了描述一维空间运动的 5 个方程, 在下一章中我们将进一步介绍速度和加速度, 然后在后续章节中会将这些概念推广到二维和三维空间中。

自我测试

1. 一艘赛艇在其速度从 20 米/秒增加到 30 米/秒的过程中, 经过的位移是 200 米, 那么它的加速度是多少?
2. 在问题 1 中, 赛艇跑完 200 米花费了多长时间?
3. 一辆赛车以 40 米/秒的速度到达终点线, 然后驾驶员开始刹车, 5 秒后汽车停了下来, 请问它的加速度是多少?
4. 在问题 3 中, 在汽车停下来之前它跑了多远距离?
5. 一个球以 25 米/秒的速度向上抛, 它达到最大高度需要多长时间?
6. 问题 5 中球的最大高度是多少?

8.4 可视化体验

在随书 CD 光盘上, 有一个名字叫 1D Racer 的演示程序, 下面是程序员 Kaveh Kahrizi 对它的简单描述:

1D 赛车程序描述了一个物体在空间中做匀加速运动的情形。首先, 请选择你想使用的运动方程, 然后选择待求的变量。随后输入 3 个已知参数的值之后, 演示程序就开始运行了。这些方程标记在游戏项目中的头文件 Game.h 中, 并且有详细的注释。该工程是在 Visual Studio 6.0 下创建、编译生成的, 在不改变项目设置的情况下, 许多文件都能很容易地加入到已有项目中, 只需配置 DirectX 类库的文件位置就可以了。运行程序要求机器中必须安装有 DirectX 8.1 或更高版本的 DirectX 组件。此外, 为了能够自己编译代码, 就必须安装 DirectX 8.1 SDK 或更高版本的 SDK 开发包, 它们都可以从微软网站 www.Microsoft.com 上免费下载到。

双击 1DRacer.exe 文件即可运行该演示程序。程序运行后, 首先会在屏幕上看到四个运动方程。用方向键可以选择使用的方程, 比如此处选择第一个 ($v_f = v_i + at$); 接下来程序会询问你想求哪个变量的值, 我们选择 a , 然后输入 3 个已知的变量值, 比如输入:

$$v_i = 0$$

$$v_f = 50$$

$$t = 10$$

此时按下回车键赛车将开始运动, 在赛车行驶过程中, 应用程序会跟踪时间、位移和当前速度, 一旦按下 P 键, 演示程序将暂停。尝试重复这个过程, 在 $t = 5$ 时暂停运行, 看看汽车跑多快, 然后和赛车到达终点时的速度进行比较; 按 Backspace 键, 可重新开始运行。

按下 Backspace 键, 并尝试用不同的方程进行测验。重复这个例子, 以查看实际运动情况。让我们重新回到例 8-8, 进行相应的单位换算之后, 可以得到下表:

已知条件	待求值
$v_i = 50 \text{ mi/hr} = 22.35 \text{ m/s}$	$\Delta x = ?$
$v_f = 0 \text{ mi/hr} = 0 \text{ m/s}$	
$a = -6.5 \text{ m/s}^2$	

利用最后一个方程 $v_f^2 = v_i^2 + 2a\Delta x$ 重新运行程序, 此时需要计算 Δx ; 然后输入公制制数值, 并且开始运行, 你能够得到接近 $\Delta x = 38.425$ 米的值吗?

也许你可能会想到利用该程序检验自我测试题的答案, 请尽量尝试构造一些题目, 并检查你能得到什么结果。你可能会在程序的运行过程中, 在不同时刻暂停程序跟踪其运动。

此外, 你可以双击 1DRacer_code.txt 文件来查看其源代码, 如果你观察该文件的最后面, 你将会看到运动方程是如何编码的。此时, 你也可能会建立一个基于文本的应用程序, 完成同样的功能。

本章总结

在本章中，我们开始进入物理知识的世界。首先复习了位移的概念，并且引出了速度和加速度的定义，随后演示了距离和位移以及速率和速度有哪些区别，最后介绍了一维空间内的 5 个运动方程，演示了 3 个量之间的关系。注意到，这些概念都是建立在 1D 空间中的，在随后两章中，我们将继续介绍速度和加速度的应用，及其他在 2D 和 3D 空间中的推广。

能力测试

选择题

1. 判断正误。速度和速率相同。
A. 正确 B. 错误
2. 如果一个物体以 240px/s 的速度匀速运动，那么它在两帧之间的位移是多少（假设当前的帧速是 30fps ）？
A. 8px B. 80px C. 270px D. 7200px
3. 下面哪个是重力加速度（ y 轴正方向向上）？
A. 32m/s^2 B. -32ft/s C. 9.8m/s D. -9.8m/s^2
4. 如果已知一个物体的初速度、末速度和它的加速度，应该用表 8-1 中的哪个方程来计算位移？
A. 方程 1 B. 方程 3 C. 方程 4 D. 方程 5
5. 如果一个物体从 30 像素的高度落下，落到地面会经过多长时间？
A. 1.9s B. 1.4s C. 2.5s D. 9.8s

练习

练习 1：速度和速率

1. 假设汽车以 65 英里/小时的速度在公路上行驶，40 分钟后，它行驶了多远？
2. 假设你编出一个移动平台供玩家跳跃，该平台仅仅能左（-）右（+）移动，并且以 -6.5px/s 的速度匀速运动。如果它的当前位置是 450 像素，那么 5 秒之后它的位置在哪里？
3. 假设游戏中的角色在 20 秒钟内跑了 28 米，那么在 20 秒时间内他的平均速度是多少？
4. 假设问题 3 中的那个人忘记了一些东西，他又返回了起点，那么整个过程他的平均速度是多少？
5. 假设在游戏中，一个物体在一帧的位置是 500 像素，并且在下一帧它的位置是 50 像素，假若游戏的帧速为 30fps ，那么它的瞬时速度是多少？

练习 2：加速度

1. 假设汽车撞上护栏被迫刹车，起始速度是 50 英里/小时，经过 3 秒后停在路上，那么汽车的加速度是多少 m/s^2 ？

2. 一辆汽车以 0.4 m/s^2 的加速度加速, 那么从 35 mi/hr 加速到 70 mi/hr 需要多长时间?
3. 一辆汽车的初始速度是 7 m/s , 它以 1.2 m/s^2 的加速度加速 3 秒后将走多远?

练习 3: 运动方程

1. 一艘赛艇行驶了 500 米, 其间的速度从 12 米/秒增加到 28 米/秒, 那么它的加速度是多少?
2. 问题 1 中, 赛艇行驶完 125 米花费多长时间?
3. 一辆赛车以 43 米/秒的速度到达终点线, 然后驾驶员开始刹车, 7 秒后汽车停下, 请问它的加速度是多少?
4. 问题 3 中汽车跑多远才能停下来?
5. 一个球以 16 米/秒的速度向上抛, 它到达最大高度需要多长时间?
6. 问题 5 中球的最大高度是多高?

程序设计

程序设计 1: 求物体的信息

1. 编写一个简单程序, 已知物体的当前位置和速度, 求出它在下一帧中的新位置(假设该物体在 1D 空间内运动, 帧速为 30fps)。
2. 编写一个简单程序, 已知物体的当前速度和加速度, 求出它在下一帧中的新速度(假设该物体在 1D 空间内运动, 帧速为 30fps)。

程序设计 2: 应用 5 个运动方程

利用 5 个运动方程, 创建一个函数库。提示: 在求解方程中的变量时, 可能会使用不止一个方程。

程序设计 3: 运行 1D Racer 演示程序

运行光盘上的 1D racer 演示程序, 它曾在本章的可视化体验中介绍过, 利用它检验这一章的习题答案。

程序设计 4: 求物体的高度

1. 编写一个简单的程序, 请求用户输入物体的高度。
2. 利用一个简单的图形, 将这个物体放置在屏幕中的正确高度上。
3. 根据用户的输入数据(按下回车键或者开始按钮), 利用重力加速度实时演示该物体的下落。

问题讨论

1. 速度和速率有什么相似之处, 有什么不同之处?
2. 对于公式 $D = v \times t$ 来说, 为什么必须在匀速运动的情况下才成立?
3. 平均速度和瞬时速度有什么不同?
4. 在什么条件下, 平均速度等于瞬时速度?
5. 加速度的单位为什么是长度单位除以时间单位的平方?
6. 在什么条件下使用方程 4 不易求出结果? 为提高性能应该怎样回避使用方程 4?

第9章 一维空间的运动导数

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 定义一个导数
- 使用导数计算瞬时速度
- 使用导数计算瞬时加速度

介绍

前面一章我们定义了速度和加速度，并介绍了如何用纯数学的方法计算它们。本章将进一步讨论速度和加速度的相关知识。首先用图像来帮助我们理解运动，然后再用图像定义导数和极限。在本章中我们简要介绍了牛顿的微积分概念。另外在你需要对其进一步研究时，附录 B 可以提供一些参考。

9.1 速度的图像和导数

首先从速度的概念入手。注意，这里速度是一个矢量，所以它是有方向的，即在一维空间中有正负之分。另外，平均速度是用总位移除以时间。为了更好的了解速度，可以画一个速度相对于时间的直角坐标系，首先我们从竖直运动开始研究，比如竖直上抛运动。

假设游戏中智能体向空中抛出一个西红柿，随着时间的增加，记录下西红柿的高度，如表 9-1 所示。

然后根据这些点画出西红柿随着时间变化的运动轨迹，如图 9-1 所示。

如图所示，随着时间的增加，西红柿先向上运动然后下落，我们可以将高度(y)看成关于时间(t)的函数，也就是说在 0 到 6 秒的过程中，随着时间的增加，高度一直在变化。从数学角度来看，我们可以把它写成 $y=f(t)$ ，读作 y 是关于 t 的函数。在后面的章节中我们也将沿用这一符号。

表 9-1 西红柿在不同时刻的高度值

时间(秒)	高度(英尺)
0	6
1	90
2	142
3	162
4	150
5	106
6	30

下面利用图 9-1 重新演算一下平均速度。如果你想知道西红柿在前 4 秒的平均速度，使用纯数学的方法，在求出位移后，用位移除以 4 秒就可以了。下面，我们可以利用新定义的

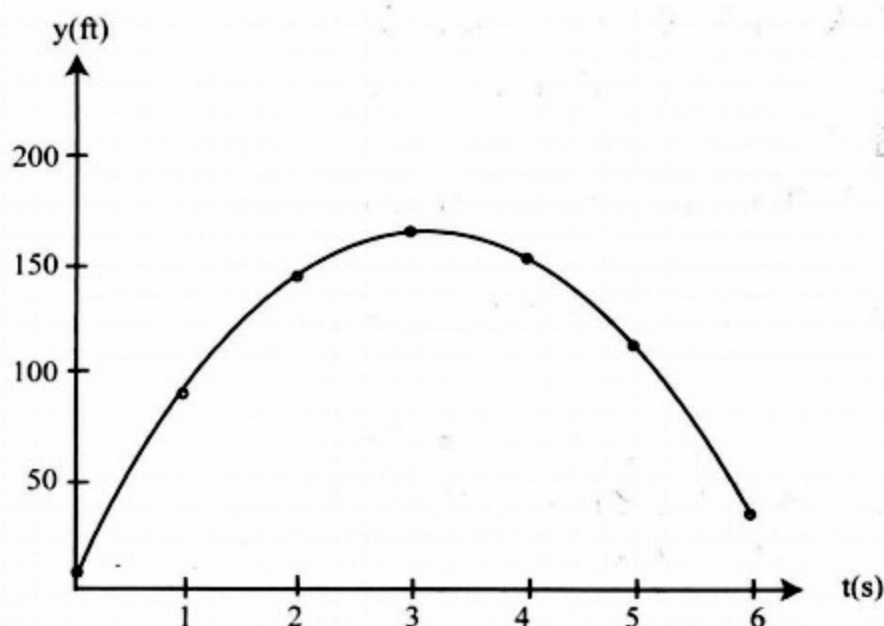


图 9-1 根据表 9-1 画出的轨迹图

符号来计算，如下所示：

$$\bar{v} = \frac{f(4) - f(0)}{4 - 0} = \frac{150 - 0}{4} = \frac{150}{4} = 37.5 \text{ ft/s}$$

平均速度

$$\bar{v} = \frac{f(b) - f(a)}{b - a}$$

两点之间线段的斜率表示平均速度的大小，时间间隔为 $a \leq t \leq b$ 。

在这里可以用图像来表示。还记得以前的斜率公式吗？斜率 = 竖直距离/水平距离。在这里竖直距离即为位移，水平距离为时间。所以，平均速度就很容易的表示为两点之间直线的斜率。如图 9-2 所示。

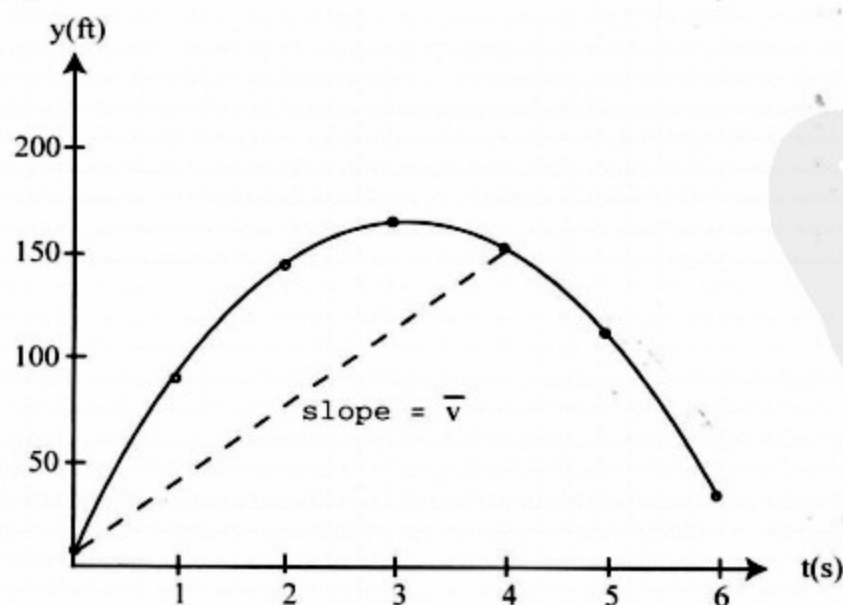


图 9-2 前 4 秒的平均速度

将这两点的坐标带入斜率公式, 然后重复前面的过程。在“位置-时间”图中任意两点之间直线的斜率就代表了在这一时间间隔内的平均速度。

【例 9-1】 计算平均速度

利用刚才的数据, 计算西红柿在 $3 \leq t \leq 5$ 的平均速度。

解答:

1. 首先需要知道在 $t = 3\text{s}$ 和 $t = 5\text{s}$ 时西红柿的高度:

$$f(3) = 162$$

$$f(5) = 106$$

2. 计算平均速度, 或者说两点之间直线的斜率:

$$\bar{v} = \frac{f(5) - f(3)}{5 - 3} = \frac{106 - 162}{2} = \frac{-56}{2} = -28\text{ft/s}$$

注意

仔细观察图像, 会发现西红柿在 3 到 5 秒中在下落, 这也就是为什么平均速度是负值。

这样我们就可以计算任意时间间隔的平均速度了。现在来看一下, 如果时间段越来越小会发生什么情况。假设我们想知道在 $t = 1\text{s}$ 时西红柿的状态。我们首先计算出西红柿在 1 到 4 秒的平均速度:

$$\bar{v} = \frac{f(4) - f(1)}{4 - 1} = \frac{150 - 90}{3} = \frac{60}{3} = 20\text{ft/s}$$

计算 1 到 3 秒的平均速度:

$$\bar{v} = \frac{f(3) - f(1)}{3 - 1} = \frac{162 - 90}{2} = \frac{72}{2} = 36\text{ft/s}$$

计算 1 到 2 秒的平均速度:

$$\bar{v} = \frac{f(2) - f(1)}{2 - 1} = \frac{142 - 90}{1} = \frac{52}{1} = 52\text{ft/s}$$

如图 9-3 所示, 时间间隔越来越小。

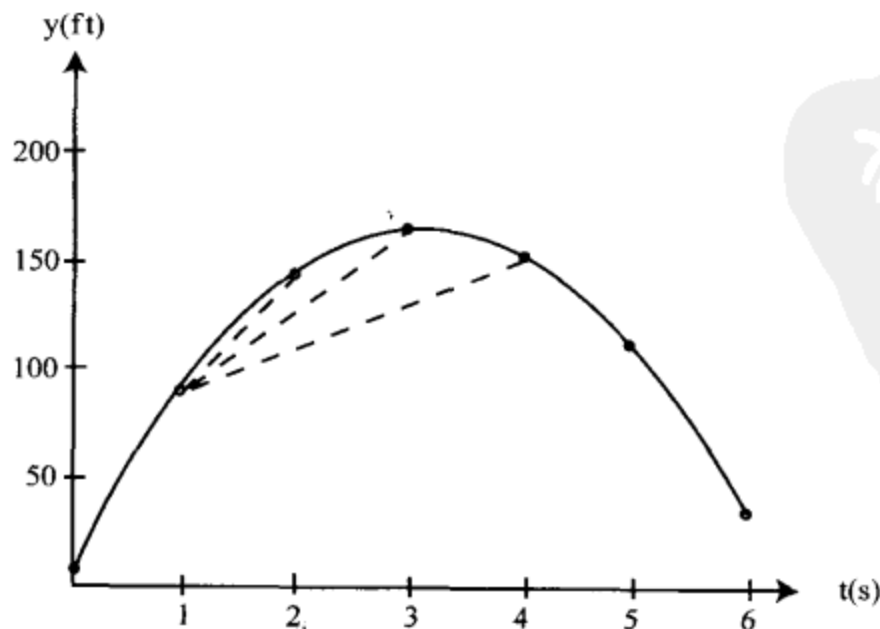


图 9-3 时间间隔无限接近 $t = 1$

你可以继续缩短时间间隔，直到线段接近于一点或者时间接近一瞬间，这就引出了瞬时速度的概念。我们来看图 9-3 中的虚线。如图 9-4，当时间间隔越来越小时，虚线就越来越接近实线。这条直线表示了在 $t = 1\text{s}$ 处的斜率。这一斜率也就是 $t = 1\text{s}$ 时的瞬时速度。

注意

图 9-4 中的实线称为切线。

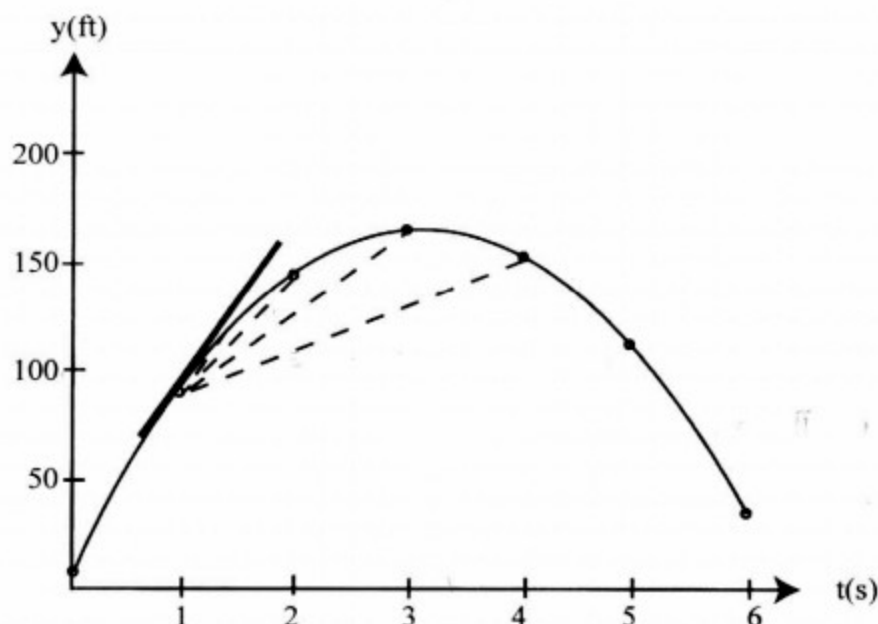


图 9-4 $t = 1$ 时的瞬时速度

在“位置-时间”图中，任意两点之间直线的斜率就代表了这一时间间隔的平均速度。某一处的斜率代表了这一点的瞬时速度。这样就引入了极限的概念。随着时间间隔越来越小，你会发现平均速度越来越趋近于某一个值，而这个值就是瞬时速度。

瞬时速度

$$v = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

$f(t)$ 表示 $t = a$ 时位置关于时间的函数

极限符号看似是强加给我们的概念，其实，它所表示的意义就是当时间间隔趋近于零时，不断的重复计算平均速度，直到最后得到瞬时速度。

【例 9-2】 计算瞬时速度

如果编一个像“Half Life”这样的游戏程序，玩家向空中扔一个雷，那么它的高度就是关于时间的函数，就像前面提到的西红柿一样。如果它的高度是 $y = f(t) = t^2 + 5$ ，计算 $t = 3\text{s}$ 时的瞬时速度。

解答：

1. 要找到 $t = 3\text{s}$ 时的瞬时速度，我们要在不断的缩小时间间隔内求平均速度。也就是说，计算当时间趋近于零时 $f(t)$ 的极限：

$$\lim_{h \rightarrow 0} \frac{f(3+h) - f(3)}{h}$$

2. 利用 $y = f(t) = t^2 + 5$ 计算高度:

$$\begin{aligned}
 & \lim_{h \rightarrow 0} \frac{f(3+h) - f(3)}{h} \\
 &= \lim_{h \rightarrow 0} \frac{(3+h)^2 + 5 - (3^2 + 5)}{h} \\
 &= \lim_{h \rightarrow 0} \frac{9 + 6h + h^2 + 5 - (9 + 5)}{h} \\
 &= \lim_{h \rightarrow 0} \frac{6h + h^2}{h} \\
 &= \lim_{h \rightarrow 0} \frac{6 + h}{1}
 \end{aligned}$$

这就是极限的作用, 当 h 趋近于零时, 瞬时速度接近 6。

无论你相信与否, 你刚刚已经计算出了一个导数, 导数就是瞬间的变化率。在这里, 就是瞬时速度, 是高度在时间间隔趋近于零时的变化率。导数的符号是'。比如 $f(t)$ 的导数是 $f'(t)$ 。

瞬时速度

$$v = f'(t)$$

$f(t)$ 为任意位移关于时间的函数。

【例 9-3】 计算导数

假如游戏中一辆赛车在屏幕上移动, 它的“位置-时间”方程为 $y = f(t) = 15t + 10$, 求在 $t = 2\text{s}$ 时的导数, 或者说此时的瞬时速度。

解答:

1. 要求出 $t = 2\text{s}$ 的瞬时速度, 我们需要求出导数, 即当时间间隔趋近于零时的平均速度, 利用极限的概念得:

$$f'(2) = \lim_{h \rightarrow 0} \frac{f(2+h) - f(2)}{h}$$

2. 利用公式 $y = f(t) = 15t + 10$ 分别计算两个位置:

$$\begin{aligned}
 & \lim_{h \rightarrow 0} \frac{f(2+h) - f(2)}{h} \\
 &= \lim_{h \rightarrow 0} \frac{(15(2+h) + 10) - (15(2) + 10)}{h} \\
 &= \lim_{h \rightarrow 0} \frac{30 + 15h + 10 - (30 + 10)}{h} \\
 &= \lim_{h \rightarrow 0} \frac{40 + 15h - 40}{h} \\
 &= \lim_{h \rightarrow 0} \frac{15h}{h} \\
 &= \lim_{h \rightarrow 0} \frac{15}{1}
 \end{aligned}$$

3. 在这里, 不用考虑 h 是多少, 导数即为 15。

在这一节中, 共计算了两个导数。你可以利用简单多项式(关于 t 的幂)来模拟这里所讨论的物体运动。同时, 也可以利用函数来描述任意物体随着时间增加的位置变化。方法都是一样的; 求在时间间隔趋近于零时的极限。这只是一个简单的介绍, 如果你想做进一步的研究, 可以参看附录 B。

自我测试

1. 回看例 9-1 中的西红柿, 找出物体在下列时间间隔中的平均速度:
 $3 \leq t \leq 6$
 $3 \leq t \leq 5$
 $3 \leq t \leq 4$
2. 回看例 9-2, 求手榴弹在 $t = 1\text{s}$ 时的瞬时速度。
3. 假设向空中抛出一物体, 它的“高度-时间”的函数为 $y = f(t) = 3t^2 + 1$, 求在 $t = 5\text{s}$ 时的瞬时速度。
4. 假设一辆汽车的运动轨迹方程为 $y = f(t) = 5t + 2$, 求在 $t = 3\text{s}$ 时物体的瞬时速度或者导数。

9.2 加速度图像和二阶导数

前面一节我们对瞬时速度做了深入的探讨, 并介绍了如何计算导数的问题。这一节将对加速度做更深一步的讨论, 还将引出有关二阶导数的概念。首先来从图像开始讨论。

这一次不用位置关于时间的图像, 而是用一个物体的速度或速率关于时间的图像。假设记录一辆赛车从静止到踩油门加速的“速度-时间”图像, 如表 9-2 所示。

我们可以通过描点法画出赛车加速的“速度-时间”图像。如图 9-5 所示。

正如图中所示, 随着时间的增加, 赛车逐渐加速。我们可以将其描述为速度 v 关于时间 t 的函数。也就是说, 速度随着时间从 0 增加到 6 在不断的变化。在数学上, 我们可以将其表示为 $v = f(t)$, 读作速度关于时间的函数。注意, 这个函数与前一节我们所提到的函数非常相似。迄今为止, 它们还是一样的。

和上次一样, 利用图 9-5 所示重新研究加速度。比如, 要求出前 6 秒赛车的平均加速度, 我们可以利用上一节的方法, 用速度差除以时间。在这一节中, 我们可以利用新的符号来表示平均加速度, 如下所示:

$$\bar{a} = \frac{f(6) - f(0)}{6 - 0} = \frac{108 - 0}{6} = 18\text{m/s}^2$$

表 9-2 赛车的“速度-时间”表

时间(秒)	速度(m/s)
0	0
1	3
2	12
3	27
4	48
5	75
6	108

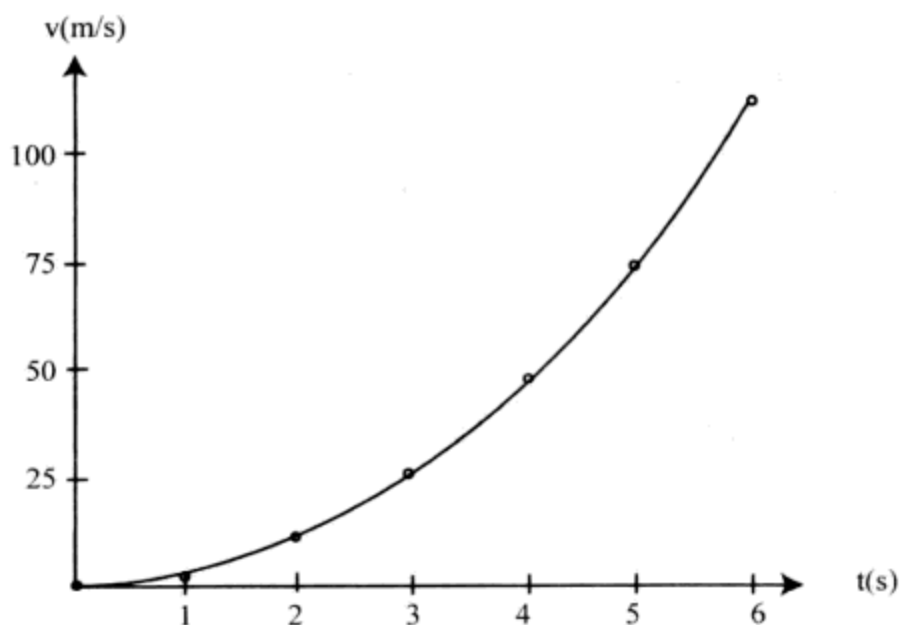


图 9-5 关于表 9-2 的“速度-时间”图

平均加速度

$$\bar{a} = \frac{f(b) - f(a)}{b - a}$$

即为在速度 v 上两点的斜率，其中 $a \leq t \leq b$ 。

我们还可以利用图像来表示这一点。还记得前面的斜率公式吗？斜率 = 竖直距离/水平距离。在这里，竖直距离为速度差，水平距离为时间。所以两点间线段的斜率为平均加速度，如图 9-6 所示。

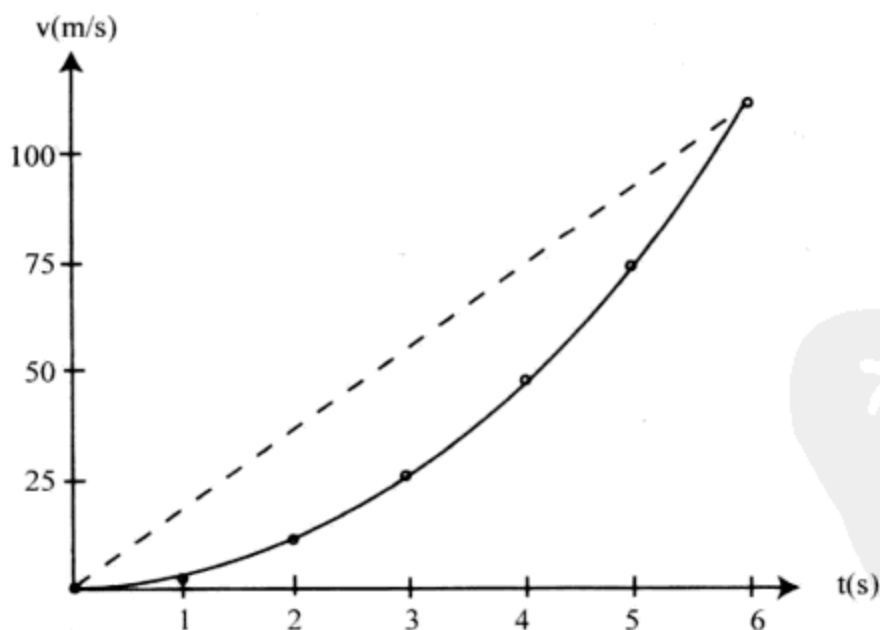


图 9-6 前 6 秒的平均加速度

把两点坐标代入斜率公式，然后重复前面的过程。“速度-时间”图像中任意两点间线段的斜率就是这一时间间隔内的平均加速度。就像“位移-时间”图像里给出的平均速度一样。

【例 9-4】 计算平均加速度

利用表 9-2 中的数据，计算赛车在 $2 \leq t \leq 4$ 时的平均加速度。

解答：

1. 观察这些数据，我们首先需要知道赛车在 $t = 2\text{s}$ 和 $t = 4\text{s}$ 时的位移是多少：

$$f(2) = 12$$

$$f(4) = 48$$

2. 计算平均加速度，或者说两点之间线段的斜率：

$$\bar{v} = \frac{f(4) - f(2)}{4 - 2} = \frac{48 - 12}{2} = \frac{36}{2} = 18\text{m/s}^2$$

这样，你就可以计算出任意时间间隔的平均加速度了。现在来看一下，如果将时间间隔变小将会出现什么情况。假如你想更好的观察赛车在 $t = 1\text{s}$ 时的加速情况，可以首先计算在 1 到 4 秒时的平均加速度：

$$\bar{a} = \frac{f(4) - f(1)}{4 - 1} = \frac{48 - 3}{3} = \frac{45}{3} = 15\text{m/s}^2$$

计算在 1 到 3 秒时的平均加速度：

$$\bar{a} = \frac{f(3) - f(1)}{3 - 1} = \frac{27 - 3}{2} = \frac{24}{2} = 12\text{m/s}^2$$

计算在 1 到 2 秒时的平均加速度：

$$\bar{a} = \frac{f(2) - f(1)}{2 - 1} = \frac{12 - 3}{1} = \frac{9}{1} = 9\text{m/s}^2$$

当时间间隔越来越小，如图 9-7 所示。

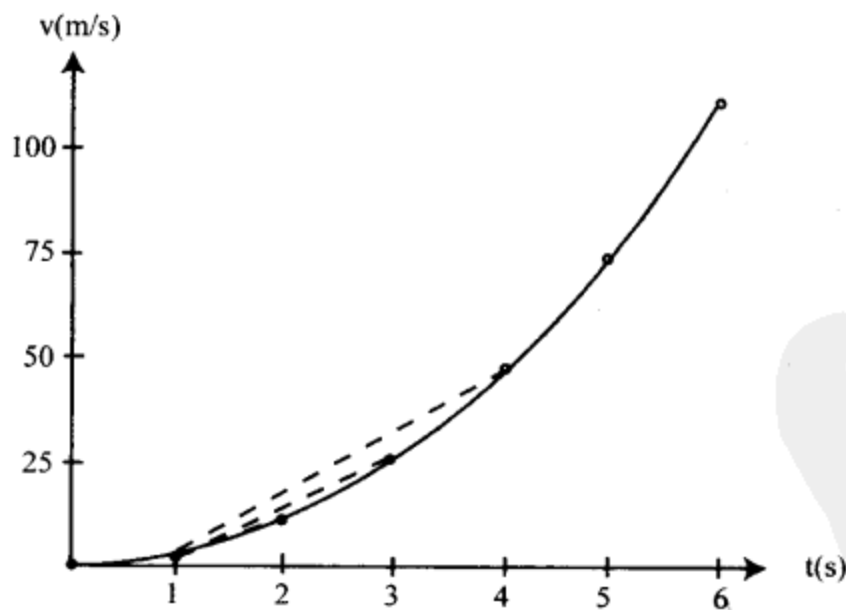


图 9-7 时间间隔接近于 1 秒时的“速度-时间”图像

你可以将时间间隔变得越来越小，最后变成一瞬间，这就得到了瞬时加速度。如图 9-7 中所示的虚线。当时间间隔变得越来越小时，虚线也就越接近实线，如图 9-8 所示。实线表示 $t = 1\text{s}$ 时的斜率。这个斜率就是此处的瞬时加速度。

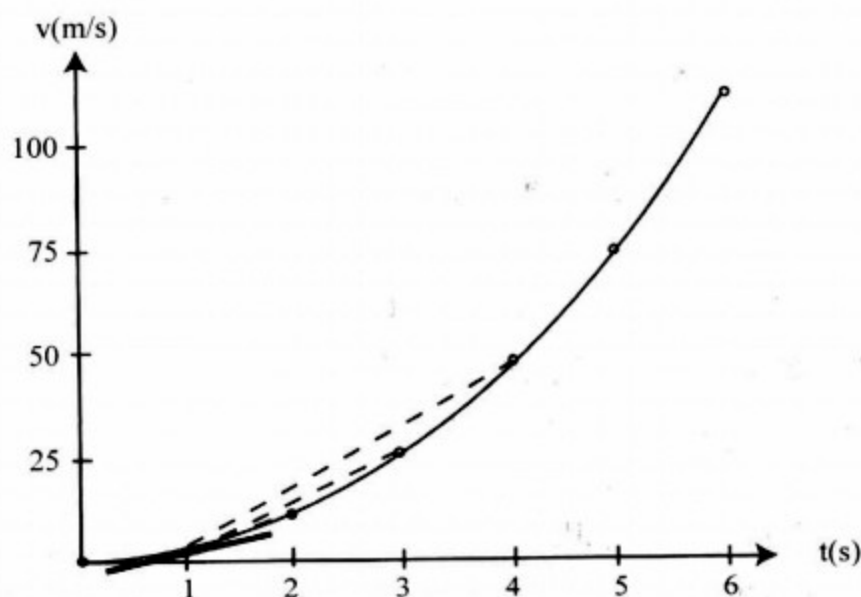


图 9-8 在 1 秒处的瞬时速度

在“速度-时间”图像中，平均加速度是在一个时间间隔内线段的斜率。瞬时加速度是图中的某一点处的斜率。这里沿用了同“位移-时间”图像中速度一样的严格的模式，同时，也引入了极限的概念。当时间间隔越来越小时，你会发现平均加速度趋近于某一数值，这个值就是瞬时加速度。

瞬时加速度

$$a = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

其中 $f(t)$ 为速度关于时间的函数。

【例 9-5】 计算瞬时加速度

假设在“Half Life”游戏中，将另一只手榴弹扔出，它的“速度-时间”的函数为 $v = f(t) = -9.8t + 25$ ，计算在 $t = 3\text{s}$ 时手榴弹的瞬时加速度。

解答：

1. 要计算 $t = 3\text{s}$ 时手榴弹的瞬时加速度，需要将平均加速度的时间间隔变得越来越小。也就是说，计算当时间变化趋近于零时的位移量：

$$\lim_{h \rightarrow 0} \frac{f(3+h) - f(3)}{h}$$

2. 利用公式 $v = f(t) = -9.8t + 25$ 计算位移量：

$$\begin{aligned} & \lim_{h \rightarrow 0} \frac{f(3+h) - f(3)}{h} \\ &= \lim_{h \rightarrow 0} \frac{-9.8(3+h) + 25 - (-9.8(3) + 25)}{h} \\ &= \lim_{h \rightarrow 0} \frac{-29.4 - 9.8h + 25 + 29.4 - 25}{h} \end{aligned}$$

$$\begin{aligned}
 &= \lim_{h \rightarrow 0} \frac{-9.8h}{h} \\
 &= \lim_{h \rightarrow 0} \frac{-9.8}{1}
 \end{aligned}$$

在这里利用了极限符号, 如果 h 趋近于零, 那么瞬时加速度就趋近于 -9.8 。

和前面一样, 你可以计算出一个导数, 加速度实际上是速度关于时间的导数。即如果 $v(t)$ 给出了瞬时速度关于时间的函数, 那么加速度就是 $v'(t)$ 。

瞬时加速度

$$a = v'(t)$$

其中 $v(t)$ 是关于时间的函数

在这里我们向前进了一大步, 如果加速度是速度函数的导数, 速度是位移函数的导数, 加速度就是位移函数的二阶导数: 即导数的导数。二阶导数记作 $f''(t)$ 。

瞬时加速度

$$a = v'(t) = y''(t)$$

其中 $v(t)$ 为速度关于时间的函数;

$y(t)$ 为位移关于时间的函数。

可能你会看不懂这一符号, 其实它就是前面步骤的重复。如果知道瞬时速度的函数, 直接求导就可以得到加速度。如果仅仅知道位移关于时间的函数, 经过两次求导(求极限值)也可以很容易的求得瞬时加速度。

【例 9-6】 计算二阶导数

让我们再来观察一次手榴弹的运动。如果我们仅有位移关于时间的函数: $y = f(t) = 10t - 4.9t^2$ 。计算在 $t = 2\text{s}$ 时的瞬时加速度。

解答:

1. 要计算在 $t = 2\text{s}$ 时的瞬时加速度, 需要求两次导数。也就是说, 计算当时间间隔趋近于零时 $f(t)$ 的极限, 然后再对其计算一次当时间间隔趋近于零时的极限。首先来计算第一个导数, 最后再将 $t = 2\text{s}$ 代入:

$$\begin{aligned}
 f'(t) &= \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h} \\
 f'(t) &= \lim_{h \rightarrow 0} \frac{10(t+h) - 4.9(t+h)^2 - (10t - 4.9t^2)}{h} \\
 f'(t) &= \lim_{h \rightarrow 0} \frac{10t + 10h - 4.9(t^2 + 2th + h^2) - 10t + 4.9t^2}{h} \\
 f'(t) &= \lim_{h \rightarrow 0} \frac{10h - 4.9(t^2 + 2th + h^2) + 4.9t^2}{h} \\
 f'(t) &= \lim_{h \rightarrow 0} \frac{10h - 4.9t^2 - 9.8th - 4.9t^2 + 4.9t^2}{h}
 \end{aligned}$$

$$f'(t) = \lim_{h \rightarrow 0} \frac{10h - 9.8th - 4.9h^2}{h}$$

$$f'(t) = \lim_{h \rightarrow 0} \frac{10 - 9.8t - 4.9h}{1}$$

2. 再一次计算导数：当 $h=0$ 时 $f'(t) = 10 - 9.8t$ 。

$$f''(t) = \lim_{h \rightarrow 0} \frac{f'(t+h) - f'(t)}{h}$$

$$f''(t) = \lim_{h \rightarrow 0} \frac{10 - 9.8(t+h) - (10 - 9.8t)}{h}$$

$$f''(t) = \lim_{h \rightarrow 0} \frac{10 - 9.8t - 9.8h - 10 + 9.8t}{h}$$

$$f''(t) = \lim_{h \rightarrow 0} \frac{-9.8h}{h}$$

$$f''(t) = \lim_{h \rightarrow 0} \frac{-9.8}{1}$$

正如你所看到的一样， $a = f''(t) = -9.8$ 。在 $t=2\text{s}$ (或者其他任意时间) 时，它的瞬时加速度为 -9.8 。

自我检测

1. 利用表 9-2 的数据，计算赛车在 $1 \leq t \leq 5$ 时的平均加速度。
2. 例 9-5 中手榴弹的速度方程为 $v = f(t) = -9.8t + 25$ ，计算在 $t=1\text{s}$ 时的瞬时加速度。
3. 假设物体的速度方程为 $v(t) = 30t + 5t^2$ ，计算在 $t=3\text{s}$ 时的加速度。
4. 假设物体的位移方程为 $y(t) = 8t + 3t^2$ ，计算在 $t=2\text{s}$ 时的加速度。

本章总结

本章中，我们不仅探讨了在图像中的速度和加速度的概念，还学习了怎样利用导数来表示速度和加速度。找到了在位移、速度和加速度之间的关系；位移的导数是速度，速度的导数是加速度。如果我们有了物体的“速度-时间”函数，就可以利用导数很容易的得到“速度-时间”函数和“加速度-时间”函数，并写成程序代码。

本章中，我们很容易的利用多项式方程(关于 t 的幂)描述了位移。同时，如果你想利用不同的方程来为一个物体的位置建模，可以参考附录 B 中的内容，那里提供了其他方程的求解方法。希望大家能够继续学习。

能力测试

多项选择

1. 下列哪一名词与导数方程无关？

A. 切线

B. 斜率

C. 半径

D. 极限

2. 判断对错：速度是加速度关于时间的导数。
A. 对 B. 错
3. 判断对错：只能利用多项式求导数。
A. 对 B. 错
4. 判断对错：加速度是位移关于时间的二阶导数。
A. 对 B. 错

练习

练习 1：速度和加速度

1. 回看例 9-1 中的西红柿，计算在下列时间间隔的平均速度：
 $1 \leq t \leq 6$
 $1 \leq t \leq 5$
 $1 \leq t \leq 4$
2. 回看例 9-2 中的手榴弹，计算在 $t = 4\text{s}$ 时的瞬时速度。
3. 如果将一物体抛向空中，它的高度关于时间函数为 $y = f(t) = 6t^2 + 2$ ，计算 $t = 4\text{s}$ 时的瞬时加速度。
4. 假设屏幕上有一辆汽车在行驶，它的位移关于时间的函数为 $y = f(t) = 2t + 10$ ，计算在 $t = 6\text{s}$ 时的导数或者瞬时速度。

练习 2：加速度和二阶导数

1. 利用表 9-2 中的数据，计算赛车在 $0 \leq t \leq 5$ 时的平均加速度。
2. 如果在例 9-5 中的手榴弹的“速度-时间”方程为 $v = f(t) = -11.8t + 12$ ，计算在 $t = 4\text{s}$ 时它的瞬时加速度。
3. 假设物体的“速度-时间”函数为 $v(t) = 15t + 4t^2$ ，计算在 $t = 3\text{s}$ 时它的加速度。
4. 假设物体的“位移-时间”函数为 $y(t) = 25t - 3t^2$ ，计算在 $t = 5\text{s}$ 时它的加速度。

程序设计

编写一段简单的程序，它的输入为物体的位置关于时间的线性方程，利用导数的概念求它的速度方程。

编写一段简单的程序，它的输入为物体的位置关于时间的多项式方程（如 $f(t) = 2t^2 + 5$ ），利用导数的概念求得它的速度方程。

问题讨论

1. 斜率与导数方程的关系。
2. 解释本章中极限的概念。
3. 曲线的切线与导数的关系。
4. 对于导数来说，位移、速度和加速度有何联系？
5. 对于导数来说，平均速度和瞬时速度有何联系？

第 10 章 二维和三维空间运动

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 在二维和三维空间里使用向量计算位移、速度和加速度
- 将向量计算应用到 5 种运动轨迹方程中
- 使用向量为抛物运动建模

介绍

我们已经广泛地讨论了有关一维空间的运动，你也许想知道怎样将这些公式应用于二维和三维空间中。这还需要用到向量。如果你对向量计算有些生疏了，可以回到第 4 章复习一下“向量计算”这一节。一旦你熟悉了如何计算向量，就会发现从一维扩展到二维、三维是多么的容易。

10.1 使用向量

二维和三维空间和一维空间的运动最大区别就是它们还有其他的方向。在一维空间里，我们使用正负号来表示位移、速度和加速度这些向量。

但要在二维和三维空间中必须结合向量来描述运动，也就是说，在第 4 章所描述的向量计算在这里可以用来表示位移、速度和加速度。无论你相信与否，这一节并没有什么新的内容，只是简单将第 8 章的内容和第 4 章的内容联系在一起。

首先来复习一下有关位移的知识，在一维空间中只有正或负两个方向。一维空间里的位移和在数轴上数字的增减是没有什么区别的。例如在图 10-1 中，物体从 50 像素点移动到 450 像素点，总的位移量是 400 像素。而第二个物体从 450 像素点移动到 50 像素点，它的位移是 -400 像素。

在二维空间中不再局限于两个方向，而是在 360° 都有方向。这就意味着仅仅用正负号不足以表示所有的方向。我们必须利用向量来表示方向。在二维空间中仍然利用末位置减去初位置来表示位移。不过并不是对两个简单的数字进行减法运算，而是对两个向量进行计算。记住，只有把向量变成分量形式才能参与计算。如图 10-2 中 P 和 P' 。

我们利用以原点为中心的向量表示物体的位置。由于我们要对其做加减法，所以我们要用分量形式表示。即用 x , y 两个方向上的距离表示，图 10-3 显示了两个向量的分量形式。

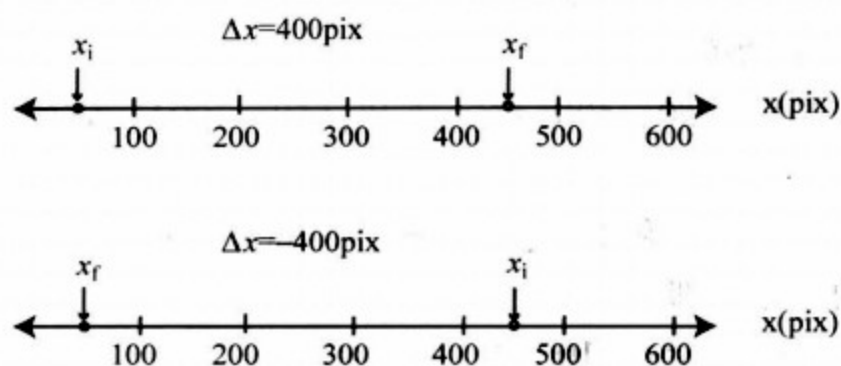


图 10-1 一维空间中的位移

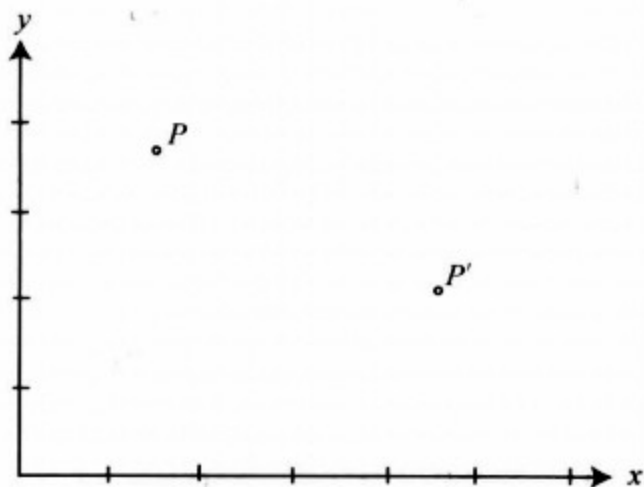


图 10-2 二维空间的起点和终点

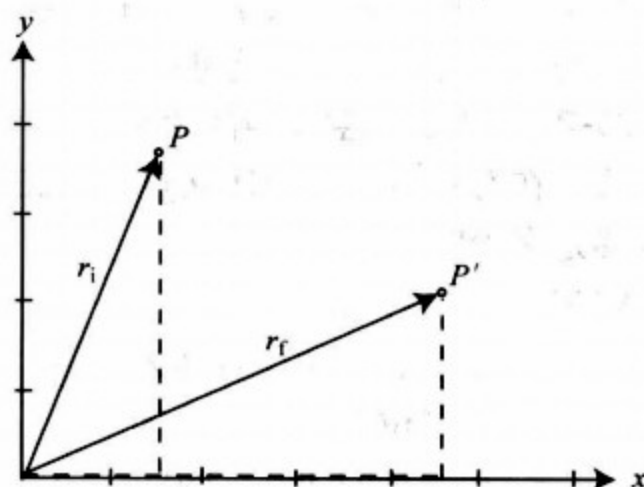


图 10-3 两点的速度表示

注意

在二维空间中进行向量计算时，就不能用 x 表示位置了，这是因为每个向量都由 x 和 y 两个分量来表示。为了避免混淆，我们用 r 来表示位置，用 Δr 表示位移。

现在我们开始研究位移，注意位移是用末位置减去初位置，所以，只要把两个向量做减法($r_f - r_i$)就可以得到位移了。如图 10-4。

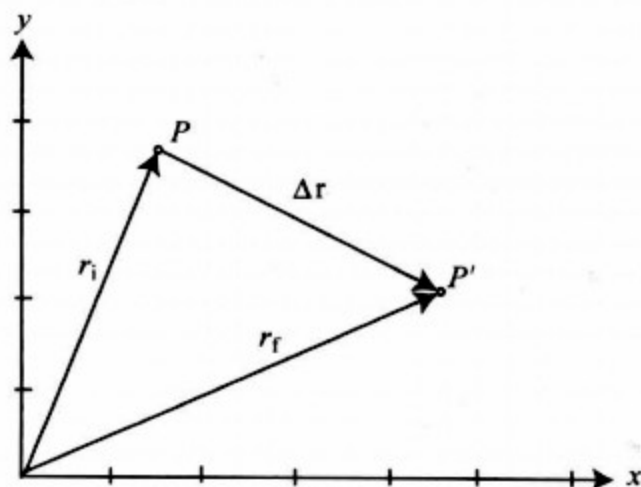


图 10-4 二维空间的位移

2D 位移

$$\Delta r = r_f - r_i$$

其中， r_i 是初始向量， r_f 是末向量。

【例 10-1】 求二维空间中的位移

假如在游戏中，物体由 $P(50, 400)$ 移动到 $P'(550, 100)$ ，它的位移量是多少？

解答：

1. 我们用向量来表示初位置和末位置： $P = [50 \ 400]$ 和 $P' = [550 \ 100]$ 。
2. 位移就是位置的变化，即

$$P' - P = [550 \ 100] - [50 \ 400] = [500 \ -300]$$

3. 总位移是向右 500 像素，向下 300 像素，记作 $[500 \ -300]$ 。

在学会使用向量来表示位置和位移后，再将它们扩展到三维空间中就很简单了：只要在所有的向量上加上 z 分量就可以了。

【例 10-2】 求三维空间中的位移

假如在游戏中一个物体从点 $P(150, 0, 250)$ 移动到点 $P'(400, 250, -300)$ ，那么它的位移是多少呢？

解答：

1. 用向量来表示初位置和末位置： $P = (150 \ 0 \ 250)$ ， $P' = (400 \ 250 \ -300)$ 。
2. 位移就是位置的变化，即

$$P' - P = [400 \ 250 \ -300] - [150 \ 0 \ 250] = [250 \ 250 \ -550]$$

3. 总的位移就是向右 250 像素，向上 250 像素，向后 550 像素，写成向量形式为 $[250 \ 250 \ -550]$ 。

如果我们看到第 4 章中声明的 `3Dvector` 类，就会发现可以用编程的方法轻松的解决有关向量计算的问题。我们只需要定义初位置和末位置的向量，然后利用减法运算即可算出结果：

```
#include "3Dvector.h"
int main()
{
    //定义二维向量
    3Dvector initial(150,0,250);
    3Dvector final(400,250,-300);
    //计算结果
    3Dvector displacement = final - initial;

    return 0;
}
```

现在我们回顾一下在一维空间中有关速度的定义。速度是位置变化的速率。在二维空间中速度仍然是位移和时间的比率。其中唯一的区别就是在二维和三维空间中使用 2D 和 3D 向量代替了正负号而已。这也意味着计算平均速度时需要用向量位移除以标量时间。

如果是在极坐标中，仅需要用向量的模除以时间，并保持方向不变即可。在直角坐标系中，我们把两个分量除以时间。两种方法的定义都是一样的。

2D、3D 的平均速度

$$\bar{v} = \frac{\Delta r}{t} = \frac{r_f - r_i}{t}$$

其中 Δr 为位移向量, t 为时间段。

【例 10-3】 计算三维空间里的平均速度

假设在游戏中智能体在 5 秒内从点 $P(150, 0, 250)$ 移动到点 $P'(400, 250, -300)$, 求它在 5 秒内的平均速度。

解答:

1. 首先用例 10-2 的方法计算出点 P 与 P' 的位移为向量 $[250 \ 250 \ -550]$ 。
2. 平均速度为位移除以时间。由于我们是在直角坐标系中计算位移, 所以我们需要将每一个分量除以 5 秒:

$$\bar{v} = \frac{\Delta r}{t} = \frac{[250 \ 250 \ -550]}{5} = [50 \ 50 \ -110]$$

平均速度的计算在编辑动画时是非常重要的。在游戏中创建和装载动画时, 我们只存储初位置和末位置, 然后通过这两个位置和平均速度用程序计算出它们之间的位置图像。最后我们利用第 4 章中定义的 `3Dvector` 类来写一个计算平均速度的函数, 如下所示:

```
3Dvector averageVelocity(const 3Dvector &Pi, const 3Dvector &Pf,
                        float intervals)
{
    // 计算开始位置与结束位置的位移
    3Dvector temp(pf.x - pi.x, pf.y - pi.y, pf.z - pi.z);
    // 用位移除以时间间隔
    temp = temp * (1 / intervals);
    // 返回结果
    return temp;
}
```

正如你所想的, 加速度的定义也有其 2D 和 3D 形式, 唯一不同的是所有的数学运算都不再对单纯的数字计算, 而是对向量进行计算。实际上所有的运动方程都是在二维空间和三维空间中完成的。在编程中用得最多的是第一、第三和第四方程。

2D 和 3D 中的移动方程

$$v_f = v_i + at$$

$$\Delta r = 1/2(v_f + v_i)t$$

$$\Delta r = v_i t + 1/2at^2$$

这些方程对于任意矢量 a , v_f , v_i 和 Δr 以及标量时间都适合。

注意，第一个方程用于跟踪随着时间的增加，速度的变化。第二和第三个方程用于跟踪位移的变化。同时注意在二维和三维空间中，要使用笛卡儿积的形式来标记速度和位移。这也意味着我们要将第一个方程用三次，第一次用来计算 x 方向的分量，第二次用来计算 y 方向的分量，第三次用来计算 z 方向的分量：

$$v_{fx} = v_{ix} + a_x t$$

$$v_{fy} = v_{iy} + a_y t$$

$$v_{fz} = v_{iz} + a_z t$$

同样，上述这一过程也可以应用到关于位移分量的计算。现在，你已经可以在你的代码中随着时间的增加一帧一帧的跟踪物体的速度和位移了。这种基于时间增加，在每一维上独立的计算方程的方法被称为分量计算。随着你在这个领域里研究的继续，你会经常用到它们。

下面是有关前三个方程的代码：

```
//利用初速度和加速度和时间计算末速度
//输入:vi - 初速度
      a - 加速度
      t - 时间
//输出:末速度
float eqOne_vf(float vi,float a,float t)
{
    return vi +a*t;
}
//根据末速度,初速度和时间计算位移
//输入:vf - 末速度
      vi - 初速度
      t - 时间
//输出:位移
float eqTwo_x(float vf,float vi ,float t)
{
    return .5f*(vf-vi)/t;
}
//根据初速度,加速度和时间计算位移
//输入:vi - 末速度
      t - 时间
      a - 加速度
//输出:位移
float eqThree_x(float vf,float vi ,float a)
{
    return vi*t+.5f*a*t*t;
}
```

任一个方程都会因为其中一个变量的改变而改变其值——例如，第三个方程经过代数变换后，可以用于求解加速度。其代码如下：


```

//根据初速度,位移和时间计算加速度
输入:vi - 初速度
      t - 时间
      a - 加速度
//输出:加速度
float eqThree_ a(float vi,float x ,float a)
{
    return (x - vi * t) * 2 / (t * t);
}

```

方程在应用于二维和三维空间时,与在一维空间中的方法是相似的。只是要注意在进行向量加减时必须是分量形式。另外,每次用时间乘以向量时,只要对每个分量进行标量乘法就可以了。

【例 10-4】 在二维空间中使用方程

假设在游戏中一辆车的当前运动速度为向 53° 方向, 10m/s , 加速度是向 30° 方向, $5\text{m/s}^2@30^\circ$, 那么, 3 秒后的速度是多少?

解答:

1. 列出已知条件:

条 件	求 值
$v_i = 10\text{m/s}@53^\circ$	$v_f = ?$
$t = 3\text{s}$	
$a = 5\text{m/s}^2@30^\circ$	

2. 由于初速度和加速度都是在极坐标中,我们需要把它们转换为直角坐标形式。记住, x 坐标等于模乘以余弦值, y 坐标等于模乘以正弦值。

条 件	求 值
$v_i = 10\text{m/s}@53^\circ$ $= [6 \ 8]$	$v_f = ?$
$t = 3\text{s}$	
$a = 5\text{m/s}^2@30^\circ$ $= [4.3 \ 2.5]$	

3. 我们需要基于列表选择一个方程。然而由于只有第一个方程反映了列表中四个量的关系,所有我们选择第一个方程。

4. 我们只需带入求 v_f 即可:

$$v_f = v_i + at = [6 \ 8] + [4.3 \ 2.5]3 = [6 \ 8] + [12.9 \ 7.5] = [18.9 \ 15.5]$$

所以 v_f 就是矢量 $[18.9, 15.5]$ 。

【例 10-5】 在三维空间中使用方程

假设在赛车游戏中,一辆赛车正等待出发。它的加速度向量是 $[3 \ 0 \ -2]$, 5 秒后它

的位移是多少?

解答:

1. 列出已知条件:

条 件	求 值
$v_i = [0 \ 0 \ 0]$	$\Delta r = ?$
$t = 5s$	
$a = [3 \ 0 \ -2]$	

2. 我们需要基于列表选择一个方程。然而由于只有第四个方程($\Delta r = v_i t + 1/2 a t^2$)反映了列表中三个量的关系, 所有我们选择第四个方程。

3. 我们只需带入求 Δr 即可:

$$\begin{aligned}\Delta r &= v_i t + 1/2 a t^2 = [0 \ 0 \ 0]5 + 1/2[3 \ 0 \ -2](5)^2 \\ &= [0 \ 0 \ 0] + 12.5[3 \ 0 \ -2] \\ &= [37.5 \ 0 \ -25]\end{aligned}$$

所以位移为 $[37.5 \ 0 \ -25]$ 。

如你所见, 第 8 章中讨论的知识也可以应用于二维和三维空间中。唯一复杂的地方就是我们要对向量进行运算, 但这些方程仍然成立, 只不过我们不再对标量进行运算, 而是对向量进行运算。但所幸的是, 向量可以写成分量形式, 简化为数量运算。

自我测试

1. 用向量的两种形式分别表示位置(40, 75)。
2. 计算屏幕上点(250, 300)和点(75, 100)之间的距离。
3. 计算三维空间中点(20, 30, 40)和点(50, 0, -50)之间的距离。
4. 计算在 10 秒之内从点(150, 200)到点(25, 0)的平均速度。
5. 计算在 10 秒之内从点(50, 400, -50)到点(250, 100, -100)的平均速度。
6. 计算在 5 秒内从 $30\text{m/s}@30^\circ$ 移动到 $25\text{m/s}@45^\circ$ 的平均加速度。
7. 计算第 6 题中两点之间的距离。
8. 如果物体的初速度为零, 加速度为 $[4 \ 1 \ 2]$, 那么 5 秒后它的位移是多少?
9. 在第 8 题中物体在 5 秒后的速度为多少?

10.2 抛物运动

游戏中最简单的运动是抛物运动。将任何一个物体扔、踢、投到空中, 它的运动都是抛物运动。回忆一下你最近一次打篮球或棒球时的情景。球即是按照抛物线的路径运动。此时可以翻到第 2 章去复习有关抛物线的知识。回想一下开口向下的抛物线, 是不是很像以一定角度发射的火箭的轨迹? 在游戏中有很多有关第一视角和第三视角射击的抛物线运动, 我们

又该如何计算呢?

正如我们前面所用的方法,二维空间中计算抛物线最简单的方法就是将向量拆分成竖直分量和水平分量,然后计算。由于分量之间是完全独立的,所以可以分别对其进行计算。你可以自己做这个简单的实验来验证这一点。找两个同样的球或者铅笔,还有一张表面光滑的桌子,其中一个球在桌子的边缘旋转,然后观察它以抛物线形式落下。另一个球从同样高度的桌子直接落下,如果它们同时开始运动,那么它们必然会同时落地。这一实验告诉我们,第一个球的水平运动并没有影响到它的竖直运动。两个球在竖直方向上的运动是相同的。如图 10-5 显示了这一过程。

如果我们从前面观察它们,会发现两球的运动是相同的,如图 10-6 所示。

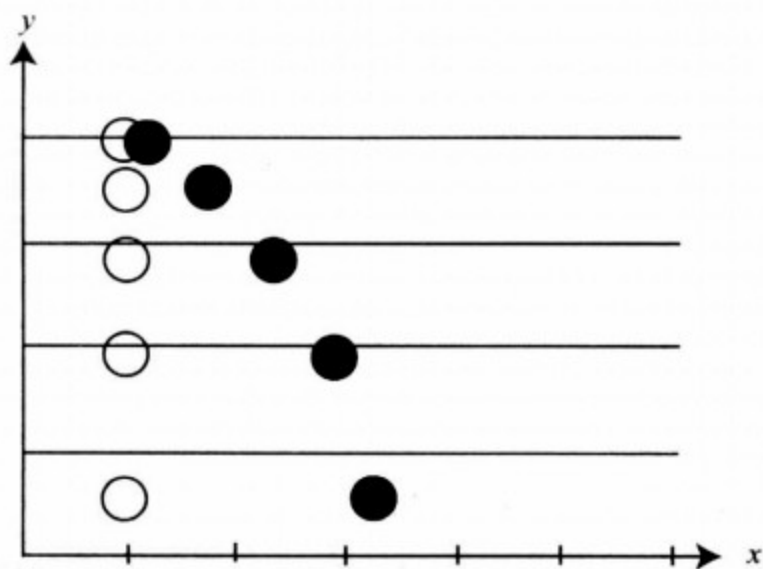


图 10-5 球运动的侧视图

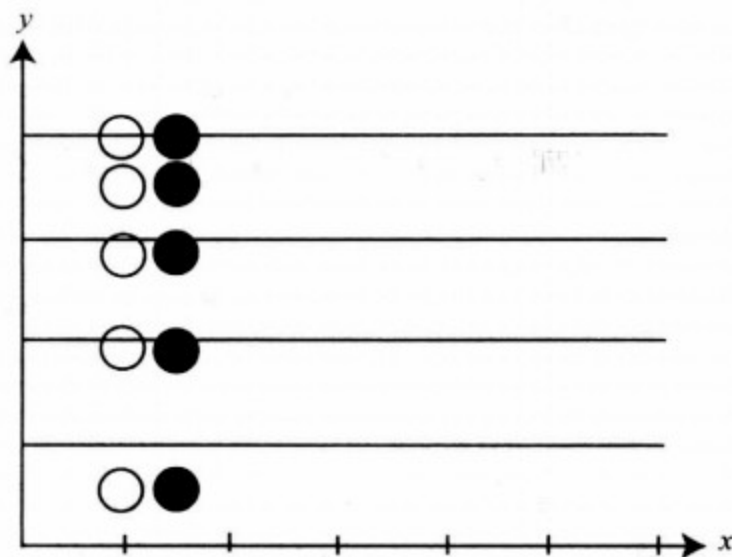


图 10-6 球运动的前视图

现在我们已经知道物体竖直方向上的运动和水平运动是独立的。我们可以将它们分开来讨论。最好能把它们分解成一维空间中的运动,也就是说用正负号来表示。先来看看竖直运动。

如果只看竖直方向的速度、位移和加速度,我们就真的回到了一维空间的竖直运动,也就是第 8 章所讨论的知识。首先来看位移,物体先运动到最高点,然后下落。速度在最高点由一个正值减小到零,在加速下落时变为负值。当然,由于重力的原因,加速度永远是 -9.8m/s^2 。由于我们运用的是一维空间中固定加速度的知识,所以在这里我们可以利用第 8 章中所讨论的第五个方程。

抛物运动的竖直分量

$$a_y = -9.8\text{m/s}^2$$

$$v_{fy} = v_{iy} + at$$

$$\Delta y = v_{iy}t + 1/2a_yt^2$$

【例 10-6】 抛物运动的竖直分量

假设你在给诸如“Spyro the Dragon”这样的游戏编程时,玩家以 20m/s 的速度, 30° 方向

发射炮弹，那么，炮弹可以在空中飞行多长时间？

解答：

1. 炮弹的初速度并不是竖直朝向空中的。所以我们需要找出竖直分量。注意可以利用角度的正弦值计算出竖直分量：

$$v_{iy} = 20\text{m/s}(\sin 30^\circ) = 10\text{m/s}$$

2. 此时我们可以回到一维竖直运动中，首先找出上升到最大高度所需的时间，它的二倍就是炮弹上升再下降所用的总时间。列出我们的已知量：

条 件	求 值
$v_i = 10\text{m/s}$	$t = ?$
$v_f = 0\text{m/s}$	
$a = -9.8\text{m/s}^2$	

3. 基于列表选择方程。在这里，我们选择第一个方程($v_f = v_i + at$)：

$$v_f = v_i + at$$

$$0\text{m/s} = 10\text{m/s} + (-9.8\text{m/s}^2)t$$

$$-10\text{m/s} = (-9.8\text{m/s}^2)t$$

$$t = 1.02\text{s}$$

4. 注意这里计算的只是上升到最大高度所用的时间。总的时间为 $t = 2.04\text{s}$ 。

现在我们把注意力转移到抛物运动的水平分量上去。水平分量的计算其实更加简单，因为在抛物运动中，水平分量的速度是不变的(不计空气阻力)。这就意味着水平分量的加速度是零。也就是说，水平速度是不变的，它等于水平位移除以时间。另外，对于一维空间的匀速运动，我们可以利用第 8 章提到的第五个方程。由于抛物运动中水平加速度为零，则代入到方程中可以简化为一个方程： $(v_x = \Delta x/t)$ 。这也是我们在计算抛物运动的水平分量时唯一需要考虑的方程。

抛物运动的水平分量

$$a_x = 0\text{m/s}^2$$

$$v_x = \Delta x/t$$

【例 10-7】 抛物运动的水平分量

假设你在给诸如“Spyro the Dragon”这样的游戏编程时，玩家以 20m/s 的速度， 30° 方向发射炮弹，假如屏幕的边缘是 30m ，炮弹在空中飞行了多长时间？

解答：

1. 炮弹的初始速度是有一定方向的，我们需要找出水平分量。注意可以利用角度的余弦值计算出水平分量：

$$v_{ix} = 20\text{m/s}(\cos 30^\circ) = 17.32\text{m/s}$$

2. 此时我们可以回到一维水平运动。列出我们的已知量：

条 件	求 值
$v = 17.32\text{m/s}$	$t = ?$
$\Delta x = 30\text{m}$	
$a = 0\text{m/s}^2$	

3. 由于抛物运动的水平运动加速度是零。也就是说速度是常量。我们只需要利用一个方程就可以计算出时间：

$$v = \Delta x / t$$

$$17.32\text{m/s} = 30\text{m} / t$$

$$t = 30\text{m} / (17.32\text{m/s})$$

$$t = 1.73\text{s}$$

炮弹从发射到最后到达屏幕边缘需要 1.73 秒。

现在我们将两个条件合在一起。前面我们曾经提到所有的抛物线向量都可以拆分为彼此独立的分量，所以其实是在对两个标量进行运算。时间是一个标量，也是连接水平和竖直运动的桥梁。例如，刚才的两个例子中运动都是由水平和竖直分量组成的。在示例 10-6 中，它花费了 2.04 秒上升到最高点然后落回地面。在它到达地面时，它在水平方向上也运动了 2.04 秒。我们可以利用这个相同的 2.04 秒计算它的水平飞行距离。现在我们来计算一下炮弹一共飞行了多远。首先来列个表：

条 件	求 值
$v = 17.32\text{m/s}$	$\Delta x = ?$
$t = 2.04\text{s}$	
$a = 0\text{m/s}^2$	

加速度是零，所以可以利用方程 $v = \Delta x / t$ 计算出水平距离：

$$v = \Delta x / t$$

$$17.32\text{m/s} = \Delta x / 2.04\text{s}$$

$$\Delta x = 35.33\text{m}$$

由于炮弹花费了 2.04 秒上升到最高点然后又落下，它也花费了 2.04 秒水平运动了 35.33 米。

现在来看这些已经比较熟悉的例子。在这些例子中，我们分别计算了水平分量和竖直分量，我们同样可以利用草图帮助进行数字运算。

【例 10-8】 从已知高度下落

假如我们做一个诸如马上枪术比赛的 2D 游戏。有一个离地面 8 米高的平台。如果一物体以 10m/s 的速度水平飞下，问落地时的水平位移是多少？

解答：

1. 首先画出运动草图。如图 10-7 所示。

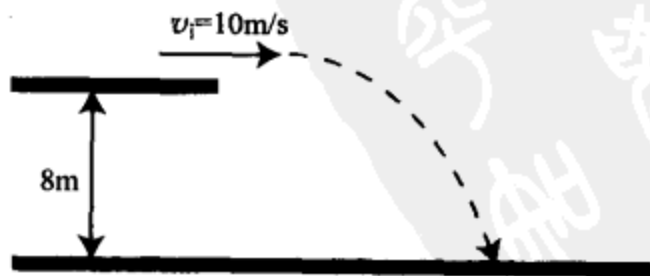


图 10-7 玩家冲出平台

2. 将向量拆分成竖直分量和水平分量两部分，列出两个表：

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 10\text{m/s}$	$\Delta x = ?$	$v_{iy} = 0\text{m/s}$	
$a_x = 0\text{m/s}^2$		$a_y = -9.8\text{m/s}^2$	
		$\Delta y = -8\text{m}$	

由于是平抛，它的初始速度完全是水平的，即 10m/s 就是水平分量，初始竖直速度为 0m/s 。还要注意因为玩家是向下运动，所以这里的竖直位移为负值。

3. 现在我们来看水平位移。由于加速度为零，则只考虑一个方程—— $v_x = \Delta x/t$ 。不过由于还不知道时间为多少，所以现在还不能运用这个方程。由于时间并不分水平或者竖直一说，它在两个表中是一样的，也就是说如果知道玩家竖直运动的时间，我们就可以利用这个时间来计算水平位移。

4. 用竖直方向的分量和一维运动的相应方程计算竖直方向的运动时间。

$$\begin{aligned}\Delta y &= v_{iy}t + 1/2a_yt^2 \\ -8\text{m} &= (0\text{m/s})t + 1/2(-9.8\text{m/s}^2)t^2 \\ -8\text{m} &= 1/2(-9.8\text{m/s}^2)t^2 \\ t^2 &= 1.633\text{s}^2 \\ t &= 1.28\text{s}\end{aligned}$$

5. 现在我们可以利用竖直方向的运动时间和表中的已知量进行计算了：

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 10\text{m/s}$	$\Delta x = ?$	$v_{iy} = 0\text{m/s}$	$t = 1.28\text{s}$
$a_x = 0\text{m/s}^2$		$a_y = -9.8\text{m/s}^2$	
$t = 1.28\text{s}$		$\Delta y = -8\text{m}$	

6. 现在我们已经有足够的信息来计算 Δx 了：

$$\begin{aligned}v_x &= \Delta x/t \\ 10\text{m/s} &= \Delta x/1.28\text{s} \\ \Delta x &= 12.8\text{m}\end{aligned}$$

在这个例子里，物体的初速度是水平方向的，现在我们来看另外的例子，不过它的初速度是有一点角度的。

在游戏中，除非必须在代码中预测物体的位置，不然我们并不对物体落在哪里进行精确计算。在前面我们所描述的游戏场景中，一旦玩家离开了原先的位置，在每一帧中，只要没有监测到物体与地面或者其他物体的碰撞，我们都可以在它的运动中加入重力作用来改变它的相应位置。下面我们来看一段程序，人物的位置和速度被存放在两个浮点型数组中，假设从屏幕上(200, 200)的地方开始让人物具有 10 像素/秒的初速度，程序如下：

```
float player_pos[2] = { 200, 200 };
float player_motion[2] = { 10, 0 };
```

在每一帧中调用下面这个函数用于更新人物的位置：

```
//目的:更新玩家的位置
//输入:无
//输出:更新后的位置
void updatePlayerPos()
{
    for(int i=0;i<2;++i)
        player_pos[i] += player_motion[i];
}
```

我们还需要调用一个函数来更新人物的速度。在 Joust 这样的游戏里，当玩家进入或与物体碰撞时，水平运动就起作用。但在这个例子里，由于重力的作用，我们还需要考虑竖直运动。所以在每一帧时，我们都要考虑在玩家下方是否有物体，如果没有，就需要在它的运动中加入重力的因素：

```
//当玩家位置下方没有物体时更新玩家的动作
//输入:accel - 固定的竖直加速度
        fps - 每秒的帧数,或者说是时间间隔
//输出:更新 player_motion[1](竖直速度)
void updatePlayerVertMotion(float accel,float fps)
{
    if(/* 检查玩家位置下方是否为空 */)
    {
        player_motion[1] = player_motion[1] + accel * fps;
    }
}
```

这段代码也可以解决下面的问题，唯一不同的是玩家在 y 方向有一个负的初速度。你可能被这样的描述弄糊涂了。在本书中，沿 y 方向的负速度通常表示向下运动，但是要记住，在游戏里，竖直方向的负速度意味着向上运动。

【例 10-9】 从已知高度跳下

假如你做了一个类似 Joust 的游戏，在一个离地面高度为 8 米的平台边缘，如果玩家以 10m/s, 30° 的方向跳下，问他在水平方向运动了多少？

解答：

1. 首先画出草图，如图 10-8。
2. 把向量拆分成水平分量和竖直分量，并分别列在两个表中：

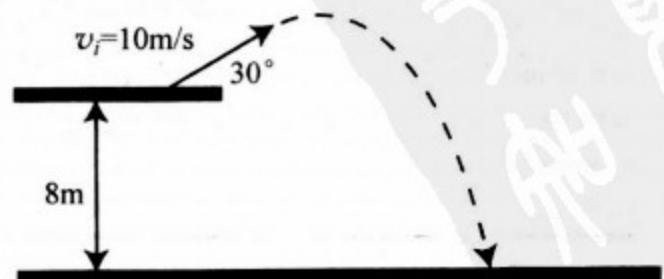


图 10-8 玩家从边缘跳下

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 10\cos 30^\circ$ $= 8.66\text{m/s}$	$\Delta x = ?$	$v_{iy} = 10\sin 30^\circ$ $= 5\text{m/s}$	
$a_x = 0\text{m/s}^2$		$a_y = -9.8\text{m/s}^2$ $\Delta y = -8\text{m}$	

注意即使玩家首先需要向上跳, 但 Δy 仍为 -8m 。因为 Δy 是位移而不是实际的路程。

3. 我们要计算水平位移, 首先要从水平列表着手。由于加速度为零, 所以我们只需要考虑一个方程—— $v_x = \Delta x/t$ 。但是我们并不知道时间是多少。不过无论在水平方向还是竖直方向, 时间都是一样的。也就是说我们只要能知道玩家从跳起到落下所用的时间是多少, 那么我们就可以利用这个时间计算水平位移。

4. 用竖直方向的分量和一维运动的相应方程计算竖直方向的运动时间:

$$\Delta y = v_{iy}t + 1/2a_yt^2$$

$$8\text{m} = (5\text{m/s})t + 1/2(-9.8\text{m/s}^2)t^2$$

5. 我们可以利用二次方程计算, 也可以先求出 v_{fy} , 再用其他方程计算时间。由于第一种方法的计算中含有平方, 比较麻烦, 所以我们利用第二种方法:

$$v_{fy}^2 = v_{iy}^2 + 2a\Delta y$$

$$v_{fy}^2 = (5\text{m/s})^2 + 2(-9.8\text{m/s}^2)(-8\text{m})$$

$$v_{fy}^2 = 181.8$$

$$v_{fy} = -13.48\text{m/s}$$

(注意最后要取负根, 因为玩家是向下运动撞击到地面的。)

6. 将 v_{fy} 带入另一个方程求出时间:

$$v_{fy} = v_{iy} + a_yt$$

$$-13.48\text{m/s} = 5\text{m/s} + (-9.8\text{m/s}^2)t$$

$$t = 1.89\text{s}$$

7. 现在再回头来看这个表, 我们计算出他从跳起到最后落到地面的时间, 也就是说我们知道了水平运动的时间:

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 10\cos 30^\circ$ $= 8.66\text{m/s}$ $= 5\text{m/s}$ $a_x = 0\text{m/s}^2$ $t = 1.89\text{s}$	$\Delta x = ?$	$v_{iy} = 10\sin 30^\circ$ $v_{fy} = -13.48\text{m/s}$ $a_y = -9.8\text{m/s}^2$ $\Delta y = -8\text{m}$	$t = 1.89\text{s}$

8. 现在有足够的信息可以计算 Δx 了:

$$v_x = \Delta x/t$$

$$8.66\text{m/s} = \Delta x / 1.89\text{s}$$

$$\Delta x = 16.33\text{m}$$

玩家在水平方向运动了 16.33 米。

在例 10-8 和例 10-9 中, 我们最终求出了水平位移。这种情况非常常见, 现在我们要再看一个计算竖直位移的例子。

【例 10-10】 射击小球

假设在游戏中智能体从地面向距离 20 米的墙壁射击, 如果小球离开枪口的初速度为 25m/s , 角度为 40° , 那么小球与墙的相撞点在多高的位置?

解答:

1. 首先画出草图, 如图 10-9。

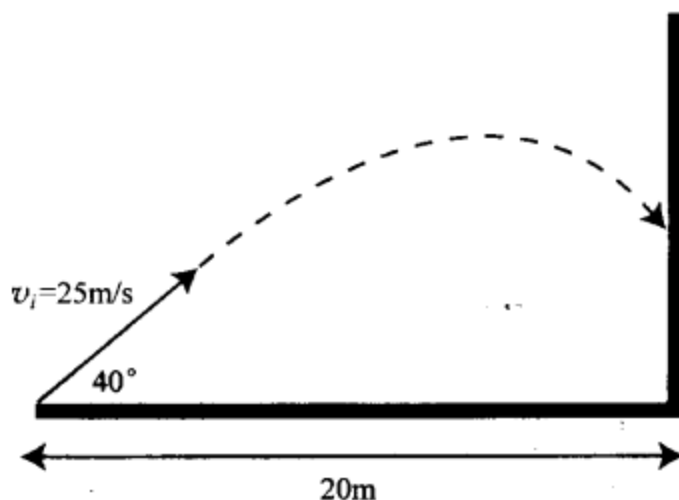


图 10-9 射击小球的运动草图

2. 把向量拆分成水平分量和竖直分量, 并分别列在两个表中:

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 25\cos 40^\circ$		$v_{iy} = 25\sin 40^\circ$	
$= 19.15\text{m/s}$		$= 16.07\text{m/s}$	
$a_x = 0\text{m/s}^2$	$\Delta y = ?$	$a_y = -9.8\text{m/s}^2$	
$\Delta x = 20\text{m}$			

3. 我们要计算竖直位移, 首先要从竖直列表着手。不过我们并没有足够的信息进行计算。但是由于时间是没有方向的, 那么在两个表中它的值是相同的, 也就是说我们能知道小球飞行 20 米所用的时间, 我们就可以利用这个时间计算竖直位移。

4. 由于加速度为零, 所以只能利用下面这个方程来计算时间:

$$v_x = \Delta x / t$$

$$19.15\text{m/s} = 20\text{m} / t$$

$$t = 20\text{m} / (19.15\text{m/s})$$

$$t = 1.04\text{s}$$

5. 再来看这个表，现在我们可以利用时间计算竖直位移了：

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 25\cos 40^\circ$		$v_{iy} = 25\sin 40^\circ$	
$= 19.15\text{m/s}$	$t = 1.04\text{s}$		$\Delta y = ?$
$= 16.07\text{m/s}$		$a_y = -9.8\text{m/s}^2$	
$a_x = 0\text{m/s}^2$		$t = 1.04\text{s}$	
$\Delta x = 20\text{m}$			

6. 现在有足够的信息计算 Δy 了：

$$\Delta y = v_{iy}t + 1/2a_yt^2$$

$$\Delta y = (16.07\text{m/s})(1.04\text{s}) + 1/2(-9.8\text{m/s}^2)(1.04\text{s})^2$$

$$\Delta y = 11.48\text{m}$$

相撞点在 11.48 米高度的地方。

注意

所有的例子中单位都是公米制，如果用不同的单位衡量，那么就需要在计算前将单位统一。

我们所有的例子都是在二维环境中进行的，你也许想知道如何将其转换到三维空间中，其实所有的抛物运动都是二维图形，在二维平面中进行计算，我们只要用数学方法，把它转移到其他平面中进行，不一定必须是 xy 平面，就好像水平竖直平面方向不与 x 轴和 y 轴平行一样。水平方向可以在 xz 平面内，可是竖直方向永远只有一个。

自我测试

1. 从悬崖上以 25° 方向 15m/s 的速度抛球，求 3 秒后球从悬崖上落下了多少米？
2. 在猫和老鼠中，Tom 在一个高 1.5 米的餐桌上追 Jerry，最后 Jerry 忽然跑开，Tom 以 5m/s 的速度从桌子的边缘落下，则 Tom 落地时水平位移是多少？
3. 在游戏中，一个站在 10 米高的建筑物上的敌人以 37° 方向 15m/s 的初速度向你扔石头，那么至少要离建筑物多远才能避免被石头砸到？
4. 用弹弓以 37° 方向 15m/s 的速度向 16 米远的墙上射鸡蛋，鸡蛋会打在墙多高的位置？
5. 假如在四分位以 35° 方向 20m/s 的速度传球，那么接球队员应该站在多远处？

10.3 可视化检测

在光盘上，你会找到一个名为 Cannon Physics 的演示程序，下面是 Michael Fawcett 和 Mike Wigand 关于这个程序的描述：

这个炸弹程序看起来很像一个小巧的坦克大战或者火星克隆游戏。它可以通过一个简单的界面将人们聚集在一起，并能够借助一些简单的小球进行操作。同时我们使这个游戏的框架尽量最小，以便你能够更加关注物理代码，而不是陷于弄清楚什么是碰撞代

码, 什么是 Win32 代码, 什么是图像引擎之类的问题当中。

双击运行这个演示程序, 可以看到在黑色的背景下, 屏幕下方有两个小炮, 当鼠标移向它们时上方会显示一些提示信息。两个小炮可以向各个方向发射炮弹, 并且每一个炮弹的运行轨迹都由具体参数固定下来, 借助中间指令, 你无须记住操作它们的热键。

以 1 号玩家的身份开始, 用右箭头改变视角为 60° , 用上箭头改变作用力为 50N, 保持炮弹质量为 1kg。按下空格键发射炮弹。这时你可以看到上面的数据开始变化, 而炮弹的飞行路径是一条白色的虚线。当炮弹最后停止飞行的时候, 会出现一个设置窗口, 你可以改变这些既定的设置, 并按 OK 键进行确认。在屏幕的左上方, 你可以看到炮弹的初速度为 10m/s。在第 12 章中, 将介绍如何计算这些值, 这里仅仅是让程序自行运算。同时当炮弹发射时告诉你炮弹的速度为 10m/s, 方向是 60° 。

现在轮到 2 号玩家开始发射炮弹, 用右箭头改变视角为 120° , 用上箭头改变作用力为 50N, 保持炮弹质量为 1kg。其实, 也是使用了与 1 号玩家同样的设置, 只是现在需要向左方发射炮弹, 也就是说初速度为 10m/s, 方向是 120° 。在按下空格键发射炮弹之前, 请在纸上计算炮弹能够到达的最大高度, 你的演算可能会是如下这样:

水平方向		竖直分量	
条件	求值	条件	求值
$v_x = 10\cos 120^\circ$			
$= 10\text{m/s}$		$v_{iy} = 10\sin 120^\circ$	
≈ 8.660	$\Delta y = ?$		
$a_x = 0\text{m/s}^2$		$a_y = -9.8\text{m/s}^2$	
		$v_{fy} = 0\text{m/s}$	

对于竖直方向, 你会利用方程 5 计算高度:

$$v_{fy}^2 = v_{iy}^2 + 2a\Delta y$$

$$(0\text{m/s})^2 = (8.660)^2 + 2(-9.8\text{m/s}^2)\Delta y$$

$$\Delta y = 3.8263$$

可能你很可能计算到这里就停下了。但要注意的是你仅仅计算出了 Δy , 它是物体在 y 方向的变化量, 炮弹离地面的高度是这个值吗? 不, 炮弹距离发射位置还有一定高度, 在屏幕上方你会发现初始高度为 2.87m, 所以你需要加上这个 y 坐标值, 因此试验的具体高度应该是 6.69m。

现在按下空格键看看发生了什么。最大高度显示在屏幕上方。它与你计算的高度值吻合吗? 十分吻合。你还可以用不同的数据再做几次试验。

要注意的是, 屏幕上仅显示最后一次试验的数据。这一次, 发射前方有一堵墙, 在设置窗口, 选择下一步和墙, 然后输入墙的有关数据: 左 5m、宽 1m、高 20m。

再以 1 号玩家的身份进行发射, 事先动手计算炮弹打在墙上的高度。如果你仍然使用前面的设置: 发射角 60° 、作用力为 50N、质量为 1kg, 发射的初速度还是 10m/s。墙在正前方 5m 处——距离炮台的高度为 3m, 那么炮弹击中墙的位置点的高度为 Δy 和水平距离(飞行

距离) Δx 分别是多少? 最后不要忘记加上炮台的高度。

当你计算完 Δx 和 Δy 后, 就可以按空格键发射炮弹了。同时按下 P 键进行暂停, 以便获得当前的高度值。再按空格键, 让暂停的炮弹继续飞行, 并击中墙。它的当前高度是多少? 你记录了吗?

在设置窗口出现之前, 你可以让墙远离发射台或者不按下一步而让炮弹重新放回。接下来再以 2 号玩家身份试验。改变炮弹的反射角为 120° , 质量为 1kg , 而用上箭头将作用力加大到 200N 。按空格键把炮弹发射出去, 这时你可以看到屏幕上方显示的初速度为 20m/s 。

当设置窗口出现时, 选择一堵新的墙: 左 25m 、宽 1m 、高 15m 。

现在又轮到 1 号玩家了。我们保持发射角 60° 、质量 1kg , 用上箭头把作用力加大到 200N , 同上次一样, 炮弹的初速度为 20m/s 。使用这些设置, 必须使墙和炮台的距离在 23m 以上。在纸上计算一下, 然后再按空格键发射炮弹, 看看你的计算结果是否正确。你会发现炮弹消失了!

它是否跑到墙的另一边了呢? 没错, 炮弹穿墙而过! 现在 2 号玩家以偏角 120° 、作用力 200N 、质量 1kg 再做这个实验, 同样会使炮弹获得 20m/s 的初速度。假如墙距离炮台在 23m 远处, 那么炮弹会不会消失呢? 还是在纸上先计算一下, 然后再按空格键发射炮弹来验证你的计算结果。你会发现炮弹再次消失了!

这是最后的挑战。用作用力 250N 、质量 1kg , 那么该选择多大的偏角才能使发射的炮弹击中另一架炮台呢? 假如前方炮台在 46m 远处, 看看你能否计算出这个结果, 然后再利用本程序来验证你的计算值, 看看是否吻合。

本章总结

本章中, 我们结合了第 4 章和第 8 章所学的内容, 将向量应用到了位移、速度和加速度当中。我们可以用向量来处理有关运动的 5 个方程, 并用向量检验了在二维(或三维)空间中一种特殊的运动形式——抛物运动。将向量分量应用到抛物运动中可以大大的简化向量的计算。我们发现利用向量的定义可以很简便的将一维空间中定义的物理概念扩展到二维和三维空间中。

能力测试

多项选择

- 判断对错。在二维和三维空间中, “末位置 - 初位置” 的值就是位移。
A. 对
B. 错
- 如果一物体在 5 秒内从点 $(250, 300)$ 移动到点 $(125, 150)$, 它的平均速度是多少?
A. $[125 \quad 150]$
B. $[-125 \quad -150]$
C. $[-25 \quad -30]$
D. $[25 \quad 30]$

3. 如果一物体的中心点在 $[25\text{m} \ 150\text{m} \ -300\text{m}]$ ，随后落下，问三秒后它的中心点在什么位置？
- A. $[-19\text{m} \ 106\text{m} \ -344\text{m}]$ B. $[25\text{m} \ 150\text{m} \ -300\text{m}]$
C. $[25\text{m} \ 135\text{m} \ -300\text{m}]$ D. $[25\text{m} \ 106\text{m} \ -300\text{m}]$
4. 在同一时间同一高度，一个球直接落下，而另一个球从一个平面滚下来，问直接落下的那个球会先着地吗？
- A. 会 B. 不会
5. 判断对错：有关运动的 5 个方程不能应用到二维或三维空间中。
- A. 对 B. 错
6. 判断对错：抛物运动中，水平速度保持不变。
- A. 对 B. 错

练习

练习 1：速度和速率

1. 在笔直的公路上，将巡航控制设置在 65mi/hr ，问 40 分钟后，你走了多远？
2. 假设在游戏中，玩家要跳上一个移动的平台，这个平台只能够左(-)右(+)移动，并以恒定的速度 -6.5 像素/秒运动，如果当前他在 450 像素/秒的位置，问 5 秒后他移动到什么位置(假设他没有转向)？
3. 假设智能体在 20 秒里跑了 28 米，问在这 20 秒里，它的平均速度是多少？
4. 假设在第 3 题中，玩家意识到他落下了一些东西，于是他又返回到他的初始位置，问在这一次的运动中他的平均速度是多少？
5. 假设在游戏中一物体在当前帧的 500 像素点位置。在下一帧中它的位置在 50 像素点，假设帧速度为 30fps ，物体的即时速度为多少？

练习 2：加速度

1. 假设要在 3 秒内将车从 50mi/hr 的速度减速到零，问加速度是多少？
2. 汽车的加速度为 0.4m/s^2 ，要将它的速度从 35mi/hr 提到 70mi/hr 需要多长时间？
3. 汽车的初始速度为 7m/s ，加速度为 1.2m/s^2 ，问 3 秒后它能走多远？

练习 3：有关运动的方程

1. 一个游艇在 500m 的距离从 12m/s 加速到 28m/s ，问它的加速度为多少？
2. 如果在题 1 中游艇移动 125m 需要多长时间？
3. 赛车到达终点时的速度是 43m/s 。此时驾驶员刹车制动，7 秒后将赛车停下来，问车的加速度是多少？
4. 在题 3 中赛车最终停下来滑行了多远？
5. 以 16m/s 的初速度向上抛球，球到达最大高度需要多长时间？
6. 在题 5 中，球的最大高度是多少？

程序设计

程序设计 1：编程输入物体的相关信息

1. 编写一段程序，输入物体当前的位置和速度，返回它在下一帧中的新位置(假设在一维空间中运动且速度为 30fps)。
2. 编写一段程序，输入物体当前速度和加速度，返回下一帧时它的新速度(假设在一维空间中运动且速度为 30fps)。

程序设计 2：运用五个运动方程

创建一个有关五个运动方程的函数库。注意：每个方程都必须至少使用一次，以求出函数中的变量。

程序设计 3：运行比赛演示程序

运行 CD 中的有关本章可视化检测的演示程序。利用它来检查本章中你的习题答案。

程序设计 4：编程输入物体高度

1. 编写一段程序要求输入物体的高度。
2. 在程序中运用简单的图形，将物体放在正确的高度。
3. 接收用户的输入(按下 Enter 键或点开始按钮)，显示物体落到屏幕底端的过程，其中物体的加速度为重力加速度。

讨论问题

1. 速度与速率之间有什么异同？
2. 对于公式 $D = v \times t$ ，为什么要求速度为常量？
3. 平均速度和瞬时速度之间有什么区别？
4. 在什么情况下，平均速度等于瞬时速度？
5. 为什么加速度的单位是位移除以时间的平方？
6. 在什么情况下方程 4 不能求解，而你又是怎样回避利用方程 4 的？

新解
PDG

第11章 牛顿定律

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 计算质量
- 计算物体受力
- 描述正交力
- 计算摩擦力
- 描述牛顿第一定律
- 运用牛顿第二定律计算加速度
- 描述牛顿第三定律

介绍

之前我们已经讨论了如何描述物体运动：物体移动了多远，它运动的速度，它的加速或减速有多快。在之前讨论这些问题时，你有没有考虑过是什么引起了运动呢？本章我们在讨论物体如何运动之前，首先讨论是什么引起了物体的运动。

11.1 力

当我们准备用程序来模拟物体的运动时，首先要对我们所要移动的物体做受力分析。物体所受力的总和决定了它的运动模式。有些力是很显然的：比如敌人打或踢玩家，玩家举起一个物体、抛球或发射导弹等等。但是有些作用力并不那么容易观察到：比如重力，摩擦力，空气阻力或者支持力等等。

我们先分别对它们进行分析，首先从重力开始。

之前讨论过重力加速度，并在很多公式中以常量 g 出现。

物体运动的竖直分量加速度为 $-g$ 或 -9.8m/s^2 。现在我们可以利用重力加速度和物体的质量求出物体所受重力。重力是一个向量，方向指向地心。物体所受重力其实就是自身质量与重力加速度的乘积。

重力

$$w = mg$$

其中 m 为物体质量, g 为重力加速度(地球上重力加速度为 -9.8m/s^2)

在大多数游戏中,我们不需要对重量和质量进行区分。只有在一些特殊的游戏(或者其他情况下)才需要区分。例如在某些场合下,玩家的物理运动或者刹车时都需要计算加速度。下面的函数是用来计算重力的,注意这里返回的并不是一个值,而是一个包含三个值的数组。这主要是因为重力是一个向量而不是标量。

```
//根据物体的质量计算其重量
//输入:mass-物体的质量
      grav-重力加速度
//输出:表示物体重量的三维数组
float *calcWeight3D(float mass, float grav)
{
    //用来保存返回时的物体重量
    //由于重力加速度只在 y 轴上起作用,则仅有 weight[1]的值发生了改变
    float weight[3] = { 0, };

    //计算重量,假定 grav 是一个负数
    weight[1] = mass * grav;

    //返回结果
    return weight;
}
```

注意

物体的重量因重力加速度的不同是可以变化的。比如月球上的重力加速度是地球上的 $1/6$, 那么物体在月球上的重量就是在地球上重量的 $1/6$ 。但是质量是不会改变的。

现在我们来进一步研究重力和质量的关系。二者之间最大的区别是质量是一个标量,而重量是矢量。虽然在世界的任何一种语言中二者都是可以互换的,但它们作为不同的物理概念是有所不同的。

也许你对牛顿这个单位还不熟悉,那么我们先来介绍一下。如果质量以千克为单位,那么用重力加速度乘以质量最后得到的一个单位是 $\text{kg} \times \text{m/s}^2$ 。由于这样念起来很不方便,我们就用牛顿(伟大的物理学家)的名字来命名这个单位,不过也许你更习惯于用磅来表示重量,那么我们就需要对其进行转换。

牛顿转换公式

$$1\text{N} = 1\text{kg} \times \text{m/s}^2$$

$$1\text{N} = 0.2248\text{lbs}$$

注意

大多数情况下，我们都用牛顿来表示重力，这也是我们为什么用公米制的原因。

【例 11-1】 计算重量

物体的质量为 50kg，那么它重多少磅？

解答：

1. 先计算它有多少牛顿：

$$w = mg = 50\text{kg}(-9.8\text{m/s}^2) = -490\text{kg} \times \text{m/s}^2 = -490\text{N}$$

2. 把牛顿转换成磅：

$$-490\text{N}\left(\frac{0.2248\text{lbs}}{1\text{N}}\right) = 110.152\text{lbs}$$

在游戏编程中，经常会发现物体的重量习惯用磅来表示，那么我们需要先把它转换为牛顿再利用它计算质量。

【例 11-2】 计算质量

如果地球上一个物体的重量为 200lbs，它的质量是多少？

解答：

1. 将磅转换为牛顿：

$$-200\text{lbs}\left(\frac{1\text{N}}{0.2248\text{lbs}}\right) \approx -890\text{N}$$

2. 利用重力公式计算质量：

$$w = mg$$

$$-890\text{N} = m(-9.8\text{m/s}^2)$$

$$m = -890\text{N} / -9.8\text{m/s}^2 = 90.78\text{kg}$$

你是否想过为什么我们始终受到重力的作用，但却并不是一直在下落呢？应该有一种力阻止我们掉向地心。没错，是大地阻止我们掉下。大地提供了这样的力——支持力。支持力作用在物体表面，抵消重力并保证它不向下落。如果撤走支撑物体的表面，物体将下落到一个提供这种支持力的平面。支持力是正交的，它总是垂直于物体表面。在第 4 章中用确定面向量的叉乘来解决正交面的问题。

所以说，如果站在地面上，重力把你向下拽，那么支持力就保持你不向下运动。

【例 11-3】 垂直于面的支持力

一辆汽车的重量是 1000lbs，需要多大的力才能支持住它？

解答：

1. 把磅转换成牛顿：

$$-1000\text{lbs}\left(\frac{1\text{N}}{0.2248\text{lbs}}\right) \approx -4448.4\text{N}$$

支持力必须抵消重力，以使汽车不至于飘到空中或陷入地下，所以支持力为 4448.4N。

注意

重量为负值，这是因为它的方向指向地心。也就是说支持力为正值。

【例 11-4】斜面的支持力

重量为 0.25lbs 的球在倾角为 30° 的斜面上滚动，它受到的支持力是多少？

解答：

1. 把磅转换成牛顿：

$$-0.25\text{lbs}\left(\frac{1\text{N}}{0.2248\text{lbs}}\right) \approx -1.112\text{N}$$

2. 注意支持力是永远垂直于接触面的。所以这次不能简单地把方向定为重力的反方向。如图 11-1 所示，它等于重力在斜面上的分量：

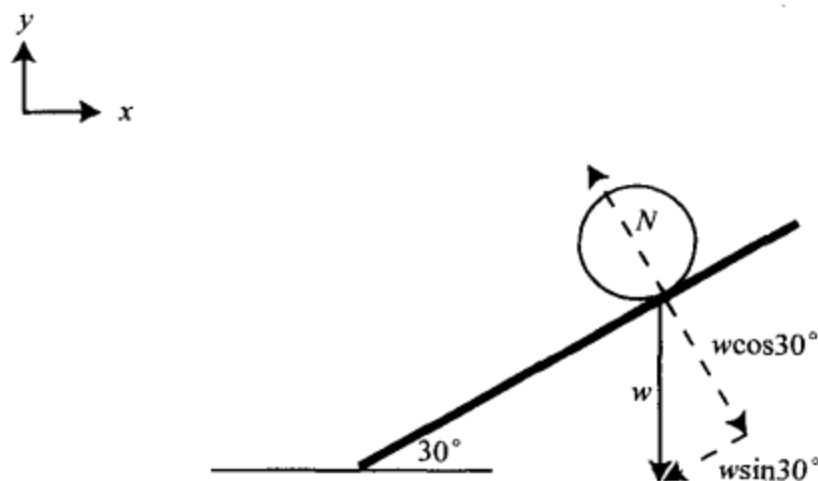


图 11-1 作用于斜面上的支持力

3. 重力在斜面上的分量垂直于斜面，即 $w \cos 30^\circ = -1.112\text{N}(\cos 30^\circ) = -0.963\text{N}$ 。所以，支持力应该是 0.963N。

注意

支持力减小了物体竖直向下的加速度。

另外一种容易被忽略的力是摩擦力。在游戏中，对于编程人员来说，忽略摩擦力不失为一种好的方法，假如摩擦力可以忽略不计，而玩家又没有注意到它，那么大可不必花费太多的时间在它上面。但是如果试图模拟现实情况的话，就需要在计算摩擦力上面花费一些时间了，这完全取决于你要如何控制游戏中的速度。

注意

摩擦力在很多游戏中起着很重要的作用。比如在赛车游戏 GTA 中，Vice City 和 Gran Turismo 3 在潮湿的路面上更容易打滑，而在 Mario 64 中，Mario 可以在冰面上滑过。也不是每一个游戏都需要考虑摩擦力，但在有些游戏中摩擦力起着核心的作用。

实际上，摩擦力分为两种形式：静摩擦力和滑动摩擦力。静摩擦力可以使物体保持稳定的状态，而滑动摩擦力则可以使物体减速。

一般首先计算静摩擦力。如果物体所受的其他力的总和小于静摩擦力，那么物体保持静止。一旦其他的力大于静摩擦力，物体开始运动，静摩擦力变成滑动摩擦力。两种摩擦力都取决于它们的接触面。接触面越光滑，摩擦力越小。也就是说金属物体在冰上的摩擦力要小于橡胶轮胎在干燥地面上的摩擦力。要计算摩擦力，必须要知道摩擦系数。

表 11-1 列出了几种常见物体表面的摩擦系数，只需要记住接触面越光滑，系数越小就可以了。

注意

用希腊字母 μ 来表示摩擦系数。

表 11-1 摩擦系数

接触面	静态系数(μ_s)	滑动系数(μ_k)
干燥齿轮	0.6	0.4
润滑齿轮	0.1	0.05
塑料齿轮	0.041	0.04
刹车内圈	0.4	0.3
橡胶齿轮(干燥)	1.0	0.9
橡胶齿轮(湿润)	0.30	0.25
金属与冰	0.022	0.02
钢轮	0.74	0.57
铝合金	0.61	0.47
铜合金	0.53	0.36
镍合金	1.1	0.53
有机玻璃	0.94	0.40
铜芯玻璃	0.68	0.53
橡胶与谷物(水平放置)	0.62	0.48
橡胶与谷物(竖直放置)	0.54	0.32

现在我们可以利用这些系数来计算两种摩擦力了。

静摩擦力：

$$F_s = -\mu_s N$$

其中 N 为支持力。

注意静摩擦力的方向通常与支持力的垂直方向相反。

滑动摩擦力：

$$F_k = -\mu_k N$$

其中 N 为支持力。

【例 11-5】 计算摩擦力

在赛车游戏中，汽车的重量大约是 1500lbs。根据橡胶与地面的摩擦情况，计算车的静摩擦力和滑动摩擦力。

解答：

1. 将磅转化成牛顿：

$$-1500\text{lbs} \left(\frac{1\text{N}}{0.2248\text{lbs}} \right) \approx -6672.6\text{N}$$

2. 计算支持力。如路面是水平的，那么支持力应该是 6672.6N。

3. 根据表 11-1 中橡胶与干燥路面的摩擦系数 1.0，计算静摩擦力。

$$F_s = -\mu_s N = -1.0(6672.6\text{N}) = -6672.6\text{N}$$

4. 根据表 11-1 中橡胶与干燥路面的摩擦系数 0.9, 计算滑动摩擦力。

$$F_k = -\mu_k N = -0.9(6672.6\text{N}) = -6005.3\text{N}$$

现在我们该把所学的关于摩擦力和支持力的知识应用到游戏中了。将一个盒子放在斜面上, 斜面的倾角不断变大, 当到达某一角度时, 物体开始下滑, 那么这个角度是多少呢? 首先要计算物体所受的支持力, 第二步计算使物体下滑的下滑力。我们利用支持力和接触面的摩擦系数可以确定要使物体运动需要多大的力。一旦下滑力大于这个力, 盒子就开始下滑。在这里, 为了计算方便, 我们把重力看作标量。

```
//判断斜面上的物体是否会移动
//输入:angle - 斜面的倾角
      weight - 物体的重量
      fric_coeff - 两物体表面的静摩擦力
//输出:假如物体移动则返回真,否则为假
bool checkForMotion(float angle,float weight,float fric_coeff)
{
    //计算支持力
    float normal = weight * cosf(angle * PI/180);
    //计算下滑力
    float perpForce = weight * sinf(angle * PI/180);
    //计算保持物体静止的静摩擦力
    float stat_friction = fric_coeff * normal;
    //物体移动则返回真,否则为假
    return perpForce > stat_friction;
}
```

现在把问题加深一步, 计算物体的加速度大小。如果我们用下滑力减去滑动摩擦力, 就能得到物体所受的合力。根据 $F = ma$, 得到 $a = F/m$, 我们可以很容易的求出加速度的大小。注意, 当 checkForMotion() 输出为真时, 才能够调用这个函数。因为如果返回值为假的话, 最后算出的结果会是物体向上运动。

```
//判断斜面上的物体是否会移动
//输入:angle - 斜面的倾角
      weight - 物体的重量
      fric_coeff - 两物体表面的静摩擦力
      mass - 物体的质量
//输出:物体的加速度
float calcAccel(float angle,float weight,float fric_coeff,float mass)
{
    //计算支持力
    float normal = weight * cosf(angle * PI/180);
    //计算下滑力
    float perpForce = weight * sinf(angle * PI/180);
    //计算保持物体静止的静摩擦力
```

```
float kin_friction=fric_coeff*normal;
//计算物体所受的合力
float total_force=perpForce-kin_friction;
//返回物体的加速度
return total_force/mass;
}
```

讨论完物体所受的所有重要的力之后，下一步我们将它们加在一起，计算合力，计作 ΣF 。

注意

Σ 经常用于表示总和。

在一维空间中只有正负两个方向，所以计算 ΣF 是很容易的。只要把正负数相加就可以了。例如，我们可以分析一下你坐下看书时的竖直方向上的力。重力是负值，而椅子给你的支持力是正值。把它们相加，得到的合力为零。如图 11-2 所示。

现在扩展到二维空间中分析。如果你坐在轮椅上，后面有人推你，那么你受的力就既有水平方向的也有竖直方向的。我们只需要用第 4 章所讲的有关向量分量的计算就可以了。记住要将水平分量和竖直分量分开来计算。

分析合力和它们分量的最好方法就是对自由物体进行受力分析。标准的受力分析图用箭头表示物体所受的力。有一个标准是非常有用的，因为只有这样，我们在其他地方看到这种图时，可以很快分析出来它是否符合标准。自由物体的受力分析图有两个基本标准：

- 箭头的长度表示力的大小。
- 所有箭头的起点都在物体的中心，指向力的方向。

现在我们利用自由物体受力图来分析轮椅上的你的受力情况。我们一共讨论了四种力：重力、支持力、摩擦力和推力。如图 11-3 所示。

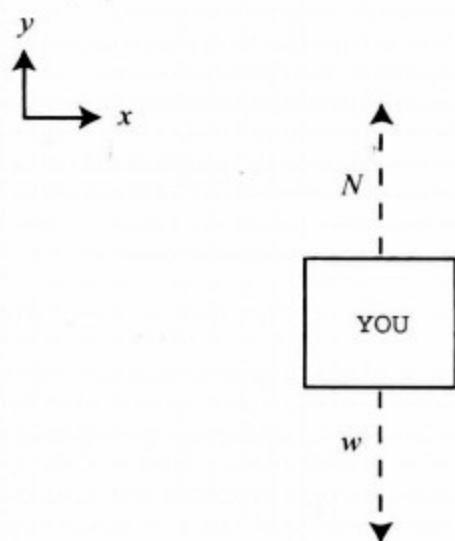


图 11-2 一维空间中的作用力

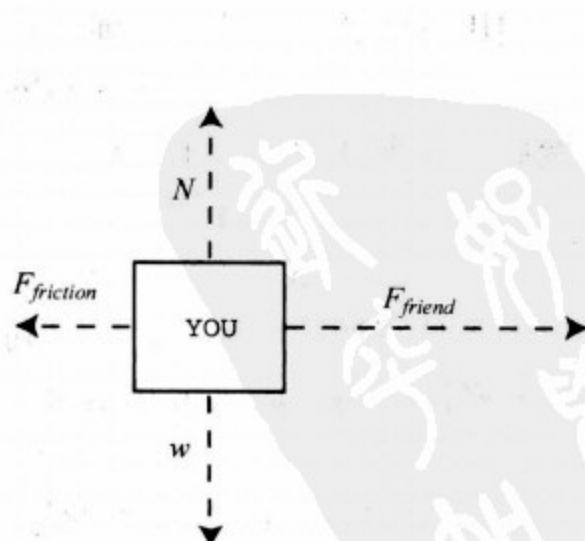


图 11-3 简单的自由物体受力分析图

简单看一下这个图就可以发现合力应该指向推力的方向。现在我们对合力有了感性的认识，该进行数字计算了。注意，在二维空间计算向量时，只能在分量上进行加减。

【例 11-6】 在二维空间中计算合力

假如在游戏里一个冰的世界中，智能体坐在雪橇上在冰面上滑行。绳子的倾斜角为 60° 。玩家用 100N 的力推它，如果雪橇负重为 80lbs 。问合力是多少？

解答：

1. 在自由物体受力图中画出所有的力，如图 11-4 所示。
2. 分解向量，画出所有的分量，如图 11-5 所示。

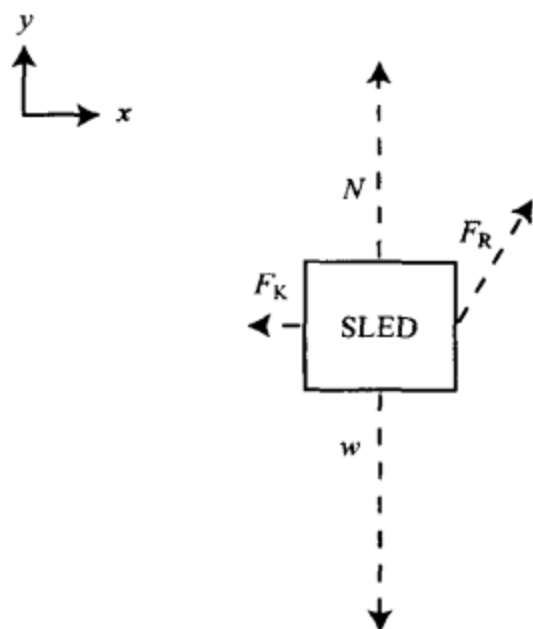


图 11-4 自由物体所受的合力

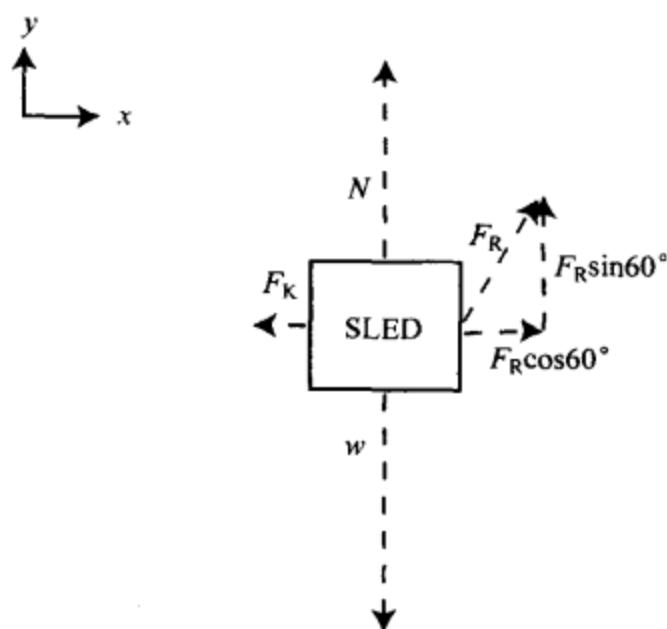


图 11-5 受力框图

3. 计算所有分力的值。首先计算重力，把磅转换成牛顿：

$$w = -80\text{lbs} \left(\frac{1\text{N}}{0.2248\text{lbs}} \right) \approx -355.9\text{N} = [0 \quad -355.9]$$

我们用正弦和余弦值计算绳子拉力的分量：

$$F_R = [100\cos 60^\circ \quad 100\sin 60^\circ] = [50 \quad 86.6]$$

然后计算支持力，为了保持平衡，支持力和方向向下的合力是相等的。我们可以利用绳子拉力的向上分量进行计算：

$$N = -(-355.9\text{N} + 86.6\text{N}) = 355.9\text{N} - 86.6\text{N} = 269.3\text{N} = [0 \quad 269.3]$$

接下来计算滑动摩擦力，可以利用公式 $F_k = -\mu_k N$ ，其中 μ_k 可以参照表 11-1 中的值。注意，滑动摩擦力方向永远和相对运动方向相反。

$$F_k = -\mu_k N = -0.02(269.3\text{N}) = -5.4\text{N} = [-5.4 \quad 0]$$

4. 现在我们只要将这些向量值相加就可以了，注意要用分量形式！

$$\begin{aligned} w + F_R + N + F_k &= [0 \quad -355.9] + [50 \quad 86.6] + [0 \quad 269.3] + [-5.4 \quad 0] \\ &= [44.6 \quad 0] \end{aligned}$$

所以合力为 $[44.6 \quad 0]$ 。

自由物体受力图在分析物体受力方面是非常有用的。我们可以很容易的从受力分析图中知道合力是多少。同时它可以将特殊方向上的力拆分成向量形式进行受力分析。不过，在三维空间中就很难画出这样清晰的分析图了。在这种情况下，可以对向量各方向的分量进行计算，最后再算出合力。下一节我们将介绍合力是怎样影响物体运动的。

自我测试

1. 一个怪物的质量为 80kg ，问它重多少磅？
2. 一个武器在地球上重量是 10lbs ，问它的质量是多少？
3. 如果地球上一物体重 25lbs ，把它放在桌子上，支持力是多少？
4. 50kg 的马车停放在 20° 的斜坡上，问斜坡的支持力是多少？
5. 假设问题 4 中的马车带有橡胶轮子，那么它与刚下过雨的斜坡之间滑动摩擦力是多少？
6. 图 11-6 中物体所受合力是多少？

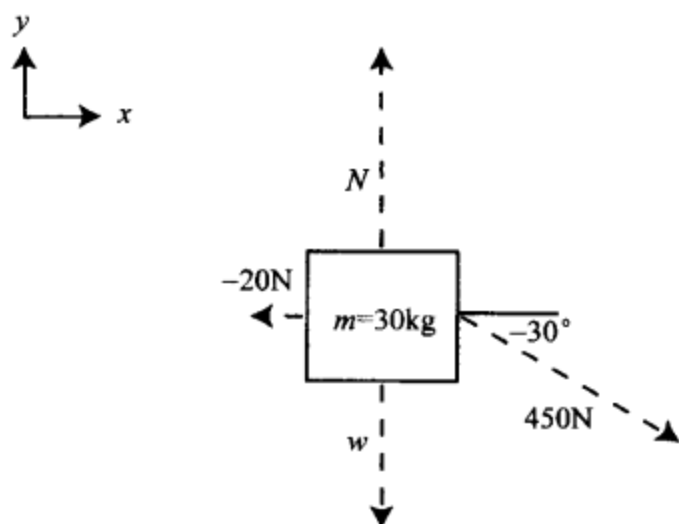


图 11-6 另一个自由物体的受力图

7. 画出下面情况的受力分析图。一个人坐在轮椅上，人重 150lbs ，车重 10lbs ，你用 1200N @ -60° 的力推他，滑动摩擦系数是 0.3 。
8. 计算问题 7 中的合力是多少。

11.2 牛顿定律

刚才我们介绍了如何计算合力，牛顿定律告诉我们力影响物体运动。在 17 世纪后期，牛顿不仅发现了影响物体运动的三大定律（不是用于量子物理），而且给出了定量计算它们的方法。下面我们开始一一进行介绍。

牛顿第一定律是这样描述的：当物体所受合力为零时，它将保持原有运动状态不变。一个很明显的例子就是，当你坐在椅子上不动时，重力使你向下运动，支持力使你向上运动，二者合力为零，而你就保持不动。

如果 $F_{\text{net}} = 0$, 物体的运动状态保持不变。

换句话说, 如果是静止物体, 那么它将继续保持静止, 直到有其他力改变其运动状态为止。同样, 如果物体运动, 那么它将保持原有的方向和速度一直运动下去, 直到有其他力改变其状态为止。你也许会想, 在地上滚动的球速度会越来越慢, 直到最后停下, 其实球的合力不是零, 不要忘记还有摩擦力的存在, 它就是使物体最终停下来的外力。

【例 11-7】将牛顿第一定律应用于冰上曲棍球

在冰上曲棍球游戏中, 桌面摩擦力忽略不计。一名球员击中球, 球的初速度为 $[10 \ 5]$ 。3 秒后球的速度是多少? (没有相撞)

解答:

由于没有摩擦力, 那么合外力为零, 则小球仍然保持匀速即 $[10 \ 5]$ 。

注意

在例 11-7 中, 只有当球受到撞击时速度才会改变。

牛顿第一定律告诉我们当合力为零时物体的运动规律, 但如果合力不为零, 又会怎样呢? 我们知道合外力会影响到物体的运动, 那么具体又是怎样影响的呢? 牛顿第二定律将告诉我们答案。

牛顿第二定律

$$F_{\text{net}} = ma$$

其中 m 为物体质量, a 为加速度。

注意

当物体所受到的合力为零时, 其加速度一定是零。这是牛顿第一定律的内容。

这条定律中告诉我们很多内容。首先, 一个物体所受合力越大, 速度改变的越快。其次, 两物体如果受到合力相同, 那么质量较小的物体速度改变的快。最重要的是, 我们可以根据牛顿第二定律利用质量和合力计算物体的加速度。

【例 11-8】牛顿第二定律

在例 11-6 中, 雪橇所受合力为 $[44.6 \ 0]$, 问它的加速度是多少?

解答:

1. 例 11-6 中雪橇的重量是 -355.9 N 。利用重量求质量:

$$\begin{aligned} w &= mg \\ -355.9 \text{ N} &= m(-9.8 \text{ m/s}^2) \\ m &= 36.3 \text{ kg} \end{aligned}$$

2. 利用合力和质量计算加速度:

$$\begin{aligned} F_{\text{net}} &= ma \\ \begin{bmatrix} 44.6 \\ 0 \end{bmatrix} &= 36.3 \begin{bmatrix} a_x \\ a_y \end{bmatrix} \end{aligned}$$

$$a = \begin{bmatrix} 1.2 \\ 0 \end{bmatrix}$$

在计算了加速度之后,我们就可以把它应用于第 10 章中的五个运动方程中去了。最重要的是要记住牛顿第二定律中的合力可以随时改变。只有在合力起作用时,物体才有加速度。一旦外力为零,加速度即为零,速度保持不变(牛顿第一定律)。不过,在游戏中是一帧一帧进行计算的,只要合力为零,下一帧加速度即为零。所以,我们从计算物体受力开始编程,继而决定加速度、速度和位移,最终我们可以得到随着时间增加而得的物体的运动轨迹。

【例 11-9】 牛顿第二定律和五个运动方程

在游戏的俯视图中,玩家要滚动石头去打开门,石头本身是静止的,重是 50lbs,假设玩家以 400N@30°的力推石头,滑动摩擦系数为 0.25,那么在 1、2、3 秒后石头移动了多远?

解答:

1. 首先从物体受力分析图开始。如图 11-7。

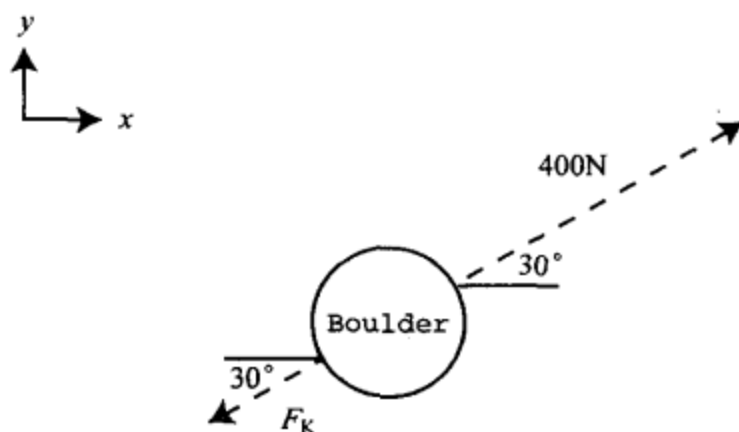


图 11-7 石头的受力分析图

2. 计算 F_K :

$$w = -50\text{lbs} = -222.42\text{N}$$

$$N = 222.42\text{N}$$

$$F_K = -\mu_K N = -0.25(222.42\text{N}) = -55.605\text{N}(\text{opposite of } F_P)$$

3. 计算合力。由于所有的力都以分量形式相加,所以把力拆分成分量形式:

$$F_P = 400\text{N}@30^\circ = [346.41 \quad 200]$$

$$F_K = 55.605\text{N}@210^\circ = [-48.16 \quad -27.80]$$

$$F_{\text{net}} = F_P + F_K = [298.25 \quad 172.20]$$

4. 要运用牛顿第二定律,首先要求出物体的质量:

$$w = mg$$

$$-222.42\text{N} = m(-9.8\text{m/s}^2)$$

$$m = 22.7\text{kg}$$

5. 利用合力和质量求出加速度:

$$F_{\text{net}} = ma$$

$$\begin{bmatrix} 298.25 \\ 172.20 \end{bmatrix} = 22.7 \begin{bmatrix} a_x \\ a_y \end{bmatrix}$$

$$a = \begin{bmatrix} 13.14 \\ 7.59 \end{bmatrix}$$

6. 求出加速度以后, 我们就可以像前几章那样利用五个运动方程求出物体位移了。由于物体从静止开始运动, 所以初速度为 $\begin{bmatrix} 0 & 0 \end{bmatrix}$ 。如果知道加速度、初速度和时间, 就可以利用公式 4 求出物体的位移:

$$\Delta r = v_i t + 1/2 a t^2$$

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} t + 0.5 \begin{bmatrix} 13.14 \\ 7.59 \end{bmatrix} t^2$$

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = 0.5 \begin{bmatrix} 13.14 \\ 7.59 \end{bmatrix} t^2$$

7. 对三个时间分别计算位移:

在 $t = 1\text{s}$ 时

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = 0.5 \begin{bmatrix} 13.14 \\ 7.59 \end{bmatrix} (1)^2 = \begin{bmatrix} 6.57 \\ 3.80 \end{bmatrix}$$

在 $t = 2\text{s}$ 时

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = 0.5 \begin{bmatrix} 13.14 \\ 7.59 \end{bmatrix} (2)^2 = \begin{bmatrix} 26.28 \\ 15.18 \end{bmatrix}$$

在 $t = 3\text{s}$ 时

$$\begin{bmatrix} r_x \\ r_y \end{bmatrix} = 0.5 \begin{bmatrix} 13.14 \\ 7.59 \end{bmatrix} (3)^2 = \begin{bmatrix} 59.13 \\ 34.16 \end{bmatrix}$$

虽然在编程中不经常用到牛顿第三定律, 但是不介绍它显然是不完整的。下面来简要介绍一下牛顿第三定律。你可能听说过“每一个作用力, 都有一个对应的反作用力”。牛顿第三定律就是以这种等大反向力为基础的。比如在战斗游戏中, 我攻击了对手, 也就是说我给对手施加了一个力。与此同时, 对手身上也产生了一个反作用力在我的拳头上。这也是为什么我的拳头最终会停下来(而且为什么会疼)的原因。

牛顿第三定律

对于每一个力, 都有一个反作用力。或者说, 当两物体相碰撞时, 它们之间就会产生大小相等、方向相反的一对作用力。

这一章实际上介绍了有关物理模拟的大致框架。在研究物体运动轨迹之前, 必须先要弄清楚是什么导致了物体的运动, 也就是力。当你将作用在物体上的所有的力都模拟出来时, 将它们相加即可得到合力。如果合力为零, 物体将保持匀速运动。如果合力不为零, 你可以利用质量求出加速度, 继而决定物体可以运动多快, 可以走多远。

自我测试

1. 将这本书放在桌子上，且没有其他东西触碰它，那么 2 秒后它在哪里呢？
2. 假如你在太空站里，那里没有摩擦力，你向同伴以 5m/s @ 45° 抛一支铅笔，问铅笔到达同伴手中时的速度是多少？
3. 如图 11-8 所示的物体加速度是多少？

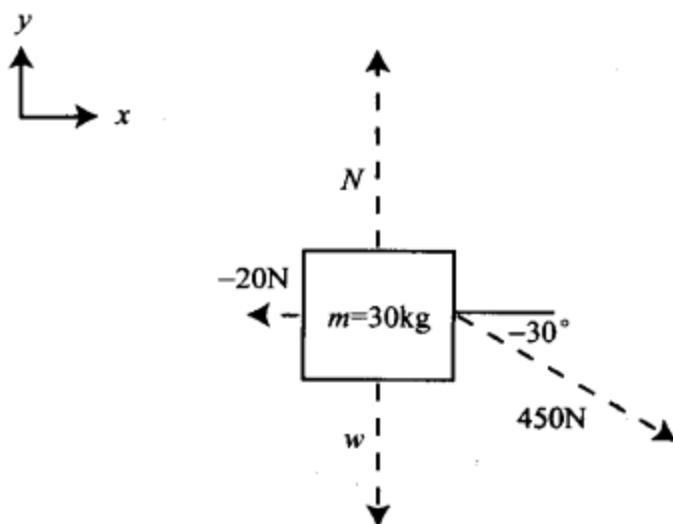


图 11-8 题 3 物体的受力分析图

4. 如果第 3 题中的物体从静止开始运动，5 秒后，它的位移是多少？
5. 在游戏的俯视图中，玩家需要将装满宝藏的卡车推回家。卡车重 35lbs，如果玩家以 200N @ -60° 的恒力推车，滑动摩擦系数为 0.05，问卡车的加速度为多少？
6. 题 5 中的卡车在 3 秒后的位移是多少？

本章总结

本章中，我们花了大量的时间来讨论是什么引起了物体的运动，并将它们与前面所讨论的物体运动联系起来。我们首先简短的介绍了质量和重量的区别，这样可以利用质量来确定物体将怎样运动。另外还讨论了几种不同的影响物体运动的力，即接触力、支持力和摩擦力。随后对物体上作用的力求和，再利用牛顿定律确定物体将怎样运动。最后我们回到了五个运动方程来讨论物体相对于时间的运动轨迹。

能力测试

多项选择

1. 判断对错：质量和重量是一回事。
A. 对
B. 错
2. 下面哪个选项不能提供力？
A. 时间
B. 重量
C. 摩擦力
D. 大地

程序设计 2：编程求物体重量

1. 编写一个简单的程序，根据物体的重量和所受的力，求物体的加速度。
2. 编写一个简单的程序，按照用户的输入，根据物体的重量和所受的力，显示物体随着时间增加在屏幕上的运动轨迹。

讨论问题

1. 质量和重量之间有什么区别？
2. 牛顿第一定律和第二定律有什么联系？
3. 你能想出一个将牛顿第三定律应用到游戏中的例子吗？
4. 静摩擦力与滑动摩擦力之间有什么区别？
5. 牛顿第二定律是如何应用到五个运动方程中的？

新学网
PDG

第 12 章 能 量

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 确定并计算功
- 确定并计算动能和势能
- 运用功能原理计算速度
- 机械能守恒定律

介绍

你是否注意到在玩过山车时，不需要任何引擎同样可以越过顶峰？另外，在途中根本没有人造的减速，速度却一直在变化。这就是机械能在起作用。

本章将能量作为另一种描述运动的方法来讨论。由于游戏的类型不同，它的效率可能也有所不同。但不失为一种候选的方法。

注意

在 Metroid Prime 中对 morph ball 运用能量定律就显示出了显著的效果。

12.1 功和动能

在研究能量模型之前，我们先定义两个量：机械能和功。这两个词在我们的日常生活中是很常用的，所以你一定对它有一定的了解。但要注意的是：在物理中它有着严格的定义。功等于力乘以(力方向上的)位移。

功

$$W = F\Delta x$$

其中 Δx 为位移， F 为位移方向上的力。

假如功定义为力乘以位移，也就是说物体移动必然有功。这也许与你对功的理解相矛盾。比如，当你费尽力气推一堵墙时，虽然你感觉你做了很大的功，但从物理的角度上讲，你根本没有做功，因为墙没有移动。

有关这个定义的另一很重要的方面是，力和位移必须在同一方向。也许是反方向的力，但必须与位移同一方向。由于力和功都是向量，所以只有方向相同，才能够直接相乘。

比如，游戏中智能体需要推着装满财宝的货车，如图 12-1 所示。它可能以一定的角度来推车，但货车只能在平面上水平运动。也就是说你必须用力在水平方向的分量计算功，而不是全部的力。

在这个例子之前，我们需要先讨论有关向量的内容。在第 4 章中提到了几种计算向量的方法。本章需要用到包括点乘、单位化和向量乘法的有关知识。为了使代码更易读，本章中的程序都是围绕向量定义展开的：

2D vector

```
typedef struct
{
    float x;
    float y;
} vector2D;
```

3D vector

```
typedef struct
{
    float x;
    float y;
    float z;
} vector3D;
```

这样使用向量可以简化数学运算。你也可以用前面章节提到的或者其他方法来替代它。这个向量库提供了我们所需计算的向量的所有细节。

我们对于功要介绍的最后一个细节就是它的单位。如果现在你还没有找到答案，最容易的办法就是进行物理模拟。如果我们采用公米制，那么力的单位就是牛，功的单位就是牛·米，我们把它称作焦耳(J)，即 $1\text{J} = 1\text{N} \cdot \text{m}$ 。如果我们使用英制，那么功的单位就是英尺·磅(ft·lbs)。

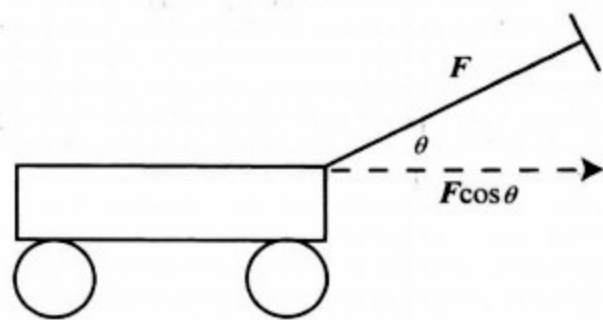


图 12-1 以一定角度推车

焦耳(J)

$$1\text{J} = 1\text{N} \cdot \text{m}$$

【例 12-1】 计算功

假设游戏中玩家必须至少将墙推动 2m 才可以得到奖励。如果玩家的水平推力为 1500N，摩擦力为 50N，问外界对墙做的总功是多少？

解答：

首先从功的定义开始， $W = F\Delta x$ 。此时合力为 $1500\text{N} - 50\text{N} = 1450\text{N}$ ，位移为 2m。

$$\begin{aligned}
 W &= F\Delta x \\
 &= 1450\text{N}(2\text{m}) \\
 &= 2900\text{N} \cdot \text{m} \\
 &= 2900\text{J}
 \end{aligned}$$

有关计算功的编程也是非常简单的。下面的函数根据推力、摩擦力和位移返回功的值：

```
float calculateWork(float force, float friction, float displacement)
{
    //计算合力
    float netForce = force - friction;

    //乘以位移
    float temp = displacement * netForce;
    return temp;
}
```

这个函数输出单位为焦耳的一个浮点型数值。在计算时要确保单位的正确性。有关这些知识在第 6 章里有详细的介绍。

注意

$\text{N} \cdot \text{m}$ 和 J 是完全等价的，根据喜好用哪一个都可以。

【例 12-2】 计算有偏角的力所做的功

我们重新回到那个马车的例子。假如给了马车一个 $1600\text{N}@60^\circ$ 的力，摩擦力为 25N ，问将马车推动 5m ，所做的功是多少？

解答：

1. 在这里力和位移成一定的角度，而位移的方向是水平的。也就是说只有力在水平方向的分量做功：

$$\begin{aligned}
 F_{\text{Cx}} &= F_{\text{C}}(\cos\theta) \\
 &= 1600\text{N}(\cos 60^\circ) \\
 &= 1600\text{N}(0.5) \\
 &= 800\text{N}
 \end{aligned}$$

2. 现在计算合力：

$$\begin{aligned}
 F_{\text{net}} &= F_{\text{Cx}} - F_{\text{f}} \\
 &= 800\text{N} - 25\text{N} \\
 &= 775\text{N}
 \end{aligned}$$

3. 最后，我们根据功的定义计算功：

$$\begin{aligned}
 W &= F_{\text{net}}\Delta x \\
 &= 775\text{N}(5\text{m}) \\
 &= 3875\text{N} \cdot \text{m} \\
 &= 3875\text{J}
 \end{aligned}$$

根据游戏物理模型，经常需要计算有偏角的力。下面的函数输出有偏角的力所做的功：

```
float calculateAngledWork(vector2D vect,float friction,float displacement)
{
    float temp;
    //注意弧度的转换...参见第3章
    temp = cos(DegreesToRads(vect.y));

    //计算水平合力;
    float horizForce = vect.x * temp;

    float work = calculateWork(horizForce,friction,displacement);
    return work;
}
```

在这里，我们首先看 2Dvector 类型的应用。我们首先计算力的正弦和余弦，然后计算水平方向上的力，最后利用例 12-1 中有关功的函数进行计算。

【例 12-3】 计算力和位移在不同角度时所做的功

在游戏的俯视图中，合力大小为 2000N@60°，物体的位移却是 3m@30°，那么力对物体所做的功是多少？

解答：

1. 本题最大的挑战就是两个向量有着不同的倾角。首先，将向量拆分成分量形式可以使计算更加容易：

$$F = 2000\text{N}@60^\circ = [2000(\cos 60^\circ) \ 2000(\sin 60^\circ)] = [1000 \ 1732]$$

$$\Delta r = 3\text{m}@30^\circ = [3(\cos 30^\circ) \ 3(\sin 30^\circ)] = [2.6 \ 1.5]$$

2. 现在我们需要通过点乘找到力在位移方向上的分量。注意，力和位移单位向量的点乘得到的是投影的长度。投影与位移单位向量的乘积才是力在位移方向上的分量：

$$\Delta \hat{r} = [2.6/3 \ 1.5/3] = [0.866 \ 0.5]$$

$$|F_r| = F \cdot \Delta \hat{r} = [1000 \ 1732] \cdot [0.866 \ 0.5] = 1732$$

$$F_r = |F_r| \cdot \Delta \hat{r} = 1732[0.866 \ 0.5] = [1500 \ 866] = 1732\text{N}@30^\circ$$

3. 如果力和位移在同一方向(30°)，我们只需要把它们模相乘即可。利用功的定义计算：

$$\begin{aligned} W &= F_r \Delta r \\ &= 1732\text{N}(3\text{m}) \\ &= 5196\text{N} \cdot \text{m} \\ &= 5196\text{J} \end{aligned}$$

对应这一例子的程序代码要用到第 4 章中定义的一些复杂计算。同时它也遵循前面例子中提到的一些注释：


```
float calculateAngledWorkFromVector(vector2D initial, vector2D change)
```

```
{
```

```
    vector2D force, displacement, displacementNormal, temp;
```

```
    float projectionLength, work;
```

```
    //将极坐标转换为直角坐标
```

```
    force = polarToCartesian(initial);
```

```
    displacement = polarToCartesian(change);
```

```
    //单位化位移单位向量
```

```
    displacementNormal = normalizeVector(displacement);
```

```
    //投影的长度为力与位移单位向量的点乘
```

```
    projectionLength = dotProduct(force, displacementNormal);
```

```
    //显示投影长度
```

```
    cout << "Projection Length" << projectionLength << "\n";
```

```
    //计算投影与位移单位向量的乘积
```

```
    temp = scalarMultiply(projectionLength, displacementNormal);
```

```
    //重新转换成极坐标,注意弧度的转换
```

```
    temp = cartesianToPolar(temp);
```

```
    //最终,完成计算
```

```
    work = temp.x * change.x;
```

```
    return work;
```

```
}
```

这个函数首先将极坐标转化为直角坐标,这样我们才能对力和位移进行计算。

第二步利用 `normalizeVector()` 函数将位移向量单位化,然后用 `dotproduct()` 函数计算投影长度,再用它与 `displacementNormal` 相乘得到最终需要计算的向量。

要想完全理解这一函数需要花费一定的精力。您可以看一下示例文件,并在实际操作中检验其功能。这是一个快速点乘和计算单位向量的方法:

```
vector3D normalizeVector(vector3D vect)
```

```
{
```

```
    float mag, square;
```

```
    vector3D temp;
```

```
    square = squareVector(vect);
```

```

mag = sqrt(square);
temp.x = vect.x/mag;
temp.y = vect.y/mag;
temp.z = vect.y/mag;
return temp;
}

```

注意，我们首先得到向量的模，然后根据原向量，把向量化为分量。

```

float dotProduct(vector3D a, vector3D b)
{
    vector3D temp;
    float scalar;
    temp.x = a.x*b.x;
    temp.y = a.y*b.y;
    temp.z = a.z*b.z;
    scalar = sumVector(temp);
    return scalar;
}

```

点乘是将所有对应分量的乘积进行相加。注意，点乘的结果是一个标量。如果对这些函数有什么疑问可以回到第4章进行复习。

注意

由于点乘的结果是一个标量，所以一旦力和位移在同一方向，我们就可以忽略一些步骤对模直接进行乘法计算。

下面开始定义动能。动能是物体由于运动而具有的能量。物体移动的越快，动能越大。在这一点上，它和动量相似(速度×质量)。

动能的定义是质量的一半乘以速率的平方，即 $1/2mv^2$

动能

$$KE = 1/2mv^2$$

其中 m 为质量， v 为速度。

注意这个定义是基于速率而不是速度(速率是速度的标量形式，也就是说将速度除去方向)。

由于动能也是一个标量，所以无所谓方向问题。物体的动能随着速度的变化而变化，所以可以和研究速度一样，计算出一段时间里的初动能和末动能。

另外，使用公米制更便于计算。如果速度以 m/s 为单位，质量以 kg 为单位，那么动能的单位是焦耳：

$$\text{kg(m/s)}^2 = \text{kg(m}^2/\text{s}^2) = (\text{kg} \cdot \text{m/s}) \cdot \text{m} = \text{N} \cdot \text{m} = \text{J}$$

【例 12-4】 计算动能

假设仪器可以测出棒球的即时速度。问质量为 145g，速度为 90mi/hr 的棒球动能是多少？

解答：

1. 首先转换单位，把速度变成单位为 m/s 的形式，而质量转换成用 kg 表示：

$$\left(\frac{90\text{mi}}{1\text{hr}}\right)\left(\frac{1609\text{mi}}{1\text{mi}}\right)\left(\frac{1\text{hr}}{60\text{min}}\right)\left(\frac{1\text{min}}{60\text{s}}\right) = 40.225\text{m/s}$$

$$145\text{g} = 0.145\text{kg}$$

2. 现在我们可以将它们带入动能方程：

$$KE = 1/2mv^2 = 1/2(0.145\text{kg})(40.225\text{m/s})^2 = 117.3\text{J}$$

编程计算也十分简单。下面的函数可以计算物体的动能，其中质量以 kg 为单位，速度以 m/s 为单位：

```
float calculateKineticEnergy(float mass, float speed)
{
    float KE;
    KE = (mass/2)*(pow(speed,2));
    return KE;
}
```

我们已经介绍了功和动能。现在我们用新的定理来表示两者之间的关系，这就是功能原理。它的内容是，外界对物体所做功的总和等于动能的改变量。

功能原理

$$W = \Delta KE = KE_f - KE_i$$

或

$$F\Delta X = 1/2mv_f^2 - 1/2mv_i^2$$

其中力和位移在同一方向上， v 为速率。

这一定理主要是说，如果有力加在物体上，物体就会加速或减速。这和牛顿第二定律是一致的，用这种方法可以根据物体的位置计算出物体的末速度，让我们用一个例子向大家展示怎样运用这个定理。

【例 12-5】 功能原理

假设为一个棒球游戏编程，游击手需要将球扔给一垒手，这就需要他对球做功。如果球在 400N 的作用力下移动了 0.75m(从他头后到他发球的那个点)，棒球重 145g，问球飞出时的速度是多少？

解答：

在这里我们可以利用功能原理进行计算：

$$W = \Delta KE = KE_f - KE_i$$

$$\begin{aligned}
 F\Delta x &= 1/2mv_f^2 - 1/2mv_i^2 \\
 400\text{N}(0.75\text{m}) &= 1/2(0.145\text{kg})(v_f)^2 - 0 \\
 300\text{J} &= 0.0725\text{kg}(v_f)^2 \\
 4137.93\text{m}^2/\text{s}^2 &= v_f^2 \\
 64.33\text{m/s} &= v_f
 \end{aligned}$$

功能原理为后面将要介绍的定理打下了基础，每一个定理在游戏编程中都起着特殊作用。下面的函数即是利用功能原理计算末速度：

```

float calculateWorkEnergy(float force, float mass,
                           float displacement, float velocityInitial)
{
    float work, vFinal;
    work = calculateWork(force, displacement);
    vFinal = work / ((mass/2) - calculateKE(mass, velocityInitial));
    velocityFinal = sqrt(Final);
    return vFinal;
}

```

在这个函数中有相当多的计算。我们首先利用力和位移计算对物体做的功，然后把功直接代入含有末速度表达式的方程，把功移到等式右边，随后，就可以利用开方计算末速度了。

在游戏编程中，计算末速度比计算加速度更有用。所以你会发现这个定理用处非常广泛。但要记住对于一个物理模型，计算的方法有很多，我们可以很容易的利用牛顿定律和五个运动方程，但这样的话，则需要更多的步骤。作为一个编程人员，我们要根据情况运用最优化的方法进行计算，而有时候功能原理是最佳的方法。

自我测试

1. 一艘拖船以 5500N 的力将一条船拖过海港，假如拖船拖了 5km，那么它对轮船所做的功是多少？
2. 假如游戏中智能体用 30N@ -60°的力拉着马车走了 5m，那么它做了多少功？
3. 第 2 题中马车重 50kg，从静止开始拖动马车 5m 后，马车的速度是多少？
4. 一辆 3000kg 的汽车，以 55mi/hr 的速度运动，它的动能是多少？
5. 一发重 2g 速度为 300m/s 的子弹动能是多少？
6. 若第 5 题中枪膛长 50cm，那么子弹在枪膛中的平均推力是多少？

12.2 势能和能量守恒定律

我们已经定义了功和动能，在我们运用另一种运动形式的公式之前，再介绍一种能量形式——重力势能(GPE)。重力势能是根据物体距离地面的高度来衡量的能量。比如，我们举

起一本书，就使书具有了重力势能。但是书并没有动，你怎么知道它具有了能量呢？没错，是由于重力的原因。只有当你松手时势能才会释放。

但是我们怎样计算书的能量呢？我们用物体的质量和它距离地面的高度来衡量。实际上就是物体所受重力和它的高度的乘积。

重力势能

$$\text{GPE} = mgy$$

其中 m 为质量， g 为重力加速度， y 为高度

注意

还有其他形式的势能，但在游戏编程中并不很重要，可以忽略不计。以后我们仅对重力势能进行讨论，在后面用 PE 代表重力势能。

为了与前面章节保持一致，我们仍用公米制，即如果质量单位为 kg，加速度单位使用 m/s^2 ，高度单位用米，那么势能的单位为焦耳(J)，和动能一样。同时我们也要注意势能也和动能一样，是标量。其实很多能量都是没有方向的，也就是说我们可以忽略 g 的方向，直接使用 9.8m/s^2 。

【例 12-6】 重力势能

我们将一本 1.5 磅的书举到离地面 2m 位置，那么它的重力势能是多少？

解答：

1. 由于定义中是基于质量而不是重量。所以我们首先计算质量：

$$1.5\text{lbs} \left(\frac{1\text{N}}{0.2248\text{lbs}} \right) = 6.6726\text{N}$$

$$w = mg$$

$$-6.6726\text{N} = m(-9.8\text{m/s}^2)$$

$$m = 0.6809\text{kg}$$

2. 利用定义计算：

$$\text{PE} = mgy = 0.6809\text{kg}(9.8\text{m/s}^2)(2\text{m}) = 13.35\text{J}$$

势能的计算是十分简单的。在还没有加入重力计算之前，要知道该怎样将这一势能常量用于游戏编程中。在大多数游戏中都需要满足如下定义(使用公米制)：

```
#define GRAVITY 9.81
```

我们可以利用上面定义的这个常量计算。下面的函数以焦耳为单位输出重力势能：

```
float calculatePotentialEnergy(float mass, float height)
```

```
{
```

```
    float PE;
```

```
    PE = mass*GRAVITY*height;
```

```
    return PE;
```

```
}
```

我们已经介绍了动能和势能，现在利用它们来为一些运动形式建模。你想知道过山车的原理是什么吗？想象一下上次你乘坐时的情景。

当翻越了第一座山后，过山车是利用了其他动力还是使用了制动工具？如果是老式过山车那它就没有任何动力。它只是利用了机械能守恒定律。这一定律告诉我们能量不能被创造也不能消灭，它只是形式的转化，在这里就是我们所讨论的动能和势能。

能量守恒定律

$$KE_i + PE_i = KE_f + PE_f$$

或

$$1/2mv_i^2 + mgy_i = 1/2mv_f^2 + mgv_f$$

想象一下当过山车到达第一座山时，它所处的位置非常高，也就是说它具有很大的势能，但此时移动的很慢，即具有很小的动能。当它从第一个高坡滑下，随着高度的减小，过山车的速度逐渐变大，势能转化为动能。随着时间的增加，过山车到达山的底端，几乎所有的势能都转化为动能，此时过山车具有最大的速度。随着过山车爬上第二个小山，势能又转化为动能。这是一个不间断的循环，所以总的能量保持不变，如图 12-2 所示。

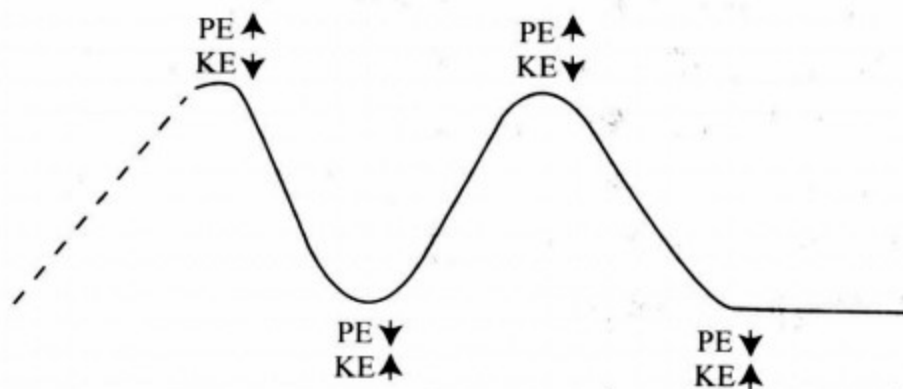


图 12-2 传统的过山车模型

【例 12-7】机械能守恒定律

假设你为游戏“Indiana Jones”编程。当 Indy 跳进一辆迷你卡车在一座小山上行驶时，卡车开始位于 100m 的高处，他和车的总重量为 300 磅，问当他到达第一座山的底端时速度是多少？

解答：

1. 注意我们需要的是质量而不是重量，所以首先计算质量：

$$300\text{lbs} \left(\frac{1\text{N}}{0.2248\text{lbs}} \right) = 1334.52\text{N}$$

$$w = mg$$

$$-1334.52\text{N} = m(-9.8\text{m/s}^2)$$

$$m = 136.18\text{kg}$$

2. 现在可以把这些量代入机械能守恒定律计算 v_f ：

$$\begin{aligned}
1/2mv_i^2 + mgy_i &= 1/2mv_f^2 + mgy_f \\
1/2(136.18\text{kg})(0)^2 + (136.18\text{kg})(9.8\text{m/s}^2)(100\text{m}) \\
&= 1/2(136.18\text{kg})(v_f)^2 + (136.18\text{kg})(9.8\text{m/s}^2)(0\text{m}) \\
0 + (136.18\text{kg})(9.8\text{m/s}^2)(100\text{m}) &= 1/2(136.18\text{kg})(v_f)^2 + 0 \\
(136.18\text{kg})(9.8\text{m/s}^2)(100\text{m}) &= 1/2(136.18\text{kg})(v_f)^2 \\
133451.96\text{J} &= 68.09\text{kg}(v_f)^2 \\
1960\text{m}^2/\text{s}^2 &= v_f^2 \\
v_f &= 44.27\text{m/s}
\end{aligned}$$

注意

机械能守恒定律适用于过山车运动中的任意两点，而不只是最高点和最低点。

另外要注意的是在等式两边均有 m ，在等式两边均除以 m ，可以消去 m ，也就是说这一定律与质量无关。

机械能守恒定律并不局限于过山车游戏，它适用于任何没有外力干预的运动形式。过山车是一个外力不起主要作用的很好的例子。这里之所以说外力不起主要作用，是由于如果我们要使运动看起来更接近实际，就必须考虑一些如摩擦力或空气阻力的因素。在现实生活中，一些能量由于摩擦力或空气阻力转换成热能或声波能而损失掉，但如果要想精确的计算这些能量代价比较大。这里我们用一种简单的方法来处理这一运动模式。在等式的右边增加一项专门代表热能和声波能，用以减小动能和势能，使其更加接近实际情况。随着时间的变化，汽车将由于能量的缓慢消耗而逐渐变慢。

机械能守恒定律(改进型)

$$KE_i + PE_i = KE_f + PE_f + E_0$$

或

$$1/2mv_i^2 + mgy_i = 1/2mv_f^2 + mgy_f + E_0$$

其中 E_0 代表其他能量(如热能或声波)。

当我们利用改进型公式时，要注意热能和声波能都是相对于两种主要能量较小的能量。这是一个增加游戏真实性的简单(快速)的方法。

【例 12-8】 机械能守恒定律

现在我们回到前面的“Indiana Jones”游戏的例子，Indy 跳进一辆迷你卡车在一座小山上行驶。这一次我们要使实验更加接近真实情况。当他刚跳入时，卡车位于 100m 的高处，他和车的总重量为 300 磅，如果有大约 2000J 的能量转换成热能和声波能损失掉，那么到达山脚时的速度是多少？

解答：

1. 注意两个定义都需要质量而不是重量，所以首先将重量转换为质量：

$$300\text{lbs}\left(\frac{1\text{N}}{0.2248\text{lbs}}\right) = 1334.52\text{N}$$

$$w = mg$$

$$-1334.52\text{N} = m(-9.8\text{m/s}^2)$$

$$m = 136.18\text{kg}$$

2. 现在把这些量代入改进型的公式, 求 v_f :

$$1/2mv_i^2 + mgy_i = 1/2mv_f^2 + mgy_f + E_0$$

$$1/2(136.18\text{kg})(0)^2 + (136.18\text{kg})(9.8\text{m/s}^2)(100\text{m}) = 1/2(136.18\text{kg})(v_f)^2 + (136.18\text{kg})(9.8\text{m/s}^2)(0\text{m}) + 2000\text{J}$$

$$0 + (136.18\text{kg})(9.8\text{m/s}^2)(100\text{m}) = 1/2(136.18\text{kg})(v_f)^2 + 0 + 2000\text{J}$$

$$(136.18\text{kg})(9.8\text{m/s}^2)(100\text{m}) = 1/2(136.18\text{kg})(v_f)^2 + 2000\text{J}$$

$$133451.96\text{J} = 68.09\text{kg}(v_f)^2 + 2000\text{J}$$

$$131451.96\text{J} = 68.09\text{kg}(v_f)^2$$

$$1930.56\text{m}^2/\text{s}^2 = v_f^2$$

$$v_f = 43.94\text{m/s}$$

利用改进型机械能守恒定律进行计算时, 由于考虑了能量损失, 最后的速度比用机械能守恒定律计算的要小, 如果这一变化仍然不是很明显, 还可以增加损失能量的值。如果是在冰面上滑行, 我们又可以选择更小的能量损失。随着经验的增加, 你就会知道 E_0 该有多大了。

注意

E_0 是一个变量, 我们可以根据自己的感觉适当的定义。

此时, 我们已经有很多方法来为运动形式建模了。你可能经常会想到利用牛顿第二定律和五个运动方程, 但在某些情况下, 利用能量的方法计算会比较简单。其实, 选择哪种方法完全根据所需建模的运动形式决定。

自我测试

1. 高 300m 的悬崖上有一块重 50kg 的石头, 问它的重力势能是多少?
2. 如果第 1 题中的石头掉下来, 那么当它砸到地面时的速度是多少?
3. 一对淘气鬼从地面向天空射水球, 如果水球的初速度为 20m/s, 问水球能达到的最大高度是多少?
4. 计算第 3 题中, 当水球到达最高点的一半高度时的速率。

本章总结

本章中我们介绍了另一种为物体运动建模的方法。这一次主要利用能量和守恒定律来决定物体到底应该跑多快或爬多高。我们首先定义了功和动能, 再由此引出功能原理。随后定

义了重力势能，并将它和动能一起应用到了机械能守恒定律中。这一定律可以应用到像过山车这样的运动当中。在某些情况下，它为我们提供了另一种解决问题的方法。

能力测试

多项选择

1. 判断对错：力和位移相乘时必须保证它们是在同一方向上。
A. 对 B. 错
2. 下面哪一项是向量？
A. 功 B. 力 C. 动能 D. 势能
3. 判断对错：可以利用功能原理计算速度。
A. 对 B. 错
4. 机械能守恒定律是利用质量还是重量？
A. 质量 B. 重量
5. 一个 50-lb 的物体距离地面 5 英尺高，问此时它的重力势能是多少？
A. 339J B. 250J C. 1112J D. 2500J

练习

练习 1：功和动能

1. 一艘拖船以 3000N 的力将一条船拖过海港，假如拖船拖了 7km，那么它对轮船所做的功是多少？
2. 假如游戏中智能体用 $27\text{N}@ -40^\circ$ 的力拉着马车走了 20m，那么它做了多少功？
3. 第 2 题中马车重 55kg，从静止开始拖动马车 20m 后，马车的速度是多少？
4. 一辆 1700kg 的汽车，以 65mi/hr 的速度运动，它的动能是多少？
5. 一发重 1.35g 速度为 270m/s 的子弹动能是多少？
6. 若第 5 题中枪膛长 35cm，那么子弹在枪膛中的平均推力是多少？

练习 2：势能和守恒定律

1. 高 300m 的悬崖上有一块重 50kg 的石头，问它的重力势能是多少？
2. 如果第 1 题中的石头掉下来，那么当它砸到地面时的速度是多少？
3. 一对淘气鬼从地面向天空射水球，如果水球的初速度为 21m/s，问水球能达到的最大高度是多少？
4. 计算第 3 题中，当水球到达最高点的一半高度时的速率。

程序设计

程序设计 1：编写程序求物体的重量、速度或高度

1. 编写一个简单的程序根据物体的重量和速度，输出它的动能。

2. 编写一个简单的程序根据物体的重量和高度，输出它的重力势能。
3. 编写一个简单的程序根据物体的重量和高度，每隔一米计算一次物体的速度，直到物体落到地面。

程序设计 2：数学方程、功的定理和机械能守恒定律

扩展数学方程数据库，加入功能原理和机械能守恒定律

讨论问题

1. 在计算功的时候，为什么需要将力和位移都转换到直角坐标系中？
2. 为什么在计算重力势能时 9.8m/s^2 不再是负值？
3. 在哪种游戏场景中(除过山车外)需要使用机械能守恒定律？

新学网
PDG

第 13 章 动量和碰撞

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 计算当物体碰撞到固定的水平或竖直平面时的向量反射
- 计算当物体碰撞到固定的非水平或竖直平面时的向量反射
- 定义和计算动量
- 定义和计算冲量
- 根据动能定理计算末速度
- 计算碰撞后两物体的速度

介绍

我们已经讨论了在虚拟世界中单独的物体如何运动的问题。下一步，我们来研究有关碰撞的问题。两台球相撞或一个玩家跳起踢中对手时，会发生什么情况？前面章节中我们只讨论了孤立物体的运动形式。现在，我们开始讨论物体之间是如何相互作用的。

首先，我们研究当其中一个物体静止时的碰撞情况，然后再研究两运动物体之间的碰撞。

13.1 有关静止物体的碰撞

和静止物体的碰撞是最简单的碰撞形式，比如台球碰到边栏。边栏不动，我们只要考虑球的运动就可以了。这种情况下可以用向量反射来研究物体的运动，这种运动存在着一种对称性，球射入的角度必然等于它射出的角度，即入射角等于反射角，如图 13-1 所示。

如果壁面是在水平方向或是竖直方向，那么就可以十分容易编出程序解决问题。同样，如果静止物体是水平或竖直平行长方体，那么也是十分容易解决的。现在我们回到台球撞边栏的例子。利用第 10 章中的知识，把球的速度写成向量的分量形式。如果边栏是竖直方向的，那么在碰撞时只要翻转其水平方向的分量就可以了，如图 13-2 所示。

与此相似，如果边栏是水平的，那么我们就在其碰撞的同时翻转竖直方向的分量。

向量轴平行反射

如果边是竖直方向的，则 $v_f = [-v_{ix} \ v_{iy}]$ ；

如果边是水平方向的，则 $v_f = [v_{ix} \ -v_{iy}]$ ；

入射向量： $v_i = [v_{ix} \ v_{iy}]$ 。

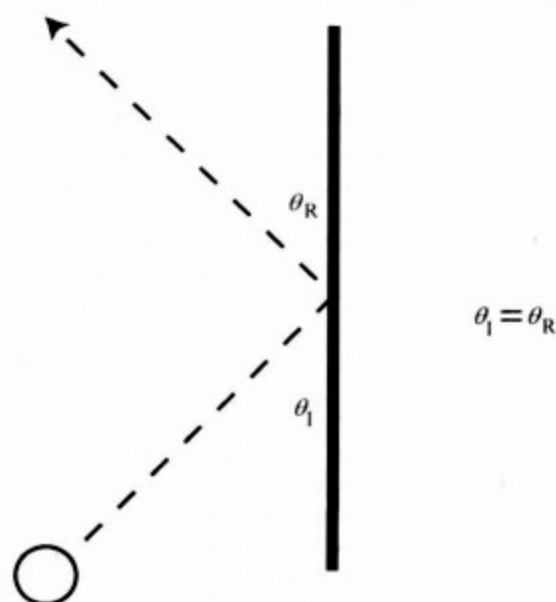


图 13-1 向量反射

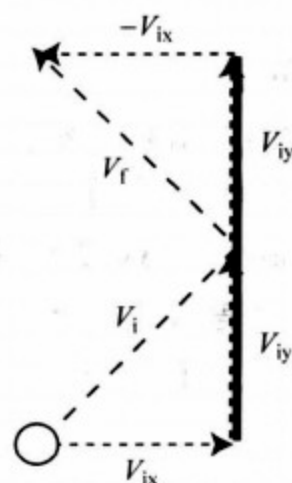


图 13-2 对于竖直墙面所产生的向量反射

【例 13-1】 向量的轴平行反射和非轴平行反射

如果在弹球游戏编程中，你想为球与球杆的碰撞建模。假设球的速度为 $[40\ 75]$ ，球碰撞到球杆之后的速度是多少？

解答：

在这个游戏中球在屏幕上来回滚动，当球碰到竖直方向的球杆时，竖直分量不变，水平分量翻转，所以末速度是 $[-40\ 75]$ 。

我们已经讨论了轴平行向量反射，现在开始研究非轴平行反射。当静止的物体边缘不与 x 、 y 轴平行时，会出现什么样的情况呢？当然，入射角必须等于反射角，但是这比仅翻转一个分量的情况要复杂的多。我们先讨论二维空间，再将其扩展到三维空间中。

在这种情况下，如果入射向量是 v_i ，静止的边界是 B ，那么反射向量就是 v_r ，如图 13-3 所示。

我们可以把这种情况编成函数：

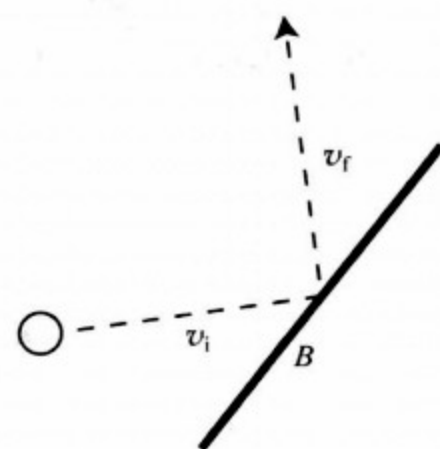


图 13-3 非轴向量反射

```
vector2D axisAlignedCollision(vector2D vect, char b)
```

```
{
```

```
    vector2D temp = vect;
```

```
    if(b == 'v')
```

```
        temp.x = temp.x * -1;
```

```
    else if(b == 'h')
```

```
        temp.y = temp.y * -1;
```

```
    return temp;
```

```
}
```


这种特殊的函数用来处理边界与轴平行的碰撞。通常，我们不需要输入任何信息，因为有关向量反射的处理过程是写入到碰撞的程序中的。注意，碰撞检测只是工作的一半，作为游戏的设计者和开发者，我们还需要考虑程序的输出。

我们首先要找到静止物体的边界垂线 N 。在第 1 章中我们曾经介绍，互相垂直的两条直线的斜率互为负倒数。也就是说，如果 B 的斜率是 $\Delta y/\Delta x$ ，那么 N 的斜率就是 $-\Delta x/\Delta y$ ，用向量形式表示就是如果 $B = [\Delta x \ \Delta y]$ ，那么 $N = [\Delta y \ -\Delta x]$ 。我们把 N 单位化，计算分量，单位化后的向量写作 N' ，如图 13-4 所示。

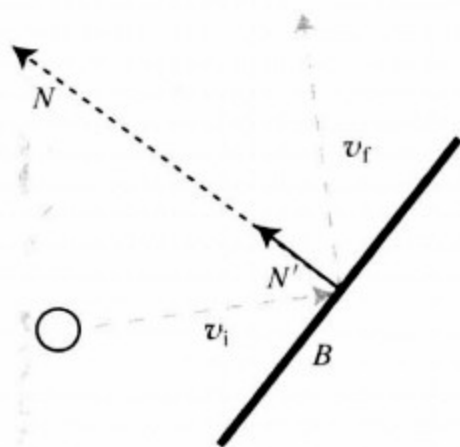


图 13-4 基于边界线的法线

下一步是计算 $-v_i$ 在 N' 上的射影 P 。

注意

不可以直接求 P ，我们需要首先翻转 v_i ，再进行点乘。

我们用第 4 章中点乘的方法来求射影 $P = -v_i \cdot N'$ 。

为了使 P 和垂线的方向相同，我们用 P 的模乘以 N' ： $P = (-v_i \cdot N') * N'$ ，如图 13-5 所示。

为了计算 v_f ，我们需要一些几何知识。如图 13-6 所示，我们计算新向量 V ，将 v_i 和 P 首尾相接，做加法 $V = v_i + P$ 。

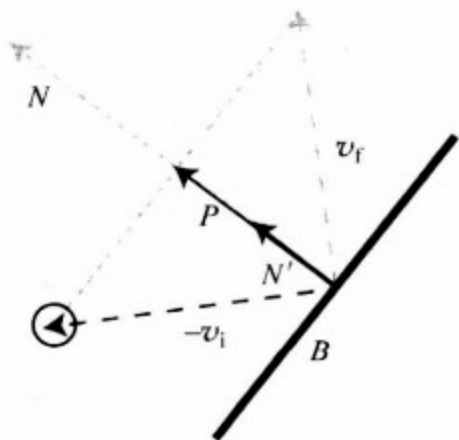
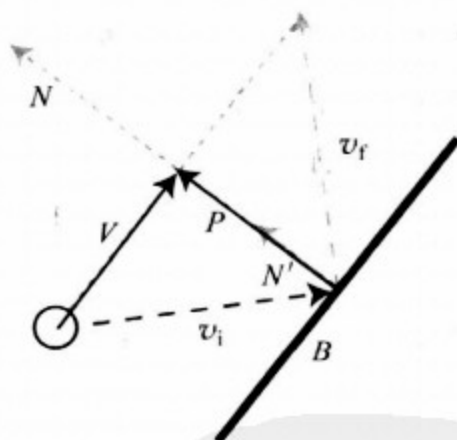


图 13-5 向量增加

图 13-6 新增加的向量 V

在图 13-6 中，我们发现 $P + V = v_f$ ，现在带入 V ：如果 $V = v_i + P$ ，而 $P + V = v_f$ ，那么 $v_f = P + (v_i + P)$ 。最后我们得到所需的反射线 $v_f = 2 \cdot P + v_i$ 。

向量的非轴平行反射

$$v_f = 2 \cdot P + v_i$$

其中 v_i 为初速度， P 为 $-v_i$ 基于分界线法线的发射向量。

【例 13-2】二维向量的非轴平行反射

如果在弹球游戏编程中又增加了一些物体，且你想为球与这些障碍物的碰撞问题建模。

假设在下一帧球就会与过点(50, 25)和(200, 250)的直线上的边界碰撞, 如果球的入射速度是[50 10], 那么反射速度是多少?

解答:

1. 计算表示边界的向量:

$$\begin{aligned} B &= [\Delta x \quad \Delta y] \\ &= [200 \quad 250] - [50 \quad 25] \\ &= [150 \quad 225] \end{aligned}$$

2. 计算边界的垂直向量(法向量):

$$\begin{aligned} N &= [\Delta y - \Delta x] \\ &= [225 - 150] \end{aligned}$$

3. 将 N 单位化:

$$\|N\| = \sqrt{225^2 + (-150)^2} = \sqrt{50625 + 22500} = \sqrt{73125} \approx 270.416$$

$$N = [225 \quad -150]$$

$$N' = \hat{N} \left[\frac{225}{270.416} \quad \frac{-150}{270.416} \right] \approx [0.832 \quad -0.555]$$

4. 用 P 的模乘以 N' 来计算向量 P :

$$\begin{aligned} P &= (-v_i \cdot N') \cdot N' \\ &= ([-50 \quad -10] \cdot [0.832 \quad -0.555]) \cdot N' \\ &= (-36.05) \cdot N' \\ &= (-36.05) \cdot [0.832 \quad -0.555] \\ &\approx [-30 \quad 20] \end{aligned}$$

5. 最后一步计算反射向量:

$$\begin{aligned} v_f &= 2 \cdot P + v_i \\ &= 2 \cdot [-30 \quad 20] + [50 \quad 10] \\ &= [-60 \quad 40] + [50 \quad 10] \\ &= [-10 \quad 50] \end{aligned}$$

下面我们来看一看怎样编写一个计算二维空间中碰撞的函数:

```
vector2D nonAxisAlignedCollision(vector2D a, vector2D b)
{
    vector2D temp, temp2, temp3, length, reflection;
    float projection;

    temp = normalizeVector(a);

    // 求反向量
    temp2 = scalarMultiply(-1, b);

    // 利用点乘求投影的长度
    projection = dotProduct(temp2, temp);
```

```

length = scalarMultiply(projection,temp);

//反射值是投影长度的二倍
reflection = scalarMultiply(2,length);

//将reflection反转后再求和
temp3 = sumVectors(scalarMultiply(-1,reflection),temp2);

//返回结果
return temp3;
}

```

这个函数使用的方法和前面的例子是一样的。首先对垂直于边界的向量进行单位化。然后翻转碰撞物体的向量和计算点乘来求它的模。把这个模和垂直单位向量相乘，再将得到的向量变成原来的二倍，最后用入射向量加上翻转后的物体向量，这样我们就完成了一次非轴平行向量的碰撞。

你也许想知道怎样将其扩展到三维空间中，如果我们知道了碰撞所在的平面，就可以用同样的方法进行计算。唯一不同的就是三维空间中碰撞是在平面上发生的。两个向量决定一个平面，所以，我们可以利用两向量的叉乘找到法向量。下面我们来看一个三维空间的例子。

【例 13-3】 三维空间中的非轴平行向量

假设你要编一个 3D 的弹球游戏，假设在下一帧球就与 $[200 \ 250 \ -300]$ $[50 \ 200 \ -25]$ 决定的平面进行碰撞，那么如果球的入射速度是 $[100 \ -50 \ -50]$ ，那么反射速度是多少？

解答：

1. 用第 4 章的方法计算面的垂直向量：

$$\begin{aligned}
 N &= [200 \ 250 \ -300] \times [50 \ 200 \ -25] \\
 &= [(250 \cdot -25) - (-300 \cdot 200)] \mathbf{i} - (200 \cdot -25) \mathbf{j} + (200 \cdot 200 - (250 \cdot 50)) \mathbf{k} \\
 &= [(-6250 + 60000) \mathbf{i} - (5000) \mathbf{j} + (40000 - 12500) \mathbf{k}] \\
 &= [53750 \mathbf{i} - 10000 \mathbf{j} + 27500 \mathbf{k}]
 \end{aligned}$$

$$\|N\| = \sqrt{53750^2 + (-10000)^2 + 27500^2} = \sqrt{3745312500} \approx 61199$$

$$N' = \hat{N} \left[\frac{53750}{61199} \quad \frac{-10000}{61199} \quad \frac{27500}{61199} \right] \approx [0.8783 \quad -0.1634 \quad 0.4494]$$

2. 用计算 N' 上射影的方法计算 P ：

$$\begin{aligned}
 P &= (-v_i \cdot N') \cdot N' \\
 &= ([-100 \ 50 \ 50] \cdot [0.8783 \ -0.1634 \ 0.4494]) \cdot N' \\
 &= (-73.53) \cdot N' \\
 &= (-73.53) \cdot [0.8783 \ -0.1634 \ 0.4494] \\
 &\approx [-64.5814 \ 12.0148 \ -33.0444]
 \end{aligned}$$

3. 最后我们计算反射向量:

$$\begin{aligned}
 v_f &= 2 \cdot P + v_i \\
 &= 2 \cdot [-64.5814 \quad 12.0148 \quad -33.0444] + [100 \quad -50 \quad -50] \\
 &= [-129.1628 \quad 24.0296 \quad -66.0888] + [100 \quad -50 \quad -50] \\
 &= [-29.1628 \quad -25.9704 \quad -116.0888]
 \end{aligned}$$

这和计算二维空间中非轴平行向量反射的方法很相似。唯一不同的是我们使用三维向量进行计算:

```

vector3D nonAxisAlignColl(vector3D a, vector3D b, vector3D velocity)
{
    vector3D temp, normal, velocityTemp, velocityFinal, length, reflection;

    velocityTemp = scalarMultiply(-1, velocity);

    float projection;
    temp = crossProduct(a, b);

    // 计算面的垂直向量
    normal = normalizeVector(temp);

    // 计算投影
    projection = dotProduct(velocityTemp, normal);
    // 计算投影长度
    length = scalarMultiply(projection, normal);

    // 得出最终向量
    reflection = scalarMultiply(2, length);
    velocityFinal = sumVectors(reflection, velocity);
    return velocityFinal;
}

```

如果你需要知道函数 dotProduct、normalizeVector 和 scalarMultiply 的定义方法, 可以参看前面的章节或查看示例代码。

我们已经讨论了与静止物体发生碰撞的情况。在这一章节的其余部分, 我们将讨论运动物体之间发生的碰撞。

自我测试

1. 假设我们以俯视的角度编写一个 2D 游戏, 桌台的四边分别平行于 x 和 y 轴, 如果一个球以 $[-30 \quad 50]$ 的速度碰到屏幕上沿, 那么碰撞后的速度是多少?
2. 假如我们以俯视的角度编写一个 2D 游戏, 桌台的一边过点 $(40, 150)$ 和 $(240, 300)$, 如果一个球以 $[80 \quad 20]$ 的速度碰到这个边, 那么碰撞后的速度是多少?

3. 假如我们编写一个 3D 游戏, 向量 $[30 \ 100 \ -50]$ 和 $[200 \ -50 \ -25]$ 定义了一个面, 那么如果一个球以 $[20 \ -10 \ -50]$ 的速度碰到这个面, 那么碰撞后的速度是多少?

13.2 向量和冲量

要研究两个移动物体的碰撞问题, 首先要定义一个量——动量。动量(p)是质量乘以速度。也就是说物体越重, 动能越大。这就是拖拉机很难突然刹车的原因。另外, 速度越快, 动量越大, 所以速度为 100mph 的跑车也不容易停下来。

动量

$$p = mv$$

其中 m 为质量, v 为速度。

注意

由于动量是基于变化的速度的, 所以我们需要讨论瞬时动量和一段时间中动量的改变。

我们可以快速的找到它的单位, 如果用米制单位, 质量单位为千克, 速度单位为米/秒, 那么动量的单位就是 $\text{kg} \cdot \text{m/s}$ 。注意, 我们经常把它和牛顿的单位搞混, 要记住 $1\text{N} = 1\text{kg} \cdot \text{m/s}^2$ 而不是 $\text{kg} \cdot \text{m/s}$ 。

如果我们进一步研究这个定义就会发现, 动量是由一个标量(时间)乘以一个矢量(速度), 所以它是一个矢量, 它的方向和速度方向一致。在一维空间中我们可以用正负数来表示它的方向。而在二维和三维空间中我们用含有分量的向量对这个量进行编程。我们分别处理它的分量, 就像处理一维空间中的情况一样。

【例 13-4】 计算一维动量

一辆卡车重 5500lbs, 以速度 60mph 行驶在公路上, 那么它的动量是多少?

解答:

1. 注意卡车的重量是以磅为单位, 而动量需要以千克为单位的质量。所以我们需要做一下转化:

$$5500\text{lbs} \left(\frac{1\text{N}}{0.2248\text{lbs}} \right) = 24\,466.19\text{N}$$

$$w = mg$$

$$24\,466.19\text{N} = m(9.8\text{m/s}^2)$$

$$m = 2496.55\text{kg}$$

2. 速度也需要转化为 m/s 的形式:

$$\left(\frac{60\text{mi}}{1\text{hr}} \right) \left(\frac{1609\text{m}}{1\text{mi}} \right) \left(\frac{1\text{hr}}{3600\text{s}} \right) = 26.82\text{m/s}$$

3. 现在我们把这两个值代入到定义中:

$$p = mv$$

$$p = 2496.55\text{kg}(26.82\text{m/s})$$

$$p = 66\,957.47\text{kg} \cdot \text{m/s}$$

下面是计算动量的函数：

```
float momentum(float velocity, float mass)
{
    float momentum;
    momentum = mass*velocity;
    return momentum;
}
```

这是一个一维空间的例子，和速度一样，正数表示向前的动量。下面来看三维空间中的情况。

【例 13-5】 计算三维动量

如果一艘飞船的质量为 4500kg，当时的速度是 $\begin{bmatrix} 30 \\ 50 \\ -20 \end{bmatrix}$ ，那么它当时的动量是多少？

解答：

这次我们用 3D 向量表示速度，所以得到的动量也是 3D 向量的形式：

$$p = mv$$

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = m \begin{bmatrix} 30 \\ 50 \\ -20 \end{bmatrix} = 4500 \begin{bmatrix} 30 \\ 50 \\ -20 \end{bmatrix} = \begin{bmatrix} 135\,000 \\ 225\,000 \\ -90\,000 \end{bmatrix}$$

用程序计算 3D 动量也是非常容易的。下面的函数就是用标量乘以向量的方法计算动量：

```
vector3D momentum3D(vector3D velocity, float mass)
{
    vector3D temp;
    temp = scalarMultiply(mass, velocity);
    return temp;
}
```

我们已经介绍了动量，下面需要介绍另一个量——冲量。首先复习一下第 11 章中的牛顿定律：

$$F = ma$$

代入加速度：

$$F = m(v_f - v_i)/t$$

$$F = (mv_f - mv_i)/t$$

你认出上面的分数了吗？它就是动量的改变量：

$$F = (\Delta p)/t$$

无论你相信与否，这就是牛顿第二定律的最初的形式。最开始，它的形式就是说力等于动量的变化率。但最后大多数版本都把它写成 $F = ma$ 的形式。

最后我们将等号两边乘以时间：

$$Ft = (\Delta p)$$

等式的左边就是冲量。

冲量

$$\text{冲量} = Ft$$

其中 F 为合力， t 是一个很小的时间量。

冲量是力在一瞬间传递的。比如用高尔夫球棒击球，就是把冲量传给了球。球棒接触球的时间只有几分之一秒，但是球却飞走了，这就是冲量的起源：

$$Ft = (\Delta p)$$

这告诉我们冲量改变了物体的动量。换句话说，一个力快速改变了动量。这就是说物体的质量不变，但速度却变化了。我们把这个公式叫做动量定理。

动量定理

$$\text{冲量} = Ft = (\Delta p)$$

其中 F 为合力， t 是一个很小的时间量， p 为动量增量。

在概念上，动量定理和牛顿第二定律是一致的。它们都是描述物体经过力的作用导致速度的改变。但是动量更加适用于编程。现在再看一下这个高尔夫球的例子。我们可以用第 11 章的方法通过球杆击球的力计算球的加速度，然后利用运动方程计算出物体的速度，而我们同样可以更快地利用动量定理计算速度。如果我们知道球的质量和初速度（此时为零），就可以计算出球杆击球的结果，这个末速度就又变成了随后运动的初速度。动量定理的计算过程和牛顿第二定律是一样的，只是步骤较少而已。

由于运动发生的非常之快，球棒接触球只有几分之一秒的时间，所以球的运动可以看作两个片段——快速与球棒碰撞以及随后的运动过程。

【例 13-6】 动量定理

假设你正在编写一个高尔夫球的游戏，球重 45g(0.045kg)。如果运动员用球棒传给球

的冲量是 $\begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix}$ (单位是 $\text{kg} \cdot \text{m/s}$)，那么球的速度是多少？

解答：

我们直接运用冲量定理，代入已知量：

$$\text{冲量} = \Delta p = mv_f - mv_i$$

$$\begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix} = 0.045 \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} - 0.045 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

(球位于 T 形架上, 初速度为零)

$$\begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix} = 0.045 \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \frac{1}{0.045} \begin{bmatrix} 3 \\ 2 \\ -4 \end{bmatrix}$$

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} 66.67 \\ 44.44 \\ -88.89 \end{bmatrix}$$

下面是动量定理的程序:

```
vector3D impulse3D (vector3D final, vector3D initial, float mass)
{
    vector3D impulse, momentumFinal, momentumInitial;
    momentumFinal = momentum3D(final, mass);
    momentumInitial = momentum3D(initial, mass);
    impulse = subtractVectors(momentumFinal, momentumInitial);
    return impulse;
}
```

注意

这里我们计算出的速度将变成后面运动的初速度。下一节我们将介绍当两个运动物体碰撞时动量定理的运用。但要记住, 动量是处理碰撞的根本之所在。

自我测试

1. 一块重 100lbs 的石头以 15m/s 速度水平滚动, 它的动量是多少?
2. 在 3D 游戏中, 重 2000kg 的卡车速度是 $\begin{bmatrix} 50 \\ 25 \\ -10 \end{bmatrix}$, 问此时它的动量是多少?
3. 在一个格斗游戏中, 一个重 250lbs 且保持不动的玩家被另一个玩家以 5000kg * m/s 的冲量踢打。问这个玩家将要以什么样的速度飞出?
4. 在第 3 题中, 施力者也受到了同样的动量作用, 假设它是以 5m/s 的速度踢向他的对手, 问他的末速度是多少?

5. 假设你正在做一个 3D 的格斗游戏。玩家给了一个物体 $\begin{bmatrix} 2500 \\ 6000 \\ -1000 \end{bmatrix}$ 的冲量, 物体重 250lbs,

那么物体的末速度将是多少?

13.3 碰撞建模

上一节我们讨论了有关将动量定理应用在一个物体上的问题。现在来看一下两个运动的物体在碰撞时将发生怎样的情况。再来看一下动量定理:

$$Ft = (\Delta p)$$

两边同除以时间:

$$F = (\Delta p)/t$$

牛顿第三定律告诉我们, 在每次碰撞中, 都会有两个等大反向的力。在这里一个物体将力作用于另一个物体时, 它也受到另一个物体等大的反向力的作用:

$$F_1 = -F_2$$

现在我们带入动量定理:

$$(\Delta p_1)/t = -(\Delta p_2)/t,$$

或

$$\Delta p_1 = -\Delta p_2$$

再将动量定义带入到等式两边:

$$m_1 v_{1f} - m_1 v_{1i} = -(m_2 v_{2f} - m_2 v_{2i})$$

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

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f}$$

最后一行被称作动量定理的变形。它告诉我们在碰撞时, 总的动量不变。动量只是从一个物体传递到另一个物体。

动量定理的变形

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f}$$

下标 1 表示一个物体, 下标 2 表示第二个物体。

这条定理只适用于两物体单独碰撞。如果受到外力作用, 动量定理就不成立了。但是由于碰撞时速度很快, 所以很少有外力介入。

注意

动量是一个矢量, 所以我们可以用同样的方法处理一维、二维和三维的情况。

【例 13-7】动量定理的变形

在一个三维的弹球游戏中, 一个重 0.5kg 的母球以 $[50 \ 10 \ -30]$ 的速度碰到另一个重 0.45kg 的静止的球, 如果碰撞后母球的速度为零, 那么第二个球的末速度是多少?

解答:

直接运用动量定理的变形式, 直接代入已知量:

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f}$$

$$0.5\text{kg} \begin{bmatrix} 50 \\ 10 \\ -30 \end{bmatrix} + 0.45\text{kg} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 0.5\text{kg} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + 0.45\text{kg} \begin{bmatrix} v_{2fx} \\ v_{2fy} \\ v_{2fz} \end{bmatrix}$$

$$0.5\text{kg} \begin{bmatrix} 50 \\ 10 \\ -30 \end{bmatrix} = 0.45\text{kg} \begin{bmatrix} v_{2fx} \\ v_{2fy} \\ v_{2fz} \end{bmatrix}$$

$$\begin{bmatrix} 25 \\ 5 \\ -15 \end{bmatrix} = 0.45\text{kg} \begin{bmatrix} v_{2fx} \\ v_{2fy} \\ v_{2fz} \end{bmatrix}$$

$$\begin{bmatrix} 55.56 \\ 11.11 \\ -33.33 \end{bmatrix} = \begin{bmatrix} v_{2fx} \\ v_{2fy} \\ v_{2fz} \end{bmatrix}$$

在这里可以看到母球通过碰撞将动量全部传递给第二个球。

下面把它编成程序函数：

```
vector3D conserveMomentum(vector3D object1Start, vector3D
    object1Stop, vector3D object2Start, float mass1, float mass2)
{
    vector3D temp, temp2, sum, result;

    // 首先计算等式左边
    temp = scalarMultiply(mass1, object1Start);
    temp2 = scalarMultiply(mass2, object2Start);
    sum = sumVectors(temp, temp2);

    // 用第二个物体的质量除以等式左边的值得到这一向量
    result = scalarDivide(mass2, sum);
    return result;
}
```

我们可以利用这个函数计算碰撞后物体的速度。还可以对具有不同的初态和末态的物体进行计算。

这仅仅是一个简单的例子，在真实的情况下，两个物体在碰撞之前可能都在运动。

【例 13-8】完全非弹性碰撞

假设以俯视的角度为一个橄榄球游戏编程。一个重 180lbs 的玩家带着球从球场的一条边线奔跑，速度为 $[0 \ 30]$ 。此时他的一个对手，重 220lbs，以 $[25 \ 5]$ 的速度追逐。当两人撞到一起时的速度是多少？

解答:

1. 两个人的重量单位都是磅, 但是这个定理是基于千克的, 所以要用第 11 章的方法, 将它们转换成质量:

$$180\text{lbs}\left(\frac{1\text{N}}{0.2248\text{lbs}}\right) = 800.71\text{N}$$

$$m_1 = \frac{w}{g} = \frac{800.71\text{N}}{9.8\text{m/s}^2} = 81.71\text{kg}$$

$$220\text{lbs}\left(\frac{1\text{N}}{0.2248\text{lbs}}\right) = 978.65\text{N}$$

$$m_2 = \frac{w}{g} = \frac{978.65\text{N}}{9.8\text{m/s}^2} = 99.86\text{kg}$$

2. 现在我们知道了质量的大小, 把所有的已知量代入定理:

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f}$$

$$81.71\text{kg}\begin{bmatrix} 0 \\ 30 \end{bmatrix} + 99.86\text{kg}\begin{bmatrix} 25 \\ 5 \end{bmatrix} = 81.71\text{kg}\begin{bmatrix} v_{1fx} \\ v_{1fy} \end{bmatrix} + 99.86\text{kg}\begin{bmatrix} v_{2fx} \\ v_{2fy} \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 2451.3 \end{bmatrix} + \begin{bmatrix} 2496.5 \\ 499.3 \end{bmatrix} = 81.71\text{kg}\begin{bmatrix} v_{1fx} \\ v_{1fy} \end{bmatrix} + 99.86\text{kg}\begin{bmatrix} v_{2fx} \\ v_{2fy} \end{bmatrix}$$

$$\begin{bmatrix} 2496.5 \\ 2950.6 \end{bmatrix} = 81.71\text{kg}\begin{bmatrix} v_{1fx} \\ v_{1fy} \end{bmatrix} + 99.86\text{kg}\begin{bmatrix} v_{2fx} \\ v_{2fy} \end{bmatrix}$$

3. 由于最后两人一同飞出, 所以最后两个人的速度相同, 也就是说 $v_{1fx} = v_{2fx}$ 且 $v_{1fy} = v_{2fy}$, 因此将它们称为 v_{fx} 和 v_{fy} :

$$\begin{bmatrix} 2496.5 \\ 2950.6 \end{bmatrix} = 81.71\text{kg}\begin{bmatrix} v_{fx} \\ v_{fy} \end{bmatrix} + 99.86\text{kg}\begin{bmatrix} v_{fx} \\ v_{fy} \end{bmatrix}$$

$$\begin{bmatrix} 2496.5 \\ 2950.6 \end{bmatrix} = 181.57\text{kg}\begin{bmatrix} v_{fx} \\ v_{fy} \end{bmatrix}$$

$$\begin{bmatrix} 13.75 \\ 16.25 \end{bmatrix} = \begin{bmatrix} v_{fx} \\ v_{fy} \end{bmatrix}$$

由于他们最后的速度相同, 我们才能计算它。还有另一种情况需要考虑, 即如果两个物体碰撞后速度的 v_{1f} 、 v_{2f} 不相同, 那我们就需要用另一个方程帮助计算速度。这需要用到第 1 章的联合法或代入法知识。如果我们把 v_{1f} 看作 x , 把 v_{2f} 看作 y , 那么就和解方程组的过程一样, 先利用联合法或代入法解出一个速度, 然后代入方程, 解出另一个速度。为了能够找出第二个方程, 我们再来看另一种碰撞。

每个碰撞的情况都介于弹性碰撞和完全非弹性碰撞之间。弹性碰撞是一种没有动能损失的碰撞。碰撞后两个物体就像台球一样弹开。例 13-8 即是一种非常好的弹性碰撞的例子, 两个物体粘在一起, 且速度相同。而现实情况经常介于这两种情况之间: 两个物体的碰撞有

能量损失，但两个物体的速度不相同。我们可以用还原系数(ε)来表示能量损失的大小。在弹性碰撞中 $\varepsilon = 1$ ，非弹性碰撞中 $\varepsilon = 0$ 。

线性碰撞

$$(v_{1f} - v_{2f}) = -\varepsilon(v_{1i} - v_{2i})$$

其中， $0 < \varepsilon < 1$ 。

我们进一步来看这个公式。如果把 0 代入 ε ，那么等式右边就等于 0， v_{1f} 等于 v_{2f} ；如果把 1 代入 ε ，那么方程就变成 $(v_{1f} - v_{2f}) = -(v_{1i} - v_{2i})$ 的形式。这就是多数程序员选择弹性碰撞的原因。为了保持游戏的真实性，我们会给 ε 赋值。 ε 值越靠近 0，碰撞时的能量损失越大，越靠近 1，就越像弹性碰撞，比如台球。

【例 13-9】 弹性碰撞

如果我们以俯视角度给一个游戏编程，游戏是关于两个相同的台球的弹性碰撞的问题，一个球的速度是 $[30 \ 20]$ ，另一球的速度是 $[-40 \ 10]$ ，那么弹性碰撞后它们的速度各是多少？

解答：

1. 为了计算两个末速度，需要使用两个方程。我们先把已知量代入动量定理的变形式中：

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f}$$

$$m_1 \begin{bmatrix} 30 \\ 20 \end{bmatrix} + m_2 \begin{bmatrix} -40 \\ 10 \end{bmatrix} = m_1 v_{1f} + m_2 v_{2f}$$

由于 $m_1 = m_2$ ，所以质量被抵消了：

$$\begin{bmatrix} 30 \\ 20 \end{bmatrix} + \begin{bmatrix} -40 \\ 10 \end{bmatrix} = v_{1f} + v_{2f}$$

$$\begin{bmatrix} -10 \\ 30 \end{bmatrix} = v_{1f} + v_{2f}$$

2. 用线性碰撞建立第二个方程， $\varepsilon = 1$ ：

$$(v_{1f} - v_{2f}) = -\varepsilon(v_{1i} - v_{2i})$$

$$(v_{1f} - v_{2f}) = -1(v_{1i} - v_{2i})$$

$$(v_{1f} - v_{2f}) = -v_{1i} + v_{2i}$$

$$(v_{1f} - v_{2f}) = \begin{bmatrix} -30 \\ -20 \end{bmatrix} + \begin{bmatrix} -40 \\ 10 \end{bmatrix}$$

$$(v_{1f} - v_{2f}) = \begin{bmatrix} -70 \\ -10 \end{bmatrix}$$

3. 我们利用联合法或代入法都可以解出方程，下面我们利用联合法：

$$v_{1f} - v_{2f} = \begin{bmatrix} -10 \\ 30 \end{bmatrix}$$

$$v_{1f} - v_{2f} = \begin{bmatrix} -70 \\ -10 \end{bmatrix}$$

用上面的方程减去下面的方程：

$$0 + 2v_{2f} = \begin{bmatrix} 60 \\ 40 \end{bmatrix}$$

$$v_{2f} = \begin{bmatrix} 30 \\ 20 \end{bmatrix}$$

代入上面的方程解得 v_{1f} ：

$$v_{1f} + \begin{bmatrix} 30 \\ 20 \end{bmatrix} = \begin{bmatrix} -10 \\ 30 \end{bmatrix}$$

$$v_{1f} = \begin{bmatrix} -40 \\ 10 \end{bmatrix}$$

由于是弹性碰撞，两个球的动量传递是弹性的。但是如果 $\varepsilon = 0.9$ ，那么计算后两个速度都会变小。

我们看这个计算弹性碰撞的程序。这个函数计算两物体的末速度，其中假设知道它们当前的速度，且二者质量相同，还原系数为 1：

```
#define RESTITUTION 1
void elasticCollision(vector2D object1, vector2D object2, float
    mass)
{
    vector2D object1Temp, object2Temp, sum, sum2,
    object1Velocity, object2Velocity, temp;

    // 首先计算等式左边
    object1Temp = scalarMultiply(mass, object1);
    object2Temp = scalarMultiply(mass, object2);
    sum = sumVectors(object1Temp, object2Temp);

    // 通过与还原系数相乘来设置等式右边

    // 这里仅仅是为了演示的目的

    object1Temp = scalarMultiply(RESTITUTION, object1Temp);
    object2Temp = scalarMultiply(RESTITUTION, object2Temp);

    // 为了这个目的，现在计算出等式的右边

    object1Temp = scalarMultiply(-1, object1Temp);
    sum2 = sumVectors(object1Temp, object2Temp);
```

```
//现在我们已经有了两个向量来完成计算
temp = subtractVectors(sum,sum2);
object2Velocity = scalarDivide(2,temp);

object1Velocity = subtractVectors(sum,object2Velocity);

//打印结果
cout<<"The first object's resulting velocity vector is ";
printVector(object1Velocity);
cout<<"The second object's resulting velocity vector is \n";
printVector(object2Velocity);
}
```

在这里，如果我们立即用还原系数乘以两个速度，就能得到最后的速度。它提供了一种解决问题的方法，但同样也仅限于解决这一问题。

现在我们已经在处理不同类型的碰撞方面有了坚实的基础。下面将展示如何利用这个基础进行真实的模拟。

自我测试

1. 在 3D 弹球游戏中，一个重 0.5kg 的球以 $[-40 \ 50 \ -20]$ 的速度碰撞到另一个重 0.45kg 的球，碰撞后第一个球静止，那么求第二个球的速度。
2. 假设你在为一个 3D 的赛车游戏编程。一辆重 1500kg 的汽车以 $[30 \ -20 \ -40]$ 的速度撞上另一辆重 1800kg 以速度 $[-50 \ 40 \ -20]$ 运动的汽车。如果碰撞后它们的速度相同，那么它们最后的速度是多少？
3. 假设你在为 3D 赛车游戏中汽车碰撞的部分编程，但这一次汽车并不会粘在一起运动。一辆重 1500kg 的汽车以 $[30 \ -20 \ -40]$ 的速度撞上另一辆重 1800kg 以速度 $[-50 \ 40 \ -20]$ 运动的汽车。假设 $e = 0.5$ ，那么它们最后的速度是多少？

13.4 可视化检测

在光盘中，你可以找到一个名字为 Basement Billiards 的演示程序。下面是程序员对它所做的简单描述：

Basement Billiards 是一个非常实用的 2D 环内碰撞演示程序。虽然这个程序是为 OpenGL 以及 3D 空间所写，但只要将视角进行旋转，对于 Z 轴反方向仍然适用——针对传统排列，Y 轴的反方向朝下，这使得所有的计算都能够基于 XY 平面进行。动量守恒定律可以通过完全弹性碰撞演示，比如两小球之间发生的碰撞，或者通过非弹性碰撞演示，比如球与铁栏之间发生的碰撞。完整的游戏代码放在函数 DrawGLScene 里，这个函数包含在文件 OGL_Wrapper.cpp 中。不难理解，代码的每个部分通常被放在一个单独的函数里，并且给出了详细的注释。所有有关碰撞的

计算被单独地放在“Shooting Code”里。而所有关于线性或圆弧碰撞的细节则位于“Prediction Code”中。由于这只是一个演示程序，所以并没有引入游戏的机制。比如玩家轮流发射或者一次性发射完它们口袋里所有的小球的情况在代码中没有很好的处理。

你可以按方向键来进行镜头移动，还可以按 A 或者 Z 键来对镜头进行缩小或放大。按 C 键可以固定镜头的焦距，以便从桌面中心确定球杆的当前位置。点击球杆开始瞄准，按左、右方向键也可以进行瞄准，再按上、下方向键增大或者减小作用力，按 Shift 键校准准心。点击大球杆左上角具有 English 字样的位置，以便增加子弹数量。无论任何时候，只要点击右键就会放弃瞄准和射击，而按开火箭则可以发射子弹。如果需要，可以刷新并重新开始。按 Esc 键结束游戏。F1 键可以在窗口和全屏幕之间切换。

——Kaveh Kahrizi

下面我们开始演示。双击 Pool.exe 运行程序。这时可以看到一个桌球平台，现在可以正式开始了。请将注意力集中在各类碰撞上——球与球之间以及球与缓冲器之间的碰撞。这个游戏的代码非常适合初级程序员，清单 13-1 给出了有关这个游戏中各类碰撞句柄的描述。来看看你是否能够得出每段代码所表示的公式。

清单 13-1 Basement Billiards

```

////////////////////////////////////
//Ball-to-ball collision
////////////////////////////////////
    for (int j=i+1;j<16;++j)
    {
        if (ball[i].collision(ball[j])&&!invalid_collision[i][j])
        {
            //使主球旋转有效
            if(!i)
            {
                apply_english=true;
            }

            invalid_collision[i][j]=invalid_collision[i][i]=1;

            GLT_VECTOR3 intersect={0,};
            //根据两球的中心计算向量的单位长度
            //这是一个特殊的例子,主球将会碰撞那些预先瞄准好的球,且将其设定为弹性碰撞
            if (i==0&j==predicted_ball)
            {
                intersect[0]=ball[j].x-col.x;
                intersect[1]=ball[j].y-col.y;
                predicted_ball=0;
            }
        }
    }

```

```

ball[0].x = col.x
ball[0].y = col.y
}
//否则,作一般处理
else
{
    intersect[0] = ball[j].x - ball[i].x;
    intersect[1] = ball[j].y - ball[i].y;
}

gltNormalize(intersect);

//一点小的优化;如果球没有移动,就不必计算向量
bool b1_moving = false;
bool b2_moving = false;

if(ball[i].vec[0] || ball[i].vec[1])
    b1_moving = true;
if(ball[j].vec[0] || ball[j].vec[1])
    b2_moving = true;

//将两球向量单位化
if(b1_moving)
    gltNormalize(ball[i].vec);
if(b2_moving)
    gltNormalize(ball[j].vec);

if(ball[i].vec[0] > .95f && fabs(ball[i].vec[1]) > .05f)
    bool what_the_heck = true;
//计算两球的角度及向量碰撞
float angle1 = 0, angle2 = 0;
if(b1_moving)
    angle1 = acosf(gltDotProduct(ball[i].vec, intersect));
if(b2_moving)
    angle2 = acosf(fabs(gltDotProduct(ball[j].vec, intersect)));

//仅当碰撞无效,且其发生于前一帧中该碰撞尚未处理完时才会做如下处理,此时该碰撞将会被
//忽略
//if(angle > (90 * GLT_PI_DIV_180) && angle > 270 * 90 * GLT_PI_DIV_180)
//continue;

if(i == 0)
    started = true;
//以下变量保存相应球的向量
//comp1 将保存引发碰撞的球体的平移分量值;comp2 将保存被碰撞的球体的平移分量值
float b1_comp1 = 0, b1_comp2 = 0, b2_comp1 = 0, b2_comp2 = 0;

```



```

//计算 ball1 的分量(由[i]定义)
if (b1_ moving)
{
    b1_ comp1 = sinf (angle1) * ball[i].velocity;
    b1_ comp2 = cosf (angle1) * ball[i].velocity;
}
//计算 ball2 的变量(由[j]定义)
if (b2_ moving)
{
    b2_ comp1 = sinf (angle2) * ball[j].velocity;
    b2_ comp2 = cosf (angle2) * ball[j].velocity;
}

//计算垂直于碰撞向量的向量
GLT_VECTOR3 norm_ intersect = { 0, };
gltCrossProduct (world_ up,intersect,norm_ intersect);

//综合得出球运动向量
GLT_VECTOR3 b1_ vcomp1 = { 0, };
GLT_VECTOR3 b1_ vcomp2 = { 0, };
GLT_VECTOR3 b2_ vcomp1 = { 0, };
GLT_VECTOR3 b2_ vcomp2 = { 0, };

//计算 ball1 的组合向量([i])
if (gltGetAngleDeg (ball[i].vec,norm_ intersect) < 90)
{
    b1_ vcomp1 [0] = norm_ intersect [0] * b1_ comp1;
    b1_ vcomp1 [1] = norm_ intersect [1] * b1_ comp1;
}
else
{
    b1_ vcomp1 [0] = -norm_ intersect [0] * b1_ comp1;
    b1_ vcomp1 [1] = -norm_ intersect [1] * b1_ comp1;
}
b1_ vcomp2 [0] = -intersect [0] * b2_ comp2;
b1_ vcomp2 [1] = -intersect [1] * b2_ comp2;

ball[i].vec [0] = b1_ vcomp1 [0] + b1_ vcomp2 [0];
ball[i].vec [1] = b1_ vcomp1 [1] + b1_ vcomp2 [1];

ball[i].velocity = sqrtf (ball[i].vec [0] * ball[i].vec [0] +
                           ball[i].vec [1] * ball[i].vec [1]);

//计算 ball2 的组合向量([j])
if (gltGetAngleDeg (ball[j].vec,norm_ intersect) < 90)
{

```

```

        b2_ vcomp1[0] = norm_ intersect[0] * b2_ comp1;
        b2_ vcomp1[1] = norm_ intersect[1] * b2_ comp1;
    }
else
{
    b2_ vcomp1[0] = -norm_ intersect[0] * b2_ comp1;
    b2_ vcomp1[1] = -norm_ intersect[1] * b2_ comp1;
}
b2_ vcomp2[0] = intersect[0] * b1_ comp2;
b2_ vcomp2[1] = intersect[1] * b1_ comp2;

//最后,将每个球的组合向量相加,赋值给这个球
ball[j].vec[0] = b2_ vcomp1[0] + b2_ vcomp2[0];
ball[j].vec[1] = b2_ vcomp1[1] + b2_ vcomp2[1];
ball[j].velocity = sqrtf(ball[j].vec[0] * ball[j].vec[0] +
                        ball[j].vec[1] * ball[j].vec[1]);

//击球前,更新球的向量
ball[i].nvec[0] = -ball[i].vec[1];
ball[i].nvec[1] = ball[i].vec[0];
ball[j].nvec[0] = -ball[i].vec[1];
ball[j].nvec[1] = -ball[j].vec[0];
glTNormalize(ball[i].nvec);
glTNormalize(ball[j].nvec);

//soundPlayer.playSample(collision);
}

//利用缓冲来修正一些小的碰撞
else
{
    invalid_collision[i][j] = invalid_collision[j][i] = 0;
}
}
}
}
}

```

本章总结

本章中，我们对碰撞进行了深入的研究。首先我们对两种简单的碰撞形式进行了讨论：一个物体去碰撞一个静止的物体。当物体碰撞到水平或竖直固定物体时，向量反射的计算是最简单的。非轴平行向量的计算则要用到更多的向量数学知识。接下来我们给出了动量和冲量的定义。这样我们就可以研究两个运动物体之间的碰撞了。最后我们讨论了弹性碰撞和完全非弹性碰撞，同时还介绍了两种不同的计算末速度的公式。

能力测试

多项选择

- 判断对错：入射角一定等于反射角。
A. 对 B. 错
- 判断对错：将物体放在有边界的盒子里可以避免复杂的非轴平行向量反射。
A. 对 B. 错
- 下列哪一项不是向量？
A. 动量 B. 冲量 C. 有角度的碰撞 D. 面的法线
- 下列 ϵ (还原系数) 中，哪个最接近弹性碰撞？
A. 0 B. 0.25 C. 0.5 D. 0.95
- 判断对错：在没有其他外力作用下，碰撞可以使两物体都停下来。
A. 对 B. 错

练习

练习 1：固定物体的碰撞

- 假设我们以俯视的角度编写一个 2D 游戏，桌台的四边分别平行于 x 和 y 轴，如果一个球以 $[80 \ -30]$ 的速度碰到屏幕上沿，那么碰撞后的速度是多少？
- 假如我们以俯视的角度编写一个 2D 游戏，桌台一边过点 $(350, 200)$ 和 $(440, 250)$ ，如果一个球以 $[50 \ 60]$ 的速度碰到这个边，那么碰撞后的速度是多少？
- 假如我们编写一个 3D 游戏，向量 $[60 \ 90 \ -45]$ 和 $[180 \ -30 \ -75]$ 定义了一个面，如果一个球以 $[20 \ -5 \ -35]$ 的速度碰到这个面，那么碰撞后的速度是多少？

练习 2：动量和冲量

- 一块重 65lbs 的石头以 11m/s 速度水平滚动，它的动量是多少？
- 在 3D 游戏中，重 1500kg 的卡车速度是 $\begin{bmatrix} 25 \\ 40 \\ -50 \end{bmatrix}$ ，问此时它的动量是多少？
- 在一个格斗游戏中，一个重 210lbs 且保持不动的玩家被另一个玩家以 $3500\text{kg} \cdot \text{m/s}$ 的冲量踢打。问这个玩家将要以什么样的速度飞出？
- 在第 3 题中，施力者也受到了同样的动量作用，假设他是以 3.5m/s 的速度踢向他的对手，问他的末速度是多少？

- 假设你正在为一个 3D 的格斗游戏。玩家给了一个物体 $\begin{bmatrix} 2000 \\ 4500 \\ -1350 \end{bmatrix}$ 的冲量，物体重 215lbs，

那么物体的末速度将是多少？

练习 3：碰撞建模

1. 在 3D 弹球游戏中，一个重 0.45kg 的球以 $[-10\ 50\ -15]$ 的速度碰撞到另一个重 0.10kg 的球，碰撞后第一个球静止，求第二个球的速度。
2. 假设你在为一个 3D 的赛车游戏编程。一辆重 3000kg 的汽车以 $[60\ -5\ -45]$ 的速度撞上另一辆重 1500kg 以 $[-70\ 60\ -5]$ 速度运动的汽车。如果碰撞后它们的速度相同，那么它们最后的速度是多少？
3. 假设你在为 3D 的赛车游戏中汽车碰撞的部分编程，但这一次汽车并不会粘在一起运动。一辆重 1000kg 的汽车以 $[60\ -10\ -45]$ 的速度撞上另一辆重 1700kg 以 $[-45\ 20\ -55]$ 速度运动的汽车。假设 $e=0.5$ ，那么它们最后的速度是多少？

程序设计**程序设计 1：编写程序显示盒子中的物体**

1. 编写一段简单的程序以显示盒子中的物体在屏幕上做线性运动，确保每一次碰撞都是向量反射。
2. 旋转第 1 题中盒子的边界，以使得碰撞不再是轴平行的。

程序设计 2：数学方程、向量反射、碰撞方程

扩展数学方程库，加入有关向量反射方程和碰撞方程。

讨论问题

1. 当谈论到向量反射时，什么是轴平行？
2. 什么是还原系数，在碰撞中如何使用它？
3. 描述一下从弹性碰撞到完全非弹性碰撞的范围，举例说明。
4. 我们现在都有哪些不同的求末速度的方法(包括以前章节中提到的)？



第14章 旋转运动

本章目标

在阅读完本章并完成练习后，你将能够做到以下几点：

- 定义和计算角位移
- 定义和计算角速度和切向速度
- 定义和计算角加速度和切向加速度
- 相对于旋转运动修改五个运动方程
- 定义和计算转矩
- 相对于旋转运动修改牛顿第二定律
- 定义和计算转动动量
- 定义和计算角冲量

介绍

上一章我们讨论了线性碰撞(正面碰撞)。现在，你也许想知道其他那些非质点对称碰撞。回忆一下你打台球的情景，想击打球的中心，但你是否真的击中了呢？大多数情况下我们并没有做到。这样球就偏移了原有的方向，同时还在旋转。如果你继续研究就会发现，碰撞产生了两种结果：一是使物体向其他方向运动，二是使球开始旋转。我们已经知道了如何处理第一种结果，这一章中，我们将介绍最后一个难题——旋转。

14.1 圆周运动

毫无疑问，我们已经掌握了关于圆周运动的所有概念。在前面的章节中，我们介绍的位移、速度和加速度都将应用于旋转运动。现在先看一下计算机读取光盘时，磁盘的运动情况如图 14-1 所示。

点 C 表示旋转轴。光盘上一点 P 以半径 r 绕 C 旋转。经过 Δt ， P 点沿着圆形路径移动了一段距离(s)，我们称之为弧长。如图 14-2。

在图 14-2 中，我们可以看到经过一段时间后的旋转角 θ 。在第 3 章中，我们曾经介绍说角应该用弧度表示，还讨论了如何在角度和弧度之间进行转化，即旋转角表示角位移，用弧长除以半径进行计算。

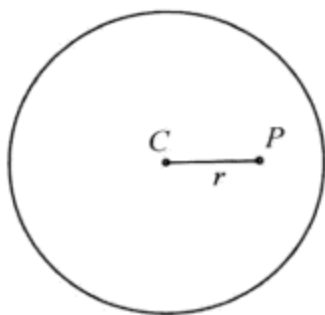


图 14-1 位于光盘上的点

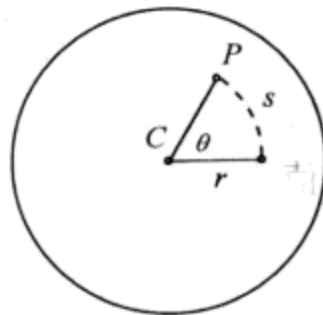


图 14-2 点 P 的运动轨迹

角位移

$$\theta = s/r$$

其中 s 是弧长, r 是半径

注意

如果弧长等于半径, 那么 $\theta = 1$, 这就是弧度的最早定义。

下面是用弧长和半径计算弧度的一个函数:

```
float angDisplacement(float arc, float radius)
{
    float theta;
    theta = arc/radius;
    return theta;
}
```

我们已经定义了角位移, 现在来看一下平均角速度 ω 。你也许已经猜到其实它就是弧度的变化率。换句话说就是用弧度除以时间, 就像线速度一样。

平均角速度

$$\bar{\omega} = \frac{\Delta\theta}{\Delta t} = \frac{\theta_f - \theta_i}{t_f - t_i}$$

其中 θ 是角位移, t 是时间。

注意

希腊字母 ω 用来表示角速度, 读作 omega。

如果我们进一步观察平均角速度的定义, 就能猜到它的单位是什么。由于角速度是用弧度(rad)和时间(s)来衡量的, 所以它的单位是弧度/秒(rad/s)。

【例 14-1】 光盘的平均角速度

如果光盘的转速是 9500rpm(圈每分钟), 那么它的角速度是多少?

解答:

1. 计算角位移, 一圈相当于 2π 弧度, 则

$$9500\text{rev}\left(\frac{2\pi^{\text{R}}}{1\text{rev}}\right) = 19000\pi^{\text{R}}$$

2. 把 1 分钟转换成 60 秒。
3. 将这两个值代入定义中:

$$\omega = \frac{\Delta\theta}{\Delta t} = \frac{19000\pi^{\text{R}}}{60\text{s}} \approx 316.67^{\text{R}}/\text{s}$$

下面这段代码是将这一方法写成一个函数:

```
float avgAngularVelocity(float arcStart, float arcEnd, float time,
    float radius)
{
    float initialDisplacement, endDisplacement, omega;

    //计算角位移
    initialDisplacement = arcStart/radius;
    endDisplacement = arcEnd/radius;

    //应用公式
    omega = (endDisplacement - initialDisplacement) / time;
    return omega;
}
```

这种方法利用了弧的起始点、终结点、半径和总的时间来计算平均速度。在这个函数中, 首先计算角位移, 然后代入公式计算平均角速度。

通常, 我们解决问题时总要找一工作量最小的方法。前面提到的这个函数就非常有用, 只要知道弧长和半径就可以计算了。我们在编程时要牢记这一点。

现在我们区分一下线速度和角速度。在例 14-1 中, 由于 CD 是刚体, 所以在它上面的所有点都有同样的加速度, 而离圆心较远的点要比靠近圆心的点的线速度大。

在第 8 章中讨论了平均线速度和瞬时线速度的区别。在这里我们同样要研究它们。瞬时角速度是在一瞬间的平均角速度。

现在我们像在第 8 章中研究线速度时一样, 用加速度表示速度变化率来定义角加速度。换句话说线加速度相当于线速度的变化量除以时间, 而在旋转运动中, 平均角加速度 $\bar{\alpha}$ 相当于角速度的变化率。

平均加速度

$$\bar{\alpha} = \frac{\Delta\omega}{\Delta t} = \frac{\omega_f - \omega_i}{t_f - t_i}$$

其中 ω 为角速度, t 为时间。

注意

希腊字母 α 用来表示角加速度, 读作 alpha。

如果我们进一步看角加速度的单位 rad/s^2 ，就会发现它和线加速度的单位 m/s^2 很相似。另外瞬时角加速度相当于一瞬间的平均角加速度。

【例 14-2】 计算轮子的平均角加速度

假设你在为赛车游戏“Need for Speed Under Ground”编程，开始时轮子的角速度是 5rad/s ，两秒后，角速度为 15rad/s ，那么平均角加速度是多少？

解答：

只要严格按照定义计算就可以了：

$$\bar{\alpha} = \frac{\Delta\omega}{\Delta t} = \frac{\omega_f - \omega_i}{t_f - t_i} = \frac{15\text{rad/s} - 5\text{rad/s}}{2\text{s}} = \frac{10\text{rad/s}}{2\text{s}} = 5\text{rad/s}^2$$

就像光盘一样，轮子上每个点的角加速度都相等，但是根据与旋转轴的距离不同，线加速度不同。

我们可以用函数计算 α ，即用角速度的变化量除以时间：

```
float avgAngAcceleration(float angVelBegin, float angVelEnd, float time)
{
    float alpha;
    alpha = (angVelEnd - angVelBegin)/time;
    return alpha;
}
```

在这个函数中，我们直接输入时间间隔，而不是用末时刻减去初时刻。如果时间够短，这个函数也可以对瞬时角加速度进行计算。即可以通过转化，输入角度进行计算。

现在我们重新定义了圆周运动中的位移、速度和时间。这些概念都是从直线运动中借用过来的。你也许猜到我们还要借用第 8 章的五个运动方程，唯一不同的是变量发生了变化。位移由 Δx 变成了 $\Delta\theta$ ，速度由 v 变成了 ω ，加速度由 a 变成了 α 。表 14-1 中列出了转化后的方程。

表 14-1 加速度的计算方程

方程 1	$\omega_f = \omega_i + \alpha t$
方程 2	$\bar{\omega} = \frac{\omega_i + \omega_f}{2}$
方程 3	$\Delta\theta = 1/2(\omega_i + \omega_f)t$
方程 4	$\Delta\theta = \omega_i t + 1/2\alpha t^2$
方程 5	$\omega_f^2 = \omega_i^2 + 2\alpha\Delta\theta$

使用的过程和原来一样，只不过我们用角度变量代替了线性变量。

【例 14-3】 轮子的命运

假设你在为一个“轮子的命运”的游戏编程，轮子的角初速度是 8rad/s ，角加速度是 -2rad/s^2 ，那么 3 秒后轮子的角位移是多少？

解答:

1. 和以前在线性运动中要做的一样, 我们先在列表中列出已知量和要求的量:

条件	求值
$\omega_i = 8\text{rad/s}$	$\Delta\theta = ?$
$t = 3\text{s}$	
$\alpha = -2\text{rad/s}^2$	

2. 根据列表我们选择第 4 个方程。

3. 把已知量代入方程, 求 $\Delta\theta$:

$$\Delta\theta = \omega_i t + 1/2 \alpha t^2$$

$$\Delta\theta = (8\text{rad/s})(3\text{s}) + 1/2(-2\text{rad/s}^2)(3\text{s})^2$$

$$\Delta\theta = 24\text{rad} - 9\text{rad}$$

$$\Delta\theta = 15\text{rad}$$

刚才, 我们着重介绍了线性运动和圆周运动的相似点。它们之间唯一的区别就是物体上所有的点的角速度和角加速度都相等, 但根据点到圆心的距离, 它们的线速度和线加速度不同。下面进一步看看线性变量和圆周变量的关系, 首先从线速度和角速度开始。

在这一章的开始, 我们定义了角位移等于弧长除以半径:

$$\Delta\theta = \frac{\Delta s}{r}$$

为了求角速度我们需要把它除以时间, 等式两边都除以时间:

$$\omega = \frac{\Delta\theta}{\Delta t} = \frac{1}{r} \left(\frac{\Delta s}{\Delta t} \right)$$

现在我们迈进了一大步。我们曾经介绍过如果要求瞬时角速度就要使时间间隔非常小。在时间变小的同时, 弧长也越来越趋近于直线。当它变成一个极短的直线时, 我们就可以用 $(\Delta s / \Delta t)$ 计算线速度, 方程就变成如下形式:

$$\omega = \frac{v}{r}$$

下面再看一下例 14-2, 如果时间段慢慢变小, θ 变小, s 趋近于弧上的一个切线。这就是为什么把这一点的线速度叫做切向速度 v_t 。如图 14-3 所示。

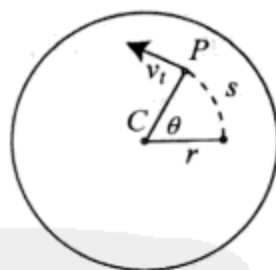


图 14-3 位于点 P 的角速度

切向速度

$$v_t = \omega r$$

其中 ω 是角位移, r 是半径。

编程计算切向速度很简单, 下面就是一个例子:

```
float tangVelocity(float omega, float radius)
{
    float velT;
    velT = omega*radius;
    return velT;
}
```

这个函数根据 ω 和时间计算切向速度。

观察这个切线速度的最好例子就是找一端绑着线的小球或者一个 yo-yo 球。抓住绳子的一端开始摆动小球，你会发现球会始终以固定的距离绕着你的手转。此时如果小球与绳子脱离会怎样呢？实际上，在它们分离的一瞬间，小球沿着被释放那一点的切线方向飞出。如图 14-4 所示。

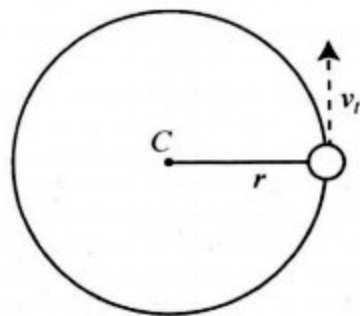


图 14-4 旋转的小球

【例 14-4】投棒球

假设你在为一个类似全明星球赛的游戏编程，投球者的前臂长 0.5m，球飞出时的即时角速度是 100rad/s，那么球刚飞出时的初始线速度是多少？

解答：

由于 $v_t = \omega r$ ，所以求得切向速度是 100rad/s $\times$ 0.5m，速度方向是球员释放球时的切向方向。我们同样可以这样表示线加速度和角加速度的关系。我们先从速度的关系入手：

$$v_t = \omega r$$

为了计算加速度我们把等式两边同时除以时间：

$$\frac{v_t}{\Delta t} = \frac{\omega}{\Delta t}(r)$$

$$a_t = \alpha r$$

切向加速度

$$a_t = \alpha r$$

其中， α 是瞬间角加速度， r 是半径。

【例 14-5】再次研究游戏“轮子的命运”

假设你再次为游戏“轮子的命运”编程，轮子的初角速度是 8rad/s，10 秒后停下，如果轮子的半径是 3m，那么需要多大的切向加速度刹车才行？

解答：

1. 首先需要计算角加速度，列出已知量：

条件	求值
$\omega_i = 8\text{rad/s}$	$\alpha = ?$
$t = 10\text{s}$	
$\omega_f = 0\text{rad/s}$	

2. 选择方程 1 计算:

$$\omega_f = \omega_i + \alpha t$$

$$0 \text{ rad/s} = 8 \text{ rad/s} + \alpha(10 \text{ s})$$

$$\alpha = -0.8 \text{ rad/s}^2$$

3. 用角加速度算出切向加速度:

$$a_t = \alpha r$$

$$a_t = -0.8 \text{ rad/s}^2 (3 \text{ m})$$

$$a_t = -2.4 \text{ m/s}^2$$

我们看到, 线性运动中的位移、速度和加速度等概念都可移植到角度运动中来。线性运动是基于平移的, 而角度运动是基于旋转的。不过要注意, 刚体上所有的角速度和角加速度都是相同的, 但线速度和线加速度却根据它们距离中心的远近不同而不同, 下一节, 我们将运用这些定义来介绍是什么引起了旋转运动。

自我测试

1. 旋转木马在 30 秒内转了 10 圈, 它的角速度是多少?
2. 如果旋转木马的角初速度是 5 rad/s , 在给了它一个推力后, 3 秒后它的速度变为 15 rad/s , 那么角加速度是多少?
3. 当旋转木马达到最大速度 15 rad/s , 然后以 -3 rad/s^2 的速度变慢, 需要多长时间它才可以停下来?
4. 如果当旋转木马达到最大速度 15 rad/s , 以 -3 rad/s^2 的速度变慢, 那么停止时它的角位移是多少?
5. 如果在旋转木马上坐的孩子没有抓紧, 他从半径为 2.5 m 、角速度为 15 rad/s 的木马上摔下来, 那么孩子摔出的线速度是多少?

14.2 旋转力学

在前面章节中描述了角的运动方程, 现在我们已经可以讨论角的位移、速度和加速度, 最后我们需要做的就是弄清楚是什么引起了旋转运动。在第 11 章中, 牛顿定律告诉我们是什么导致了线性运动, 牛顿第二定律说, 加在物体上的合力决定了物体的加速度。此前假设所有的力都作用在物体的质心上, 所以最终结果只是简单的直线运动。但是, 如果力作用在物体的某一个顶点呢? 它不仅能使物体做平移运动, 还能使物体绕着它的质心运动。

如果物体受到的是切向力而不是对质心的力, 那么我们就可以利用牛顿第二定律计算:

$$F_t = ma_t$$

把等式两边同时乘以重心到力的作用点的半径:

$$F_t(r) = ma_t(r)$$

根据前一节的知识, 我们得到:

$$a_t = r\alpha$$

即

$$F_t(r) = m(r\alpha)(r) = mr^2\alpha$$

方程左边表示力矩，它使物体绕着重心旋转。

力矩

$$\tau = mr^2\alpha$$

其中 m 是质量， r 是半径， α 是角加速度。

注意

如果继续沿用公制，我们就可以得到它的单位 $\text{N} \cdot \text{m}$ 。

现在来比较一下线性运动和旋转运动的相似性。刚才我们提到了牛顿第二定律， $F = ma$ 。我们把力矩看成线性运动中的力，是否可以得到旋转运动中的牛顿第二定律？我们知道 $\tau = mr^2\alpha$ ，质量就相当于旋转惯量(I)等于 mr^2 。这就是说力矩等于旋转惯量乘以角加速度。看起来和牛顿第二定律的形式一样，不是吗？

牛顿第二定律的变形

$$\tau = I\alpha$$

其中 $I = mr^2$ ， α 是角加速度。

正如力产生加速度一样，力矩产生角加速度。就像以前一样，如果我们知道了角加速度就可以利用方程求出加速度和角位移。

【例 14-6】 撞车

如果给一个如偷车贼的游戏编程。在游戏中，一辆汽车以 $5000\text{N} \cdot \text{m}$ 的力矩撞上另一辆车的后保险杠，如果被撞车的质量是 1200kg ，保险杠距离重心的长度是 3m ，那么产生的角加速度是多少？

解答：

直接利用前面介绍的方程：

$$\tau = mr^2\alpha$$

$$5000\text{N} \cdot \text{m} = 1200\text{kg}(3\text{m})^2(\alpha)$$

$$\alpha = 0.463\text{rad/s}^2$$

就像处理线性运动一样，如果知道角加速度，就可以利用五个运动方程求出角速度和位移。记住，在编程中要处理的是每一帧图像的变化。

我们同样可以把动量和动能两个方程转化为旋转运动的形式。回忆下它们是怎样处理线性运动的。它们都是利用速度或速率进行计算，然后乘以时间。我们同样可以把这个概念应用于旋转力学。首先来看动能。

如果旋转惯量相当于质量，角速度相当于速度，那么我们就可以猜出旋转动能的定义。

旋转动能

$$KE_R = 1/2 I \omega^2$$

其中, $I = mr^2$, ω 是角速度。

注意

前面我们用 KE_T 表示动能, 这里我们 KE_R 表示旋转动能。

【例 14-7】 滚动的球

如果你给一个如“Crash Bandicoot”的游戏编程。一个重 0.5kg 的球从 10m 高的山坡上滚下山, 如果球的旋转惯量是 $(2/5)mr^2$, 球到达山脚的线速度是多少?

解答:

我们把旋转动量加到能量守恒定律中进行计算:

$$PE_i + KE_{Ti} + KE_{Ri} = PE_f + KE_{Tf} + KE_{Rf}$$

$$mgy_i + 1/2mv_i^2 + 1/2I\omega_i^2 = mgy_f + 1/2mv_f^2 + 1/2I\omega_f^2$$

$$mgy_i + 1/2m(0)^2 + 1/2I(0)^2 = mg(0) + 1/2mv_f^2 + 1/2I\omega_f^2$$

$$mgy_i = 1/2mv_f^2 + 1/2I\omega_f^2$$

$$0.5\text{kg}(9.8\text{m/s}^2)(10\text{m}) = 1/2(0.5\text{kg})v_f^2 + 1/2((2/5)(0.5\text{kg})r^2)\omega_f^2$$

$$49\text{J} = 0.25\text{kg}(v_f^2) + 0.1\text{kg}(r^2)\omega_f^2$$

注意由于 $v = r\omega$, 则 $v^2 = r^2\omega^2$:

$$49\text{J} = 0.25\text{kg}(v_f^2) + 0.1\text{kg}(v_f^2)$$

$$49\text{J} = 0.35\text{kg}(v_f^2)$$

$$49\text{J} = 0.35\text{kg}(v_f^2)$$

$$140\text{m}^2/\text{s}^2 = v_f^2$$

$$v_f \approx 11.832\text{m/s}$$

用数学方法计算比较复杂, 但是我们还是要试着编写一个用质量、高度和旋转惯量来计算线速度的函数:

```
#define GRAVITY 9.81
void LinearSpeed()
{
    //定义一些浮点数
    float mass, initialHeight, inertia, energy = 0.0f;
    float halfMass, halfInertiaMass linearSpeed, temp = 0.0f;

    cout<<"Let's calculate the linear speed of a rolling
    object!\n";
    cout<<"First, we need some information...\n";
    cout<<"Please specify a mass for the ball in kg\n";
    cin>>mass;
```

```

cout<<"Next, give us an initial height for the ball in
meters\n";
cin>>initialHeight;

cout<<"Lastly, give us an inertia for the ball\n";
cin>>inertia;
cout<<"\n";

//首先计算那些可以确定的值
energy = mass*initialHeight*GRAVITY;

//存储式: 1/2(m)vf^2 的值
halfMass = mass/2;

//存储式子: 1/2(inertia)*(mass) r^2 * wf^2 的值

halfInertiaMass = inertia*mass/2;

//将其值保存
temp = energy/(halfMass+halfInertiaMass);

//求平方根, 单位为 m/s
linearSpeed = sqrt(temp);

cout<<"The final linear speed is "<<linearSpeed<<"
meters/second\n";

}

```

在这段代码中出现了很多量, 我们来研究一下。首先来看最初的势能。变量 `halfMass` 存储的是质量的一半。这样可以省略将质量除以 2 的代码。在计算出惯性乘以质量的一半后, 把这个值赋给 `halfInertiaMass`。然后可以用上面的两个量计算出能量。通过计算 `temp` 的值的平方来求得球到达山底时的线性速度。另外调用 `math.h` 头文件使用平方运算。这样就可以减少 CPU 的运算量。

数学编程的技巧就在于充分理解方程中的每一个量, 然后编写一个能使 CPU 尽可能快速计算的程序。提高计算的速度并不是一件很容易的工作。

我们要介绍的另一个量是转动动量。重复前面的对比就会发现, 转动动量是转动惯量和角速度的乘积。

旋转惯量

$$L = I\omega$$

其中 $I = mr^2$, ω 是角速度。

还记得在第 13 章中如何将牛顿第二定律从 $F = ma$ 转换到动量定理吗？最后我们得出结论是作用在物体上的冲量的改变量导致了动量的改变。现在我们同样可以将这一概念应用到旋转运动中。任何非对心冲量导致了非对心动量的变化。

我们看到，旋转运动是线性运动概念的扩展。公式的形式都一样，只不过里面使用的是旋转运动的量。如果你已经充分明白了线性运动概念的含义，就可以把这些概念同旋转运动中的概念对应起来。当我们有了足够的基础时，就可以编写诸如 Havok 和 MathEngine 这样的商业物理驱动程序。实际上，许多商业游戏驱动系统都可以互相合并，这样我们就能够方便灵活的应用它们。比如 2003 年，Unreal Tournament 就和 MathEngine 的 Karma 物理库合并。虽然，它们并没有公布具体的算法，但是这确实是一个研究物理系统的好机会。另外，在有了足够基础的前提下我们也可以起草自己的模拟程序。不要在合上书后，想当然的认为，你已经可以做到这些了。我希望各位放下这本书，然后开始新的学习历程去研究其他资料。这只是起点！

自我测试

1. 假如再给旋转木马一个力，使得旋转木马有了 6rad/s^2 的加速度。如果半径是 2.5m ，且总的质量为 500kg ，那么这个人给木马的力矩是多少？
2. 假如编写一个疯狂出租车的游戏，一辆重 1000kg 的车被 $3000\text{N} \cdot \text{m}$ 的力矩撞在距离重心 2m 的地方，那么角加速度是多少？

本章总结

本章中，我们重新讨论了在旋转运动中的物理概念，首先用弧长来定义角位移。然后扩展到角速度和角加速度。同时我们也看到五个运动方程也可以应用到角度值中，并扩展到牛顿第二定律和旋转运动的动量定律。现在你已经可以解决有关线性运动和旋转运动的问题了。

能力测试

多项选择

1. 判断对错：角位移就是弧长除以半径。
A. 对
B. 错
2. 下面哪一项是角速度的单位？
A. m/s
B. m/s^2
C. rad/s
D. rad/s^2
3. 判断对错：五个运动方程不能用于旋转运动。
A. 对
B. 错
4. 下列哪一项的单位是 $\text{N} \cdot \text{m}$ ？
A. 角加速度
B. 力矩
C. 旋转动能
D. 角冲量

5. 判断对错：机械能守恒定律可以扩展到旋转能量中。

A. 对

B. 错

练习

练习 1：圆周运动

1. 旋转木马在 55 秒内转了 21 圈，它的角速度是多少？
2. 如果旋转木马的角初速度是 6rad/s ，在给了它一个推力后，4 秒后它的速度变为 24rad/s ，那么角加速度是多少？
3. 当旋转木马达到最大速度 12rad/s ，然后以 -2rad/s^2 的速度变慢，需要多长时间它才可以停下来？
4. 如果当旋转木马达到最大速度 12rad/s 后，以 -3rad/s^2 的速度变慢，那么直到停止时它的角位移是多少？
5. 如果在旋转木马上坐的孩子没有抓紧，他从半径为 1.3m 、角速度为 11rad/s 的木马上摔下来，那么孩子摔出的线速度是多少？

练习 2：旋转力学

1. 假如再给旋转木马一个力，使得旋转木马有了 7rad/s^2 的加速度。如果半径是 1.3m ，且总的质量为 490kg ，那么这个人给木马的力矩是多少？
2. 假如编写一个疯狂出租车的游戏，一辆重 1300kg 的车被 $3300\text{N} \cdot \text{m}$ 的力矩撞在距离重心 1.83m 的地方，那么角加速度是多少？

程序设计

程序设计 1：编写程序求得物体状态

编写简单的程序，输入有关物体的三方面信息和用户要计算的值的数量。选择适当的方程计算所要求的数值。

程序设计 2：编写图形程序

编写一个简单的程序，就像在例 14-7 中所描述的那样演示球滚下山的情景。

程序设计 3：数学方程和圆周运动

扩展数学方程库，修改所有的物理方程以使它们适应旋转运动。

讨论问题

1. 线性运动和圆周运动有哪些相同点和不同点？
2. 非对心碰撞与变换这一章的内容有何联系？
3. 在游戏场景中，如何运用切向速度或加速度？
4. 如果商业物理引擎是可用的，为什么我们还要学习这些物理概念？

附录 A 一些数学和物理公式

数学和物理中的一些特殊符号，如表 A-1 所示：

表 A-1 一些特殊符号

符 号	名 称	描 述
α	Alpha	常用来表示角
Δ	Delta	变化量
ϵ	Epsilon	恢复系数
$f'(t)$		关于参数 t 的函数 f 的派生函数
μ	Mu	摩擦系数
ω	Omega	多用于描述角速度
π	Pi	圆周率常数 3.14159265
θ	Theta	多用于描述角

第 1 章 点和直线

直线方程

描述直线的一般式方程是 $Ax + By = C$ ，其中 A 和 B 不能同时为零，这个方程的图像是一条直线。

直线的斜率

$$\text{slope} = m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

对于直线方程的一般式 $Ax + By = C$ ，它的斜率为 $m = -A/B$ 。

平行直线

如果两条直线平行，那么它们的斜率相等，即 $m_1 = m_2$ 。

垂直直线

如果两条直线互相垂直，那么它们的斜率乘积等于 -1 ，即 $m_1 m_2 = -1$ 或者 $m_1 = -\frac{1}{m_2}$ ，
或者 $m_2 = -\frac{1}{m_1}$ 。

直线方程的其他形式

直线的斜切式方程: $y = mx + b$

直线的点斜式方程: $(y - y_1) = m(x - x_1)$ 。其中, 坐标 (x_1, y_1) 是直线上的任意一点。

求解直线方程组

这主要用于判断平面上两条直线的关系:

- 一个解 如果两条直线的斜率相等, 那么它们相交。
- 无穷多个解 如果两条直线的斜率相等, 且截距也相等, 那么它们重合。
- 无解 如果两条直线的斜率相等, 但截距不相等, 那么它们平行。

第2章 相关几何知识

勾股定理

在直角三角形中, 斜边 c 的平方等于两条直角边 a 和 b 的平方之和: $c^2 = a^2 + b^2$ 。

平面上两点之间距离

$$P_1P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

其中, $P_1(x_1, y_1)$ 和 $P_2(x_2, y_2)$ 是直线 P_1P_2 上的两个点。

3D 空间里两点的距离公式

$$P_1P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

其中, $P_1(x_1, y_1, z_1)$ 和 $P_2(x_2, y_2, z_2)$ 是空间直线 P_1P_2 上的两个点。

直角平面上两点的中点公式

$$M\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

点 M 为 $P_1(x_1, y_1)$ 和 $P_2(x_2, y_2)$ 之间的中点。

3D 空间的中点公式

$$M\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}, \frac{z_1 + z_2}{2}\right)$$

点 M 为 3D 空间中 $P_1(x_1, y_1, z_1)$ 和 $P_2(x_2, y_2, z_2)$ 之间的中点。

竖直抛物线的对称轴

$y = a(x - h)^2 + k$, 其中 (h, k) 是抛物线的顶点坐标, 对称轴是 $x = h$ 。

水平抛物线的对称轴

$x = a(y - k)^2 + h$, 其中 (h, k) 是抛物线的顶点坐标, 对称轴是 $y = k$ 。

圆的方程

$$(x - h)^2 + (y - k)^2 = r^2$$

其中, 圆心是 $O(h, k)$, 半径是 r 。

单位圆方程

$$x^2 + y^2 = r^2$$

其中, 圆心是 $O(0, 0)$, 半径是 r 。

球方程

$$(x - h)^2 + (y - k)^2 + (z - l)^2 = r^2$$

其中, 球心是 $O(h, k, l)$, 半径是 r 。

单位球方程

$$x^2 + y^2 + z^2 = r^2$$

其中, 球心是 $O(0, 0, 0)$, 半径是 r 。

圆与圆的碰撞检测

两圆方程为 $(x - h_1)^2 + (y - k_1)^2 = r_1^2$ 和 $(x - h_2)^2 + (y - k_2)^2 = r_2^2$, 如果 $\sqrt{(h_2 - h_1)^2 + (k_2 - k_1)^2} \leq (r_1 + r_2)$, 那么两个圆相交。

优化的圆与圆的碰撞检测

两圆方程为 $(x - h_1)^2 + (y - k_1)^2 = r_1^2$ 和 $(x - h_2)^2 + (y - k_2)^2 = r_2^2$, 如果 $(h_2 - h_1)^2 + (k_2 - k_1)^2 \leq (r_1 + r_2)^2$, 那么两个圆相交。

优化的球与球的碰撞检测

两球方程为 $(x - h_1)^2 + (y - k_1)^2 + (z - l_1)^2 = r_1^2$ 和 $(x - h_2)^2 + (y - k_2)^2 + (z - l_2)^2 = r_2^2$, 如果 $(h_2 - h_1)^2 + (k_2 - k_1)^2 + (l_2 - l_1)^2 \leq (r_1 + r_2)^2$, 那么两个球相交。

第3章 三角函数

角度转换成弧度

$$\text{角度} \times \left(\frac{\pi^R}{180^\circ} \right) = \text{弧度}$$

弧度转换成角度

$$\text{弧度} \times \left(\frac{180^\circ}{\pi^R} \right) = \text{角度}$$

三角函数公式

$$\sin \alpha = \frac{\text{opp}}{\text{hyp}} = \frac{b}{c}$$

$$\cos \alpha = \frac{\text{adj}}{\text{hyp}} = \frac{a}{c}$$

$$\tan \alpha = \frac{\text{opp}}{\text{adj}} = \frac{b}{a}$$

其中, opp 为直角三角形中角 α 的对边, adj 为直角三角形中角 α 的邻边, hyp 为直角三角形的斜边。

其他三角函数

$$\csc \alpha = \frac{1}{\sin \alpha} = \frac{\text{hyp}}{\text{opp}}$$

$$\sec \alpha = \frac{1}{\cos \alpha} = \frac{\text{hyp}}{\text{adj}}$$

$$\cot \alpha = \frac{1}{\tan \alpha} = \frac{\text{adj}}{\text{opp}}$$

其中, opp 为直角三角形中角 α 的对边, adj 为直角三角形中角 α 的邻边, hyp 为直角三角形的斜边。

正弦函数周期

对于正弦函数 $y = \sin(Bx)$, 它的周期是 $\frac{360^\circ}{|B|}$ 。

正弦函数振幅

对于正弦函数 $y = A \sin(x)$, 它的振幅是 $|A|$ 。

单位圆性质

$$\cos^2 \alpha + \sin^2 \alpha = 1$$

正切和余切函数

$$\tan \alpha = \frac{\sin \alpha}{\cos \alpha}$$

$$\cot \alpha = \frac{\cos \alpha}{\sin \alpha}$$

负角

$$\sin(-\alpha) = -\sin \alpha$$

$$\cos(-\alpha) = \cos \alpha$$

$$\tan(-\alpha) = -\tan \alpha$$

正弦函数的和与差

$$\sin(\alpha_1 + \alpha_2) = \sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2$$

$$\sin(\alpha_1 - \alpha_2) = \sin \alpha_1 \cos \alpha_2 - \cos \alpha_1 \sin \alpha_2$$

余弦函数的和与差

$$\cos(\alpha_1 + \alpha_2) = \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2$$

$$\cos(\alpha_1 - \alpha_2) = \cos \alpha_1 \cos \alpha_2 + \sin \alpha_1 \sin \alpha_2$$

第 4 章 向量运算

矢量和标量的标准

标量只有大小，没有方向。矢量既有大小，也有方向。

位移

物体的位移 = 末向量 - 始向量。

$$\Delta x = x_f - x_i$$

极坐标

$$\text{Vector } A = \|A\| @ \theta$$

其中 $\|A\|$ 是向量 A 的大小， θ 是向量 A 的方向。

笛卡儿坐标

$$\text{Vector } B = b_1 \hat{i} + b_2 \hat{j}$$

其中， $\hat{i}$ 和 $\hat{j}$ 分别表示 x 方向和 y 方向上的单位向量。

极坐标转换成笛卡儿坐标

对于向量 $A = \|A\| @ \theta$,

$$A = a_1 \hat{i} + a_2 \hat{j}$$



其中 $a_1 = \|A\| \cos\theta$, $a_2 = \|A\| \sin\theta$

笛卡儿坐标转换为极坐标

对于向量 $B = b_1 \hat{i} + b_2 \hat{j}$,

$$\|B\| = \sqrt{(b_1)^2 + (b_2)^2}, \theta = \tan^{-1}\left(\frac{b_2}{b_1}\right).$$

3D 笛卡儿坐标

对于向量 $B = b_1 \hat{i} + b_2 \hat{j} + b_3 \hat{k}$

其中, $\hat{i}$, $\hat{j}$ 和 $\hat{k}$ 分别表示 x 方向, y 方向上和 z 方向上的单位向量。

向量交换率

$$A + B = B + A$$

A 和 B 表示任何向量。

2D 向量加法

$$A + B = (a_1 + b_1) \hat{i} + (a_2 + b_2) \hat{j}$$

对于向量 $A = a_1 \hat{i} + a_2 \hat{j}$, $B = b_1 \hat{i} + b_2 \hat{j}$ 。

3D 向量加法

$$A + B = (a_1 + b_1) \hat{i} + (a_2 + b_2) \hat{j} + (a_3 + b_3) \hat{k}$$

对于向量 $A = a_1 \hat{i} + a_2 \hat{j} + a_3 \hat{k}$, $B = b_1 \hat{i} + b_2 \hat{j} + b_3 \hat{k}$ 。

向量减法

$$A - B = (a_1 - b_1) \hat{i} + (a_2 - b_2) \hat{j}$$

对于向量 $A = a_1 \hat{i} + a_2 \hat{j}$, $B = b_1 \hat{i} + b_2 \hat{j}$ 。

3D 向量减法

$$A - B = (a_1 - b_1) \hat{i} + (a_2 - b_2) \hat{j} + (a_3 - b_3) \hat{k}$$

对于向量 $A = a_1 \hat{i} + a_2 \hat{j} + a_3 \hat{k}$, $B = b_1 \hat{i} + b_2 \hat{j} + b_3 \hat{k}$ 。

标量与向量(极坐标形式)相乘

$$cA = c \|A\| @ \theta$$

对于任意常数 c 和向量 $A = \|A\| @ \theta$ 。

标量与向量(笛卡儿坐标形式)相乘

$$cA = ca_1 \hat{i} + ca_2 \hat{j}$$

对于任意常数 c 和向量 $A = a_1 \hat{i} + a_2 \hat{j}$ 。

2D 向量的标准化

$$\hat{A} = \frac{1}{\|A\|} A = \left[\frac{a_1}{\|A\|} \quad \frac{a_2}{\|A\|} \right]$$

其中向量 $A = [a_1 \ a_2]$ 。

3D 向量的标准化

$$\hat{A} = \frac{1}{\|A\|} A = \left[\frac{a_1}{\|A\|} \quad \frac{a_2}{\|A\|} \quad \frac{a_3}{\|A\|} \right]$$

其中向量 $A = [a_1 \ a_2 \ a_3]$ 。

2D 向量的点乘

$$A \cdot B = a_1 b_1 + a_2 b_2$$

其中, A 和 B 分别表示向量 $A = [a_1 \ a_2]$, $B = [b_1 \ b_2]$ 。

3D 向量的点乘

$$A \cdot B = a_1 b_1 + a_2 b_2 + a_3 b_3$$

其中, A 和 B 分别表示向量 $A = [a_1 \ a_2 \ a_3]$, $B = [b_1 \ b_2 \ b_3]$ 。

判断两个向量垂直

如果 $A \cdot B = 0$, 那么 $A \perp B$ 。

正、负点乘

如果 $A \cdot B < 0$ (负数), 那么它们之间的夹角 $\theta > 90^\circ$

如果 $A \cdot B > 0$ (正数), 那么它们之间的夹角 $\theta < 90^\circ$

其中 θ 为向量 A 和 B 之间的夹角。

两个向量之间的夹角

$$A \cdot B = \|A\| \|B\| \cos \theta$$

其中, θ 为向量 A 和 B 之间的夹角。

向量的叉乘

$$\mathbf{A} \times \mathbf{B} = [(a_2 b_3 - a_3 b_2)(a_3 b_1 - a_1 b_3)(a_1 b_2 - a_2 b_1)]$$

其中, 向量 $\mathbf{A} = [a_1, a_2, a_3]$, $\mathbf{B} = [b_1, b_2, b_3]$ 。

垂直向量

向量 $\mathbf{A} \times \mathbf{B}$ 同时垂直于向量 $\mathbf{A}$ 和向量 $\mathbf{B}$ 。

向量叉乘的交换率不成立

$$\mathbf{A} \times \mathbf{B} \neq \mathbf{B} \times \mathbf{A}$$

事实上, $\mathbf{A} \times \mathbf{B} = -(\mathbf{B} \times \mathbf{A})$, 这对于任意 3D 向量 $\mathbf{A}$ 和 $\mathbf{B}$ 均成立。

平面的一般式

$$\text{平面向量} = (\mathbf{A} \hat{\times} \mathbf{B}) = \frac{\mathbf{A} \times \mathbf{B}}{\|\mathbf{A} \times \mathbf{B}\|}$$

这个公式对于任意的 3D 向量 $\mathbf{A}$ 和 $\mathbf{B}$ 都成立。

两向量之间的夹角

$$\|\mathbf{A} \times \mathbf{B}\| = \|\mathbf{A}\| \|\mathbf{B}\| \sin\theta$$

这个公式对于任意的 3D 向量 $\mathbf{A}$ 和 $\mathbf{B}$ 都成立。

第 5 章 矩阵运算

矩阵相等

两个矩阵相等的条件是:

- 它们的元素个数相等
- 所有元素完全一样

矩阵加法

如果两个矩阵的行和列数都相同, 那么只要将它们的所有元素对应相加即可。

矩阵减法

如果两个矩阵的行和列数都相同, 那么只要将它们的所有元素对应相减即可。

常数和矩阵的乘积

将该常数乘以矩阵中所有元素即可。

$$cA = \begin{bmatrix} ca_{00} & ca_{01} & \cdots \\ ca_{10} & \ddots & \\ \vdots & & ca_{nn} \end{bmatrix}$$

对于任意矩阵 A 和常数 c 均成立。

两个 2×2 矩阵相乘

$$AB = \begin{bmatrix} (a_{00}b_{00} + a_{01}b_{10}) & (a_{00}b_{01} + a_{01}b_{11}) \\ (a_{10}b_{00} + a_{11}b_{10}) & (a_{10}b_{01} + a_{11}b_{11}) \end{bmatrix}$$

对于任意矩阵 $A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ 和 $B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$ 该运算法则均成立。

如何定义矩阵的乘积?

对于乘积 AB , 要求矩阵 A 的列数与矩阵 B 的行数相等。

矩阵乘积的大小

如果 AB 是确定的, 那么矩阵 AB 的大小可以从矩阵 A 的行数与矩阵 B 的列数得到。

矩阵乘积的交换率不成立

$$AB \neq BA$$

其中 A 、 B 为任意矩阵。

3 阶转置矩阵

如果矩阵

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

那么它的转置矩阵

$$A^T = \begin{bmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{bmatrix}$$

置换矩阵

对于任意矩阵 A , 都可以找到一个元素 a_{mn} , 并且使得元素存在于其转置矩阵 A^T 中。

第 6 章 变换

2D 加法变换

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix}$$

其中, dx 为 x 方向上的变化率, dy 为 y 方向上的变化率。

3D 加法变换

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$$

其中, dx 为 x 方向上的变化率, dy 为 y 方向上的变化率, dz 为 z 方向上的变化率。

2D 乘法变换

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中, dx 为 x 方向上的变化率, dy 为 y 方向上的变化率。

3D 乘法变换

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中, dx 为 x 方向上的变化率, dy 为 y 方向上的变化率, dz 为 z 方向上的变化率。

2D 缩放变换

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中, Sx 为 x 方向上的缩放因子, Sy 为 y 方向上的缩放因子。

3D 缩放变换

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中, Sx 为 x 方向上的缩放因子, Sy 为 y 方向上的缩放因子, Sz 为 z 方向上的缩放

因子。

2D 旋转变换

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

其中, θ 为旋转角。

3D 旋转变换(z 轴)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中, θ 为旋转角。

3D 旋转变换(x 轴)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中, θ 为旋转角。

3D 旋转变换(y 轴)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

其中, θ 为旋转角。

2D 组合变换

对于任意一个 2D 组合矩阵:

$$\begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

都可以找到一个缩放因子或者旋转因子(转角 θ) r , 并且还可以找到一个平移因子 t 。

3D 组合变换

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

可以找到一个缩放因子或者旋转因子(转角 θ) r , 并且还可以找到一个平移因子 t 。

矩阵在 OpenGL 和 DirectX 中的变换

对于任意矩阵 A 和 B 以及它们的转置矩阵 A^T 和 B^T , 使得:

$$AB = B^T A^T$$

具体方法如下:

$$\begin{matrix} A & B & A^T \\ \begin{bmatrix} r_{00} & r_{01} & t_x \\ r_{10} & r_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} & = & \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} r_{00} & r_{10} & 0 \\ r_{01} & r_{11} & 0 \\ t_x & t_y & 1 \end{bmatrix} \end{matrix}$$

第 7 章 单位换算

公米制

kilo- hecto- deca-**meters** deci- centi- milli- **grams**

表 A-2 单位换算表

长度单位	时间
1 m = 39.37 in = 3.281 ft.	1 s = 1000 ms.
1 in = 2.54 cm.	1 min. = 60 s
1 km = 0.621 mi.	1 hr. = 3600 s
1 mi = 5280 ft. = 1.609 km.	1 year = 365.242 days

计算机存储单位换算

$$1 \text{ byte} = 8 \text{ bits}$$

$$1 \text{ kilobyte} = 1024 \text{ bytes}$$

$$1 \text{ megabyte} = 1024 \text{ kilobytes}$$

$$1 \text{ gigabyte} = 1024 \text{ megabytes}$$

第 8 章 一维空间运动

匀速运动的位移

对于任意固定速度 v , 位移 = 速度 $\times$ 时间 ($D = v \times t$)

帧与帧之间的位移

新位置 = 老位置 + 速度 $\times$ 时间

时间为变换一帧的时间 (通常是 1/30 每秒)

平均速度

$$\bar{v} = \frac{\Delta x}{t} = \frac{x_f - x_i}{t}$$

其中, Δx 为物体在时间间隔 t 内发生的位移。

加速度

$$a = \frac{\Delta v}{\Delta t} = \frac{v_f - v_i}{t_f - t_i}$$

末速度 (公式 1)

$$v_f = v_i + at$$

平均速度 (公式 2)

$$\bar{v} = \frac{v_i + v_f}{2}$$

仅对匀速直线运动成立。

平均位移 (公式 3)

$$\Delta x = 1/2(v_i + v_f)t$$

匀加速位移 (公式 4)

$$\Delta x = v_i t + 1/2at^2$$

匀加速运动的末速度 (公式 5)

$$v_f^2 = v_i^2 + 2a\Delta x$$



第9章 一维空间的运动导数

平均速度

$$\bar{v} = \frac{f(b) - f(a)}{b - a}$$

这个公式是求物体在斜坡上两个不同位置之间运动的平均速度。对于时间的要求是： $a \leq t \leq b$ 。

即时速度

$$v = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

当时间 $t = a$ 时，函数 $f(t)$ 给出了在该时刻物体的位置。因此，即时速度还可以变形为：

$$v = f'(t)$$

对于任意位置，物体的即时速度就是一个关于时间的方程 $f(t)$ 。

平均加速度

$$\bar{a} = \frac{f(b) - f(a)}{b - a}$$

这个公式是求物体做变加速运动时，在两个不同位置之间的平均加速度。时间的要求是： $a \leq t \leq b$ 。

瞬时加速度

$$a = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

函数 $f(t)$ 给出了加速度与时间的变化关系。

瞬时加速度

$$a = v'(t)$$

函数 $v(t)$ 给出了速度与时间的变化关系。

瞬时加速度

$$a = v'(t) = y''(t)$$

其中函数 $v(t)$ 给出了速度与时间的变化关系，函数 $y(t)$ 给出了加速度与时间的变化关系。

第 10 章 二维和三维空间运动

2D 位移

$$\Delta r = r_f - r_i$$

其中, r_i 为初向量, r_f 为末向量。

2D 和 3D 的平均速度

$$\bar{v} = \frac{\Delta r}{t} = \frac{r_f - r_i}{t}$$

对于任意的位移向量 Δr 和时间间隔 t 都成立。

2D 和 3D 空间物体的运动公式

$$v_f = v_i + at$$

$$\Delta x = 1/2(v_f + v_i)t$$

$$\Delta x = v_i t + 1/2at^2$$

对于向量 a , v_f , v_i 和 Δx , 时间间隔 t 都成立。

炮弹的竖直分量

$$a_y = -9.8\text{m/s}^2$$

$$v_{fy} = v_{iy} + at$$

$$\Delta y = v_{iy}t + 1/2a_yt^2$$

炮弹的水平分量

$$a_x = 0\text{m/s}^2$$

$$v_x = \Delta x/t$$

第 11 章 牛顿定律

重量

$$w = mg$$

其中, m 为物体的质量, g 是重量加速度, 地球上物体的重力加速度 $g = -9.8\text{m/s}^2$ 。

牛顿单位

$$1\text{N} = 1\text{kg} \times \text{m/s}^2$$

$$1\text{N} = 0.2248\text{lbs}$$

静摩擦力

$$F_s = -\mu_s N$$

其中, N 为正压力。

动摩擦力

$$F_k = -\mu_k N$$

其中, N 为正压力。

物体受力图

1. 带箭头线段代表力。
2. 从物体的中心出发, 朝适当的方向画带箭头线段。

牛顿第一定律

如果 $F_{\text{net}} = 0$, 那么物体将保持原来的运动状态不变。

牛顿第二定律

$$F_{\text{net}} = ma$$

其中, m 是物体的质量, a 是物体运动的加速度。

牛顿第三定律

对于施予同一物体的任何一个力, 都有一个与之等大、反向的力反作用于施力物体。

第 12 章 能量

功

$$W = F\Delta x$$

其中, F 是物体所受的力, Δx 是在外力 F 作用下产生的位移。

动能

$$KE = 1/2mv^2$$

其中, m 是物体的质量, v 是物体的运动速度。

功能定理

$$W = \Delta KE = KE_f - KE_i$$

或

$$F\Delta x = 1/2mv_f^2 - 1/2mv_i^2$$

也就是说,外力对物体所做的功等于物体动能的变化量。

重力势能

$$\text{GPE} = mgy$$

其中, m 是物体的质量, g 是重力加速度, y 是物体相对于参考平面所具有的高度。

机械能守恒定律(一般方程)

$$KE_i + PE_i = KE_f + PE_f$$

或

$$\frac{1}{2}mv_i^2 + mgy_i + \frac{1}{2}mv_f^2 + mgy_f$$

机械能守恒定律(改进型)

$$KE_i + PE_i = KE_f + PE_f + E_o$$

或

$$\frac{1}{2}mv_i^2 + mgy_i + \frac{1}{2}mv_f^2 + mgy_f + E_o$$

其中, E_o 代表其他能,例如热能、光能。

第 13 章 动量和碰撞

轴平行向量反射

如果轴是竖直固定的,那么 $v_f = [-v_{ix} \ v_{iy}]$; 如果轴是水平固定的,那么 $v_f = [v_{ix} \ -v_{iy}]$, 所以我们可以得出向心速度 $v_i = [v_{ix} \ v_{iy}]$ 。

轴平行向量反射

$$v_f = 2 \times P + v_i$$

其中, v_i 是入射向量, P 为 $-v_i$ 在边界线法向量上的射影。

动量

$$p = mv$$

其中, m 为物体的质量, v 为物体的运动速度。

冲量

$$\text{冲量} = Ft$$

其中, F 为物体所受的外力, t 表示一个非常小的时间。

冲量定理

物体的冲量 $= Ft = (\Delta p)$

其中, F 为外力, t 表示一个非常小的时间, p 表示动量变化。

动量守恒定律

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f}$$

其中, 下标 1 和下标 2 分别代表两个物体的有关参数。

线性碰撞

$$(v_{1f} - v_{2f}) = -\varepsilon(v_{1i} - v_{2i})$$

其中, ε 是摩擦系数, 且 $0 < \varepsilon < 1$ 。

第 14 章 旋转运动**角位移**

$$\theta = s/r$$

其中, s 为弧长, r 为半径。

平均角速度

$$\bar{\omega} = \frac{\Delta\theta}{\Delta t} = \frac{\theta_f - \theta_i}{t_f - t_i}$$

其中, ω 是角位移, t 是运动时间。

平均角加速度

$$\bar{\alpha} = \frac{\Delta\omega}{\Delta t} = \frac{\omega_f - \omega_i}{t_f - t_i}$$

其中, ω 是角速度, t 是运动时间。

线速度

$$v_t = \omega r$$

其中, ω 是角速度, r 是半径。

角加速度

$$a_t = \alpha r$$

其中, α 是瞬间角速度, r 是半径。



附录 B 推荐资源

数学

Dolciani, Mary P. et al. *Algebra 2 and Trigonometry*. Boston: Houghton Mifflin, 1992.

Finney, Ross L. et al. *Calculus: Graphical, Numerical, Algebraic*. Reading: Addison-Wesley, 1995.

Lecky-Thompson, Guy. *Infinite Game Universe: Mathematical Techniques*. Rockland: Charles River Media, 2001.

Leduc, Steven A. *Linear Algebra*. Lincoln: Cliffs Notes, 1996.

Stewart, James. *Calculus*. Pacific Grove: Brooks/Cole, 2002.

Strang, Gilbert. *Introduction to Linear Algebra*. Wellesley: Wellesly-Cambridge Press, 1993.

Thompson, Silvanus P. and Martin Gardner. *Calculus Made Easy*. New York: St. Martin's Press, 1998 (original copyright 1910).

物理

Bourg, David M. *Physics for Game Developers*. Cambridge: O'Reilly, 2002.

Gonick, Larry and Art Huffman. *The Cartoon Guide to Physics*. New York: HarperPerennial, 1990.

Huetinck, Linda. *Physics*. Lincoln: Cliffs Notes, 2001.

Serway, Raymond A. and Jerry S. Faughn. *College Physics*. Philadelphia: Saunders College Publishing, 2003.

3D 游戏引擎

- DeLoura, Mark, ed. *Game Programming Gems*. Rockland: Charles River Media, 2000.
- DeLoura, Mark, ed. *Game Programming Gems 2*. Rockland: Charles River Media, 2001.
- Dunn, Fletcher and Ian Parberry. *3D Math Primer for Graphics & Game Development*. Wordware, 2002.
- Eberly, David H. *3D Game Engine Design*. San Francisco: Morgan Kaufmann Publishers, 2001.
- Foley, J.D. and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Reading: Addison-Wesley, 1982.
- LaMothe, Andre. *Tricks of the Windows Game Programming Gurus*. Indianapolis: Sams, 1999.
- Lengyel, Eric. *Mathematics for 3D Game Programming & Computer Graphics*. Rockland: Charles River Media, 2001.
- Watt, Alan and Fabio Policarpo. *3D Games: Real-time Rendering and Software Technology*. Reading: Addison-Wesley, 2001.

通用编程技术

- Eckel, Bruce. *Thinking in C++, Volume I: Introduction to Standard C++, Second Edition*. Upper Saddle River: Prentice Hall, 2000.
- Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Reading: Addison-Wesley, 2001.
- Prata, Stephen. *C++ Primer Plus, Third Edition*. Indianapolis: Sams, 1998.

数字资源
PDG

附录 C 自我测试答案

第 1 章 点和直线

1.1 点的定义

1. $A(2, 1)$, $B(-1, -3)$, $C(-3, 0)$, $D(0, 3)$, $E(3, -3)$
2. 答案见图 1-21。
3. $(-2, 1, -4)$
4. $(2, 3, 2)$

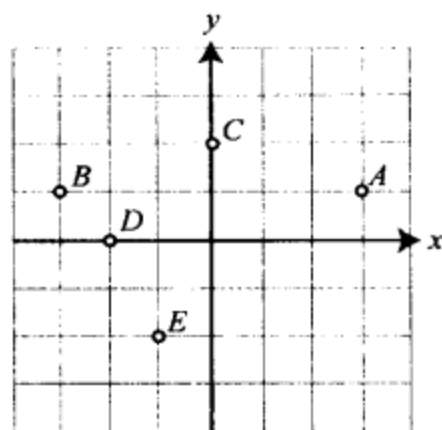


图 1-21 A 点到 E 点在图中的位置

1.2 线的定义

1. 是
2. 是
3. 是
4. 否
5. 答案见图 1-22。
6. 答案见图 1-23。
7. 答案见图 1-24。

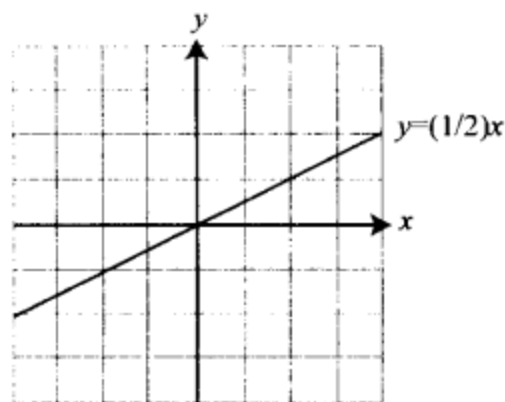


图 1-22 $x - 2y = 0$ 的图像

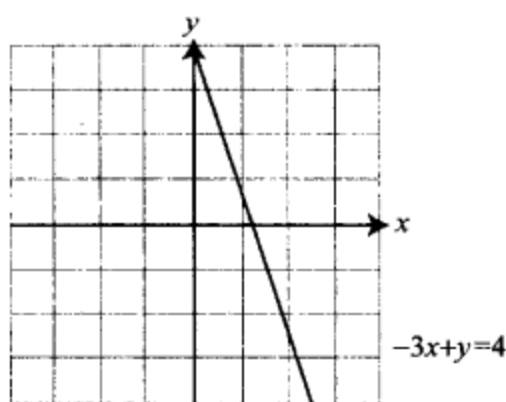


图 1-23 $-3x + y = 4$ 的图像

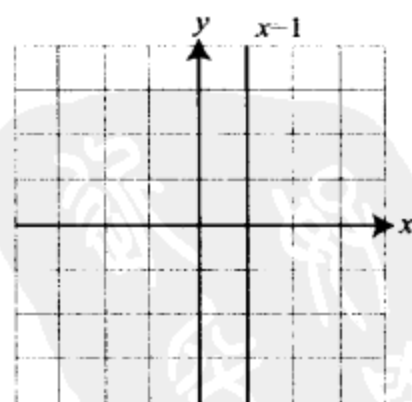


图 1-24 $x = 1$ 的图像

1.3 线的属性

1. $m = -2$

2. $m = -2$
3. $m = 1/2$
4. $m = 3$
5. $m = 3/2$
6. $m =$ 未定义
7. $m = 1/2$
8. $m = 0$
9. 它们是平行的。
10. 垂直的
11. 水平的
12. $m = -2/3$
13. $m = 1/5$
14. $m = 0$
15. $m = -1$
16. $(y - 10) = -2(x)$ 或 $y = -2x + 10$
17. $(y - 5) = -2(x - 3)$ 或 $y = -2x + 11$
18. $(y + 1) = 1/2(x - 2)$ 或 $y = 1/2x - 2$
19. $(y - 4) = 3(x - 1)$ 或 $y = 3x + 1$
20. 点 $(2, 0, -1)$ 和向量 $(1, 4, 6)$
21. 点 $(-3, 1, 5)$ 和向量 $(3, 7, -7)$

1.4 在相交检测中的应用

1. $m_1 = -1, b_1 = 7, m_2 = 1/3, b_2 = 2/3$; 唯一解
2. $m_1 = 1/3, b_1 = 2, m_2 = 1/3, b_2 = -2$; 零解
3. $m_1 = -1/4, b_1 = 2, m_2 = -1/4, b_2 = 2$; 有限个解
4. $m_1 = 1, b_1 = -5, m_2 = -5, b_2 = 0$; 唯一解
5. $m_1 = -1/3, b_1 = -1/3, m_2 = -1/3, b_2 = -5/6$; 零解
6. $m_1 = 3/5, b_1 = 8/5, m_2 = -3/5, b_2 = 8/5$; 唯一解
7. $(2, 1)$
8. $(6, 0)$
9. $(0, 3)$
10. $(5, -4)$

第2章 相关几何知识

2.1 两点间的距离

1. $b = 18$

2. No
3. 130
4. $\sqrt{12200} \approx 110.45$
5. (90, 105)
6. (60, 85, 25)

2.2 抛物线

1. (4, 7)
2. (1, 5)
3. (-3, 2)
4. (-12, 0)
5. (-1, 0)
6. 答案见图 2-17。
7. 答案见图 2-18。

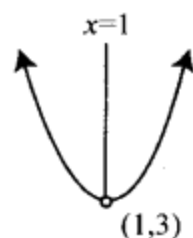


图 2-17 $y = 10(x-1)^2 + 3$

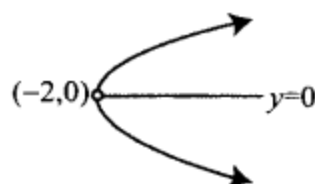


图 2-18 $x = y^2 - 2$

2.3 圆和球

1. center = (30, 10), radius = 20
2. center = (-20, 90), radius = 10
3. center = (-50, 0), radius = 25
4. $(x-40)^2 + (y+25)^2 = 900$
5. $x^2 + y^2 = 225$
6. $(x+10)^2 + (y-40)^2 = 1000$
7. center = (10, 30, 50), radius = 40
8. center = (0, 0, 0), radius = 10
9. center = (50, 0, -40), radius = 1
10. $(x-40)^2 + (y+25)^2 + (z-30)^2 = 100$
11. $x^2 + y^2 + z^2 = 484$
12. $(x-10)^2 + y^2 + (z+60)^2 = 3400$

2.4 在碰撞检测中的应用

1. 是

2. 否

3. 设置一层球, 当检测到大球发生碰撞时检测小球。

第3章 三角函数

3.1 角度和弧度

1. 答案见图 3-16。

2. 答案见图 3-17。

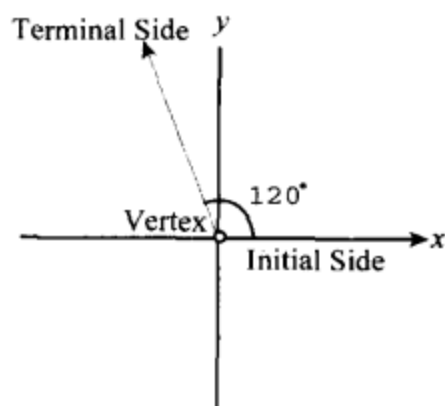


图 3-16 题 1 的答案

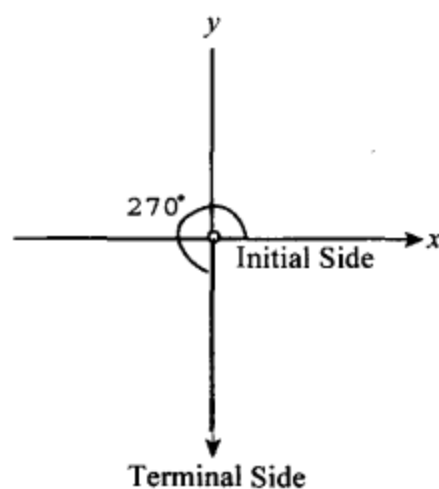


图 3-17 题 2 的答案

3. 答案见图 3-18。

4. $\left(\frac{\pi^R}{3}\right)$

5. $\left(\frac{3\pi^R}{2}\right)$

6. $\left(\frac{\pi^R}{4}\right)$

7. 135°

8. 60°

9. 72°

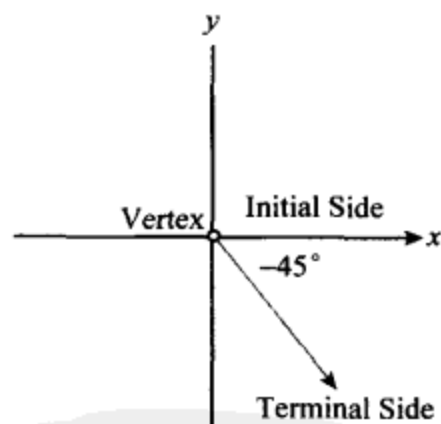


图 3-18 题 3 的答案

3.2 三角函数

1. 0.7071

2. -0.5736

3. -0.7002

4. $\sin\alpha = 7/25 = 0.28$, $\cos\alpha = 24/25 = 0.96$, $\tan\alpha = 7/24 = 0.29$

5. $\alpha = 16.26^\circ$

6. -1.4281

7. 2. 9238
8. 1. 4142
9. per = 120° , amp = 5
10. per = 360° , amp = 3
11. per = 90° , amp = 1
12. per = 360° , amp = 5
13. per = 720° , amp = 2
14. per = 180° , amp = $1/2$

3.3 三角函数的性质

1. $\sin(180^\circ) = 0$ 和 $\cos(180^\circ) = -1$
2. $\sin^2(180^\circ) + \cos^2(180^\circ) = 0^2 + (-1)^2 = 1$, 所以应该是 180° 。
3. $\tan(30^\circ) = \sin(30^\circ)/\cos(30^\circ) = 0.5/0.8660 = 0.5774$
4. $\sin(2\alpha) = \sin(\alpha + \alpha) = \sin\alpha\cos\alpha + \cos\alpha\sin\alpha$
5. $\cos(2\alpha) = \cos\alpha\cos\alpha - \sin\alpha\sin\alpha = \cos^2(\alpha) - \sin^2(\alpha)$
6. 适用正弦函数的和的性质:

$$\sin(90^\circ + 30^\circ) = \sin 90^\circ \cos 30^\circ + \cos 90^\circ \sin 30^\circ = \cos 30^\circ + 0 = 0.8660$$

第4章 向量运算

4.1 向量与标量

1. 标量仅仅是一个数字, 仅表示大小。而向量同时具有大小和方向。
2. 向量
3. 标量
4. +60 英尺
5. -60 英尺
6. +130 像素
7. 470 像素
8. 0

4.2 极坐标与笛卡儿坐标

1. $A = 17.68 \hat{i} + 17.68 \hat{j}$
2. $B = 13\text{units}@22.6^\circ$
3. $C = 100 \hat{i} + 173.21 \hat{j}$
4. $D = 10\text{units}@323^\circ$



4.3 向量加减和减法

1. $A = 3.5 \hat{i} + 19.7 \hat{j}$

2. $B = 10\text{units}@53.1^\circ$

3. $A + B = 9.5 \hat{i} + 27.7 \hat{j}$

4. $A - B = -2.5 \hat{i} + 11.7 \hat{j}$

5. $C + D = 13 \hat{i} + 5 \hat{j}$

6. $F + G = -3 \hat{i} - \hat{j} + 5 \hat{k}$

7. $F - G = 7 \hat{i} - 7 \hat{j} - 7 \hat{k}$

4.4 标量与向量相乘

1. 仅仅有大小

2. $(-1/3)A = -4\text{m/s}@43^\circ$

3. $6B = 18 \hat{i} - 6 \hat{j}$

4. $\hat{C} = \begin{bmatrix} \frac{24}{26} & \frac{10}{26} \end{bmatrix} = \begin{bmatrix} \frac{12}{13} & \frac{5}{13} \end{bmatrix}$

5. $\hat{D} = \begin{bmatrix} \frac{0}{25} & \frac{7}{25} & \frac{24}{25} \end{bmatrix}$

4.5 点乘

1. 0

2. 7

3. 是

4. 74.81°

5. 否

4.6 叉乘

1. $[-12 \ -13 \ -3]$

2. $\begin{bmatrix} \frac{-12}{17.94} & \frac{-13}{17.94} & \frac{-3}{17.94} \end{bmatrix} \approx [-0.67 \ -0.72 \ -0.17]$

3. 44.9°

第5章 矩阵运算

5.1 相等矩阵

1. 3×1

2. 元素在第一行, 第 0 列
3. 4×5
4. 4
5. 否
6. 否

5.2 矩阵的加减运算

1. $C + D = \begin{bmatrix} -3 & 4 \\ -1 & 6 \end{bmatrix}$
2. $C - D = \begin{bmatrix} 7 & 2 \\ -1 & -6 \end{bmatrix}$
3. $F + G = \begin{bmatrix} -2 & 8 \\ 9 & 3 \\ 5 & 6 \end{bmatrix}$
4. $F - G = \begin{bmatrix} -2 & 2 \\ -7 & 5 \\ 9 & -6 \end{bmatrix}$

5.3 矩阵与标量相乘

1. $-3A = \begin{bmatrix} 6 & -9 & -15 \\ 0 & -3 & 12 \\ -30 & 0 & 3 \end{bmatrix}$
2. $1/2B = \begin{bmatrix} 3 & -4 & -1 \\ 0 & 5 & -2 \\ 6 & 0 & -8 \end{bmatrix}$
3. $X = \begin{bmatrix} -36 & 52 & 64 \\ 0 & -8 & -40 \\ 96 & 0 & 20 \end{bmatrix}$

5.4 矩阵相乘

1. $\begin{bmatrix} 7 & 13 \\ 0 & 10 \end{bmatrix}$
2. $\begin{bmatrix} 10 & 0 \\ -5 & 7 \end{bmatrix}$
3. 不确定
4. 不确定

鄧子勉 贈
PDG

$$5. \begin{bmatrix} -4 \\ -4 \\ 36 \end{bmatrix}$$

$$6. \begin{bmatrix} -1 & 5 \\ 21 & 0 \\ -1 & 15 \end{bmatrix}$$

5.5 矩阵转置

$$1. \begin{bmatrix} 7 & 0 \\ 13 & 10 \end{bmatrix}$$

$$2. \begin{bmatrix} 0 & 1 \\ -5 & 4 \end{bmatrix}$$

$$3. \begin{bmatrix} 1 & 2 & 0 \end{bmatrix}$$

$$4. \begin{bmatrix} 2 & 0 & -4 \\ -3 & 1 & 5 \\ 0 & -2 & 10 \end{bmatrix}$$

$$5. \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 3 \\ -1 & 5 & -2 \\ 0 & 1 & 0 \end{bmatrix}$$

$$6. \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & -1 & 0 \end{bmatrix}$$

第6章 变换

6.1 平移

$$1. D'(70, 60)$$

$$2. E'(-10, 180)$$

$$3. F'(140, -20)$$

$$4. G'(250, -330, -50), H'(140, -300, -40), J'(-350, -250, -100)$$

$$5. L'(-300, 50), M'(-150, -60), N'(-130, 20)$$

$$6. \text{是}, G'(250, -330, -50), H'(140, -300, -40), J'(-350, -250, -100)$$

6.2 缩放

$$1. A'(-25, 10), B'(-5, 10), C'(-15, 20)$$

$$2. A'(-12.5, 60), B'(-2.5, 60), C'(-7.5, 120)$$

3. $D'(0, 300, -1000)$, $E'(-500, 1000, -200)$, $F'(-200, 0, -3000)$

4. $D'(0, 30, -50)$, $E'(-100, 100, -10)$, $F'(-40, 0, -150)$

6.3 旋转

1. $A'(-33.66, 38.30)$, $B'(69.28, 40)$, $C'(-93.3, 61.6)$

2. $D'(-0.8725, 20, -49.9925)$, $E'(-9.6495, 0, 20.1715)$, $F'(29.9955, 20, -0.5235)$

3. $G'(0, 91.91, -49.49)$, $H'(-50, 84.84, 56.56)$, $J'(-20, 212.13, -212.13)$

4. $L'(-30, 0, -100)$, $M'(-100, -50, -20)$, $N'(0, -20, -300)$

6.4 组合变换

$$\begin{bmatrix} 0 & 1 & -40 \\ -1 & 0 & 60 \\ 0 & 0 & 1 \end{bmatrix}$$

2. $A'(10, 70)$, $B'(-40, 30)$, $C'(60, 110)$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & (-y_c + x_c) \\ -1 & 0 & (x_c + y_c) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$[x' \ y' \ 1] = [x \ y \ 1] \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ (-y_c + x_c) & (x_c + y_c) & 1 \end{bmatrix}$$

5. $D'(-141.4, 30, 141.4)$, $E'(-35.4, 150, -35.4)$, $F'(-42.4, 100, 14.1)$

$$[x' \ y' \ z' \ 1] = [x \ y \ z \ 1] \begin{bmatrix} -0.707 & 0 & 0.707 & 0 \\ -0.707 & 0 & -0.707 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

第7章 单位换算

7.1 公米制

1. 3 500 厘米

2. 0.035 千米

3. 0.004 567 千米

4. 0.056 千克

5. 3 000 克

7.2 不同单位制间的转换

1. 4 320 分钟
2. 25.4 厘米
3. 19 308 米
4. 362.025 公里/小时
5. 100.562 5 米/秒
6. 78 936 mi/hr²

7.3 计算机转换

1. 170
2. 109
3. 10110000
4. 11001101
5. 838 860 800 字节
6. 32 768 字节
7. 1.47 兆字节
8. 22.12 千兆字节

第8章 一维空间运动

8.1 速度和速率

1. 16.25 米
2. 185 像素位置处
3. 1.25 米/秒
4. 0 米/秒
5. -150px/s

8.2 加速度

1. 1.12m/s²
2. -3.58m/s²
3. 3.7 秒
4. 8.6 米

8.3 运动方程

1. 1.25m/s²



2. 8 秒
3. -8m/s^2
4. 100 米
5. 2.55 秒
6. 31.9 米

第 9 章 一维空间的运动导数

9.1 速度的图像和导数

1. 当 $3 \leq t \leq 6$ 时, 平均速度 = -44ft/s
当 $3 \leq t \leq 5$ 时, 平均速度 = -28ft/s
当 $3 \leq t \leq 4$ 时, 平均速度 = -12ft/s
2. $v = 2\text{ft/s}$
3. $v = 30\text{ft/s}$
4. $v = 5\text{ft/s}$

9.2 加速度图像和二阶导数

1. 180m/s^2
2. -9.8
3. 60
4. 6

第 10 章 二维和三维空间运动

10.1 使用向量

1. $[40 \ 75] = 85@62^\circ$
2. $[-175 \ -200]$
3. $[30 \ -30 \ -90]$
4. $[-12.5 \ -20]$
5. $[20 \ -30 \ -5]$
6. $[-1.66 \ 0.54]$
7. $[109.15 \ 81.70]$
8. $[50 \ 12.5 \ 25]$
9. $[20 \ 5 \ 10]$

10.2 抛物运动

1. 边缘以下 25.08 米



2. 与桌子相距 2.77 米
3. 与建筑物相距 31.44 米
4. 墙上 3.26 米的位置
5. 球场上 19.18 米的位置

第 11 章 牛顿定律

11.1 力

1. -176.24 磅
2. 4.54 千克
3. 111.2N
4. 460.45N
5. $F_s = -\mu_s N = -0.3(460.45) = -138.13\text{N}$ 和 $F_k = -\mu_k N = -0.25(460.45) = -115.11\text{N}$
6. $F_{\text{net}} = [369.7 \ 0]$
7. 答案见图 11-9。
8. $F_{\text{net}} = [74.7 \ 0]$

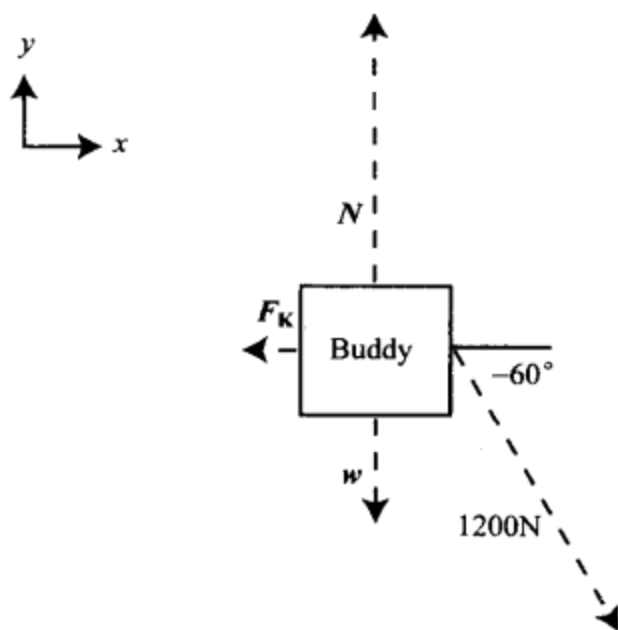


图 11-9 题 7 的受力分析图

11.2 牛顿定律

1. 在相同的位置
2. $5\text{m/s}@45^\circ$
3. $a = [12.32 \ 0]$
4. 154m

$$5. a = [6.05 \quad -10.48]$$

$$6. \Delta r = [27.22 \quad -47.15]$$

第 12 章 能量

12.1 功和动能

$$1. 27\,500\,000\text{J}$$

$$2. 75\text{J}$$

$$3. 1.732\text{m/s}$$

$$4. 906\,408\text{J}$$

$$5. 90\text{J}$$

$$6. 180\text{N}$$

12.2 势能和能量守恒定律

$$1. 147\,000\text{J}$$

$$2. 76.68\text{m/s}$$

$$3. 20.4\text{m}$$

$$4. 14.14\text{m/s}$$

第 13 章 动量和碰撞

13.1 有关静止物体的碰撞

$$1. [-30 \quad -50]$$

$$2. [41.6 \quad 71.2]$$

$$3. [38.634 \quad 24.478 \quad 30.132]$$

13.2 向量和冲量

$$1. 680.88\text{kg} \cdot \text{m/s}$$

$$2. \begin{bmatrix} 100\,000 \\ 50\,000 \\ -20\,000 \end{bmatrix}$$

$$3. 44\text{m/s}$$

$$4. 39\text{m/s}$$

$$5. \begin{bmatrix} 22.03 \\ 52.87 \\ -8.81 \end{bmatrix}$$

数字资源
PDG

13.3 碰撞建模

1. $[-44.44 \quad 55.56 \quad -22.22]$
2. $[-13.636 \quad 12.727 \quad -29.091]$
3. $v_{1f} = [-34.455 \quad 29.091 \quad -23.636]$, $v_{2f} = [4.545 \quad -0.909 \quad -33.636]$

第14章 旋转运动

14.1 圆周运动

1. $(2/3)\pi\text{rad/s}$
2. 5rad/s^2
3. 5s
4. 37.5rad
5. 37.5m/s

14.2 旋转力学

1. $18\,750\text{N}\cdot\text{m}$
2. 0.75rad/s^2

蘇子卿

PDG

附录 D 光盘内容

随书 CD 光盘包含了所有的演示程序文件和可执行文件，它能够帮助你更好地学习本书内容，下面大致介绍一下光盘上的具体内容。

请阅读光盘根目录下的 ReadMe.txt 文件，获得使用光盘的更多信息，包括运行程序的最低机器配置、安装信息、技术支持等。

技术支持

如果你对光盘内容有何异议，请访问我们的网站：

www.prenhall.com/gamedesdev

装载 CD 文件

请将光盘插入电脑的光驱内，如果你的机器设置为光盘自动运行，那么当你第一次插入光盘时会自动安装程序，你也可以将光盘内容复制到硬盘上，这样当不插入光盘时仍能使用它们。

注意

该 CD-ROM 采用了较长的、大小混合形式的文件名，因此能确保您的 CD 光驱在保护模式下能正常运行。

练习文件

该光盘包含了本书所有的练习文件，它们可以在根目录下的对应文件夹中找到。

第三方程序

该光盘还包含了来自著名公司的第三方程序和演示，它们都经过精心的挑拣，以用于巩固你的专业技能。

请注意，本光盘上的一些程序可能会是试用版本，请在使用 30 天后进行注册，以支持这些程序的提供者。请仔细查看软件的说明文件，弄清楚如何才能完成注册。

- Circles 演示：演示了两圆移动直到相交的全过程，其中以平面边缘为边界(第 2 章)。
- Matrix Vortex 演示：它是一个交互性演示程序，直观地演示了向量以及矩阵的不同运算规则(第 4 章)。

- Transformations 演示：它可以帮助读者理解对物体进行 2D 变换的原则(第 6 章)。
- 1D Racer 演示：演示了具有恒定加速度的物体在一维空间中的运动(第 8 章)。
- Cannon Physics 演示：它是一个简单的炮弹发射游戏(第 10 章)。
- Basement Billiards 演示：演示了 2D 球体之间的碰撞，同时也说明了许多关于碰撞响应的相关概念(第 13 章)。

术 语 表

加速度 (acceleration) 速度的变化率, 用以表示物体运动速度变化的快慢。

仿射 在仿射变换中, 物体的形状、大小都保持不变。

振幅 用以表示正弦和余弦曲线的峰值。

入射角 一个物体同规定物体发生碰撞前的瞄准角度。

反射角 一个物体同规定物体发生碰撞后被反射飞出时的角度。

角位移 θ 为一个矢量, 用于表示量度旋转中角度的变化, 其值等于弧度除以旋转半径。

角动量 $L = I\omega$, 其中, $I = mr^2$ 。 I 为惯性系数, ω 为角速度。

弧长 s 一段圆形路径的长度。

数组 用于组织任意形式的信息 (比如数字、变量、文本甚至是数组)。

平均角加速度 α 角速度的变化速度, 其值为角速度的变化率除以时间。

平均角速度 $\bar{\omega}$ 角位移变化的速率 (其值为角度变化量除以时间)。

轴对齐包围盒 盒的边界线 (长和宽) 完全平行或垂直于 x 或 y 轴。

对称轴 一条水平或垂直直线, 它经过抛物线的顶点时恰好把其分为完全相同的两半。

笛卡儿坐标 x 、 y 、 z 轴代表物体的位移分量分别平行于这三条轴。

笛卡儿坐标系 一条平行的 x 轴和一条垂直的 y 轴组成一个平面, 这个平面上的任意一点均可用 (x, y) 序列唯一表示, 如果加上 z 轴, 则可以表示三维空间点的位置 (x, y, z) 。

圆 到一个定点的距离相等的所有二维空间的点的集合, 这一距离称为半径。

类 面向对象设计中的基本类型。

还原系数 ε 用于测量碰撞时的能量消耗。例如, 弹性碰撞的 $\varepsilon = 1$, 一次完全非弹性碰撞的 $\varepsilon = 0$ 。

通解 所有点皆符合给定的方程式。

关联 指将多个矩阵组合, 并转化为一个新矩阵的过程。

动量守恒定律 总动量在碰撞前后保持不变, 只是从一方转移到另一方。

同构矩阵 任何具有相同行和列的矩阵。

非等比缩放 在缩放变换中对 x 、 y 方向采用不同的缩放因子 S_x 、 S_y 。

位移 一个表示距离的矢量, 即距离大小及方向。

弹性碰撞 指两个物体碰撞时, 无动能损耗; 非弹性碰撞则有动能损耗。

元素 矩阵中存放的每个数据。

欧拉旋转 通常使用三个变量 roll、pitch 和 yaw 表示物体分别沿 3 个坐标轴进行旋转的过程, 它不同于另外一种旋转方法——四元数。

公式 有一些参数和常数组成的表示各参数之间换算关系的一个方程式。

受力分析图 一种标准的示意图, 用箭头表示施加在物体上的有一定受力方向的力, 所有箭头都源于物体中心, 箭头长度和大小对应。

函数 根据某种规则输入一些信息而得到另一信息或结果的方法。

周期 360° 或 $2\pi^R$, 它表示正弦和余弦曲线的一个频率区间。

重力势能 GPE 由于物体相对于参考位置具有一定高度而具有的能量。

冲量定理 冲量是指在很短的时间内力作用于物体,以至于可以把力看作一个常数。力迅速地使物体状态改变导致了物体动能的变化。也就是说,如果质量不变,物体要么加速运动,要么做减速运动。

唯一解 仅有一个坐标值满足于某个方程,这个坐标值就是该方程的唯一解。

惯性 I 描述物体保持原运动状态的特性,它只与物体的质量有关,其值是 $I = mr^2$ 。

瞬时角加速度 指在尽可能短的时间内的平均角加速度。

瞬时角速度 指在尽可能短的时间内的平均角速度。

瞬时速度 指在尽可能短的时间内的平均速度。

插值 指在两个预定点之间放入第三个值。

关键帧 在动画或 flash 影片剪辑制作中,指那些起过渡作用的重要的帧。

动能 物体因为运动而具有的能量。

动摩擦力 物体开始运动后,使物体慢下来的力。

机械能守恒定律 能量可以转化,但不能产生也不会消失。

线性方程组 利用等式的规律,根据两个给定的等式求解两个未知量。

矩阵 仅限于存储数字的数组。

中点 两点之间等距离的位置。

动量 p 其值等于物体质量乘以速度。

牛顿第二定律推论 牛顿第二定律的旋转形式: $F = ma$ (线性) 和 转矩 $= Ia$ (rotational). ($\tau = I\alpha$), 其中, I 指物体具有的惯性, 即 $I = mr^2$, α 为角度。

压力 物体由于受到重力作用而施加于支撑它的物体表面的一种承受力。

单位化 缩小一个矢量的模化为 1 的过程。

直交 垂直的同义词。

抛物线 一种对称的凸起曲线或弧线。

平行线 两条永不相交的且拥有相同斜率的直线。

完全非弹性碰撞 指两个物体碰撞之后,不再分开并以相同的速度运动。

周期 用以度量三角函数曲线重复的频率。

垂直线 两条直线相交成 90° 。

极坐标 坐标平面上的点用标准长度和标准角度表示。

抛射体 任何被抛出、踢出或射击出去的且在空中运行的物体。

勾股定理 对于任意直角三角形,斜边的平方等于两直角边的平方和 ($a^2 + b^2 = c^2$)。

象限 笛卡儿坐标系中的一个区域。

半径 圆或球上任意一点到圆心或球心的距离。

旋转势能 $KE_R = 1/2 I \omega^2$, 其中 $I = mr^2$, ω 为角加速度。

标量 只有大小,没有方向的物理量。

坡度 形容斜坡陡峭的程度。

解集 符合两个方程的所有点的集合。

速率 度量物体运动快慢程度的物理量。

球体 在三维空间中,到定点的距离等于定长的所有点的集合,定点称为圆心,定长为半径。也可由一个半圆围绕其圆心旋转得到。

静摩擦力 在物体将动未动之际,阻止物体运动的力。

结构体 由用户定义的编程结构,用于定义新的数据。

替代法 用来解两个线性方程组的方法之一。

表面张力 一个垂直于物体表面的矢量, 长度为 1。

线性方程组 把两个方程放在一起求解。

切向速度 v_t 旋转物体边缘点的线速度。

转化 在 2D 或 3D 空间移动物体的运动方式, 包括前后移动或上下移动, 将物体放大或缩小, 以及绕一个点或一条定轴旋转等方式。

变换 将物体做前后、左右、上下、内外移动的运动方式。

转置 矩阵里的一种操作, 例如以一个竖行来代替横行。

约束比例 当缩小或放大物体时, 大部分比例依然保持不变(例如角度)。

单位圆 其半径大小为 1, 圆心位于坐标原点的圆。

矢量 既有大小又有方向的物理量。

向量反射 移动物体在碰撞到某一固定物体后的反方向运动。

速度 一种矢量性质的速率, 即速率加上方向。

顶点 一个点, 例如在抛物线中, 顶点就是弧的顶端或底端, 最大值或最小值。

重量 由于地球引力而作用于物体的力。

功 一个标量, 其大小可以由力乘以该力作用下物体的位移求得。

功能原理 外力对物体所做的总功, 等于物体动能的增加量。

y-截点 一条直线与 y 轴的交点, 此交点到坐标原点的距离叫做截距。

新解
PDG