

# Steve Marschner Peter Shirley

with

Michael Ashikhmin

Michael Gleicher

Naty Hoffman

Garrett Johnson

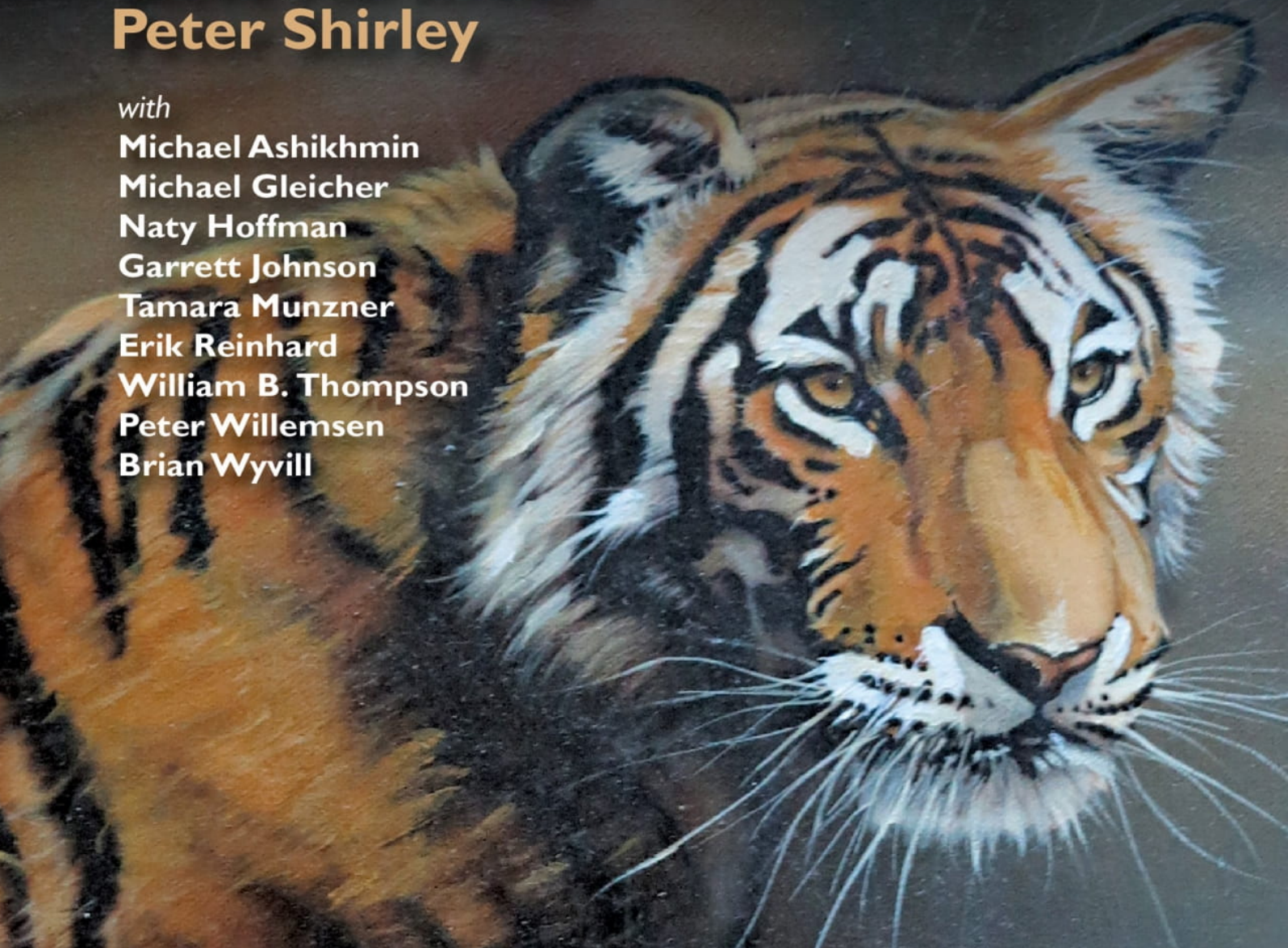
Tamara Munzner

Erik Reinhard

William B. Thompson

Peter Willemssen

Brian Wyvill



## 计算机图形学基础

第五版 中文译本

作者: Steve Marschner & Peter Shirley

组织: Cornell University & NVIDIA

时间: February 2021

版本: Fifth edition

译者: buding, Hugo HU, Ninohana



Stay hungry. Stay foolish. —Steve Jobs

# 前言

本版的《计算机图形学基础》包括对阴影着色、光线反射和路径追踪等材料的大量重写，以及对全书谬误的许多订正。本书对基于物理的材料和基于物理的渲染等技术进行了更好的介绍，这些技术在实际应用中逐渐占据主导地位。现在这些材料得到了更好的整合，我们认为这本书很好地匹配了目前许多教师组织图形学课程的方式。

本书的组织结构与第四版基本相似。在多年来对本书进行修订的过程中，我们努力保留了早期版本中非正式的、直观的表述风格，同时也提高了本书的一致性、准确性和完整性。我们希望读者能够发现，本书是一个吸引人的平台，适合于各种计算机图形学课程。

## 关于封面

封面图片来自 J.W.Baker 的《水中之虎》（画布上的拉丝和喷枪亚克力，16 英寸 x20 英寸），[www.jwbart.com](http://www.jwbart.com)。

老虎的主题是指 Alain Fournier（1943-2000）1998 年在康奈尔大学的一次研讨会上的精彩演讲。他的演讲是对老虎动作进行的令人回味的口头描述。他总结了自己的观点：

尽管在过去的 35 年里，计算机图形学的建模和渲染已经有了巨大的进展，但我们仍然无法自动模拟在河中游泳的老虎所有精彩细节。我所说的自动是指不需要艺术家或专家进行仔细的手动调整的方式。

坏消息是，我们还有很长的路没走。

好消息是，我们还有很长的路要走。

## 线上资源

本书的网址是 <http://www.cs.cornell.edu/~srm/fcg5/>。我们将继续维护本书的勘误表和课程链接，以及与本书风格相符的教学材料。本书中的大多数图片都是 Adobe Illustrator 格式的，我们很乐意根据需要特定图片转换为可移植格式。请随时通过 [srm@cs.cornell.edu](mailto:srm@cs.cornell.edu) 或 [ptrshrl@gmail.com](mailto:ptrshrl@gmail.com) 与我们联系。

## 致谢

以下人士提供了对于本书各版本的有用信息、评论或者反馈：Ahmet O˘guz Aky˘uz, Josh Andersen, Beatriz Trinch˘ao Andrade Zeferino Andrade, Bagossy Attila, Kavita Bala, Mick Beaver, Robert Belleman, Adam Berger, Adeel Bhutta, Solomon Boulos, Stephen Chenney, Michael Coblenz, Greg Coombe, Frederic Cremer, Brian Curtin, Dave Edwards, Jonathon Evans, Karen Feinauer, Claude Fuhrer, Yotam Gingold, Amy Gooch, Eungyoung Han, Chuck Hansen, Andy Hanson, Razen Al Harbi, Dave Hart, John Hart, Yong Huang, John “Spike” Hughes, Helen Hu, Vicki Interrante, Wenzel Jakob, Doug James, Henrik Wann Jensen, Shi Jin, Mark Johnson, Ray Jones, Revant Kapoor, Kristin Kerr, Erum Arif Khan, Mark Kilgard, Fangjun Kuang, Dylan Lacewell, Mathias Lang, Philippe Laval, Joshua Levine, Marc Levoy, Howard Lo, Joann Luu, Mauricio Maurer, Andrew Medlin, Ron Metoyer, Keith Morley, Eric Mortensen, Koji Nakamaru, Micah Neilson, Blake Nelson, Michael Nikelsky, James O’Brien, Hongshu Pan, Steve Parker, Sumanta Pattanaik, Matt Pharr, Ken Phillis Jr, Nicol˘o Pinciroli, Peter Poulos, Shaun Ramsey, Rich Riesenfeld, Nate Robins, Nan Schaller, Chris Schryvers, Tom Sederberg, Richard Sharp, Sarah Shirley, Peter-Pike Sloan, Hannah Story, Tony Tahbaz, Jan-Phillip Tiesel, Bruce Walter, Alex Williams, Amy Williams, Chris Wyman, Kate Zebrose, and Angela Zhang。

Ching-Kuang Shene 和 David Solomon 允许我们借用他们的例子。Henrik Wann Jensen、Eric Levin、Matt Pharr 和 Jason Waltman 慷慨地提供了图片。Brandon Mansfield 帮助改进了关于光线追踪的分层包围体的探讨。Philip Greenspun ([philip.greenspun.com](http://philip.greenspun.com)) 热心地允许我们使用他的照片。John “Spike” Hughes 帮助改进了对抽样理论的探讨。Wenzel Jakob 的 Mitsuba 渲染器在创建许多图形方面非常宝贵。我们非常感谢 J.W. Baker 帮助创作了 Pete 设想的封面。他除了是一位才华横溢的艺术家之外，也是一位非常愉快的工作伙伴。

本书的章节注释中引用了许对编写本书有帮助的著作。然而，有几本影响了本书内容和表现形式的关键文献值得在此特别表彰。其中包括两本经典的计算机图形学教材，我们都是从这两本教材中学习的基础知识——《计算机图形学：原理与实践》(Foley、Van Dam、Feiner 和 Hughes, 1990 年) 和《计算机图形学》(Hearn 和 Baker, 1986 年)。其他文本包括 Alan Watt 的两本有影响力的书籍 (Watt, 1993, 1991), Hill 的《使用 OpenGL 的计算机图形》(Francis S. Hill, 2000), Angel 的《交互式计算机图形学：使用 OpenGL 的自上而下方法》(Angel, 2002), Hugues Hoppe 的华盛顿大学论文 (Hoppe, 1994) 和 Rogers 的两篇优秀的图形学文章 (D. F. Rogers, 1985, 1989)。

我们要特别感谢 Alice 和 Klaus Peters 鼓励 Peter 撰写本书的第一版，感谢他们在帮助完成本书方面的伟大才能。他们对作者的耐心以及竭尽所能奉献于使本书成为最好的书籍对本书的出版起了重要的作用。如果没有他们的非凡努力，这本书肯定不存在。

Steve Marschner, 伊萨卡, 纽约

Peter Shirley, 盐湖城, 犹他州

2021 年 2 月



## 作者

Steve Marschner 是康奈尔大学的计算机科学教授。他于 1993 年在布朗大学获得理学学士学位，1998 年在康奈尔大学获得博士学位。在 2002 年加入康奈尔大学之前，他在微软研究院和斯坦福大学担任研究职务。他是 2015 年 SIGGRAPH 计算机图形学成就奖的获得者和 2003 年技术学院奖的共同获得者。

Peter Shirley 是英伟达公司的杰出研究科学家。他曾在印第安纳大学、康奈尔大学和犹他大学担任学术职务。他于 1985 年获得里德学院的物理学学士学位，1991 年获得伊利诺伊大学的计算机科学博士学位。

# 目录


前言	i
致谢	ii
作者	iii
<b>第 1 章 图形学介绍</b>	<b>1</b>
1.1 图形学领域	1
1.2 主要应用	2
1.3 图形 API	2
1.4 图形管道	3
1.5 数值问题	3
1.6 效率	5
1.7 图形程序设计和编码	6
<b>第 2 章 基础数学知识</b>	<b>10</b>
2.1 集合和映射	10
2.2 求解二次方程	14
2.3 三角函数	15
2.4 向量	20
2.5 积分	22
2.6 密度函数	22
2.7 曲线和曲面	22
2.8 线性插值	22
2.9 三角形	22
2.10 离散概率	22
2.11 连续概率	22
2.12 蒙特卡洛积分	22
<b>第 3 章 光栅图像</b>	<b>23</b>
3.1 光栅设备	24
3.2 图像、像素和几何	29
3.3 RGB 颜色	33
3.4 阿尔法合成	34
3.5 FAQ	36

3.6 练习	36
<b>第 4 章 光线追踪</b>	<b>37</b>
4.1 基础光线追踪算法	37
4.2 透视图	37
4.3 计算视线	37
4.4 光线相交	37
4.5 阴影	37
4.6 历史笔记	37
<b>第 5 章 表面着色</b>	<b>38</b>
5.1 点状光源	38
5.2 基本反射模型	38
5.3 环境照明	38
<b>第 6 章 线性代数</b>	<b>39</b>
6.1 行列式	39
6.2 矩阵	41
6.3 计算矩阵和行列式	45
6.4 特征式和矩阵对角式	45

# 第 1 章 图形学介绍

计算机图形学 (*computer graphics*) 这个术语描述了任何使用计算机来创建和操作图像的场景。本书介绍了可用于创建各种图像的算法和数学工具——逼真的视觉效果、丰富的技术插图或精美的计算机动画。图形可以是二维的，也可以是三维的；图像可以是合成的，也可以是通过处理照片产生的。本书是关于基本算法和数学的书，特别是那些用于制作三维物体和场景的合成图像的算法。

实际上，研究计算机图形学不可避免地需要了解特定的硬件和文件格式，通常还需要了解一两个图形 API（参见 1.3 节），而计算机图形学又是一个快速发展的领域，这些知识的具体内容也在不断更新变化。因此在本书中，我们尽量避免依赖任何特定的硬件或 API，同时鼓励读者结合自己软/硬件环境的相关文档来学习本书。幸运的是，计算机图形学领域的术语和概念足够标准，本书的讨论应能很好地反映到大多数环境。

 **笔记** API: 应用程序编程接口

本章定义了一些基本术语，并提供了一些历史背景，以及与计算机图形学相关的信息源。

## 1.1 图形学领域

对任何领域强加分类都是危险的，但大多数图形学从业者都对计算机图形学的以下主要领域达成一致：

- **建模** (*modeling*) 涉及到通过数学将形状和外观属性描述为可存储在计算机中的形式。例如，可以将咖啡杯描述为一组有序的三维点、一些连接这些点的插值规则和一个描述光线如何与杯子相互作用的反射模型。
- **渲染** (*rendering*) 是一个从艺术中继承下来的术语，涉及到从三维计算机模型创建着色图像。
- **动画** (*animation*) 是一种通过图像序列创造运动错觉的技术。动画使用建模和渲染，但增加了随时间移动的关键问题，这通常在基本建模和渲染中不会涉及到。

还有许多其他涉及计算机图形学的领域，至于它们是否属于图形学的核心领域，仁者见仁，智者见智。这些内容至少都会在本书中提及。此类相关领域包括以下内容：

- **用户交互** (*user interaction*) 涉及输入设备（例如鼠标和手写板）、应用程序、对用户的图像反馈以及其他感官反馈之间的接口。从历史上看，该领域与图形有关，主要是因为计算机图形学的研究人员最早接触到了现在无处不在的输入和输出设备。
- **虚拟现实** (*virtual reality*) 试图让用户沉浸在一个三维虚拟世界中。这通常要求至少有立体图形和对头部运动的反应。对于真正的虚拟现实，还应该提供声音和力量的反馈。因为这一领域需要先进的三维图形和先进的显示技术，所以它通常与图形学密切相关。
- **可视化** (*visualization*) 试图通过视觉显示让用户深入了解复杂的信息。通常，在一个可视化问题中有一些图形学问题需要解决。

- 图像处理 (*image processing*) 涉及对二维图像的操作，在图形学和视觉领域均有应用。
- 三维扫描 (*three-dimensional scanning*) 使用测距技术来创建测定的三维模型。这类模型对于创造丰富的视觉图像很有帮助，而处理这种模型往往需要图形算法。
- 计算摄影 (*computational photography*) 使用计算机图形学、计算机视觉和图像处理方法来实现拍摄物体、场景和环境的新方法。

## 1.2 主要应用

几乎任何工作都可以在一定程度上涉及到计算机图形学，但计算机图形学技术的主要消费者包括以下行业：

- 视频游戏 (*video games*) 越来越多地使用复杂的三维模型和渲染算法。
- 卡通片 (*cartoons*) 通常由三维模型直接渲染得到。许多传统的二维卡通片会使用由三维模型渲染的背景，这使得连续移动的视角不必占用艺术家大量的时间。
- 视觉效果 (*visual effects*) 几乎使用了所有类型的计算机图形学技术。几乎每部现代电影都使用数字合成技术来叠加背景与单独拍摄的前景。许多电影还使用三维建模和动画来创造合成环境、物体甚至人物，而大多数观众都不会怀疑这是假的。
- 动画电影 (*animated films*) 使用了许多视觉效果会用到的技术，但不一定要追求图像的真实性。
- CAD/CAM是指计算机辅助设计 (*computer-aided design*) 和计算机辅助制造 (*computer-aided manufacturing*)。这些领域利用计算机技术在计算机上设计零件和产品，然后利用这些虚拟设计来指导制造过程。例如，许多机械零件是在三维计算机建模软件包中设计的，然后在计算机控制的铣削设备上自动生产。
- 仿真 (*simulation*) 可以被视为精确的视频游戏。例如，飞行模拟器使用复杂的三维图形来模拟驾驶飞机的体验。这样的模拟对于安全关键领域的初始培训（例如驾驶汽车）以及对于有经验的用户的场景培训（比如在实际操作中成本太高或太危险的特定灭火情况）都是非常有用的。
- 医学影像 (*medical imaging*) 根据扫描的病人数据创建有意义的图像。例如，计算机断层扫描 (*CT, computed tomography*) 数据集是由包含密度值的大型三维矩阵组成的。计算机图形学被用来创建阴影图像，帮助医生从这些数据中提取最突出的信息。
- 信息可视化 (*information visualization*) 为不一定具有“自然的”视觉描述的数据创建图像。例如，十只不同股票价格的时间趋势没有明显的视觉描述，但巧妙的图形技术可以帮助人类看到这些数据的模式。

## 1.3 图形 API

使用图形库的一个关键部分在于和图形 API (*graphics API*) 打交道。应用程序接口 (*API, application program interface*) 是执行一系列相关操作的标准函数集合，而图形 API 是执行诸



如将图像和三维表面绘制到屏幕上的窗口等基本操作的函数集合。

每个图形程序都需要使用两个相关的 API：一个图形 API 用于视觉输出，一个用户界面 API 用于从用户那里获得输入。目前有两种主流的图形 API 和用户界面 API 模式：第一种是集成的方式，以 Java 为例，其中图形和用户界面工具包是集成的、可移植的包，作为语言的一部分得到完全的标准化和支持。第二种是以 Direct3D 和 OpenGL 为代表的模式，其中绘图命令是与 C++ 等语言相联系的软件库的一部分，而用户界面软件是一个独立的实体，可能因系统而异。在后一种方法中，编写可移植的代码会有点困难，尽管对于简单的程序，可能会使用可移植的库层来封装系统特定的用户界面代码。

无论你选择什么样的 API，基本的图形调用将大致相同，而且本书的概念也适用。

## 1.4 图形管道

今天，每台台式电脑都有一个强大的三维图形管道 (*graphics pipeline*)。这是一个特殊的软/硬件子系统，可以有效地绘制透视的三维图元。这些系统一般都对包含共享顶点的三维三角形的处理过程进行了优化。管道中的基本操作是将三维顶点位置映射到二维屏幕位置，并对三角形进行着色，好让它们看起来足够真实，同时又以适当的从后到前 (*back-to-front*) 的顺序呈现。


尽管如此按照有效的从后到前顺序绘制三角形曾是计算机图形学中最重要研究问题，但现在几乎都是用 *z-buffer* 来解决，它使用一个特殊的内存缓冲区，以蛮力的方式解决问题。

事实证明，图形管道中使用的几何操作几乎完全可以在四维坐标空间中完成，该空间由三个传统的几何坐标和第四个有助于透视的齐次坐标 (*homogeneous coordinate*) 组成。这些四维坐标是用  $4 \times 4$  矩阵和四维向量来操作的。因此，图形管道包含了许多能有效处理、合成这些矩阵和向量的机制。这种四维坐标系是计算机科学中最微妙、最美丽的结构之一，同时也是学习计算机图形学时需要跨越的最大障碍。每本图形学书籍的第一部分都有很大一部分内容是关于这些坐标的。

图像生成的速度很大程度上取决于需要绘制的三角形的数量。由于许多应用中交互性比视觉质量更重要，因此需要尽量减少用于表示模型的三角形数量。此外，如果从远处看模型，需要的三角形数量比从近处看模型时要少。这说明使用不同的细节级别 (*LOD, level of detail*) 来表示一个模型是很有用的。

## 1.5 数值问题

许多图形程序实际上只是和三维相关的数值代码。数值问题在这类程序中往往是至关重要的。在“过去”，要以健壮和可移植的方式处理这些问题是非常困难的，因为机器内部对数值有不同的表示，更糟糕的是，处理异常的方式也不尽相同，互不兼容。幸运的是，几乎所有的现代计算机都遵循 IEEE 浮点数 (*IEEE floating-point*) 标准 (IEEE 标准协会, 1985)。该标准允许程序员对“某些数值条件会如何被处理”这种问题做出许多方便的假设。

 **笔记** IEEE 浮点数运算有两种零的表示方法，一种表示法将零视为正数，另一种则将零视为负数。虽然  $-0$  和  $+0$  之间的区别一般不重要，但以防万一还是应该牢记于心的。

尽管 IEEE 浮点数标准有很多在编写数值算法时很有价值的特性，但就图形学中遇到的大多数情况而言只有少数几个特性是至关重要的。首先，也是最重要的，是要了解 IEEE 浮点标准中实数的三个“特殊”值：

1. 无穷大 ( $\infty$ )：这是一个比其他所有有效数字都要大的有效数字。
2. 负无穷大 ( $-\infty$ )：这是一个比其他所有有效数字都要小的有效数字。
3. 非数字 (NaN)：这是一个由不确定后果的操作产生的无效数字，比如  $0$  除以  $0$ 。

IEEE 浮点数标准的设计者制定了一些对程序员来说非常方便的规定，而其中不少规则都与上述三个处理异常（比如除以零）的特殊值有关。在这些情况下，异常会被记录下来，但在很多场景中，程序员可以忽略它。具体来说，对于任何正实数  $a$ ，以下涉及除以无穷数的规则都是成立的：

$$\begin{aligned} +a/(\infty) &= +0 \\ -a/(\infty) &= -0 \\ +a/(-\infty) &= -0 \\ -a/(-\infty) &= +0 \end{aligned}$$

其他涉及无穷数的运算的表现与人们所期望的一样。同样对于正实数  $a$ ，其行为如下：

$$\begin{aligned} \infty + \infty &= +\infty \\ \infty - \infty &= \text{NaN} \\ \infty \times \infty &= \infty \\ \infty/\infty &= \text{NaN} \\ \infty/a &= \infty \\ \infty/0 &= \infty \\ 0/0 &= \text{NaN} \end{aligned}$$

布尔表达式中涉及无穷数的规则与预期一致：

1. 所有有限的有效数字都小于  $+\infty$ ；
2. 所有有限的有效数字都大于  $-\infty$ ；
3.  $-\infty$  小于  $+\infty$ 。


涉及 NaN 值的表达式规则比较简单：

1. 任何包括 NaN 的算术表达式的结果都是 NaN；
2. 任何涉及 NaN 的布尔表达式都是假的。

也许 IEEE 浮点数标准最有用的地方在于规定了“如何处理除以零的问题”；对于任何正实数  $a$ ，以下涉及除以零值的规则都成立：

$$+a/ + 0 = +\infty$$

$$-a/ + 0 = -\infty$$

 **笔记** 必须留意可能出现负零 (-0) 的情况！

如果程序员能对 IEEE 规则善加利用，那么许多数字计算会变得更加简单。例如，考虑表达式：

$$a = \frac{1}{\frac{1}{b} + \frac{1}{c}}$$

这种表达式会在电阻和透镜相关的问题中出现。如果除以 0 会导致程序崩溃（在 IEEE 浮点数标准出现前的许多系统中都是如此），那么就需要两个 if 语句来检查 b 或 c 的小值或零值。相反，在 IEEE 浮点数标准中，如果 b 或 c 为零，我们会如愿获得 a 的零值。另一种避免特殊检查的常用技术是利用 NaN 的布尔特性。请看下面的代码段：

```
a = f(x)
if(a > 0) then
  do something
```

在这里，函数  $f$  可能会返回“丑陋”的值，如  $\infty$  或 NaN，但  $if$  条件仍然是明确的：当  $a = \text{NaN}$  或  $a = -\infty$  时为假，而  $a = +\infty$  为真。在决定返回哪些值时要小心，通常  $if$  语句可以在不检查的情况下做出正确选择。这使得程序更小、更健壮、更高效。

## 1.6 效率

并没有什么神奇的规则能让代码更加高效。想要高效需要仔细地做权衡，而这些权衡对于不同的架构是不同的。然而在可预见的未来，一个好的启发式方法是，程序员应该更多地关注内存访问模式而不是操作数。这与 20 年前的最佳启发式方法相反，出现这种转变是因为这 20 年间内存的速度没有跟上处理器的速度。由于这一发展趋势仍在继续，有限、连贯的内存访问对优化的重要性应该只会增加。

一个合理的优化代码的方法是按以下顺序进行，只采取那些需要的步骤：

1. 用最直接的方式编写代码。根据需要即时计算中间结果，而不是将其存储起来；
2. 在优化模式下进行编译；
3. 使用现有的任何分析工具来找到关键瓶颈；
4. 检查数据结构以寻找改善局部性的方法。尽可能让数据单元大小与目标架构上的缓存/页面大小相匹配；
5. 如果分析工具表明数值计算存在瓶颈，请检查编译器生成的汇编代码是否存在优化空间。重写源代码以解决发现的任何问题。

这些步骤中最重要的是第一个。大多数所谓的“优化”使代码更难读，却没有加快速度。此


外，前期花在代码优化上的时间通常更适合用来纠正错误或增加功能。另外要注意旧版《计算机图形学基础》中给出的建议；一些经典的技巧（比如使用整数而不是实数）可能不再加快速度，因为现代 CPU 在执行浮点数操作时一般都能像执行整数操作一样快。在所有情况下都应该用分析工具来确定任何针对特定机器、编译器的优化是否真的有意义。


## 1.7 图形程序设计和编码

某些常见的策略在图形编程中往往很有用。在本节中，我们提供了一些建议，当你实现书中学到的方法时，你可能会发现这些建议很有帮助。

### 1.7.1 类型设计

所有图形程序的一个关键部分是为几何实体（如向量和矩阵）以及图形实体（如 RGB 颜色和图像）提供良好的类型或例程，这些例程应该尽可能的简洁和高效。一个普遍的设计问题是，位置和位移是否应该放在不同的类型中，因为它们有不同的操作。例如，一个位置乘以二分之一没有几何意义，而位移的二分之一却有意义（Goldman, 1985; DeRose, 1989）。在这个问题上人们几乎没有一致的意见，还可能会在图形学从业者中引发数小时的激烈争论，但为了举例说明，我们假设不做这种区分。


 **笔记** 我坚信 KISS（“保持简单、愚蠢”，“*keep it simple, stupid*”）原则，从这个角来看，“将位置、位移放在两个类型中”的论点并不令人信服，不足以证明增加的复杂性。-P.S.

 **笔记** 我喜欢把点和向量分开，因为它使代码更易读，并能让编译器捕捉到一些错误。-S.M.

这意味着要编写的一些基本类型包括：

- **二维向量** (*vector2*)。一个用于存储  $x$  和  $y$  分量的二维向量类型，它将分量存储在一个长度为 2 的数组中，这样就可以很好地支持索引操作。你还应该包括向量加法、向量减法、点积、叉积、标量乘法和标量除法的操作。
- **三维向量** (*vector3*)。一个类似于二维向量的三维向量类型。
- **h 向量** (*hvector*)。一个有四个分量的齐次向量（见第 8 章）。
- **RGB**。RGB 颜色类型存储三个分量，此外还应该包括 RGB 加法、RGB 减法、RGB 乘法、标量乘法和标量除法之类的操作。
- **变换** (*transform*)。一个用于变换的  $4 \times 4$  矩阵，还应该包括一个矩阵乘法和成员函数来应用于位置、方向和表面法向量。如第七章所示，这些都是不同的。
- **图像** (*image*)。一个具有输出操作的、由 RGB 像素组成的二维数组。


此外，由你决定要不要为时间间隔、正交基和坐标系添加类型。


 **笔记** 你也可以为单位向量建立一个特殊的类型，尽管我觉得不是很有必要。-P.S.



## 1.7.2 单精度浮点数与双精度浮点数的比较

现代架构表明，减少内存的使用和保持内存访问的连贯性是实现高效的关键，所以最好使用单精度数据。然而为了避免数字问题又需要双精度数据进行运算。这方面的权衡取决于你的程序，但最好能在类型的定义中设定一个默认值。

 **笔记** 我建议了几何计算中使用双精度浮点数，在颜色计算中使用单精度。对于占据大量内存的数据，如三角形网格，我建议存储为单精度数据，但当通过成员函数访问数据时，要转换为双精度。—P.S.

 **笔记** 我主张所有的计算都使用单精度浮点数，直到发现在代码的特定部分必须使用双精度浮点数。—S.M.

## 1.7.3 调试图形程序

如果你四处打听可能会发现，随着程序员经验的增加，他们越来越少地使用传统的调试器。其中一个原因是，对于复杂的程序来说，使用这样的调试器相比简单的程序在使用时更加困难；另一个原因是，最严重的错误是概念上的错误（也就是说实现了错误的东西），这就很容易把大量的时间浪费在检查变量值上，而不能发现真正的错误。我们发现几种调试策略对图形学开发特别有用。

### 1.7.3.1 科学的方法

在图形程序中，有一种替代传统调试的方法往往很有用。它的缺点在于它与计算机程序员在职业生涯早期被教导不要做的事情非常相似，所以这样做时你可能会觉得“很调皮”：我们先创建一个图像，观察它有什么问题。然后我们对导致问题的原因提出一个假设，并对其进行测试。例如在一个光线追踪程序中，可能会有许多看起来有些随机的暗色像素。这是典型的“阴影失真 (*shadow acne*)”问题，大多数人在写光线追踪程序时都会遇到。传统的调试在这里是没有用的，相反，我们必须意识到阴影射线是打在被着色的表面上的。我们可能会注意到，黑点的颜色是环境色，所以直接照明是缺失的。直接照明可以在阴影中被关闭，所以你可以假设这些本该在亮处的点被错误地标记为在阴影中。为了验证这个假设，我们可以关闭阴影检查并重新编译。这将表明这些是错误的阴影测试，我们可以继续我们的调查工作。这种方法在实践中有时很好用的关键原因是，我们从不需要发现一个错误的值、或真正明确我们的概念性错误。相反，我们只是通过实验缩小了概念性错误的范围。通常情况下，只需要几次试验就可以追踪到事情的真相，这种类型的调试是很愉快的。

### 1.7.3.2 图像作为编码调试的输出

在许多情况下，从图形程序中获得调试信息的最简单方式是输出图像本身。如果你想知道在每个像素上都会运行的计算中的某个变量值，那你可以临时修改你的程序，把这个值直接复制到输出图像上，并跳过通常要进行的其他计算。例如，如果你怀疑表面法线导致了着


色的问题，你可以直接将法向量复制到图像上（ $x$  转为红色， $y$  转为绿色， $z$  转为蓝色），结果就是在计算中实际使用的向量的颜色编码图。或者，如果你怀疑某个特定的值有时超出了它的有效范围，就可以让你的程序在发生这种情况的地方使用鲜红色的像素。其他常见的技巧包括用明显的颜色画出表面的背面（当它们应该是不可见的情况下）、用物体的 ID 给图像着色、或者用像素的计算量来着色。

### 1.7.3.3 使用调试器

在某些场景中（特别是当正确步骤似乎导致结果不一致时），当我们想要观察程序中究竟发生了什么时，调试器仍然是无法取代的。麻烦在于图形程序往往涉及多次执行相同的代码（例如，每个像素执行一次或每个三角形执行一次），这导致从一开始就在调试器中逐步执行代码是完全不现实的。而且最困难的错误通常只发生在复杂的输入上。

一个有用的方法是为该错误“设置一个陷阱”。首先，确保你的程序是确定的——在单线程中运行并确保所有的随机数都是由固定的种子计算出来的。然后找出有错误的像素或三角形，并在你怀疑不正确的代码前添加一个只会在可疑情况下执行的语句。例如，如果你发现像素 (126, 247) 出现了错误，那么就添加

```
if(x = 126 and y = 247) then
    print "blarg!"
```


 **笔记** 使用固定的随机数种子的特殊调试模式很有用。

如果你在打印语句上设置一个断点，你就可以在你感兴趣的像素被计算出来之前进入调试器。有些调试器有一个“条件断点”功能，可以在不修改代码的情况下达到同样的效果。

在程序崩溃的情况下，传统的调试器对于确定崩溃的地点很有用。然后你应该在程序中开始回溯，使用断言和重新编译从而找出程序出错的地方。这些断言应该被留在程序中，以备将来可能出现的错误。这也就意味着要避免传统的步进过程，因为这样就不会在程序中添加有价值的断言。

### 1.7.3.4 调试过程的数据可视化

通常情况下，我们很难理解你的程序在做什么，因为它在最终出错之前计算了大量的中间结果。这种情况类似于测量大量数据的科学实验，有一个共通的解决办法：为自己制作合理的图表和插图以了解数据的含义。例如在光线追踪器中，你可能会编写将光线树可视化的代码，这样你就可以看到哪些路径对一个像素的贡献，或者在图像重采样程序中，你可能会做一些图来显示输入中的所有采样点。编写代码以可视化程序内部状态所花费的时间，也可以在优化程序时通过更好地了解程序的行为来得到回报。

 **笔记** 我喜欢把调试中打印出来的语句格式化，这样输出的结果恰好是一个 MATLAB® 或 Gnuplot 脚本，可以做出有帮助的图。-S.M.

## 说明

关于软件工程的讨论受到《Effective C++》系列 (Meyers, 1995, 1997)、《Extreme Programming Explained: Embrace Change》(Beck & Andres, 2004) 和《The Practice of Programming》(Kernighan & Pike, 1999) 的影响。关于实验性调试的讨论基于和 Steve Parker 的讨论。

有许多与计算机图形学有关的年度会议,包括 ACM SIGGRAPH 和 SIGGRAPH Asia、Graphics Interface、Game Developers Conference (GDC)、Eurographics、Pacific Graphics、High Performance Graphics、Eurographics Symposium on Rendering 以及 IEEE VisWeek。通过网络搜索, 可以很容易找到这些会议的名称。

## 第2章 基础数学知识

图形学的大部分内容只是将数学公式直译成代码。数学公式越干净，写出的代码就越干净，所以本书的大部分内容都聚焦于“如何使用正确的数学公式完成工作”这个话题。本章回顾了高中和大学数学中的各种工具，旨在作为参考，而不是作为一个教程。本章可能看起来像是一个各种主题组成的大杂烩，事实上它也确实如此：每个主题之所以被选择是因为它在“标准”数学课程中的略微不寻常、因为它在图形学中具有核心地位、或者因为我们一般不从几何角处理它。除了对本书所使用的符号进行回顾外，本章还强调了标准本科课程中有时会跳过的几个要点，比如三角形的重心坐标。本章不会严肃探究这些知识，相反，我们会着重于直觉和几何解释。而对线性代数的讨论则被推迟到了第六章中我们学习矩阵变换之前。我们鼓励读者浏览本章以熟悉所涉及的主题，并在需要时参考本章。本章末尾的练习也许能帮助你确定哪些主题需要复习。

### 2.1 集合和映射

映射 (*mapping*)，也叫函数 (*function*)，是数学和编程的基础。像程序中的函数一样，数学中的映射接受某个类型 (*type*) 的参数，并将其映射到 (返回) 一个特定类型的对象。在程序中，我们说的是“类型”；在数学中，我们说的是“集合”。当一个对象是一个集合的成员时，我们使用  $\in$  符号。比如说，

$$a \in S,$$

可以读作“ $a$  是集合  $S$  的成员”。给定任何两个集合  $A$  和  $B$ ，我们可以通过取两个集合的笛卡尔积 (*cartesian product*) 来创建第三个集合，表示为  $A \times B$ 。这个集合  $A \times B$  由所有可能的有序对  $(a, b)$  组成，其中  $a \in A$  且  $b \in B$ 。作为缩写，我们用符号  $A^2$  表示  $A \times A$ 。我们可以扩展笛卡尔积，从三个集合中创建一个新集合，该集合包含所有可能的有序三元组，以此类推，从任意多的集合中创建任意长的有序元组。

常见的集合包括：

- $\mathbb{R}$ ——实数；
- $\mathbb{R}^+$ ——非负实数 (包括 0)；
- $\mathbb{R}^2$ ——真实二维平面中的有序对；
- $\mathbb{R}^n$ —— $n$  维笛卡尔空间中的点；
- $\mathbb{Z}$ ——整数；
- $S^2$ ——单位球面上的三维点 ( $R^3$  中的点) 的集合。

请注意，虽然  $S^2$  是由三维空间中的点组成的，但它也在一个可以用两个变量进行参数化的表面上，所以它可以被认为是一个二维集合。映射的记号使用箭头和冒号，例如：



$$f: \mathbb{R} \mapsto \mathbb{Z},$$

你可以把它理解为“有一个叫做  $f$  的函数，它将一个实数作为输入，并将其映射成一个整数”。这里，箭头前面的集合被称为函数的定义域 (*domain*)，右边的集合被称为目标 (*target*)<sup>1</sup>。计算机程序员可能更愿意使用下面的等价表达：“有一个叫  $f$  的函数，它有一个实数参数并返回一个整数”。换句话说，上面的集合映射符号等同于常见的编程符号：

$$\text{integer } f(\text{real}) \leftarrow \text{等价} \rightarrow f: \mathbb{R} \mapsto \mathbb{Z}$$

因此，“冒号-箭头”符号可以被认为是一种编程语法，就这么简单。

点  $f(a)$  被称为  $a$  的像 (*image*)，一个集合  $A$  (定义域的子集) 的像是目标的子集，目标包含  $A$  中所有点的像。整个定义域的像被称为函数的值域 (*range*)。

### 2.1.1 逆映射

如果我们有一个函数  $f: \mathbf{A} \mapsto \mathbf{B}$ ，那么可能存在一个反函数 (*inverse function*)  $f^{-1}: \mathbf{B} \mapsto \mathbf{A}$ ，这是通过  $f^{-1}(b) = a, b = f(a)$  定义的。该定义只在所有  $b \in \mathbf{B}$  是函数  $f$  下一些点的像 (也就是说，值域和目标相等) 且只有一个这样的点 (也就是说，在  $f(a) = b$  中，只有一个让等式成立的  $a$ ) 的情况下有效。这种映射或函数称为双射 (*bijection*)。双射将每一个  $a \in \mathbf{A}$  映射到唯一的  $b \in \mathbf{B}$ ，对于每一个  $b \in \mathbf{B}$ ，正好有一个  $a \in \mathbf{A}$ ，使  $f(a) = b$  (图 2.1)。例如，一组骑手和一组马匹之间的双射关系表明，每个人都骑着一匹马，每匹马都被骑着。这两个函数是骑手 (马) 和马 (骑手)，它们都是彼此的反函数。不是双射的函数没有反函数 (图 2.2)。

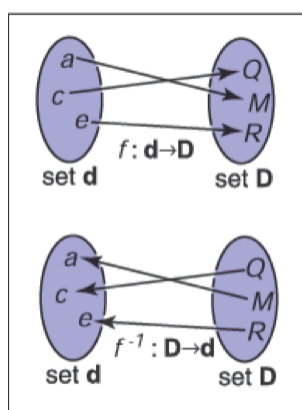


图 2.1: 一个双射  $f$  及其反函数  $f^{-1}$ 。注意， $f^{-1}$  也是一个双射。

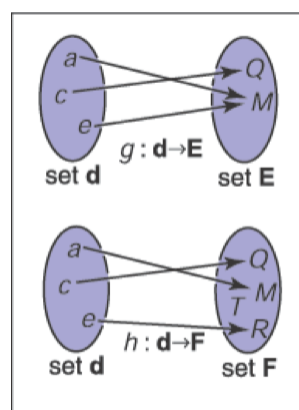


图 2.2: 函数  $g$  不具有反函数，因为集合  $\mathbf{d}$  中的两个元素映射到了集合  $\mathbf{E}$  中的同一个元素。函数  $h$  没有反函数，因为集合  $\mathbf{d}$  中没有元素映射到集合  $\mathbf{F}$  中的元素  $T$ 。

<sup>1</sup> 译注：此处的目标 (*target*) 指的是陪域 (*codomain*)。关于这点的详细讨论请参考 <https://math.stackexchange.com/questions/440235/range-of-function-vs-target-of-function> 中的回答。

一个双射的例子是  $f: \mathbb{R} \mapsto \mathbb{R}$ ,  $f(x) = x^3$ 。它的反函数为  $f^{-1}(x) = \sqrt[3]{x}$ 。这个例子表明, 标准的符号可能有些笨拙, 因为  $x$  在  $f$  和  $f^{-1}$  中都被用作虚拟变量 (*dummy variable*)。有时使用不同的虚拟变量会更直观, 比如写成  $y = f(x)$  和  $x = f^{-1}(y)$  的形式。这就有了更直观的  $y = x^3$  和  $x = \sqrt[3]{y}$ 。一个没有反函数的例子是  $\text{sqr}: \mathbb{R} \mapsto \mathbb{R}$ ,  $\text{sqr}(x) = x^2$ 。它没有反函数的原因有两个: 首先  $x^2 = (-x)^2$ , 其次定义域中没有成员映射到目标的负数部分。请注意, 如果我们把定义域和值域限制在  $\mathbb{R}^+$ , 还是可以定义一个反函数的。如此一来  $\sqrt{x}$  就是一个有效的反函数。

### 2.1.2 区间

通常情况下, 我们希望让一个函数处理的实数局限在某个范围内, 实现这种约束的一个方法就是规定一个区间 (*interval*)。区间的一个例子是“0 和 1 之间的实数、不包括 0 或 1”, 我们把它表示为  $(0, 1)$ 。因为它不包括其端点, 所以被称为开区间。相对应的闭区间则包含其端点, 用方括号表示:  $[0, 1]$ 。这种符号可以混合使用; 比如,  $[0, 1)$  包括 0, 但不包括 1。当写一个区间  $[a, b]$  时, 我们假定  $a \leq b$ 。表示一个区间的三种常见方法如图 2.3 所示。区间的笛卡尔积经常被使用。例如, 为了表示点  $x$  在三维的单位立方体中, 我们说  $x \in [0, 1]^3$ 。

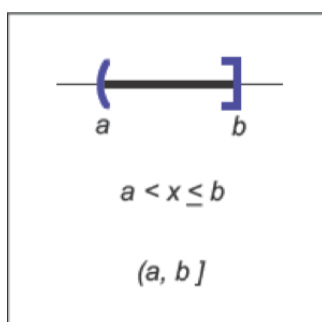


图 2.3: 三个表示方式是等效的, 都能用于表示“从  $a$  到  $b$ 、包含  $b$  但不包含  $a$  的实数区间”。

区间在与常常与集合运算结合在一起使用: 交集 (*intersection*)、并集 (*union*) 和差集 (*difference*)。例如, 两个区间的交集是它们公共点的集合, 符号  $\cap$  用来表示交集。例如,  $[3, 5) \cap [4, 6] = [4, 5)$ 。而并集使用符号  $\cup$  来表示两个区间中的点合并后产生的集合。例如,  $[3, 5) \cup [4, 6] = [3, 6]$ 。差集运算符与前两个运算符不同, 它根据参数顺序产生不同结果。减号用于差集运算, 它返回在左侧区间但不在右侧区间的点。例如,  $[3, 5) - [4, 6] = [3, 4]$ , 以及  $[4, 6] - [3, 5] = [5, 6]$ 。这些操作在区间图就能很直观地表现出来 (图 2.4)。

### 2.1.3 对数函数

尽管今天对数 (*logarithm*) 已不像使用计算器之前那么普遍了, 但它在具有指数项的方程问题中仍然很有用。根据定义, 每个对数都有一个基数  $a$  (*base a*, 也称“底”)。 $x$  的以  $a$  为底的对数通常被写作  $\log_a x$ , 并且它被定义为若要得到  $x$  需要将  $a$  乘幂的指数值。即:

$$y = \log_a x \iff a^y = x.$$

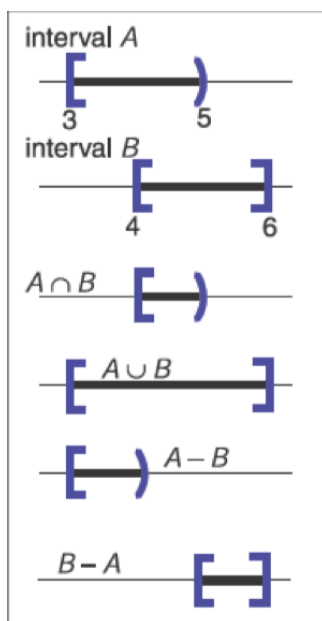


图 2.4: 在  $[3, 5)$  和  $[4, 6]$  上的区间操作。

请注意，以  $a$  为底数的对数函数和以  $a$  为底数的幂函数是互逆的。这个基本定义可引申出几个结论：

$$a^{\log_a(x)} = x;$$

$$\log_a(a^x) = x;$$

$$\log_a(xy) = \log_a x + \log_a y;$$

$$\log_a(x/y) = \log_a x - \log_a y;$$

$$\log_a x = \log_a b \log_b x.$$

当我们将微积分应用于对数时，经常会出现特殊数字  $e = 2.718\dots$ 。以  $e$  为底的对数被称为自然对数 (*natural logarithm*)。我们一般用简写  $\ln$  来表示：

$$\ln x \equiv \log_e x.$$

“ $\equiv$ ”符号的意思是“根据定义恒等”。像  $\pi$  一样，特殊数字  $e$  在很多情况下都会出现。除了  $e$  以外，在许多领域还使用特定的基数进行操作，并在其写法中省略基数，即  $\log x$  的形式。例如，天文学家经常使用以 10 为底的对数，理论计算机科学家经常使用以 2 为底的对数。由于计算机图形学借鉴了许多其他领域的技术，我们将避免这种缩写。

对数的导数和指数的导数阐明了为什么自然对数是“自然的”：

$$\begin{aligned} \frac{d}{dx} \log_a x &= \frac{1}{x \ln a} \\ \frac{d}{dx} a^x &= a^x \ln a \end{aligned}$$

上面的常数乘数只有在  $a = e$  时才等于 1。

## 2.2 求解二次方程

一个二次方程 (quadratic equation) 满足如下格式:

$$Ax^2 + Bx + C = 0,$$

其中  $x$  是一个未知的实数, 而  $A$ 、 $B$  和  $C$  是已知的常数。如果你能联想到  $y = Ax^2 + Bx + C$  的二维  $xy$  图, 那么该二次方程的解就是函数  $y$  与  $x$  轴交点的横坐标值。因为  $y = Ax^2 + Bx + C$  是一条抛物线, 根据抛物线是否未相交、相切或相交于  $x$  轴, 将有零个、一个或两个实数解 (图 2.5)。

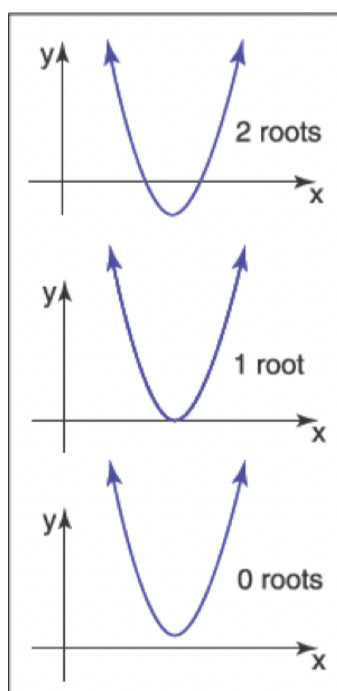


图 2.5: 二次方程根的几何解释是抛物线与  $x$  轴的交点。

为了求解二次方程, 我们先将方程除以  $A$ :

$$x^2 + \frac{B}{A}x + \frac{C}{A} = 0$$

然后, 我们通过“构造平方项”的方式来合并项:

$$\left(x + \frac{B}{2A}\right)^2 - \frac{B^2}{4A^2} + \frac{C}{A} = 0$$

将常数部分移到等号右边, 然后取平方根, 得到:

$$x + \frac{B}{2A} = \pm \sqrt{\frac{B^2}{4A^2} - \frac{C}{A}}.$$



从两边减去  $\frac{B}{2A}$ ，然后用分母  $2A$  合并分式，就得到了我们熟悉的形式：<sup>1</sup>

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}. \quad (2.1)$$

此处的“ $\pm$ ”符号意味着有两个解，一个是正号，一个是负号。因此， $3 \pm 1$  等于 2 或 4。请注意，决定实数解数量的项是：

$$D \equiv B^2 - 4AC$$

这被称为二次方程的判别式 (*discriminant*)。如果  $D > 0$ ，有两个实数解 (也叫根 (*root*)); 如果  $D = 0$ ，有一个实数解 (一个双根 (*a "double" root*)); 如果  $D < 0$  则没有实数解。

例如， $2x^2 + 6x + 4 = 0$  的根是  $x = -1$  和  $x = -2$ ，方程  $x^2 + x + 1 = 0$  没有实数解。这些方程的判别式分别是  $D = 4$  和  $D = -3$ ，且我们都按预期得到了解的数量。在程序中，我们通常先评估  $D$  值，如果  $D$  为负数，则直接返回无根 (*no roots*) 而不取平方根。

## 2.3 三角函数

在图形学中，我们在许多情况下都会用到基本三角函数。一般来说它平平无奇，但能记住基础定义往往是有帮助的。

### 2.3.1 角

尽管我们认为角的概念是理所当然的，但我们应重温其定义，以便我们将角的概念扩展到球体上。角是在两条半直线 (从一个原点出发的无限射线) 或方向之间形成的，如图 2.6 所示，两条射线会形成两个角，我们还需要通过一些约定来决定使用哪个角。角 (*angle*) 是由射线在单位圆上切出的弧段的长度来定义的。一个常见的约定是使用较小的弧长，而角的符号由两条射线被指定的顺序决定。在该约定的约束下，所有角都在  $[-\pi, \pi]$  范围内。

这两个角中的任何一个都可以通过两条射线“切割”的单位圆的弧长来度量。因为单位圆的周长是  $2\pi$ ，所以两个可能的角总和为  $2\pi$ ，这种弧长的单位是弧度 (*radians*)。另一个常见的单位是角度 (*degrees*)，圆的周角是  $360^\circ$ 。因此，一个弧度为  $\pi$  的角的角度是  $180^\circ$ ，通常表示为  $180^\circ$ 。角度和弧度之间的转换是

$$\begin{aligned} &= \frac{180}{\pi} \\ &= \frac{\pi}{180} \end{aligned}$$

<sup>1</sup> 一个稳健的实现会使用等价的表达式  $\frac{2C}{(-B \mp \sqrt{B^2 - 4AC})}$  来根据  $B$  的符号计算其中一个根 (练习 7)。

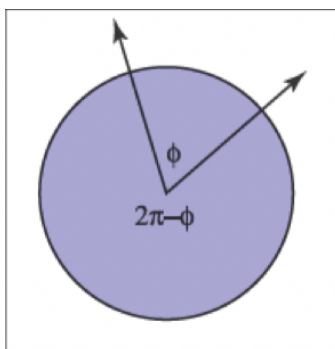


图 2.6: 两条射线将单位圆切割成两条弧线。任意一条弧线的长度都是这两条射线“之间”的有效角。我们可以使用“较小弧线长度是角”的约定，或者这种约定：“将两条射线按一定顺序指定、确定角  $\phi$  的弧是从第一条射线到第二条射线逆时针扫过的弧。

### 2.3.2 三角函数

给出一个边长为  $a$ 、 $o$ 、 $h$  的直角三角形，其中  $h$  是最长边的长度（总是与直角相对），或者叫斜边（*hypotenuse*），勾股定理（*Pythagorean theorem*）描述了一种重要的关系：

$$a^2 + o^2 = h^2$$

你可以从图 2.7 中看到这一点，大正方形的面积为  $(a + o)^2$ ，四个三角形的面积合计为  $2ao$ ，中心正方形的面积为  $h^2$ 。

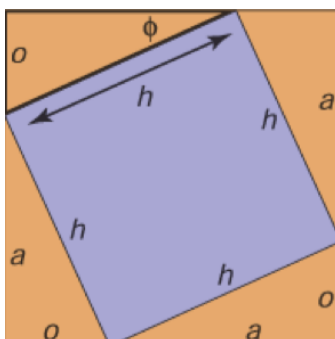


图 2.7: 勾股定理的几何证明。

由于三角形和中心正方形均匀地细分了较大的正方形，所以我们可以很容易地取得如下公式  $2ao + h^2 = (a + o)^2$ 。通过直角三角形，我们可以定义角  $\phi$  的  $\sin$  值和  $\cos$  值，以及其他基于比值的三角表达式：

$$\sin \phi \equiv o/h$$

$$\csc \phi \equiv h/o$$

$$\cos \phi \equiv a/h$$

$$\sec \phi \equiv h/a$$

$$\tan \phi \equiv o/a$$

$$\cot \phi \equiv a/o$$

这些定义允许我们建立极坐标系 (*polar coordinates*), 其中任何一个点都被编码为其与原点的距离及其与  $x$  轴正半轴的角 (带符号), 如图 2.8 所示。请注意, 角的范围是  $\phi \in (-\pi, \pi]$ , 且正数角是从  $x$  轴正半轴逆时针方向旋转取得的。“将逆时针方向映射到一个正数”这个约定是主观性的, 但它被应用于图形学很多场景中, 因此值得一记。

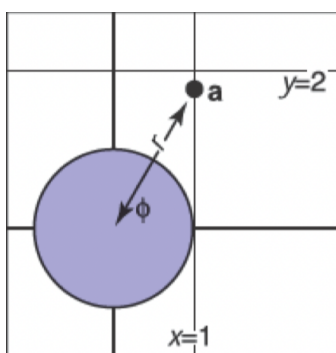


图 2.8: 点  $(x_a, y_a)$  的极坐标为  $(r_a, \phi_a) = (2, \frac{\pi}{3})$ 。

三角函数是周期性的, 可以接受任何角作为参数。例如,  $\sin(A) = \sin(A + 2\pi)$ 。这意味着当我们考虑整个  $\mathbb{R}$  域时, 这些函数是不可逆的。这个问题可以通过限制标准反函数的值域来避免。这在几乎所有现代数学库中都是以标准方式进行的 (例如, Plauger (1991))。定义域和值域如下:

$$\begin{aligned} \text{asin} : [-1, 1] &\mapsto [-\pi/2, \pi/2] \\ \text{acos} : [-1, 1] &\mapsto [0, \pi] \\ \text{atan} : \mathbb{R} &\mapsto [-\pi/2, \pi/2] \\ \text{atan 2} : \mathbb{R}^2 &\mapsto [-\pi, \pi] \end{aligned} \tag{2.2}$$

最后一个函数  $\text{atan 2}(s, c)$  往往很有用。它采用与  $\sin A$  成比例的值  $s$  以及与  $\cos A$  成比例的值  $c$ , 两者具有相同的因子, 最后返回值  $A$  (假定因子为正数)。一种思考方式是它返回了二维笛卡尔点  $(c, s)$  在极坐标中的角 (图 2.9)。

### 2.3.3 有用的恒等式

本节未经推导地列出了各种有用的三角恒等式。

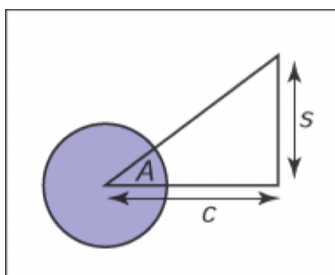


图 2.9: 函数  $\text{atan2}(s, c)$  返回  $A$  的角, 这在图形学中往往很有用。

移位恒等式 (*shifting identities*):

$$\sin(-A) = -\sin A$$

$$\cos(-A) = \cos A$$

$$\tan(-A) = -\tan A$$

$$\sin(\pi/2 - A) = \cos A$$

$$\cos(\pi/2 - A) = \sin A$$

$$\tan(\pi/2 - A) = \cot A$$

毕达哥拉斯恒等式 (*Pythagorean identities*):

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

$$\sec^2 A - \tan^2 A = 1$$

$$\csc^2 A - \cot^2 A = 1$$

加减法恒等式 (*addition and subtraction identities*):

$$\sin(A + B) = \sin A \cos B + \sin B \cos A$$

$$\sin(A - B) = \sin A \cos B - \sin B \cos A$$

$$\sin(2A) = 2 \sin A \cos A$$

$$\cos(A + B) = \cos A \cos B - \sin A \sin B$$

$$\cos(A - B) = \cos A \cos B + \sin A \sin B$$

$$\cos(2A) = \cos^2 A - \sin^2 A$$

$$\tan(A + B) = \frac{\tan A + \tan B}{1 - \tan A \tan B}$$

$$\tan(A - B) = \frac{\tan A - \tan B}{1 + \tan A \tan B}$$

$$\tan(2A) = \frac{2 \tan A}{1 - \tan^2 A}$$

半角恒等式 (*half-angle identities*):



$$\sin^2(A/2) = (1 - \cos A)/2$$

$$\cos^2(A/2) = (1 + \cos A)/2$$

乘法恒等式 (*product identities*):

$$\sin A \sin B = -(\cos(A + B) - \cos(A - B))/2$$

$$\sin A \cos B = (\sin(A + B) + \sin(A - B))/2$$

$$\cos A \cos B = (\cos(A + B) + \cos(A - B))/2$$

以下恒等式适用于边长为  $a$ 、 $b$  和  $c$  的任意三角形，每边相对的角分别为  $A$ 、 $B$ 、 $C$  (图 2.10),

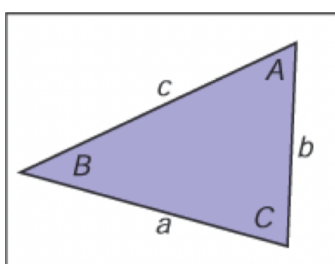


图 2.10: 三角定律的几何示意图。

$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c} \quad (\text{正弦定理})$$

$$c^2 = a^2 + b^2 - 2ab \cos C \quad (\text{余弦定律})$$

$$\frac{a+b}{a-b} = \frac{\tan\left(\frac{A+B}{2}\right)}{\tan\left(\frac{A-B}{2}\right)} \quad (\text{正切定律})$$

三角形的面积也可以根据这些边长计算:

$$\text{三角形面积} = \frac{1}{4} \sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}.$$

### 2.3.4 立体角和球面三角学

本节中的传统三角学涉及平面上的三角形。三角形也可以定义在非平面上，这在很多领域都会出现，比如天文学中就有定义在单位半径球体上的三角形。这些球面三角形 (*spherical triangles*) 的边是球面上的大圆 (单位半径的圆) 的线段。对这类三角形的研究是一个称为球面三角学 (*spherical trigonometry*) 的领域，其在图形学中一般并不常用，但当它出现时便不可忽视。我们不会在这里讨论它的细节，但希望读者知道当相关问题出现时存在这样一个研究领域，并且该领域有很多有用的规则，例如球面余弦定律和球面正弦定律。关于使用球面三角学机制的一个例子，请看一篇关于三角形光照采样 (*sampling triangle lights*) (投影到一个球面三角形) 的论文 (Arvo, 1995b)。

对计算机图形学而言，立体角 (*solid angles*) 的概念更加重要。角可以让我们量化一些东西，比如“在我的视野中这两根柱子的距离是多少”，而立体角则能让我们量化“那架飞机在我的视野中覆盖了多少面积”这样的问题。对于传统的角，我们把线段投射到单位圆上，并在单位圆上测量投射的两点之间的弧长。我们经常使用这个弧长作为角，以至于许多人都会忘记这个定义，因为在当下它对我们来说是如此直观。立体角也同样简单，但它们可能看起来更令人困惑，因为我们大多数人都是在成年后才了解它们的。对于立体角，我们把“看到”飞机的可见方向投射到单位球体上，并测量其面积。这个面积就是立体角，与“弧长就是角”的方式相同。角以弧度测量，总和为  $2\pi$  (单位圆的总长度)，而立体角则以立体弧度 (*steradians*) 测量，总和为  $4\pi$  (单位球的总表面积)。

## 2.4 向量

向量描述了长度和方向，它总是用箭头表示。如果两个向量具有相同的长度和方向，则它们是相等的，即使我们认为它们处于不同的位置 (图 2.11)。您应该尽可能地将向量视为箭头，而不是坐标或者数字。在某些时候，我们将不得不在我们的程序中降向量表示为数字，但即使在代码中，他们也应该被操作为对象，并且只有低级向量操作应该知道他们的数字表示 (DeRose, 1989)。向量将表示为粗体字符，例如 **a**。向量的长度表示为  $\|\mathbf{a}\|$ 。单位向量是长度为一的任何向量。零向量是零长度的向量，零向量的方向未定义。

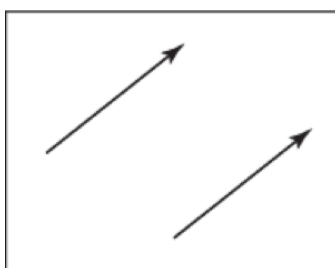


图 2.11: 这两个向量是相同的，因为它们拥有相同的长度和方向。

向量可以用来表示许多不同的东西。例如，它们可用于存储偏移量，也称为位移。如果我们知道“宝藏埋在秘密会议地点以东两步，以北三步”，那么我们就可以知道偏移量，但我们还不知道从哪里开始。向量也可以用来存储一个位置，换句话说表达位置或点。位置可以表示为来自另一个位置的位移。通常，有一些已知的原点位置，所有其他位置都存储为相对于它的偏移量。请注意，位置不是向量。正如我们将讨论的那样，您可以将两个向量相加。然而，在计算一个位置的加权平均值时，将两个位置相加通常没有意义，除非它是计算位置加权平均值时的中间操作 (Goldman, 1985 年)。添加两个偏移量确实有意义，所以这就是偏移量是向量的原因之一。但这强调，一个位置不是偏移量；它是来自特定原点的偏移量。偏移量本身不是位置。

### 2.4.1 向量运算

向量具有我们应用于实数的大多数通常算术运算。两个向量相等当且仅当它们具有相同的长度和方向。根据平行四边形规则将两个向量相加。这条规则表明，两个向量的和是通过将任一向量的尾部放在另一个向量的头部来求得的（图 2.12）。和向量是与两个向量共同“构成三角形”的向量。平行四边形是通过按任一顺序求和而形成的。这强调向量加法是可交换的：

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$$

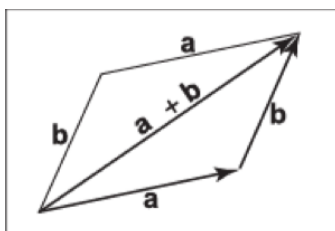


图 2.12: 两个向量通过从头到尾排列来实现相加。这一过程可以按任意顺序实现。

请注意，平行四边形法则只是形式化了我们对位移的直觉。想象沿着一个向量从头走到尾，然后沿着另一个向量走。净位移只是平行四边形的对角线。您还可以为向量添加一元负号： $-\mathbf{a}$ （图 2.13）是一个与  $\mathbf{a}$  长度相同但方向相反的向量。这使我们也可以定义减法：

$$\mathbf{b} - \mathbf{a} = -\mathbf{a} + \mathbf{b}$$

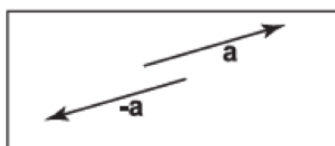


图 2.13: 向量  $-\mathbf{a}$  拥有与向量  $\mathbf{a}$  相同的长度但是方向相反。

您可以使用平行四边形法则可视化向量减法（图 2.14）。我们可以写成

$$\mathbf{a} + (\mathbf{b} - \mathbf{a}) = \mathbf{b}$$

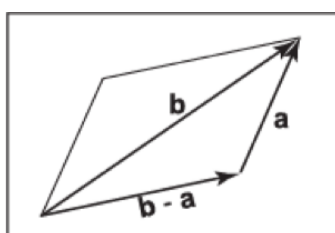


图 2.14: 向量减法只是第二个参数反转的矢量加法。

向量也可以相乘。事实上，有几种乘法涉及向量。首先，我们可以通过乘以实数  $k$  来缩放向量。这只是乘以向量的长度而不改变其方向。例如， $3.5\mathbf{a}$  是与  $\mathbf{a}$  方向相同的向量，但它的

长度是  $a$  的 3.5 倍。我们在本节后面讨论涉及两个向量的两种乘积，点乘和叉乘，以及第 6 章中涉及三个向量（行列式）的乘积。

## 2.5 积分

## 2.6 密度函数

## 2.7 曲线和曲面

## 2.8 线性插值

## 2.9 三角形


## 2.10 离散概率


## 2.11 连续概率

## 2.12 蒙特卡洛积分

## 第3章 光栅图像


大多数 CG (computer graphics) 图片在某种光栅显示器上呈现给用户。光栅显示器以矩形像素阵列的方式呈现图片。一个常见的例子是平板显示器或平板电视，它们有一个由微小的发光像素组成的矩形阵列，每个发光像素都可以被设置成不同的颜色从而得到任何想得到的图像。不同的颜色是通过混合不同强度的红、绿、蓝光实现的。大多数打印机，如激光打印机和喷墨打印机，也是光栅设备。它们以扫描为基础：实际上没有由像素组成的物理网格，但图像是通过在网格上选定的点上沉积墨水而按顺序铺设的。

 **笔记** 像素 (Pixel) 是图像元素 (picture element) 的简称

 **笔记** 打印机的颜色更复杂，涉及至少有四种颜料的混合物。

光栅也被普遍应用到了图像输入设备上。一块数码相机包括一个由光感像素网格组成的图像传感器，每个光感像素都能记录颜色和落在其上的光线强度。一台桌面扫描器包括一个由像素组成的线性阵列，它每秒横向扫描页面并不断记录来产生像素网格。


由于光栅在设备中相当普遍，光栅图像是储存、处理图像的最常见方式。一幅光栅图像只是一个储存了每个像素的像素值（通常是一个由三个数字表示的颜色，分别代表红、绿、蓝）的二维数组。内存中存储的光栅图像可通过被存储图像的每个像素值来——控制显示器上单个像素的颜色从而被显示出来。

 **笔记** 也可能是因为光栅图像如此便利以至于光栅设备如此普遍。

但我们并不总是以这种方式显示图片。我们可能想要改变图像的大小和朝向，修正颜色，甚至将图像显示在运动的三维空间平面上。即使在电视上，显示器也很少会有和图像相同的像素数量。诸如此类的考虑打破了图像像素和显示像素之间的直接联系。最好把光栅图像看作是对要显示的图像的独立设备描述，而把显示设备看作是接近该理想图像的一种方式。

除了使用像素阵列，还有其他方式描述图像。矢量图像通过存储图形的描述来绘制图像——由线条和曲线围成的色彩区域——不包含对特定像素网格的引用。本质上，这相当于存储如何显示图像的说明而不是存储需要显示的像素。矢量图像的主要优势在于他们是分辨率独立的，因此在高分辨率设备上具有很好的显示效果。而相对应的，其缺点在于在显示前图像需要被栅格化。矢量图像经常用于文本、图表、机械制图、对清晰度和精确度要求较高的应用场合以及不需要摄影图像和复杂阴影的场景。

在这个章节，我们讨论光栅图像和光栅显示器的基础，特别需要注意标准显示器的非线性问题。当我们在后面的章节中讨论计算图像时，像素值如何与光强度建立联系的细节是很重要的并需要牢记于心。

 **笔记** 或者说：你需要知道图像中这些数值的真正含义。



## 3.1 光栅设备

在讨论抽象的光栅图像之前，了解使用这些图像的一些具体设备的基本操作是很有意义的。一些熟悉的光栅设备可以被归入一个简单的层次结构中：

- 输出
  - 显示器
    - 透射式显示器：液晶显示器（LCD）
    - 自发光式显示器：发光二极管（LED）显示器
  - 打印
    - 二值图像：喷墨打印机
    - 连续色调（continuous-tone）图像：热升华打印机
- 输入
  - 二维阵列传感器：数码相机
  - 一维阵列传感器：平板扫描仪

### 3.1.1 显示器

目前的显示器，包括电视和数字电影放映机以及计算机显示器和投影仪，几乎普遍基于固定的像素阵列。它们可以分为自发光式显示器，它使用直接发射可控光量的像素，和透射式显示器，其中像素本身不发光，而是改变它们允许通过的光量。透射式显示器需要光源来照亮它们：在直视型显示器中，光源是阵列后面的背光；在投影仪中，光源是一盏灯，它发出的光穿过阵列后投射到屏幕上。而自发光式显示器便是其自身的光源。

发光二极管（LED）显示器是自发光式显示器的一个例子。每个像素由一个或多个 LED 组成，这是一种基于通过电流强度决定发光亮度的半导体设备（基于有机/无机半导体）（图 3.1）。

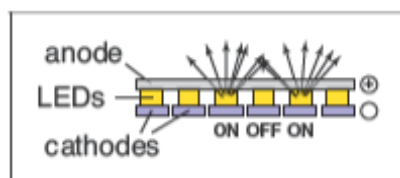


图 3.1: 发光二极管（LED）显示器的运行机制。

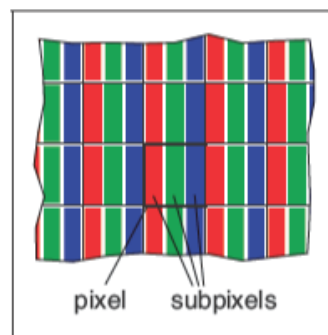
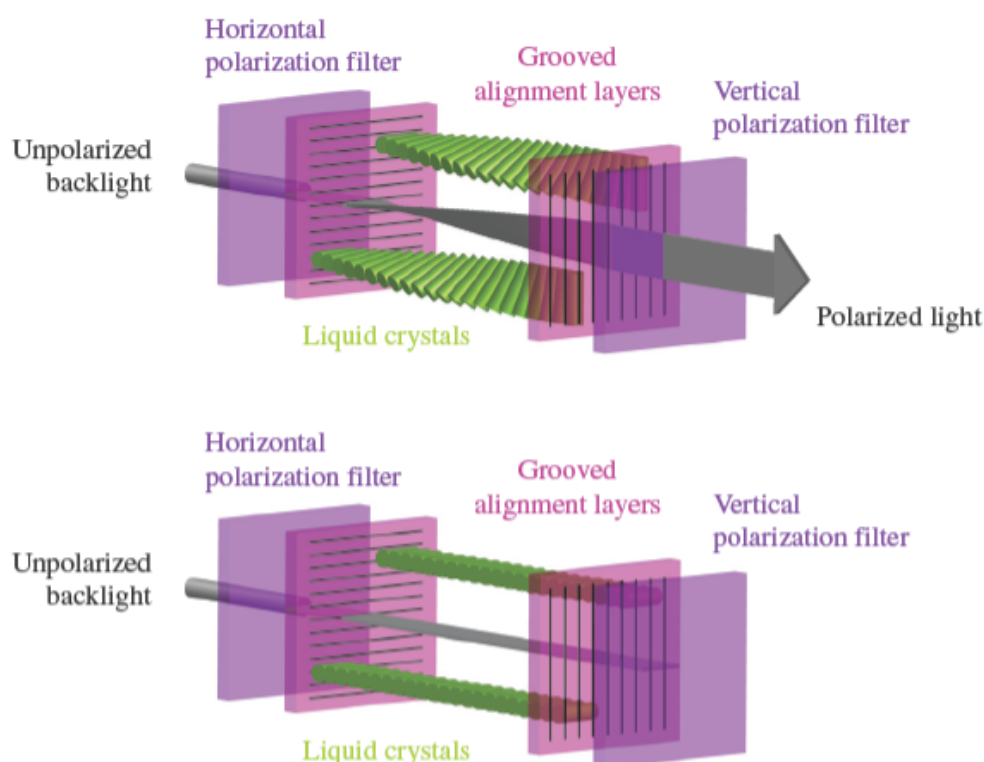


图 3.2: 组成平板显示器中单个像素的红、绿、蓝子像素。

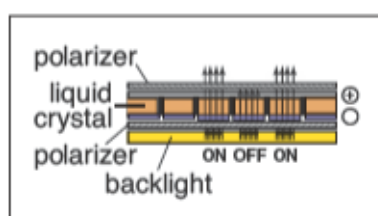
彩色显示器上的像素被细分为 3 个独立控制的子像素，一个红色、一个绿色、一个蓝色，每个 LED 都由不同材料组成从而发出不同颜色的光（图 3.2）。当从一定距离外观看显示器时，眼睛无法分开每个独立的子像素，最后感知到的颜色便是红、绿、蓝三色混合成的颜色。

液晶显示器 (LCD) 是透射式显示器的一个例子。液晶是一种材料，其分子结构使其能够旋转通过它的光的偏振，并且旋转的程度可以通过施加的电压来调节。LCD 像素 (图 3.3) 后面有一层偏振膜，因此它被偏振光照亮——让我们假设光是水平偏振的。



**图 3.3:** 上图是一个处于开启状态的 LCD 显示器像素，其中液晶单元旋转光线的偏振方向，因此光线能通过前端偏振片。下图是一个处于关闭状态的 LCD 显示器像素，其中前端偏振片阻挡了所有通过后端偏光片的光线。图由 Reinhard, Khan, Akyilz and Johnson(2008) 提供。


像素前的第二层偏振膜只传导垂直偏振光。如果液晶层两端施加电压至不改变偏振方向，则所有光都被阻挡且像素处于“关闭”状态（最小光强度）。如果电压被设置以让液晶层改变偏振方向至 90 度，则所有从像素后方进入的光都将从前端逃离，此时像素处于完全“开启”状态（具有最大光强度）。而中等的电压会部分旋转偏振角，这样前端偏振片会阻挡部分光线，最终产生的光强度介于最大值与最小值之间（图 3.4）。就像彩色 LED 显示器一样，彩色 LCD 显示器的像素中也包含红、绿、蓝子像素，实质上是三个独立的、覆盖有红、绿、蓝三种颜色过滤器的像素。



**图 3.4:** 液晶显示器 (LCD) 的运行机制。

任何形式的有固定像素阵列的显示器——包括以上这些或其他技术——都有一个由阵列大小决定的基础的固定分辨率。对于显示器或图像，分辨率仅仅意味着像素阵列的尺寸：如

果桌面显示器有  $1920 \times 1200$  的分辨率，那么它总共有 2304000 个像素分布在 1920 列和 1200 行的像素阵列中。

 **笔记** 显示器的分辨率有时被叫做“本地分辨率”因为许多显示器都可以通过内置转换处理其他分辨率的图片。

### 3.1.2 打印设备

在纸上永久记录图像的过程与在显示器上短暂地显示图片相比有许多限制。在印刷中，颜料被分布于纸上或其他打印媒介上，当光从纸上反射后便能形成想要的图案。打印机是类似于显示器的光栅设备，但许多打印机只能打印二值图像，即每个网格位置要么沉积颜料，要么不沉积，没有可能的中间值。

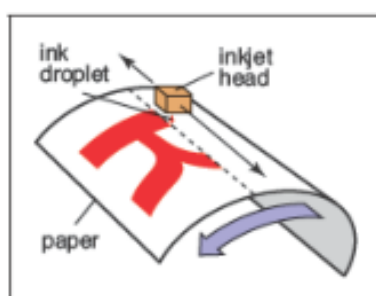



图 3.5: 喷墨打印机的运行机制。

喷墨打印机（图 3.5）是一种通过扫描形成光栅图像的设备。喷墨打印头容纳了含有颜料的液体墨水，并能通过电子控制将墨水以微小墨滴的形式喷出。喷墨头在纸上横向移动，当它经过应该接受墨水网格位置时便会将墨滴喷射出；想要保持空白的位置则不会喷射墨滴。喷墨头每次扫描（sweep）之后，纸张都会略微前移，随后网格的下一行被喷射到纸上。彩色打印机使用数个打印头，每个喷头喷射一种不同的颜色的墨水，因此每个网格位置可接受任何由不同颜色的墨滴混合成的颜色。由于所有墨滴都一样，一台打印二值图像的喷墨打印机在每个网格格点上要么有墨滴要么没有墨滴，没有中间的色彩度（shade）。

喷墨打印机没有物理像素阵列，其分辨率由墨滴的尺寸以及喷墨头每次扫描后纸张前进的距离决定。许多喷墨打印机的打印头中有多个喷嘴，允许其在一次横向移动中完成多次扫描，但最终是由纸张前移距离决定行距而非喷嘴间距。

 **笔记** 也存在连续喷墨打印机，它以连续的螺旋路径打印包裹在旋转的滚筒周围的纸张，而不是来回移动打印头。

热敏染料转印工艺是连续色调印刷工艺的一个例子，这意味着可以在每个像素上沉积不同量的染料 - 它不像喷墨打印机那样全有或全无（图 3.6）。将含有彩色染料的供体色带压在纸张（或染料接收器）与包含加热组件线性阵列的打印头之间，图片的每列像素分配一个加热组件。当纸张和色带经过打印头时，加热组件通过打开和关闭状态来在需要染料的区域加热色带，使染料从色带扩散到纸张。对几种染料中的每一种重复此过程。由于较高的温度会导致更多的染料被转印，因此可以控制在每个网格位置沉积的每种染料的量，从而产生连续

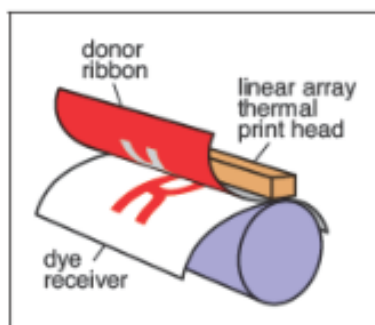



图 3.6: 热敏染料转印打印机的运行机制。

的颜色范围。打印头中加热组件的数量在页面横向方向上确定了固定的分辨率，但沿页面的分辨率由加热、冷却速率与纸张速度的关系决定。

与显示器不同，打印机的分辨率是根据像素密度而不是像素总数来描述的。因此，如果热敏染料转印打印机的打印头上每英寸的加热元素间距为 300，则其整个页面的分辨率为每英寸 300 个像素 (ppi)。如果选择沿页面的分辨率相同，我们可以简单地说打印机的分辨率为 300ppi。将点放在每英寸 1200 个网格点上的喷墨打印机被描述为分辨率为每英寸 1200 个点 (dpi)。由于喷墨打印机是二值设备，因此至少出于两个原因，它需要更精细的网格。由于边缘是突兀的黑/白边界，因此需要非常高的分辨率来避免出现锯齿 (stair-stepping/aliasing) (详见第 9.3 节)。打印连续色调图像时，需要高分辨率通过打印称为半色调的不同密度点图案来模拟中间色。

 **笔记** 术语“DPI”经常被用于表达“PPI”的含义，但“DPI”应被用于涉及二值设备的情况，而“PPI”应被用于涉及连续色调设备的情况。

### 3.1.3 输入设备

光栅图像不能凭空产生，并且任何不是由算法产生的图像都需要通过光栅输入设备记录 (通常是相机或扫描仪)。即使是 3D 场景渲染而成的图像也通常使用照片作为纹理映射 (详见第 11 章)。光栅输入设备需要对每个像素进行光记录，并且 (就像输出设备一样) 它们通常基于传感器阵列。

数码相机就是一种二维阵列输入设备。相机中的图像传感器是一个拥有光敏像素网格的半导体设备。两种常见的阵列类型是电荷耦合元件 (CCDs, charge-coupled devices) 和互补金属氧化物半导体 (CMOS, complimentary metal-oxide-semiconductor) 图像传感器。相机的镜头将要被照相的景物的图片传感器上，然后每个像素记录落在其上的光能，最终产生构成输出图像的数字 (图 3.7)。就像颜色通过红、绿、蓝子像素进行显示一样，许多相机通过使用“色彩滤镜阵列” (color-filter array) 或“马赛克” (mosaic) 来让每个像素只能感知红光、绿光或蓝光，而图像处理软件则在称为“去马赛克” (demosaicking) 的过程中填充缺失值 (图 3.8)。

其他相机使用 3 个独立阵列或阵列中的 3 个独立层级来在每个像素上记录独立的红、绿、蓝值，直接产生可用的彩色图片而无需更进一步地处理。相机分辨率由阵列中像素的固定数量决定，并且通常被描述为像素总数：一台阵列有 3000 列和 2000 行的相机产生分辨率 3000\*2000



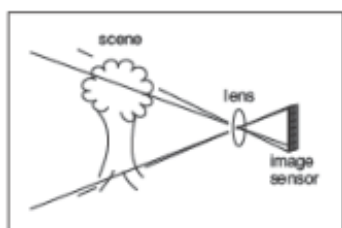


图 3.7: 数码相机的运行机制。



图 3.8: 许多彩色数字相机使用类似于上图所示的“拜尔马赛克” (Bayer mosaic) 的色彩滤镜阵列。每个像素记录红光、绿光或蓝光。

的图像，其中有六百万像素，称为“6 兆像素” (MP, megapixel) 相机。需要注意马赛克传感器并不记录一幅完整的图像，因此记录相同像素数但具有独立红色、绿色和蓝色记录值的相机比使用马赛克传感器的相机记录更多有关图像的信息。

**笔记** 买相机的人通常使用兆 (mega) 来表示  $10^6$ ，而不是兆字节 (megabytes) 代表的数量级  $2^{20}$ 。

平板扫描仪也记录每个像素网格的红色、绿色和蓝色值。但与热敏染料转印打印机一样，它使用一维阵列横扫正在被扫描的页面，每秒进行多次记录 (图 3.9)。页面的横向分辨率由阵列的尺寸确定，并且页面的纵向分辨率由记录频率和扫描头移动的速度确定。彩色扫描仪具有  $3 \times n_x$  阵列，其中  $n_x$  是页面的横向像素数，包括三行由红色、绿色和蓝色滤镜覆盖 (的子像素)。在记录三种颜色的时间之间有适当的延迟，这允许在每个网格点进行三个独立的颜色记录。与连续色调打印机一样，扫描仪的分辨率以每英寸像素数 (ppi) 为单位进行报告。

**笔记** 由于许多扫描仪可以通过内置转换产生其他分辨率的图像，扫描仪的分辨率有时被称为“光学分辨率”。

有了这些关于我们的图像来自哪里以及它们将去哪里的具体信息，我们现在将更抽象地讨论图像，就像我们在图形算法中使用它们一样。

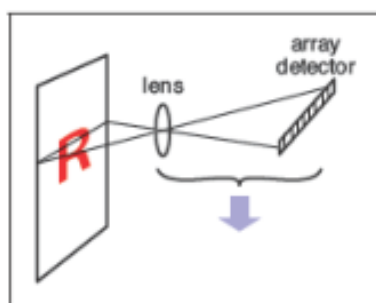



图 3.9: 平板扫描仪的运行机制。



## 3.2 图像、像素和几何


我们知道光栅图像是一个大的像素阵列，每个像素都存储了图像在该网格点的信息。我们也看到了数种输出设备如何处理我们发给它的图像，以及输入设备如何从物理世界中由光形成的图像中获取它们。但是对于计算机中的计算，我们需要一个独立于任何设备细节的方便抽象，我们可以用它来推理如何生成或解释存储在图像中的值。

当我们记录或再现图像时，它们采取光能的二维分布的形式：显示器发出的光作为位置在显示器表面上的函数；落在相机图像传感器上的光作为传感器平面上位置的函数；反射率 (reflectance)，或反射光的分数（相对于吸收的），作为一张纸上位置的函数。因此，在物理世界中，图像是在二维区域（几乎总是矩形）上定义的函数。因此，我们可以将图像抽象为函数

 **笔记** “像素不是小正方形！”——Alvy Ray Smith (1995)


$$I(x, y) : R \rightarrow V,$$

其中  $R \subset \mathbb{R}^2$  是一个矩形区域，而  $V$  是一个可能像素值的集合。最简单的例子是一个理想化的灰度图像，其矩形中的每个点只有亮度（而不是颜色），因此我们可以说  $V = \mathbb{R}^+$ （非负实数）。一幅理想化的彩色图像，每个像素都有红色、绿色、蓝色值，有  $V = (\mathbb{R}^+)^3$ 。我们将在下一小节讨论  $V$  的其他可能性。

 **笔记** 有没有其他不是矩形的光栅设备？

光栅图像与连续图像的这个抽象概念有什么关系？从具体的例子来看，照相机或扫描仪的一个像素是对该像素周围某个小区域的图像平均颜色的记录。一个具有红、绿、蓝子像素的显示器像素被设计成图像在像素面上的平均颜色由光栅图像中的相应像素值控制。在这两种情况下，像素值是图像颜色的局部平均值，它被称为图像的点样本。换句话说，当我们找到一个像素的值  $x$  时，它意味着“这个网格点附近的图像的值是  $x$ ”。图像作为函数的采样表示的想法将在第十章中进一步探讨。

一个平淡但重要的问题是像素在二维平面内如何定位。这只是一个约定俗成的问题，但创建一个一致的约定很重要！在本书中，光栅图像由一对  $(ij)$  索引，指示像素的列  $(i)$  和行  $(j)$ ，从左下角开始计数。如果图像具有  $n_x$  列、 $n_y$  行像素，则左下角的像素为  $(0, 0)$ ，右上角为像素  $(n_x - 1, n_y - 1)$ 。我们需要二维实际屏幕坐标来指定像素位置。我们将像素的采样点放置在整数坐标处，如图 3.10 中的  $4 \times 3$  屏幕所示。

 **笔记** 在一些 API 和许多文件格式中，图片的行从上到下组织，因此  $(0, 0)$  点在左上角。这是由于历史原因：模拟电视广播的行就是从顶端开始。

图像的矩形区域具有宽度  $n_x$  和高度  $n_y$ ，并且以此网格为中心，这意味着它延伸到每侧的最后一个采样点之外半个像素。所以  $n_x \times n_y$  图像的矩形域是：

$$R = [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5].$$

同样，这些坐标只是约定，但稍后在实现相机和视图变换时记住它们非常重要。

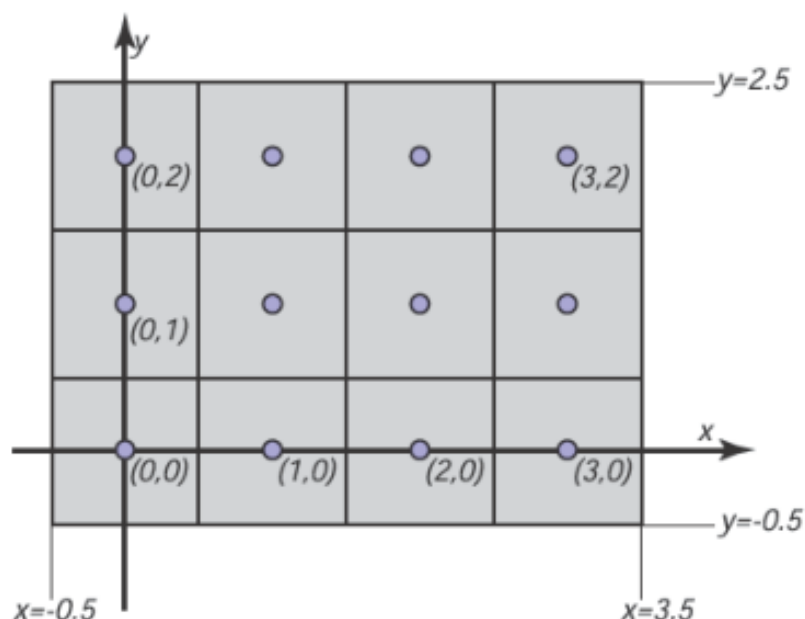




图 3.10: 一个拥有 4 个像素 \* 3 个像素的屏幕的坐标系。注意在部分 API 中  $y$  轴会指向下方。


 **笔记** 一些系统将坐标系移动了半个像素，以将采样点放置在整数之间的中间位置、将图像边缘放置在整数上。

### 3.2.1 像素值

到目前为止，我们已经用实数描述了像素的值，表示图像中某个点的强度（可能分别用于表示红色，绿色和蓝色）。这表明图像应该是一个由浮点数组成的阵列，其中每个像素存储一个（对于灰度或黑白图像）或三个（对于 RGB 彩色图像）32 位浮点数。当需要其精度和值范围时，有时会使用这种格式，但是图像具有大量的像素，并且用于存储和传输图像的内存和带宽总是稀缺的。在这种格式下，仅一张 1000 万像素的照片就会消耗大约 115MB 的 RAM。

 **笔记** 为什么是 115MB 而不是 120MB？

对于要直接显示的图像，需要的范围更小。虽然可能的光强度范围原则上是无限制的，但任何给定的设备都有一个明确的最大强度，所以在许多情况下，像素有一个有边界的范围是完全足够的，为了简单起见，通常取值为  $[0, 1]$ 。例如，8 位图像的可能值是 0,  $1/255$ ,  $2/255$ , ...,  $254/255$ , 1。用浮点数存储的图像，允许很大的数值范围，通常被称为高动态范围 (HDR) 图像，以区别于固定范围或用整数存储的低动态范围 (LDR) 图像。关于高动态范围图像的技术和应用的深入讨论，详见第 20 章。

 **笔记** 分母是 255 而非 256，这很不雅观，但能准确表示 0 和 1 是很重要的。

此处有一些像素格式以及它们的代表性应用：

- 1 位灰度图像——文本和其他不需要中间灰度的图像（需要高分辨率）
- 8 位 RGB 固定范围颜色（每个像素 24 位）——网络和电子邮件应用，消费者照片
- 8 或 10 位固定范围 RGB（每个像素 24 位或 30 位）——到计算机显示器的数码接口

- 12-14 位固定范围 RGB（每个像素 36-42 位）——专业摄影的原始相机图像
- 16 位固定范围 RGB（每个像素 48 位）——专业摄影与打印；用于固定范围图像处理的中间格式
- 16 位固定范围灰度图像（每个像素 16 位）——放射学和医学图像
- 16 位“半精度”浮点 RGB——HDR 图像；实时渲染的中间格式
- 32 位浮点 RGB——通用目的中间格式，用于 HDR 图像的软件渲染和处理。

减少用于存储每个像素的位数会导致图像中出现两种不同的伪影（artifact）类型（或人为引入的缺陷）。一、用固定范围的值对图像进行编码会产生裁剪（clipping），当其他比最大值更亮的像素被设置或裁剪为最大可表示的值。例如，一张阳光明媚的景物照片可能包括比白色表面亮得多的再反射；当图像被转换为固定范围显示时，这些反射会被剪掉（即使它们是由相机测量的）。第二，以有限的精度对图像进行编码会导致量化的伪影（quantization artifact），或带状现象（banding），当需要将像素值四舍五入到最接近的可表示值时，就会在强度或颜色上出现明显的跳跃。带状现象在动画和视频中特别隐蔽，在静止的图像中带状现象可能并不令人讨厌，但当它们来回移动时就变得非常明显。

### 3.2.2 显示器强度与伽马（Gamma）

所有的现代显示器都是使用数字输入作为一个像素的“值”，并将其转换为一个强度等级。真正的显示器在关闭时有一些非零强度，因为屏幕会反射一些光线。为了我们的目的，我们可以认为这是“黑色”，而显示器完全打开则是“白色”。我们假设对像素颜色的数字描述从 0 到 1 不等。黑色是 0，白色是 1，介于黑色和白色之间的灰色是 0.5。请注意，这里的“一半”指的是来自像素的物理光量，而不是外观。人类对强度的感知是非线性的，这不会成为本讨论的一部分；更多内容详见第 19 章。

要在显示器上产生正确的图像，有两个关键问题必须了解。第一个问题是，显示器对于输入是非线性的。例如，如果你给显示器的三个像素输入 0、0.5 和 1.0，显示的强度可能是 0、0.25 和 1.0（关闭、四分之一全开和全开）。作为这种非线性的近似描述，显示器通常以  $\gamma$ （“gamma”）值为特征。这个值是下面公式中的自由度：

$$\text{显示强度} = (\text{最大强度}) a^\gamma \quad (3.1)$$

其中  $a$  是介于 0 和 1 之间的输入像素值。比如，如果一台显示器有 2.0 的伽马值，并且我们输入  $a = 0.5$ ，则显示强度是四分之一最大可能强度（因为  $0.5^2 = 0.25$ ）。注意  $a = 0$  映射到 0 强度、 $a = 1$  映射到与  $\gamma$  值无关的最大强度。使用  $\gamma$  描述显示器的非线性度只是一个近似值；我们不需要很高的精度来估计设备的  $\gamma$  精度。衡量非线性度的一个很好的视觉方法是找到能产生介于黑白之间的中间强度的  $a$  值。这个  $a$  会如下所示：

$$0.5 = a^\gamma$$

如果我们能找到这样一个  $a$ ，就能通过在两边取对数来推断出  $\gamma$ ：

$$\gamma = \frac{\ln 0.5}{\ln a}$$

我们可以通过一个标准技术找到这个  $a$ ，即在“输入  $a$ ”的灰色像素正方形旁边显示一个黑白像素的棋盘图案（图 3.11），然后要求用户调整  $a$ （例如用滑块），直到两边的平均亮度一致。当你从远处看这幅图时（如果你是近视眼，也可以不戴眼镜），当  $a$  产生的中间强度介于黑色和白色之间时，图的两边看起来差不多。这是因为模糊的棋盘混合了偶数的白色和黑色像素，所以整体效果是介于白色和黑色之间的统一颜色。

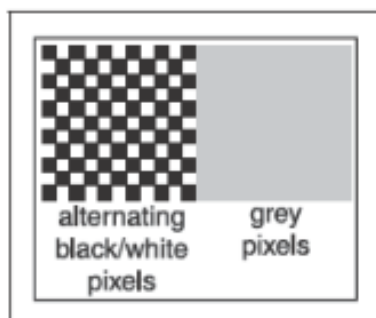



图 3.11: 交替的黑、白像素在一段距离外看是介于黑色、白色之间的中间值。显示器的伽马值可通过找到一个看起来与黑白图案有相同强度的灰色值来推断出来。

一旦我们知道了  $\gamma$ ，我们就可以对我们的输入进行伽马校正 (gamma correct)，使  $a = 0.5$  的值在显示时具有介于黑与白之间的中间强度。这是通过如下转换实现的；

$$a' = a^{\frac{1}{\gamma}}$$

当把以上公式代入等式 3.1，我们得到了：

$$\text{显示强度} = (a')^{\gamma} = (a^{\frac{1}{\gamma}})^{\gamma} (\text{最大强度}) = a (\text{最大强度})$$

 **笔记** 使用模拟接口的显示器在水平方向上更难迅速改变强度，因此水平黑白线的效果比棋盘格更好。

实数显示器的另一个重要特征是，它们采取量化的输入值。因此，虽然我们可以在浮点范围  $[0, 1]$  内操作强度，但显示器的详细输入是一个固定大小的整数。这个整数最常见的范围是 0-255，可以用 8 比特的存储空间来保存。这意味着  $a$  的可用值不是  $[0, 1]$  中的任何数值，而是：

$$a \text{ 的可能值} = \left\{ \frac{0}{255}, \frac{1}{255}, \frac{2}{255}, \dots, \frac{254}{255}, \frac{255}{255} \right\}$$


这意味着可能的显示强度值近似于以下值：

$$\left\{ M\left(\frac{0}{255}\right)^{\gamma}, M\left(\frac{1}{255}\right)^{\gamma}, M\left(\frac{2}{255}\right)^{\gamma}, \dots, M\left(\frac{254}{255}\right)^{\gamma}, M\left(\frac{255}{255}\right)^{\gamma} \right\}$$

其中  $M$  是最大强度。在需要控制确切强度的应用中，我们必须实际测量 256 种可能的强度，并且这些强度在屏幕上的不同点可能不同，特别是对于 CRT 设备。它们也可能随视角而变化。幸运的是，很少有应用需要如此精确的校准。

## 3.3 RGB 颜色

大多数计算机图形图像是根据红-绿-蓝 (RGB) 颜色定义的。RGB 颜色是一个简单的空间，允许直接转换给大多数计算机屏幕的控制。在本节中，我们会从用户角、以便利操作为目标讨论 RGB 颜色。第 18 章对颜色进行了更全面的讨论，但是 RGB 色彩空间的机制将使我们能够使用大多数图形学程序。RGB 色彩空间的基本思想是通过混合三个主要光源来显示颜色：一个红色、一个绿色和一个蓝色。灯光以加色方式混合。

 **笔记** 在小学你可能学习到了红、绿、蓝是三原色，并且例如，黄色 + 蓝色 = 绿色。这是减色混合，这与显示器中发生的更熟悉的加色混合有根本的不同。

在 RGB 加色混合中我们得到了 (图 3.12)：

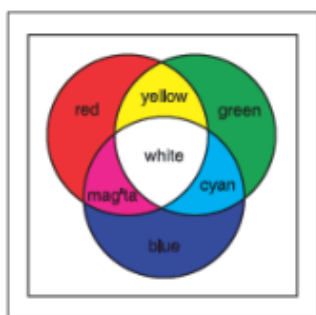


图 3.12: 红、绿、蓝三色的加色混合规则。

红色 + 绿色 = 黄色,

绿色 + 蓝色 = 青色,

蓝色 + 红色 = 品红,

红色 + 绿色 + 蓝色 = 白色

“青色”是一种蓝绿色，“品红”是一种紫色。

如果我们被允许将主灯从完全关闭（用像素值 0 表示）调至完全打开（用 1 表示），我们就可以创造出所有可以在 RGB 显示器上显示的颜色。红色、绿色和蓝色的像素值创建了一个三维的 RGB 颜色立方体，它有一个红色、一个绿色和一个蓝色轴。轴的最小坐标值范围是 0 到 1。图 3.13 是彩色立方体的图示。

立方体角落的颜色是：



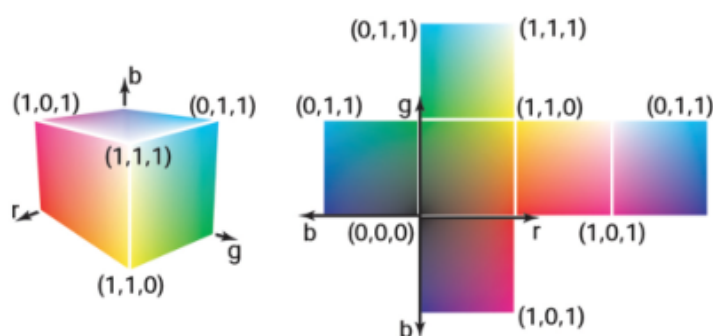


图 3.13: RGB 颜色立方体和其完全展开的面。任何 RGB 颜色都是立方体上的一个点。

黑色 =  $(0, 0, 0)$ ,

红色 =  $(1, 0, 0)$ ,

绿色 =  $(0, 1, 0)$ ,

蓝色 =  $(0, 0, 1)$ ,

黄色 =  $(1, 1, 0)$ ,

品红 =  $(1, 0, 1)$ ,

青色 =  $(0, 1, 1)$ ,

白色 =  $(1, 1, 1)$

实际的 RGB 等级通常以量化的形式给出，就像第 3.2.2 节中讨论的灰阶 (grayscale) 一样。每个分量都用一个整数来指定。这些整数最常见的大小是每个分量一个字节，所以 RGB 的三个分量都是 0 到 255 之间的整数。这三个整数加起来占了三个字节，也就是 24 位。因此，一个拥有“24 位色彩”的系统，三原色中的每一种都有 256 个可能的级别。第 3.2.2 节中讨论的伽玛校正问题也分别适用于 RGB 的每个分量。


## 3.4 阿尔法合成

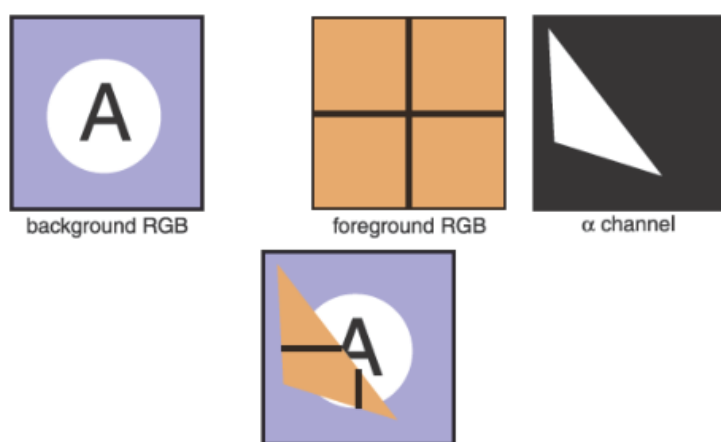
通常情况下，我们只想部分地覆盖一个像素的内容。一个典型的例子发生在合成中，我们有一个背景，想在上面插入一个前景图像。对于前景中不透明的像素，我们只需替换背景像素。对于完全透明的前景像素，我们不改变背景像素。对于部分透明的像素，必须小心谨慎。当前景物体有部分透明区域时，如玻璃，就会出现部分透明像素。但是，前景和背景必须混合的最常见的情况是，前景物体只覆盖了部分像素，要么是在前景物体的边缘，要么是有子像素孔，如远处的树叶之间。

如果我们想在背景物体上混合一个前景物体，最重要的信息是像素覆盖率 (pixel coverage)，它告诉我们前景层所覆盖的像素的比例。我们可以称这个分数为  $\alpha$ 。如果我们想要将一个前景色  $c_f$  合成到背景色  $c_b$  上，而且被前景覆盖的像素比例是  $\alpha$ ，那我们就可以用下面这个公式：

$$c = \alpha c_f + (1 - \alpha)c_b \quad (3.2)$$

对于一个不透明的前景层，这个公式的解释是前景物体覆盖了像素矩形内的比例为  $\alpha$  的区域，而背景物体覆盖了剩余的区域，也就是比例为  $(1 - \alpha)$  的区域。对于透明层（想想在玻璃或描图纸上画的图像，使用半透明的油漆），这个公式的解释是前景层阻挡了从背景透过来的那部分光线（比例为  $(1 - \alpha)$ ），并贡献了一部分（比例为  $\alpha$ ）自己的颜色来替代被移除的部分。图 3.14 显示了一个使用公式 3.2 的例子。

 **笔记** 由于前景层和背景层的权重相加为 1，如果前景层和背景层的颜色相同，颜色就不会改变。



**图 3.14:** 上图是一个使用公式 3.2 的合成示例。前景图像在被放在背景图像之上之前，实际上是被  $\alpha$  通道裁剪过的。合成结果显示在底部。

一个图像中所有像素的  $\alpha$  值可能被存储在一个单独的灰度图像中，然后被称为阿尔法遮罩 (alpha mask) 或透明度遮罩 (transparency mask)。或者信息可以被存储为 RGB 图像中的第四个通道，在这种情况下，它被称为阿尔法通道 (alpha channel)，因此图像可以被称为 RGBA 图像。对于 8 位图像，每个像素占 32 位，这在许多计算机架构中是一个大小方便的数据块。

尽管公式 3.2 常被使用，但有一些情况下  $\alpha$  会有其他不同的用法。(Porter & Duff, 1984)。

### 3.4.1 图像存储

大多数 RGB 图像格式的红、绿、蓝三色通道各使用 8 比特。这样一来，一张 100 万像素的图像大约需要 3 兆字节的原始信息。为了减少存储需求，大多数图像格式允许某种压缩。在高层次上，这种压缩是无损 (lossless) 的或有损 (lossy) 的。在无损压缩中，没有信息被丢弃，而在有损系统中，一些信息会不可恢复地丢失。流行的图像存储格式包括：

- **jpeg**. 这种有损格式是根据人类视觉系统的阈值来压缩图像块。这种格式对自然图像很有效。
- **tiff**. 这种格式最常用于保存二进制图像或无损压缩的 8 位或 16 位 RGB，尽管还有许多其他选择。

- ppm. 这种非常简单的无损、未压缩的格式最常用于 8 位 RGB 图像，尽管还有许多其他选择。
- png. 这是一套无损格式，有一套很好的开放源码管理工具。

由于压缩和变体的原因，为图像编写输入/输出例程可能会涉及到。幸运的是，人们通常可以依靠库中的例程来读写标准文件格式。对于快速、肮脏的应用来说，简单的价值高于效率，一个简单的选择是使用原始的 PPM 文件，这通常可以简单地通过将存储在内存中的图像的数组转储到一个文件中来编写，并在头部追加适当的头文件。

## 3.5 FAQ

### 3.5.1 为什么他们不把显示器做成线性的，而避免所有这些伽马业务？

理想情况下，显示器的 256 种可能的强度看起来应该是均等分隔的，而不是按照能量线性分隔。因为人类对强度的感知本身就是非线性的，伽玛值在 1.5 和 3 之间（取决于观看条件），会使强度在主观上近乎均匀。这样一来，伽玛就成了一种特性。否则，制造商会把显示器做成线性的。


## 3.6 练习

通过拍摄自然图像（最好是扫描的照片，而不是可能已经应用了拜尔马赛克的数码照片），并创建一个由红/绿/蓝通道交错组成的灰度图像，来模拟从拜尔马赛克获得的图像。这模拟了数码相机的原始输出。现在从该输出创建一个真正的 RGB 图像，并与原始图像进行比较。


## 第4章 光线追踪

计算机图形学的基础任务之一就是渲染三维物体：取一个由许多几何物体在三维空间中排列组成的场景，并计算出一个从一个特定视角看这些物体的二维图像。几个世纪以来，建筑师和工程师都是这样做的，他们绘制图纸，向他人传达他们的设计。

从根本上说，渲染是一个将一系列物体作为输入，并产生一个像素阵列的过程。无论如何，渲染涉及到考虑每个物体对每个像素的贡献，它可以以两种常见的方式规划。在物序渲染（object-order rendering）中，每个物体依次被处理，并且对于每个物体，其影响的每个像素都会被找到并被更新。在图序渲染中（image-order rendering），每个像素依次被处理，并且对于每个像素，所有影响它的物体都会被找到再计算像素值。你可以从循环嵌套的角度来考虑区别：在图序渲染中，循环“对于每个像素”在嵌套的外面，而在物序渲染中，循环“对于每个物体”在嵌套的外面。

 **笔记** 如果输出是矢量图像而不是光栅图像，渲染就不会涉及到像素，但我们在本书中将会假定输出光栅图像。

图序渲染和物序渲染可以计算出完全一样的图像，但它们计算出的特效不同，性能特点也大相径庭。在我们讨论完这两种方法后，我们将在第9章探讨这两种方法的比较优势，但一般来说，图像顺序渲染的工作方式更简单，可以产生的特效也更灵活，但通常（尽管不一定）需要更多的执行时间来产生一个类似的图像。

 **笔记** 在光线追踪器中，很容易计算出准确的阴影和反射，而这在物序渲染框架中是很棘手的。

光线追踪是一种渲染3D场景的图序渲染算法，我们会先讨论它，因为无需开发用于物序渲染的数学机器就有可能得到一个工作的光线追踪器。

### 4.1 基础光线追踪算法

### 4.2 透视图

### 4.3 计算视线

### 4.4 光线相交

### 4.5 阴影

### 4.6 历史笔记

## 第 5 章 表面着色

### 5.1 点状光源

### 5.2 基本反射模型

### 5.3 环境照明



## 第6章 线性代数

图形程序中最通用的工具恐怕就是改变或转换点和向量的矩阵了。在接下来的章节中，我们将学习如何将一个向量表示为一个只有一列的矩阵，并且通过矩阵乘法用不同的方式表示该向量。我们还将描述如何使用这种乘法来完成对向量的改变，比如缩放、旋转和平移。在这一章中，我们从几何学的角来回顾线性代数基础，专注于直感和在二维、三维上行之有效的算法。

对自己线性代数知识掌握有信心的读者可跳过这个章节，尽管如此，这里还是有些小道消息可能会对你有所启发，例如行列式的发展以及对奇异值和特征值分解的讨论等。

### 6.1 行列式

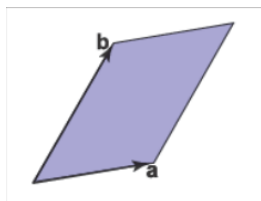


图 6.1: 这个平行四边形的带符号面积是  $|ab|$ ，并且在这种情况下面积为正。

我们通常认为行列式是在线性方程的解中出现的。然而，在此处，我们有意将行列式视为另一种向量相乘的方式。对于二维向量  $a$  和  $b$ ，行列式  $|ab|$  是  $a$  和  $b$  形成的平行四边形的面积（图 6.1）。这是一个带符号的面积，并且当  $a$  和  $b$  右旋时符号为正，左旋则为负，也就是  $|ab| = -|ba|$ 。在二维中，我们可以把“右旋”理解为逆时针旋转第一个向量至与第二个向量形成最小的角。三维中的行列式则必须同时计算三个向量。对于三个三维向量  $a$ 、 $b$  和  $c$ ，行列式  $|abc|$  是这三个向量形成的平行六面体（三维平行四边形；一个修剪的三维盒子）的带符号体积（图 6.2）。

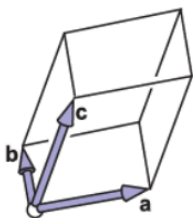


图 6.2: 所示平行六面体的带符号体积由行列式  $|abc|$  表示，在这种情况下，体积为正，因为这个向量以右旋作为基准。

要计算一个二维行列式，首先需要确定它的一些性质。我们注意到对平行四边形的其中一条边进行缩放，与它面积缩放的比例是相同的（图 6.3）：

$$|(ka)b| = |a(kb)| = k|ab|$$

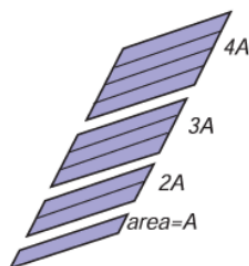


图 6.3: 沿一个方向缩放平行四边形会以相同比例改变面积。

此外，“修剪”一个平行四边形不会改变它的面积（图 6.4）：

$$|(a + kb)b| = |a(b + ka)| = |ab|$$

最终我们发现，行列式拥有如下特性：

$$|a(b + c)| = |ab| + |ac| \quad (6.1)$$

如图 6.5 所示，我们可以滑动两个平行四边形中间的边来形成一个平行四边形而不改变原来两个平行四边形的面积。

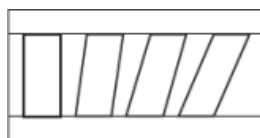


图 6.4: 剪切平行四边形不会改变其面积。这四个平行四边形底相同，因此面积也相同。

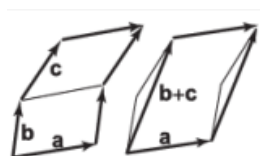


图 6.5: 方程式 6.1 后面的几何结构。左边两个平行四边形可以被剪切成右边的平行四边形。

现在让我们假设  $a$  和  $b$  的笛卡尔表示法：

$$\begin{aligned} |ab| &= |(x_a X + y_a Y)(x_b X + y_b Y)| \\ &= x_a x_b |XX| + x_a y_b |XY| + y_a x_b |YX| + y_a y_b |YY| \\ &= x_a x_b (0) + x_a y_b (+1) + y_a x_b (-1) + y_a y_b (0) \\ &= x_a y_b - y_a x_b \end{aligned}$$

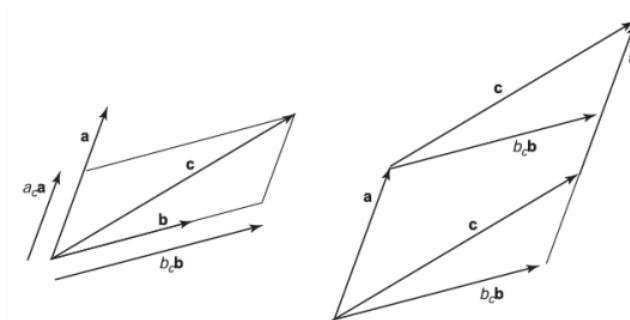
简而言之，对于任何向量  $v$ ，都有  $|vv|=0$ ，即平行四边形四边共线，因此没有面积。

在三维空间中，三个三维向量  $a$ 、 $b$  和  $c$  的行列式记作  $|abc|$ 。这些向量的笛卡尔表示对于平行六面体和平行四边形具有相似的规则，并且我们可以像对二维那样进行类似的拓展：

$$\begin{aligned}
 |abc| &= |(x_a X + y_a Y + z_a Z)(x_b X + y_b Y + z_b Z)(x_c X + y_c Y + z_c Z)| \\
 &= x_a y_b z_c - x_a z_b y_c - y_a x_b z_c + y_a z_b x_c + z_a x_b y_c - z_a y_b x_c
 \end{aligned}$$

可以看到, 这种方式下行列式的计算随着维度的升高变得越来越糟糕。我们将在章节 6.3.2 讨论易错更少的计算方法。

 **笔记** 例二



**图 6.6:** 左侧的向量  $c$  可以用两个基向量表示成  $a_c a + b_c b$ 。我们看到右侧, 由  $a$  和  $c$  形成的平行四边形是由  $b$  和  $a$  形成的平行四边形的剪切版本。

行列式自然产生于计算一个向量的表达式作为其他两个的线性组合——举个例子, 如果我们希望用向量  $c$  表示一个向量组合  $a$  和  $b$ :

$$c = a_c a + b_c b$$

参照图 6.6 得到:

$$|(b_c b) a| = |c a|$$

因为这些平行四边形只是彼此的剪切版本。解出  $b_c$  为

$$b_c = |c a| / |b a|$$

类比得出

$$a_c = |b c| / |b a|$$

这是克莱姆法则的二维版本, 我们将在章节 6.3.2 中再次探讨。

## 6.2 矩阵

矩阵是一个遵循某些算术规则的数值元素的数组。这是一个两行三列的矩阵示例:

$$\begin{bmatrix} 1.7 & -1.2 & 4.2 \\ 3.0 & 4.5 & -7.2 \end{bmatrix}$$

计算机图形学频繁地用到矩阵解决各种问题, 例如空间变换的表示。在我们的讨论中, 假设矩阵的元素都是实数。这个章节描述矩阵算法的力学原理和“方形”矩阵(译者注: 即方阵, 行数和列数相同的矩阵)的行列式。

### 6.2.1 矩阵计算

矩阵乘以一个常数会得到一个矩阵，其中每个元素都会乘以这个常数，例如：

$$2 \begin{bmatrix} 1 & -4 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 2 & -8 \\ 6 & 4 \end{bmatrix}$$

矩阵还可以逐个元素对应相加，例如：

$$\begin{bmatrix} 1 & -4 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & -2 \\ 5 & 4 \end{bmatrix}$$

对于矩阵乘法，我们将第一个矩阵的行与第二个矩阵的列“相乘”：

$$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{i1} & \cdots & a_{im} \\ \vdots & & \vdots \\ a_{r1} & \cdots & a_{rm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1j} & \cdots & b_{1c} \\ \vdots & & \vdots & & \vdots \\ b_{m1} & \cdots & b_{mj} & \cdots & b_{mc} \end{bmatrix} = \begin{bmatrix} p_{11} & \cdots & p_{1j} & \cdots & p_{1c} \\ \vdots & & \vdots & & \vdots \\ p_{i1} & \cdots & p_{ij} & \cdots & p_{ic} \\ \vdots & & \vdots & & \vdots \\ p_{r1} & \cdots & p_{rj} & \cdots & p_{rc} \end{bmatrix}$$

因此元素  $p_{ij}$  的乘积结果是：

$$p_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{im}b_{mj} \quad (6.2)$$

只有当左边矩阵的列数与右边矩阵的行数相同时，才能取得两个矩阵的乘积。例如：

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 6 & 7 & 8 & 9 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 12 & 17 & 22 & 27 \\ 24 & 33 & 42 & 51 \end{bmatrix}$$

矩阵乘法在大多数情况下都不满足交换律：

$$\mathbf{AB} \neq \mathbf{BA} \quad (6.3)$$

同样，如果  $\mathbf{AB} = \mathbf{AC}$ ，它并不一定满足  $\mathbf{B} = \mathbf{C}$ 。幸运的是，矩阵乘法满足结合律和分配律：

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$$

$$(\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$$

### 6.2.2 矩阵运算

我们想得到一个逆矩阵。首先我们知道实数  $x$  的倒数是  $1/x$ ，并且他们的乘积为 1。我们需要一个矩阵  $\mathbf{I}$  看作是一个“矩阵 1”。它只存在于方阵中，并称之为单位方阵；它由主对角线的 1 和其它地方的 0 组成。例如，4×4 的单位阵是：

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

矩阵  $\mathbf{A}$  的逆矩阵  $\mathbf{A}^{-1}$  是使得  $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$  的矩阵。例如：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} = \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix} \text{ 因为 } \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$\mathbf{A}^{-1}$  的逆记作  $\mathbf{A}$ ，因此  $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ 。两个矩阵乘积的逆是两个矩阵逆的乘积，但顺序交换。

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1} \quad (6.4)$$

章节 6.3 我们将回到这个计算逆的问题。

矩阵  $\mathbf{A}$  的转置  $\mathbf{A}^T$  拥有与之相同的数字，但行和列被交换了。如果我们把  $\mathbf{A}^T$  的条目记作  $a'_{ij}$ ，则：

$$a_{ij} = a'_{ji}$$

例如

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

两个矩阵乘积的转置遵循于方程 6.4 相似的规则：

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

方阵的行列式就是这个矩阵的行列式，被认为是一组向量。这个行列式与刚才讨论的矩阵运算有几个不错的关联，在这里列出以供参考：

$$|\mathbf{AB}| = |\mathbf{A}||\mathbf{B}| \quad (6.5)$$

$$|\mathbf{A}^{-1}| = \frac{1}{|\mathbf{A}|} \quad (6.6)$$

$$|\mathbf{A}^T| = |\mathbf{A}| \quad (6.7)$$

### 6.2.3 矩阵形式的向量运算

在图形中，我们使用一个方阵来变换一个表示为矩阵的向量。比如有一个二维向量  $a = (x_a, y_a)$  要绕原点旋转 90 度得到向量  $a' = (-y_a, x_a)$ ，可以用一个  $2 \times 2$  的矩阵和一个  $2 \times 1$  的矩阵来计算乘积，称为列向量。这个计算的矩阵形式是：

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_a \\ y_a \end{bmatrix} = \begin{bmatrix} -y_a \\ x_a \end{bmatrix}$$

我们也可以通过这个矩阵的转置左乘（“前相乘”）一个行向量得到相同的结果：

$$\begin{bmatrix} x_a & y_a \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -y_a & x_a \end{bmatrix}$$

现如今，使用列向量的后乘法是绝对标准的，但在很多早期的书籍和系统中，会看到行向量和前乘法。



我们也可以用矩阵形式来编码仅对向量的操作。如果我们把点积的结果看作是一个  $1 \times 1$  的矩阵，可以写作：

$$ab = a^T b$$

例如，如果我们取两个三维向量，会得到：

$$\begin{bmatrix} x_a & y_a & z_a \end{bmatrix} \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix} = [x_a x_b + y_a y_b + z_a z_b]$$

一组相关的向量乘积是两个向量的外积，可以表示成左侧是列向量，右侧是行向量的矩阵乘法： $\mathbf{a}\mathbf{b}^T$ 。结果是一个由  $a$  和  $b$  中所有条目对应的乘积构成的矩阵。对于三维向量，我们有：

$$\begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix} \begin{bmatrix} x_b & y_b & z_b \end{bmatrix} = \begin{bmatrix} x_a x_b & x_a y_b & x_a z_b \\ y_a x_b & y_a y_b & y_a z_b \\ z_a x_b & z_a y_b & z_a z_b \end{bmatrix}$$

用向量运算的角度考虑矩阵乘法通常是有用的。为了说明，我们使用三维中的例子，想象一个  $3 \times 3$  的矩阵，它是三个三维向量在两种方式下的集合：要么是三个列向量并排，要么是三个行向量堆叠形成。比如，一个矩阵向量乘法  $\mathbf{y} = \mathbf{A}\mathbf{x}$  的结果可以解释为一个向量，它的项是  $\mathbf{x}$  和  $\mathbf{A}$  行的点积。把它的行向量记作  $\mathbf{r}_i$ ，得到：

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} -\mathbf{r}_1- \\ -\mathbf{r}_2- \\ -\mathbf{r}_3- \end{bmatrix} \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix}$$

$$y_i = \mathbf{r}_i \cdot \mathbf{x}$$

或者我们也可以想象成  $\mathbf{A}$  的三列  $\mathbf{c}_i$  乘积的和，用  $\mathbf{x}$  的项加权：

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3 \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\mathbf{y} = x_1 \mathbf{c}_1 + x_2 \mathbf{c}_2 + x_3 \mathbf{c}_3$$


带着同样的想法，可以将矩阵与矩阵的乘积  $\mathbf{AB}$  理解为一个数组，其中包含  $\mathbf{A}$  的所有行和  $\mathbf{B}$  的所有列的对应点积（参看 6.2）；看作一个矩阵  $\mathbf{A}$  和  $\mathbf{B}$  所有列向量的乘积的集合，从左到右排列；看作一个  $\mathbf{A}$  的所有行向量与矩阵  $\mathbf{B}$  的乘积，从上到下堆叠；或看作  $\mathbf{A}$  的所有列和  $\mathbf{B}$  的所有行对应外积的和（练习 8）。

### 6.2.4 特殊类型的矩阵

单位矩阵是对角矩阵的一种，其中所有非零元素都沿着对角线出现。对角线由那些从左上角开始列索引与行索引计数相同的元素组成。

单位矩阵也具有与它的转置相同的性质，这种矩阵被称为对称矩阵。

单位矩阵也是一个正交矩阵，因为它的每一个列都被认为是一个向量，长度为 1，并且这些列彼此正交。各行也是如此（练习 2）。任何正交矩阵的行列式都是 +1 或 -1。

 **笔记** 正交矩阵的概念对应于标准正交基的概念，而不仅仅是一组正交向量——这是术语中的一个不幸的小差错。

正交矩阵有一个非常有用的性质：它们几乎是它们自己的逆。用一个正交矩阵乘以它的转置，可以得到恒等式：

$$\mathbf{R}^T \mathbf{R} = \mathbf{I} = \mathbf{R} \mathbf{R}^T \quad \text{正交 } \mathbf{R}$$

这很显然，因为  $\mathbf{R}^T \mathbf{R}$  的项就是  $\mathbf{R}$  的列之间的点积。非对角线上的项是正交向量之间的点积，对角线上的项是（单位长度）列与它们自身的点积。

例 3

矩阵

$$\begin{bmatrix} 8 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

是对角矩阵，因此是对称矩阵，但不是正交矩阵（列是正交的但不是单位长度）。

矩阵

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 9 & 7 \\ 2 & 7 & 1 \end{bmatrix}$$

是对称矩阵，但不是对角矩阵或正交矩阵。

矩阵

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

是正交矩阵，但不是对角矩阵或对称矩阵。

## 6.3 计算矩阵和行列式

## 6.4 特征式和矩阵对角式