实时渲染第四版

关于本书

英文名称：《Real-Time Rendering, Fourth Edition》

原作者：Tomas Akenine-Moller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, Sebastien Hillaire

翻译者：Morakito

版本：v1.0

时间：2023.11.20

内容简介：本翻译版涵盖了 RTR4 中的第 1 章-第 26 章中的内容。附录内容详见在线网站，参考文献目录详见参考文献网站和参考文献合集，已涵盖勘误内容（截止到 2023.10.30）。由于本人才疏学浅，翻译难免有误，望各位不吝惜指正。本翻译仅供交流学习，如有侵权，请联系删除。

相关链接

- 源文件仓库：https://github.com/Morakito/Real-Time-Rendering-4th-CN

- 在线网站：https://www.realtimerendering.com
- 勘误网站：https://www.realtimerendering.com/corrigenda.html
- 参考文件网站：https://www.realtimerendering.com/refs.html
- 参考文献合集仓库：https://github.com/QianMo/Real-Time-Rendering-4th-Bibliography-Collection
- wolai 文件：https://www.wolai.com/fkGSwxLu2pjWD7kiBY1V7W

序言

（下文来自毛星云的自序，共勉）

> All our dreams can come true, if we have thecourage to pursue them.

> 沃尔特·迪斯尼——我们所有的梦想都可以成真，只要我们有勇气去追求它们。

依稀记得那还是 F4 红遍大街小巷，满城都飘扬着《流星雨》的年代。

那个时候的电子游戏，无论是投币式的街机游戏，还是网吧里的《反恐精英》、《流星蝴蝶剑》、《仙剑奇侠传》、《星际争霸》、《帝国时代》等引领时代的游戏界的璀璨明珠，总能深深地吸引住每个纯真无邪孩童的心，绚烂的游戏画面总是让孩童们流连忘返。

那个时候，每次放学后唯一单纯的想法，就是悄悄溜到学校附近的网吧，和电脑游戏亲密接触。口袋里有邻花钱的时候就能玩上一会儿，没有零花钱的时候就痴痴地站在屏幕前面看别人操纵着荧幕前的剑侠闯荡世界。年少的我单纯地认为，游戏世界中存在着一个无比恢弘的世界，那是可以装下梦想的地方。应该是我对游戏的痴迷，对游戏开发梦想的虔诚，让我走向了研习游戏开发的这条道路。

还记得那个香樟树覆盖的夏天，年幼无知的我在一帮同学中吹牛说：我长大后，一定要自己开发出比这些还牛还要好玩的游戏。

现在想想，这几年走过的路途，真应了那句话，“现在的努力，都是为了小时候吹过的牛逼”。

这些年来，在学习游戏编程的道路上有过惊喜，有过坎坷，有过自豪，有过怅惘，走了不少弯路，也算是最终走上了正途，小有所成。于是，我单曲循环着五月天的《有

些事情现在不做一辈子都不会做了》，打开 Word，打开 Visual Studio，把自己这么多年来的游戏开发经验和心得用文字凝聚起来，开始为大家写这本书。

而这么一写，就是一整年。

经过一年夙兴夜寐，终于，赶在 22 岁生日之前，近百万字的书稿随着一声响指而初具雏形。

“谨以此书献给父母，因养育之恩无以回报。谨以此书献给母校南京航空航天大学和乌克兰国立航空航天大学，因赐予我一颗不甘平庸、上下求索的心。谨以此书献给所有怀揣游戏开发梦想的人们，因为，你们不是一个人在战斗。”

当在书稿的开头写下这三个“谨以”的时候，我终于开始觉得，这一年的夜以继日，这一年的披星戴月，都是值得的。

然而，因为岁月积累的关系，这本书中渗透的编程思想或许不能和编程界中的泰斗们同日而语。但是，我可以捂着胸口问心无愧地说，我把这些年自己悟出来的关于游戏编程的学习方法和真知灼见，毫无保留地呈现给了大家。大家能看到的眼前的这些句子和代码，全都是经过一遍又一遍的深思熟虑，一遍又一遍的修改，然后小心谨慎地敲出来的。

详细研究过游戏编程的朋友们都应该有这样的共识：“中国人写的书水平上不去，外国人写的书水平有了，但是翻译得往往都强差人意，理解不了”。也许正是这个原因，国内游戏编程的入门门槛一直很高，DirectX 一直被人们认为是很难学的。很多怀揣游戏开发梦想的热血青年们，信誓旦旦地开始着手学习游戏编程的时候，却被晦涩难懂的游戏编程教材拒之梦想门外，碰了一鼻子灰，从此和最初的梦想失之交臂。我想，这正是导致国产游戏业界的萎靡，国产游戏一直很难成长起来的原因之一。

在这样的环境的激励下，这本倾注我一年多心血的书出现了，它的创作初衷便是渴望能够改变这样的现状。

愿这本书，能帮到那些热爱游戏编程、怀揣游戏开发梦想，却苦于难以入门的人们，让他们少走弯路。

愿这本书，能为国产游戏、国产游戏引擎的崛起，开启一扇门，迎接新的黎明。

我有一个梦想，将来的某一天，大家都能玩到蕴含着中国上下五千年本土文化的优质游戏大作。

我有一个梦想，有一天，西游记能出 ACT，让老外去体会中国文化西游记中”斗战胜佛”的打击快感，那一定比西方的动作巅峰之作《战神》、《鬼泣》更加深邃。

我有一个梦想，有一天，上海滩能出沙盒游戏，而不是玩《GTA》感受美国梦，亦或是玩着《热血无赖》体验国外公司强行塞给我们的“中国文化”。

我有一个梦想，有一天，不少 3A 大作不需要汉化，因为是我们自己的游戏，配音是中文，文化也是中国的。

我有一个梦想，将来的某一天，国产游戏能像中国的其他产业一样，以一个领跑者的姿态，面对全世界，面对全宇宙，器宇轩昂，扬眉吐气。

这会是由我们一起去完成的梦想。

我等着我们的好消息。

浅墨 2013 年 5 月于乌克兰

希望我们可以一起努力，翻过那座山。

目录

Chapter 5 Shading Basics 着色基础

8.2 从场景到屏幕

8.2.1 HDR 显示编码

8.2.2 色调映射

色调再现变换

曝光

8.2.3 颜色分级

Chapter 9 Physically Based Shading 基于物理的着色

9.1 光的物理学

9.1.1 粒子

9.1.2 介质

9.1.3 表面

9.1.4 次表面散射

9.2 相机

9.3 The BRDF

9.4 光照（Illumination）

9.5 菲涅尔反射

9.5.1 外反射

9.5.2 典型的菲涅尔反射值

电介质的菲涅尔反射率

金属的菲涅尔反射率

半导体的菲涅尔反射值

水中的菲涅尔反射率

参数化的菲涅尔值

9.5.3 内反射

9.6 微观几何（Microgeometry）

Chapter 16 Polygonal Techniques 多边形技术

16.1 三维数据的来源

16.2 曲面细分和三角形划分

16.2.1 着色问题

16.2.2 边界开裂和 T 顶点

16.3 整合

16.3.1 合并

16.3.2 定向

16.3.3 实体性

16.3.4 法线平滑和折痕边缘

16.4 三角形扇，三角形带和三角形网格

16.4.1 三角形扇

16.4.1 三角形带

16.4.3 三角形网格

16.4.4 缓存无关的网格布局

16.4.5 顶点和索引缓冲区/数组

16.5 简化

16.5.1 动态简化

16.6 压缩和精度

Chapter 17 Curves and Curved Surfaces 曲线和曲面

17.1 参数化曲线

Chapter 26 RealTime Ray Tracing 实时光线追踪

Chapter 1 Introduction 简介

实时渲染是指在计算机上快速生成图像，它是计算机图形学中互动性最强的领域。屏幕上会显示一张图像，观察者在看到图像之后会做出一些反应和操作，这些反馈紧接着又会对下一张图像的生成产生影响。这个包含反应和渲染的循环会以一个很高的速度发生，以至于观察者根本意识不到自己正在观察一系列相互独立的图像，而是沉浸在这样一个动态的过程中。

通常会使用每秒显示的帧数（frames per second，FPS）或者赫兹（Hertz，HZ）来衡量图像显示的速率。如果图像以每秒一帧的速率进行显示，那么就几乎没有交互感可言，用户会意识到每一张新图像的到来，这是一个痛苦的过程。当图像显示速率达到 6 FPS 左右时，会逐渐开始增加交互性。电子游戏的目标是 30，60，72 或者更高的 FPS，在这样的图像显示速率下，用户可以将注意力集中在自身的行动和反应上。

电影放映机（movie projector）会以 24 FPS 的速率来进行显示，但是它会使用一个快门系统（shutter system）来将每帧重复显示 2–4 次，从而避免画面出现闪烁。这被称为刷新率（refresh rate），其单位是赫兹（Hz），它和上文中所说的显示速率（display rate）是两个概念。以刚才的电影放映机为例，一个可以每帧照亮三次的快门，其刷新率为 72 Hz。同样的，LCD 显示器的刷新率和显示速率也是两个不同的概念。

在显示器上观看以 24 FPS 出现的图片也许是可以接受的，但是更高的帧率可以有效降低最小反应时间。当显示延迟大于 15 毫秒的时候，就会对交互的流畅感产生干扰[1849]。举个例子，头戴的 VR 显示设备一般需要 90 FPS 来最小化延迟。

实时渲染也不仅仅只包含交互性，如果渲染速度是唯一的衡量标准的话，那么任何能够快速响应用户指令，并在屏幕上绘制图像的应用程序都符合这个条件。实时渲染通常指的是将三维场景渲染成二维图像。

交互性和三维场景是实时渲染的充分条件，但是第三个元素也已经逐渐成为了其定义的一部分，即图形加速硬件（graphics acceleration hardware）。许多人都认为，在 1996 年上市的 3Dfx Voodoo 1 图形加速卡是消费级显卡的开端[408]。随着这个市场的快速发展，如今每个电脑，平板和手机中都内置了相应的图形处理器。图 1.1 和图 1.2 展示了一些优秀的实时渲染案例，它们都使用了硬件加速来进行渲染：

图 1.1 《极限竞速 7》的游戏内画面。

图 2.2 《巫师 3》中的鲍克兰港。

图形硬件的进步推动了交互式计算机图形学领域的爆炸式发展。本文将重点关注用于提高渲染速度和图像质量的相关方法，同时也会介绍一些加速算法和图形 API 的特性，以及它们的局限性。我们不可能对每一个话题都进行深入，因此我们的目标是介绍关键性的概念和术语，介绍该领域中最健壮和最实用的算法，同时提供一些深入学习的方向指引。我们尝试提供一些工具，来帮助您更好地理解这个领域，同时希望这份尝试能够对得起您阅读本书所付出的时间和努力。

1.1 内容概述

下面是对各个章节的简要概述。

第 2 章–图形渲染管线（The Graphics Rendering Pipeline）。实时渲染的核心是一组操作步骤，它将场景描述作为输入，并将其转换为我们能够看得见的图像。

第 3 章–图形处理单元（The Graphics Processing Unit）。现代 GPU 使用固定功能单元和可编程单元的组合，来实现渲染管线的各个阶段。

第 4 章–变换（Transforms）。变换是用于控制物体位置、朝向、尺寸、形状以及相机位置、相机视角的基本工具。

第 5 章–着色基础（Shading Basics）。我们首先会讨论材质和光源的定义，以及它们如何用于实现所需的表面外观，无论这个表面是写实的还是风格化的。该章节还会介绍其他与外观表现相关的话题，例如使用抗锯齿、透明度和伽马矫正来获得更高的图像质量。

第 6 章–纹理（Texturing）。实时渲染中最强大的工具之一，就是能够快速访问图像，并将其显示在表面上。这个过程叫做纹理化，有各种各样来实现它的方法。

第 7 章–阴影（Shadows）。在场景中添加阴影效果可以增强画面的真实感和表现力。我们会在这个章节中介绍几种目前比较流行的、能够快速计算阴影的方法。

第 8 章–光和颜色（Light and Color）。在我们实现基于物理的渲染之前，我们首先需要了解如何对光和颜色进行量化。并且在我们执行基于物理的渲染之后，我们还需要将最终结果转换为可以显示的数值，并考虑屏幕属性和观察环境对它的影响。本章节将会介绍以上两个主题。

第 9 章–基于物理的着色（ Physically Based Shading）。我们将从头开始建立起对基于物理的着色模型的理解。在本章节中，我们首先会从潜在的物理现象开始，介绍一系列包含各种渲染材质的着色模型，最后介绍这些材质的混合方法和过滤方法，从而避免材质相互重叠并保持表面外观。

第 10 章–局部光照（Local Illumination）。本章节研究了刻画复杂光源的算法。我们在进行表面着色的时候，需要考虑到光线是从某些物理对象中发射出来的，这些物体具有独特的形状。

第 11 章–全局光照（Global Illumination）。本章节研究了模拟光线和场景多次相交的算法，这可以大大增强图像的真实感。我们还会研究环境光遮蔽（ambient occlusion）和定向遮蔽（directional occlusion），介绍在漫反射表面和镜面表面上渲染全局光照效果的方法，以及一些很有前景的统一方法。

第 12 章–图像空间特效（Image-Space Effects）。图形硬件擅长进行高速的图像处理。在本章节中，我们首先会讨论图像过滤技术和重投影（reprojection）技术，然后我们将介绍几种常见的后处理特效：镜头光晕（lens flare），动态模糊（motion blur）和景深（depth of field）。

第 13 章–超越多边形（Beyond Polygons）。对于描述物体形状而言，三角形并不总是速度最快或者效果最逼真的方法，一些诸如基于图像，点云（point clouds），体素（voxels）或者其他样本集合的替代方法，都有着各自的独特优势。

第 14 章–体积和半透明渲染（Volumetric and Translucency Rendering）。本章节的重点是体积材质的表示，以及体积材质与光线相互作用的理论与实践。它可以模拟的现象有很多，大到大范围的大气效果，小到毛发纤维的光线散射等。

第 15 章–非真实感渲染（Non-Photorealistic Rendering）。尝试将一个场景渲染得更加逼真，只是众多渲染方式中的一种，除此之外还有很多其他风格，比如卡通渲染和水彩效果等。同时我们还会对直线和文本生成技术进行讨论。

第 16 章–多边形技术（Polygonal Techniques）。几何数据的来源有很多，有时候我们需要对其进行修正，才能更好更快的进行渲染。本章节讨论了有关多边形数据表示和多边形数据压缩的相关内容。

第 17 章–曲线和曲面（Curves and Curved Surfaces）。使用一些更加复杂的表面表达方式可以提供很多优势，例如可以在渲染质量和渲染速度之间进行权衡，可以具有更加紧凑的表示以及可以生成更加平滑的表面。

第 18 章–管线优化（Pipeline Optimization）。对于一个已经使用了高效算法，并且正在运行的应用程序，我们还可以使用各种优化技术来进一步提高它的运行效率。本章节讨论了如何找应用程序的性能瓶颈（bottleneck）并对其进行处理，以及多线程优化等问题。

第 19 章–加速算法（Acceleration Algorithms）。当我们让一个程序成功运行起来之后，下一步就是让它运行得更快。本章节讨论了各种各样的剔除技术（culling），以及层次细节（LOD，level of detail）等技术。

第 20 章–高效着色（Efficient Shading）。场景中的大量光源会严重降低性能表现；在无法确定一个片元最终是否可见之前对其进行着色计算，也是重要的性能开销来源（过度绘制）。本章节中我们会介绍很多方法，来解决着色过程中可能会出现低效率问题。

第 21 章–虚拟现实和增强现实（Virtual and Augmented Reality）。这些领域有着特殊的挑战和技术，例如以一个较高且稳定的帧率，来高效的生成逼真的图像。

第 22 章–相交测试技术（Intersection Test Methods）。相交测试对于渲染，用户交互和碰撞检测而言十分重要。在本章节中，我们介绍了许多用于几何相交测试的高效算法，并对其进行了深入讨论。

第 23 章–图像硬件（Graphics Hardware）。本章节对一些硬件组件进行了重点关注，例如颜色深度，帧缓冲以及其他基本结构类型等。同时提供了一个具有代表性的 GPU 案例学习。

第 24 章–展望未来（The Future）。预测未来的技术发展趋势，以及对读者的建议。

我们还完成了一章有关碰撞检测（Collision Detection）的内容，以及一章有关实时光线追踪的内容（Real–Time Ray Tracing），这里限于篇幅，我们将其放在了配套网站 realtimerendering.com 上，你可以在这里下载到相关内容，同时网站上还有关于线性代数以及三角学的附录内容。

1.2 符号和定义

首先我们需要解释一下本书中所用到的数学符号。如果你想对本小节或者本书中的术语做更加深入的了解，你可以在 realtimerendering.com 上获得我们的线性代数附录。

1.2.1 数学符号

表 1.1 中总结了大部分我们将用到的数学符号，这里我们将对其中的一些概念进行详细描述。

请注意表格中的规则也有一些例外，这主要是因为着色方程中所使用的符号，在相关文献中已经非常完善且统一了，例如 L 代表辐射度（radiance），E 代表辐照度（irradiance），σ_s 代表散射系数（scattering coefficient）等。

角度和标量都是取自于 $\mathbb{R}$ （实数集），即它们都是实数。向量和点使用粗体的小写字母进行表示，其各个分量的表示如下：

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

该向量以列向量的形式给出，这种表达形式在计算机图形学中被广泛使用。在本书中的某些地方我们会使用行向量 (v_x, v_y, v_z) 来表示向量或者点，之所以不使用形式更加正确的 $(v_x, v_y, v_z)^T$，只是因为前者阅读起来更加容易。

类型	数学标记	例子
角度（angle）	小写希腊字母	$\alpha_i, \phi, \rho, \eta, \gamma_{242}, \theta$
标量（scalar）	小写斜体	a, b, t, u_k, v, w_{ij}
向量，点（vector，point）	小写粗体	$\mathbf{a}, \mathbf{u}, \mathbf{v}_s, \mathbf{h}(\rho), \mathbf{h}_z,$
矩阵（matrix）	大写粗体	$\mathbf{T}(\mathbf{t}), \mathbf{X}, \mathbf{R}_x(\rho),$
平面（plane）	π :一个向量和一个标量	$\pi : \mathbf{n} \cdot \mathbf{x} + d = 0,$ $\pi_1 : \mathbf{n}_1 \cdot \mathbf{x} + d_1 = 0$
三角形（triangle）	$\triangle$ + 三个顶点	$\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2, \triangle \mathbf{cba}$
线段（line segment）	两个顶点	$\mathbf{uv}, \mathbf{a}_i\mathbf{b}_j$
几何实体（geometry entity）	大写斜体	A_{OBB}, T, B_{AABB}

表 1.1 本书中所用的公式总结

使用齐次（homogeneous）坐标表示法，一个坐标可以使用四个值来进行表示，即 $\mathbf{v} = (v_x \quad v_y \quad v_z \quad v_w)^T$，其中 $\mathbf{v} = (v_x \quad v_y \quad v_z \quad 0)^T$ 代表一个向量，$\mathbf{v} = (v_x \quad v_y \quad v_z \quad 1)^T$ 代表一个点。有时我们会使用只包含三个分量的向量或者点，我们会尽量避免关于使用何种表示类型的歧义。对于矩阵运算而言，使用相同符号形式的点和向量是十分有用的，更多内容详见第 4 章中有关变换的部分。在某些算法中，使用数字索引来代替 x, y, z 下标会很方便，例如 $\mathbf{v} = (v_0 \quad v_1 \quad v_2)^T$。所有这些有关向量和点的符号规则，同样也适用于只包含两个分量的向量，在二维向量的情况中，我们会直接跳过向量的第三个分量。

矩阵值得我们多进行一些解释。常用的矩阵尺寸包括 2×2，3×3，4×4，这里我们将以 3×3 矩阵 $\mathbf{M}$ 为例，来回顾矩阵的访问方式，其他尺寸矩阵的操作也类似。矩阵 $\mathbf{M}$ 的（标量）元素记为 $m_{ij}, 0 \leq (i, j) \leq 2$，其中的 i 代表该元素所在的行，j 代表该元素所在的列，如方程 1.1 所示：

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{02} \end{pmatrix} \tag{1.1}$$

方程 1.2 中的符号也代表一个 3×3 矩阵，这种表达形式用于从矩阵 $\mathbf{M}$ 中分离向量： $\mathbf{m}_{,j}$ 代表第 j 个列向量； $\mathbf{m}_{i,}$ 代表第 i 个行向量（以列向量形式进行表示）。与向量与点一样，如果使用起来更加方便的话，列向量也可以使用 x, y, z, w 来进行索引：

$$\mathbf{M} = \begin{pmatrix} \mathbf{m}_{,0} & \mathbf{m}_{,1} & \mathbf{m}_{,2} \end{pmatrix} = \begin{pmatrix} \mathbf{m}_x & \mathbf{m}_y & \mathbf{m}_z \end{pmatrix} = \begin{pmatrix} \mathbf{m}_{0,}^T \\ \mathbf{m}_{1,}^T \\ \mathbf{m}_{2,}^T \end{pmatrix} \tag{1.2}$$

我们使用 $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ 来表示一个平面，它包含了定义平面所需的数学公式，即平面的法线 $\mathbf{n}$ 以及标量 d 。其中平面法线是一个描述平面朝向的向量，对于更一般的表面（例如曲面），法线描述了表面上某个特定点的朝向；而对于平面而言，平面上所有点都具有相同的法线。 π 通常被用作为代表平面的数学符号，平面 π 会将空间一分为二，其中位于正半空间中的点满足 $\mathbf{n} \cdot \mathbf{x} + d > 0$ ；位于负半空间中的点满足 $\mathbf{n} \cdot \mathbf{x} + d < 0$ 。剩下所有的点都位于平面 π 上。

一个三角形可以使用三个顶点 $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ 来进行定义，记为 $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ 。

表 1.2 展示了其他的一些数学运算符及其符号表示，你可以在配套网站 realtimerendering.com 上找到线性代数附录，其中包含了点乘、叉乘、行列式以及模长操作符的相关解释。转置操作符可以将一个列向量转换为一个行向量，反之亦然，这样我们就可以将一个列向量写在一行中，例如 $\mathbf{v} = \begin{pmatrix} v_x & v_y & v_z \end{pmatrix}^T$ 。表中的第四个操作符在《*Graphics Gems IV*》[735]中有详细介绍，这是一个作用于二维向量的一元操作符，它作用于向量 $\mathbf{v} = \begin{pmatrix} v_x & v_y \end{pmatrix}^T$ 上，并会生成一个与其垂直的向量，例如 $\mathbf{v}^\perp = \begin{pmatrix} -v_y & v_x \end{pmatrix}^T$ 。

序号	数学标记	说明
1	$\cdot$	点乘
2	$\times$	叉乘
3	$\mathbf{v}^T$	向量 $\mathbf{v}$ 的转置
4	$^\perp$	一元操作符，垂直点乘操作符
5	$\lvert \cdot \rvert$	矩阵的行列式
6	$\lvert \cdot \rvert$	标量的绝对值

7	$\|\cdot\|$	范数（长度和模长）
8	x^+	将 x 的最小值限制在 0
9	$x^\mp$	将 x 限制在 0 到 1 之间
10	$n!$	阶乘
11	$\binom{n}{k}$	二项式系数

表 1.2：一些数学操作符的标记

我们使用 $|a|$ 来表示标量 a 的绝对值，使用 $|\mathbf{A}|$ 来表示矩阵 $\mathbf{A}$ 的行列式。有时我们还会使用 $|\mathbf{A}| = |\mathbf{a} \quad \mathbf{b} \quad \mathbf{c}| = \det(\mathbf{a}, \mathbf{b}, \mathbf{c})$ 这种表示方式，其中 $\mathbf{a}, \mathbf{b}, \mathbf{c}$ 分别是矩阵 $\mathbf{A}$ 的列向量。

第 8 和第 9 个操作符是限制操作符（clamp），它在着色计算中经常使用。操作符 8 会将输入值的负数部分限制到 0：

$$x^+ = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{otherwise}, \end{cases} \tag{1.3}$$

操作符 9 则会将输入值限制在 0 到 1 之间：

$$x^\pm = \begin{cases} 1, & \text{if } x \geq 1, \\ x, & 0 < x < 1, \\ 0, & \text{otherwise}, \end{cases} \tag{1.4}$$

操作符 10 是阶乘（factorial）操作符，其定义如下所示，请注意 $0! = 1$：

$$n! = n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1 \tag{1.5}$$

操作符 11 是组合数，也叫做二项式系数，其定义方程 1.6：

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{1.6}$$

除此之外，我们一般将 $x = 0$，$y = 0$，$z = 0$ 这三个平面叫做坐标平面（coordinate planes）或者轴对齐平面（axis-aligned planes）。将 $\mathbf{e}_x =$

$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T, \mathbf{e}_y = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T, \mathbf{e}_z = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$ 叫做主轴（main axes）或者主方向（main direction）；或者分别叫做 x 轴， y 轴和 z 轴。这组向量通常也会被称为标准基（standard basis）。除了特殊说明之外，我们将会使用标准正交基（即由相互垂直的单位向量所组成的基底）。

我们将同时包含 a, b，以及两者之间所有数字的范围区间记为 $[a, b]$。如果我们只想要 a, b 之间的数字，而不想要 a, b 本身的话，那么我们可以将其记为 (a, b)。我们也可以将开闭区间进行组合使用，例如： $[a, b)$ 代表包括 a 在内，但是不包括 b 在内的， a, b 之间的所有数字。

序号	函数	描述
1	$\text{atan2}(y, x)$	二元反正切函数
2	$\log(n)$	n 的自然对数

表 1.3：一些特殊的数学函数的表示方法

$\text{atan2}(y, x)$ 是一个 C 语言中的数学函数，它在本文中经常使用，因此值得我们去关注一下。它是数学函数 $\arctan(x)$ 的一个拓展，它俩的主要区别在于 $-\frac{\pi}{2} < \arctan(x) < \frac{\pi}{2}$，而 $-\pi \leq \text{atan2}(y, x) \leq \pi$；并且后者包含一个额外的参数输入。这个函数的常见应用是用来计算 $\arctan(y/x)$，当 $x = 0$ 时，分母就为 0 了。而拥有两个参数的 $\text{atan2}(y, x)$ 则可以避免这一点。

在本书中， $\log(n)$ 始终代表了自然对数，即 $\log_e(n)$，而不是以 10 为底的对数 $\log_{10}(n)$。

颜色使用一个三维向量来进行表示，例如 $(red, green, blue)$，其中每个分量的范围都是 $[0, 1]$。

1.2.2 几何定义

几乎所有图形硬件使用的渲染图元（primitive，也叫做 drawing primitives）都是点、线和三角形。

> 我们所知道的唯二例外就是 Pixel-Planes[502]，它可以绘制球体；以及 NVIDIA NV1 芯片，它可以绘制椭球体。

在本书中，我们会将一个几何实体（geometric entities）的集合称作为模型（model）或者物体（object）。场景（scene）是指环境中所有待渲染模型的集

合，同时场景中还包含了材质信息，灯光信息，以及观察信息等。

这里的物体可以是一辆车，一栋建筑甚至是一条直线。在实际中，一个物体中包含了一系列的渲染图元，但是也有例外，物体也可以是其他更加高级的几何表现形式，例如 Bezier 曲线（Bezier curves）、Bezier 曲面或者是细分曲面（subdivision surface）。同时，一个物体也可以同时包含其他的物体，例如一辆车包含了四个车门以及四个轮子等。

1.2.3 着色

按照约定俗成的计算机图形学惯例，本书中的“着色（shading）”和“着色器（shader）”以及相关的派生词，常常被用来指向两个相关但是完全不同的概念：一个是计算机生成的视觉外观，例如：“着色模型（shading model）”，“着色方程（shading equation）”，“卡通渲染（toon shading）”等；另一个是渲染系统中的可编程组件，例如：“顶点着色器（vertex shader）”，“着色器语言（shading language）”等。在这两种不同的情况下，你可以通过上下文来推断出它具体指向的含义。

深入阅读和资源

我们能够给你提供的、最重要的资源，就是本书的配套网站：realtimerendering.com，其中包含了最新信息的链接以及每章相应的网站。实时渲染的研究领域也是实时变化的，在本书中，我们试图关注那些最基本的概念，以及那些不太可能过失的技术。在这个网站上，我们可以展示与当今软件开发者有关的信息，并且我们有能力将其进行不断更新。

Chapter 2 The Graphics Rendering Pipeline 图形渲染管线

> Anonymous——“A chain is no stronger than its weakest link.”

> 佚名——“链条的坚固程度取决于它最薄弱的环节。”（一着不慎，满盘皆输。）

本章节介绍了实时图形学中的核心组件，它被称为“图形渲染管线（graphics rendering pipeline）”，也被简称为“管线”。渲染管线的核心功能就是利用给定的虚拟相机、三维物体、光源等信息，来生成或者渲染（render）一张二维图像。因此，渲染管线是实时渲染中的底层工具，其功能如图 2.1 所示。最终图像中物体的位置和形状，由其几何结构，环境特征以及相机位置所决定。而物体的外观则会受到材质属性、光源、纹理（应用在物体表面上的图像）以及着色方程的影响。

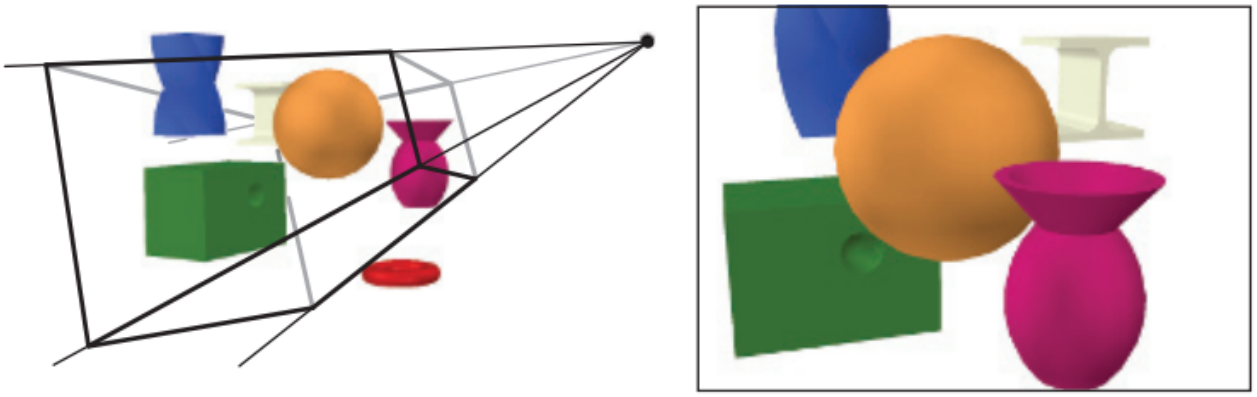


图 2.1：在左图中，虚拟相机位于“金字塔”的顶端（四条直线汇聚的地方），只有位于可视空间内部的物体才会被渲染。对于一个透视渲染的图像而言（如这里的例子），可视空间是一个截锥体（frustum，复数形式为 frusta），通常被叫做视锥体，即一个带有矩形底座的、被截断的金字塔形状。右图显示了虚拟相机所看到的画面，请注意左图中的红色甜甜圈并没有出现在右图中，这是因为它位于视锥体之外；同时左图中扭曲的蓝色棱柱体被视锥体的上平面裁剪了。

我们将会介绍渲染管线中的各个阶段，在本章节中，我们将会重点关注各个阶段的功能而不是实现方式，有关如何应用这些阶段的内容，将在后续章节中进行详细介绍。

2.1 渲染管线的架构

在现实世界中，流水线（pipeline）的概念有很多种表现形式，从工厂的装配线到快餐厨房等，它也同样适用于图形渲染领域。一个流水线中会包含若干个阶段，每个阶段负责完成总任务中的一部分任务[715]。

流水线阶段可以并行执行，其中每个阶段都依赖于前一个阶段的结果。在理想状态下，一个非流水线化的系统可以被划分为 n 个流水线阶段，从而提升 n 倍的速度，这种性能上的提升是使用流水线的主要原因。举个例子，一组员工就可以快速制作一大批的三明治：一个人负责准备面包，一个人负责添加肉片，另一个人负责添加其他配料。每个人都会将结果传递给流水线上的下一个人，同时立即开始下一个三明治的制作。如果每个人都需要 20 秒的时间来完成各自的任务，那么每 20 秒就可以生产一个三明治，一分钟就可以生产 3 个。虽然流水线阶段可以并行进行，但是整个流水线的效率会被执行速度最慢的那个阶段所影响。比如说：如果给三明治加肉片的阶段变得比之前更复杂了，现在需要 30 秒时间才能完成，那么现在流水线的最快速度就是一分钟生产两个三明治了。对于这个三明治流水线而言，加肉阶段就是整个流水线的瓶颈，因为它决定了整个生产过程的最终速度。在等待加肉阶段完成之前，加料阶段是“饥饿（starved）”的（用户也是）。

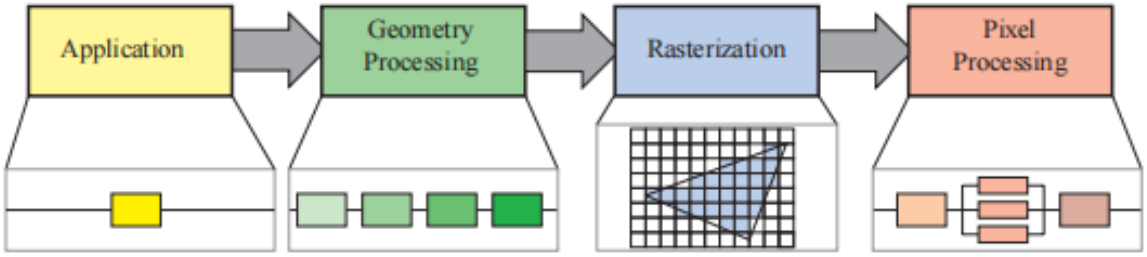


图 2.2：渲染管线的基本结构，包含以下四个阶段：应用阶段、几何处理阶段、光栅化阶段和像素处理阶段。每个阶段本身也可以是一个流水线，例如几何处理阶段下方的图示（包含 4 个子阶段）；有些阶段也可以（部分）并行化执行，例如像素处理阶段下方的图示（三个子阶段并行执行）。在上图中，应用阶段是一个单独的处理过程，但是这个阶段同样也可以流水线化或者并行化。另外请注意，光栅化会查找一个图元内部的像素，例如一个三角形内部的像素。

在实时渲染领域中也可以找到这样的流水线结构，一种粗略的划分方法是将渲染管线分为四个阶段——应用阶段（application）、几何处理阶段（geometry processing）、光栅化阶段（rasterization）和像素处理阶段（pixel processing），如图 2.2 所示。这个结构（渲染管线的引擎）是实时计算机图形程序的核心，也是后续章节的基础概念。每个阶段本身通常也是一个流水线，这意味着每个阶段也是由几个子阶段构成的。在这里我们将功能性阶段（functional stage）

（如上图所示）和其实现结构的概念（the structure of implementation）区分开来。一个功能性阶段有一个特定的任务需要执行，但是并没有指定它在管线中的实现方式。一个给定的实现方式也可能会将两个功能性阶段合并成一个功能单元，或者使用可编程核心来执行；同时它也可以将一个很耗时的功能性阶段，划分为几个硬件单元来完成。

渲染速度可以用每秒帧数（FPS）来进行表示，即每秒显示的帧数；也可以用赫兹（Hz）来进行表示，这个单位代表了 $1/seconds$ ，即更新的频率。通常我们也会直接使用渲染一张图像所花费的时间（毫秒）来表示渲染速度，生成每帧图像所花费的时间往往并不相同，这取决于每帧中所执行计算的复杂度大小。FPS 可以用来表示某一帧的速率，也可以表示一段时间内的平均性能。Hz 一般被设定成一个固定的值，多用于硬件设备中，例如显示器等。（FPS：帧数，Hz：硬件刷新率）。

顾名思义，应用阶段（application）是由应用程序进行驱动的，它在软件中进行实现，运行在通用 CPU 上。这些 CPU 一般都具有多个核心，可以并行处理多个线程（thread）的任务，这使得 CPU 可以高效执行由应用阶段所负责的各种任务，一般 CPU 会负责碰撞检测，全局加速算法，动画，物理模拟等任务，具体会执行哪些任务取决于应用程序的类型。下一个主要阶段是几何处理阶段（geometry processing），它负责处理变换（transform），投影（projection）以及其他所有和几何处理相关的任务。这个阶段需要计算哪些物体会被绘制，应该如何进行绘制，以及应当在哪里绘制等问题。几何阶段通常运行在硬件处理单元（GPU）上，它包含一系列的可编程单元和固定操作硬件。光栅化阶段（rasterization）通常会将构成一个三角形的三个顶点作为输入，找到所有位于三角形内部的像素，并将其转发到下一个阶段中。最后一个阶段是像素处理阶段（pixel processing），对于每个像素而言，都会执行一个程序来决定它的颜色；并执行深度测试，来判断这个像素是否可见；这里还可以执行一些逐像素的操作，例如将新计算的颜色和之前的颜色进行混合。光栅化阶段和像素处理阶段同样完全运行在 GPU 上。所有这些阶段及其内部的子流水线阶段，将会在接下来的四个小节中进行讨论。有关 GPU 如何处理这些阶段的细节，详见第 3 章。

2.2 应用阶段

由于应用阶段通常都运行在 CPU 上， 因此开发者可以完全控制在应用阶段发生的事情，开发者可以决定应用程序的具体实现方式，也可以在之后对其进行修改优化，从而提高程序的性能表现。对应用阶段的修改也会影响后续阶段的性能表现。例如：一个应用阶段中的算法或者设置，可以减少后续需要进行渲染的三角形数量。

尽管如此，有一些应用阶段中的任务也可以让 GPU 来进行执行，即通过使用一个叫做计算着色器（compute shader）的独立模式，该模式会将 GPU 视为一个高度并行的通用处理器，而忽略其专门用于图形渲染的特殊功能。

在应用阶段的最后，需要进行渲染的几何物体会被输入到几何处理阶段中，这些几何物体被称作为渲染图元（rendering primitive），即点、线和三角形。这些图元最终可能会出现在屏幕上（或者是任何正在使用的显示设备上），这也是应用阶段中最重要的任务。

由于这一阶段基于软件实现（software-based），其中一个结果就是，它并没有像之后的几何处理、光栅化和像素处理等阶段一样，被进一步划分出子阶段。

> 由于 CPU 自身就是一个规模很小的流水线，因此我们也可以说，应用阶段也被进一步划分为了几个子阶段，但是这与本章节的主题关系不大。

但是为了提升效率，应用阶段通常也会利用多个处理器核心来并行化执行，在 CPU 设计中，这被称为多核（multi-core）构造，因为它可以在同一阶段执行多个进程。在章节 18.5 中我们介绍了一些多核心调度的方法。

碰撞检测（collision detection）通常会在这个阶段中实现。当检测到两个物体之间的碰撞之后，会产生相应的响应，并返回给碰撞物体，同时也返回给力反馈设备（如果有的话）。应用阶段同样也是处理其他来源输入的地方，例如键盘、鼠标或者头戴式显示器等，会根据不同的输入，从而采取不同的操作。此外，一些加速算法例如特殊的剔除算法（第 19 章）等，以及渲染管线剩余部分无法处理的一切问题，都会在应用阶段中完成。

2.3 几何处理阶段

运行在 GPU 上的几何处理阶段会负责大部分的逐三角形（per-triangle）和逐顶点（per-vertex）操作。将几何处理阶段再细分下去，可以划分为以下几个功能性阶段：顶点着色（vertex shading）、投影（projection）、裁剪（clipping）和屏幕映射（screen mapping），如图 2.3 所示.

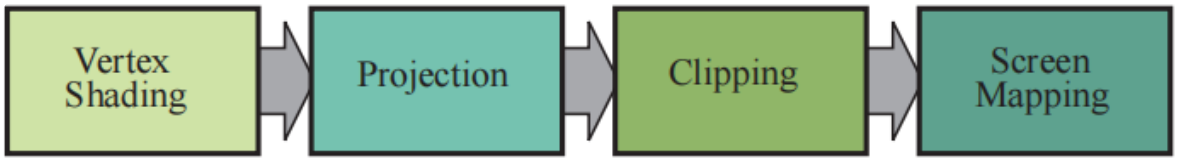


图 2.3：将几何处理阶段进一步划分成一个包含若干功能性阶段的流水线。

2.3.1 顶点着色

顶点着色（vertex shading）的任务主要有两个，一个是计算顶点的位置，另一个是计算那些开发人员想要作为顶点数据进行输出的任何参数，例如法线（normal）和纹理坐标（texture coordinate）等。在早些时候，物体的光照是逐顶点计算的，通过将光源应用于每个顶点的位置和法线，从而计算并存储最终的顶点颜色；然后再通过对顶点颜色进行插值，来获取三角形内部像素的颜色，因此这个可编程的顶点处理单元被命名为顶点着色器（vertex shader）[1049]。随着现代 GPU 的出现，以及几乎全部的着色计算都在逐像素的阶段进行，因此顶点着色阶段变得越来越通用，甚至可能并不会在该阶段中进行任何的着色计算，当然这也取决于开发人员的意图，我们仍然可以在顶点着色器中进行着色计算。顶点着色器如今是一个更加通用的单元，它负责计算并设置与每个顶点都相关的数据。例如在章节 4.4 和章节 4.5 中，顶点着色器可以用来计算物体的动画。

首先我们描述一下顶点位置是如何被计算出来的，它需要一组顶点坐标来作为输入。在物体最终进入屏幕的过程中，它需要在不同的空间（space）或者坐标系（coordinate system）下，进行若干次变换。最开始时，模型位于自身的模型空间（model space）中，也可以简单地认为它没有进行任何变换。每个模型都可以与一个模型变换（model transform）相关联，以便调整自身的位置和朝向。我们可以将若干个模型变换和同一个模型相关联，这样我们就能够在不复制这个模型的前提下，在一个场景中放置同一个模型的多个副本（也叫做实例，instance），每个实例都拥有各自不同的位置和朝向（即模型变换）。

模型变换会对模型的顶点和法线进行变换，模型本身所处的坐标系叫做模型坐标系（model coordinates），它的坐标也被称为模型坐标。当对这些坐标进行模型变换之后，这个模型便位于世界坐标系（world coordinate）或者叫做世界空间（world space）中。世界空间是唯一的，当各个模型经过各自的模型变换之后，所有的模型便都位于一个相同的空间中。

如前文所述，只有被相机（或者观察者）看到的模型才会被渲染，相机在世界空间中有一个位置参数和方向参数，用于放置相机和调整相机的朝向。为了便于后续的投影操作和裁剪操作，相机和世界空间中的所有模型，都会应用观察变换（view transform），观察变换的主要目的是将相机放置在原点上，并调整相机的朝向，使其看向 z 轴负半轴的方向，同时 y 轴指向上方，x 轴指向右方。本文中我们将会使用指向负 z 轴的约定，有一些书籍则更喜欢让相机看向正 z 轴的方向，二者之间的区别主要是语义上的，因为二者之间的转换十分简单。在应用观察变换之后，模型的具体位置和具体方向取决于底层图形 API 的实现方式。这样形成的空间被称作相机空间

（camera space），或者是更加常见的观察空间（view space）和眼睛空间（eye space）。图 2.4 是观察变换的一个例子，它展示了观察变换是如何对相机和模型产生影响的。模型变换和观察变换都是使用 4×4 矩阵实现的，我们将在第 4 章讨论这个话题。这里我们需要认识到，顶点的坐标和法线，都可以按照程序员所喜欢的任何方式来进行计算。

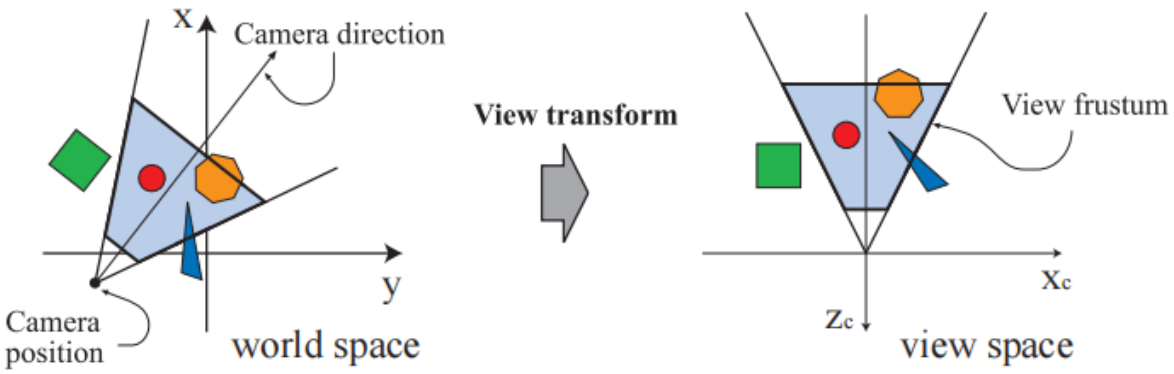


图 2.4：左图是一个俯视图，虚拟相机按照用户的想法进行放置，在这个空间中，正 z 轴指向上方。观察变换重新定位了这个空间，使得相机位置位于原点，看向负 z 轴的方向，同时正 y 轴指向上方，正 x 轴指向右方。这样做可以使得投影操作和裁剪操作变得更加简单。图中的蓝色区域是可视空间，这里假设相机进行的是透视观察，因为此时视野的形状是一个视锥体。类似的变换操作可以应用于任何类型的投影。

接下来，我们将描述顶点着色的第二类输出。为了创建一个真实的场景，仅仅是渲染物体的位置和形状是不够的，我们还需要对物体的外观信息进行建模，包括物体的材质（material）信息以及光源照射在物体表面上的效果。从最简单的颜色描述到基于物理的详细描述，材质和光源可以通过很多方式进行建模。

确定光照作用于材质上所产生的效果，这个操作被称为着色，它涉及到在模型的不同位置上计算着色方程。通常来说，其中一些计算是在模型顶点的几何处理阶段中执行的，其他计算可能会在逐像素处理中完成。顶点上可以存储各种各样的数据，例如顶点位置、法线、颜色或者着色方程所需要的其他数值信息。顶点着色的结果（可能是颜色、向量、纹理坐标或者其他类型的着色数据）会被发送到光栅化阶段中进行插值，并在像素处理阶段中用于计算表面的着色。

顶点着色在 GPU 中会以顶点着色器的形式体现，我们将在本书中进行更加深入的讨论，尤其是第 3 章和第 5 章。

作为顶点着色的一部分，渲染系统还会进行投影操作和裁剪操作，这两个操作会将整个可视空间变换为一个标准立方体（standard cube），其端点位于 $(-1,-1,-1)$ 和 $(1,1,1)$ 处，立方体的边长为 2。可以使用不同的范围来定义相同的空间，例如

$0 \leq z \leq 1$，这个标准立方体被称为规范可视空间（canonical view volume）。首先会进行投影操作，这是在 GPU 上的顶点着色器中完成；有两种常见的投影方法，一种是正交投影（orthographic），也可以叫做平行投影（parallel）；另一种是透视投影（perspective），如图 2.5 所示。事实上，正交投影是平行投影中的一种，其他类型的平行投影也有一些应用，尤其是在建筑领域，例如：斜平面投影（oblique）和轴测投影（axonometric），有一款老式街机游戏《Zaxxon》就使用了轴测投影方式。

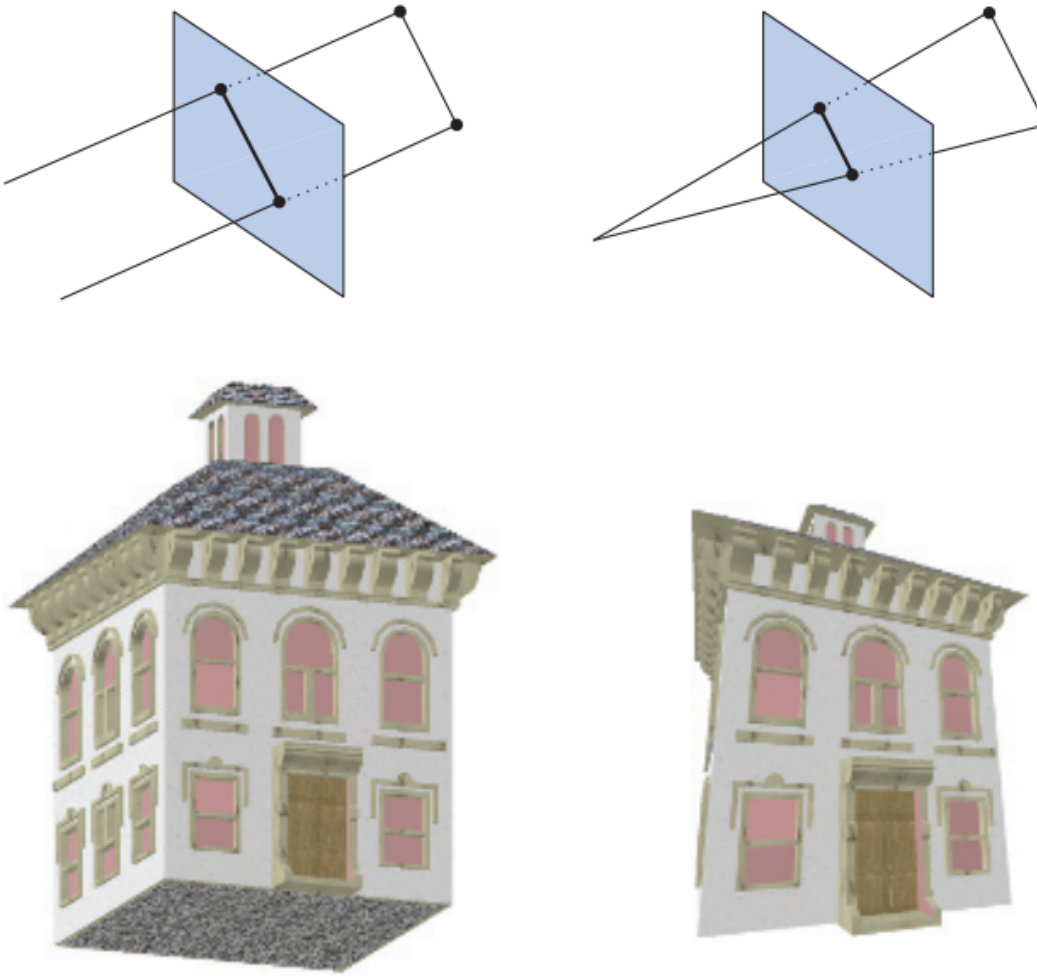


图 2.5：左侧是正交投影，或者叫做平行投影；右侧是透视投影。

请注意，投影操作是通过投影矩阵（章节 4.7）完成的，因此有时候它会和其余几何变换连接起来。

正交视图的可视空间通常是一个长方体，正交投影会将这个可视空间变换为一个标准立方体。正交投影最主要的特征就是，投影变换之前的平行线，在正交投影之后仍然是平行的。这样的投影变换由一个位移变换和一个缩放变换组成。

透视投影要更加复杂一点。在透视投影中，距离相机越远的物体，在投影变换之后就越小；此外，平行线在透视投影之后也会在视界处汇聚，也就是说，透视投影模拟了

我们感知物体大小的方式。从几何学上看，透视投影的可视空间（也叫做视锥体）是一个具有矩形底面的截断金字塔，这个视锥体也会被投影变换为一个标准立方体。正交投影和透视投影都可以使用一个 4×4 矩阵来进行描述（第 4 章），在投影变换之后，模型所处的坐标系被称为裁剪坐标系（clip coordinates），在坐标除以 w 分量之前，事实上它们都是齐次坐标（homogeneous coordinate），我们将会在第 4 章中进行详细讨论。GPU 的顶点着色器必须始终输出这种类型的坐标，以便于下一个功能性阶段（裁剪）可以正确执行。

尽管这些投影变换矩阵会将模型从一个空间变换到另一个空间，但是它们仍然被叫做投影，这是因为在显示之后，坐标的 z 分量并不会被存储在生成的图像中，而是存储在一个叫做 z-buffer 的地方，详见章节 2.5。通过这种方式，模型便从三维空间投影到了二维空间中。

2.3.2 可选的顶点处理

每个渲染管线中，都会有刚才所描述的顶点处理阶段，当完成顶点处理之后，还有几个可以在 GPU 上执行的可选操作，它们的执行顺序如下：曲面细分（tessellation）、几何着色（geometry shading）和流式输出（stream out）。是否使用这些可选操作，一方面取决于硬件的功能（并不是所有 GPU 都支持这些功能），另一方面取决于程序员的意愿。这些功能相互独立，而且一般并不是很常用，详细内容见第 3 章。

第一个可选阶段是曲面细分（tessellation），想象现在有一个使用三角形进行表示的弹性小球，我们可能会遇到质量和性能的取舍问题。在 5 米外观察这个小球可能看起来会很不错，但是如果离近了看，我们会发现部分三角形，尤其是小球轮廓边缘处的三角形会非常明显。如果我们给这个小球添加更多的三角形来提高表现质量，那么当这个小球距离相机很远，仅仅占据屏幕上几个像素的时候，我们会浪费大量的计算时间和内存。这个时候，使用曲面细分可以为一个曲面生成数量合适的三角形，同时兼顾质量和效率。

在前文中我们已经讨论了一些有关三角形的话题，但是到目前为止，我们在管线中只对顶点进行了处理。这些顶点可以用来表示点、线、三角形或者其他物体，顶点也可以用来描述一个曲面（例如一个球体），这样的曲面可以通过几个部分组合而成，每个部分也都由一组顶点构成。曲面细分阶段本身也包含了一系列子阶段——壳着色器（hull shader）、曲面细分器（tessellator）和域着色器（domain shader），它们可以将当前的顶点集合（通常）转换为更大的顶点集合，从而创建出更多的三角形。

场景中的相机位置可以用来决定需要生成多少个三角形：当距离相机很近时，则生成较多数量的三角形；当距离相机很远时，则生成较少数量的三角形。

下一个可选阶段是几何着色器（geometry shader），这个着色器出现的比曲面细分着色器更早，因此在 GPU 上也更加常见。它和曲面细分着色器的相似点在于，它也将各种类型的图元作为输入，然后生成新的顶点。这是一个较为简单的阶段，因为它能够创建的范围是有限的，能够输出的图元则更加有限。几何着色器有好几种用途，其中最流行的一种就是用来生成粒子。想象我们正在模拟一个烟花爆炸的过程，每颗火花都可以表示为一个点，即一个简单的顶点。几何着色器可以将每个顶点都转换成一个正方形（由两个三角形组成），这个正方形会始终面朝观察者，并且会占据若干个像素，这为我们提供了一个更加令人信服的图元来进行后续的着色。

最后一个可选阶段叫做流式输出（stream out）。这个阶段可以让我们把 GPU 作为一个几何引擎，我们可以选择将这些处理好的数据输入到一个缓冲区中，而不是将其直接输入到渲染管线的后续部分并直接输出到屏幕上，这些缓冲区中的数据可以被 CPU 读回使用，也可以被 GPU 本身的后续步骤使用。这个阶段通常会用于粒子模拟，例如我们刚才所举的烟花案例。

以上三个阶段会按照曲面细分、几何着色和流式输出的顺序进行执行，每一个阶段都是可选的。无论我们选择使用其中的哪些阶段，最终我们都会拥有一组使用齐次坐标进行表示的顶点，并输入到管线的下一阶段中，并检查这些顶点是否会被相机看见。

2.3.3 裁剪

只有完全位于可视空间内部，或者部分位于可视空间内部的图元，才需要被发送给光栅化阶段（以及后续的像素处理阶段），然后再将其绘制到屏幕上。完全位于可视空间内部的图元，将会按照原样传递给下一阶段；完全位于可视空间之外的图元，将不会传递给下一阶段，因为它们是不可见的，也不会被渲染；而对于那些一部分位于可视空间内部，一部分位于可视空间外部的图元，则需要进行额外的裁剪操作。例如：一个顶点在外，一个顶点在内的线段会被可视空间裁剪（clip），裁剪之后会生成一个新的顶点，用来替代可视空间之外的那个顶点，这个新顶点位于线段和可视空间的交点处。我们使用投影矩阵将可视空间变换为一个标准立方体，这意味着所有的图元都需要被这个标准立方体所裁剪。使用观察变换和投影变换保证了裁剪操作的一致性，即图元始终只需要针对这个标准立方形进行裁剪即可。

裁剪过程如图 2.6 所示，除了可视空间的六个裁剪平面之外，用户还可以定义额外的裁剪平面来对物体进行裁剪，这种操作被称为切片（sectioning），图 19.1 展示了这一过程。

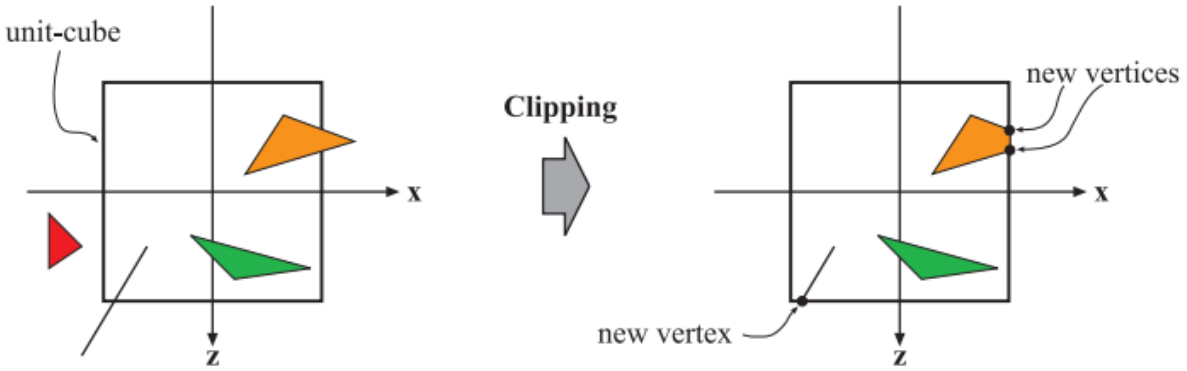


图 2.6：在投影变换之后，只有位于标准立方体内部的图元（对应视锥体内部的模型），才需要进行后续处理。因此完全位于标准立方体外部的图元将会被直接丢弃，完全位于标准立方体内部的图元将会被保留。而与标准立方体相交的图元，将会被标准立方体裁剪，即生成一些新的顶点，而位于立方体之外的旧顶点将会被直接丢弃。

这里我们会使用投影变换生成的四维齐次坐标，来完成这个剪切操作，齐次坐标在透视空间中的三角形上进行的插值，通常并不是线性的，同时我们需要使用齐次坐标的第四个值，以便在透视投影之后进行正确的插值和裁剪。最后会进行透视除法（perspective division），将得到的三角形位置转换到三维标准化设备坐标系（normalized device coordinates，NDC）中。前文中我们提到，这个标准立方体的范围是 $(-1,-1,-1)$ 到 $(1,1,1)$，几何处理阶段的最后一步就是将这个空间转换为窗口坐标系。

2.3.4 屏幕映射

只有位于可视空间内部的图元才会被传递到屏幕映射（screen mapping）阶段。当这些图元进入这一阶段时，其坐标还是三维的，其中 x 坐标和 y 坐标会被转换为屏幕坐标（screen coordinate），屏幕坐标和 z 坐标在一起，被称作窗口坐标（window coordinate）。假设这个场景会被渲染到一个窗口中，窗口左下角的坐标为 (x_1,y_1)，右上角的坐标为 (x_2,y_2)，其中 $x_1<x_2, y_1<y_2$。屏幕映射包含了一个缩放操作，映射后的新 x,y 坐标会被称为屏幕坐标。z 坐标同样也会被映射到 $[z_1,z_2]$ 的范围中，默认是 $z_1=0, z_2=1$（OpenGL 中的范围是 $[-1,+1]$，DirectX 中的范围是 $[0,1]$），但是这些范围也可以使用相应的 API 来进行修改。窗口坐标和这个被重映射的 z 值一起，都会被传递到光栅化阶段中。屏幕映射的过程如图 2.7 所示。

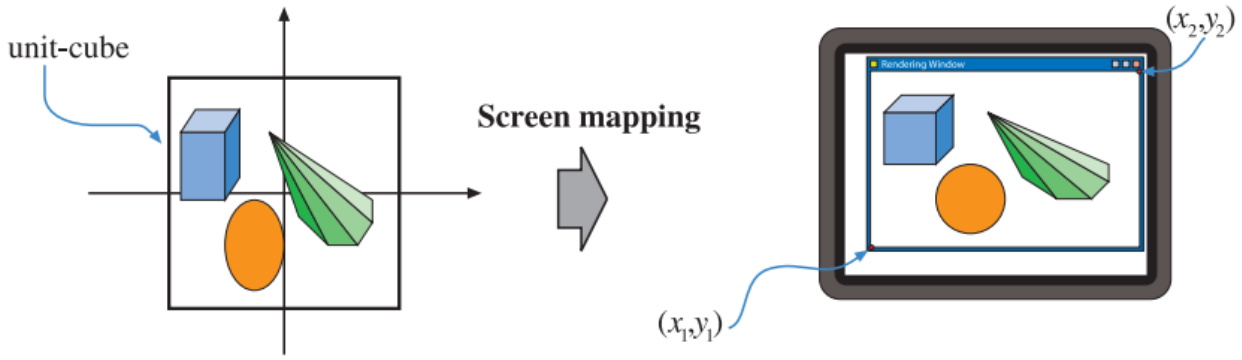


图 2.7：投影变换之后的图元都位于标准立方体内部，屏幕映射负责找到图元在屏幕上的坐标位置。

接下来我们将描述整型数值和浮点型数值与像素（以及纹理坐标）之间的对应关系，给定一个使用笛卡尔坐标进行描述的水平像素数组，那么最左侧像素的中心点坐标为 0.5，该像素左边界的坐标为 0.0。OpenGL 以及 DirectX 10 之后的 API 均采用了这一规定。因此，$[0, 9]$ 范围的像素占据了 $[0.0, 10.0)$ 的范围，也就是说，转换规则为：

$$d = \mathsf{floor}(\mathsf{c}), \tag{2.1}$$

$$c = d + 0.5, \tag{2.2}$$

其中 d 代表了像素的位置索引（整型），c 代表了像素内的连续值（浮点类型）[692]。

虽然所有 API 都定义像素值从左向右会不断增大，但是在某些情况下，OpenGL 和 DirectX 在竖直方向上的起始位置是不一样的。

> “Direct3D”是 DirectX 的三维图形 API，DirectX 中还包含了许多其他 API，例如输入控制和音频控制等。在讨论这个 API 的时候，我们一般不去区分“Direct3D”和 DirectX 之间的区别，这里我们统一使用“DirectX”。

OpenGL 倾向于全部使用笛卡尔坐标系，始终将最左下角的像素视为最小像素（即起始位置的像素）；而在 DirectX 中，根据上下文的不同，有时候会将左上角的位置作为最小像素。在这个问题上并没有一个标准答案，每个规定都有着自己的逻辑。例如：在 OpenGL 中，$(0, 0)$ 点位于屏幕的左下角；而在 DirectX 中，这个点位于屏幕的左上角。当我们想要迁移 API 的时候，这个差异是十分重要的。

2.4 光栅化阶段

现在我们已经有了被正确变换和正确投影的顶点数据，以及它们相应的着色数据（在几何处理阶段中获取的），下一阶段的目标是找到位于待渲染图元（例如三角形）中的所有像素值（pixel，picture element 的缩写）。我们将这个过程称作为光栅化（rasterization），我们可以将其划分成两个子阶段：三角形设置（triangle set up，也叫做图元装配，primitive assembly）和三角形遍历（triangle traversal），其过程如图 2.8 所示。这里需要注意的是，这两个子阶段也同样适用于点和线，只是因为三角形是更加常见的图元，因此这两个子阶段的名字中才带有了三角形。光栅化也被称为扫描变换（scan conversion），这是一个将屏幕空间中二维顶点，转换到屏幕上像素的过程，其中每个顶点都对应一个 z 值（深度缓冲）和各种各样的着色信息。光栅化也可以被认为是一个几何处理阶段和像素处理阶段之间的同步点，因为光栅化阶段的三角形，是由几何处理阶段输出的顶点组成的，并且最终会输出到像素处理阶段中。

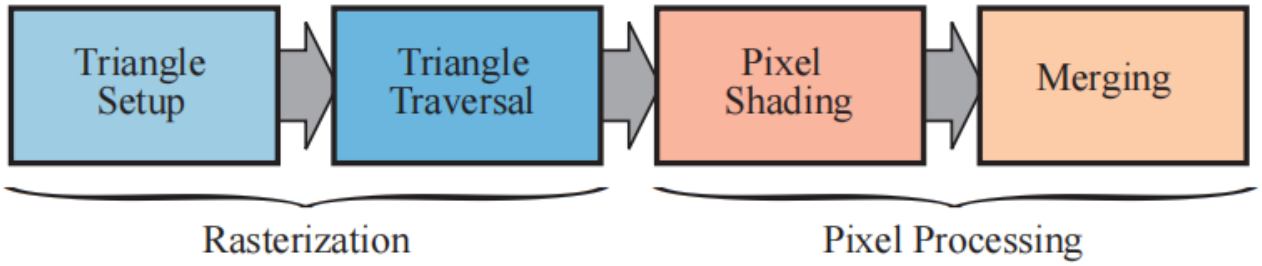


图 2.8：左侧：光栅化阶段被划分为了两个功能性阶段，分别被称为三角形设置和三角形遍历。右侧：像素处理阶段也被划分为了两个功能性阶段，分别被称为像素着色和合并。

判定某个三角形和屏幕上的哪些像素重合，取决于我们如何实现 GPU 管线。例如：你可以使用点采样（point sample）来判定某个点是否位于三角形内部。最简单的方式就是直接将每个像素的中心点来作为该像素的样本，如果该像素的中心点位于三角形内部的话，那么我们就认为该像素也位于三角形的内部。我们还可以通过超采样（supersampling）或者多重采样抗锯齿技术（multisampling antialiasing），来对每个像素进行多次采样，详见章节 5.4.2。另一种方法是使用保守光栅化（conservative rasterization），即当某个像素只要有一部分与三角形重叠时，我们就认为该像素位于三角形内部，详见章节 23.1.2。

2.4.1 三角形设置

三角形的微分（differential）、边界方程（edge equation）和其他数据，都会在这个阶段进行计算，这些数据可以用于三角形遍历（章节 2.4.2），以及对几何处理阶

段产生的各种着色数据进行插值。这个功能一般会使用固定功能的硬件实现。

2.4.2 三角形遍历

在这一阶段，会对每个被三角形覆盖的像素（中心点或者样本点在三角形内部的像素）进行逐个检查，并生成一个对应的片元（fragment）。我们可以在章节 5.4 中找到更加详细的采样方法。找到那些位于三角形内部的点或者样本，这个过程通常被称为三角形遍历，并且会对三角形三个顶点上的属性进行插值，来获得每个三角形片元的属性（第 5 章），这些属性包括片元的深度，以及几何阶段输出的相关着色数据等。McCormack 等人[1162]提供了有关三角形遍历的更多信息。在光栅化阶段也会对三角形进行透视正确的插值[694]（章节 23.1.1）。片元内部的像素或者样本会被输入到像素处理阶段中，下面我们将会对其进行介绍。

2.5 像素处理阶段

经过之前若干阶段的处理，这里我们已经找到了所有位于三角形（或者其他图元）内部的像素。像素处理阶段也可以被划分为像素着色（pixel shading）和合并（merging）两个阶段，如图 2.8 所示。在像素处理阶段，会对图元内部的的像素（或者样本）进行逐像素（或者逐样本）的计算和操作。

2.5.1 像素着色

这里会使用插值过的着色数据作为输入，来进行逐像素的着色计算，其结果是生成一个颜色值或者多个颜色值，这些颜色值会被输入到下一阶段中。三角形设置和三角形遍历使用了专门的硬件单元进行执行，而像素着色阶段则是由可编程的 GPU 核心来执行的。为此，程序员需要为像素着色器（在 OpenGL 中叫做片元着色器）提供一个实现程序，这个程序中包含了任何我们想要的着色计算操作。这里可以使用各种各样的技术，其中最重要的一个技术就是纹理化（texturing），详见第 6 章。简单来说，纹理化就是将一个图像或者多个图像“粘合（gluing）”在物体表面，从而实现各种各样的效果和目的。图 2.9 展示了一个纹理化的简单例子，一般这些纹理都是二维图像，但是有时候也可以是一维图像或者三维图像。简单来说，像素着色阶段最终会输出每个片元的颜色值，这些颜色值会被输入到下一个子阶段中。

图 2.9：左上角是一个没有纹理的龙模型，右边的纹理会被“粘合”在模型的表面，其结果如左下角图像所示。

2.5.2 合并

颜色缓冲（color buffer）是一个矩形阵列，它存储了每个像素中的颜色信息（即颜色的红绿蓝分量）。在之前的像素着色阶段中，我们计算了每个片元的颜色，并将其存储在颜色缓冲中，而合并阶段的任务就是将这些片元的颜色组合起来。这个阶段也被叫做 ROP，意思是“光栅操作管线（raster operations pipeline）”或者“渲染输出单元（render output unit）”，这取决于你问的是谁。与像素着色阶段不同，执行这一阶段的 GPU 子单元，并不是完全可编程的；但它仍然是高度可配置的，可以支持实现各种效果。

合并阶段还负责解决可见性问题，即当整个场景被渲染的时候，颜色缓冲应当只包含那些相机可见的图元颜色。对于大部分或者几乎所有的图形硬件而言，这个操作是通过 z-buffer（深度缓冲）实现的[238]。z-buffer 具有与颜色缓冲相同的尺寸，对于其中的每个像素，它存储了目前距离最近的图元 z 值。这意味着，当一个图元要被渲染到某个像素上时，会计算这个图元的 z 值，并将其与 z-buffer 中的对应像素深度进行比较。如果这个新的 z 值比当前 z-buffer 中的像素深度更小，说明这个新图元距离相机更近，会挡住原来的图元，因此需要使用新图元的 z 值和颜色值来对 z-buffer 和颜色缓冲进行更新；如果新图元的 z 值大于对应像素在 z-buffer 中的 z 值，说明这个新图元距离相机更远，则 z-buffer 和颜色缓冲将会保持不变。z-buffer 算法十分简单，其时间复杂度为 $O(n)$（n 为需要被渲染的图元数量）；且这个算法适用于任何能够计算出 z 值的图元。同时请注意，z-buffer 允许图元以任意顺序进行渲染，这也是它流行的另一个原因。但是 z-buffer 在每个屏幕像素上，只存

储了一个深度值，因此它不适用于透明物体的渲染。透明物体必须要等到所有的不透明物体都渲染完成之后，才能进行渲染，而且需要严格按照从后往前的顺序进行渲染，或者使用一个顺序无关的透明算法（章节 5.5）。透明物体的渲染是 z-buffer 算法的主要弱点之一。

我们刚才提到使用颜色缓冲来存储每个像素的颜色，使用 z-buffer 来存储每个像素的 z 值。但是还有一些其他的通道和缓冲可以用来过滤和捕获片元的信息，例如与颜色缓冲相关联的透明通道（alpha channel），它存储了每个像素的不透明度（opacity，章节 5.5）。在一些较老的 API 中，透明通道也可以被用来进行透明测试（alpha test），来选择性的丢弃一些像素。如今这样的片元丢弃操作可以在像素着色器中完成，而且任何我们想要的计算都可以用来触发这个丢弃操作。这种类型的测试可以用来确保那些完全透明的片元，不会对 z-buffer 产生影响（章节 6.6）。

模板缓冲（stencil buffer）是一个离屏缓冲区（offscreen buffer），它可以用来记录被渲染图元的位置信息，通常它的每个像素包含 8 bit。图元可以通过各种各样函数来被渲染到模板缓冲中，同时模板缓冲可以用来控制渲染到颜色缓冲和 z-buffer 中的内容。举个例子：假设现在有一个实心圆被写入到了模板缓冲中，现在我们通过一个操作，可以只允许后续图元被渲染到这个实心圆所在位置的颜色缓冲中。模板缓冲十分强大，可以用于生成一些特殊效果。所有这些在管线末尾的功能都被叫做光栅操作（raster operation，ROP）或者混合操作（blend operation）。我们也可以将当前颜色缓冲中的颜色，与三角形中正在处理的颜色相混合，从而实现一些透明效果或者颜色样本累积的效果。上文中我们提到，混合操作通常并不是完全可编程的，一般只能通过使用 API 来进行配置。但是某些 API 支持光栅顺序视图（raster order view），也可以被称作像素着色器排序，它支持可编程的混合操作。

系统中的所有缓冲区在一起，被统称为帧缓冲（frame buffer）。

当图元到达并通过光栅化阶段时，这些从相机角度可见的图元将会被显示在屏幕上， 屏幕上所显示的内容就是颜色缓冲中的内容。由于渲染需要花费一定时间，为了避免观察者看到图元渲染并显示在屏幕上的过程，一般都会使用双缓冲机制（double buffering），这意味着场景的渲染都会在屏幕外的后置缓冲区中进行。当场景被渲染到后置缓冲区之后，后置缓冲区会与显示在屏幕上的前置缓冲区（front buffer）交换内容。这个交换的过程通常发生在垂直回扫（vertical retrace）的过程中，因此这样做是可行的。

有关不同缓冲和缓冲方法的更多内容，详见章节 5.4.2，章节 23.6 和章节 23.7。

2.6 回顾整个管线

点，线和三角形是用来构建模型和物体的渲染图元。假设现在有一个交互式的计算机辅助设计（computer aided design，CAD）程序，用户正在查看一个华夫饼机的设计模型。现在我们将跟随这个模型，完整通过整个图形渲染管线，它包含四个主要阶段：应用阶段、几何处理阶段、光栅化阶段和像素处理阶段。场景将以透视视角渲染到一个屏幕窗口中，在这个简单的例子中，华夫饼机的模型由线段（展示各个部分的边界）和三角形（展示模型的表面）组成，华夫饼机还有一个可以打开的盖子，上面有一些三角形具有制造商标志的二维纹理。在这个例子中，除了纹理贴图的使用发生在像素处理阶段之外，其余所有的表面着色计算都在几何处理阶段完成。

应用阶段

CAD 程序允许用户选择并移动部分模型，例如用户可能会选中华夫饼机的盖子，并通过拖动鼠标来打开它。应用程序阶段需要将用户的鼠标移动，转换为相应的旋转矩阵，然后确保这个旋转矩阵在渲染的时候会被正确应用。另一个例子是播放相机动画，让相机沿着预定的路线进行移动，从不同的视角来展示这个华夫饼机。程序需要按照动画的设定，来对相机的相关参数进行更新，例如位置、视角方向等。在渲染每一帧之前，应用程序需要将相机位置，光照和模型的图元信息，都发送给管线的下一阶段——几何处理阶段。

几何处理阶段

对于透视视角而言，我们这里假设应用阶段已经提供了一个投影矩阵（$\mathbf{P}$）；同样的，对于每个物体而言，应用程序都会计算出一个矩阵（$\mathbf{MV}$），这个矩阵描述了观察变换以及物体本身的位置和朝向。在我们的例子中，华夫饼机的底座会对应一个矩阵，而盖子则会对应另外一个矩阵。在几何阶段，物体的顶点和法线会通过这个矩阵被变换到观察空间中，然后使用材质和光照信息，来进行顶点着色以及其他一些计算。然后会使用一个由用户提供的投影矩阵，来将相机的可视空间变换为一个标准立方体，所有位于标准立方体外部的图元都会被直接丢弃，所有与标准立方体相交的图元都会被裁剪，从而获得完全位于标准立方体内部的图元。最后顶点会被映射到屏幕上的窗口中。当所有这些逐三角形和逐顶点的操作完成之后，生成的结果数据会被输入到光栅化阶段。

光栅化阶段

所有在前一阶段中被保留下来的图元，在这个阶段中都会进行光栅化，即找到所有位于图元内部的像素，然后将其发送到管线的像素处理阶段。

像素处理阶段

这一步的目标是计算出每个可见图元所覆盖像素的颜色值。与纹理（图像）相关联的三角形会使用这些纹理进行渲染。图元的可见性通过使用 z-buffer 算法来进行解决，也可以使用可选的图元丢弃操作和模板测试。每个物体都会被轮流处理，最终生成一副图像，然后显示在屏幕上。

总结（Conclusion）

本章节所描述的渲染管线，是几十年来，面向实时渲染程序的 API 以及图形硬件发展而来的结果。很重要的一点是，这并不是唯一可能的渲染管线，离线渲染中也有一套渲染管线，但是经历了与实时渲染完全不同的演化路径。用于电影制作的渲染以往都会采用微多边形（micropolygon）渲染管线[289, 1734]，但是最近几年来，光线追踪和路径追踪已经渐渐占据了上风。这些技术在建筑可视化领域和设计可视化领域中也有很多应用，它们会在章节 11.2.2 中提及。

多年以来，开发人员只能使用图形 API 所定义的固定渲染管线（fixed-function pipeline）来完成这个过程。固定渲染管线之所以名字中带有“固定”，是因为其中的功能是基于硬件进行实现的，而这些硬件无法进行灵活的编程调整。固定渲染管线的最后一个重要机器，是于 2006 年推出的 Ninendo Wii。另一方面，可编程 GPU 可以准确的确定，渲染管线中的每个阶段应用的都是哪些操作。在本书中，我们假设所有的开发都是使用可编程 GPU 来完成的。

补充阅读和资源

Blinn 的《A Trip Down the Graphics Pipeline》[165]是一本从头开始编写软件渲染器的老书，这是一个很好资源，你可以从中了解如何实现一个渲染管线的细节，它介绍了一些诸如裁剪和透视插值的关键算法。《OpenGL：Programming Guide》（简称“Red Book”，即红宝书）[885]是一本高龄的书（但是会经常更新），它提供了有关图形渲染管线的详细描述，以及所使用算法的详细描述。在我们的配套网站 realtimerendering.com 上，提供了各种各样的管线程序、渲染引擎的实现以及其他信息。

Chapter 3 The Graphics Processing Unit 图形处理单元

> Jen-Hsun Huang——“The display is the computer.”
>
> 黄仁勋——“显示器就是电脑”。（Nvidia 创始人兼 CEO；1963—）

在历史上，图形加速首先开始于三角形上的像素扫描线颜色插值，并在屏幕上显示这些颜色值。它包括访问图像数据的能力，这允许将纹理运用在表面上；添加用于插值和深度测试的硬件，这提供了内置的可见性检查。由于这些过程在渲染管线中被大量使用，因此它们都需要专门的硬件单元来进行执行，从而提高性能表现。在多次迭代的过程中，渲染管线增加了很多部分，并且其中的每个部分也增加了更多的功能。这些专用的图形硬件与 CPU 相比，唯一的优势就是速度，在实时渲染中，速度是至关重要的。

在过去的二十年中，图形硬件发生了不可思议的变化。第一个包含硬件顶点处理的消费级芯片（NVIDIA GeForce256）于 1999 年发布。NVIDIA 创造了图形处理单元（graphics processing unit，GPU）这个术语，用来区别 GeForce256 和过去所使用的光栅化芯片，并将这个术语沿用了下来。在接下来的几年时间里，GPU 从一个可配置的复杂固定功能管线，发展为一个高度可编程的“白板”，开发者可以在这个白板上实现自己的图形算法。各种可编程的着色器（shader）是控制 GPU 的主要手段。为了获得更高的效率，渲染管线中的有些部分仍然只是可配置的，而并非是可编程的，但是 GPU 的整体发展趋势是可编程性和灵活性[175]。

GPU 专注于一组高度并行化的任务，从而获得了很高的处理速度，它使用专门的硬件来实现 z-buffer，来能够快速访问纹理图像和其他缓冲区，还可以快速寻找哪些像素被一个三角形所覆盖。我们将在第 23 章，讨论这些硬件单元是如何实现各自的功能的。目前更重要的是，需要尽早了解 GPU 是如何实现可编程着色器的并行化处理的。

在章节 3.3 中，我们解释了着色器是如何进行工作的，而现在我们需要知道的是，着色器核心是一个用于执行某些相对独立任务的小型处理器，例如将一个世界空间中的顶点位置，变换到屏幕空间中；或者是计算一个像素（被三角形覆盖）的颜色。由于

每帧都会有成千上万的三角形被发送到屏幕上，因此每秒可能会有数十亿次的着色器调用（shader invocation），即运行着色器程序的单独实例。

首先我们需要知道一点，由于在存储中访问数据需要花费一定时间，因此延迟是所有处理器都会面临的一个问题。考虑延迟的一个基本方法是，数据距离处理器越远（物理意义上的距离），访问所需要等待的时间就越长，章节 23.3 详细讨论了这个延迟问题。相对于本地寄存器而言，访问内存碎片中的信息则需要花费更多的时间，章节 18.4.1 对内存访问进行了深入讨论。这个问题的关键在于，等待数据检索意味着此时处理器处于停滞状态，这会降低性能表现。

3.1 数据并行结构

不同的处理器架构使用了不同的策略来避免停滞。CPU 经过优化，可以处理大量的数据结构和大型代码段，CPU 一般都具有多个处理器，每个处理器都以串行的方式来执行代码，但是有限的 SIMD 向量处理是一个小例外。为了最小化延迟所带来的影响，CPU 芯片中的大部分面积都是高速的本地缓存，这些缓存中存满了接下来可能会用到的数据。CPU 还会使用一些智能技术来避免停滞，例如分支预测（branch predication）、指令重排序（instruction reordering）、寄存器重命名（register renaming）和缓存预取（prefetching）等[715]。

而 GPU 则采用了不同的策略，GPU 芯片中的很大一片面积都是大量的处理器，也叫做着色器核心（shader core），GPU 芯片中通常会有数千个着色器核心。GPU 是一个流处理器，它会依次处理有序的相似数据。由于数据的相似性（例如一组顶点或者像素），因此 GPU 可以通过大规模并行的方式来处理这些数据。另一个重要因素是，这些着色器调用都是尽可能独立的，即它们不需要来自邻近调用的信息，也不需要共享可写入的内存位置。有时候为了使用一些新功能，这个规定会被打破，但是这种例外会带来潜在的额外延迟，因为一个处理器可能会需要等待另一个处理器执行结束之后，才能开始工作。

GPU 专门对吞吐量（throughput）进行了优化，吞吐量指的是数据能够被处理的最大速度。但是这种快速处理是有代价的，由于用于缓存和控制逻辑的芯片面积较少，因此每个着色器核心的延迟，通常都会比 CPU 处理器所遇到的延迟要大[462]。

假设一个模型网格需要被光栅化，它生成了两千个需要进行处理的片元，那么像素着色器将会被调用两千次。想象现在我们有一个世界上性能最弱的 GPU，它只包含了一个着色处理器（shader processor）。它首先会为两千个片元中的第一个执行着色器程序，这个着色处理器会对存储在寄存器中的值进行一些算术操作。这里的寄存器

是本地的，它的访问速度很快，因此处理器不会发生停滞。然后处理器遇到了一个访问纹理的指令，例如：对于一个给定的表面位置，程序需要知道纹理上对应位置的像素颜色，而这个纹理是一个独立的资源，并不是像素着色程序本地内存中的一部分，因此可能会涉及到访问纹理的操作。读取内存数据需要花费成百上千个时钟周期，而在此期间 GPU 处理器将不会进行任何操作。此时处理器将会停滞，并等待这个纹理颜色值的返回。

为了让这个弱鸡的 GPU 变得更好一点，我们为每个片元的本地寄存器都提供了一小片存储空间。现在这个着色处理器不会在访问纹理的时候发生停滞了，而是会进行切换并执行另一个片元的着色器程序，即两千个片元中的第二个。这个片元切换的过程是很快的，除了需要注意第一个片元当前执行的是哪个指令之外，不会有其他任何影响。现在处理器会执行第二个片元的着色器程序，与第一个片元一样，这里进行了一些相同的算术函数，然后会再次遇到这个纹理访问的问题，然后着色器核心会再次切换到第三个片元的着色器程序。最终，所有 2000 个片元都会以这种形式依次进行处理，然后着色处理器会返回到第一个片元，这时已经获取到了纹理的颜色，因此着色器程序可以继续进行执行。处理器会以这种方式继续执行，直到遇到另一个会导致停滞的指令，或者是程序执行完成。在这种执行方式下，处理单个片元会花费更长的时间，但是整体的片元执行时间将会大大减少。

在这种架构中，当遇到会令着色处理器停滞的指令时，我们通过切换并执行其他片元程序的方式，来让 GPU 时刻保持忙碌，从而避免延迟。更进一步，GPU 可以将指令执行的逻辑与数据分离开来，这种设计叫做单指令、多数据（single instruction, multiple data，SIMD），这种设计会在固定数量的着色器程序上，以一个固定的步长来执行完全相同的指令。与使用一个独立的逻辑单元和调度单元来执行每个程序相比，SIMD 的优势在于，可以使用更少硅芯片（也意味着更小的功耗）来用于处理数据和进行切换。使用现代 GPU 的术语来说，每个片元的像素着色器调用都可以被称为一个线程（thread），但是这里所说的线程不同于 CPU 上的线程，它还包括用于存储着色器输入数据的存储空间，以及用于着色器执行的任何寄存器空间。这些使用相同着色器程序的线程会被打包成组，NVIDIA 将其称作为一个 warp，AMD 将其称作为一个 wavefront。一个 warp/wavefronts 负责调度一定数量的 GPU 处理核心，可能是 8 到 64 个，并且都会使用 SIMD 处理。每个线程都会被映射到一个 SIMD 通道（SIMD lane）。

假设我们现在有 2000 个线程要执行，NVIDIA GPU 中的一个 warp 包含 32 个线程，$2000/32 = 62.5$，这就意味着我需要分配 63 个 wrap 来执行这些线程，其中有一个 warp 只被占用了一半。一个 wrap 的执行和单个 GPU 处理器类似，同一个 warp

内的 32 个处理器，都会以一个固定的步长来执行着色器程序，所有线程都会执行完全相同的指令，即当一个线程遇到存储读取的时候，所有线程都会同时遇到这个存储读取。这个读取信号表明了这个 warp 中的所有线程都会发生停滞，并等待各自的返回结果。此时这个 warp 会切换到另一个包含 32 个线程的 warp，然后由 32 个处理器进行执行。这个交换过程与我们之前单个处理器的切换过程一样快，在交换 warp 的过程中，每个线程中的数据不会被修改，每个线程都有着各自的寄存器，并且每个 warp 都会记录下当前正在执行的指令。交换一个 warp，只是将一组处理器核心指向另一组需要被执行的线程而已，没有其他的额外开销。每个 warp 只会在所有任务都完成之后，才会进行交换，详情如图 3.1 所示。

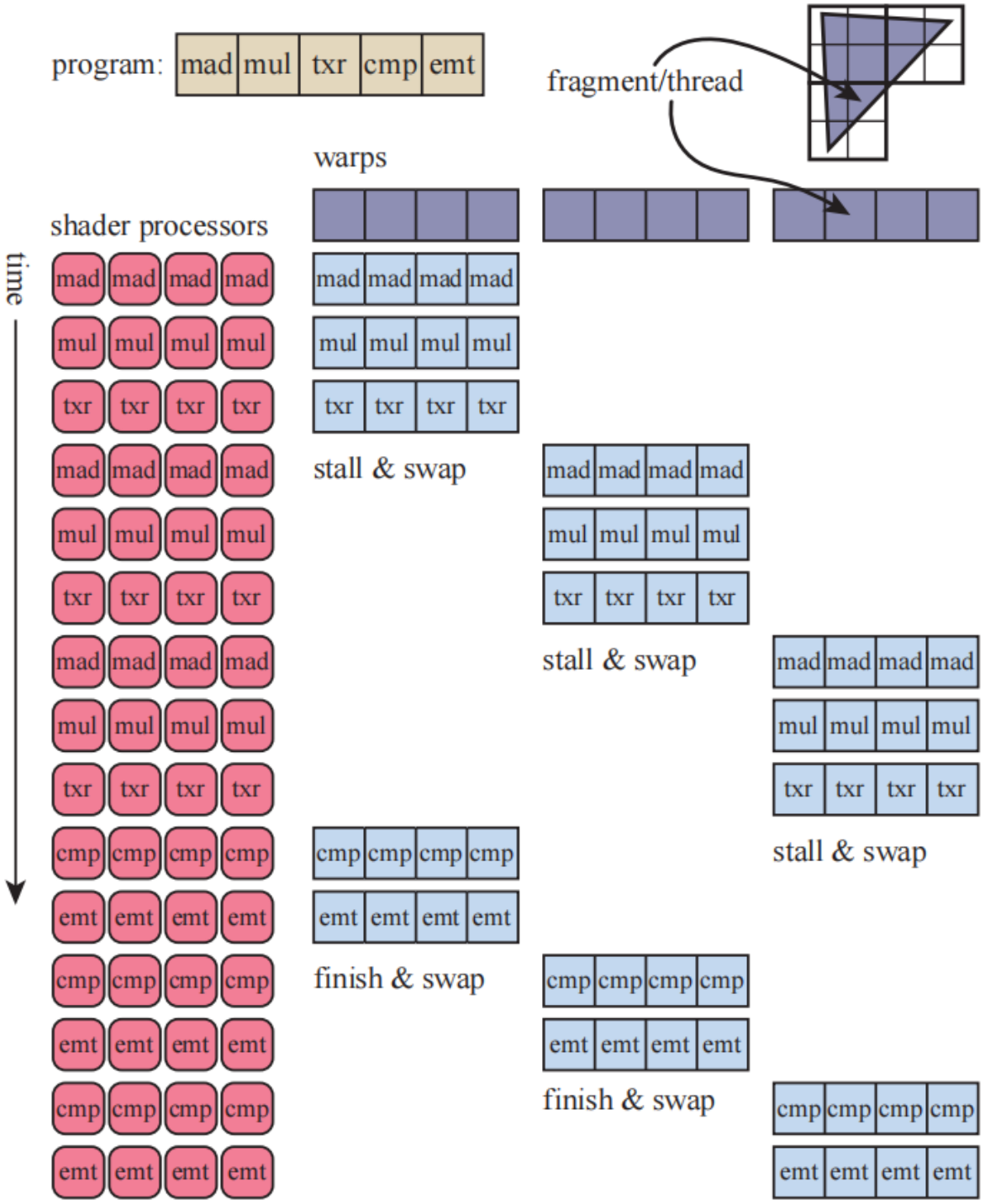


图 3.1：一个简化的着色器执行例子。一个三角形的片元（被称作线程），被打包成很多个 warp。为了简化表示，上图中的每个 warp 都只包含四个线程，而真正的 warp 一般包含 32 个线程。这个等待执行的着色器程序中包含五个指令。这四个处理器为一组，开始执行第一个 warp 中的指令，直到在遇到“txr”指令时，检测到了一个停滞状态，这个指令需要花费一些时间来从存储中获取数据。然后第二个 warp 会被切换到处理器中，同样执行着色器程序中的前三条指令，直到再次检测到一个停滞状态，第三个 warp 也同样如此。然后处理器会再次切换回第一个 warp，继续进行执行，如果这个时候“txr”指令所

需要的数据还没有返回的话，那么处理器才会真正的停滞下来，直到这些数据可用的时候。每个 warp 都会按照顺序轮流执行。

在我们的这个简单例子中，在存储中读取纹理所带来的延迟，可能会导致 warp 进行交换。由于这个切换的成本非常低，所以实际上的 warp 可以通过交换来获得更低的延迟。还有一些其他的技术可以用来优化执行效率[945]，但是 warp 交换是 GPU 上最重要的延迟隐藏技术。这个过程的工作效率还涉及到好几个其他因素，例如：如果线程的数量很少，那么就只能创建很少的 warp，这可能就没法有效隐藏延迟。

影响执行效率的另一个重要特征是着色器程序的结构，其中最重要的一个因素就是每个线程所使用的寄存器数量。我们现在假设 GPU 上可以同时存在两千个线程，每个线程中运行的着色器程序所需要使用的寄存器数量越多，那么 GPU 上能够同时存在的线程数量和 warp 数量也就越少。数量较少的 warp 意味着，可能无法通过 warp 交换来缓解处理器核心的停滞，正在执行的 warp 被称作“in flight”，其数量被称为占用率（occupancy）。更高的占用率意味着存在更多用于处理的 warp，也意味着会有更少的空闲处理器，较低的占用率往往意味着性能表现不佳。存储读取的频率同样也会对延迟产生影响，Lauriten [993]指出了着色器所使用的寄存器数量，以及共享存储是如何影响 GPU 占用率的。Wronski [1911, 1914]讨论了理想中的 GPU 占用率，是如何跟着着色器操作类型而发生改变。

另一个影响整体运行效率的因素是由“if”语句和循环语句导致的动态分支（dynamic branching）。假设现在我们的着色器程序中遇到了一个“if”语句，如果所有线程都进入了相同的分支，那么这个 warp 可以不用管其他的分支，继续执行进入的那个分支即可。但是，如果其中有几个线程，甚至是只有一个线程进入了其他的分支，那么这个 warp 就必须把两个分支都执行一遍，然后再根据每个线程的具体情况，丢弃不需要的结果[530, 945]。这个问题叫做线程发散（thread divergence），它意味着有一些线程需要去执行一个循环操作，或者是进入了所在 warp 中其他线程都没有进入的“if”分支，这会导致其他的线程空转。

所有的 GPU 实现都应用了这些架构思想，虽然这样会导致系统具有严重的限制，但是这也提供了大量的每瓦计算能力。理解这个系统是如何进行操作的，可以帮助开发人员更加高效的利用 GPU 性能。在接下来的小节中，我们会讨论 GPU 是如何实现渲染管线的，可编程着色器是如何进行操作的，以及每个 GPU 阶段的演变与功能。

3.2 GPU 管线概述

我们在第 2 章中描述了概念上的几何处理阶段、光栅化阶段和像素处理阶段，GPU 则在硬件上实现了这些阶段，它们被划分为了几个不同的硬件阶段，每个阶段都有着不同程度的可配置性和可编程性。图 3.2 中展示了这些硬件阶段，并使用不同的颜色来标注它们的可配置性和可编程性。这里需要注意的是，这些物理阶段的划分与第 2 章中展示功能阶段有所不同。

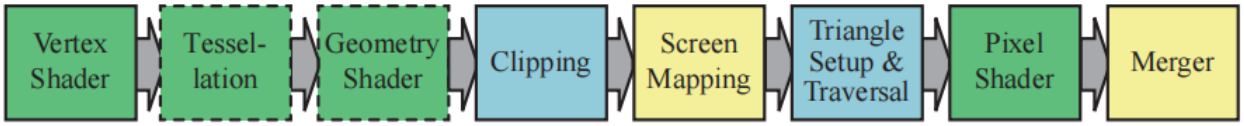


图 3.2：渲染管线的 GPU 实现。根据用户能够对其操作的控制程度，对这些阶段进行了颜色编码。其中绿色代表了这个阶段是完全可编程的，虚线边框代表了这个阶段是可选的。黄色代表了这个阶段是可配置的，但并是非可编程的，例如我们可以为合并阶段设置不同的混合模式，但是无法完全对其进行编程控制。蓝色代表了这些阶段中的功能是完全固定的。

我们这里所描述的是 GPU 的逻辑模型（logical model），它通过图形 API 的形式暴露给程序员。如第 18 章和第 23 章中所讨论的，这个逻辑模型的具体实现，即物理模型（physical model），取决于硬件供厂商。逻辑模型中某个固定功能阶段，可能是通过为相邻的可编程阶段添加一些指令实现的。管线中的某个简单程序，也可能会被划分成子程序，然后由相互独立的子单元进行执行，或者是完全由一个独立的 pass 执行。这个 GPU 逻辑模型可以帮助你理解影响性能的因素，但是不应当将其误认为是 GPU 的实际实现方式。

顶点着色器（vertex shader）是一个完全可编程的阶段，它用于实现渲染管线中的几何处理阶段。几何着色器（geometry shader）也是一个完全可编程的阶段，它可以对图元（点、线或者三角形）的顶点进行操作，它也可以用于进行一些逐图元的着色操作、销毁图元或者是创建新图元等。曲面细分（tessllation）阶段和几何着色器都是可选的阶段，但并不是所有的 GPU 都支持这两个阶段，尤其是移动设备上的 GPU。

裁剪、三角形设置和三角形遍历阶段，都由固定功能的硬件进行实现。屏幕映射受到窗口（window）和视口（viewport）设置的影响，其内部包含了一个简单的缩放和重定位功能。像素着色器阶段是一个完全可编程的阶段。合并阶段尽管不是可编程的，但是它是高度可配置的，我们可以为其设定各种各样的操作。合并阶段实现了渲染管线中的合并功能，负责修改维护颜色缓冲、z-buffer、混合、模板缓冲以及其他任何与输出相关的缓冲区。像素着色器和合并阶段一起，组成了第 2 章中概念化的像素处理阶段。

随着时代的进步，GPU 流水线已经从之前的硬编码操作，向着更加灵活、更加可控的方向演变。而可编程着色器阶段的引入是这个演变过程中最重要的一步，下一小节我们将对各种可编程阶段的共同特性进行描述。

3.3 可编程着色器阶段

现代的着色器程序都使用了统一的着色器设计，这意味着顶点着色器、像素着色器、几何着色器以及与曲面细分相关的着色器，都共享一个通用的编程模型，它们内部的指令集架构（instruction set architecture，ISA）都是相同的。在 DirectX 中，实现这个模型的处理器叫做公共着色器核心（common-shader core），具有这个核心的 GPU 被称作拥有一个统一的着色器架构。这种架构背后的思想是，这些着色处理器可以用于执行很多的任务，而 GPU 可以视情况来对这些处理器进行分配。例如：相对于由两个三角形组成的一个大正方形，一组由大量微小三角形所组成的网格将会需要更多的顶点着色处理。一个具有独立顶点着色器核心池和像素着色器核心池的 GPU 意味着，为了保持所有的核心都处于忙碌状态，那么需要预先规定顶点着色器和像素着色器的任务比例。而通过统一的着色器核心，GPU 可以视情况来平衡这个负载。

想要对整个着色器编程模型进行描述有点过于复杂了，这远远超出了这本书的范围，而且已经有很多文档、书籍和网站做到了这件事情。着色器使用了类 C 的着色器语言（shading language）进行编写，例如 DirectX 中的 High-Level Shading Language（HLSL）和 OpenGL 中的 OpenGL Shading Language（GLSL）。为了提供硬件独立性，HLSL 可以被编译成虚拟机器码，它也被叫做中间语言（intermediate language，简称为 IL 或者 DXIL），这种中间表示还可以允许着色程序进行离线编译和存储，它会被 GPU 驱动程序转换为特定的 ISA。游戏主机上通常会避免这个中间语言步骤，因为主机系统只有一个 ISA。

32 位精度的浮点标量和浮点向量是最基础的数据类型，尽管向量只是着色器代码的一部分，而且并不被上面所提到的硬件原生支持。在现代 GPU 上，同样原生支持 32 位整数和 64 位浮点数。浮点向量通常用来表示位置（ $xyzw$ ）、向量、矩阵中的某一行、颜色（ $rgba$ ）或者纹理坐标（ $uvwq$ ）等。整数通常用来表示计数器、索引或者位掩码等。诸如结构体、数组和矩阵等聚合类型，也同样被 GPU 支持。

一次 draw call 会调用图形 API 来绘制一组图元，渲染管线也会相应执行它所对应的着色器。每个可编程的着色器阶段都包含两种类型的输入：统一输入（uniform input），它是指在一次 draw call 中不会发生改变的常量；可变输入（varying input），来自三角形的顶点或者光栅化的数据。例如：像素着色器中的光源颜色可

能是一个统一的值，在一次 draw call 中并不会发生变化；而三角形的表面位置则会根据屏幕上的每个像素发生变化，因此这个位置数据是可变的。纹理是一种特殊的统一输入，在过去，纹理总是会被视为一个应用于网格表面的颜色图像，而如今纹理可以是任意的阵列数据。

底层虚拟机为不同类型的输入提供了不同的寄存器，常量寄存器（constant register）用于存储统一输入，其数量远大于可变寄存器（用于存储可变输入输出）。这是因为可变输入输出只需要为每个顶点或者每个像素存储独立的数据即可，因此其所需要的数量是天然有限的。而统一输入在一次 draw call 中只会存储一次，然后会在所有的顶点或者像素中进行重用。虚拟机还有用于临时存储的通用临时寄存器（temporary register），所有类型的寄存器，都可以使用存储在临时寄存器上的整数数值，来进行数组索引。着色器虚拟机的输入输出如图 3.3 所示。

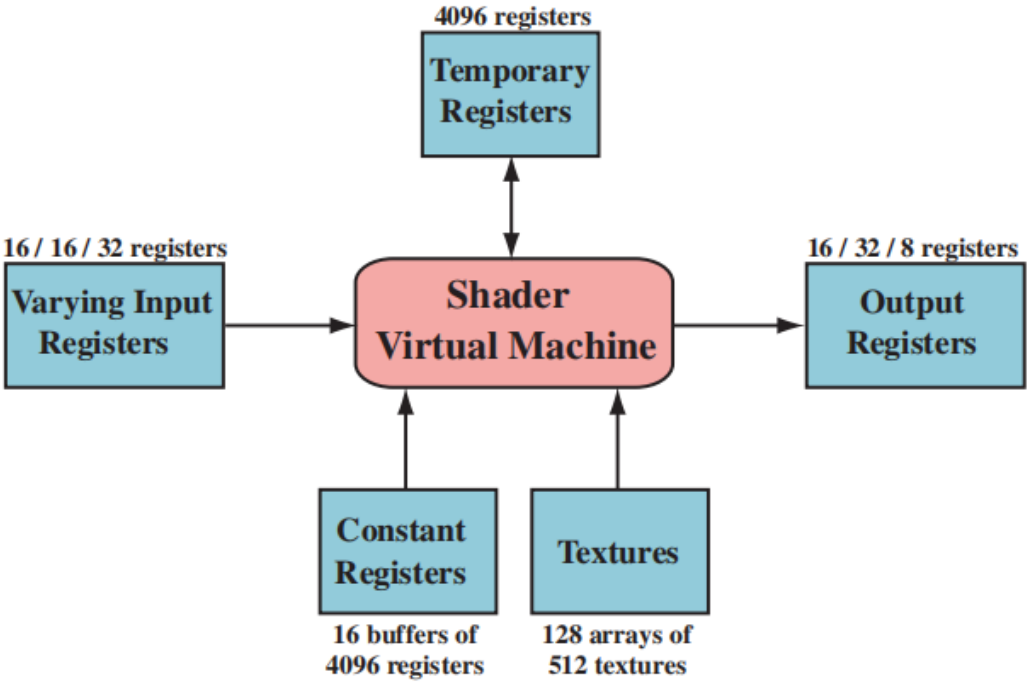


图 3.3：着色器模型（Shader Model）4.0 下，统一的虚拟机架构和寄存器布局。每个部分旁边都显示了最大的可用数量，从左到右的数字分别代表顶点着色器、几何着色器和像素着色器的限制。

图形计算中的常见操作和运算都可以在现代 GPU 上高效执行，着色语言通过操作符来暴露最常用的操作，例如加法和乘法的操作符是 + 和 *；其余的操作可以通过使用内置函数（intrinsic function）来提供，这些内置函数针对 GPU 进行了专门优化，例如：`atan()`，`sqrt()`，`log()`。还有一些函数可以提供更加复杂的操作，例如向量的标准化、求反射向量、向量的叉乘、求矩阵的转置以及行列式等。

流程控制（flow control）这个术语，是指使用分支指令来改变代码执行的流程。与流程控制相关的指令是用来实现一些高级语言结构的，例如“if”、“case”以及各种类型的循环。着色器支持两种类型的流程控制，包含静态流程控制（static flow control）以及动态流程控制（dynamic flow control）。其中静态流程控制的分支情况会基于统一输入的值，这意味着在一次 draw call 中，该代码的流程是恒定不变的。静态流程控制最主要的好处在于，它可以在各种不同的情况下（例如不同数量的光源）使用相同的着色器，并且在这个过程中没有任何的线程发散，因为所有调用都会进入相同的代码路径。动态流程控制则基于可变输入的值，它意味着每个片元都可以执行不同的代码，其功能比静态流程控制更加强大，但是也更消耗性能，尤其是当着色器调用之间，代码流程不规则变化时。

3.4 可编程着色及其 API 的演变

可编程着色框架的想法可以追溯到 1984 年 Cook 所提出的着色树（shade tree）[287]，图 3.4 展示了一个简单的着色器及其着色树结构。在 1980s，基于着色树的想法，开发了 RenderMan Shading Language [63, 1804]，如今它与其他不断发展的规范一起，例如 Open Shading Language（OSL）项目等，仍用于电影制作的渲染[608]。

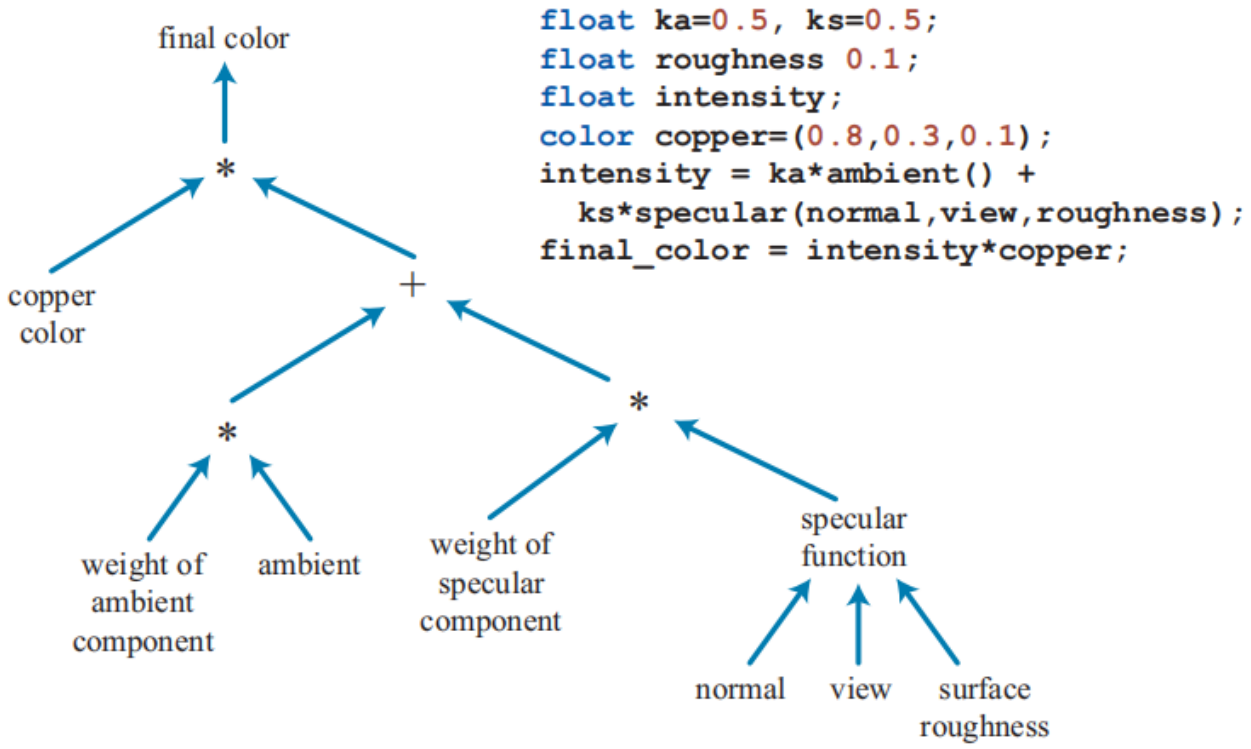


图 3.3：左侧是一个铜材质的着色树，右侧则是该着色树所对应的着色语言代码[287]。

3dfx 交互公司于 1996.10.1，首次引入了消费级的图形硬件，图 3.5 展示了从这年开始的时间线。3dfx 的 Voodoo 显卡可以高质量的渲染游戏《雷神之锤（Quake）》，其强大的性能使得它被广泛采用。该显卡实现了一个固定功能的管线，在 GPU 原生支持可编程着色器之前，还可以通过多个 pass 来实现实时可编程着色的操作。1999 年发行的《雷神之锤 3：竞技场》，其脚本语言（游戏引擎）也在这一领域获得了广泛的商业成功。在本章节开始的时候，我们提到 NVIDIA 的 Geforce256 是第一个被称作 GPU 的图形硬件，但是它仅仅只是可配置的，而不是可编程的。

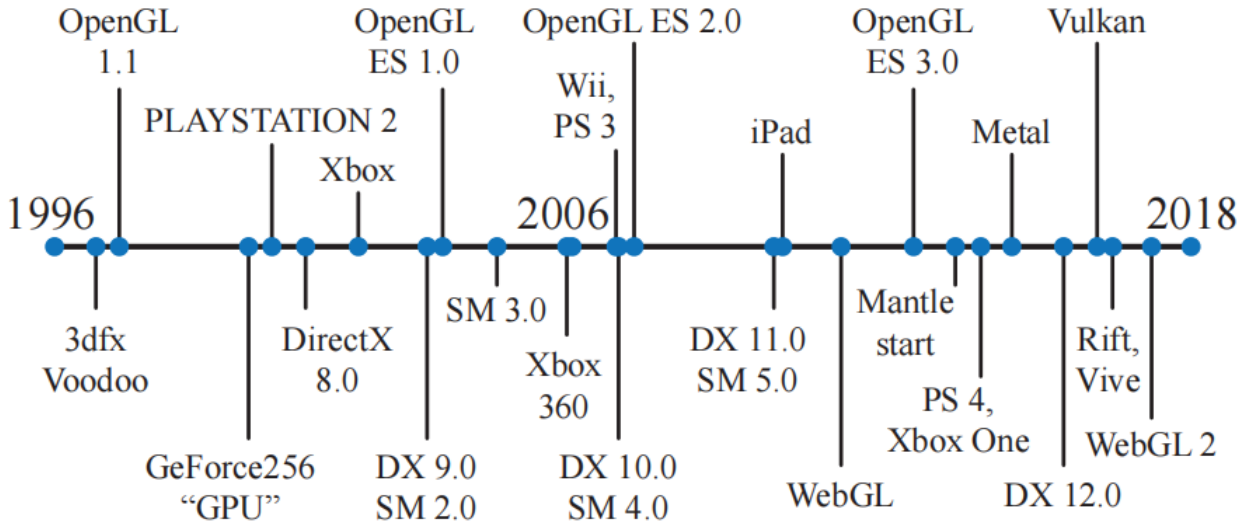


图 3.5：一些图形 API 和图形硬件发布的时间线。

2001 年初，NVIDIA 推出了 Geforce3 显卡，这是第一个支持可编程顶点着色器的 GPU [1049]，它通过 DirectX 8.0 来暴露相关接口，并可以扩展到 OpenGL。这些着色器通过一种类似于汇编的语言来进行编程，并通过驱动程序，来将其动态地转换为微代码。DirectX 8.0 同样也包含了像素着色器，但是它并没有实现真正的可编程性，它仅支持“有限”程序被驱动程序转换为纹理混合状态，进而将其链接到“寄存器组合器（register combiner）”。这些程序不仅受限于长度（最大只支持 12 条指令），同时也缺少很多重要的功能。Peercy 等人[1363]对 RenderMan 的研究表明，依赖纹理读取（dependent texture reads，是指需要使用第一个纹理读取的返回值，来确定第二个纹理读取的位置）和浮点数据对于实现真正的可编程性至关重要。

这时的着色器不允许包含流程控制（分支），如果着色器中包含分支，那么就需要将两个分支都执行一次，然后在结果中进行选择或者插值来模拟分支。DirectX 定义了着色器模型（Shader Model）的概念来区分具有不同功能的着色器。2002 年微软推出了包含 Shader Model 2.0 的 DirectX 9.0，它支持真正可编程的顶点着色器和像素着色器。同样的功能也可以在 OpenGL 下使用各种扩展来实现。Shader Model 2.0

支持了任意的依赖纹理读取，并支持 16 位的浮点数值，这最终满足了 Peercy 等人所确定的需求。Shader Model 2.0 还扩展了着色器资源的范围（例如指令、纹理以及寄存器），着色器因此可以生成更加复杂的特效，同时增加了对流程控制的支持。随着着色器程序长度和复杂度的不断增加，这使得使用汇编模型来开发着色器变得越来越繁琐。幸运的是，DirectX 同样支持了 HLSL，这种着色语言由 Microsoft 和 NVIDA 合作开发。大约在同一时间，OpenGL ARB（Architecture Review Board，架构审查委员会）也推出了 GLSL，这是一种用于 OpenGL 的类似着色语言[885]。这些语言的设计深受 C 语言语法和设计哲学的影响，同时也包含了来自 RenderMan 着色语言中的一些元素。

Shader Model 3.0 于 2004 年推出，并增加了动态流程控制，这使得着色器更加强大。它还将可选的功能特性纳入了需求列表，进一步扩大了可使用资源的范围，在顶点着色器中添加了对纹理读取的有限支持。新世代的游戏主机分别于 2005 年底（Microsoft Xbox 360）和 2006 年底（Sony 计算机娱乐 PS3）推出，他们都配备了支持 Shader Model 3.0 的 GPU。Nintendo 于 2006 年底推出了 Wii 主机，它是最后一个仅支持固定功能 GPU 的著名主机。如今纯固定功能的管线已经彻底消失了，着色器语言已经发展到了可以使用各种工具创建和管理它们的地步。图 3.6 展示了一种这样的工具，它使用了 Cook 的着色树概念。

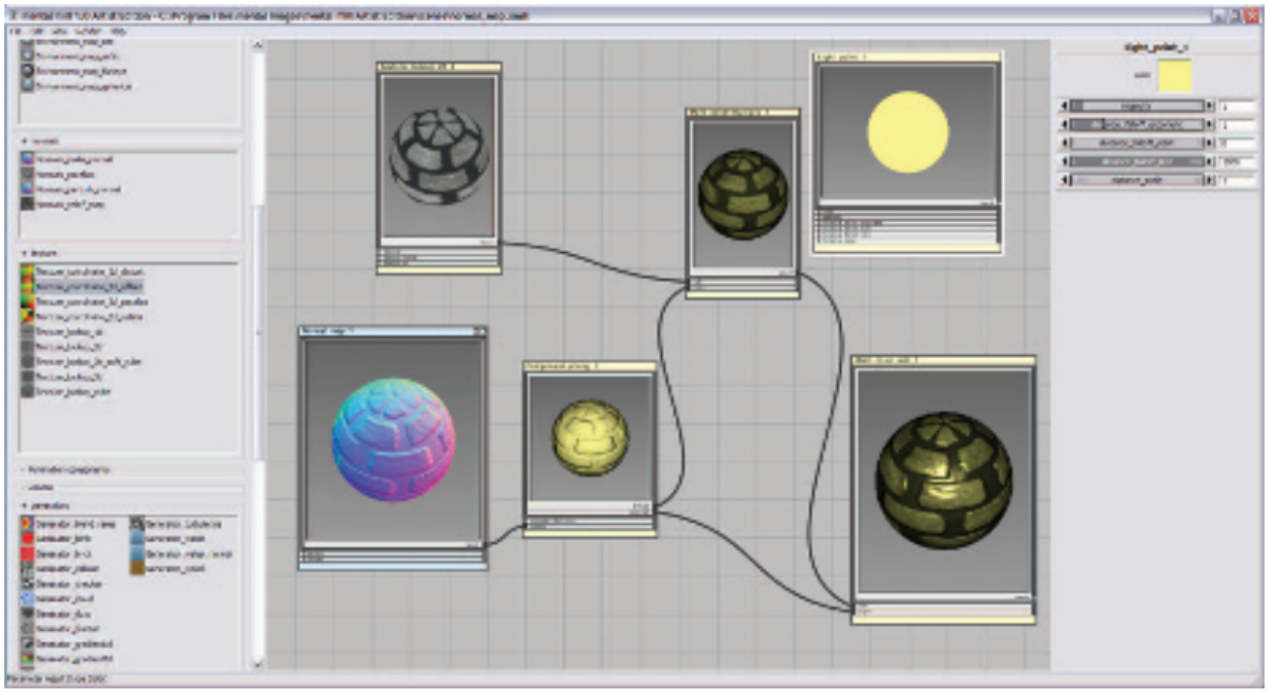


图 3.6：一个用于 shader 编辑的可视化工具。各种功能都被封装到了不同的功能块中，如左侧的可选按钮。当你选中一个功能块的时候，你可以对它的参数进行调整，如右侧所示。每个功能块的输入输出都和其他的功能块相连接，最终能够计算出着色的结果，如中间区域的右下角所示。

着色器可编程性的下一次跨域也出现在 2006 年底，DirectX 10.0 推出了 Shader Model 4.0 [175]，它引入了几个重要特性，例如几何着色器和流式输出。Shader Model 4.0 包含了一个针对所有着色器（顶点着色器、像素着色器和几何着色器）的统一编程模型，即我们在前文中描述过的标准着色器设计。并且它进一步扩大了资源范围，同时支持了整数类型的数据（包括位运算等操作）。在 OpenGL 3.3 中所引入的 GLSL 3.0，也推出了一个类似的着色器模型。

在 2009 年发布的 DirectX 11 和 Shader Model 5.0 中，增加了曲面细分着色器和计算着色器，计算着色器也被叫做 DirectCompute。这次发布还关注了如何提高 CPU 多线程处理的效率，这将在章节 18.5 中进行讨论。OpenGL 在 4.0 版本中添加了曲面细分，在 4.3 版本中添加了计算着色器。DirectX 和 OpenGL 的发展方式不太一样，但是二者都针对特定的发布版本，设定了一定的硬件支持级别。Microsoft 控制着 DirectX API，因此他可以直接跟独立硬件厂商（IHV，例如 AMD、NVIDIA 和 Intel）、游戏开发者以及计算机辅助设计软件公司进行合作，从而确定需要暴露哪些功能。OpenGL 则是由一个硬件和软件开发商联盟进行开发的，并由非盈利性的 Khronos 组织进行管理。由于所涉及的公司数量很多，因此 OpenGL 中一些功能特性的发布时间，往往比 DirectX 晚。但是 OpenGL 允许使用供应商专用的，或者是通用的扩展，从而能够在官方支持发布之前，提前使用最新的 GPU 功能。

图形 API 的下一个重大变化是由 AMD 于 2013 年提出的 Mantle API，它是 AMD 和电子游戏开发商 DICE 一起合作开发的，其核心想法是去除用于图形驱动程序的大量开销，将控制权直接交给开发者。除此之外，该技术还进一步支持了 CPU 多线程的高效处理，这一类 API 专注于如何减少 CPU 花费在驱动上的时间，以及如何更加高效的利用 CPU 的多个核心（第 18 章）。这个由 Mantle 首创的想法被 Microsoft 所采纳，并于 2015 年推出了全新的 DirectX 12.0。这里请注意，DirectX 12 并没有增加更多的 GPU 功能——DirectX 11.3 和 DirectX12 具有完全相同的硬件特性。这两个 API 都可以用于向虚拟现实系统发送并显示图形，例如 Oculus Rift 和 HTC Vive。DirectX 12 是对 API 的彻底重构，它可以更好的映射到现代 GPU 架构。低开销的驱动程序对于 CPU 驱动所导致性能瓶颈的应用程序非常有用，或者是可以利用更多的 CPU 处理核心来获得更好的图形性能[946]。这些特性想要从早期图形 API 进行移植是非常困难的，并且过于简单的实现反而可能会导致降低性能表现[249, 699, 1438]。

Apple 于 2014 年推出了自家叫做 Metal 的低开销 API，Metal 首先用于移动设备，例如 iPhone 5s 和 iPad Air。一年后，新的 Mac 电脑也可以通过 OS X El Capitan 来使用 Metal。除了提高效率之外，减少 CPU 的占用还可以降低功耗，这对于移动

设备而言是非常重要的。Metal 有着属于自己的着色器编程语言，同时适用于图形程序和 GPU 计算程序。

AMD 将自身 Mantle 的工作贡献给了 Khronos 组织，后者于 2016 年推出了新一代的 API，叫做 Vulkan。与 OpenGL 一样，Vulkan 可以用于多个操作系统。Vulkan 使用了一种被称为 SPIR-V 的全新高级中间语言，它可以同时用于着色器表示和通用 GPU 计算。这些预编译的着色器代码是可移植的，因此可以在任何支持该功能特性的 GPU 上进行使用[885]。Vulkan 也可以被用于非图形的 GPU 计算，这些计算通常并没有一个用于显示画面的窗口[946]。与其他低开销驱动的 API 相比，Vulkan 的一个显著区别在于，它具有十分强大的跨平台特性，可以在从工作站到移动设备的很多系统上进行使用。

在移动设备上一般会使用 OpenGL ES，其中“ES”代表的是嵌入式系统（embedded system），因为这个 API 是针对移动设备进行开发的；这是由于标准 OpenGL 中的一些调用结构十分臃肿和缓慢，并且还需要对其中很少使用到的功能进行支持。OpenGL ES 1.0 于 2003 年发布，它是 OpenGL 1.3 的一个简化版本，描述了一个固定功能的管线。虽然 DirectX 是和支持它的硬件同步发布的，但是移动设备上的图形支持没有以同样的方式进行。例如：2010 年发布的初代 iPad，它实现了 OpenGL ES 1.1。OpenGL ES 2.0 于 2007 年发布，它基于 OpenGL 2.0，提供了可编程着色功能，但是并不包含固定功能的组件，因此无法向后兼容 OpenGL ES 1.1。OpenGL ES 3.0 于 2012 年发布，它提供了许多功能，例如多重渲染目标（Mutiple Render Target，MRT）、纹理压缩、变换反馈（transform feedback）、实例化、更广泛的纹理格式和纹理模式，以及对着色器语言的改进等。OpenGL ES 3.1 中添加了计算着色器，3.2 中添加了几何着色器和曲面细分着色器以及其他一些特性。第 23 章详细讨论了移动设备的架构。

OpenGL ES 的一个分支是基于浏览器的 WebGL，它通过 JavaScripts 进行调用。它的第一个版本于 2011 年发布，并且在大多数移动设备上都可以使用，其功能大概相当于 OpenGL ES 2.0。与 OpenGL 一样，我们可以通过使用扩展来添加更多的高级 GPU 特性。WebGL 2 假设了 OpenGL ES 3.0 是支持的。

WebGL 特别适合用于在课堂上进行教学和试验：

- 它是一个跨平台的 API，可以在所有 PC 上和几乎所有的移动设备上运行。
- 它由浏览器提供驱动程序。即使某个浏览器并不支持某个 GPU 或者是某个扩展，通常也会有另一个浏览器是支持的。

- 其代码是解释执行的，而不是编译后执行的。这意味着其开发环境十分轻量，只需要一个文本编辑器就可以进行开发。
- 大部分浏览器都内嵌了一个调试器（debugger），可以对运行在任何网站上的代码进行审查和调试。
- 你可以将程序上传到一个网站来进行部署，例如 Github。

还有一些更加高级的场景图或者是特效库，例如 three.js [218]，我们可以通过它来很方便的实现更加复杂的效果，例如阴影算法、后处理、基于物理的渲染以及延迟渲染等。

3.5 顶点着色器

顶点着色器是图 3.2 中所描述的功能流水线中的第一个阶段。虽然这是程序员可以直接进行控制的第一个阶段，但是值得注意的是，在进入这个阶段之前，就已经存在一些数据计算了。这在 DirectX 中叫做输入汇编器（input assembler）[175, 530, 1208]，几个数据流被编织在一起，形成了顶点集合和图元集合，并向下发送给管线。例如：一个物体可以用一组位置和一组颜色来进行表示，输入汇编器会通过创建带有位置信息和颜色信息的顶点，从而创建这个物体的三角形（点或者线段）。第二个物体可以使用与第一个物体相同的位置数组（但是其模型变换矩阵不同），以及一个不同的颜色数组，与数据表示相关的内容详见章节 16.4.5。在输入汇编器中还支持实例化（instancing），这允许一个物体可以在每个实例中，使用不同的数据的来进行绘制，所有这些绘制都只对应一次 draw call，有关实例化使用的内容详见章节 18.4.2。

一个三角形网格由一组顶点构成，每个顶点都对应物体表面上的一个特定位置。除了位置之外，每个顶点上还具有一些其他可选的属性，例如颜色和纹理坐标。表面法线同样也会在网格顶点的位置上进行定义，虽然这看起来有点奇怪。在数学上，每个三角面都有一个明确定义的表面法线，而且可以直接使用这个三角形法线来进行着色计算，这看起来是更加合理的。但是在渲染过程中，三角形网格通常会被用来表示一个潜在的曲面，而顶点法线则被用来表示这个曲面的朝向，而不是这个三角形网格本身的朝向。我们会在章节 16.3.4 中讨论计算顶点法线的方法。图 3.7 展示了两个三角形网格的侧视图，分别代表了两个不同的曲面，其中一个曲面是光滑的，而另外一个曲面则带有尖锐的折痕。

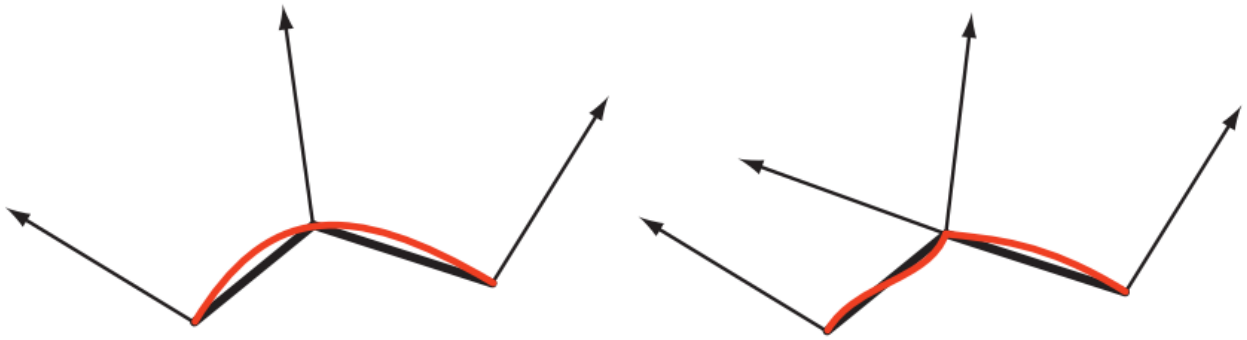


图 3.7：两个用于表示曲面（红色部分）的三角形网格（黑色加粗线条，并带有顶点法线）的侧视图。左侧平滑的顶点法线用于表示一个光滑的曲面；右侧的中间顶点上具有两个方向不同的顶点法线，用于表示曲面上的折痕。

顶点着色器是处理这些三角形的第一个阶段。但是用于描述三角形是如何组成的数据（点和点之间的关系）对于顶点着色器而言是不可用的，正如顶点着色器的字面意思，它只会对传入的顶点进行处理。顶点着色器提供了一种用于修改、创建或者忽略三角形顶点数据的方法，这些数据可以是颜色、法线、纹理坐标和位置等。通常顶点着色器程序会将顶点从模型空间变换到齐次裁剪空间中（章节 4.7），在最极端的情况下，顶点着色器也必须要输出顶点的位置。

顶点着色器与之前所描述的统一着色器模型非常相似，每个传入的顶点都会经过顶点着色器的处理，然后该程序会输出一些数据，这些数据将会用于三角形或者线段的插值。顶点着色器无法创建或者销毁顶点，并且一个顶点上的计算结果也无法传递给另一个顶点，因为每个顶点都是单独处理的，GPU 上任意数量的着色处理器都可以并行的应用于输入的顶点流上。

输入装配（input assembly）通常是一个在顶点着色器之前发生的阶段，这也是物理模型通常不同于逻辑模型的一个例子。在物理模型中，获取数据来创建一个顶点可能会发生在顶点着色器中，这是因为驱动程序会悄悄的在每个着色器前添加一些适当的指令，这些过程对于程序员而言是不可见的。

后续的章节中展示了一些顶点着色器能够实现的效果，例如用于动画关节的顶点混合以及轮廓渲染（描边）等。顶点着色器的其他用于包括：

- 物体生成：仅创建一次模型，并通过顶点着色器对其进行变形。
- 使用蒙皮技术和变形技术来设置角色的身体动画和面部动画。
- 程序化变形：例如旗帜、布料和水面的运动。

- 粒子创建：通过向流水线发送简并（无面积）网格，并根据需要来设定它们的位置，从而来模拟粒子效果。
- 透镜畸变、热雾、水波纹、书页卷曲以及其他特效，可以通过将整个帧缓冲的内容作为一个纹理，然后将其应用在一个正在经历变形，并且屏幕对齐的网格上进行实现。
- 通过使用顶点纹理来获取并应用地形的高度场[40, 1227]。

图 3.8 展示了使用顶点着色器完成的一些变形操作。

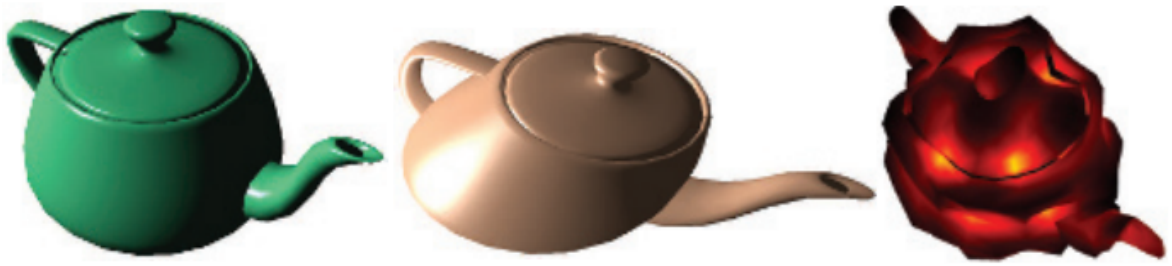


图 3.8：左侧是一个正常的茶壶。中间的茶壶使用顶点着色器来对其进行了剪切变换。右图则使用了噪声函数来创建了一个扭曲的场，从而对模型进行了扭曲。

3.6 曲面细分阶段

曲面细分阶段允许我们绘制曲面，GPU 的任务就是将每个曲面描述都转换成一组三角形。曲面细分阶段是一个可选的 GPU 特性，它首次出现在 DirectX 11 中。OpenGL 4.0 和 Open GL ES 3.2 也同样支持曲面细分着色器。

使用曲面细分阶段有几个好处。描述一个曲面往往要比提供三角形网格本身更加紧凑，除了节省内存之外，当场景中存在一些不断变化的角色或者物体时，这个功能还可以防止 CPU 与 GPU 之间的总线带宽成为程序的性能瓶颈。对于一个给定的相机视角，曲面细分可以生成适当的三角形数量，这样的曲面可以被高效渲染。例如：现在有一个距离相机很远的小球，它仅仅需要使用很少的三角形即可；当这个小球距离相机很近的时候，它也可以用几千个三角形来进行表示。从而获得更好的效果。这个控制层次细节（level of detail）的能力允许应用程序控制自己的性能开销，例如：为了在一个性能较弱的 GPU 上仍然保持合适的帧率，可以使用较低质量的网格。使用平坦表面进行表示的模型也可以转换为更加细密的三角形网格[1493]，并根据需要来进行弯曲变形；或者也可以进行细分，从而减少着色计算的压力[225]。

曲面细分阶段同样包含三个字阶段。在 DirectX 中，它们分别叫做壳着色器（hull shader）、曲面细分器（tessellator）和域着色器（domain shader）。在 OpenGL

中，壳着色器叫做细分控制着色器（tessellation control shader），域着色器叫做细分评估着色器（tessellation evaluation shader），这个名称对于功能的描述更加清晰，但是稍微有点冗长。固定功能的曲面细分器在 OpenGL 中叫做图元生成器（primitive generator），正如它的名字那样，它确实可以生成图元。

我们将在第 17 章深入讨论如何指定曲线和曲面，并对其进行细分操作，这里我们只简要总结一下每个细分阶段的目的。首先，壳着色器的输入是一个特殊的面片（patch）图元，它包含了若干个定义细分表面、Bezier 面片、以及其他类型曲线元素的控制点。壳着色器包含两个功能：第一，它会告诉细分器需要生成多少个三角形，以及如何对它们进行配置；第二，它会对每个控制点进行处理。壳着色器也可以选择对输入的面片进行修改，根据要求添加或者移除一些控制点。壳着色器会将处理好的控制点和曲面细分相关的控制数据一起发送给域着色器，如图 3.9 所示。

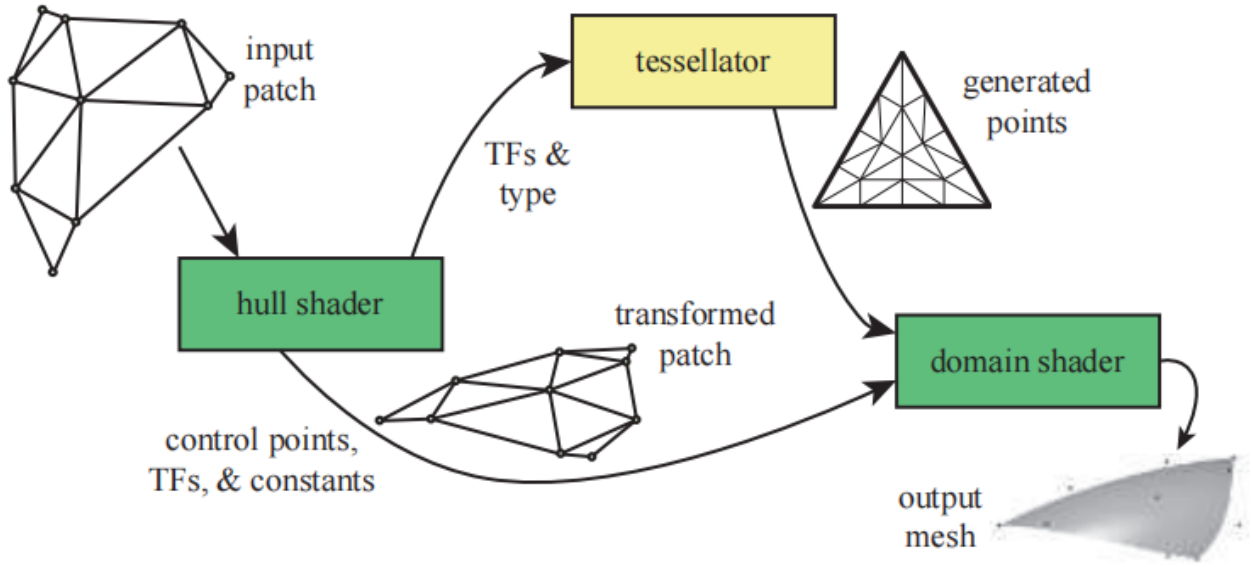


图 3.9：曲面细分阶段。壳着色器会将一组控制点所定义的面片作为输入，它将曲面细分因子（TF）及其类型发送给固定功能的细分器。控制点集会按照壳着色器进行变换，并与细分因子以及相关的面片常量一起，被发送给域着色器。曲面细分器会创建一系列的顶点，并生成该顶点的重心坐标。这些参数经过域着色器的处理，最终会生成一个三角形网格（图中展示了用于参考的控制点）。

曲面细分器是流水线中一个固定功能的阶段，并且只用于曲面细分着色器。它的任务是添加新的顶点，并发送给域着色器进行处理。壳着色器还会给曲面细分器发送一些额外信息，来告诉它我们需要的是哪一种细分曲面：三角形、四边形（quadrilateral）还是等值线（isoline）。其中等值线是一组线条，有时候也会用于头发渲染[1954]。壳着色器发送给曲面细分器的另一个重要参数是曲面细分因子（tessellation factor，在 OpenGL 中叫做曲面细分等级，tessellation level），它

有两种类型，分别是内边缘和外边缘。内边缘因子有两个，它决定了在三角形或者四边形内部进行细分的次数；外边缘因子决定了每个外部边缘被分割的次数（章节 17.6）。图 3.10 展示了增大曲面细分因子的不同效果。通过对参数的分开控制，无论曲面内部是如何细分的，我们都可以让相邻的曲面边界与曲面细分相匹配，这种边缘匹配避免了在面片相接触的地方产生裂缝，或者是产生其他的着色瑕疵。这些顶点也会被指定重心坐标（章节 22.8），它代表了顶点在曲面上的相对位置。

图 3.10：改变曲面细分因子所带来的影响。这个 Utah 茶壶由 32 个面片组成，从左到右的内部曲面细分因子和外部曲面细分因子分别是 1，2，4，8。[1493]

壳着色器总会输出一个面片和一个控制点集。但是，壳着色器也可以通过向曲面细分器发送一个为 0 或者更低（或者一个非数字，NaN）的外部细分水平，来表示要丢弃一个面片。否则曲面细分器就会生成一个网格，并将其发送给域着色器。当域着色器每次进行调用的时候，都会使用来自壳着色器的曲面控制点，来计算每个顶点的输出值。域着色器具有一个和顶点着色器类似的数据流模式，来自曲面细分器的每个顶点都会被处理，并生成一个相应的输出顶点。然后生成的三角形会被输入到管线中的下一个阶段。

虽然这个系统听起来十分复杂，但是其中的每个着色器都可以相当简单，它采取这样一个结构的目的是为了提高效率。进入壳着色器的面片通常会很少或者基本没有被修改，壳着色器还可以使用面片的估计距离或者屏幕尺寸，来动态计算曲面细分因子，例如用于地形渲染[466]。又或者，这些面片可以全部由应用程序进行计算，然后壳着色器可以简单的为所有面片都提供一组固定的值。曲面细分器的功能要更加复杂一点，但是它的功能是完全固定的：生成顶点并计算它们的位置，指定它们所构成的三角形和线段。为了提高计算效率，这个数据放大的过程会在着色器之外执行[530]。域着色器会为每个顶点生成重心坐标，并在面片的计算方程中使用重心坐标来生成顶点的位置、法线、纹理坐标以及其他需要的顶点信息。图 3.11 展示了这个过程的一个例子。

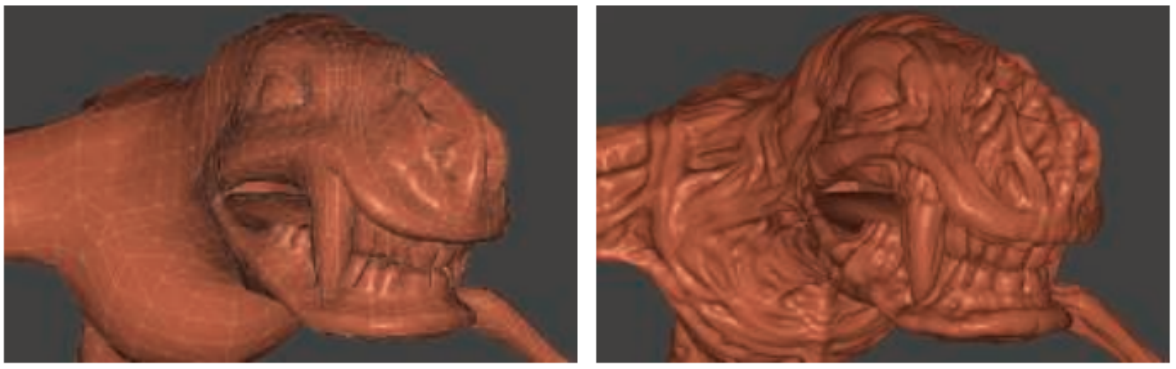


图 3.11：左图是由大概 6000 个三角形所组成的原始网格。右图中的每个三角形都使用了 PN 三角形（一种特殊的 Bezier 面皮）细分进行了细分和替换。[1301]

3.7 几何着色器

几何着色器可以将一种图元转换为另一种图元，这是曲面细分着色器所无法实现的。例如：我们可以为每个三角形都创建边界线段，从而将一个三角形网格转换成一个线框模型。或者我们可以使用面向观察者的狭长四边形来替换边界线段，从而生成一个具有较粗边界的线框渲染[1492]。随着 2006 年底发布的 DirectX 10，几何着色器被添加到了硬件加速的图形管线中。它位于管线中的曲面细分着色器之后，同样也是可选的。虽然它是 Shader Model 4.0 中的一部分，但是它并没有在早期的着色模型中进行使用。Open GL 3.2 和 OpenGL ES 3.2 同样支持几何着色器。

图 3.12：几何着色器的输入是一些简单类型：点、线段、三角形。最右边的两个图元包含了与线段和三角形相邻的顶点。也可以使用一些更加复杂的面片类型。

几何着色器的输入是一个独立物体和与其相关联的顶点。这些物体通常由一个条状三角形、一个线段或者仅仅是一个点组成，其他扩展的图元也可以在几何着色器中进行定义和处理。特别地，可以传入一个三角形之外的三个附加顶点；并且可以使用一条折线上的两个相邻顶点，如图 3.12。在 DirectX 11 和 Shader Model 5.0 中，我们还可以传入一些更加复杂的面片，这个面片最多可以拥有 32 个控制点。这也表明，在曲面细分阶段生成面片的效率要很高[175]。

几何着色器会对这些输入的图元进行处理，会输出 0 个或者更多数量的顶点，这些顶点可以是点、折线或者条状三角形。请注意，在几何着色器中生成的图元不可以直接

输出，但是可以通过编辑顶点、添加新图元和移除其他图元的方式，来对网格进行选择性的修改。

几何着色器的设计目的是对输入的顶点数据进行修改，或者是创建有限数量的副本。例如：其中一个用途是生成六个变换后的数据副本，从而同时渲染一个立方体的六个面，详见章节 10.4.3。它也可以用于高效的创建级联阴影贴图（cascaded shadow map，CSM），从而生成高质量的阴影。其他利用几何着色器的算法包括：从点数据中创建大小可变的粒子；沿着模型轮廓挤压出鳍片从而模拟毛发渲染；找到物体的边缘从而用于阴影算法等。在图 3.13 中展示了更多的例子，我们将在本书的剩余部分讨论几何着色器的这些实际应用。

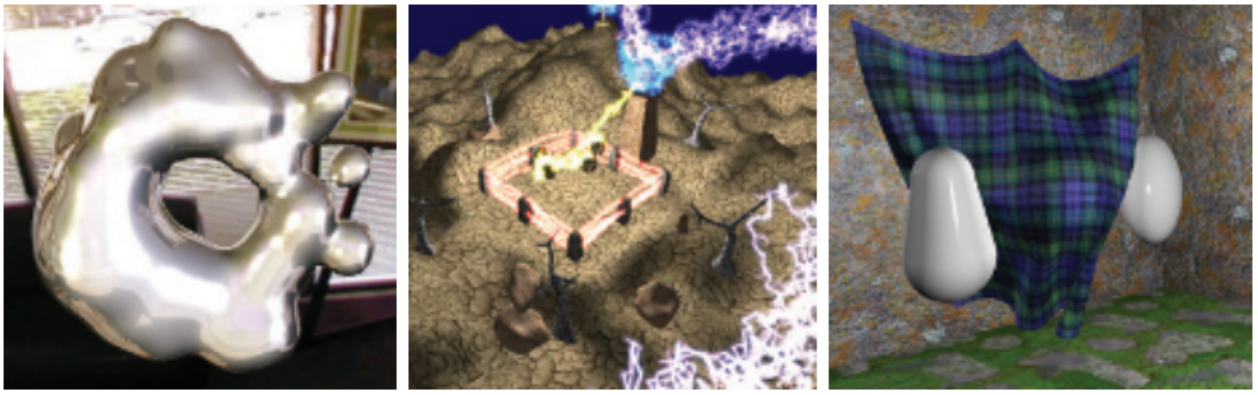


图 3.13：一些几何着色器（GS）的应用。左边：使用几何着色器来对元球（metaball）进行动态等值面曲面细分。中间：通过使用几何着色器来完成线段的分形细分，并使用几何着色器来创建用于显示闪电的告示牌。右边：使用流式输出的顶点和几何着色器进行布料模拟。[1300]

DirectX 11 为几何着色器添加了使用实例化的功能[530, 1971]，它允许几何着色器在任意给定的图元上执行一定次数，在 OpenGL 中这是通过使用一个调用计数器来指定的。几何着色器最多可以输出四个数据流（stream），每个数据流都可以被发送到渲染管线的下一阶段进行处理，所有这些数据流都可以选择性地发送到流式输出的渲染目标中。

几何着色器会保证按照图元的输入顺序来输出图元。这个排序会对执行性能产生影响，因为如果有很多个着色器核心并行执行的话，那么为了保证图元的输出顺序与输入顺序相同，则必须要将所有执行后的结果保存下来并进行排序。这个因素和其他的一些因素一起，不利于几何着色器在一次调用中大量复制或者创建图形[175, 530]。

当一次 draw call 命令被提交之后，渲染管线中只有三个地方可以在 GPU 上创建工作：光栅化、曲面细分着色器和几何着色器。其中考虑到所需要的资源和内存，几何着色器的行为是最不可预测的，因为它是完全可编程的。在实践中，几何着色器很少

会被使用，因为它和 GPU 并行计算的优势并不相符；在一些移动设备上，几何着色器是使用软件进行实现的，因此在这些设备上也不鼓励使用几何着色器[69]。

3.7.1 流式输出

使用 GPU 管线的标准方式是通过顶点着色器向 GPU 发送数据，然后将生成的结果三角形进行光栅化，最后在像素着色器中对这些数据进行处理。在过去，数据总是会直接穿过整个管线，直接输出到屏幕上，中间生成的数据都无法进行访问。流式输出（stream output）的想法是在 Shader Model 4.0 中引入的，在顶点被顶点着色器处理完成之后（这里还可以选择曲面细分和几何着色器），这些数据除了被发送到光栅化阶段之外，还可以通过一个流（即一个有序数组）来进行输出。事实上，我们还可以完全关闭光栅化阶段，然后将管线作为一个纯粹的、非图形的流处理器。这些处理过的数据可以通过流式输出从管线中返回，从而允许对其进行迭代处理。这类操作在模拟流动的水体，或者其他粒子特效的时候十分有用，我们将在章节 13.8 对这个问题进行讨论。它还可以用于对模型进行蒙皮操作，然后让这些顶点数据可以重复使用（章节 4.4）。

流式输出只能以浮点数的形式返回数据，因此它可能会占用很多存储空间。流式输出是作用于图元的，而不是直接作用于顶点的，这就意味着如果我们将一个网格输入到管线中，这个网格中的每个三角形都会生成自己的集合，每个集合都会包含三个输出的顶点，而原始网格中任何共享的顶点都会丢失。因此，在实际使用中，通常会直接将顶点作为一个点集图元直接输入到管线中。在 OpenGL 中流式输出被叫做变换反馈（transform feedback），因为流式输出的大部分用途都是对顶点进行变换，然后再将它们返回进行其他处理。在流式输出中，图元保证会按照它们的输入顺序进行输出，这意味着流式输出需要对输入到管线中的顶点顺序进行维护[530]。

3.8 像素着色器

如上一章所述，在顶点着色器、曲面细分和几何着色器完成操作之后，输出的片元将会进行裁剪和设置，从而进行下一步的光栅化。光栅化是管线中相对固定的处理步骤，它不具备任何可编程性，仅仅是某些地方可以进行自定义配置。在这一步中，会对每个三角形进行遍历，从而确定它所覆盖的像素，光栅化器也会粗略计算一个三角形所覆盖的像素单元格面积（章节 5.4.2）。三角形中部分与像素重叠、或者完全与像素重叠的部分被叫做一个片元（fragment）。

三角形顶点上的数值，包括 z-buffer 中的 z 值在内，每个被三角形所覆盖的像素都会使用这些数据进行插值。这些插值生成的数据会被发送到像素着色器（pixel

shader）中，在像素着色器中会对片元进行处理。在 OpenGL 中像素着色器被称为片元着色器（fragment shader），这可能是一个更好的名称，但是为了保证前后文的一致性，我们会在本书中将其叫做像素着色器。点图元和线段图元也会为其所覆盖的像素创建相应的片元，并发送到像素着色器中。

跨三角形执行的插值操作是由像素着色器程序所指定的。通常我们会使用透视正确的插值（perspective-correct interpolation），这样像素表面位置之间的世界空间距离，就会随着物体之间距离的增加而增加，即物体的深度值呈线性分布。一个典型的例子就是渲染一条延伸到地平线尽头的铁轨，在距离相机很远的地方，铁轨的两个轨道看起来会更加靠近，因为在靠近地平线的地方，每个像素所代表的实际距离会更远。我们还可以使用其他类型的插值操作，例如屏幕空间插值，它不会考虑透视投影所带来的影响。在 DirectX 11 可以进一步对插值的时机和插值的方式进行控制[530]。

在编程中，顶点着色器程序的输出，在经过三角形（或者线段）插值之后，会成为像素着色器程序的输入。随着 GPU 的不断发展，其他的一些输入数据也逐渐暴露给了像素着色器，例如：从 Shader Model 3.0 开始，我们可以在像素着色器中使用片元的屏幕空间位置；以及会有一个 flag 来标识三角形的哪一侧是可见的，这一点有时候是很重要的。例如在一个 pass 中，我们想要在一个三角形的正面和背面渲染不同材质的时候，这个 flag 会很有用。

有了这些输入数据，通常像素着色器就可以计算并输出一个片元的颜色值。它也可以生成一个不透明度或者选择性修改片元的深度值，在后续的合并阶段中，这些数值可以用于修改存储在像素上的值。光栅化阶段生成的深度值也可以在像素着色器中进行修改。模板缓冲（stencil buffer）通常是不可修改的，而是会直接将其发送给合并阶段；在 DirectX 11.3 中，也允许着色器对模板缓冲进行修改。在 Shader Model 4.0 中，诸如雾效计算和透明度测试（alpha test）等操作，都从合并阶段转移到了像素着色器中进行执行[175]。

像素着色器还有一个独有能力，那就是将一个输入片元丢弃，也就是说不生成任何输出。图 3.14 展示了像素着色器丢弃片元的几个例子。在过去的固定渲染管线中，裁剪平面（clip plane）是一个只能进行配置的选项，后来则可以在顶点着色器中进行指定。随着我们拥有了可以在像素着色器中丢弃片元的能力，这个功能还可以以任何我们想要的方式进行实现，例如决定裁剪空间是“与”操作还是“或”操作。

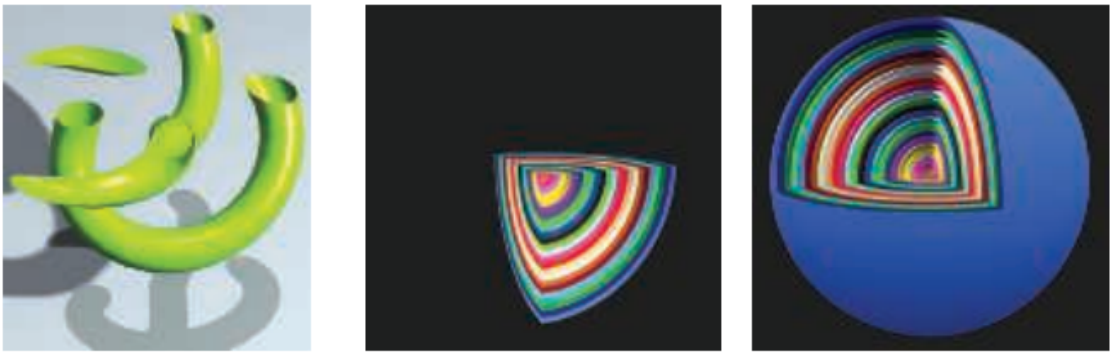


图 3.14：用户自定义的裁剪平面。左侧：使用单个水平裁剪平面对物体进行了切割。中间：嵌套的球体被三个平面所裁剪。右侧：只有位于全部三个裁剪平面之外的球体表面才会被裁剪掉。[218]

在早期 GPU 管线中，像素着色器只能将结果输出到合并阶段，然后直接显示在屏幕上。随着时间的推移，像素着色器可以执行的指令数量大大增加，这种增长催生了多重渲染目标（multiple render target，MRT）的想法。像素着色器并不会直接将生成的结果输出到颜色缓冲和 z-buffer 中，而是会为每个片元生成多组数值，并存储到不同的缓冲区中，每个缓冲区被称为一个渲染目标（render target，RT）。渲染目标通常具有相同的 x, y 维度，有些 API 也允许其拥有不同的尺寸，但是渲染区域只会以其中最小的那个维度为准。某些架构会要求所有的渲染目标都具有相同的位深度，甚至要求必须是完全一样的数据格式。根据 GPU 的不同，能够使用的渲染目标的数量一般是 4 个或者 8 个。

即使有着种种限制，但是 MRT 仍然是一种强大的工具，可以用于高效执行各种算法。我们可以在一个 pass 中进行如下操作：在第一个渲染目标中生成颜色图像，在第二个渲染目标中生成对象标识符，并在第三个渲染目标中生成世界空间距离。MRT 的这种能力催生了一种不同类型的渲染管线，它被称作延迟着色（deferred shading），在延迟着色中，可见性计算和着色计算是在两个单独的 pass 中完成的。第一个 pass 计算并存储了每个像素上的物体位置和材质信息，并在之后的 pass 中来高效计算光照以及其他效果。我们将在章节 20.1 中详细讨论这类渲染方法。

像素着色器也有一些限制，通常它只能在输入片元的对应位置上来写入渲染目标，它无法读取相邻像素上的计算结果。也就是说，当执行一个像素着色器程序的时候，它既不能将自己的结果输出到相邻像素上，也无法访问相邻像素当前的变化；相反，它的计算结果只会对其自身的像素产生影响。但是，像素着色器的这种限制并不像它听起来这么严重。我们在一个 pass 中输出图像之后，可以在之后 pass 的像素着色器中，来访问图像中的任何数据；相邻像素也可以使用一些图像处理技术来进行处理，详见章节 12.1。

刚才我们提到，像素着色器无法得知或者影响相邻像素的计算结果，但是这个规则也有一些例外的情况。其中一个情况是，像素着色器可以在计算梯度或者导数的时候，获取相邻片元的信息。像素着色器提供了任何插值数据在 x, y 方向上每个像素的变化量，这些数据对于各种计算和纹理寻址而言都十分有用，其中的梯度信息对于纹理过滤（texture filtering）来说尤其重要（章节 6.2.2）。在纹理过滤中，我们需要知道一个像素到底覆盖了纹理图像上的多大面积。所有的现代 GPU 都通过处理 2×2 的片元组（被称作一个 quad，四边形）来实现这个特性。当像素着色器需要获取一个梯度数据的时候，便会返回相邻片元之间的差异信息，如图 3.15 所示。统一的处理器核心具备访问相邻数据的能力，这些数据被保存在同一个 warp 上的不同线程中，因此可以计算出梯度信息并在像素着色器中进行使用。这种实现的一个后果是，在受到动态流程控制影响的着色器部分，将无法访问梯度信息，即“if”语句或者是包含一定迭代次数的循环语句。同一组中的所有片元，都必须使用同一组指令来进行处理，以便所有四个像素的结果对于计算梯度都具有意义。这是一个基础性的限制，它甚至存在于离线渲染系统中[64]。

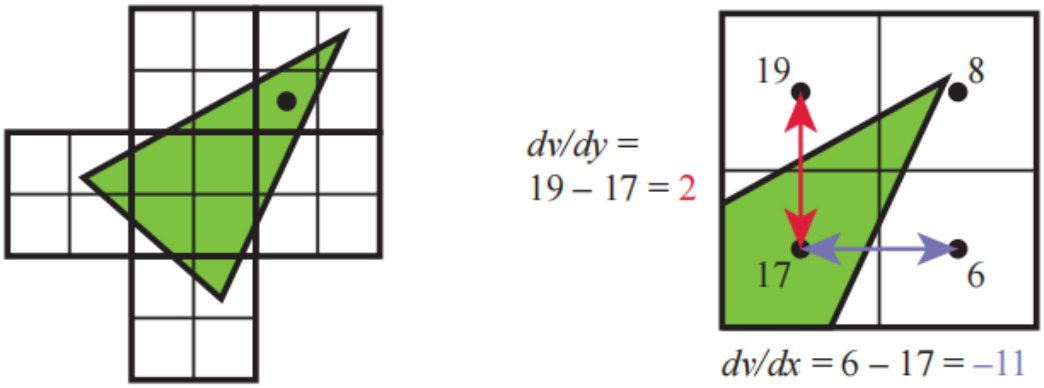


图 3.15：左图中，一个三角形被光栅化为了好几个 2×2 的 quad，其中有一个使用黑点进行标记的像素。右图展示了其梯度计算的细节，图中展示了这个 quad 中每个像素的 v 值。请注意，这个 quad 中有三个像素并没有被三角形所覆盖，但是它们仍然会被 GPU 处理，以便能够计算出所需要的梯度信息。右图使用了 quad 中的两个相邻像素，来计算左下角像素的 x 方向上和 y 方向上的梯度。

DirectX 11 引入了一种缓冲类型，它叫做无序访问视图（unordered access view，UAV），这个缓冲允许在其任何位置上进行写入。最开始的时候，这个缓冲只允许像素着色器和计算着色器进行写入，在 DirectX 11.1 的时候，这个缓冲的写入权限被扩展到了所有的着色器[146]。在 OpenGL 4.3 中这个缓冲叫做着色器存储缓冲区对象（shader storage buffer object，SSBO），这两个名字的描述都有一定的道理。像

素着色器可以以任意的顺序并行化运行，这个存储缓冲区会被所有的像素着色器共享。

通常来说，当两个着色器程序在“相互竞争”的影响同一个值时，可能会导致这个值产生一个无法预期的结果，这时候就需要某种机制来避免数据竞争问题（data race condition，简称数据冲突，data hazard）。例如：如果一个像素着色器的两次调用试图在同一时间读取并修改同一个检索值，那么可能就会引发错误，两次调用都会检索到完全相同的原始数据，并在本地对其进行修改，然后再将其写回原地址，但是后一次的写入会覆盖掉前一次的写入，也就是说只有一次着色器的调用能够对这个数值进行修改。GPU 通过着色器专用的原子操作，来避免这个问题[530]。但是原子操作意味着，当某个内存位置正在被另一个着色器读写时，另一个等待访问的着色器此时将会停滞下来。

虽然原子操作避免了数据冲突问题，但是很多算法实际上都需要一个特定的执行顺序。例如：在绘制一个红色的透明三角形之前，我们可能还会想要在一个更远的地方，绘制一个蓝色的透明三角形，然后再将蓝色与红色混合。我们需要在一个像素上调用两个不同的像素着色器（每个三角形一个），正确的执行顺序是：蓝色三角形的着色器先执行，然后再执行红色三角形的着色器。在一个标准的渲染管线中，片元处理的结果会在合并阶段之前进行排序。在 DirectX 11.3 中引入了光栅器有序视图（rasterizer order view，ROV），它强制保证了执行的顺序。ROV 和 UAV 相类似，它们都以相同的方式进行读写，它们之间最主要的区别在于，ROV 保证会以特定的顺序来访问数据，这大大增加了这些缓冲区（着色器可以进行访问）的实用性。例如：ROV 可以使得像素着色器来编写自己的混合方法，而不需要通过合并阶段来完成[176]，因为它可以直接访问并写入 ROV 中的任何位置。它的代价就是，如果检测到了一个无序访问，那么像素着色器的调用就会停滞下来，直到在它之前绘制的三角形被处理完成为止。

3.9 合并阶段

正如章节 2.5.2 中我们所讨论的，在合并阶段中，我们会将每个独立片元的颜色和深度进行组合，并最终形成帧缓冲。DirectX 将这个阶段叫做输出合并（output merger）；OpenGL 将其称为逐样本操作（per-sample operation）。在大多数传统渲染管线的示意图中（包括本书中的），模板缓冲和深度缓冲的相关操作都将在这个阶段执行。如果一个片元是可见的，那么这个阶段将会发生的另一个操作是颜色混合（color blending）。当然，对于一个不透明表面而言，并没有发生真正意义上的

颜色混合，仅仅是将之前存储的颜色替换成了当前片元的颜色而已，真正的片元混合和颜色存储，通常会发生在透明度和合成操作（章节 5.5）。

想象现在我们有一个片元，它在光栅化阶段生成，并通过了像素着色器。然后在应用 z-buffer 的时候我们发现，它被之前已经渲染过的某些片元所遮挡，这个片元最终并不会显示在屏幕上。也就是说，我们在像素着色器中对这个片元所进行的处理和计算都是无意义的。为了避免这种性能浪费，许多 GPU 都会在执行像素着色器之前，进行一些合并测试[530]。片元的深度值（以及其他任何可以使用的内容，例如模板缓冲或者裁剪测试，即 scissor）可以用于对可见性进行测试，不可见的片元将会被直接剔除，这个功能被称作为 early-z [1220, 1542]。像素着色器可以修改片元的深度值，或者是直接将整个片元丢弃；但是如果在像素着色器中存在这种类型操作的话，那么通常将无法使用 early-z，这会降低整个管线的效率。DirectX 11 和 OpenGL 4.2 可以允许像素着色器强制开启 early-z，尽管会存在一些限制[530]，在章节 23.7 中，我们可以看到更多有关 early-z 和其他 z-buffer 的优化技术。使用 early-z 可以大大提升渲染管线的性能表现，详见章节 18.4.5。

合并阶段位于固定功能阶段（例如三角形设置）和完全可编程着色器阶段的中间地带。尽管合并阶段并不是完全可编程的，但是它是高度可配置的。特别是颜色混合，可以为其设置大量不同的操作，最常见的颜色混合方式涉及颜色和透明度的乘法、以及加减操作等，也有一些其他的操作，例如最小值、最大值和位逻辑操作等。DirectX 10 中添加了双色源混合（dual source-color blending），它允许将像素着色器中的两种颜色与帧缓冲颜色相混合，但是它无法和多重渲染目标一起使用。MRT 是支持混合操作的，DirectX 10.1 中引入了在每个单独缓冲区上执行不同混合操作的功能。

在上一小节的末尾我们提到，DirectX 11.3 提供了一种方法，可以通过 ROV 来实现可编程的混合操作，其代价是牺牲一部分性能。ROV 与合并阶段都可以保证绘制的顺序，即输出不变性（output invariance）。无论像素着色器生成的结果顺序如何，API 都要求其结果需要按照输入的顺序（逐对象、逐三角形）进行排序，然后再输入到合并阶段。

3.10 计算着色器

GPU 不仅可以用来实现传统的图形渲染管线，还可以用于很多非图形的领域，例如用于计算股票期权的估计价值，以及用于训练深度学习的神经网络等，这种使用硬件的方式叫做 GPU 计算（GPU computing）。诸如 CUDA 和 OpenCL 等平台，会将 GPU 视为一个巨大的并行处理器来进行控制，而不需要使用 GPU 专门用于图形渲染

的功能，这类框架通常使用带有扩展的 C 或者 C++ 来实现的，以及专门为 GPU 创建的库等。

DirectX 11 引入了计算着色器（compute shader），它是利用 GPU 进行计算的一种方式。计算着色器是一种特殊的着色器，但是它并没有锁定在图形管线中的固定位置。它与渲染的过程密切相关，因为它是通过图形 API 来进行调用的。计算着色器与顶点着色器、像素着色器以及其他着色器可以一起进行使用，它利用了管线中相同的统一着色器处理器池。与其他着色器一样，它也有一组输入数据，并且也可以访问输入缓冲区和输出缓冲区（例如纹理）。warp 和线程的概念在计算着色器中会更加明显，例如：每个计算着色器的调用，都会获得一个可以访问的线程 ID。DirectX 11 中还有一个线程组（thread group）的概念，每个线程组包含 1–1024 个线程。这些线程组可以通过使用 x, y, z 坐标进行指定，这样设计的主要目的是便于在着色器代码中进行调用。在每个线程组中，线程之间会共享一小部分内存，在 DirectX 11 中是 32KB。计算着色器是由线程组进行执行的，也就是说一个线程组中的所有线程，都是同步执行的[1971]。

计算着色器的其中一个优势在于，它可以访问在 GPU 上生成的数据。由于在 GPU 和 CPU 之间进行通讯是一件效率很低的事情，因此如果我们能够将数据驻留在 GPU 上，并在 GPU 上进行计算，那么就可以大幅提高性能表现[1403]。计算着色器的一个普遍用途就是后处理计算，即以某种方式来对图像进行修改。线程之间共享内存，这意味着某些图像采样的中间结果可以与相邻线程进行共享。例如：使用计算着色器来确定图像的分布或者平均亮度，我们可以发现其运行效率大约是在像素着色器上进行处理的两倍[530]。

计算着色器对于实现许多功能都非常有用，例如：粒子系统、网格处理（例如面部动画[134]）、剔除[1883, 1884]、图像过滤[1102, 1710]、改进深度精度[991]、阴影[865]、景深效果[764]、以及其他可以使用 GPU 处理器来完成的任务等。Wihlidal [1884]讨论了计算着色器是如何比曲面细分中的壳着色器更加高效的，图 3.16 展示了计算着色器的其他一些用途。



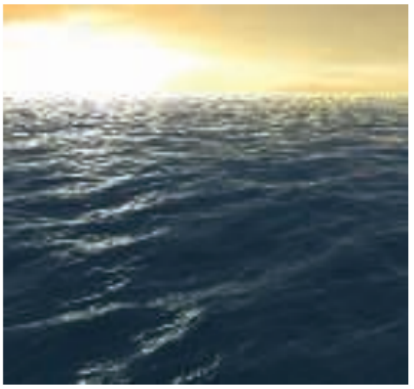


图 3.16：使用计算着色器的例子。左图使用计算着色器来模拟被风吹动的头发，而头发本身的渲染则是使用曲面细分来完成的。中间的图片使用计算着色器来快速完成模糊效果。右图使用计算着色器来模拟海面上的波浪。[1301]

到此为止，我们回顾了渲染管线的 GPU 实现。还有很多其他的方式可以使用和组合 GPU 功能，从而执行各种与渲染相关的过程，为了利用这些功能而修改调整的相关理论和算法，是本书的中心主题。下一章我们将重点关注变换和着色。

补充阅读和资源

Giesen 的图形管线之旅[530]详细讨论了 GPU 的许多方面，解释了 GPU 组成部分的工作原理。Fatahalian 和 Bryant 的课程[462]通过一系列的详细讲座，讨论了 GPU 的并行特性。在关注使用 CUDA 进行 GPU 计算的同时，Kirk 和 Hwa 书籍的导论部分[903]讨论了 GPU 的演变和设计哲学。

在正式学习着色器编程之前，还有一些其他的工作要做。例如 OpenGL Superbible [1606]和 OpenGL Programming Guide [885]这样的书中包含了大量有关着色器编程的资料。OpenGL Shading Language 是一本有点老旧的书[1512]，它专注于与着色器相关的算法，但是并没有涵盖最新的几何着色器和曲面细分着色器。在本书的网站上 realtimerendering.com，你可以找到最新的推荐书籍。

Chapter 4 Transform 变换

> Robert Penn Warren——
> “What if angry vectors veer
> Round your sleeping head, and form.
> There's never need to fear
> Violence of the poor world's abstract storm.”

> 罗伯特·佩恩·沃伦——
> “要是愤怒的航船改变了方向
> 围绕着你沉睡的脑袋，和身体
> 那就永远不必去害怕
> 穷苦世界抽象风暴之暴行”
> （美国第一任桂冠诗人；1905—1989）

变换（transform）是指以点、向量、颜色等实体作为输入，并以某种方式对其进行转换的一种操作。对于计算机图形学从业者而言，熟练掌握变换相关的知识是非常重要的。通过各种变换操作，我们可以对物体、光源和相机进行移动、变形以及设定动画；我们还可以确保所有的计算都在同一个坐标系下进行，以及使用不同的方式来将物体投影到一个平面上。这里我们只列举了变换所能完成的部分操作，但是足以证明变换在实时图形学中的重要性，或者可以说，在任何领域图形学中的重要性。

线性变换（linear transform）是指一种仅保留向量加法和标量乘法的变换，具体来说就是：

$$\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{y}) = \mathbf{f}(\mathbf{x} + \mathbf{y}) \tag{4.1}$$

$$k\mathbf{f}(\mathbf{x}) = \mathbf{f}(k\mathbf{x}) \tag{4.2}$$

例如：现在有一个变换 $\mathbf{f}(\mathbf{x}) = 5\mathbf{x}$，它代表将输入向量 $\mathbf{x}$ 的每个分量都乘以 5。为了证明这个变换是线性的，它必须满足上述两个条件（即方程 4.1 和方程 4.2）：第一个条件是成立的，因为任意两个向量先乘以 5 再相加，等同于两个向量相加再乘以 5；第二个标量乘法的条件也是显然成立的（方程 4.2）。这个函数叫做一个缩放

变换（scaling transform），因为它改变了一个物体的大小（尺寸）。旋转变换（rotation transform）是另一种线性变换，它将一个向量以原点为中心进行旋转。缩放变换以及旋转变换，以及事实上所有应用于三维向量的线性变换，都可以使用一个 3×3 矩阵来进行表示；但是， 3×3 的矩阵尺寸通常是不够的。

函数 $\mathbf{f}(\mathbf{x}) = \mathbf{x} + (7, 3, 2)$ 是一个非线性变换，其中的向量 $\mathbf{x}$ 是一个三维向量。对两个不同的向量分别执行这个函数，其结果为每个向量各自加上 $(7, 3, 2)$ 。让一个向量加上另一个固定的向量，意味着完成了一次平移变换（translation），即它以相同的程度对所有输入向量进行了移动，这是一种非常有用的变换类型。我们还可以将各种各样的变换组合在一起，例如：我们可以先将一个物体缩放为原来的一半，然后再将其移动到一个不同的位置上。到目前为止，我们使用这些变换的方式还很简单，而且我们很难以这种方式（ 3×3 矩阵）来将各种变换组合在一起。

我们可以使用仿射变换（affine transform）的形式来将线性变换和平移变换组合在一起，仿射变换通常存储在一个 4×4 的矩阵中。仿射变换是指先进行一次线性变换，然后再进行一次平移操作的变换。我们使用齐次符号（homogeneous notation）来表示这样的四维向量，点和方向也会使用同样的方式来进行表示（小写的粗体字母），二者之间的区别是：代表方向的向量，其 w 分量为 0；代表点的向量，其 w 分量为 1。例如：方向向量可以表示为 $\mathbf{v} = \begin{pmatrix} v_x & v_y & v_z & 0 \end{pmatrix}^T$ ，点可以表示为 $\mathbf{v} = \begin{pmatrix} v_x & v_y & v_z & 1 \end{pmatrix}^T$ 。在本章节中，我们将大量使用线性代数中的术语和操作，你可以在 realtimerendering.com 中找到有关线性代数的附录。

实时渲染中所使用的平移、旋转、缩放、对称和剪切矩阵都是仿射类型的。仿射变换最主要的特征就是它保证了直线的平行性（即两个平行的直线在变换之后仍然是平行的），但是其长度和角度可能会发生一些变化。一个仿射变换也可以表示为一系列独立仿射变换的组合。

本章将从最基本的仿射变换开始说起，章节 4.1 可以看成是这些简单变换的“参考手册”，我们会在接下来的章节中描述更加复杂的矩阵；然后会对四元数（quaternion）进行讨论和介绍，四元数是一个非常强大的变换工具；再接下来，我们会讨论有关顶点混合和变形的内容，这是两种用于描述网格动画的方式，它们简单且有效；最后我们会介绍投影变换矩阵。表 4.1 总结了本章节中所涉及大部分变换的符号、函数和属性；这里还需要说明一点的是，正交矩阵的逆矩阵（inverse）就是它的转置矩阵（tanspose）。

符号	名称	特征描述
$\mathbf{T}(\mathbf{t})$	平移矩阵	移动一个点；仿射变

$\mathbf{R}_x(\rho)$	旋转矩阵	绕 x 轴旋转 ρ 弧度；绕 y, z 轴 是类似的；正交矩阵 & 仿
$\mathbf{R}$	旋转矩阵	任意的旋转矩阵；正交矩阵 &
$\mathbf{S}(\mathbf{s})$	缩放矩阵	根据传入的向量 $\mathbf{s}$，沿 x, y, z 放；仿射变换
$\mathbf{H}_{ij}(s)$	剪切变换	将分量 i 相对于分量 j 进行剪切 $\{x, y, z\}$；仿射变
$\mathbf{E}(h, p, r)$	欧拉变换	根据给定的欧拉角（yaw，pitc 矩阵的朝向；正交矩阵 &
$\mathbf{P}_o(s)$	正交投影矩阵	平行投影到一个平面或者是体积
$\mathbf{P}_p(s)$	透视投影矩阵	透视投影到一个平面或者
$\text{slerp}(\hat{\mathbf{p}}, \hat{\mathbf{r}}, t)$	球面插值	根据参数 t，在四元数 $\hat{\mathbf{p}}, \hat{\mathbf{r}}$ 中 四元数

表 4.1：对本章中所讨论的大部分变换的总结。

这些变换是操控几何物体的基本工具，大部分图形应用程序的接口都允许开发者对这些矩阵进行自定义。有时候还有一些库会对本章节中所讨论的矩阵操作进行封装，以便开发者进行调用；但是，深入理解这些函数调用背后的矩阵操作和变换细节是非常有价值的。了解矩阵在各种函数调用之后做了些什么仅仅是一个开始，深入理解这些矩阵本身的属性，可以帮助我们在图形学上走得更远。例如：当我们在处理一个正交矩阵的时候，这些对矩阵的知识让我们知道，正交矩阵的逆矩阵实际上就是它的转置矩阵，我们可以通过获取其转置矩阵的方式，来快速求解正交矩阵的逆矩阵。类似这样的知识可以帮助我们写出速度更快的代码。

4.1 基本变换

本小节会介绍最基本的变换操作，例如平移、旋转、缩放、剪切、变换组合、刚体变换、法线变换以及计算逆矩阵等。对于有经验的读者而言，本小节可以看作是这些简单变换的参考手册；而对于初学者而言，本小节可以作为对矩阵变换的介绍与入门。对于本章剩余部分以及本书其他章节而言，本小节的内容是必要的背景和前置知识。这里我们将从最简单的平移变换开始讲起。

4.1.1 平移

我们可以使用一个平移（translation）矩阵 $\mathbf{T}$ 来描述从一个位置到另一个位置的变化，这个矩阵需要输入一个向量 $\mathbf{t}=(t_x,t_y,t_z)$ 来作为参数，从而对物体进行变换。矩阵 $\mathbf{T}$ 的形式如下：

$$\mathbf{T}(\mathbf{t})=\mathbf{T}(t_x,t_y,t_z)=\begin{pmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{pmatrix} \tag{4.3}$$

图 4.1 展示了平移变换的一个具体效果。我们可以很容易的看出，点 $\mathbf{p}=(p_x,p_y,p_z,1)$ 在经过 $\mathbf{T}(\mathbf{t})$ 变换之后，生成了一个新的顶点 $\mathbf{p}'=(p_x+t_x,p_y+t_y,p_z+t_z,1)$，很显然这是一个平移变换。请注意，一个向量 $\mathbf{v}=(v_x,v_y,v_z,0)$ 和矩阵 $\mathbf{T}$ 相乘之后，是不会受到影响的，因为一个方向向量是无法被平移的；相比之下，点和向量都会受到其他仿射变换的影响。平移矩阵的逆矩阵 $\mathbf{T}^{-1}(\mathbf{t})=\mathbf{T}(-\mathbf{t})$，即向量 $\mathbf{t}$ 取反即可；通俗的来理解就是，平移矩阵的逆矩阵就是反向平移相同距离的矩阵。

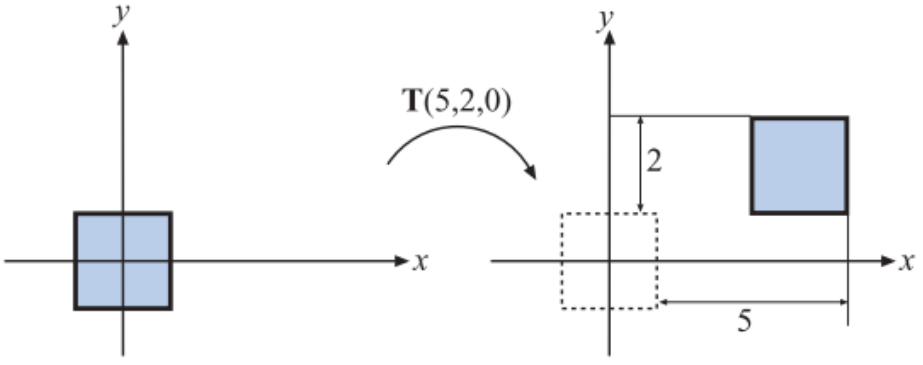


图 4.1：左图中的正方形使用了一个矩阵 $\mathbf{T}$（5，2，0） 来进行平移变换，其中正方形向右移动了 5 个单位，向上移动了 2 个单位。

这里我们需要提及一下有关矩阵表示的问题。有时候我们会在计算机图形学中看到另一种有效的矩阵表示方法，这种方法会将平移向量放在变换矩阵的最下面一行（即第四行），DirectX 中就使用了这种表示方式。在这种表示方式中，矩阵元素的顺序会被颠倒，即应用程序会以从左往右的方式来读取平移变量；这种向量和矩阵的表示方法被称作行优先（row-major）表示法（也叫做行主序）。而本书中将会采用列优先（column-major）表示法（也叫做列主序），使用哪一种表示方法仅仅是符号上的区别。当我们采用行主序表示法，那么变换矩阵在内存中进行存储时，代表位移的四

个分量会位于所有 16 个分量的最后四位，其中前三位代表了具体的位移分量，最后一位是 1。

4.1.2 旋转

旋转（rotation）变换可以将一个向量（位置或者方向），绕着一个过原点的旋转轴旋转一定的角度。与平移变换一样，旋转变换也是一个刚体变换（rigid-body transform），也就是说，被变换点之间的距离并不会发生改变，并且保持了手性（handedness），即它不会导致物体左右两边相互交换；这两个特性对于计算机图形学中调整物体位置和朝向而言非常有用。方向矩阵（orientation matrix）是一个与相机视角和物体朝向相关的旋转矩阵，它定义了物体在空间中的朝向，即物体向上和向前的方向。

我们可以很容易地推导出二维空间中的旋转矩阵。假设我们现在有一个向量 $\mathbf{v} = (v_x, v_y)$，我们可以将其参数化表示为 $\mathbf{v} = (v_x, v_y) = (r\cos\theta, r\sin\theta)$。如果我们将向量 $\mathbf{v}$ 顺时针旋转 ϕ 度，那么可以获得一个新向量 $\mathbf{v} = (r\cos(\theta + \phi), r\sin(\theta + \phi))$，这个变换过程可以写成如下形式：

$$
\begin{aligned}
\mathbf{v} &= \begin{pmatrix} r\cos(\theta+\phi) \\ r\sin(\theta+\phi) \end{pmatrix} = \begin{pmatrix} r(\cos\theta\cos\phi - \sin\theta\sin\phi) \\ r(\sin\theta\cos\phi - \cos\theta\sin\phi) \end{pmatrix} \\
&= \underbrace{\begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}}_{\mathbf{R}(\phi)} \underbrace{\begin{pmatrix} r\cos\theta \\ r\sin\theta \end{pmatrix}}_{\mathbf{v}} = \mathbf{R}(\phi)\mathbf{v}
\end{aligned} \tag{4.4}
$$

在方程 4.4 的推导过程中，我们使用了三角函数的二角和差公式来将 $\cos(\theta + \phi), \sin(\theta + \phi)$ 进行展开。在三维空间中，我们经常使用的旋转矩阵是 $\mathbf{R}_x(\phi)$，$\mathbf{R}_y(\phi)$，$\mathbf{R}_z(\phi)$，它们分别代表了绕 x, y, z 轴旋转 ϕ 度，它们的具体形式如下：

$$
\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.5}
$$

$$
\mathbf{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.6}
$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.7}$$

如果我们将这个 4×4 矩阵的第四行和第四列删除，那么就可以获得一个 3×3 矩阵。对于一个绕坐标轴旋转 ϕ 度的 3×3 旋转矩阵而言，矩阵的迹（trace：矩阵主对角线上的元素之和，等于矩阵的特征值之和）是一个与坐标轴无关的常量，其计算公式为[997]：

$$\text{tr}(\mathbf{R}) = 1 + 2\cos\phi \tag{4.8}$$

图 4.4 展示了旋转矩阵的具体作用效果。除了使得物体绕 i 轴旋转 ϕ 度之外，旋转矩阵 $\mathbf{R}_i(\phi)$ 的另外一个特征是，它会使得所有位于 i 轴上的点保持不变。请注意，矩阵 $\mathbf{R}$ 也可以用来表示绕任意轴进行旋转的旋转矩阵，我们可以通过对上面提到的三个绕坐标轴旋转的旋转矩阵 $\mathbf{R}_x(\phi)$ ， $\mathbf{R}_y(\phi)$ ， $\mathbf{R}_z(\phi)$ 进行组合，来获得一个绕任意轴旋转的旋转矩阵。我们将在章节 4.2.1 来讨论这个组合过程，章节 4.2.4 也会提到有关直接绕任意轴进行旋转的方法。

所有旋转矩阵的行列式值都为 1，并且它们都是正交矩阵。这个特征对于任意数量旋转矩阵相乘的结果也同样成立。我们还可以通过 $\mathbf{R}_i^{-1}(\phi) = \mathbf{R}_i(-\phi)$ 的方式，来获得旋转矩阵的逆矩阵，也就是说旋转矩阵的逆矩阵，相当于以相反方向旋转同样的角度。

示例：绕某个点旋转

假设我们想让一个物体以点 $\mathbf{p}$ 为中心，绕 z 轴旋转 ϕ 度，那么其变换过程是什么样的呢？图 4.2 展示了这个旋转过程。绕某个点进行旋转的意思就是，这个点本身并不会被旋转影响，我们可以先将物体进行平移，使得点 $\mathbf{p}$ 和原点重合，这个过程可以通过平移矩阵 $\mathbf{T}(-\mathbf{p})$ 来完成；然后我们可以对物体进行想要的旋转操作，即 $\mathbf{R}_z(\phi)$ ；最后我们再使用反向平移矩阵 $\mathbf{T}(\mathbf{p})$ 来将这个物体平移回原来的位置。我们将这个三个变换组合在一起，即可获得总的变换矩阵 $\mathbf{X}$ ：

$$\mathbf{X} = \mathbf{T}(\mathbf{p})\mathbf{R}_z(\phi)\mathbf{T}(-\mathbf{p}) \tag{4.9}$$

这里请注意这三个变换的顺序，最先应用的变换矩阵位于方程的最右边，然后接下来的若干个变换依次左乘。

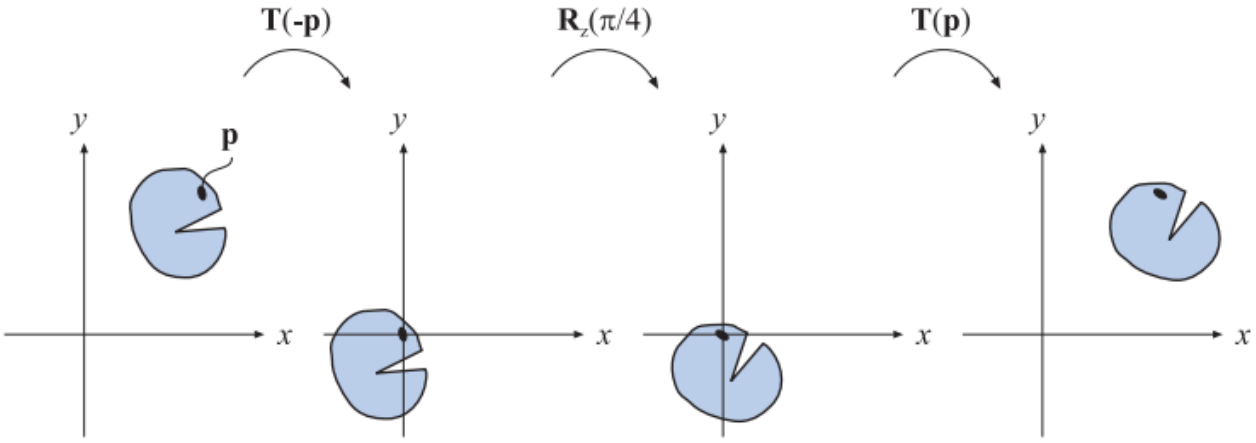


图 4.2：绕点 $\mathbf{p}$ 旋转的过程示意图。先将物体和点 $\mathbf{p}$ 平移到坐标原点，然后让物体绕原点旋转一定角度，最后再将物体和点 $\mathbf{p}$ 反向平移回最开始的位置。

4.1.3 缩放

缩放（scaling）矩阵 $\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z)$ 可以分别在 x, y, z 方向上，使用缩放因子 s_x, s_y, s_z 来对物体进行缩放，也就是说缩放矩阵可以放大或者缩小物体，缩放因子 $s_i, i \in \{x, y, z\}$ 的值越大，物体在该方向上的尺寸也就会变得越大；如果我们将缩放矩阵的其中一个分量设置为 1，那么物体在该方向上的尺寸将会保持不变。方程 4.10 展示了缩放矩阵 $\mathbf{S}$ 的具体形式：

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.10}$$

图 4.4 展示了缩放矩阵的具体效果。当 $s_x = s_y = s_z$ 的时候，这个缩放操作被称作均匀缩放（uniform），否则会被称作非均匀缩放（nonuniform）；有时候我们也会使用术语各向同性（isotropic）和各向异性（anisotropic）来代指均匀和不均匀。缩放矩阵的逆矩阵可以表示为 $\mathbf{S}^{-1}(\mathbf{s}) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$，即按照缩放比例进行反向缩放。

在使用齐次坐标来表示变换矩阵的时候，我们还可以通过直接操作位于矩阵 $(3, 3)$ 位置（矩阵最右下角的元素）上的元素，来创建一个均匀缩放变换，这个操作会改变齐次坐标的 w 分量，因此每个被缩放矩阵变换的点（这里只能是点，而不是方向向量，因为方向向量的 w 分量为 0），其坐标中的各个分量，都会按照 w 分量的倒数

进行缩放。例如：为了创建一个比例为 5 的均匀缩放，我们可以将缩放矩阵中 $(0,0),(1,1),(2,2)$ 这三个位置上的元素设置为 5；或者是直接将矩阵 $(3,3)$ 位置上的元素设置为 $1/5$ 。这两个均匀缩放矩阵的形式不同，但效果确是相同的，具体矩阵形式如下所示：

$$\mathbf{S}=\begin{pmatrix}5&0&0&0\\0&5&0&0\\0&0&5&0\\0&0&0&1\end{pmatrix},\mathbf{S}'=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1/5\end{pmatrix}. \tag{4.11}$$

相对于使用矩阵 $\mathbf{S}$ 来进行均匀缩放，我们在使用 $\mathbf{S}'$ 的时候必须遵守齐次坐标的规则。这个缩放矩阵的效率并不高，因为它在齐次坐标过程中引入了除法；即如果矩阵最右下角（位置 $(3,3)$ ）上的元素值为 1，则不需要进行除法操作。当然了，如果我们所编写的系统根本没有针对 1 进行测试和优化（即当点坐标的 w 分量为 1，则跳过齐次坐标除法），那么其实也就没有额外的开销。

如果向量 $\mathbf{s}$ 中包含 1 个或者 3 个为负的分量，那么我们就获得了一个反射矩阵（reflection matrix），或者叫做镜像矩阵（mirror matrix）；如果有两个为负的分量，那么这个矩阵会将物体旋转 180 度（中心对称）。这里需要注意是，如果一个旋转矩阵和一个反射矩阵相乘，那么生成的结果仍将是一个反射矩阵。例如下列两个矩阵相乘的结果仍然是一个反射矩阵：

$$\underbrace{\begin{pmatrix}\cos(\pi/2)&\sin(\pi/2)\\-\sin(\pi/2)&\cos(\pi/2)\end{pmatrix}}_{\text{rotation}}\underbrace{\begin{pmatrix}1&0\\0&-1\end{pmatrix}}_{\text{reflection}}=\begin{pmatrix}0&1\\-1&0\end{pmatrix} \tag{4.12}$$

通常在检测到一个镜像变换的时候，都会进行一些特殊处理。例如：一个顶点为逆时针顺序定义的三角形，在经过反射矩阵变换之后，其顶点顺序将会变成顺时针；顶点顺序的改变会导致错误的光照效果和背面剔除。我们可以通过计算左上角 3×3 行列式的值，来判断一个给定的缩放矩阵是否为一个反射矩阵。如果缩放矩阵的行列式为负数，则说明该矩阵是一个反射矩阵。例如：方程 4.12 中的行列式值为 $0\cdot0-(-1)\cdot(-1)=-1$ 。

示例：按任意方向进行缩放

缩放矩阵 $\mathbf{S}$ 只能沿坐标轴方向进行缩放，我们需要使用复合变换来实现在其他方向上的缩放操作。假设我们现在有一个给定的方向 $\mathbf{f}$ ，我们需要首先将这个方向分解为三

个标准正交向量 $\mathbf{f^x}, \mathbf{f^y}, \mathbf{f^z}$ 。然后构建旋转变换矩阵 $\mathbf{F}$ ，这个矩阵可以用于修改坐标系的基底：

$$\mathbf{F} = \begin{pmatrix} \mathbf{f^x} & \mathbf{f^y} & \mathbf{f^z} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.13}$$

这个变换的核心思想就是，将由 $\mathbf{f^x}, \mathbf{f^y}, \mathbf{f^z}$ 给出的坐标系与标准坐标轴重合，然后再使用标准的缩放矩阵对其进行缩放，最后再将坐标系反向变换回去即可。这个过程的第一步是乘以矩阵 $\mathbf{F}$ 的转置矩阵（也就是它的逆矩阵），然后再乘以缩放矩阵（左乘），最后再乘以矩阵 $\mathbf{F}$ 来将坐标系变换回去，这个过程的总变换如下所示：

$$\mathbf{X} = \mathbf{F}\mathbf{S}(\mathbf{s})\mathbf{F}^{\mathbf{T}} \tag{4.14}$$

4.14 剪切

剪切（shear）变换是另一种基本变换，它可以用来对整个场景进行扭曲，从而创造出一种迷幻的效果，或者是对单个模型的外观进行扭曲。剪切变换一共包含 6 个基本矩阵，它们分别是 $\mathbf{H}_{xy}(s)$ ， $\mathbf{H}_{xz}(s)$ ， $\mathbf{H}_{yx}(s)$ ， $\mathbf{H}_{yz}(s)$ ， $\mathbf{H}_{zx}(s)$ ， $\mathbf{H}_{zy}(s)$ 。其中第一个下标用于表示哪个坐标会被剪切矩阵改变，第二个下标表示将会使用哪个坐标来进行剪切。方程 4.15 展示了剪切矩阵的一个例子 $\mathbf{H}_{xz}(s)$ ，从中我们可以发现，剪切矩阵的两个下标可以用来找到参数 s 在矩阵中的位置：下标中的 x （其索引为 0）代表了第 0 行，下标中的 z （其索引为 2）代表了第 2 列，因此我们可以知道参数 s 位于剪切矩阵的第 0 行，第 2 列，即位置 $(0, 2)$ 上：

$$\mathbf{H_{xz}}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.15}$$

将这个矩阵和一个顶点相乘，会生成一个新的顶点： $\begin{pmatrix} p_x + sp_z & p_y & p_z \end{pmatrix}^T$ ，图 4.3 生动的展示了一个单位正方形被剪切的过程。通过向相反的方向进行剪切，我们可以获得 $\mathbf{H}_{ij}(s)$ 的逆矩阵，即 $\mathbf{H}_{ij}^{-1}(s) = \mathbf{H}_{ij}(-s)$ 。

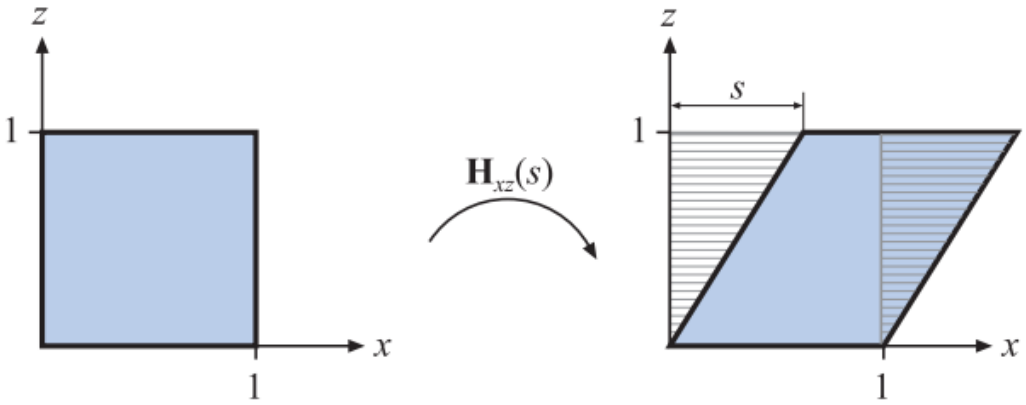


图 4.3：使用 $\mathbf{H}_{xz}(s)$ 对单位正方形进行剪切的过程。坐标中的 y 值和 z 值都不受这个变换的影响，而新的 x 值，由原来的 x 值加上 s 与 z 值的乘积组成，这个变换让原来的单位正方形变得向右倾斜。这个变换的过程是面积保持（area-preserving）的，即变换前后的区域面积不会发生改变，我们可以看到左右两个虚线区域的面积是一样。

你还可以使用一种稍微不同的剪切矩阵：

$$\mathbf{H}'_{\mathbf{xy}}(s,t) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.16}$$

这里的剪切变换包含两个输入参数，它们代表了这两个坐标（ x, y ）都会被第三个坐标（ z ）剪切。这个特殊的剪切矩阵可以通过上面一般的剪切矩阵组合而成，即 $\mathbf{H}'_{\mathbf{xy}}(s,t) = \mathbf{H}_{ik}(s)\mathbf{H}_{jk}(t)$ ，其中的 k 代表了 第三个坐标分量的索引。具体使用哪一种形式的剪切矩阵仅仅是一个偏好上的问题。最后我们需要注意的是，任何剪切矩阵的行列式值都为 $|\mathbf{H}| = 1$ ，这意味着剪切变换是一种体积保持（volume-preserving）的变换，即变换前后，物体的体积并不会发生改变（在三维空间中是体积，在二维空间中则是面积），图 4.3 中也展示了这个特性。

4.1.5 变换的连接

由于矩阵之间的乘法不具备交换律（noncommutativity），因此矩阵在乘法式子中的顺序十分重要。变换的连接是与顺序相关的。

我们举一个矩阵顺序相关的例子：假如我们现在有两个矩阵 $\mathbf{S}, \mathbf{R}$ ，其中矩阵 $\mathbf{S}(2, 0.5, 1)$ 是一个缩放变换，它将坐标的 x 分量缩放为原来的 2 倍，将 y 分量缩放为原来的 0.5 倍。 $\mathbf{R}_z(\pi/6)$ 是一个旋转矩阵，它绕 z 轴（在右手坐标系中，这里的

z 轴指向书页的外面）顺时针旋转了 $\pi/6$ 的角度。这两个矩阵可以用两种不同的方式相乘，它们变换的结果是完全不同的，如图 4.4 所示：

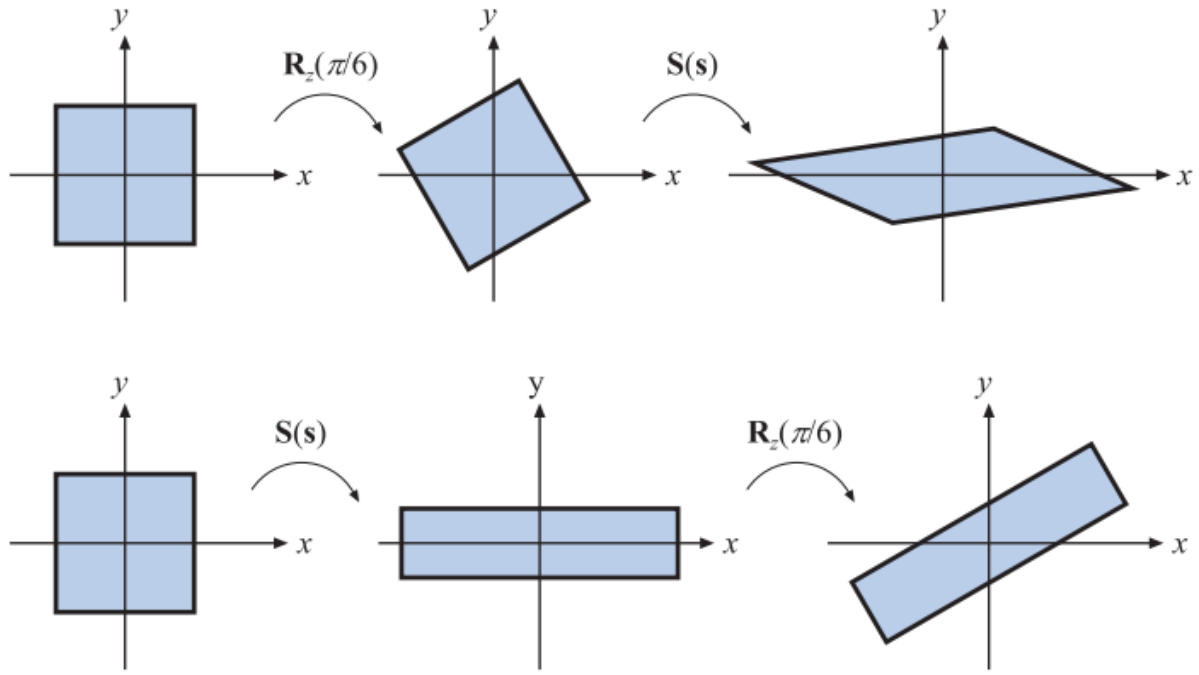


图 4.4：上图展示了矩阵相乘的顺序依赖性。在第一行中，先进行了旋转变换 $\mathbf{R}_z(\pi/6)$，然后再进行了缩放变换 $\mathbf{S}(\mathbf{s})$，其中 $\mathbf{s}=(2,0.5,1)$，总变换为 $\mathbf{S}(\mathbf{s})\mathbf{R}_z(\pi/6)$；在第二行中，两个向量交换了相乘的顺序，总变换为 $\mathbf{R}_z(\pi/6)\mathbf{S}(\mathbf{s})$，这两个组合变换的结果完全不同。对于任意的矩阵 $\mathbf{M},\mathbf{N}$ 而言，$\mathbf{MN}\neq\mathbf{NM}$，即矩阵的乘法不具备交换律。

将一系列矩阵组合成一个独立矩阵有一个好处，那就是可以获得更高的执行效率。例如：想象我们现在有一个包含几百万顶点的游戏场景，场景中的所有物体都需要进行缩放、旋转和平移变换。这里我们并不会将所有的顶点都与这三个变换矩阵挨个相乘，因为这样做的效率实在是太低了；我们会将这三个矩阵连接成一个独立的矩阵，然后对所有顶点都应用这个相同的组合变换矩阵。这个组合矩阵可以写作 $\mathbf{C}=\mathbf{TRS}$，请注意这里的矩阵顺序，缩放矩阵 $\mathbf{S}$ 应当首先作用于顶点，因此它出现在组合矩阵的最右侧。这个组合矩阵的顺序意味着 $\mathbf{TRSp}=(\mathbf{T}(\mathbf{R}(\mathbf{Sp})))$，其中 $\mathbf{p}$ 是待变换的顶点。顺便说一句，$\mathbf{TRS}$ 顺序是场景图系统中，最为常用的变换组合顺序。

值得注意的是，虽然矩阵连接的结果是与顺序相关的，但是矩阵和矩阵之间可以进行分组计算，也就是说，矩阵乘法是具有结合律的。例如：假设我们按照 $\mathbf{TRSp}$ 的顺序进行计算，并且在计算的过程中，我们还想顺便计算一下刚体变换 $\mathbf{TR}$ 的值；那么我们可以将这两个矩阵组合在一起进行计算，即 $(\mathbf{TR})(\mathbf{Sp})$，这样我们就可以先获得 $\mathbf{TR}$ 的值，然后再利用这个值去计算 $\mathbf{TRSp}$。

4.1.6 刚体变换

现在我们想象这样的一个过程，当一个人抓起一个固体物体（比如说从桌子上拿起一支笔），然后将其移动到另一个位置上（可能是衬衫口袋里），在这个过程中，仅仅是物体的位置和朝向发生了变化，其形状和大小并没有受到任何影响。我们将这样只包含平移和旋转的变换叫做刚体变换（rigid-body transform），它具有保持长度、角度和手性的特点。

任何刚体变换矩阵 $\mathbf{X}$，都可以写成一个平移矩阵 $\mathbf{T}(\mathbf{t})$ 和一个旋转矩阵 $\mathbf{R}$ 的连接，方程 4.17 展示了刚体变换矩阵 $\mathbf{X}$ 的一般形式：

$$\mathbf{X} = \mathbf{T}(\mathbf{t})\mathbf{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.17}$$

刚体变换矩阵 $\mathbf{X}$ 的逆矩阵可以这样来计算：

$$\mathbf{X}^{-1} = (\mathbf{T}(\mathbf{t})\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{T}(\mathbf{t})^{-1} = \mathbf{R}^{\mathbf{T}}\mathbf{T}(-\mathbf{t})$$

为了计算这个逆矩阵，旋转矩阵 $\mathbf{R}$ 左上角 3×3 的子矩阵会被转置（旋转矩阵的逆矩阵），平移矩阵 $\mathbf{T}$ 的参数符号会被取反；然后再将这两个新矩阵调换顺序相乘，便可以获得所需的逆矩阵。计算刚体变换矩阵 $\mathbf{X}$ 的逆矩阵，另一种方法是将矩阵 $\mathbf{R}, \mathbf{T}$ 按下列方式进行考虑（符号表示详见方程 1.2）：

$$\begin{gathered} \bar{\mathbf{R}} = \begin{pmatrix} \mathbf{r}_{,0} & \mathbf{r}_{,1} & \mathbf{r}_{,2} \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0^T, \\ \mathbf{r}_1^T, \\ \mathbf{r}_2^T, \end{pmatrix}, \\ \mathbf{X} = \begin{pmatrix} \bar{\mathbf{R}} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix}, \end{gathered} \tag{4.18}$$

其中 $\mathbf{r}_{,0}$ 代表了旋转矩阵中的第一列（即第一个逗号取值可以是 0–2，而第二个下标的值始终为 0），$\mathbf{r}_0^T$ 代表了旋转矩阵中的第一行。还需要注意的是，方程中的 $\mathbf{0}$ 代表一个 3×1 的零向量。在这种表示方式下，方程 4.19 给出了矩阵 $\mathbf{X}$ 的逆矩阵的计算过程：

$$\mathbf{X}^{-1} = \begin{pmatrix} \mathbf{r}_0 & \mathbf{r}_1 & \mathbf{r}_2 & -\bar{\mathbf{R}}^{\mathbf{T}}\mathbf{t} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.19}$$

示例：调整相机的朝向

在图形程序中有一个十分常见的操作，即调整相机的朝向，使其观察某一个指定的点。有一个叫做 gluLookAt() 的函数可以完成这个操作（来自 OpenGL 工具库，简称为 GLU）。这里我们将会具体展示其执行的流程，虽然这个函数现在已经很少用到了，但是这个函数所对应的操作仍然十分常见。假设我们现在有一个位于点 $\mathbf{c}$ 的相机，我们想让这个相机看向位于点 $\mathbf{l}$ 的物体，并且此时相机指向上方的方向为 $\mathbf{u}'$，如图 4.5 所示。

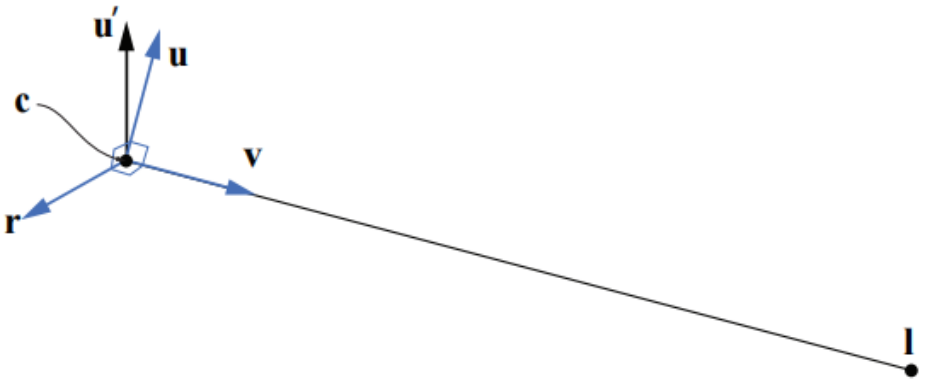


图 4.5：计算一个变换矩阵，使得位于点 $\mathbf{c}$ 的相机看向点 $\mathbf{l}$，并且向上的向量为 $\mathbf{u}'$。为了实现这个目的，我们需要计算 $\mathbf{r}, \mathbf{u}, \mathbf{v}$。

这里我们需要计算一个包含三个向量 $\{\mathbf{r}, \mathbf{u}, \mathbf{v}\}$ 的基底。我们首先计算观察向量 $\mathbf{v} = (\mathbf{l} - \mathbf{c})/\|\mathbf{l} - \mathbf{c}\|$，即从相机位置指向目标位置的单位向量；然后计算指向右方的向量 $\mathbf{r} = (\mathbf{v} \times \mathbf{u}')/\|\mathbf{v} \times \mathbf{u}'\|$；在相机调整朝向之后，向量 $\mathbf{u}'$ 通常并不会精确地指向相机向上的方向，因此最终指向相机上方的向量，是之前两个向量叉乘的结果，即 $\mathbf{u} = \mathbf{r} \times \mathbf{v}$，这个向量同时垂直于向量 $\mathbf{v}, \mathbf{r}$，并且由于这两个向量都是单位向量，因此计算出的向量 $\mathbf{u}$ 也是一个单位向量。接下来我们将构建相机的变换矩阵 $\mathbf{M}$，这里的核心思想是，先将相机平移到坐标原点 $(0, 0, 0)$ 的位置上；然后再对基底进行变换，使得向量 $\mathbf{r}$ 指向 $(1, 0, 0)$，向量 $\mathbf{u}$ 指向 $(0, 1, 0)$，向量 $\mathbf{v}$ 指向 $(0, 0, -1)$，这个过程可以通过如下变换完成：

$$
\begin{aligned}
\mathbf{M} &= \underbrace{\begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ -v_x & -v_y & -v_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{change of basis}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{translation}} \\
&= \begin{pmatrix} r_x & r_y & r_z & -\mathbf{t} \cdot \mathbf{r} \\ u_x & u_y & u_z & -\mathbf{t} \cdot \mathbf{u} \\ -v_x & -v_y & -v_z & -\mathbf{t} \cdot \mathbf{v} \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{4.20}
$$

请注意，这里我们将平移矩阵和基底变换矩阵组合在了一起，第一步进行的应当是平移变换，因此平移矩阵 $-\mathbf{t}$ 会位于总变换矩阵的右侧。有一种方法可以用来帮助记忆向量 $\mathbf{r}, \mathbf{u}, \mathbf{v}$ 在矩阵中的具体位置：我们想要让向量 $\mathbf{r}$ 和向量 $(1,0,0)$ 重合，所以当 $(1,0,0)$ 与基底变换矩阵相乘的时候，其结果矩阵中的第一行，一定是向量 $\mathbf{r}$ 中的元素；此外，矩阵中第二行和第三行的向量一定会垂直于 $\mathbf{r}$，即 $\mathbf{r} \cdot \mathbf{x} = 0$。同理，当我们使用同样的想法来思考向量 $\mathbf{u}, \mathbf{v}$ 的时候，便可以获得上述的基底变换矩阵。

4.1.7 法线变换

矩阵可以用于对点、线、三角形和其他几何物体进行变换，这些矩阵同样也可以对这些线或者三角形表面的切向量（tangent vector）进行变换，然而有一个重要的几何属性并不能总是使用这些矩阵直接进行变换，即表面法线（以及顶点的光照法线）。图 4.6 展示了如果使用同样的矩阵同时对几何物体及其法线进行变换的结果。

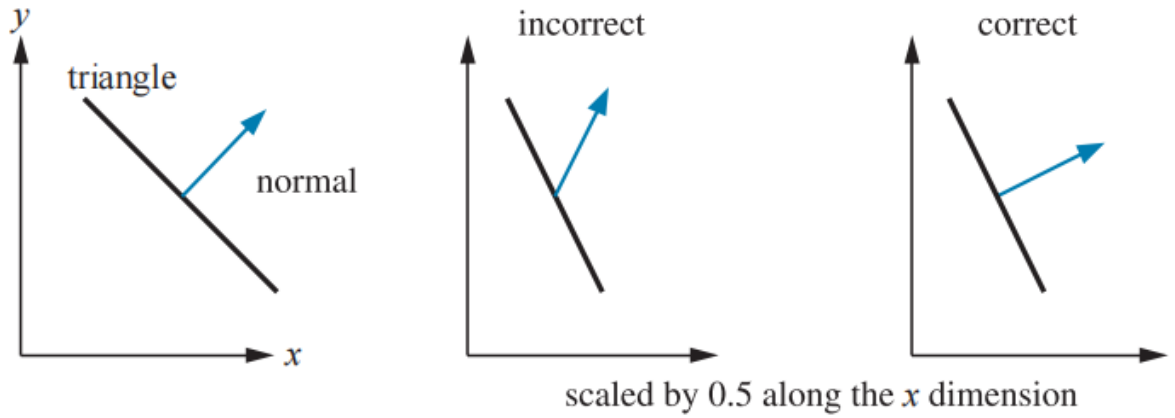


图 4.6：左侧是原始的几何体，它从侧面展示了一个三角形及其表面法线。在中间的图中，使用了一个缩放矩阵来将几何体在 x 轴方向上缩放为原来的 0.5 倍，并使用这个缩放矩阵对法线进行了变换，变换后的法线并不是这个三角形的表面法线。右图中则展示了法线正确变换后的结果。

对法线正确的变换方法是：使用原始变换矩阵的伴随矩阵（adjoint）的转置矩阵来对其进行变换，而不是使用原始变换矩阵本身[227]。你可以在我们网站上的线性代数附录中，找到有关计算伴随矩阵的相关内容，这里你需要知道的是：矩阵的伴随矩阵是始终存在的。法线在经过变换之后，其长度可能会发生变化，因此在变换后通常还需要对法线进行归一化处理。

法线变换的传统方法是，计算原始变换矩阵的逆矩阵的转置[1794]，即 $(\mathbf{M}^{-1})^{\mathbf{T}}$，这个方法现在也是可以使用的。但是我们并不需要完整求出这个逆矩阵，这样会做很浪费性能；并且有时候这个逆矩阵可能并不存在，因为逆矩阵是通过伴随矩阵除以矩阵的行列式获得的，但是矩阵的行列式可能会为 0。行列式为 0 的矩阵被称为奇异矩阵（singular matrix），其逆矩阵是不存在的。

进一步说，即使我们只计算一个 4×4 矩阵的伴随矩阵，也是十分耗时的，而且我们通常也不需要这样做。由于法线仅仅只是一个向量，因此平移变换对其是没有影响的；此外，大多数模型变换都是仿射变换，这些变换并不会修改齐次坐标的 w 分量，即它们没有执行投影操作。在这些条件下，我们只需要计算左上角 3×3 子矩阵的伴随矩阵，即可完成对法线的变换。

实际上，我们甚至都不需要计算这个伴随矩阵。假设我们已经知道了这个变换矩阵是完全由平移、旋转和均匀缩放（没有被拉伸或者压缩）操作组合而成的，那么首先平移变换并不会对法线产生影响，其次均匀缩放仅仅会对法线的长度产生影响，剩下的就是一系列旋转变换了，这些旋转变换最终会组合在一起，形成一个总的旋转变换矩阵。上文中我们说到，我们可以使用原始变换矩阵的逆转置矩阵，来对法线进行变换；而旋转矩阵本身是一个正交矩阵，其逆矩阵和转置矩阵是一样的，也就是说，我们对一个旋转矩阵进行两次转置（或者两次求逆）操作之后，得到的矩阵就是最初的旋转矩阵本身。总而言之，在这样的条件下（进行了平移、旋转和均匀缩放操作），我们可以直接使用模型的变换矩阵来对法线进行变换；但是如果模型变换涉及了非均匀缩放或者投影操作，那么就无法使用这种方法对法线进行变换了。

最后我们需要考虑是否要对法线进行归一化处理，如果模型变换只涉及平移或者旋转的话，那么法线的长度是不会发生改变的，因此也就不需要进行归一化处理。但是如果变换中涉及了非均匀缩放的话，那么这个均匀缩放系数（如果是已知的，或者是已经被提取出来的话，有关提取这个均匀缩放系数的话题会在章节 4.2.3 中进行讨论）也可以直接用来对法线进行归一化处理（直接按照均匀缩放系数进行反向缩放即可）。例如：如果我们知道了一系列的均匀变换最终会导致物体比之前放大 5.2 倍，并且法线应用了这个变换矩阵的话，我们可以直接将法线的长度除以 5.2 即可（即缩小为原来的 5.2 倍）。除此之外，我们还可以将这一步放在对法线变换之前，让左上

角 3×3 子矩阵除以这个缩放系数，从而构建一个会生成归一化结果的法线变换矩阵。

还有一点需要注意一下，如果表面法线是从变换之后的三角形中计算出来的话（例如使用三角形的边向量进行叉乘，从而获得垂直于三角形表面的法线），那么法线变换的问题就不需要进行考虑了。切向量的本质和法线并不相同，它可以直接使用原始变换矩阵进行变换。

4.1.8 计算逆矩阵

有很多计算都需要使用逆矩阵，例如在不同的坐标系间来回切换的时候。根据一个变换的可用信息不同，我们可以使用以下三种方法来计算一个逆矩阵：

- 如果某个矩阵是一次很简单的变换，或者是一系列带参简单变换的组合的话，那么我们可以通过反转“参数”和矩阵次序的方式，来获得这个矩阵的逆矩阵，即进行一系列的反向变换。例如：现在有一个变换矩阵 $\mathbf{M} = \mathbf{T}(\mathbf{t})\mathbf{R}(\phi)$，则其逆矩阵为 $\mathbf{M}^{-1} = \mathbf{R}(-\phi)\mathbf{T}(-\mathbf{t})$。这种求取逆矩阵的方式十分简单且准确，这对于渲染一个大世界而言是十分重要的[1381]。
- 如果一个矩阵是正交矩阵的话，那么其逆矩阵和转置矩阵是相等的，即 $\mathbf{M}^{-1} = \mathbf{M}^{\mathbf{T}}$。旋转矩阵是一个正交矩阵，并且任意数量的旋转矩阵组合在一起仍然是一个旋转矩阵，因此其结果也是正交的。
- 如果这些信息都不知道的话，那么我们还可以使用伴随矩阵法、Cramer 法则、LU 分解法、高斯消元法等方法来计算一个矩阵的逆矩阵。伴随矩阵法和 Cramer 法则通常要更好一些，因为它们所涉及的分支操作较少；在现代的 GPU 架构中，最好还是避免使用分支结构。在章节 4.1.7 中我们提到了如何使用伴随矩阵法来对法线进行正确的变换。

在进行性能优化的时候，我们还可以对逆矩阵的计算目的进行考虑。例如：如果这个逆矩阵仅仅是用来对向量进行变换的话，那么通常我们只需要获得左上角 3×3 子矩阵的逆矩阵即可（例如章节 4.1.7 中对法线的变换）。

4.2 特殊的矩阵变换和操作

在本小节中，我们会介绍并推导若干个对于实时渲染非常重要的矩阵变换和操作。首先我们会介绍欧拉变换（以及如何提取欧拉变换的参数），这是一种描述方向和旋转

的直观方法。然后我们会介绍如何在单个变换矩阵中，提取出一组基本变换（分解变换矩阵）。最后我们会推导一种绕任意轴旋转物体的方法。

4.2.1 欧拉变换

欧拉变换可以构建一个旋转矩阵，将我们自身（相机）或者其他物体指向一个特定的方向，这是一种十分直观的方式，其名字来源于伟大的瑞士数学家 Leonhard Euler (1707–1783)。

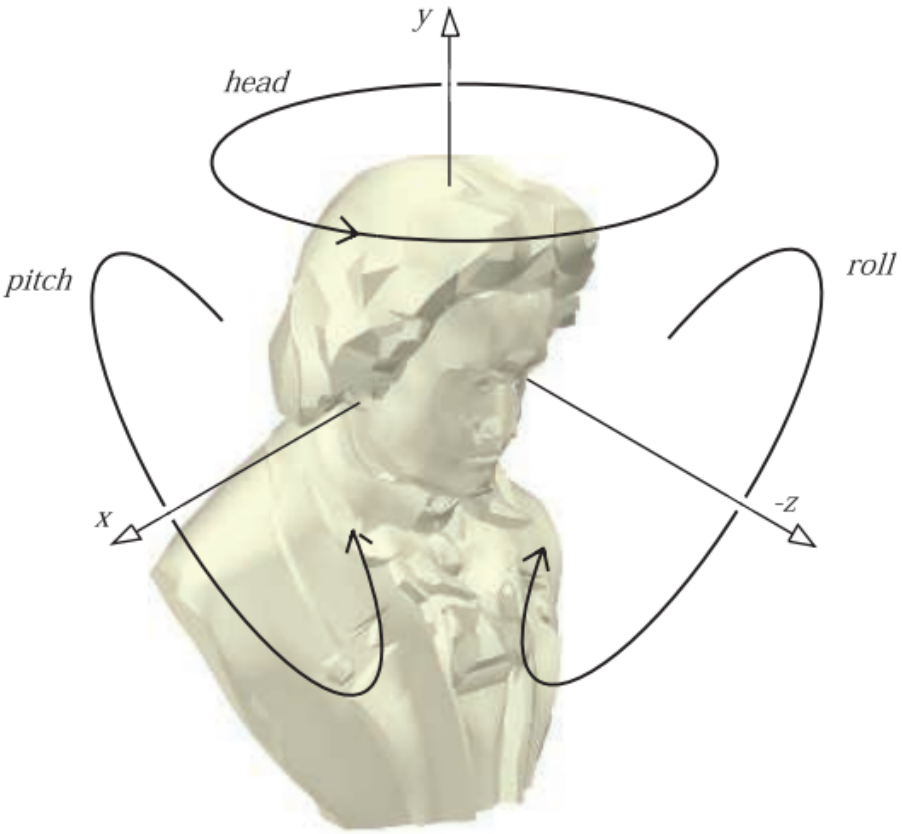


图 4.7：欧拉变换和改变头部角度（head）、俯仰角（pitch）以及滚转角（roll）之间的关系。相机的默认位置是看向 z 轴负半轴，同时 up 方向与 y 轴正半轴重合。

首先，我们需要有一个默认的观察方向，通常来说都会让这个方向指向 z 轴负半轴，并且头部指向 y 轴正半轴，如图 4.7 所示（其实就是在应用观察变换矩阵之后，相机的位置和朝向）。欧拉变换是三个旋转矩阵相乘的结果，如图 4.7 中所示的旋转。欧拉变换通常会使用 $\mathbf{E}$ 来进行表示，其具体形式如下：

$$\mathbf{E}(h,p,r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h) \tag{4.21}$$

在欧拉变换中，矩阵的组合方式一共有 24 种[1636]（6 个两轴旋转，6 个三轴旋转；以及内旋外旋两种方式。 $12 \times 2 = 24$ ），方程 4.21 所展示的组合方式，是图形学中最为常用的一种组合方式。由于矩阵 $\mathbf{E}$ 是由一系列旋转矩阵连接组成的，那么矩阵 $\mathbf{E}$ 本身自然也是一个正交矩阵，因此该矩阵的逆矩阵可以表示为 $\mathbf{E}^{-1} = \mathbf{E}^{\mathbf{T}} = (\mathbf{R}_z^T \mathbf{R}_x^T \mathbf{R}_y^T) = \mathbf{R}_y^T \mathbf{R}_x^T \mathbf{R}_z^T$ ，当然，一般直接使用 $\mathbf{E}^T$ 会更加方便。

其中的欧拉角参数 h, p, r 代表了每个方向（head 头部，pitch 俯仰角，roll 滚转角）上绕轴旋转的角度。有时候我们会将这三个旋转角度都叫做 roll，例如头部角度叫做 y–roll，俯仰角叫做 x–roll 等。这几个角度的名称，在不同学科领域中的称呼不太一样，例如在飞行模拟中，头部角度 head 通常被叫做偏航角（yaw）。

这种变换的方式是非常直观的，因此不使用专业术语也可以很形象对其进行描述。例如：改变 head 角度会让观察者摇头；改变俯仰角会让观察者点头；改变滚转角会让观察者倾斜头部。这里我们使用了 head，pitch，roll 来描述旋转的方向，而不是使用绕 x, y, z 轴。这里需要注意的是，欧拉变换不仅仅可以用来调整相机的方向，还可以用来调整任意物体的朝向；同时，欧拉变换不仅可以用于世界空间中，同样也可以用于局部参考坐标系中。

值得注意的是，在一些欧拉角的表示中，会让 z 轴指向上方；这虽然会让人感到一些困惑，但是它确实只是一种符号表示上的差异，本质上都是等价的。计算机图形学中，在如何看待和表示世界这个问题上，确实存在着一些分歧，即： y 轴向上（y–up）还是 z 轴向上（z–up）。在包括 3D 打印在内的大部分工业制造领域，都将 z 轴正方向作为世界空间中向上的方向；而在航空和航海领域中，则将 z 轴负方向作为世界空间中向上的方向。建筑和 GIS 领域通常都会使用 z–up，因为建筑设计和地图一般都是在二维上进行的（ x, y ）；与多媒体相关的建模系统一般会使用 y–up，这与我们在计算机图形学中的描述是相符的，即相机屏幕也是 y 轴正方向指向上方。这两种表示方法之间的差异仅仅是旋转 90° 而已（或者是一个对称），但是如果我们不清楚此时使用的是哪一种表示方法的话，可能会导致出现一些问题。在这里，如果没有额外说明的话，我们将会默认使用 y–up。

这里我们还想指出一点，相机的 up 方向和世界空间中 up 方向没有什么特殊关系，我们可以左右倾斜自己的头部，这时我们眼睛的视野也会相应的倾斜，此时头部的 up 方向和世界空间中的 up 方向并不是重合的。再举一个例子：如果现在世界空间使用了 y–up，那么我们的相机将会以一个鸟瞰视角俯瞰整个世界；这是因为我们的相机视角向前旋转了 90° ，其在世界空间中的 up 方向变成了 $(0, 0, -1)$ 。在这种观察角度下，相机的 up 方向将不再指向世界空间中的正 y 轴，而是变成了世界空间中的负 z 轴；但是对于相机而言，自身的 y–up 仍然是成立的。

欧拉角在小角度变换和调整观察者方向方面十分有用，但是它也有一些严重的限制，即我们很难将两组欧拉角组合在一起。例如：在两组欧拉角之间进行插值，并不是简单地对每个分量分别进行插值就可以完成的。事实上，两组表示形式完全不同的欧拉角，可能会给出完全相同的方向，因此对这两组欧拉角进行插值的话，中间生成的任何一组欧拉角，在理想情况下都不应当导致物体发生旋转。这也是使用其他方向表示方法（例如四元数）的原因之一，我们将在之后的小节中对四元数进行详细讨论，这是十分有价值的。使用欧拉角也会导致一个叫做万向节死锁（gimbal lock）的问题，我们将在章节 4.2.2 中对其进行介绍。

4.2.2 从欧拉变换中提取参数

在某些情况下，我们需要从一个代表欧拉变换的矩阵中，提取出各个方向上所改变的角度，即欧拉变换的参数 h, p, r 。这个过程如下所示：

$$\mathbf{E}(h,p,r) = \begin{pmatrix} e_{00} & e_{01} & e_{02} \\ e_{10} & e_{11} & e_{12} \\ e_{20} & e_{21} & e_{22} \end{pmatrix} = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h) \qquad (4.22)$$

齐次变换矩阵是 4×4 的，这里我们只使用了左上角 3×3 子矩阵，因为这已经能够提供旋转矩阵的所有必要信息了；也就是说，在完整的 4×4 欧拉变换矩阵中，除了最右下角的元素是 1 之外，其他剩余的元素均为 0。

我们将三个旋转矩阵相乘，可以获得以下结果：

$$\mathbf{E} = \begin{pmatrix} \cos r \cos h - \sin r \sin p \sin h & -\sin r \cos p & \cos r \sin h + \sin r \sin p \\ \sin r \cos h + \cos r \sin p \sin h & \cos r \cos p & \sin r \sin h - \cos r \sin p \\ -\cos p \sin h & \sin p & \cos p \cos h \end{pmatrix} \qquad (4.23)$$

在方程 4.23 中，我们可以很明显的看出 $\sin p = e_{21}$ ；此外，我们可以令 e_{01} 除以 e_{11} 来计算 r ，令 e_{20} 除以 e_{22} 来计算 h 。具体的参数提取方程如下：

$$\frac{e_{01}}{e_{11}} = \frac{-\sin r}{\cos r} = -\tan r, \qquad \frac{e_{20}}{e_{22}} = \frac{-\sin h}{\cos h} = -\tan h \qquad (4.24)$$

也就是说，我们可以使用 atan2(y, x) （包含两个参数的反正切函数，详见第 1 章）来从矩阵 $\mathbf{E}$ 中提取欧拉角的参数 h （head），p （pitch），r （roll），即：

$$
\begin{aligned}
h &= \mathsf{atan2}(-e_{20}, e_{22}) \\
p &= \arcsin(e_{21}) \\
r &= \mathsf{atan2}(-e_{01}, e_{11})
\end{aligned} \tag{4.25}
$$

但是，这里我们还需要处理一种特殊情况，即当 $\cos p = 0$ 的时候，我们会遇到被称为万向节死锁的问题（在章节 4.2.1 的末尾提到过），此时旋转角度 r, h 将会围绕着同一个旋转轴进行旋转（尽管它俩的旋转方向可能不同，这取决于旋转角度 p 是 $-\pi/2$ 还是 $\pi/2$），在这种情况下，我们只需要计算其中任意一个角度即可。如果我们假设 $h = 0$ [1769]，那么此时矩阵 $\mathbf{E}$ 为：

$$
\begin{aligned}
\mathbf{E_1} &= \begin{pmatrix}
\cos r \cos h - \sin r \sin h & 0 & \cos r \sin h + \sin r \cos h \\
\sin r \cos h + \cos r \sin h & 0 & \sin r \sin h - \cos r \cos h \\
0 & 1 & 0
\end{pmatrix} \\
&= \begin{pmatrix}
\cos(r+h) & 0 & \sin(r+h) \\
\sin(r+h) & 0 & \cos(r+h) \\
0 & 1 & 0
\end{pmatrix}, p = \pi/2
\end{aligned} \tag{4.25.1}
$$

$$
\begin{aligned}
\mathbf{E_2} &= \begin{pmatrix}
\cos r \cos h + \sin r \sin h & 0 & \cos r \sin h - \sin r \cos h \\
\sin r \cos h - \cos r \sin h & 0 & \sin r \sin h + \cos r \cos h \\
0 & -1 & 0
\end{pmatrix} \\
&= \begin{pmatrix}
\cos(r-h) & 0 & \sin(r-h) \\
\sin(r-h) & 0 & \cos(r-h) \\
0 & -1 & 0
\end{pmatrix}, p = -\pi/2
\end{aligned}
$$

当 $\cos p = 0$ 的时候，由于方程 4.23 中的 $e_{01}, e_{11}, e_{20}, e_{21}$ 都包含 $\cos p$ 项，因此都为 0（如方程 4.25.1 所示），所以我们就无法使用方程 4.25 中来解析出 h, r 的值。而且我们可以在方程 4.25.1 中看到，此时方程仅和 $r + h$ 或者 $r - h$ 有关；这里我们只需要假设其中任意一个旋转角度为 0，然后求出另一个参数即可；例如 $h = 0$，即 $\cos h = 1, \sin h = 0$，那么此时的欧拉变换矩阵可以写成：

$$
\mathbf{E} = \begin{pmatrix}
\cos r & -\sin r \cos p & \sin r \sin p \\
\sin r & \cos r \cos p & -\cos r \sin p \\
0 & \sin p & \cos p
\end{pmatrix} \tag{4.26}
$$

我们可以看出，由于旋转角度 p 并不会对矩阵第一列的值产生任何影响，因此当 $\cos p = 0$ 的时候，我们可以使用 $\sin r / \cos r = \tan r = e_{10}/e_{00}$，即 $r = \text{atan2}(-e_{10}, e_{00})$ 来计算出 r。

另外请注意，函数 arcsin 的定义域是 $-\pi/2 \leq p \leq \pi/2$，这意味着如果我们使用了一个在这个范围之外的 p 值，来创建一个欧拉变换矩阵 $\mathbf{E}$ 的话，那么我们将无法获取最初的 p 值。参数 h, p, r 的组合并不是唯一的，也就是说有很多组欧拉角的组合，都可以生成效果一样的变换。更多有关欧拉角转换的内容，可以阅读 Shoemake 于 1994 年发布的论文[1636]。我们上述所提到的方法只是一个很简单的版本，它可能会导致数值不稳定（numerical instability）的问题，我们可以牺牲一些性能来避免这个问题[1362]。

当我们使用欧拉变换的时候，有时会发生一种叫做万向节死锁（gimbal lock）的现象，即在旋转的过程中失去了一个自由度。例如：假设我们现在按照 x, y, z 的顺序进行变换，然后绕 y 轴旋转了 $\pi/2$ 的角度，即执行了第二个旋转；这个旋转操作会使得局部坐标系中的 z 轴与原始的 x 轴重合，最终导致绕 z 轴旋转的操作是多余的。

在数学上，我们已经在方程 4.25.1 和方程 4.26 中看到了万向节死锁的现象，在方程中我们假设了 $\cos p = 0$，即 $p = \pm\pi/2 + 2\pi k$，其中 k 是一个正整数。此时整个矩阵只跟一个角度有关系，如方程 4.25.1 所示，根据 p 值的不同，这个角度可能是 $r + h$ 或者 $r - h$。也就是说，原来的参数 r, h 代表了两个不同的自由度，而现在只有一个自由度了。

在建模系统中，通常会使用 x, y, z 顺序的欧拉角，每个旋转都对应着一个局部坐标轴。在不同的系统中，通常都会使用不同的欧拉角顺序，例如：在动画中使用 z, x, y 顺序，在动画和物理中也会使用 z, x, z 顺序。所有这些分别指定三个旋转轴的方法都是有效的，刚才提到的最后一个有效顺序 z, x, z，在某些应用程序中会更加优越，因为只有当绕 x 轴旋转 π （旋转半周，180°）的时候才会出现万向节死锁。没有任何一个序列能够完美的避免万向节死锁问题，但是欧拉角仍然是最为常用的角度表示方法和旋转表示方法，因为动画师们在进行动画制作的时候，更加喜欢使用曲线编辑器来指定某个角度随着时间的变化，这种方法十分直观和便于理解[499]。

示例：约束变换

想象我们现在拿着一个虚拟的扳手，这个扳手正抓着一个螺栓；为了让这个螺栓安装到位，我们需要绕着 x 轴来旋转扳手。现在假设我们通过操作输入设备（例如鼠标、VR 手套、太空球等）给出了一个旋转矩阵，这个旋转矩阵用于控制扳手的运动。但

是如果直接将这个旋转矩阵应用到扳手上的话可能会导致一定的问题，因为扳手只应当绕着 x 轴进行旋转。我们将这个输入的变换矩阵称为 $\mathbf{P}$ ，为了限制这个变换矩阵只围绕着 x 轴进行旋转，需要使用本小节中描述的方法，将变换矩阵中的欧拉角度提取出来，然后创建一个新的变换矩阵 $\mathbf{R}_x(p)$ 即可。这个新的变换矩阵 $\mathbf{R}_x(p)$ ，就是我们希望应用于扳手的变换矩阵，它将会使得扳手只绕着 x 轴进行旋转（如果此时输入的 $\mathbf{P}$ 包含绕 x 轴旋转的变换的话）。

4.2.3 矩阵分解

到目前为止，我们所做的工作都建立在这样的一个假设上，即已经知道了原始变换矩阵及其变换过程；但是更多的时候，我们实际上并不清楚这些信息。例如：我们只知道有一个变换矩阵和被变换的物体，现在的任务是从在这个总变换矩阵中，分解出各种各样的子变换矩阵，这个过程叫做矩阵分解（matrix decomposition）。

矩阵分解有很多用途，例如：

- 提取物体的缩放因子。
- 找到一个指定坐标系中所需要的变换（例如：某些系统和变换不允许使用任意的 4×4 矩阵）。
- 确定一个物体是否只经历了刚体变换。
- 在只有物体的变换矩阵可用的情况下，在动画的关键帧之间进行插值。
- 移除一个旋转变换矩阵中的剪切变换。

在前文中我们其实已经展示了两个矩阵分解的例子，例如从一个刚体变换中提取出平移矩阵和旋转矩阵（章节 4.1.6）；从一个正交矩阵中提取出欧拉角（章节 4.2.2）。

在上文中的两个例子中我们可以看到，从一个变换矩阵中提取出平移矩阵是很简单的，我们只需要找到 4×4 矩阵中的最后一列元素即可。我们还可以对变换矩阵的行列式进行检查，如果行列式的值是一个负数，那么就说明这个矩阵包含一个反射变换。而想要分离出旋转、缩放和剪切变换则需要更多的的努力。

幸运的是，有几篇关于这个话题的文章，它们都提供了可用的在线代码。Thomas [1769]和 Goldman [552, 553]各自都为不同类型的转换提供了对应方法。Shoemake [1635]对他们处理仿射矩阵的方法进行了改进，该算法是独立于参考系的，并且试图对原始的变换矩阵进行分解，从而提取出刚体变换矩阵。

4.2.4 绕任意轴旋转

有时候，能够让一个物体绕着任意轴旋转是一件很方便的事情，假设我们现在有一个归一化的旋转轴 $\mathbf{r}$，我们希望创建一个矩阵，能够让物体绕着这个旋转轴 $\mathbf{r}$ 旋转 α 度。

为了完成这个绕任意轴旋转的操作，我们首先需要进行一次空间变换，将旋转轴 $\mathbf{r}$ 与 x 轴重合。我们可以通过一个旋转矩阵 $\mathbf{M}$ 来完成，然后当我们真正需要的旋转变换完成之后，再使用旋转矩阵 $\mathbf{M}^{-1}$ 将其旋转回最初的位置[314]，整个变换过程如图 4.8 所示：

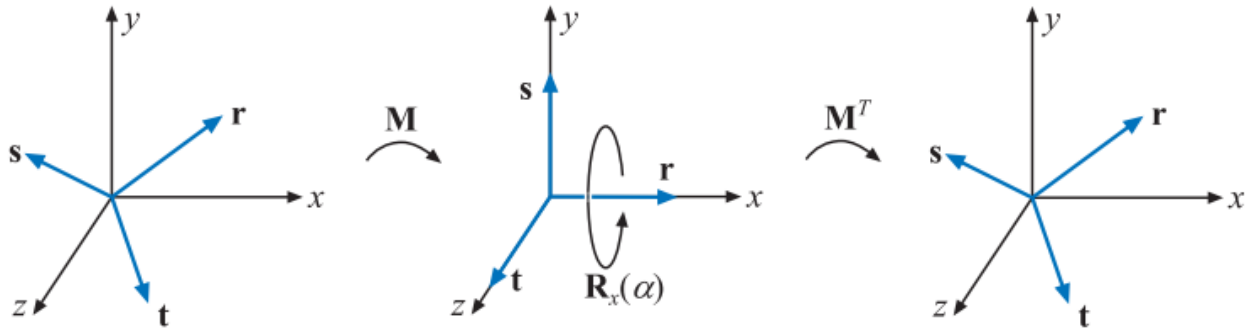


图 4.8：上图展示了绕任意轴 $\mathbf{r}$ 进行旋转的变换流程。我们首先需要找到一组标准正交基 $\mathbf{r}, \mathbf{s}, \mathbf{t}$。然将这个基底与标准基底（坐标轴）重合，即使得旋转轴 $\mathbf{r}$ 与 x 轴重合；然后进行绕 $\mathbf{x}$ 轴旋转 α 度的操作；最后再反向旋转回来。

为了计算这个旋转矩阵 $\mathbf{M}$，我们需要找到两个与旋转轴 $\mathbf{r}$ 正交，且彼此相互正交的轴，从而构建一组标准正交基。其实我们只要能找到第二个轴 $\mathbf{s}$ 即可，因为第三个轴 $\mathbf{t}$ 可以通过前两个轴叉乘的结果获得，即 $\mathbf{t} = \mathbf{r} \times \mathbf{s}$。有一种数值稳定的方法是：找到原始旋转轴 $\mathbf{r}$ 中的最小分量，然后将其设置为 0，再交换剩下两个分量的值，最后再将刚才两个不为 0 的分量中的任意一个取反即可（这里先设置为 0，交换再取反的操作，实际上就是找到与旋转轴 $\mathbf{r}$ 同一平面的垂直向量）。其数学表达式为[784]：

$$
\begin{aligned}
\bar{\mathbf{s}} &= \begin{cases} (0, -r_z, r_y), & \text{if } |r_x| \le |r_y| \text{ and } |r_x| \le |r_z|, \\ (-r_z, 0, r_x), & \text{if } |r_y| \le |r_x| \text{ and } |r_y| \le |r_z|, \\ (-r_y, r_x, 0), & \text{if } |r_z| \le |r_x| \text{ and } |r_z| \le |r_y|, \end{cases} \\
\mathbf{s} &= \bar{\mathbf{s}} / ||\bar{\mathbf{s}}||, \\
\mathbf{t} &= \mathbf{r} \times \mathbf{s}.
\end{aligned} \tag{4.27}
$$

上述方程保证了 $\bar{\mathbf{s}}$ 和 $\mathbf{r}$ 是正交的（垂直），并且 $\mathbf{t}$ 也与 $\mathbf{r}, \mathbf{s}$ 正交，即 $(\mathbf{r}, \mathbf{s}, \mathbf{t})$ 是一组标准正交基。Frisvad [496]提供了一种没有任何分支操作的代码实现，它的速度很快，但是相对精度较低。Max [1147]和 Duff 等人[388]改进了 Frisvad 方法的精度。

无论你使用了哪种方法进行求解，最终都会获得一组标准正交基，从而构建旋转矩阵 $\mathbf{M}$：

$$\mathbf{M} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix} \tag{4.28}$$

在经过旋转矩阵 $\mathbf{M}$ 的变换之后，向量 $\mathbf{r}$ 与 x 轴重合，向量 $\mathbf{s}$ 与 y 轴重合，向量 $\mathbf{t}$ 与 z 轴重合。最终我们获得了绕归一化旋转轴 $\mathbf{r}$ 旋转 α 度的变换矩阵，如下所示：

$$\mathbf{X} = \mathbf{M}^T\mathbf{R}_x(\alpha)\mathbf{M} \tag{4.29}$$

换句话说，我们通过变换，先将旋转轴 $\mathbf{r}$ 与 x 轴重合（使用旋转矩阵 $\mathbf{M}$），然后绕 x 轴旋转 α 度，最后再使用旋转矩阵 $\mathbf{M}$ 的逆矩阵进行反向变换即可。其中，由于旋转矩阵 $\mathbf{M}$ 是一个正交矩阵，因此其逆矩阵就是其转置矩阵 $\mathbf{M}^T$。

Goldman [550]还提出了另外一种绕任意归一化轴 $\mathbf{r}$ 旋转 ϕ 度的方法，这里我们直接将最终的变换矩阵展示出来：

$$\mathbf{R} = \begin{pmatrix} \cos\phi + (1-\cos\phi)r_x^2 & (1-\cos\phi)r_xr_y - r_z\sin\phi & (1-\cos\phi \\ (1-\cos\phi)r_xr_y + r_z\sin\phi & \cos\phi + (1-\cos\phi)r_y^2 & (1-(4.30)\phi \\ (1-\cos\phi)r_xr_z - r_y\sin\phi & (1-\cos\phi)r_yr_z + r_x\sin\phi & \cos\phi + \end{pmatrix}$$

在章节 4.3.2 中，我们还提出了另一种使用了四元数来解决这个问题的方法；同时，我们还提出了用于解决相关问题的高效算法，例如将一个向量转换为另一个向量等。

4.3 四元数

四元数是 William Rowan Hamilton 爵士于 1843 年发明的，当时是作为复数（complex number）的扩展，但是直到 1985，才被 Shoemake 引入计算机图形学的领域中[1633]。

> 公平来说，Robinson [1502]在 1958 年就使用了四元数来进行刚体模拟。

四元数可以用于表示旋转和方向，它在很多地方都比欧拉角和矩阵表示更加优秀。任何三维方向都可以表示为一个绕特定轴的简单旋转，给定一个旋转轴和旋转角度，我们可以直接将其转换为一个四元数，或者是从一个四元数中提取出旋转轴和旋转角

度；但是对任意方向上的欧拉角进行转换是很困难的。四元数可以用于稳定且恒定速度的方向插值，这是欧拉角很难实现的。

复数由一个实部和一个虚部组成，每个复数都可以使用两个实数进行表示，其中第二个实数要乘以 $\sqrt{-1}$ 。类似地，四元数由四个部分组成，前三个值与旋转的轴有关，而旋转角度会对四个值都产生影响（章节 4.3.2）。由于四元数和向量都有四个分量，因此这里我们给四元数的符号加上一个小帽子来以示区别，即 $\hat{\mathbf{q}}$ 。在本小节中，我们将从一些四元数的数学背景开始讲起，然后用它来构建各种有用的变换。

4.3.1 数学背景

我们首先从四元数的定义开始。

定义：四元数 $\hat{\mathbf{q}}$ 可以使用下面的方法来进行定义，这些定义方法都是等价的。

$$
\begin{aligned}
\hat{\mathbf{q}} &= (\mathbf{q}_v, q_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w, \\
\mathbf{q}_v &= iq_x + jq_y + kq_z = (q_x, q_y, q_z), \\
i^2 &= j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k.
\end{aligned} \tag{4.31}
$$

其中变量 q_w 是四元数 $\hat{\mathbf{q}}$ 中的实数部分（实部），变量 $\mathbf{q_v}$ 是四元数 $\hat{\mathbf{q}}$ 中的虚数部分（虚部）， i, j, k 叫做虚数单位。四元数的结构和复数类似，但是复数只有一个虚部，而四元数则包含三个虚部。

对于虚部 $\mathbf{q_v}$ ，我们可以将其看作为一个三维向量，可以对其应用诸如加法、缩放、点乘、叉乘以及其他的向量操作。利用四元数的定义，我们可以得到两个四元数 $\hat{\mathbf{q}}, \hat{\mathbf{r}}$ 之间的乘法运算，如方程 4.32 所示。这里需要注意的是，虚数单位之间的乘法是不具备交换律的。

$$
\begin{aligned}
\textbf{乘法：}\quad \hat{\mathbf{q}}\hat{\mathbf{r}} &= (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w) \\
&= i(q_y r_z - q_z r_y + r_w q_x + q_w r_x) \\
&\quad + j(q_z r_x - q_x r_z + r_w q_y + q_w r_y) \\
&\quad + k(q_x r_y - q_y r_x + r_w q_z + q_w r_z) \\
&\quad + q_w r_w - q_x r_x - q_y r_y - q_z r_z \\
&= (\mathbf{q}_v \times \mathbf{r}_v + r_w \mathbf{q}_v + q_w \mathbf{r}_v, q_w r_w - \mathbf{q}_v \cdot \mathbf{r}_v).
\end{aligned} \tag{4.32}
$$

从上述计算公式中我们可以看出，我们在计算两个四元数的乘法时，同时使用了点乘和叉乘。

除了四元数的定义和乘法计算公式之外，下面是四元数的加法、共轭、范数以及其他定义：

加法：
$$\hat{\mathbf{q}} + \hat{\mathbf{r}} = (\mathbf{q}_v, q_w) + (\mathbf{r}_v, r_w) = (\mathbf{q}_v + \mathbf{r}_v, q_w + r_w)$$

共轭：
$$\hat{\mathbf{q}}^* = (\mathbf{q}_v, q_w)^* = (-\mathbf{q}_v, q_w)$$

模长：
$$\begin{aligned} n(\hat{\mathbf{q}}) &= \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*} = \sqrt{\hat{\mathbf{q}}^*\hat{\mathbf{q}}} = \sqrt{\mathbf{q}_v \cdot \mathbf{q}_v + q_w^2} \\ &= \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}. \end{aligned} \tag{4.33}$$

虚数单位：
$$\hat{\mathbf{i}} = (\mathbf{0}, 1)$$

当 $n(\hat{\mathbf{q}}) = \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}$ 化简之后，我们可以看到最终结果的虚部消失了，只剩下实部（即一个实数），这个实数叫做虚数 $\hat{\mathbf{q}}$ 的模长，有时候我们也会使用 $\|\hat{\mathbf{q}}\| = n(\hat{\mathbf{q}})$ 来表示一个虚数的模长[1105]。我们使用符号来 $\hat{\mathbf{q}}^{-1}$ 表示一个四元数的逆，四元数的逆有这样一个性质，即 $\hat{\mathbf{q}}^{-1}\hat{\mathbf{q}} = \hat{\mathbf{q}}\hat{\mathbf{q}}^{-1} = 1$ 。根据虚数模长的定义，我们可以推导出：

$$n(\hat{\mathbf{q}})^2 = \hat{\mathbf{q}}\hat{\mathbf{q}}^* \iff \frac{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}{n(\hat{\mathbf{q}})^2} = 1 \tag{4.34}$$

从方程 4.34 我们可以推导出四元数逆的计算公式：

四元数的逆：
$$\hat{\mathbf{q}}^{-1} = \frac{1}{n(\hat{\mathbf{q}})^2}\hat{\mathbf{q}}^* \tag{4.35}$$

方程 4.35 使用了四元数的标量乘法，其计算方法可以从方程 4.32 中推导出来，即

$$\begin{aligned} s\hat{\mathbf{q}} &= (\mathbf{0}, s)\,(\mathbf{q}_v, q_w) = (s\mathbf{q}_v, sq_w) \\ \hat{\mathbf{q}}s &= (\mathbf{q}_v, q_w)\,(\mathbf{0}, s) = (s\mathbf{q}_v, sq_w) \end{aligned}$$

这也意味着复数的标量乘法具备交换律，即： $s\hat{\mathbf{q}} = \hat{\mathbf{q}}s = (s\mathbf{q}_v, sq_w)$ 。

利用上面我们提到的基本定义，可以推导出以下的四元数运算规则：

$$
\begin{aligned}
&\textbf{共轭法则：} && (\hat{\mathbf{q}}^*)^* = \hat{\mathbf{q}}, \\
& && (\hat{\mathbf{q}} + \hat{\mathbf{r}})^* = \hat{\mathbf{q}}^* + \hat{\mathbf{r}}^*, \\
& && (\hat{\mathbf{q}}\hat{\mathbf{r}})^* = \hat{\mathbf{r}}^*\hat{\mathbf{q}}^*.
\end{aligned} \tag{4.36}
$$

$$
\begin{aligned}
&\textbf{模长法则：} && n\left(\hat{\mathbf{q}}^*\right) = n(\hat{\mathbf{q}}) \\
& && n(\hat{\mathbf{q}}\hat{\mathbf{r}}) = n(\hat{\mathbf{q}})n(\hat{\mathbf{r}})
\end{aligned} \tag{4.37}
$$

$$
\begin{aligned}
&\textbf{乘法分配律：} && \hat{\mathbf{p}}(s\hat{\mathbf{q}} + t\hat{\mathbf{r}}) = s\hat{\mathbf{p}}\hat{\mathbf{q}} + t\hat{\mathbf{p}}\hat{\mathbf{r}} \\
& && (s\hat{\mathbf{p}} + t\hat{\mathbf{q}})\hat{\mathbf{r}} = s\hat{\mathbf{p}}\hat{\mathbf{r}} + t\hat{\mathbf{q}}\hat{\mathbf{r}} \\
&\textbf{乘法结合律：} && \hat{\mathbf{p}}(\hat{\mathbf{q}}\hat{\mathbf{r}}) = (\hat{\mathbf{p}}\hat{\mathbf{q}})\hat{\mathbf{r}}
\end{aligned} \tag{4.38}
$$

单位四元数 $\hat{\mathbf{q}} = (\mathbf{q}_v, q_w)$ 的长度（模长）为 $n\left(\hat{\mathbf{q}}\right) = 1$ 。因此对于某些长度为 1 的三维向量 $\mathbf{u}_q$ ，我们可以将这个四元数改写成如下形式：

$$
\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi) = \sin\phi\mathbf{u}_q + \cos\phi = \cos\phi + \sin\phi(\mathbf{u}_{qx} + \mathbf{u}_{qy} + \mathbf{u}_{qz}) \tag{4.39}
$$

当且仅当 $\mathbf{u}_q \cdot \mathbf{u}_q = 1 = \|\mathbf{u}_q\|^2$ 的时候，我们才能够这样改写，这是因为：

$$
\begin{aligned}
n(\hat{\mathbf{q}}) = n\left(\sin\phi\mathbf{u}_q, \cos\phi\right) &= \sqrt{\sin^2\phi\left(\mathbf{u}_q \cdot \mathbf{u}_q\right) + \cos^2\phi} \\
&= \sqrt{\sin^2\phi + \cos^2\phi} = 1
\end{aligned} \tag{4.40}
$$

在下一小节中我们将会看到，单位四元数可以通过一种十分高效的方式来创建旋转和方向；但是在那之前，我们还需要介绍一些针对单位四元数的操作。

对于复数而言，一个二维单位向量可以被写成 $\cos\phi + i\sin\phi = e^{i\phi}$ 的形式（复变函数中的欧拉公式）；而对于一个四元数来说，其等价形式如下：

$$
\hat{\mathbf{q}} = \sin\phi\mathbf{u}_q + \cos\phi = e^{\phi\mathbf{u}_q} \tag{4.41}
$$

对于一个单位四元数而言，其对数运算和幂运算遵循以下规则，这些规则可以从方程 4.41 中推导出来：

$$\begin{aligned}\textbf{Logarithm}: &\quad \log(\hat{\mathbf{q}}) = \log\left(e^{\phi \mathbf{u}_q}\right) = \phi \mathbf{u}_q \\ \textbf{Power}: &\quad \hat{\mathbf{q}}^t = (\sin\phi \mathbf{u}_q + \cos\phi)^t = e^{\phi t \mathbf{u}_q} = \sin(\phi t)\mathbf{u}_q + \cos(\phi t\end{aligned} \tag{4.42}$$

4.3.2 四元数变换

我们现在将研究四元数中的一个子集，即单位四元数（unit quaternion），它们的模长为 1。单位四元数可以用于表示任何的三维旋转，而且这种表示方式非常紧凑和简单。

> 译者：这里推荐一个 3b1b 等人制作的互动视频网站 https://eater.net/quaternions（以及相应的四元数科普视频），上面展示了四元数是如何作用于三维空间的旋转。其中最重要的一个理解（译者自己的理解）就是：一个复数的虚部代表了空间在单位圆上进行旋转，实部代表了这个空间的拉伸。

这里我们将介绍为什么单元四元数对于旋转和方向如此有用。假设现在有一个点（或者向量） $\mathbf{p} = \begin{pmatrix} p_x & p_y & p_z & p_w \end{pmatrix}^T$，首先我们将这个点坐标的四个分量对应放入一个四元数 $\hat{\mathbf{p}}$ 的各个分量中；假设我们现在还有一个单位四元数 $\hat{\mathbf{q}} = (\sin\phi \mathbf{u}_q + \cos\phi)$，我们对这两个四元数进行如下操作：

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1} \tag{4.43}$$

这个操作意味着将四元数 $\hat{\mathbf{p}}$（即点 $\mathbf{p}$）以 $\mathbf{u}_q$ 为旋转轴，旋转了 2ϕ 角度。请注意，由于 $\hat{\mathbf{q}}$ 是一个单位四元数，因此根据方程 4.35，有 $\hat{\mathbf{q}}^{-1} = \hat{\mathbf{q}}^*$。这个绕任意轴旋转的过程如图 4.9 所示。

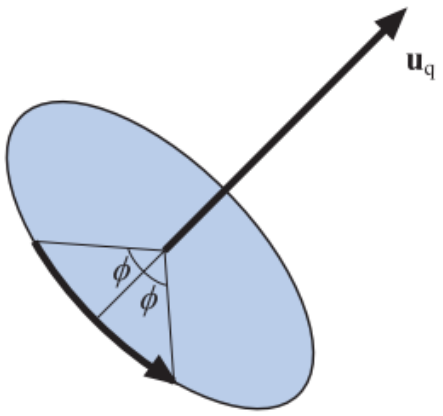


图 4.9：通过使用一个单位四元数 $\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q + \cos\phi)$ 完成的旋转操作，绕轴 $\mathbf{u}_q$ 旋转了 2ϕ 。

任意非零倍数的 $\hat{\mathbf{q}}$ 都能表示这个相同的旋转变换，也就是说， $\hat{\mathbf{q}}$ 和 $-\hat{\mathbf{q}}$ 所表示的旋转变换也都是一样的（不考虑缩放），因此我们可以这样理解，将旋转轴 $\mathbf{u}_q$ 和实部 q_w 取反，可以创建一个与原始四元数完全相同的旋转变换。由于单位四元数 $\hat{\mathbf{q}}$ 首先将目标绕轴旋转 ϕ ，然后 $-\hat{\mathbf{q}}$ 再将目标绕轴旋转 ϕ ，并且两次旋转的方向是相同的，因此最终旋转了 2ϕ 。同时这也意味着我们从变换矩阵中提取出的四元数，可能是 $\hat{\mathbf{q}}$ 或者 $-\hat{\mathbf{q}}$ 中的任意一个。

给定两个单位四元数 $\hat{\mathbf{q}}$ 和 $\hat{\mathbf{r}}$ ，我们首先对 $\hat{\mathbf{p}}$ （可以被认为一个点或者是一个向量）应用四元数 $\hat{\mathbf{q}}$ 所代表的旋转变换，然后再应用四元数 $\hat{\mathbf{r}}$ 所代表的变换，整个过程可以用方程 4.44 进行描述：

$$\hat{\mathbf{r}}\left(\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*\right)\hat{\mathbf{r}}^* = (\hat{\mathbf{r}}\hat{\mathbf{q}})\hat{\mathbf{p}}(\hat{\mathbf{r}}\hat{\mathbf{q}})^* = \hat{\mathbf{c}}\hat{\mathbf{p}}\hat{\mathbf{c}}^* \tag{4.44}$$

上述公式中的 $\hat{\mathbf{c}} = \hat{\mathbf{r}}\hat{\mathbf{q}}$ 也是一个单位四元数，它代表两个单位四元数 $\hat{\mathbf{q}}$ 和 $\hat{\mathbf{r}}$ 的组合变换。

矩阵转换

很多时候我们需要将好几种不同的变换组合在一起，这些变换大多数都以矩阵的形式给出，因此我们需要一种方法来将方程 4.43 中的四元数旋转，转换为矩阵形式。一个四元数 $\hat{\mathbf{q}}$ ，可以转换为一个变换矩阵 $\mathbf{M}^{\mathbf{q}}$ ，其形式如下[1633, 1634]：

$$\mathbf{M}^q = \begin{pmatrix} 1 - s\left(q_y^2 + q_z^2\right) & s\left(q_x q_y - q_w q_z\right) & s\left(q_x q_z + q_w q_y\right) & 0 \\ s\left(q_x q_y + q_w q_z\right) & 1 - s\left(q_x^2 + q_z^2\right) & s\left(q_y q_z - q_w q_x\right) & 0 \\ s\left(q_x q_z - q_w q_y\right) & s\left(q_y q_z + q_w q_x\right) & 1 - s\left(q_x^2 + q_y^2\right) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.45}$$

上述方程中的 $s = 2/(n(\hat{\mathbf{q}}))^2$ ，对于一个单位四元数而言，可以将其化简为：

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2\left(q_y^2 + q_z^2\right) & 2\left(q_x q_y - q_w q_z\right) & 2\left(q_x q_z + q_w q_y\right) & 0 \\ 2\left(q_x q_y + q_w q_z\right) & 1 - 2\left(q_x^2 + q_z^2\right) & 2\left(q_y q_z - q_w q_x\right) & 0 \\ 2\left(q_x q_z - q_w q_y\right) & 2\left(q_y q_z + q_w q_x\right) & 1 - 2\left(q_x^2 + q_y^2\right) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.46}$$

当我们完成这个四元数的构建，在将其转换为矩阵形式的时候，上述转换方程中不包含任何三角函数运算，因此这个转换在实际应用中是很高效的。

将一个正交矩阵 $\mathbf{M}^{\mathbf{q}}$ 反向转换为一个单位四元数 $\hat{\mathbf{q}}$ 要更加复杂一点，这个过程的关键在于，我们可以通过消元法，将方程 4.46 中正交矩阵 $\mathbf{M}^{\mathbf{q}}$ 的某几项进行做差，即：

$$\begin{aligned} m_{21}^q - m_{12}^q &= 4q_w q_x \\ m_{02}^q - m_{20}^q &= 4q_w q_y \\ m_{10}^q - m_{01}^q &= 4q_w q_z. \end{aligned} \tag{4.47}$$

方程 4.47 中的每一项都含有 q_w ，也就是说，如果我们知道了 q_w 的值是多少的话，我们就可以计算出向量 $\mathbf{v}_q$ 的各个分量，那么这个单位四元数 $\hat{\mathbf{q}}$ 也就计算出来了。q_w 可以通过计算矩阵 $\mathbf{M}^{\mathbf{q}}$ 的迹（trace：矩阵主对角线上的元素之和，等于矩阵的特征值之和）获得，计算方程如下：

$$\begin{aligned} \operatorname{tr}\left(\mathbf{M}^q\right) &= 4 - 2s\left(q_x^2 + q_y^2 + q_z^2\right) = 4\left(1 - \frac{q_x^2 + q_y^2 + q_z^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2}\right) \\ &= \frac{4q_w^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} = \frac{4q_w^2}{(n(\hat{\mathbf{q}}))^2}. \end{aligned} \tag{4.48}$$

我们通过方程 4.48 求出矩阵 $\mathbf{M}^{\mathbf{q}}$ 的迹，然后就可以计算出单位四元数的实部 q_w ，最后计算出四元数虚部的其他几项即可，具体方程如下：

$$\begin{aligned} q_w &= \tfrac{1}{2}\sqrt{\operatorname{tr}\left(\mathbf{M}^q\right)}, \quad q_x = \frac{m_{21}^q - m_{12}^q}{4q_w}, \\ q_y &= \frac{m_{02}^q - m_{20}^q}{4q_w}, \quad q_z = \frac{m_{10}^q - m_{01}^q}{4q_w}. \end{aligned} \tag{4.49}$$

为了在计算过程中保持数值稳定[1634]，我们应当尽可能地避免除以一个过小的数值。为了实现这个目的，我们首先设 $t = q_w^2 - q_x^2 - q_y^2 - q_z^2$ ，然后又因为 $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$ （单位四元数的模长为 1），可得：

$$
\begin{aligned}
m_{00} &= t + 2q_x^2, \\
m_{11} &= t + 2q_y^2, \\
m_{22} &= t + 2q_z^2, \\
u &= m_{00} + m_{11} + m_{22} = t + 2q_w^2,
\end{aligned}
\tag{4.50}
$$

上述方程反过来意味着， $m_{00}, m_{11}, m_{22}, u$ 中最大的那一项，对应决定了 q_w, q_x, q_y, q_z 中哪一项是最大的。如果我们计算出来实部 q_w 是最大的那一项，我们便可以使用方程 4.49 计算出这个四元数的其他分量，这样就不会因为除以一个过小的数值而带来精度上的误差。如果计算出来实部 q_w 并不是最大的那一项，那么我们可以使用下列方程进行计算，从而避免精度误差：

$$
\begin{aligned}
4q_x^2 &= +m_{00} - m_{11} - m_{22} + m_{33}, \\
4q_y^2 &= -m_{00} + m_{11} - m_{22} + m_{33}, \\
4q_z^2 &= -m_{00} - m_{11} + m_{22} + m_{33}, \\
4q_w^2 &= \operatorname{tr}\left(\mathbf{M}^q\right).
\end{aligned}
\tag{4.51}
$$

总而言之，方程 4.50 可以用来计算四元数虚部分量 q_x, q_y, q_z, q_w 中的最大值，根据最大值的不同适当选取方程 4.51 或者方程 4.47 来计算 $\hat{\mathbf{q}}$ 的其余分量。 Schuler [1588]提出了一种无分支的变体方法，但是该方法计算了四次平方根。

球面线性插值

球面线性插值（spherical linear interpolation）是指，给定两个单位四元数 $\hat{\mathbf{q}}$ ， $\hat{\mathbf{r}}$ 以及一个参数 $t \in [0, 1]$ ，然后计算出一个插值的四元数。球面线性插值对于物体动画非常有用，但是对于相机的方向插值来说，却并不常用，因为相机的 up 向量在插值的过程中可能会发生倾斜，这通常是一个干扰的效果。

球面线性插值的数学表达式如下：

$$
\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \left(\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1}\right)^t \hat{\mathbf{q}}
\tag{4.52}
$$

但是，对于软件实现而言，通常会使用更为合适的 slerp 函数来代表球面线性插值，其形式如下：

$$
\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \operatorname{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(1-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin(\phi t)}{\sin\phi}\hat{\mathbf{r}}
\tag{4.53}
$$

其中 $\cos\phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$ [325]。对于给定的 $t \in [0,1]$，slerp 函数所计算出的四元数是唯一的（当且仅当 $\hat{\mathbf{q}}$ 不与 $\hat{\mathbf{r}}$ 反向。）。这些插值生成的四元数组合在一起，构成了一个四维单位球面上，从 $\hat{\mathbf{q}}(t=0)$ 到 $\hat{\mathbf{r}}(t=1)$ 的最短弧。如图 4.10 所示，四元数 $\hat{\mathbf{q}}$，$\hat{\mathbf{r}}$ 与原点一起构成了一个平面，这个平面与四维单位超球相交，投影到三维空间中便形成了一个圆，而这些插值出来的四元数所形成的圆弧便位于这个圆上。随着参数 t 的均匀变化，这个插值出来的四元数会绕着一个固定的轴，以一个恒定的角速度进行旋转；像这样一个具有恒定变化速度（即加速度为 0）的曲线被叫做测地线（geodesic curve）[229]。球面上的大圆（great circle）是由经过原点的平面与球体相交形成的，这个圆上的一部分被叫做大弧（great arc）。

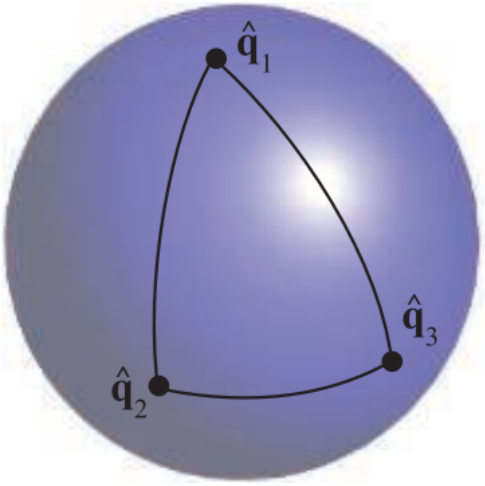


图 4.10：单位四元数可以表示为单位球面（实际上是一个四维空间中的超球，投影到了三维空间中）上的点。slerp 函数用于对两个四元数进行插值，插值形成的路径是球面上的一段圆弧。需要注意的是，从 $\hat{\mathbf{q}}_1$ 插值到 $\hat{\mathbf{q}}_2$，与从 $\hat{\mathbf{q}}_1$ 插值到 $\hat{\mathbf{q}}_3$，再插值到 $\hat{\mathbf{q}}_2$ 并不是一回事。虽然它们插值的起点和终点是相同的，但是中间具体的路径和方向是不同的。

slerp 函数非常适合用于在两个方向之间进行插值，这个插值过程会围绕一个固定的旋转轴，且旋转速度是恒定的；而在使用欧拉角进行方向插值的时候则并不是这样的（旋转轴不固定，旋转速度也不恒定）。在实际实现中，由于 slerp 函数中涉及到了很多三角函数的计算，因此效率是很低的。Malyshau [1114]讨论了如何将四元数集成到渲染管线中，他指出：当直接在像素着色器中对四元数进行归一化，而不是使用 slerp 函数来调整三角形的朝向时，在 90 度角内的最大误差为 4 度，这个误差在光栅化三角形的时候是可以接受的。Li [1039, 1040]提供了一种效率更高的增量方法来计算 slerp，并且不会对精度产生任何影响。Eberly [406]提出了一种仅使用加法和乘法来快速计算 slerp 的方法。

当现在有两个以上的方向时，例如 $\hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_{n-1}$，我们希望可以从从 $\hat{\mathbf{q}}_1$ 插值到 $\hat{\mathbf{q}}_2$，再插值到 $\hat{\mathbf{q}}_3$，直到最终插值到 $\hat{\mathbf{q}}_{n-1}$，使用 slerp 函数是最直接的方式。例如现在我们到达了 $\hat{\mathbf{q}}_{i-1}$，正要向 $\hat{\mathbf{q}}_i$ 方向前进，我们将使用 $\hat{\mathbf{q}}_i$，$\hat{\mathbf{q}}_{i-1}$ 来作为 slerp 的参数进行插值；当到达了 $\hat{\mathbf{q}}_i$，要向 $\hat{\mathbf{q}}_{i+1}$ 方向前进时，我们使用 $\hat{\mathbf{q}}_i$，$\hat{\mathbf{q}}_{i+1}$ 作为 slerp 的参数进行插值。但是这样的操作会导致在插值过程中的方向发生突变，如图 4.10 所示；这和点的线性插值情况相类似，如图 17.3 的右上角部分所示。当你阅读完第 17 章有关样条线的内容之后，你可以重新浏览一下下面的内容，可能会有额外的收获。

一个更好的方法是使用某种样条线来进行插值。我们在四元数 $\hat{\mathbf{q}}_i$ 与 $\hat{\mathbf{q}}_{i+1}$ 之间引入了 2 个新的中间四元数 $\hat{\mathbf{a}}_i$，$\hat{\mathbf{a}}_{i+1}$。我们可以使用 $\hat{\mathbf{q}}_i, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, \hat{\mathbf{q}}_{i+1}$ 这四个四元数，来定义一个球面立方插值（Spherical cubic interpolation）。这两个中间四元数可以通过下面的方程进行计算（Shoemake [1633]给出了另一种推导方式）：

$$\hat{\mathbf{a}}_i = \hat{\mathbf{q}}_i \exp\left[-\frac{\log\left(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i-1}\right) + \log\left(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i+1}\right)}{4}\right]. \tag{4.54}$$

我们将使用 $\hat{\mathbf{q}}_i, \hat{\mathbf{a}}_i$ 来构建一个光滑的三次样条线，从而对四元数进行光滑的球面插值：

$$\begin{aligned}&\mathbf{squad}\left(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t\right) = \\ &\mathbf{slerp}\left(\mathbf{slerp}\left(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, t\right), \mathbf{slerp}\left(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t\right), 2t(1-t)\right)\end{aligned} \tag{4.55}$$

从上述方程中我们可以看到，**squad** 函数是通过重复使用 slerp 进行球面线性插值构建而成的（章节 17.1.1 中包含了有关于对点重复线性插值的内容）。这个插值形成的路径，会通过每一个 $\hat{\mathbf{q}}_i$，但是并不会通过这些用于构建样条线的 $\hat{\mathbf{a}}_i$ ——它们用于指示位于初始位置时的切线方向。

将一个向量旋转到另一个向量

一个常见的操作是将方向 $\mathbf{s}$ 以最短路径变换到另一个方向 $\mathbf{t}$，使用四元数可以大大简化这个变换过程，这也显示了四元数与其表示方法的密切关系。首先我们要将 $\mathbf{s}$ 和 $\mathbf{t}$ 进行归一化；其次通过 $\mathbf{u} = (\mathbf{s} \times \mathbf{t})/\|\mathbf{s} \times \mathbf{t}\|$ 来计算归一化的旋转轴（同时垂直于向量 $\mathbf{s}$ 和 $\mathbf{t}$）；然后根据 $e = \mathbf{s} \cdot \mathbf{t} = \cos(2\phi)$，$\|\mathbf{s} \times \mathbf{t}\| = \sin(2\phi)$，其中 2ϕ 是向量 $\mathbf{s}$ 和 $\mathbf{t}$ 之间的夹角。这时候我们就可以表示从 $\mathbf{s}$ 变换到 $\mathbf{t}$ 的四元数了，即 $\hat{\mathbf{q}} =$

$(\sin\phi\mathbf{u}, \cos\phi)$；我们将其展开可得 $\hat{\mathbf{q}} = \left(\frac{\sin\phi}{\sin 2\phi}(\mathbf{s}\times\mathbf{t}), \cos\phi\right)$，又因为 $\cos\phi = \sqrt{\frac{e+1}{2}}$，进一步化简可得[1197]：

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}}(\mathbf{s}\times\mathbf{t}), \frac{\sqrt{2(1+e)}}{2}\right) \tag{4.56}$$

使用这种方式构建四元数（相比于对叉积 $\mathbf{s}\times\mathbf{t}$ 的结果进行标准化），避免了当 $\mathbf{s}$ 和 $\mathbf{t}$ 几乎指向相同方向时造成的数值不稳定[1197]。但是当 $\mathbf{s}$ 和 $\mathbf{t}$ 指向相反方向的时候，此时 $e = -1$，两种方法都会出现数值不稳定的情况，因为分母为 0；当检测到这种特殊情况出现的时候，可以使用任何一个垂直于 $\mathbf{s}$ 的向量作为旋转轴，然后再将 $\mathbf{s}$ 旋转到 $\mathbf{t}$ 。

有时我们可能会需要一个表示 $\mathbf{s}$ 旋转到 $\mathbf{t}$ 的矩阵，上文中的方程 4.46 给出了单位四元数所表示的旋转矩阵，我们可以从这个矩阵出发，将方程 4.56 带入其中并进行化简，可得[1233]：

$$\mathbf{R}(\mathbf{s},\mathbf{t}) = \begin{pmatrix} e+hv_x^2 & hv_xv_y - v_z & hv_xv_z + v_y & 0 \\ hv_xv_y + v_z & e+hv_y^2 & hv_yv_z - v_x & 0 \\ hv_xv_z - v_y & hv_yv_z + v_x & e+hv_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.57}$$

在这个计算过程中，会使用到以下的中间计算过程：

$$\begin{aligned} \mathbf{v} &= \mathbf{s}\times\mathbf{t} \\ e &= \cos(2\phi) = \mathbf{s}\cdot\mathbf{t}, \\ h &= \frac{1-\cos(2\phi)}{\sin^2(2\phi)} = \frac{1-e}{\mathbf{v}\cdot\mathbf{v}} = \frac{1}{1+e}. \end{aligned} \tag{4.58}$$

我们可以看到，通过对方程进行简化，我们避免了平方根运算和三角函数运算，因此这是一种效率很高的计算方式。请注意方程 4.57（使用四元数将一个向量旋转到另一个向量）和方程 4.30（使用欧拉变换将一个向量旋转到另一个向量），二者的结构是类似的，但是使用四元数变换的方程 4.57 中，并没有涉及对三角函数的计算，因此其效率更高。

当向量 $\mathbf{s}$ 和 $\mathbf{t}$ 平行或者接近平行的时候，此时 $\|\mathbf{s}\times\mathbf{t}\| \approx 0$，我们需要格外小心。当 $\phi \approx 0$ 的时候，我们可以返回单位矩阵；当时当 $2\phi \approx \pi$ 的时候，我们可以绕任意

轴旋转 π 弧度，这个轴可以是向量 $\mathbf{s}$ 与任何其他不平行于 $\mathbf{s}$ 的向量之间的叉乘（章节 4.2.4）。Moller 和 Hughes 使用了 Householder 矩阵（初等反射矩阵，根据一个向量和平面，生成一个反射的向量），来以一种不同的方法来处理这个特殊情况[1233]。

4.4 顶点混合

想象现在有一个数字人的手臂，他由两部分组成，分别是前臂和上臂，这个模型可以使用刚体变换（章节 4.1.6）来进行动画处理，如图 4.11 左半边所示。但是这两部分之间的关节并不像一个真正的手臂肘关节，这是因为我们使用了两个独立的子模型，即这个肘关节由两个独立模型的重叠部分所组成。这个由静态模型组合而成的物体，并没有解决如何使得关节灵活柔韧（flexible）的问题，很明显，使用一个单独的物体进行动画处理，可以获得更好的效果。

顶点混合（vertex blending）是一种解决这个问题的常见方法[1037, 1903]，它还有几个其他的名称，例如：线性蒙皮混合（linear-blend skinning）、包络（enveloping）或者骨架子空间变形（skeleton-subspace deformation）。虽然该算法的确切来源尚不清楚，但是定义骨骼，并让模型皮肤随着骨骼的变化产生相应的反应则是计算机动画中的基本概念[1100]。在最简单的形式中，数字人的前臂和上臂可以像以前一样分别进行动画处理，但是在关节处，这两个独立的部分会通过一个具有弹性的“皮肤（skin）”连接起来。这部分皮肤由两组不同的顶点组成，其中一部分顶点由前臂的变换矩阵所驱动，另一部分顶点则被上臂的变换矩阵所驱动。这意味着位于关节处的“皮肤”，其三角形顶点不再使用统一的矩阵进行变换，而是每个三角形都使用不同的矩阵来进行变换，如图 4.11 所示。

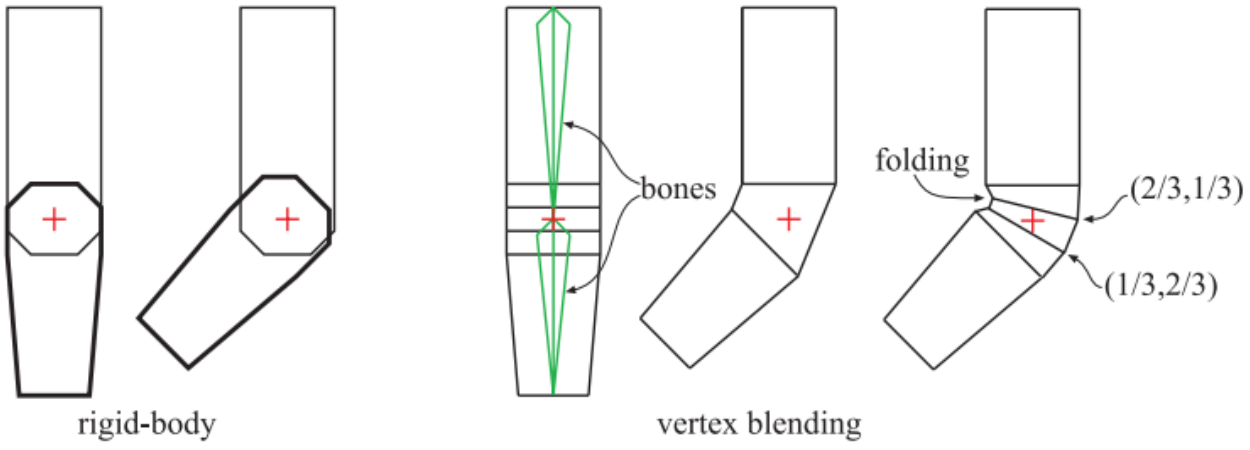


图 4.11：左侧的手臂由前臂和上臂两部分组成，并使用了刚体变换进行动画处理，其关节处表现的并不是很真实自然。在右侧，对整个手臂模型使用了顶点混合方法。其中，位于中间的手

臂展示了，当一个连接两部分的简单皮肤直接覆盖在肘关节时会发生什么。最右侧的手臂展示了使用顶点混合进行处理的结果，其中一些顶点使用了不同的权重进行混合：$(2/3, 1/3)$ 意味着应用在该顶点上的变换有 $2/3$ 来自上臂，有 $1/3$ 来自前臂。在最右侧的图中，还展示了使用顶点混合的一个缺点，即可以看见肘关节内部的折叠。我们还可以使用更多的骨骼节点，以及更加合理的权重选择来获得更好的动画效果。

更进一步说，我们可以允许单个顶点被若干个不同的矩阵所变换，并将这些变换所得到的结果加权混合在一起。这是通过为动画物体设置一个骨架来完成的，通过用户自定义的权重，每个骨骼的变换都可能会对其他的顶点产生影响。由于整个手臂都可能是“有弹性的”，也就是说，所有的顶点都可能会受到多个矩阵变换的影响，因此也可以将整个网格模型称作为包裹骨骼的蒙皮（skin），如图 4.12 所示。许多商业化的建模软件都具有这样的骨架–骨骼建模功能，尽管名字中含有骨骼，但是这个骨骼也不一定是刚性的。例如：Mohr 和 Gleicher [1230]提出了一种想法，可以通过增加额外关节的方法，来实现诸如肌肉膨胀的效果。James 和 Twigg [813]讨论了如何使用可拉伸与可挤压骨骼来进行动画蒙皮。

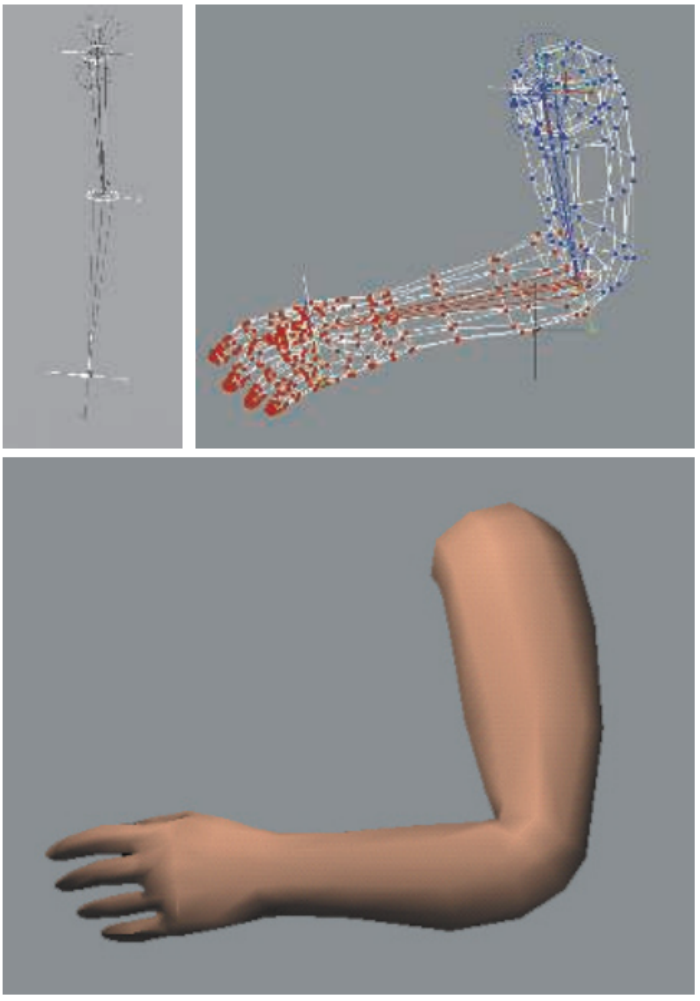


图 4.12：顶点混合的一个真实案例。左上角的图展示了这个手臂的两个骨骼，此时手臂处于伸展状态。右上角的图展示了手臂的顶点和网格模型，其中使用颜色标注了该顶点属于哪个骨骼。在底部的图中，被着色处理的手臂处于一个稍微不同的位置，呈弯曲状态。[968]

顶点混合的数学表达式如方程 4.59 所示，其中 $\mathbf{p}$ 是原始的顶点坐标，$\mathbf{u}(t)$ 是变化后的顶点位置，t 代表了动画时间。方程的具体形式如下：

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \quad \text{where} \quad \sum_{i=0}^{n-1} w_i = 1, \quad w_i \geq 0 \tag{4.59}$$

上述方程的含义是：有 n 个骨骼会对点 $\mathbf{p}$ 的世界空间坐标产生影响；w_i 的值代表了骨骼 i 对于顶点 $\mathbf{p}$ 坐标的影响权重；矩阵 $\mathbf{M}_i^{-1}$ 代表骨骼 i 的模型变换矩阵，它负责将骨骼变换到世界空间中。通常来说，骨骼的控制关节位于自身坐标系的原点位置，例如对于前臂骨骼而言，其控制关节为肘关节，则肘关节应当位于这个前臂骨骼模型空间的原点处；这样可以直接对模型的旋转矩阵进行动画处理，从而可以对整个前臂进行移动，并且移动结束之后，肘关节仍然位于模型空间的原点处。$\mathbf{B}_i(t)$ 是一个世界变换矩阵，代表了第 i 个骨骼的世界变换，它会随着时间进行变化，从而让物体动起来；它通常是由若干个矩阵连接而成的，例如上一层级（父级）的变换以及局部的动画矩阵等。

Woodland 对一种维护和更新动画矩阵 $\mathbf{B}_i(t)$ 的方法进行了深入讨论[1903]。首先找到所有会对顶点产生影响的骨骼，然后每个骨骼都会将这个顶点坐标转换到自身的坐标系中，从而生成一个变换后的新顶点，而该顶点的最终位置，是通过在这些新顶点之间进行插值获得的。在一些关于蒙皮的讨论中，模型变换矩阵 $\mathbf{M}_i$ 被认为是世界变换矩阵 $\mathbf{B}_i(t)$ 的一部分，因此并不会将 $\mathbf{M}_i$ 显式展示出来。在这里我们将其展示了出来，因为这是一个十分有用的矩阵，它总会成为矩阵连接过程的一部分。

实际上，矩阵 $\mathbf{B}_i(t)$ 和矩阵 $\mathbf{M}_i^{-1}$ 在每一帧种都会进行连接组合，最终生成的变换矩阵会被用于对顶点进行变换。点 $\mathbf{p}$ 会被不同骨骼的变换矩阵进行变换，并通过使用权重 w_i ，来将这些变换进行加权混合——这也是顶点混合的名称由来。这些权重的值通常都是非负的，且它们的和为 1，即点 $\mathbf{p}$ 会被变换到若干个位置上，并在这些位置之间进行插值。换句话说，对于一个固定的时刻 t ，最终插值出来的点 $\mathbf{u}$ 位于点集 $\mathbf{B}_i(t)\mathbf{M}_i^{-1}\mathbf{p}, \quad i = 0 \cdots n-1$ 的凸包（convex hull）内部。通常我们也可以使用方程 4.59 来对顶点的法线进行变换，当然这也取决于所使用的变换类型：如果骨骼

被大量拉伸或者挤压（非均匀缩放）的话，就需要使用矩阵 $\mathbf{B}_i(t)\mathbf{M}_i^{-1}$ 的逆来对法线进行变换，这在章节 4.1.7 中讨论过。

顶点混合非常适合在 GPU 上运行，我们可以将网格中的顶点集合放入 GPU 的静态缓存中（只需要发送一次），之后重复使用这个缓存即可。在每一帧中，只有骨骼节点发生了变化（顶点数量和连接关系不会发生变化，即 GPU 上的静态缓存也不会发生变化），我们使用一个顶点着色器，来计算骨骼节点对缓存中存储网格的影响。通过这种方式，可以最大程度的减少在 CPU 上进行的处理，以及与 GPU 之间的数据传输（会带来大量延迟），这使得 GPU 可以高效地渲染这个网格。最简单的情况是可以同时使用模型的整套骨骼矩阵，否则的话就需要拆分模型，并复制一些骨骼。还可以将骨骼的变换存储在顶点可以访问的纹理中，从而避免触及寄存器的限制。通过使用四元数来表示旋转变换，那么每个变换都可以被存储在两张纹理中[1639]（一张存储顶点的位置，一张存储用于旋转的单位四元数）。如果可以使用无序访问视图（UAV）的话，还可以对蒙皮的动画结果进行重复使用[146]。

我们还可以指定超出 $[0, 1]$ 范围之外的权重值，或者是让权重之和不为 1。当然这只在使用一些其他的混合算法时才有意义，例如章节 4.5 中的变形目标（morph target）等。

图 4.13：左图展示了在使用线性蒙皮混合算法时关节会出现的问题；右图则使用了双四元数混合算法，它明显改善了关节处的表

现。[1693]

我们刚才介绍的是最基础的顶点混合算法，它当然会存在一些缺点，例如我们不希望出现的折叠、扭曲或者是自相交等现象[1037]，如图 4.13 所示。一个更好的方案是使用双四元数混合（dual quaternion）[872, 873]，这个技术可以让蒙皮保持原始形状的刚性，从而避免四肢出现像“糖纸”一样的扭曲；这个技术的效果良好，并且计算成本小于线性蒙皮混合的 1.5 倍，因此被广泛采用。但是双四元数混合技术会导致蒙皮发生膨胀，Le 和 Hodings [1001]提出了中心旋转蒙皮（center-of-rotation）作为双四元数的替代，他们认为：局部变换应当是一种刚体变换，并且具有相似权重 w_i 的顶点，应当也具有相似的变换。这种技术预先计算了每个顶点的旋转中心，并对其施加了正交（刚体）约束，从而防止肘部折叠以及糖纸弯曲现象。在运行时，该技术类似于线性蒙皮混合，它运行在 GPU 上，在顶点的旋转中心进行线性蒙皮混合，紧接着再进行四元数混合的操作。

4.5 变形

在动画中，从一个三维模型变形到另一个三维模型是非常有用的[28, 883, 1000, 1005]。假设在 t_0 时刻有一个正在展示的模型，我们希望它能够在 t_1 时刻变形为另一个模型。在 t_0 和 t_1 时刻之间，我们会使用某种插值方法，得到一个不断变化的“混合”模型。图 4.14 展示了变形的一个例子。

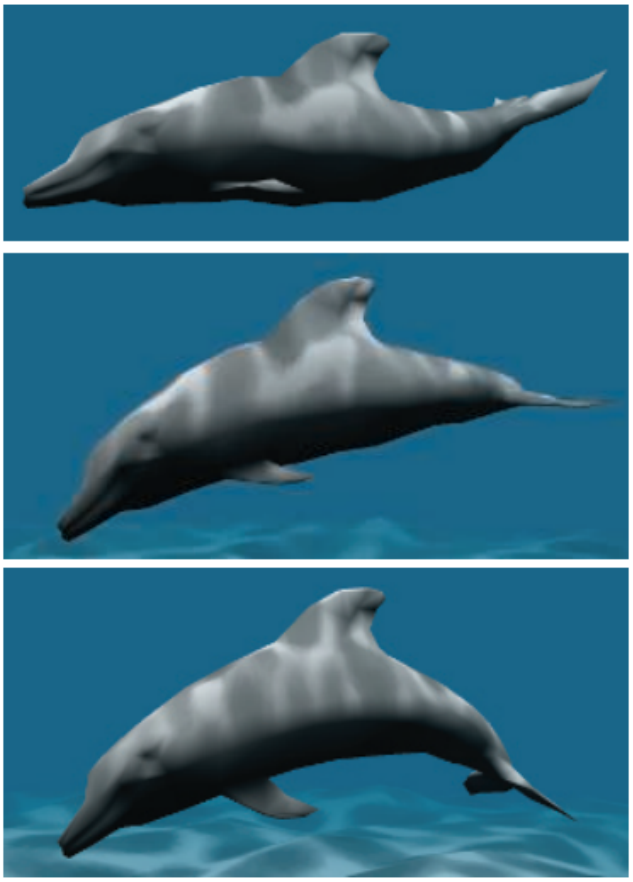


图 4.14：顶点变形（vertex morphing）。每个顶点上都定义了两组位置和法线，在每一帧中，顶点着色器会对这两组位置和法线进行线性插值，从而获得中间的位置和法线。

变形主要会涉及到两个问题：顶点对应（vertex correspondence）问题和插值（interpolation）问题。给定两个任意的模型，它们具有不同的拓扑结构，不同数量的顶点以及不同的网格连通性。我们通常要从设置两个模型顶点之间的对应关系开始，这是一个十分困难的问题，但是在这个领域中，也已经有很多相关的研究了。我们建议感兴趣的读者可以参考 Alexa 的相关调研[28]。

如果两个模型之间已经有了一一对应的顶点了，即对于第一个模型中的每个顶点，我们都可以在第二个模型中找到唯一相对应的顶点。这种情况下的变形操作是很简单的，我们只需要在对应的顶点属性之间进行插值就可以获得中间数据了，例如直接使用线性插值（章节 17.1 中介绍了更多的插值方法）。对于 $t \in [t_0, t_1]$，为了计算这个变形的顶点，首先我们需要计算 $s = (t - t_0) / (t_1 - t_0)$（即当前时刻 t 占总时间的比例），然后将两个对应顶点进行线性混合：

$$\mathbf{m} = (1 - s)\mathbf{p}_0 + s\mathbf{p}_1, \tag{4.60}$$

其中 $\mathbf{p}_0$ 与 $\mathbf{p}_1$ 对应了同一个顶点不同时刻（t_0, t_1）的状态。

变形目标（morph target）或者形状混合（blend shape）[907]是变形技术的其中一种，用户可以对其进行更加直观的控制，图 4.15 展示了其基本思想。

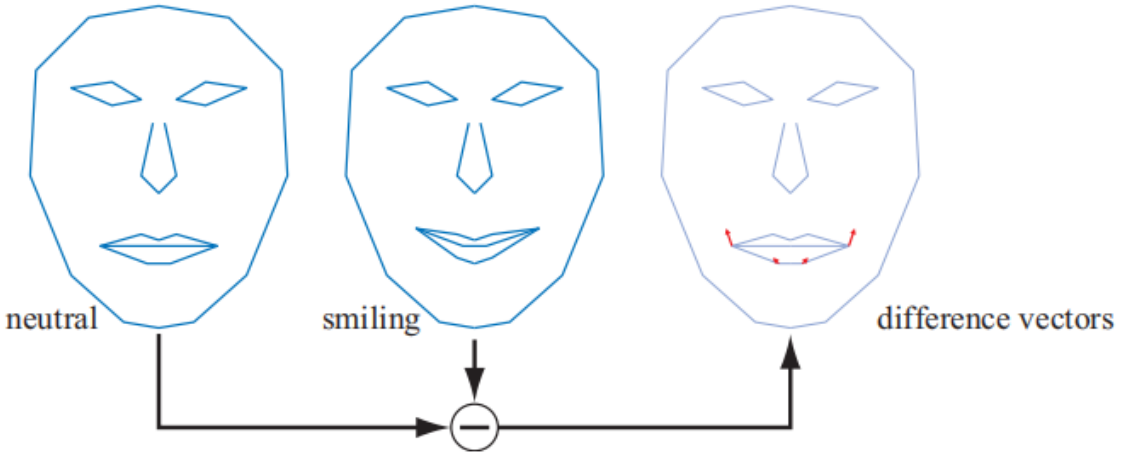


图 4.15：给定两个嘴巴的姿态，可以计算一组差分向量，用于控制插值。在变形目标中，我们可以通过调整差分向量的权重，从而在中性面部（标准面部姿态，类似于 T Pose）上“添加”运动。在这个例子中，当权重为正的时候，我们可以获得一个嘴角上扬的笑脸；当权重为负的时候，我们可以获得一个相反的结果，即嘴角下垂的哭脸。

我们从一个标准的中性模型（neutral model）开始，在这个例子中，这个中性模型是一个没有表情的脸，我们将其标记为 $\mathcal{N}$。此外还需要有一组不同的面部姿态，而在这个例子中我们只有一个笑脸作为额外的面部姿态。一般来说，我们可以允许 $k \geq 1$ 个不同的姿态，我们使用 $\mathcal{P}_i, i \in [1, \ldots, k]$ 来标记它们。在预处理阶段，我们通过使用每个不同的姿态来减去中性姿态，从而计算出姿态之间的差分向量，即 $\mathcal{D}_i = \mathcal{P}_i - \mathcal{N}$。

此时我们有了一个中性模型以及一组不同姿态 $\mathcal{D}_i$，我们可以使用下列方程计算得到一个变形的模型 $\mathcal{M}$：

$$\mathcal{M} = \mathcal{N} + \sum_{i=1}^{k} w_i \mathcal{D}_i \tag{4.61}$$

我们在中性模型的基础上，通过使用不同的权重 w_i，根据需要来将不同姿态的特征进行加权混合。在图 4.15 中，如果设定此时的权重为 $w_1 = 1$，那么我们就会获得一个笑脸，如中间的面部表情；如果设定此时的权重为 $w_1 = 0.5$，那么就可以获得一个半微笑的脸，以此类推。同时我们还可以使用一个负数权重，或者是使用和不为 1 的权重。

对于这个简单的面部模型而言，我们还可以再添加一个带有“悲伤”眉毛的面部姿态，通过使用一个负数权重，从而获得一个“开心”的眉毛。由于这些顶点的位移都是可叠加且相互独立的，因此这个眉毛姿态可以和和嘴巴姿态同时使用。

变形目标是一个强大的技术，它为动画师提供了很多控制能力，因为模型中的不同特征，可以进行独立操控。Lewis 等人[1037]介绍了姿态空间变形技术（pose-space deformation），该技术将顶点混合和变形目标结合在一起。Senior [1608]使用了预计算的顶点纹理来存储和检索目标姿态之间的位移。对于支持流式输出和顶点 ID 的硬件，允许在单个模型上使用多个变形目标，并在 GPU 上专门计算这个效果。此外，还可以使用一个低分辨率的网格，然后通过曲面细分和位移映射（displacement mapping）来生成一个高分辨率的网格，这样避免了在一个高分辨率模型上，对每个顶点进行蒙皮和处理的开销[1971]。

图 4.16：《声名狼藉：次子》的游戏画面，角色 Delsin 的面部使用了形状混合来制作表情动画。所有的这些面部表情，都是基于一个没有表情的面部（中性），

然后通过修改不同的权重，从而使得面部表情看起来不同。

图 4.16 展示了一个同时使用蒙皮和变形的例子。Weronko 和 Andreason [1872]在《教团：1886》中使用了蒙皮和变形。

4.6 几何缓存回放

在一些过场动画中，我们可能会希望使用一些极高质量的动画，这些高质量动画使用上述我们所提到的任何方法和技术都无法实时实现。一种简单的方法是，将所有帧的顶点数据预先计算并存储起来， 然后在游戏运行的时候，从磁盘上读取这些数据并对顶点进行更新。但是对于一个包含 30000 个顶点的模型而言，一段简单的动画可能就会占据 50 MB/s 的带宽。Gneiting [545]提供了若干种方法，可以将内存消耗降低到原来的 10%，其具体做法如下：

首先会对数据进行量化处理，例如：使用 16 位整数来存储每个顶点的位置和纹理坐标，这一步操作会损失一些精度和信息，因为这个压缩过程是不可逆的，在压缩之后无法恢复原始的数据。为了进一步对数据进行压缩，还使用了基于空域（spatial）和时域（temporal）的预测，并对这些差异信息（同一顶点的不同帧，同一帧中的相邻顶点）进行了编码。对于空域压缩而言，可以使用平行四边形预测（parallelogram prediction）技术[800]。对于三角形条带上的一个顶点，我们可以使用该顶点所在的三角形，以当前边为基准，将其反射到当前边的另一侧；这两个三角形构成了一个平行四边形，然后使用这个平行四边形来预测下一个顶点的位置，然后对原顶点和新顶点之间的差异进行编码。在良好预测的情况下，大多数差异数值都会接近于 0，这对于许多压缩方案来说都是很理想的。与 MPEG 压缩类似，还可以在时域上进行预测，即每隔 n 帧进行一次空域压缩。还可以在时域上对两帧之间进行预测，例如：如果一个顶点被一个增量向量从第 $n-1$ 帧移动到了第 n 帧，那么它很有可能会以类似的增量，再次移动到第 $n+1$ 帧。这些技术大大减少了所需要的存储空间，从而使得动画系统可以实时使用流式数据。

4.7 投影

在真正渲染一个场景之前，场景中所有的相关物体都需要被投影到某个平面上，或者是某个简单空间中。在投影变换完成之后，才会进行裁剪操作和渲染操作（章节 2.3）。

到目前为止，本章节中所涉及的诸多变换，并不会对齐次坐标的 w 分量（第四个分量）产生影响；也就是说，点和向量在经过变换之后，仍然会保持它们的类型：点的

w 分量为 1，向量的 w 分量为 0。同时，这些变换矩阵的第四行（最底行）都是 $(0,0,0,1)$ 。而现在我们所要讨论的投影变换，将会涉及到对齐次坐标 w 分量的修改。对于透视投影矩阵（perspective projection matrice）而言，其第四行包含了对向量和点类型的操作；而且在变换之后通常还需要进行均匀化（homogenization）过程，也就是说， w 分量在透视投影变换之后可能并不为 0 或者 1，因此需要将齐次坐标的每个分量都除以 w 分量，这样才能获得非齐次的点坐标。而对于正交投影（orthographic projection），本小节我们首先会进行讨论这种投影方式，这种投影方式比较简单而且会经常使用，但它并不会对齐次坐标的 w 分量产生影响。

在本小节中，我们假设在经过观察变换之后，相机会看向负 z 轴，同时 y 轴指向上，x 轴指向右，即一个标准的右手坐标系。有些书籍和一些图形 API（例如 DirectX）中，在这里会使用一个左手坐标系，即相机看向正 z 轴。使用哪一种手性的坐标系都是可以的，最终也会生成相同的效果。

4.7.1 正交投影

正交投影的一个特征是，投影之前的平行线，在投影之后仍然会保持平行。这也就意味着，当使用正交投影来观察一个场景的时候，无论场景中的物体距离相机多远，它们的大小都不会发生改变。矩阵 $\mathbf{P}_o$ 是一个简单的正交投影矩阵，它将点坐标的 x, y 分量保持不变，然后将 z 分量直接归零，即投影到平面 $z=0$ 上。其具体形式如下：

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.62}$$

图 4.17 展示了正交投影的效果。由于矩阵 $\mathbf{P}_o$ 的行列式为 0，因此它是不可逆的，换句话说，我们将一个物体从三维空间变换到二维平面上，是没有办法将丢失的维度信息恢复的。使用这样的正交投影观察场景会遇到一个问题，那就是它会将在视口内的所有点（无论 z 坐标是正数还是负数）都投影到投影平面上；通常我们会将点坐标的 z 分量（ x, y 分量也有相应的限制，即视口的长宽）限制到一个区间内，一般使用 n （近裁剪平面，也叫做前平面或者 hither）和 f （远裁剪平面，也叫做后平面或者 yon）来进行表示。这里我们对可视空间进行了限制，这也是下一步变换的目的之一。

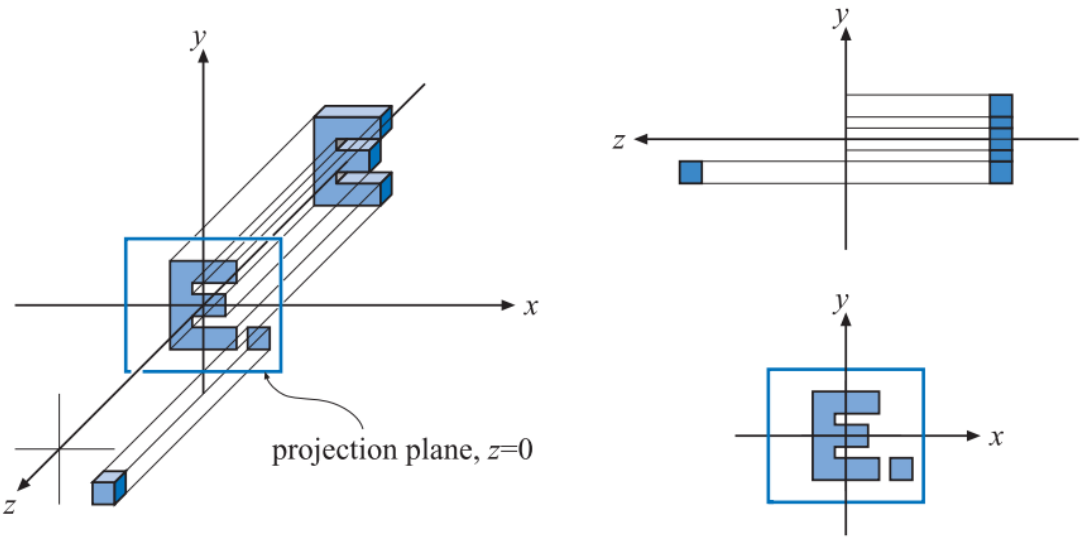


图 4.17：由方程 4.62 生成的简单正交投影，图中展示了这个过程的三个观察视图。这个投影可以看作是观察者沿着负 z 轴进行观察，这意味着这个投影操作实际上只是省略了（或者设置为零） z 坐标，同时保持 x 和 y 坐标不变。请注意，位于平面 $z = 0$ 两边的物体，都会被投影到投影平面上。

通常我们会使用一个六元组（ l, r, b, t, n, f ）来描述一个正交投影矩阵，它们分别代表了左侧、右侧、底部、顶部、近裁剪平面以及远裁剪平面。这个矩阵会将代表可视空间的轴对齐包围盒（axis-aligned bounding box，简称 AABB，详见章节 22.2）转换为一个位于原点的轴对齐立方体。这个 AABB 的最小角是 (l, b, n) ，最大角是 (r, t, f) 。这里我们需要额外注意的是，由于相机此时看向的是负 z 轴，因此 $n > f$ 。而通常人们的直觉是：表示近距离的数值应当要比一个表示远距离的数值小，因此这里一般会让用户按照直观感受来设置远近裁剪平面的数值，然后在程序中再对它们进行相应的调整。

在 OpenGL 中，这个轴对齐立方体的范围是从 $(-1, -1, -1)$ 到 $(1, 1, 1)$ ；而在 DirectX 中这个范围则是 $(-1, -1, 0)$ 到 $(1, 1, 1)$ 。这个立方体被称为规则观察体（canonical view volume，CVV），此时所在的空间被称为规范化设备坐标系（normalized device coordinates，NDC；也叫做齐次裁剪空间）。整个变换过程如图 4.18 所示。将观察空间（view space）转换为 NDC 空间的原因是，这样可以使得裁剪操作更加高效，也使得裁剪操作有了统一的前置标准。

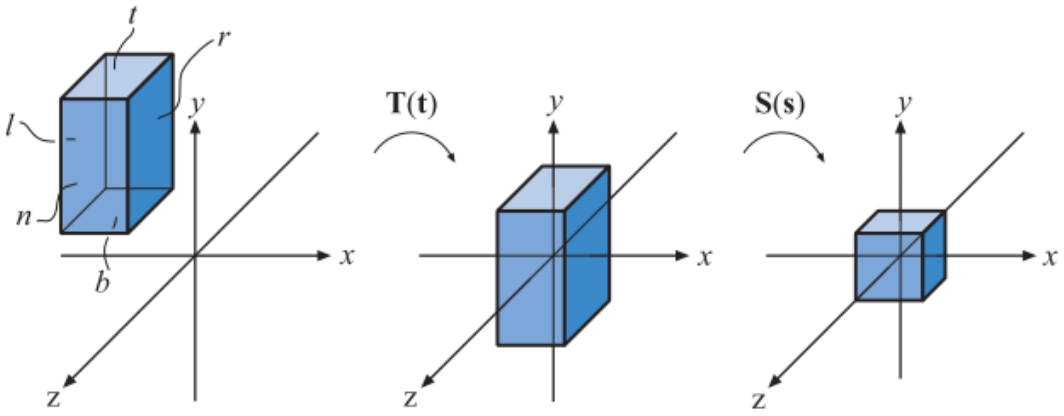


图 4.18：将一个 AABB 转换为一个规则观察体。左侧图中的 AABB 首先会进行平移，使其几何中心位于原点处；然后再对 AABB 进行缩放，使其变成右图中的规则观察体。

在转换为规则观察体之后，需要渲染的几何体顶点会被这个立方体所裁剪。位于立方体内的几何体最终会被保留，然后通过屏幕映射的方式，将剩余的正方形区域渲染到屏幕上。详细的正交投影矩阵如下所示：

$$\begin{aligned}\mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.\end{aligned} \tag{4.63}$$

如方程 4.63 所示，整条投影矩阵 $\mathbf{P}_o$ 可以写成一个平移矩阵 $\mathbf{T}(\mathbf{t})$ 和一个缩放矩阵 $\mathbf{S}(\mathbf{s})$ 的组合，其中：

$$\mathbf{s} = (2/(r-l), 2/(t-b), 2/(f-n)),$$
$$\mathbf{t} = (-(r+l)/2, -(t+b)/2, -(f+n)/2)$$

这个矩阵是可逆的，其逆矩阵为：

$$\mathbf{P}_o^{-1} = \mathbf{T}(-\mathbf{t})\mathbf{S}((r-l)/2, (t-b)/2, (f-n)/2)$$

> 当且仅当 $n \neq f$ ， $l \neq r$ ， $t \neq b$ ；否则，不存在逆矩阵。

在计算机图形学中，在投影变换之后通常会使用一个左手系，即视口的 x 轴指向右方， y 轴指向上方， z 轴指向视口内侧。由于 AABB 的远裁剪平面 z 值要比近裁剪平面小，因此正交投影通常还包会含一个镜像的变换操作。为了看到这个镜像变换操作，这里我们假设原始 AABB 和最终变换的规则观察体的尺寸是一样的，此时 AABB 的范围会从对应 (l, b, n) 的 $(-1, -1, 1)$ 到对应 (r, t, f) 的 $(1, 1, -1)$ 。我们将参数带入方程 4.63，可得：

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.64}$$

此时的正交投影矩阵是一个镜像变换矩阵，正是这个镜像操作会将右手系的观察空间（相机看向负 z 轴）变换为左手系的齐次裁剪空间。

OpenGL 会将点坐标的深度值（ z 分量）映射到 $[-1, 1]$ 中（不用进行额外处理），而 DirectX 则是将其映射到 $[0, 1]$ 中。为了获得在 DirectX 中使用的正交投影矩阵，我们可以在正交投影变换之后，通过应用一个简单的缩放和平移矩阵来完成这个映射操作，即先将范围 $[-1, 1]$ 缩放为原来的一半即 $[-0.5, 0.5]$ ，再将其沿着 z 轴正方形平移 0.5 个单位到 $[0, 1]$ 。这个缩放平移矩阵具体形式如下：

$$\mathbf{M}_{st} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.65}$$

现在我们将正交投影的变换矩阵，与这个对深度值进行缩放和平移的矩阵结合在一起，可以得到最终的正交投影变换矩阵，根据深度值映射的不同可能会有细微的区别，在 DirectX 中，这个矩阵的具体形式如下：

$$\mathbf{P}_{o[0,1]} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.66}$$

在 DirectX 中一般会使用这个矩阵的转置矩阵，因为 DirectX 使用了行优先（row-major）的规则来写入矩阵。

4.7.2 透视投影

透视投影要比正交投影更加复杂，在计算机图形程序中也更加常用。在透视投影中，投影变换前的平行线在投影之后通常就不再平行了，相反，这些平行线可能会在它们的尽头汇聚成一个点。透视投影更加符合我们人眼观察这个世界的模式，因为它具有近大远小的特点。

首先我们先从最简单的情况开始，推导投影到 $z = -d, d > 0$ 平面上的透视投影矩阵。由于在投影变换之前还需要进行一次观察变换，这里我们为了简化这个转换过程的理解难度，所以直接在世界空间中进行推导。在这个最基本的推导完成之后，我们会将其扩展为更加常规的矩阵形式，例如 OpenGL 中所使用的透视投影矩阵[855]。

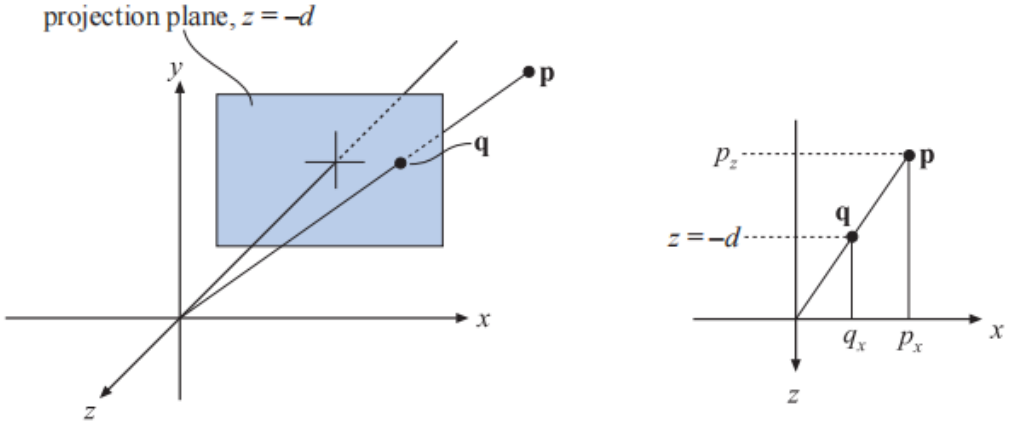


图 4.19：上图是用于推导透视投影矩阵的几何符号。点 $\mathbf{p}$ 投影到平面 $z = -d, d > 0$ 上，最终生成一个新的顶点 $\mathbf{q}$ 。投影是从相机的角度执行的，在本例中相机位于原点。使用右图中的相似三角形，可以推导出点 $\mathbf{q}$ 的 x 分量。

假设现在的相机位于坐标原点处，我们希望将点 $\mathbf{p}$ 投影到平面 $z = -d, d > 0$ 上，最终生成一个新的顶点 $\mathbf{q} = (q_x, q_y, -d)$ ，这个过程如图 4.19 所示。通过图中的相

似三角形，我们可以推导出点 $\mathbf{q}$ 的 x 分量，如下所示：

$$\frac{q_x}{p_x} = \frac{-d}{p_z} \quad \Longleftrightarrow \quad q_x = -d\frac{p_x}{p_z}. \tag{4.67}$$

同理我们可以推导出点 $\mathbf{q}$ 的其他分量，如 $q_y = -dp_y/p_z$ 和 $q_z = -d$ 。将上述公式整合在一起，就可以获得这个透视投影矩阵 $\mathbf{P}_p$ ，即：

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \tag{4.68}$$

方程 4.68 中的透视变换矩阵可以产生正确的透视投影，具体方程如下：

$$\begin{aligned} \mathbf{q} = \mathbf{P}_p\mathbf{p} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix} \end{aligned} \tag{4.69}$$

方程 4.69 中的最后一步，将新顶点的所有分量都除以 w 分量（在这个例子中是 $-p_z/d$ ），从而将 w 分量设置为 1。由于我们将顶点投影到了 $z = -d, d > 0$ 上，因此新顶点的 z 分量始终都为 $-d$ 。

在直觉上，是很容易理解为什么齐次坐标可以用于表示投影操作的。这个齐次化过程（将 w 分量设置为 1）的一种几何解释是：这个过程会将点 (p_x, p_y, p_z) 投影到 $w = 1$ 所在的平面上。

与正交投影类似，透视投影并没有真正地将所有物体都投影到了一个平面上（这个过程是不可逆的），而是将视锥体变换成了一个规则观察体。在透视投影的时候，我们会假设视锥体从 $z = n$ 开始，并在 $z = f$ 结束，其中 $0 > n > f$ 。在 $z = n$ 这个平面上，视锥体的截面是一个长方形，其最小角（左下角）是 (l, b, n) ，最大角（右上角）是 (r, t, f) ，如图 4.20 所示。透视投影的整个过程，可以这样理解：首先将

视锥体的远裁剪平面按一定规则，缩放到与近裁剪平面一样的尺寸，即将视锥体变成一个长方体；然后再按照正交投影的方式，将其变换成一个规则观察体。

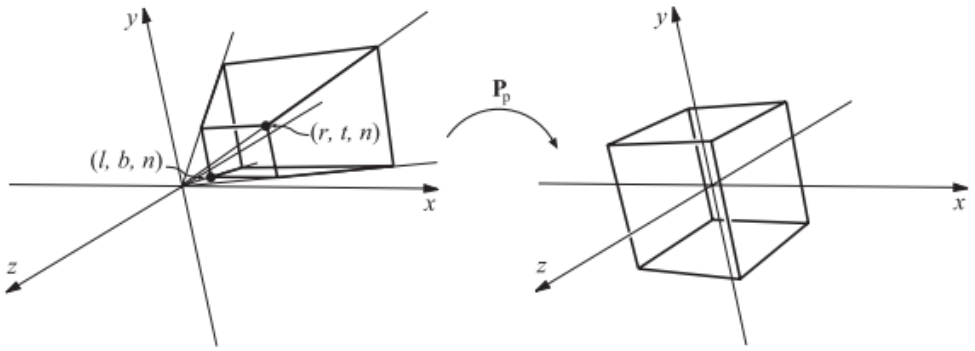


图 4.20：透视投影矩阵 $\mathbf{P}_p$ 将视锥体转换为一个标准立方体，也叫做规则观察体。

参数（l, r, b, t, n, f）决定了相机视锥体的范围，视野的水平视场角由视锥体的左右平面（即 l, r）所决定；同理视野的垂直视场角由视锥体的上下平面（即 b, t）所决定。视场角越大，相机能够看到的内容也就越多。我们也可以通过设定 $r \neq -l, t \neq -b$ 来构建一个不对称的视锥体，这种视锥体通常会用于 3D 立体观察（3D 电影）和虚拟现实（VR）中（章节 21.2.3）。

视场角（field of view，FOV）是提供场景感的重要因素，与电脑屏幕相比，眼睛本身就有一个物理上的视场角（人类单眼的水平视场角最大可达 156 度，双眼的水平视场角最大可达 188 度；人类两眼的重合视场角为 124 度，单眼的舒适视场角为 60 度；当集中注意力时，视场角约为 25 度。），其中水平视场角的计算方法为：

$$\phi = 2 \arctan(w/(2d)) \tag{4.70}$$

其中 ϕ 是视场角，w 是物体垂直于视线的宽度，d 是物体到相机的距离。例如：一个 25 英寸（对角线距离）的显示器大约宽 22 英寸，如果这个显示器在距离 12 英寸远的地方，那么水平视场角应为 85°；20 英寸远时水平视场角为 58°；30 英寸远时水平视场角为 40°。这个视场角的计算公式也可以用于将相机镜头尺寸转换为视场角，例如：相机的感光元件宽 $35mm$，镜头长 $50mm$，则其视场角为 $\phi = 2 \arctan(36/(2 \cdot 50)) = 39.6^\circ$。

如果使用比人眼物理视场角更小的视场角，会减弱透视的感觉，因为观察者在场景中的视野会被放大；如果使用一个更大的视场角的话，会使得场景中的物体看起来很扭曲（例如使用相机的广角镜头），尤其是在靠近屏幕边缘的地方，会夸大近距离物体

的比例。然而，视场角越大，意味着视野越广阔，可以让观察者感觉看到的物体更大，更加令人印象深刻；其优势在于可以为用户提供更多的环境信息。

我们使用透视投影矩阵来将视锥体转换为规则观察体，这个矩阵的具体形式如下：

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4.71}$$

在使用这个矩阵对一个点进行透视投影之后，我们会获得一个新的顶点 $\mathbf{q} = (q_x, q_y, q_z, q_w)^T$，这个新顶点的 w 分量在变换之后通常并会在 0–1 之间，为了获得这个三维的投影点 $\mathbf{q}$，我们需要将该点的四个分量都除以 w 分量，即：

$$\mathbf{p} = \left(\frac{q_x}{q_w}, \frac{q_y}{q_w}, \frac{q_z}{q_w}, 1 \right) \tag{4.72}$$

矩阵 $\mathbf{P}_p$ 总会确保在平面 $z = f$ 上的点被映射到平面 $z = +1$ 上；在平面 $z = n$ 上的点被映射到平面 $z = -1$ 上。

超出近裁剪平面和远裁剪平面的物体会被裁剪，因此并不会出现在场景中。我们也可以将透视投影的远裁剪平面设置在无穷远处（$f \to \infty$），那么方程 4.71 中的矩阵将变成如下形式：

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & 1 & -2n \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4.73}$$

综上所述，在投影变换（包括正交投影和透视投影）之后，还会进行裁剪操作和齐次化操作（homogenization，坐标除以 w 分量），最终将其转换到 NDC 空间中（规范化设备坐标系，normalized device coordinate）。

为了获得可以在 OpenGL 中使用的透视变换矩阵，我们首先需要将方程 4.71 中的矩阵乘以 $\mathbf{S}(1,1,-1,1)$，这与正交投影中的操作类似，仅仅是将矩阵的第三列取反。在这个镜像操作之后，远近裁剪平面的值都会变为正数，即 $0 < n' < f'$，这也比较符合用户的直觉经验（距离越远，数字越大）。此时的 n', f' 代表了远近裁剪平面沿观察方向（负 z 轴）上的距离，下面是以供参考的 OpenGL 透视变换矩阵：

$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}. \quad (4.74)$$

还有一个更简单的矩阵设置方法，即只提供垂直视场角 ϕ 和宽高比 $a = w/h$（其中 $w : width$， $h : height$，它们代表了屏幕的分辨率），以及取反之后的远近裁剪平面 n', f'，这样我们可以将上述方程简化改写为：

$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} c/a & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \quad (4.75)$$

其中 $c = 1.0/\tan(\phi/2)$，这个矩阵过去是使用 gluPerspective() 函数生成的，它是 OpenGL 实用工具库的一部分（OpenGL Utility Library，简称 GLU），但是现在已经过时了。

有些图形 API（例如 DirectX）会将近裁剪平面映射到平面 $z = 0$ 上（而不是平面 $z = -1$ 上），同时将远裁剪平面映射到平面 $z = 1$ 上。另外，DirectX 使用了左手坐标系来定义其投影矩阵（即观察空间是一个左手系），这意味着在 DirectX 中，相机在经过观察变换之后，会看向正 z 轴，其远近裁剪平面的值都是正数。下面是 DirectX 中的透视投影矩阵：

$$\mathbf{P}_{p[0,1]} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f'}{f'-n'} & -\frac{f'n'}{f'-n'} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4.76}$$

由于 DirectX 在其文档中使用行优先来表示变换矩阵，因此通常会使用这个矩阵的转置矩阵。

透视投影带来的一个影响是，计算出的深度值并不会随着输入的 p_z 值线性变化，使用方程 4.74–4.76 中的任意一个透视变换矩阵对点 $\mathbf{p}$ 进行变换，我们可以获得一个新的顶点 $\mathbf{v}$：

$$\mathbf{v} = \mathbf{P}\mathbf{p} = \begin{pmatrix} \cdots \\ \cdots \\ dp_z + e \\ \pm p_z \end{pmatrix} \tag{4.77}$$

这里我们直接忽略了 v_x，v_y，其中的常数 d, e 取决于所使用的矩阵，如果我们使用方程 4.74 中的矩阵，则其中的常数为：

$$\begin{aligned} d &= -\left(f' + n'\right) / \left(f' - n'\right), \\ e &= -2f'n' / \left(f' - n'\right), \\ v_w &= -p_z. \end{aligned}$$

为了将这个点转换到 NDC 空间中，我们需要让点 $\mathbf{v}$ 的各个分量除以 w 分量，即：

$$z_{\text{NDC}} = \frac{dp_z + e}{-p_z} = d - \frac{e}{p_z} \tag{4.78}$$

在 OpenGL 中，z_{NDC} 的范围是 $[-1, +1]$；我们从方程 4.78 中可以看出，输出的深度值 z_{NDC} 和输入的 p_z 成反比。

例如：如果此时用户设定的 $n' = 10, f' = 110$，并且 p_z 位于沿负 z 轴 60 个单位时（即 n', f' 的中点），那么此时该点的 NDC 坐标深度为 0.833，而不是 $[-1, +1]$ 的

中点 0。图 4.21 展示了随着近裁剪平面位置（n'）发生变化时，所对应 NDC 坐标的深度变化，我们将在章节 23.7 中进行进一步讨论。

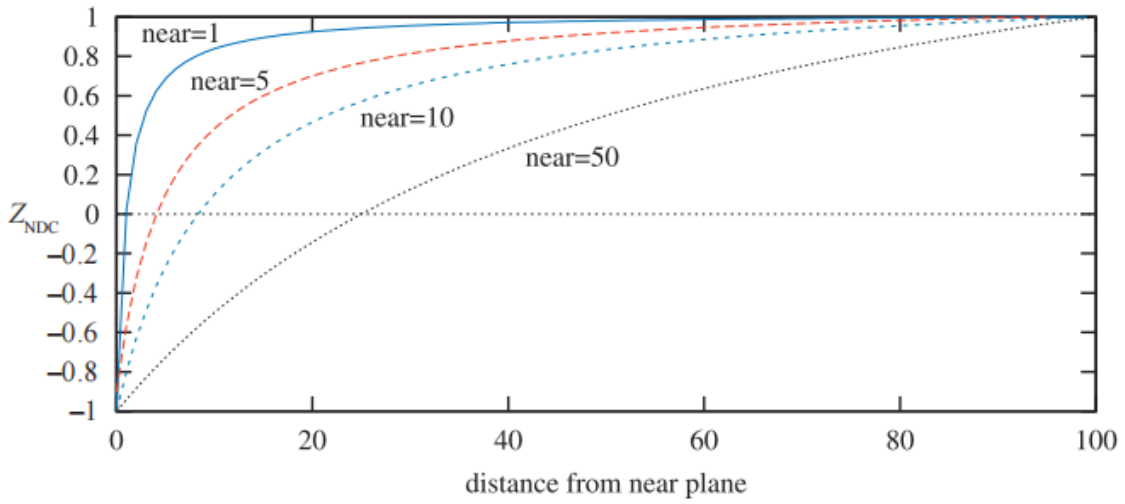


图 4.21：随着近裁剪平面位置（n'）发生变化时，所对应 NDC 坐标的深度变化。这里我们保持 $f' - n'$ 的值为常数 100。随着近裁剪平面距离原点越来越近，靠近远裁剪平面的点只占据了 NDC 深度空间的一小部分，这使得 z-buffer 在较远距离上的物体深度表示变得不那么精确。

有一些方法可以用来提高深度值的精度，其中反向 z-buffer（reversed z）是一种比较常用的方法，当使用浮点数据来表示 z-buffer 的时候，反向 z-buffer 会存储 $1 - z_{\text{NDC}}$ 的值[978]；当使用整数来表示 z-buffer 的时候，反向 z-buffer 会反向存储 z_{NDC} 的值，图 4.22 展示了不同情况下的对比结果。Reed [1472]（网站为 Depth Precision Visualized | NVIDIA Developer）通过模拟发现：使用浮点数的反向 z-buffer 可以提供最好的准确率；而反向 z-buffer 也是整数深度缓冲区（通常是 24 位整数）的首选方法。对于标准映射而言（即不使用反向 z-buffer），正如 Upchurch 和 Desbrun 所建议的那样[1803]，在变换中使用分离的投影矩阵可以降低误差率。例如：相对于使用组合矩阵 $\mathbf{Tp}, \mathbf{T} = \mathbf{PM}$ 进行变换，最好是使用分离的矩阵 $\mathbf{P}(\mathbf{Mp})$。同时，在 $[0.5, 1]$ 的范围内，由于 fp32 的尾数（定点数）为 23 位，这使得 fp32（32 位浮点数）和 int24（24 位整型）具有相近的准确性。之所以让 z_{NDC} 和 $1/p_z$ 成正比，是因为这样设计可以让硬件实现变得更加简单，同时还可以使得深度压缩变的效率更高，我们将在章节 23.7 中进一步讨论这个话题。

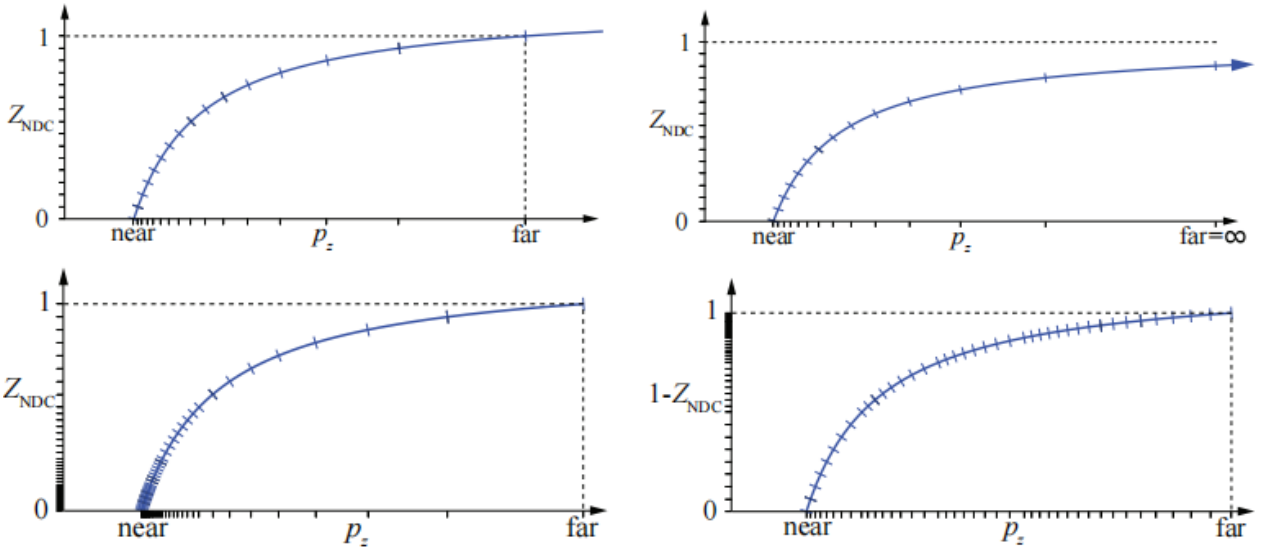


图 4.22：使用不同方法设置 DirectX 变换之后的深度缓冲，即 $z_{\text{NDC}} \in [0, +1]$ 。左上角：使用标准的整数类型深度缓冲，这里使用了 4 位整数进行演示（因此 y 轴上有 16 个标记）。右上角：将远裁剪平面设置为 ∞ ， x, y 轴只发生了微小的移动，这意味着这样做并不会损失太多精度。左下角：使用了包含 3 个指数位和 3 个尾数位的浮点类型深度缓冲。可以看到浮点数在 y 轴上的分布并不是均匀的，而在 x 轴上这个拥挤现象会更加严重。右下角：使用了反向的浮点类型深度缓冲，即存储了 $1 - z_{\text{NDC}}$ ，其分布表现良好（均匀）。

Lloyd [1063]提出，可以使用深度值的对数来提高阴影贴图（shadow map）的精度。Lauritzen 等人[991]提出可以使用前一帧的 z-buffer 来确定当前帧的最大近裁剪平面和最小远裁剪平面。Kemen 提出[881]，对于屏幕空间中的深度，可以对每个顶点使用下列的重映射：

$$\begin{array}{ll} z = w\left(\log_2\left(\max\left(10^{-6}, 1+w\right)\right) f_c - 1\right), & [\text{ OpenGL}] \\ z = w\log_2\left(\max\left(10^{-6}, 1+w\right)\right) f_c/2, & [\text{ DirectX }] \end{array} \tag{4.79}$$

其中 w 是顶点经过投影矩阵变换之后，点坐标的 w 分量， z 是顶点着色器输出的 z 值；方程中的常数 $f_c = 2/\log_2(f+1)$ ，其中 f 是远裁剪平面的值。当这个变换（方程 4.79）仅应用于顶点着色器中的，在 GPU 的光栅化阶段中，片元的深度值仍然会使用在三角形顶点的非线性深度之间进行线性插值获得。由于对数函数是一个单调函数，因此只要分段线性插值（piecewise linear interpolation）与精确的非线性插值之间，所获得的深度值差异很小，那么遮挡剔除硬件与深度压缩技术就仍然可以使用；在具有足够的曲面细分时，上述结论在大多数情况都是成立的。但是，上述方程也可以用于对每个片元进行变换，我们在顶点着色器中输出顶点 $e = 1 + w$ 的值，然后在光栅化阶段，让 GPU 在三角形上进行插值，从而获得其他片元的 e 值。然后在像素着色器中使用公式 $\log_2(e_i)\, f_c/2$ 来对片元的深度进行修改，其中 e_i 是对

三角形顶点上 e 值进行插值获得的。当 GPU 不支持浮点类型的 z-buffer，并且所渲染场景的深度很大时，这是一个很好的替代方法。

Cozzi [1605]提出可以使用多个视锥体，从而将精度提高到任何我们所期望的准确率。这个方法的核心思路是，将大的视锥体在深度方向上划分成若干个不重叠的小视锥体，这些小视锥体结合在一起就是原来的大视锥体。这些小视锥体会按照从后往前的顺序进行渲染，首先会清除颜色缓冲和深度缓冲，然后将所有需要渲染的物体，分类到与之重叠的每个小视锥体中；而对于每个小视锥体，则会生成各自的投影矩阵并清除自身的深度缓冲，然后渲染每个与小视锥体重叠的物体。

补充阅读和资源

网站 http://immersivemath.com/ila/index.html [1718]提供了一个与本章有关的交互式书籍，它鼓励你通过交互操作与调整数值的方式，来建立对于线性代数的直觉。其他的交互式学习工具和有关变换的代码库，都可以在 realtimerendering.com 上找到链接。

Farin 与 Hansford 编写了书籍《The Geometry Toolbox》[461]，它是无痛建立矩阵直觉的最好书籍之一。另一本非常有用的 3D 数学书籍是 Lengyel 所编写的《Mathematics for 3D Game Programming and Computer Graphics》[689]（这本书有中文版）。换一种角度看，Hearn 和 Baker [689]（《计算机图形学》），Marschner 和 Shirley [1129]（虎书），以及 Hughes（《计算机图形学原理及实践 3rd》）等人[785]所编写计算机图形学书籍中，也都包含了有关矩阵基础的内容。Ochiai 等人的课程介绍了矩阵的基础知识，以及矩阵指数和矩阵对数的相关内容，这些内容在计算机图形学中十分常用。《Graphics Gems》系列书籍[72, 540, 695, 902, 1344]提供了各种与变换相关的算法，其中很多算法都有在线的代码实现。Golub 和 Van Loan 所编写的《Matrix Computations》[556]是一本学习通用矩阵技术的入门书籍。更多有关骨架子空间变形、顶点混合以及形状插值的内容，可以阅读 Lewis 等人的 SIGGRAPH 论文[1037]。

Hart 等人[674]和 Hanson [663]提供了对于四元数的可视化理解（还有 3b1b 的视频以及互动视频网站）。Pletinckx [1421]和 Schlag [1566]提出了一种在四元数之间进行平滑插值的方法。Vlachos 和 Isidoro [1820]推导了对于四元数的 C^2 连续插值。沿着曲线计算一致性坐标系的问题与四元数插值相关，Dougan [374]提出了解决该问题的方案。

Alexa [28]、Lazarus 与 Verroust [1000]调查了一系列不同的变形技术。Parent 的书籍[1354]是一本学习计算机动画的良好地方。（详细的论文及书籍链接可以在 realtimerendering.com 网站上找到）.

Chapter 5 Shading Basics 着色基础

> Vincent Van Gogh——“A good picture is equivalent to a good deed.”

> 文森特·梵高——“一副好画等于一件好事。”（荷兰后印象派画家，1853—1890）

当我们在渲染三维物体的图像时，场景中的模型不仅仅需要有正确的几何形状，还应当具备我们想要的材质外观。根据应用程序的不同，这些外观具有非常广泛的范围，从真实感渲染（即物体外观几乎和真实世界中的一模一样），到各种各样的由于创造性而选择的风格化外观等。图 5.1 同时展示了这两种不同的风格。

图 5.1：第一行是使用虚幻引擎渲染的写实风景场景。第二行来自 Campo Santo 的游戏《看火人（Firewatch）》，它采用了一种插画式（illustrative art）的艺术风格。

本章节将会讨论一些有关着色的内容，这些内容同时适用于真实感渲染与风格化渲染。在第 15 章中我们将专门讨论风格化渲染，这是本书中的重要部分之一。而第 9–14 章，我们会着重讨论基于物理的通用渲染方法，这些方法常用于真实感渲染中。

5.1 着色模型

想要确定渲染对象的材质外观，第一步是选择一个着色模型（shading model），这个模型用于描述物体的颜色是如何随着表面朝向、观察方向和光照等因素的变化而变化的。

例如：我们现在将使用 Gooch 着色模型[561]的一个变体来渲染物体，Gooch 着色模型是一种非真实感渲染（non-photorealistic rendering，NPR）的着色模型，非真实感渲染是第 15 章的主要内容。Gooch 着色模型被设计用于增加技术插图（工程制图）中细节的易读性。

Gooch 着色模型的基本思想是比较表面法线和光源的位置：如果法线直接指向了光源，那么就会使用一种暖色调来给表面着色；如果法线没有指向光源，则会使用一种冷色调来给表面着色；如果法线位于这两个状态之间，则会在冷暖色调之间进行插值，当然用户也可以自定义设置表面的颜色。在这个例子中，我们还会给表面添加一个风格化的高光效果，从而使得物体表面上的某些地方变得更加闪亮。图 5.2 展示了 Gooch 着色模型的渲染结果。

着色模型中通常会包含一些用于控制外观表现的属性，确定物体材质外观的下一步，就是设定这些属性的具体数值。我们刚才提到的 Gooch 着色模型只有一个属性，即表面颜色，如图 5.2 底部所示。

图 5.2：一种风格化的着色模型，由 Gooch 着色模型和高光效果组成。顶部的图像展示了一个使用这个着色模型渲染的复杂对象（中国龙），它具有一个中性色调的表面颜色。下部的图像中展示了一系列具有不同表面颜色的材质球。[1172]

和大多数着色模型一样，相对于观察方向和光照方向的表面朝向，也会对 Gooch 着色模型产生影响。为了便于着色计算，这些方向通常都会使用归一化向量（单位向量）来进行描述，如图 5.3 所示。

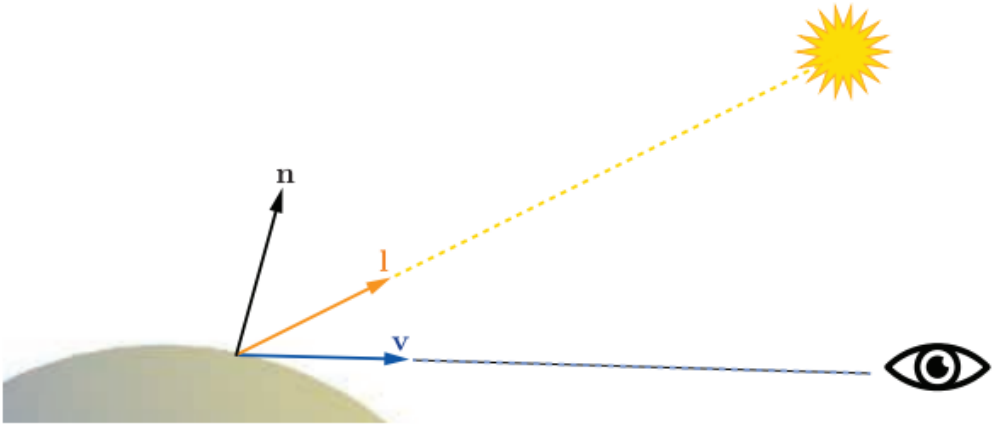


图 5.3：Gooch 着色模型中输入的单位向量：表面法线 $\mathbf{n}$，观察方向 $\mathbf{v}$ 以及光照方向 $\mathbf{l}$。这些单位向量在其他着色模型也被大量使用。

我们有了定义着色模型的全部输入参数，现在我们可以看看 Gooch 着色本身的数学定义：

$$\mathbf{c}_{\text{shaded}} = s\mathbf{c}_{\text{highlight}} + (1-s)\left(t\mathbf{c}_{\text{warm}} + (1-t)\mathbf{c}_{\text{cool}}\right). \tag{5.1}$$

在这个方程中，我们使用了以下的中间计算过程：

$$\begin{aligned}
\mathbf{c}_{\text{cool}} &= (0, 0, 0.55) + 0.25\mathbf{c}_{\text{surface}}, \\
\mathbf{c}_{\text{warm}} &= (0.3, 0.3, 0) + 0.25\mathbf{c}_{\text{surface}}, \\
\mathbf{c}_{\text{highlight}} &= (1, 1, 1), \\
t &= \frac{(\mathbf{n}\cdot\mathbf{l})+1}{2}, \\
\mathbf{r} &= 2(\mathbf{n}\cdot\mathbf{l})\mathbf{n} - \mathbf{l}, \\
s &= (100(\mathbf{r}\cdot\mathbf{v}) - 97)^{\mp}.
\end{aligned} \tag{5.2}$$

上述的一些数学表达式，在其他着色模型的计算中也会经常用到。例如 clamp（限制）操作，尤其是将数值 clamp 到 0、或 0 到 1，它在着色计算中十分常用。这里我们使用了 $x^{\mp}$ 来代表 clamp 操作，我们曾在章节 1.2 的符号约定中提到过，这里是将高光混合系数的计算结果限制到 0–1 范围内。上述计算过程中出现了三次单位向量点乘的操作，这在着色计算中也十分常见；两个向量的点乘结果，就是各自长度与它们之间夹角余弦值的乘积，而单位向量的模长为 1，因此两个单位向量点积的结果，就是这两个向量夹角的余弦值，这个余弦值可以用于衡量两个向量之间的对齐程度。由余弦组成的简单函数是一种令人愉快的数学表达式，它可以精确描述两个方向之间的夹角关系，例如着色模型中光线方向和表面法线之间的关系。

另一个常见的着色操作是在两个颜色之间，使用一个 0–1 之间的标量来进行线性插值。这个操作的基本形式为 $t\mathbf{c}_{\text{a}} + (1-t)\mathbf{c}_{\text{b}}$，其中 $t \in [0, 1]$，当 t 在 0–1 之间变化的时候，最终的结果会在 $\mathbf{c}_{\text{a}}$ 和 $\mathbf{c}_{\text{b}}$ 之间进行插值。在这里的着色模型中，包含了两次线性插值的操作，第一次是在 $\mathbf{c}_{\text{warm}}$ 和 $\mathbf{c}_{\text{cool}}$ 之间进行插值，从而获得表面颜色；第二次是在插值出来的表面颜色和高光颜色 $\mathbf{c}_{\text{highlight}}$ 之间进行插值，从而获得最终输出的颜色。线性插值是一个十分常用的操作，通常着色器都会提供内置（built-in）的函数来进行线性插值，一般叫做 lerp 或者 mix 。

表达式 $\mathbf{r} = 2(\mathbf{n}\cdot\mathbf{l})\mathbf{n} - \mathbf{l}$ 计算了光线方向 $\mathbf{l}$ 相对于表面法线 $\mathbf{n}$ 的反射向量（简单画图就可以推导出来）。尽管反射操作并不像之前的 clamp 操作和线性插值那样用的那么频繁，但是也较为常见，因此大多数着色器语言也都内置了一个 reflect 函数来计算反射向量。

通过将这些操作，与各种各样的数学公式以及着色参数以不同的方式组合在一起，我们就可以获得各种各样的着色模型，这些着色模型可以是风格化的，也可以是写实的。

5.2 光源

在上一小节中的示例着色模型中，光源对着色结果的影响是很简单的，它只提供了一个用于着色的主要方向。但是现实世界中的光照是非常复杂的，可能会有很多个光源，每个光源的大小、形状、颜色以及强度都可能会不相同；而间接光照的情况就更加复杂了，我们将会在第 9 章讨论这些话题，基于物理的真实感渲染，需要将所有这些参数都纳入考虑。

相反，风格化的着色模型可能会以许多不同的方式来计算光照，这取决于应用程序和视觉风格的需要。而对于一些高度风格化的着色模型而言，可能完全没有光照的概念，或者只是用它来提供一些简单的方向性（例如上一小节中提到的 Gooch 着色模型）。

对于着色模型而言，光照的复杂性还体现在于，如何使得表面以二元方式（表面材质只有有光和无光两种情况），来对无光或者有光环境做出相应的反应；使用这种着色模型进行渲染的物体表面，在有光照的时候会表现出一种外观，而在没有光照的时候则会表现出另一种外观。这也意味着有一些标准可以用来区分这两种情况：距离光源的距离，阴影（我们将在第 7 章讨论有关阴影的话题），表面是否背对光源（即表面法线 $\mathbf{n}$ 和光照方向 $\mathbf{l}$ 之间的夹角大于 90° ），或者以上这些因素的组合。

这里我们首先使用一个连续变化的光照强度，来替代有光还是无光的二元条件。这可以通过对完全有光和完全无光之间进行插值获得，但是这也意味着，我们能够设定的光照强度是有限的，即从 0 到 1；或者使用一个没有范围限制的光照强度，并使用其他一些方式来对着色过程产生影响。后者的一个常见做法是将着色模型分为 lit（受到光照）和 unlit（没有受到光照）两部分，并使用光强系数 k_{light} 来对有光部分进行缩放，具体形式如下：

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + k_{\text{light}}\, f_{\text{lit}}(\mathbf{l}, \mathbf{n}, \mathbf{v}) \tag{5.3}$$

方程 5.3 中的 k_{light} 只是代表了光照的强度系数，如果还想要表示光源的 RGB 颜色 $\mathbf{c}_{\text{light}}$ 的话，我们可以将其扩展成如下形式：

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \mathbf{c}_{\text{light}}\, f_{\text{lit}}(\mathbf{l}, \mathbf{n}, \mathbf{v}), \tag{5.4}$$

如果模型会接收场景中的多个光照的话，则：

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} \mathbf{c}_{\text{light}_i} f_{\text{lit}}(\mathbf{l}_i, \mathbf{n}, \mathbf{v}). \tag{5.5}$$

其中 unlit 部分 $f_{\text{unlit}}(\mathbf{n}, \mathbf{v})$ 对应“完全不受光照影响时的外观”，此时着色模型会将光照信息二元化，即分为完全有光条件和完全无光条件。根据我们所希望获得的视觉风格，以及应用程序的需要，这个 unlit 部分的着色可以有着各种不同的形式。例如当 $f_{\text{unlit}}() = (0, 0, 0)$ 的时候，会使得物体表面在不接受光照时表现为纯黑色；或者，unlit 的部分也可以表现出某种风格化的外观，例如上文中提到的 Gooch 着色模型，它对于没有接收到光照的表面，会显示一个冷色调的颜色。通常来说，着色模型中的这部分会表现出间接光照或者叫做环境光照的效果（即不是直接由光源直接照射到物体表面上所产生的光照效果），例如来自天空盒的光照，或者是光线在周围物体之间进行弹射而形成的光照。我们将在第 10 章和第 11 章讨论其他形式的光照。

我们在前文中提到，如果表面上一点的法线 $\mathbf{n}$，与光照方向 $\mathbf{l}$ 之间的夹角大于 $90°$ 的话，则说明光源发出的光线来自表面的背后，那么这个光源就不会对这个点产生任何光照效果。一般来说，光线方向与表面法线方向之间的关系，会对着色过程产生影响，而这个例子则可以认为是一般情况中的一个特例。虽然光线方向和表面法线之间的关系是基于物理的，但是这个关系也可以通过简单的几何原理推导出来；而且这个关系对于很多非基于物理的、风格化着色模型也很有用。

为了对表面进行着色，我们可以将光照对表面的影响可视化为一组平行的射线（ray），照射到表面上的射线密度代表了光照的强度。图 5.4 中展示了受光表面的横截面，光线（射线）之间的间距 d，与表面法线 $\mathbf{n}$ 和光照方向 $\mathbf{l}$ 之间夹角的余弦值成反比。也就是说，照射到表面上的光线总强度（总能量），与表面法线 $\mathbf{n}$ 和光照方向 $\mathbf{l}$ 之间夹角的余弦值成反比，我们之前已经提到了，两个单位向量之间夹角的余弦值，就等于这两个向量点乘的结果。在这里可以看到为什么我们要将光线的方向向量 $\mathbf{l}$ 定义为从表面碰撞点指向光源，而不是沿着真实的光线方向；如果我们让光线方向指向表面碰撞点的话，那么我们就要在每次点乘运算之前，将光线方向取反。

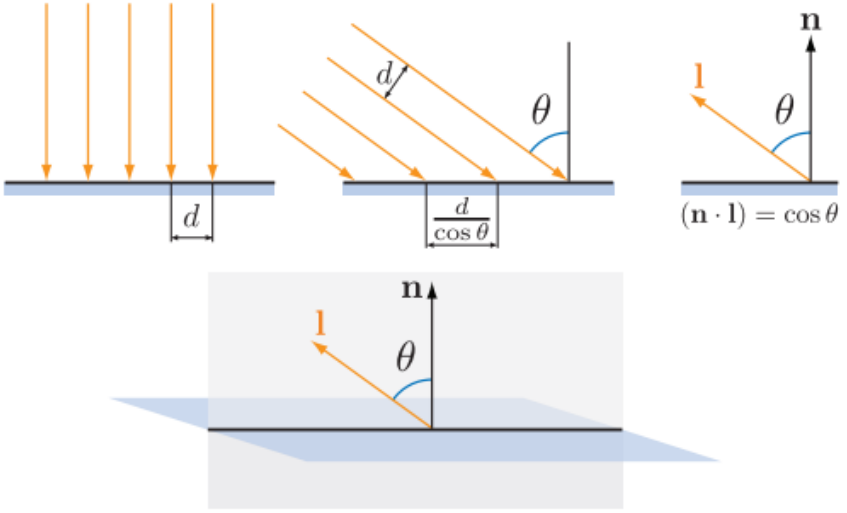


图 5.4：上面一行的图展示了光线照射到一个表面上的横截面示意图。其中左图是光线垂直照射在表面上，中图是光线以一个倾斜角度照射在表面上，右图中使用向量点乘来计算了夹角的余弦值。下面一行的图展示了这个横截面（包含光线方向和表面法线）与整个表面的相对位置。

更准确的说，当点乘结果为正的时候，光线强度才与点乘结果成正比；当点乘结果为负的时候，对应了光线来自于表面背后的情况，此时光线对于表面的着色并没有影响。所以在将光线的着色结果与点乘结果相乘之前，我们需要将点乘结果限制到 0，我们使用章节 1.2 中所提到的符号 x^+ 来表示限制到 0 的操作，它代表了如果输入的数值为负数，则直接返回 0。考虑光线方向和表面法线之间的关系对着色结果的影响，我们将获得以下方程：

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} (\mathbf{l}_i \cdot \mathbf{n})^+ \mathbf{c}_{\text{light}_i} f_{\text{lit}}(\mathbf{l}_i, \mathbf{n}, \mathbf{v}) \tag{5.6}$$

支持多光源的着色模型，通常具有类似于方程 5.5 或者是方程 5.6 的结构，前者形式更加普遍一些，后者则是基于物理的着色模型所需要的。方程 5.6 也可以用于构建风格化的着色模型，因为它有助于确保光照效果的一致性，尤其是对于那些背对光源，或者是处于阴影中的表面。但是有些着色模型确实不太适合这种结构，对于这样的模型，可以使用方程 5.5 中的结构。

对于接受光照的 lit 部分，最简单的选择就是直接为其设定一个恒定的颜色，即：

$$f_{\text{lit}}() = \mathbf{c}_{\text{surface}} \tag{5.7}$$

我们将其带入方程 5.6 中的，可以获得如下的着色模型：

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} (\mathbf{l}_i \cdot \mathbf{n})^+ \mathbf{c}_{\text{light}_i} \mathbf{c}_{\text{surface}} \tag{5.8}$$

上述方程中的 lit 部分，其实就是 Lambertian（兰伯特）着色模型，该模型由 Johann Heinrich Lambert [967]于 1760 年提出！（原文中特地使用了感叹号）。Lambertian 模型可以用于对理想漫反射表面的着色，例如完美的哑光表面等。这里我们只是简单描述了 Lambertian 模型，我们将在第 9 章对其进行更加严格的讨论。Lambertian 模型本身可以用于生成简单的着色，同时它也是许多着色模型中所用到的关键构件。

从方程 5.3–5.6 我们可以看到，一个光源可以通过两个关键参数来与着色模型进行相互作用：指向光源的光线方向 $\mathbf{l}$，以及光线的颜色 $\mathbf{c}_{\text{light}}$。有各种不同类型的光源，其主要区别在于这两个参数在场景中的变化方式。

接下来我们将讨论几种常见类型的光源，它们都有一个共同点：对于一个给定的表面位置，每个光源都只会从单一方向 $\mathbf{l}$ 照射到表面上。换句话说，从着色点的位置看向光源，光源是一个无穷小的点。现实世界中的光源严格上来说并不是这样的，但是大多数光源自身的尺寸，相对于它们与着色表面之间距离来说都非常小，因此我们将光源抽象成一个点是一个合理的近似。在章节 7.1.2 和章节 10.1 中，我们将讨论有关“面光源”的话题，面光源会从一系列方向照亮一个表面。

5.2.1 方向光

方向光（Directional light）是光源模型中最简单的一个。对于场景中的方向光，其光线方向 $\mathbf{l}$ 和颜色 $\mathbf{c}_{\text{light}}$ 都是恒定的，除了光线的颜色 $\mathbf{c}_{\text{light}}$ 可能会被场景中的阴影所减弱。真实世界中的光源都是具有确切位置的，但是在渲染中，方向光是没有位置这个概念的。方向光是一个抽象的概念，当场景到光源的距离相对于场景尺寸而言很大的时候，方向光的效果会很好。例如：一个 20 英尺外的泛光灯对桌面上的一个立体模型进行照明，我们便可以把这个泛光灯看作是一个方向光。方向光的另一个例子就是地球上被太阳所照亮的场景，但是如果我们讨论的是太阳系内的行星是如何被太阳照亮的话，那么就不太能将太阳看成一个方向光了。

方向光的概念可以在某种程度上加以扩展，即在保持光线方向 $\mathbf{l}$ 不变的同时，改变光线的颜色 $\mathbf{c}_{\text{light}}$。这通常是出于场景演出或者其他的一些创造性目的，从而将光线的效果绑定到场景中的特定部分。例如：一个区域可能会有两个嵌套（一个在另一个的

内部）的方形空间，其中会将外部空间的 $\mathbf{c}_{\text{light}}$ 设定为 $(0,0,0)$，即纯黑；将内部空间的 $\mathbf{c}_{\text{light}}$ 设定为另一个恒定的颜色，然后当角色在场景中移动的时候，会通过对这两个空间的颜色进行平滑插值，从而获得方向光的颜色。

5.2.2 精确光源

精确光源是英文是 punctual light，其中 punctual 本身有守时、精确的意思，但是这里并不是指光源会准时赴约，而是指灯光有一个确定的位置，并不是像方向光那样没有位置概念，也可以叫做定点光源。精确光源和现实世界中的光源不同，它没有面积、形状和尺寸的概念。“punctual”来自于拉丁语“punctus”，其意思是“点（point）”；这里我们使用精确光源来描述那些从单一局部位置上发出光线的光源。使用点光源来描述那些向各个方向均匀发射光线的光源。也就是说，点光源和聚光灯是两种不同形式的精确光源。对于表面上的着色点 $\mathbf{p}_0$，以及位于点 $\mathbf{p}_{\text{light}}$ 的精确光源，其光线方向 $\mathbf{l}$ 为从点 $\mathbf{p}_0$ 指向点 $\mathbf{p}_{\text{light}}$ 的向量，具体计算方程如下：

$$\mathbf{l} = \frac{\mathbf{p}_{\text{light}} - \mathbf{p}_0}{\left\|\mathbf{p}_{\text{light}} - \mathbf{p}_0\right\|}. \tag{5.9}$$

上述方程是向量归一化的一个例子：一个向量除以其模长，可以生成一个相同方向的单位向量。与我们前文所提到的着色操作类似，向量归一化在着色计算中也经常使用，大多数着色语言都会提供内置的函数来完成这个操作。但是有时候我们可能会需要这个计算中的一些中间结果，那么就要将这个归一化过程使用更加基础的操作来分步执行。对于精确光源，其光线方向的分步计算过程如下所示：

$$\begin{aligned} \mathbf{d} &= \mathbf{p}_{\text{light}} - \mathbf{p}_0, \\ r &= \sqrt{\mathbf{d} \cdot \mathbf{d}}, \\ \mathbf{l} &= \frac{\mathbf{d}}{r}. \end{aligned} \tag{5.10}$$

由于两个向量点乘的结果等于两个向量的长度再乘以其夹角的余弦值，而 0° 的余弦值是 1.0，那么一个向量点乘自身的结果就等于其长度的平方。因此，我们可以让一个向量点乘自身再开平方，从而计算其长度。

通常我们所需要的中间数据是 r，即精确光源和当前着色点之间的距离。除了使用 r 来对光线方向进行归一化之外，还可以用于计算光线颜色随着距离的衰减，我们将在之后的几个章节中进一步讨论。

点光源/泛光灯

向所有方向都均匀发射光线的精确光源被叫做点光源（point light）或者泛光灯（omni light）。对于点光源而言，光线的强度 $\mathbf{c}_{\text{light}}$ 会随着距离 r 的变化而变化，其中唯一的变化源就是上文提到的距离衰减。图 5.5 展示了这个颜色变暗的原因，这与图 5.4 中的几何推导过程相类似，都使用了余弦因子。对于一个给定的表面，从一个点光源发出的光线，其间距与光源到表面的距离成正比。

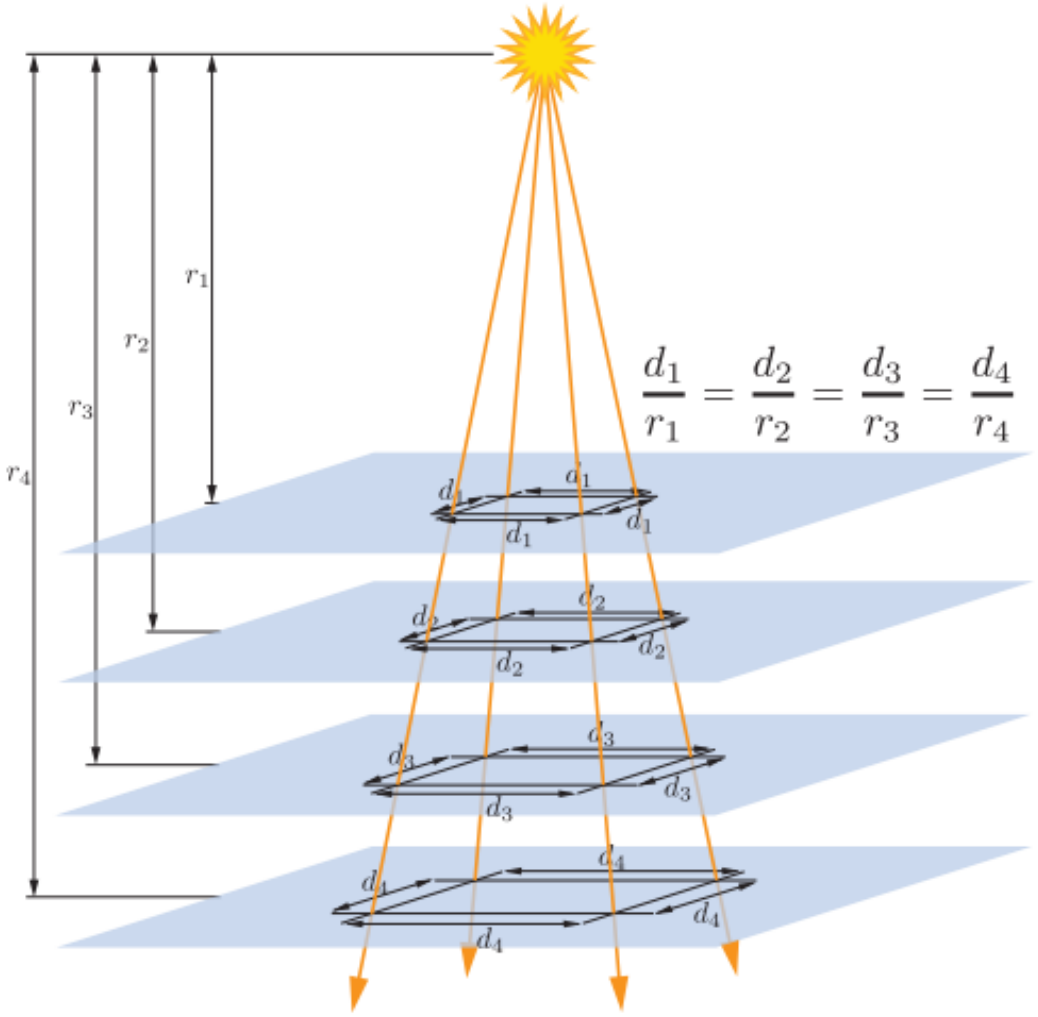


图 5.5：从点光源发出的光线，其间距随着距离 r 的增大而增大，由于这个间距会在两个维度上同时增长，因此光线的强度（即光线的颜色）与 $1/r^2$ 成正比。

与图 5.4 中的余弦因子略有不同的是，这个间距会沿着表面上的两个方向维度同时增加，因此光线的强度（即光线的颜色 $\mathbf{c}_{\text{light}}$ ）与平方距离的倒数成正比，即与 $1/r^2$ 成正比。这个特性使得我们可以使用一个单一的光线参数，来指定 $\mathbf{c}_{\text{light}}$ 在空间中发生的变化，这个参数记为 $\mathbf{c}_{\text{light}_0}$ ，即 $\mathbf{c}_{\text{light}}$ 在固定参考距离 r_0 处的值：

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0}\left(\frac{r_0}{r}\right)^2 \tag{5.11}$$

方程 5.11 通常被称作光线的平方反比衰减（inverse-square light attenuation），虽然这个方程在从技术上来说，的确正确描述了一个点光源的距离衰减，但是它仍然存在一些问题，会使得它在实际着色计算的使用中并不理想。

第一个问题发生在相对较小的距离上。当距离 r 的值趋近于 0 的时候，光线颜色 $\mathbf{c}_{\text{light}}$ 的值会迅速无界增长；当距离 r 为 0 的时候，此时方程就出现了除以 0 的情况。为了解决这个问题，一个常用的优化方式是给分母加上一个较小的数值 ϵ [861]，即：

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0} \frac{r_0^2}{r^2 + \epsilon}. \tag{5.12}$$

数值 ϵ 的取值取决于应用程序本身的设定，例如，虚幻引擎中的设定为 $\epsilon = 1cm^2$ [861]。

CryEngine [1591]和寒霜引擎[960]则使用了另一种优化方式，即将距离 r 限制到一个最小的值 r_{min}，即：

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0} \left(\frac{r_0}{\max(r, r_{\text{min}})} \right)^2. \tag{5.13}$$

与前一种方法中使用一个有点随意的数值 ϵ 不同，方程 5.13 中的 r_{min} 有一个具体的物理解释，即发光物体的物理半径。比 r_{min} 还要小的距离 r，对应了位于光源内部的着色表面，这在现实中是不可能发生的。

和第一个问题相反的是，平方反比衰减的第二个问题会发生在相对较大的距离上。这个问题与视觉效果无关，而是与性能有关。尽管光线强度会随着距离的增大而不断减小，但是它永远也不会变成 0；为了提高渲染效率，我们希望光线强度会在某个有限的距离处衰减到 0（第 20 章）。有很多不同的方法可以对平方反比方程进行修改，从而达到这一目的，在理想情况下，这个修改方法应当尽可能少地引入变化。同时，为了避免在光线影响范围的边界处出现尖锐的截断，最好是在修正过后，函数的值及其导数会在同一位置处达到 0。一种解决方案是将平方反比方程乘以具有所需属性的窗函数（window function）。虚幻引擎[861]和寒霜引擎[960]所采用的窗函数方程[860]如下：

$$f_{\text{win}}(r) = \left(1 - \left(\frac{r}{r_{\text{max}}}\right)^4\right)^{+2} \tag{5.14}$$

方程中的 $+2$ 意味着，在进行平方操作之前，需要对值进行限制，如果值为负数的话，则将其设置为 0。图 5.6 展示了三条曲线，分别是平方反比曲线，方程 5.14 中的窗函数曲线，以及二者相乘后的结果。

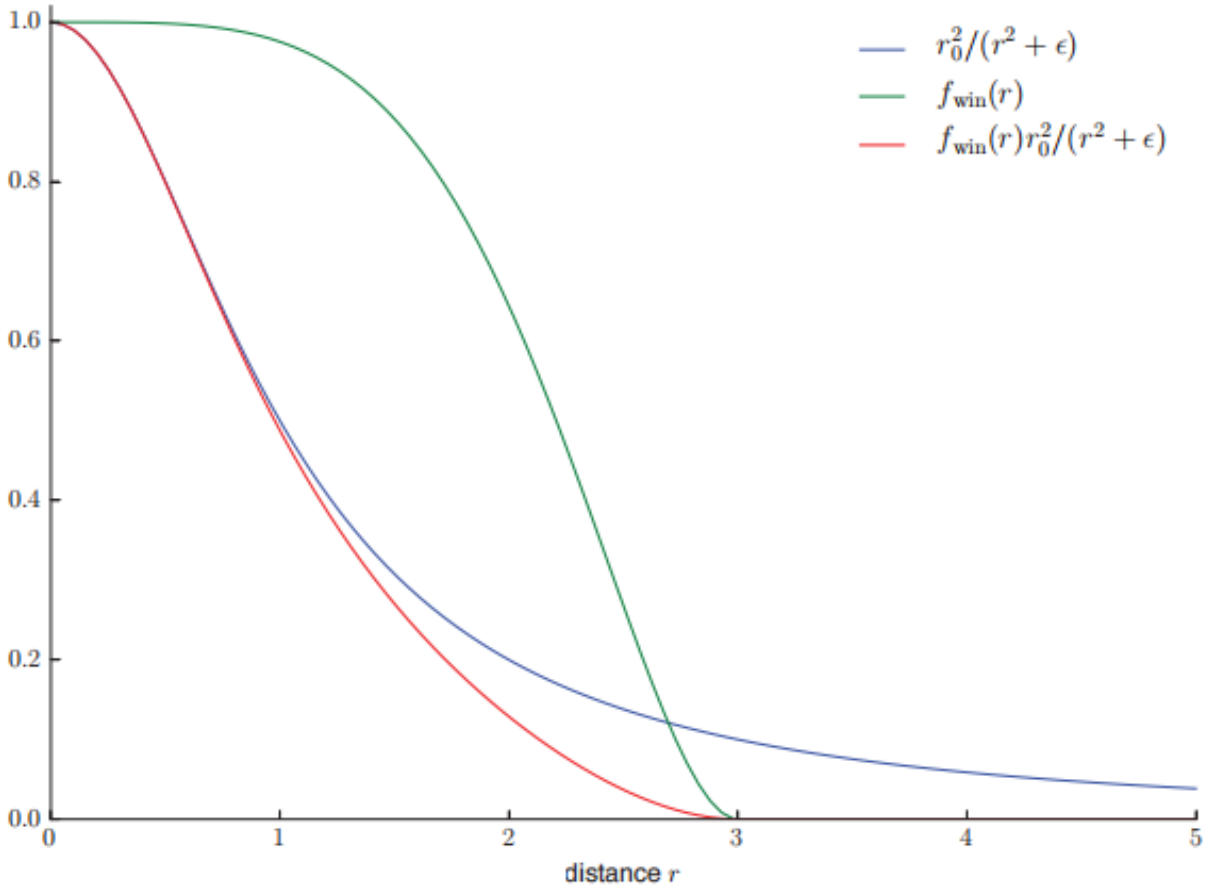


图 5.6：图中展示了三条曲线，其中蓝色曲线是平方反比曲线（使用了 ϵ 来避免分母为 0，其中 $\epsilon = 1$），绿色曲线是方程 5.14 中描述的窗函数（其中 $r_{max} = 3$），红色曲线是蓝色曲线和绿色曲线相乘后的结果。

具体使用哪种方法取决于应用程序的需要。例如：当距离衰减函数在一个较低的空间频率上进行采样时（例如光照贴图和逐顶点着色），使得 r_{max} 处的导数为 0 尤为重要。CryEngine 并没有使用光照贴图或者逐顶点着色，因此它采用了一个更加简单的修正方式，即在距离为 $0.8r_{max}$ 和 r_{max} 之间，直接使用线性衰减来替换原来的平方反比衰减[1591]。

对于某些应用而言，没有必要精确匹配平方反比曲线，因此完全可以使用其他的一些函数来进行代替。这可以有效的将方程 5.11–5.14 推广为如下形式：

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0} f_{\text{dist}}(r), \tag{5.15}$$

其中 $f_{\text{dist}}(r)$ 是一些关于距离的函数，这样的函数被称为距离衰减函数（distance falloff function）。在某些情况下，由于性能开销受到限制，因此我们需要使用一些非平方反比的衰减函数。例如：在《正当防卫 2》中，需要性能开销很低的光照计算，这就要求所使用的距离衰减函数必须简单易算，同时其过渡也要足够平滑，不能出现逐顶点的光照瑕疵（Artifacts）[1379]，它所使用的距离衰减函数如下所示：

$$f_{\text{dist}}(r) = \left(1 - \left(\frac{r}{r_{\max}}\right)^2\right)^{+2} \tag{5.16}$$

在其他情况下，衰减函数的选择也可能会取决于一些艺术风格的考虑。例如：虚幻引擎可以用于制作写实游戏和风格化游戏，对于这两种完全不同的画面风格，虚幻引擎提供了两种光线衰减的模式：一种是方程 5.12 中所描述的平方反比衰减模式；另一种是指数衰减模式，在这种模式中，可以通过调整参数，来创建各种各样的衰减曲线[1802]。《古墓丽影（2013）》的开发者使用了一个样条线编辑工具来绘制衰减曲线[953]，从而可以更好地控制曲线的形状。

聚光灯

与点光源不同的是，现实世界中几乎所有光源的光照，不仅会随着距离的改变而发生变化，同样也会随着方向的改变而发生变化，这种变化可以表示为一个方向性的衰减函数 $f_{\text{dir}}(\mathbf{l})$，它与距离衰减函数组合在一起，便完整定义了光照强度在空间中的变化：

$$\mathbf{c}_{\text{light}} = \mathbf{c}_{\text{light}_0} f_{\text{dist}}(r) f_{\text{dir}}(\mathbf{l}) \tag{5.17}$$

选择不同的 $f_{\text{dir}}(\mathbf{l})$ 可以生成不同的光照效果，其中一种很重要的效果是聚光灯（spotlight），它会将光线投射在一个圆锥体的内部。聚光灯的方向衰减函数围绕其方向向量 $\mathbf{s}$ 具有旋转对称性，因此这个衰减可以表示为向量 $\mathbf{s}$，与到达表面的反向光线向量 $-\mathbf{l}$ 之间夹角 θ_s 的函数。这里的光线向量需要进行取反，这是因为我们是在着色表面上定义光线向量 $\mathbf{l}$ 的，该向量会指向光源，而这里我们需要这个向量指向远离光源的方向。

大多数聚光灯函数都会使用包含夹角 θ_s 的余弦表达式，这是着色计算中表达角度最常用的方式（正如我们之前所看到的一些着色方程）。通常聚光灯都会有一个本影角（umbra angle）θ_u，对于 $\theta_s \geq \theta_u$ 的所有光线，会将其距离衰减函数限制为 $f_{\text{dir}}(\mathbf{l}) = 0$。这个角度可以用于剔除，其原理类似于我们之前在点光源所提到的最

大衰减距离 $r_{\max}$ 。聚光灯通常还具有一个半影角（penumbra angle） θ_p ，它定义了一个位于内部的小圆锥体，位于这个小圆锥体内部的光线，具有最大的光线强度。如图 5.7 所示：

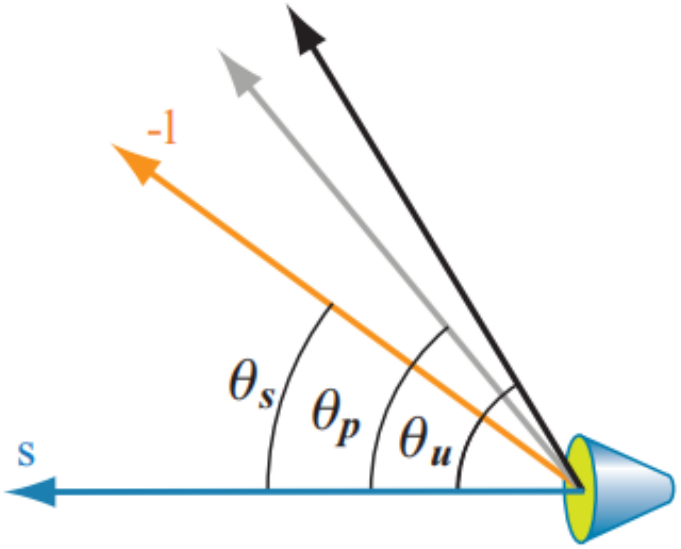


图 5.7：对于一个常见的聚光灯而言： θ_s 是光源方向 $\mathbf{s}$ 与指向表面方向 $-\mathbf{l}$ 的夹角； θ_p 代表了光源半影的范围； θ_u 代表了光源本影的范围。

有各种各样的方向衰减函数可以用于聚光灯，但是它们的形式基本都相似。例如：寒霜引擎[960]中使用的方向衰减函数是 $f_{\mathrm{dir}_F}(\mathbf{l})$ ，而 three.js（一个浏览器图形库）[218]所使用的则是 $f_{\mathrm{dir}_T}(\mathbf{l})$ ，它们的具体形式如下所示：

$$
\begin{aligned}
t &= \left(\frac{\cos\theta_s - \cos\theta_u}{\cos\theta_p - \cos\theta_u}\right)^{\mp}, \\
f_{\mathrm{dir_F}}(\mathbf{l}) &= t^2, \\
f_{\mathrm{dir_T}}(\mathbf{l}) &= \ \mathrm{smoothstep}\ (t) = t^2(3-2t).
\end{aligned}
\tag{5.18}
$$

回顾一下，我们在章节 1.2 中提到过，符号 $x^{\mp}$ 代表了会将 x 的值限制到 0–1 之间。其中的 smoothstep 函数是一个三次多项式，通常用于着色计算中的平滑插值，在大多数着色语言中，它都是一个内置的函数。

图 5.8 展示了我们到目前为止讨论过的一些光照类型。

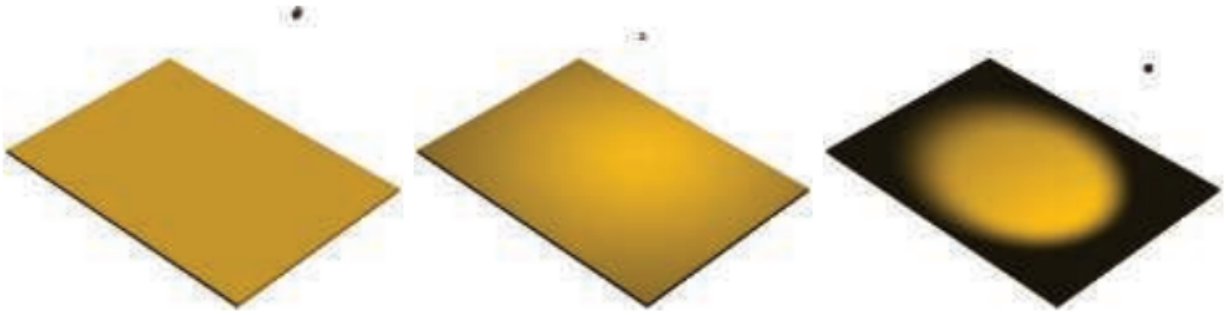


图 5.8：上图展示了三种不同的光照类型，从左到右分别是：方向光；没有距离衰减的点光源；有平滑过渡的聚光灯。请注意在第二幅表示点光源的图中，由于光线和表面之间夹角的变化，因此会在边缘的地方变暗。

其他精确光源（Other Punctual Lights）

还有很多其他的方式可以让一个精确光源的光线强度 $\mathbf{c}_{\text{light}}$ 发生变化。

$f_{\text{dir}}(\mathbf{l})$ 函数也不仅仅局限于上面所讨论的简单聚光灯衰减函数，它可以表示任何类型的方向变化，包括从真实世界光源测量而来的复杂表格数据模式。照明工程学会（Illuminating Engineering Society， IES）为此类度量定义了标准的文件格式，这些 IES 配置文件可以从许多照明设备的制造商处获得，并且已经在游戏《杀戮地带：暗影坠落》[379, 380]中进行了实际应用，除此之外也在例如虚幻[861]和寒霜[960]等游戏引擎中进行了应用。Lagarde 给出了一个有关解析和使用该文件格式的良好总结[961]。

《古墓丽影（2013）》[953]中有着一类特殊的精确光源，它可以对沿世界方向的 x, y, z 轴应用独立的衰减函数。除此之外，还可以使用自定义编辑的曲线来让光线强度随时间发生变化，例如生成一个闪烁的火炬效果。

在章节 6.9 中，我们将讨论如何使用纹理贴图来改变光线的强度和颜色。

5.2.3 其他光源类型

方向光和精确光源的主要特征是如何计算光线方向 $\mathbf{l}$，我们可以使用其他一些方法来计算光线方向，从而定义不同类型的光源。例如：除了上文中所提到的光源类型之外，《古墓丽影（2013）》中还有一种特殊的胶囊光，它使用了一个线段来作为光源，而不是一个点[953]。对于每个需要进行着色的像素，会将该像素与距离线段上最近点的方向作为光线方向 $\mathbf{l}$。

只要着色器使用了光线方向 $\mathbf{l}$ 和光线强度 $\mathbf{c}_{\text{light}}$ 来计算着色方程，那么我们可以使用任何方法来计算这些值。

到目前为止，我们所讨论的光源类型都是抽象的；而在现实世界中，光源具有大小和形状，它们会从多个方向照亮表面上的一个点。在渲染领域中，这种光源类型被称为面光源（area light），它们在实时应用程序中的应用正在稳步增加。面光源渲染技术可以分为两类：一类是模拟光线被部分遮挡，从而导致阴影边缘软化（软阴影）的方法（章节 7.1.2）；另一类是模拟面光源对表面着色影响的方法（章节 10.1）。第二类光照效果在光滑镜面上体现得最为明显，我们可以通过镜面反射，清晰得看到面光源的大小和形状。尽管方向光和精确光源已经不再像以前那样无处不再了，但是它们也不太可能被废弃。对面光源的近似方法已经有了一定的发展，其实现成本也相对便宜，因此得到了广泛的应用。同时，不断提高的 GPU 性能也允许使用比过去更加复杂的技术。

5.3 实现着色模型

为了在实际中进行应用，这些着色方程和光照方程必须要在代码中进行实现。在本小节中，我们将讨论在设计与编写此类实现时，需要考虑的一些关键注意事项，同时我们还会介绍一个简单的实现案例。

5.3.1 计算频率

当设计一个着色实现的时候，我们需要评估它的计算频率（frequency of evaluation），并对计算进行划分。首先，需要确定一个给定的计算结果在整个 draw call 中是否总为常数。尽管 GPU 的计算着色器可以用于一些开销很大的计算，但是如果这个计算结果是常数的话，那么可以让应用程序来完成（即在 CPU 上完成），然后通过统一的着色器输入传递给图形 API，这样可以大大减少重复计算量。

即使在这一类常数计算结果中，计算频率也会有很大的变化范围，我们从“只计算一次”开始讨论。最简单的情况就是着色方程中的常数子表达式，这种情况也适用于那些包含极少变化的计算因子，例如硬件配置和安装选项等。这样的着色计算可能会在着色器编译阶段就直接完成了，在这种情况下，我们甚至不需要设置一个统一的着色器输入。又或者，这些计算可以在安装和加载应用程序时，在离线预计算的过程中完成。

另一种情况是，一个着色计算的结果会在应用程序运行的过程中发生变化，但是其变化的频率很慢，因此并不需要每一帧都进行更新。例如：在一个虚拟的游戏世界中，

光照效果取决于一天中所处的时间，如果这个计算开销很大的话，则可以将其分摊到多个帧中进行计算。

其他的情况包括每帧仅执行一次的计算，例如连接观察变换和透视变换矩阵；或者每个模型一次的计算，例如对依赖于模型位置信息的光照参数进行更新；或者每个 draw call 一次的计算，例如更新模型上的材质参数。根据计算频率对统一的着色器输入进行分组，可以大大提高应用程序的效率；还可以通过最小化常量更新，来提升 GPU 的性能[1165]。

如果一个着色计算的结果在一次 draw call 中会发生变化，那么它就无法通过统一的着色器输入来将其传递给着色器。相反，它必须由第 3 章中所描述的可编程着色器阶段中的一个来进行计算，如果需要的话，还可以使用可变着色器输入，来将其传递给其他的着色器阶段。理论上而言，着色计算可以在任意一个可编程阶段中执行，每个阶段都对应着一个不同的计算频率：

- 顶点着色器（Vertex Shader）——在每个曲面细分前的顶点上进行计算。
- 壳着色器（Hull Shader）——在每个表面面片上进行计算。
- 域着色器（Domain shader）——在每个曲面细分后的顶点上进行计算。
- 几何着色器（Geometry shader）——在每个图元上进行计算。
- 像素着色器（Pixel shader）——在每个像素上进行计算。

在实际实现中，大部分着色计算都是逐像素执行的，虽然这些计算通常都是在像素着色器中完成，但是使用计算着色器来实现的例子也越来越多，我们会在第 20 章讨论几个例子；而其他的几个阶段主要用于几何操作，例如变换和变形等。为了理解为什么要这么做，我们现在对逐顶点（per-vertex）和逐像素（per-pixel）的着色计算结果进行一些比较。在一些较老的书籍中，这两种不同的着色方式有时被称为 Gouraud shading [578]和 Phong shading [1414]，但是现在这些术语已经不常使用了。这次对比测试使用了一个类似于方程 5.1 中描述的着色模型，不同之处在于，我们对其进行了修改，使其可以支持多个光源，完整的着色模型会在稍后给出，届时我们会详细介绍这个示例实现。

图 5.9 展示了在不同顶点密度的模型上，进行逐像素着色和逐顶点着色的结果。对于最底部的龙模型而言，它是一个非常密集的网格，它的顶点数量很大，逐像素着色和逐顶点着色的结果差别很小；但是对于中间的茶壶而言，顶点着色会导致可见的着色错误，例如棱角形状的高光；对于最上面的三角形平面而言，顶点着色的结果是明显错误的。这些错误的原因是由于着色方程中的一部分项（尤其是高光部分），在网格

表面上具有非线性变化的值。这使得它们并不适合在顶点着色器中进行实现，因为顶点着色器的计算结果，会在光栅化阶段被线性插值，然后再输入到像素着色器中。

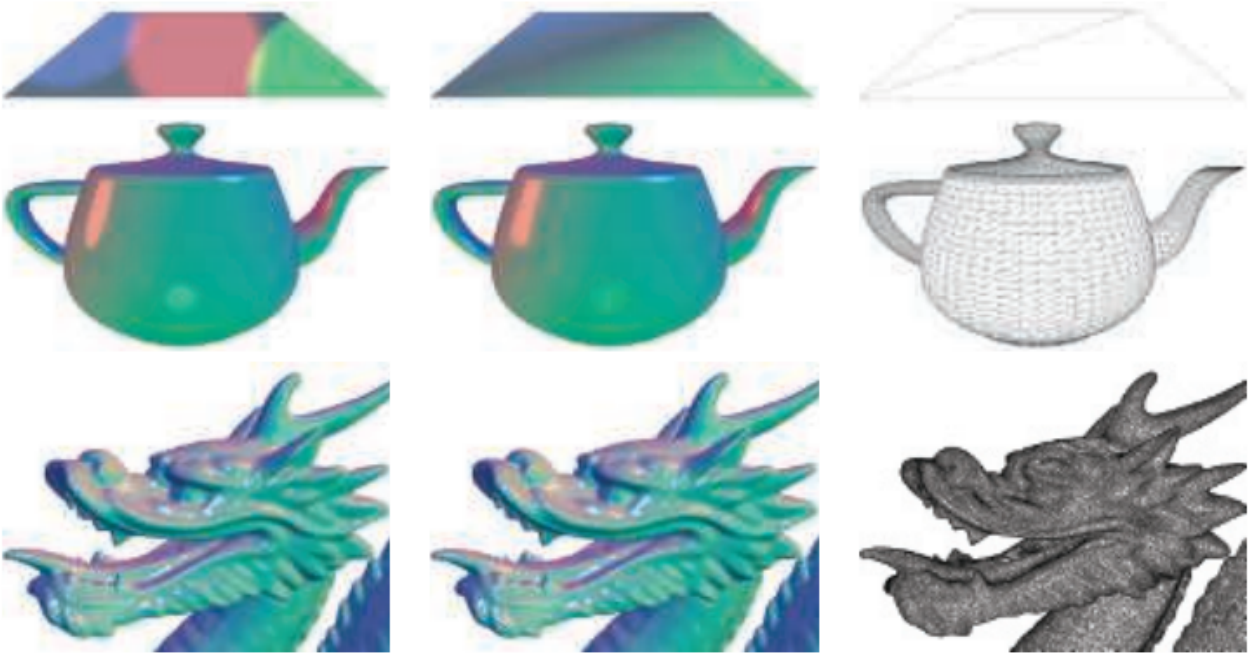


图 5.9：使用方程 5.19 中所描述的着色模型，来对比逐顶点着色和逐像素着色之间的区别，其结果分别显示在了三个不同顶点密度的模型上。最左侧的一列图片展示的是逐像素计算的结果；中间一列图片展示的是逐顶点计算的结果；最右侧一列图片展示了模型的线框渲染结果，用于显示顶点密度。[1172]

原则上来说，我们可以在像素着色器中只计算着色模型的高光（specular highlight）部分，并在顶点着色器中计算着色模型的剩余部分，这样应该就不会产生视觉瑕疵了，而且在理论上会节省一些计算量。但是在实践中，这种混合的实现方式往往并不是最优选择。首先，着色模型线性变化的部分往往是计算成本最小的部分，并且以这种方式将计算过程分离开来，也会带来额外的性能开销，例如重复的计算和额外的可变输入等，从而导致弊大于利。

正如上文中所提到的，在大多数实现中，顶点着色器负责的操作都是非着色的，例如几何变换和变形操作。几何表面上的属性在被转换为合适的坐标空间后，最终会被顶点着色器输出，并在三角形上进行线性插值，然后作为可变着色器输入，被传递到像素着色器中，这些属性通常包含了表面位置、表面法线、以及可选的表面切向量（如果需要进行法线映射的话）等。

需要注意的是，即使顶点着色器总是会输出单位长度的的表面法线，但是光栅化插值也可能会改变它们的长度，如图 5.10 左侧所示，因此需要在像素着色器中重新将法线进行归一化（将长度缩放为 1）。虽然如此，在顶点着色器中输出单位长度的法线仍然十分重要，如果顶点之间的法线长度变化很大的话（例如顶点混合的副作用），

这将会使得法线插值的结果发生倾斜，如图 5.10 的右侧所示。由于这两种影响，因此在实际的实现中，通常会对插值前和插值后的向量都进行归一化，即在顶点着色器和像素着色器中进行归一化。

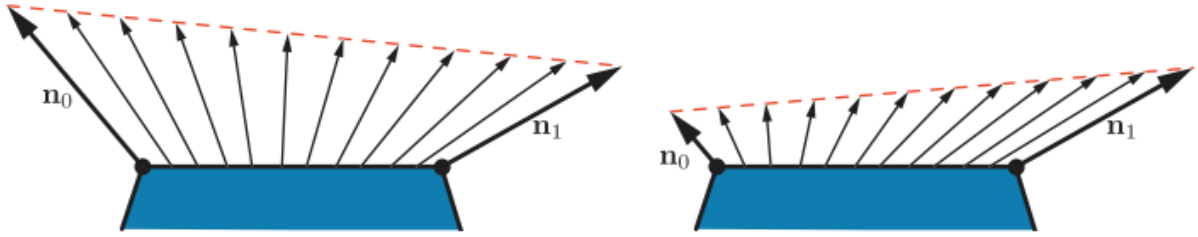


图 5.10：左图中的两个顶点法线 $\mathbf{n}_0$ 和 $\mathbf{n}_1$ 都是单位向量，我们可以看到，表面上插值而来的中间向量，其长度均小于单位长度。右图中法线 $\mathbf{n}_0$ 的长度明显小于 $\mathbf{n}_1$ ，这会导致插值结果偏向于两个法线中长度较长的那个。

与表面法线不同，指向特定位置的向量（例如观察向量和光线向量），通常并不会进行插值。相反，在像素着色器中， 会使用插值而来的表面位置来计算这些向量。除了对向量进行归一化之外（正如我们所看到的，在任何情况下，像素着色器中的向量在使用之前，都需要进行归一化），这些向量都会使用向量减法来进行计算，这样做的速度很快。如果出于某种原因，我们需要对这些向量进行插值的话，那么在插值之前，一定不要对它们进行归一化，因为这样会得到错误的结果，如图 5.11 所示。

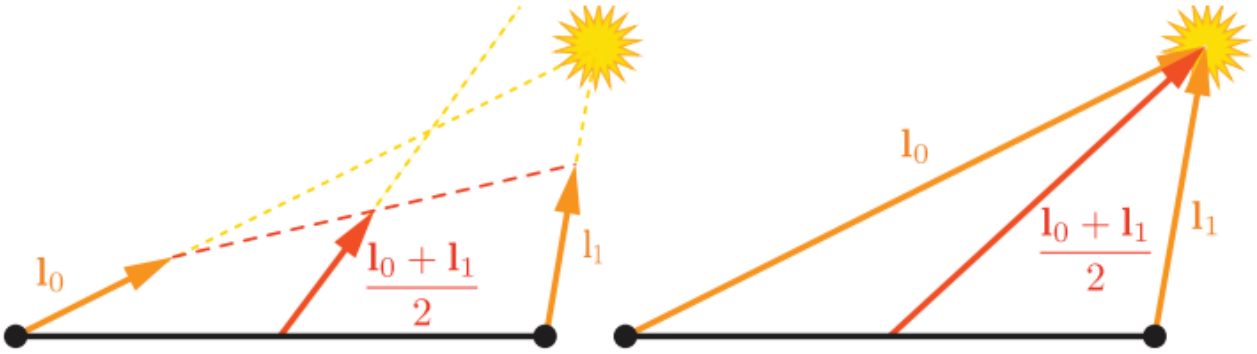


图 5.11：上图展示的是对两个光线向量进行插值。左图中，在进行插值操作之前，对光线向量进行了归一化，这会导致插值出来的光线方向是错误的；右图中，对两个没有进行归一化的光线向量进行插值，会得到正确的插值向量。

在上文中我们提到，顶点着色器会将表面的几何属性转换到“合适的坐标系”中，最后再输入到像素着色器中。而相机位置和光源位置，通常也需要由应用程序来将其转换到相同的坐标系中，然后再通过统一变量传递给像素着色器。这样做的好处是显而易见的，我们不需要在像素着色器中，再对这些向量进行大量的坐标变换，这样可以最小化像素着色器中的重复计算。但是现在有一个问题，到底哪个坐标系才是“合适的

坐标系”呢？可能的坐标系包括全局的世界空间，或者是相机的局部坐标系（观察空间），或者是更为罕见的、当前渲染模型的局部坐标系。这个坐标系的选择，通常需要将整个渲染系统作为一个整体，综合考虑系统的性能、灵活性和简单性。例如：如果渲染场景中包含大量的光源，那么可以选择全局的世界空间来避免改变光源的位置；或者最好是选择相机空间，这样可以优化与观察方向相关的像素着色器操作，并提高渲染精度（详见章节 16.6）。

尽管大多数的着色器实现（包括我们即将要讨论的示例实现），都遵循了我们上面描述的大纲，但是当然也有一些例外情况。例如：一些应用程序出于风格上的原因，选择了基于逐图元的着色计算，这样的风格通常被称作平面着色（flat shading），图 5.12 展示了两个平面着色的例子。

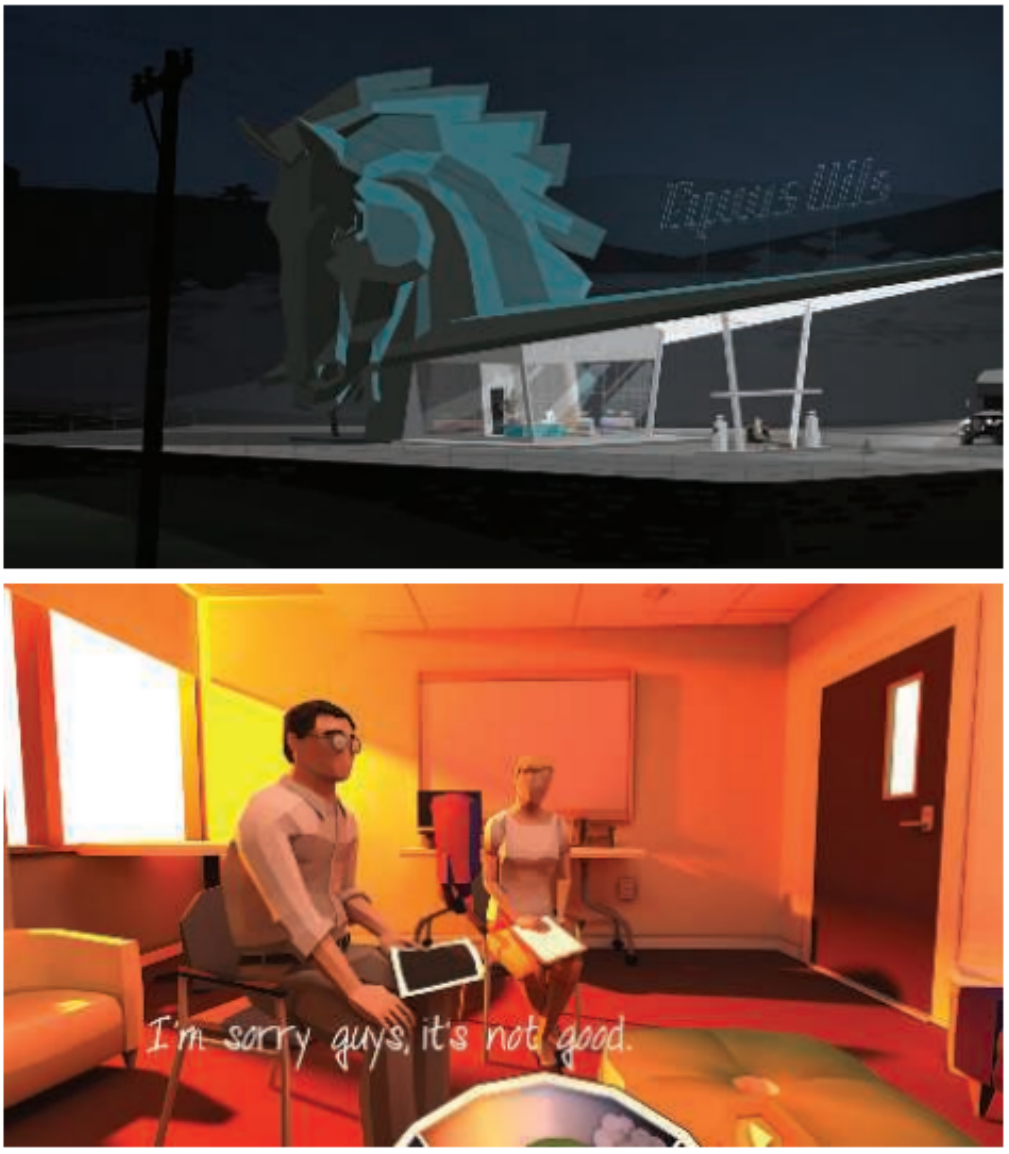


图 5.12：两个使用平面着色作为画面风格的游戏：上图来自《肯塔基零号公路》，下图来自《癌症似龙》。

原则上来说，平面着色的效果可以在几何着色器中完成，但是最近的实现通常都是使用顶点着色器来完成的。这是通过将每个图元的属性和第一个顶点绑定起来，同时禁用顶点插值来实现的。禁用顶点插值（可以对每个顶点单独执行）将会导致第一个顶点的属性会被应用到该图元中的每个像素上。

5.3.2 实现示例

这里我们将展示一个着色模型的实现示例。在上文中我们提到，我们将要实现的着色模型类似于方程 5.1 中所描述的 Gooch 着色模型，不同的是这个模型进行了扩展，可以支持多个光源，其数学描述如下：

$$\mathbf{c}_{\text{shaded}} = \frac{1}{2}\mathbf{c}_{\text{cool}} + \sum_{i=1}^{n} \left(\mathbf{l}_i \cdot \mathbf{n}\right)^{+} \mathbf{c}_{\text{light}_i} \left(s_i \mathbf{c}_{\text{highlight}} + \left(1 - s_i\right) \mathbf{c}_{\text{warm}}\right), \tag{5.19}$$

其中的一些中间变量如下所示：

$$\begin{aligned} \mathbf{c}_{\text{cool}} &= (0, 0, 0.55) + 0.25\mathbf{c}_{\text{surface}}, \\ \mathbf{c}_{\text{warm}} &= (0.3, 0.3, 0) + 0.25\mathbf{c}_{\text{surface}}, \\ \mathbf{c}_{\text{highlight}} &= (2, 2, 2), \\ \mathbf{r}_i &= 2\left(\mathbf{n} \cdot \mathbf{l}_i\right)\mathbf{n} - \mathbf{l}_i, \\ s_i &= \left(100\left(\mathbf{r}_i \cdot \mathbf{v}\right) - 97\right)^{\mp}. \end{aligned} \tag{5.20}$$

方程 5.19 的形式符合我们在方程 5.6 中描述的多光源结构，为了方便对照，这里我们回顾一下方程 5.6 的具体形式：

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}\left(\mathbf{n}, \mathbf{v}\right) + \sum_{i=1}^{n} \left(\mathbf{l}_i \cdot \mathbf{n}\right)^{+} \mathbf{c}_{\text{light}_i} f_{\text{lit}}\left(\mathbf{l}_i, \mathbf{n}, \mathbf{v}\right)$$

对于方程 5.19 而言，其中的 lit 项和 unlit 项分别是：

$$\begin{aligned} f_{\text{unlit}}\left(\mathbf{n}, \mathbf{v}\right) &= \frac{1}{2}\mathbf{c}_{\text{cool}}, \\ f_{\text{lit}}\left(\mathbf{l}_i, \mathbf{n}, \mathbf{v}\right) &= s_i \mathbf{c}_{\text{highlight}} + \left(1 - s_i\right)\mathbf{c}_{\text{warm}}, \end{aligned} \tag{5.21}$$

调整 unlit 项中的冷色贡献值，可以让结果看起来更像原来的方程。

在大多数典型的渲染应用程序中，诸如 $\mathbf{c}_{\text{surface}}$ 之类的可变材质属性都会存储在顶点数据中，或者更加常见的做法是存储在一张纹理中（第 6 章）。这里为了让这个示例的简单易懂，我们假设 $\mathbf{c}_{\text{surface}}$ 在整个模型中都是恒定的。

这个着色模型的实现将会使用着色器的动态分支功能，来所有的光源进行遍历；这是一种很简单直接的方法，它在一些比较简单的场景中可以正常工作，但是对于一些拥有很多光源的大型复杂场景而言，它的效率就很低了。有关如何处理场景中大量光源的渲染技术，将在第 20 章中进行介绍。此外，为了简单起见，这里我们只会支持点光源这一种光源类型。虽然本小节中的实现非常简单，但是它遵循了前面我们提到的最佳实践原则。

着色模型通常并不是单独实现的，而是在一个更大的渲染上下文框架（context）中实现的。这个示例是在一个简单的 WebGL 2 应用程序中实现的，它由 Tarek Sherif [1623]所开发的“Phong-shaded Cube”WebGL 2 修改而来，虽然它很简单，但是同样的原理也适用于更加复杂的渲染框架。

我们将讨论一些 GLSL 着色器代码，以及 JavaScript WebGL 调用的示例，本小节的目的并不是教授 WebGL API 的细节，而是为了展示通用的实现原则。我们将从“由内而外”的顺序来讲解整个实现过程，首先是从像素着色器开始，然后是顶点着色器，最后是应用程序端的图形 API 调用。

着色器源文件中应当包含对着色器输出输出的定义，这样的着色器代码才是正确完整的。正如我们在章节 3.3 所讨论的，使用 GLSL 的术语来进行描述，即着色器输入会分为两类：第一类是一组统一输入，这些值是由应用程序进行设置的，并且在一次 draw call 的过程中会保持不变；第二类会包含一系列的可变输入，这些值可以在着色器调用（像素着色器或者顶点着色器）之间发生变化。下面给出了一些像素着色器的可变输入（在 GLSL 中使用 in 来进行标记），以及像素着色器的输出值（使用 out 来进行标记）：

GLSL

```
in vec3 vPos ;
in vec3 vNormal ;
out vec4 outColor ;
```

这个像素着色器只有一个输出，即最终的着色颜色。而像素着色器的输入与顶点着色器的输出是相匹配的，这些参数在输入像素着色器之前，会在三角形上进行插值。这个像素着色器有两个可变输入：表面位置和表面法线，二者均位于应用程序的世界空

间坐标系中。统一输入的数量要大得多，所以简单起见，这里我们只展示两个与光源有关的定义：

```GLSL
struct Light {
    vec4 position ;
    vec4 color ;
};
uniform LightUBlock {
    Light uLights [ MAXLIGHTS ];
};
uniform uint uLightCount ;
```

由于这些光源都是点光源，因此每个光源定义中都只包含一个位置和一个颜色。这两个数据都被定义成了 vec4 类型而不是 vec3 类型，以符合 GLSL std140 数据布局标准的限制。虽然在这个例子中， std140 布局会浪费一些空间，但是这简化了确保 CPU 和 GPU 之间数据布局一致性的任务，这也是为什么我们在这个示例中使用它的原因。 Light 数组是在一个被标记为 uniform 的代码块内部定义的，这是 GLSL 的一个特性，用于将一组统一变量绑定到一个缓冲区对象中，从而加快数据传输的速度。 Light 数组的长度被定义为应用程序在一次 draw call 中所允许的最大光源数量。我们将在下文中看到，应用程序会在着色器编译之前，使用正确的数值（在这个例子中是 10）来替换着色器源代码中的字符串 MAXLIGHTS 。统一的整数 uLightCount 代表了在这次 draw call 中实际可用的光源数量。

接下来，我们来看一下像素着色器的具体代码：

```GLSL
vec3 lit ( vec3 l, vec3 n, vec3 v) {
    vec3 r_l = reflect (-l, n) ;
    float s = clamp (100.0 * dot (r_l , v) - 97.0 , 0.0 , 1.0) ;
    vec3 highlightColor = vec3 (2 ,2 ,2) ;
    return mix ( uWarmColor , highlightColor , s);
}

void main () {
    vec3 n = normalize ( vNormal );
    vec3 v = normalize ( uEyePosition .xyz - vPos );
    outColor = vec4 ( uFUnlit , 1.0) ;

    for ( uint i = 0u; i < uLightCount ; i ++) {
```

```
        vec3 l = normalize ( uLights [i]. position . xyz - vPos );
        float NdL = clamp ( dot (n, l) , 0.0 , 1.0) ;
        outColor . rgb += NdL * uLights [i]. color . rgb * lit (l,n,v)
    }
}
```

我们有一个计算 lit 项的函数定义，它会在 main() 函数中进行调用，总的来说，这是方程 5.20 和方程 5.21 的一个简单 GLSL 实现。请注意， $f_{unlit}()$ 和 c_{warm} 的值是通过统一变量输出的，由于这些变量的值在一次 draw call 中都是常数，因此可以由应用程序来计算这些值，从而节省一些 GPU 的计算周期。

这里展示的像素着色器使用了一些内置的 GLSL 函数。其中 reflect() 函数会在第二个输入向量（平面法线）所定义的平面上，将第一个输入向量进行反射；在本例中，被反射的是光线方向。由于我们希望光线向量和反射向量都是指向远离表面的方向，因此我们需要将先光线方向取反，然后再输入到 reflect() 函数中。 clamp() 函数有三个输入参数，后两个参数决定了第一个参数被限制的范围。一个特殊的限制范围是 0–1 之间（对应 HLSL 中的 saturate() 函数），这个运算的速度是很快的，对于大多数 GPU 而言，几乎没有什么开销。这也是我们在这里使用这个函数的原因，尽管我们只需要将这个参数限制到 0，因为我们已经知道了它的结果不会超过 1。函数 mix() 同样包含三个输入参数，它会根据第三个参数（位于 0–1 之间），在前两个参数之间进行线性插值；在这个例子中是根据参数 s ，在暖色和高光颜色之间进行插值。在 HLSL 中这个函数叫做 lerp() ，意思是“线性插值（linear interpolation）”。最后，函数 normalize() 会将输入向量除以这个向量的模长，即将其长度缩放为 1。

现在让我们来看看顶点着色器，由于我们已经在像素着色器中看到一些统一定义的例子了，因此这里我们不会再展示顶点着色器的统一定义，但是可变输入和输出的定义还是值得看一下的：

GLSL

```
layout ( location = 0) in vec4 position ;
layout ( location = 1) in vec4 normal ;
out vec3 vPos ;
out vec3 vNormal ;
```

请注意，我们之前提到过，顶点着色器的输出项是与像素着色器的输入项匹配的。这些输入参数还包括指定如何在顶点数组中排列数据的指令。顶点着色器的代码如下：

GLSL

```
void main () {
    vec4 worldPosition = uModel * position ;
    vPos = worldPosition.xyz ;
    vNormal = (uModel * normal ).xyz ;
    gl_Position = viewProj * worldPosition ;
}
```

这些都是顶点着色器中的常见操作，它将表面位置和法线转换到世界空间中，并将其传递给像素着色器以用于着色计算。最终，表面位置会被变换到裁剪空间中，并传递给 gl_Position ， gl_Position 是一个光栅化器所使用的、特殊的系统定义变量，它是任何顶点着色器所必须的输出。

需要注意的是，法线向量并没有在顶点着色器中进行归一化，这是因为原始网格数据中的法线长度就已经为 1 了，并且应用程序没有执行任何可能会改变法线长度的操作，例如顶点混合或者非均匀缩放等。模型变换矩阵中确实包含一个均匀缩放的比例系数，但是它会按比例改变所有法线的长度，因此并不会导致图 5.10 右侧所展示的问题。

示例中的应用程序使用 WebGL API 来进行各种渲染设置和着色器设置。每个可编程的着色器阶段都可以单独进行设置，然后它们会被绑定到一个程序对象上。以下是像素着色器设置的代码：

JavaScript

```
var fSource = document.getElementById ("fragment").text .trim () ;

var maxLights = 10;
fSource = fSource.replace(/MAXLIGHTS/g, maxLights.toString () );

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl. shaderSource ( fragmentShader , fSource );
gl. compileShader ( fragmentShader );
```

请注意代码中提到的“片元着色器（fragment shader）”，这个术语被 WebGL（以及它所基于的 OpenGL）所使用。但是正如本书前面所提到的那样，虽然“像素着色器（pixel shader）”在某些方面确实不那么精确，但是它是更加常见的用法，因此我们会在整本书中都使用像素着色器这个名称。这段代码也将 MAXLIGHTS 字符串替换为适合的值，大多数渲染框架会都执行类似这样的着色器预编译操作。

还有很多应用程序端的代码会用于设置统一输入变量、初始化顶点数组、清除、绘制等，这些代码可以都可以在示例程序[1623]中找到，并且有很多 API 指南它们进行了说明解释。本小节的目标是，了解着色器是如何被视为独立处理器的，以及它们自身独特的编程环境。因此我们有关示例着色程序的讨论到此就结束了。

5.3.3 材质系统

渲染框架很少只会实现一个单独的着色器，因此通常需要一个专门的系统，来处理应用程序中用到的各种各样的材质、着色模型和着色器。

正如前面章节中所介绍的那样，着色器是某个 GPU 可编程着色阶段中的一个程序。因此着色器是一个底层（low-level）的图形 API 资源，而不是艺术家可以直接进行交互的东西；相比之下，材质（material）对表面的视觉外观进行了封装，它才是直接面向艺术家的资源。材质有时候也会描述一些非视觉方面的内容，例如碰撞特性（物理材质）等，但是我们并不会对其进行进一步讨论，因为它超出了本书的范围。

虽然材质是通过着色器实现的，但是这并不是简单的一对一关系。在不同的渲染场景中，有时候相同的材质可能也会使用不同的着色器，一个着色器也可能会被多种材质所共享。最常见的情况就是参数化材质，在最简单形式中，材质参数化需要两种材质实体：材质模板（material template）和材质实例（material instance）。每个材质模板都描述了一类材质，并且包含了一组参数，根据参数类型的不同，可以为其指定具体的数值、颜色或者纹理贴图等。每个材质实例对应了一个材质模板，以及一组包含所有参数的特定值。一些渲染框架（例如虚幻引擎[1802]）允许构建一个更加复杂的层次材质结构，其中的材质模板可以由其他的多层次模板派生而来。

这些参数可以在运行时，通过统一输入来传递给着色器程序；或者是在着色器编译阶段，通过替换着色器中的一些值来完成。最常见的编译时参数类型就是一个布尔开关，使用这个布尔值来控制一个给定的材质特性是否会被激活。这样的开关可以由艺术家通过材质 GUI 中的一个勾选框来进行设置；也可以通过材质系统在程序上进行设置，例如：材质系统可以自动对渲染质量进行调整，来降低远处物体的着色成本，而这个修改在视觉效果上则可以忽略不记。

虽然材质参数可能与着色模型中的参数一一对应，但是情况也并非总是如此的。材质可以对给定的着色模型参数进行修改，例如将表面颜色修改成一个固定的值。又或者，可能会将多个材质参数，以及顶点插值结果和纹理值作为输入，通过一系列复杂的操作，来计算着色模型中的某一个参数。在某些情况下，诸如表面位置、表面朝向甚至是时间等参数，都会对计算产生影响。基于表面位置和表面朝向的着色，在地形材质中尤其常见，例如：可以使用高度和表面法线来控制积雪的效果，具体实现方式

是在高海拔的水平表面以及近乎水平的表面上，混合叠加一个白色。基于时间的着色在动画材质中十分常见，例如闪烁的霓虹灯标志。

材质系统最重要的任务之一，就是将各种着色器功能划分为独立的元素，并控制这些元素的组合方式。这种组合是十分有用的，例如以下几种情况：

- 将表面着色和几何处理结合在一起，例如刚体变换、顶点混合、曲面细分、实例化以及裁剪等。这些功能是独立变化的：表面着色依赖于材质，而几何处理则依赖于模型网格。因此，将这些功能分开编写，并使用材质系统对它们进行按需组合是十分方便的。
- 将表面着色和一些合并操作组合在一起，例如像素丢弃（discard）和像素混合（blending），这与移动端的 GPU 尤其相关，因为其中的混合通常是在像素着色器中执行的。通常我们都希望独立于着色材质来选择这些特殊操作。
- 将计算着色模型参数的操作，与计算着色模型本身的操作组合在一起。这样做可以在计算其他着色模型的时候，复用之前已经实现过的操作和函数。
- 将独立可选的材质特性相互组合，并与逻辑选择、着色器的剩余部分组合在一起。这种方式允许独立编写每个材质特性的实现。
- 将着色模型及其参数计算，与光源计算组合在一起：即计算每个光源在着色点上的颜色 $\mathbf{c}_{light}$ 和方向 $\mathbf{l}$。例如延迟渲染等技术（在第 20 章中进行讨论）可以改变这种组合的结构。在支持多种此类技术的渲染框架中，这增加了额外的复杂性。

如果图形 API 能提供这种着色器代码模块化的核心功能，那就太方便了。然而悲伤的是，与 CPU 代码不同，GPU 着色器并不允许在编译后再去链接代码片段，即每个着色器阶段的程序都会被编译成一个独立的单元。着色器阶段之间的分离确实提供了有限的模块化，这有点像上述清单中的第一项：将表面着色（通常在像素着色器中执行）与几何处理（通常在其他着色器阶段执行）相结合。但是这种类比是不完美的，因为每个着色器还会执行其他的操作，并且还需要处理其他类型的组合。考虑到这些限制的存在，材质系统能够实现上述这些组合类型的唯一方法，就是在源代码级别进行组合。这主要涉及字符串操作（例如连接和替换），这通常是通过类似于 C 语言风格的预处理指令来完成，例如 #include，#if 和 #define。

早期的渲染系统包含相对较少的着色器变体，并且通过都是手动编写的。这样做有一定的好处，例如：我们可以在充分了解着色器程序的基础上去，对每个变体都进行优化。但是随着变体数量的不断增加，这种方法很快就变得不切实际了。考虑到着色器

中的所有不同部分和选项时，着色器变体的数量可能会十分巨大，这也是为什么模块化和可组合性是如此重要的原因。

当设计一个用于处理着色器变体的系统时，需要解决的第一个问题就是：不同选项之间的选择，是在运行时使用动态分支来完成的，还是在编译时通过条件预处理来完成。在一些较老的 GPU 硬件上，动态分支通常是不可能的，或者是非常低效的，因此在运行时进行选择是不可取的。因此所有变体都是在编译时进行处理的，包括不同光源类型的所有可能组合[1193]。

相比之下，当前的 GPU 硬件可以很好的支持动态分支，尤其是当该分支在一次 draw call 中对所有的像素做相同处理的时候。如今很多功能变体，例如光源数量，都是在运行时处理的。但是，向着色器中添加大量功能变体会带来另一种不同的开销：寄存器计数的增加，以及占用率的降低，从而导致性能下降（详见章节 18.4.5）。因此在编译阶段处理变体仍然是很有价值的，它能够避免包含那些从不被执行的复杂逻辑。

作为一个例子，让我们想象一个支持三种不同类型光源的应用程序。其中有两个光源类型是很简单的：点光源和方向光。第三种光源则是一个广义的聚光灯，它支持表格照明模式以及其他复杂的特性，这需要使用大量的着色器代码来实现。但是这种广义的聚光灯在应用程序中使用的很少，只有大概不到 5%。在过去，为了避免动态分支，我们会为三种光源类型的（可能出现的）每种计数组合都生成一个独立的着色器变体。尽管现在可能并不需要这种方式了，但是编译两个独立的变体仍然是很有用的，其中一个用于聚光灯数量大于等于 1 的情况，另一种用于聚光灯数量恰好为 0 的情况。第二种情况的着色器代码更加简单，也更加常用（对应 95% 的使用场景），其寄存器使用量也更低（意味着可以实现更高的占用率），因此具有更好的性能表现。

现代的材质系统会同时使用运行时着色器变体和编译时着色器变体。尽管现在我们并不只在编译阶段进行处理，但是总体的复杂性和变体数量仍在不断增加，因此仍然有大量的着色器变体需要进行编译。例如：在游戏《命运：邪神降临》（命运的一个大型 DLC）的某些区域中，在一帧内使用了超过 9000 个预编译的着色器变体[1750]。可能存在的变体数量则更为巨大，例如：Unity 渲染系统中有着接近 1000 亿个可能的着色器变体。虽然只有那些实际用到的变体才会进行编译，但是必须要对着色器编译系统进行重新设计，才能处理数量如此巨大的潜在变体[1439]。

材质系统的设计者采用了不同的策略来解决这些设计目标，虽然这些策略有时候会表现出互斥的系统结构[342]，但是这些策略确实可以被整合在同一个系统中。这些策略包含以下内容：

- 代码复用——在共享文件中实现可复用的函数，使用预处理指令 #include，可以在任何需要的着色器中访问这些函数。
- 做减法——如果一个着色器中聚合了大量的功能特性，它通常会被称作为“超级着色器（ubershader or supershader）”[1170, 1784]，灵活使用编译时的预处理指令与动态分支的组合，来移除那些无用的部分，并在互斥的备选方案中进行切换。
- 做加法——将各种单位功能定义成具有输入输出连接器的节点，这些功能节点可以被组合在一起。这有点类似于代码复用策略，但是更加结构化。这些节点的组合可以通过使用文本[342]或者一个可视化编辑器来完成，后者旨在让非工程师的成员（例如技术美术）更易于编写新的材质模板[1750, 1802]。但是这种可视化方案也存在一些缺陷，即只能访问部分的着色器，例如：虚幻引擎中的材质图形编辑器，只能作用于着色模型输入参数的相关计算[1802]，如图 5.13 所示。
- 基于模板——现在有一个定义好了的接口，只要符合这个接口的定义，那么就可以接入不同的实现。这要比加法策略更加正式一点，通常会用于更大的模块中。这种接口的一个常见例子是，将着色模型参数的计算与着色模型本身的计算分离开来，例如：虚幻引擎[1802]中有着不同的“材质域（material domain）”，它包含了用于计算着色模型参数的表面域（Surface domain），以及用于计算缩放系数（这个系数用于对一个给定光源的 c_{light} 进行调整）的光照函数域（Light Function domain）。在 Unity 引擎[1437]中也存在一个类似的“表面着色器（surface shader）”结构。值得注意的是，延迟渲染技术会使用 G-buffer 来作为接口，强制要求执行一个类似的着色器结构（即统一的着色模型），我们会在第 20 章中讨论它。

图 5.13：虚幻引擎的材质编辑器。请注意右侧最长的那个节点，该节点的输入连接器对应了渲染引擎所使用的各种着色输入，包括着色模型所需的所有参数。

对于更加具体的例子，《WebGL Insights》[301]这本书中的一些章节，讨论了各种引擎是如何控制其着色器管线的。除了组合之外，现代的材质系统还有几个重要的设计考虑事项，例如：如何以最小的着色器代码重复来支持多个平台。这会产生一些额外的着色器变体，以解决平台、着色器语言以及 API 之间的性能差异和功能差异。《命运》的着色器系统[1750]是这类问题的一个代表性解决方案，它使用了一个专门的预处理层，能够使用自定义的着色器语言来进行着色器编写。这允许开发人员编写与平台无关的材质，并将其自动转换为不同的着色语言和着色实现。虚幻引擎[1802]和 Unity 引擎[1436]也都具有类似的系统。

材质系统还需要保证良好的性能，除了专门的着色器变体编译之外，材质系统还可以执行一些常见的其他优化。《命运》和虚幻引擎的着色器系统，会自动检测在一次 draw call 中结果为常数的计算（例如章节 5.3.2 示例程序中的暖色和冷色计算），并将其移动到着色器之外进行。另外一个例子是《命运》中所使用的作用域系统（scope system），它用来区分不同更新频率的常量（例如：每帧更新一次，每个光源更新一次，每个物体更新一次），并在适当的时候对每一组常量进行更新，从而减少 API 的开销。

正如我们所看到的，实现一个着色方程是很简单的，重要的是决定哪些部分是能够简化的，如何计算各种表达式的出现频率，以及用户如何能够对材质外观进行修改和控制。渲染管线最终输出的是一个颜色值和混合值，本章的剩余内容包含了有关抗锯

齿、透明度以及图像显示的部分，这些内容会详细介绍如何对这些输出的值进行修改和组合，并最终进行显示。

5.4 锯齿和抗锯齿

想象现在有一个巨大的黑色三角形，正在白色的背景上缓缓移动。由于屏幕上的网格单元被三角形所覆盖，因此其像素值的强度应该会平滑的下降。但是通常在各种基础渲染器中发生的情况是，一旦网格单元的中心被三角形所覆盖，这个单元的像素颜色就会立即从白色变为黑色。标准的 GPU 渲染器也不例外，如图 5.14 最左侧一列所示。

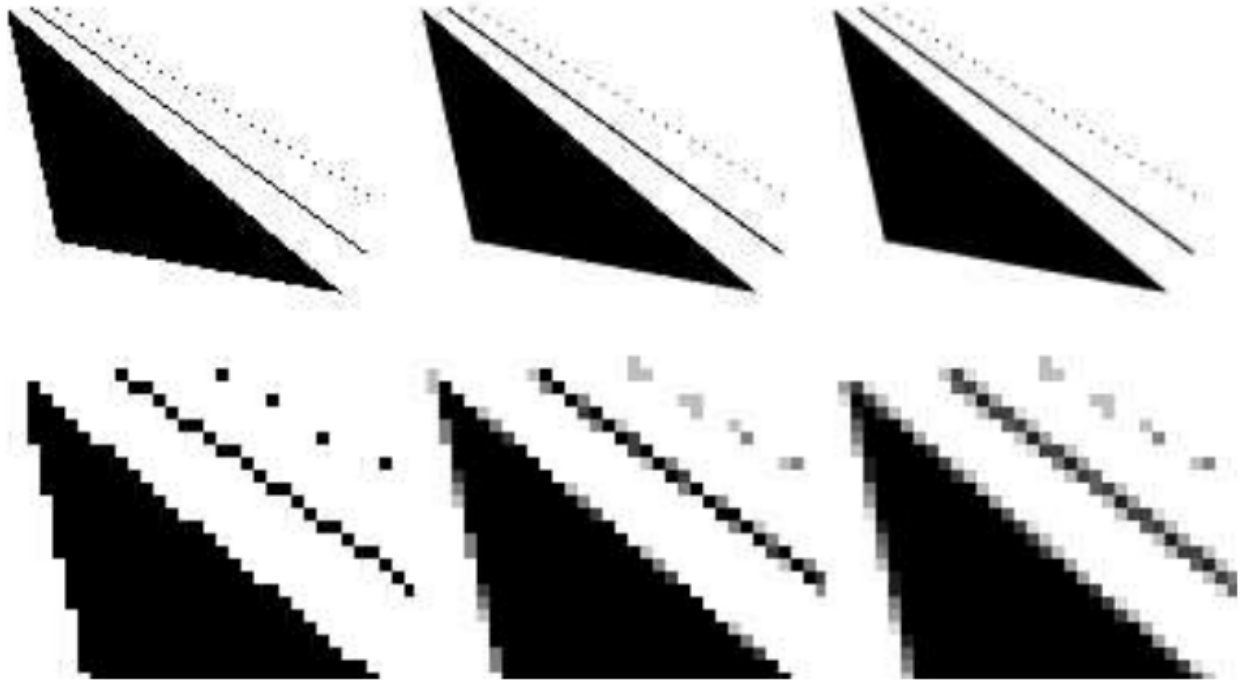


图 5.14：第一行图像展示了三个具有不同抗锯齿级别的三角形、线和点。第二行图像是上面图像放大数倍之后的结果。最左边一列图像在每个像素中仅使用了一个样本，即没有使用抗锯齿技术；中间一列图像在每个像素上使用了 4 个样本（网格模式）；最右边一列图像在每个像素上使用了 8 个样本（4×4 棋盘格模式，采样了正方形区域中的一半）。

三角形会以像素的形式显示出来，一个网格像素要么被覆盖，要么不被覆盖，绘制出来的线也有类似的问题。由于这个原因，因此三角形和线段的边界会呈现出锯齿状，这个视觉瑕疵被称作为“锯齿（the jaggies）”，当物体运动起来的时候就会变成“爬虫（the crawlies）”。更正式地说，这个问题被称作为“走样（aliasing）”，避免这个问题地技术被称作“反走样（antialiasing）”。（图像领域的走样和反走样，我们会统一翻译为锯齿和抗锯齿）

采样理论（sampling theory）和数字滤波（digital filtering）的话题内容非常巨大，大到已经足够来写另一本书了[559, 1447, 1729]。由于这是渲染中的一个关键领域，因此下面我们将会介绍有关采样和滤波的基本理论。然后我们会关注当前在实时渲染中，我们能够做些什么来缓解锯齿现象。

5.4.1 采样和滤波理论

渲染图像的过程本质上是一个采样任务。这是因为图像的生成实际上就是对三维场景进行采样的过程，从而获得图像（离散像素数组）中每个像素的颜色值。为了能够使用纹理映射技术（第 6 章），纹素（texel，texture pixel 的缩写，和像素的概念类似，是纹理中的基本单元）也必须进行重新采样，从而在不同的条件下都获得良好的效果。为了在动画中生成一系列图像，通常会以一定的时间间隔来进行采样。本小节将会介绍有关采样（sample）、重建（reconstruction）和过滤（filter）的内容。为了简单起见，大部分资料都会以一维形式呈现，这些概念可以很自然的扩展到二维图像上，用于对二维图像进行处理。

图 5.15 展示了一段连续信号如何以均匀间隔进行采样的过程，这个过程也被叫做离散化（discretized）。采样过程的目标是使用数字方式来表示信息，但是这样做会减少信息量，因此还需要对采样后的信号进行重建，这个重建的过程是通过对采样信号进行滤波实现的。

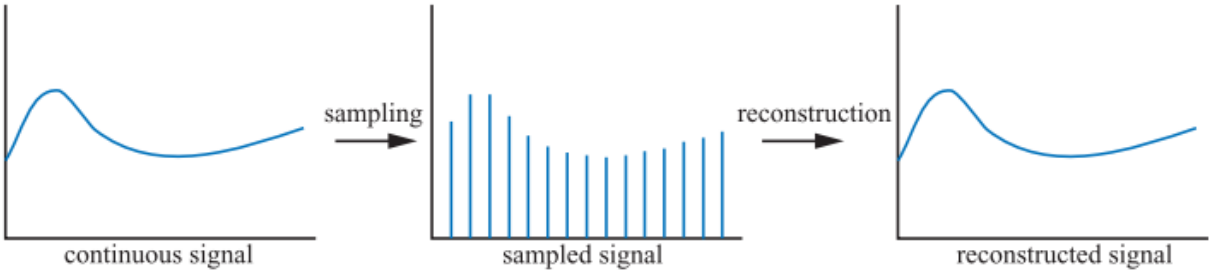


图 5.15：对一个连续信号（左图）进行采样（中图），然后通过重建方法来将其恢复成原始信号（右图）。

每当进行采样的时候，都可能会发生走样现象，这是我们不希望出现的瑕疵，因此我们需要努力对抗走样，从而生成令人愉悦的图像。在老西部片中有一个经典的走样案例，就是使用电影摄像机来拍摄旋转的马车轮子。由于车轮辐条的旋转速度要比相机记录图像的速度快得多，因此车轮可能会看起来旋转得很慢（向前旋转或者向后旋转），甚至可能会看起来根本没有发生旋转，如图 5.16 所示。之所以会出现这种现象，是因为车轮图像是在一系列时间步长内进行拍摄的，这种现象被称作时域走样（temporal aliasing）。

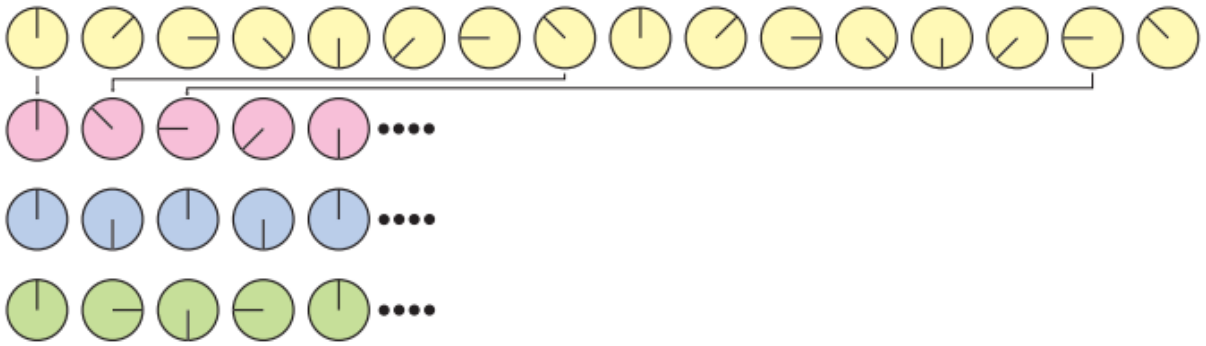


图 5.16：第一行展示了一个旋转的车轮（原始信号）。第二行对信号进行了采样，但是采样并不充分，使得车轮看起来向着相反的方向旋转，这是由于采样率（sample rate）过低而导致走样的一个例子。第三行的采样率正好是每旋转一周采样两次，因此我们无法确定车轮的旋转方向，这就是 Nyquist 极限。在第四行中，采样率大于第三行的采样率，我们可以看到车轮向着正确的方向旋转。

计算机图形学中常见的走样例子，是光栅化线段或者三角形边界上出现“锯齿（jaggies）”；以及当一个具有格子图案的纹理被缩小显示时，会出现被称为“萤火虫（firefly）”的高亮闪烁（摩尔纹）（章节 6.2.2）。

当一个信号的采样率过低时，就会出现走样现象（在光栅化游戏中，屏幕的分辨率大体决定了采样率，因此屏幕分辨率越高，走样和锯齿现象就越少）。此时采样信号的频率会比原始信号低，如图 5.17 所示。为了对一个信号进行正确的采样（即可以从采样出来的样本中，重建原始信号），采样率必须要在采样信号最大频率的两倍以上。这通常被称作采样定理（sampling theorem），对应的采样率叫做 Nyquist 率[1447]或者 Nyquist 极限，它由瑞典科学家 Harry Nyquist （1889—1976）在 1928 发现的。图 5.16 中同样展示了 Nyquist 极限的例子。事实上，该定理使用了术语“最大频率”，这意味着原始信号必须是有限频宽（band-limited）的，即没有任何频率会超过这个“最大频率”上限。换而言之，相对于相邻样本之间的间距，信号必须要足够平滑。

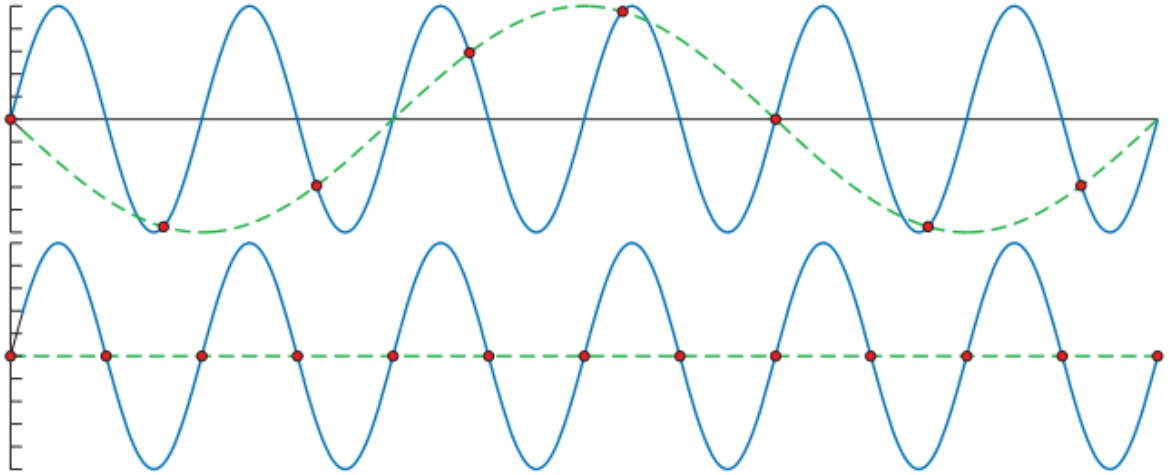


图 5.17：图中的蓝色实线代表了原始信号，红点代表了均匀间隔的采样点，绿色虚线代表了重建后的信号。第一行展示了采样率过低时的情况，重建后的信号看起来频率较低，即原始信号出现了走样。第一行的采样率是原始信号频率的两倍，重建后的信号是一条水平线。这可以证明，如果采样率稍微增加一点点，可能就可以对原始信号进行完美重建。

当使用点样本（像素点渲染）对三维场景进行采样的时候，正常情况下是不会有频宽限制的，这是因为三角形的边界，阴影的边界以及其他会产生不连续信号的现象，会导致三维场景中的频率是没有上限的[252]。此外，无论采样样本排列的有多么紧密，场景中的物体都可以被继续缩小，使得它们根本无法被采样。因此，在使用点样本来渲染场景的时候，我们是无法完全避免走样现象的，而且我们通常都会使用点采样进行采样。但是有时候，我们也可以知道一个信号是有限频宽的，其中一个例子就是在将纹理应用到曲面上的时候。因为我们此时知道了像素的采样率，同时也可以计算出纹理采样的频率，如果这二者之比小于 Nyquist 极限的话，那么我们就不需要采取什么特殊手段，直接就可以进行正确的纹理采样；但是如果这个比例太高的话，我们就需要利用各种算法，来对纹理采样进行频宽限制（章节 6.2.2）。

重建

给定一个有限频宽的采样信号后，现在我们将讨论如何从采样信号中重建原始信号。为了实现这个目的，我们必须使用一个滤波器，图 5.18 展示了三种常见的滤波器。这里需要注意的是，滤波函数的积分面积应当始终为 1，否则重建后的信号就会被放大或者缩小。

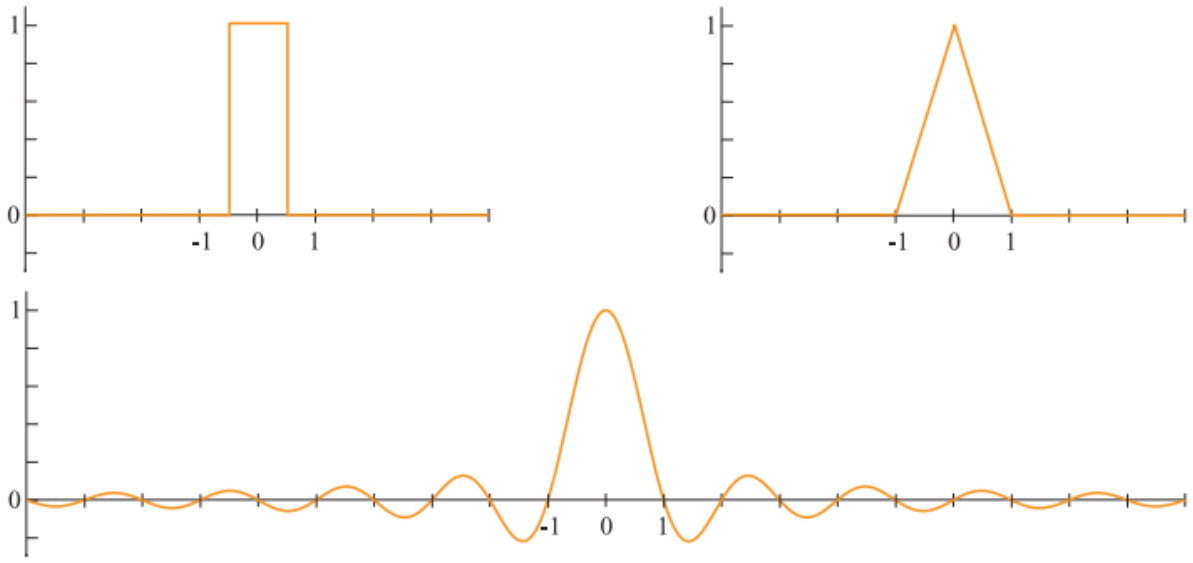


图 5.18：左上角为 box 滤波器，右上角是 tent 滤波器，下方是 sinc 滤波器（即 $\sin x/x$ ）。

在图 5.19 中，我们使用了 box 滤波器（box 意味盒子或者方框，这里就不对形状进行翻译了）来对一个采样信号进行重建。这是一种最糟糕的滤波器，因为它重建出来的信号是一段不连续的阶梯状。尽管如此，由于 box 滤波器非常简单，因此在计算机图形学中也被经常使用。如图 5.19 所示，我们将 box 滤波器放置在每个采样点上，然后对其进行缩放，使得滤波器的顶端和采样点重合（因为 box 滤波器最大值为 1），所有这些缩放和平移过后的 box 函数之和，就是右图中重建出的信号。

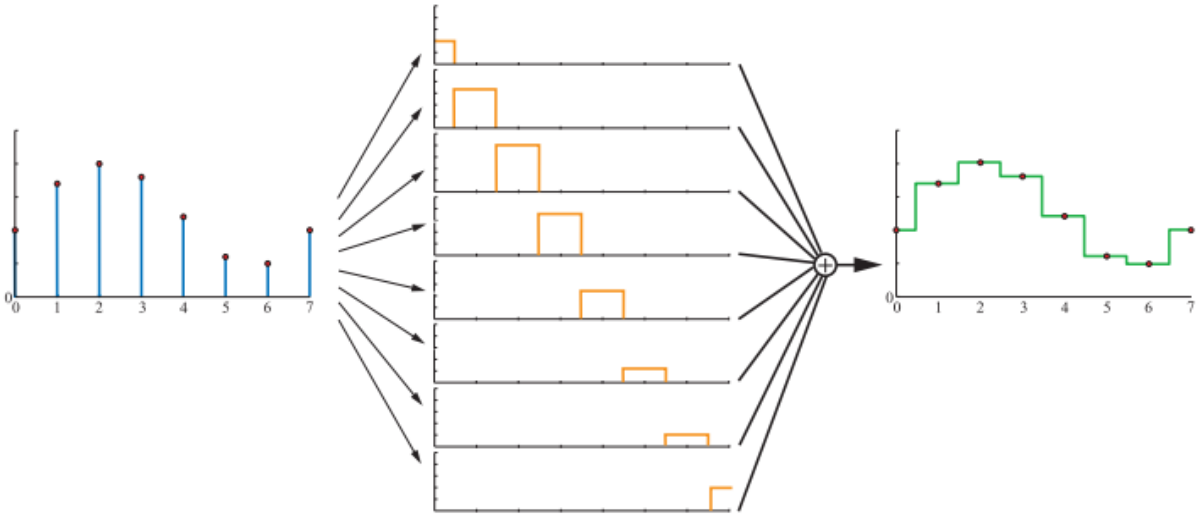


图 5.19：使用 box 滤波器对采样信号（左）进行重建。这是通过将 box 滤波器放置在每个采样点上得到的，并在 y 轴方向上对其进行缩放，使得滤波器的高度与采样点相一致。对这些变换后的 box 函数求和，即可获得重建后的信号（右）。

box 滤波器也可以使用其他任意的滤波器来进行替换，在图 5.20 中，使用了 tent 滤波器（tent 意为帐篷，tent 滤波器也叫做三角滤波器或者帐篷滤波器）来对采样信号进行重建。请注意，由于 tent 滤波器实现了相邻采样点之间的线性插值，因此它比 box 滤波器更好，因为重建之后的信号是连续的。

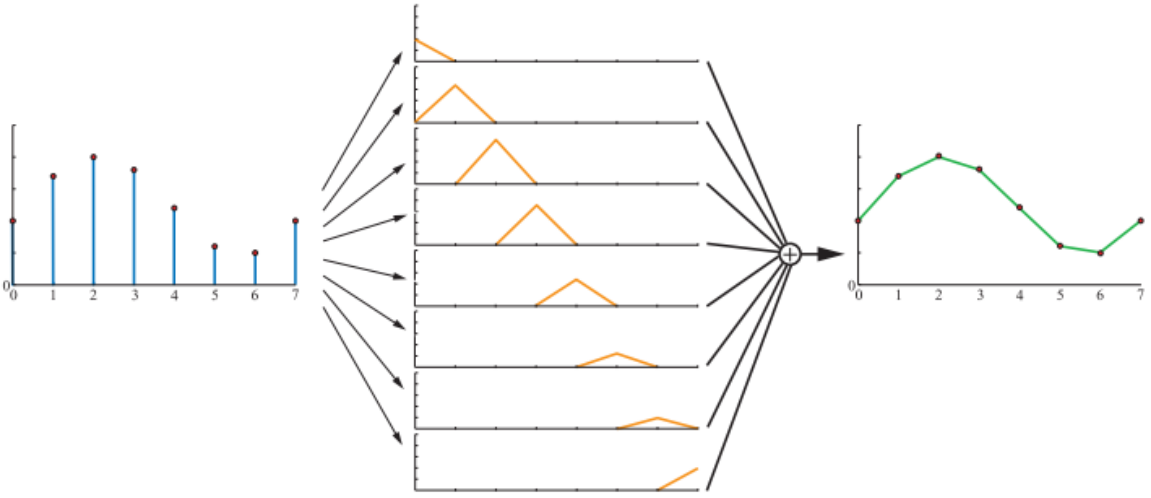


图 5.20：使用 tent 滤波器来对采样信号（左）进行重建。右图展示了重建后的连续信号。

但是，tent 滤波器仍然有不足的地方，即重建后信号的平滑性较差，信号会在采样点的位置处发生突变。这与 tent 滤波器并不是一个完美的重建滤波器有关，为了获得较为完美的重建信号，我们必须使用一个理想的低通滤波器（low-pass filter）。将一个信号进行分解，最终可以表示为若干正弦波（$\sin(2\pi f)$）的组合，其中 f 是该分量的频率。低通滤波器有一个特点，它会移除所有高于某个特定频率的分量，这个特定频率是由低通滤波器本身所决定的。从直观上来看，低通滤波器消除了信号的尖锐特征，即滤波器对信号进行了模糊处理。sinc 滤波器是一个理想的低通滤波器（如图 5.18 底部所示），其数学表达式为：

$$\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \tag{5.22}$$

傅里叶分析理论[1447]解释了为什么 sinc 滤波器是一个理想的低通滤波器。简单来说，原因如下：理想的低通滤波器是频域（frequency domain）中的 box 滤波器，当它与信号相乘的时候，会移除所有超出滤波器宽度的频率。将这个 box 滤波器从频域转换到空间域（spatial domain），便可以得到一个 sinc 滤波器。同时频域中的乘法运算也会被转换为卷积（convolution）运算，这是我们在本小节中一直在使用的概念，只是我们并没有实际描述卷积这个术语。

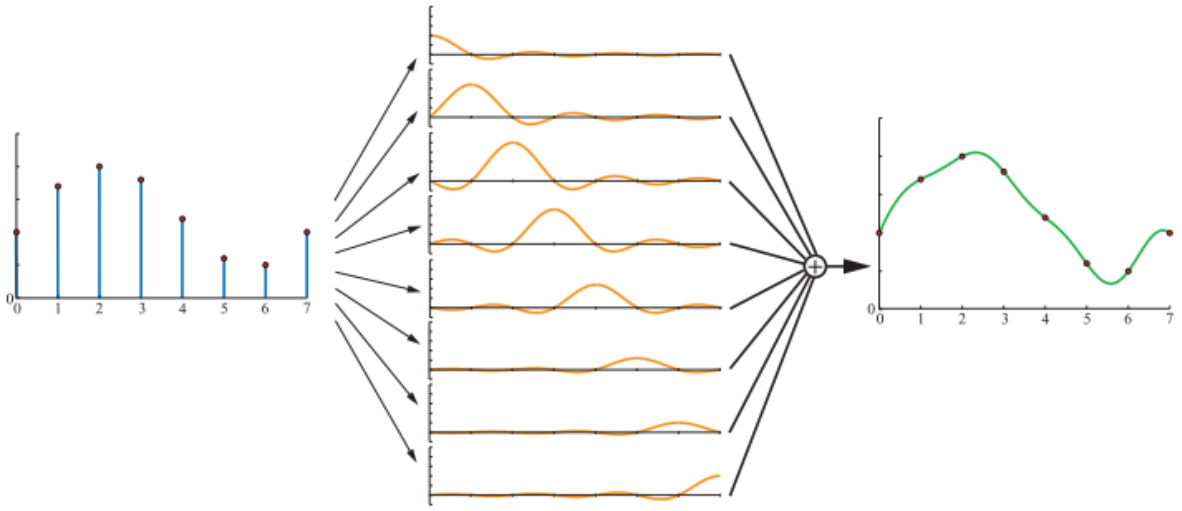


图 5.21：这里我们使用了 sinc 滤波器来重建采样信号，sinc 滤波器是一个理想的低通滤波器。

使用 sinc 滤波器进行信号重建，可以获得更加平滑的结果，如图 5.21 所示。采样过程会在信号中引入高频分量（在函数图像中则体现为突变点），低通滤波器的任务就是移除这些高频分量。事实上，sinc 滤波器消除了所有频率高于采样率 $1/2$ 的正弦波。当采样率为 1 的时候（即采样信号的最大频率必须小于 $1/2$），方程 5.22 所描述的 sinc 滤波器便是一个完美的重建滤波器。更加一般的描述是，假设采样率为 $f_s/2$（即意味着相邻样本之间的间距为 $1/f_s$），对于这种情况，完美的重建滤波器就是 sinc $(f_s x)$，它会消除所有高于 $f_s/2$ 的频率，这对于信号的重采样十分有用（下一小节的内容）。但是由于 sinc 滤波器的宽度是无限的，而且在某些区域为负，因此在实际中很少会进行使用。

在低质量的 box 滤波器，tent 滤波器和不实用的 sinc 滤波器之间，还有一个可用的中间区域，最广泛使用的滤波函数[1214, 1289, 1413, 1793]介于这两个极端之间，所有这些滤波函数都与 sinc 函数有一些类似，但是不同的是，这些滤波函数所能够影响的像素数量是有限的。与 sinc 函数类似，这些滤波函数在其部分区域上为负，但是对于应用程序而言，负值滤波器通常是不可取且不实用的，因此我们一般会使用非负波瓣的滤波器（它们通常称为高斯滤波器，因为这些滤波函数要么来自于高斯曲线，要么类似于高斯曲线）[1402]。在章节 12.1 中我们会详细讨论滤波函数以及它们的应用。

在使用任意符合要求的滤波器之后，我们便可以得到一个连续的信号；然而，在计算机图形学中，我们并不能直接显示这样的连续信号，因为最终显示在屏幕上的都是离散的点。但是我们可以使用这些重建后的连续信号，即对连续信号进行重采样，将其离散化，即放大或者缩小信号。我们将在下一小节中讨论这个话题。

重采样

重采样（resampling）用于放大或者缩小采样信号。假设原始的采样点位于整数坐标上（0，1，2……），即样本之间具有相同的单位间隔。此外，假设在重采样之后，我们希望新的样本点能够以 a 为间隔均匀分布。当 $a > 1$ 时，则缩小（minification）了采样信号（降采样，dowmsampling），即增大采样间隔，使用了更少的采样点；当 $a < 1$ 时，则放大（magnification）了采样信号（上采样，upsampling），即减小采样间隔，使用了更多的采样点。

信号放大（上采样）是这两种情况中较为简单的一种，因此我们先从它开始讨论。假设采样信号按照上一小节的方式进行重建，从直观上来看，现在的信号已经被完美重建，并且是平滑连续的。我们现在所要做的就是，以期望的间隔对重建后的信号进行重新采样，这个过程如图 5.22 所示。

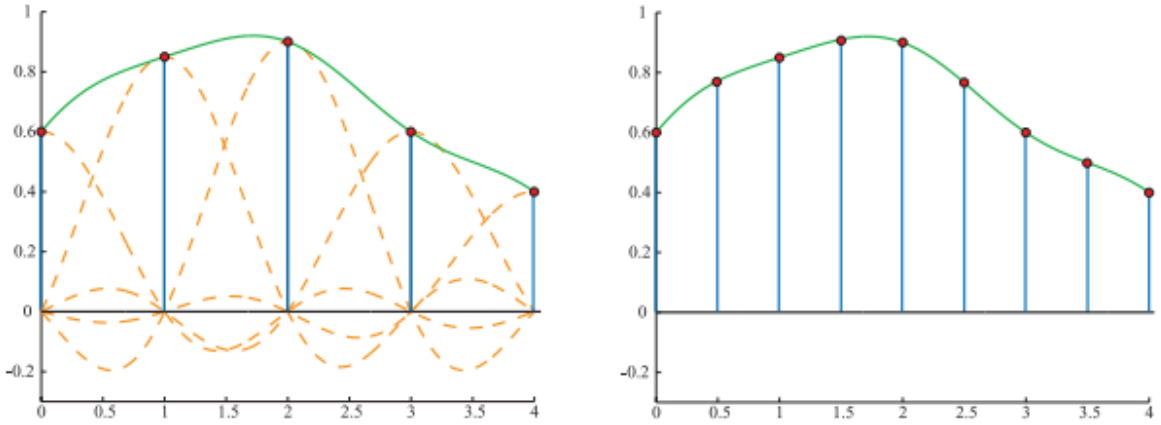


图 5.22：左图展示的是采样信号（红点）以及重建后的连续信号（绿色曲线）。右图中，重建后的信号会以两倍的采样率进行重采样，即进行了放大（上采样）。

但是当发生信号缩小（下采样）的时候，这种技术就不起作用了，因为原始信号的采样率过高，直接降低采样率导致无法避免走样。相反，应当使用一个 $sinc(x/a)$ 滤波器，来从被采样的信号中创建一个连续信号[1447, 1661]，然后再按照所需的间隔进行重采样，这个过程如图 5.23 所示。换而言之，通过在这里使用 $sinc(x/a)$ 来作为滤波器，降低了低通滤波器的采样率，从而去除了更多的高频信号。如图 5.23 所示，滤波器的重采样率降低为原始采样率的一半。将这个方法与数字图像联系起来，类似于先对图像进行模糊（去除高频内容），然后再以一个较低的分辨率对图像重新采样。

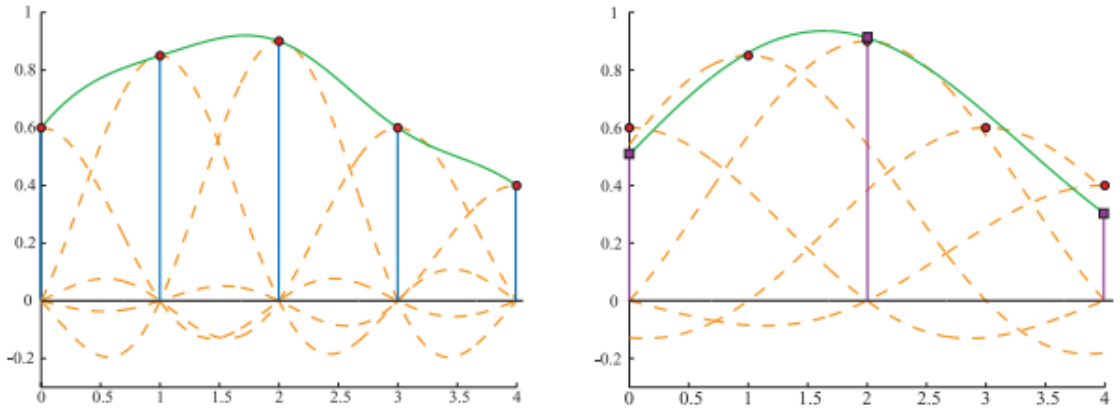


图 5.23：左图展示的是采样信号（红点）以及重建后的连续信号（绿色曲线）。右图中的采样率减半，sinc 滤波器被水平缩放为原来的两倍。

现在我们有了采样理论和滤波理论作为基础，现在可以讨论在实时渲染领域中，用于减少走样（抗锯齿）的各种算法了。

5.4.2 基于屏幕的抗锯齿

三角形的边缘如果没有被很好地采样和过滤，就会产生明显的视觉瑕疵。同理，阴影边界、镜面高光以及其他颜色剧烈变化的现象，也会导致类似的问题。本小节所讨论的算法有助于提高这些情况下的渲染质量，这些算法有一个共同点，它们都是基于屏幕的（screen based），即这些算法只会对渲染管线输出的样本进行操作处理。有一点需要注意，不存在最好的抗锯齿算法，每种算法在质量，捕捉尖锐细节（或者其他现象），处理运动物体，内存开销，GPU 开销以及速度等方面，都具有不同的优势。

在图 5.14 的黑色三角形例子中，存在的一个问题就是较低的采样率，即我们只在每个像素网格单元的中心，采样了一个单独的样本，因此我们只能了解到这个像素的中心是否被三角形所覆盖。我们可以在每个屏幕像素网格中使用更多的样本，并以某种方式将这些样本的结果混合起来，这样可以计算出更好的像素颜色。这个过程如图 5.24 所示。

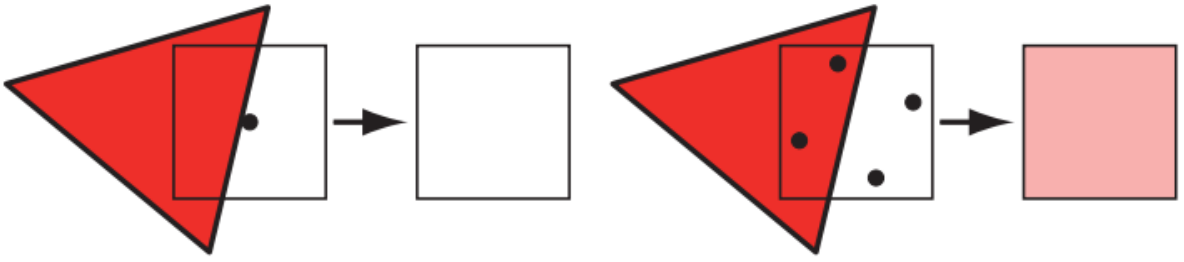


图 5.24：左图中渲染了一个红色的三角形，在像素的中心有一个样本。虽然这个像素的很大一部分都被三角形所覆盖，但是由于这个三角形并没有覆盖到像素中心的样本，因此像素最终的颜色值是还是白色的。右图中，每个像素内有四个采样点，其中有两个被三角形所覆盖，在对样本进行混合之后，最终输出了一个粉色的像素值。

基于屏幕的抗锯齿算法，其通用策略是使用一个针对屏幕的采样模式（即多个采样点），然后对这些样本进行加权求和，最终生成像素的颜色 $\mathbf{p}$，其数学表达如下所示：

$$\mathbf{p}(x,y) = \sum_{i=1}^{n} w_i \mathbf{c}(i,x,y) \tag{5.23}$$

其中 n 是一个像素内的样本数量； $\mathbf{c}(i,x,y)$ 是一个采样颜色； w_i 是一个权重，代表了该样本对像素整体颜色的贡献值，范围是 $[0,1]$ 。样本点的位置可以通过该样本在样本序列 $1,\ldots\ldots,n$ 中的序号获得，也可以使用整数标注的亚像素位置 (x,y) 来进行表示。换句话说，对于每个像素网格单元而言，其采样点的位置都是不同的，采样的模式也可以因像素而异。实时渲染系统（以及其他大部分的渲染系统）中的样本通常都是点样本，因此函数 $\mathbf{c}$ 可以看作是两个函数的组合：首先是用一个函数 $\mathbf{f}(i,n)$ 用于检索屏幕上需要采样的位置 (x_f,y_f)；然后再对屏幕上的这个位置进行采样，即精确检索这个样本点所对应的颜色值。为了计算在特定亚像素位置上的样本，需要事先选择采样方案，并对渲染管线进行配置，这通常是在逐帧（或者是逐程序）进行设置的。

抗锯齿过程中的另一个变量是代表每个样本权重的 w_i，这些权重的和应当为 1。实时渲染系统中所使用的大部分抗锯齿方法，都会给每个样本赋予相同的权重，即 $w_i = \frac{1}{n}$ 。图形硬件的默认模式是只对像素中心进行一次采样，这是上面抗锯齿方程中最简单的情况。在这种情况下，只有一个样本，该样本对应的权重为 1，采样函数 $\mathbf{f}$ 总是会返回被采样像素的中心位置。

对每个像素计算多个完整样本的抗锯齿方法被称为“超采样（supersampling）”方法。在概念上最简单的超采样方法是全屏抗锯齿（full-scene antialiasing，FSAA），通常也被称为超采样抗锯齿（supersampling antialiasing，SSAA）。这个方法会以更高的屏幕分辨率来渲染整个场景，然后再通过对相邻像素样本进行滤波（卷积），从而生成一个图像。例如：假设我们现在需要一张分辨率为 1280×1024 的图像，我们使用该方法时，首先需要离屏渲染一张分辨率为 2560×2048 的图像，然后将屏幕上每 2×2 像素区域内的颜色值进行平均，然后再显示到屏幕上；对于最终生成

的图像而言，每个像素都对应了四个采样点，并使用一个 box 滤波器进行过滤。请注意，这个过程对应了图 5.25 中的 2×2 网格采样。这个方法的开销很大，因为每个子样本都有一个 z-buffer 深度，它们都需要进行完整的着色和填充，FSAA 最大的优点就是实现起来很简单。该方法还有一个低质量的版本，即只在一个屏幕轴向上，以两倍的采样率进行采样，也被称为 1×2 超采样或者 2×1 超采样，但是通常为了简单起见，都会使用 2 的幂次分辨率和 box 滤波器。NVIDIA 的动态超分辨率（dynamic super resolution）功能是一种更加精细的超采样方法，该方法会以更高的分辨率来渲染场景，并使用包含 13 个样本的高斯滤波器来生成最终的显示图像 [1848]。

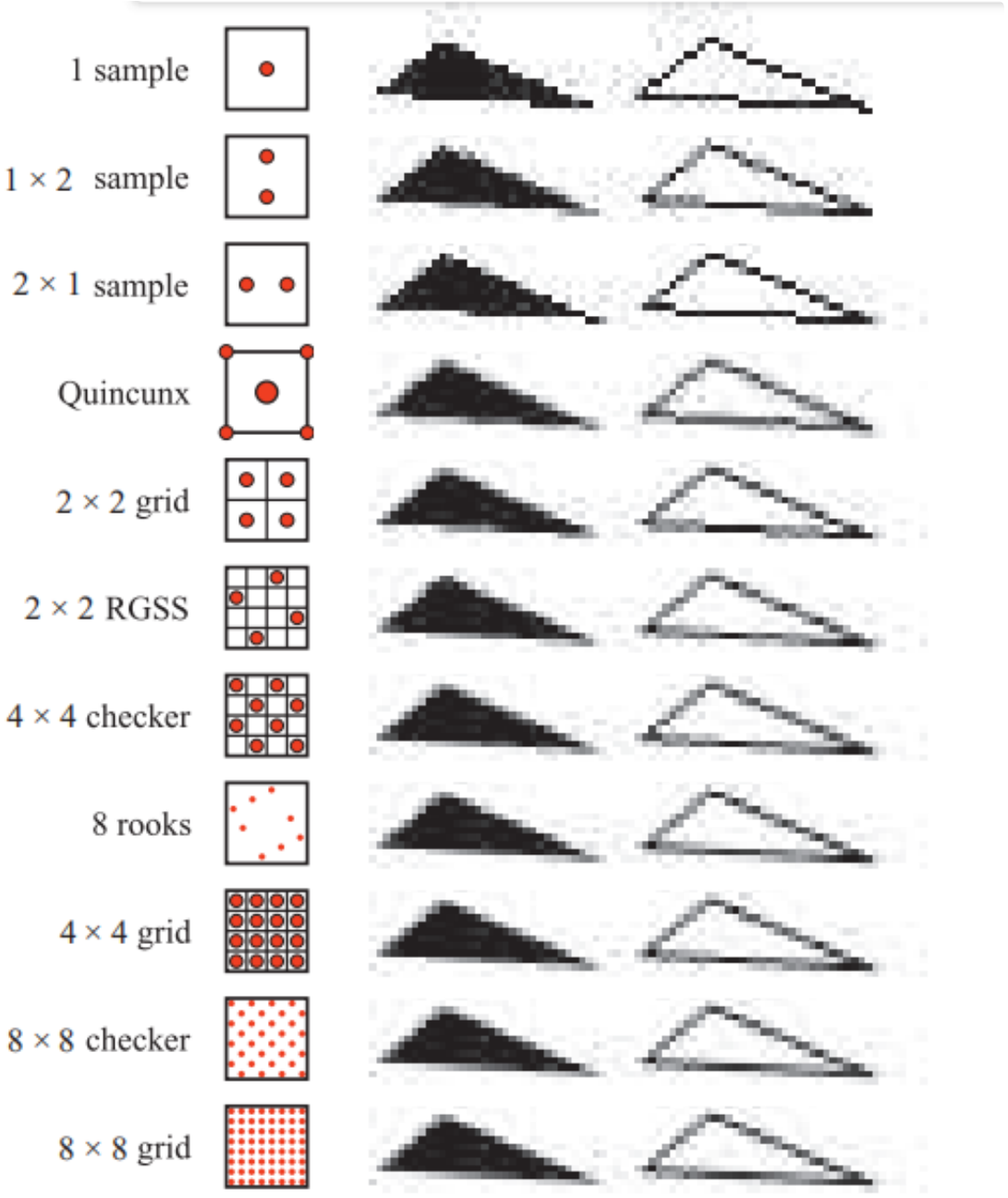


图 5.25：一些采样方案的对比，按照每个像素采样次数从少到多进行排列。其中 Quincunx 方法包含五个采样点，其中四个和其他像素共享拐角，中心采样点贡献了最终颜色的一半权重。2×2 RGSS 可以比 2×2 grid 在接近水平的边缘处，捕获更多的灰度等级。同样的，虽然 8 rooks 方法使用的样本数量更少，但是它可以比 4×4 grid 捕获更多的灰度等级。

一种与超采样相关的采样方法利用了累积缓冲区（accumulation buffffer）的思想[637, 1115]，这种方法并不会使用一个巨大的离屏缓冲区，而是使用了一个与所需图像相同分辨率的缓冲区，不同之处在于，这个缓冲区的每个通道中包含了更多的颜色比特。为了对场景进行 2×2 的采样，该方法每帧会生成四张图像，每张图像都是让视图在 x, y 方向上分别移动了半个像素距离生成的，这样四张图像便分别对应了像素网格中不同的采样点。这个方法需要每帧预渲染四张图像，并将最终处理好的图像再复制到屏幕上，这些巨大的额外成本使得该方法难以应用于实时渲染系统中。如果不考虑性能的话，这种方法可以用于生成高质量的图像，因为具体累计多少张图像是没有限制的，每个像素中都可以使用任意数量的样本，这些样本也可以位于亚像素的任意位置[1679]。在过去，累积缓冲区是一个单独的硬件模块，被 OpenGL API 直接支持，但是在 OpenGL 3.0 中被弃用了。在现代的 GPU 中，可以通过在输出缓冲区中使用更高精度的颜色格式，从而在像素着色器中实现这个累积缓冲区的概念。

当物体边缘，镜面高光以及尖锐阴影等现象导致颜色发生突变的时候，就需要使用额外的样本来进行处理。通常会对这些情况进行一些额外处理来避免锯齿，例如：让阴影变得更加柔和，让高光变得更加光滑从而避免锯齿；一些特殊的物体类型（例如电线）可以通过增加尺寸，从而保证它们在沿着其长度的方向上的每个位置，都至少占据一个像素[1384]。物体边缘的锯齿仍然是一个主要的采样问题，可以采用一些分析方法，在渲染的过程中对物体的边缘进行检测并考虑它带来的影响，但是与直接增加采样点相比，这些检测方法的开销会更大，健壮性也更差。但是，GPU 一些新特性打开了抗锯齿的新思路[327]，例如保守光栅化（conservative rasterization）和光栅器有序视图（rasterizer order view）。

诸如超采样和累积缓冲区等技术，它们生成的新样本与普通像素一样，都会独立完整地进行着色计算并维护深度信息。由于每个样本都需要通过像素着色器，因此这样做的总体收益相对较低，成本相对较高。

多重采样抗锯齿（Multisampling antialiasing，MSAA）通过只进行一次逐像素的表面着色计算，并在多个样本之间共享结果，从而大大降低了计算成本。假设每个屏幕像素的每个片元（fragment）上有四个 (x, y) 子样本位置，每个样本都具有独立的颜色信息和深度信息（z-depth），但是对于该像素的每个片元，像素着色器只会进

行一次计算。如果所有的样本位置都被这个片元所覆盖，那么则在像素的中心位置来计算这个着色样本；而如果这个片元只覆盖了部分样本，那么着色样本的选择可以进行移动，从而更好地表示所覆盖的位置。这样做有一些好处，例如可以避免对纹理边缘的着色采样。这个位置调整被叫做质心采样（centroid sampling）或者质心插值（centroid interpolation），如果 MSAA 的功能被启用的话，那么 GPU 会自动对样本的位置进行调整。质心采样避免了非三角形的问题，但是可能会导致梯度计算返回不正确的结果[530, 1041]。如图 5.26 所示。

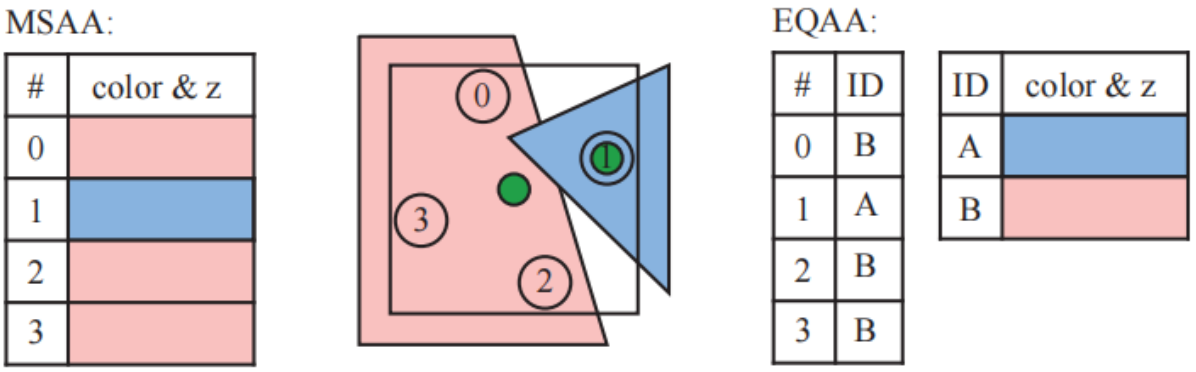


图 5.26：中间的图展示了一个像素被两个物体重叠的情况，其中红色物体覆盖了三个样本，蓝色物体覆盖了一个样本。图中的绿色点代表了像素着色器计算的位置，由于红色三角形覆盖了像素的中心，因此这个位置被用于着色器计算；而蓝色物体的像素着色器将在 1 号样本的位置上进行计算。对于 MSAA 而言，所有的四个样本位置都存储了单独的颜色信息和深度信息，右图展示了 EQAA 的 2f4x 模式，四个样本对应了四个 ID 值，这个 ID 值用于在另一个只包含颜色信息和深度信息的表格中进行检索。

MSAA 的速度要比纯超采样的方案快，因为每个片元只会进行一次着色计算，它专注于以更高的速率来对片元覆盖的像素范围进行采样，并共享计算出的着色结果。通过对采样和覆盖的进一步解耦，从而可以节省更多的内存，这反过来也可以使得对抗锯齿进行加速——对内存的访问越少，渲染的速度就越快。NVIDIA 在 2006 年引入了覆盖采样抗锯齿（coverage sampling antialiasing，CSAA），AMD 随后也提出了增强质量抗锯齿（enhanced quality antialiasing，EQAA）。这些技术以更高的采样率，同时只存储片元所覆盖的范围来进行实现，例如：EQAA 的“2f4x”模式只存储了两组颜色和深度信息，这些信息会在四个样本位置之间进行共享。并且这些信息也不再和具体的样本位置相绑定，而是存储在一张额外的表格中，四个样本仅需要各自使用一个 bit，来指定哪组颜色深度信息与该位置相关联即可，如图 5.26 所示。片元所覆盖的样本数量最终决定了该片元对像素颜色的贡献权重。如果此时存储的颜色数量超出了存储上限（两组），那么就会丢弃一个已存储的颜色信息，并将其所关联的样本标记为未知，这些被标记为未知的样本最终不会对像素颜色产生贡献[382,

383]。对于大多数场景而言，一个像素同时被三个不透明的片元所覆盖，并且这些片元还是以完全不同的方式进行着色的，这种情况发生的概率非常低，因此这个方案在实际应用中表现良好[1405]。虽然 EQAA 有着一定的性能优势，但是为了获得最高质量的画面效果，《极限竞速：地平线 2（Forza Horizon 2）》还是采用了 4× MSAA [1002]。

当场景中所有的几何体都被渲染到一个多样本缓冲区（multiple-sample buffer）之后，会进行一个解析（resolve）操作。在这个过程中，会对样本的颜色进行平均化，并最终决定像素的颜色。值得注意的是，当使用高动态范围颜色值（HDR）的时候，使用多重采样可能会导致一些问题，一般在这种情况下，为了避免出现视觉瑕疵，都会在解析之前对颜色值进行色调映射[1375]。色调映射的开销可能会很大，因此可以使用一个更简单的色调映射近似函数或者其他方法[862, 1405]。

在默认情况下，MSAA 会使用 box 滤波器进行解析。ATI 于 2007 年引入了自定义滤波器抗锯齿（custom filter antialiasing，CFAA）技术[1625]，该项技术允许使用一个更宽或者更窄的 tent（帐篷形）滤波器来进行解析，从而可以稍微扩展到周围的像素单元格上。EQAA 后续也支持了这种滤波器，CFAA 因此被取代了。在现代的 GPU 上，像素着色器或者计算着色器可以访问 MSAA 的样本，并使用任何我们所需要的滤波器来进行重建，甚至可以访问周围像素的样本。一个范围更大的滤波器可以减少锯齿和走样，但是它也会失去尖锐的细节。Pettineo 发现[1402, 1405]，使用具有 2 到 3 个像素宽度的三次平滑滤波器和 B 样条滤波器，可以获得最好的整体效果。使用其他类型的滤波器还会引入性能问题，因为在一个自定义着色器中，即使使用默认的 box 滤波器来进行解析，也需要花费很长的时间，并且使用范围更大的滤波器核（卷积核），意味着需要访问更多的样本，这也会增加开销。

NVIDIA 内置的 TXAA 同样支持使用一个更大范围（超过一个像素范围）的重建滤波器，从而获得更好的抗锯齿效果。TXAA 和比较新的 MFAA（多帧采样抗锯齿 multi-frame antialiasing）都使用了时域抗锯齿（temporal antialiasing，TAA），TAA 是一种通过之前几帧的结果来对当前帧进行优化的抗锯齿技术。从某种程度上来看，这种技术之所以成为可能，是因为它允许程序员设置每帧的 MSAA 采样模式[1406]。这种基于时域的抗锯齿技术，可以用于解决旋转车轮的走样问题，也可以提到边缘的的渲染质量。

想象一下我们可以通过“自定义”的采样模式来生成一系列图片，其中每一帧都使用了不同的像素内样本位置来进行采样。这种偏移（offset）是通过在投影矩阵上附加一个微小位移来实现的[1938]。生成和用于平均的图像数量越多，渲染结果就越好。这种使用多重偏移图像的思路也被用于时域抗锯齿的算法中，可以通过 MSAA 或者其

他方法来生成一张图像，并和之前几帧（一般是 2–4 帧）的图像进行混合[382, 836, 1405]。一般越老的图像被赋予的权重就越少[862]，如果当前的相机和场景没有发生移动的话，这种方法可能会导致闪烁的现象，因此通常只对当前帧和前一帧赋予相同的权重。由于每一帧中的样本都位于不同的像素子位置，因此这些样本的加权和，相较于比单帧加权而言，可以获得更好的边缘覆盖估计。因为上述原因，在一个渲染系统中，使用最新生成的两帧在一起进行平均，可以获得更好的结果，而且最吸引人的是，我们不需要对每一帧进行额外的处理。甚至我们可以使用时域采样来生成一个较低分辨率的图像，然后再将其放大到显示器的分辨率[1110]。除此之外，一些光照方法或者其他技术往往需要许多样本才能获得一个较好的结果，可以通过将当前结果和之前几帧进行混合，从而在一帧中降低样本的使用数量[1938]。

虽然基于时域的抗锯齿技术可以在没有增加额外采样成本的情况下，为静态场景提供抗锯齿，但是这类算法也会存在一些问题。例如：如果每一帧的权重不同的话，那么会导致静态场景中的物体出现闪烁现象（shimmer）；场景中快速移动的物体，或者是相机的快速移动，会导致画面出现鬼影（ghosting），即由于之前帧的贡献，会导致物体后面出现拖曳的痕迹。一种解决鬼影现象的方法是，只对缓慢移动的物体使用时域抗锯齿[1110]。另一个重要的方法是使用重投影（reprojection，章节 12.2）来更好地将前一帧中的物体与当前帧中的物体关联起来。在这种方案中，物体会将生成的运动向量（motion vector）存储在“速度缓冲区（velocity buffer，章节 12.5）”中。这些运动向量用于将前一帧和当前帧关联起来，即从当前的像素位置减去对应的向量，从而找到前一帧中该表面位置所对应的彩色像素。当然，在当前帧中不太可能成为表面一部分的样本将会被丢弃[1912]。由于这类基于时域的抗锯齿算法，并不会引入额外的样本，并且开销相对较小，因此近年来人们对这类算法产生了强烈的兴趣和广泛的使用。其中一部分的人关注时域抗锯齿的原因，是因为延迟渲染技术（章节 20.1）并不兼容 MSAA 和其他的多重采样抗锯齿技术[1486]。时域抗锯齿的实现方法也各不相同，并且根据应用程序的内容和目标的不同，已经有一系列用于避免瑕疵和改善质量的技术被开发出来了[836, 1154, 1405, 1533, 1938]。例如：Wihlidal [1885]展示了如何将 EQAA、时域抗锯齿和各种其他采用棋盘格采样模式的滤波技术组合在一起，在保证渲染质量的同时，降低像素着色器调用的次数。Iglesias-Guitian 等人[796]总结了之前的工作，并提出了了他们自己的方案，使用像素的历史信息并对其进行预测从而最小化滤波瑕疵。Patney 等人[1357]扩展了 Karis 和 Lottes 在虚幻 4 [862]中使用的 TAA，将其用于虚拟现实（VR）应用，并增加了可变大小的采样以及对眼睛的运动补偿（章节 21.3.2）。

采样模式

高效的采样模式（sampling pattern）是减少瑕疵和时间开销等方面的关键因素。Naiman 指出[1257]，在水平边缘与垂直边缘的附近的锯齿，对人类的视觉影响最大，其次便是倾角接近 45 度的边缘。旋转网格超采样（rotated grid supersampling，RGSS）使用了一个旋转后的正方形采样模式来进行采样，可以在像素内提供更多的水平分辨率和垂直分辨率，图 5.25 展示了这个采样模式的一个例子。

RGSS 是一个类似拉丁超立方体（Latin hypercube）和 N-rooks 采样的模式，其中 n 个采样点被放置在一个 $n \times n$ 的网格内，每行和每列都各有一个样本点[1626]。在 RGSS 采样模式中，这四个采样点分别位于 4×4 子像素网格的单独行和单独列中。与常规的 2×2 采样模式相比，这种旋转采样的方式尤其适合捕获接近水平或者接近垂直的边缘；而在常规的采样模式中，这些边缘很可能会覆盖偶数个的样本，因此其提供的有效程度较低。

N-rooks 采样模式只是创建良好采样模式的基础，其本身还不够好，例如：当 N-rooks 的采样点集中在亚像素网格的对角线上时，如果此时覆盖像素的边缘几乎平行于对角线，那么 N-rooks 会给出一个很差的采样结果，如图 5.27 所示。

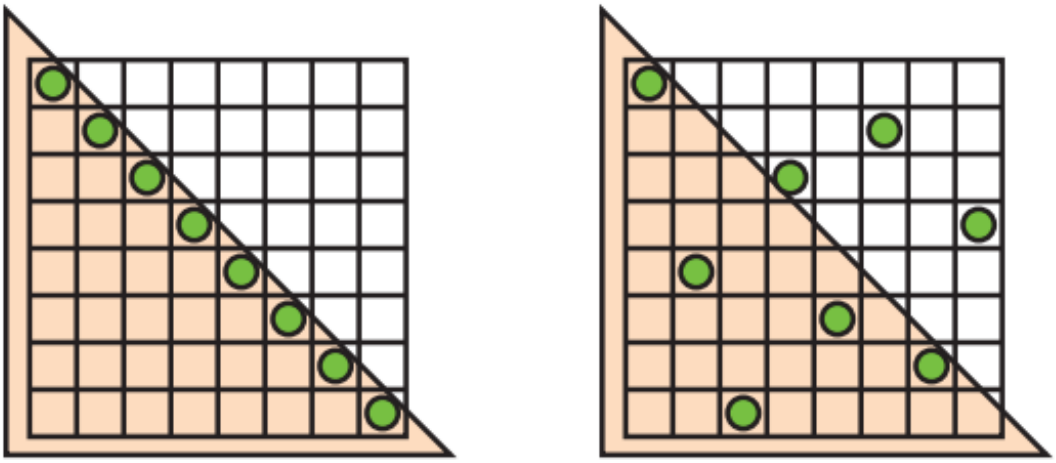


图 5.27：N-rooks 采样。左侧是一个合法的 N-rooks 采样模式，但是对于平行于对角线的三角形边界，它的捕获效果会很差，因为当这个三角形稍微移动一点， 那么所有的样本位置就会全部位于三角形内部或者全部位于三角形之外。右图也是一个合法的 N-rooks 采样模式，它可以更加有效地捕获这种边缘以及其他类型的边缘。

为了更好的进行采样，我们希望避免将两个样本放在一起。同时我们希望将这些样本均匀分布在这个像素区域中。为了获得这样的采样模式，我们可以将例如拉丁超立方体（Latin hypercube）采样的分层抽样（stratified sampling）技术与其他方法相结

合，例如抖动（jittering）采样，Halton 序列以及泊松圆盘采样（Poisson disk sampling）[1413, 1758]。

在实践中，GPU 生产制造商通常会将这种采样模式进行硬件实现，从而进行多重采样抗锯齿。图 5.28 展示了一些在实践中使用的 MSAA 采样模式。对于时域抗锯齿而言，由于每帧之间的样本位置可以发生变化，因此具体的采样覆盖模式是程序员按照自己的设计想法来进行实现的，例如：Karis [862]发现一个基础的 Halton 序列，就可以比任何 GPU 提供的 MSAA 模式表现得更好。Halton 序列在空间中生成的样本是随机的，并且样本之间的差异很小，也就是说，生成的这些样本，它们在空间中分布得很均匀，并且没有聚类（clustered）的现象[1413, 1938]。

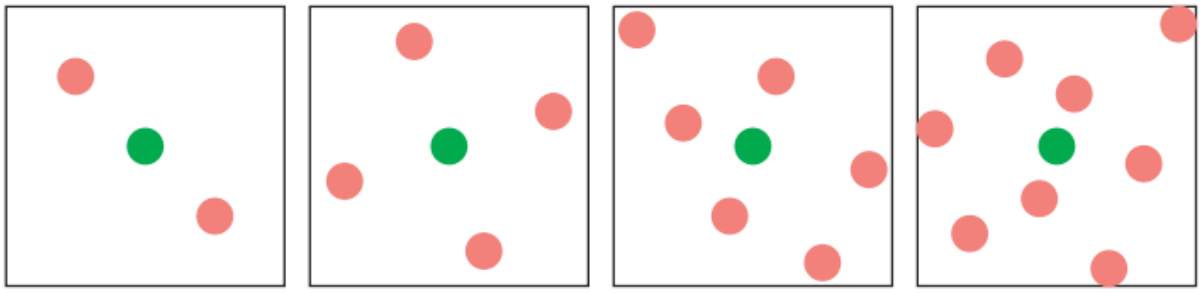


图 5.28：AMD 和 NVIDIA 图形硬件加速器（显卡）所采用的 MSAA 采样模式，其中绿色的点代表了着色样本的位置，红色的点计算并保存了采样点的位置。从左到右分别是 $2\times$ ， $4\times$ ， $6\times$ （AMD）和 $8\times$ （NVIDIA）MSAA 的采样模式。

虽然使用亚像素网格模式，可以更好地近似每个三角形是如何覆盖像素网格的，但是它并不是最理想的方法。一个场景可以由屏幕上任意小的物体组成，这意味着没有任何一个采样率可以完美地捕获它们。如果这些微小的物体或者特征形成了某种图案，那么此时以恒定的间隔对其进行采样，就会产生摩尔纹（Moire fringes）和其他的干涉图案。在这种情况下，超采样所使用的网格模式尤其容易产生走样现象。

一种解决方案是使用随机采样（stochastic sampling）（译者注：这里的 stochastic 也是随机的意思，一般用于形容一个过程是随机的，而 random 一般用于形容一个变量是随机的），这种方法可以生成一个更加随机的采样模式，例如图 5.28 中的所使用的模式。想象现在远处有一把细齿梳子，屏幕上的每个像素都包含了好几个梳齿。若此时采用的采样模式是规则不变的，并且它正好和梳齿出现的频率相对应，那么就会产生严重的瑕疵。而如果我们拥有一个不那么有序的采样模式，就可以避免这种情况发生。使用这种随机化的采样模式，虽然可以避免瑕疵的出现，但是也会引入图像噪声的问题，但是庆幸的是，人类的眼睛对于噪声更加宽容[1413]。使用几种不同结构的采样模式可以缓解瑕疵，但是如果当采样模式在像素之间重复的

话，还是没法避免瑕疵的出现。一种解决方案是在每个屏幕像素上，都使用完全不同的采样模式；或者随着时间的推移，对每个采样位置进行修改。在过去的几十年中，交错采样（interleaved sampling），索引采样（index sampling）偶尔会得到硬件的支持，其中每组像素都可以使用不同的采样方式。例如：ATI 的 SMOOTHVISION 允许每个像素包含 16 个样本，并且支持多达 16 种用户自定义的采样模式，这些采样模式可以混合在一个重复出现的模式（例如 4×4 像素块）中。Molnar [1234]以及 Keller、Heidrich [880]发现，当对每个像素使用相同的模式时，使用交错随机采样（interleaved stochastic sampling）可以最大程度上减少形成的锯齿瑕疵。

其它一些由 GPU 支持的算法也值得关注。NVIDIA 早期提出了一个五点型（Quincunx）方法[365]，该方法是一种实时抗锯齿方案，它可以让一个样本同时影响多个像素。“Quincunx”指的是五个物体的排列方式，其中四个位于正方形内部，第五个位于正方形的中心，例如六面骰子中五个点的图案。五点型多重采样抗锯齿便使用了这种模式，它将四个外部样本放置在正方形像素的四个角落上，如图 5.25 所示。每个角落上的样本值，都会被分发给其相邻的四个像素。不同于对每个样本进行平均加权（其他大多数实时方案都采用了这种策略），该方案的中心样本权重为 $\frac{1}{2}$，每个角落样本的权重为 $\frac{1}{8}$。由于这种共享策略，平均每个像素只需要两个样本，而且其结果要比双样本的 FSAA 方法要好得多[1678]。这种五点型的采样策略，优点类似于二维的 tent 滤波器，如上一小节所描述的那样，它是优于 box 滤波器的。

通过对每个像素仅使用一个样本，五点采样的策略也可以用于时域抗锯齿[836, 1677]。每一帧都在前一帧的基础上，在每个轴上偏移半个像素，偏移的方向会在两帧之间交替进行（抖动）。前一帧用于提供角落样本，并使用双线性插值来快速计算每个像素的贡献值，并将结果与当前帧进行平均。每一帧的总权重都是相等的，这意味着在静态视角下也不会出现闪烁瑕疵。但是沿着轴向进行移动的物体仍然会出现一些问题，但是这个方案本身的代码实现十分简单，并且由于一帧中的每个像素仅使用了一个样本，因此可以提供更好的外观表现。

在一帧中，五点型方法通过在像素边界共享样本信息，实际上每个像素只有两个样本的较低开销。RGSS 模式可以捕捉到更多接近水平边界和垂直边界的层次信息。首次为移动端开发的 FLIPQUAD（字面意思是翻转四边形）采样模式，结合了这两个方法的特点[22]，其优势在于，每个像素仅进行了两次采样，同时可以获得类似于 RGSS（每个像素四次采样）的质量。该采样模式如图 5.29 所示。Hasselgren 等人[677]探索了其他使用共享样本思路的低开销采样模式。

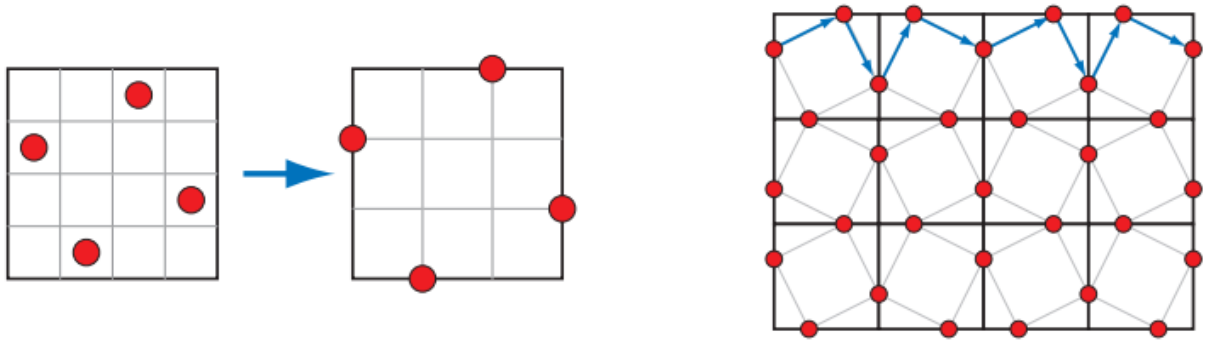


图 5.29：左图是现实的 RGSS 采样模式，每个像素采样四次。通过将这些样本移动到像素边缘，便可以与周围像素共享样本。但是为了实现样本共享，周围相邻像素都需要将样本模式进行对称处理，如右图所示。这种采样模式叫做 FLIPQUAD，每个像素只需要采样两次。

与五点型采样类似，双样本的 FLIPQUAD 方法也可以用于时域抗锯齿，在两帧之间共享样本。Drobot [382, 383, 1154]在其混合重建抗锯齿方法（hybrid reconstruction antialiasing，HRAA）中解决了使用哪种双样本采样模式才是最好的问题，他研究了用于时域抗锯齿的不同采样策略，他发现在测试的五种采样策略中，FLIPQUAD 是最好的。棋盘格模式也可以用于时域抗锯齿，El Mansouri [415]讨论了如何使用双样本的 MSAA 来创建一个棋盘格渲染，从而降低着色器的开销，同时解决瑕疵问题。Jimenez [836]提供了一种混合式的解决方案，该方案使用了 SMAA，时域抗锯齿和一系列其他技术；在该方案中，可以根据引擎的渲染负载，来实时改变抗锯齿的质量。Carpentier 和 Ishiyama [231]对边缘进行采样，并将采样网格旋转了 45 度，他们将这种时域抗锯齿策略和 FXAA（稍后会进行讨论）相结合，来提高在高分辨率显示器上的渲染效率。

形态学方法（Morphological Methods）

锯齿通常会在边缘处产生，例如几何图形，尖锐阴影，或者高亮所形成的边缘处，我们可以利用锯齿产生的原因和结构，对其进行针对性的优化，从而获得更好的抗锯齿效果。Reshetov [1483]于 2009 年提出了一种类似的算法，称之为形态学抗锯齿（morphological antialiasing，MLAA）。其中“形态（morphological）”意味着它与物体的结构或者形状有关。早在 1983 年，Bloomenthal [170]就完成了这个领域的早期工作[830]。Reshetov 的论文重新激发了大家对多重采样方法的替代方法的研究，其核心思路是对边缘的搜索和重建[1486]。

这种形式的抗锯齿是作为一个后处理（post-process）来执行的。也就是说，以通常的方式来渲染一张图像，然后将这个渲染结果输入到一个专门进行抗锯齿处理的过程中。自 2009 年以来已经发展了多种此类型的技术，这些技术依赖于一些额外的缓冲区（例如深度缓冲和法线缓冲）来生成更好的结果，例如亚像素重建抗锯齿

（subpixel reconstruction antialiasing，SRAA）[43, 829]，但是它只适用于几何边缘的抗锯齿处理。一些分析方法，例如几何缓冲区抗锯齿（geometry buffer antialiasing，GBAA）和距离边缘抗锯齿（distance-to-edge antialiasing，DEAA）等技术，让渲染器在渲染的过程中计算一些有关三角形边缘的额外信息，例如边缘到像素中心的距离[829]。

最普通的方法只需要使用颜色缓冲，这意味还可以使用阴影，高亮，以及各种之前应用于后处理阶段的技术（例如描边渲染，silhouette edge rendering，详见章节15.2.3），来对边缘效果进行改善，例如：定向局部抗锯齿（directionally localized antialiasing，DLAA）[52, 829]基于以下的观察进行实现：接近垂直的边缘应当被水平模糊，同样地，接近水平的边缘应当与其相邻像素进行垂直模糊。

一些更加复杂的边缘检测方法，尝试找到包含任意角度边缘的像素，并确定其覆盖范围。对潜在边缘的相邻像素进行检查，从而尽可能重建原始边缘所在的位置。然后通过考虑边缘对像素的影响，来融合相邻像素的颜色，图 5.30 展示了这个流程的原理图。

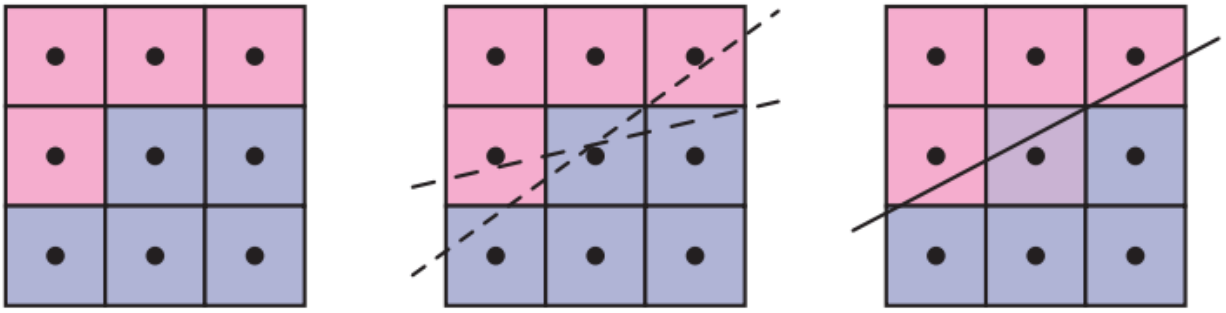


图 5.30：形态学抗锯齿。左边是一副带有锯齿的图像，我们的目标是找出潜在边缘的方向。在中间，该算法会对相邻像素进行检查，来标记一条可能存在的边缘，图中展示了两条可能的边缘位置。右图中，一个最佳猜测（best-guess）的边缘会根据像素的覆盖率，来将相邻像素的颜色混合到中心像素中，从而实现抗锯齿效果。

Iourcha 等人[798]通过检查像素中 MSAA 的样本，来计算更好的结果，从而改进了边缘检测的效果。值得注意的是，基于边缘预测和混合的算法，其结果的精度要比基于样本的算法更高，例如：一种每个像素采样四次的算法，只能给物体边缘混合提供 5 个不同的级别：即没有样本覆盖，或者有 1，2，3，4 个样本覆盖。而边缘预测算法所估测出的边缘位置，可以具有更多的位置情况，因此可以提供更好的结果。

在某几种情况下，基于图像的算法可能会导致错误的结果。第一，如果两个物体之间的颜色差异低于该算法的阈值，则可能无法有效检测到这个边缘。第二，如果某个像素被三个或者三个以上的表面所覆盖，那么就很难对这种情况进行边缘预测。第三，

具有高对比度或者高频元素的表面，其颜色可能会在像素之间快速变化，从而导致算法遗漏一些边缘。第四，当使用了形态学抗锯齿算法时，画面上的文字质量通常会受到影响。第五，物体的拐角处对于这种算法来说是一个挑战，使用一些算法后，可能会导致这些拐角变成圆角。第六，因为我们一般假设边缘都是直线的，因此弯曲的线段可能会被错误处理。第七，单独一个像素上的变化，就可能会导致边缘重建的方式发生巨大变化，从而在帧与帧之间产生鬼影和瑕疵；可以通过使用 MSAA 覆盖蒙版来提高边缘检测的准确性，从而改善这一现象[1484]。

形态学抗锯齿通常只会使用已经提供的信息，例如：某个物体的宽度可能不足一个像素（例如电线或者绳子），只要它没有覆盖到这个像素的中心，那么屏幕上就会出现一个像素的空白。在这种情况下，可以通过增加采样数来提高渲染质量，但是基于图像的抗锯齿算法并不能做到这一点。除此之外，这类算法的执行时间是不确定的，它取决于当前所观察到的内容，例如：一片草地视图，其所需的抗锯齿处理时间，可能是一片天空视野的三倍[231]。

综上所述，基于图像的方法能够以较小的内存使用和处理开销，来提高抗锯齿效果，因此这类算法被很多应用程序所使用。仅仅使用纯颜色缓冲的算法可以很好地与渲染管线相解耦，使得它们易于修改和禁用，甚至可以暴露为 GPU 驱动中的选项。其中两种最流行算法分别是快速近似抗锯齿（fast approximate antialiasing，FXAA）[1079, 1080, 1084]和亚像素形态学抗锯齿（subpixel morphological antialiasing，SMAA）[828, 830, 834]，其中的部分原因是因为，二者都为各种设备平台提供了可靠免费的源代码实现。两种算法都只使用了颜色缓冲，而 SMAA 的优势在于，它还可以访问 MSAA 的样本。二者都包含了一系列可选的设置，用于在渲染速度和渲染质量之间进行平衡，其每帧开销基本都在 1 到 2 毫秒内，这也是电子游戏愿意花费给抗锯齿处理的时间。最后，这两种算法也可以额外应用时域抗锯齿技术[1812]。Jimenez [826]提供了一种改进的 SMAA 实现，其速度比 FXAA 更快，并描述了一种时域抗锯齿方案。总而言之，我们推荐读者去阅读 Reshetov 和 Jimenez 所撰写的形态学技术综述[1486]，以及它们在电子游戏中的应用。

5.5 透明度，Alpha，合成

光线通过半透明（semitransparent）物体的方式有很多种，对于半透明渲染算法而言，这些效果可以大致分为基于光线（light-based）的半透明效果与基于视图（view-based）的半透明效果。基于光线的半透明效果是指，半透明物体会导致光线发生衰减和偏移，从而使得场景中的其他物体会被照亮，或者呈现出不同的渲染效果。基于视图的半透明效果是指透明物体自身的渲染效果。

本小节中，我们会讨论最简单形式的基于视图的半透明效果，即半透明物体本身会作为背后物体颜色的衰减器（attenuator）。在后面的章节中，我们会更加详细的讨论基于视图和基于光线的半透明效果，例如毛玻璃（frosted glass）、光线折射、由于透明物体的厚度而导致的光线衰减、以及因视角变化而带来的反射率（reflectivity）变化和透射率（transmission）的变化。

一种产生透明错觉的方法被称作点阵剔除半透明（screen-door transparency）[1244]，该方法的思路是使用一个像素对齐的棋盘格来对图案进行填充，从而渲染透明效果的三角形。也就是说，每个三角形像素都会被选择性地渲染，从而使得后面得物体变得部分可见。通常来说，如果屏幕上被渲染出来的像素比较紧凑密集，那么棋盘格本身的图案是不可见的。这种方法的主要缺点是，在屏幕上的一个区域内，通常只有一个物体的半透明效果比较令人信服；如果有多个半透明物体重叠在一起的话，这种方法就非常失真了。例如：如果一个红色的透明物体与一个绿色的透明物体，会在一个蓝色物体前面发生重叠的话，那么红绿蓝中的三种颜色中，只有两种可以出现在棋盘格图案上。除此之外，棋盘格的比例一般只能设置到 50%，虽然也可以通过使用其他更大的像素蒙版，来给出其他的百分比混合效果，但是这样会导致半透明效果失真[1245]。

这种点阵剔除的方法，其优点就是十分简单，透明物体可以在任意时间，以任意的顺序出现，并且不需要特殊的硬件支持。通过使得所有物体在棋盘格所覆盖的像素位置上变得不透明，而在其他地方变得透明，从而实现了半透明的效果。同样的思路也可以用于对镂空纹理（cutout texture）的边缘进行抗锯齿处理，不同的是这是在亚像素级别上实现的，使用了一个叫做 alpha to coverage（简称 A2C）的功能（章节 6.6）。

Enderton 等人[423]提出了随机透明度（stochastic transparency）的方法，它将亚像素级别的遮罩和随机采样的方法相结合，通过使用随机点状图案来表示一个片元的 alpha 覆盖率，可以生成一个合理，但是有噪声的图像，如图 5.31 所示。为了使得透明效果看起来较为合理，每个像素都需要进行大量的采样，并且每个亚像素样本都需要相当大的内存空间。它吸引人的地方在于，它并不需要进行混合操作，而且抗锯齿，透明度以及其他会创建部分覆盖像素的现象，都可以使用这个算法来实现。

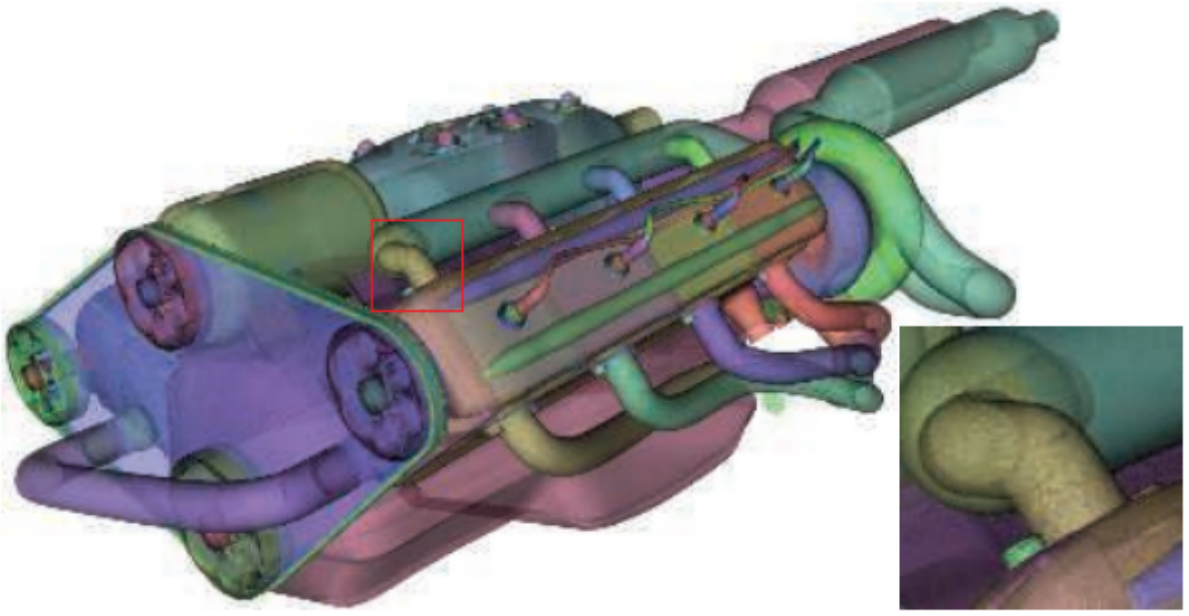


图 5.31：随机透明度。图中放大区域内展示了该算法产生的噪声。[1301]

大部分透明度算法都需要将透明物体的颜色，与其背后物体的颜色混合在一起，为此，需要使用 alpha 混合（alpha blending）的概念。当一个物体渲染到屏幕上时，每个像素都关联了一个 RGB 颜色和一个 z-buffer 深度值，我们还会为每个被物体所覆盖的像素，定义一个额外的分量，这个分量被叫做 alpha (α) 。alpha 描述了一个给定像素上，片元的透明度和像素覆盖率：当 alpha 为 1.0 的时候，意味着这个物体是不透明的，并且完全占据了这个像素内的所有区域；当 alpha 为 0.0 的时候，意味着这个像素完全没有被掩盖（obscured），即这个片元是完全透明的。

一个像素的 alpha 值可以用于表示不透明度，覆盖率或者同时表示这两者，具体需要依情况而定。例如：一个肥皂泡的边缘可能占据了四分之三个像素，即 0.75；并且它可能是几乎透明的，例如可以让十分之九的光穿过这个肥皂泡，即透明度为 0.1，那么其 alpha 值为 $0.75 \times 0.1 = 0.075$ 。但是，如果我们使用了 MSAA 或者类似的抗锯齿方案，那么样本本身就考虑到了对像素的覆盖率，即像素内四分之三的样本都会受到肥皂泡的影响。那么此时对于每个样本，我们将会使用 0.1 的不透明度来作为其 alpha 值。

5.5.1 混合顺序

为了使得一个物体看起来更加透明，我们需要将 alpha 小于 1 的物体渲染在场景的最前面。被这个物体所覆盖的每个像素，都会从像素着色器中接收到一个 $\mathrm{RGB}\alpha$ 值（也叫做 RGBA）。通过使用 **over** 运算符，来将这个片元的值和像素的原始颜色进行混合，其数学形式如下所示：

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + (1 - \alpha_s)\,\mathbf{c}_d \quad [\textbf{over} \quad \text{operator}] \tag{5.24}$$

其中的 $\mathbf{c}_s$ 代表透明物体的颜色（被称作源颜色 source），α_s 代表了这个物体的 alpha 值，$\mathbf{c}_d$ 代表该像素在混合之前的颜色（被称作目标颜色 destination），$\mathbf{c}_o$ 代表了将这个透明物体放置在现有场景前（**over** 操作）而生成的结果颜色。当渲染管线此时输入了物体的颜色 $\mathbf{c}_s$ 和 alpha 值 α_s 时，像素的原始颜色 $\mathbf{c}_d$ 会被替换为生成的结果颜色 $\mathbf{c}_o$；如果这个输入的 $\mathrm{RGB}\alpha$ 值实际上是一个不透明的颜色（即 $\alpha_s = 1.0$）的话，那么方程 5.24 会简化为直接使用这个输入物体的颜色来替换像素的原始颜色。

举例：混合：现在有一个红色的半透明物体被渲染到一个蓝色的背景上，假设物体上某个像素的 RGB 值为 $(0.9, 0.2, 0.1)$，背景的 RGB 值为 $(0.1, 0.1, 0.9)$，同时这个物体的不透明度为 0.6。那么这两个颜色的混合计算表达式为：

$$0.6(0.9, 0.2, 0.1) + (1 - 0.6)(0.1, 0.1, 0.9)$$

最后生成的结果颜色值为 $(0.58, 0.16, 0.42)$。

这个 **over** 运算符可以让一个被渲染的物体具有半透明的外观，这种方式实现的透明效果是可行的，因为从某种意义上来说，只要能透过这个物体看到其背后物体的样子，我们就会认为这个东西是透明的[754]。例如我们可以使用 **over** 运算符来模拟现实世界中轻薄布料（gauzy fabric）的效果，由于布料上的丝线（thread）是不透明的，因此布料背后的物体会被部分模糊。在实际实现中，松散布料的 alpha 覆盖率会随着视角的变化而变化[386]。也就是说，这里我们认为 alpha 值模拟了材质对像素的覆盖程度。

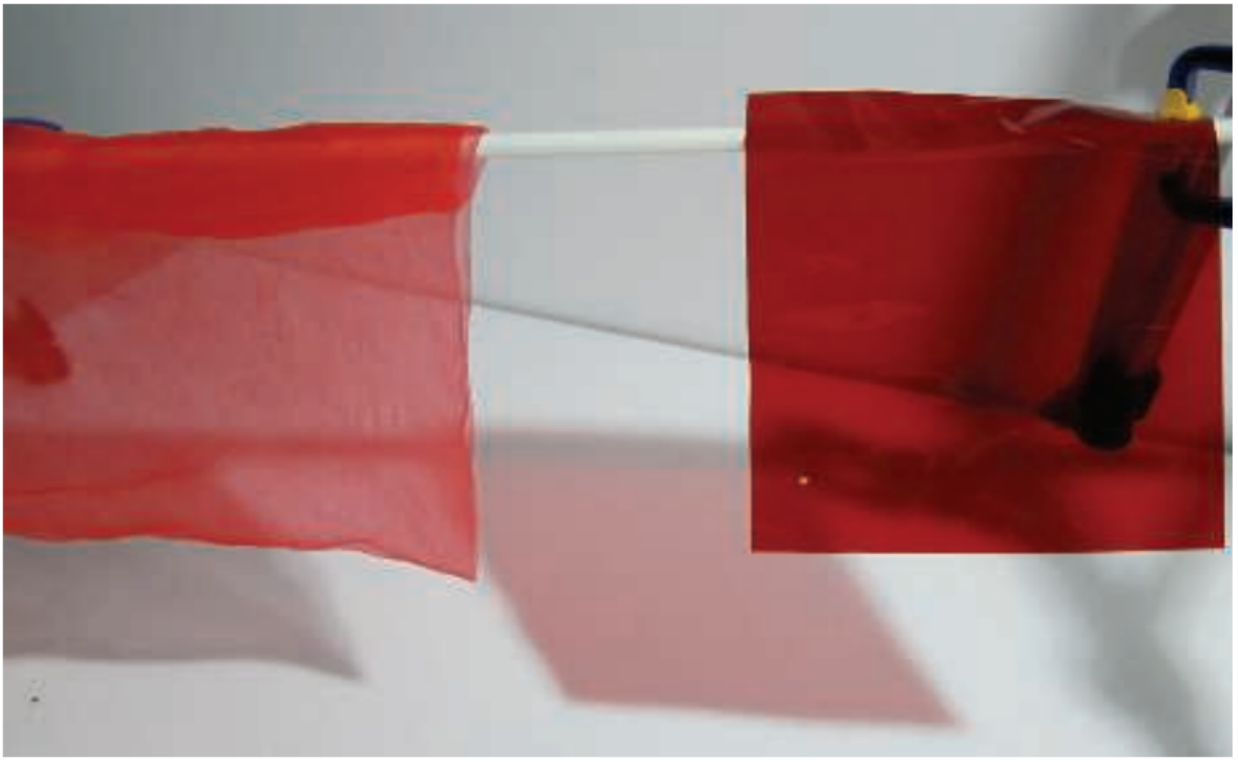


图 5.32：左侧是一个红色的方形布料，右侧是一个红色的塑料滤镜，二者的透明效果是完全不同的。请注意观察二者阴影效果的不同。

在模拟其他透明效果的时候，尤其是通过彩色玻璃或者彩色塑料来进行观察的情况下， **over** 运算符模拟的效果就不那么令人信服了。在现实世界中，在一个蓝色物体前面放置一个红色滤镜，通常会使得这个蓝色物体变得更暗，因为这个物体所反射出来的光线（蓝色光，波长较短），大部分都无法通过这个红色滤镜（只允许通过红色光，波长较长），由于通过的光线变少了，因此会显得更暗，如图 5.32 所示。而使用 **over** 运算符来进行模拟的时候，其结果是红色的一部分和蓝色的一部分被相加在一起。最好是能够将这两种颜色相乘，再加上透明物体自身的反射颜色，这种基于物理透射率的半透明效果，会在章节 14.5.1 和章节 14.5.2 中进行讨论。

在基本的混合阶段运算符中， **over** 是一个常用于透明效果的运算符[199, 1429]。另一个有一定用途的操作符是叠加混合（additive blending），在这个运算符中，参与混合的两个像素值只是简单的相加，其数学表达式为：

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + \mathbf{c}_d \tag{5.25}$$

这种混合模式适用于模拟各种发光效果，例如闪电或者火花，这些效果并不会对背后的像素产生影响，而只是使像素本身变得更亮[1813]。但是这种模式的生产的透明度效果看起来并不正确，因为不透明表面看起来并没有被过滤[1192]。对于一些层状的

半透明表面，例如烟雾或者火焰，这种叠加混合的模式具有饱和现象颜色的效果[1273]。

为了能够正确的渲染透明物体，我们需要在不透明物体绘制完成之后，再去绘制透明物体。具体的实现方式是先关闭混合操作，渲染所有的不透明物体；然后再启用 **over** 运算符，渲染所有的透明物体。理论上来说我们可以一直启用 **over**，因为不透明物体的 alpha 值为 1.0，它并不会导致 destination 颜色被替换成 source 颜色；但是这样做增加额外的开销，并且没有实际的收益。

z-buffer 存在的一个限制是，每个像素位置上仅会存储距离相机最近的那个深度值，如果有好几个透明物体都重叠在同一个像素上的话，那么就无法通过维护 z-buffer 来解决所有可见物体的效果了。因此在任意给定像素上叠加透明表面时，通常需要严格按照从后往前的顺序进行渲染，如果不这样进行处理的话，会导致错误的感知暗示。一种实现这种排序的方法是，将透明物体沿着观察方向，按照各自质心到相机的距离大小进行排序。这种粗略的排序可以很好地运行，但是在各种情况下都存在一些问题。首先，这样的排序仅仅是一个近似，位于远处的物体也可能会错误地出现在较近物体的前面；并且相互贯通重叠的物体，无法针对所有视角都进行正确显示，除非将每个网格都划分成单独的块，图 5.33 左侧图片展示了一个这样的情况。即便是一个凹形的单个网格，也可能会出现沿着观察方向的排序问题，即在屏幕上显示为与自身发生重叠。

图 5.33：左图中的模型使用了 z-buffer 来进行透明渲染，以任何一个顺序来渲染网格，都会出现严重的错误。右图中，使用深度剥离可以提供一个正确的渲染结果，但是代价是需要使用一个额外的 pass。

尽管如此，由于这种方法十分简单，速度很快，并且不需要使用额外的内存空间或者特殊的 GPU 特性，因此这种对透明度物体进行粗略排序的方法仍然被广泛使用。如果使用了这种方法，那么最好是在渲染透明物体的时候，关闭深度缓冲的写入替换功能。也就是说，z-buffer 仍然可以用于正常的深度测试，但是通过测试的表面并不会改变已有的 z-buffer 内容，即最近的不透明表面深度仍然会保持不变。通过使用这种方法，所有后来出现的半透明物体，至少都位于不透明表面前面（深度测试），这样就不会因为旋转相机而导致物体排序出现变化时，造成透明物体突然出现或者突然消失。也有一些其他的技术可以用于改善透明物体的外观，例如将一个透明物体连续渲染两次，先渲染一次背面，然后再渲染一次正面[1192, 1255]。

我们可以对 **over** 运算符的方程进行一些修改，使得从前向后的混合也可以得到相同的结果，这种混合模式被称作为 **under** 运算符。

$$\begin{aligned}\mathbf{c}_o &= \frac{\alpha_d \mathbf{c}_d + (1-\alpha_d)\,\alpha_s \mathbf{c}_s}{\mathbf{a}_o} \quad [\textbf{over} \quad \text{operator}] \\ \mathbf{a}_o &= \alpha_s\,(1-\alpha_d) + \alpha_d = \alpha_s - \alpha_s\alpha_d + \alpha_d\end{aligned} \tag{5.26}$$

请注意， **under** 运算符要求目标颜色维护一个 alpha 值，而 **over** 运算符则不需要。换而言之，目标物体（更靠近相机的透明表面）并不是一个不透明物体，因此需要有一个 alpha 值。 **under** 的数学公式与 **over** 很像，只是将源（ s ）和目标（ d ）进行了交换。另外需要注意的是，alpha 的计算公式是与顺序无关的，因为在我们交换了方程中的源 alpha 和目标 alpha 之后，仍然可以得到相同的 alpha 结果。

计算 alpha 的方程来自于将片元的 alpha 值，看作为该片元对像素的覆盖率。Porter 和 Duff 指出[1429]，由于我们并不知道每个片元覆盖像素区域的形状，因此我们假设每个片元都会按照其 alpha 的比例，来覆盖另一个片元。例如：如果 $\alpha_s = 0.7$ ，即意味着这个像素会以某种方式被划分成两个区域，其中有 70% 的部分被源片元所覆盖，另外 30% 则没有被覆盖；这里假设目标片元的覆盖率为 $\alpha_d = 0.6$ ，在没有其他条件的情况下，目标片元将会按照覆盖率与源片元进行重叠。图 5.34 展示了该公式的几何解释。

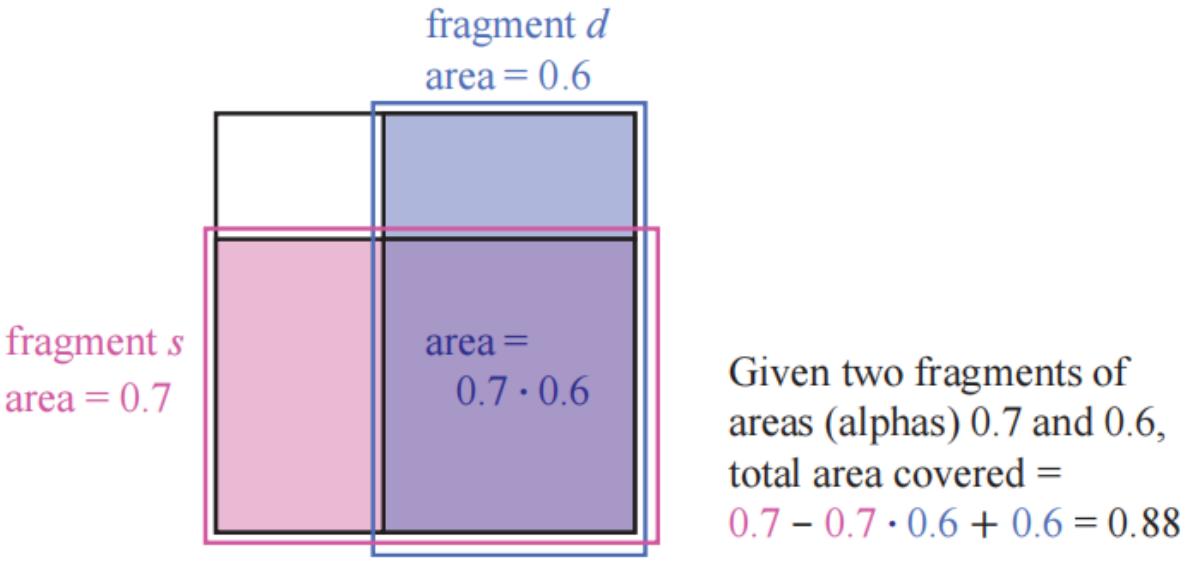


图 5.34：图中展示了一个像素和两个片元 s, d 。我们将这两个片元沿着不同的轴进行对齐，每个片元都会按照一定的比例覆盖另一个片元，也就是说，这两个片元是不相关的。两个片元所覆盖的区域大小与 **under** 运算符输出的 alpha 值相等（ $\alpha_s - \alpha_s\alpha_d + \alpha_d$ ）。从公式中我们可以看出来，这意味着将这两个区域相加，然后再减去二者重叠的部分。

5.5.2 顺序无关的透明度算法

under 运算符用于将所有透明物体都绘制到一个单独的颜色缓冲中，然后再使用 **over** 运算符将这个颜色缓冲合并到场景的不透明视图上。 **under** 运算符的另一个用途是执行被称为深度剥离（depth peeling）的算法[449, 1115]，该算法是一种顺序无关的透明度算法（order-independent transparency，OIT），这里的顺序无关意味着：应用程序不需要对透明物体进行排序。深度剥离的核心思路是使用两个 z-buffer 和多个 pass。首先，第一个 pass 会将所有表面的 z-depth 信息记录在第一个 z-buffer 中，包括所有的透明表面；在第二个 pass 中，只会渲染所有的透明物体，如果一个透明物体的 z-depth 与第一个 z-buffer 中的深度值相匹配，那么我们可以知道，这个透明物体是距离相机最近的，并将其 $\mathrm{RGB}\alpha$ 值保存在一个单独的颜色缓冲区中；并且此时我们会进行“剥离”操作，具体方式是使用距离相机第二近的透明物体的 z-depth（如果存在的话）来更新第一个 z-buffer 中的对应位置，此时最靠近相机的那个透明物体便被剥离了出去，代替它的是第二近的透明物体。接下来的一系列 pass 会继续按照这种方式来剥离透明物体，并在透明物体的颜色缓冲区上，使用 **under** 运算符来进行混合。我们会在一定数量的 pass 操作之后终止算法，然后将透明图像（透明物体的颜色缓冲区）混合到不透明图像上，如图 5.35 所示。

图 5.35：每个深度剥离的 pass 都会绘制其中一层的透明物体。左图是第一个 pass 的渲染结果，它代表的是相机直接可见的透明图层（位于最前面）。中间展示了第二个 pass 的渲染结果，每个像素上都是距离相机第二近的透明表面，在这个例子中代表了透明物体的背面。右图展示了第三个 pass 的渲染结果，每个像素上都是距离相机第三近的透明表面。图 14.33 展示了最终的渲染结果。

深度剥离算法有好几种变体，例如：Thibieroz [1763]给出了一种可以从后向前进行渲染的算法，其优势在于可以将透明值进行立即混合，这意味着我们不需使用额外的 alpha 通道。深度剥离算法的一个问题在于，我们并不知道到底要使用多少个 pass 才能捕获所有的透明层，一种基于硬件的解决方案是，通过使用一个像素绘制计数器来记录当前 pass 中写入了多少像素；如果在一个 pass 中没有任何的像素被写入，那么就说明完成了深度剥离。使用 **under** 运算符的优点在于，最重要的那一个透明层（眼睛最先看到的）是最先被渲染的。每个透明表面总是会增加所覆盖像素的 alpha 值，如果一个像素的 alpha 值接近 1.0 的话，意味着这个像素已经接近不透明了，而对于后续需要进行混合的颜色（位于更远处的透明表面的颜色），其影响可以忽略不计[394]。我们可以通过设置一个固定的 pass 数量，或者当一个 pass 所渲染的像素数量低于某个设定的最小值时，将算法终止，从而减少从前到后的剥离次数。但是这种方式不利于从后向前的剥离过程，因为最重要的那一层位于最前面，它通常是最后被绘制的，如果过早终止算法的话，可能会丢失这些最重要的透明信息。

尽管深度剥离算法可以有效渲染透明物体，但是这个算法的效率并不高，因为每次剥离的过程，都是对所有透明物体的一次独立渲染 pass。 Bavoil 和 Myers [118]提出了一种双重深度剥离算法（dual depth peeling），在每个 pass 中，会剥离当前最近和最远的两层透明表面，从而使得 pass 数量减半。Liu 等人[1056]探索了一种桶排序方法（bucket sort method），可以在一个 pass 中最多捕获 32 层透明表面；该方

法的一个缺陷在于，它需要一个相当大的内存空间，来维护所有透明层的排序顺序。如果再使用 MSAA 或者类似的抗锯齿技术的话，会极大增加内存开销。

如何以一个可交互的速率来正确混合透明物体，这个问题的关键不在于我们缺少这样的有效算法，而是在于如何将这些算法在 GPU 上进行高效实现。Carpenter 于 1984 年提出了 A-buffer [230]，这是多重采样的另一种方式。在 A-buffer 中，每个被渲染的三角形（片元），都会为其所覆盖地每个屏幕单元格创建一个覆盖掩码（coverage mask），每个像素上都会存储与其相关的所有片元。不透明片元会剔除位于它们后面的片元，就像 z-buffer 一样；而所有透明表面的片元都会被存储下来。当所有的信息都被构建完成之后，会通过遍历片元和解析样本的方式，来生成最终的结果。

在 GPU 上创建片元链表（linked list）的想法，是通过 DirectX 11 中暴露的新特性实现的[611, 1765]，这些新特性包括章节 3.8 中提到的无序访问视图（UAV）和原子操作（atomic operation）。基于 MSAA 的抗锯齿可以通过访问覆盖掩码和计算每个样本的像素着色器结果来实现。该算法的原理是对每个透明表面进行光栅化，并将生成的片元插入到一个长数组中。然后连同颜色信息和深度信息一起，会生成一个单独的指针结构，该结构会将每个片元与该像素的前一个片元相链接。然后会执行一个单独的 pass，它会渲染一个填充屏幕的四边形，以便在每个屏幕像素位置上调用像素着色器，这个着色器会根据已有的指针链接，找到每个像素上所有的透明片元；检索到的每个透明片元，都会与前面的片元按照深度进行排序，然后将排序后的链表，按照从后往前的顺序进行混合，从而生成最终的颜色。由于这里的混合操作是通过像素着色器执行的，因此如果需要的话，我们可以为每个像素都指定不同的混合模式。GPU 和 API 的持续发展降低了使用原子操作符的开销[914]，从而提升了性能表现。

A-buffer 的优点在于，只需要为每个像素分配所需的片元即可，不会浪费额外的存储空间，GPU 上的链表实现也是如此。但是从某种意义上来说，这也可能是一个缺点，因为在渲染之前，我们并不知道到底需要多少的存储空间（链表可能会很长）；而且对于一个带有头发、烟雾或者其他具有大量重叠透明物体的场景而言，它所能产生的片元数量将会是十分巨大的。Andersson 指出[46]，对于一些复杂的游戏场景，可能会有多达 50 个透明网格（如树叶）和 200 个半透明粒子会发生重叠。

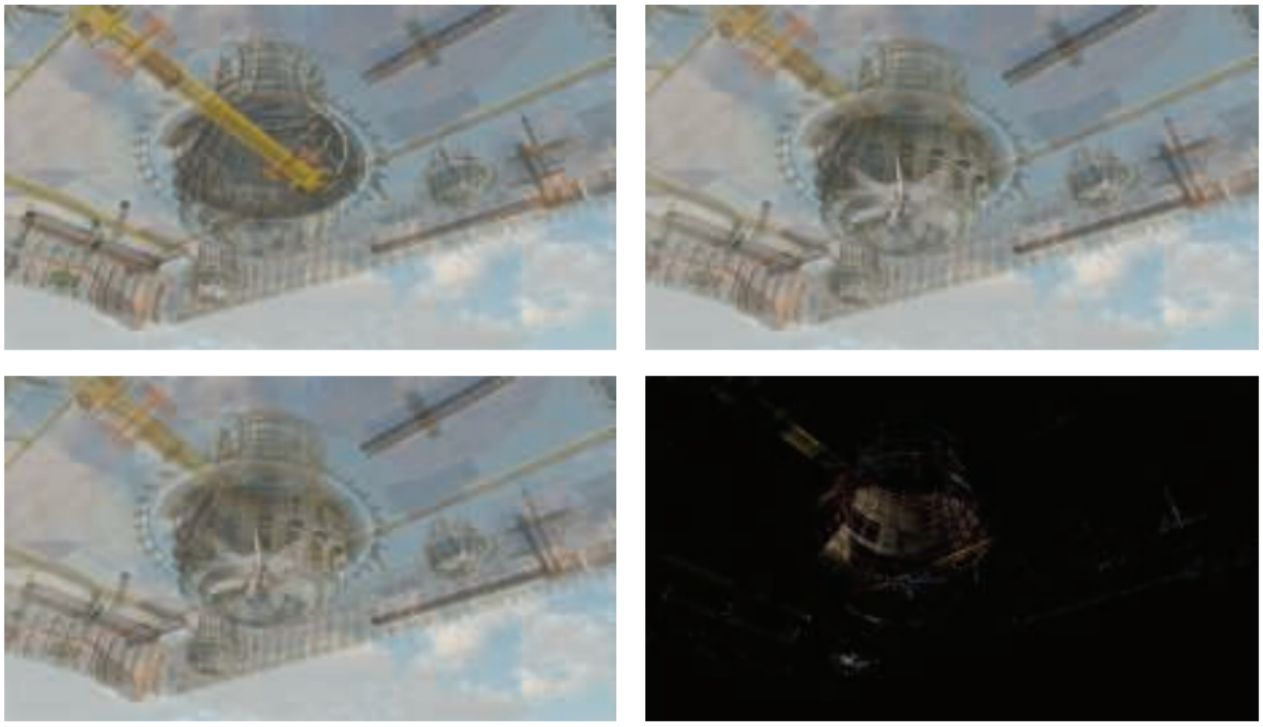


图 5.36：左上角执行的是传统的、从后向前的 alpha 混合，由于错误的排序顺序，会导致渲染出错。右上角使用了 A-buffer，给出了一个完美的，非交互式的结果。左下角使用了多层 alpha 混合的渲染结果。右下角展示了 A-buffer 和多层 alpha 混合之间的差异，为了便于观察，将结果颜色放大了四倍[1532]。

GPU 通常会有预先分配好的存储资源，例如缓冲区和数组，链表方法也不例外。可以由用户来决定到底使用多少内存，如果内存耗尽的话，则会造成一些很明显的瑕疵。Salvi 和 Vaidyanathan [1532]提出了一种解决这个问题的方法，即多层 alpha 混合（multi-layer alpha blending），它使用了一个由 Intel 引入的 GPU 特性，该特性叫做像素同步（pixel synchronization），如图 5.36 所示。这一特性可以用于实现可编程混合，并且开销要比原子操作小。他们的方法重新定义了存储和混合，以便能够在内存耗尽的时候，适当降低质量。此外，粗略的排序对这种方法也有一定的帮助。DirectX 11.3 引入了光栅器有序视图（章节 3.8），这是缓冲区的一种类型，它允许在支持该功能的任何 GPU 上实现这种透明方法[327, 328.。移动设备也有着被称为 tile 本地存储（tile local storage）的类似技术，允许在这些设备上使用多层 alpha 混合[153]。然而这种机制也有一定的性能成本，因此这类算法的开销可能会很大[1931]。

这种方法建立在 Bavoil 等人[115]提出的 k-buffer 的思想上，其中前几个可见层会被尽可能地保存和排序，而更深的层则会被尽可能地丢弃和合并。Maule 等人[1142]使用了一个 k-buffer，并使用加权平均（weighted average）来描述这些较远的透明层。即该算法会保留前面若干个透明层，并对它们进行排序；而对于后续的透明层，

会将多个层划分为一组，并使用加权平均的方式进行混合。基于加权求和[1202]和加权平均[118]的透明技术，都是顺序无关的，并且都是单 pass 的，因此可以运行在几乎所有的 GPU 上。但是它们的问题都在于，没有考虑到物体的前后顺序，例如：使用 alpha 来表示覆盖率，一条淡红色围巾叠加在一条淡蓝色围巾的上面，会给人一种紫罗兰色的感觉；而正确地结果是一条带有一些蓝色的红色围巾。虽然对于几乎不透明的物体而言，这种方法产生的效果会很差，但是这类算法对于可视化很有用，而且对于高度透明的表面和粒子也很有效，如图 5.37 所示。

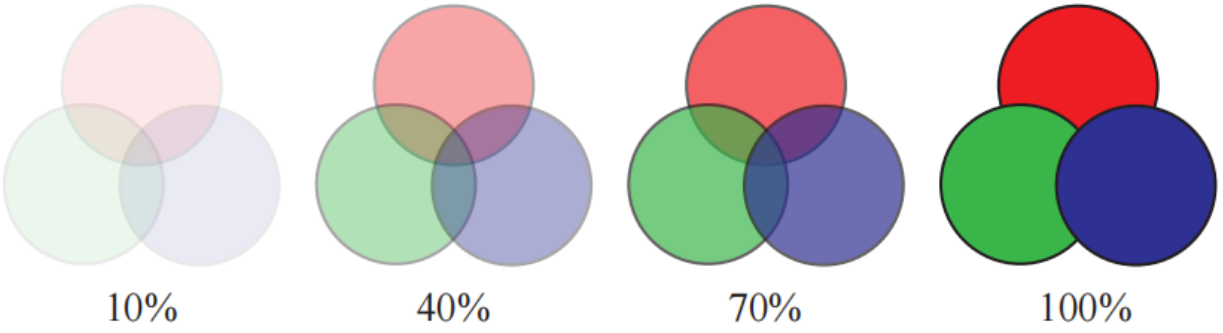


图 5.37：随着不透明度的增加，物体之间的前后顺序会变得越来越重要。[394]

加权和（weighted sum，WS）透明度的方程为：

$$\mathbf{c}_o = \sum_{i=1}^{n} (\alpha_i \mathbf{c}_i) + \mathbf{c}_d \left(1 - \sum_{i=1}^{n} \alpha_i\right) \tag{5.27}$$

其中方程中的 n 代表透明表面的数量， $\mathbf{c}_i$ 和 α_i 代表了第 i 个透明表面的颜色值和 alpha 值， $\mathbf{c}_d$ 是场景中不透明部分的颜色。方程中包含两个求和的部分，当透明表面被渲染的时候，这两部分会被分别累加并存储；并且在透明 pass 的最后，每个像素都会对方程 5.27 进行计算。这种方法的问题有两个：1，第一个求和式的结果会饱和，即产生超过 $(1.0, 1.0, 1.0)$ 的颜色；2，由于第二个求和式的结果也可能会大于 1，因此背景颜色可能会产生反作用。

由于加权和方程的上述问题，因此通常都会使用加权平均方程：

$$
\begin{aligned}
\mathbf{c}_{\text{sum}} &= \sum_{i=1}^{n} \left(\alpha_i \mathbf{c}_i\right), \quad \alpha_{\text{sum}} = \sum_{i=1}^{n} \alpha_i, \\
\mathbf{c}_{\text{wavg}} &= \frac{\mathbf{c}_{\text{sum}}}{\alpha_{\text{sum}}}, \quad \alpha_{\text{avg}} = \frac{\alpha_{\text{sum}}}{n}, \\
u &= \left(1 - \alpha_{\text{avg}}\right)^n, \\
\mathbf{c}_o &= (1-u)\mathbf{c}_{\text{wavg}} + u\mathbf{c}_d.
\end{aligned}
\tag{5.28}
$$

第一行代表了在透明渲染的过程中，生成的两个独立缓冲区的结果，每个透明表面对 $\mathbf{c}_{\text{sum}}$ 的贡献，都会受到其 alpha 值的影响：越是不透明的表面（alpha 越大）所贡献的颜色就越多，越是透明的表面（alpha 越小）所贡献的颜色就越少。令 $\mathbf{c}_{\text{sum}}$ 除以 α_{sum} ，我们便可以得到一个加权平均的透明颜色；而 α_{avg} 则是所有 alpha 值的平均值。 u 代表了对于 n 个透明表面而言，将平均 alpha 应用 n 次之后，对目标（不透明场景）可见性的估计值（即大约有 $u\%$ 的部分不可见，即有 $(1-u)\%$ 的部分可见，即 alpha 值）。最后一行实际上就是 **over** 操作符，其中 $(1-u)$ 代表了源的 alpha 值。

加权平均有一个限制，即对于具有相同 alpha 值的透明表面而言，无论它们的顺序如何，加权平均都会均匀混合它们的颜色。McGuire 和 Bavoil [1176, 1180]引入了加权混合的顺序无关透明度渲染算法，得到了更加可信的结果。在他们提出方程中，表面到相机的距离也会对权重产生影响，即越靠前的表面会产生越大的影响。此外，方程中的 u 不再使用平均 alpha 值来进行估计，而是用 1 减去 $(1-\alpha_i)$ 各项相乘的结果，从而给出了这一组透明表面真实的 alpha 覆盖率。这个方法可以产生在视觉上更加可信的结果，如图 5.38 所示。

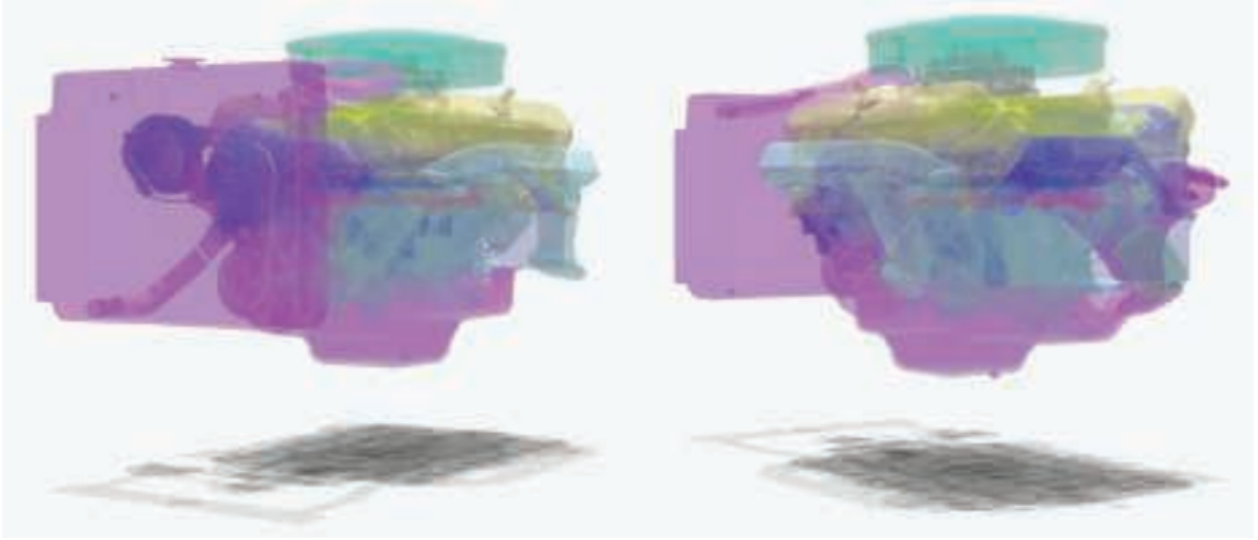


图 5.38：图中展示了在两个不同的相机位置上，观察同一个引擎模型的结果，它们都使用了加权混合的顺序无关透明度算法。按照距离进行加权，有助于弄清哪些表面更加靠近观察者[1185]。

这种按照距离进行加权的方法存在一个缺点，即在一个很大的场景中，彼此靠近的两个物体会拥有几乎相同的权重，这使得最终计算的结果与使用加权平均方法的结果相差不大。除此之外，随着相机到透明物体的距离发生变化，深度权重也可能会发生变化，但是这种变化是平缓的，并不会造成突变的效果。

McGuire 和 Mara [1181, 1185]对这种方法进行了扩展，使其包含了合理的颜色透射效果。上文中我们曾经提到，本小节中所讨论的透明度算法，都只是对像素覆盖率进行模拟，从而实现对颜色的混合，而不是对颜色进行过滤。为了得到颜色过滤的效果，需要在像素着色器中获取不透明场景的颜色，将每个透明表面的颜色与它所覆盖的像素颜色相乘，然后将结果存储到第三个缓冲区中。在这个缓冲区中，不透明物体现在会被透明物体染色，并在接下来解析半透明缓冲区的时候，使用第三个缓冲区来代替不透明场景。与使用覆盖率来模拟透明度的方法不同，颜色透射是顺序无关的，因此这个方法也是有效的。

还有一些其他的算法，它们用到了这里所介绍技术的部分元素和思路。例如：Wyman [1931]按照内存需求、插入和合并的方法、是否使用了 alpha 覆盖率或者几何覆盖率，以及如何处理被丢失的片元，对之前的透明度算法进行了分类。通过寻找先前研究中的空白与不足，他展示了两种被发现的新方法。其中一种方法叫做随机分层 alpha 混合（stochastic layered alpha blending），该方法使用了 k-buffer，加权平均以及随机透明度；另一种方法是 Salvi 和 Vaidyanathan 方法的变体，使用了覆盖掩码来代替了 alpha。

考虑到透明物体的类型，渲染方法以及 GPU 特性，目前还没有一种能够完美渲染透明物体的解决方案。我们推荐感兴趣的读者参考一下 Wyman 的论文[1931]以及 Maule 等人对交互式透明度算法的详细调查[1141]。McGuire 的演讲[1182]为该领域提供了一个更加开阔的视野，内容贯穿了与透明度相关的各种现象，例如体积照明，颜色透射和折射等，这些内容将会在本书的后续章节中，进行更加深入的讨论。

5.3.3 Alpha 预乘与合成

over 操作符也可以用于将照片或者物体的渲染图混合在一起，这个过程被称为合成（compositing）[199, 1662]。在这种情况下，每个像素的 alpha 值会与物体的 RGB 值存储在一起，alpha 通道所构成的图像有时候也被称为无光粗糙层

（matte，也叫做哑光），它展示了物体的轮廓形状，图 6.27 展示了该图层的一个例子。这个 $\mathrm{RGB}\alpha$ 图像可以用于与其他元素和背景进行混合。

使用合成 $\mathrm{RGB}\alpha$ 数据的一种方式是 alpha 预乘（premultiplied alpha，也被称为关联的 alpha），它的意思是在使用这些 RGB 值之前，要先将它们与对应的 alpha 值相乘。在 alpha 预乘之后，会使得 over 方程变得更加高效：

$$\mathbf{c}_o = \mathbf{c}'_s + (1 - \alpha_s)\,\mathbf{c}_d \tag{5.29}$$

其中 $\mathbf{c}'_s$ 代表了已经 alpha 预乘过的源通道，它代替了方程 5.25 中的 $\alpha_s\mathbf{c}_s$ 项。由于在混合期间已经叠加过了源颜色，因此 alpha 预乘还可以在不改变混合状态的情况下，直接使用 **over** 操作符和叠加混合。请注意，对于已经 alpha 预乘过的 $\mathrm{RGB}\alpha$ 值而言，尽管也可以使用它们来创建一个特别明亮的半透明值，但是其中的 RGB 分量通常并不会大于其 alpha 值。

渲染合成图像很适合使用预乘 alpha 的方法，在黑色背景上渲染抗锯齿的不透明物体默认会提供预乘值。假设一个白色 $(1, 1, 1)$ 的三角形，在其边缘上覆盖了某个像素的 40%，这时通过一些抗锯齿方法（非常精确），该像素值会被设置成为 0.4 的灰度值，即我们会将这个像素值保存为 $(0.4, 0.4, 0.4)$。如果还要存储 alpha 值的话，那么其 alpha 值也为 0.4，因为这是三角形所覆盖的区域面积。该像素最终的 $\mathrm{RGB}\alpha$ 值为 $(0.4, 0.4, 0.4, 0.4)$，这是一个预乘过的像素值。

另一种存储图像的方法是使用未相乘的 alpha（unmultiplied alpha），也被称为不关联的 alpha，甚至是令人费解的术语——未预乘的 alpha（nonpremultiplied alpha）。未相乘的 alpha 就是其字面意思：存储的 RGB 并不乘以其 alpha 值。在刚才的那个白色三角形的例子中，不相乘的颜色值为 $(1, 1, 1, 0.4)$。这种表示的方式的好处在于，它存储了三角形的原始颜色，但是这种颜色在被显示之前，还需要再乘以它的 alpha 值。在执行过滤和混合操作的时候，最好是使用 alpha 预乘的颜色数据；因为如果使用未相乘的 alpha 的话，诸如线性插值的一些操作无法正确执行[108, 164]。物体边缘也会产生黑色条纹状的瑕疵[295, 648]。章节 6.6 对此进行了深入讨论。另外，alpha 预乘也可以使得理论表达更加整洁[1662]。

对于图像处理的应用程序而言，使用不关联的 alpha 可以在不影响图像原始数据的情况下，对照片进行遮罩处理。此外，不关联的 alpha 意味着可以使用颜色通道的全精度范围；也就是说，在将未相乘的 $\mathrm{RGB}\alpha$ 值转换到用于计算机图形计算的线性空间时，需要格外注意转换的正确性。例如：现在没有浏览器可以对其正确转换，也不可能做这样的转换，因为现在的预期结果就是不正确的[649]。支持 alpha 通道的文件

格式包括 PNG（仅支持不关联的 alpha），OpenEXR（仅支持关联的 alpha）和 TIFF（同时支持两种关联类型的 alpha）。

一个与 alpha 通道相关的概念是色键抠像（chroma key）[199]，这是视频制作中的一个术语，即演员在绿色背景或者蓝色背景前进行拍摄，然后再与其他背景进行混合。在电影工业中这个过程被叫做绿幕（green-screening ）或者蓝幕（blue-screening）。这个技术的核心思路是将某个特定的色调（用于电影工业），或者某个精确的值（用于计算机图形学）看作是透明的，当检测到这个颜色的时候，就会显示背景图像。这项技术允许仅使用 RGB 颜色来标注物体轮廓，并不需要使用额外的 alpha 通道。这个方案有一个缺点，即图像中的物体要么是完全不透明的，要么就是完全透明的，即 alpha 只有 1.0 和 0.0 两个取值。例如：GIF 格式允许将一种颜色指定为透明颜色。

5.6 显示编码

在计算光照，纹理效果或者是其他操作的时候，我们会假设所使用的值是线性的（linear）。通俗来讲，线性意味着加法和乘法可以生成预期的效果。但是为了避免各种视觉瑕疵，显示缓冲区和纹理中使用了非线性的编码方式，这是我们必须要考虑到的。一个简单粗略的答案是：将着色器输出的颜色范围设置为 $[0,1]$，并将输出的结果缩放为原来的 $1/2.2$ 次方倍，这个过程被称作为伽马矫正（gamma correction）；而对于输入的颜色和纹理则要进行相反的处理，即要乘以 2.2 次方。在大多数情况下，你可以让 GPU 来执行这些转换操作，本小节的目的是简要总结如何进行伽马矫正以及为什么要进行伽马矫正。

我们将从阴极射线管（cathode-ray tube，CRT）开始，在数字成像的早期阶段，通常会使用 CRT 显示器来进行成像。这些设备的显示 radiance 和输入电压之间具有指数关系。当应用在一个像素上的能量增加时（电压增加），该像素的 radiance 并不会线性增长，例如：假设电压与发光强度之间的指数比为 2，此时将应用于该像素上的电压设置为原来的 50%，那么实际的发光强度为 $0.5^2 = 0.25$，即原来的四分之一[607]。虽然液晶和其他显示技术有着不同于 CRT 显示器的亮度响应曲线，但是由于它们都是通过转换电路制造的，因此可以模拟 CRT 显示器的响应方式。

这个指数函数几乎与人类眼睛对亮度的敏感程度相反，这个幸运的巧合也使得这种编码方式符合人类的视觉感知[1431]。也就是说，在可显示范围内，一对相邻编码值 N 和 $N+1$ 之间的感知差异大致是恒定的。使用和阈值对比度（threshold contrast）类似的方法进行测量，我们可以在很大范围的条件下，检测到大约 1% 的亮度差异（即将人眼对亮度的感知曲线以及 CRT 显示器的响应曲线进行对比）。当颜色存储

在有限精度的显示缓冲区中时，这个近似最优的分布能够最大程度地减少条带瑕疵（章节 23.6）。对使用相同编码方式的纹理，也可以起到同样的优化效果。

显示转换函数（display transfer function）描述了显示缓冲区中的值与实际显示器发光强度之间的关系，因此它也被称为电光转换函数（electrical optical transfer function，EOTF）。显示转换函数是硬件的一部分，对于不同类型的显示器而言（例如计算机显示器，电视以及电影放映机），具有不同的标准。对于这一过程的另一端（图像和视频捕获设备）也有一个相应的标准转换函数，它被称为光电转换函数（optical electric transfer function，OETF）[672]。

当对用于显示的线性颜色值进行编码时，我们的目标是抵消显示转换函数的影响，这样计算出的任何颜色值才能发出相应水平的亮度，例如：如果我们计算出的结果翻倍了，那么我们希望显示器输出的亮度也能相应的翻倍。为了保证这种关系，我们需要应用显示转换函数的逆，来抵消其非线性效应，这个消除显示器响应曲线的过程也被称作为伽马矫正（gamma correction），关于进行伽马矫正的原因会在接下来进行说明。当对纹理进行编码的时候，我们需要应用显示转换函数来生成用于着色的线性颜色值。图 5.39 展示了在显示过程中编码和解码的使用。

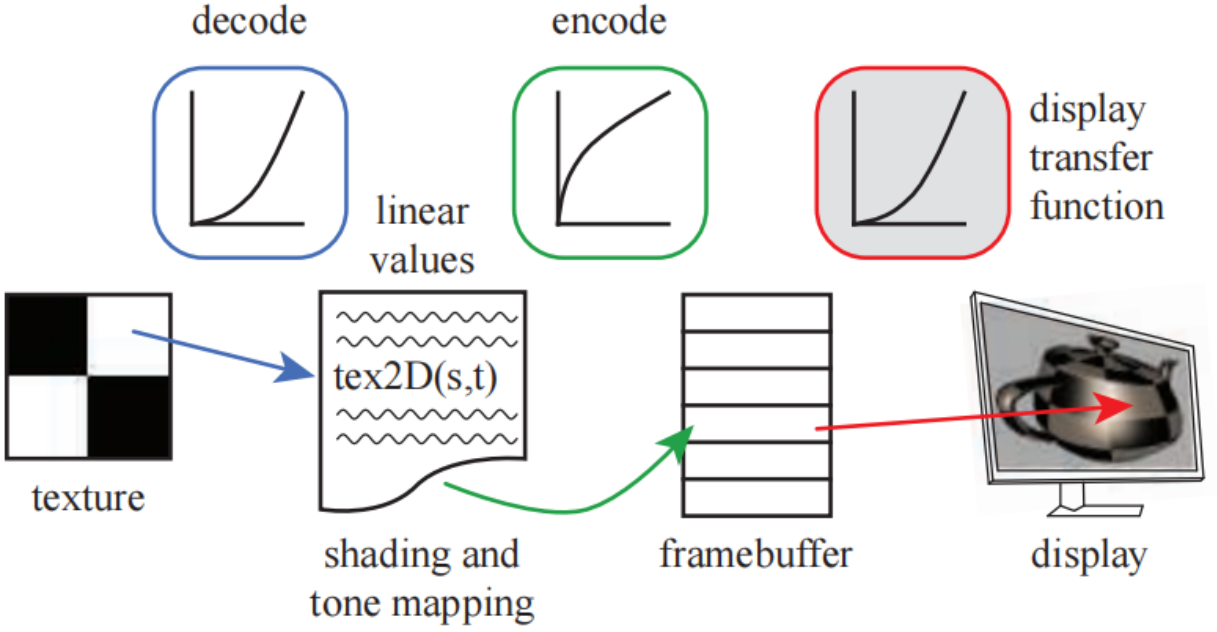


图 5.39：最左边，GPU 着色器访问了一个 PNG 格式的彩色纹理，该纹理的非线性编码值（蓝色过程）被转换为了线性值。在进行着色和色调映射（章节 8.2.2）之后，最终计算出的颜色值进行了编码（绿色过程），并存储在帧缓冲中。帧缓冲中的颜色值与显示转换函数（红色过程），最终决定了显示器上的发光强度。绿色的编码过程与红色的显示转换函数相抵消，因此显示器实际的发光强度与线性的计算值成正比。

PC 显示器的标准转换函数由一个被称为 $sRGB$ 的颜色空间所定义。大部分控制 GPU 的 API 可以被设置为：当读取纹理和写入颜色缓冲时，自动完成适当的 sRGB 转换[491]。例如章节 6.2.2 中所讨论的，mipmap 在生成的时候也会采用 sRGB 编码。在纹理上进行双线性插值的时候，为了保证插值结果的正确性，首先会将纹理中的颜色值转换为线性值，然后再进行插值计算。alpha 混合的时候也是类似的，先将存储在缓存中的值解码为线性值，然后执行混合操作，再对混合结果进行编码和写入。

在渲染的最后阶段，当颜色值被写入帧缓冲中并等待显示的时候，应用这个转换是十分重要的。如果在编码之后再进行一些其他的后处理操作，那么此时是在对非线性的颜色值进行处理，这通常是不正确的，会导致明显的瑕疵。显示编码可以被认为是一个压缩过程，它最大程度上保留了颜色值的感知效果[491]。有一个很好的方法可以用于理解这个转换过程：我们总是会使用线性值来执行实际的计算，每当要显示结果或者访问可显示的图像（例如彩色纹理）时，我们就需要使用适当的编解码转换，来将数据转换到合适的编码格式中。

如果我们需要手动进行 sRGB 转换的话，可以使用标准的转换方程，或者一些简化过的版本。在实际操作中，每个颜色通道的比特数会对显示结果进行控制，例如：对于消费级的显示器而言，一般都是 8bit 的，它可以提供 $[0, 255]$ 共计 256 个不同的颜色等级。这里我们为了方便讲解，将显示编码的级别表示为 $[0.0, 1.0]$ 之间的一个浮点数，线性值也位于这个范围中。我们常用 x 来表示线性值，用 y 来表示帧缓冲中的非线性值。为了将线性值 x 转换为 sRGB 编码的非线性值 y，我们需要对 x 应用 sRGB 显示转换函数的逆，即：

$$y = f_{\mathrm{sRGB}}^{-1}(x) = \begin{cases} 1.055x^{1/2.4} - 0.055, & \text{where } x > 0.0031308 \\ 12.92x, & \text{where } x \leq 0.0031308 \end{cases} \tag{5.30}$$

其中 x 代表了 RGB 值中的一个通道，我们会对每个通道的线性颜色值都应用这个方程，然后由生成的非线性值来驱动显示器。如果我们手动应用这个转换函数的话，那么需要格外的小心，一般会有两个常犯的错误：1，对已经编码过的颜色值（而不是线性值）进行编码处理；2，对一个颜色进行编码两次或者解码两次。

这个变换过程是一个两段函数，两个表达式实际上都是简单的乘法，因为硬件设备需要这个变换是完全可逆的[1431]。其中第一行表达式包含一个指数运算，它几乎适用于 $[0.0, 1.0]$ 的所有范围。考虑到偏移量和尺度，这个函数可以被近似为下面的简单形式[491]：

$$y = f_{\text{display}}^{-1}(x) = x^{1/\gamma} \tag{5.31}$$

其中 $\gamma = 2.2$，希腊字母 γ 是“伽马矫正”的名称来源。

计算出的数值结果需要进行编码才能正确显示；类似地，使用相机拍摄的图片，在用于计算之前也必须将其转换为线性值。我们在显示器或者电视机上看到的任何颜色，都具有被显示编码过的 RGB 三元组，我们可以通过屏幕截图或者颜色选择器来获得具体的数值。这些值会以例如 PNG，JPEG，GIF 等文件格式来进行存储，这些文件格式可以直接发送到帧缓冲中并显示在屏幕上，而并不需要进行编码转换。换而言之，我们在屏幕上看到的任何图像，都是已经经过显示编码的数据；在使用这些颜色进行着色计算之前，我们都必须将其转换为线性值。将 sRGB 格式解码为线性值的方程如下：

$$x = f_{\text{sRGB}}(y) = \begin{cases} \left(\dfrac{y + 0.055}{1.055}\right)^{2.4}, & \text{where } y > 0.04045, \\ \dfrac{y}{12.92}, & \text{where } y \leq 0.04045, \end{cases} \tag{5.32}$$

其中 y 代表了归一化的显示通道值，即存储在图像或者帧缓冲中的值，其范围是 $[0.0, 1.0]$。这个解码函数刚好与之前的 sRGB 编码方程相反，这意味着如果我们在一个着色器中读取了一张图片（解码过程），并且不对它进行任何处理直接输出（编码过程）的话，它将和处理之前的结果完全相同。解码函数其实和显示转换函数是完全一样的，因为存储在纹理中的值已经被编码过了，而且它能够正确地显示在显示器上；二者之间不同的是，解码函数将输入转换为一个线性值，而显示转换函数则是将其转换为一个线性响应显示。

更加简单的伽马显示转换函数，是直接将方程 5.31 取逆：

$$x = f_{\text{display}}(y) = y^{\gamma}. \tag{5.33}$$

有时我们会看到一组更加简洁的转换函数，特别是在移动和浏览器应用程序中 [1666]：

$$\begin{aligned} y &= f_{\text{simpl}}^{-1}(x) = \sqrt{x} \\ x &= f_{\text{simpl}}(y) = y^2 \end{aligned} \tag{5.34}$$

也就是说，直接对线性值开平方就完成了编码过程，并直接用于显示；反过来，直接取编码值的平方根就完成了解码过程，二者互为反函数。这种转换仅仅是一个很粗略的近似，但是要比完全忽略伽马矫正的过程好上不少。

如果我们不进行伽马矫正的话，那么数值较小的线性值在屏幕上会显得太暗，同时某些颜色的色调可能会发生改变。假设我们设置 $\gamma = 2.2$ ，如果我们希望屏幕上像素的发光强度与计算出来的线性值成正比的话，这意味着要必须要将这个线性值提升到原来的 $1/2.2$ 次方倍。线性值 0.1 对应的发光强度为 0.351，0.2 对应 0.481，0.5 对应 0.730。如果我们不进行编码的话，直接将这些线性值输入到显示器中，会导致显示器的发光强度低于所需要的值。需要注意的是，线性值为 0.0 和 1.0 在经过编码之后不会有任何改变。在引入伽马矫正之前，场景建模人员通常会手动拉高暗表面的颜色值，从而使得它们在显示变换之后，不至于显得太暗。

图 5.40：上图展示了两盏聚光灯照亮同一个平面的情况。左图中的两盏灯的亮度分别为 0.6 和 0.4，中间重叠的部分直接将两盏灯的亮度相加，即 0.6+0.4=1.0，并没有使用伽马矫正；对非线性值进行加法运算，会导致结果出错。我们可以观察到，左边的灯光要比右边的更亮，但是重叠的部分却出奇的亮。右图中的亮度值在相加之前进行了伽马矫正，两盏灯相较于左图都要更亮，但是中间重叠的部分会显得比较自然。

忽略伽马矫正的另一个问题是，对基于物理的线性 radiance 的正确着色计算，是在非线性值上进行的，图 5.40 展示了这种情况的一个例子。

图 5.41：左图中，一个黑色背景（为了方便展示，图中显示的是灰色）上有一个白色三角形，其边缘覆盖了四个像素，像素内标注了真实的覆盖率。如果不进行伽马矫正的话，像素整体会

偏暗，从而扭曲人眼对于边缘的感知，如右图所示。

忽略伽马矫正也会影响抗锯齿边缘的质量，例如：假设一个三角形的边界覆盖了四个平面单元格（图 5.41），其中三角形的归一化 radiance 为 1（白色），背景的 radiance 为 0（黑色）。从左到右，白色三角形分别占据了单元格的 $\frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}$，假设使用 box 滤波器来进行抗锯齿处理，我们希望处理后像素的线性 radiance 分别为 0.125，0.375，0.625 和 0.875。正确的方法是对线性值进行抗锯齿处理，然后再对结果的像素值进行编码处理。如果不这么做的话，边缘像素的 radiance 会变得很暗，这将导致人眼对于边缘的感知发生变形，如图 5.41 右侧所示。这种瑕疵被称为扭绳（roping），因为边缘看起来有点像一个扭曲的绳子[167, 1265]，图 5.42 展示了这种瑕疵的效果。

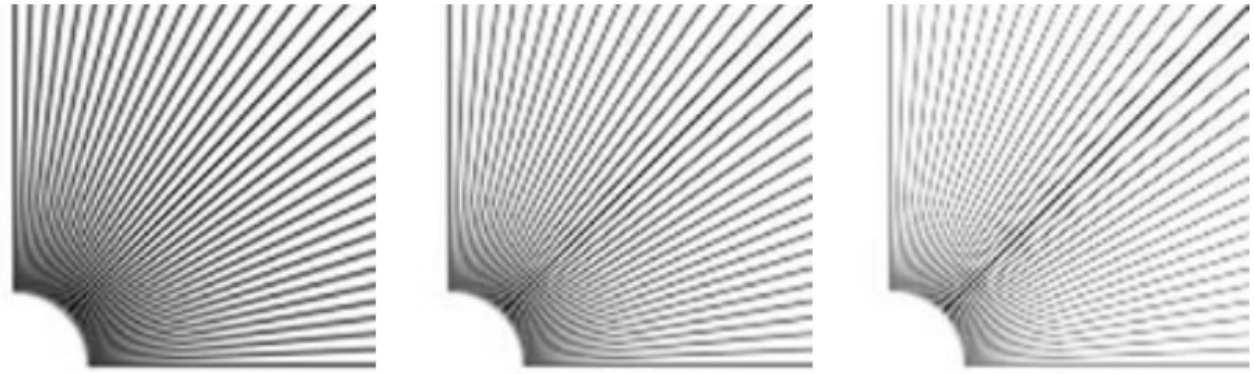


图 5.42：左边，在抗锯齿之后进行了伽马矫正。中间，在抗锯齿之后进行了部分伽马矫正。右边，没有进行伽马矫正。

sRGB 标准创建于 1996 年，现已成为大多数 PC 显示器的标准。然而这些年来，显示技术有了巨大的发展，出现了亮度更亮，色域更广的显示器。有关色彩显示和亮度的话题在章节 8.1.3 中进行了讨论；有关高动态显示范围（HDR）的编码方式在章节 8.2.1 中进行了讨论。Hart 的论文[672]介绍了有关高级显示器的详细内容。

补充阅读和资源

Pharr 等人[1413]深入讨论了采样模式和抗锯齿的问题。Teschner 的课程讲义[1758]展示了各种采样模式的生成方法。Drobot [382, 383]贯穿了之前有关实时抗锯齿的研究，解释了各种抗锯齿技术的特点和性能表现。你可以在 SIGGRAPH 课程[829]中找到各种有关形态学抗锯齿的方法。Reshetov 和 Jimenez [1486]提供了游戏中所使用的形态学以及相关时域抗锯齿工作的最新回顾。

对于透明度算法的研究，我们再次向感兴趣的读者推荐 McGuire 的演讲[1182]和 Wyman 的论文[1931]。Blinn 的论文“What Is a Pixel?”[169]在讨论像素的不同定义

时，提供了对计算机图形学多个领域的优秀介绍。Blinn 的书籍《Dirty Pixels》和《Notation, Notation, Notation》[166, 168]包含了一些介绍性的文章，其中涉及滤波和抗锯齿，alpha，合成和伽马校正等内容。Jimenez 的演讲[836]详细介绍了用于抗锯齿的最先进技术（state-of-the-art）。

Gritz 和 d'Eon [607]对伽马矫正问题有一个很好的总结。Poynton 的书[1431]对各种媒体文件中的伽玛校正，以及其他与颜色相关的主题进行了详细介绍。Selan 的白皮书[1602]是一个较新的资料来源，它解释了显示编码及其在电影行业中的应用，以及许多其他相关的信息。

Chapter 6 Texturing 纹理

> Jim Blinn——“All it takes is for the rendered image to look right.”

> 吉姆 布林——“所需要做的就是使渲染出来的图像看起来正确。”（美国计算机图形学专家，Blinn–Phong 着色模型；1949—）

表面纹理（texture）是指其外观和给人的视觉感受，就像是一幅油画的图案一样。而在计算机图形学中，纹理化则指的是一个过程，即通过使用一些图像、函数或者其他数据，来对每个表面位置的外观表现进行修改。例如：我们可以将一张砖墙的彩色图像应用于由两个三角形组成的矩形上，而不是去精确表现砖墙的几何结构。当我们观察这个砖墙矩形的时候，对应的彩色图像将会显示在这个矩形所在的位置上，这样可以使得这个矩形看起来很像真实的砖墙。除非相机十分靠近墙壁的话，否则砖墙几何细节的缺乏并不会带来明显的视觉瑕疵。

然而，除了缺乏细致的几何结构之外，一些具有纹理的砖墙也有可能无法令人信服。例如：砖墙的砂浆（砖块和砖块之间粘合物）应当是哑光的（matte），而砖块则应当是有光泽的（glossy），但是观察者会注意到，此时这两种材料表现出的粗糙度（roughness）实际上是相同的。为了产生更加令人信服的视觉表现，我们可以将第二张图像纹理应用到这个表面上，这种纹理并不会改变表面的颜色，而是会根据表面位置来修改墙壁的粗糙度。现在砖块和砂浆从图像纹理中获得了颜色，从新纹理中获得了各自所对应的粗糙度值。

通过上述两张纹理图像，观察者可以看到，现在所有的砖块都是有光泽的，而所有的砂浆都是哑光的。但是现在还有一些问题，那就是每个砖块看起来都非常平整，而真实的砖块表面通常都是坑坑洼洼的，是不规则的。我们可以通过使用凹凸映射（bump mapping），对砖块表面的着色法线进行一些修改，从而使得其在渲染之后看起来并不平整。这类纹理贴图通过对矩形的原始表面法线进行抖动处理，从而改变光照的计算结果。

如果从一个接近平行的角度来观察这个砖墙的话，那么这种崎岖不平的错觉便会露出马脚，因为现实中的砖块要比砂浆更加突出，因此我们应该是看不到砂浆的。而且即使是从一个垂直的视角进行观察，砖块也应当会在砂浆上投射出阴影。视差映射（parallax mapping）会在渲染平面时，使用一个特殊纹理来使其变形；视差遮蔽映

射（parallax occlusion mapping）会对高度纹理进行光线投射，从而提高渲染的真实感。位移映射（displacement mapping）通过使用位移贴图，从而对模型的三角形高度进行修改。图 6.1 展示了一个具有颜色贴图和凹凸贴图的例子。

图 6.1：通过将颜色贴图和凹凸贴图应用到这条鱼身上，从而增加其细节表现。

以上所举的例子都可以通过使用纹理技术来解决，它们使用了越来越复杂的算法。在本章节中，我们将详细介绍有关纹理处理的相关技术。首先，我们会给出了纹理化过程的一般框架。然后，我们将重点关注在纹理表面上应用纹理贴图的过程，因为这是实时渲染中最为流行的纹理使用形式。我们也会简要讨论有关程序化纹理的内容，也会介绍一些使用纹理贴图来影响表面的常见方法。

6.1 纹理管线

纹理化（texturing）是一种用于描述表面材质以及对表面进行修饰加工的有效技术，一种理解纹理的方法是，思考单个着色像素会如何发生变化。正如前一章中所提到的，着色计算需要考虑材质和光源的颜色，以及其他各种复杂的因素；如果场景中还有透明物体的话，那么还需要考虑透明度对着色计算的影响。纹理的工作原理是通过修改着色方程中所使用的参数，从而对最终的着色结果产生影响，而这些参数通常会随着表面位置的变化而变化。例如：对于上文中的砖墙例子，会根据着色点在表面上的位置信息，将该点的颜色替换为砖墙图像中的对应颜色。为了与屏幕上的像素（pixel）概念有所区别，图像纹理中的像素通常被称为纹素（texel）。粗糙度纹理修改了表面的粗糙度值，凹凸纹理修改了表面着色法线的方向，因此每个纹理都对最终着色方程的计算结果产生了影响。

纹理化的过程可以被描述为一个更加一般的纹理管线。稍后我们将会介绍许多术语，但是请你牢记：我们将会详细描述这个纹理管线中的每个部分。

纹理化的起点首先是空间中的一个具体位置，这个位置可以在世界空间中，但是通常都会放在模型的参考坐标系中，因为当模型发生移动的时候，纹理也会随之移动。这里我们使用 Kershaw 提出的术语[884]：这个空间点会应用一个投影函数（projector function）来获得一组数字，它被称为纹理坐标（texture coordinates），这个纹理坐标将用于访问和采样纹理，这个过程被称为纹理映射（texture mapping）。有时候纹理图像本身会被称为纹理贴图（texture map），尽管这并不是严格正确的。

在使用纹理坐标访问纹理之前，还需要使用一个或者多个转换函数（corresponder function），来将纹理坐标转换到纹理空间中。 转换后的纹理空间位置用于在纹理中获取像素值，例如：纹理空间位置可以是图像纹理中的数组索引，从而检索到对应位置上的像素值。检索到的像素值可能还需要使用一个值转换函数（value transform function）来进行转换，最终这些新值会用于对表面的某些属性进行修改，例如材质或者着色法线等。图 6.2 详细展示了一个纹理的应用过程。这个纹理管线之所以要设计得如此复杂，是因为其中的每个步骤都可以为用户提供一些有用的控制。需要注意的是，并不是每次纹理应用过程都需要激活管线中的所有步骤。

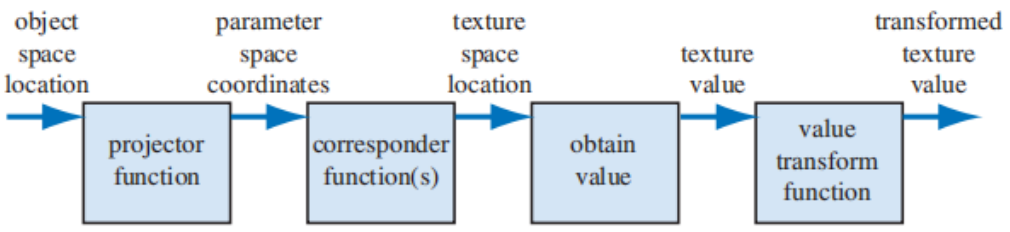


图 6.2：针对单个纹理的广义纹理管线 。

例如：对于一个具有砖墙纹理的三角形，我们在其表面上进行采样时会发生如下情况（如图 6.3 所示）：首先我们会在该物体的局部参考系中，找到对应的采样位置 (x, y, z)，这里假设它是 $(-2.3, 7.1, 88.2)$。然后会对这个位置坐标应用一个投影函数，就像世界地图是三维地球的二维投影那样，这里的投影函数通常会将一个三维向量 (x, y, z) 转换为一个二维向量 (u, v)。本例中所使用的投影函数，实际上与正交投影是等价的（章节 2.3.1），它就像幻灯片放映机一样，将砖墙图像投影到三角形表面上；并且为了最后能将图象值返回到墙面上，其表面上的点都会被转换为一个 0–1 范围内的数值对，这里我们假设转换后的值是 $(0.32, 0.29)$，这个数值对也被称为纹理坐标或者 UV 坐标。这个纹理坐标将用于查找纹理贴图在此位置上的颜色值。假设这里我们所使用的砖墙纹理分辨率为 256×256，因此使用转换函数，将纹理坐标 (u, v) 各自乘以 256，即 $(81.92, 74.24)$。在丢弃小数部分之后，我们在砖墙图像中进行检索，找到索引值为 $(81, 74)$ 的颜色值，这里假设这个颜色值为 $(0.9, 0.8, 0.7)$。同时，我们所使用的纹理颜色位于 sRGB 颜色空间中，因此如果要在着色方程中使用这个颜色值，还需要将其转换到线性空间中，即 $(0.787、0.604、0.448)$ （章节 5.6）。

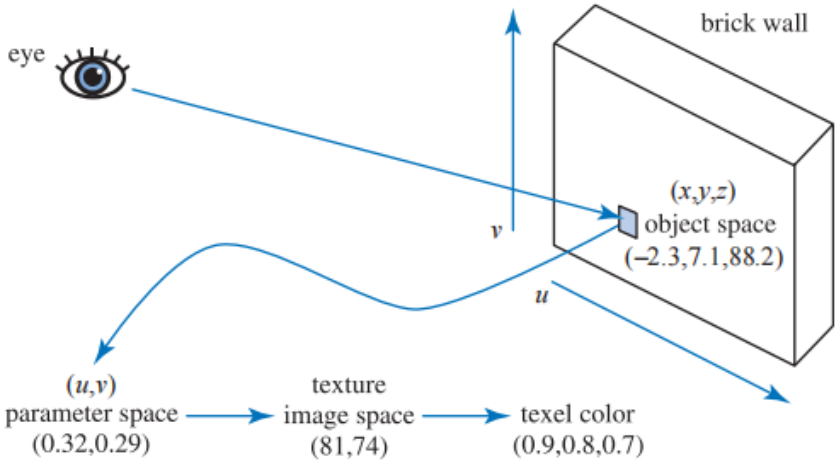


图 6.3：砖墙纹理管线过程中的参数变化。

6.1.1 投影函数

纹理处理的第一步是获取表面的位置，并将其投影到纹理坐标空间（texture coordinate space）中，这个纹理坐标空间通常是一个二维 (u, v) 空间。常见的建模软件都允许艺术家定义每个顶点的 (u, v) 坐标。这些 (u, v) 坐标可以从投影函数（projector function）或者网格展开算法（mesh unwrapping algorithm）中进行初始化，艺术家也可以像编辑顶点位置那样，对 (u, v) 坐标进行编辑。投影函数的作用

通常是将空间中的三维坐标转换为二维纹理坐标，在建模软件中常用的投影函数包括球面投影（spherical）、柱面投影（cylindrical）和平面投影（planar）等[141, 884, 970]。

投影函数还可以有其他的输入参数，例如：表面法线可以用来选择应用于该表面的平面投影方向（一共六个）。在面片接缝处（即 UV 接缝）常常会出现纹理匹配的问题，Geiss [521, 522]讨论了一种将 UV 接缝混合在一起的技术。Tarini 等人[1740]描述了立方体映射技术（polycube maps），在该方法中，一个模型会被映射到一组立方体投影上，空间中的不同区域会被映射到不同的立方体上。

而其他的一些投影函数实际上根本就不是投影操作，而是隐含在了表面创建和曲面细分中。例如：参数化曲面的定义本身就包含了一组天生的 (u, v) 坐标，如图 6.4 所示。纹理坐标也可以从其他不同的参数中生成，例如观察方向、表面温度（热力图）或者任何其他可以想象的东西。投影函数的最终目标是生成纹理坐标，将其作为一个与位置有关的函数来进行推导，只是其中的一种方法。

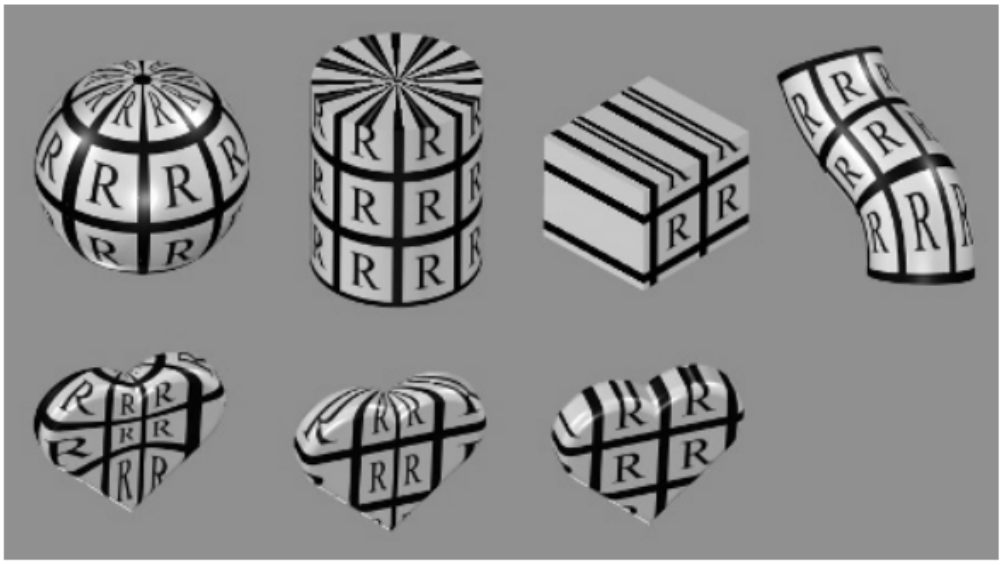


图 6.4：不同的纹理投影方法。第一行从左到右分别是：球面、柱面、平面和自然 (u, v) 投影。第二行则展示了这些投影方法应用于同一个物体的结果（不包含自然投影）。

非交互式的渲染器会经常调用这些投影函数，这是渲染过程本身的一部分。虽然有时候单个投影函数就可以用于整个模型的投影操作，但是艺术家通常会使用一些工具来将模型进行细分，并单独应用不同的投影函数[1345]，如图 6.5 所示。

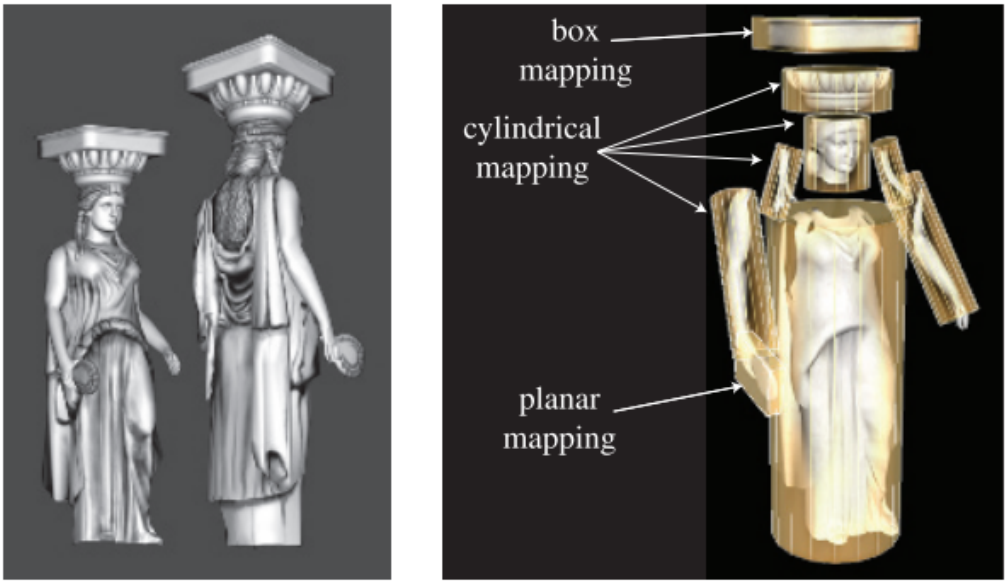


图 6.5：展示了如何在单个模型上使用不同的纹理投影方法。右图中的 box 映射由六个平面映射组成，其中每个立方体面都对应一个平面映射。

在实时渲染中，通常会在建模阶段使用投影函数，并将结果数据存储在顶点上。但是情况并非总是如此，有时在顶点着色器或者像素着色器中应用投影函数是会带来一些好处，这样做可以提高精度，并有助于实现包括动画（章节 6.4）在内的各种效果。有时候一些渲染方法有着自己独特的投影函数，它们会进行逐像素的计算，例如环境映射（environment mapping，章节 10.4）

球面投影（spherical projection，图 6.4 左侧）会将表面点投影到一个以某点为中心的假想球体上，Blinn 和 Newell 在其环境映射方案（章节 10.4.1）中所使用的投影方法便是球面投影，方程 10.30 对这个函数进行了描述。但是这种投影方法与那一小节中所描述的顶点插值方法，都存在相同的问题。

柱面投影（cylindrical projection）计算纹理坐标 u 的方法与球面投影相同，而纹理坐标 v 则是沿圆柱体轴的距离。这种投影方法对于具有中心轴的物体而言十分有用，例如旋转的表面。柱面投影的缺点是，当一个表面几乎垂直于圆柱体的中轴时，就会发生畸变（如柱面投影两端的圆形表面）。

平面投影（planar projection）就像一束 x 射线一样，它会沿着一个方向进行平行投射，并将纹理应用到所有的表面上。平面投影使用了正交投影方法（章节 4.7.1）。这种类型的投影在贴花（decal）应用中十分有用（章节 20.2）。

与投影方向平齐的表面会发生严重的扭曲，因此艺术家经常需要手动将模型分解成接近平面的小块（即建模流程中的分 UV）。有一些工具可以通过对网格进行展开，或者创建一组接近最优的平面投影，来帮助减少这种扭曲现象。我们的目标是，让每个

多边形在纹理区域中尽量占据更加公平的份额，同时尽可能多的保持网格连通性。网格的连通性是非常重要的，因为在网格的接缝处很容易出现采样瑕疵。一个具有良好展开效果的网格，可以使得艺术家的后续工作变得更加轻松[970, 1345]。章节 16.2.1 中讨论了纹理扭曲是如何对渲染产生负面影响的。图 6.6 中所展示的 UV 图和纹理图，与图 6.5 中的雕像相对应。这个展开过程是一个更大的研究领域（网格参数化，mesh parameterization）的其中一部分，有兴趣的读者可以参考 Hormann 等人[774]的 SIGGRAPH 课程讲义。

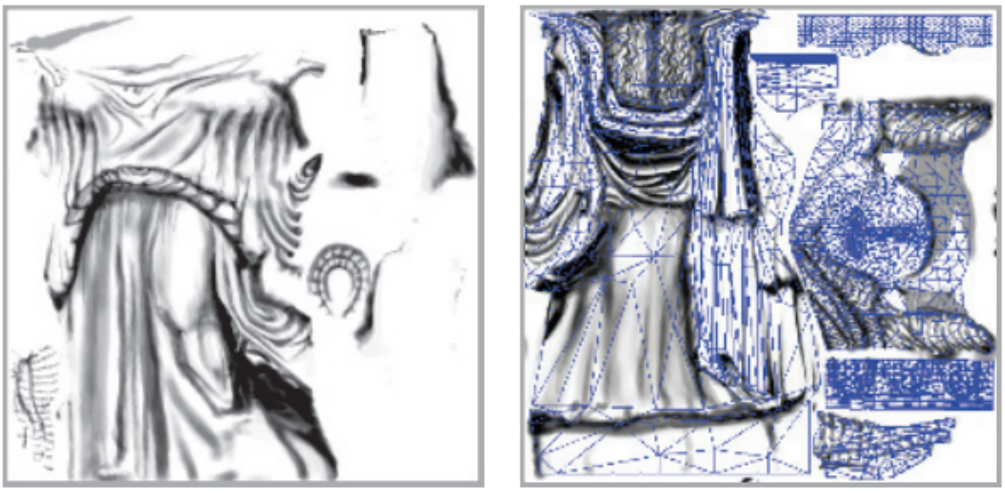


图 6.6：几张用于渲染雕像模型的小纹理，它们被打包存储在两个较大的纹理中。右图展示了这个雕像的三角形网格是如何被展开的，以及是如何将纹理映射到三角形网格上的。

纹理坐标空间并不总是一个二维平面，有时候它也可能是一个三维体积，在这种情况下，纹理坐标会被表示为一个包含三个分量的向量 (u, v, w)，其中 w 是沿着投影方向的深度。有一些系统会使用多达四个坐标，通常是 (s, t, r, q) [885]，其中 q 代表了齐次坐标中的第四个值。它的作用类似于电影放映机或者幻灯片投影机，随着投影距离的增加，投影产生的纹理大小也会相应增加。例如，它可以用于在舞台或者其他表面上，投影一个装饰性的聚光灯图案（被称为 gobo）[1597]。

另一类重要的纹理坐标空间是方向性的，纹理空间中的每个点都需要通过输入方向来进行访问。将这种空间进行可视化的一种方法是，将其作为单位球体上的点，每个点位置上的法线代表了用于访问该位置纹理的输入方向。使用这种方向性参数化的、最常见的纹理类型就是立方体贴图（cube map，章节 6.2.4）。

值得注意的是，一维的纹理图像和投影函数也有各自的用途。例如：对于一个地形模型而言，它的表面颜色可以由该点对应的高度决定，即低地是绿色的，山峰是白色的。线条同样也可以被纹理化，其中一个应用场景是，将雨渲染为一组带有半透明图

像纹理的长线条。这样的纹理也可以用于将一个值转换为另一个值，例如将其作为一个一维查找表（lookup table，LUT）。

由于多个纹理可以被应用到同一个表面上，因此可能需要定义多组纹理坐标。但是无论怎样使用这些纹理坐标，其核心思想都是相同的：这些纹理坐标会在表面上进行插值，并用于检索纹理值。然而，在进行插值之前，这些纹理坐标还需要使用转换函数进行变换。

6.1.2 转换函数

转换函数（corresponder function）用于将纹理坐标转换为纹理空间中的具体位置，它们提高了在表面上应用纹理的灵活性。转换函数的其中一个例子是：使用 API 选择现有纹理中的一部分来进行显示；并且在后续操作中都只会用到这个子图像。

另一类转换函数是矩阵变换，应用于顶点着色器或者像素着色器中，它们允许对表面上的纹理进行平移、旋转、缩放、剪切或者投影操作。正如章节 4.1.5 中所讨论的，这些变换操作的顺序是很重要的。令人惊讶的是，纹理的变换顺序必须与预期的变换顺序相反，这是因为纹理变换实际上是对决定图像可见位置的遮罩空间产生了影响；图像本身并没有被变换，真正发生变换的是定义图像位置的空间。

另一类转换函数控制了图像的应用方式，我们知道，当纹理坐标 (u, v) 在 $[0, 1]$ 范围内时，表面上才会出现图像，但是如果纹理坐标位于这个范围之外呢？转换函数决定了此时会发生什么。在 OpenGL 中，这种类型的转换函数被称为“包装模式（wrapping mode）”；而在 DirectX，则被称为“纹理寻址模式（texture addressing mode）”。这种类型的常见转换函数包括：

- **wrap**（DirectX），**repeat**（OpenGL）或者 **tile**：图像会在表面上进行重复。在算法实现中，纹理坐标的整数部分会被直接丢弃（这使得纹理坐标在 $[0, 1]$ 范围不断重复）。这种模式对于让材质图像在表面上不断重复而言十分有用，并且通常都是默认的模式。
- **mirror**：图像在表面上不断重复，但是每重复一次就会被镜像（翻转）一次。例如：图像在纹理坐标为 0–1 之间时表现正常，然后在 1–2 之间时会进行反转，然后在 2–3 之间时表现正常，然后再反转，以此类推。这种模式可以为纹理边缘提供一些连续性。
- **clamp**（DirectX）或者 **clamp to edge**（OpenGL）：位于 $[0, 1]$ 范围外的纹理坐标会被限制在这个范围内。这种模式会导致图像边缘的不断重复，其的优点在

于：当在纹理边缘附近发生双线性插值时，这个模式可以避免从纹理的相反边缘处采样[885]。

- **border**（DirectX）或者 **clamp to border**（OpenGL）：位于 $[0,1]$ 范围外的纹理坐标所采样到的值，会被设定为同一个颜色，它被称为边框颜色（border color），这个边框颜色是可以自己定义的。这种模式可以很好地将贴花渲染到一个单色表面上，因为纹理的边缘颜色会与这个边框颜色进行平滑地混合。

如图 6.7 所示。每个纹理轴上所分配的转换函数可以是不同的，例如：可以在轴 u 上使用 repeat 模式，而在轴 v 上则使用 clamp 模式。在 DirectX 中还有一个 **mirror once** 模式，它可以对纹理只进行一次镜像操作，然后再使用 clamp 模式，这种特殊模式对于对称贴花而言十分有用。

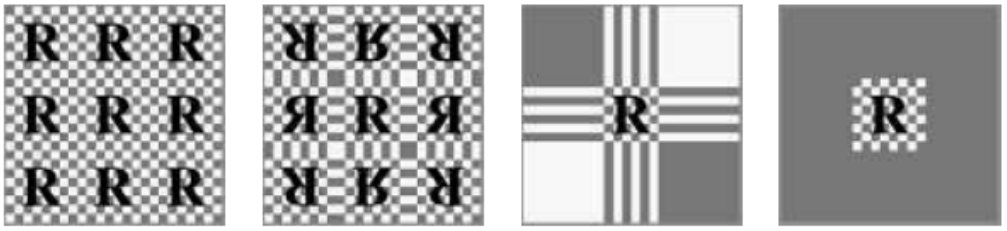


图 6.7：纹理图像的不同寻址模式，从左到右分别是 repeat，mirror，clamp 和 border。

纹理的重复平铺是一种为场景添加更多视觉细节的廉价方法，但是通常来说，这种方法在纹理重复三次之后就看起来不太自然了，因为人眼会识别出这种重复的图案。避免这种周期性（periodicity）问题的一个常见方法是，将纹理值与另一个非重复平铺的纹理相结合。这种方法可以被极大扩展，例如 Andersson [40]所描述的商用地形渲染系统，该系统可以根据地形类型、高度、坡度等因素，对多种纹理进行组合。同时纹理图像也会与几何模型（例如灌木和岩石）在场景中的位置有关。

另一个避免周期性问题的方法是，使用着色器程序来实现一些特殊的转换函数，从而将纹理图案和瓦片（tile）贴图进行随机重组。Wang tiles 就是这种方法的一个例子，它是一组边缘匹配的方形瓦片贴图，在纹理化过程中，会对集合中的瓦片贴图进行随机选择[1860]。Lefebvre 和 Neyret [1016]实现了一个类似的转换函数，来避免纹理图案的重复问题，它使用了依赖纹理读取和纹理表格。

最后一种被应用的转换函数是隐式的，并且与图像的大小有关。纹理通常会应用在 uv 坐标的 $[0,1]$ 范围内。例如砖墙的例子，通过将该范围内的纹理坐标乘以图像的分辨率，便可以得到对应的像素位置。这种将纹理坐标限制在 $[0,1]$ 范围内的优势在

于：可以使用不同分辨率的纹理贴图，而且不需要修改存储在模型顶点中的纹理坐标值。

6.1.3 纹理值

在使用转换函数生成纹理空间坐标之后，便可以使用这个坐标来获取对应的纹理值。对于图像纹理而言，这是通过检索图像中的纹素信息来完成的，我们将在章节 6.2 中详细讨论这个话题。在实时渲染中所使用的绝大多数纹理都是图像纹理，但是也可以使用一些函数来程序化生成纹理的内容。在使用程序化纹理的情况下，从纹理空间位置获得纹理值的过程并不涉及内存查找，而是变成了对一个函数进行计算。有关程序化纹理的内容将在章节 6.3 中进一步描述。

最直接的纹理值就是 RGB 三元组，它可以用于替换或者修改表面的颜色，当然也可以只返回简单的灰度值。另一种可以返回的数据类型是 $RGB\alpha$ ，如章节 5.5 所述。其中的 α 值（alpha）通常代表了颜色的不透明度，它决定了该颜色对像素的影响程度。纹理贴图中不仅仅可以存储颜色数据，还可以存储任何其他类型的数据，例如表面粗糙度等。在章节 6.7 中我们讨论了凹凸映射（bump mapping），届时我们会看到纹理贴图中存储的其他类型数据。

从纹理中返回的值可以在使用之前进行选择性地转换，这些转换一般都是在着色器程序中执行的。一个常见的例子是，将数据从无符号范围（0.0 到 1.0）重新映射到有符号范围（– 1.0 到 1.0）内，它可以用来在纹理贴图中存储法线数据。

6.2 图像纹理

在图像纹理化的过程中，二维图像被有效地附着在一个或者多个三角形的表面上。在上一小节中，我们讨论了如何根据纹理坐标来计算对应的纹理空间位置；现在我们将解决从给定位置的图像纹理中，获取纹理值的相关问题和算法。在本章节的剩余部分中，我们将图像纹理（image texture）将简称为纹理（texture）；此外，当我们提到像素的单元格（pixel's cell）时，它实际上指的是围绕该像素的屏幕网格单元（screen grid cell）。正如章节 5.4.1 中所讨论的那样，像素实际上是一个被显示出来的颜色值，它会（而且应该，为了更好的质量）受到与其相关网格单元之外的样本影响。

在本小节中，我们将特别关注于快速采样和过滤纹理图像的方法。章节 5.4.2 讨论了走样和锯齿问题，尤其是在渲染物体边缘的时候；纹理同样也会遇到采样问题，不同的是，它们会出现在被渲染的三角形内部。

像素着色器可以通过将纹理坐标传给 texture2D 等函数，并调用它们来访问纹理；这些纹理坐标位于 (u, v) 纹理坐标系中，会通过转换函数来将其映射到 $[0.0, 1.0]$ 范围中，然后再由 GPU 负责将这个值转换为纹素坐标。在不同的图形 API 中，纹理坐标系统有两个主要的区别。在 DirectX 中，纹理的左上角对应 $(0, 0)$，右下角是对应 $(1, 1)$，这与大多数图像类型存储数据的方式相匹配，因为位于图像顶部的数据会被存储在图像文件的起始位置；而在 OpenGL 中，左下角的位置对应了 $(0, 0)$，这刚好是将 DirectX 的纹理坐标系，按照 y 轴翻转后的结果。纹素具有整数类型的坐标，但是我们会经常想要访问两个纹素之间的位置，并在它们之间进行插值，这就引出了一个问题：像素中心的浮点坐标到底是什么？Heckbert [692]讨论了两种可能的模式：截断（truncate）和舍入（round）。DirectX 9 将每个纹素的中心定义在 $(0.0, 0.0)$ 处，它采用了舍入方法。但是这个系统稍微有点混乱，因为对于 DirectX 左上角像素（原点）而言，该像素的左上角坐标为 $(-0.5, -0.5)$。DirectX 10 学习了 OpenGL 的纹理坐标系统，让每个纹素的中心值为 $(0.5, 0.5)$，即使用了截断方法，或者更准确地说是向下取整（floor），即小数部分会被丢弃。向下取整是一个更加直观的系统，它可以很好的用语言进行表述，例如：当我们说一个像素位于坐标 $(5, 9)$ 时，实际上我们指的是沿 u 轴方向上从 5.0–6.0 的范围，以及沿 v 轴方向上从 9.0–10.0 的范围。

依赖纹理读取（dependent texture read）是一个值得解释的术语，它包含两个定义。第一个定义是针对移动设备而言的，当我们使用 texture2D 或者类似方式访问纹理，并在像素着色器内手动计算纹理坐标，而不是使用从顶点着色器传入的、未修改的纹理坐标时[66]，就会发生依赖纹理读取。这里提到的手动计算纹理坐标，包括任何对输入纹理坐标的修改，甚至是像交换 u 和 v 这样的简单操作。对于那些不支持 OpenGL ES 3.0 的老旧移动 GPU 而言，在没有依赖纹理读取的情况下会具有更高的效率，因为纹理数据可以被预先读取。另一个定义对于早期的桌面 GPU 十分重要，在这种情况下，当一个纹理坐标依赖于之前的纹理值结果时，就会发生依赖纹理读取。例如：一个纹理可能会改变表面的着色法线，这反过来又会改变用于访问立方体贴图（cube map）的坐标。这种功能在早期的 GPU 上是受限的，甚至是不存在的。如今绝大部分 GPU 都支持了这个功能，但是这样的读取操作可能会对性能产生影响，这取决于在一个 batch 中计算的像素数量，以及其他的一些因素，更多信息详见章节 23.8。

GPU 所使用的纹理尺寸通常为 $2^m \times 2^n$，其中 m 和 n 为非负整数，这样的纹理被称为 2 次幂（power–of–two，POT）纹理。现代 GPU 可以处理任意大小的非 2 次幂（non–power–of–two，NPOT）纹理，这允许将生成的图像也视为纹理。但是一些老旧的移动端 GPU 可能并不支持 NPOT 纹理的 mipmap（章节 6.2.2）。不同的

图形加速器对于纹理尺寸有着不同的上限，例如：DirectX 12 允许一个纹理最多包含 16384^2 个纹素。

假设现在我们有一个尺寸为 256×256 的纹理，并且我们想将其显示在一个正方形上，只要此时屏幕上投影后正方形尺寸与纹理的尺寸大致相同，那么这个正方形上的纹理看起来就几乎与原始图像一模一样。但是设想一下，如果在投影之后，这个正方形所覆盖的像素尺寸是原始图像的十倍（放大，magnification）；又或者在投影之后，正方形只能覆盖屏幕上的一小部分（缩小，minification），这时会发生什么呢？这个问题的答案取决于我们在这两种不同情况下，使用了什么样的采样方法和过滤方法。

本小节所讨论的图像采样方法和过滤方法，将会应用于每个从纹理读取中的值。然而我们所期望的结果是在最终图像中的避免锯齿和走样的出现，在理论上，这需要对最终的像素颜色进行采样和过滤。这里二者的区别在于，是对着色方程的输入进行过滤，还是对着色方程的输出进行过滤。实际上，只要着色方程的输入和输出是线性相关的（例如颜色），那么对单个纹理值的过滤与对最终颜色的过滤其实是等效的。然而，许多存储在纹理中的着色器输入参数，例如表面法线和粗糙度，与着色器的输出之间具有非线性关系，标准的纹理过滤方法不能很好地处理这些纹理，最终会导致出现走样，章节 9.13 将对这种纹理的过滤方法进行了讨论。

6.2.1 放大

在图 6.8 中，一个尺寸为 48×48 的纹理被纹理化到了一个正方形表面上，相对于纹理尺寸而言，这个正方形到相机的距离非常近，因此底层的图形系统需要将这个纹理进行放大（magnification）。最常见的放大过滤技术是邻近过滤（nearest neighbor，实际使用了 box 滤波器，详见章节 5.4.1）和双线性插值（bilinear interpolation）。还有一种方法叫做三次卷积插值（cubic convolution），它使用了 4×4 或者 5×5 范围纹素的加权和，它可以实现更高质量的放大。虽然现在的硬件并不原生支持三次卷积插值（也被称为双三次插值，bicubic interpolation），但是它可以在着色器程序中进行执行。

图 6.8：将 48×48 的纹理放大到 320×320 。左：邻近过滤，每个像素都会选择距离其最近的纹素。中间：双线性过滤，使用四个最近像素的加权平均值。右：立方滤波（三次滤波），使用 5×5 范围内最近像素的加权平均值。

在图 6.8 的最左侧，我们使用了邻近过滤的方法。这种放大技术的一个特点是，单个纹素可能会变得十分明显。这种效果被称为像素化（pixelation）；因为该方法在放大的时候，会选取距离每个像素中心最近的纹素，从而产生了块状外观。虽然这种方法的质量有时会很差，但是它的好处在于，只需要为每个像素获取一个纹素即可。

在图 6.8 的中间，我们使用了双线性插值（有时也会叫做线性插值）方法。对于每个像素而言，这种过滤方法需要找到四个相邻的纹素，并在二维上进行线性插值，从而获得混合后的像素值。虽然双线性插值的结果是比较模糊的，但是它并不会像邻近过滤那样出现锯齿。你可以做一个简单的小实验，尝试眯着眼睛来看左边的图像，你就会发现图像的锯齿也消失了，因为这样做（眯着眼睛观察）的效果其实和低通滤波器是大致相同的，并且更能展示面部的特征。

这里我们回到本章一开始提到的砖块纹理例子：在不舍弃小数的情况下，我们会获得坐标 $(p_u, p_v) = (81.92, 74.24)$ 。这里我们使用与 OpenGL 同样的纹理坐标系，其原点位于左下角，它与标准的笛卡尔坐标系是相匹配的。我们的目标是在四个最近的纹素中心之间，建立一个局部坐标系，并在这四个纹素中心之间进行插值，最终获得该点的像素值，如图 6.9 所示。为了找到 4 个最近的相邻像素，我们从采样位置减去像素中心的分数部分 $(0.5, 0.5)$ ，得到 $(81.42, 73.74)$ 。在去掉中心的小数部分之后，距离最近的 4 个像素范围即为 $(x, y) = (81, 73)$ 到 $(x + 1, y + 1) = (82, 74)$ 。在这个例子中，分数部分 $(0.42, 0.74)$ 是该采样点在这个局部坐标系（由相邻的四个纹素中心构成）中的位置，我们将这个位置表示为 (u', v') 。

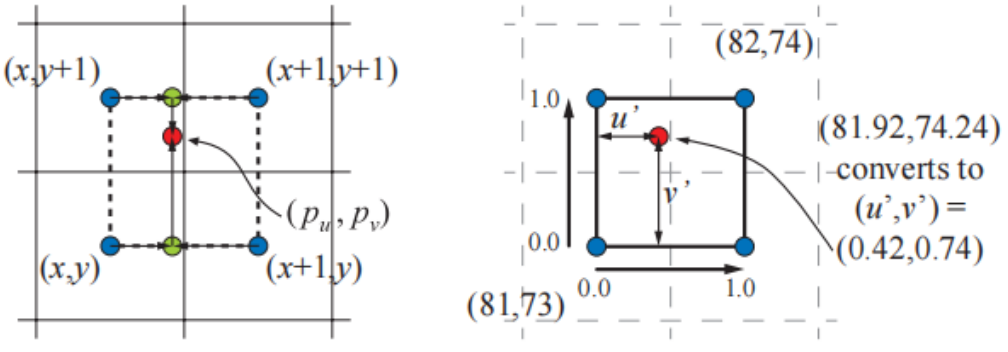


图 6.9：双线性插值。涉及的四个纹素由左边的四个正方形表示，其中蓝点代表了纹素的中心点。右边则展示了由四个纹素中心所形成的局部坐标系。

这里我们将纹理访问函数定义为 $t(x, y)$，该函数会返回对应纹素的颜色，其中 x 和 y 是整数。那么任意位置 (u', v') 的双线性插值颜色可以按照以下两步进行计算：首先，使用下方的两个纹素颜色 $t(x, y)$ 和 $t(x+1, y)$，按照参数 u' 进行插值，即 $(1-u')t(x, y) + u't(x+1, y)$；再使用上方的两个纹素颜色 $t(x, y+1)$ 和 $t(x+1, y+1)$，按照参数 u' 进行插值，即 $(1-u')t(x, y+1) + u't(x+1, y+1)$，如图 6.9 左侧的绿色圆圈。然后对这两个值在竖直方向上，按照参数 v' 进行插值，即将上述过程结合起来，最终 (p_u, p_v) 处的双线性插值颜色 $\mathbf{b}$ 为：

$$\begin{aligned}\mathbf{b}(p_u, p_v) &= (1-v')\left((1-u')\,\mathbf{t}(x, y) + u'\mathbf{t}(x+1, y)\right) \\ &\quad + v'\left((1-u')\,\mathbf{t}(x, y+1) + u'\mathbf{t}(x+1, y+1)\right) \\ &= (1-u')\,(1-v')\,\mathbf{t}(x, y) + u'\,(1-v')\,\mathbf{t}(x+1, y) \\ &\quad + (1-u')\,v'\mathbf{t}(x, y+1) + u'v'\mathbf{t}(x+1, y+1).\end{aligned} \tag{6.1}$$

从直观上来说，距离采样位置越近的纹素，对其最终颜色值的影响也就越大，这也是我们在方程 6.1 中所描述的。右上角 $(x+1, y+1)$ 处的纹素对其的影响是 $u'v'$。注意这个等价关系：右上角纹素的影响力，等于左下角纹素与采样点之间所形成的矩形面积。回到我们的例子中，这意味着从该纹素上检索到的像素值，会被乘以 0.42×0.74，即 0.3108。从右上角纹素开始，按照顺时针顺序，其他三个纹素会被分别乘以 0.42×0.26， 0.58×0.26 和 0.58×0.74，这四个权重的和为 1.0。

一种用于解决放大模糊的常见方法是使用细节贴图（detail texture）。这个纹理代表了表面上的精细细节，例如手机上的划痕和地形上的灌木丛等。这些细节会作为一个独立的纹理贴图，以不同的尺度被覆盖在放大后的纹理上。细节贴图中包含了高频的重复图案，它与低频的放大纹理相结合，在视觉效果上类似于使用单张高分辨率的纹理。

双线性插值会在两个方向上进行线性插值，但是有些情况下我们其实并不需要线性插值。例如：一个纹理是由类似于棋盘格的黑白像素组成的，使用双线性插值使得原本的黑白结果变成了平滑的灰度值。这时候我们需要对结果进行通过重新映射，例如我们可以让所有低于 0.4 的灰色都变成黑色，所有高于 0.6 的灰色都变成白色，而那些介于两者之间的灰色，则被拉伸用以填补空白，使得纹理看起来更像是原本的棋盘格，同时也在纹理之间进行了一些混合过渡，如图 6.10 所示。

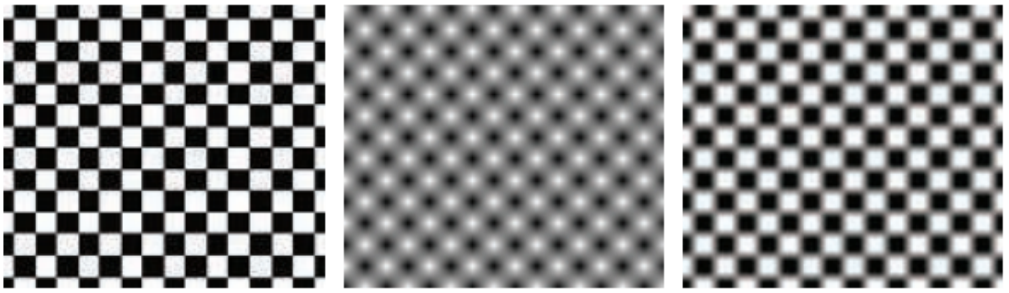


图 6.10：从左到右分别是邻近过滤，双线性插值，和部分重新映射的结果，原本的纹理是一个 2×2 的黑白棋盘格。需要注意的是，由于纹理和图像网格并不是完美匹配的，因此邻近过滤方法所生成的正方形大小略有不同。

使用一个更高分辨率的纹理也会出现类似重新映射的效果。例如：假设现在每个黑白方格都由 4×4，总计 16 个像素组成（原本是 1×1），那么在每个方格的中心周围，插值出来的颜色也将是全黑或者全白的。

在图 6.8 的右侧，我们使用了双三次插值（bicubic filter），它大幅去除了剩余的方块感。需要注意的是，双三次插值比双线性插值的计算成本更高，但是许多的高阶滤波器都可以被表示为重复的线性插值[1518]（详见章节 17.1.1），因此可以通过若干次简单的线性插值，来充分利用纹理单元中用于线性插值操作的 GPU 硬件。

如果我们觉得双三次插值的计算成本太大的话，那 Quilez [1451]提出了一种更加简单的技术，它在一组 2×2 纹素之间，使用了一个平滑的曲线来进行插值。接下来我们首先会先介绍这个平滑曲线，然后再描述这个技术的详细过程。最常用的两个平滑曲线分别是 smoothstep 曲线和 quintic（五次）曲线[1372]，它们的数学表达如下：

$$\underbrace{s(x) = x^2(3-2x)}_{\text{smoothstep}} and \underbrace{q(x) = x^3\left(6x^2 - 15x + 10\right)}_{\text{quintic}} \tag{6.2}$$

当我们想从一个值平滑插值到另外一个值时，这些函数是非常有用的。smoothstep 曲线具有 $s'(0) = s'(1) = 0$ 的性质（C^1 连续），它在 0–1 之间是非常平滑的。quintic 曲线具有类似的性质，唯一不同是 $q''(0) = q''(1) = 0$（C^2 连续），即曲线在开始和结束处的二阶导数也是 0。这两条曲线如图 6.11 所示。

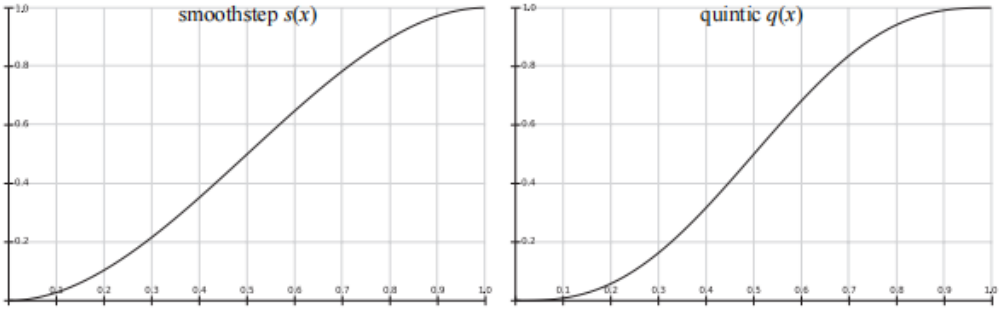


图 6.11：smoothstep 曲线 $s(x)$（左侧）和 quintic 曲线 $q(x)$（右侧）。

该方法首先会计算出位置 (u', v')（与方程 6.1 和图 6.9 中使用相同的方法），然后将采样位置与纹理尺寸相乘并加上 0.5。结果的整数部分暂时先保留，小数部分会存储在 u' 和 v' 中，它们的取值范围是 $[0, 1]$。然后再将 (u', v') 变换为 $(t_u, t_v) = (q(u'), q(v'))$，这个结果仍然在 $[0, 1]$ 的范围内；将这个结果减去 0.5，再加上原来的整数部分。然后将得到的坐标 u 和坐标 v 分别除以纹理宽度和纹理高度。此时，会将这个新的纹理坐标作为参数，传给 GPU 提供的双线性插值查找函数。需要注意的是，这种方法在每个纹素上只会给出一个值，这意味着如果纹素位于 RGB 空间平面上的话，那么这种类型的插值将给出一个平滑的，但仍然是阶梯状的外观，这可能并不总是我们想要的，如图 6.12 所示。

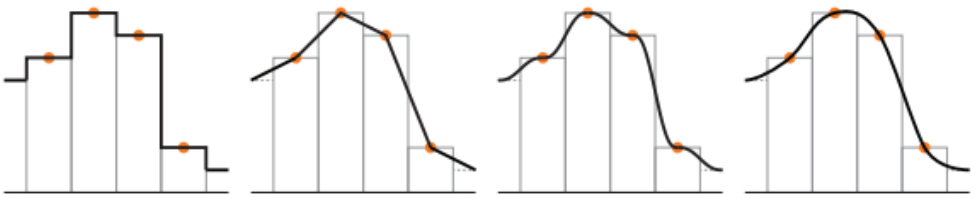


图 6.12：放大一维纹理的四种不同方式。其中橙色圆圈代表了纹素中心以及对应纹理值（高度）。从左到右分别是：邻近过滤、线性插值、在相邻纹素之间使用 quintic 曲线、三次插值。

6.2.2 缩小

当纹理被压缩时，平面上的一个像素单元格可能会占据好几个纹素，如图 6.13 所示。为了正确获得这个像素的颜色值，我们应当将这几个纹素对像素的影响整合起来。然而，精确确定某个像素附近所有纹素对其的影响是很难的，而且想要以实时的速度来完美地实现这一点几乎是不可能的。

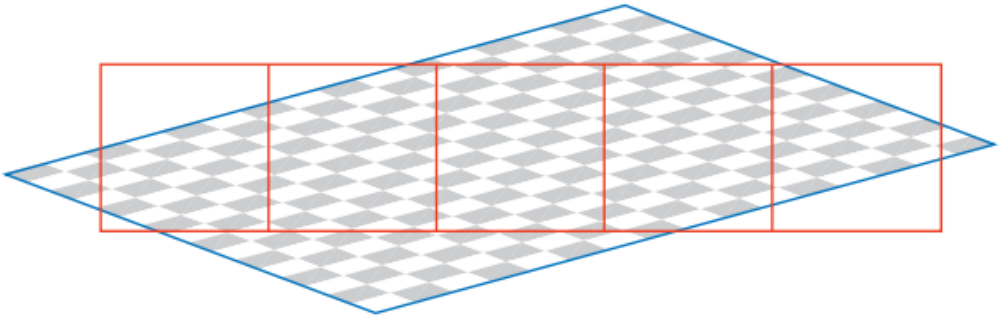


图 6.13：缩小（minification）：通过一排像素来观察一个具有棋盘格纹理的正方形，大致显示了每个像素是如何被多个纹素所影响的。

由于上述的一些限制，因此在 GPU 上使用了一些特殊方法来解决这个问题。其中最简单的方法便是使用邻近过滤（nearest neighbor），其工作原理与用于纹理放大的滤波器完全相同，即直接选择位于像素单元中心可见的纹素值，来作为自身的像素值，但是这个方法可能会产生严重的锯齿问题，如图 6.14 第一行的图片。因为这种方法只是在影响像素的众多纹素中，选择一个纹素来代表该点的颜色，当表面相对于相机发生移动的时候，这种瑕疵会变得更加明显，这种在运动中所产生的瑕疵也被称为时域锯齿（temporal aliasing）。

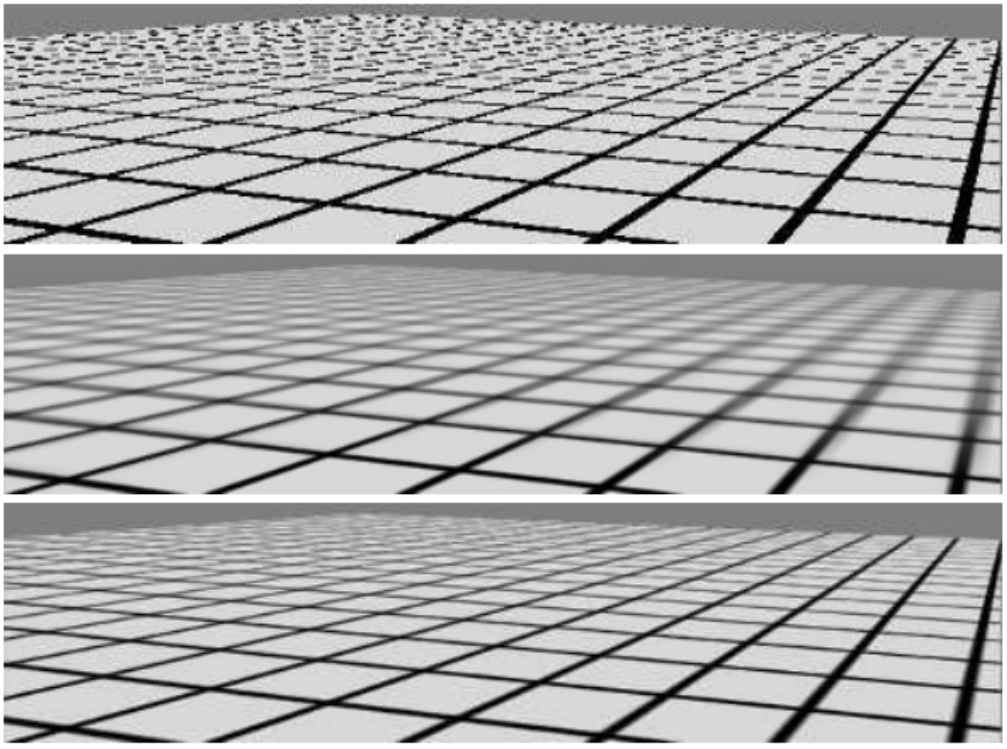


图 6.14：第一行图像使用了点采样（邻近过滤），第二行图像使用了 mipmap，第三行图像使用了 SAT。

另一个常用的滤波器是双线性插值，其工作原理与上文中的放大滤波器完全相同；但是对于纹理缩小而言，这种方法只比邻近过滤稍微好一点，因为它只能将四个纹素对于像素的影响混合起来；而当一个像素受到超过四个纹素的影响时，双线性插值就会很快失效并产生锯齿。

还有一些更好的解决方法，正如章节 5.4.1 中所讨论的，走样（锯齿）问题可以通过采样技术和滤波技术来解决。纹理的信号频率取决于纹素在屏幕上的间隔距离，根据 Nyquist 极限，我们只需要确保纹理的信号频率不大于采样频率的一半即可。例如：假设现在有这样一个图像，它由黑白相间的线条组成，其中每个线条占据了两个纹素，即频率为 $\frac{1}{2}$ 。为了在屏幕上能够正确显示这个纹理，那么采样频率必须至少为 $2 \times \frac{1}{2} = 1$ ，即至少一个像素。因此对于纹理而言，一般一个像素应当最多对应一个纹素，这样可以避免走样和锯齿。

根据上面我们的分析，为了实现这个目标，我们要么提高像素的采样频率，要么降低纹理的信号频率。上一章节中我们所讨论的抗锯齿方法，主要是通过提高像素采样率来实现的；但是这些方法能够增加的采样率是有限的。为了更加充分地解决这个问题，我们需要降低纹理的频率，因此提出了各种纹理缩小算法。

所有纹理抗锯齿算法背后的基本思想都是相同的：对纹理进行预处理，创建某种数据结构，从而快速近似计算一组纹素对像素的影响。对于实时渲染而言，这些算法在执行过程中具有使用固定时间开销和资源开销的特点。通过这种方式，每个像素会采集固定数量的样本，并将这些结果组合起来，从而计算大量纹素对像素的影响。

Mipmap

mipmap [1889]是最流行的纹理抗锯齿方法，现如今所有的图形加速器都会支持这种方法。其中 Mip 是拉丁语（multum in parvo）的缩写，它的意思是“一个很小的地方上有很多东西”，这是一个很好的名字，因为 mipmap 的过程其实就是将原始纹理反复过滤成更小的图像。

在使用 mipmap 滤波器的时候，在实际渲染发生之前，原始纹理图像会生成一系列较小尺寸的版本。原始纹理（第 0 级）会被下采样到原始尺寸的四分之一，每个新生成的纹素值，通常为原始纹理中四个相邻纹素的平均值，这个新生成的纹理（第 1 级）有时也会被叫做原始纹理的子纹理（subtexture）。这个下采样的过程会被递归执行，直到最终生成的某个纹理的维度为 1。这个过程如图 6.15 所示，这组图片的集合通常被称为一个 mipmap 链（mipmap chain）。

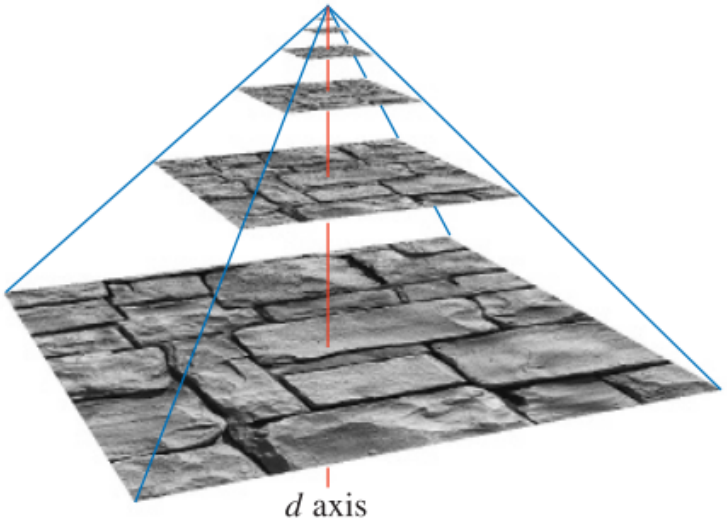


图 6.15：金字塔的底部是原始纹理图像，它对应了 mipmap 的第 0 级，将每个 2×2 区域内纹素的平均值作为下一级别的 mipmap。纵轴是第三个纹理坐标 d，在这个图中，d 并不是线性的，它用于在两个纹理级别之间进行插值。

生成高质量 mipmap 的两个重要因素分别是：使用良好的过滤和伽马校正。生成 mipmap 的常用方法是将每 2×2 的纹素进行平均，从而获得下一级 mip 所对应的纹素值。具体使用的是一个 box 滤波器，虽然这可能是最糟糕的一个滤波器，使用 box 滤波器可能会导致较差的质量，因为它会对低频信息进行模糊，同时保留一些会产生锯齿的高频信息[172]。最好是使用高斯、Lanczos、Kaiser 或者类似的滤波器，这些滤波器的源代码基本都有免费高效的开源实现[172, 1592]，同时有一些 API 还支持在 GPU 上进行过滤操作。在靠近纹理边缘进行过滤的时候的地方，需要注意纹理的包装模式（wrapping mode）。

对于在非线性颜色空间中进行编码的纹理（例如大多数的彩色纹理），在过滤时忽略伽玛校正会修改该层级 mipmap 的感知亮度[173, 607]。如果使用了未校正的 mipmap，相机距离物体越远，物体整体看起来就会越暗，对比度和表面细节也会受到影响。由于这个原因，因此将这种纹理（例如颜色纹理）从 sRGB 颜色空间转换到线性颜色空间是十分重要的（章节 5.6），我们会在线性空间中完成 mipmap 的生成和过滤，然后将生产的结果转换回 sRGB 颜色空间中并进行存储。大多数图形 API 都支持 sRGB 纹理，因此可以在线性空间中正确生成 mipmap，并将结果存储在 sRGB 中。当访问 sRGB 纹理的时候，它们首先会被转换到线性空间中，以便正确地执行放大（magnification）和缩小（minification）操作。

我们在之前提到，一些纹理参数与最终的着色颜色之间具有非线性关系，这会给过滤带来一些问题，而 mipmap 的生成对这个问题特别敏感，因为在这个过程中，需要对

上百或者上千个像素进行过滤。为了获得最佳的结果，通常需要一些专门的 mipmap 生成方法。这些方法将会在章节 9.13 中进行详细介绍。

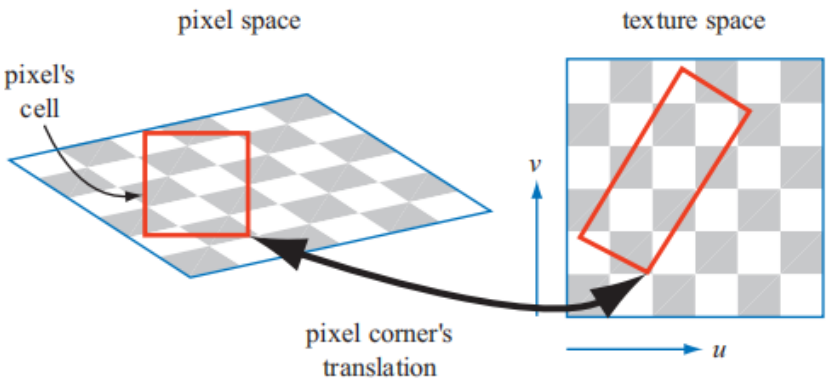


图 6.16：左边是一个正方形的像素单元格及其它的纹理视图。右边是这个像素单元格在纹理上的投影。

在纹理化时访问 mipmap 的过程也很简单。由于屏幕像素占据了纹理中的一个区域，因此当某个像素被投影到纹理上时（如图 6.16 所示），它会包含一个或者多个纹素。这里使用像素单元格的边界并不是严格正确的，使用这种方式只是为了简化表示。位于单元格外的纹素也会对像素的颜色产生影响，详见章节 5.4.1，这里的目标是大致确定纹素对像素的影响程度。对于第三个纹理坐标 d （在 OpenGL 称之为 λ ，也称为纹理细节级别，texture level of detail），有两种常用的计算方法：一种是利用像素单元格投影后所形成的四边形，取其中较长的那个边来对像素的覆盖范围进行近似[1889]；另一种方法是使用四个梯度中绝对值最大的那个作为度量，这四个梯度分别是 $\partial u/\partial x, \partial v/\partial x, \partial u/\partial y, \partial v/\partial y$ [901, 1411]，它们代表了纹理坐标相对于屏幕轴向的变化量，例如 $\partial u/\partial x$ 代表了一个像素所对应的纹理值 u ，沿着屏幕 x 轴的变化量。你可以在 Williams [1889]、Flavell [473]或者 Pharr [1411]的论文中了解更多有关这些方程的内容。McCormack 等人[1160]讨论了采用最大绝对值方法所引入的走样，并提出了一个替代方程。Ewins 等人[454]对几种质量相当的算法，以及它们所对应的硬件成本进行了分析。

在 shader Model 3.0 或者更新版本的像素着色器中，可以直接使用这些梯度值。由于这些梯度基于相邻像素值之间的差异，如果像素着色器中包含了受动态流程控制（章节 3.8）的部分，那么这部分是无法访问这些梯度信息的。如果我们想要在这部分（例如一个循环）中读取纹理的话，那么就必须提前计算梯度。需要注意的是，顶点着色器并不能访问梯度信息，当需要顶点纹理化操作的时候，才会在顶点着色器中计算梯度或者细节级别等信息，并提供给 GPU，以供后续阶段使用。

计算第三个纹理坐标 d 的目的是确定沿着 mipmap 金字塔轴进行采样的层级，如图 6.15 所示。我们的目标是使得像素与纹素的比例至少为 $1:1$，以达到 Nyquist 极限。这里计算坐标 d 的重要原则是，当一个像素单元格内包含多个纹素时，就需要增大 d，从而访问尺寸更小，更模糊的 mipmap 层级。我们使用三元组 (u, v, d) 来访问纹理的 mipmap，其中 d 类似于纹理级别，但 d 并不是一个整数值，而是级别之间距离的分数值。我们会对 d 两侧的 miamap 分别进行采样，即使用坐标 (u, v) 来从这两个 mipmap 中分别进行双线性插值，获得两个纹理值。最后按照参数 d 再对这两个纹理值进行一次线性插值。整个过程被称为三线性插值（trilinear interpolation），并且会逐像素地执行。

用户可以通过细节层次偏移（level of detail bias，LOD bias），来对坐标 d 进行一定的控制，这是一个加在坐标 d 上的偏移量，它影响了纹理的相对感知锐度。如果我们将 mipmap 金字塔向上移动（即增大 d），那么纹理会看起来更加模糊。对于任何给定的纹理，根据图像类型和使用方式的不同，良好的 LOD 偏移也是不同的。例如：对于有些模糊的第 0 级 mipmap 图像，可以使用一个负偏移量；而对于过滤不良（产生锯齿）的纹理图像则可以使用一个正偏移量。这个偏移量可以针对整个纹理进行指定，也可以在像素着色器中逐像素指定。为了获得更加精细的控制，可以由用户来提供坐标 d，或者是提供用于计算它的梯度。

mipmap 的好处在于，它并不是去单独计算每个纹素对像素的影响，而是对预先生成的纹素集合进行访问和插值，无论纹理压缩的程度如何，这个过程的时间开销是固定的。然而，mipmap 也存在几个缺陷[473]，其中一个主要的问题就是过度模糊（overblurring）。我们假设现在有一个像素单元格，它在 u 方向上覆盖了大量的纹素，而在 v 方向上只覆盖了少量的纹素，这种情况通常发生在相机以一个掠射角度来观察纹理表面的时候。在这种情况下，需要沿着纹理的其中一个轴进行缩小，沿着另一个轴进行放大，这会导致像素在纹理上的投影区域是一个长宽比很大的矩形；而我们在访问 mipmap 时，只能检索纹理上的正方形投影区域，无法检索矩形投影区域。为了避免走样，我们会选择较长的那个边所形成的正方形，来作为对像素单元格覆盖率的近似度量，这导致检索到的样本往往会相对模糊。这种现象可以在图 6.14 的 mipmap 图像中看到，图片右侧向远处延申的线条会变得过度模糊。

Summed-Area 表（SAT）

另一种能够避免过度模糊的方法是面积积分表（summed-area table，SAT，也可以叫做求和面积表）[312]，后文中我们会简称为 SAT。想要使用这种方法，首先要创建一个尺寸与纹理相同的数组，但是颜色存储的精度要更高（例如：每个红绿蓝颜色分量都会占据 16 个 bit）。

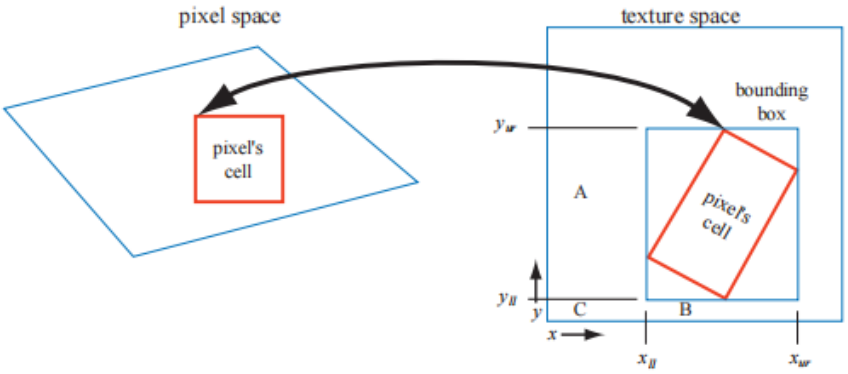


图 6.17：像素单元格被反向投影到纹理上，并被一个轴对齐矩形包围盒所包围，这个包围盒的四个角会用于访问 SAT。

在数组中的每个位置上，该位置上的纹素会和 $(0,0)$ 处的纹素（原点）构成一个矩形，计算并存储区域中所有纹素值的总和。在纹理化的过程中，屏幕上像素在纹理上的投影区域是一个矩形；然后会通过 SAT 来确定这个矩形区域的平均颜色，并将其作为该像素的纹理颜色。这个计算过程如图 6.17 所示，具体的平均颜色计算公式如下：

$$\mathbf{c} = \frac{\mathbf{s}\left[x_{ur}, y_{ur}\right] - \mathbf{s}\left[x_{ur}, y_{ll}\right] - \mathbf{s}\left[x_{ll}, y_{ur}\right] + \mathbf{s}\left[x_{ll}, y_{ll}\right]}{\left(x_{ur} - x_{ll}\right)\left(y_{ur} - y_{ll}\right)} \tag{6.3}$$

其中的 x 和 y 代表了矩形的纹理坐标，$\mathbf{s}\left[x, y\right]$ 代表了该坐标所对应的 SAT 值。这个方程的原理是：首先获取右上角到原点这个大矩形的 SAT 值，然后再根据相邻矩形顶点，获得两个小矩形 A 和小矩形 B 的 SAT 值，并将它们减去；其中右下角区域 C 的 SAT 值被减去了两次，因此最后还要再加上一个区域 C 的 SAT 值。请注意，坐标 (x_{ll}, y_{ll}) 位于区域 C 的右上角，即坐标 $(x_{ll}+1, y_{ll}+1)$ 是包围盒的左下角。

图 6.14 的第三行使用了 SAT 方法，图像右侧向远处地平线延申的线条变得更加清晰了，但是中间对角相交的线条仍然是很模糊的。这个问题的原因在于，当我们沿着对角线观察纹理的时候，像素投影所生成的区域是一个沿对角线的细长矩形，该矩形对应的包围盒中包含了大量无关的纹素，例如：在图 6.17 中，想象此时像素的投影区域是一个横跨纹理对角线的细长区域，它所对应的包围盒几乎会占据整个纹理，而真正位于像素投影区域内的纹素数量则很少。此时这种方法会对整个纹理矩形进行平均，这个结果包含了大量的无关纹素值，从而导致模糊的产生。

SAT 是各向异性过滤（anisotropic filtering）算法[691]的其中一个例子，这类算法用于检索非正方形投影区域的纹理值，SAT 对于接近水平方向或者竖直方向的投影区域最为有效。还需要注意的是，对于 16×16 或者尺寸更小的纹理，SAT 需要至少

两倍的内存；而对于尺寸更大的纹理，则需要更高的存储精度，因为像素值的和会很大，精度过低可能会导致数值溢出。

SAT 可以提供更好的质量，并且额外的内存开销还算合理，因此它在现代的 GPU 上也被广泛应用[585]。高质量的过滤方法对于高级渲染技术的质量而言至关重要。例如，Hensley 等人 [718, 719]提出了一个高效的实现，并展示了使用 SAT 采样来改善 glossy 反射的方法。其他使用区域采样的算法也可以通过 SAT 方法进行改进，例如如景深[585, 719]，阴影贴图[988]，和模糊反射[718]等。

无约束的各向异性过滤

对于目前的图形硬件而言，想要进一步改进纹理过滤的质量，最常见的方法就是重用现有的 mipmap 硬件。其基本思想是将像素单元格反向投影到纹理上，然后再对纹理上的四边形区域进行多次采样，最后将采样的结果整合在一起，作为该像素的颜色。在之前我们所提到的方法中，会在 mipmap 中的一个正方形区域内进行采样，这可能会导致采样到很多无关纹素，使得表面变得模糊。这里我们将要介绍的算法，并不是使用单个 mipmap 采样区域来对该投影形成四边形进行近似，而是会使用多个正方形来进行近似。我们使用四边形中较短的那个边来确定 d 的值（而在原始的 mipmap 中，通常会使用较长的边来确定 d ），这样做会使得每个 mipmap 样本的平均面积更小（包含了更少的无关像素，因此会减少模糊的出现）。而四边形的长边则被用来创建一条与其平行，并且穿过四边形中点的各向异性线（line of anisotropy）。当各向异性的比例在 $1:1$ 和 $2:1$ 之间时，我们会沿着这条线取两个样本（如图 6.18 所示）；各向异性的比例越高，沿轴采集的样本就越多。

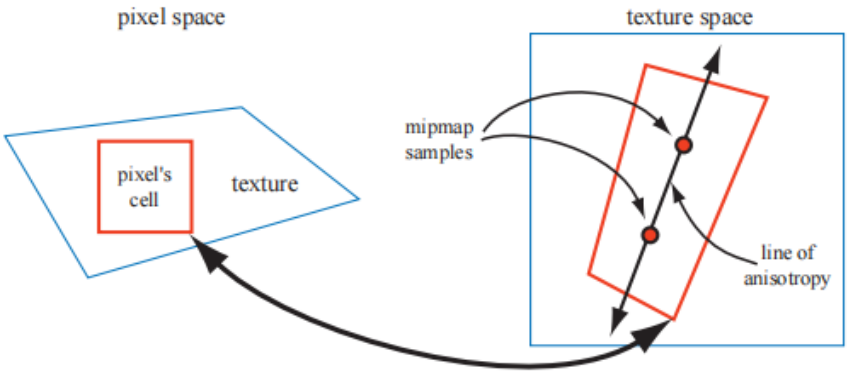


图 6.18：各向异性过滤。像素单元格的反向投影会形成一个四边形，在较长的边之间构建一条各向异性线。

这种方法对各向异性线的方向没有要求，因此它并不会出现类似 SAT 那样的限制。而且它也不需要比 mipmap 使用更多的纹理内存，因为它只是在 mipmap 算法的基础上，对采样方法进行了改进而已，图 6.19 展示了一个各向异性过滤的例子。

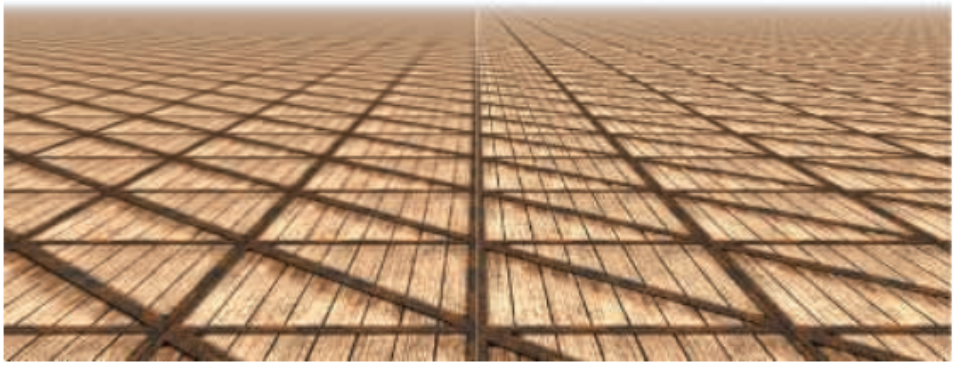


图 6.19：mipmap 和各向异性之间的对比。左侧使用了三线性插值的 mipmap，右侧使用了 $16:1$ 的各项异性 mipmap。在向地平线延申的方向上，各向异性过滤可以提供更加清晰的结果与最少的锯齿。[218]

这种沿轴采样的想法最初是由 Schilling 等人提出的，并应用在了他们的 Texram 动态存储设备中[1564]。Barkans 描述了该算法在 Talisman 系统[103]中的应用；McCormack 等人[1161]提出了一个名为 Feline 的类似系统。Texram 的原始方法是，让沿着各向异性轴的采样点 （也称为探针）具有相同的权重，而 Talisman 系统则只为轴两端的两个探针赋予一半的权重；Feline 系统使用一个高斯滤波核来对一组探针进行加权。这些算法的质量接近于软件采样算法，例如椭圆加权平均（Elliptical Weighted Average，EWA）滤波器，这个滤波器会将像素的影响区域，转换为纹理上的一个椭圆区域，并通过滤波核来对椭圆内的纹素进行加权[691]。Mavridis 和 Papaioannou 提出了几种可以在 GPU 上，使用着色器代码实现 EWA 过滤的方法[1143]。

6.2.3 体积纹理

对图像纹理直接进行扩展可以得到三维图像数据，它通过坐标 (u, v, w) 或者 (s, t, r) 来进行访问，例如：医学成像数据可以生成三维网格，通过在网格中移动成像平面，可以看到这些数据的二维切片。一个类似的想法是，使用这种数据形式来表示体积光，通过在体积内部找到该位置所对应的值，并结合光照方向，可以计算出表面上一点的光照。

如今大部分 GPU 都支持体积纹理（volume texture）的 mipmap，由于在体积纹理的单个 mipmap 级别内，需要使用三线性插值来进行过滤，因此在不同 mipmap 级别之间，需要四线性插值（quadrilinear interpolation）来进行过滤。由于需要对 16 个纹素的结果进行求平均，因此可能会导致一些精度不足的问题，这可以通过使用更

高精度的体积纹理来进行解决。Sigg 和 Hadwiger [1638]讨论了这个问题以及其他与体积纹理相关的问题，并提供了用于过滤和其他操作的高效方法。

虽然体积纹理对于存储空间的要求比较高，并且过滤的计算成本也比较高，但它确实具有一些特殊的优势。由于可以直接使用纹理坐标来表示三维的空间位置，因此可以跳过为三维网格寻找一个良好二维参数化表示的复杂过程（UV 拆分）。这避免了二维参数化时经常出现的扭曲和接缝问题。体积纹理也可以用来表示木材或者大理石等材质的体积结构，具有这种纹理的模型，看起来就像是使用这种材料雕刻出来的一样。

使用体积纹理来对表面进行纹理化操作是非常低效的，因为体积纹理中的绝大部分样本都没有被使用。Benson 和 Davis [133]以及 DeBry 等人[334]，讨论了将纹理数据存储在稀疏八叉树中的方法，这种方法非常适合交互式的三维绘画系统，因为我们在创建表面的时候，不需要显式地指定它的纹理坐标，同时八叉树结构可以将纹理细节保留到任何我们想要的级别。Lefebvre 等人[1017]讨论了在现代 GPU 上实现八叉树纹理的细节；Lefebvre 和 Hoppe [1018]提出了一种将稀疏体积数据打包成较小纹理的方法。

6.2.4 立方体贴图

另一种类型的纹理叫做立方体纹理（cube texture）或者立方体贴图（cube map），它具有六个正方形的纹理，立方体的六个面分别对应了这个六个正方形纹理。访问立方体贴图需要使用一个包含三个分量的纹理坐标向量，这个向量代表了从立方体中心向外发射的射线方向。这个射线与立方体交点的计算过程如下：向量中绝对值最大的那个分量，决定了射线会射向哪个立方体表面（例如：向量 $(-3.2, 5.1, -8.4)$ 代表了射线会射向 $-z$ 面）。将剩余的两个坐标分量分别除以最大分量的绝对值（即 8.4 ），此时这两个分量的大小位于 $[-1, 1]$ 内，然后再将其重新映射到 $[0, 1]$ 中以计算纹理坐标，例如：坐标 $(-3.2, 5.1)$ 会被映射为 $((-3.2/8.4 + 1)/2，(5.1/8.4 + 1)/2) \approx (0.31, 0.80)$ 。立方体贴图对于表示方向函数的值而言非常有用；它们最常用于环境映射中（章节 10.4.3）。

6.2.5 纹理表示

在应用程序中处理大量纹理的时候，有一些方法可以提高性能表现，有关纹理压缩的内容会在章节 6.2.6 中进行介绍，而本小节的重点则是纹理图集（texture atlas），纹理数组（texture array）以及无绑定的纹理（bindless textures），所有这些技术的目的都是为了在渲染过程中避免纹理的切换，因为切换纹理是有一些额外开销的。

在章节 19.10.1 和章节 19.10.2 中，我们还会介绍了纹理流（texture streaming）和纹理转码（texture transcoding）。

图 6.20：左图：纹理图集，其中 9 个较小的图像被组合为一个大纹理。右图：一种更加现代的方法是将较小的图像设置为纹理数组，大多数现代 API 中都有类似的概念。

为了能够使得 GPU 批量处理尽可能多的任务，一般来说最好尽可能地避免改变它的状态（章节 18.4.2）。为此，可以将多个图像放入一个尺寸更大的纹理中，这个纹理被叫做纹理图集，如图 6.20 左侧所示。这里需要注意的是，图集中子纹理的形状和尺寸可以是任意的（如图 6.6 所示），Noll 和 Stricker [1286]提出了用于优化子纹理布局的方法。在生成和访问图集 mipmap 的时候也需要格外当心，因为 mipmap 的上一层级中可能会包含几个独立的、不相关的形状；Manson 和 Schaefer [1119]提出了一种通过考虑表面参数化来优化 mipmap 创建的方法，该方法可以生成更好的结果。Burley 和 Lacewell [213]提出了一个叫做 Ptex 的系统，在该系统中，位于细分表面的每个四边形，都具有属于自己的迷你纹理，这样做的优点在于，避免了在网格上分配唯一的纹理坐标，并且在纹理图集不相连接部分的接缝处也不会出现瑕疵。为了能够实现跨四边形的过滤操作，Ptex 系统使用一些邻接数据结构（adjacency data structure）。虽然 Ptex 系统最初的目标是用于渲染，但是 Hillesland [746]提出了 packed Ptex，它将每个表面的子纹理都放入一个纹理图集中，在过滤的时候使用相邻表面作为填充从而避免间接取值。Yuksel [1955]提出了网格颜色纹理（mesh color texture），该方法对 Ptex 进行了改进。Toth [1780]实现了一种方法，即当 filter tap 超出 $[0,1]^2$ 的范围时，则将它们丢弃，从而为 Ptex-like 的系统提供了高质量的跨表面过滤。

当使用 wrapping/repeat 或者 mirror 模式的时候，是无法使用纹理图集的，因为这些模式会对整个纹理产生影响，导致我们无法对子纹理进行正确的设置。另一个问题

发生在为图集生成 mipmap 时，图集中的子纹理可能会与另一个子纹理相互混合。当然这个问题也有对应的解决方案，可以将子纹理的分辨率设置为 2 的整数次幂，然后在将每个子纹理放入纹理图集之前，提前为它们生成 mipmap 层次结构[1293]。

对于上述的这些问题，一个更简单的解决方案是使用一种被称为纹理数组（texture array）的 API 结构，它完全避免了 mipmap 和 repeat 模式所带来的问题[452]，如图 6.20 右侧所示。一个纹理数组中的所有子纹理都需要具有相同的尺寸、格式、mipmap 层次结构和 MSAA 设置。与纹理图集一样，纹理数组只需要进行一次设置，然后就可以通过着色器中的索引来访问数组中的任何元素，这种方法要比分别绑定每个子纹理快 5 倍[452]。

现代图形 API 对于无绑定纹理（bindless texture）的支持，也有助于避免状态切换所带来的额外开销[1407]。如果没有无绑定纹理的话，则需要使用 API 来将纹理绑定到特定的纹理单元中，这会带来很多问题，其中一个问题是纹理单元数量的是有上限的，这使得程序员的工作变得更加复杂。在这种情况下，是由驱动程序来确保纹理驻留在 GPU 端的。对于无绑定纹理而言，纹理的使用数量是没有上限的，因为每个纹理都只通过一个 64 位的指针（有时称为句柄 handle）来与其数据结构相关联。可以通过多种方式来访问这些句柄，例如通过 uniform buffer、可变数据、其他纹理，以及着色器存储缓冲对象（shader storage buffer object，SSBO）等。应用程序需要确保纹理驻留在 GPU 端。无绑定纹理避免了驱动程序中任何类型的绑定开销，这使得渲染速度更快。

6.2.6 纹理压缩

固定压缩比的纹理压缩（fixed-rate texture compression）是一种直接解决内存、带宽以及缓存问题的解决方案[127]。通过让 GPU 对纹理进行实时的解码压缩，能够使得纹理占据更少的内存，从而增加有效的缓存大小。同样重要的是，这样的纹理使用起来会更加高效，因为在访问纹理时，对于内存带宽的开销变少了。能够对纹理进行压缩，也意味着能够支持更大尺寸的纹理。例如：在 512×512 的分辨率下，每个纹素使用 3 个字节的未压缩纹理将会占用 768 kB 空间；而在使用纹理压缩之后（例如压缩比为 $6:1$），一个 1024×1024 尺寸的纹理也只需要 512 kB 的空间。

有许多用于图像文件的压缩格式，例如 JPEG 和 PNG 格式，但是在硬件中实现这些图像解码的成本很高（有关纹理转换编码的内容，详见章节 19.10.1）。S3 开发了一种被称为 S3 纹理压缩（S3 Texture Compression，S3TC）的方案[1524]，它被作为 DirectX 的标准纹理压缩方法，被称为 DXTC，在 DirectX 10 中则被称为 BC（块压缩，Block Compression）；此外，它也是 OpenGL 事实上的标准，因为几乎所

有 GPU 都支持这种纹理压缩方法。这种方法的优点在于，可以创建大小固定的压缩图像，具有独立编码的片段，并且解码过程十分简单（因此速度很快）。图像的每个压缩部分都可以被单独解码，没有共享查找表或者其他依赖关系，这简化了解码过程。

DXTC/BC 压缩方案有七种变体，它们之间有一些共同的特性。这种压缩方案的编码是在 4×4 范围内的纹素块上（也称为 tile）上完成的，每个纹素块都可以进行单独编码；这个编码过程是基于插值的，对于每个编码量，会存储两个参考值（即颜色）。它会在两个参考值之间所构成的直线上选择一个值，即在两个参考颜色之间进行插值。这种压缩方案最终只会存储两个参考颜色，以及每个像素的短索引值。

Name(s)	Storage	Ref colors	Indices	Alpha	Comment
BC1/DXT1	8 B/4 bpt	RGB565×2	2 bpt	–	1 line
BC2/DXT3	16 B/8 bpt	RGB565×2	2 bpt	4 bpt raw	color same as BC1
BC3/DXT5	16 B/8 bpt	RGB565×2	2 bpt	3 bpt interp.	color same as BC1
BC4	8 B/4 bpt	R8×2	3 bpt	–	1 channel
BC5	16 B/8 bpt	RG88×2	2 × 3 bpt	–	2× BC4
BC6H	16 B/8 bpt	see text	see text	–	For HDR; 1–2 lines
BC7	8 B/4 bpt	see text	see text	optional	1–3 lines

表 6.1：不同的纹理压缩格式。表中的所有压缩方法都作用于 4×4 的纹素块上。表中的 Storage 列代表了最终存储所占据的空间，其中前面的是每个纹素块的所占据的字节数（byte，B），后面的是每个纹素所占据的 bit 数（bits per texel，bpt）。在 Ref colors 列中，前面的英文符号代表具体的通道情况，后面的数字代表了每个通道所占据的 bit 数。例如：RGB565 代表了红色通道有 5 个 bit，蓝色通道有 6 个 bit，绿色通道有 5 个 bit。

具体的编码方式在七个变体之间有所不同，它们之间的区别如表 6.1 所示。表中的“DXT”代表了 DirectX 9 中的名称，而“BC”则代表了 DirectX 10 及之后的名称。从表中我们可以看出，BC1 有两个 16 bit 的 RGB 参考值（红色 5 bit，绿色 6 bit，蓝色 5 bit），每个纹素都有一个 2 bit 的插值因子，用于在参考值或者两个中间值中进行选择；因为 2 bit 可以代表四个不同的值，其插值结果共有四个，分别是两端的参考值以及两个中间值。与未压缩的 24 bit RGB 纹理相比，这种压缩方法具有 $6:1$ 的纹理压缩比。

> 另一种 DXT1 模式为透明像素保留了四个可能的插值因子中的一个，此时由于 alpha 值的存在，可能的像素颜色数量会被限制为三个——两个参考值及其平均值。

BC2 与 BC1 对于颜色的编码方式是相同的，但是为每个纹素添加了 4 个 bit，用于存储 alpha 值。到了 BC3 中，每个纹素块对 RGB 数据的编码方式与 DXT1 相同；不同之处在于，BC3 的 alpha 数据使用两个了 8 bit 的参考值，以及使用了每个纹素 3 bit 的插值因子进行编码，3 bit 可以产生 8 个不同的值，一共会产生 8 个插值结果，分别是两端的参考值以及六个中间值，因此每个纹素可以在一个参考 alpha 值，或者六个中间值中进行选择。BC4 只有一个通道，它与 BC3 中的 alpha 值使用了相同的编码方式。BC5 包含了两个通道，每个通道都与 BC3 中的 alpha 值使用了相同的编码方式。

BC6H 针对高动态范围（HDR）纹理进行压缩，在这个格式中，每个原始纹素的 RGB 通道都是一个 16 bit 的浮点数。BCH6 会使用 16 个 byte 来存储一个纹素块，即每个纹素 8 bit。BC6H 具有两种模式，其中一种模式用于单条颜色线（类似于上面的压缩方法），另一种模式用于两条颜色线。在双颜色线模式下，每个纹素块都可以从一小组分区中进行选择。两个参考颜色也可以使用增量编码的方式，从而获得更好的精度表现，并且还可以根据使用的模式，应用不同的精度。在 BC7 中，每个纹素块可以有 1 到 3 条颜色线，每个纹素使用 8 bit 进行存储；BC7 的目标是对 8 bit RGB 或者 RGBA 纹理进行高质量压缩。它与 BC6H 之间具有许多相同的属性，不同之处在于，BC7 用于 LDR 纹理的压缩，而 BC6H 则用于 HDR 纹理的压缩。BC6H 和 BC7 在 OpenGL 中分别被称为 BPTC_FLOAT 和 BPTC。这些压缩技术不仅可以应用于二维纹理，同样也可以应用于立方体贴图或者体积纹理的压缩。

这些压缩方案的主要缺点在于，它们都是有损压缩，也就是说，通常我们无法从压缩纹理中还原出原始图像。在 BC1–BC5 中，我们只使用了 4 或者 8 个插值出的值，来表示全部的 16 个像素。如果一个纹素块中包含大量不同的颜色值，那么就会不可避免的产生一些颜色信息的损失。在实际应用中，如果使用得当，这些压缩方案通常可以提供可接受的图像保真度。

BC1–BC5 存在的一个问题是，由于仅仅使用了两个颜色参考值，因此在解码之后，一个纹素块中的所有颜色都会位于 RGB 空间中的一条直线上，例如：我们无法在一个纹素块中同时表示红色、绿色和蓝色。BC6H 和 BC7 支持更多的颜色线，因此可以提供更高的质量。

OpenGL ES 选择了另一种压缩算法，被称为 Ericsson 纹理压缩（Ericsson texture compression，ETC）[1714]，这种压缩算法被内置在 API 中。该方案具有与 S3TC 相同的特点，即快速解码、随机访问、无间接查找和固定压缩比。它将 4×4 的纹素块编码为 64 bit，即每个像素使用 4 bit，其基本思想如图 6.21 所示。每 2×4 块（或者 4×2，取决于哪个的质量最好）会存储一个基色（base color）。每个纹素

块还会从一个很小的静态查找表中选择四个常量，纹素块中的每个纹素，都可以选择机上其中的一个值，这会逐像素的修改其亮度值。这种压缩算法的图像质量与 DXTC 相当。

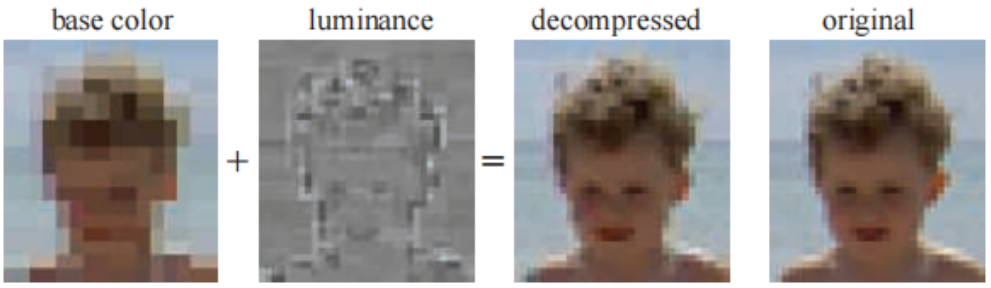


图 6.21：ETC 压缩方法会对像素块的颜色进行编码，然后通过逐像素的亮度修改来获得最终的纹理颜色。

在 OpenGL ES 3.0 中包含的 ETC2 [1715]中，使用了未使用的 bit 组合方式，来为原始 ETC 算法添加更多的模式。其中一个未使用的 bit 组合是压缩表示（例如：64 bit），它代表了这个纹素块会被解压为与另一个压缩表示完全相同的图像。例如：在 BC1 中，将两个参考颜色设置为相同颜色是没有意义的，因为这个纹素块中的所有纹素都将具有完全相同的颜色；而现在只要将一个参考颜色设置为该恒定颜色，就可以让整个纹素块都变成这个颜色。在 ETC 中，一种颜色也可以从第一个有符号颜色中进行增量编码获得，当然计算可能会造成数值溢出，这种情况被用来表示其他压缩模式。ETC2 中添加了两个新模式，在第一种模式中，每个纹素块都有四种不同的颜色；第二个模式被称为最终模式，该模式是 RGB 空间中的一个平面，旨在处理参考颜色之间的平滑过渡。Ericsson alpha 压缩（Ericsson alpha compression，EAC）[1868]用于对只包含一个通道的图像（例如 alpha）进行压缩，这种压缩方法类似于基本的 ETC 压缩，二者的区别在于，前者只针对一个通道进行压缩，最终生成的图像会使用 4 bit 来存储一个纹素。EAC 压缩方法还可以与 ETC2 相结合；此外，还可以使用两个 EAC 通道来对法线进行压缩（随后将详细介绍这个话题）。ETC1、ETC2 与 EAC 都是 OpenGL 4.0 核心配置文件、OpenGL ES 3.0、Vulkan 和 Metal 中的一部分。

对于法线贴图（将在章节 6.7.2 中进行讨论）的压缩需要注意一些问题，因为针对 RGB 颜色进行设计的压缩格式，通常并不适用于法线的 xyz 数据。大多数法线贴图的存储方法，都会利用已知法线为单位长度这个事实，并进一步假设其 z 分量为正（对于切线空间法线而言，这是一个合理的假设）。这样我们就可以只存储法线的 x 和 y 分量，然后在运行时计算出 z 分量即可：

$$n_z = \sqrt{1 - n_x^2 - n_y^2} \tag{6.4}$$

这种表示方法本身就带会来一些压缩效果，因为我们最终只需要存储两个分量即可。并且，由于大多数 GPU 并不原生支持三分量的纹理，一般只会原生支持四分量的纹理，但是这样的话就会浪费一个分量；而将三分量法线转换为两分量进行存储，就可以避免对第四个分量空间的浪费。对 x 分量和 y 分量的进一步压缩，通常是使用 BC5 或者 3Dc 格式来实现的，如图 6.22 所示。由于每个纹素块中的参考值限制了 x 分量和 y 分量的最大值和最小值，因此它们可以被视为在 xy 平面上的轴对齐包围盒。3 bit 的插值因子可以产生 8 个插值数据，即每个轴可以在 8 个值中任选其一，因此这个包围盒被划分为一个 8×8 的潜在法线网格。或者也可以使用两个 EAC 通道来存储 x 分量和 y 分量，然后再按照方程 6.4 来计算 z 分量。

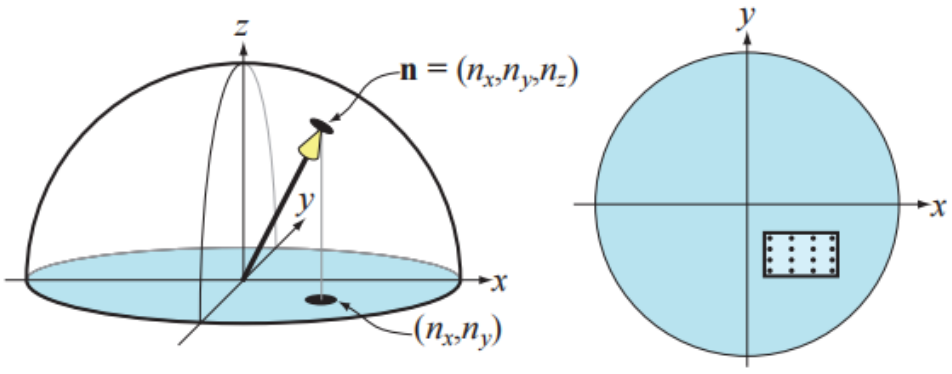


图 6.22：左侧：球面上的单位法线只需要编码 x 分量和 y 分量即可。左侧：对于 BC4/3Dc， xy 平面上的一个轴对齐包围盒限制了法线的范围，每个 4×4 的纹素块可以在这个 8×8 网格中选择法线（为了清晰，这里只显示 4×4 法线）。

在不支持 BC5/3Dc 或者 EAC 压缩格式的硬件上，一种常见的备用方法（fallback）[1227]是使用 DXT5 格式纹理，并将两个分量分别存储在绿色和 alpha 通道中（因为这两个通道的存储精度最高），剩下的红色通道和蓝色通道则没有使用到。

PVRTC [465]也是一种纹理压缩格式，它可以在 Imagination Technologies 的硬件（叫做 PowerVR）上使用，它最广泛的应用是在 iphone 和 ipad 上。它同样也是对 4×4 的纹素块进行压缩，同时它对每个纹素都提供了两种存储方案，分别是 2 bit 或者 4 bit。其核心思想是，提供两个图像的低频（平滑）信号，这两个信号是通过对相邻纹素块进行插值获得的，然后在解码图像的时候，每个纹素会使用 1 bit 或者 2 bit 的插值因子，来在这两个信号之间进行插值。

自适应可伸缩纹理压缩（Adaptive scalable texture compression，ASTC）[1302]的不同之处在于，它可以将 $n \times m$ 的纹理块压缩成 128 bit，其中纹素块的尺寸范围可以从 4×4 到 12×12，纹素块尺寸的不同会导致不同的压缩比：当纹素块尺寸为 4×4 时，每个纹素会使用 8 bit 进行存储；当纹素块尺寸为 12×12 时，每个纹素仅会使用 0.89 bit 进行存储。ASTC 使用了大量技巧来压缩索引表示，并且纹素块都可以选择不同的颜色线数量和端点（参考值）编码。此外，ASTC 可以处理任意 1–4 通道的纹理，包括 LDR 纹理和 HDR 纹理。ASTC 是 OpenGL ES 3.2 及后续版本中的一部分。

上面提到的所有纹理压缩方案都是有损的，而且不同压缩方案所花费的压缩时间也是不同的。花费较长的时间（几秒钟甚至几分钟）来进行压缩，可以获得更好的压缩质量；因此这个纹理压缩的过程通常是离线预处理的，即将压缩好的图像存储起来，并在之后进行使用。或者，我们可以只花费较短的时间（几毫秒）来进行压缩，这样获得的压缩质量通常较低，但是好处在于，我们可以在接近实时的情况下，对纹理进行压缩并立即使用。一个常见的例子是天空盒（章节 13.3），当云的移动速度很慢时，天空盒每隔一秒左右才会刷新。纹理的解压缩过程是非常快的，因为它是使用固定功能的硬件来实现的；这种差异被称为数据压缩的不对称性（data compression asymmetry），即数据的压缩过程要比解压过程花费更长的时间。

图 6.23：在纹理压缩过程中，每个通道使用 16 bit 与 8 bit 的效果对比。从左到右分别是：原始纹理；对每个通道 8 bit 的纹理使用 DXT1 进行压缩；对每个通道 16 bit 的纹理使用 DXT1 进行压缩，并在着色器中进行了重新归一化。为了更清楚地展示效果差异，使用了较强的光照来渲染纹理。

Kaplanyan [856]提出了几种可以提高压缩纹理质量的方法：对于包含颜色贴图和法线贴图的纹理，建议每个通道使用 16 bit；对于颜色纹理而言，可以对这 16 个 bit 进行直方图归一化（histogram renormalization），然后在着色器中使用比例常量和偏移常量（每个纹理）来反转它的效果。直方图归一化是一种将图像中使用的颜色值扩展到整个范围（例如 $[0, 255]$）的技术，这是一种增强对比度的有效方法。每个分量

使用 16 bit，可以确保在重新归一化之后，直方图中不会出现未使用的位置，从而减少了许多纹理压缩方案可能会引入的带状瑕疵，如图 6.23 所示。此外，如果图像中有 75% 的像素值大于 116/255 的话，Kaplanyan 建议对这种纹理使用线性颜色空间，而不是将纹理存储在 sRGB 颜色空间中。对于法线贴图而言，他还注意到 BC5/3Dc 压缩方案通常会独立于 y 分量来压缩 x 分量，这意味着并不总是能找到最佳的法线；相反，他建议对法线使用以下的误差度量：

$$e = \arccos\left(\frac{\mathbf{n} \cdot \mathbf{n}_c}{\|\mathbf{n}\| \, \|\mathbf{n}_c\|}\right) \tag{6.5}$$

其中 $\mathbf{n}$ 是原始法线， $\mathbf{n}_c$ 是经过压缩和解压缩的对应法线。

需要注意的是，也可以在不同的颜色空间中来进行纹理压缩，这可以对纹理压缩过程进行加速，一个常用的颜色空间变换是 RGB $\rightarrow$ YCoCg [1112]，其数学形式如下：

$$\begin{pmatrix} Y \\ C_o \\ C_g \end{pmatrix} = \begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 0 & -1/2 \\ -1/4 & 1/2 & -1/4 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{6.6}$$

其中的 Y 是亮度项（luminance）， C_o 代表了橙色色度（chrominance）， Cg 代表了绿色色度。这个变换过程的逆变换开销也很低：

$$G = (Y + C_g), \quad t = (Y - C_g), \quad R = t + C_o, \quad B = t - C_o \tag{6.7}$$

这两个变换都是线性的，方程 6.6 是一个矩阵和向量的乘法，这个运算本身就是线性的（详见方程 4.1 和方程 4.2）。这一点十分重要，因为我们可以在纹理中存储 YCoCg 颜色，而不是 RGB 颜色；而且纹理硬件同样可以在 YCoCg 颜色空间中进行过滤操作，然后像素着色器可以根据需要，再将颜色转换回 RGB 颜色空间中。需要注意的是，这个转换过程本身是有损的，根据情况的不同，这对结果而言可能会很重要，也可能是无关紧要的。

还有一种可逆的 RGB $\rightarrow$ YCoCg 变换，其数学形式可以总结为：

$$\begin{cases} C_o = R - B \\ \quad t = B + (C_o \gg 1) \\ C_g = G - t \\ \quad Y = t + (C_g \gg 1) \end{cases} \iff \begin{cases} t &= Y - (C_g \gg 1) \\ G &= C_g + t \\ B &= t - (C_o \gg 1) \\ R &= B + C_o \end{cases} \tag{6.8}$$

方程中的符号 $\gg$ 代表了位运算中的右移操作，这意味着可以在 24 bit 的 RGB 颜色与相应的 YCoCg 表示之间来回转换，并且不会有任何损失。需要注意的是，如果 RGB 中的每个分量都有 n 个 bit，那么色度值 C_o 和 C_g 便会各有 $n+1$ 个 bit，从而保证变换是可逆的；而亮度值 Y 只需要 n 个 bit。Van Waveren 和 Castano [1852] 使用了有损的 YCoCg 变换，从而在 CPU 和 GPU 上实现了对 DXT5/BC3 格式纹理的快速压缩。它们将 Y 存储在 alpha 通道中（因为它具有最高的精度），将 C_o 和 C_g 存储在 RGB 的前两个通道中。这个压缩过程非常快，因为 Y 是单独进行压缩和存储的；对于 C_o 和 C_g 分量，他们构建一个二维包围盒，并选择能够产生最佳效果的对角线包围盒。请注意，对于在 CPU 上动态创建的纹理，最好也压缩 CPU 上对纹理进行压缩；而当纹理是通过 GPU 渲染创建的时候，通常最好也是在 GPU 上对纹理进行压缩。YCoCg 变换和其他亮度–色度（luminance–chrominance）的变换，在图像压缩中十分常用，其中的色度分量会在 2×2 的像素块上进行平均，这样做可以减少 50% 的存储空间，并且通常效果也很好，因为像素之间的色度变化是十分平缓的。Lee–Steere 和 Harmon [1015]将其进一步转换到 HSV（hue–saturation–value）颜色空间中，并在 x 和 y 方向上对色调和饱和度进行 4 倍的下采样，最终将其存储为单通道的 DXT1 纹理。Van Waveren 和 Castano 还描述了法线贴图的快速压缩方法[1853]。

Griffin 和 Olano [601]的一项研究表明，当多个纹理应用在一个具有复杂着色模型的几何物体上时，可以使用的质量很低的纹理，同时并不会带来任何可以感知的明显差异。因此，根据纹理使用情况的不同，可以对其质量进行一定程度的降低。Fauconneau [463]提出了一种 DirectX 11 纹理压缩格式的 SIMD 实现。

6.3 程序化纹理

上文中我们所提到的方法，都是给定一个纹理空间中的坐标位置，然后在图像中进行查找，从而获得纹理值。还有一种方法是对函数进行求值，然后作为对应位置上的纹理值，这就是程序化纹理（procedural texture）。

在过去，程序化纹理通常运用于离线渲染中，在实时渲染中更加常见的则是图像纹理，这是因为现代 GPU 中的图像纹理硬件是非常高效的，可以在一秒钟内执行数十亿次的纹理访问操作。然而，目前的 GPU 架构正在向着更低的计算成本，以及更昂贵（相对）的存储访问发展，也就是说，存储访问和带宽限制越来越成为 GPU 的性能瓶颈。这些趋势使得程序化纹理在实时渲染中得到了更加广泛的应用。

考虑到体积纹理的高昂存储成本，因此程序化的体积纹理，是过程化生成中一个特别有吸引力的应用。这样的纹理可以通过很多技术进行合成，其中最为常见的一种方法是，使用一个或者多个噪声函数来生成纹理值[407, 1370, 1371, 1372]，如图 6.24 所示。这些噪声函数通常都是以连续 2 次幂的频率进行采样的，这被称为 octave（八度）。每个 octave 都有一个权重，这个权重通常会随着频率的增加而下降，而这些加权样本的总和被称为湍流（turbulence）函数。

图 6.24：两个使用的实时程序化纹理生成的体积纹理。左边的大理石是使用光线步进（ray marching）渲染的半透明体积纹理。右边的物体是使用复杂的程序化木材着色器[1054]生成的图像，并将其在“真实”环境（背景）中进行了渲染。

由于计算噪声函数的成本较大，因此三维数组中的点通常都是预先计算好的，并使用这些点进行插值从而获得相应的纹理值。有许多方法可以通过使用颜色缓冲混合，来快速生成这些三维数组[1192]。Perlin [1373]提出了一种快速且实用的方法，来对这个噪声函数进行采样，并展示了一些实际用途。Olano [1319]提供了一种可以允许在存储纹理和执行计算之间进行权衡的噪声生成算法。McEwan 等人[1168]实现了在着色器中计算经典噪声（classic noise）和单纯形噪声（simplex noise）的方法，该方法无需任何查找，并提供了完整的源代码。Parberry [1353]使用动态规划，来将计算分摊到若干个像素上，从而加快了噪声的计算速度。Green [587]给出了一种更高质量的方法，它更适用于接近交互式帧率的应用程序，因为它在一次查找中使用了 50 个像素着色器指令。Perlin 提出的原始噪声函数[1370, 1371, 1372]可以被进一步改进，Cook 和 DeRose [290]提出了另一种表示方法，称为小波噪声（wavelet noise），它避免了锯齿问题，同时只略微增加了计算成本。Liu 等人[1054]通过使用各种噪声函数，来模拟不同的木材纹理和表面抛光效果。我们还推荐读者进一步阅读 Lagae 等人[956]有关这个主题的最新报告。

其他的程序化方法也是可行的。例如：通过测量从每个位置到一组“特征点”（分散在空间中）的距离，来构建蜂窝状纹理（cellular texture）。还可以用不同的方法对这些生成的最近距离进行映射，例如：改变颜色或者着色法线，从而生成看起来像细胞、石板、蜥蜴皮肤以及其他自然图案的纹理。Griffiths [602]讨论了如何高效地在GPU 上找到最近邻居，并生成蜂窝状纹理的方法。

另一种类型的程序化纹理是物理模拟或者其他交互过程的结果，例如水面波纹或者扩展裂缝。在这种情况下，程序化纹理可以根据动态条件的不同，有效地生产无限种变化。

当生成一个程序化的二维纹理时，参数化问题（UV）可能要比创建纹理更加困难，因为在传统的纹理制作过程中，可以对 UV 拉伸或者 UV 接缝进行手动调整，而程序化纹理则不行。一种解决方案是直接在表面上合成纹理，从而完全避免参数化的问题；在复杂表面上执行这种操作，在技术上是具有挑战性的，这也是一个十分活跃的研究领域，详情请参阅 Wei 等人[1861]有关该领域的综述。

对程序化纹理的抗锯齿处理，要比图像纹理的抗锯齿既困难和又容易。一方面，类似 mipmap 这样的预计算方法是不可行的，这会给程序员带来极大的负担。另一方面，在生成程序化纹理的时候，我们实际上已经知道了有关于纹理内容的“内部信息”，因此可以对其进行调整，从而避免锯齿现象。这对于叠加了多个噪声函数的程序化纹理尤其正确，因为每个噪声函数的频率都是已知的，因此我们可以主动丢弃那些会导致锯齿的频率，这实际上会使得计算成本变得更低。对于其他类型的程序化纹理而言，有着各种各样的抗锯齿技术[407, 605, 1392, 1512]。Dorn 等人[371]对过去的工作进行了讨论总结，并提出了一些重新构造纹理函数的过程，以避免过高的频率，即有限频宽（band-limited）的频率。

6.4 纹理动画

应用于表面上的图像纹理也不一定是静态的。例如：视频源（video source）可以用作一种随帧变化的特殊纹理。

纹理坐标也不一定是静态的。无论是在网格数据本身对纹理坐标进行修改，还是通过顶点着色器或者像素着色器中的函数，来对纹理坐标进行修改，应用程序设计人员都可以显式地改变帧与帧之间的纹理坐标。想象一下，现在我们有了一个已经建模好的瀑布模型，并且它已经被一个图像纹理化了，使得它看起来很像瀑布。假设纹理坐标 v 是水流的方向，为了让水流动起来，必须从每一帧的坐标 v 中减去一定的数值。纹理坐标的减法操作会使得纹理本身看起来正在向前移动。

可以通过对纹理坐标应用变换矩阵来生成更加精细的效果。除了平移之外，它还允许其他的线性变换操作，例如缩放、旋转和剪切[1192, 1904]，图像扭曲（image warping）和变形转换（morphing transforms）[1729]，以及广义投影[638]等。通过在 CPU 或者着色器中应用变换函数，可以生成更复杂的效果。

通过使用纹理混合（texture blending）技术，还可以实现其他的动画效果。例如：从一个大理石纹理出发，将其渐变为一个肉质纹理，从而使得雕像看起来像是活过来一样[1215]。

6.5 材质映射

纹理的一个常见用途是对材质属性进行修改，从而影响着色方程的计算结果。现实世界中的物体通常都会具有不同的表面材质属性，为了模拟这样的物体，像素着色器可以从纹理中读取纹理值，并在计算着色方程之前，使用它们来修改材质的参数。纹理最常修改的参数就是表面是颜色，这种纹理通常被称为反照率颜色贴图（albedo color map）或者漫反射颜色贴图（diffuse color map）。但是，理论上任何参数都可以被纹理进行修改，例如：替换、相乘或者以其他方式等。例如在图 6.25 中的表面，应用了三种不同的纹理来替换常量值。

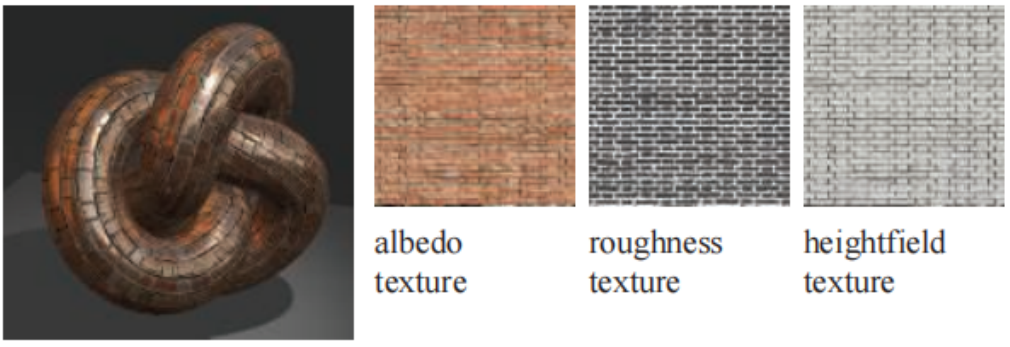


图 6.25：金属砖块和灰浆材质。右边的纹理分别代表了表面颜色、粗糙度（越亮越粗糙）和凹凸高度（越亮起伏程度越大）。[218]

材质的纹理还有更加广泛的用途。纹理还可以用来控制像素着色器本身的流程以及函数功能，而不是简单地修改着色方程中的参数。具有不同着色方程和参数的多个材质，可以同时被应用于一个表面上，通过使用一个特殊纹理，来指定表面的哪些区域具有哪些材质，从而在每个区域上执行不同的着色代码。例如：在具有生锈区域的金属表面上，可以使用一个特殊贴图来指示生锈的位置，根据纹理查找的结果，来决定是执行生锈着色器还是执行闪亮的金属着色器（章节 9.5.2）。

着色模型中的一些输入参数（例如表面颜色），与着色器的最终颜色输出具有线性关系。包含类似输入参数的纹理可以使用标准技术来进行过滤，从而避免瑕疵和锯齿。对于包含非线性着色输入参数的纹理而言，例如粗糙度或者凹凸贴图（章节 6.7），则需要更多的注意和处理，来避免锯齿和走样的出现。使用将着色方程考虑在内的滤波技术，可以改善这类纹理的结果，这些技术将在章节 9.13 中进行讨论。

6.6 Alpha 映射

alpha 值可以用于 alpha 混合或者 alpha 测试，它们能够实现许多效果，例如树叶渲染、爆炸效果和远处物体等。本小节将讨论如何使用带有 alpha 通道的纹理，以及这些方法的各种限制和对应的解决方案。

一种与纹理相关的效果是贴花（decal）。例如：假设我们想在茶壶上放置一张花的图片，我们并不需要花的整个画面，只需要花所在的部分即可；对于那些不包含花的纹素，我们可以将它的 alpha 值设为 0，使这些部分变得透明，这样它就没有任何效果了。因此，通过正确设置贴花纹理的 alpha 值，我们可以使用贴花纹理来替换对应的底层表面，或者将二者混合。通常，我们可以使用一个 clamp 转换函数与透明边框，将贴花的单个副本（相对于重复纹理）应用到表面上。图 6.26 展示了贴花实现的示意图，有关贴花技术的更多信息，详见章节 20.2。

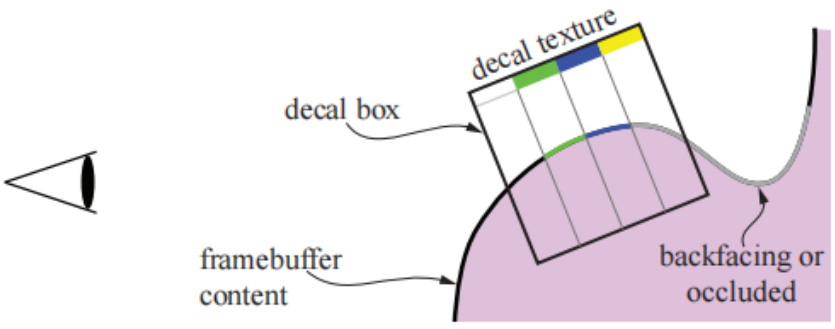


图 6.26：实现贴花的一种方法。首先使用场景来渲染一个帧缓冲，然后再渲染一个 box，对于 box 内的所有顶点，贴花纹理会被投影到帧缓冲上。box 中最左侧的纹素是完全透明的，因此它不会影响帧缓冲；box 中的黄色纹素是不可见的，因为它会被投影到表面的隐藏部分。

alpha 值的另一个类似应用是制作镂空（cutout）效果。假设我们制作了一个灌木的贴花图像，并将其应用到场景中的一个矩形上。这个原理和贴花是相同的，不同之处在于，这个灌木图像并不是和底层表面对应的，而是会绘制在它背后的任何几何图形上。通过这种方式，可以仅仅使用单个矩形，便能够渲染出具有复杂轮廓的物体。

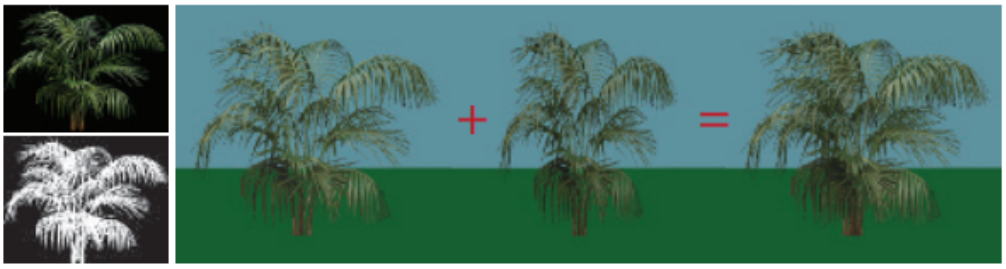


图 6.27：左边是灌木纹理贴图及其对应的 1 bit alpha 通道贴图。右边是灌木贴图渲染在单个矩形上的效果；通过添加旋转 90 度的第二个矩形副本，可以创建一个廉价的三维灌木丛。

在这个灌木的例子中，如果观察者围绕它进行旋转，这个视觉错觉就会露陷，因为这个灌木矩形实际上并没有厚度。一种解决方案是将这个灌木矩形复制一份，并沿着树干旋转 90 度，使得两个灌木矩形相互垂直。这两个矩形构成了一个廉价的三维灌木效果，它有时会被称为“交叉树”[1204]，从地面上进行观察的时候，这种视错觉是相当有效的，如图 6.27 所示。Pelzer [1367]讨论了一个类似的配置方法，他使用了三个镂空纹理来表示草。在章节 13.6 中，我们将讨论一种称为广告牌（billboard）的方法，它仅使用单个矩形便可以实现这种渲染效果。如果观众移动到地面以上，那么这种视错觉也会露陷，因为在灌木上面可以看到两个明显切口，如图 6.28 所示。为了解决这个问题，可以使用不同的方式来添加更多的镂空纹理（例如切片、分支、层等），从而提供更加令人信服的模型，章节 13.6.5 中讨论了一种生成此类模型的方法，图 19.31 则展示了另一种方法。

图 6.28：从离开地面一点的地方来观察这个“交叉树”灌木，这种视错觉就消失了。

将 alpha 贴图和纹理动画结合在一起，可以产生令人信服的特殊效果，例如闪烁的火炬，植物生长，爆炸和大气效果等。

使用 alpha 贴图来渲染物体有几个可选的方式。alpha 混合（章节 5.5）允许透明度值为小数，从而实现了物体边缘的抗锯齿，以及部分透明物体的渲染。但是 alpha 混合需要在渲染不透明三角形之后，再去渲染透明的三角形，并且需要严格按照从后向前的顺序进行渲染。一个无法实现的例子是刚才提到的简单交叉树，它包含了两个镂空纹理，在这个例子中，没有任何一个渲染顺序是正确的，因为两个四边形相互交叉，每个四边形都位于另一个四边形的前面。而且，即使在理论上可以对其进行排序并得到正确的渲染顺序，这样做通常也是十分低效的，例如：一块土地上可能会有成千上万的草叶，这些草叶都是用镂空纹理进行表示的。每个网格物体都由多个单独的叶片所组成，因此显式地对每个叶片进行排序是非常不切实际的。

在渲染的时候，有几种不同的方法可以改善这个问题。一种是使用 alpha 测试，即在像素着色器中丢弃 alpha 值低于给定阈值的片元，其做法如下：

$$\text{if}(\text{texture.a} < \text{alphaThreshold}) \quad \text{discard} \tag{6.9}$$

其中 texture.a 代表了从纹理中检索到的对应 alpha 值，参数 alphaThreshold 是用户提供的阈值，它将决定哪些片元会被丢弃。这个二元的可见性测试允许以任意顺序来渲染三角形，因为透明的片元都会被丢弃。通常我们希望对 alpha 值为 0.0 的任何片元都执行这个操作，将完全透明的片元直接丢弃还有额外的好处，它节省了后续着色器处理和合并的计算成本，同时还可以避免将 z-buffer 中的像素错误地标记为可见[394]。而对于镂空纹理而言，我们通常会将这个阈值设置为大于 0.0 的值，比如 0.5 或者更高。或者再进一步，直接忽略 alpha 值，不使用它们进行混合；这样做可以避免乱序所带来的瑕疵，但是结果的质量会很低，因为只有两个级别的透明度（完全不透明和完全透明）是可用的。另一种解决方案是对每个模型执行两个 pass：第一个 pass 用于渲染实体（不透明）的镂空样本，同时写入 z-buffer；另一个 pass 则用于渲染半透明的样本，此时并不会写入 z-buffer。

alpha 测试还有另外两个问题，即过度放大[1374]和过度缩小[234, 557]，当 alpha 测试与 mipmap 一起使用的时候，如果不进行特殊处理，那么效果会很差，如图 6.29 顶部所示，树木的叶子会比预期变得更加透明。这可以用这样一个例子来进行解释：假设我们现在有一个具有四个 alpha 值的一维纹理，即 $(0.0, 1.0, 1.0, 0.0)$；而在平均之后，下一个 mipmap 层级所对应的纹素值会变为 $(0.5, 0.5)$，最高层级 mipmap 所对应的纹素值为 (0.5)。现在，假设我们现在使用 $\alpha_t = 0.75$ 来作为 alpha 测试的阈值。当访问第 0 级的 mipmap 时，4 个纹素中的 1.5 个可以通过 alpha 测试，不会被丢弃。但是，当访问剩下两个 mipmap 层级时，由于 $0.5 < 0.75$，因此所有纹素都会被丢弃。图 6.30 展示了另一个例子。

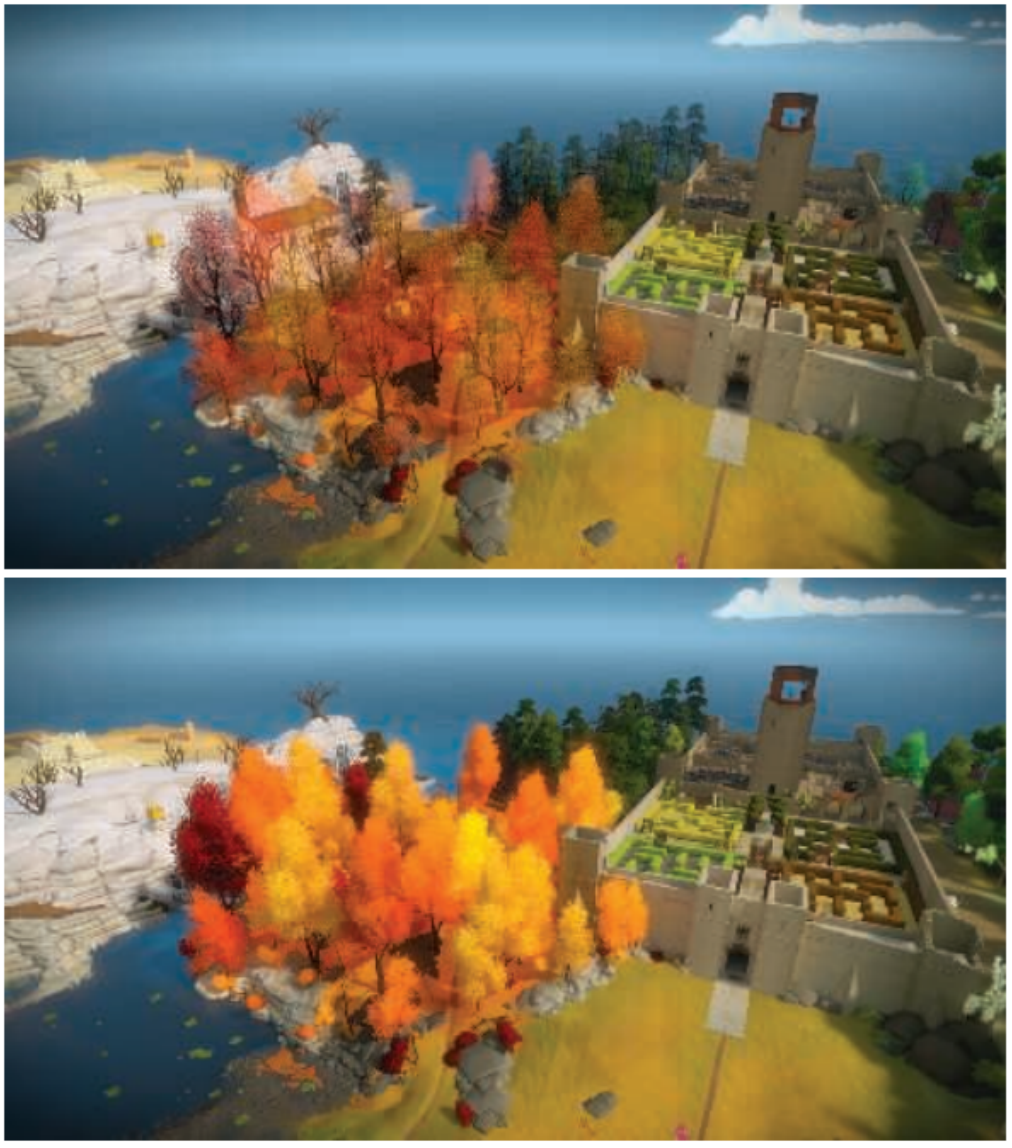


图 6.29：上图：alpha 测试与 mipmap 没有进行任何的修正。下图：根据覆盖率重新调整 alpha 值的 alpha 测试结果。

Castano [234]提出了一个作用于 mipmap 创建过程中的简单解决方案，它工作得很好。对于第 k 个 mipmap 层级，覆盖范围 c_k 的定义如下：

$$c_k = \frac{1}{n_k} \sum_i (\alpha(k,i) > \alpha_t) \tag{6.10}$$

其中 n_k 为第 k 级 mipmap 中的纹素个数， $\alpha(k,i)$ 为第 k 级 mipmap 在像素 i 处的 alpha 值， α_t 为方程 6.9 中用户提供的 alpha 阈值。这里我们假设 $\alpha(k,i) > \alpha_t$ 的结果为 1 或者 0。需要注意的是， $k = 0$ 代表了最低层级的 mipmap，即原始图像。对于每个 mipmap 层级，我们都会找到一个新的 mipmap 阈值 α_k ，而不是一直使用原始阈值 α_t ，这样做可以使得 c_k 等于 c_0 （或者尽可能得接近），这个过程这可以

使用二分查找来实现。最后，使用 α_t/α_k 来对第 k 级 mipmap 中所有纹素的 alpha 值进行缩放，如图 6.29 的底部所示，NVIDIA 的纹理工具也支持这个方法。Golus [557]给出了一个该方法的变体，该方法并没有对 mipmap 进行修改，而是随着 mipmap 层级的增加，在着色器中对 alpha 进行放大。

图 6.30：第一行是叶子图案不同 mipmap 层级的结果，更高的 mipmap 层级放大了叶子的可见性。在第二行中，mipmap 进行了阈值为 0.5 的 alpha 测试处理，这个过程展示了随着物体与相机距离的增加，其在屏幕上占据像素数量越来越少。[557]

Wyman 和 McGuire [1933]提出了一种不同的解决方案，其中方程 6.9 中的代码在理论上被替换为：

$$\text{if}(\text{texture.a} < \text{random}()) \quad \text{discard} \tag{6.11}$$

随机函数在 $[0,1]$ 中返回一个均匀的值（0–1 均匀分布），这意味着平均而言，这个操作会产生正确的结果。例如：如果检索到的纹理 alpha 值为 0.3，那么意味着该片元将会有 30% 的概率被丢弃。这是随机透明的一种形式，每个像素仅包含一个样本 [423]。在实践中，会将这个随机函数替换为哈希函数，从而避免时空上（temporal，spatial）的高频噪声：

$$\begin{aligned}&\text{float hash2D}(x,y)\{\\&\quad\text{return fract}(1.0e4 * \sin(17.0 * x + 0.1 * y) *\\&\quad(0.1 + \text{abs}(\sin(13.0 * y + x)))); \}\end{aligned} \tag{6.12}$$

通过对上述函数进行嵌套调用，可以创建一个三维哈希，它返回一个位于 $[0,1)$ 内的数字，即：

$$\text{float hash3D(x,y,z) \{ return hash2D(hash2D(x,y),z); \}}$$

哈希函数的输入是，物体局部空间坐标除以该坐标相对于屏幕空间（x 方向和 y 方向）的最大导数，然后再对其进行 clamp 操作。需要进一步注意来获得在 z 方向上的稳定性，该方法最好与时域抗锯齿技术相结合。这个方法随着距离的增加而渐隐，在近距离时我们根本不会看到任何随机效果。该方法的优点在于，每个片元的结果在平均上都是正确的。Castano 的方法[234]则是为每个 mipmap 层级都创建一个修正过的阈值 α_k；然而，这个值可能会随着 mipmap 层级的不同而发生变化，这可能会降低最终的质量，并需要艺术家进行手动调整。

alpha 测试会在放大情况下显示出波纹瑕疵（ripple artifact），这可以通过将 alpha 贴图预先计算为一个距离场[580]来避免（章节 15.5 也对这个问题进行了讨论）。

Alpha To Coverage，以及类似的透明度自适应抗锯齿（transparency adaptive antialiasing），这些方法会将片元的透明度值转换为像素内覆盖的样本数[1250]。这个想法类似于章节 5.5 中所介绍的点阵剔除半透明方法（screen-door transparency），只不过这里的方法作用于亚像素级别。假设每个像素有四个样本位置，现在有一个片元覆盖了一个像素，但是由于镂空纹理的影响，其透明度为 25%（即 75% 不透明）。Alpha To Coverage 模式使得该片元变得完全不透明，尽管它只覆盖了四个样本中的三个。这个模式对镂空纹理而言十分有用，例如重叠的草叶[887, 1876]。由于绘制的每个样本都是完全不透明的，因此最近的叶子会将以相同的方式，在其边缘处遮挡后面的物体。由于此时关闭了 alpha 混合，因此不需要进行排序，就可以正确地混合半透明的边缘像素。

Alpha To Coverage 对于 alpha 测试的抗锯齿而言是非常好的，但是在 alpha 混合的时候，可能会出现瑕疵。例如：两个具有相同 alpha 覆盖百分比的 alpha 混合片元，将会使用相同的亚像素模式，这意味着一个片元将会完全覆盖另一个片元，而不是与之混合。Golus [557]讨论了使用着色器指令 $\mathrm{fwidth}()$ 来对边缘进行锐化处理，如图 6.31 所示。

图 6.31：使用不同技术进行渲染的叶子纹理，在叶子的边缘部分使用了 alpha 覆盖。从左到右分别是：alpha 测试，alpha 混合，Alpha To Coverage，Alpha To Coverage+ 边缘锐化。[557]

对于任何 alpha 映射的使用，理解双线性插值对颜色值的影响原理是很重要的。想象两个相邻的纹素： $rgb\alpha = (255, 0, 0, 255)$ 代表了不透明红色，其相邻的 $rgb\alpha = (0, 0, 0, 2)$ 代表了几乎完全透明的黑色。那么两个纹素中间位置上的 $rgb\alpha$ 值是多少呢？最简单的插值方法可以获得 $(127, 0, 0, 128)$ ，这个 rgb 值是一个“较暗”的红色。然而，这个结果实际上并没有变暗，它是一个预乘了 alpha 的全红色。如果我们要对 alpha 进行插值的话，为了得到正确的插值结果，我们需要确保被插值的颜色在插值之前就已经预乘了 alpha。例如：假设刚才那个几乎透明的邻居纹素被设置为 $rgb\alpha = (0, 255, 0, 2)$ ，即几乎完全透明的绿色。这个颜色没有预乘 alpha，在插值时得到的结果是 $(127, 127, 0, 128)$ ，这个几乎完全透明的绿色会把结果变成（预乘了 alpha）黄色。这个相邻纹素的 alpha 预乘版本是 $(0, 2, 0, 2)$ ，它会给出正确的预乘颜色结果，即 $(127, 1, 0, 128)$ 。这个结果才是有意义的，因为生成的颜色以红色为主，还有一抹很难察觉的绿色。

如果忽略双线性插值给出的预乘结果，则会导致贴花物体和镂空物体周围出现黑色的边缘。在上文的例子中，插值产生的“暗”红色结果会被渲染管线的后续阶段视为未相乘 alpha 的颜色，这会使得边缘变成黑色。即使使用了 alpha 测试，也会看到这种效果。最好的策略就是在进行双线性插值之前，就进行 alpha 预乘[490, 648, 1166, 1813]。WebGL API 支持这个功能，因为合成对于网页而言十分重要。然而，双线性插值通常是由 GPU 执行的，而在执行这个操作之前，着色器并不能对纹素值进行操作。图片也无法以 PNG 等文件格式进行预乘，因为这样做会失去色彩精度。这两个因素结合在一起，会导致在使用 alpha 映射时，产生默认的黑色边缘。一个常见的解决方法是对镂空纹理进行预处理，将透明的“黑色”纹素涂上来自附近不透明纹素的颜色[490, 685]。所有透明区域通常都需要使用这种方式来重新进行绘制，这个过程可以手动完成或者自动完成，这样 mipmap 的各个层级也可以避免边缘问题[295]。同样值得注意的是，在使用 alpha 值来生成 mipmap 的时候，也应当使用 alpha 预乘的颜色值[1933]。

6.7 凹凸映射

本小节介绍了一大类小尺度细节（small-scale detail）的表现技术，我们统称为凹凸映射（bump mapping）。所有的这些方法，通常都是通过逐像素地修改着色例程来实现的；它们在没有添加任何额外几何形状的前提下，提供了比纹理映射更加立体的外观。

物体表面上的细节可以分为三种尺度：覆盖大量像素的宏观特征（macro-feature）、覆盖几个像素的细观特征（meso-feature，或者叫做介观）以及尺寸远

大小于一个像素的微观特征（micro-feature）。这些类别并不是完全固定的，因为在动画或者交互的过程中，观察者可能会在许多不同的尺度上来观察同一个物体。

宏观几何（macrogeometry）由顶点、三角形或者其他几何图元构成。当我们创建一个三维角色时，四肢和头部通常都是在宏观尺度上进行建模的。而微观几何则会被封装在着色模型中，通常在像素着色器中进行实现，并使用纹理贴图作为着色方程的输入参数。着色模型可以用来模拟表面微观几何与光线之间的相互作用，例如：表面有光泽的物体在微观上是光滑的，而漫反射的表面在微观上则是粗糙的。角色的皮肤和衣服看起来具有不同的材质，这是因为它们使用了不同的着色器，或者至少在这些着色器中使用了不同的参数。

细观几何（meso-geometry，介观几何）描述了宏观尺度和微观尺度之间的一切特征。它包含的细节过于复杂，以致于无法使用单个三角形来进行有效地渲染；但是这个尺度对于观察者而言已经足够大了，可以在几个像素上分辨出表面曲率的细微变化。例如：角色脸上的皱纹、肌肉组织的细节、衣服上的褶皱和接缝，都属于细观范畴。一类统称为凹凸映射技术的方法，通常用于建模这样的细观尺度，它们会在像素级别上调整着色参数，使得观众能感知到基础几何形状之外的微小扰动，同时使得基础几何形状保持平坦。不同类型凹凸映射技术的主要区别在于它们表示细节特征的具体方式。细观几何的变化因素包括现实主义水平和细节特征的复杂程度，例如：数字艺术家通常会在模型中雕刻细节，然后使用软件将这些几何元素转换为一个或多个纹理贴图，例如凹凸贴图以及可能的缝隙暗化（crevice-darkening）纹理。

Blinn 在 1978 年提出了在纹理中编码细观尺度细节的想法[160]。他观察到：如果在着色过程中，我们使用稍微扰动的表面法线来代替真实的表面法线，这样表面看起来就像是具有了小尺度细节一样。他将描述表面法线扰动的数据存储在数组中。

其中最关键的想法是，我们不是使用纹理贴图来改变光照方程中的颜色参数，而是使用纹理贴图来修改表面的法线数据。表面原本的几何法线保持不变；我们只是修改了光照方程中所使用的法线；这个操作并没有物理意义上的等效操作，虽然我们对表面法线进行一些修改，但是表面本身在几何意义上仍然是保持光滑的。就像是每个顶点上都有一个法线会给人一种错觉，即三角形之间的表面全都是光滑的；逐像素地修改法线，会改变对三角形表面本身的感知，但是并不会真正地修改这个三角形的几何形状。

对于凹凸映射而言，表面法线必须相对于某个参照系改变自身的方向。为了实现这个操作，每个顶点都会存储一个切线坐标系（tangent frame），它也称为切线空间基底（tangent-space basis）。这个参考系用于将光线转换到表面着色点的局部空间中（反之亦可），从而计算扰动法线的效果。对于一个应用了法线贴图的多边形表

面，除了顶点法线之外，我们还存储了所谓的切线向量（tangent vector）和副切线向量（bitangent vector）。其中副切线有时也被错误地称为副法线（binormal vector）[1025]。

切线向量和副切线向量代表了物体空间坐标系中，法线贴图本身的坐标轴，因为这里的目标是将光线转换到相对于法线贴图的局部空间中，详见图 6.32。

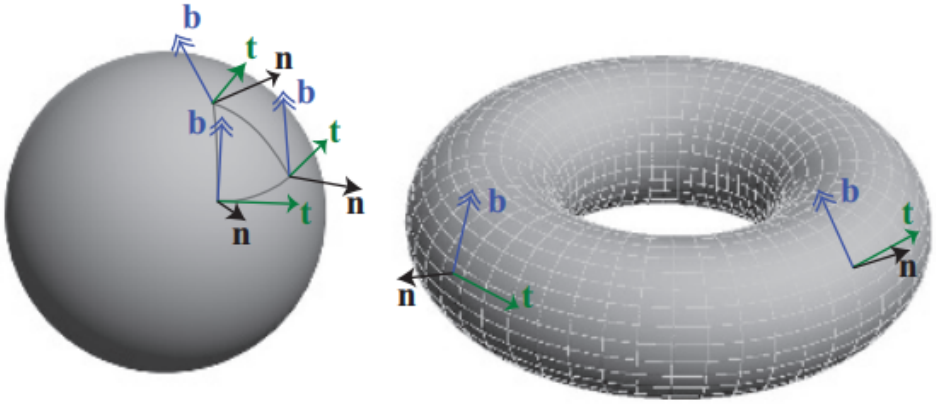


图 6.32：上图展示了两个球面化的三角形，并展示了其切线坐标系。诸如球体和圆环面这样的形状，有一个很自然的切线空间基底，正如圆环面上的经纬度线所展示的那样。

这三个向量，法向量 $\mathbf{n}$，切线向量 $\mathbf{t}$，和副切线向量 $\mathbf{b}$，构成一个基底矩阵：

$$\begin{pmatrix} t_x & t_y & t_z & 0 \\ b_x & b_y & b_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.13}$$

这个矩阵有时会缩写为 TBN，它用于将光线方向（对于给定顶点）从世界空间转换到切线空间中。这些向量并不需要严格的相互垂直，因为法线贴图本身就可能会被拉伸扭曲，以适应表面。但是使用非正交基会引入纹理的偏移，这也意味着可能需要更多的存储空间，同时还可能对性能产生影响，例如无法通过简单的转置操作来求取逆矩阵[494]。一种节省存储空间的方法是，只存储顶点的切线和副切线，然后使用它们的叉乘来计算法线；然而，这种方法只有在 TBN 矩阵的手性（handedness）总是相同的情况下才有效[1226]。但是实践中的很多模型都是对称的，例如：飞机、人体、文件柜以及其他对称物体等。由于纹理会占据大量的存储空间，为了降低对称模型的纹理存储空间，因此它们通常会被镜像到对称模型上；即物体上的纹理只有一半会被存储下来，然后再使用纹理映射，将其应用在模型对称的两侧上。在这种情况下，切空间的手性在对称的两侧是不同的，不满足手性相同的假设。当然还可以在每

个顶点上存储额外的 bit 信息来指示手性，这样也可以避免存储法线。如果存在这个额外的手性 bit 的话，它会被用来修正切线和副切线的叉乘结果（取反），从而生成正确的法线。如果切线坐标系是正交的，也可以将基底存储为四元数（章节 4.3），这样既提高了存储效率，又可以节省每个像素上的一些计算量[494, 1114, 1154, 1381, 1639]；这样做可能会稍微损失一些质量，尽管在实践中这种质量损失是很难察觉到的。

这种切线空间的思想对于其他算法而言也很重要。正如下一章中所讨论的，虽然许多着色方程只依赖于表面的法线方向，但是拉丝铝或者丝绒等材质，也需要知道观察者与光线相对于表面的相对方向，而切线坐标系则可以用来定义表面上的材质朝向。Lengyel [1025]和 Mittring [1226]的文章对这一领域进行了广泛的介绍。Schuler 提出了一种在像素着色器中，动态计算切线空间基底的方法，从而无需逐顶点地存储预先计算好的切线坐标系。Mikkelsen [1209]对该技术进行了改进，并推导出了一种不需要任何参数化的方法，而是使用表面位置的导数和高度场的导数来计算扰动法线。然而，与使用标准的切线空间映射相比，这些技术可能会导致更少的显示细节，并且可能会产生与美术工作流相关的问题[1639]。

6.7.1 Blinn 方法

Blinn 原始的凹凸映射方法是，在纹理的每个纹素上存储两个带符号的值，b_u 和 b_v。这两个值代表了沿图像 u 轴和 v 轴法线的改变量。也就是说，这个纹理值（通常是双线性插值而来的）用于对两个垂直于法线的向量进行缩放，然后将这两个向量加到法线上，从而来改变法线的方向。b_u 和 b_v 这两个值描述了表面在该点处的朝向，如图 6.33 所示。这种类型的凹凸贴图纹理通常被称为偏移矢量凹凸贴图（offset vector bump map）或者偏移贴图（offset map）。

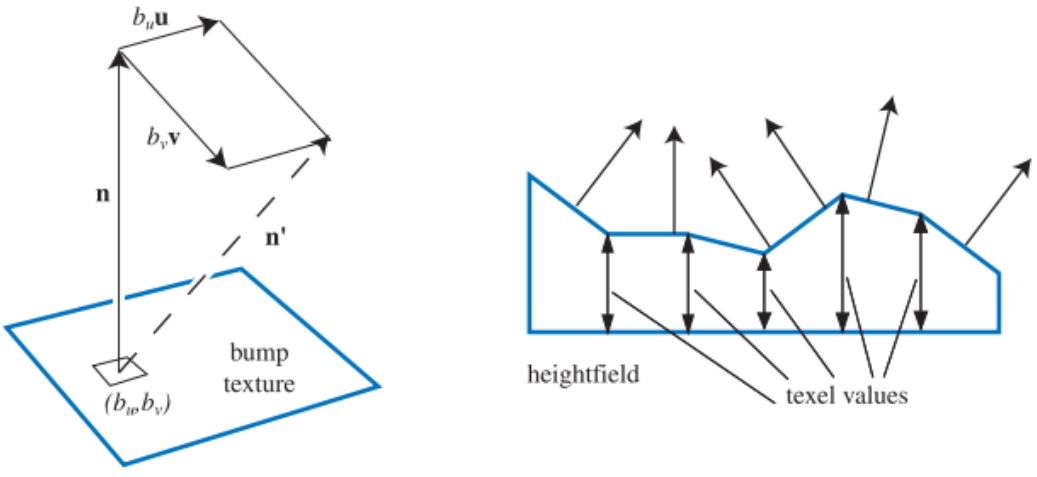


图 6.33：左图中，在凹凸纹理中检索到了对应的值 (b_u, b_v)，法线 $\mathbf{n}$ 会在 u 和 v 两个方向上被这个值进行修改，最终得到 $\mathbf{n}'$（这个新法线是非归一化的）。右图展示了高度场及其对着色法线的影响，这些法线可以在不同高度之间进行插值，从而获得更加平滑的效果。

另一种用于表示凸起的方法是，使用高度场来修改表面法线的方向。每个单色的纹理值都代表了一个高度：在纹理中，白色代表较高的区域，黑色则代表较低的区域（反之亦然），如图 6.34 所示。这是首次创建或者扫描凹凸贴图时所用的格式，它也是由 Blinn 在 1978 年提出的。这个由纹理值定义的高度场，用于推导出 u 和 v 的带符号值，类似于第一种方法中使用的值。具体的推导方式是通过相邻列之间的差来得到 u 的斜率，以及相邻行之间的差来得到 v 的斜率[1567]。一种变体是使用 Sobel 滤波器，它会为直接相邻的纹素赋予更大的权重[535]。

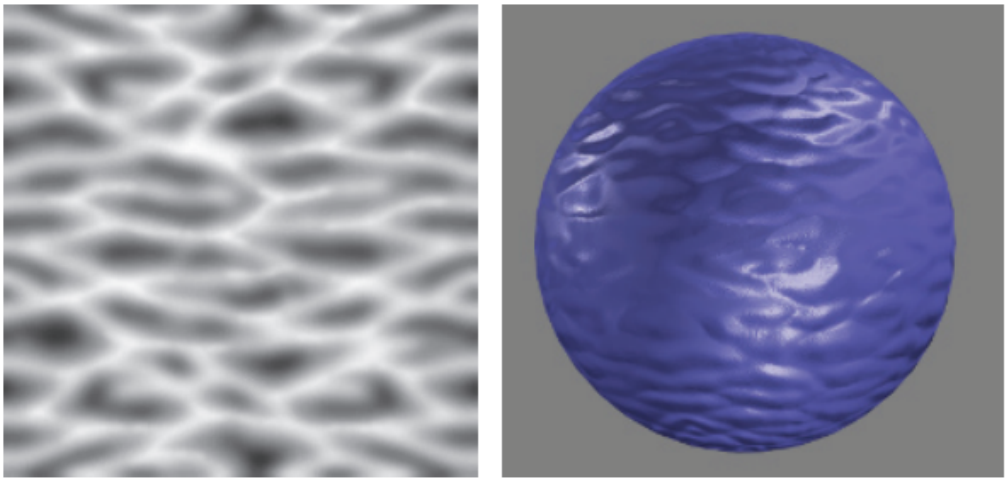


图 6.34：波浪高度场凹凸图像及其在球体上的应用效果。

6.7.2 法线映射

凹凸贴图的一个常用方法是直接存储法线贴图（normal map），其算法和结果与 Blinn 的方法在数学上是等价的；二者的不同之处在于存储格式以及像素着色器中的计算操作。

法线贴图将 (x, y, z) 映射到 $[-1, 1]$ 中，例如：对于 8 bit 纹理而言，x 轴上的值 0 表示 -1.0，值 255 则表示 1.0。图 6.35 给出了一个例子：$[128, 128, 255]$ 是一个浅蓝色，具有该颜色值的平面，其法线为 $[0, 0, 1]$。

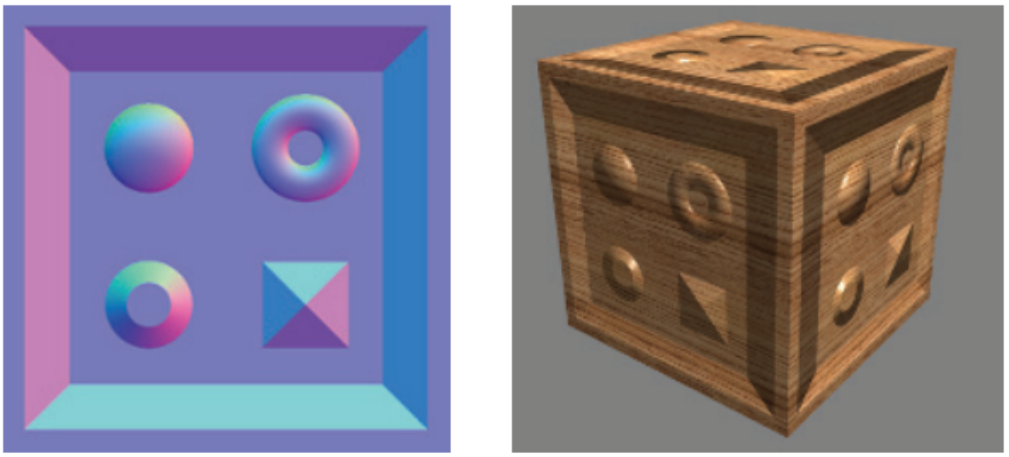


图 6.35：使用法线贴图进行凹凸映射。图像的每个颜色通道，实际上都是表面法线坐标的一个分量。红色通道代表了法线在 x 方向上的偏移；法线贴图的颜色越红，代表了修正后的法线越指向右。绿色通道是 y 方向上的偏移，蓝色通道是 z 方向上的偏移。右边是使用法线贴图生成的图像，请注意立方体顶部的扁平外观。

这种法线贴图的表示方法，最初是作为世界空间法线贴图[274, 891]引入的，现在在实践中已经很少使用了。对于这种类型的映射而言，扰动方法是直截了当的：在每个像素位置上，从法线贴图中直接检索对应的法线，并使用这个法线方向和光线方向来计算表面上该位置的着色结果。法线贴图也可以在物体的局部坐标系中进行定义，这样模型在旋转之后，法线仍然是有效的。然而，世界空间和模型空间的表示方法，都将法线贴图与特定方向的特定几何形状相绑定，这大大限制了法线贴图的可复用性。

相反，扰动法线通常会在切线空间中进行存储和检索，即相对于表面本身的空间。这允许表面发生形变，以及在最大程度上保证了法线贴图的可复用性。切线空间中的法线映射也可以很好地进行压缩，因为 z 分量（与未受干扰的表面法线相对齐的分量）的符号通常可以假设为正。

法线映射可以很好地增强物体表面的真实感，如图 6.36 所示。

图 6.36：一个在游戏场景中使用法线贴图进行凹凸映射的例子。左上角：没有应用右侧的两个法线贴图。左下角：应用了法线贴图。右边：法线贴图。

与颜色纹理的过滤相比，对法线贴图的过滤是一个困难的问题。一般来说，法线与着色结果之间的关系并不是线性的，因此标准的滤波方法可能会产生令人讨厌的瑕疵。想象我们正看着由闪闪发光的白色大理石砌成的楼梯。在某些观察角度下，楼梯的顶部或者侧面会反射出明亮的高光。然而，楼梯的平均法线是 45 度角，当我们从较远处观察这个楼梯时，它的高光将会出现在完全不同的观察角度下，与原来的楼梯截然相反。当带有锐利高光的凹凸贴图没有被正确的过滤时，可能会出现一些分散观察者注意力的闪光效果，这些高光部分的出现和消失，取决于光线镜面反射的着色点法线落在哪里。

Lambertian 材质表面是一种特殊情况，法线贴图对着色结果的影响几乎是线性的；Lambertian 的着色操作几乎完全是一个点乘，这是一个线性运算。取一组法线的平均值，并对平均后法线进行点乘运算；与取单个法线的点乘结果，然后再进行平均是一样的，其数学形式如下：

$$\mathbf{l} \cdot \left(\frac{\sum_{j=1}^{n} \mathbf{n}_j}{n} \right) = \frac{\sum_{j=1}^{n} (\mathbf{l} \cdot \mathbf{n}_j)}{n}. \tag{6.14}$$

请注意，这个平均法线向量在使用之前并没有进行归一化处理。方程 6.14 表明，标准的滤波操作和 mipmap 几乎可以对 Lambertian 表面产生正确的结果；但是这个结果实际上并不是完全正确的，因为 Lambertian 着色方程其实并不是一个简单的点乘，而是一个 clamp 过的点乘，即 $\max(\mathbf{l} \cdot \mathbf{n}, 0)$ ，这个 clamp 操作使得点积不再线

性。如果光线以掠射角度打到表面上，这会使得表面变得过暗，但是在实践中，这通常并不会令人反感。还需要注意的是，一些通常用于法线贴图的纹理压缩方法（例如从另外两个分量中重建 z 分量）并不支持非单位长度的法线，因此使用非归一化的法线贴图可能会造成压缩困难。

在非 Lambertian 表面的情况下，需要将着色方程的输入参数作为一个整体来进行过滤，而不是单独对法线贴图进行过滤，这样可能会产生更好的效果。相关技术将在章节 9.13 中进行讨论。

最后，还可以从高度贴图 $h(x,y)$ 中推导出法线贴图，具体的做法如下[405]：首先，使用中心差分来计算 x 和 y 方向上梯度的近似值：

$$\begin{aligned} h_x(x,y) &= \frac{h(x+1,y) - h(x-1,y)}{2}, \\ h_y(x,y) &= \frac{h(x,y+1) - h(x,y-1)}{2}. \end{aligned} \tag{6.15}$$

则在纹素 (x,y) 处的非归一化法线为：

$$\mathbf{n}(x,y) = (-h_x(x,y), -h_y(x,y), 1) \tag{6.16}$$

在进行实际计算的时候需要格外注意纹理的边界。

地平线映射（horizon mapping）[1027]可以用来进一步增强法线贴图，它可以让凸起的部分在其表面上投射阴影。这是通过预先计算额外的纹理来实现的，每个纹理都与表面平面的方向相关联，在每个纹素中存储该方向上的地平线角度。更多信息详见章节 11.4。

6.8 视差映射

凹凸映射和法线映射存在的一个问题是，表面上凹凸的位置永远不会随着视角的变化而变化，也不会发生互相遮挡的情况。例如：如果我们以一个掠射视角，沿着真正的砖墙进行观察，从某个角度开始，我们就看不到砖与砖之间的砂浆了，这是因为砂浆相对于砖块而言是凹下去的。墙壁的凹凸贴图永远无法表现这种类型的遮挡关系，因为它仅仅是改变了表面的法线。想要实现这种表面自遮挡效果，最好是让这些凸起对表面上每个像素渲染的位置产生实际影响。

视差映射（parallax mapping）的想法是由 Kaneko [851]于 2001 年提出的，并由 Welsh [1866]进行了改进和推广。视差（parallax）这个概念指的是，当观察者的位置发送移动时，物体的位置也会相对发生移动；当观察者移动时，凸起部分应当看起来具有高度感。视差映射的核心思想是通过检查可见物体的高度，对像素中应当看到的内容进行有根据的猜测。

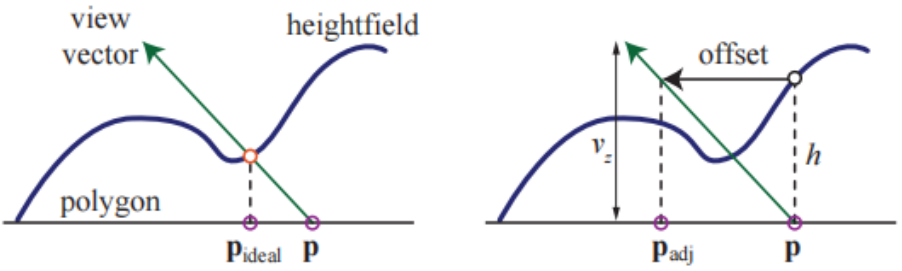


图 6.37：左边是视差映射的目标：根据观察向量穿过高度场的位置，找到表面上被观察到的实际位置。视差映射通过获取矩形位置上的高度，并使用它来找到新的位置 $\mathbf{p}_{adj}$ 来对 $\mathbf{p}_{ideal}$ 进行一阶近似。[1886]

对于视差映射而言，表面的凸起会被存储在一个高度场纹理中。当观察表面上的某个像素时，将会检索该位置对应的高度场值，并将其用于对纹理坐标进行移动，从而对表面上的不同部分进行检索。具体移动多少距离，取决于检索到的高度值以及眼睛到表面上该点的角度，这个过程如图 6.37 所示。这个高度场值要么存储在一个单独的纹理中，要么被打包在其他纹理未使用的颜色通道或者 alpha 通道中（在打包不相关纹理的时候必须十分小心，因为这可能会对压缩质量产生负面影响）。在用于移动坐标之前，高度场的值还会被缩放和偏移：缩放的大小取决于高度场在地表之上（或者地表之下）延伸的高度；偏移取决于不发生变化的“海平面”高度。给定纹理坐标位置 $\mathbf{p}$，调整后的高度场高度 h，以及归一化的观察向量 $\mathbf{v}$，并且高度值为 v_z，水平分量为 $\mathbf{v}_{xy}$，那么经视差调整后的新纹理坐标 $\mathbf{p}_{adj}$ 为：

$$\mathbf{p}_{\text{adj}} = \mathbf{p} + \frac{h \cdot \mathbf{v}_{xy}}{v_z} \tag{6.17}$$

请注意，与大多数着色方程不同，这里执行计算的空间坐标系是很重要的，因为观察向量需要位于切线空间中。

虽然这只是一个很简单的近似，但是如果凸起高度变化得相对缓慢时，这种移动近似在实践中的效果相当好[1171]。相邻纹素大概率具有相同的高度，因此使用原始位置的高度来对新位置的高度进行估计，这个想法是有一定道理的。然而，这种方法在掠射视角下会失效，因为当观察向量近乎平行于表面时，一个很小的高度变化，可能就

会导致一个相当大的纹理坐标偏移，此时的近似会失效，因为检索到的新位置与原始表面位置的高度相关性很小，或者是根本没有高度相关性。

为了改善这个问题，Welsh [1866]引入了偏移限制（offset limit）的概念，这样做的目的是对移动量进行限制，使其永远不会大于检索到的高度值，其数学形式为：

$$\mathbf{p}'_{\text{adj}} = \mathbf{p} + h \cdot \mathbf{v}_{xy} \tag{6.18}$$

请注意，这个方程的计算速度要比原来的快。这种方法的几何解释是：该位置的高度定义了一个半径，超过这个半径位置就无法进行移动，如图 6.38 所示。

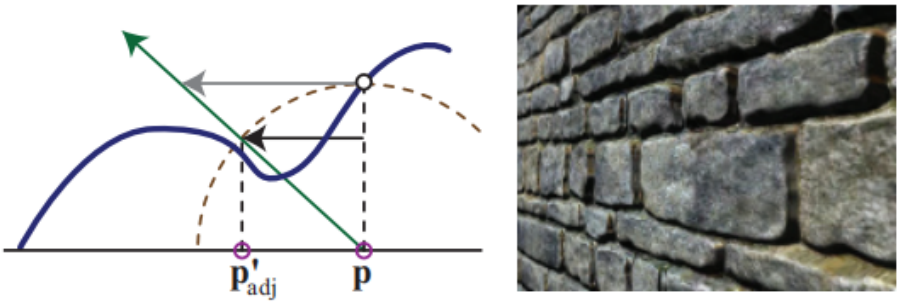


图 6.38：在视差偏移限制中，偏移量的最大值就是原始位置的高度，例如左图中的虚线圆弧。左图中的灰色偏移代表了原始的偏移结果，而黑色偏移代表了限制偏移后的结果。右图则是使用这种技术渲染出来的砖墙。

在垂直于表面的观察角度上，由于 v_z （观察向量的 z 分量）接近于 1，因此这个方程几乎与原始方程完全相同。而在掠射观察角度上，偏移量的影响是有限的。从视觉上看，这减小了在掠射观察角度下的凹凸程度，但是相比于对视察纹理进行随机采样（偏移量很大，无法进行控制）要好得多。随着观察视角的变化，还会存在纹理游动（texture swimming）的问题；对于立体渲染而言，观察者会同时感知两个视点，这要求必须提供一致的深度按时[1171]。即使有着这些缺点，但是具有偏移限制的视差映射，只需要花费一些额外的像素着色器指令即可实现，并且相较于基本的法线映射而言，它可以大幅改善图像的质量。Shishkovtsov [1631]通过在凹凸贴图的法线方向上，对估计位置进行移动，从而改善视差遮挡的阴影效果。

6.8.1 视差遮挡映射

凹凸映射并不会基于高度场来对纹理坐标进行修改，它仅仅是改变了该位置上的着色法线。视差映射提供了一个简单的近似高度场效果，它假设一个像素的高度与其邻居的高度大致相同，但是这个假设很快就会被打破。视差映射所产生的凸起并不会相互

遮挡，也不会产生阴影。我们真正想要的是在像素处可见的内容，即观察向量第一次与高度场相交的地方。

为了能够更好地解决这个问题，一些研究人员提出沿着观察向量进行光线步进（ray marching），直到找到与表面高度场的第一个交点（或者近似交点）。这项工作可以在像素着色器中完成，可以通过访问纹理来检索到所需的高度数据。我们将这些方法的研究归类为视差映射技术的一个子集，这些技术以这样或者那样的方式，来利用光线步进解决问题[192, 1171, 1361, 1424, 1742, 1743]。

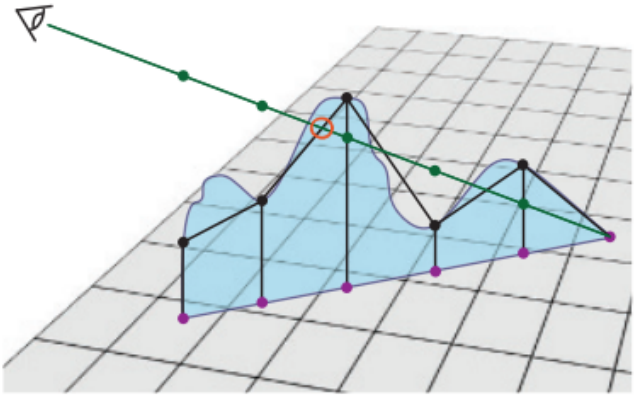


图 6.39：绿色的观察射线被投影到表面上，并以一定的间隔进行采样（紫色点），在每个采样点上，会检索对应位置的高度值。该算法的目标是，找到观察射线与近似弯曲高度场（黑色线段）的第一个交点。

这类算法被称为视差遮挡映射（parallax occlusion mapping，POM）或者浮雕映射（relief mapping）等。其关键思想是：首先沿着观察方向的投影向量，对固定数量的高度场纹理样本进行测试。在掠射观察角度下，通常会生成数量更多的样本，以便不会遗漏这个最近交点[1742, 1743]。会对沿着光线方向的每个三维样本位置进行检索，并将其转换到纹理空间中进行处理，从而确定该位置是高于该点的高度场，还是低于该点的高度场。一旦找到了一个低于高度场的样本，就使用这个低于高度场的样本，与前一个高于高度差的样本，来找到一个交点位置，如图 6.39 所示。然后使用额外的法线贴图、颜色贴图以及任何其他纹理，来对该位置进行着色。使用多层高度场，可以用于生产悬垂结构（overhang）、独立的重叠表面和双面浮雕等效果，详见章节 13.7。高度场追踪方法也可以用来让凹凸不平的表面在自身上投射阴影，包括硬阴影[1171, 1424]和软阴影[1742, 1743]，如图 6.40 所示。

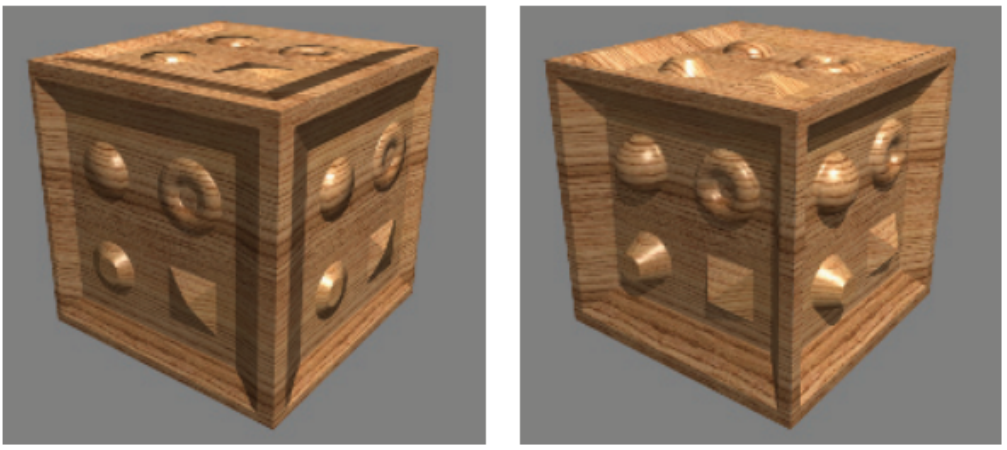


图 6.40：没有使用光线步进的视差映射（左）与使用光线步进的视差映射（右）之间的对比。在没有使用光线步进的时候，立方体的顶部会变得很平坦（因为顶部处于掠射观察视角）。使用光线行进也可以产生自阴影效果。

关于这个话题，有着大量的相关文献，虽然所有的这些方法都是沿着一条光线进行步进的，但是它们之间也有一些不同之处。有些方法可以直接使用简单的纹理来检索高度数据，但是也可以使用更加高级的数据结构，以及更加高级的求根方法（寻找交点）。有些技术还可能涉及到着色器丢弃像素或者写入深度缓冲区的操作，这些额外的操作可能会对性能产生影响。下面我们对大量方法进行了总结，但是请你记住，随着 GPU 的不断发展，最好的方法也会不断发展。这种“最佳”方法取决于生成的效果，以及光线步进过程中所需的步进次数。

确定两个均匀间隔样本之间的交点，实际上是一个求根问题。实际上，在实践中，高度场更多地会被视为深度场，每个纹素的矩形平面代表了改表面的最大高度；这样平面上的初始点会位于高度场之上。在找到了位于高度场表面上方的最后一个点，以及位于表面下方的第一点之后，Tatarchuk [1742, 1743]使用了一次迭代的割线法（secant method）来找到近似解。Policarpo 等人[1424]在这两个点之间，使用二分查找（binary search）来寻找最近的交点。Risser 等人[1497]使用迭代的割线法来加快收敛速度，这里涉及一个权衡问题：固定间隔的采样可以通过并行完成；迭代方法虽然在整体上会进行较少的纹理访问操作，但是它在执行的过程中必须等待结果的返回，并执行效率较低的依赖纹理读取。暴力求解方法（brute-force methods）总体上表现良好[1911]。

对于高度场进行足够多的采样是很重要的。McGuire 和 McGuire [1171]建议对 mipmap 查找进行偏移处理，并使用各向异性的 mipmap 来确保对高频高度场（例如尖刺或者头发）的正确采样。我们也可以用比法线贴图更高的分辨率来存储高度场纹

理。一些渲染系统甚至不会去存储法线贴图，它们更喜欢使用交叉滤波器（cross filter），来从高度场中动态生成法线信息[40]，方程 16.1 展示了这个方法。

还有一种提高性能和采样精度的方法，即不以固定的间隔来对高度场进行初始采样，而是尝试跳过中间的空白区域，即在光线步进的过程中，动态调整下一次步进的步长。Donnelly [367]将高度场预处理成一组体素，在每个体素中存储它距离高度场表面的距离，在这种方式下，光线在步进过程中，可以快速跳过中间的空白区域，其代价是增加了每个高度场的存储空间。Wang 等人[1844]使用了五维位移映射（five-dimensional displacement mapping）的方案，来记录从各个方向和各个位置到地面的距离，这种方案支持更加复杂的表面、自阴影以及其他效果，其代价是内存开销会很大。Mehra 和 Kumar [1195]使用了定向距离图（directional distance map）来达到类似的目的。Dummer [393]提出了锥形步进映射（cone step mapping）的思想，Policarpo 和 Oliveira [1426]对其进行了改进。这里的核心思想是，为每个高度场位置存储一个圆锥半径（cone radius），这个半径定义了在射线方向上，与高度场最多只有一个交点的区间。尽管这个方法需要依赖纹理读取，但是它允许沿着射线方向进行快速跳跃，并且不会错过任何可能的交点；这种方法的另一个缺点是，它需要进行预计算来生成锥形步进图，这使得该方法无法用于动态变化的高度场。Schroders 和 Gulik [1426]提出了四叉树浮雕映射（quadtree relief mapping）方法，这是一种在遍历过程中跳过空间的分层方法。Tevs 等人[1426]使用了“最大化 mipmap”来支持动态步长，同时最小化了预计算成本，这里的“最大化 mipmap”是指，在生成 mipmap 的时候，我们并不是取上一级像素颜色的平均值，而是取其中的最大值来作为下一级 mipmap 的值（译者注：这种结构类似于 Hi-z）。Drobot [377]还使用了存储在 mipmap 中的类四叉树结构，来遍历进行加速，并提出了一种在不同高度场之间进行混合的方法，它允许一种地形类型转换为另一种地形类型。

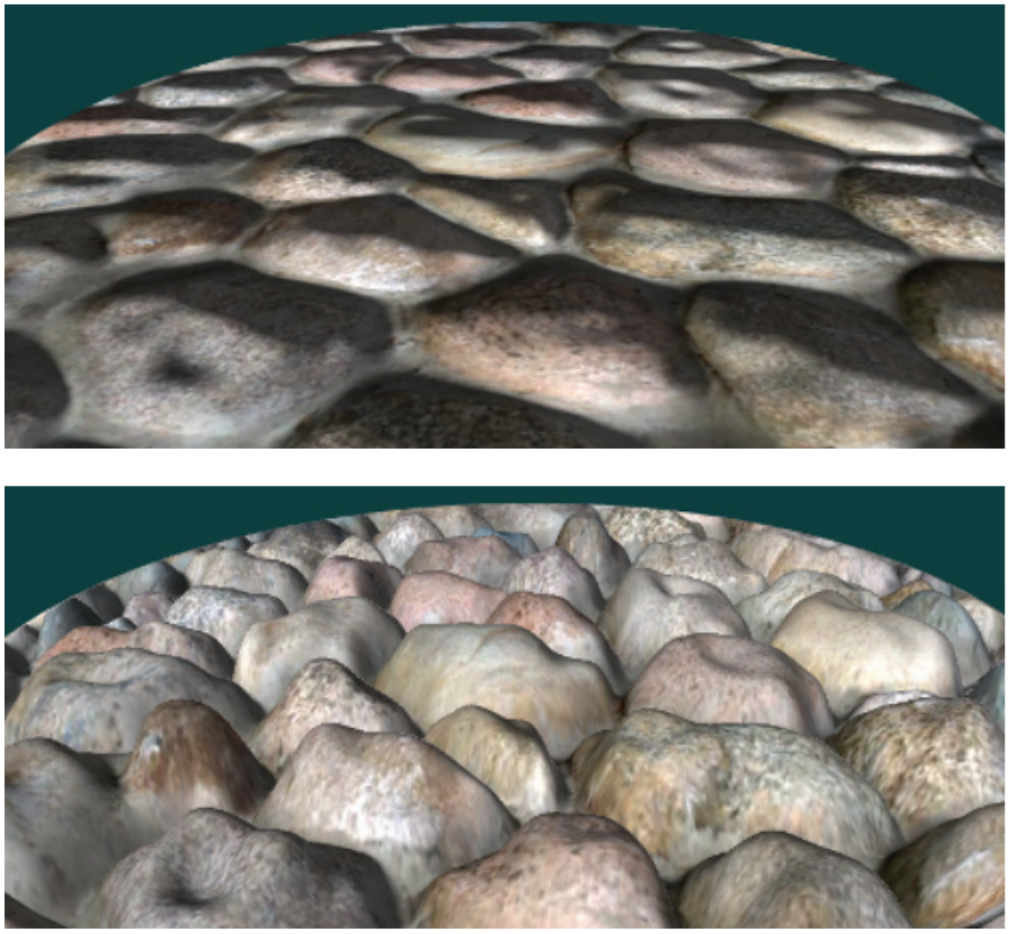


图 6.41：法线映射和浮雕映射。法线映射不会产生自遮挡现象；浮雕映射在具有重复纹理的轮廓边缘上会出现问题，因为纹素矩形更像是高度场的视图，而不是真正的边界定义。

上述所有方法都存在一个问题，即这种视差错觉会在物体的轮廓边缘处消失，转而会显示出原始表面的平滑轮廓，如图 6.41 所示。这个问题的关键在于，被渲染的三角形仅仅定义了哪些像素应当由像素着色器进行计算，而不是表面上的实际位置，因此位于三角形之外的像素都不会受到视差映射的影响。此外，对于曲面而言，视察映射的轮廓问题将会变得更加复杂。Oliveira 和 Policarpo [1325, 1850]描述并开发了一种使用二次轮廓近似技术的方法。Jeschke 等人[824]和 Dachsbacher 等人[323]对之前的工作进行了回顾，并给出了一种更加通用且健壮的方法，能够正确处理轮廓和曲面等情况。Hirche [750]首先提出了将网格中的每个三角形向外挤压形成棱柱（prism）的想法，然后渲染这个棱柱，计算所有可能会出现高度场中的像素。这种类型的方法称为外壳映射（shell mapping），因为向外扩展的网格在原始模型上，形成一个单独的外壳。尽管计算成本很高，但是通过在与光线相交时保持棱柱的非线性特性，可以实现无瑕疵的高度场渲染。图 6.42 展示了这种令人印象深刻的技术案例。

图 6.42：视差遮挡映射，又名浮雕映射，将其应用在石径上，可以使得石头看起来更加真实。图中的石径地面实际上是一组简单的三角形，外加一个高度场。

6.9 纹理光源

纹理还可以用于为光源添加更加丰富的视觉表现，并且允许光源具有非常复杂的强度分布函数或者聚光灯函数。对于所有光线都限制在一个圆锥（cone），或者一个截锥体（frustum）内的光源而言，投影纹理（projective texture）可以用来调节光线的强度[1192, 1597, 1904]。这样就可以实现有形状的聚光灯、有图案的光源、甚至是“幻灯片投影仪”效果（如图 6.43 所示）。这些光源通常被称为 gobo 或者 cookie，这是在专业剧院和电影照明中用于描述 cutou 镂空（cutout）的术语。在章节 7.2 节，我们讨论了如何使用类似于投影映射的方法，来实现阴影效果。

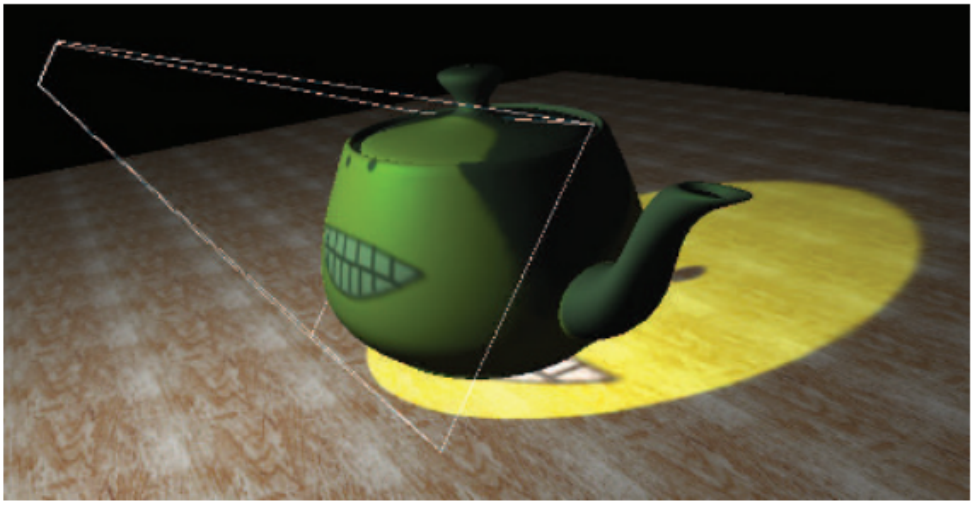


图 6.43：投影纹理光源。纹理被投影到茶壶和地面上，并用于控制投影截锥体内的光线分布（位于截锥体外的光线，其强度为 0）。

对于不局限于截锥体范围，而是照亮所有方向的光源而言，可以使用立方体贴图来调节它的强度，而不是像其他光源一样使用二维的投影纹理。一维纹理可用于定义任意的距离衰减函数；将它与二维角度衰减贴图结合，可以实现复杂的体积照明模式[353]。一种更加普遍的做法是，使用三维（体积）纹理来控制光线的衰减[353, 535, 1192]，这种方法可以实现包括光束（light beam）在内的任意体积效果。与其他所有的体积纹理一样，这种技术方案也是内存密集型的。如果光源的体积效果沿三个轴对称的话，那么可以将数据在每个八个象限中进行镜像，从而将内存占用减少到原来的八分之一。

任何类型的光源都可以添加纹理，从而实现额外的视觉效果。纹理光源允许艺术家们，通过对纹理进行编辑和修改，从而很轻松地控制照明效果。

补充阅读和资源

Heckbert 撰写了一篇关于纹理映射理论的良好综述[690]，以及一篇有关该主题的、更加深入的报告[691]。Szirmay-Kalos 和 Umenhoffer [1731]对视差遮挡映射和位移（displacement）方法进行了出色而深入的研究。有关法线表示的更多信息，可以在 Cigolle 等人[269]和 Meyer 等人[1205]的工作中找到。

《Advanced Graphics Programming Using OpenGL》[1192]一书广泛涵盖了使用纹理算法的各种可视化技术。有关三维程序化纹理的更多信息，你可以参考《Texturing and Modeling: A Procedural Approach》[407]一书。《Advanced Game Development with Programmable Graphics Hardware》[1850]一书中包含了很多实现视差遮挡映射的技术细节，Tatarchuk 的演讲[1742, 1743]以及 Szirmay-Kalos 和 Umenhoffer 的调查[1731]也包含了这部分内容。

对于程序化纹理和程序化建模，我们在互联网上最喜欢的网站就是 Shadertoy，该网站上展示了许多有价值、并且令人着迷的程序化纹理函数，并且你可以很轻松地修改任何示例，并查看相应的渲染结果。

你还可以访问本书的配套网站 realtimerendering.com，来获取更多的补充资源。

Chapter 7 Shadows 阴影

> Tolstoy——“All the variety, all the charm, all the beauty of life is made up of light and shadow.”

> 列夫·托尔斯泰——“属于生命的一切多样性，一切魅力，一切美好，都是由光和影构成的。”（19 世纪中期俄国批判现实主义作家；1828—1910）

对于创建逼真的图像、为用户提供物体位置的视觉暗示而言，阴影效果是非常重要的。本章节将重点介绍计算阴影的基本原理和基本思想，并对最重要、最流行的几种实时阴影算法进行介绍。我们还会简要讨论一些不太流行，但是体现了重要原则和重要思想的方法。我们并不会在本章节中花费大量时间来覆盖所有的内容和方法，因为已经有两本非常全面的书籍[412, 1902]，对阴影领域进行了深入研究和讨论。相反，我们将专注于调查文献（surveying article）和相关技术演讲，这些内容在出现之前，相关方法就已经被提出并实现了。也就是说，本书主要侧重于经过实战检验的阴影技术。

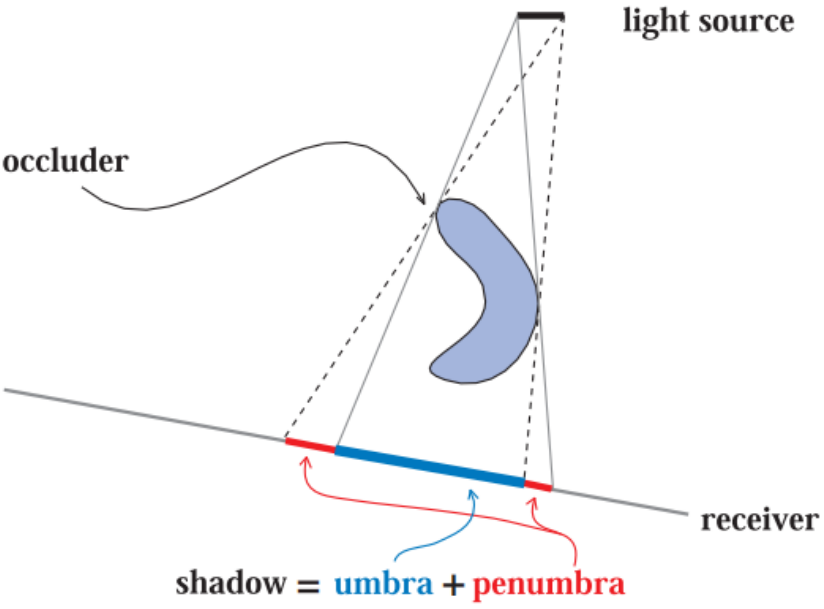


图 7.1：阴影相关的术语：光源（light source），遮挡物（occluder），接收物（receiver），阴影（shadow），本影（umbra）和半影（penumbra）。

本章节中所使用的相关术语（terminology）如图 7.1 所示，其中遮挡物（occluder）是指向接收物（receiver）投射阴影的物体。精确光源（定点光源），即那些没有面积的光源，只会产生完全被阴影覆盖的区域（本影），这样的阴影有时候会被称为硬阴影（hard shadow）。如果使用了面光源或者体积光源，那么则会产生软阴影（soft shadow）。每个阴影都有一个完全被阴影覆盖的区域，称为本影（umbra）；以及一个部分被阴影覆盖的区域，称为半影（penumbra）。我们可以通过模糊的阴影边缘，来辨别出软阴影；然而需要注意的是，仅仅使用低通滤波器来对硬阴影的边缘进行模糊，通常是无法正确渲染出软阴影效果的。从图 7.2 中我们可以看出，一个正确渲染的软阴影应当有这样的现象，即遮挡物越靠近接收物，阴影的边缘就会越清晰。软阴影的本影区域并不等同于由精确光源所产生的硬阴影；相反，软阴影的本影区域会随着光源的变大而减小。如果光源足够大，或者接收物距离遮挡物足够远的话，那么本影区域甚至可能会完全消失。通常我们都更加希望实现软阴影效果，因为半影的边缘会让观众知道这个黑色的区域确实是一个阴影。而硬阴影具有明确的阴影边界，它通常看起来不太真实，有时候甚至会被误解为实际的几何特征，比如表面上的折痕等。然而，硬阴影的渲染速度要比软阴影快得多。

图 7.2：图中包含了硬阴影和软阴影。当遮挡物靠近接收物的时候，箱子所产生的阴影是尖锐的。人所产生的阴影在接触点（脚部）附近是尖锐的，随着与遮挡物之间的距离逐渐增加，阴影边缘也逐渐软化。远处的树枝也会产生柔和的阴影效果。[1711]

比半影效果更加重要的是先有阴影。如果没有阴影作为视觉暗示的话，场景往往看起来会很假，更加难以进行感知。正如 Wanger 所说的那样[1846]，有一个不准确的阴影，通常要比完全没有阴影强得多，因为人类的眼睛对于阴影的形状是相当宽容的。例如：在地板上应用一个纹理化的模糊黑色圆圈，就可以让观察者感觉这个人确实在地面上。

在接下来的几个小节中，我们将超越这些简单的建模阴影，介绍一些根据场景中的遮挡物，实时自动计算阴影的方法。第一部分我们首先会先处理在平面上投射阴影的特殊情况，第二部分则会涵盖更加一般的阴影算法，即在任意表面上投射阴影的算法。同样我们会介绍硬阴影和软阴影的实现方法。最后我们会介绍一些适用于各种阴影算法的优化技术。

7.1 平面阴影

最简单的情况就是物体在平面上投射阴影。本小节将会介绍几种平面阴影（planar shadow）算法，每种算法在阴影的柔和度和真实感方面都有所不同。

7.1.1 投影阴影

在这个方法中，三维物体会被渲染两次，其中第二次用于创建阴影。我们可以推导出这样一个矩阵，它将物体的顶点投影到一个平面上[162, 1759]。考虑图 7.3 中的情况，光源位于点 $\mathbf{l}$，待投影的顶点位于点 $\mathbf{v}$，投影后平面上对应的顶点位于点 $\mathbf{p}$。这里我们先推导一个特殊情况，即阴影平面位于 $y = 0$ 时的投影矩阵，然后再将这个结果推广到任意平面上。

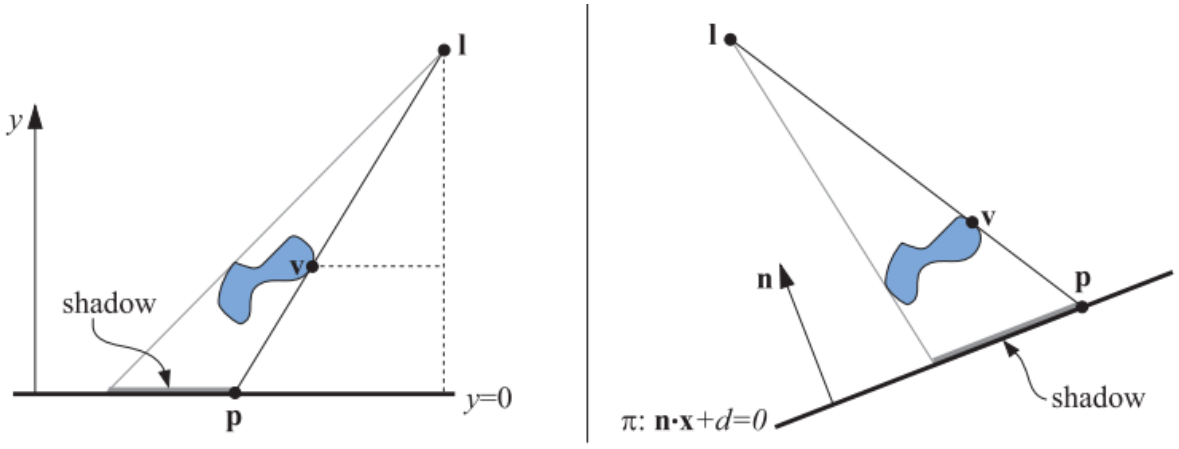


图 7.3：左图：位于点 $\mathbf{l}$ 的光源，在平面 $y=0$ 上投下阴影。顶点 $\mathbf{v}$ 被投影到平面上，所产生的投影点被记作点 $\mathbf{p}$，可以利用相似三角形来推导投影矩阵。右图：阴影投射到任意平面 π：$\mathbf{n}\cdot x+d=0$ 上。

首先我们推导出 x 坐标上的投影操作。根据图 7.3 左侧的相似三角形，我们可以得到：

$$\frac{p_x-l_x}{v_x-l_x}=\frac{l_y}{l_y-v_y}\quad\Longleftrightarrow\quad p_x=\frac{l_yv_x-l_xv_y}{l_y-v_y}. \tag{7.1}$$

同理，z 坐标为 $p_z=(l_yv_z-l_zv_y)/(l_y-v_y)$，由于投影平面位于 $y=0$ 处，因此投影后顶点的 y 坐标为 0。然后我们可以将这些方程转换为投影矩阵 $\mathbf{M}$：

$$\mathbf{M}=\begin{pmatrix} l_y & -l_x & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & -l_z & l_y & 0\\ 0 & -1 & 0 & l_y \end{pmatrix} \tag{7.2}$$

我们可以很容易验证 $\mathbf{Mv}=\mathbf{p}$，这意味着矩阵 $\mathbf{M}$ 确实是个投影矩阵。

在一般情况下，阴影投射到的平面并不是 $y=0$，而是任意平面 $\pi:\mathbf{n}\cdot x+d=0$，图 7.3 右侧展示了这种一般情况。我们的目标是找到一个矩阵，来将点 $\mathbf{v}$ 投影到点 $\mathbf{p}$。为了实现这个目的，我们让光线从点 $\mathbf{l}$ 发出，经过点 $\mathbf{v}$，最终与与平面 π 相交，这样我们就获得了投影点 $\mathbf{p}$，这个过程的数学形式如下：

$$\mathbf{p}=\mathbf{l}-\frac{d+\mathbf{n}\cdot\mathbf{l}}{\mathbf{n}\cdot(\mathbf{v}-\mathbf{l})}(\mathbf{v}-\mathbf{l}). \tag{7.3}$$

方程 7.3 也可以转换为投影矩阵，如方程 7.4 所示，这个矩阵也满足 $\mathbf{Mv}=\mathbf{p}$：

$$\mathbf{M}=\begin{pmatrix} \mathbf{n}\cdot\mathbf{1}+d-l_xn_x & -l_xn_y & -l_xn_z & -l_xd\\ -l_yn_x & \mathbf{n}\cdot\mathbf{1}+d-l_yn_y & -l_yn_z & -l_yd\\ -l_zn_x & -l_zn_y & \mathbf{n}\cdot\mathbf{1}+d-l_zn_z & -l_zd\\ -n_x & -n_y & -n_z & \mathbf{n}\cdot\mathbf{l} \end{pmatrix} \tag{7.4}$$

按照预期，当投影平面为 $y=0$，即 $\mathbf{n}=(0,1,0)$，$d=0$ 时，方程 7.4 中的矩阵会退化为方程 7.2 中的矩阵。

想要渲染平面阴影效果，只需要将这个矩阵应用到要在平面 π 上投射阴影的物体上，然后再将投影后的物体渲染为深色，并且不接收光照即可。在实践中，我们还必须采取一些措施，来避免让投影后的三角形被渲染到投影表面之下。一种可行的方法是对投影平面进行一些偏移，使得投影后的三角形总是会渲染在投影平面的前面。

一种更加稳妥安全的方法是，先绘制投影平面，然后在关闭 z-buffer 的情况下，再去绘制投影后的三角形，然后再像往常一样渲染剩余的几何图形。这样投影后的三角形总是会绘制在投影平面的前面，因为这里我们关闭了深度测试。

如果投影平面存在边界，例如它是一个矩形，那么阴影可能会落在矩形范围之外，这看起来会很奇怪。为了解决这个问题，我们可以使用一个模板缓冲区（stencil buffer），首先将接收物分别绘制到屏幕和模板缓冲区上，然后关闭 z-buffer，使用模板缓冲区作为遮罩，只在有颜色的地方绘制投影后的三角形，最后再正常渲染场景的剩余部分。

另一种阴影算法是将三角形渲染到一个纹理中，然后再将其应用到投影平面上。这个纹理其实是一种光照贴图（light map），它可以调节底层表面的强度（详见章节 11.5.1）。正如我们将看到的，这种将投影阴影渲染到纹理中的思路，也可以在曲面上产生半影效果和阴影效果。这种技术的一个缺点在于，纹理可能会被放大，使得单个纹素会覆盖多个屏幕像素，从而产生一些阴影锯齿。

如果画面中的阴影情况没有发生变化的话，即光源和遮挡物之间没有发生相对移动，那么这个纹理便可以重复使用。实际上，如果阴影情况没有发生变化的话，大多数阴影技术都可以对中间的计算结果进行重用。

所有的遮挡物必须位于光源和地面接收物之间。如果光源位于遮挡物下方，那么则会出现反阴影（antishadow）现象[162]，因为遮挡物上的每个顶点，都会通过光源所在的位置进行投影。图 7.4 展示了正确的阴影效果，以及错误的反阴影效果。如果我们要投影一个位于接收平面以下的物体，此时也会出现错误，因为事实上它不应该会产生阴影。

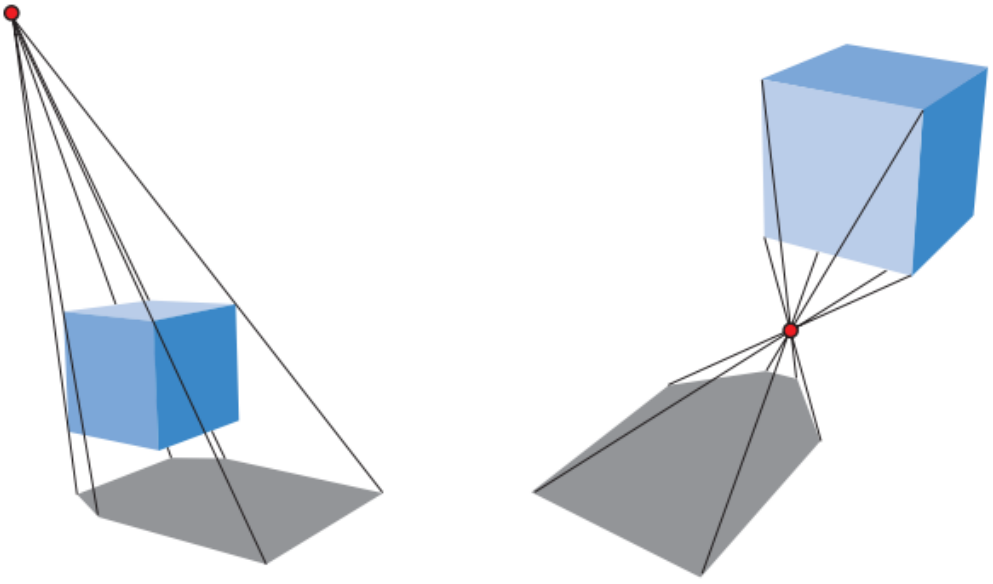


图 7.4：左图展示了一个正确的阴影效果；右图中则出现了一个错误的反阴影效果，因为光源低于在物体最顶端的顶点。

我们可以对生成的阴影三角形进行显式剔除和修剪，从而避免上述的诸多瑕疵。下面我们将介绍一种更加简单的方法，它利用现有的 GPU 管线来执行带有剔除的投影操作。

7.1.2 软阴影

通过使用各种技术，投影阴影也可以变得很柔和。在本小节中，我们将介绍 Heckbert 和 Herf [697, 722]提出的一种算法，它可以产生软阴影效果。该算法的目标是在地平面上生成一个纹理，这个纹理中包含了一个软阴影。然后，我们还会介绍一种不太准确，但是效率更高的方法。

当光源具有大小和面积时，便会形成软阴影。一种对面光源进行近似的方法是，在其表面上放置几个精确光源，来模拟对面光源的采样。对于每个精确光源，都会单独计算它的光照效果，将其存储在一张图像中，再将图像累积到缓冲区中；这些图像的平均值就是一幅带有软阴影的图像。请注意，在理论上而言，任何可以生成硬阴影的算法都可以用这种累积技术来生成半影效果；但是在实践中，由于应用的交互性限制了每帧画面的生成时间，因此这种方法通常是不可取的，它的效率实在是太低了。

Heckbert 和 Herf 使用一种基于视锥体的方法来生成阴影，这个方法的核心思路是将光源作为一个观察者，将地面作为视锥体的远裁剪平面；这个视锥体足够宽，可以讲所有的遮挡物包含在内。

通过生成一系列地面纹理，并将其累积叠加在一起，便可以形成一个包含软阴影的纹理。我们在面光源表面上进行采样，将每个采样点作为一个精确光源，并在每个光源位置上渲染代表地面的图像，然后将遮挡物投影到该图像上。将所有的这些图像相加并取平均值，便可以生成地面的阴影纹理，如图 7.5 左侧所示。

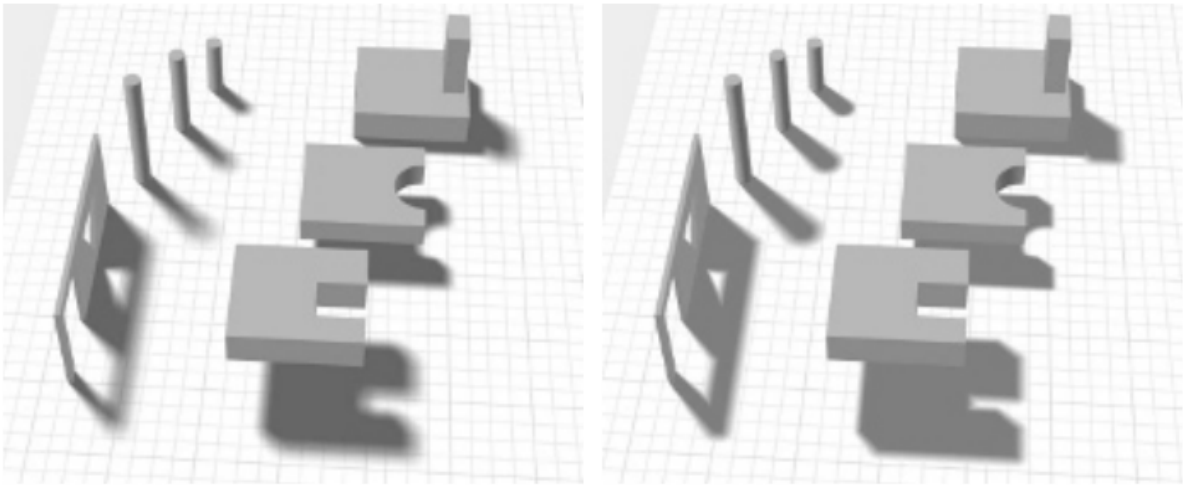


图 7.5：左图使用了 Heckbert 和 Herf 的方法进行渲染，一共包含 256 个 pass。右图是 Haines 的单 pass 方法，但是在 Haines 的方法中，本影的范围太大了，尤其是在门口和窗户附近。

对面光源进行采样，然后逐个生成阴影的方法存在一个问题，它看起来像是几个来自精确光源的阴影相互重叠在一起，而不是面光源自然形成的软阴影。而且，对于 n 个阴影 pass，它只能生成 $n+1$ 个不同的阴影；虽然可以使用大量 pass 来得到更加准确的结果，但是计算开销实在过高。这种方法可以用于获取 ground-truth 图像，以便对其他更快算法的质量进行测试对比。

更加高效的方法是使用卷积（即滤波）。在某些情况下，对单个精确光源生成的硬阴影进行模糊，效果就足够了，并且可以生成半透明的纹理，从而与场景中的其他内容进行合成，如图 7.6 所示。然而，在物体和地面相接触的地方，均匀的模糊看起来可能会很假。

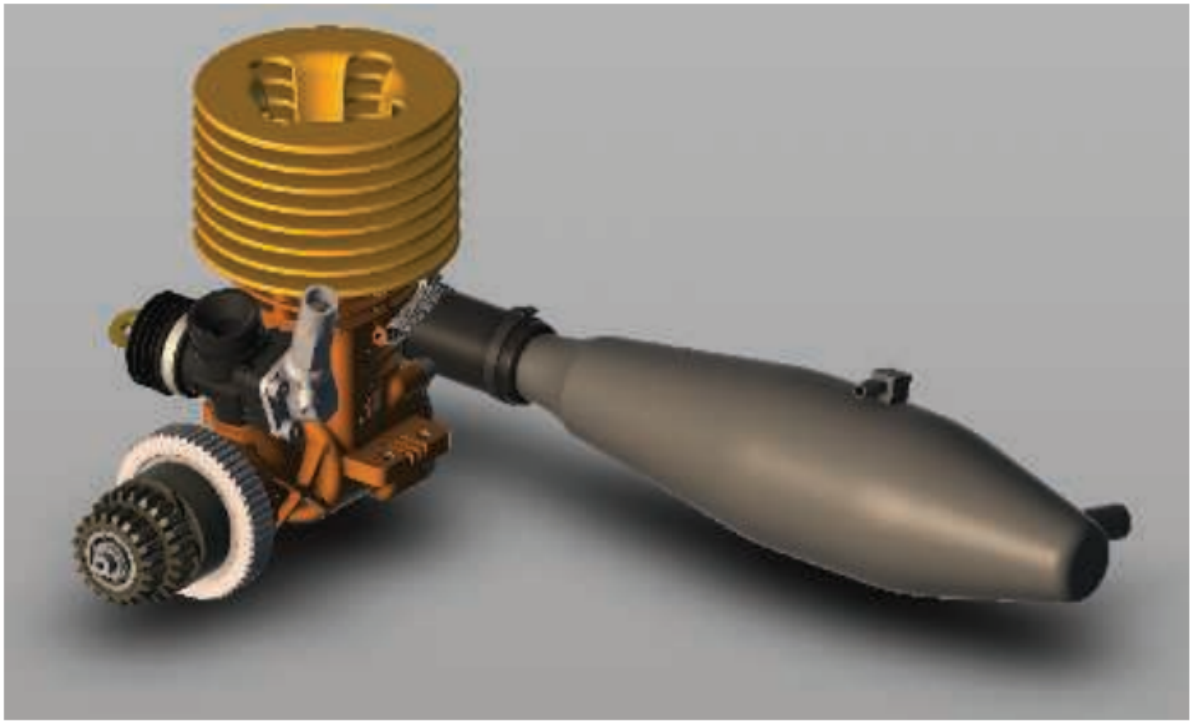


图 7.6：下落阴影（drop shadow）。从上方对遮挡物进行渲染，然后再对图像进行模糊处理，最后应用在地平面上，从而形成了图中的阴影纹理。

还有许多其他方法可以提供更好的近似软阴影效果，但是需要一些额外的开销。例如：Haines [644]从投影生成的硬阴影出发，用从阴影中心（黑色）到阴影边缘（白色）的渐变，来渲染阴影的轮廓边缘，从而生成看起来很真（plausible）的半影效果，如图 7.5 右侧所示。然而，这些模拟出来的半影效果在物理上是不正确的，因为它们还应当延伸到轮廓边缘内的区域中。Iwanicki [356, 806]借鉴了球谐函数（spherical harmonic）的思想，使用椭球体来对遮挡物进行近似，从而产生柔和的阴影。所有的这些方法，都使用了各种不同的近似方法，也都有各自的缺点，但是相比于对大量阴影图像进行平均，这些算法要高效得多。

7.2 曲面上的阴影

可以将平面阴影的思想和方法扩展到曲面上，其中一个简单的方法是，将生成的阴影图像作为一个投影纹理 [1192, 1254, 1272, 1597]。我们可以从光源的角度来理解阴影，凡是可以被光源看见的物体都会被照亮，而光源看不到的区域就是阴影区域。假设遮挡物从光源的视角被渲染为黑色纹理（而不是白色），然后这个黑白纹理可以被投影到接收阴影的表面上。实际上，接收物上的每个顶点，都会为其计算一个 (u, v) 纹理坐标，并将阴影纹理应用在其表面上，接收物上的这些纹理坐标可以由应用程序显式计算而来。这与上一小节中的地面阴影纹理有一些不同，在上一小节中，物体会

被投影到特定的物理平面上；而在本小节的方法中，这些阴影图像是由光源渲染形成的，就像放映机中的一帧底片一样。

在渲染的时候，投影后的阴影纹理会对接收物的表面进行修改，它也可以与其他的阴影方法结合使用，有时候，它主要是用于帮助感知物体的位置。例如：在一个平台跳跃游戏中，即使角色完全处于阴影中[1343]，其下方可能总是会存在一个下落阴影，这个阴影主要是用来帮助玩家感知角色的位置。一些更加复杂精细（elaborate）的算法可以生成更好的结果，例如：Eisemann 和 Decoret [411]预设了一个矩形的顶光源，并生成了一个包含物体水平切片的阴影图像栈，然后再将其转换为 mipmap 或者类似的结构。通过使用 mipmap，每个切片的对应区域（即对应的 mipmap 层级）与其到接收物的距离成正比，这意味着越远的切片将会投射出更加柔和的阴影（mipmap 层级越高越模糊）。

当然，纹理投影方法存在着一些严重的缺陷。首先，应用程序必须识别出哪些物体是遮挡物，哪些物体是阴影的接收物。同时，应用程序必须保证接收物要比遮挡物到光源的距离更远，否则阴影便会“向后投射”，即上一节中提到的反阴影现象。此外，这里的遮挡物是无法遮挡自己的，也就是说无法产生自阴影效果。在接下来的两个小节中，我们将介绍生成正确阴影的算法，它们并不需要这种干预或者限制。

值得注意的是，各种光源的照明模式可以通过使用预先创建的投影纹理来进行定义。例如聚光灯的投影纹理是一个简单的正方形，里面有一个圆来定义光源的照明范围。可以使用由水平线构成的投影纹理，来生成百叶窗（venetian blind）效果。这种类型的纹理被称为光线衰减遮罩（light attenuation mask）、cookie 纹理或者 gobo 贴图。还可以简单地将两个纹理进行相乘，从而将预先构建的光照模式，与动态创建的投影纹理相结合，此类光源在章节 6.9 中进行了深入讨论。

7.3 阴影体算法

基于 Crow 的阴影体（shadow volume）思想[311]，Heidmann 于 1991 年提出了一种改进的方法[701]，该方法巧妙地利用了模板缓冲区，可以在任意物体上投射阴影。它可以在任何 GPU 上进行使用，因为该方法的唯一要求就是，GPU 要支持模板缓冲区。该方法并不是基于图像的（与下一小节中所描述 shadow map 算法不同），因此它避免了采样的问题，可以在各处产生正确且锐利的阴影效果。有时候，这反而可能是一个缺点，例如：角色的衣服上可能会有褶皱，它们会产生薄而硬的阴影。由于该算法的计算成本难以预测[1599]，因此现在已经很少使用阴影体算法了。在这里，我们将对该算法进行简要介绍，因为该算法涉及了一些重要原理，我们将在此基础上进行后续的讨论。

首先，让我们想象一个点和一个三角形，将这个点和三角形的顶点相连接，并延伸到无穷远处，这样就形成了一个底面无穷远的三面金字塔。位于三角形下方的部分（即不包括这个点的部分），是一个被截断的无限金字塔；而位于三角形上方的部分，则只是一个简单的金字塔，如图 7.7 所示。现在我们假设这个点实际上是一个点光源，任何位于截断金字塔内部（在三角形下方）的物体都处于阴影中，这个空间便被称为 shadow volume，一般叫做阴影体或者阴影锥。

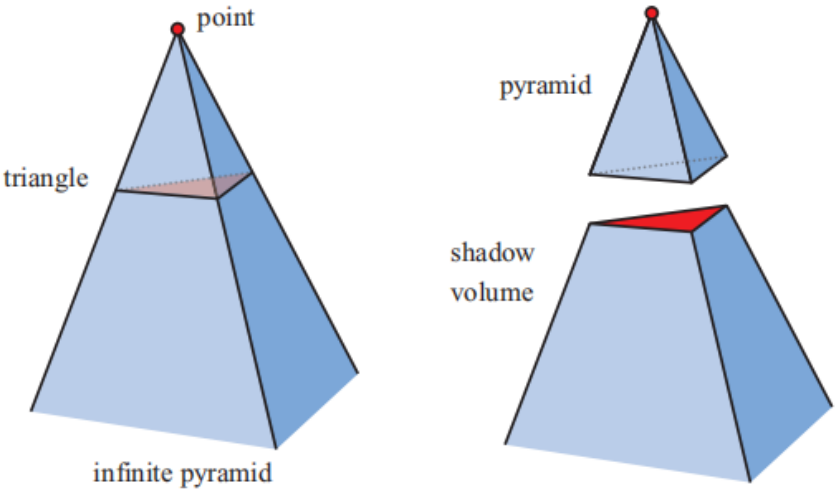


图 7.7：左图：点光源发出的光线穿过三角形的顶点，并向远处延伸，形成一个无限的金字塔。右图：上半部分是一个金字塔，下半部分是一个被截断的无限金字塔，又被称为阴影体（shadow volume）。所有位于阴影体内部的几何体都处于阴影中。

假设我们现在正在观察某个场景，沿着从眼睛发出并穿过一个像素的光线，直到这个光线击中了要显示在屏幕上的场景物体。在光线从发出到与场景物体相交的过程中，每当光线穿过了阴影体的正面（即面向观察者的一面）时，我们就让计数器加 1；也就是说，每当光线进入阴影区域时，计数器就会增加。以同样的方式，每当光线穿过截断金字塔的背面时，我们便将相同的计数器减 1，这代表了光线从阴影区域中射出。我们会一直持续这个过程，增加或者减少计数器的值，直到光线击中了场景中的物体。此时，如果计数器大于 0，则说明该像素位于阴影中；如果计数器小于 0，则说明该像素位于阴影之外。当场景中有多个三角形投射阴影时，这个原则也同样适用，如图 7.8 所示。

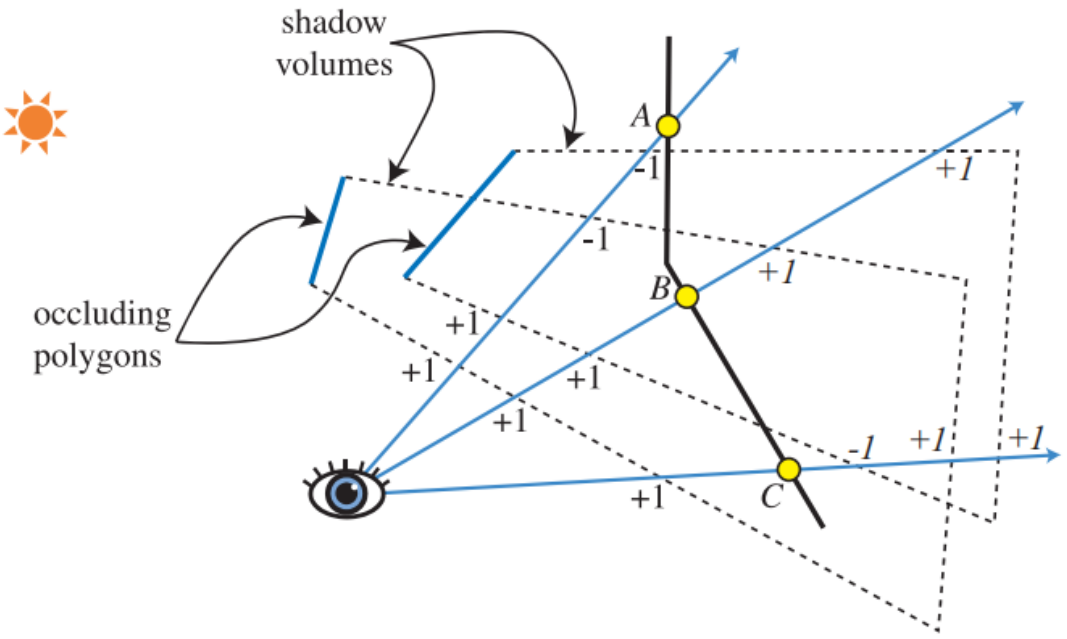


图 7.8：使用两种不同的计数方法来对阴影体交点进行计数，上图展示了这个过程的二维侧视图。在 z-pass 计数方法中，当光线穿过阴影体的正面三角形时，会增加计数；当光线穿过阴影体的背面三角形时，会减少计数。因此对于图中的点 A 而言，光线进入了两个阴影体，计数加 2；然后又离开两个阴影体，计数减 2，最后的净计数为 0，因此点 A 并不在阴影中，会被光源照亮。在 z-fail 计数方法中，会从表面以外的地方开始计数（这些计数在图中以斜体进行显示）。对于通过点 B 的光线而言，如果使用 z-pass 计数方法，则光线通过了两个正面三角形，计数加 2；如果使用 z-fail 计数方法，该光线通过了两个背面三角形，计数也加 2。点 C 展示了如何对 z-fail 的阴影体进行限制。经过点 C 的光线首先穿过了一个正面三角形，计数减 1；然后又离开了两个阴影体（通过了结束端点，这是该方法正常运行所必要的条件），计数加 2。最终的计数结果不为 0，因此该点位于阴影中。这两种方法都可以对可见表面上的点，给出相同的计数结果。

使用射线来完成这件事是很耗时的[701]，有一个更加聪明的解决方案，即使用模板缓冲区，模板缓冲区可以帮助我们完成这个计数过程。第一步，清空模板缓冲区。第二步，将场景中使用无光材质的物体渲染到帧缓冲中，并将着色结果写入颜色缓冲区中，将深度信息写入到 z-buffer 中。第三步，关闭对 z-buffer 和颜色缓冲的更新（当然我们仍然会使用 z-buffer 来进行深度测试），然后我们绘制阴影体的正面三角形。在这个过程中，我们将模板操作被设置为：在绘制三角形的地方，模板缓冲区中对应位置上的值加 1。第四步，使用模板缓冲区来执行另一个 pass，这次我们只绘制阴影体的背面三角形。在这个 pass 中，我们将模板操作被设置为：在绘制三角形的地方，模板缓冲区中对应位置上的值减 1。只有当阴影体各个表面上的像素可以被相机看见时（即没有被其他几何图形遮挡），模板缓冲区中的值才会增加和减少。在这个 pass 完成之后，模板缓冲区中保存了屏幕上每个像素的阴影状态（即是否位于

阴影区域中）。最后，我们再次渲染整个场景，这次只会渲染使用受光材质的物体，并且渲染的结果只会显示在模板缓冲区中值为 0 的地方，值为 0 代表了光线离开阴影体与它进入阴影体的次数相等，也就是说，这个像素位置位于阴影之外，它会被光源照亮。

这种计数方法是阴影体算法背后的基本思想。图 7.9 展示了一个由阴影体算法生成的阴影。还有一些更加高效的方法，可以在一个 pass 中实现这个算法[1514]。然而，当场景物体穿过相机的近裁剪平面时，会出现计数错误的问题；还有一种被称为 z-fail 的解决方法，它会对隐藏在可见表面后面（而不是前面）的交点进行计数[450, 775]。图 7.8 对这个方法进行了简要概述。

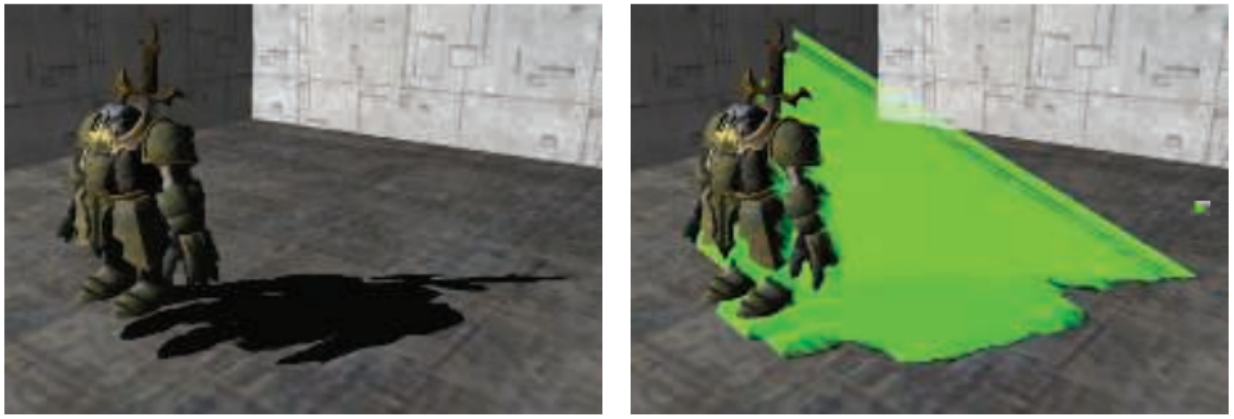


图 7.9：阴影体算法。左图中的角色投射了一个影子，右边展示了这个模型所产生的阴影体三角形。[1280]

由于阴影体金字塔有三个面，也就是说，对于每个会能够阴影的三角形而言，我们都要为其创建三个四边形（对应截断金字塔的四个面），这会导致大量的过度绘制（overdraw）。一个由 1000 个三角形组成的球体，会创建 3000 个四边形，并且每个四边形都可能会横跨整个屏幕。一种解决方案是，只沿着物体的轮廓边缘来绘制这些四边形，例如：上述这样一个球体可能只有 50 条轮廓边缘[1702]，因此只需要生成 50 个四边形即可。我们可以使用几何着色器，来自动生成这样的轮廓边缘。一些剔除和 clamp 技术，也可以用于降低填充成本[1061]。

然而，阴影体算法有着一个严重的缺点，那就是极端的不稳定性。想象现在场景中有一个简单的小三角形，如果相机和光源位于同一位置，那么此时阴影体算法的开销是最小的，因为它所形成的四边形不会覆盖屏幕上的任何像素，它们都侧面朝向相机，完全不可见；只有这个三角形本身会影响渲染结果。假设现在相机围绕这个三角形进行旋转，并让它一直处于视野中。随着相机远离光源，阴影体的四边形将会变得越发可见，覆盖更多的屏幕像素，从而导致需要更多的计算开销。如果相机恰好移动到这个三角形所形成阴影区域中，那么阴影体将会覆盖整个屏幕；与我们最初的观察视野

相比，这种情况需要花费大量的时间来进行阴影计算。这种不稳定性使得阴影体算法在交互式应用中几乎无法使用，因为对于这些应用而言，稳定的帧率是非常重要的。相机看向光源的方向也可能会导致算法的计算成本发生不可预测的陡增，其他的一些场景也会出现这种情况。

由于上述的这些原因，如今的应用程序在很大程度上都放弃了阴影体算法。然而，考虑到在 GPU 上访问数据的新方法正在不断发展，以及研究人员对于这些功能的巧妙再利用，阴影体算法可能会在未来的某一天重新被广泛使用。例如，Sintorn 等人[1648]对改进效率的阴影体算法进行了综述，并提出了他们自己的分层加速结构。

下一小节中我们会介绍一种新算法，它叫做阴影映射（shadow mapping），它具有更加可预测的计算开销，非常适合 GPU 执行；它也是许多应用程序中生成阴影的基础算法。

7.4 阴影贴图

1978 年，Williams [1888]提出了一种通用的、基于 z-buffer 的渲染器，它可以在任意物体上快速生成阴影。其核心想法是从光源的位置出发，使用 z-buffer 来渲染整个场景，然后再生成阴影效果。能够被光源“看见”的任何物体都会被照亮，光源“看不见”的物体则都处于阴影中。实际上在图像渲染的时候，我们最终只需要这个 z-buffer 即可，即我们只需要场景的深度信息；因此在这个特殊的场景渲染中，我们可以关闭光照、纹理等选项，也不用向颜色缓冲写入任何值。

在从光源视角渲染整个场景之后，z-buffer 中的每个像素现在代表了最靠近光源的物体深度值。我们将这个 z-buffer 中的内容叫做阴影贴图（shadow map），有时候也会称为阴影深度图（shadow depth map）或者阴影缓冲区（shadow buffer）等。为了使用这个阴影贴图来生成阴影效果，我们会从相机的位置来对场景进行第二次渲染。在渲染每个图元的时候，对于该图元所覆盖的每个像素位置，我们都会将其与阴影贴图进行深度比较：如果着色点比阴影贴图中对应位置到光源的距离更远，则说明该点位于阴影中，否则该点不在阴影中。该算法是利用纹理映射实现的，如图 7.10 所示。阴影映射是一种十分流行的算法，因为它的计算成本相对来说是可预测的。创建阴影贴图的开销，与需要渲染的图元数量大致呈线性关系，并且访问时间是常量。在光源和物体不发生移动的场景中（例如一些计算机辅助设计应用 CAD 中），我们可以只生成一次阴影贴图，并在每一帧中进行重复使用。

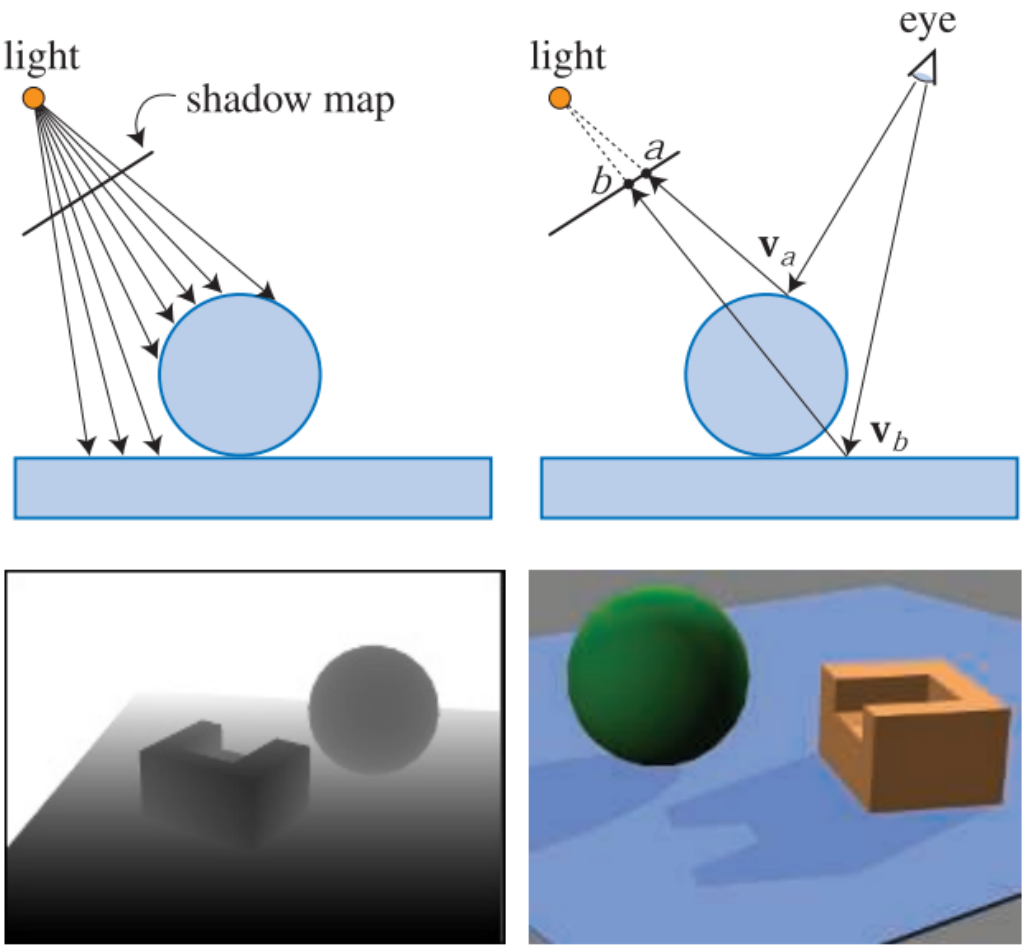


图 7.10：阴影映射。左上角：通过存储视图中的表面深度，来构建阴影贴图。右上角：眼睛正看向场景中的两个位置，其中在点 $\mathbf{v}_a$ 看到了球体，而这个点可以在阴影贴图的纹素 a 处找到，存储在纹素 a 处深度值大于点 $\mathbf{v}_a$ 到光源的距离，因此点 $\mathbf{v}_a$ 会被照亮。而对于矩形表面上的点 $\mathbf{v}_b$，其相对于光源的深度要大于存储在纹素 b 处的深度值，因此点 $\mathbf{v}_b$ 位于阴影中。左下角：光源视角下的场景深度图，颜色越白，代表深度值越大。右下角：使用阴影贴图进行渲染的场景。

当只生成单个 z-buffer 的时候，光源只能“看”向一个特定的方向，就像相机一样。对于一个遥远的方向光来说（例如太阳），这个光源的视野会非常大，对于相机能够看到的任何物体，该光源都可以让其产生阴影。这样的光源会使用正交投影来进行渲染，其视野需要在 x 方向上足够宽，在 y 方向上足够高，才能保证“看到”场景中的这些物体。局部光源也需要尽可能地进行类似的调整，如果局部光源距离 shadow caster（能够投射阴影的物体）足够远的话，那么一个视锥体就足以包含这些物体了；如果局部光源是一个聚光灯的话，那么它有一个与光照范围天然关联的视锥体，位于视锥体之外的一切物体都不会被照亮。

如果一个局部光源位于场景内部，并且被 shadow caster 所包围，一种典型的解决方案是使用一个六视图立方体（six-view cube），这类似于环境映射[865]中所使用

的立方体贴图（cube map），它被称为全向阴影贴图（omnidirectional shadow map）。全向阴影贴图的主要挑战在于，需要避免在两个独立阴影贴图接缝处出现瑕疵。King 和 Newhall [895]深入分析了这个问题并提供了解决方案，Gerasimov [525]则提供了一些实现上的细节。Forsyth [484, 486]提出了一种通用的、适用于全向光源的多视锥体分割方案，该方案可以在需要的方向上提供更高分辨率的阴影贴图。Crytek [1590, 1678, 1679]根据每个视图的投影视锥体在屏幕空间中的覆盖率，从而为点光源的六个视图分别设定阴影贴图分辨率，并将所有贴图存储在一个很大的纹理图集中。

并不是场景中的所有物体都需要被渲染到光源的视野中。首先，只有能够投射阴影的物体才需要进行渲染，我们将其称为 shadow caster（遮挡物，能够投射出阴影的物体）。例如：如果我们事先知道了地面只能够接收阴影，但是无法投射阴影，那么它就不需要被渲染到阴影贴图中。

根据定义，shadow caster 是指那些位于光源视锥体内部的物体，这个视锥体可以通过一些方法来进行扩张或者收缩，这使得我们可以安全地忽略掉场景中的一些 shadow caster [896, 1812]。想象一组可以被相机看见的 shadow caster，这组物体位于沿着光源观察方向的最大距离内，任何超出这个最大距离的物体，都无法在阴影接收物上投下阴影，无论这个接收物是可见的还是不可见的。同样的，对于一组可见的阴影接收物，其位置也可能会小于光源的原始视图边界（在 x 方向上和 y 方向上），如图 7.11 所示。另一种情况是，如果光源位于相机视锥体的范围内，那么位于相机视锥体外的任何物体，都无法在接收物上投下阴影。只对相关物体进行渲染，不仅仅可以节省渲染时间，还可以减少光源视锥体所需的大小，提高阴影贴图的有效分辨率，从而提高阴影贴图的质量。此外，让光源视锥体的近裁剪平面尽可能地远离光源，同时让远裁剪平面尽可能地靠近光源，对于提升阴影贴图的质量也有一定帮助，因为这样做可以提高 z-buffer 的有效精度[1792]（详见章节 4.7.2）。

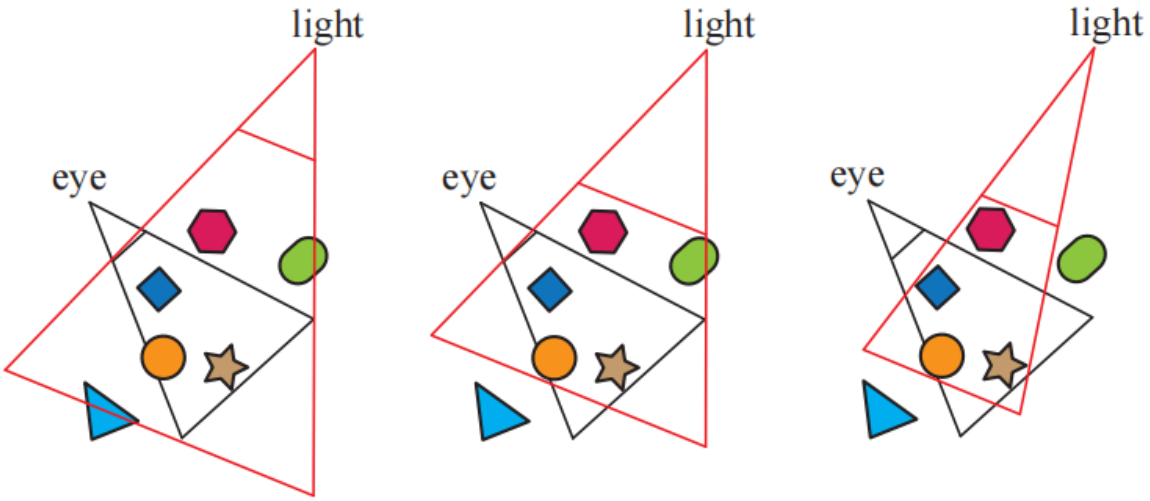


图 7.11：左侧：光源的视野内包含相机的截锥体。中间：对光源的远裁剪平面进行了调整，使得光源视锥体只包含相机可见的阴影接收物，即将作为 shadow caster 的蓝色三角剔除在外；同时还对近裁剪平面进行了调整，从而提高 z-buffer 的有效精度。右侧：光源视锥体的两侧进行收缩，只包含相机可见的阴影接收物，将绿色胶囊剔除在外。

阴影贴图的一个缺点是：阴影质量取决于阴影贴图的分辨率（单位为像素）以及 z-buffer 的数值精度。由于在进行深度比较的时候，需要对阴影贴图进行采样，因此该算法容易受到锯齿问题的影响，尤其是在物体之间的接触点附近。一个常见的问题是自阴影锯齿（self-shadow aliasing），即一个三角形错误地对自身投射阴影，这个现象通常会被称为“表面痤疮（surface acne）”或者“阴影痤疮（shadow acne）”。产生这个问题主要有两个原因：其中一个原因是处理器精度的数值限制；另一个原因是几何上的，由于受到阴影贴图分辨率的限制，因此一个点状样本的值会被用来代表一个小范围区域的深度。也就是说，为光源生成的样本，几乎永远不会与屏幕样本位于相同的位置（例如：像素通常会在屏幕样本的中心进行采样）。当光源的存储深度值与观察表面的深度值进行比较时，光源的值可能会略低于表面的值，从而导致自阴影现象的发生。图 7.12 展示了这些错误所带来的影响。

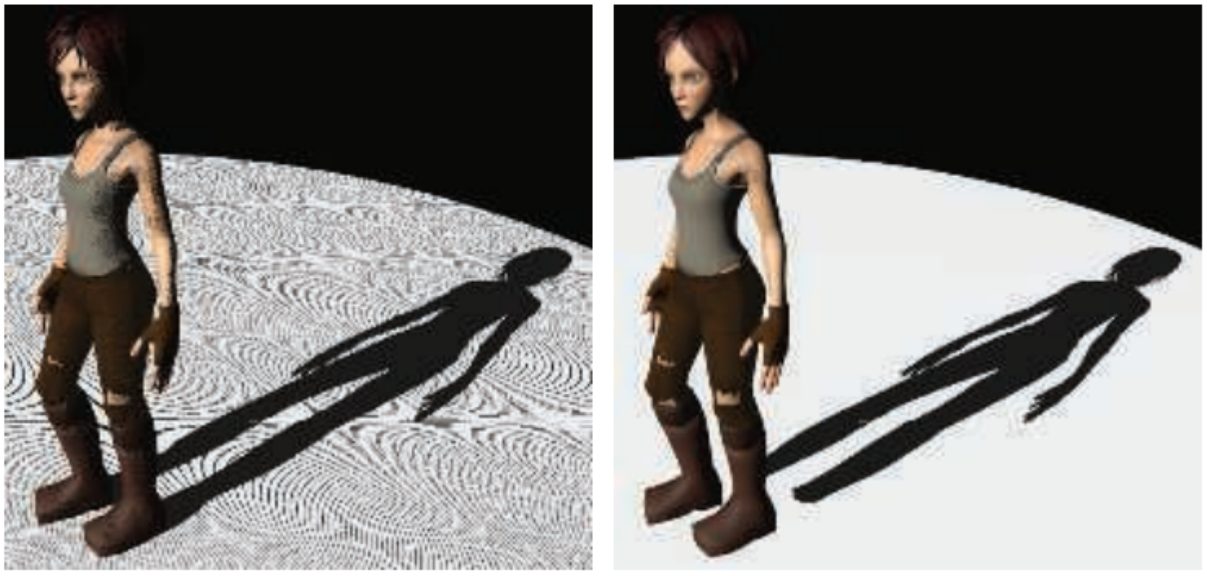


图 7.12：阴影映射的偏移瑕疵。左图中所设置的偏移量太小，因此发生了自阴影现象。右图中所设置的偏移量太大，导致鞋子没有投射出与自身相接触阴影。同时阴影贴图的分辨率也太低，使得阴影具有块状外观。

一种帮助避免（但并不总是能消除）各种阴影贴图瑕疵的常见方法是引入偏移量因子。当对阴影贴图中检索到的深度，与待测位置的深度进行比较时，可以从接收物的深度中减去一个很小的偏移量，如图 7.13 所示。这个偏移量可以是一个恒定的值[1022]，但是当接收物并没有完全面对光源的时候（表面法线与光线方向之间的夹角

很大），这样做可能会失败。一种更加有效的方法是，使用接收物相对于光源的角度，来对偏移量进行修正，从而避免这个问题，即表面相对于光源的倾斜角度越大，所应用的偏移量也就越大。这种偏移量称为斜率修正的偏移（slope scale bias）。这两种偏移量都可以通过使用相应的命令（例如 OpenGL 的 $\mathrm{glPolygonOffset}$），来让每个多边形稍稍远离光源。请注意，当表面正对光源的时候，并不会因斜率修正偏移而向后移动，即此时的斜率修正偏移为 0。因此，为了避免可能的精度误差，我们会将斜率修正偏移与一个恒定偏移结合在一起使用。斜率修正偏移也经常会被 clamp 到某些最大值，因为当表面和光源呈掠射夹角时（接近 90 度），此时的正切值会很大。

图 7.13：阴影偏移。表面正上方有一个光源，这个表面会被渲染到一个阴影贴图中，图中的灰色竖线代表了阴影贴图的像素中心，阴影贴图会在 × 位置处记录遮挡深度。我们想要知道，表面上的这三点是否会被光源照亮，最近的阴影贴图深度值会使用相同颜色的 × 进行表示。左图中，如果没有添加偏移，那么蓝色和橙色样本将会被错误地认为处于阴影中，因为与对应的阴影贴图深度相比，它们距离光源更远。在中间，从每个样本中都减去了恒定的深度偏移，使得每个样本都更加靠近光源，但是此时蓝色样本仍然会被认为处于阴影中，因为它距离光源还是要更远一点。右图中，在构建阴影贴图的时候，会根据每个多边形的斜率，对其偏移量进行修正，然后再让其按照偏移量稍稍靠近光源（样本），这样所有的样本深度都要比阴影贴图中的深度更加靠近光源，因此它们都会被光源照亮。

Holbert [759, 760]引入了法线补偿偏移（normal offset bias），它将接收物的世界空间位置，沿着其表面法线的方向稍稍移动一点，移动的距离与光源方向和表面法线之间夹角的正弦值成正比，如图 7.24 所示。这个操作不仅改变了样本的深度值，还改变了它在阴影贴图上进行深度测试的 xy 坐标。当光线与表面之间的角度更加接近掠射角度时，这个偏移量就会增加，因为此时我们希望样本距离表面足够远，从而避免自阴影现象。这种方法可以理解为将样本移动到阴影接收物上方的“虚拟表面”上。这个偏移量是一个世界空间下的距离，因此 Pettineo [1403]建议可以根据阴影贴图的深度范围，来对这个偏移量进行缩放。Pesce [1391]提出了沿相机观察方向进行偏

移的想法，这也可以通过调整阴影贴图中的坐标来实现。其他的一些偏移方法将在章节 7.5 中进行讨论，因为这些阴影方法还需要对几个相邻样本进行深度测试。

过大的偏移量会导致所谓的漏光（light leak）或者 Peter Panning 问题，即物体看起来像是悬浮在表面上方一样。这种瑕疵的出现，是因为物体接触点下方的区域（例如脚下的地面），被向前偏移得太多，因此并没有接收到阴影。

避免自阴影问题的一种方法是，只将物体背面渲染到阴影贴图中，这种方法被称为第二深度阴影映射（second-depth shadow mapping）[1845]，它在许多情况下都表现良好，尤其是对于那些无法手动调整偏移量的渲染系统。但是当物体是双面渲染的、非常薄的、或者是相互接触时，这种方法就会出现问题。如果一个模型是双面渲染的，也就是说其网格两侧都是可见的（例如：一个棕榈叶或者一张纸），那么此时就可能会发生自阴影现象，因为背面和正面位于相同的位置。同样地，如果不进行偏移，那么在模型轮廓边缘处，或者很薄的物体附近也可能会出现问题，因为在这些地方，模型的背面和正面靠得很近。此时进行一点点偏移，可以帮助避免表面阴影痤疮，但是这样会更容易漏光，因为在接收物和遮挡物的接触点上，接收物并没有与遮挡物的背面相分离，如图 7.14 所示。具体选择哪种方案可以视情况而定，例如：Sousa 等人[1679]发现，使用正面深度来生成太阳的阴影，使用背面深度来生成室内灯光的阴影，这种方案对于他们的应用而言效果最好。

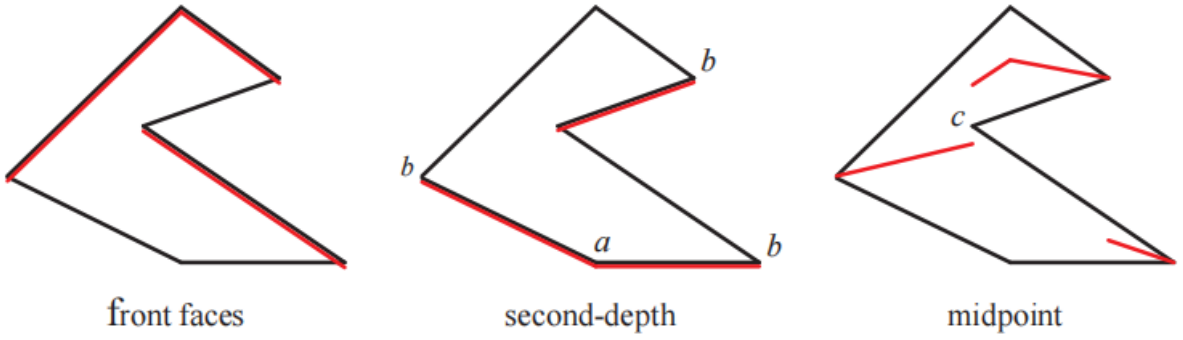


图 7.14：位于正上方的光源所生成的阴影贴图。左图：使用红色标记了面对光源的表面，它们将被记录到阴影贴图中。这些表面可能会错误地对自身投射阴影（即“阴影痤疮”），所以需要向远离光源的方向进行偏移。在中间：只有背面的三角形表面会被渲染到阴影贴图中。如果将遮挡物向下（远离光源）进行偏移，可能会让光线泄漏到靠近位置 a 的地面上；如果将遮挡物向上（靠近光源）进行偏移，可能会导致在轮廓边界 b 附近，被错误地被认为处于阴影中，但是实际上这些地方应当会被照亮。右图：在阴影贴图上每个位置上，会在最近的正面三角形与背面三角形的中点处，形成一个中间表面。但是这样可能会在点 c 附近发生漏光（这也可能发生在第二深度阴影贴映射中），因为最近的阴影贴图样本，可能位于该位置（点 c）左侧的中间表面上，该点会更加靠近光源。

需要注意的是，对于阴影映射而言，能够投射阴影的物体必须是“水密的（watertight）”（或者被称为流形 manifold、封闭 closed、或者固体 solid，详见章节 16.3.3），或者换句话说，它必须能够将自己的正面和背面都渲染到阴影贴图中，否则这个物体可能无法投射一个完整的阴影。Woo [1900]提出了一种通用方法，该方法试图在使用正面深度或者背面深度进行阴影处理之间，找到一个折衷的方法。其核心想法是，将固体对象渲染到阴影贴图中，并追踪两个最靠近光源的表面，这个过程可以通过深度剥离（depth peeling），或者其他与透明相关的技术来实现。两个物体之间的平均深度会形成一个中间层，我们使用这个中间层的深度来作为阴影贴图，这种方法有时会称为双重阴影贴图（dual shadow map）[1865]。如果物体足够厚的话，那么可以最小化自阴影和漏光所带来的瑕疵。Bavoil 等人[116]讨论了处理潜在瑕疵的方法，以及其他的实现细节。这种方法的主要缺点在于，它使用了两个阴影贴图，这会带来额外的开销（计算开销和存储开销）。Myers [1253]讨论了一种由艺术家来控制遮挡物和接收物之间深度层的方案。

随着相机在场景中的移动，光源的视野范围也会随着 shadow caster 的变化而变化，这种变化反过来又会导致阴影在帧与帧之间略有不同。这是因为光源的阴影贴图从光源处采样了一组方向，这些采样方向与上一帧中的采样方向略有不同。对于方向光而言，对应的解决方案是，对每个后续生成的阴影贴图进行限制，使得它们在世界空间中保持相同的相对纹素位置[927, 1227, 1792, 1810]。也就是说，我们可以将阴影贴图视为在整个世界中，强加上一个二维网格的参考网格，每个网格单元都代表了阴影贴图上的一个像素样本。当我们在场景中移动的时候，阴影贴图会为这些相同的网格单元生成不同的样本集合。换句话说，光源的视图投影会被限制到这个网格中，从而保持帧与帧之间的一致性。

7.4.1 分辨率增强

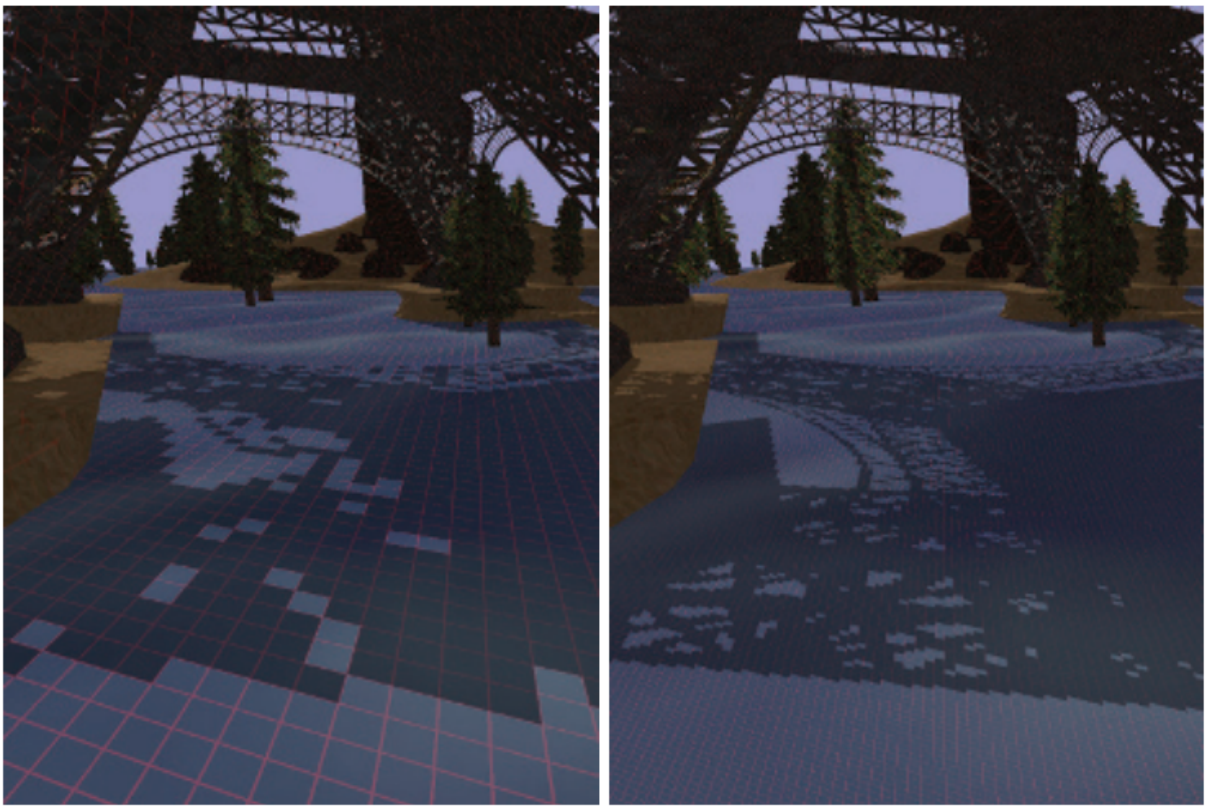


图 7.15：左边的图像是使用标准的阴影映射生成的；右边的图像则使用了 LiSPSM。上图还展示了每个阴影贴图纹素在地面上的投影。这两个阴影贴图具有相同的分辨率，不同之处在于，LiSPSM 对光源矩阵进行了修改，在更靠近观察者的地方提供了更高的采样率。

与纹理的使用方式相类似，在理想情况下，我们希望阴影贴图中的一个纹素，大约能够覆盖屏幕上的一个像素。如果我们有一个与相机位置相同的光源，那么此时阴影贴图中的纹素可以和屏幕空间上像素完美地一一对应（并且我们看不见任何阴影，因为相机所观察到的物体，都会被这个光源照亮）。一旦光源的方向有所改变，那么阴影贴图中一个纹素所占据的屏幕像素比例就会发生改变，此时就可能会产生一些瑕疵和锯齿。如图 7.15 所示，图中的阴影是块状的，并且轮廓也很不清晰，因为阴影贴图中的每个纹素都关联了大量的前景像素，这种不匹配的现象被称为透视走样（perspective aliasing）。当一个表面几乎与光线平行，并且它刚好面向观察者时，单个阴影贴图的纹素也会覆盖大量像素，这个问题被称为投影走样（projective aliasing）[1792]，如图 7.16 所示。提高阴影贴图的分辨率可以减少块状阴影的出现，但是需要额外的内存开销和处理开销。

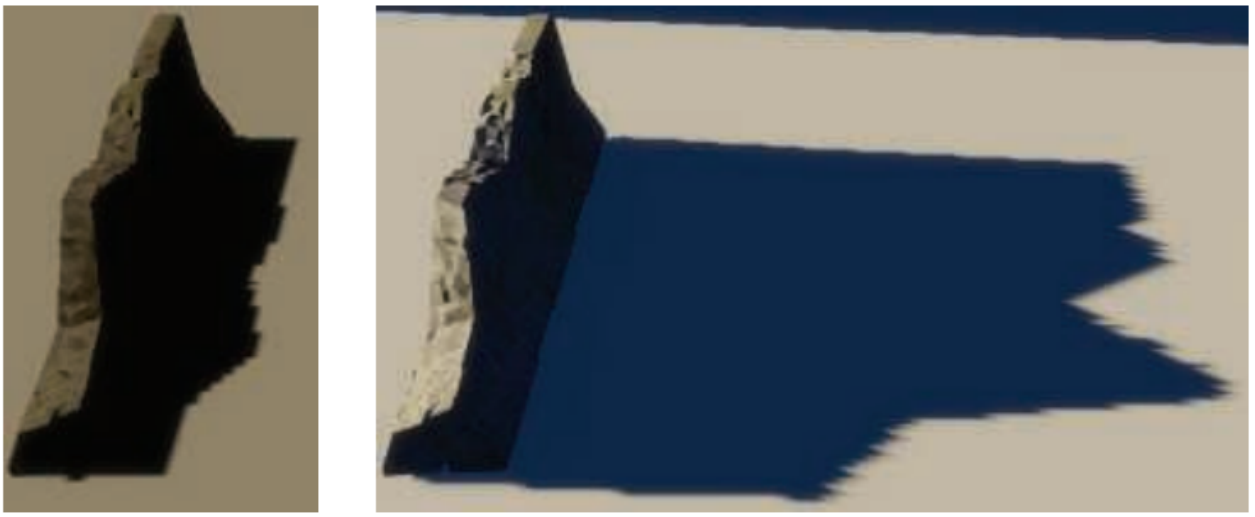


图 7.16：左图中的光源几乎位于正上方，阴影的边缘有点粗糙，因为与相机的视图分辨率相比，阴影贴图的分辨率比较低。右图中的光线以掠射角度照射到物体上，每个阴影贴图中的纹素，在水平方向上都会覆盖相当多的屏幕像素，所以会产生更加锯齿状的阴影边缘。

还可以使用另一种方法来创建光源的采样模式，使其更加接近相机的采样模式，这是通过改变场景向光源的投影方式来实现的。通常我们认为观察区域是对称的，观察方向位于视锥体的中心。然而观察方向仅仅决定了视图平面的朝向，并没有决定哪些像素会被采样。定义视锥体的窗口可以在这个平面上进行移动、倾斜或者旋转，从而创建一个四边形，提供一种完全不同的、从世界空间到观察空间的映射方式。这个四边形仍然会以固定的间隔进行采样，这是由在 GPU 中使用的线性变换矩阵的本质特点所决定的。但是我们可以通过改变光源的观察方向以及观察窗口的边界，来改变不同位置上的采样率，如图 7.17 所示。

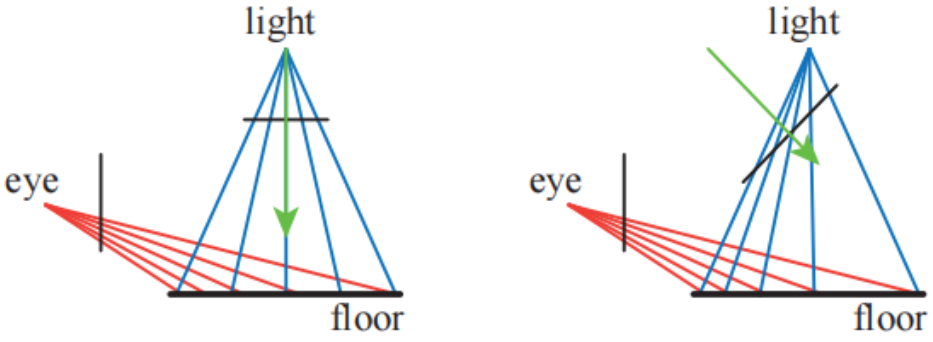


图 7.17：对于位于正上方的光源，左图中地面上的采样率和眼睛的采样率不匹配。在右图中，通过改变光源的观察方向和投影窗口，使得采样率发生偏移，在左侧靠近眼睛的地方，拥有更高密度的纹素。

将光源的观察视图映射到人眼的观察视图，大概有 22 个自由度[896]，对这个解空间的探索引出了好几种不同的算法，这些算法试图将光源的采样率与相机的采样率进行

更好地匹配。这些方法包括透视阴影贴图（perspective shadow map，PSM）[1691]、梯形阴影贴图（trapezoidal shadow maps，TSM）[1132]和光源空间透视阴影贴图（light space perspective shadow map，LiSPSM）[1893, 1895]，具体示例详见图 7.15 和图 7.26。这类技术被统称为透视变形（perspective warping）方法。

这些矩阵扭曲（matrix-warping）算法的一个优点是，除了对光源的投影矩阵进行修改之外，不需要进行其他额外的工作。每种方法都有自己的优缺点[484]，因为每种方法都可以在某些特定的几何情况和照明情况下，来帮助匹配采样率，但是在其他场景情况下的采样率会发生恶化。Lloyd 等人[1062, 1063]分析了 PSM、TSM 和 LiSPSM 之间的等价性，并对这些方法的采样和锯齿问题进行了很好的概述。当光源方向垂直于观察方向时（例如头顶上的光源），这些方案的效果最好，因为在这种情况下，在对光源的透视变换进行修正之后，可以在靠近眼睛的地方放置更多的样本。

当光源位于相机前方并且指向相机的时候，这些矩阵扭曲技术就会失效。这种情况被称为视锥决斗（dueling frusta），因为光源的视锥体和相机的视锥体相对；或者更加通俗的说法是“车灯下的鹿（deer in the headlights）”（译者注：这是一句英语俗语，意思是慌张、呆若木鸡）。在这种情况下，在靠近相机的位置处依然需要更多的阴影贴图样本，但是这种线性扭曲的方法只会让情况变得更加糟糕[1555]。上述这些问题以及其他的一些问题，例如阴影质量的突变[430]，相机运动过程中阴影质量的不稳定性[484, 1227]，使得这些方法不再受大家欢迎。

在靠近观察者的位置上添加更多的样本，这是一个很好的想法，基于这种思路，有一些算法会为给定的视图生成多张阴影贴图。当 Carmack 在 2004 年的 Quakecon 大会上发表相关主题演讲时，这个思路的第一次提出就产生了显著影响。Blow 独立实现了这样的一个系统[174]，其核心想法很简单：生成一组固定数量的阴影贴图（它们的分辨率可能是不同的）来覆盖场景中的不同区域。在 Blow 的方案中，生成了四张阴影贴图，它们嵌套在了观察者周围；通过这种方式，位于观察者附近的物体就可以获得更高分辨率的阴影贴图，而位于远处的物体则会获得较低分辨率的阴影贴图。Forsyth [483, 486]提出了一个类似的想法，即为不同集合的可见物体，生成不同的阴影贴图；在这种方法中，不需要对跨越两个阴影贴图边界的物体过渡问题进行处理，因为每个物体有且只有一个与其关联的阴影贴图。Flagship 工作室开发了一个系统，它将这两种理念相融合：第一个阴影贴图用于观察者附近的动态物体，第二个阴影贴图用于观察者附近的静态对象，第三个阴影贴图用于整个场景中的静态物体。每帧只需要重新生成第一个阴影贴图即可，另外两个阴影贴图只需要生成一次，因为它们所包含的光源和几何物体都是静态的。虽然上述这些渲染系统现在都已经很老了，

但是为不同物体和不同情况生成多个阴影贴图的思想，从那时起就成为了阴影映射算法的共同主题，其中，这些阴影贴图有些是预计算生成的，有些则是动态生成。

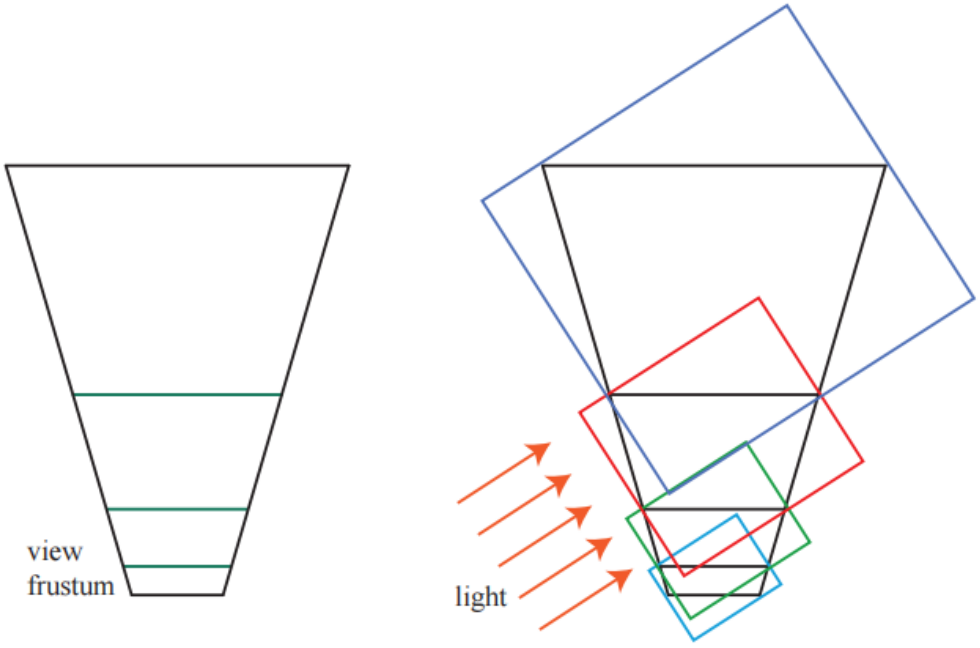


图 7.18：左图中，相机的视锥体被划分成了四个区域。右图中，为每个划分后的视锥体区域创建了对应的包围盒，图中的方向光会生成四个阴影贴图，每个包围盒决定了该阴影贴图的渲染范围。[430]

2006 年 Engel [430]、Lloyd 等人[1062, 1063]、Zhang 等人[1962, 1963]，根据这个相同的基本思想，独立进行了各自的研究；这个想法是在观察方向的垂直方向上，将视锥体划分成若干个区域，如图 7.18 所示。

> Tadamura 等人[1735]早在七年前就提出了这个想法，但是直到其他研究人员发现了这个想法的实用性之后，它才真正产生了巨大影响。

随着深度（距离相机的距离）的增加，每个视锥体区域的深度范围大约是前一个区域的 2 到 3 倍[430, 1962]。对于每个划分后的视锥体区域，光源都可以生成一个紧密包裹该区域的视锥体，然后根据这个视锥体来生成阴影贴图。通过使用纹理图集（texture atlas）或者纹理数组（texture array），可以将不同区域的阴影贴图打包成一个较大的纹理对象，从而减少缓存访问所带来的延迟。图 7.19 展示了这种方法所带来的质量提升。Engel 将这种算法的命名为级联阴影贴图（cascade shadow maps，CSM），Zhang 则将其称为平行分割阴影贴图（parallel-split shadow maps），这两个名词都在相关文献中都出现过，它们实际上是完全等价的[1964]，但是一般 CSM 会更加常用一些 。

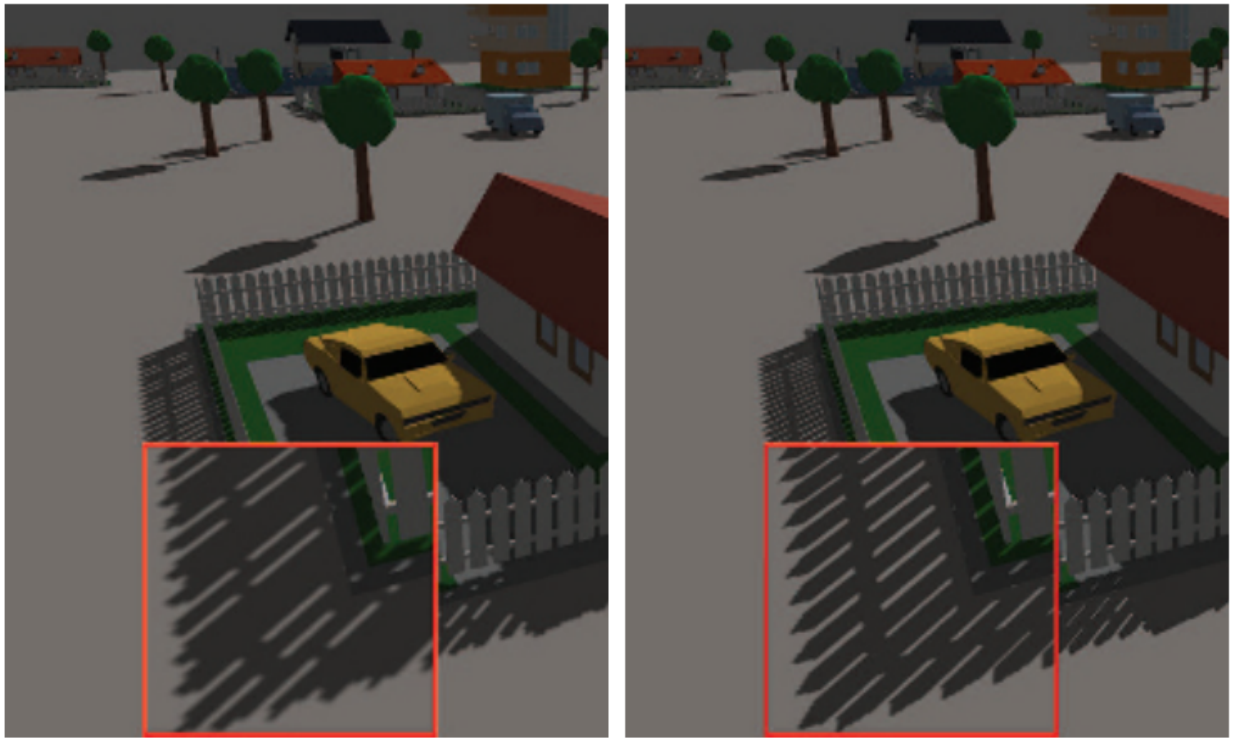


图 7.19：左图：场景的可见区域很大，导致单张阴影贴图使用了 2048×2048 的分辨率，从而避免透视走样（锯齿）。右图：在观察方向上使用了四个 1024×1024 的阴影贴图，大大提高了阴影的质量[1963]。上图红框中展示了围栏前角的放大结果，可以看到右侧的阴影质量更高。

这种算法的实现起来十分简单，它能够覆盖较大范围的场景区域，并且可以得到合理的阴影结果，具有较强的健壮性。“视锥决斗（dueling frusta）”的问题也可以通过在靠近相机的位置上使用更高的采样率来解决，并且这种算法没有严重的最坏情况问题，也就是说，它在几乎所有的光照条件和场景条件下都表现良好。由于上述的这些优点，级联阴影映射被广泛应用在各种图形程序中。

图 7.20：阴影级联的可视化。图中紫色、绿色、黄色和红色分别代表了从近到远的四个级

联阴影。

虽然可以使用透视变形，来将更多的样本打包到单个阴影贴图的细分区域中，但是通用的做法是为每个级联（cascade）使用单独的阴影贴图[1783]。如图 7.18 和图 7.20 所示，从相机的视角来看，每个阴影贴图所覆盖的区域是不同的。较小的观察空间对应了更近的阴影贴图，它需要提供更多的样本。在不同的阴影贴图之间划分 z 深度的范围，这可能是一件既简单又复杂的任务，这个任务被称为 z 划分（z-partitioning）[412, 991, 1791]。其中一种方法是使用对数划分方法[1062]，即对于每个级联阴影贴图而言，其远裁剪平面与近裁剪平面的距离之比是相同的，这个比例如下：

$$r = \sqrt[c]{\frac{f}{n}} \tag{7.5}$$

其中 n 是整个场景视锥体的近裁剪平面， f 是整个场景视锥体的远裁剪平面； c 是阴影贴图的数量， r 是最终生成的比例。例如：如果场景中最靠近相机的物体深度为 1 米，最远离相机的物体深度为 1000 米，一共划分为三个阴影贴图，那么这个比例是 $r = \sqrt[3]{1000/1} = 10$ 。也就是说，最近区域所对应的远近裁剪平面的深度分别是 1 和 10，下一个区域是 10 和 100，最后是 100 和 1000 米。初始的近裁剪平面深度，对这种划分方法有着很大的影响，如果近裁剪平面的深度仅为 0.1 米，那么 10000 的立方根将会达到 21.54，这是一个相当高的比例，最终的深度划分为：0.1，2.154，46.42，1000。这意味着最后一个阴影贴图需要覆盖更大范围的区域，从而降低了精度和阴影的质量。在实际应用中，这种划分方法可以为近裁剪平面附近的区域提供了更大的阴影贴图分辨率（采样率），但是如果该区域中没有物体的话，那么这个更大分辨率的阴影贴图就会被浪费。避免这种问题的一种方法是，将划分距离设置为对数分布（logarithmic distribution）和等距分布（equidistant distribution）的加权混合[1962, 1963]；如果我们能够确定场景的紧密视图包围盒的话就更好了。

这个问题的困难之处在于如何设置近裁剪平面的深度。如果将近裁剪平面设置得距离相机太远，那么可能会错误地剔除某些靠近相机的物体，这会产生非常严重的视觉瑕疵。对于过场动画而言，艺术家可以提前精确设定这个值[1590]，但是对于一般的交互式场景而言，这个问题就变得十分具有挑战性了。Lauritzen 等人[991, 1403]提出了一种叫做样本分布阴影贴图（sample distribution shadow maps，SDSM）的方法，它使用了前一帧中的深度划分值，然后通过两种方法中的其中一种，来确定一个更好的划分方式。

第一种方法是遍历深度（深度缓冲）的最小值和最大值，并使用它们来设置远近裁剪平面的深度。这是使用 GPU 上的 reduce 操作来完成的，其中计算着色器或者其他的着色器，会对一系列越来越小的缓冲区进行分析，输出的缓冲区又会被作为输入，重新送入这个着色器中，直到最终留下一个 1×1 大小的缓冲区。通常情况下，会根据场景中物体的移动速度，对这些值进行一点调整。如果不采取纠正措施的话，从屏幕边缘进入画面的物体，可能会对当前帧产生一些一些影响和问题，尽管这个问题在下一帧中就会被很快纠正。

第二种方法同样会对深度缓冲中的深度值进行分析，并生成一个直方图，其中记录了深度值的分布。这个直方图除了用于寻找最合适（紧密包裹场景中需要渲染的物体）的远近裁剪平面之外，还应当存在一些不包含任何物体和深度的间隙。通常会在实际包含物体的深度区间中进行划分，从而为级联阴影贴图提供更多的深度精度。

在实践中通常会使用第一种方法，它的速度很快（通常在每帧 1 毫秒内），并且能够生成较好的结果，因此它在很多应用中都被采纳[1405, 1811]，如图 7.21 所示。

图 7.21：深度包围盒对阴影质量的影响。左图中，没有使用特殊处理来调整远近裁剪平面的深度。在右边，使用了 SDSM 方法来寻找更加紧密的包围盒。注意上图中左侧边缘附近的窗框、二楼花盆下方的区域、以及一楼的窗户，松散的视图包围盒会导致采样不足，从而产生了瑕疵和锯齿。指数划分的阴影贴图可以用于渲染这些特定的图像，但是提高深度精度的思路对所有的阴影贴图技术而言都是有效的。

级联阴影贴图与使用单个阴影贴图具有一些相同的问题，由于光线样本会在帧与帧之间发生变化，从而产生闪烁瑕疵，并且当物体在级联之间进行移动的时候，这种问题

可能会变得更加严重。有很多方法可以用来在世界空间中保持稳定的采样点，每种方法都有各自的优点[41, 865, 1381, 1403, 1678, 1679, 1810]。当一个物体跨域两个级联阴影贴图的公共边界时，阴影的质量会发生突变，一种解决方法是让划分的深度区域稍微重叠，位于重叠区域的样本会收集相邻阴影贴图的结果，并对其进行混合[1791]；或者也可以使用抖动法（dithering），来在该区域内采集单个样本[1381]。

由于级联阴影贴图技术十分流行，人们投入了相当大的努力来进一步提高算法的效率和质量[1791, 1964]。如果阴影贴图的视锥体内没有发生任何变化，那么就不需要重新计算该阴影贴图。对于每个光源而言，可以通过预先计算，找到光源视锥体中的哪些物体是可见的，其中又有哪些物体会产生阴影，从而构建一个 shadow caster 的列表[1405]。由于人眼很难精确判断生成的阴影是否是物理正确的，因此我们可以在级联阴影映射和其他阴影算法中，采取一切取巧的方法（trick）。其中一种方法是使用低层级细节（LOD）的模型来作为实际投射阴影的代理模型[652, 1812]；另一种方法是移除那些很小的遮挡物[1381, 1811]。位于较远处的阴影也不需要每帧都进行更新，因为在理论上而言，这类阴影对于视觉感知的重要性很低；但是当一些很大的物体在远处移动时，这个想法可能会产生一些瑕疵，因此需要小心使用[865, 1389, 1391, 1678, 1679]。Day [329]提出了一种将远处阴影贴图在帧与帧之间进行重复使用的想法，这个想法的依据是，静态阴影贴图中的大部分内容，在帧与帧之间都是相同的，可以重复使用，只有阴影的边缘部分可能会发生变化，因此这部分需要重新渲染。诸如《毁灭战士（2016）》这样的游戏，维护了一个巨大的阴影贴图图集，只在物体发生移动的地方重新生成阴影贴图即可[294]。而对于更远的级联阴影贴图，可以让其完全忽略动态物体，因为这些物体的阴影对于场景感知的贡献很少。在某些场景中，可以使用一个高分辨率的静态阴影贴图，来代替这些位于较远位置的级联阴影贴图，这样可以显著减少工作量[415, 1590]。对于一些大场景而言，可能会有一个十分巨大的静态阴影贴图，在这种情况下，可以使用一个稀疏纹理系统（sparse texture system）（章节 19.10.1）[241, 625, 1253]。级联阴影贴图还可以与烘焙的光照贴图结合使用，或者是与其他更适合某些特定场景的阴影技术结合使用[652]。Valient [1811]的演讲是十分值得关注的，他介绍了在各种电子游戏中所使用的自定义阴影系统，以及相关的技术。在章节 11.5.1 中详细讨论了预计算的光照和阴影算法。

生成若干个独立的阴影贴图意味着要为每个贴图都渲染一组几何图形。有个想法是在单个 pass 中，将遮挡物渲染到一组阴影贴图中，有许多提高效率的方法都是建立在这个想法上的。可以使用几何着色器来复制物体数据，并将其发送到多个视图中[41]；实例化的几何着色器允许一个物体输出到多达 32 个深度纹理中[1456]。多视口扩展（multiple-viewport extensions）可以执行诸如将物体渲染到特定纹理数组切片中之类的操作[41, 154, 530]，章节 21.3.1 在虚拟现实的背景下，对其进行了更加

详细地讨论。视图共享技术的一个可能的缺点是，用于生成阴影贴图的所有遮挡物（无论这些阴影贴图是否会用到它们）都必须要发送到管线中执行，而不是只发送与当前阴影贴图有关的遮挡物[1791, 1810]。

现实中的我们正处于来自世界各处数十亿光源的阴影中，其中只有少数光源会照射到我们的身上。而在实时渲染中，如果所有的光源都处于启用状态，那么一些具有多个光源的大型场景可能直接会被计算量所淹没。如果视锥体中的一个区域是相机看不见的，那么就不需要对该区域所接收到的阴影进行计算[625, 1137]。Bittner 等人[152]使用相机的遮挡剔除（occlusion culling）（章节 19.7），来找到所有可见的阴影接收物，然后再从光源的角度，将所有潜在的阴影接收物渲染到一个模板缓冲区中，这个蒙版记录了光源可以看到的全部阴影接收物。为了生成阴影贴图，他们从光源的视角出发，使用遮挡剔除来渲染物体，并使用之前生成的蒙版来剔除那些没有阴影接收物的物体。相机所使用的各种的剔除技术也同样适用于这里的光源。由于 irradiance 会随着距离的平方而衰减，因此通常会在一定的阈值距离后将光源剔除。例如：章节 19.5 中所介绍的入口剔除（portal culling）技术，可以发现是哪些光源会对单元格产生了影响。这是一个相当活跃的研究领域，因为它所带来的性能提升是相当客观的[1330, 1604]。

7.5 PCF

对阴影贴图技术进行简单的扩展，我们就可以获得质量不错的伪软阴影效果。当单个光源样本覆盖了许多屏幕像素的时候，会导致阴影看起来呈现块状，本小节中所介绍的方法，还可以帮助改善阴影贴图的分辨率问题。这个方法类似于章节 6.2.1 中所介绍的纹理放大技术，我们在对阴影贴图进行检索的时候，并不是简单地从阴影贴图中取出对应的单个样本，而是会检索 4 个最近的样本。不同的是，这里的所提到的方法并不会在这些深度值之间进行插值，而是先将它们与表面深度进行比较，然后在对比较的结果进行插值。也就是说，这四个纹理样本的深度会先与表面深度进行比较，然后确定每个样本点到底是位于光照区域还是阴影区域。我们使用 0 来代表位于阴影区域，1 来代表位于光照区域，然后再对这个 0–1 结果进行双线性插值，从而计算光源对于该表面位置的实际贡献有多大。这种过滤的结果会产生人为的软阴影，这些半影效果的变化取决于阴影贴图的分辨率、相机的位置以及其他的一些因素，例如：阴影贴图使用更高的分辨率，会使得阴影边缘变得更加柔和。尽管这种软阴影并非是物理正确的，但是有一点半影和平滑的效果，总比没有要好得多。

这种从阴影贴图中检索多个样本，并将结果进行混合的想法被称为百分比接近滤波（Percentage-Closer Filtering，PCF）[1475]。我们已经知道，面光源可以产生柔

和的软阴影效果，到达表面上某个位置的光线数量，与该位置所能看见的光源面积大小成比例。而 PCF 算法试图通过反转这个过程，来对一个精确光源或者方向光的软阴影进行近似。PCF 并没有从一个表面位置上来计算光源可见区域的面积，而是从靠近该位置的一组表面位置上，来计算精确光源的可见性，如图 7.22 所示。PCF 的名称“percentage-closer filtering”就表明了算法的最终目标，即找到光源可见样本所占的百分比。我们使用这个百分比来近似照射到表面位置的光线数量。

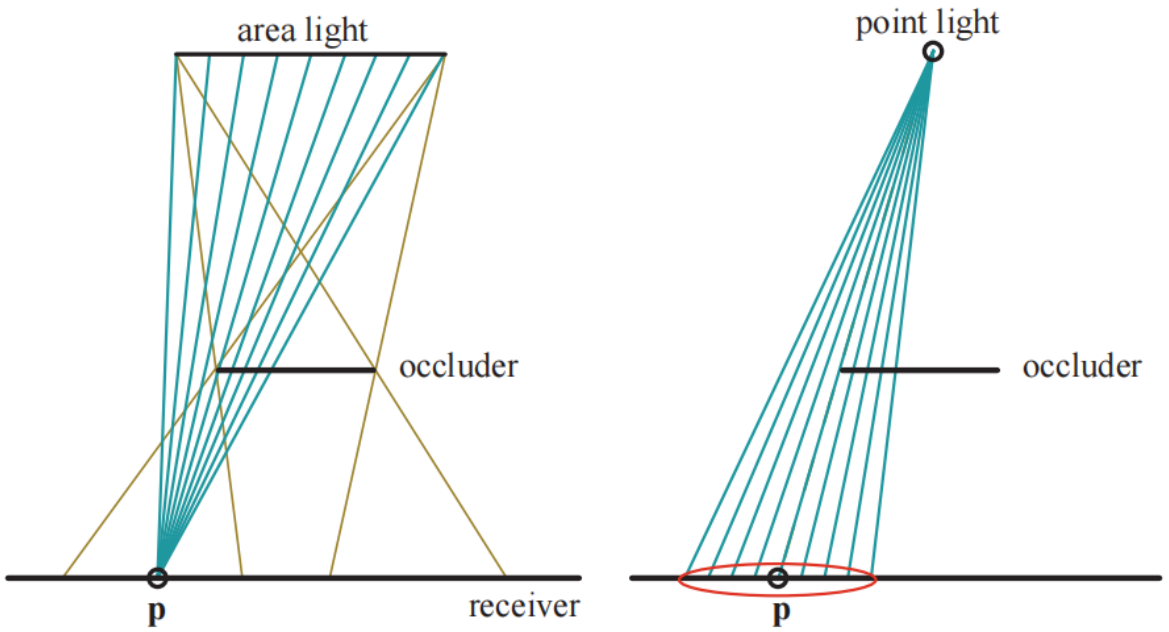


图 7.22：左图中，来自面光源的棕色线代表了半影的范围。对于阴影接收物上的点 $\mathbf{p}$，可以对面光源表面进行采样，并通过计算哪些采样点没有被遮挡物遮挡，从而获得该点所接收到的光线数量。右图中的点光源并不会产生半影现象。PCF 通过反转左图中过程，来对面光源效果进行近似：在给定的位置上，对阴影贴图附近的可比较区域进行采样，从而计算有多少样本会被照亮（百分比）。图中红色椭圆代表了在阴影贴图上进行采样的区域，在理想情况下，这个圆形区域的半径，与接收物和遮挡物之间的距离成正比。

在 PCF 算法中，这些计算光源可见性的样本位置在原始位置附近生成的，它们都具有大致相同的深度值，但是对应了阴影贴图中不同位置的纹素。会在这些位置上计算光源的可见性，然后再对生成的布尔结果（被光源照亮或者不会被光源照亮）进行混合，从而获得软阴影的效果。需要注意的是，这个过程并不是物理正确的：我们并没有直接对光源进行采样，而是对表面本身进行采样，再计算可见性百分比。由于表面与遮挡物之间的距离并不会对阴影结果产生影响，因此所生成的软阴影都具有尺寸相似的半影区域。尽管如此，这种方法在大多数情况下都可以提供一个合理的近似效果。

一旦确定了采样区域的宽度，下一个重要的问题是，如何以避免锯齿瑕疵的方式来进行采样。如何对附近的阴影贴图位置进行采样和过滤，有很多可以调整的参数选项，例如：采样区域的宽度、采集的样本数量、采样的模式以及如何对结果进行加权等。在图形 API 支持功能较少的情况下，可以采用一种类似于双线性插值的特殊纹理采样模式，通过访问相邻的 4 个位置来加速采样过程。在获得样本值之后，我们首先会将各个样本值与一个给定的值进行比较，计算通过测试的比例，然后再对结果进行混合[175]。然而，以一个规则的网格模式来对最近邻居纹素进行采样，会产生肉眼可见的锯齿和瑕疵；可以使用一个联合双边滤波，来对结果进行模糊处理，同时将物体的边缘信息作为联合双边滤波的权重，这样可以在提高阴影质量的同时，避免阴影泄漏到其他表面上[1343]。有关这种过滤技术的更多信息，详见章节 12.1.1。

DirectX 10 为 PCF 引入了单指令双线性滤波的支持，从而给出了一个更加平滑的结果[53, 412, 1709, 1790]。与上文提到的最近邻居采样方法相比，它提供了相当大的视觉改进，但是规则采样所带来的锯齿瑕疵仍然无法解决。为了最小化规则采样所带来的问题，其中一种解决方案是使用一个预先计算好的泊松分布模式来对区域进行采样，如图 7.23 所示；泊松分布会将样本分散开来，使它们既不相互靠近，也会不处于规则模式。我们知道，无论使用什么样的分布来进行采样，对每个像素使用相同的采样位置，都会产生这种规则的采样模式[288]。因此我们需要给每个像素上的采样位置添加一点随机性，通过将采样位置绕像素中心进行随机旋转，从而避免产生规则采样所带来的瑕疵，但是代价是会产生噪声问题。Castano 指出[235]，泊松采样所产生的噪声十分平滑，并且具有一定的特殊风格，因此在画面上会显得特别明显；他在双线性采样的基础上，提出了一种高效的高斯加权采样方法。

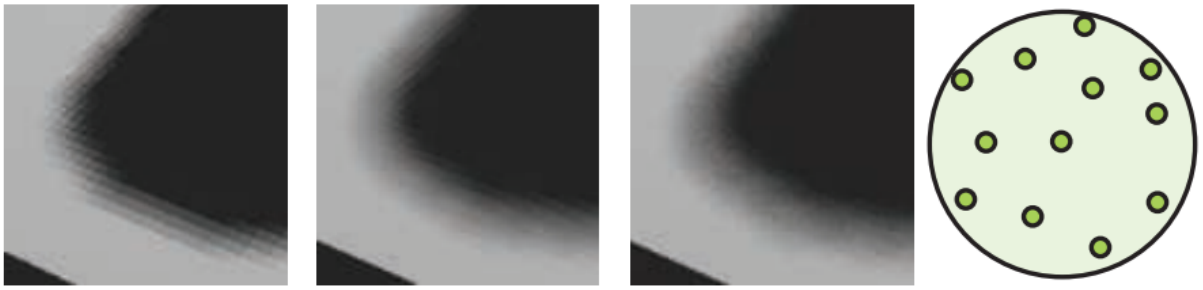


图 7.23：最左边的图片展示了 4×4 网格模式下，使用最近邻居采样的 PCF 结果。最右边的图片展示了圆盘上包含 12 个点的泊松采样模式，使用这种模式对阴影贴图进行采样，可以获得稍微改进的结果（第二张图片），但是仍然可以看到一些锯齿。在第三张图片中，采样模式围绕中心进行逐像素的随机旋转，可以看到结构化阴影锯齿变成了噪声（不那么令人反感）。

自阴影问题和漏光问题，即阴影痤疮和 Peter Panning，在 PCF 中会变得更加糟糕。假设样本在阴影贴图上所占据的大小不超过一个纹素，那么斜率修正偏移仅仅会根据

表面与光源的夹角，来使得表面稍稍远离光源。此时如果表面上单个位置在更大的区域内进行采样，会导致一些测试样本被真实表面所遮挡。

人们提出了一些不同的额外偏移因子，可以在一定程度上减少自阴影问题出现的概率。Burley [212]提出了偏移锥（bias cone）的想法，每个测试样本都会根据自身与原始样本的距离，来成比例地向光源进行移动。Burley 建议将比例系数设置为 2.0，并添加一个较小的恒定偏移量，如图 7.24 所示。

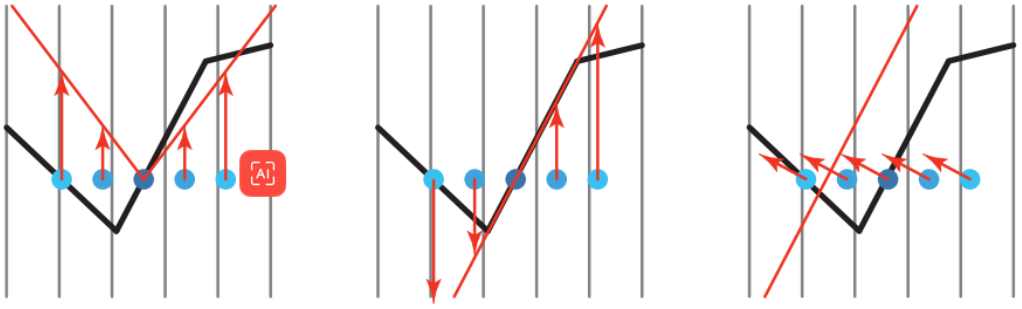


图 7.24：额外的阴影偏移方法。对于 PCF，会在原始样本位置（即图中五个点的中心位置）周围采集几个测试样本，所有的样本都应当被照亮。在左图中，会构建一个锥形的偏移量，并将样本向上移动。可以进一步增加这个锥形的陡峭程度，从而将位于右侧的样本上移足够的高度，使其能够被照亮；但这样做的风险是，其他真正被阴影所覆盖的样本位置（图中未显示）可能会发生漏光。在中间，所有的样本都被移动到了接收表面所在的平面上，这种方法对于凸面的效果很好，但是对于凹面可能会适得其反，如左侧所示。右图中，法线偏移会使得样本沿着表面的法线方向进行移动，移动的距离与表面法线和光线方向之间夹角的正弦值成正比。对于中心样本而言，可以认为是将其移动到位于原始表面上方的一个假想表面上，这种偏移方式不仅会影响样本的深度值（前面两种偏移方法也会影响），还会改变用于测试阴影贴图的纹理坐标（即向水平方向发生了移动）。

Schuler [1585]、Isidoro [1585]和 Tuft [1790]所提出的技术基于了这样一个观察：接收物表面本身的斜率，应当用于调整剩余样本的深度值。在这三位学者的方法中，Tuft 的方程[1790]最容易应用于级联阴影贴图。Dou 等人[373]对这个概念进行了完善和扩展，他解释了 z-depth 是如何以一种非线性的方式进行变化的。这些方法都基于了这样一个假设，即邻近的样本位置位于三角形所定义的同一平面上。这类技术被称为接收物平面深度偏移（receiver plane depth bias）或者其他类似的术语，它在许多情况下都十分精确，因为位于这个假想平面上的样本位置确实位于表面上；如果模型是凸的话，那么这个假想平面会位于原始表面的前方（法线方向）。如图 7.24 所示，位于凹陷表面附近的样本可能会被遮挡隐藏起来。将常量偏移、斜率修正偏移、接收物平面偏移、视图偏差和法线偏移组合使用，可以有效解决自阴影问题，但是对每个场景进行手动调整，仍然是有必要的[235, 1391, 1403]。

PCF 算法存在这样的一个问题，由于每个采样区域的宽度保持不变，因此阴影会表现出均匀柔和的外观，即所有阴影区域都具有相同的半影宽度。它在某些情况下是可以接受的，但是在遮挡物和接收物相接触的地方，会表现得不太正确，如图 7.25 所示。

图 7.25：PCF 和 PCSS。左图中，使用了一点点 PCF 滤波后的硬阴影。中间的图展示了由 PCF 生成的、具有恒定柔和宽度的软阴影。右图中，在物体与地面相接触的地方，PCSS 具有适当硬度的、可变宽度的软阴影效果。

7.6 PCSS

2005 年，Fernando [212, 467, 1252]提出了一种具有影响力的方法，它被称为百分比接近软阴影（percentage-closer soft shadow，PCSS）。它试图通过对阴影贴图上的附近区域进行搜索，来找到所有可能的遮挡物，并使用这些遮挡物与表面位置的平均距离，来决定采样区域的宽度，其方程如下：

$$w_{\text{sample}} = w_{\text{light}} \frac{d_r - d_o}{d_o} \tag{7.6}$$

其中 d_r 是接收物到光源的距离， d_o 是遮挡物到光源的平均距离。换句话说，随着遮挡物到接收物的平均距离越远（遮挡物距离光源越来越近， d_o 变小），表面采样区域的宽度就越大。我们可以重新观察图 7.22，想象一下如果此时遮挡物发生移动会产生什么效果，然后再来理解一下方程 7.6 的含义。图 7.2、图 7.25 和图 7.26 都展示了这样的例子。

如果没有发现遮挡物，那么该位置会被光源完全照亮，不需要任何进一步的处理；同样的，如果该位置被完全遮挡，那么也不需要进行后续处理。在其他情况下，我们需要对感兴趣的区域进行采样，并计算光源的近似贡献。为了降低计算成本，我们可以根据采样区域的宽度，来动态调整采样的数量。也可以使用一些其他的技术，例如：

位于较远位置的软阴影对于视觉感知而言没有那么重要，因此可以使用较低的采样率。

PCSS 的一个缺点是，它需要在阴影贴图中一个较大的区域范围内进行采样，从而寻找遮挡物的平均深度。使用上一小节中提到的旋转泊松圆盘采样模式，可以帮助隐藏欠采样导致的锯齿瑕疵[865, 1590]。Jimenez [832]注意到，泊松采样在运动情况下可能是不稳定的，他还发现使用介于抖动和随机之间的函数所形成的螺旋模式，在运动情况下具有更好的表现。

Sikachev 等人[1641]详细讨论了如何使用 SM 5.0 中的特性，来更快地实现 PCSS，这个特性是由 AMD 引入的，它通常被称为接触硬化阴影（contact hardening shadows，CHS）。这个新版本的算法还解决了基础 PCSS 的另一个问题，即半影区域的大小会受到阴影贴图分辨率的影响，如图 7.25 所示。可以通过首先生成阴影贴图的 mipmap，然后选择最接近于（由用户定义）世界空间尺度的 mipmap 层级，来最小化这个问题。该算法会对 8×8 的区域进行采样，从而找到平均遮挡物深度，这个操作只需要 16 次 $\mathrm{GatherRed}()$ 纹理调用即可。一旦计算出了半影区域的估计，就可以使用较高分辨率的 mipmap 来渲染较硬的阴影，使用较低分辨率的 mipmap 来渲染较软的阴影。

CHS 算法已经被大量的电子游戏所采用[1351, 1590, 1641, 1678, 1679]，并且对它的研究还在继续。例如：Buades 等人[206]提出了可分离的软阴影映射（separable soft shadow mapping，SSSM），其中对网格采样的 PCSS 过程被划分为了可分离的部分，可以在像素之间尽可能地重复使用相关元素，从而进一步提高效率。

对于那些每个像素采样多个样本（多 spp）的算法，有一个已被证明的、可以用于加速计算的概念是分层最小/最大阴影贴图（hierarchical min/max shadow map）。虽然阴影贴图的深度通常无法被平均，但是每个 mipmap 层级的最小值和最大值，对于算法加速而言还是很有用的。也就是说，可以生成两组 mipmap，其中一个用于保存每个区域中的最大 z-depth（这个 mipmap 有时会称为 HiZ，层次 z 缓冲），另一个则用于保存每个区域中的最小 z-depth。对于给定的采样位置、深度以及采样区域，这两个 mipmap 可以用于快速判断待测样本是完全光照的还是完全阴影的。例如：如果待测样本的 z-depth，大于 mipmap 相应区域中存储的最大 z-depth，那么则说明该样本一定位于阴影中，因此不需要对其进行后续的采样。这种类型的阴影贴图可以使得计算光源可见性的任务变得更加高效[357, 415, 610, 680, 1064, 1811]。

PCF 方法通过对附近的接收物位置进行采样，来计算光照百分比，从而模拟软阴影效果。PCSS 方法通过找到附近遮挡物的平均深度，从而决定测试样本的采样区域宽度。这些算法并不会直接去考虑光源的表面面积，而是会对着色点附近的表面进行采

样，这些算法都会受到阴影贴图分辨率的影响。PCSS 背后的主要假设是：平均遮挡物深度是一个对半影区域大小的合理估计。但是当两个遮挡物（例如路灯和远处的山）都部分遮挡了同一个像素时，上述假设就不成立了，并且可能会导致瑕疵。在理想情况下，我们希望能够确定从单个表面位置上，能够看到多少面光源。一些研究人员使用 GPU 来实现反向投影（backprojection），这个方法的思路是：将接收物的每个表面位置都视为一个观察点，将面光源视为观察平面的一部分，并尝试将遮挡物投影到这个平面上。Schwarz 和 Stamminger [1593]以及 Guennebaud 等人[617]对之前的工作进行了总结，并提出了他们自己的改进方法。Bavoil 等人[116]采用了一个不同的方法，他们使用深度剥离来创建多层阴影贴图。反向投影算法可以生成很好的结果，但是其逐像素的计算开销实在太大，因此它们还没有（到目前为止）在交互式应用程序中进行使用。

7.7 过滤阴影贴图

Donnelly 和 Lauritzen 提出了方差阴影贴图（variance shadow map，VSM）[368]，该方法允许对生成后的阴影贴图进行过滤操作。该算法将原始深度值存储在一个贴图中，并将原始深度值的平方存储在另一个贴图中。在生成这些贴图的时候，可以使用 MSAA 或者其他的抗锯齿技术。这些贴图可以被模糊化、生成 mipmap、放入面积积分表（SAT）[988]中或者应用其他方法。将这些贴图视为可过滤的纹理具有巨大的好处，因为当从贴图中检索数据的时候，可以使用一系列成熟的采样和过滤技术。

这里我们将稍微深入地介绍一下 VSM 算法，以便大致了解它是如何工作的；此外，对这类算法中的所有方法，都使用了相同类型的测试。有兴趣进一步了解该领域信息的读者，可以阅读相关的参考资料，同时我们也推荐读者阅读 Eisemann 等人[412]所撰写的书，它对这个话题进行了广泛的介绍和讨论。

首先，VSM 算法会在接收物的表面位置处，对深度图进行一次采样，从而返回最近遮挡物的平均深度，我们将这个平均深度称为一阶矩（first moment），记作 M_1。当 M_1 大于阴影接收物 t 的深度时，则认为该接收物位置完全处于光照中；当 M_1 小于阴影接收物 t 的深度时，则使用以下方程：

$$p_{\max}(t) = \frac{\sigma^2}{\sigma^2 + (t - M_1)^2} \tag{7.7}$$

其中 $p_{\max}$ 代表了位于光照下的样本最大百分比， σ^2 代表了方差， t 是阴影接受物的深度， M_1 阴影贴图中的平均期望深度。上文中我们提到 VSM 还会生成一个贴图用于存储阴影贴图深度值的平方，我们将这个贴图中的样本称为二阶矩（second moment），记作 M_2 ，我们使用 M_1 和 M_2 来计算方差，方程如下：

$$\sigma^2 = M_2 - M_1^2 \tag{7.8}$$

$p_{\max}$ 代表了接收物可见性百分比的上界，实际使用的光照百分比 p 不能大于这个值。这个上界来自于 Chebyshev's inequality（切比雪夫不等式）的单侧变体，这个方程试图使用概率论中的方法，来估计表面位置上的遮挡物分布有多少超出了表面到光源的距离。Donnelly 和 Lauritzen 证明了，对于具有固定深度的平面遮挡物和平面接收物而言， $p = p_{\max}$ ；因此方程 7.7 可以作为对许多真实阴影的良好近似。

Myers [1251]对这种方法的有效性建立了一种直观理解。阴影边缘附近的深度方差会增加，深度值差异越大，方差也越大。方程 7.7 分母中的 $(t - M_1)^2$ ，是可见性百分比的重要决定因素，如果这个值略高于零，就意味着平均遮挡物深度会稍微靠近光源一些（相较于接收物而言），并且值 $p_{\max}$ 会接近于 1（完全被照亮），这种情况会发生在半影完全被光源照亮的边缘处。随着向半影区域内部不断移动，平均遮挡物深度会越来越靠近光源，因此 $(t - M_1)^2$ 项会变大， $p_{\max}$ 会减小。与此同时，方差本身在半影区域内也会发生变化，方差在半影边缘处接近于零，当遮挡物深度不同且面积相等时，方差会达到最大。这些项相互制衡抵消，会在半影区域内形成一个线性变化的阴影，图 7.26 展示了 VSM 与其他算法的对比结果。

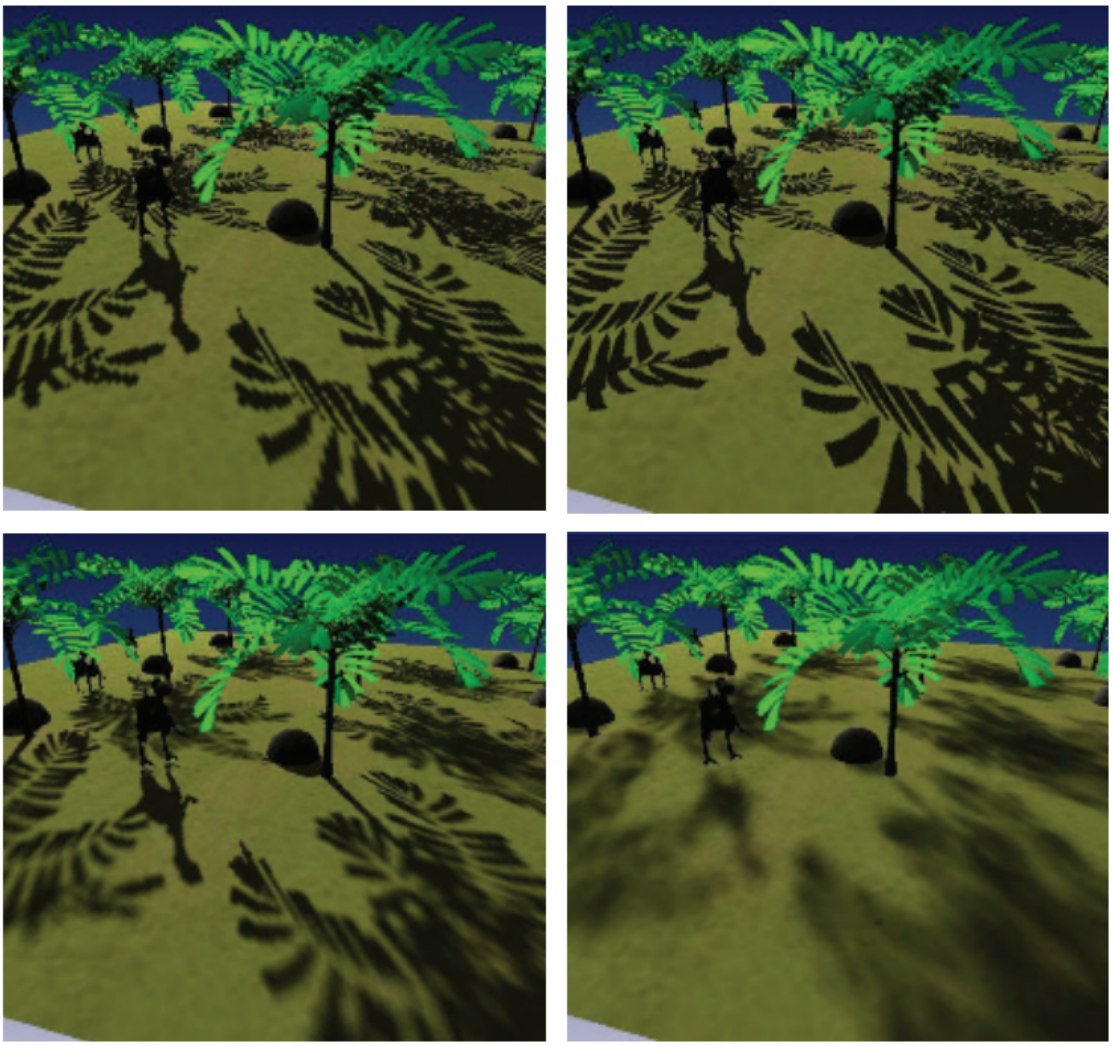


图 7.26：左上角：标准的阴影映射。右上角：透视阴影映射，在靠近观察者的地方增加了阴影贴图的纹素密度。左下角：PCSS，随着遮挡物与接收物之间的距离增加，阴影会逐渐软化。右下角：VSM，它具有恒定的软阴影宽度，每个像素仅使用了一个方差贴图的样本来进行渲染。

VSM 的一个重要特点是，它可以以一种十分优雅的方式，来处理由几何形状导致的表面偏差问题。Lauritzen [988]给出了如何使用表面斜率，来修改二阶矩的推导过程。VSM 的一个缺点是，数值稳定性可能会导致一些偏差和其他问题，例如：方程 7.8 中使用一个相似的值来减去另一个较大的值，这种类型的计算往往会加剧数值精度不足的问题，可以通过使用浮点纹理来避免这个问题。

总的来说，由于 GPU 的纹理能力（专门优化过的）得到了有效利用，VSM 算法可以在处理时间相同的情况下，显著提高阴影质量。PCF 需要更多的测试样本，从而避免在生成软阴影的时候产生噪声，但是这也会花费更多的时间；而 VSM 可以只使用单个高质量的样本，来确定整个区域的可见性百分比，并产生平滑的半影效果。这意味着在算法的限制范围内，可以在不增加额外开销的前提下，对阴影进行任意软化。

与 PCF 一样，滤波核的宽度决定了半影区域的宽度。通过寻找接收物和最近遮挡物之间的距离，来动态调整滤波核的宽度，从而生成令人信服的软阴影。对于宽度缓慢增加的半影区域，mimap 生成的样本对于半影区域覆盖率的估计结果较差，会产生方形锯齿。Lauritzen [988]详细介绍了如何使用 SAT（面积积分表）来生成质量更好的阴影，如图 7.27 所示。

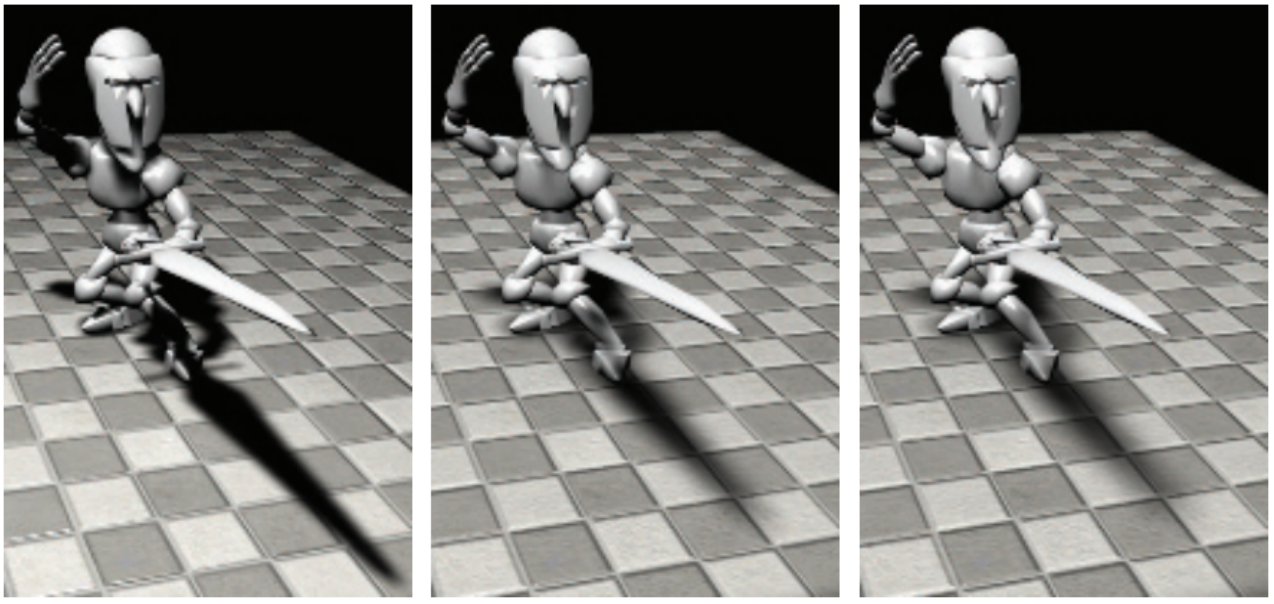


图 7.27：方差阴影映射（VSM），其中遮挡物到光源的距离，从左到右逐渐增加。[1300]

当两个或者两个以上的遮挡物覆盖同一个接收物，并且其中一个遮挡物十分靠近接收物时，方差阴影映射的半影区域会失效。上文中提到的切比雪夫不等式，将会产生一个与正确的光照百分比无关的最大光照百分比。由于最近遮挡物只能遮挡部分光线，从而使得方程近似值出现较大误差，这会导致漏光问题，即被完全遮挡的区域仍然能够接收到光线，如图 7.28 所示。通过在更小的区域上采集更多的样本，可以将 VSM 转化为 PCF 的形式，从而解决这个问题。与 PCF 一样，这需要对性能和质量进行权衡，但是对于较低阴影深度复杂度的场景而言，VSM 的效果已经很好了。Lauritzen [988]提出了一种由艺术家控制的方法来改善这个问题，即将较低光照百分比的区域视为完全位于阴影中，然后再将其余百分比范围重新映射到 0%–100% 之间，这种方法以半影区域的缩小为代价，从而减少了漏光现象。虽然漏光是一个严重的限制问题，但是 VSM 很适合用于从地形中生成阴影，因为这种类型的阴影很少会涉及多个遮挡物[1227]。

图 7.28：左图：使用了 VSM 来生成茶壶的阴影。右图：一个位于上方的三角形（未显示在图中）在茶壶上投射下阴影，产生了令人反感的阴影瑕疵。

由于能够使用滤波技术来快速生成平滑的阴影，因此滤波阴影映射（filtered shadow mapping）引起了人们的强烈兴趣，该技术最主要的挑战就是解决各种漏光问题。Annen 等人[55]提出了卷积阴影贴图（convolution shadow map），它对 Soler 和 Sillion 平面阴影算法[1673]背后的思想进行了扩展，其核心思路是在傅里叶展开中，对阴影深度进行编码。与 VSM 算法一样，这种贴图也可以被过滤，这个方法最终会收敛到正确的阴影效果，从而减少了漏光问题。

卷积阴影映射的一个缺点是，它需要对若干项进行计算和访问，这大大增加了计算成本和存储成本[56, 117]。Salvi [1529, 1530]和 Annen 等人[56]同时并独立地提出了基于指数函数来使用单个项的想法，这种方法被称为指数阴影贴图（exponential shadow map，ESM），这种方法会将深度值的指数，及其二阶矩存储到两个缓冲区中。指数函数更接近于阴影贴图测试时的阶跃函数（即在光照中或者不在光照中），因此它可以显著降低漏光瑕疵。同时，它避免了卷积阴影映射中另一个被称为 ringing 的问题，即在刚刚超过原始遮挡物深度时，可能会发生轻微的光线泄漏，从而导致环状漏光。

存储指数会由一个限制，就是其二阶矩的数值可能会非常大，有可能会超出浮点数的范围。为了提高精度，并让指数函数下降得更快，可以生成 z-depth，因为这个深度值是线性的[117, 258]。

与 VSM 相比，指数阴影映射（ESM）的质量有所提高；并且与卷积阴影贴图相比，它所占用的存储空间更低，性能更好，因此在这三种滤波方法中，指数阴影映射方法引起了人们的最大兴趣。Pettineo [1405]提出了一些其他方面的改进方法，例如使用 MSAA 来改善结果，并让其具备一些有限的透明度能力等，他还描述了如何使用计算着色器来提高过滤的性能。

Peters 和 Klein [1398]最近提出了矩阴影映射（moment shadow mapping）。它可以提供更好的阴影质量，代价是使用了四个或者更多数量的矩，增加了存储成本。可以通过使用 16 bit 整数来存储这些矩，从而降低开销。Pettineo [1404]实现了这种新方法，并将其与另一种方法指数方差阴影贴图（exponential variance shadow map，EVSM）进行了比较，他还提供了一个包含多种变体的代码库。

级联阴影贴图技术也可以应用于滤波贴图，从而提高精度[989]。与标准的级联阴影贴图相比，级联 ESM 的一个优点在于，可以为所有级联设置相同的偏移因子[1405]。Chen 和 Tatarchuk [258]详细介绍了级联 ESM 会遇到的各种漏光问题和其他瑕疵，并提出了一些解决方案。

滤波贴图可以被认为是一种廉价形式的 PCF，它只需要很少的样本；与 PCF 一样，这样产生的阴影具有恒定的半影宽度。这些滤波方法还可以与 PCSS 一起使用，从而提供可变宽度的半影效果[57, 1620, 1943]。一种对矩阴影映射的扩展方法，还可以提供光线散射和透明等效果[1399]。

7.8 体积阴影技术

透明物体会衰减并改变光线的颜色。对于某些类型透明物体，可以使用类似于章节 5.5 中所讨论的技术来模拟这种效果。例如：在某些情况下可以生成第二种类型的阴影贴图，将透明物体渲染到这个阴影贴图中，并且存储最近的深度值，以及颜色或者 alpha 覆盖率。如果一个接收物没有被不透明阴影贴图所遮挡的话，则进一步将其与透明的阴影贴图进行测试，如果这个物体被透明阴影贴图所遮挡的话，则根据需要来检索贴图中对应位置的颜色或者覆盖率[471, 1678, 1679]。这个想法让人想起章节 7.2 中的阴影投影和光源投影，存储下来的深度值会避免在透明物体与光源之间的接收物上进行投影。但是这种技术无法用于透明物体本身的渲染。

自阴影问题对于头发、云等物体的真实感渲染而言至关重要，因为这些物体要么是很小的，要么是半透明的，单一深度的阴影贴图无法用于处理这些情况。Lokovic 和 Veach [1066]首先提出了深度阴影贴图（deep shadow map）的概念，其中阴影贴图的每个纹素，都存储了光线随着深度的增加，如何进行衰减的函数。这个函数通常会由不同深度的一系列样本来近似，其中每个样本都有一个不透明度值。对于给定位置的深度，会使用阴影贴图中的两个样本（一个深度大于该位置，一个深度小于该位置）来计算阴影的效果。而 GPU 所面临的挑战是，如何高效地生成和计算这些函数。与一些顺序无关的透明算法相同（章节 5.5），这些算法使用了类似的方法，并

且也会遇到类似的挑战，例如：如何使用一些紧凑的数据存储形式，来准确表示每个函数。

Kim 和 Neumann [894]首先提出了一种基于 GPU 的方法，他们称之为不透明度阴影贴图（opacity shadow map），在一组固定的深度值中生成仅存储不透明度的贴图。Nguyen 和 Donnelly [1274]给出了一种这种方法的改进版本，生成了如图 17.2 所示的图像。然而，由于深度切片是平行且均匀的，因此需要使用大量的切片，来隐藏因线性插值而产生的不透明度瑕疵。Yuksel 和 Keyser [1953]通过创建更加贴合模型形状的不透明度图，来提高效率和质量；这样做可以减少所需的深度切片层数，因为这种方法中的每一层都更具代表性，对于最终图像的计算更加重要。

为了避免使用固定的切片设置，人们提出了一系列的自适应技术。Salvi 等人[1531]提出了自适应体积阴影贴图（adaptive volumetric shadow map），其中阴影贴图的每个纹素，都存储了不透明度和切片深度。使用像素着色器的一些操作来对光栅化后数据流（例如表面不透明度）进行有损压缩，这样可以避免需要大量内存来收集所有样本，并能够在一个集合中对它们进行处理。该技术类似于深度阴影贴图[1066]，但是压缩步骤是在像素着色器中动态完成的。将函数表示限制为一个较小的、固定数量的不透明度–深度值对，可以使得 GPU 上的压缩操作和检索操作效率更高[1531]。这样做的成本要比简单的混合高，因为曲线需要进行读取、更新和回写，而且它还取决于用来表示曲线的控制点数量。在这种情况下，该技术还需要硬件支持 UAV 和 ROV 功能（详见章节 3.8 末尾）。图 7.29 展示了使用这种技术的一个例子。

图 7.29：使用自适应体积阴影贴图来渲染的头发和烟雾阴影。[1531]

在游戏《GRID2》中，使用了自适应体积阴影映射方法，来渲染逼真的烟雾，其平均成本在每帧 2 毫秒以内[886]。Furst 等人[509]对他们为电子游戏实现的深度阴影贴图技术进行了描述，并提供了实现代码，他们使用链表来存储深值度和 alpha 值，并使用指数阴影映射，来生成光照区域和阴影区域之间的柔和过度。

对阴影算法的探索仍在继续，将各种算法和技术综合在一起使用变得越来越普遍。例如：Selgrad 等人[509]研究了如何使用链表来存储多个透明样本，并使用分散写入的计算着色器来生成贴图。他们将深度阴影贴图、滤波贴图以及其他一些技术元素结合在一起，为高质量的软阴影效果提供了一个更加通用的解决方案。

7.9 不规则 z-buffer

之所以各种各样的阴影贴图方法那么流行，有以下几个原因。第一，这类算法的成本是可预测的，并且随着场景大小的不断增加，它们的成本也不会发生暴涨，最坏的情况就是与图元数量成线性相关。第二，这类算法可以很好地被映射到 GPU 上，因为它们依赖于光栅化来对光源的观察视野进行规则采样。然而正是因为这种离散采样，问题就出现了，因为相机所看到的位置与光源所看到的位置并不是一一对应的。当光源对表面的采样频率低于相机的采样频率时，就会出现各种锯齿问题。即使二者的采样率比较合适，也会存在偏移问题，因为光源对表面的采样位置，与相机所看到的位置略有不同。

阴影体（shadow volume）算法提供了一个精确的、解析的解决方案，光源与表面的相互作用会生成一组三角形，对于任何的给定位置，它都可以判断该位置被光源照亮，还是位于阴影区域中。当在 GPU 上实现该算法时，它有一个严重的缺点，那就是不可预测的、不稳定的计算成本。近年来对这类算法的探索和改进[1648]是相当诱人的，但是到目前为止，它还没有在任何商业应用中进行使用。

从长远来看，另一种阴影测试的方法十分具有潜力：光线追踪。它的基本思想很简单，尤其是对于阴影效果而言，我们将在章节 11.2.2 中对其进行详细介绍。我们会从阴影接收物的表面位置出发，向光源发射一条射线，如果有物体阻挡了光线，则说明该表面位置位于阴影中。许多光线追踪渲染器的大部分代码，都在构建和使用分层的加速结构，从而最小化每条光线所需测试的物体数量。在动态场景中逐帧构建并更新这些加速结构，是一个在几十年前就被提出的话题，同时它现在也是一个持续研究的领域。

另一种方法是使用 GPU 的光栅化硬件来观察场景，除了存储 z 深度之外，还会存储每个网格单元中，光源遮挡物的额外边缘信息[1003, 1607]。例如：假设在阴影贴图的每个纹素上，都存储一个与网格单元重叠的三角形列表，这样的三角形列表可以通过保守光栅化（conservative rasterization）来生成，在保守光栅化中，如果三角形的任何部分与像素发生重叠（而不仅仅是与像素中心重叠，详见章节 23.1.2），就会生成一个片元。这种方案的一个问题是，通常需要对每个纹素所存储的数据量进行限制，这反过来又可能会导致对每个接收物表面位置状态判定的不准确。考虑到 GPU 的现代链表原理[1943]，在每个像素上存储更多的数据当然是可能的。然而，除了物理内存的限制之外，在每个纹素列表中存储可变大小的数据，会导致 GPU 的并行处理变得十分低效，因为一个 warp 中可能只有几个片元线程需要进行检索和计算，而其余的片元线程都处于空闲状态，没有任何任务可以执行。在构造着色器的时候，避免由于动态“if”语句和循环而导致的线程发散，对于性能而言至关重要。

上述的方法都是在阴影贴图中存储三角形或者其他数据，然后再利用这些数据来检测接收物的位置，还有一种选择是将问题反转过来，即我们在阴影贴图中存储接收物的位置，然后再用三角形对其进行检测。这种将接收物位置存储下来的想法，首先由 Johnson 等人[839]、Aila 和 Laine [14]提出，它被称为不规则 z-buffer（irregular z-buffer，IZB）。这个名字实际上有些误导人，因为这个缓冲本身是一个正常的、规则形状的阴影贴图。相反，缓冲区中所包含的内容是不规则的，因为阴影贴图中的每个纹素，将会存储一个或者多个接收物的位置，又或者是根本没有存储任何数据，如图 7.30 所示。

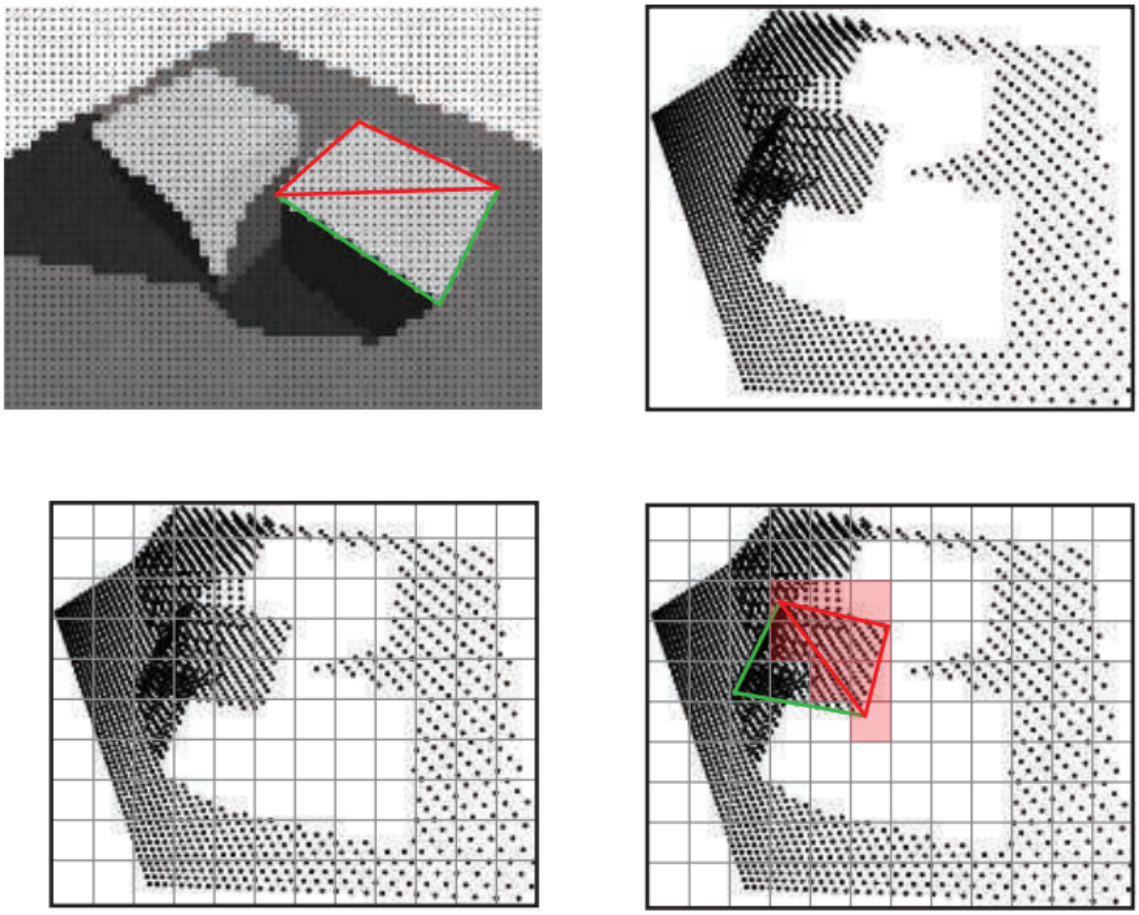


图 7.30：不规则 z-buffer。左上角：相机视野在像素中心处生成了一组点，图中展示了两个三角形所形成的一个立方体面。右上角：从光源视角生成了这些点。左下角：在这些点上覆盖了一个阴影贴图网格，对于每个纹素而言，会根据其网格单元内的所有点来生成一个列表。右下角：通过使用保守光栅化，来对红色三角形进行阴影测试，与其发生重叠的每个纹素被显示为浅红色，这些纹素列表中的所有点，都会和三角形进行测试，从而计算光照的可见性。[14]

使用 Sintorn 等人[1645]和 Wyman 等人[1645]所提出的方法，使用一个多 pass 的算法来创建 IZB，并对其内容在光源下的可见性进行测试。首先，会从相机视角来对场景进行渲染，从而找到可见表面的 z 深度。这些表面点会被转换到光源的视角下，并根据它们来生成光源的视锥体，使得这个视锥体紧密包裹这些点。然后将这些点存储在光源的 IZB 中，即每个点都会被放入对应纹素的列表中。需要注意的是，有些列表可能是空的，因为光源所看到的空间和物体表面，相机并不一定会看到。最后使用保守光栅化，来将遮挡物渲染到光源的 IZB 中，从而判断是否有任何点会被遮挡，被遮挡的点位于阴影中。这里使用了保守光栅化，它能够确保即使一个三角形并没有覆盖到纹素的中心，但是这个纹素中所包含的点也会用于对光源的可见性进行测试。

这个可见性测试会在像素着色器中进行。测试本身可以看作是光线追踪的一种形式，它会从图像点的位置处，向光源发射一条光线。如果这个光线与三角形平面的交点位于这个三角形内部，并且这个待测点的位置距离光源更远，那么就说明它会被遮挡。

当所有的遮挡物都被光栅化之后，光源的可见性结果将会被用于表面着色。这个测试过程也被称为截锥体追踪（frustum tracing），因为这个三角形可以被认为是定义了一个截锥体，它会对位于截锥体内部的点进行检查。

想要使得这种方法在 GPU 上很好地运行，需要格外仔细地编写代码。Wyman 等人[1930, 1932]指出，他们最终版本的代码，要比最初的原型快了两个数量级。这种性能提升的部分原因是对算法的直接改进，例如：剔除表面法线背对光源的图像点（因为它们永远不会被这个光源照亮）；同时避免为空纹素生成片元。其他的一些性能提升，来自对 GPU 数据结构的改进，以及在每个纹素中使用较短的、长度相似的点列表，来最小化线程发散所带来的效率降低。为了便于说明算法原理，图 7.30 展示了一个较低分辨率的阴影贴图，其中有一些纹素具有很长的点列表，而最理想的情况是，每个列表中仅包含一个图像点。使用更高分辨率的阴影贴图可以生成较短的列表，但是这也增加了用于遮挡物测试的片元数量。

从图 7.30 的左下角可以看出，由于透视效应，左侧地面上可见点的密度要明显高于右侧。使用级联阴影贴图有助于降低这些区域的列表长度，这主要是通过在更靠近相机的区域，使用较高分辨率的阴影贴图来实现的。

这种方法避免了其他方法会有的采样问题和偏移问题，并且可以提供完美的尖锐阴影。出于审美和感知的原因，通常我们都更希望实现软阴影的效果，但是这可能会与附近的遮挡物产生偏移问题，例如 Peter Panning。Story 和 Wyman [1711, 1712]对混合的阴影技术进行了探索，其核心思想是利用遮挡物的距离，来对 IZB 和 PCSS 所产生的阴影进行混合，当遮挡物距离较近时，则使用硬阴影；当遮挡物距离较远时，则使用软阴影，如图 7.31 所示。阴影质量对于附近物体而言通常是最重要的，因此，可以通过仅在选定的物体子集上使用这个技术，来降低 IZB 的成本。这种方法已经成功地应用在了电子游戏中，本章节最开始的地方，就展示了这种技术渲染的结果，如图 7.2 所示。

图 7.31：左图：PCF 会为所有物体都生成均匀的软阴影。中间：PCSS 会根据与遮挡物的距离，来对阴影进行软化，但是与板条箱左下角重叠的树枝阴影，会产生一些瑕疵。右图：将来自 IZB 的尖锐阴影与来自 PCSS 的柔和阴影进行混合，给出了一个改进的结果。[1711]

7.10 其他应用

将阴影贴图视为空间体积的一种定义来区分明暗，也有助于确定物体的哪些部分位于阴影区域中。Gollent [555]介绍了 CD Projekt 的地形阴影系统，是如何计算每个区域被遮挡的最大高度的，这个最大高度信息不仅可以用于生成地形阴影，还可以用来对树木和场景中的其他元素进行阴影处理。为了找到每个区域的最大遮挡高度，会从太阳的视角来生成可见区域的阴影贴图，然后会在每个地形高度场的位置上，检查对太阳上的能见度。如果某个高度位于阴影中，那么会在这个高度的基础上，增加一个固定的步长，继续对可见性进行检查，直到太阳进入视野中，然后再使用二分搜索来找到准确的最大遮挡高度。换句话说，我们会沿着一条垂直于地面的线前进，通过迭代来找到这个垂线与阴影贴图表面相交的位置，这里我们将阴影贴图看作是一个高度场表面，该表面会将光照区域和阴影区域分开。相邻位置的高度会通过插值生成，以便能够在任何位置上找到这个遮挡高度。图 7.32 展示了一个使用这种技术来生成地形高度场的软阴影的例子。我们将在第 14 章中看到更多光线步进穿越明暗区域的用法。

图 7.32：每个高度场位置都会计算第一次看见太阳时的高度，然后使用这个高度信息来渲染地形的阴影。请注意阴影边缘的树木，是如何被正确渲染出阴影的。[555]

最后一个值得一提的方法是渲染屏幕空间阴影（screen-space shadow）。由于分辨率的限制，阴影贴图通常无法在较小的几何形状上产生准确的遮挡。尤其是在渲染人脸的时候，因为我们会特别容易注意到人脸上的任何视觉瑕疵，例如：由于错误的

遮挡关系，可能会导致鼻孔内部被照亮，这看起来会非常不和谐。当然我们也可以使用更高分辨率的阴影贴图，或者针对感兴趣区域使用单独的阴影贴图，这样做都缓解这个问题，但是还可以换一个思路，使用已经存在的数据结果来生成阴影。在大多数现代渲染引擎中，来自相机视角的深度缓冲，会在较早的 pass 中生成，它在后续的渲染阶段中都是可以使用的。存储在深度缓冲的数据可以被视为一个场景高度场。通过对这个深度缓冲的迭代采样，我们可以执行一个光线步进的过程（详见章节 6.8.1），并检查到光源的方向是否会被遮挡。由于这个过程涉及到对深度缓冲的重复采样，因此其计算成本很高，但是这样做可以为过场动画中的特写镜头提供高质量的渲染结果，因此在这方面额外花费几毫秒通常也是合理的。这种方法由 Sousa 等人[1678]提出，并且在今天的许多游戏引擎中被广泛使用[384, 1802]。

这里我们对本章节中的内容进行简要总结，其中阴影映射算法是目前为止最为常用的，它可以将阴影投射到任意表面上。当阴影投射在一个较大的区域时（例如室外场景），使用级联阴影贴图（CSM）可以提高采样质量。通过 SDSM 算法，来为近裁剪平面找到一个合适的最大距离，可以进一步提高精度。PCF 可以在一定程度上软化阴影；PCSS 及其变体方法，可以赋予阴影接触硬化的特点，即让 shadow caster 在接触处的阴影变得尖锐；而 IZB 可以提供精确的硬阴影效果。滤波阴影贴图可以快速计算的软阴影效果，并且当遮挡物距离接收物较远时（例如地形），它的效果特别好。最后，屏幕空间中的相关技术可以提高精度，但是其计算成本较高。

在本章节中，我们重点介绍了当前应用程序所使用的关键概念和关键技术。每种方法都有自己的优势，具体选择使用哪种方法，取决于场景的尺度大小、物体类型（静态物体还是动态物体）、材质类型（不透明、透明、头发或者烟雾）、光源的数量和光源的类型（静态光源还是动态光源；局部光源还是方向光；点光源、聚光灯还是面光源），以及诸如底层纹理如何隐藏瑕疵和锯齿等因素。随着 GPU 能力的不断发展和提高，因此我们期望在未来几年里，能够看到与硬件相匹配的新算法出现。例如：章节 19.10.1 中所描述的稀疏纹理技术，已经应用在了阴影贴图的存储方面，它可以提高阴影贴图的分辨率[241, 625, 1253]。Sintorn，Kampe 和其他人[850, 1647]使用了一种创造性的方法，来将二维阴影贴图转换为三维体素集合（有关体素的内容详见章节 13.10）。使用体素的一个优点是，每个体素都可以被划分为光照状态或者阴影状态，因此需要最少的存储空间。一个高度压缩的稀疏体素八叉树表示，可以为大量的光源和静态遮挡物存储阴影信息。Scandolo 等人[1546]将他们的压缩技术与基于双重阴影贴图的区间方案相结合，从而获得更高的压缩比。Kasyan [865]使用了体素锥形追踪（详见章节 13.10），来从面光源中生成软阴影，图 7.33 展示了这样的一个例子。图 13.33 展示了更多锥形追踪阴影的效果。

图 7.33：第一行图片是使用基本的软阴影近似技术生成的。第二行图片展示了基于体素的面光源阴影效果，它是在一个体素化的场景中，使用锥形追踪来生成的。请注意观察两幅图片中汽车的漫反射阴影。第二行图片中的光照效果还可以随着一天中时间的变化而变化。[865]

补充阅读和资源

本章节中所介绍的算法，都基于了这样一个原则，即阴影算法需要具有什么样的特点才能用于交互式渲染中，答案是：可预测的、稳定的质量和性能。我们尽可能避免在渲染的这一子领域中，对所做的研究进行详尽分类和详尽介绍，因为已经有两本优秀的书籍关注了这个主题。Eisemann 等人[412]的《Real-Time Shadows》一书，关注了在交互式渲染中所使用的阴影算法，它对各种算法及其优势和成本进行了讨论。SIGGRAPH 2012 的一个课程提供了本书的摘录，同时引用了一些较新的工作和算法

[413]。他们在 SIGGRAPH 2013 课程中的演讲，可以在他们的网站 www.realtimeshadows.com 中找到。Woo 和 Poulin 的书《Shadow Algorithms Data Miner》[1902]对交互式渲染和批量渲染中所用到的各种阴影算法进行了综述。这两本书都提供了相应的参考文献，包含了该领域中的数百篇研究论文。

Tuft 的两篇文章[1791, 1792]对常用的阴影映射技术，以及其中涉及的相关问题进行了很好的综述。Bjørge [154]提出了一系列适用于移动设备的流行阴影算法，并对各种算法生成的图像进行了对比。Lilley 的演讲[1046]对实用阴影算法进行了坚实且广泛的综述，其关注的重点是地理信息系统（GIS）中的地形渲染。Pettineo [1403, 1404]和 Castano [235]的博客文章中，包含了大量的实用技巧和解决方案，以及一个 demo 代码库，因此特别具有使用价值。Scherzer 等人[1558]针对硬阴影的工作进行了简短总结。Hasenfratz 等人[675]对软阴影算法进行了调研，虽然这个调研如今已经过时了，但是它在一定程度上，涵盖了大量软阴影算法的早期工作。

Chapter 8 Light and Color 光与颜色

> John Keats——“Unweave a rainbow, as it erewhile made. The tender-person’d Lamia melt into a shade.”

> 约翰·济慈——“如刚才拆解彩虹那般，让光鲜娇嫩的 Lamia 黯然失色。”（19 世纪初期英国浪漫派诗人；1795—1821）

在前几章中我们所讨论的许多 RGB 颜色值，都代表了光线的强度和着色。而在本章节中，我们将学习由这些值所度量的各种物理光量，为后续章节从一个更加物理的角度来讨论渲染，打下坚实的基础。我们还将了解更多有关渲染过程中经常被忽视一部分内容，即：将表示场景线性光量的颜色，转换为最终要进行显示的颜色。

8.1 光量

任何基于物理的渲染方法，其第一步都是以一种精确的方式，来对光进行量化（quantify）。在本小节中，我们首先会介绍辐射度量学（radiometry），因为它与光线的物理传输过程紧密相关。然后我们紧接着会讨论光度学（photometry），它会根据人眼的灵敏度，对光线值进行加权。我们对颜色的感知，实际上是一种心理物理学（psychophysical）现象，即对物理刺激的心理感知。有关颜色感知的内容将在色度学（colorimetry）一节中进行讨论。最后，我们会讨论使用 RGB 颜色值来进行渲染的有效性。

8.1.1 辐射度量学

辐射度量学（radiometry）研究的是对电磁辐射（electromagnetic radiation）的测量，这种辐射会以波的形式进行传播，我们将在章节 9.1 中进行更加详细地讨论。具有不同波长的电磁波，往往会具有不同的特性。波长是指两个具有相同相位的相邻点之间的距离，例如两个相邻峰值之间的距离。在自然界中，电磁波的波长范围很广，既有长度不到百分之一纳米的伽马波，也有到长达数万公里的极低频（extreme low frequency，ELF）无线电波。人眼所能看到的电磁波，实际上只占据这个范围中的很一小部分，从大约从 400 纳米的紫光延伸到 700 多纳米的红光，如图 8.1 所示。

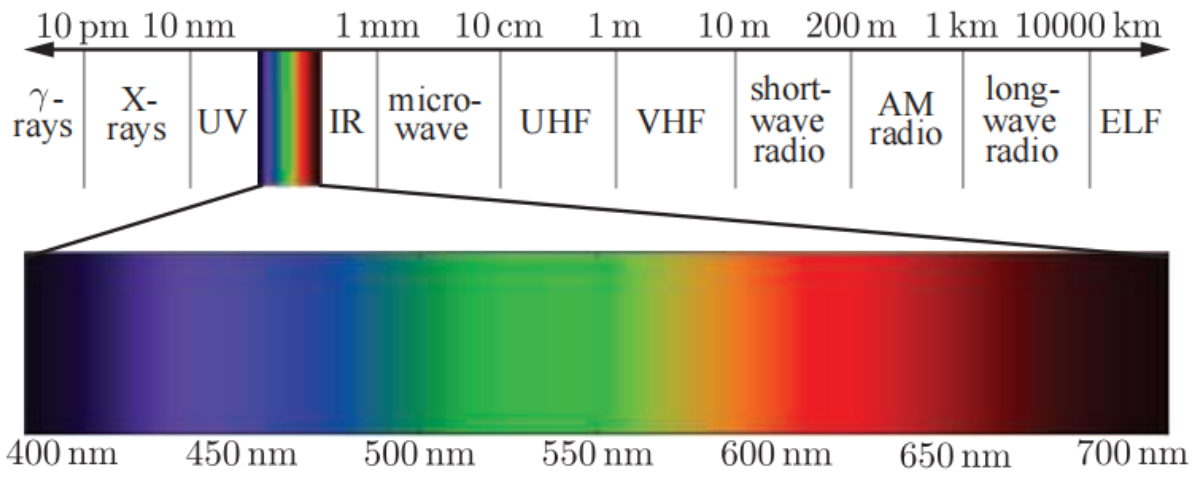


图 8.1 可见光的波长范围，以及这个范围在整个电磁波谱范围内所占据的位置。

各个辐射量（radiometric quantity）的存在是为了对电磁辐射的各个方面进行测量和度量，例如：总能量、功率（随时间变化的能量）以及相对于面积、方向或者二者的功率密度等，表 8.1 对这些物理量进行了总结。

Name	Symbol	Units
radiant flux	Φ	watt (W)
irradiance	E	$\mathrm{W/m^2}$
radiant intensity	I	W/sr
radiance	L	$\mathrm{W/(m^2 sr)}$

表 8.1：辐射量及其单位。

在辐射度量学中，最基本的单位是辐射通量（radiant flux） Φ，辐射通量是指辐射能量随时间的流动变化，又叫做功率（power），其单位为瓦特(watts，W)。

> （译者注：在后文中，irradiance 和 radiance 都不会进行翻译，中文翻译容易混淆）

辐照度（irradiance）是辐射通量相对于面积的密度，即 $d\Phi/dA$ 。irradiance 是相对于一个面积来进行定义的，这个面积可能是空间中的一个假想区域，但是在渲染中一般都是物体的表面。irradiance 的单位是瓦特每平方米（ W/m^2 ）。

在我们讨论下一个物理量之前，我们首先需要介绍一下立体角（solid angle）的概念，它是对二维角度概念的三维扩展。角度可以被认为是对平面上连续方向集合大小的度量，其弧度值等于这组方向与半径为 1 的封闭圆相交，所产生圆弧的长度。类似地，立体角是三维空间中连续方向集合大小的度量，其单位为立体弧度

（steradians，缩写为 sr），它由这组方向与半径为 1 的封闭球体相交，所产生面片的面积进行定义的[544]。立体角使用符号 ω 表示。

在二维空间中，弧度值为 2π 的角度（ 360° ）可以覆盖整个单位圆。将其扩展到三维空间中，立体弧度值为 4π 的立体角，将会覆盖整个单位球体的面积。图 8.2 展示了一个立体弧度为 1 的立体角。

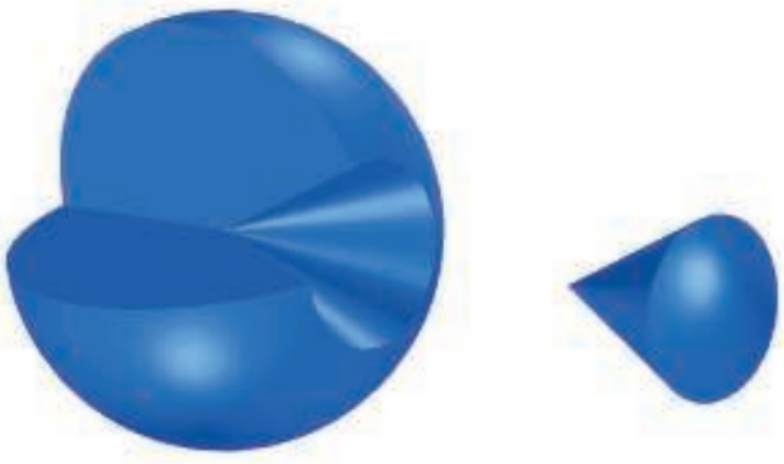


图 8.2：从球体的剖面图上移除一个立体弧度为 1 的圆锥体。立体角与具体的形状本身无关，最关键的是球体表面上的覆盖率。

现在我们可以引入辐射强度（radiant intensity） I ，即辐射通量相对于方向的密度，更准确地说，是相对于立体角的密度（ $(d\Phi/d\omega)$ ）。它的单位是瓦特每立体弧度（ W/sr ）。

最后，辐射度（radiance） L 是对单条光线中电磁辐射的度量。更精确地说，它是辐射通量相对于面积和立体角的密度（ $d^2\Phi/dAd\omega$ ）。这里的面积位于垂直于光线的平面上，如果想要在其他方向上对表面施加辐射，则必须使用余弦因子进行校正。我们可能还会遇到一些其他对于 radiance 的定义，它们使用了术语“投影面积”来代表这个校正因子。

radiance 是传感器（例如眼睛或者相机）所直接测量的对象（更多细节详见章节 9.2），因此它对渲染而言至关重要。计算着色方程的目的就是沿着给定的光线，计算从着色点到相机的 radiance；沿着这条光线计算出来的结果 L ，与第 5 章中的 $\mathbf{c}_{\text{shaded}}$ 在物理上是等价的。radiance 的公制单位是瓦特每平方米每立体弧度（ W/m^2sr ）。

环境中的 radiance 可以被认为是五个变量（或者六个变量，将波长考虑在内）的函数，它被称为辐射分布（radiance distribution）[400]；其中有三个变量指定了位置，另外两个变量指定了方向，这个分布函数描述了在空间中任何地方传播的任何光

线。根据上面的描述，我们可以这样来理解渲染过程：将眼睛和屏幕定义为一个点和一组方向（例如从眼睛出发，穿过每个像素的光线），然后使用这个函数，在这组方向上对眼睛所在的位置进行评估。章节 13.4 中所讨论的基于图像的渲染，使用了一个与之相关的概念，被称为光场（light field）。

在着色方程中，radiance 通常会以 $L_o(\mathbf{x}, \mathbf{d})$ 或者 $L_i(\mathbf{x}, \mathbf{d})$ 的形式出现，它代表了从点 x 发出，或者进入点 x 的 radiance 具体是多少。方向向量 $\mathbf{d}$ 表示了光线的方向，按照惯例，它总是会指向远离着色点 x 的方向。虽然这种惯例在 L_i 的情况下，可能有点令人困惑，因为此时光线方向 $\mathbf{d}$ 与光线实际的传播方向相反，之所以这样进行设计，是因为它对点乘之类的运算十分方便。

radiance 的一个重要特性是，在忽略了雾等大气效应影响的前提下，radiance 不会受到传播距离的影响。换句话说，无论一个表面与相机的距离有多远，它们都将具有相同的 radiance。当距离相机越远时，这个表面所覆盖的像素就越少，但是来自表面上每个像素的 radiance 是恒定的。

大多数光波中都包含了许多不同波长的单色光，这通常可以被可视化为一个光谱功率分布（spectral power distribution，SPD），它是一个展示了光线能量如何在不同波长之间分布的图片，图 8.3 给出了三个 SPD 的例子。值得注意的是，尽管图 8.3 第二行和第三行所展示 SPD 具有很大的差异，但是它们被人眼所感知到的颜色是相同的。很显然，人眼是一种精度很差的光谱分析仪。我们将在章节 8.1.3 中详细讨论有关颜色视觉的内容。

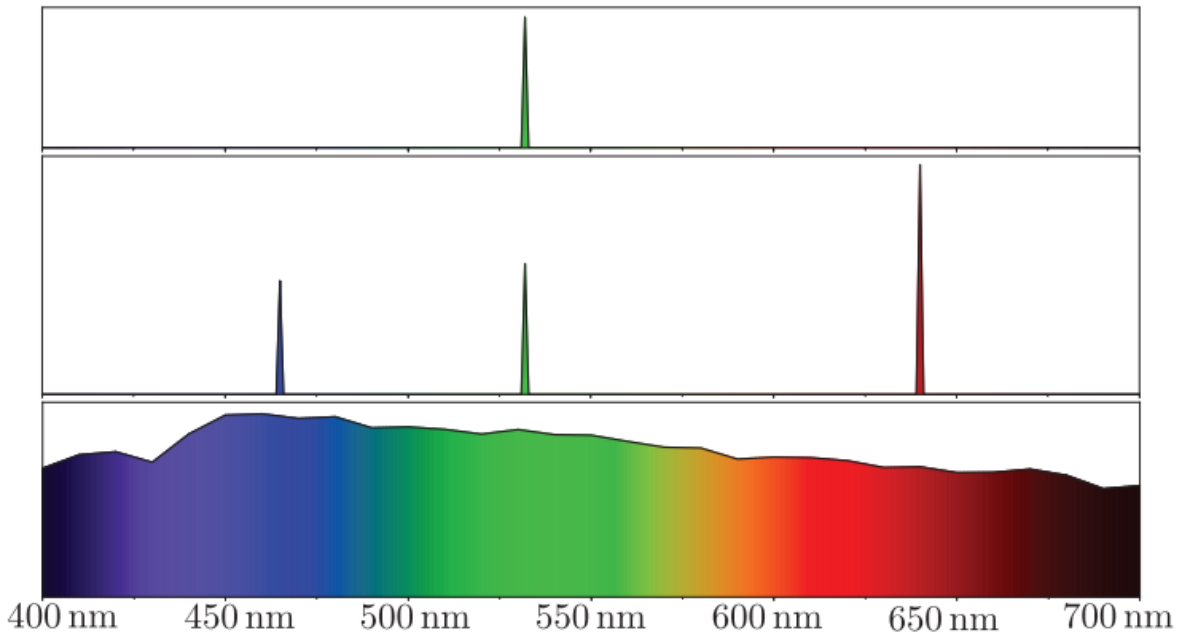


图 8.3：三种不同光波的光谱功率分布（SPD）。第一行是一个绿色激光的 SPD，它有一个非常狭窄的光谱分布，其波形类似于图 9.1 中的简单正弦波。第二行的 SPD 是由相同的绿色激

光，外加两个额外的激光所组成的，分别是红色激光和蓝色激光。将这些激光的波长和相对强度转换到 RGB 激光投影显示器上，会显示出中性白色。第三行的 SPD 是标准的 D65 光源，这是一个典型的中性白色参考值，旨在代表室外的自然光照，类似第三行的 SPD，其能量连续地分布在可见光谱上，是典型的自然光。

之前我们介绍的所有辐射量都具有光谱分布。由于这些分布是波长上的能量密度，因此它们的单位是各个辐射量原始单位再除以纳米。例如：irradiance 光谱分布的单位是瓦特每平方米每纳米（W/m^2ns）。

完整的 SPD 对于渲染而言十分笨重，尤其是在交互式速率下；因此在实际应用中，这些辐射量会使用 RGB 三元组来进行表示。在章节 8.1.3 中，我们将会解释这些三元组与光谱分布之间的关系。

8.1.2 光度学

辐射度量学仅仅对物理量进行了研究，它完全没有考虑人眼的感知。与此相关的一个领域被称为光度学（photometry），它与辐射度量学类似，不同之处在于，它会根据人眼的敏感度，对辐射度量学中的一切事物进行加权处理。通过乘以 CIE 光度曲线（CIE photometric curve），辐射度量学中的计算结果可以被转换为相应的光度单位。CIE 光度曲线是一条以 555 纳米为中心的钟形曲线，它代表了人眼对各种波长光线的响应程度[76, 544]，如图 8.4 所示。

> 更加完整、准确的名称应当是“CIE 感光光谱照明效率曲线”（CIE photopic spectral luminous efficiency curve）。其中“phototopic”一词指的是每平方米亮度超过 3.4 坎德拉的照明条件，在这种照明条件下，人眼的视锥细胞（cone cell）会活跃起来。有一个与之对应的“scotopic”CIE 曲线，其中心在 507 纳米左右，这是当人眼适应黑暗（低于 0.034 坎德拉每平方米）之后对应的情况，即没有月亮的夜晚或者更暗的照明条件，在这种照明条件下，人眼的视杆细胞（rod cell）会活跃起来。

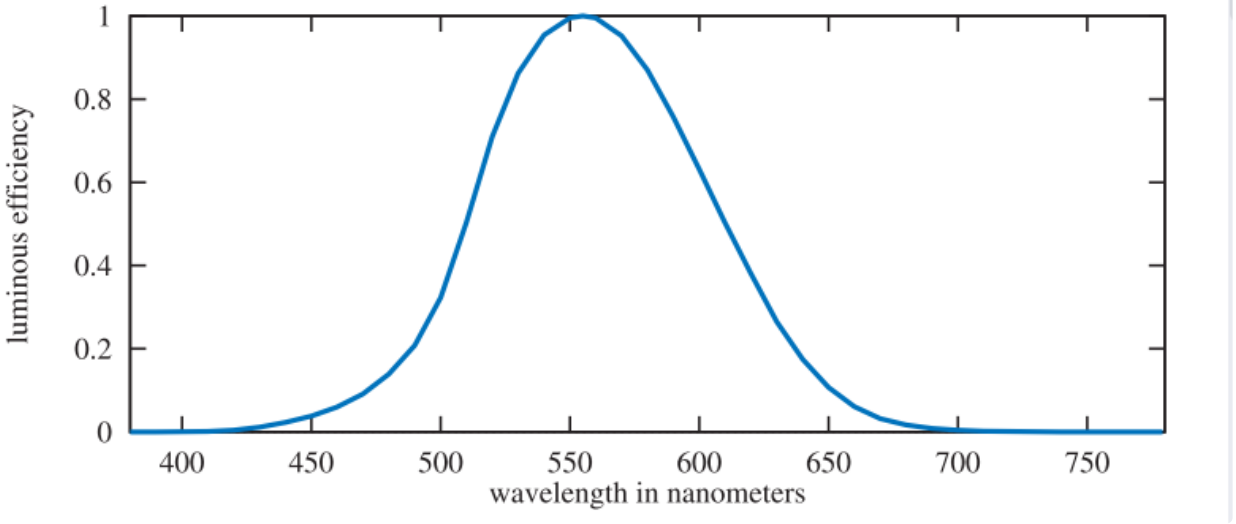


图 8.4：光度曲线。

这个转换曲线与测量单位，是光度学理论和辐射度量学理论之间的唯一区别。每个辐射物理量都有一个对应的光度学物理量，表 8.2 给出了它们的名称和单位。这些光度学物理量的单位都有预期的对应关系（例如：lux 的单位是 lumen 每平方米）。虽然逻辑上来讲，lumen（流明）应该是个基本单位，但是在历史上，candela（坎德拉）则被定义为基本单位，而其他单位都是从坎德拉中派生出来的。在北美，照明设计师仍然会使用已被废弃的英制测量单位，而不是使用 lux（勒克斯），这个英制单位叫做英尺烛光（foot-candle，fc）。无论哪种情况，大多数测光仪都会对 illuminance 进行测量，这个单位在照明工程（illumination engineering）中十分重要。

Radiometric Quantity: Units	**Photometric Quantity: Units**
radiant flux: watt (W)	luminous flux: lumen (lm)
irradiance: W/m^2	illuminance: lux (lx)
radiant intensity: W/sr	luminous intensity: candela (cd)
radiance: $W/(m^2 sr)$	luminance: cd/m^2 = nit

表 8.2：辐射度量学和光度学中的物理量，以及各自的单位。

Luminance 通常用来描述平面的亮度。例如：高动态范围（high dynamic range，HDR）电视屏幕的峰值亮度通常在 500 到 1000 尼特（nit）之间。相比之下，晴朗天空的亮度大约为 8000 尼特，60 瓦的电灯泡约为 12 万尼特，地平线上的太阳约为 60 万尼特[1413]。

8.1.3 色度学

在章节 8.1.1 中我们已经看到，人眼对于颜色的感知与光线的 SPD（光谱功率分布）密切相关。同时我们还知道了，二者之间并不是简单的一一对应关系。图 8.3 第二行和第三行所展示的 SPD 完全不同，但给人的感知却是完全相同的。色度学（colorimetry）研究的就是 SPD 和颜色感知之间的关系。

人眼可以分辨大约 1000 万种不同的颜色。对于颜色感知，人眼的视网膜上有着三种不同类型的视锥感受器（细胞），每种感受器对于不同波长的光线都有不同的反应。其他动物的眼睛则有着不同数量的颜色感受器，有些动物的眼睛甚至多达 15 个[260]。因此，对于一个给定的 SPD，我们的大脑只会从这些感受器中接收到三种不同的信号，这就是为什么只用三个数字就可以精确地表示任何颜色的原因[1707]。

但是具体使用哪三个数字来表示一个特定的颜色呢？CIE（Commission Internationale d'Eclairag）提出了一套测量颜色的标准条件，并根据这些标准条件进行了配色实验（color-matching）。在配色实验中，将三种颜色的光源投射在一个白色的屏幕上，并使它们的颜色叠加在一起，形成一个色块。一个用于匹配的待测颜色会投影在这个色块旁边，这个待测颜色是只有一个波长的单色光（激光）。然后观察者可以使用校准到 $[-1, 1]$ 加权范围内的旋钮，来调整这三种颜色的灯光，直到与待测颜色相匹配为止。为了和某些特殊颜色相匹配，有时可能会需要一个负权重，这样的权重意味着，对应的光源会被添加到待测颜色的色块中。图 8.5 展示了三种光源（被叫做 r 、 g 和 b ）的一组测试结果，这三个光源几乎是单色的，每个光源的能量分布都狭窄地聚集在以下波长的附近： r 为 645 纳米，g 为 526 纳米，b 为 444 纳米。将每组匹配权重与待测波长的颜色相关联的函数，被称为颜色匹配函数（color-matching function）。

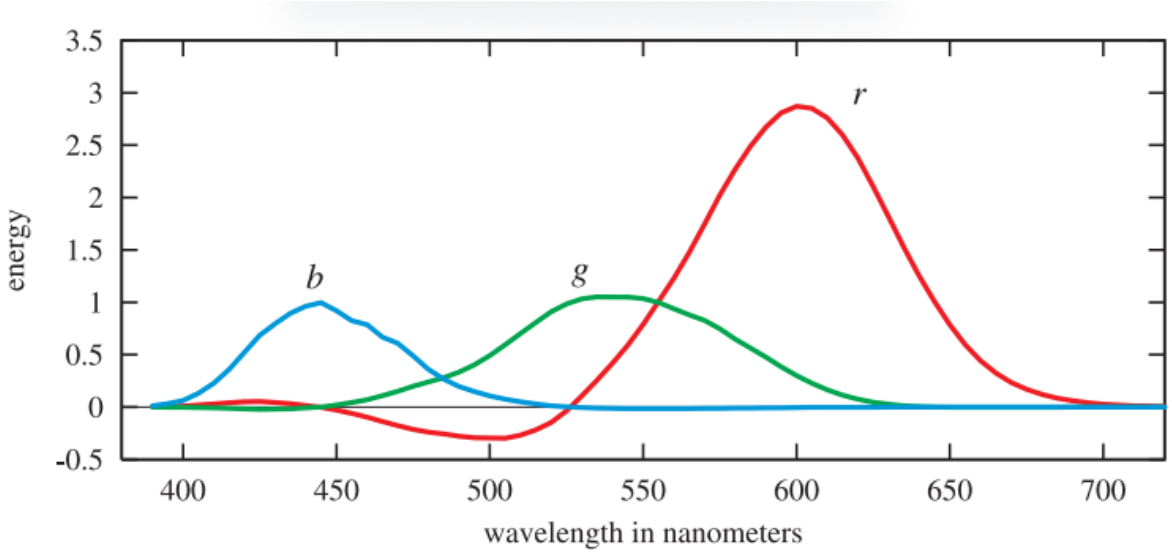


图 8.5： r 、 g 和 b 的二度颜色匹配曲线，来自 Stiles 和 Burch[1703]。这些颜色匹配曲线，不能与配色实验中所使用的纯波长光源的光谱分布曲线相混淆。

这些函数可以将一个 SPD 转换为三个值。对于给定的单一波长的光源，可以从曲线图中读取到三种颜色光的权重，通过设置对应旋钮，调节后的叠加色块在屏幕上可以和另一个色块给人相同的颜色感知。对于任意的 SPD，可以将其与颜色匹配函数相乘；每条结果曲线下的面积（即积分）便给出了三种颜色光的相对数量，然后对三种颜色的光进行调节，从而与该 SPD 产生的感知颜色相匹配。完全不同的 SPD 也可能会解析成相同的三个权重值，即这些 SPD 在人眼看起来是完全相同的。给出相同权重的不同 SPD，被称为同色异谱（metamer）。

3 个加权的 r 、 g 、 b 光源无法直接表示所有人眼可见的颜色，因为它们的颜色匹配函数在某些波长上具有负权重。CIE 提出了三种不同的假想光源，这些光源的颜色匹配函数在所有可见光波长范围内都为正。这些光源的颜色匹配函数，是原始 r 、 g 和 b 颜色匹配函数的线性组合。新的颜色匹配函数要求其光源的 SPD 在某些波长上为负，这在物理上是不可能发生的，因此这些光源都是无法实现的数学抽象光源。它们的颜色匹配函数分别被表示为 $\bar{x}(\lambda)$ ， $\bar{y}(\lambda)$ 和 $\bar{z}(\lambda)$ ，如图 8.6 所示。其中颜色匹配函数 $\bar{y}(\lambda)$ 与上文所提到的光度曲线（图 8.4）相同，通过该曲线，可以将光线的 radiance 转换为 luminance。

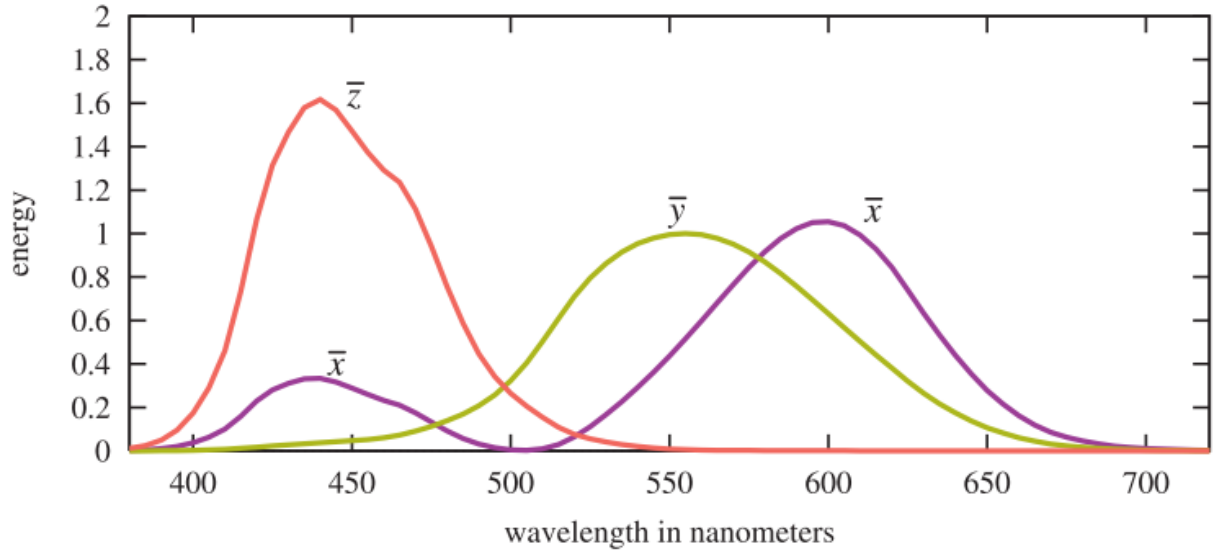


图 8.6： Judd–Vos 修正的 CIE（1978）2 度颜色匹配函数。请注意两段 $\bar{x}$ 其实是同一条曲线的两个部分。

与前一组颜色匹配函数一样，使用 $\bar{x}(\lambda)$ ， $\bar{y}(\lambda)$ 和 $\bar{z}(\lambda)$ ，通过乘法和积分可以将任意的 SPD $s(\lambda)$ 转换为三个数字：

$$X = \int_{380}^{780} s(\lambda)\bar{x}(\lambda)d\lambda, \quad Y = \int_{380}^{780} s(\lambda)\bar{y}(\lambda)d\lambda, \quad Z = \int_{380}^{780} s(\lambda)\bar{z}(\lambda)d\lambda \tag{8.1}$$

这些 X 、 Y 、 Z 三色值（tristimulus values）是定义在 CIE XYZ 空间中的权重。通常来说，将颜色划分为亮度（luminance）和色度（chromaticity）是很方便的。其中色度是与亮度无关的颜色特征，例如：两种蓝色（一种深，一种浅）可以具有相同的色度，但是它们的亮度不同。

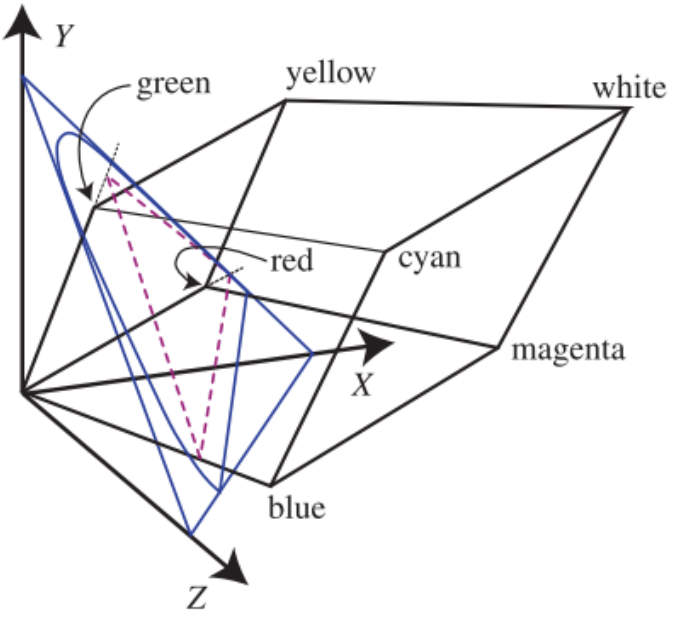


图 8.7：图中的 RGB 颜色立方体是使用 CIE RGB 原色形成的，它位于 XYZ 空间中；这个立方体被投影到了平面 $X+Y+Z=1$ 上，投影轮廓使用紫色虚线进行表示。蓝色的轮廓线代表了所有可能的色度值范围。从坐标原点发射出的每条射线，其上的坐标点都具相同的色度值，它们仅在亮度上存在差异。

为此，CIE 通过将颜色投影到平面 $X+Y+Z=1$ 上，从而定义了一个二维的色度空间，如图 8.7 所示。该色度空间中的坐标称为 x 和 y ，其计算方法如下：

$$
\begin{aligned}
x &= \frac{X}{X+Y+Z}, \\
y &= \frac{Y}{X+Y+Z}, \\
z &= \frac{Z}{X+Y+Z} = 1-x-y.
\end{aligned}
\tag{8.2}
$$

这里的 z 值并不会提供额外的信息，因此通常会被省略。使用色度坐标（chromaticity coordinate） x 和 y 值绘制的图片，被称为 CIE 1931 色度图（chromaticity diagram），如图 8.8 所示。图中弯曲的黑色轮廓线代表了可见光谱颜色所覆盖的范围，连接光谱两端的直线称为紫色线（purple line）。图中的黑点代表了 D65 光源的色度，它是一种常用的白点（white point），常用于定义白色或者无色（achromatic）的色度值。

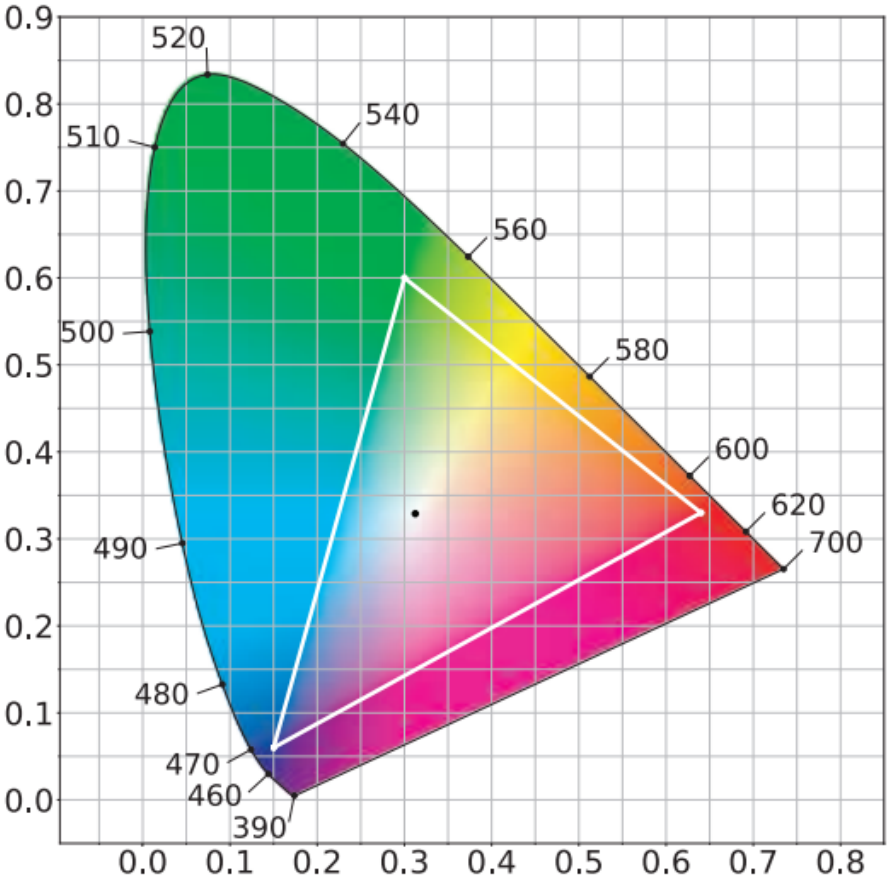


图 8.8：CIE 1931 色度图。图中的黑色曲线用相应的纯色波长进行了标记。图中的白色三角形代表了所覆盖的色域；黑点代表了白点，这个白点同样用于 sRGB 和 Rec. 709 色彩空间。

总而言之，我们从一个颜色匹配实验开始，使用三种单一波长的光源，并测量了每种光源需要多大的权重，才能与其他波长的光在视觉感知上相匹配。有些情况下可能会出现负数权重，此时这三种纯色光还必须被添加到待测光源中，才能相互匹配。这个实验给出了一组颜色匹配函数，将它们进行一定的线性组合，从而生成一个不包含负数权重的新颜色匹配函数。有了这个非负的颜色匹配函数集合，我们可以将任意 SPD 转换为一组定义颜色的色度和亮度的 XYZ 坐标值，还可以在保持亮度不变的前提下，进一步使用 xy 来描述颜色的色度值。

对于一个给定的颜色点 (x, y)，画一条从白点穿过该颜色点的线，并与区域边界（光谱线或者紫色线）相交。该颜色点与区域边界的相对距离，代表了颜色的激发纯度（excitation purity）；而区域边界上的交点，则定义了主导该颜色的主波长（dominant wavelength）。这些色度学中的术语在图形学中很少使用，相反，我们通常会使用饱和度（saturation）和色调（hue）这两个术语，其中饱和度与颜色的激发纯度有关，色调与颜色的主波长有关，但是需要注意的是，它们之间并不是强相

关的。你可以在 Stone [1706]以及其他人[456, 789, 1934]所撰写的书中，找到有关饱和度和色调的精确定义。

色度图仅仅描述了一个平面，它仅包含了颜色的色调信息，想要完整地描述一个颜色，还需要第三个维度 Y ，即亮度 luminance。这三个坐标在一起，定义了所谓的 xyY 坐标系。色度图对于理解颜色在渲染中的使用，以及理解渲染系统中的限制而言非常重要。电视或者计算机显示器会通过使用 R 、 G 、 B 颜色值来呈现颜色，每个颜色通道都会控制一个显示原色（display primary），这个原色会发出具有特定 SPD 的光线。三种原色都会按其各自的颜色值进行缩放，这些颜色值被叠加在一起，从而生成一个被观众所感知的单一 SPD。

色度图中的三角形代表了电视或者计算机显示器中的色域，这个三角形的三个顶点分别对应了三原色，即屏幕上所能显示的、饱和度最高的红色、绿色和蓝色。色度图的一个重要属性是，可以将这些限制颜色（三原色）用直线连接起来，从而展示整个显示系统的颜色限制。这里的直线代表了通过将这三种原色进行混合，所能够显示的颜色范围。色度图中的白点代表了显示系统在 R 、 G 、 B 颜色值相等时所产生的色度。你需要时刻注意，一个显示系统的完整色域实际上是一个三维空间，而色度图只展示了这个空间在二维平面上的投影。更多信息请参阅 Stone 撰写的书籍[1706]。

在图形渲染中有几个值得关注的 RGB 色域空间，每个空间都由 R 、 G 、 B 三原色和一个白点进行定义。为了对它们进行比较，我们将使用一种不同类型的色度图，它被称为 CIE 1976 UCS （uniform chromaticity scale）色度图。它是 CIELUV 颜色空间的一部分，CIE 将 CIELUV 颜色空间（以及另一个颜色空间 CIELAB）作为 XYZ 颜色空间的替代方案[1707]，其目的是提供更加均匀的视觉感知。在 CIE XYZ 颜色空间中，在视觉感知上具有相同差异的一对颜色，它们之间的距离可能会相差 20 倍；CIELUV 在此基础上进行了改进，将这个比例降低到最多 4 倍。这种增强的感知均匀性，使得 1976 年的色度图在比较 RGB 空间的色域方面，要比 1931 年的好得多。对感知均匀色彩空间的持续研究，最近诞生了 $\mathrm{IC_TC_P}$ [364]颜色空间和 $\mathrm{J_za_zb_z}$ 颜色空间[1527]，这些颜色空间在视觉感知上，要比 CIELUV 颜色空间更加均匀，尤其是对于现代高亮度和高饱和度的显示器而言。然而，基于这些颜色空间的色度图还没有被广泛采纳，因此我们将在本章节中继续使用 CIE 1976 UCS 色度图，如图 8.9 所示。

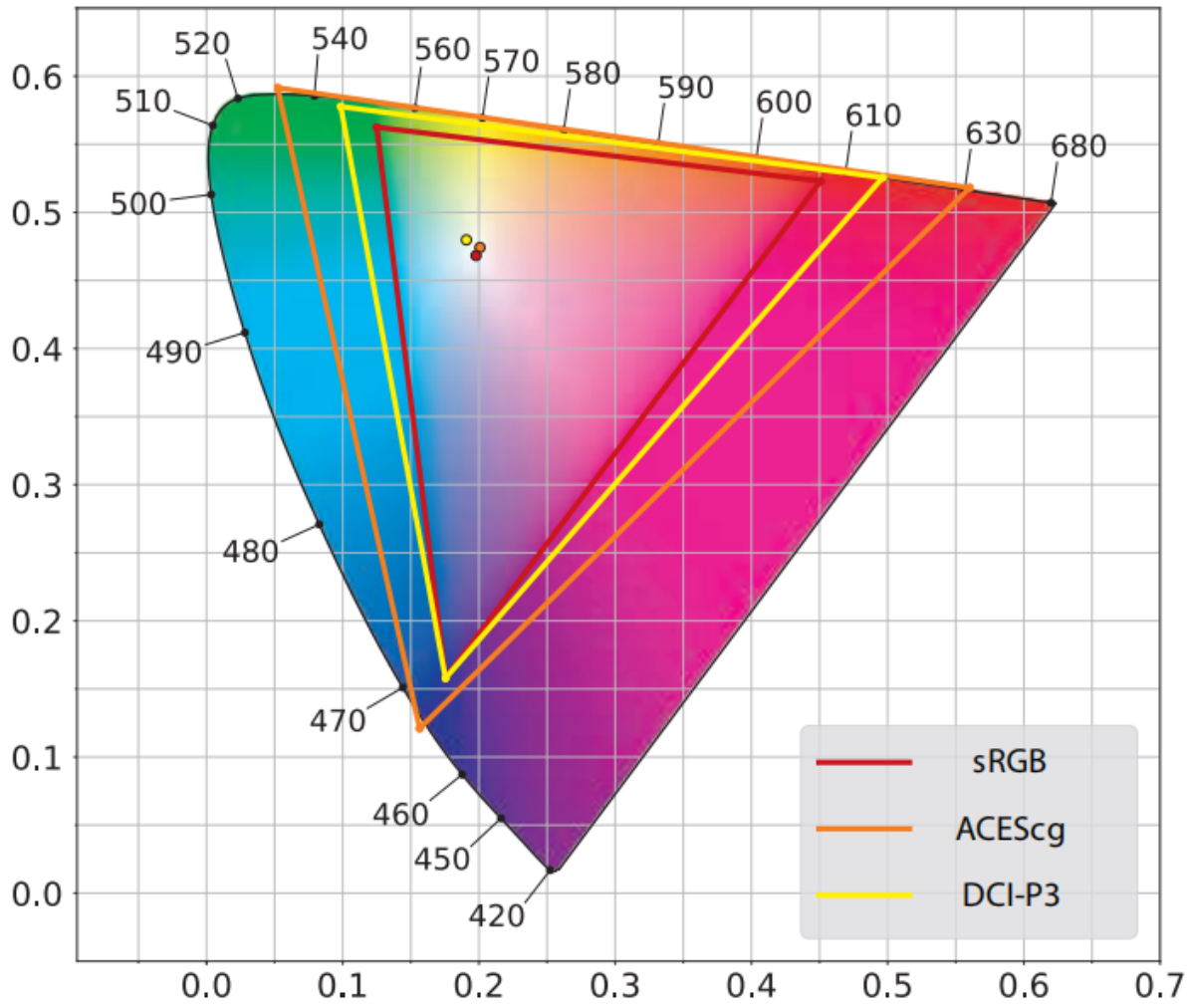


图 8.9：CIE 1976 UCS 色度图展示了三个 RGB 色彩空间，以及它们各自的白点，分别是：sRGB、DCI–P3 和 ACEScg。sRGB 色度图也可以用于 Rec. 709 颜色空间，因为它们具有相同的三原色和白点。

在图 8.9 所展示的三个 RGB 颜色空间中，sRGB 是目前在实时渲染中最常用的。需要注意的是，在本小节中，我们所使用的“sRGB 颜色空间”，指的是具有 sRGB 三原色和白点的线性颜色空间，而不是章节 5.6 中所讨论的非线性 sRGB 颜色编码。大多数计算机显示器都是为 sRGB 颜色空间所设计的，同样的三原色和白点也适用于 Rec. 709 颜色空间，这个颜色空间常用于高清电视显示器，因此它对主机游戏非常重要。然而，越来越多的显示器正在拥有更宽的色域，一些用于照片编辑的计算机显示器，常常会使用 Adobe 1998 色彩空间（未显示在图 8.9 中）。DCI–P3 色彩空间最初是为电影制作开发的，现在也被广泛使用，从 iPhone 到 Mac，苹果的产品线都采用了这种颜色空间，其他的设备制造商也在效仿这种做法。虽然超高清（ultra–high definition，UHD）内容和显示器，都会指定使用具有极宽色域的 Rec. 2020 来作为色彩空间，但是在许多情况下，DCI–P3 实际上也会被用作 UHD 的色彩空间。Rec. 2020 颜色空间并没有在图 8.9 中展示，但是它的色域图与第三个颜色空间

ACEScg 非常接近。ACEScg 色彩空间是由美国电影艺术与科学学院（Academy of Motion Picture Arts and Sciences，AMPAS）开发的，它专门用于剧情片电影（feature film）的计算机图形渲染。ACEScg 并不打算用作实际显示的颜色空间，而是专门用于在渲染中的工作颜色空间（working color space），在渲染完成之后，会将颜色转换到适当的显示颜色空间中。

虽然目前 sRGB 颜色空间在实时渲染中无处不在（ubiquitous），但是以后可能会逐渐使用具有更宽色域的颜色空间。对于那些针对高色域显示器开发的应用程序，使用更高色域的颜色空间可以增强颜色表现[672]，但是即使是使用 sRGB 或者 Rec. 709 颜色空间的显示器，在应用程序中使用更高色域的颜色空间也能带来相当的好处。当在不同的颜色空间中执行乘法等常规渲染操作时，会得到不同的结果[672, 1117]；有证据表明，在 DCI–P3 或者 ACEScg 等高色域颜色空间中执行这些操作，生成的结果要比在线性的 sRGB 颜色空间中更加准确[660, 975, 1118]。

从 RGB 空间到 XYZ 空间的转换是线性的，可以根据 RGB 空间的三原色和白点生成一个矩阵[1048]；通过矩阵求逆和相乘运算，派生出的矩阵可以从 XYZ 空间转换到任意的 RGB 空间，或者在两个不同的 RGB 空间之间进行转换。需要注意的是，转换后的 RGB 颜色值可能是一个负数，也有可能会大于 1。这些颜色值是位于颜色空间色域外的颜色，换句话说，这样的颜色在目标 RGB 空间中是不可能出现的；当然有各种方法可以将这些颜色映射到目标 RGB 色域中[785, 1241]。

一种常用的转换操作是将 RGB 颜色转换为灰度亮度值（grayscale luminance value），由于这个亮度值与系数 Y 实际上是相同的，因此这个操作只是将 RGB 颜色转换到 XYZ 的“Y 部分”。换句话说，它是 RGB 系数向量与 RGB–to–XYZ 转换矩阵中间一行之间的点乘。对于 sRGB 和 Rec. 709 颜色空间而言，这个灰度转换方程为[1704]：

$$Y = 0.2126R + 0.7152G + 0.0722B \tag{8.3}$$

方程 8.3 又让我们想起了图 8.4 中所展示的光度曲线（photometric curve），这个光度曲线代表了标准观察者的眼睛，对于不同波长光线的反应程度，将其与三种原色光的光谱功率分布相乘，再对相乘的结果在可见光波长范围内进行积分，最终生成的结果就是方程 8.3 中的三个权重值。之所以这个灰度强度值并不直接等于红、绿、蓝三原色的值，是因为人睛对于不同波长的光线具有不同的敏感程度。

色度学可以告诉我们两种颜色对于人眼的刺激程度是否匹配，但是无法预测它们表现出来的颜色外观。对于给定 XYZ 颜色的刺激条件，其视觉外观在很大程度上取决于

照明条件、周围的颜色以及人眼之前所接受到的颜色刺激等因素。颜色外观模型（color appearance model，CAM），例如 CIECAM02 试图解决这些问题，并对最终的感知到的颜色外观进行预测[456]。

颜色外观建模是更广泛的视觉感知领域中的一部分，其中还包括遮罩（masking）等效果[468]。在这种情况下，使用高频、高对比度的图案往往可以掩盖瑕疵（flaw）。换句话说，类似波斯地毯这样的纹理，将有助于掩饰色带以及其他着色瑕疵，这也意味着我们可以花费较少的渲染精力来处理这种表面。

8.1.4 使用 RGB 颜色进行渲染

严格来说，RGB 颜色值代表的是感知量，而不是真实的物理量，使用 RGB 颜色值来进行基于物理的渲染，在技术上来说是一个分类错误。正确的方法应当是在光谱物理量上执行所有的渲染计算，然后通过密集采样或者是投影到适当的基底上，并在最后将其转换为用于屏幕输出的 RGB 颜色值。

例如：渲染中最常见的操作之一，就是计算物体表面的反射光线。正如表面的光谱反射率曲线（spectral reflectance curve）所描述的那样，物体表面对于不同波长的光线，具有不同的反射率，对于某些特定波长的光线，表面对其的反射率要明显高于其他波长的光线。计算反射光线颜色的严格正确方法，应当是将入射光的 SPD 与各个波长的光谱反射率相乘，从而得到反射光的 SPD，然后再将其转换为 RGB 颜色。相反，在一个 RGB 渲染器中，会将光线和表面的 RGB 颜色相乘，从而得到反射光的 RGB 颜色。在一般情况下，这种做法并不会给出正确的结果；为了说明这一点，我们下面将会看到一个极端的例子，如图 8.10 所示。

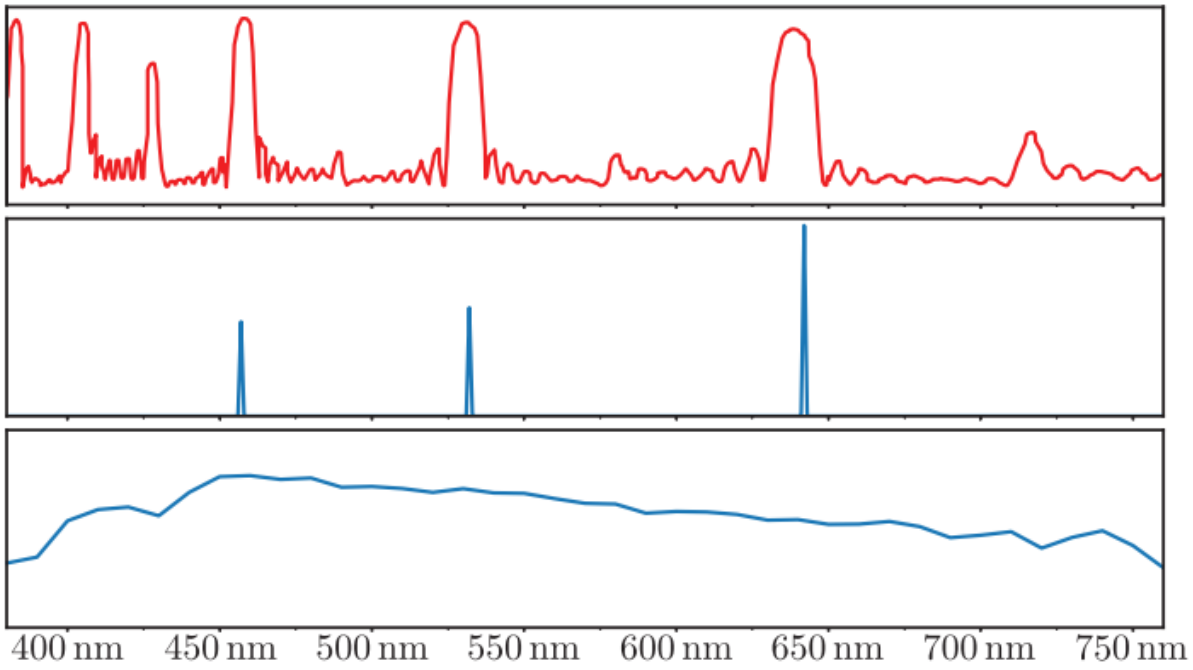


图 8.10：第一行图片展示了一种材料的光谱反射率，这种材料常用于制作投影屏幕。下面两个图展示了两种相同 RGB 颜色光源的光谱功率分布：第二行图片是 RGB 激光投影仪，第三行图片为标准 D65 光源。对于来自激光投影仪的光线，这个屏幕材料大约会反射 80%，因为它的反射峰与投影仪的原色光线相重合。但是对于 D65 光源发出的光线，这个屏幕材料只能反射不到 20%，因为该光源的大部分能量都位于屏幕的反射峰值之外。使用 RGB 颜色对这个光照情况进行渲染，这两个光源的反射光线将会具有相同的强度，这在物理上是不正确的。

图 8.10 展示了一种用于激光投影仪屏幕的材料，它在激光投影仪光线波长的窄带处，具有较高的反射率，而在其他大多数波长上的反射率都较低。这使得这个屏幕表面会反射投影仪所发出的大部分光线，同时会吸收来自其他光源的大部分光线。在这种情况下，RGB 渲染器将会产生严重的渲染错误。

但是好在图 8.10 所展示的情况只是一个极端例子，并非普遍情况。我们在实际中所遇到表面，其光谱反射率曲线要平滑得多，如图 8.11 所示。而现实生活中常见的光源 SPD 都类似于 D65 光源，激光投影仪的 SPD 只是极少数的情况。当光源的 SPD 与表面的光谱反射率都比较光滑的时候，RGB 渲染所引入的误差相对比较小。

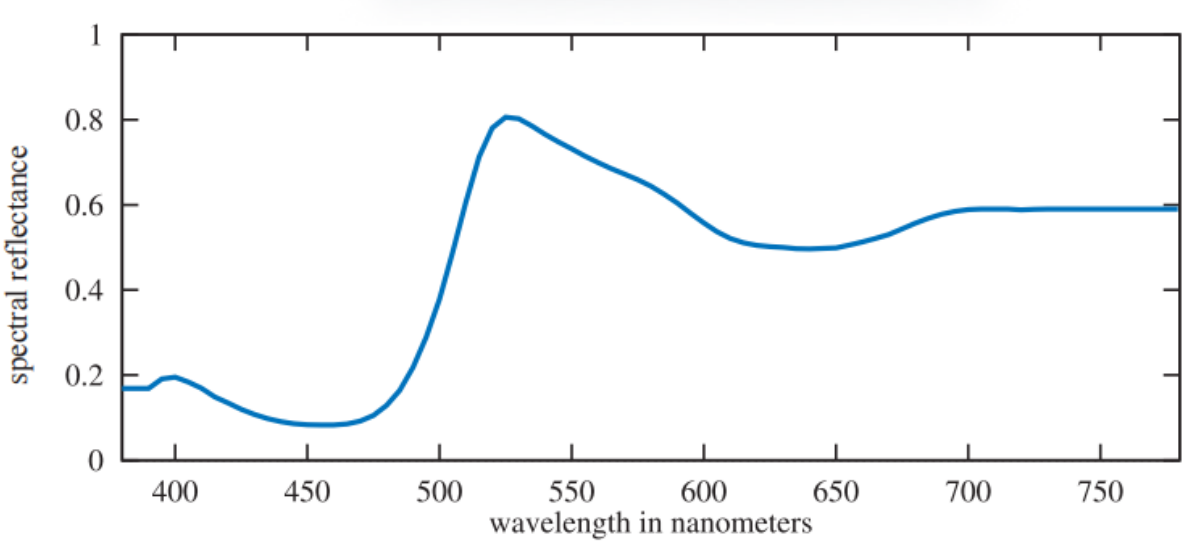


图 8.11：黄色香蕉的光谱反射率。[544]

在预测性渲染（predictive rendering）应用程序中，这些细微的错误可能会十分重要。例如：两个光谱反射率曲线在某个光源下可能会具有相同的颜色外观，但是在另一个光源下可能会具有完全不同的颜色外观。这个问题被称为同色异谱失效（metameric failure）或者光源异色（illuminant metamerism），当使用油漆来修复汽车车身表面的时候，这可能会成为一个十分严重的问题，因为当光照条件发生变化时，经过修复的汽车表面所反射出的颜色，可能会和原始颜色不同。对于试图预测这种类型效果的应用程序而言，并不适合使用 RGB 来进行渲染。

然而，对于大多数渲染系统而言，尤其是那些并不是用于预测模拟的交互式应用程序而言，RGB 渲染的效果得出奇的好[169]。即使是动画电影的离线渲染中，也只是在最近才开始使用光谱渲染，而且只是少数情况，大部分离线渲染仍然使用的是 RGB 渲染[660, 1610]。

本小节只介绍了一些颜色科学的基础知识，主要是想让读者意识到光谱与颜色三元组之间的关系，并简要讨论设备的局限性。我们将在下一小节中，讨论一个与之相关的话题，即场景的渲染颜色到显示颜色的转换过程。

8.2 从场景到屏幕

在接下来的几个章节中，我们将集中讨论基于物理的渲染问题。对于一个给定的虚拟场景，基于物理的渲染有着这样一个目标，即计算场景中可能存在的真实 radiance。然而计算完成之后，渲染的工作还远未完成，我们仍然需要确定最终的结果，即显示器帧缓冲中的像素值。在本小节中，我们将讨论与此有关的一些因素。

8.2.1 HDR 显示编码

章节 5.6 介绍了有关显示编码的内容，本小节的内容将建立在其基础上。之所以我们决定将高动态范围（high dynamic range，HDR）显示的内容推迟到本小节，因为想要理解这部分内容需要一定的背景知识，例如色域等概念，在本章之前，我们还没有对这些内容进行介绍和讨论。

章节 5.6 讨论了标准动态范围（standard dynamic range，SDR）显示器和电视机的显示编码，前者（显示器）通常会使用 sRGB 显示标准，后者（电视机）则会使用 Rec. 709 和 Rec. 1886 显示标准。这些标准都具有相同的 RGB 色域和白点（D65 光源），同时具有相似（但不完全相同）的非线性显示编码曲线。它们也具有大致相似的参考白色亮度水平（sRGB 为 80 $80\mathrm{cd}/\mathrm{m}^2$，Rec. 709/1886 为 $100\mathrm{cd}/\mathrm{m}^2$）。但是这些标准的亮度规格实际上并没有被显示器和电视制造商严格遵守，它们会倾向于制造更亮白色水平的显示器[1081]。

HDR 显示器通常会使用 Rec. 2020 和 Rec. 2100 标准。其中 Rec. 2020 定义了一个具有更宽色域的颜色空间，如图 8.12 所示，并且它与 Rec. 709 和 sRGB 色彩空间具有相同的白点（D65）。Rec. 2100 则定义了两种非线性显示编码函数：感知量化曲线（perceptual quantizer，PQ）[1213]和混合对数伽马曲线（hybrid log–gamma，HLG）。其中 HLG 编码在渲染领域中并不常用，因此我们这里将重点关注 PQ，它定义了一个 $10,000\mathrm{cd}/\mathrm{m}^2$ 的峰值亮度。

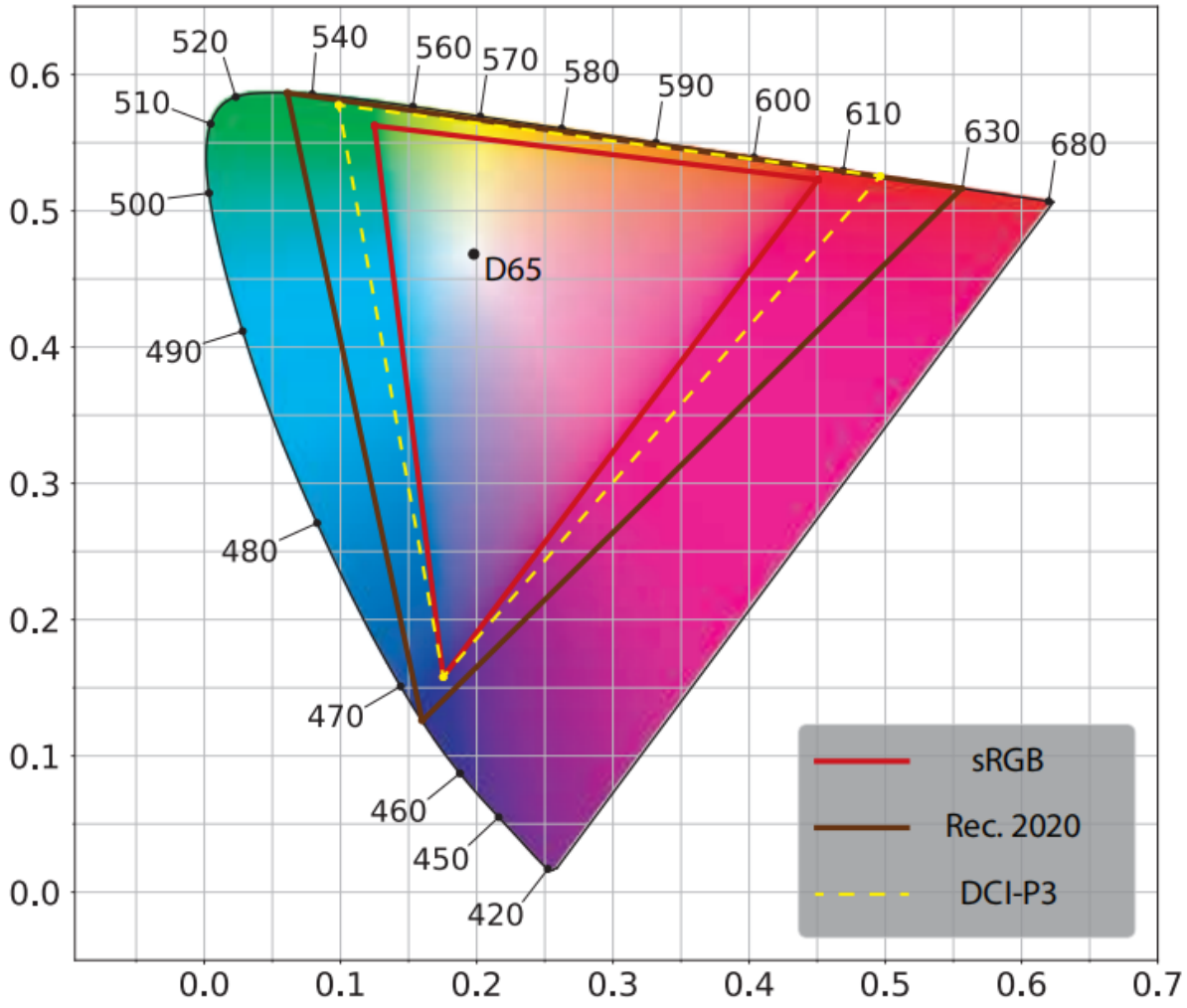


图 8.12：CIE 1976 UCS 色度图展示了 Rec. 2020 和 sRGB/Rec. 709 颜色空间的色域范围和白点（D65）。同时也显示了 DCI–P3 颜色空间的色域范围，以供和其他颜色空间进行比较。

虽然峰值亮度（peak luminance）和色域规格（gamut specification）对于编码而言十分重要，但是对于实际的显示设备而言，它们多少有些不切实际（aspirational）。在撰写本文的时候，很少会有消费级 HDR 显示器的峰值亮度水平超过 $1500\text{cd}/\text{m}^2$，实际的显示色域也要更加接近 DCI–P3 颜色空间（而不是 Rec. 2020），如图 8.12 所示。因此，HDR 显示器会在内部执行从标准规格到实际显示功能的色调映射（tone mapping）与色域映射（gamut mapping）。这个映射过程也可能会受到应用程序所传递过来的原始数据的影响，这些原始数据可能会指明实际的动态范围和色域范围[672, 1082]。

从应用程序端来看，有 3 条路径可以将图像传输到 HDR 显示器上。但是根据实际显示器和操作系统的不同，并不是所有的三种路径都是可用的。这三种路径分别是：

1. HDR10——被 PC 和主机操作系统的 HDR 显示器广泛支持。帧缓冲格式为每个像素 32 bit，其中 RGB 通道各自占据 10 个无符号整数 bit，alpha 通道占据 2

个 bit。它采用了 PQ 非线性编码和 Rec. 2020 颜色空间。每个 HDR10 显示模型都有各自的色调映射方案，没有被标准化或者文档化。

2. scRGB（线性变体）——仅支持 Windows 操作系统。名义上，它使用 sRGB 三原色和白点，但是这两者都可以被超越，因为这个标准支持小于 0 和大于 1 的 RGB 值。其帧缓冲的格式为每个 RGB 通道 16 bit，并存储线性的 RGB 值。它可以在任何的 HDR10 显示器上工作，因为驱动程序可以将其自动转换为 HDR10。scRGB 十分有用，因为它使用方便，同时向后兼容 sRGB 颜色空间。

3. 杜比视界（Dolby Vision）——一种专有格式，截止到本文撰写时，尚未在显示器或者任何主机上得到广泛支持。其帧缓冲使用了自定义的每通道 12 bit，并使用了 PQ 非线性编码和 Rec. 2020 颜色空间。显示器内部的色调映射是标准化的（但是并没有相关文档）。

Lottes [1083]指出，实际上还有第四种选择。如果仔细调整曝光和颜色，那么 HDR 显示器也可以通过常规的 SDR 信号路径进行驱动，并且能够取得良好的效果。

除了 scRGB 之外，作为显示编码步骤中的一部分，应用程序还需要将像素 RGB 值从渲染的工作空间中转换到 Rec. 2020，这个过程需要一个 $\mathbf{3} \times \mathbf{3}$ 的矩阵变换，并且需要应用 PQ 编码函数，这个 PQ 编码函数要比 Rec. 709 或者 sRGB 的编码函数复杂一些，计算成本也要更高[497]。Patry [1360]给出了 PQ 编码函数曲线的一个廉价近似。在 HDR 显示器上合成用户界面（user interface，UI）元素时需要特别注意，需要确保用户界面清晰可见（legible），并且具有一个舒适的亮度水平[672]。

8.2.2 色调映射

在章节 5.6 和章节 8.2.1 中，我们讨论了显示编码的话题，即将线性的 radiance 值，转换为用于显示设备的非线性编码值的过程。显示编码所使用的函数是显示器光电转换函数（electrical optical transfer function，EOTF）的逆函数，它确保了输入的线性值与显示器发出的线性 radiance 相匹配。在我们前面的讨论中，忽略了发生在渲染和显示编码之间的一个重要步骤，下面我们将对这个步骤进行介绍和讨论。

这个步骤叫做色调映射（tone mapping）或者色调再现（tone reproduction），它是指将场景的 radiance 转换为显示器 radiance 的过程。在这个步骤中所应用的转换函数称为端到端转换函数（end-to-end transfer function）或者场景到屏幕转换函数（scene-to-screen transform）。图像状态（image state）的概念是理解色调映射的关键[1602]，有两种基本的图像状态：场景参考（scene-referred）图像是根据场景中的 radiance 进行定义的，显示参考（display-referred）图像是根据显示器

的 radiance 进行定义的。图像状态与编码无关，在这两种状态下的图像，都可以进行线性编码或者非线性编码。图 8.13 展示了图像状态、色调映射和显示编码是如何在成像管线中进行组合的，这个成像管线用于将最初渲染生成的颜色值，转换为最终用于显示的颜色值。

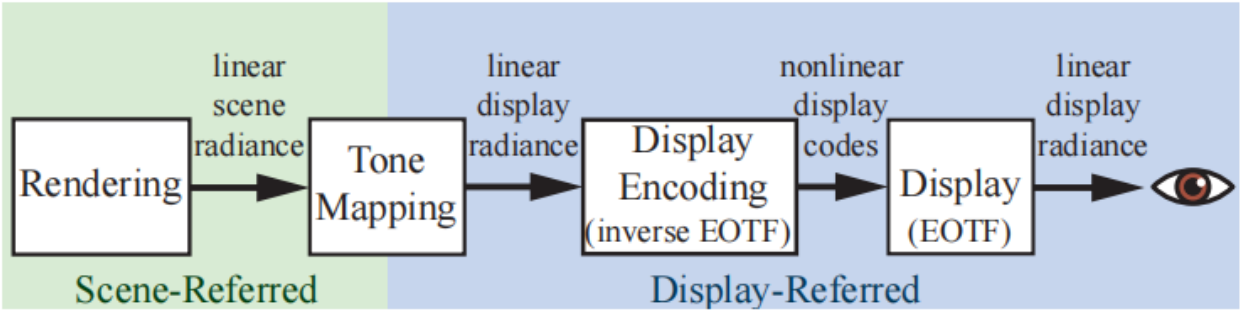


图 8.13：用于合成（渲染）图像的成像管线。我们首先会渲染线性的场景参考 radiance，然后使用色调映射，将其转换为线性的显示参考值。显示编码使用逆 EOTF，将线性的显示值转换为非线性的编码值，然后再将这些编码值传递给显示器。最后，显示器使用 EOTF 将非线性的显示值，转换为从屏幕发射出的线性 radiance，最终进入人的眼睛。

关于色调映射的目标，有几个常见的误解。首先，它不能确保从场景到屏幕的变换是一个恒等变换，不一定能够在显示器上完美再现场景的 radiance。其次，它也不是要把场景高动态范围中的每一个 bit 信息，都“塞入”到低动态范围的显示器中，尽管考虑场景渲染和显示器之间的动态范围差异确实具有十分重要的作用。

为了更好地理解色调映射的目的，最好将其视为图像再现（image reproduction）的一个实例[757]。图像再现的目标是创建一个显示参考图像，在给定显示属性和观看条件的情况下，尽可能地再现观看者在观察原始场景时所产生的感知印象，如图 8.14 所示。

图 8.14：图像再现的目标是确保由再现图像（右）所引起的感知印象，尽可能地和原始场景（左图）所引起的感知印象相接近。

有一种类型的图像再现，具有略微不同的目标，它叫做择优图像再现（preferred image reproduction）。它旨在创建一个比原始场景看起来更好的显示参考图像，我们将在章节 8.2.3 中讨论这个话题。

在实际的渲染场景中，其亮度范围可能要比显示器的显示能力高出几个数量级，因此再现与原始场景相似的感知印象，是一个具有挑战性的目标。场景中一些颜色的饱和度（纯度）也可能会远远超出显示器的能力。尽管如此，正如文艺复兴时期的画家一样，摄影、电视和电影确实在设法生成令人信服的、与原始场景相似的感知印象。通过利用人类视觉系统的某些特性，是可以实现这个目标的。

人类的视觉系统会对绝对亮度的差异进行补偿，这种能力被称为适应性（adaptation）。由于这种能力的存在，因此在一个昏暗的房间里，屏幕上再现的室外场景可以产生与原始场景相似的感知体验，尽管此时再现场景的亮度（屏幕的亮度）可能还不到原始场景的 1%。但是，这种适应性所提供的补偿并不是完美的，在较低的亮度水平下，人眼对于画面对比度的感知能力会有所下降（Stevens 效应），所感知到的“色彩度”也会下降（Hunt 效应）。

还有其他一些因素会影响再现图像的实际对比度或者感知对比度。显示器周围的环境（显示器屏幕外的亮度水平，例如房间照明的明亮程度）可能增强或者减弱感知到的对比度（Bartleson–Breneman 效应）。显示耀斑（display flare）是指通过显示器缺陷或者屏幕反射而叠加到显示器图像上的多余光线，会使得屏幕中的部分区域变得异常明亮，这种现象会在相当大的程度上降低图像的实际对比度。由于这些效应和现象的存在，如果我们想要保持与原始场景相似的感知效果的话，那么就必须提高显示参考图像的对比度和饱和度[1418]。

然而，这种对比度的增加，会加剧一个已经存在的问题。由于场景的动态范围通常要比显示器的动态范围大得多，因此我们必须选择一个狭窄的亮度值窗口来显示图像，而位于这个亮度值窗口两侧的值，会被裁剪为黑色或者白色。提高对比度会进一步缩小这个亮度值窗口的宽度。为了部分抵消暗值（小于亮度值窗口）和亮值（大于亮度值窗口）的裁剪，可以让这个亮度值窗口进行柔和的滚动，从而带回一些阴影和高光细节。

上述的这些因素会导致一个 S 型（sigmoid）的色调再现曲线，这与光化学胶片（photochemical film）[1418]所提供的曲线类似。这并非是一个偶然现象，柯达和其他公司的研究人员，对光化学胶片乳液的性质进行了仔细的调整，从而生成了有效且令人满意的图像再现。由于这些原因，“filmic”这个形容词经常会出现在有关色调映射的讨论中。

曝光（exposure）是一个色调映射中至关重要的概念。在摄影中，曝光是指对照射在胶片或者传感器上的光量进行控制。然而在渲染中，曝光是指在应用色调再现变换（tone reproduction transform）之前，对场景参考图像进行的线性缩放操作。曝光的棘手之处在于，很难确定究竟使用多大的缩放因子。色调再现变换和曝光是紧密联系在一起的，色调变换的对象，通常就是以某种方式进行曝光的场景参考图像。

通过曝光对亮度进行缩放，然后再应用色调再现变换，这个过程是一种全局色调映射（global tone mapping），会对图像中所有的像素，都应用相同的映射操作。相比之下，局部色调映射（local tone mapping）过程会基于周围像素以及其他因素，在像素与像素之间使用不同的映射方式。在实时应用程序中，几乎只会使用全局色调映射（除了少数例外[1921]），因此我们将重点讨论这种映射类型。首先我们会讨论色调再现变换，然后再讨论曝光。

需要时刻牢记的是，场景参考图像与显示参考图像在根本上是不同的。相关物理操作只有在场景参考图像上执行时才是有效的。由于显示器的限制，以及上文中我们讨论

过的各种感知效应，因此在这两个图像状态之间，总是需要一个非线性变换来进行转换。

色调再现变换

色调再现变换（tone reproduction transform）通常会被表示为一维曲线，它将场景参考的输入值，映射到显示参考的输出值。这些曲线可以独立应用于 R、G、B 值或者亮度值。对于 RGB 颜色值，转换后的结果将自动处于显示器的色域中，因为每个显示参考的 RGB 通道值都将介于 0–1 之间。然而，在 RGB 通道上执行非线性操作（尤其是裁剪），可能会导致饱和度和色相（hue）发生变化，而不是我们想要的亮度变化。Giorgianni 和 Madden 指出[537]，这种饱和度上的变化对于感知是有益的。大多数色调再现变换为了抵消 Stevens 效应（以及显示器环境效应和耀斑效应），会增强对比度，这也会使得饱和度被相应提高，从而抵消 Hunt 效应。然而，色相上的变化通常被认为是不受欢迎的，因此现代的色调变换试图在应用色调曲线之后，通过添加额外的颜色调整来减少色调上的变化。

通过将色调曲线应用于亮度值，可以避免色相和饱和度的变化（或者至少减少这种变化）。然而，最终生成的显示参考颜色，可能会超出显示器的 RGB 色域，在这种情况下，还需要将其映射回去。

色调映射的一个潜在问题是，将非线性函数应用于场景参考的像素颜色，可能会导致某些抗锯齿技术出现问题，有关这个问题（以及相应的解决方法）的内容，在章节 5.4.2 中进行了讨论。

Reinhard 色调再现算子[1478]是早期用于实时渲染中的色调变换之一。在这个映射方法中，较暗的值基本保持不变，而较亮的值则会渐变为白色。Drago 等人[375]提出了一种类似的色调映射算子，它具有调整输出显示亮度的能力，更加适合 HDR 显示器。Duiker 创建了一种近似于柯达胶片的响应曲线[391, 392]，并将其应用在了电子游戏中；这条曲线后来被 Hable [628]进一步修改，添加了更多的用户控制选项，并应用在了游戏《神秘海域 2》中。Hable 关于该曲线的演讲十分具有影响力，并导致许多游戏都使用了这个“Hable 电影曲线（Hable filmic curve）”。Hable [634]后来又提出了一种新的曲线，和他早期的工作相比，这个新曲线具有许多优点。

Day [330]提出了一种 S 形的色调映射曲线，它被 Insomniac Games 所采用，并应用在了《使命召唤：高级战争》中。Goanda [571, 572]提出一些了色调变换曲线，用于模拟胶片和数码相机传感器的响应曲线，《星之海洋 4》和其他一些游戏都采用了这种方法。Lottes [1081]指出，显示耀斑对于显示器的有效动态范围具有十分显著的影响，并且高度依赖于室内的照明条件。由于这个原因，向用户提供色调映射的调

整选项是很重要的。他提出了一种支持这种调整选项的色调再现变换方法，它可以同时使用在 SDR 显示器以及 HDR 显示器上。

学院色彩编码系统（Academy Color Encoding System，ACES）是由美国电影艺术与科学学院（AMPAS）下属的科学技术委员会提出的，作为电影和电视行业色彩管理的建议标准。ACES 系统将场景到屏幕的转换过程分为两个部分。第一部分是参考渲染转换（reference rendering transform，RRT），它将场景参考值转换为一个与设备无关的、标准输出空间中的显示参考值，这个输出空间叫做输出颜色编码规范（output color encoding specification，OCES）。第二部分是输出设备转换（output device transform，ODT），它将来自 OCES 的颜色值，转换为最终的显示器编码。有许多不同的 ODT，每一个都是为特定的显示设备和观察条件而设计的。RRT 与适当的 ODT 组合在一起，便构成了整个转换过程，这种模块化的结构便于处理各种显示器类型和观察条件。对于那些需要同时支持 SDR 显示器和 HDR 显示器的应用程序，Hart [672]建议使用 ACES 色调映射变换。

虽然 ACES 最初是为电影和电视设计的，但它的转换过程在实时渲染中的应用越来越多。虚幻引擎[1802]默认启用了 ACES 色调映射，Unity 引擎[1801]也同样支持这一功能。对于 ACES RRT 和 SDR、HDR 的 ODT 转换过程，Narkowicz [1260, 1261] 和 Patry [1359]都给出了廉价的拟合曲线。Hart [672]提出了一种 ACES ODT 的参数化版本，它可以支持许多显示设备。

HDR 显示器的色调映射需要一些额外注意，因为许多显示器也会应用一些自己的色调映射。Fry [497]介绍了一组在寒霜引擎中所使用的色调映射变换方法，它们在 SDR 显示器中使用了相对激进的色调再现曲线；对于使用 HDR10 信号路径（根据显示器的峰值亮度有一些变化）的显示器，则使用了不那么激进的色调再现曲线；对于使用杜比视界（Dolby Vision）路径的显示器，则没有应用任何色调映射，也就是说，它们依赖于显示器内置的杜比视界色调映射。寒霜引擎中色调再现变换的设计是中性的，即没有显著的对比度变化或者色相变化；这一设计的目的是，可以通过应用颜色分级（color grading，详见章节 8.2.3），来获得任何想要的对比度或者色相修改。为此，应当在 $\mathrm{IC_TC_P}$ 颜色空间中进行色调再现变换[364]，$\mathrm{IC_TC_P}$ 颜色空间中的色度轴（chrominance）和亮度轴（luminance）是正交的，而且是感知均匀的。寒霜引擎中的色调映射会随着亮度的下降，逐渐降低色度的饱和度，直到显示为白色，这种方案提供了一个没有色相变化的干净变换。

具有讽刺意味的是（Ironically），随着越来越多的资源（例如火焰效果）在早期的变换中利用了色相变化，寒霜引擎最终对变换过程进行了修改，使得用户能够在显示参

考颜色中重新引入一定程度的色相变换。图 8.15 展示了寒霜引擎的变换，以及与本节中提到的其他几种变换的对比。

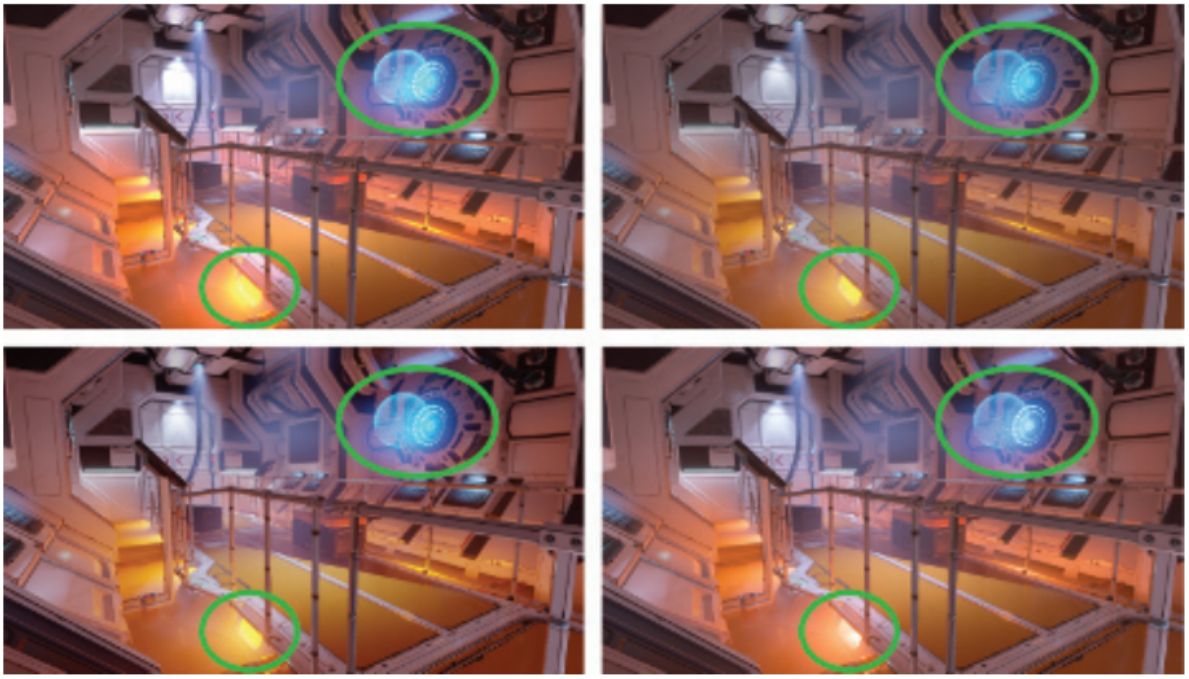


图 8.15：同一个场景使用了四种不同的色调变换。这些结果的差异主要体现在图中的绿色圆圈区域，因为那里的场景像素值特别高。左上角：直接裁剪（同时使用了 sRGB OETF）；右上角：Reinhard 方法[1478]；左下角：Duiker 方法[392]；右下角：寒霜方法（保持色相的版本）[497]。Reinhard，Duiker 和寒霜的变换，都可以保留因裁剪而丢失的高光信息。然而，Reinhard 曲线倾向于降低图像中较暗部分的饱和度[628, 629]，而 Duiker 变换则增加了较暗区域的饱和度，后者有时会被认为是更好的特性[630]。通过一些特殊设计，寒霜变换保持了饱和度和色相，避免了强烈的色相偏移，请仔细观察其他三张图片左下角的圆圈区域。

曝光

计算曝光（exposure）的常用技术依赖于对场景参考的亮度值进行分析。为了避免出现停顿，这种分析通常是对前一帧进行采样来完成。

根据 Reinhard 等人[1478]的建议，早期实现中使用的一个指标是场景亮度值的对数平均（log-average scene luminance），通常曝光是通过计算前一帧的对数平均值来确定的[224, 1674]。通过执行一系列的下采样后处理 pass，来计算这个对数平均值，直到最后仅剩下单个值为止。

我们知道，平均值往往对某些异常值过于敏感，例如：少量过于明亮的像素可能会影响整个图像的曝光，后续的实现方法通过使用亮度直方图来改善这个问题。直方图可以允许我们计算中位数，这要比平均值更加健壮稳定，不会受到少数异常值的影响。直方图中的额外数据点，还可以用于对结果进行进一步的改进，例如：在 Valve 的

《The Orange Box》中，使用了基于 95% 和中位数的启发式方法来确定曝光[1821]。Mittring 描述了如何使用计算着色器来生成亮度直方图[1229]。

到目前为止我们所讨论的曝光技术，它们都是使用像素亮度来确定曝光的，这并不是物理正确的。如果我们看看摄影实践，例如 Ansel Adams 的区域系统[10]；以及如何使用入射光的物理量来确定曝光，就可以清楚地看到，最好是单独使用光照（不受表面反照率的影响）来确定曝光[757]。这样做之所以有效，简单来说就是摄影中的曝光是用来抵消光照的，这样生成的印刷品（例如照片）主要显示的才是物体真正的表面颜色，这与人类视觉系统的颜色恒定性（color constancy）相符合。以这种方式来处理曝光，还可以确保将正确的值传递给色调变换函数，例如：电影或者电视行业中所使用的大多数色调变换，旨在将曝光后的场景参考值 0.18 映射到显示参考值 0.1，这里的 0.18 代表了主场景照明中的 18% 灰卡（译者注：中性灰度测试卡，具有 18% 的反射率）[1418, 1602]。

虽然这种曝光方法在实时应用程序中还不常见，但是它已经开始被采用了，例如：游戏《合金装备 V：原爆点》就有一个基于光照强度的曝光系统[921]。在许多游戏中，曝光水平是静态的，美术人员会根据已知的场景照明值，来为环境的不同部分手动设置曝光水平，这样做可以避免曝光水平发生意外的动态变化。

8.2.3 颜色分级

在章节 8.2.2 中，我们提到了择优图像再现（preferred image reproduction）的概念，即生成在某种意义上看起来比原始场景更好的图像。它通常会涉及到对图像颜色的创造性处理，这个过程被称为颜色分级（color grading），或者调色、校色等，它们的含义实际上都是一样的。

数字颜色分级已经在电影工业中使用了一段时间，早期的例子包括电影《逃狱三王（O Brother, Where Art Thou? ）》（2000）和《天使爱美丽（Amelie）》（2001）。颜色分级通常是通过交互式的操作，来对场景图像的颜色进行调整，直到实现想要的创意“外观”，然后再将相同的操作序列，重新应用到一个镜头或者一个序列中的所有图像上。颜色分级技术从电影传播到游戏领域中，现在在游戏中被广泛应用[392, 424, 756, 856, 1222]。

Selan [1601]展示了如何将来自一个颜色分级，或者图像编辑应用的任意颜色转换，“烘焙”到一个三维颜色查找表（LUT）中。通过将 RGB 颜色值作为 xyz 坐标输入，来从这个表中快速查找对应的新颜色；这种方式可以用于从颜色到颜色的任何映射，不过这个过程会受到 LUT 分辨率的限制。Selan 的烘焙过程从一个相同 LUT 开始，这个 LUT 会将每个输入的颜色，映射到相同的输出颜色，然后再将其“切片”从

而创建一个二维图像。然后将这个切片的 LUT 图像加载到一个颜色分级应用程序中，并对其应用定义目标创意外观所需要的操作。需要注意的是，只能对 LUT 应用颜色操作，需要避免模糊等空间操作。然后将编辑好的 LUT 保存下来，“打包”到一个三维 GPU 纹理中，并在渲染的过程中进行使用，从而动态地对渲染像素应用相同的颜色转换。Iwanicki [806]提出了一种聪明的方法，当在 LUT 中存储颜色变换的时候，可以使用最小化的最小二乘法，来减少采样误差。

在后来的出版物中，Selan [806]对两种执行颜色分级的方法进行了区分。第一种方法是对显示参考的图像数据进行颜色分级操作。第二种方法通过显示变换，对结果进行预览，然后再对场景参考的数据进行颜色分级操作。虽然显示参考的颜色分级方法更加容易操作和实现，但是对场景参考的数据进行颜色分级操作，可以生成更高保真度（fidelity）的结果。

当实时应用程序首次使用颜色分级技术时，显示参考方法占据了主导地位[756, 856]。然而，场景参考方法由于其更高的视觉质量，而获得了更多的关注[198, 497, 672]，如图 8.16 所示。对场景参考的图像数据应用颜色分级操作，还可以将色调映射曲线烘焙到颜色分级 LUT 中[672]，从而节省一些计算量，就像在游戏《神秘海域 4》中所做的那样[198]。

图 8.16：游戏《神秘海域 4》中的场景画面。最上面的截图没有使用颜色分级，下面两个截图各自应用了一个颜色分级操作。为了便于说明，我们选择了一个极端的颜色分级操作（将原始图像乘以一个高饱和度的青色）。在左下角的截图中，将颜色分级应用在了显示参考图像上（即色调映射之后）；而在右下角的截图中，将颜色分级应用在了场景参考图像上（即色调映射之前）。

在进行 LUT 查找之前，还必须将场景参考的数据重新映射到 $[0, 1]$ 范围中[1601]。在寒霜引擎[497]中，使用了感知量化曲线 OETF 来实现这个目的，尽管可以使用一些更加简单的曲线。Duiker [392]使用了一个对数曲线，而 Hable [635]则建议使用一次或者两次的平方根运算来实现。

Hable [635]对常见的颜色分级操作，以及实现中的注意事项进行了很好的综述。

补充阅读和资源

Wyszecki 和 Stiles [1934]所撰写的《Color Science》，是色度学和颜色科学领域的“圣经”。其他一些较好的色度学参考资料包括：Hunt [789]撰写的《Measuring Colour》和 Fairchild [456]撰写的《Color Appearance Models》。

Selan 的白皮书[1602]很好地概述了图像再现和“从场景到屏幕”的问题。如果读者想要进一步了解这个主题，可以参考 Hunt [788]撰写的《Reproduction of Colour》，以及 Giorgianni 和 Madden [537]撰写的《Digital Color Management》。《Ansel Adams Photography Series》中有三本书[9, 10, 11]，特别是《The Negative》，详细阐释了电影摄影中的艺术和科学，是如何影响到如今图像再现的理论和实践的。最后，Reinhard 和其他人[1480]共同撰写的《Color Imaging: Fundamentals and Applications》一书，对整个研究领域进行了全面的概述和讨论。

Chapter 9 Physically Based Shading 基于物理的着色

> John Constable ——“There is nothing ugly; I never saw an ugly thing in my life: for let the form of an object be what it may, — light, shade, and perspective will always make it beautiful.”

> 约翰·康斯太勃尔——“没有什么是丑陋的；我一生中从未见过丑陋的东西，因为不管一个物体的形状如何，光、影和透视总是会使它变得美丽。”（英国风景画家；1776—1837）

在本章节中，我们将从各个方面介绍有关基于物理的着色。首先我们会在章节 9.1 中介绍光与物质交互的物理原理；在章节 9.2–9.4 中，我们将展示这些物理原理是如何转换到着色过程中的；章节 9.5–9.7 介绍了一些构建模块，它们专门用于基于物理的着色模型；章节 9.8–9.12 介绍了包含一系列材质类型的着色模型；最后，在章节 9.13 中，我们描述了各种材质是如何混合在一起的，并介绍了一些避免瑕疵、保持材质外观的过滤方法。

9.1 光的物理学

光与物质的相互作用，是基于物理的着色的理论基础，理解这些相互作用的原理，有助于帮助我们建立一个对光本质的基础理解。

在物理光学（physical optic）中，光被认为是一种电磁横波（electromagnetic transverse wave），它使得电场（electric field）和磁场（magnetic field）在其传播方向的垂直面上来回振荡。电场和磁场的振荡是耦合的，二者的矢量相互垂直，并且长度之比也是固定的，这个比值等于相位速度（phase velocity），我们将在稍后进行讨论。

图 9.1 展示了一个简单的光波，事实上，这是最简单的一种，即一个完美的正弦函数。这个波只有一个波长（wavelength），使用希腊字母 λ 进行表示。在章节 8.1 中我们提到，人眼所感知到的颜色，与光线的波长密切相关，因此，具有单一波长的光

被称为单色光（monochromatic light）。然而在实践中所遇到的大多数光都是多色光（polychromatic light），即它们包含了许多不同的波长。

之所以我们说图 9.1 所展示的光波是简单的，是因为它是线性偏振（linearly polarized）的。这意味着对于空间中的一个不动点，电场和磁场都沿着一条直线来回运动；相比之下，在这本书中，我们所关注的是更加普遍的非偏振光（unpolarized light）。在非偏振光中，电场和磁场会在垂直于传播方向的所有方向上均匀分布。尽管这种单色的、线性偏振的光波是很简单的，但是理解它们的行为仍然十分有用，因为任何光波都可以被分解成这种简单波的组合。

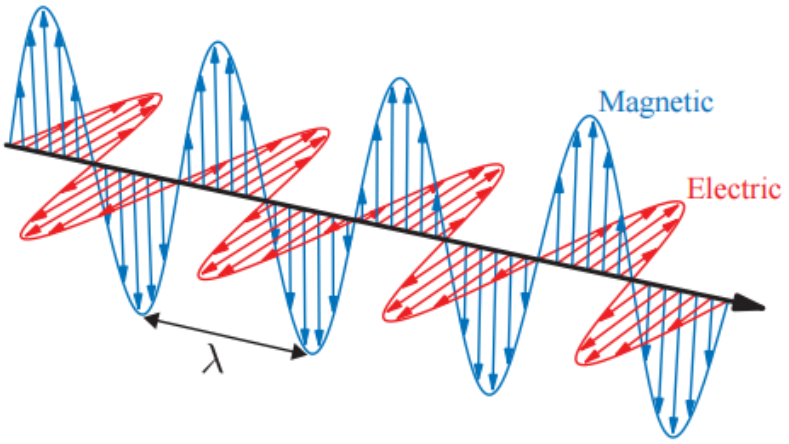


图 9.1：光是一种电磁横波。电场和磁场的矢量垂直于光线的传播方向，并以 90 度夹角相互振荡。本图中所展示的是最简单的光波，它是单色光（仅有一个固定的波长 λ），并且是线性偏振的（电场和磁场各自沿着一条直线进行振荡）。

如果我们以一个给定的相位（例如偏振峰值），对波上的一个点进行追踪，我们将看到它会以一个恒定的速度在空间中进行移动，这就是波的相位速度（phase velocity）。对于真空中的光波而言，其相位速度为 c，通常被称为光速，大约是每秒 30 万公里。

在章节 8.1.1 中，我们讨论了可见光的波长分布，单个波长的大小大约在 400 纳米–700 纳米，这个长度大约是一根蜘蛛丝的一半到三分之一左右，而一根蜘蛛丝还不到人类头发直径的五分之一，如图 9.2 所示。在光学中，通常都会使用相对于波长的尺度来描述尺寸大小，例如：我们会说蜘蛛丝的宽度大约为 $2\lambda - 3\lambda$（即 2–3 倍的波长），头发的宽度为 $100\lambda - 200\lambda$。

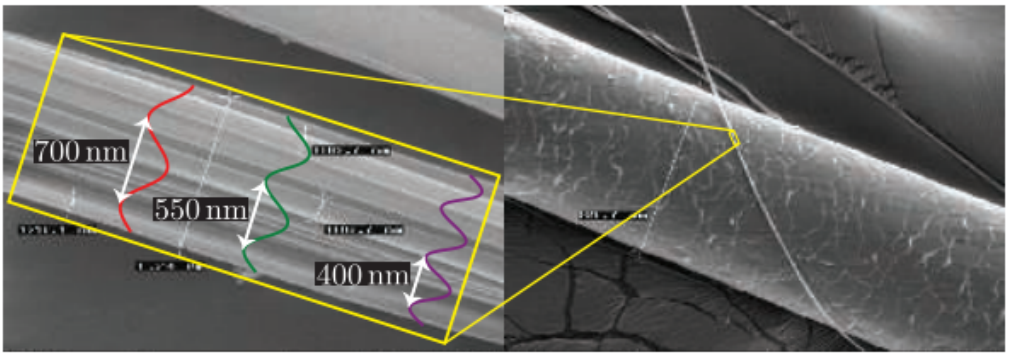


图 9.2：左图展示了不同波长的可见光与一根蜘蛛丝，蜘蛛丝的长度略微超过 1 微米。右图是一根蜘蛛丝和一根人类头发的直径对比。

光波会携带能量，这个能量流的强度等于电场强度和磁场强度的乘积，由于电场和磁场交替振荡，二者强度是成比例的，因此光线能量流的强度与电场强度的平方成正比。这里我们会关注电场，这是因为电场对物质的影响要比磁场大得多。在渲染中，我们关注的是随着时间变化的平均能量流，它与振幅的（amplitude）平方成正比，这个平均能量流的强度就是 irradiance，通常使用字母 E 来进行表示。我们在章节 8.1.1 中介绍了 irradiance 和其他光量的关系。

光波会以线性的方式进行叠加，总体波是各个分量波之和。然而，由于 irradiance 和振幅的平方成正比，这似乎导致了一个悖论，例如：将两个相同的光波进行叠加，会不会导致 irradiance“1+1=4”的情况？由于 irradiance 描述的是能量流，这会不会违反能量守恒能量？对于这两个问题，前者的答案是“有时会”，而后者的答案则是“不会”。

为了对刚才的两个问题进行说明，我们来看看以下这个简单的例子：现在有 n 个单色光要进行叠加，除了相位之外，其他的属性完全相同。这 n 个波的振幅均为 a，使用 E_1 来表示每个波的 irradiance，E_1 和 a^2 成正比，也就是说，$E_1 = ka^2$，其中 k 是常数。

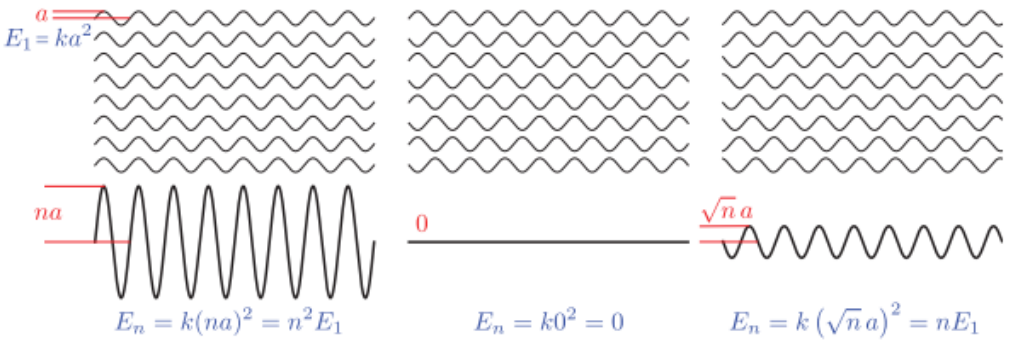


图 9.3：n 个具有相同频率、相同偏振和相同振幅的单色光，它们以三种不同的方式进行叠加。从左到右分别是：相长干涉，相消干涉和不相干叠加。

每种情况的下面都展示了对应组合波的振幅和 irradiance。

图 9.3 展示了三种不同的叠加情况。左图中，所有的波都具有相同的相位，叠加之后会相互加强，其 irradiance 是单个波的 n^2 倍，是 n 个独立波 irradiance 之和的 n 倍，这种情况被称为相长干涉（constructive interference）。在中间的图中，每一对波都处于相反的相位，两两相互抵消，叠加之后的振幅和 irradiance 均为 0，这种情况被称为相消干涉（destructive interference）。

相长干涉和相消干涉是两种特殊的相干叠加（coherent addition），即波峰和波谷以一种一致的方式进行排列。根据相对相位关系的不同，n 个相同波相干叠加生成的组合波，其 irradiance 为单个波的 $0 - n^2$ 倍。

然而通常来说，波都是以不相干的的方式进行叠加的（incoherent addition），这意味着它们的相位都是相对随机的，如图 9.3 右侧所示。在这种情况下，组合波的振幅为 $\sqrt{n}a$，其 irradiance 等于 n 个独立波的 irradiance 线性叠加，这正如我们所期望的那样。

看起来相长干涉和相消干涉违反了能量守恒定律，但实际上并没有。这是因为图 9.3 并没有展示完整的情况——它仅仅展示了波在一个位置上的相互作用。当波在空间中进行传播的时候，它们之间的相位关系会发生变化，如图 9.4 所示。在某些位置上会发生相长干涉，组合波的 irradiance 会大于单个波 irradiance 之和；而在某些位置上则会发生相消干涉，组合波的 irradiance 会小于单个波 irradiance 之和。这并不违反能量守恒定律，因为通过相长干涉而获得的能量，与通过相消干涉而损失的能量是相等的。

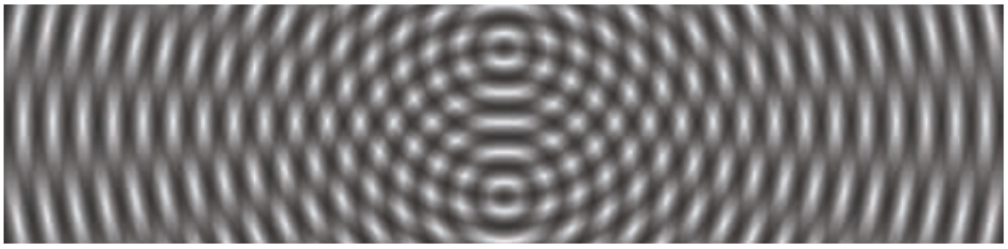


图 9.4：图中展示了两个单色光，它们以相同的频率从两个点源进行传播。两个波在空间的不同位置上，会发生相长干涉和相消干涉。

当物体中的电荷发生振荡时，就会发出电磁波，而引起振荡的部分能量（热能、电能、化学能）会被转化为光能，然后从物体上辐射出去。在渲染中，这样的物体会被视为光源，我们在章节 5.2 中首次讨论了光源，在第 10 章中，它们将从一个更加基于物理的角度，来对其进行描述。

当光波被发射出来之后，它们穿过空间，直到遇到一些可以相互作用的物质。大多数光与物质交互作用，其核心现象都很简单，与上面所讨论的发射情况非常类似。振荡的电场会推动和拉动物质中的电荷，使它们也依次发生振荡。振荡的电荷会发射出新的光波，这个过程会将入射光波的一些能量重新定向到新的方向。这个作用过程被称为散射（scattering），它是各种光学现象的基础。

散射出的光波与原始光波具有相同的频率。通常情况下，当原始光波包含多个频率的光波时，每个频率的光波都会单独与物质发生作用。而对于某个频率的入射光，也不会对其他频率的散射光产生能量贡献，除了一些特定和相对罕见的情况之外，例如荧光（fluorescence）和磷光（phosphorescence），我们不会在本书中讨论这些情况。

一个孤立的分子会使光波向着各个方向进行散射，并使光的强度发生一些方向性的变化。但是更多的光线还是会沿着接近原始传播轴的方向进行散射，包括向前的方向或者向后的方向。分子作为散射体（scatterer）的有效性（分子附近的光波被散射的可能性），会随着波长的变化而发生很大的改变。

在渲染中，我们主要关心的是包含许多分子的集合，光线在与这些聚集体发生相互作用的时候，其结果可能会与孤立分子的情况有所不同。从附近分子散射出的光波通常是相干的，因此会相互干涉，因为它们大概率都具有相同的入射波。本小节的剩余部分将专门讨论若干种多分子散射的特殊情况。

9.1.1 粒子

在理想气体（ideal gas）中，分子之间互不影响，即它们的相对位置是完全随机且不相关的，虽然这是一种抽象概念，但是对于标准大气压下的空气而言，这是一个相当好的模型。在这种情况下，从不同分子散射出的波，其相位差异是随机且不断变化的，因此这些散射波是不相干的，其能量呈线性叠加，如图 9.3 右侧所示。即从 n 个分子散射出的总光能，是单分子散射的 n 倍。

相反，如果分子紧密地排列在比光波长小得多的团簇（cluster）中，那么每个团簇中的散射光波都是同相位的，即会发生相长干涉，如图 9.3 左侧所示。即从由 n 个分子组成的团簇散射出的总光强，是单分子散射的 n^2 倍，也是相同数量分子在理想气体散射中的 n 倍。这种关系意味着，对于一种密度恒定的物质，将分子聚集成团簇，将会显著增加散射光的强度。在保持整体分子密度不变的情况下，增加每个团簇的大小，将会进一步增加散射光的强度，直到团簇的直径接近光的波长，当超过这个临界值的时候，额外增加团簇的大小将并不会进一步增加散射光的强度[469]。

这个过程解释了为什么云和雾的散射光是如此强烈。因为这两种现象都是由冷凝（condensation）产生的，所谓冷凝，是指空气中的水分子聚集成越来越大的团簇的过程。这会显著增加光的散射，即使水分子的总体密度并没有发生变化。有关云渲染的内容，将在章节 14.4.2 中进行讨论。

当我们在讨论光线散射的时候，粒子（particle）这个术语同时代表了孤立的分子与包含多个分子的团簇。由于直径小于波长的多分子粒子散射，是孤立分子散射的增强版本（由于相长干涉），因此它会表现出相同的方向性变化和波长依赖性。这种类型的散射在大气粒子中被称为瑞利散射（Rayleigh scattering），在固体中则被称为丁达尔散射（Tyndall scattering）

当粒子的大小超过光波长的时候，散射波在不再是同相的，这会改变散射的性质。散射光的方向越来越倾向于前向（即光波的原始传播方向），此时波长的依赖性会降低，直到所有的可见光波长都被均匀散射，这种类型的散射被称为米氏散射（Mie scattering）。章节 14.1 会对瑞利散射和米氏散射进行了更加详细的描述。

9.1.2 介质

光在均匀介质（homogeneous medium）中进行传播是另一种重要的情况，均匀介质是指一个充满均匀分布的相同分子的空间。这里所说的均匀分布，并不是指分子间距要像晶体一样完全规则，如果液体和非晶体是纯净物（所有的分子都是相同的），并且没有间隙或者气泡的话，也可以认为它们在光学上是均匀的。

在均匀介质中，散射波会整齐排列，除了沿着最初的传播方向之外，它们会在所有方向上都会产生相消干涉。当原始波与所有的单分子散射波结合之后，最终的结果与原始波基本相同，只有相位速度和振幅（在某些情况下）会有所不同。最后叠加的结果并不会表现出任何散射现象，因为散射现象已经被相消干涉有效抑制了。

原始波和新波的相位速度之比，决定了介质的光学特性，这被称为折射率（index of refraction，IOR），通常使用字母 n 来进行表示。有些介质是具有吸收性的（absorptive），它们会将部分光能转换为热量，这会导致光波的振幅随着距离呈指数级下降，这个下降率由衰减率（attenuation index）定义，使用希腊字母 $\kappa(kappa)$ 来进行表示。n 和 κ 通常会随波长的变化而变化；这两个数字在一起，完整定义了一个介质会如何影响给定波长的光，它们通常被组合成一个复数 $n + i\kappa$，称为复折射率（complex index of refraction）。折射率对分子级别上的光相互作用细节进行了抽象，使得介质可以被视为一个连续的空间体积，这大大简化了模型的复杂度。

虽然光的相位速度并不会直接影响物体外观，但是相位速度的变化则会对外观产生影响，我们稍后会进行解释。另一方面，光的吸收会对视觉效果产生直接影响，因为它降低了光的强度，并且也可以改变光的颜色（如果波长发生变化的话）。图 9.5 展示了一些光线吸收的例子。

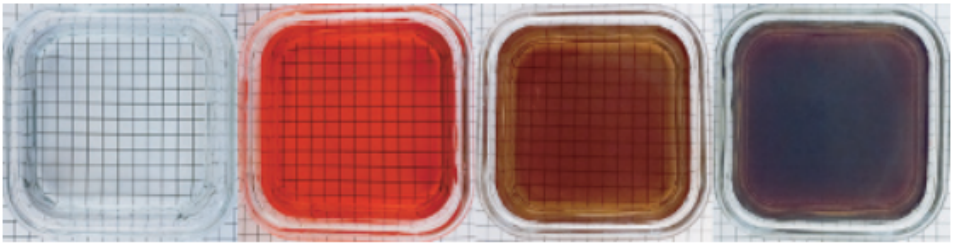


图 9.5：四个装有不同吸收性质液体的小容器。从左到右分别是：纯净水，石榴汁，茶，咖啡。

非均匀介质通常可以建模为嵌有散射粒子的均匀介质。相消干涉抑制了均匀介质中的散射现象，这是由分子的均匀排列所引起的；非均匀介质中的粒子排列并不均匀，从而会导致它们产生散射光波。分子分布中的任何局部变化，都将打破相消干涉的模式，从而允许散射光波进行传播；这种局部变化可以是一个不同分子类型的分子团，或者是气隙、气泡、密度变化等；在任何情况下，它都会像前面讨论的粒子那样，对光波进行散射，其散射性质同样取决于分子团的大小。即使是气体也可以这样进行建模，在气体情况下，“散射粒子”是由分子持续运动所引起的瞬态密度涨落（transient density fluctuation）。这个模型可以为气体建立一个具有意义的折射率 n，这有利于理解它们的光学性质。图 9.6 展示了几种光线散射的例子。

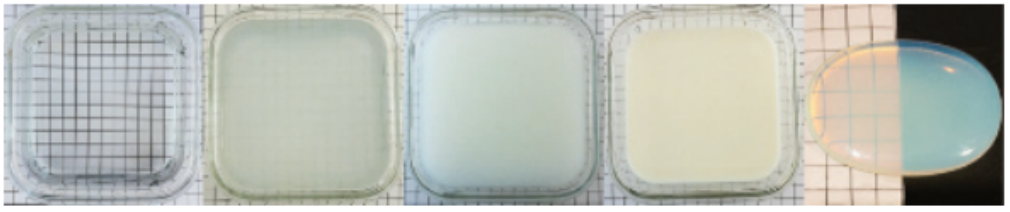


图 9.6：从左到右分别是：水，加了几滴牛奶的水，加了大约 10% 牛奶的水，全脂牛奶，乳白色玻璃。大多数牛奶的散射粒子都要比可见光的波长大，因此它的散射主要是无色的，在中间的图像上有明显的淡蓝色。在最右侧图像中，乳白色玻璃中的散射粒子要比可见光的波长小，因此蓝光的散射强度要比红光强；同时由于明暗背景的分割，透射光在左边更加明显，而散射光则在右边更加明显。

散射和吸收现象都与尺度有关。在小场景中不产生任何明显散射的介质，在较大的尺度上也可能会有相当明显的散射现象。例如：当在房间内观察一杯水时，光在空气中

的散射与在水中的吸收是几乎不可见的；然而在一个较大尺度的环境中，这两种效果可能都会十分显著，如图 9.7 所示。

图 9.7：左图中：在超过几米的范围内，水会对光产生强烈的吸收作用，尤其是红光，因此整体看起来会很蓝。右图中：光在没有严重空气污染或者雾的情况下，也会在数英里尺度的空气中发生散射。

在一般情况下，介质的外观由散射作用和吸收作用的某种组合所决定，如图 9.8 所示。散射的程度决定了灰暗的程度，较高的散射会产生不透明的外观。除了一些比较罕见的情况之外（如图 9.6 中的乳白色玻璃），固体介质和液体介质中的粒子尺寸，往往大于光的波长，并且倾向于均匀散射所有可见波长的光，因此，物体的颜色通常都是由吸收作用的波长依赖性所引起的。而介质的亮度则是由光线散射和光线吸收这两种现象共同作用的结果。特别是白色，它是高散射和低吸收结合的结果。我们将在章节 14.1 中进行更详细的讨论。

图 9.8：具有不同吸收作用和散射作用组合的液体容器。

9.1.3 表面

从光学角度来看，物体的表面（surface）是一个二维界面，它分隔了具有不同折射率的空间体积。在一般的渲染情况下，由空气组成的外部空间，其折射率约为1.003，为简单起见，通常假设空气的折射率为1；而内部空间的折射率则取决于构成物体的物质。

当光波遇到一个表面的时候，该表面的两个因素会对结果产生重要的影响：两侧的物质和表面的几何形状。我们先关注两侧物质所带来的影响，因此假设此时的表面几何形状是一个完美的平面。我们将“外部”（入射方向或者入射波产生的那一侧）折射率表示为 n_1 ，将“内部”（光波通过该表面后将发生传播的一侧）折射率表示为 n_2 。

我们在前一小节中提到过，当光波遇到材料组成不连续或者密度不连续的时候（即两个区域的折射率不同时），它们就会发生散射。分离不同折射率物体的平面，是一种特殊类型的不连续，它会以一种特定的方式来对光进行散射。这个平面边界要求平行于表面的电场分量是连续的，换句话说，表面两侧的电场矢量，到表面平面上的投影必须相匹配。这有以下几个含义：

1. 在表面上的任何散射波，与入射波要么是同相位的，要么是 180° 反相位的。因此，在这个表面上的散射波，其峰值必须与入射波的峰值或者波谷相一致。这限制了散射波只能向两个可能的方向进行移动，一个继续向前进入表面，另一个则远离表面；前者被称为透射波（transmitted wave），后者被称为反射波（reflected wave）。
2. 散射波和入射波具有相同的频率。这里我们假设的是一个单色波，但是这个原理可以应用于任何波，其对应的单色分量会具有相同的频率。
3. 当光波从一个介质移动到另一个介质时，其相位速度（即波通过介质的速度）与相对折射率 (n_1/n_2) 成比例变化。由于光波的频率是固定的，因此波长也与相对折射率 (n_1/n_2) 成比例变化。

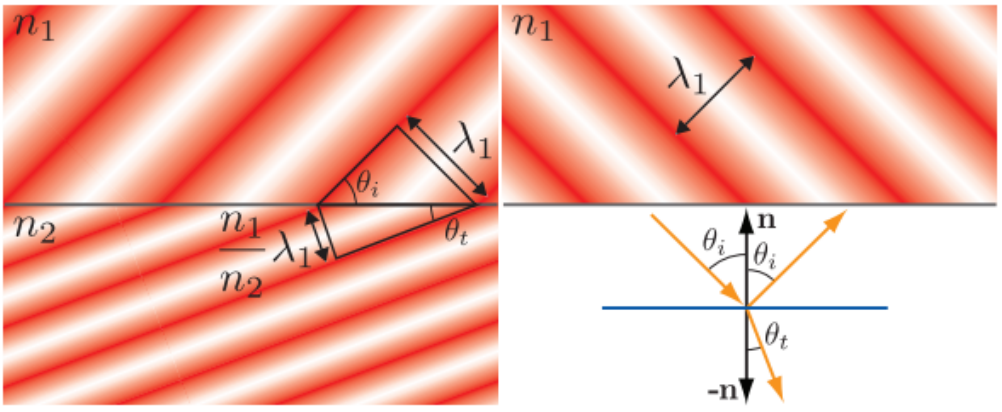


图 9.9：一个光波撞击一个平面，平面两侧物质的折射率分别为 n_1 和 n_2 。左侧图展示了入射波的侧视图，这个入射波从左上角进入，红色的强度代表了波的相位。表面下方的波间距与 (n_1/n_2) 成正比，本例中为 0.5。相位沿着表面排列，因此间距的变化会弯曲（折射）透射波的方向。图中三角形的构造展示了 Snell 定律的推导过程。为了清晰起见，右上方的图显示了反射波的情况，它与入射波具有相同的波间距，因此其方向与表面法线具有相同的夹角。右下方展示了入射波、透射波和反射波方向的矢量图。

最终结果如图 9.9 所示。反射波和入射波的方向，与表面法线之间具有相同的夹角 θ_i ；透射波的方向会以 θ_t 的角度进行弯曲（折射），它与 θ_i 的关系如下：

$$\sin(\theta_t) = \frac{n_1}{n_2}\sin(\theta_i). \tag{9.1}$$

这个折射方程被称为 Snell 定律（折射定律），它被用于全局折射效应，这将在章节 14.5.2 中进一步讨论。

虽然折射现象通常会与玻璃、晶体等透明材质联系在一起，但是它也会发生在不透明物体的表面上；当不透明物体发生折射时，光会在物体的内部进行散射和吸收。光会与物体的介质相互作用，就像图 9.8 中的装有各种液体的容器一样。对于金属而言，其内部包含了许多自由电子（不与分子结合的电子），它们会“吸收”折射光的能量，并将其重定向到反射波中，这就是为什么金属会具有较高的吸收率和较高的反射率的原因。

我们刚才所讨论的表面折射现象（反射和折射），需要折射率在小于单一波长的距离上发生突变。而折射率的逐渐变化并不会对光进行散射，而是会导致光路发生弯曲，就像折射时出现的不连续弯曲一样；当空气密度因温度不同而发生变化的时候，可以看到这种效应，例如海市蜃楼（mirage）和热变形（heat distortion，也叫做热浪），如图 9.10 所示。

图 9.10：一个由于折射率逐渐变化而导致光路弯曲的例子，图中所展示的情况是由温度变化引起的。

即使是一个具有明确边界的物体，如果它被浸没在一个具有相同折射率的物质中，那么也将没有可见的表面。因为在没有折射率变化的情况下，就不会发生反射现象和折射现象。图 9.11 展示了这样的一个例子。

图 9.11：玻璃杯里有很多装饰珠子，它的折射率与水的折射率是一样的。在水面以上，由于珠子的折射率与空气的折射率不同，因此有一个明显可见的表面。在水面以下，由于珠子表面两侧的折射率是相同的，因此这些表面是不可见的。由于这些珠子会吸收特定波长的光，因此我们能够看到这些珠子本身，并且它们会表现出不同的颜色。

到目前为止，我们一直关注的都是表面两侧的物质对光的影响，现在我们将讨论影响表面外观的另一个重要因素：几何形状。严格地说，一个完美平坦的平面表面是不可能存在的，每个表面都具有某种程度的不规则性，即使该表面只由单个原子组成。然而，比波长小得多的不规则表面，对光的行为没有影响；而比波长大得多的不规则表面，虽然会使这个表面在宏观上看起来变得倾斜，但是并不会影响其微观局部的平坦性。只有位于 1–100 个波长范围内的不规则表面，才会导致这个表面与平面具有不同的外观表现，这种现象被称为衍射现象（diffraction），将在章节 9.11 中进一步讨论。

在渲染中，我们通常会基于几何光学（geometrical optics），它忽略了光作为一种波所产生的效应，例如干涉和衍射。这相当于假设所有表面的不规则性，要么小于光的波长，要么比光的波长大得多（100 倍以上）。在几何光学中，光会被建模为一条射线，而不是一种波，当一条射线与表面相交时，相交点的局部区域会倍视为一个平

面。图 9.9 右下角的图像，可以看作是反射现象和折射现象的几何光学表示；而图 9.9 其他部分所展示的，是光作为一种波所发生的现象。从这里开始，直到章节 9.11 为止，我们将一直会在几何光学的领域中进行讨论（会将光译作光线），章节 9.11 中将专门讨论基于波动光学的着色模型。

正如我们前面所提到的，比波长大得多（1–100 个波长）的表面不规则性，会改变表面的局部朝向。当这些不规则性太小以致于无法进行渲染的时候（小于一个像素），我们则将其称为微观几何图形（microgeometry，微表面）；表面的反射方向和折射方向取决于该表面的法线，而这些微表面会改变表面上不同点处的法线，从而改变光线的反射方向和折射方向。

虽然不规则表面上的每个点，都只会在一个方向上反射光线，但是屏幕上的每个像素都覆盖了许多的表面点，这些点会在不同方向上反射光线，因此这个表面的外观，是由所有不同反射方向的光线的聚合结果所决定的。图 9.12 展示了两个在宏观尺度上具有相似形状，但是微观几何形状显著不同的表面，其外观也具有明显的差异。

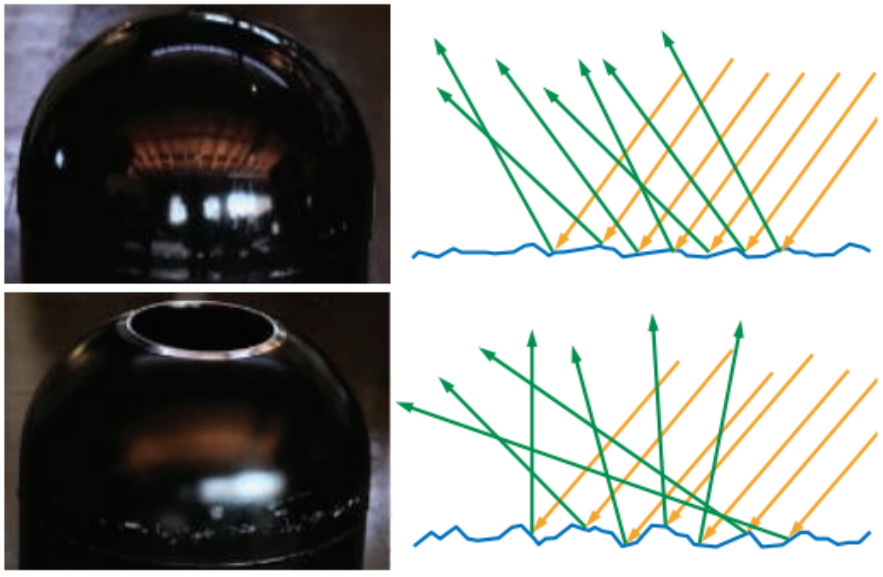


图 9.12：左侧展示了两个表面的照片，右侧使用示意图的形式，展示了它们所对应的微观几何结构。上方的表面拥有略微粗糙的微观几何形状，入射光线击中了表面上不同的点，每个点的法线方向略有差异，并在一个狭窄的锥形方向上被反射，其宏观效果是具有轻微模糊的反射。下方表面具有更加粗糙的微观几何形状，入射光线照射到的表面点，具有明显不同的法线方向，反射光线会以一个较宽的锥体进行扩散，从而导致宏观的反射效果变得更加模糊。

在渲染中，我们并不会对微观几何形状进行明确的建模，而是会以一种统计的方式对其进行处理，将该表面视为一个具有微观结构法线的随机分布。因此，我们将表面建模为，会在一个连续扩散的方向上，对光线进行反射和折射。其中扩散的宽度（锥形

范围的大小），以及反射细节和折射细节的模糊程度，取决于微观几何法线的统计方差，即表面微尺度的粗糙度（roughness），如图 9.13 所示。

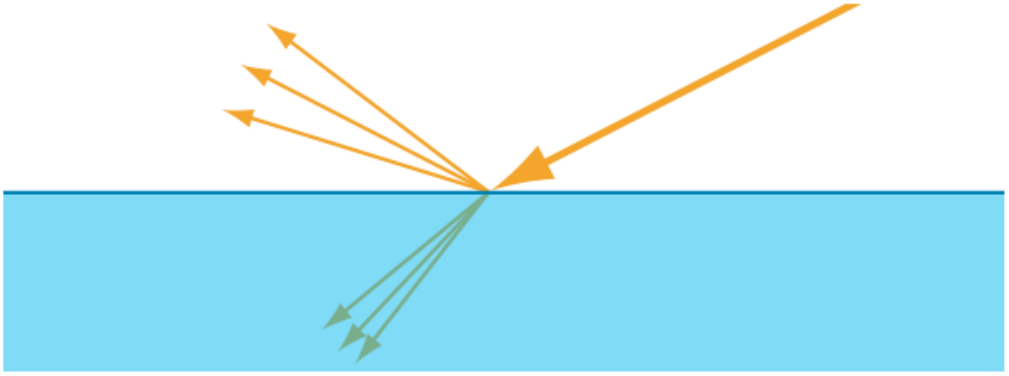


图 9.13：从宏观上看，表面可以被视为在多个方向上，对光线进行反射和折射。

9.1.4 次表面散射

进入物体内部的折射光线，会继续与内部物质发生相互作用。前面我们提到，金属具有较高的吸收率和较高的反射率，即金属表面会反射大部分的入射光线，进入金属内部的折射光线也会被迅速吸收；相比之下，非金属物体则会表现出广泛的散射行为和吸收行为，这与我们在图 9.8 中所看到的液体容器类似。透明材料具有低散射和低吸收的特性，进入物体内部的任何折射光线，最终都会从物体中折射出来。我们在章节 5.5 中讨论过一些简单的透明物体渲染方法；有关透明物体的折射问题，将在章节 14.5.2 中进行讨论。在本小节中，我们将重点关注不透明的物体，在这些物体中，透射光线在经历多次散射和吸收之后，最终有一部分会从表面重新发射出来，并被人眼观察到，如图 9.14 所示。

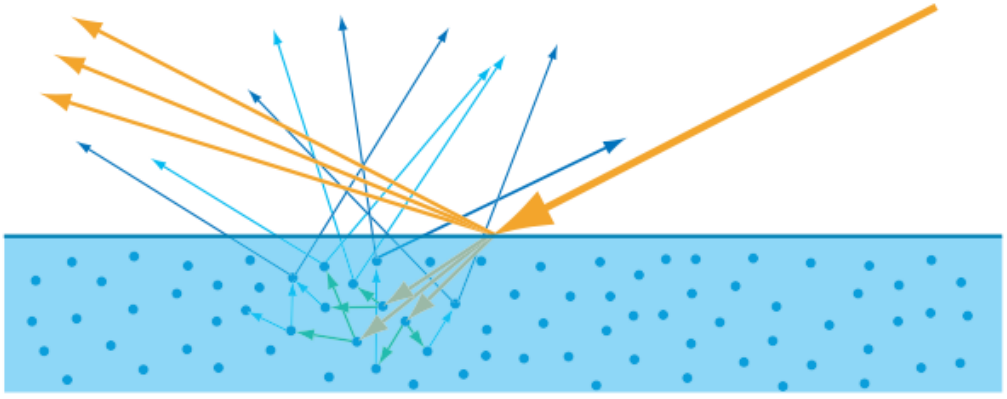


图 9.14：折射光线在穿过物体时会被吸收。在这个例子中，大部分被吸收的都是波长较长的光线，留下的主要是波长较短的蓝光。另外，光线还会在物体内部的粒子中发生散射，最终有一些折射光线会散射出表面，例如图中以不同方向离开表面的蓝色箭头。

这种次表面散射（subsurface scatter）的光线，会以相对于入射点的不同距离从表面射出，这个进出距离（entry–exit distances）的分布取决于材料中散射粒子的密度和性质，这些距离与着色尺度（像素的大小，以及着色样本之间的距离）之间的关系是很重要的。如果这个进出距离比着色尺度要小，那么可以假定它们为零，这样我们就可以将次表面散射与表面反射整合到同一个局部着色模型中，即某个着色点上的出射光线，只依赖于该点的入射光线。由于次表面散射与表面反射具有明显不同的外观表现，因此可以很容易地将它们划分为单独的着色选项，使用镜面项（specular term）控制表面反射现象，使用漫反射项（diffuse term）控制局部次表面散射（local subsurface scattering）现象。

如果这个进出距离比着色尺度要大，那么就需要专门的渲染技术，来表现光线在某一点进入表面，并从另一点离开表面的视觉效果。章节 14.6 详细介绍了这些全局次表面散射（global subsurface scattering）技术。局部次表面散射和全局次表面散射之间的差异如图 9.15 所示。

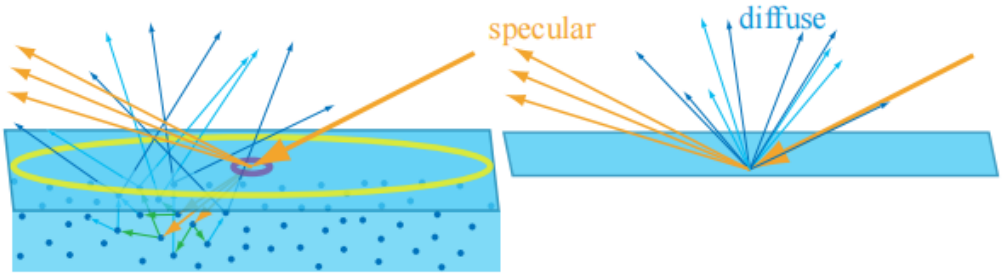


图 9.15：左图中，我们渲染了一个具有次表面散射的材质，图中的黄色圆圈和紫色圆圈代表了两种不同大小的采样区域。其中黄色圆圈代表了单个着色样本所覆盖的区域，大于次表面散射的距离，因此我们可以在渲染中忽略这些距离所带来的影响，并在局部着色模型中将次表面散射项视为漫反射项，如右图所示。如果我们向着靠近表面的方向进行移动，单个着色样本所覆盖的区域会变小，如左图中的紫色圆圈，与着色样本所覆盖的区域相比，此时次表面散射的距离很大，无法忽视这些距离所带来的影响，因此需要使用全局的着色模型和光照技术，来从这些着色样本中生成逼真的图像。

值得注意的是，局部次表面散射技术和全局次表面散射技术模拟的是完全相同的物理现象。每种情况下的最佳选择（即到底使用哪种模型和技术），不仅取决于材质的属性，还取决于观察的尺度。例如：当渲染一个孩子在玩塑料玩具的场景时，我们很可能需要使用全局次表面散射技术来准确地渲染孩子的皮肤，而对于塑料玩具而言，可能一个局部的漫反射着色模型就足够了。这是因为皮肤中的散射距离，要比在塑料中的散射距离大得多，但是如果相机足够远的话，皮肤的散射距离也可能会小于一个像

素，此时局部的着色模型对于儿童和玩具而言都是十分准确的。相反，在一个极端的特写镜头中，塑料也可能会表现出明显的非局部次表面散射现象，这时候就需要全局次表面散射技术来对玩具进行准确地渲染。

9.2 相机

如章节 8.1.1 所述，在渲染的时候，我们会计算从表面着色点到相机位置的 radiance。这模拟了一个简化了的成像系统，例如胶片相机、数码相机或者人眼。

这类成像系统中，包含了一个由许多离散小传感器组成的传感器表面。例如：眼睛中的视杆细胞和视锥细胞、数码相机中的光电二极管（photodiode）、或者胶片中的染料颗粒，这些传感器可以检测到其表面上的 irradiance，并生成一个颜色信号。这些 irradiance 传感器本身并不能产生图像，因为它们对来自所有入射方向的光线都进行了平均处理。因此，完整的成像系统还应当包括一个具有小孔（aperture，光圈）的不透光外壳，它对光线进入和撞击传感器的方向进行了限制；放置在光圈处的透镜会将光线进行聚焦，这样每个传感器就只能接收来自一小部分入射方向的光线（小孔成像原理）。外壳、光圈和透镜的综合作用，使得这些传感器具有了特定的方向性（directionally specific），即只会对分布在一小块区域和一小组入射方向上的光线进行平均。正如我们在章节 8.1.1 中所看到的，这些传感器量化了来自各个方向光流的表面密度，即测量了平均 radiance，而不是测量平均 irradiance；换句话说，传感器量化记录了单条光线的亮度和颜色。

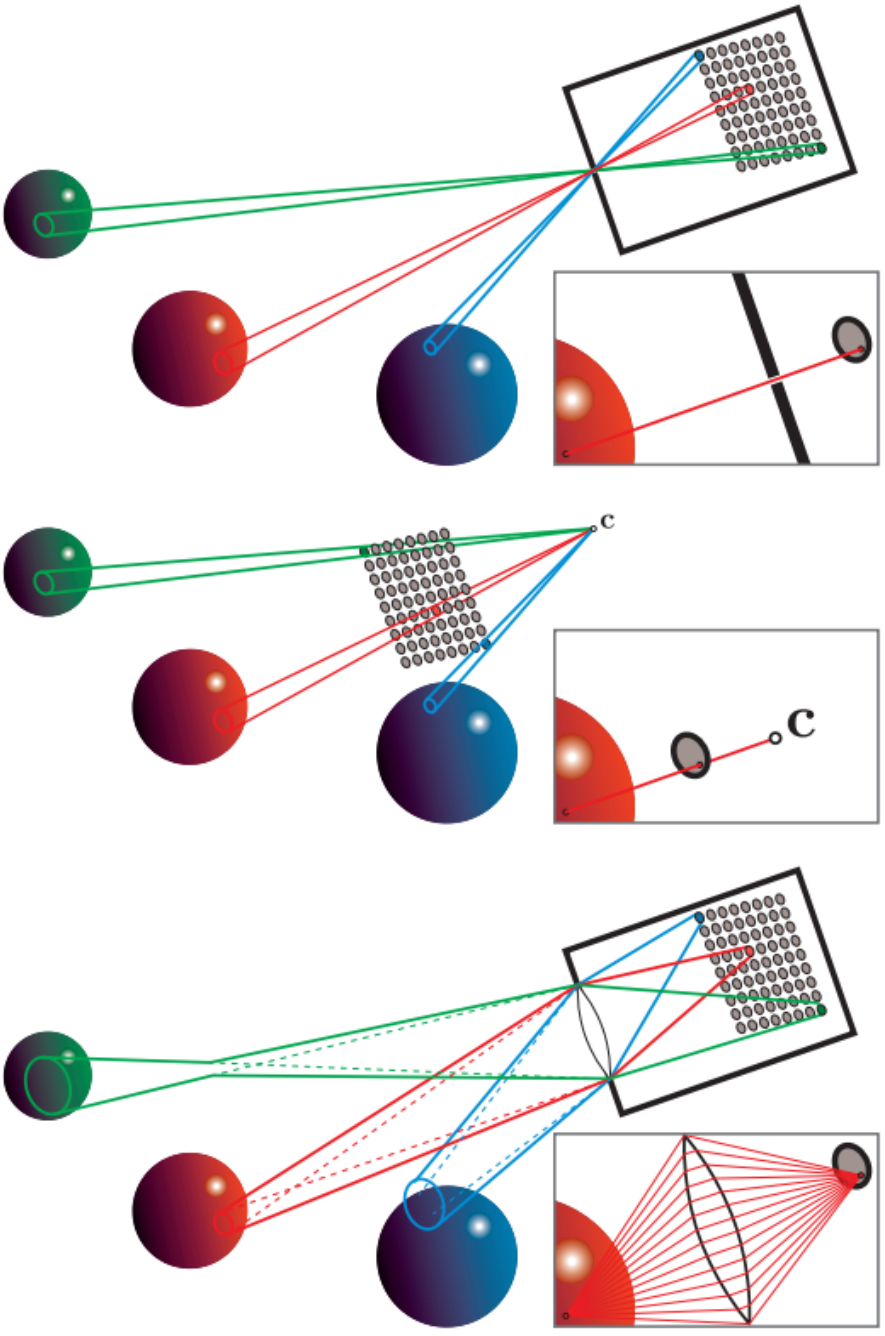


图 9.16：每个相机成像模型中都包含了一个像素传感器阵列。图中的每根实线，都代表了传感器从场景中收集到的一组光线。每个子图中的窗口图像，展示了像素传感器上的单点样本所收集的光线。第一行图片展示了一个针孔相机模型；第二行图片展示了一个典型的渲染系统模型，即一个带有相机点 c 的针孔相机；第三行图片展示了一个更加物理正确的镜头相机，其焦点位于红色球体上，另外两个球体并不在焦点上。

在历史上，渲染模拟了一种特别简单的成像传感器，它被称为针孔相机（pinhole camera），如图 9.16 的第一行所示。针孔相机没有镜头，并且具有一个非常小的光圈（在理想的情况下，它是一个没有大小的数学抽象点）。这个小光圈对传感器进行了限制，即传感器表面上的每个点都只能收集一束光线，而一个单独传感器中包含了

若干个这样的点，也就是说，每个小传感器都会收集一个锥形区域内的光线，这个圆锥的底（base）覆盖了传感器的表面，圆锥的顶点（apex）位于光圈处。图形渲染系统以一种略有不同（但是是等价的）的方式，对针孔相机进行了建模，如图 9.16 的第二行所示。针孔光圈的位置用点 c 进行表示，通常被称为“相机位置”或者“眼睛位置”。这个点也是透视变换的投影中心（章节 4.7.2）。

在渲染时，每个着色样本都对应了一条光线，即对应于传感器表面上的一个采样点。而抗锯齿的过程（章节 5.4）则可以理解为，对每个离散传感器表面收集到的信号进行重建。然而，由于图形渲染并不会受到物理传感器的限制，因此我们可以对这个过程进行更加一般化的处理，即从离散的着色样本中，重建出连续的图像信号。

虽然真正的针孔相机可以很容易地制造出来，但是对于大多数实践中所使用的相机，以及人眼来说，针孔相机都是一个糟糕的模型，其实用价值并不高。图 9.16 第三行展示了一个使用镜头的成像系统模型，这个模型中包含了一个镜头，这允许相机使用一个更大的光圈，大大增加了成像传感器所收集的光量。但是它也会导致相机的景深（depth of field）是有限的（章节 12.4），太近或者太远的物体都会被模糊（大光圈的时候，景深效果尤其明显）。

镜头除了会限制景深之外，还有一些其他的影响。传感器上的每个点（包括完美聚焦的点）都会接收一个锥形区域内的光线；而在理想化的模型中，每个着色样本都只会对应唯一的观察射线，这样处理有时候会引入数学奇异点（即无法定义的点）、数值不稳定性或者视觉瑕疵；当我们在渲染图像的时候，将真实的物理模型牢记于心，有助于我们识别和解决这些问题。

9.3 The BRDF

最终，基于物理的渲染可以归结为沿着一组观察射线，计算进入相机的 radiance。这里我们使用章节 8.1.1 中所介绍的入射 radiance 的符号表示，对于一个给定的观察射线，我们需要计算的是 $L_i(\mathbf{c}, -\mathbf{v})$，其中 $\mathbf{c}$ 是相机的位置，$-\mathbf{v}$ 是沿着观察射线的方向。这里我们使用 $-\mathbf{v}$ 是由于两个符号约定：第一，$L_i()$ 中的方向向量，总是指向远离给定点的方向，在这里就是指向远离相机位置的方向。第二，观察向量 $\mathbf{v}$ 总是会指向相机。

在渲染中，场景通常会被建模为彼此之间具有介质的物体集合（“介质”一词实际上来自于拉丁语，它的意思是“在中间”）。通常我们所讨论的介质是适量且相对干净的空气，它并不会对光线的 radiance 产生明显地影响，因此在渲染中可以被忽略。但是有时候，光线所穿过的介质会通过吸收或者散射等方式，对光线的 radiance 产生明

显影响，这样的介质被称为参与介质（participating media），因为它们参与了光线在场景中的传播过程。我们将在第 14 章中详细介绍有关参与介质的话题。而在本章节中，我们暂时假设没有参与介质的存在，因此进入相机的 radiance，与其观察方向上离开最近物体表面的 radiance 是相等的，即：

$$L_i(\mathbf{c}, -\mathbf{v}) = L_o(\mathbf{p}, \mathbf{v}) \tag{9.2}$$

其中， $\mathbf{p}$ 是观察射线与最近物体表面的交点。

根据方程 9.2，我们的新目标是计算 $L_o(\mathbf{p}, \mathbf{v})$ 。我们在章节 5.1 中曾讨论过简单的着色模型，而这里的计算过程则是其物理版本。有时候一些表面也会直接发出 radiance，但是更加常见的情况是，离开这个表面的 radiance 来自于其他地方，并通过章节 9.1 中所描述的物理作用，最终被反射到相机中。在本章节中，我们将不考虑透明物体（章节 5.5 和章节 14.5.2）和全局次表面散射（章节 14.6）的情况；换句话说，我们所关注的是局部反射现象，它将照射到当前着色点的光线，重新发射回外部，这些现象包括表面反射以及局部次表面散射，它们只依赖于入射光方向 $\mathbf{l}$ 和指向外部的观察方向 $\mathbf{v}$ 。双向反射分布函数（bidirectional reflectance distribution function，BRDF）描述了着色点表面的局部反射系数，记为 $f(\mathbf{l}, \mathbf{v})$ 。

在原始推导中[1277]，BRDF 是针对均匀表面进行定义的。也就是说，我们会假设表面上各处的 BRDF 都是相同的。然而，对于现实世界中的物体（以及渲染场景中的物体）而言，很少会有表面具有完全相同的材质属性。即使是由单一材料组成的物体（例如银质雕像），其表面上也会有划痕、暗斑、污渍或者其他的一些变化，从而导致两个不同的着色点具有不同的视觉特性。从技术上来讲，根据空间位置捕获 BRDF 变化的函数，被称为空间变化的 BRDF（spatially varying BRDF，SVBRDF）或者空间 BRDF（spatial BRDF，SBRDF）。然而，这种情况在实践中非常普遍，因此通常还是会使用 BRDF 这个较短的术语来进行描述，我们会默认假定 BRDF 与表面位置有关。

光线的入射方向和出射方向各有两个自由度。一种常用的参数化表示包括两个角度：相对于表面法线 $\mathbf{n}$ 的仰角 θ ，以及相对于表面法线 $\mathbf{n}$ 的方位角（即水平旋转） ϕ 。在一般情况下，BRDF 是一个包含四个标量变量的函数。各向同性（Isotropic）BRDF 是一个重要的特例，当光线入射方向和出射方向围绕表面法线 $\mathbf{n}$ 旋转的时候，各项同性 BRDF 会保持不变，并保持它们之间的相对角度不变。图 9.17 展示了这两种情况下所使用的输入变量。各向同性 BRDF 是一个包含三个标量变量的函数，因为我们只需要光线绕表面法线 $\mathbf{n}$ 的旋转角度，或者相机的旋转角度 ϕ 即可。也就是

说，如果将一个具有均匀各向同性材质的物体放置在转盘上，并旋转它的话，在给定的光线和相机条件下，这个物体表面在所有旋转角度上看起来都是相同的。

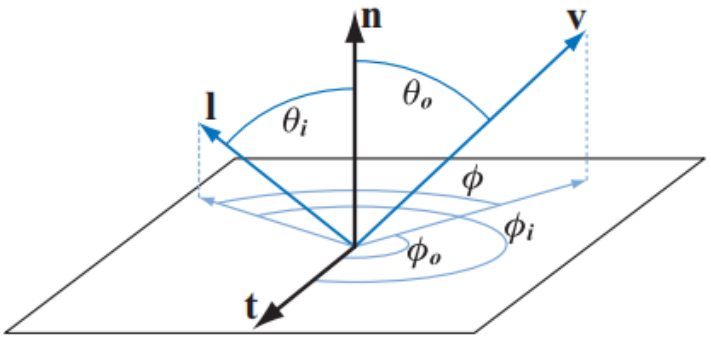


图 9.17：BRDF 输入参数。方位角 ϕ_i 和 ϕ_o 都是相对于给定切向量 $\mathbf{t}$ 定义的。而在各向同性 BRDF 中，我们只需要考虑相对方位角 ϕ 即可，不需要考虑参考切向量与 ϕ_i 、ϕ_o 。

由于我们忽略了荧光（fluorescence）和磷光（phosphorescence）等现象，因此我们可以假设，给定波长的入射光会以相同的波长被反射出来。反射光的光量会根据其波长发生变化，可以使用以下两种方法来进行模拟：第一种是将波长视为 BRDF 的额外输入变量；第二种则是将 BRDF 视为会返回一个光谱分布的值。虽然第一种方法有时会用于离线渲染[660]，但是在实时渲染中，总是会使用第二种方法。由于实时渲染器一般将光谱分布表示为一个 RGB 三元组，这意味着 BRDF 会返回一个 RGB 值。

为了计算着色点的出射光线 $L_o(\mathbf{p},\mathbf{v})$ ，我们将 BRDF 项合并到反射方程（reflectance equation）中：

$$L_o(\mathbf{p},\mathbf{v}) = \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{p},\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \tag{9.3}$$

积分符号的下标 $\mathbf{l}\in\Omega$ 意味着是，我们对位于表面单位半球（以着色点为球心，以表面法线 $\mathbf{n}$ 为半球方向）上的入射光方向 $\mathbf{l}$ 进行积分。请注意，入射光方向 $\mathbf{l}$ 会在半球范围内进行连续扫描，也就是说，它并不是某个特定的“光线方向”。这个想法的核心是：来自任何入射方向的光线，都会（通常会）产生一些 radiance。我们用 $d\mathbf{l}$ 来表示入射光方向 $\mathbf{l}$ 周围单位立体角的微分（有关立体角的内容在章节 8.1.1 讨论过）。

总而言之，反射方程表明，出射的 radiance 等于入射 radiance 乘以 BRDF 项，再乘以表面法线 $\mathbf{n}$ 和入射光方向 $\mathbf{l}$ 点积的积分（其中入射光方向 $\mathbf{l}$ 来自半球范围 Ω ）。

为了简化表述，在本章节的剩余部分中，我们将从 $L_i()$ ， $L_o()$ 和反射方程中省略表面着色点 $\mathbf{p}$ ，即：

$$L_o(\mathbf{v}) = \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \tag{9.4}$$

在计算反射方程的时候，通常会使用球坐标 ϕ 和 θ 来对半球进行参数化。在这个参数化表示下，微分立体角 $d\mathbf{l}$ 就等于 $\sin\theta_i d\theta_i d\phi_i$ 。利用这种球坐标系参数化，我们可以推导出方程 9.4 的二重积分形式：

$$L_o\left(\theta_o,\phi_o\right) = \int_{\phi_i=0}^{2\pi}\int_{\theta_i=0}^{\pi/2} f\left(\theta_i,\phi_i,\theta_o,\phi_o\right) L\left(\theta_i,\phi_i\right)\cos\theta_i\sin\theta_i d\theta_i d\phi_i \tag{9.5}$$

回顾一下，方程 9.4 中的 $(\mathbf{n}\cdot\mathbf{l}) = \cos\theta_i$ ，有关 $\theta_i,\phi_i,\theta_o,\phi_o$ 的几何意义如图 9.17 所示。

在某些情况下，我们使用一些稍微不同的参数化表示会变得更加方便，即使用仰角的余弦值 $\mu_i = \cos\theta_i, \mu_o = \cos\theta_o$ ，而不是仰角本身 θ_i,θ_o 。对于这种参数化表示，微分立体角 $d\mathbf{l}$ 就等于 $d\mu_i d\phi_i$ 。使用 (μ,ϕ) 这种参数化，表示可推导出以下的积分形式:

$$L_o\left(\mu_o,\phi_o\right) = \int_{\phi_i=0}^{2\pi}\int_{\mu_i=0}^{1} f\left(\mu_i,\phi_i,\mu_o,\phi_o\right) L\left(\mu_i,\phi_i\right)\mu_i d\mu_i d\phi_i \tag{9.6}$$

BRDF 仅在光线方向和观察方向都位于表面上方时才有定义。对于光线方向在表面下方的情况，可以通过将 BRDF 乘以零，或者首先不计算这些方向的 BRDF 来避免这种情况。但是对表面下方的观察方向呢？换句话说就是，当点积 $\mathbf{n}\cdot\mathbf{v}$ 为负的时候会发生什么情况？从理论上来说，这种情况永远都不会发生，因为此时表面是背对相机的，是不可见的。然而，在实时应用中十分常见的顶点法线插值和法线映射等操作，会在实践中产生这样的情况。可以将 $\mathbf{n}\cdot\mathbf{v}$ 的值限制到 0，或者使用 $\mathbf{n}\cdot\mathbf{v}$ 的绝对值，来避免 BRDF 对表面下方的观察方向进行计算，但是这两种方法都会产生一些瑕疵。寒霜引擎使用 $\mathbf{n}\cdot\mathbf{v}$ 的绝对值再加上一个很小的数字（0.0001），来避免发生除以 0 的情况[960]。另一种可能的方法是“soft clamp”，这意味着随着 $\mathbf{n},\mathbf{v}$ 之间的夹角超过 90° 时， $\mathbf{n}\cdot\mathbf{v}$ 的值会逐渐趋于 0。

物理定律对任何的 BRDF 都有两个约束。第一个约束是 Helmholtz 互易性（Helmholtz reciprocity），这意味着交换入射角和出射角后，其函数值相同，即：

$$f(\mathbf{l}, \mathbf{v}) = f(\mathbf{v}, \mathbf{l}) \tag{9.7}$$

在实践中，用于渲染的 BRDF 经常会违反 Helmholtz 互易性，而且并不会出现明显的瑕疵，除非是一些特别需要 Helmholtz 互易性的离线渲染算法，例如双向路径跟踪（bidirectional path tracing）。然而，当在确定某种 BRDF 在物理上是否可信时，Helmholtz 互易性是一个有效的鉴别工具。

第二个约束是能量守恒（conservation of energy）——即出射的能量不能大于入射的能量（这里不包含自发光表面，它将作为一个特殊情况进行处理）。离线渲染算法（例如 path tracing）需要能量守恒来保证收敛性；而对于实时渲染来说，我们并不需要严格保证能量守恒，但是近似的能量守恒还是很重要的。使用一个明显违背能量守恒的 BRDF 对表面进行渲染，它会看起来过亮，不太真实。

定向半球反射率（directional–hemispherical reflectance） $R(\mathbf{l})$ 是一个与 BRDF 有关的函数，它可以用来衡量 BRDF 的能量守恒程度。尽管它的名字有点令人生畏，但是定向半球反射率实际上是一个很简单的概念。对于一个给定方向的入射光线，$R(\mathbf{l})$ 测量了这个入射光线被反射到半球方向内的数量。本质上，$R(\mathbf{l})$ 测量了给定入射方向上的能量损失。这个函数的输入是入射方向 $\mathbf{l}$，$R(\mathbf{l})$ 的定义如下：

$$R(\mathbf{l}) = \int_{\mathbf{v} \in \Omega} f(\mathbf{l}, \mathbf{v})(\mathbf{n} \cdot \mathbf{v}) d\mathbf{v} \tag{9.8}$$

需要注意的是，这里的观察方向 $\mathbf{v}$，就像反射方程中的入射光方向 $\mathbf{l}$ 一样，它扫过了整个半球范围，并不是代表单一的观察方向。

还有一个与 $R(\mathbf{l})$ 类似，但是某种意义上相反的函数——半球定向反射率 $R(\mathbf{v})$（hemispherical–directional reflectance），其定义与 $R(\mathbf{l})$ 类似：

$$R(\mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v})(\mathbf{n} \cdot \mathbf{l}) d\mathbf{l} \tag{9.9}$$

如果 BRDF 满足互易性的话（即交换入射方向和出射方向，结果不变），那么半球定向反射率和定向半球反射率实际上是相等的，可以使用同一个函数来计算其中的任意一个。在二者可以交换使用的时候，可以使用定向反照率（directional albedo）这个术语来作为两个反射率的统称。

由于能量守恒，因此定向半球反射率 $R(\mathbf{l})$ 的值必须始终在 $[0, 1]$ 范围内。反射率为 0 代表着所有的入射光线都被吸收，或者以其他方式丢失；反射率为 1 代表着所有的

入射光线都会被反射。在大多数情况下，$R(\mathbf{l})$ 的值将介于这二者之间。与 BRDF 一样，$R(\mathbf{l})$ 的值也会随着波长的变化而变化，因此 $R(\mathbf{l})$ 被表示为一个 RGB 向量，以便用于渲染。由于 RGB 的每个分量（红、绿、蓝）都被限制在 $[0,1]$ 的范围内，因此 $R(\mathbf{l})$ 也可以被认为是一个颜色。需要注意的是，这个约束不适用于 BRDF。BRDF 作为一个分布函数，如果其描述的分布高度是不均匀的话，那么它在某些方向上，可以具有任意高的值（例如高光中心）。对 BRDF 能量守恒的要求是，对于所有可能的入射方向 $\mathbf{l}$，$R(\mathbf{l})$ 都不大于 1。

最简单的 BRDF 是 Lambertian，它对应于章节 5.2 中简要讨论的 Lambertian 着色模型。Lambertian BRDF 具有一个恒定值，著名的、用于区分 Lambertian 着色模型的 $(\mathbf{n}\cdot\mathbf{l})$ 项并不是 BRDF 的一部分，而是方程 9.4 的一部分。尽管 Lambertian BRDF 很简单，但是它经常用于实时渲染中，以表示局部次表面散射（尽管它正在被更加精确的模型所取代，详见章节 9.9）。Lambertian 表面的定向半球反射率也是一个常数，将方程 9.8 中 $f(\mathbf{l},\mathbf{v})$ 项作为一个常数，可以计算得到定向半球反射率 $R(\mathbf{l})$，并将其作为 BRDF:

$$R(\mathbf{l}) = \pi f(\mathbf{l},\mathbf{v}) \tag{9.10}$$

Lambertian BRDF 的恒定反射率通常被称为漫反射颜色（diffuse）c_{diff}，或者反照率（albedo）ρ。在本章节中，为了强调它与次表面散射现象的联系，我们将这个量称为次表面反照率（subsurface albedo）ρ_{ss}。有关次表面反照率的内容，将会在章节 9.9.1 中详细讨论。由方程 9.10 得到的 BRDF 可以推导出如下结果:

$$f(\mathbf{l},\mathbf{v}) = \frac{\rho_{\mathrm{ss}}}{\pi} \tag{9.11}$$

其中包含了 $1/\pi$ 项，这是因为对余弦项在整个半球范围内进行积分的结果是 π，这些项经常出现在 BRDF 中。

> 译者注：反照率 albedo 是一个行星物理学术语，用于描述天体的反射能力，其数学定义是反射辐射和入射辐射之比，是一个无量纲量。而反射率 reflectance 用于表示某一种波长光线的反射辐射和入射辐射之比。当只存在漫反射，不存在镜面反射时，albedo 才等于 diffuse，其取值范围在 $[0,1]$ 之间。

理解 BRDF 的一种方法是保持输入方向不变，将其可视化，如图 9.18 所示。对于给定的入射光方向，会在所有出射方向上展示 BRDF 的值。围绕着色点的半球部分对应了漫反射分量，因为 radiance 会在半球范围内被均匀反射。其中的椭球片对应了镜

面波瓣（specular lobe），这个波瓣代表了入射光线的出射方向，波瓣的厚度、大小与反射的模糊程度相对应。根据互易性原则，这些 BRDF 的可视化也可以被认为是不同入射方向对单个出射方向的贡献大小。

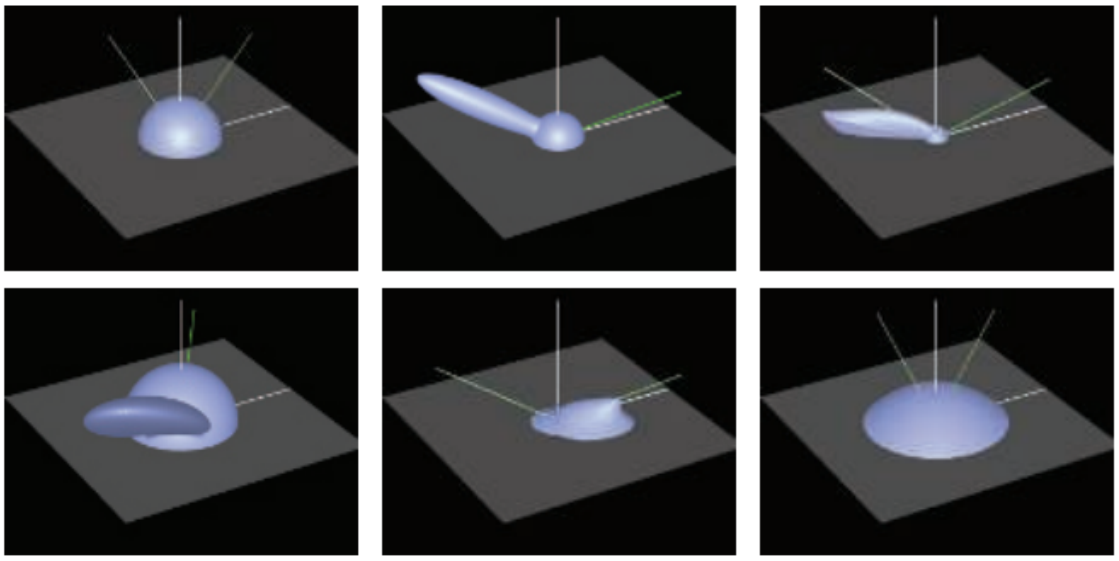


图 9.18：BRDF 的一些例子。图中右侧的绿色实线代表了入射光线的方向，中间的白色实线代表了表面法线，绿色虚线代表了理想状态下的反射方向。在第一行中，左侧展示了一个 Lambertian BRDF（一个简单半球）；中间展示了在 Lambertian 项中添加了 Blinn-Phong 高光的结果；右侧展示了 Cook-Torrance BRDF [285, 1779]，值得注意的是，镜面高光并不是最强的反射方向。在第二行中，左侧是 Ward 各向异性模型的特写，在这种情况下，反射方向是一个倾斜的镜面波瓣；中间展示了 Hapke/Lommel-Seeliger“月面（lunar surface）”BRDF [664]，它具有强烈的后反射；右侧展示了 Lommel-Seeliger 散射，尘埃表面会将光线散射到掠射角上（grazing angle）。

9.4 光照（Illumination）

反射方程（方程 9.4）中的 $L_i(\mathbf{l})$ 项（入射 radiance），代表了从场景其他部分照射到着色点表面的光线。全局光照（global illumination）算法通过模拟光线在场景中的传播和反射，来计算这个 $L_i(\mathbf{l})$ 项，这些算法会使用到渲染方程（rendering equation）[846]，其中反射方程就是渲染方程的一个特殊例子。有关全局光照的内容将在第 11 章进行讨论，而在本章节（第 9 章）和下一章（第 10 章）中，我们将关注局部光照（local illumination），它使用反射方程来计算每个表面点的局部着色情况。在这些局部光照算法中，$L_i(\mathbf{l})$ 项是默认已知的，因此不需要进行计算。

在现实场景中，$L_i(\mathbf{l})$ 包括了来自各个方向的非零 radiance，无论这些 radiance 是直接从光源发射出的，还是从其他表面反射出的。与章节 5.2 中所讨论的方向光和精确光源不同，现实世界的光源一个是覆盖了非零立体角的面光源（area light）。在本章节中，我们将假定使用有限形式的 $L_i(\mathbf{l})$，它仅由方向光和精确光源组成，而更加一般的光照环境将留到第 10 章进行讨论。有了这个限制前提，可以使得本章节的讨论更加集中有效。

虽然方向光和精确光源都是非物理的抽象光源，但是它们可以作为对物理光源的近似。这样的推导其实是很重要的，因为这样做我们可以将这些光源合并到一个基于物理的渲染框架中，同时能够把握近似所带来的误差。

假设现在我们取一个很小的、很远的面光源，并定义向量 $\mathbf{l}_c$ 指向该面光源的几何中心；同时，我们将该面光源的颜色 $\mathbf{c}_{light}$ 定义为：从正对（$\mathbf{n} = \mathbf{l}_c$）该光源的白色 Lambertian 表面所反射出的 radiance。这是一个很直观的定义，因为光线的颜色直接对应了它所产生的视觉效果。

有了以上这些定义，我们可以推导出方向光的一种极限情况：即在保持光源颜色 $\mathbf{c}_{light}$ 不变的情况下，将面光源的大小缩小到零[758]。在这种情况下，反射方程（方程 9.4）中的积分可以简化为一个 BRDF 计算，这个计算成本要低得多：

$$L_o(\mathbf{v}) = \pi f\left(\mathbf{l}_c, \mathbf{v}\right) \mathbf{c}_{\text{light}} \ \left(\mathbf{n} \cdot \mathbf{l}_c\right) \tag{9.12}$$

上述方程中的 $\left(\mathbf{n} \cdot \mathbf{l}_c\right)$ 通常会被限制到 0，从而避免表面下方的光线对这个着色点产生贡献，即：

$$L_o(\mathbf{v}) = \pi f\left(\mathbf{l}_c, \mathbf{v}\right) \mathbf{c}_{\text{light}} \ \left(\mathbf{n} \cdot \mathbf{l}_c\right)^+ \tag{9.13}$$

方程 9.13 中的符号 x^+ 代表了将 x 限制到 0，我们在章节 1.2 中讨论过。

精确光源同样也可以这样进行处理，二者唯一的区别在于，面光源不需要距离着色点很远，同时 $\mathbf{c}_{light}$ 的值会随着到光源的距离，发生平方反比衰减，如方程 5.11 所示。当场景中存在多个光源的时候，我们可以多次计算方程 9.12，并对结果求和，即：

$$L_o(\mathbf{v}) = \pi \sum_{i=1}^{n} f\left(\mathbf{l}_{c_i}, \mathbf{v}\right) \mathbf{c}_{\text{light}_i} \left(\mathbf{n} \cdot \mathbf{l}_{c_i}\right)^+ \tag{9.14}$$

其中 $\mathbf{l}_{c_i}$ 代表了第 i 个光源的方向，$\mathbf{c}_{\text{light}_i}$ 代表了第 i 个光源的颜色。请注意方程

9.14 和方程 5.6 之间的相似性。

方程 9.14 中的 π，会和 BRDF 中经常出现的 $1/\pi$ 相抵消（例如方程 9.11）。这种抵消可以将除法操作移出着色器，并且使得着色方程更加易于阅读。然而，在将学术论文中的 BRDF 用于实时着色方程时，必须谨慎处理这一点，通常情况下，BRDF 在使用之前需要乘以 π。

9.5 菲涅尔反射

在章节 9.1 中，我们从一个较高的层次讨论了光与物质的相互作用。在章节 9.3 中，我们介绍了如何使用数学形式，来表达这些相互作用的基本机制：BRDF 和反射方程。现在我们准备开始深入研究一些特定的光学现象，并对这些现象进行量化描述，以便可以在着色模型中使用它们。我们将从章节 9.1.3 中首次讨论的平面反射开始。

物体的表面是指周围介质（通常是空气）与内部物质之间的分界面，光与两种物质之间的平面分界面的相互作用，遵循菲涅尔方程（Fresnel equation），它是由 Augustin-Jean Fresnel（1788 – 1827）提出的。根据几何光学的假设，菲涅尔方程需要一个平面分界面才能生效；也就是说，我们会假设这个表面在 1–100 倍光线波长的范围内不存在任何不规则性。小于这个范围的不规则性对光线没有影响，而大于这个范围的不规则性，会明显改变表面的倾斜程度，但是并不影响表面局部的平坦性。

照射到平面上的光线会分为反射和折射两个部分。反射光线的方向（$\mathbf{r}_i$）与表面法线 $\mathbf{n}$ 之间的夹角，入射光线方向（$\mathbf{l}$）和表面法线 $\mathbf{n}$ 之间的夹角，这两个夹角是完全相同的，记为 θ_i。反射向量 $\mathbf{r}_i$ 可以由表面法线 $\mathbf{n}$ 和入射光方向 $\mathbf{l}$ 计算得出：

$$\mathbf{r}_i = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - 1 \tag{9.15}$$

如图 9.19 所示。反射光的光量（入射光的一部分）由菲涅尔反射率（Fresnel reflectance）F 来描述，它取决于入射角度 θ_i。

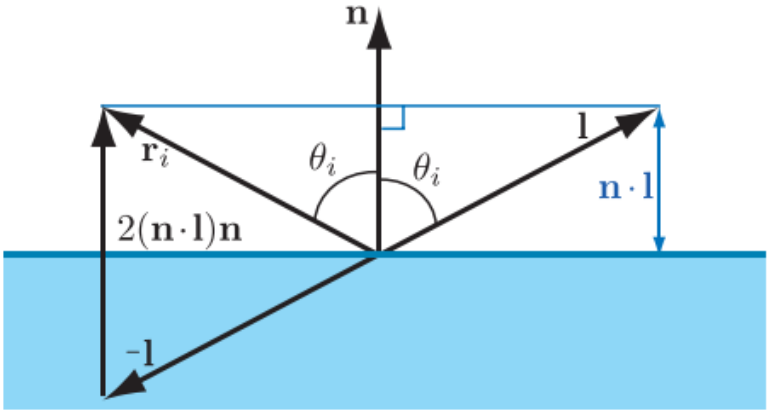


图 9.19：平面反射的示意图。入射光 $\mathbf{l}$ 在表面法线 $\mathbf{n}$ 周围发生反射，从而生成反射光 $\mathbf{r}_i$ 。整个过程如下：首先将入射光 $\mathbf{l}$ 投影到表面法线 $\mathbf{n}$ 上，我们可以得到一个缩放过的法线 $(\mathbf{n}\cdot\mathbf{l})\mathbf{n}$ ；然后我们将入射光 $\mathbf{l}$ 取反，加上两个刚才的投影向量，最终可以获得反射向量 $\mathbf{r}_i$ 。

如章节 9.1.3 所讨论的，反射和折射会受到表面两侧物质折射率的影响，我们将继续使用前面讨论时所用到过的一些符号表示。 n_1 代表了表面“上方”物质的折射率，即入射光和反射光传播的那一侧； n_2 代表了表面“下方”物质的折射率，即折射光传播的那一侧。

菲涅尔方程描述了 F 对入射角度 θ_i 、折射率 n_1 ， n_2 的依赖关系，这些方程本身有些复杂，下面我们将描述菲涅尔反射的重要特征，而不是直接给出其数学表达式。

9.5.1 外反射

外反射（external reflection）是指 $n_1 < n_2$ 的情况，也就是说，光线是从折射率较低的那一侧表面发出来的。在大多数情况下，表面的这一侧都是空气，其折射率约为 1.003。为了简单起见，我们假设 $n_1 = 1$ 。与外反射相反，光线从物体内部传播到空气中的反射过程被称为内反射（internal reflection），稍后将在章节 9.5.3 中进行讨论。

对于一种给定的物质，菲涅尔方程可以被解释为一个仅依赖于入射光角度的反射率函数 $F(\theta_i)$ 。原则上来说， $F(\theta_i)$ 的值会在可见光谱上连续变化，但是出于渲染目的，这个函数的输出会被视为一个 RGB 向量。函数 $F(\theta_i)$ 具有以下特征：

- 当 $\theta_i = 0°$ ，即当光线垂直于表面（ $\mathbf{l} = \mathbf{n}$ ）时， $F(\theta_i)$ 的值反映了物质本身的属性。这个特殊值 F_0 ，可以被认为是物质特有的镜面颜色。我们将 $\theta_i = 0°$ 的情况称为法线反射（normal incidence）（或者是法向入射，垂直入射等）。
- 随着 θ_i 的不断增大，入射光线会以越来越大的掠射角度照射到表面上， $F(\theta_i)$ 的值将不断增加，当 $\theta_i = 90°$ 时，对于任何频率的入射光， $F(\theta_i)$ 都会输出 1（即白色）。

图 9.20 以几种不同的方式可视化展示了几种物质的 $F(\theta_i)$ 函数。这些曲线是高度非线性的——当 θ_i 较小的时候，它们几乎没有变化；直到 $\theta_i = 90°$ 左右，函数值会迅速增长为 1。大部分物质的 $F(\theta_i)$ 函数会从 F_0 单调增长到 1，但是有些特殊物质（例如图 9.20 中的铝）在变白（1）之前会有轻微的下降。

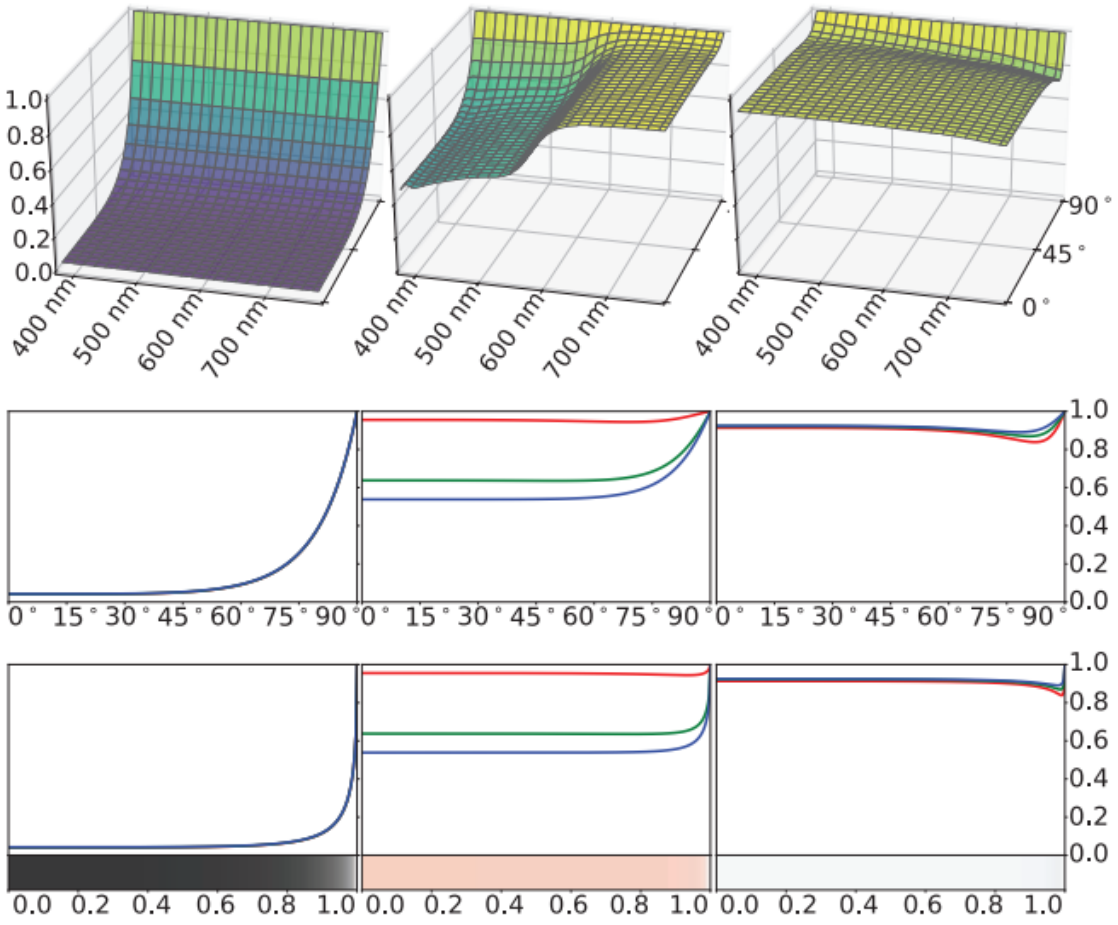


图 9.20：三种物质外反射的菲涅尔反射率 F，从左到右分别是：玻璃、铜、铝。第一行是 F 作为波长和入射角的函数的三维示意图。第二行展示了在不同入射角下，F 的光谱值转换为 RGB 颜色通道的曲线。其中玻璃的三个通道曲线是完全一致的，因为它的菲涅尔反射是无色的。在第三行中，RGB 通道曲线是根据入射角的正弦值进行绘制的，以解释图 9.21 所示的透视缩减。第三行图片中 x 轴还展示了菲涅尔反射的 RGB 颜色。

在镜面反射（mirror reflection）的情况下，光线的出射角与入射角完全相同，这意味着与表面呈掠射角度的入射光线（θ_i 的值接近 90◦），其出射光线也与表面呈掠射角度，并最终进入相机或者眼睛，因此反射率的增加主要会出现在物体的边缘处。此外，从相机的视角来看，表面反射率增加最剧烈的部分会被透视缩短（foreshortened），因此这部分所占用的像素数量相对较少。为了展示菲涅尔曲线的不同部分与其视觉突出程度（prominence）成比例，图 9.22 和图 9.20 的下半部分所展示的菲涅尔反射率曲线和色条，都是使用 $\sin(\theta_i)$ 作为参数进行绘制的，而不是直接使用 θ_i 作为参数。图 9.21 则说明了为什么 $\sin(\theta_i)$ 是一个更加合适的参数选择。

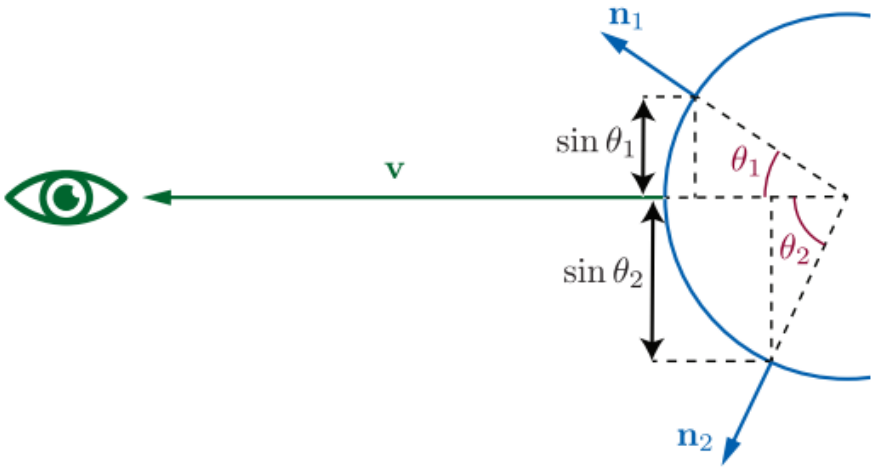


图 9.21：远离眼睛的斜面会被透视缩短。这种透视缩短与按照视角方向 $\mathbf{v}$ 和表面法线 $\mathbf{n}$ 之间夹角的正弦值，来对表面点进行投影是一致的（对于镜面反射而言，这个夹角与入射角相同）。因此，在图 9.20 和图 9.22 中，菲涅尔反射率曲线是根据入射角的正弦值进行绘制的。

从现在开始，为了强调所涉及的向量，我们使用符号 $F(\mathbf{n},\mathbf{l})$ 而不是 $F\left(\theta_i\right)$ 来表示菲涅尔函数。回想一下，θ_i 是表面法线 $\mathbf{n}$ 和入射光方向 $\mathbf{l}$ 之间的夹角。当菲涅尔函数被整合为 BRDF 的一部分时，通常会使用一个不同的向量来代替表面法线 $\mathbf{n}$，详见章节 9.8。

在有关渲染的出版物中，在掠射角度处反射率剧烈增加的现象，通常被称为菲涅尔效应（Fresnel effect）（而在其他领域中，该术语则具有不同的含义，它与无线电波的传输有关）。我们可以通过一个简单的小实验，来亲眼看到菲涅尔效应。带上你的智能手机，坐在光线明亮的地方（比如电脑显示器前）。在不打开手机屏幕显示的情况下，首先将手机靠近胸口，低头看手机，并将手机略微倾斜，使得手机屏幕能够反射出显示器。此时手机屏幕上会有一个相对较弱的显示器反射，这是因为玻璃的法线入射反射率很低。现在将手机向上抬起，使其大致位于眼睛和显示器之间，并再次调整屏幕的角度，使其能够反射出显示器。现在显示器在手机屏幕上的反射，应当几乎与显示器本身一样明亮清晰。

除了非常复杂之外，菲涅尔方程还有一些其他的特性，这些特性使得菲涅尔方程很难直接用于渲染。它需要在可见光谱上采样折射率值，这些值可能会是复数。图 9.20 中的曲线展示了一种基于特征高光颜色 F_0 的简化方法。Schlick [1568]给出了菲涅尔反射率的近似值，如下所示：

$$F(\mathbf{n},\mathbf{l}) \approx F_0 + (1 - F_0)\left(1 - (\mathbf{n}\cdot\mathbf{l})^+\right)^5 \tag{9.16}$$

这个函数实际上就是在白色和 F_0 之间进行 RGB 插值，尽管它很简单，但是这个近似还是相当准确的。

图 9.22 展示了几种偏离 Schlick 曲线的物质，它们（第二行）在变白之前会表现出明显的“下降”。事实上，之所以选择这些物质进行展示，就是因为它们在很大程度上偏离了 Schlick 近似。但是即使对于这些物质而言，Schlick 近似所带来的误差也相当微小，如图中每个曲线下面的颜色条。在极少数情况下，我们可能需要精确模拟这些材料的菲涅尔反射，那么此时可以使用 Gulbrandsen [623]给出的另一种近似方法，这种近似方法可以精确近似金属的菲涅尔方程，但是计算开销要高于 Schlick 近似。一个更加简单的选择是对 Schlick 近似进行修改，使得它可以将最后一项提高到 5 次幂以上（方程 9.18）。这种修改会改变过渡到白色的“速度”，从而使得近似更加精确。Lagarde [959]对菲涅尔方程以及几种近似方法进行了总结。

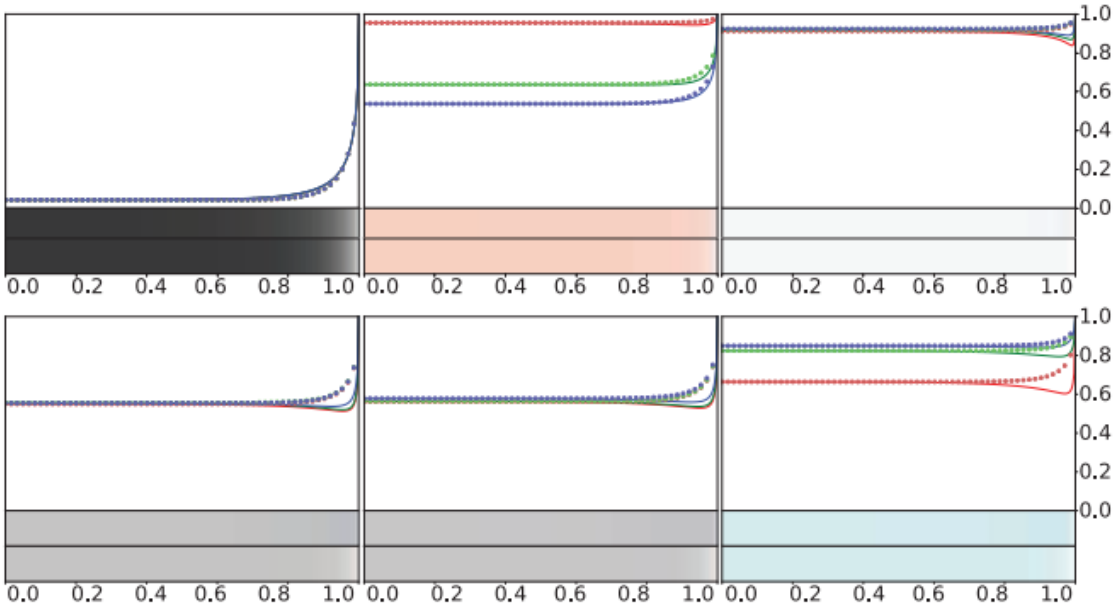


图 9.22：展示了 Schlick 的菲涅耳反射率近似值，与六种物质外反射精确值之间的对比。前三种物质与图 9.20 中所展示的相同，从左到右分别是：玻璃、铜、铝；下面三种物质分别是：铬、铁和锌。每种物质都有一个 RGB 通道曲线图，其中实线代表了精确的菲涅尔方程，虚线代表了 Schlick 近似。每个曲线图下面有两个颜色条，其中上面的颜色条代表了精确菲涅耳方程的结果，下面的颜色条代表了 Schlick 近似的结果。

当我们使用 Schlick 近似时，F_0 是控制菲涅尔反射的唯一参数。这种方式十分方便，因为 F_0 在 $[0, 1]$ 中有一个定义良好的有效值范围，很容易使用标准的颜色选择界面进行设置，并且可以使用为颜色设计的纹理格式来进行纹理化操作。此外，还可以使用许多现实世界材料的属性来作为 F_0 的参考值。F_0 也可以通过折射率计算得

出，通常假设空气折射率的近似值 $n_1 = 1$，用 n 代替 n_2，来表示物体的折射率。简化之后可以获得如下方程：

$$F_0 = \left(\frac{n-1}{n+1}\right)^2 \tag{9.17}$$

如果使用复数结果的话，这个方程甚至适用于复数折射率（例如金属的折射率）。在可见光谱中折射率变化较大的情况下，计算 F_0 的精确 RGB 值，需要首先计算密集采样波长下的 F_0，然后再使用章节 8.1.3 种所描述的方法，将得到的光谱矢量转换为 RGB 值。

在一些应用中[732, 947]使用了 Schlick 近似的更一般形式：

$$F(\mathbf{n}, \mathbf{l}) \approx F_0 + (F_{90} - F_0)\left(1 - (\mathbf{n} \cdot \mathbf{l})^+\right)^{\frac{1}{p}} \tag{9.18}$$

这个方程可以让我们控制菲涅尔曲线在 90° 时所过渡到的颜色，以及过渡的“速度”。使用这种更加一般的形式，通常是为了增加艺术可控性，但是在某些情况下，它也可以对现实进行更加精确的近似。上面我们提到，修改幂次可以使得某些材质更加接近现实。此外，对于一些材质而言，菲涅耳方程无法很好地描述它们，而将 F_{90} 设置为白色以外的颜色，则可以帮助对它们的近似，例如被细尘覆盖的表面，这些细尘颗粒的大小与单个光波的大小相当。

9.5.2 典型的菲涅尔反射值

物质根据其光学特性可以分为三大类，它们分别是电介质（dielectric），也就是绝缘体（insulator）；金属（metal），它们是导体（conductor）；半导体（semiconductor），它的性质介于电介质和金属之间。

电介质的菲涅尔反射率

我们日常生活中遇到的大多数材料都是电介质——玻璃、皮肤、木材、头发、皮革、塑料、石头和混凝土等等。水也是电介质，这可能会让你感到惊讶，因为我们知道，日常生活中的水是可以导电的，但是实际上纯水是不导电的，这种导电性是由于水中含有的各种杂质所产生的。电介质具有相当低的 F_0 值，通常为 0.06 或者更低。法线反射时的低反射率，会使得菲涅尔效应在电介质中尤为明显。电介质的光学特性在可见光谱上的变化很小，从而会导致无色的反射率值。几种常见电介质的 F_0 值如表 9.1 所示，表中的数值都是标量而不是 RGB 值，这是因为这些材质的 RGB 通道并没

有显著的差异。为了方便起见，表 9.1 中包含了线性值，以及用 sRGB 转换函数编码的 8 bit 值（通常在纹理绘制应用中会使用这种形式）。

Dielectric	Linear	Texture	Color	Notes
Water	0.02	39		
Living tissue	0.02–0.04	39–56		Watery tissues are toward the lower bound, dry ones are higher
Skin	0.028	47		
Eyes	0.025	44		Dry cornea (tears have a similar value to water)
Hair	0.046	61		
Teeth	0.058	68		
Fabric	0.04–0.056	56–67		Polyester highest, most others under 0.05
Stone	0.035–0.056	53–67		Values for the minerals most often found in stone
Plastics, glass	0.04–0.05	56–63		Not including crystal glass
Crystal glass	0.05–0.07	63–75		
Gems	0.05–0.08	63–80		Not including diamonds and diamond simulants
Diamond-like	0.13–0.2	101–124		Diamonds and diamond simulants (e.g., cubic zirconia, moissanite)

表 9.1：各种电介质的外反射 F_0 值。每个值都以线性值、纹理值（非线性编码的 8 bit 无符号整数）和色板的形式给出。如果给定的 F_0 值是一个范围，那么色板展示是该范围的中间颜色。回想一下，这些都是镜面颜色，例如：宝石通常具有十分鲜艳的颜色，但是这些颜色是由于物质内部的光线吸收而产生的，与菲涅尔反射无关。

其他电介质的 F_0 值可以通过查看表中的类似物质推断出来。对于未知的电介质而言，0.04 是一个合理的默认值（译者注：metallic 工作流中，非金属的 F_0 值一般为 0.04），这与大多数常见材料的 F_0 值相差不大。

一旦光线进入了电介质内部，那么它就有可能会被进一步散射或者吸收。章节 9.9 将详细讨论这个过程的模型。如果物体的材质是透明的，光线将继续传播，直到“从内部”照射到物体表面，我们将在章节 9.5.3 中进行详细讨论。

金属的菲涅尔反射率

金属的 F_0 值普遍很高，几乎都在 0.5 以上。一些金属的光学特性会在可见光谱上发生较大的变化，从而会产生彩色的反射值。几种金属的 F_0 值如表 9.2 所示。

Metal	Linear	Texture	Color
Titanium	0.542,0.497,0.449	194,187,179	
Chromium	0.549,0.556,0.554	196,197,196	
Iron	0.562,0.565,0.578	198,198,200	
Nickel	0.660,0.609,0.526	212,205,192	
Platinum	0.673,0.637,0.585	214,209,201	
Copper	0.955,0.638,0.538	250,209,194	
Palladium	0.733,0.697,0.652	222,217,211	
Mercury	0.781,0.780,0.778	229,228,228	
Brass (C260)	0.910,0.778,0.423	245,228,174	
Zinc	0.664,0.824,0.850	213,234,237	
Gold	1.000,0.782,0.344	255,229,158	
Aluminum	0.913,0.922,0.924	245,246,246	
Silver	0.972,0.960,0.915	252,250,245	

表 9.2：各种金属（和一种合金）的外反射 F_0 值，按照反射亮度的顺序进行排序。黄金的实际红色分量略微超出了 sRGB 色域，图中展示的是 clamp 后的值。

与表 9.1 类似，表 9.2 中包含了线性值以及用于纹理的 8 bit sRGB 编码值。然而不同之处在于，这里我们给出了 RGB 各个分量的详细数值，因为许多金属都具有彩色的菲涅尔反射。这些 RGB 值是使用 sRGB 原色（与 Rec. 709，电影电视行业所使用的一种色域）和白点定义的。黄金的 F_0 值有些不太寻常，它有着非常强烈的反射颜色，其红色通道值略高于 1（刚刚超出 sRGB/Rec.709 色域），其蓝色通道值则特别低（表 9.2 中唯一一个显著低于 0.5 的）。黄金也是最亮的金属之一，从它在表中的位置可以看出，因为表格是按照亮度进行排序的。黄金明亮且强烈的反射率，可能有助于其在历史上独特的文化意义和经济意义。

回顾一下，金属会立即吸收任何透射光线，因此它们并不会表现出任何的次表面散射或者透明度。金属的所有可见颜色都来自于其 F_0 值。

半导体的菲涅尔反射值

正如人们所料，半导体的 F_0 值介于最亮的电介质和最暗的金属之间，如表 9.3 所示。在实践中我们很少需要渲染这样的物质，因为大多数渲染场景中并没有散落着的晶体硅块。出于实践目的，应当避免使用 0.2–0.45 之间的 F_0 值，除非您有意尝试建模一个奇异的或者非真实的材质。

Substance	Linear	Texture	Color
Diamond	0.171,0.172,0.176	115,115,116	
Silicon	0.345,0.369,0.426	159,164,174	
Titanium	0.542,0.497,0.449	194,187,179	

表 9.3：图中展示了代表性半导体（晶体形式的硅）的 F_0 值，以及与明亮的电介质（金刚石）和较暗的金属（钛）的对比。

水中的菲涅尔反射率

在我们有关外反射的讨论中，我们假设要渲染的表面被空气包围。如果实际情况不是这样的话，那么反射率就会改变，因为它取决于表面两侧折射率的比值。如果我们不能假设 $n_1 = 1$ 的话，则需要将方程 9.17 中的 n 替换为相对折射率 n_1/n_2 。即下面这个更加一般化的方程：

$$F_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{9.19}$$

$n_1 \neq 1$ 最常见的情况就是水下场景，由于水的折射率大约是空气的 1.33 倍，因此水下的 F_0 值是完全不同的，电介质的这种效应要比金属更强，如表 9.4 所示。

Substance	Linear	Texture	Color
Skin (in air)	0.028	47	
Skin (in water)	0.0007	2	
Schott K7 glass (in air)	0.042	58	
Schott K7 glass (in water)	0.004	13	
Diamond (in air)	0.172	115	
Diamond (in water)	0.084	82	
Iron (in air)	0.562,0.565,0.578	198,198,200	
Iron (in water)	0.470,0.475,0.492	182,183,186	
Gold (in air)	1.000,0.782,0.344	255,229,158	
Gold (in water)	1.000,0.747,0.261	255,224,140	
Silver (in air)	0.972,0.960,0.915	252,250,245	
Silver (in water)	0.964,0.950,0.899	251,249,243	

表 9.4：各种物质在空气和水中的 F_0 值的比较。正如方程 9.19 所期望的那样，折射率接近水的电介质所受到的影响最大。相比之下，金属几乎没有受到什么影响。

参数化的菲涅尔值

一个常用的参数化是将镜面颜色（specular color） F_0 和漫反射颜色（diffuse color） ρ_{ss} 结合在一起（漫反射颜色将在章节 9.9 中进一步讨论）。这种参数化利用了以下的观察结果：金属材质没有漫反射颜色，而电介质可能存在的 F_0 值是一个有限集合。这种参数化表示包含一个 RGB 表面颜色 $\mathbf{c}_{surf}$ 和一个标量参数 m，这个参数被称为“metallic”或者“metalness”，即金属度。如果 $m = 1$，则说明该材质是金属，则将 $\mathbf{c}_{surf}$ 设置为 F_0， ρ_{ss} 设置为黑色；如果 $m = 0$，则说明该材质是非金属，则将介电值（常数，或者由其他附加参数进行控制）设置为 F_0， $\mathbf{c}_{surf}$ 设置为 ρ_{ss}。

这个“金属度”参数最初是作为布朗大学（Brown University）[1713]使用的早期着色模型中的一部分出现的；而目前形式的参数化，则首次被皮克斯（Pixar）用于电影《机器人总动员（Wall-E）》[1669]。对于迪士尼动画电影《无敌破坏王（Wreck-It Ralph）》系列中使用的迪士尼原则着色模型，Burley 添加了一个额外的标量“镜面（specular）”参数，来将电介质的 F_0 值控制在一个有限的范围内[214]。虚幻引擎[861]同样使用了这种形式的参数化，而寒霜引擎所使用的形式则有一些不同，它允许电介质使用较大范围的 F_0 值[960]。游戏《使命召唤：无限战争》则使用了一种变体，它将这些金属度参数和镜面参数打包成一个单一的值[384]，从而节省了内存空间。

对于那些使用这种金属度参数化，而不是直接使用 F_0 和 ρ_{ss} 的渲染程序，其动机包括方便用户使用，节省纹理存储空间或者 G-buffer 存储空间等。在游戏《使命召唤：无限战争》中，这种参数化以一种独特的方式进行使用。艺术家为 F_0 和 ρ_{ss} 绘制纹理，程序会自动执行一种压缩方法，将其转换为金属度参数化形式。

使用这种金属度参数化有一些缺点，即它无法表示某些特殊类型的材质，例如一个具有彩色 F_0 值的金属，其表面涂有一层电介质，在金属和电介质之间的边界上，可能会出现瑕疵[960, 1163]。

一些实时应用程序使用了另一种参数化技巧，它基于这样的一个事实：除了特殊的抗反射涂层（anti-reflective coating）之外，没有任何材料的 F_0 值低于 0.02。这个技巧可以用来抑制表面上某些代表空腔或者空洞的镜面高光。 F_0 低于 0.02 的值被用来“关闭”菲涅尔的边缘增亮效果，而不是使用单独的镜面遮挡纹理进行控制。这种技术最初是由 Schuler [1586]提出的，它现在被应用于虚幻引擎[861]和寒霜引擎[960]中。

9.5.3 内反射

虽然外反射在渲染中更加常见，但是内反射有时候也会很重要。当 $n_1 > n_2$ 的时候，就会发生内反射，也就是说，当光线在透明物体的内部传播，并“从内部”打到物体表面上时，就会发生内反射，如图 9.23 所示。

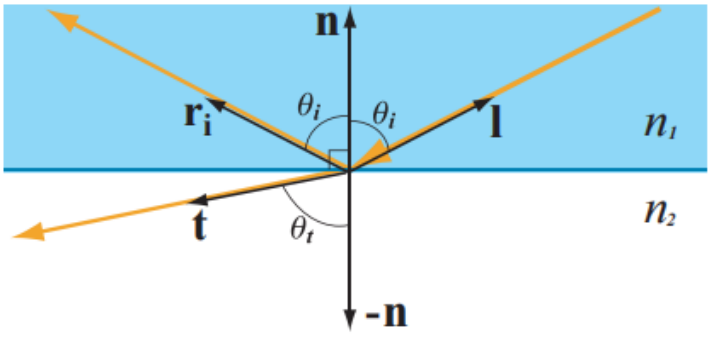


图 9.23：平面上的内反射，其中 $n_1 > n_2$

Snell 定律表明，对于内反射，有 $\sin\theta_t > \sin\theta_i$；由于 θ_t, θ_i 都在 0° – 90° 之间，因此也意味着 $\theta_t > \theta_i$，如图 9.23 所示。而在外反射的情况下，角度大小的情况正好相反（可以与图 9.9 进行比较），二者之间的不同之处是理解内反射和外反射差异的关键所在。在外反射中，对于 0–1 之间的任意一个 $\sin\theta_i$，都存在一个有效的（较小的） $\sin\theta_t$ 值。但是内反射则不是这样的，在内反射中存在一个临界角 θ_c，对于大于这个临界角的入射角 θ_i，根据 Snell 定律我们能推导出表明 $\sin\theta_t > 1$，即 $\theta_t > 90^\circ$，这是不可能发生的。在这种情况下，θ_t 实际上是不存在的，即当 $\theta_i > \theta_c$ 的时候，不会发生光线透射，入射光将会被全部反射。这种现象被称为全内反射（total internal reflection）。

菲涅尔方程是对称的，也就是说入射向量和透射向量是可以相互交换的，并且反射率保持不变。结合 Snell 定律，这种对称性意味着，内反射的 $F(\theta_i)$ 曲线将类似于外反射曲线的“压缩”版本。即内外反射的 F_0 的值是相同的，但是内反射曲线会在临界角 θ_c 处就提前达到完美反射，而外反射曲线则会在 90° 处才达到完美反射，如图 9.24 所示。从曲线中我们还可以看出，内反射的平均反射率更高，这就是为什么在水下看到的气泡，会具有高度反射的银色外观的原因。

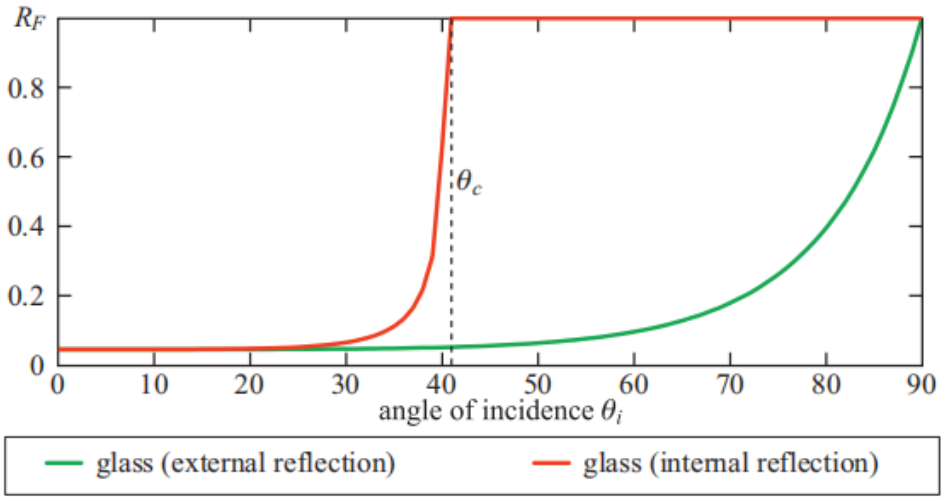


图 9.24：玻璃–空气界面内外反射率曲线的比较。在临界角 θ_c 处，内反射率曲线趋于 1.0。

内反射只会发生在电介质中，因为金属和半导体会迅速吸收在其内部传播的任何光线[285, 286]。由于电介质的折射率都在实数范围内，因此通过折射率或者 F_0 来计算内反射临界角是很简单的：

$$\sin\theta_c = \frac{n_2}{n_1} = \frac{1-\sqrt{F_0}}{1+\sqrt{F_0}} \tag{9.20}$$

方程 9.16 展示的 Schlick 近似，对于外反射而言是正确的，通过使用透射角 θ_t 来代替入射角 θ_i，它也可以用于内反射中。如果此时已经有了透射方向 $\mathbf{t}$（例如：用于渲染折射，详见章节 14.5.2），那么可以用它来计算透射角 θ_t。或者可以使用 Snell 定律，利用入射角 θ_i 来计算透射角 θ_t，但是这个计算成本会很高，并且需要两侧物质的折射率参数，而折射率可能是无法获得的。

9.6 微观几何（Microgeometry）

我们在章节 9.1.3 中讨论过，比单个像素小得多的不规则表面无法被显式建模，因此 BRDF 会转而对它们的总体效应进行统计建模。目前，我们的讨论仍然停留在几何光学领域，它假设这些不规则表面要么小于光的波长（即对光的行为没有影响），要么比光的波长大得多。而有关“波动光学”领域中的不规则表面（大约 1–100 波长大小）的影响，将在章节 9.11 中进行讨论。

每个可见的表面点上都包含了许多微表面法线（microsurface normal），这些法线将入射光向着不同方向进行反射。由于单个微表面的朝向在某种程度上来说是随机

的，因此将它们建模为某种统计分布是十分有道理的。对于大多数表面而言，其微观几何的表面法线是一个连续分布，而在宏观的表面法线处则存在一个很强的峰值。这种分布的“紧密程度”是由表面粗糙度所决定的。表面越粗糙，微观几何表面的法线分布就会越“分散”。

增加微观尺度的粗糙度，其宏观的视觉效果就是反射的环境细节会变得更加模糊。在一个小且明亮的光源下，这种模糊会导致范围更宽、颜色更暗的高光。而那些来自粗糙表面的反射光线则会更暗，因为光线会被发散到了一个更宽的锥形方向上。这一现象可以在图 9.12 中看到。

图 9.25 展示了单个微观尺度表面细节的聚合反射，是如何产生宏观可见反射的。这一组图像展示了一个由单一光源照亮的曲面，每个曲面上的凸起大小逐渐减小，直到最后一张图像中的表面凸起比单个像素还要小得多。许多个微小高光以一种统计模式，最终聚合成了宏观的高光。例如：外围单个高光的相对稀疏程度在聚合之后，就变成了远离高光中心点的相对暗度。

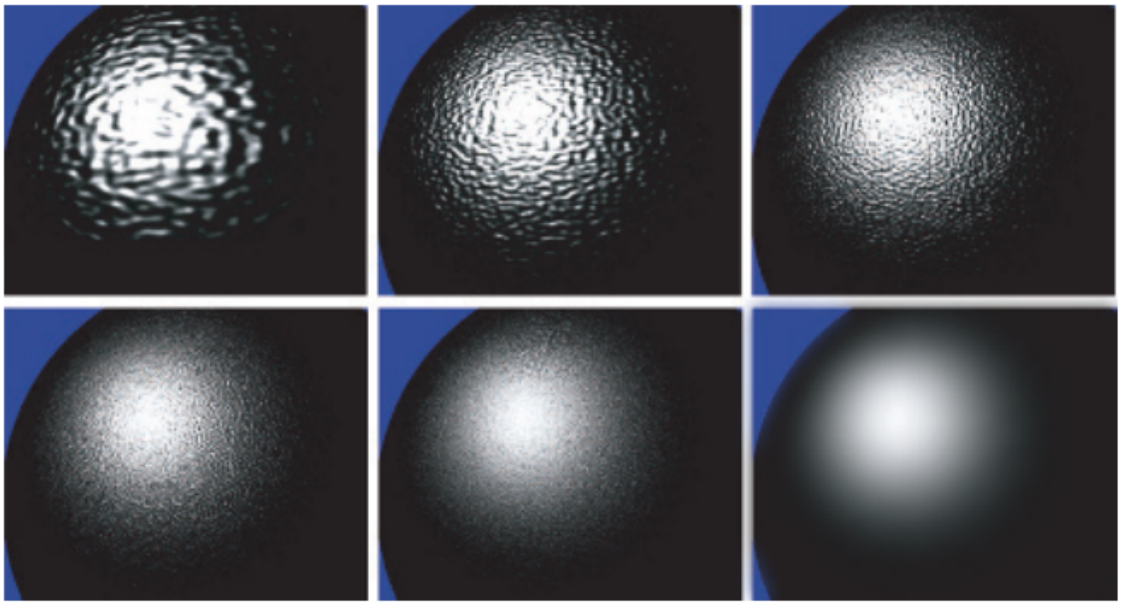


图 9.25：从可见的宏观表面细节逐渐过渡到微观尺度。图像序列按照从上到下，从左到右排布。每个图像中的表面形状和光源都是固定的，只有表面细节的尺度发生了变化。

对于大多数表面而言，微尺度表面法线的分布是各向同性的，这意味着它是旋转对称的，不具备任何固有的方向属性。而有一些表面则具有各向异性（anisotropic）的微观结构，这样的表面具有各向异性的表面法线分布，这会导致反射和高光的定向模糊，如图 9.26 所示。

图 9.26：左边是一个各向异性的表面（拉丝金属），请注意反射的定向模糊。右图是一个相似表面的显微照片，请注意表面细节的方向性。

有一些表面具有高度结构化的微观几何形状，这产生了各种微观尺度的法线分布和表面外观。纤维织物就是一个常见的例子——天鹅绒和绸缎的独特外观，就是由它们的微观几何结构所决定的[78]。有关纤维织物的模型，将在章节 9.10 中进行讨论。

虽然多重表面法线是微观几何对反射率的主要影响因素，但是其他影响因素有时候也很重要。其中遮蔽（shadowing）是指部分微尺度表面细节对光源的遮挡，使得光线无法照射到另一部分表面上，如图 9.27 左侧所示。而遮挡（masking）是指微尺度表面细节对相机的遮挡，使得相机无法看见一些表面，如图 9.27 中间所示。（译者注：后文中就不翻译 shadowing 和 masking 了，容易混淆）

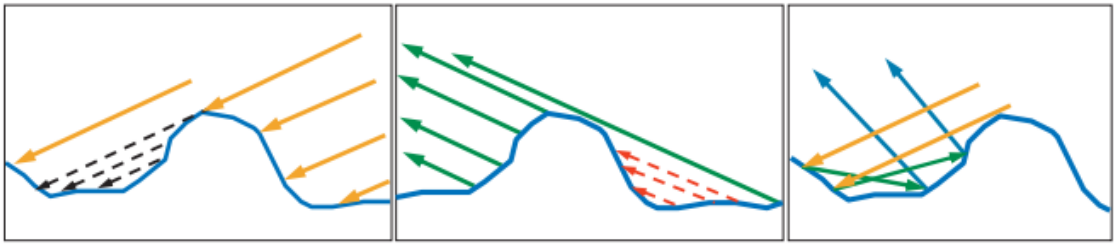


图 9.27：微观结构的几何效应。左侧：黑色虚线箭头代表了被其他微观几何遮蔽（遮挡了光线）的区域。中间：红色虚线箭头表示被其他微观几何遮挡（遮挡相机视线）的区域。右侧：展示了光线在微尺度结构之间的相互反射。

如果微观几何的高度与表面法线之间存在相关性的话，那么 shadowing 和 masking 就能够有效地改变表面的法线分布。例如：想象这样的一个表面，该表面上凸起的部分因为风化或者其他过程而变得十分光滑，而凹陷的部分仍然十分粗糙。在掠射视角下，表面的凹陷部分往往会被 shadowing 或者 masking，从而生成一个十分光滑的表面。如图 9.28 所示。

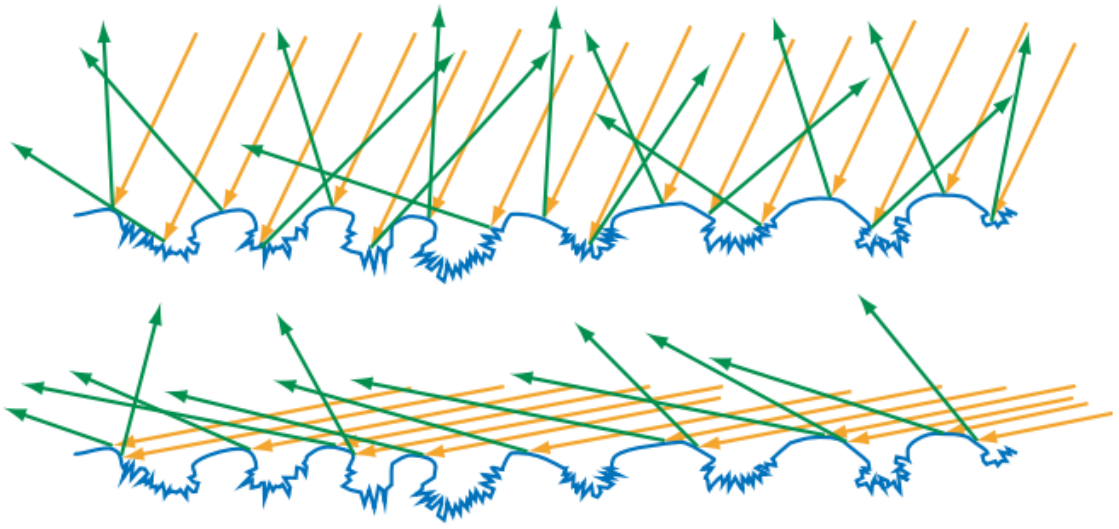


图 9.28：图示的微观几何在高度和表面法线之间具有很强的相关性，其中凸起的区域是光滑的，而凹陷的区域是粗糙的。在上面的图像中，光线以接近宏观表面法线的角度照射到表面上。在这个入射角度上，许多入射光线都可以到达粗糙的凹坑，这些光线会在不同的方向上散射。而在下面的图像中，光线以一个掠射角度照射到表面上。Shadowing 遮蔽了大部分凹坑，因此只有很少的光线能够照射到它们，大多数光线都被表面的光滑部分反射回来。在这种情况下，表面粗糙度很大程度上取决于光线的入射角度。

对于所有的表面类型而言，不规则表面的可见尺寸会随着光线入射角 θ_i 与表面法线之间夹角的增加而减小。在极端的掠射角度下，这种效应可以将观察到的不规则表面尺寸，减小到比光的波长还要短，从而使得这些不规则表面对于光线的传播行为没有任何影响。这两种效应与菲涅尔效应相结合，使得表面看起来具有很高的反射率，就像镜子一样，因为此时的观察角度和光线角度接近 90° [79, 1873, 1874]。

我们可以自己做个小实验。将一张没有光泽的纸卷成一根长管，并将它对准明亮的窗户或者电脑屏幕；当我们的视角几乎与纸张平行的时候，我们会在纸张上看到窗户或者屏幕的尖锐反射。只有这个纸质长管和电脑屏幕（或者明亮的窗户）之间的角度非常接近 90° 时，才能看到这种效果。

被微尺度表面细节遮挡的光线并不会消失，它可能会被反射到其他的微观几何上。光线在最终到达眼睛之前，可能会经历多次这种类型的反射，图 9.27 右侧就展示了这种相互反射现象。由于光线在每次反弹时都会被菲涅尔反射衰减，因此电介质中的相互反射往往都不易察觉（subtle）。而在金属材质中，这种多次弹射是任何可见的漫反射的来源，这是因为金属没有次表面散射现象。有色金属的多次反射颜色要比一次反射的更深，因为它们是光线与表面多次相互作用的结果。

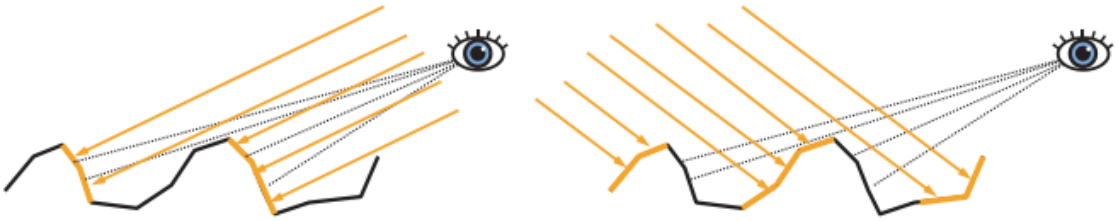


图 9.29：微尺度粗糙度引起的逆反射。左右两幅图都展示了一个粗糙表面，它们具有较低的菲涅尔反射率和较高的漫反射率，因此次表面反射在视觉上就变得十分重要。左图中，观察方向和光照方向是相似的，微观几何中明亮的部分也是最可见的部分，从而形成了明亮的材质外观。右图中，观察方向和光照方向的差别很大，在这种情况下，明亮的区域被遮挡在视野之外，而可见区域则位于阴影中，从而形成了更暗的材质外观。

到目前为止，我们已经讨论了微观几何对镜面反射率（表面反射率）的影响。在某些情况下，微尺度的表面细节也会对次表面反射率产生影响。如果微观几何的不规则性大于次表面散射发生距离的话，那么 shadowing 和 masking 就会导致逆反射（retroreflection）效应，即光线被优先反射回入射方向。这种效应的发生原因是：当观察方向和光照方向相差很大时，shadowing 和 masking 会遮挡被光线照射到的区域，如图 9.29 所示。逆反射往往会使粗糙表面表现出平坦的外观，如图 9.30 所示。

图 9.30：由于微尺度表面的粗糙度，这两个物体表现出了非 Lambertian 的、逆反射的材质外观。

9.7 微表面理论

有关微观几何对反射率影响的数学分析，被称为微表面理论（microfacet theory），许多 BRDF 模型都是建立在这个理论之上的。这个工具最初是由光学界的研究人员提出的[124]，1977 年由 Blinn [159]引入了计算机图形学，1981 年由 Cook 和 Torrance [285]再次引入。该理论的基础是，将微观几何建模为一组微表面的集合。

每个微表面都是平坦的，它具有一个微表面法线 $\mathbf{m}$；这些微表面会根据 micro–BRDF $f_\mu(\mathbf{l}, \mathbf{v}, \mathbf{m})$ 来对光线进行分别反射，将所有微表面的反射率加起来，就是整个表面的 BRDF。通常的做法是将每个微表面都当成一个完美的菲涅尔镜面，这会产生镜面的微表面 BRDF，用于模拟表面反射。当然，其他选择也是可行的，例如：漫反射的 micro–BRDF 可以用于生成一些局部的次表面散射模型[574, 657, 709, 1198, 1337]；衍射的 micro–BRDF 可以用于创建一个结合几何光学和波动光学效果的着色模型[763]。

微表面模型的一个重要性质是微表面法线 $\mathbf{m}$ 的统计分布。这个分布是由表面的法线分布函数（normal distribution function，NDF）所定义的。为了避免与高斯正态分布（Gaussian normal distribution）相混淆，有些文献则使用了术语“法线的分布（distribution of normal）”。这里我们将使用 $D(\mathbf{m})$ 来表示方程中的 NDF 项。

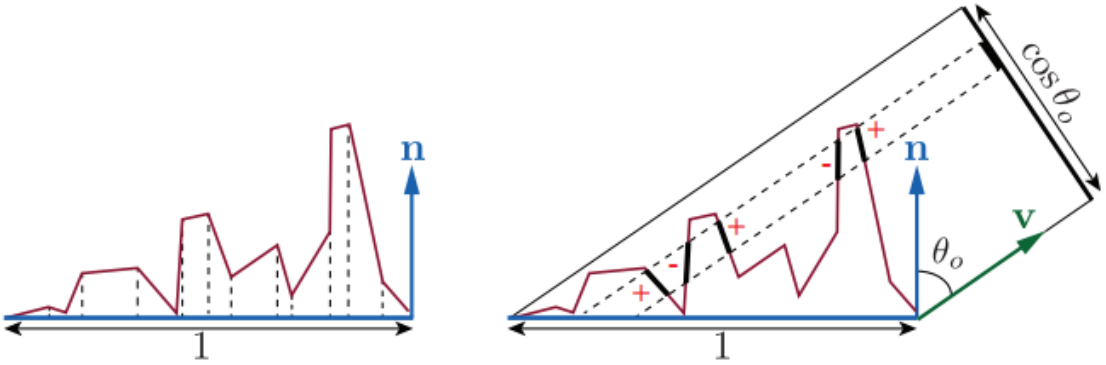


图 9.31：微表面结构的侧视图。左侧：我们可以对 $D(\mathbf{m})(\mathbf{n}\cdot\mathbf{m})$ 进行积分，将微表面区域投影到宏表面平面上，可以得到宏表面的面积（在这个侧视图中则体现为长度），按照约定是 1。右侧：对 $D(\mathbf{m})(\mathbf{v}\cdot\mathbf{m})$ 进行积分，将微表面区域投影到垂直于观察方向 $\mathbf{v}$ 的平面上，可以获得宏表面在该平面上的投影，即 $\cos\theta_o$ 或者 $(\mathbf{v}\cdot\mathbf{n})$。当多个微表面的投影相互重叠的时候，背面微表面的负投影区域会和对应的正面微表面的正投影区域相互抵消。

NDF 项 $D(\mathbf{m})$ 是微表面法线在微观几何区域上的统计分布。在整个微表面法线球面上对 $D(\mathbf{m})$ 进行积分，即可获得这个微表面的面积[708]。更加有用的是，对 $D(\mathbf{m})(\mathbf{n}\cdot\mathbf{m})$ 进行积分，即将 $D(\mathbf{m})$ 投影到宏表面平面上，所得到的宏表面面片的面积为 1（按照约定），如图 9.31 左侧所示。换句话说，投影 $D(\mathbf{m})(\mathbf{n}\cdot\mathbf{m})$ 是归一化的：

$$\int_{\mathbf{m}\in\Theta} D(\mathbf{m})(\mathbf{n}\cdot\mathbf{m})d\mathbf{m} = 1 \tag{9.21}$$

本章节之前的积分，都是在以 $\mathbf{n}$ 为中心的半球内进行的，使用符号 Ω 来表示；而这里是对整个球面的法线进行积分，因此使用符号 Θ 来表示。大多数图形出版物中都使用了这种符号，但是也有少部分参考文献[708]使用了符号 Ω 来表示完整的球体。在实践中，图形学中所使用的微观结构模型大多数是高度场（heightfield），这意味着在 Ω 以外的所有方向 $\mathbf{m}$ 上，$D(\mathbf{m}) = 0$ 。但是，方程 9.21 对于非高度场的微观结构也是有效的。

更一般地说，微表面（microsurface）和宏表面（macrosurface）在垂直于观察方向 $\mathbf{v}$ 的平面上的投影是相等的，即：

$$\int_{\mathbf{m}\in\Theta} D(\mathbf{m})(\mathbf{v}\cdot\mathbf{m})d\mathbf{m} = \mathbf{v}\cdot\mathbf{n} \tag{9.22}$$

方程 9.21 和方程 9.22 中的点积没有限制到 0，图 9.31 的右侧子图解释这个原因。方程 9.21 和方程 9.22 强制要求 $D(\mathbf{m})$ 必须是一个合法的 NDF。

从直观上来看，NDF 就好像是微表面法线的直方图。它在微表面法线更可能指向的方向上，具有更高的值。大多数表面的 NDF 都在宏观表面法线 $\mathbf{n}$ 处，具有很强的峰值。章节 9.8.1 将介绍在渲染中常用的几个 NDF 模型。

让我们再看一下图 9.31 的右侧子图。虽然有许多微表面的投影相互重叠了，但是对于渲染而言，我们只关心能够被相机看见的那些微表面，即在每个重叠集合中最接近相机的那个微表面。根据这个事实，可以提出另一种方法，来将投影后的微表面区域与宏观区域联系起来：可见微表面的投影面积之和 = 宏表面的投影面积。我们可以通过定义遮挡函数（masking function） $G_1(\mathbf{m},\mathbf{v})$ ，以数学的方式来表达这一点，$G_1(\mathbf{m},\mathbf{v})$ 给出了沿观察方向 $\mathbf{v}$ ，具有法线 $\mathbf{m}$ 的可见微表面的比例。$G_1(\mathbf{m},\mathbf{v})D(\mathbf{m})(\mathbf{n}\cdot\mathbf{m})^+$ 的球面积分，则给出了在垂直于观察方向 $\mathbf{v}$ 的平面上，宏表面的投影面积，即：

$$\int_{\in\Theta} G_1(\mathbf{m},\mathbf{v})D(\mathbf{m})(\mathbf{v}\cdot\mathbf{m})^+d\mathbf{m} = \mathbf{v}\cdot\mathbf{n} \tag{9.23}$$

图 9.32 展示了方程所描述的过程。与方程 9.22 有所不同的是，方程 9.23 中的点积被限制到 0，我们在章节 1.2 中介绍过这个操作，使用符号 x^+ 来进行表示。位于背面的微表面是不可见的，所以在这种情况下它们并不会被计算在内。向量 $G_1(\mathbf{m},\mathbf{v})D(\mathbf{m})$ 与可见法线的分布（distribution of visible normal）有关[708]。

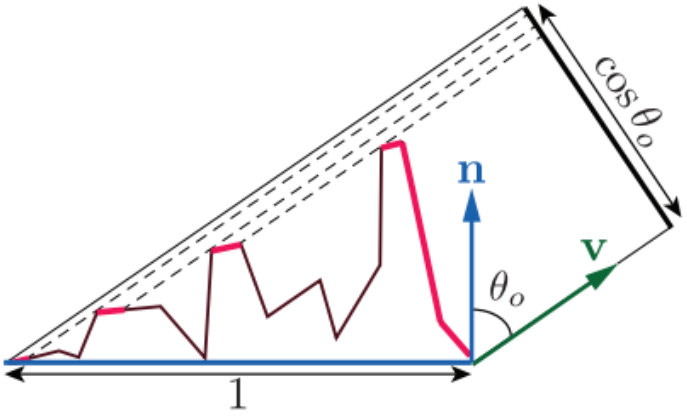


图 9.32：对可见微表面（亮红色）的投影面积进行积分，能够得到垂直于观察方向 $\mathbf{v}$ 的平面上的宏表面投影面积。

虽然方程 9.23 对 $G_1(\mathbf{m}, \mathbf{v})$ 施加了一些约束，但是却无法唯一地决定它。对于给定的微表面法线分布 $D(\mathbf{m})$ [708]，存在无数个 masking 函数满足约束条件。这是因为 $D(\mathbf{m})$ 并不能完全决定微表面的全部细节，它仅仅告诉了我们有多少个微表面的法线指向了这个方向，但是却没有告诉我们这些法线具体的排列方式。

尽管多年来已经提出了各种形式的 G_1 函数，但直到 Heitz 的一篇优秀论文[708]，才解决了（至少目前）到底使用哪种函数这一困境（dilemma）。Heitz 讨论了 Smith masking 函数，这个函数最初是为高斯正态分布而推导出来的[1665]，后来才推广到了任意的 NDF 上[202]。Heitz 表明，在文献中提到的各种 masking 函数中，只有 Smith 函数和 Torrance–Sparrow“V–cavity”函数[1779]服从方程 9.23，即在数学上是有效合法的。他进一步表明，Smith 函数要比 Torrance–Sparrow 函数更加接近随机微表面的表现。Heitz 还证明了 Smith masking 函数是唯一一个可能既服从方程 9.23，又具有法线–masking 独立性的函数。这意味着函数 $G_1(\mathbf{m}, \mathbf{v})$ 并不依赖于微表面法线 $\mathbf{m}$ 的方向，只要这个 $\mathbf{m}$ 不朝向背面即可，即只要 $\mathbf{m} \cdot \mathbf{v} \geq 0$ 即可。Smith G_1 函数的数学形式如下：

$$G_1(\mathbf{m}, \mathbf{v}) = \frac{\chi^+(\mathbf{m} \cdot \mathbf{v})}{1 + \Lambda(\mathbf{v})} \tag{9.24}$$

其中 χ^+ 是正特征函数：

$$\chi^+(x) = \begin{cases} 1, & \text{where } x > 0, \\ 0, & \text{where } x \leq 0. \end{cases} \tag{9.25}$$

每个 NDF 的 Λ （lambda）函数都是不同的。Walter 等人[1833]和 Heitz [708]在出

版物中，描述了为给定 NDF 推导 Λ 函数的过程。

Smith masking 函数确实存在一些缺点。从理论的角度来看，它的函数需求与实际的表面结构并不一致[708]，甚至可能在物理上无法实现[657]。从实践的角度来看，虽然它对随机表面的模拟是相当准确的，但是对于法线方向和 masking 之间具有较强依赖性的表面（如图 9.28 所展示的表面），尤其是当该表面具有一些重复结构的时候（例如大多数织物），Smith masking 函数的准确性预计会下降。然而，在找到更好的替代方案之前，它还是大多数渲染程序的最佳选择。

对于一个给定的微观几何描述，它包括以下几部分：micro-BRDF 项 $f_\mu(\mathbf{l},\mathbf{v},\mathbf{m})$，法线分布函数 $D(\mathbf{m})$，以及遮挡函数 $G_1(\mathbf{m},\mathbf{v})$。据此可以推导出宏表面的 BRDF [708, 1833]，即：

$$f(\mathbf{l},\mathbf{v}) = \int_{\mathbf{m}\in\Omega} f_\mu(\mathbf{l},\mathbf{v},\mathbf{m}) G_2(\mathbf{l},\mathbf{v},\mathbf{m}) D(\mathbf{m}) \frac{(\mathbf{m}\cdot\mathbf{l})^+}{|\mathbf{n}\cdot\mathbf{l}|} \frac{(\mathbf{m}\cdot\mathbf{v})^+}{|\mathbf{n}\cdot\mathbf{v}|} d\mathbf{m}. \tag{9.26}$$

这个积分在以法线 $\mathbf{n}$ 为中心的半球 Ω 上进行的，从而避免计算从表面下方照射来的光线贡献。方程 9.26 使用了联合 masking-shadowing 函数（joint masking-shadowing function） $G_2(\mathbf{l},\mathbf{v},\mathbf{m})$，来代替遮挡函数 $G_1(\mathbf{m},\mathbf{v})$。这个函数是从 G_1 中推导出来的，它给出了在两个方向上具有法线 $\mathbf{m}$ 的可见微表面的比例，这两个方向分别是观察方向 $\mathbf{v}$ 和光线方向 $\mathbf{l}$。通过引入 G_2 函数，方程 9.26 使得 BRDF 能够同时考虑 shadowing 和 masking 对着色的影响，但是它并没有考虑微表面之间的相互反射情况（详见图 9.27）。从方程 9.26 推导出的所有 BRDF 都有一个共同的局限性，即没有考虑微表面之间的相互反射，这样的 BRDF 看起来要比真实情况暗一些。在章节 9.8.2 和章节 9.9 中，我们将讨论为解决这一局限性而提出的一些方法。

Heitz [708]讨论了 G_2 函数的几个版本，其中最简单的是可分离形式，其中的 shadowing 和 masking 分别使用 G_1 函数进行评估，并将计算结果相乘：

$$G_2(\mathbf{l},\mathbf{v},\mathbf{m}) = G_1(\mathbf{v},\mathbf{m}) G_1(\mathbf{l},\mathbf{m}). \tag{9.27}$$

这种形式的 G_2 函数相当于假设 shadowing 和 masking 是两个不相关的事件，但是在现实中它们并非如此，并且这种假设会导致使用该 G_2 函数的 BRDF 过暗。

举一个极端的例子，当观察方向与光线方向相同时，在这种情况下，G_2 应当等于 G_1，因为没有任何一个可见的微表面被 shadowing，但是根据方程 9.27，G_2 将等于 G_1^2。

如果微表面是一个高度场的话（在渲染中通常会使用高度场来作为微表面模型），那么每当观察方向 $\mathbf{v}$ 和光线方向 $\mathbf{l}$ 之间的相对方位角 ϕ 等于 0° 时（如图 9.17 中的 ϕ ），$G_2(\mathbf{l},\mathbf{v},\mathbf{m})$ 应当等于 $\min(G_1(\mathbf{v},\mathbf{m}),G_1(\mathbf{l},\mathbf{m}))$ 。这种关系提出了一种通用的方法，来解释 shadowing 和 masking 之间的关系，并且可以应用于任何 G_1 函数：

$$\begin{aligned} G_2(\mathbf{l},\mathbf{v},\mathbf{m}) = \lambda(\phi)G_1(\mathbf{v},\mathbf{m})G_1(\mathbf{l},\mathbf{m}) \\ + (1-\lambda(\phi))\min\left(G_1(\mathbf{v},\mathbf{m}),G_1(\mathbf{l},\mathbf{m})\right) \end{aligned} \tag{9.28}$$

其中 $\lambda(\phi)$ 是某种随着角 ϕ 的增大，从 0 逐渐增加到 1 的函数。Ashikhmin 等人[78]建议采用标准差为 15° （约为 0.26 弧度）的高斯分布：

$$\lambda(\phi) = 1 - e^{-7.3\phi^2} \tag{9.29}$$

van Ginneken 等人[534]提出了一种不同的 λ 函数：

$$\lambda(\phi) = \frac{4.41\phi}{4.41\phi + 1} \tag{9.30}$$

无论观察方向和光线方向是否对齐，在给定表面点上的 shadowing 和 masking 仍然是相关的，这还有另一个原因，即二者都与表面点相对于曲面其他部分的高度有关。对于高度较低的表面点而言，它被 shadowing 或者 masking 的概率都会增加。如果使用 Smith masking 函数的话，那么二者的相关性可以用 Smith 高度相关的 masking–shadowing 函数来精确地解释：

$$G_2(\mathbf{l},\mathbf{v},\mathbf{m}) = \frac{\chi^+(\mathbf{m}\cdot\mathbf{v})\chi^+(\mathbf{m}\cdot\mathbf{l})}{1+\Lambda(\mathbf{v})+\Lambda(\mathbf{l})} \tag{9.31}$$

Heitz 同样描述了一种结合了方向相关性和高度相关性的 Smith G_2 函数：

$$G_2(\mathbf{l},\mathbf{v},\mathbf{m}) = \frac{\chi^+(\mathbf{m}\cdot\mathbf{v})\chi^+(\mathbf{m}\cdot\mathbf{l})}{1+\max(\Lambda(\mathbf{v}),\Lambda(\mathbf{l}))+\lambda(\mathbf{v},\mathbf{l})\min(\Lambda(\mathbf{v}),\Lambda(\mathbf{l}))} \tag{9.32}$$

其中的函数 $\lambda(\mathbf{v},\mathbf{l})$ 可以是一个经验函数，如方程 9.29 和方程 9.30 所描述的函数，也可以是一个专门为给定的 NDF 推导出来的函数[707] 。

在这些替代方案中，Heitz [708]推荐了 Smith 函数的高度相关形式（方程 9.31），因为它与高度不相关形式的 Smith 函数，具有相似的开销和更好的准确性。这种形式在实践中使用得最为广泛[861, 947, 960]，其他一些从业者使用了可分离形式的 Smith 函数（方程 9.27）[214, 1937]。

方程 9.26 中所描述的微表面 BRDF 并不会直接用于渲染，而是在给定 micro-BRDF f_μ 的情况下，推导出一个封闭形式的解（精确解或者近似解）。我们将在下一小节中，展示这种派生类型的第一个例子。

9.8 表面反射的 BRDF 模型

除了少数例外情况，在基于物理的渲染中所使用的镜面 BRDF 项，都来源于微表面理论。在镜面反射的情况下，每个微表面都是一个完美光滑的菲涅尔镜面，回顾一下，这样的镜面会将每束入射光线，都反射到一个单一的反射方向上。这就意味着除非观察方向 $\mathbf{v}$ 刚好与光线的出射方向平行，不然每个表面的 micro-BRDF $f_\mu(\mathbf{l}, \mathbf{v}, \mathbf{m})$ 都等于零。而对于给定的观察方向 $\mathbf{v}$ 和光线入射方向 $\mathbf{l}$，这种情况相当于微表面的法线 $\mathbf{m}$，与 $\mathbf{v}$ 和 $\mathbf{l}$ 的中间向量对齐，这个中间向量被称为半向量（half vector） $\mathbf{h}$，如图 9.33 所示。半向量 $\mathbf{h}$ 是通过将观察方向 $\mathbf{v}$ 和光线入射方向 $\mathbf{l}$ 相加，并将结果进行归一化计算而来的：

$$\mathbf{h} = \frac{1 + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|} \tag{9.33}$$

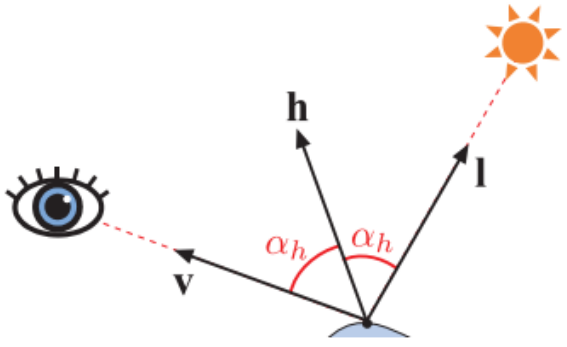


图 9.33：半向量 $\mathbf{h}$ 与观察方向 $\mathbf{v}$ 和光线入射方向 $\mathbf{l}$ 的夹角（用红色表示）相等。

菲涅尔镜面的 micro-BRDF $f_\mu(\mathbf{l}, \mathbf{v}, \mathbf{m})$ 对于所有 $\mathbf{m} \neq \mathbf{h}$ 时的情况都为零，当我们从方程 9.26 中推导高光微表面模型的时候，这个事实是十分方便的，因为它将原本的积分形式退化为了在 $\mathbf{m} = \mathbf{h}$ 处，对被积函数进行求值。这样就做可以得到高光 BRDF 项：

$$f_{\text{spec}}(\mathbf{l},\mathbf{v}) = \frac{F(\mathbf{h},\mathbf{l})G_2(\mathbf{l},\mathbf{v},\mathbf{h})D(\mathbf{h})}{4|\mathbf{n}\cdot\mathbf{l}||\mathbf{n}\cdot\mathbf{v}|} \tag{9.34}$$

有关这个推导的详细过程，可以在 Walter 等人[1833]、Heitz [708]和 Hammon [657]的出版物中找到。Hammon 还给出了一种 BRDF 实现的优化方法，即在不计算半向量 $\mathbf{h}$ 本身的情况下，通过直接计算 $\mathbf{n}\cdot\mathbf{l}$ 和 $\mathbf{n}\cdot\mathbf{v}$ 来降低计算量。

我们使用符号 f_{spec} 来表示方程 9.34 中的 BRDF 项，代表了它模拟了表面反射（镜面反射）现象。在一个完整的 BRDF 实现中，它可能还会加上一个用于模拟次表面（漫反射）着色的额外项。

为了更加直观地理解方程 9.34，我们可以这样认为：只有那些法线与半向量碰巧对齐（$\mathbf{m} = \mathbf{h}$）的微表面，才能正确地将光线从入射方向 $\mathbf{l}$ 反射到观察方向 $\mathbf{v}$ 上，如图 9.34 所示。因此，反射光线的数量取决于微表面法线 $\mathbf{m}$ 与半向量 $\mathbf{h}$ 的对齐程度，这个值是通过 $D(\mathbf{h})$ 项给出的；$G_2(\mathbf{l},\mathbf{v},\mathbf{h})$ 给出了从光线入射方向和相机观察方向都可见的微表面比例；$F(\mathbf{h},\mathbf{l})$ 给出了每个微表面反射光线的比例。这里在计算菲涅尔函数的时候，使用了半向量 $\mathbf{h}$ 来代替表面法线，例如计算方程 9.16 中的 Schlick 近似函数时。

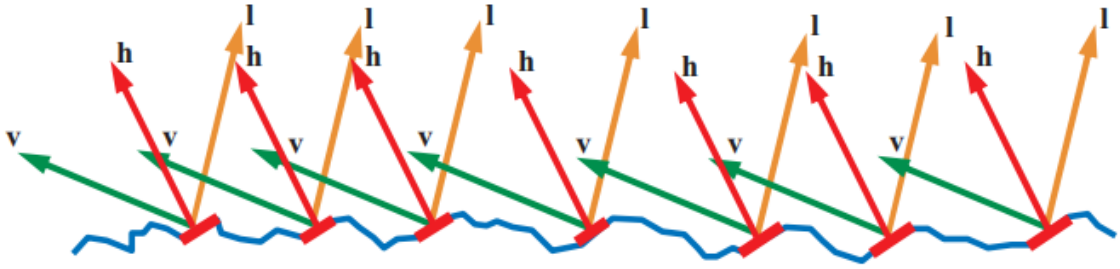


图 9.34：由微表面构成的表面。只有微表面法线与半向量 $\mathbf{h}$ 对齐的那部分（图中红色的微表面），才会参与从入射方向 $\mathbf{l}$ 到观察方向 $\mathbf{v}$ 的光线反射。

在 masking–shadowing 函数中使用的半向量还可以稍微简化一下，由于所涉及的角度永远不会超过 $90°$，因此可以移除方程 9.24，方程 9.31，方程 9.32 中的 χ^+ 项。

9.8.1 法线分布函数

法线分布函数（normal distribution function，NDF）对于要渲染的表面外观具有十分重要的影响。NDF 的形状（将其绘制在微平面法线的球体上）决定了反射光锥（镜面波瓣）的宽度和形状，反过来又决定了高光的大小和形状。NDF 影响了表面

粗糙度的整体感知，以及一些更加微妙的视觉表现，例如高光是否具有明显的边缘，还是具有一个模糊（haze）的边缘。

然而，镜面波瓣（specular lobe）并不是 NDF 形状的简单复制。镜面波瓣（也就是高光形状）被扭曲的程度还取决于表面曲率和观察视角。如图 9.35 所示，当从掠射视角观察平面的时候，这种畸变（distortion）会尤其强烈。Ngan 等人[1271]对这种畸变背后的原因进行了分析。

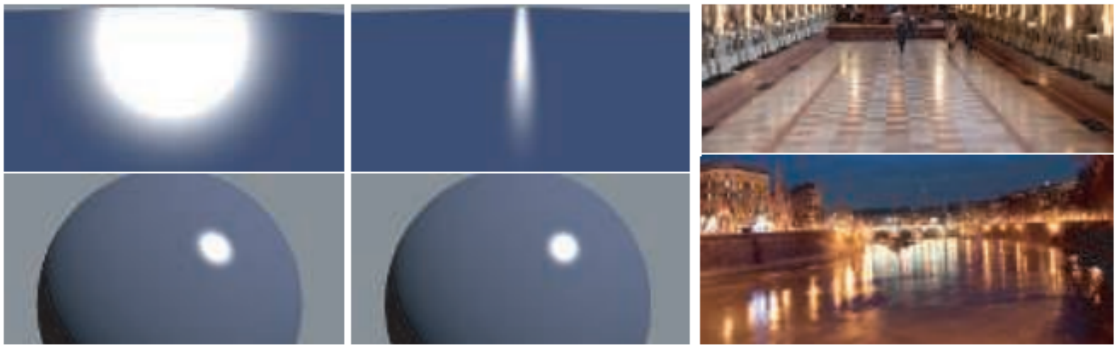


图 9.35：最左边的图像是使用非物理的 Phong 反射模型渲染而来的，这个模型的镜面波瓣关于光线的反射向量旋转对称，类似这种的 BRDF 在早期的计算机图形学中被经常使用。中间的图像使用基于物理的微表面 BRDF 进行渲染。第一行左侧和第一行中间的图片，展示了一个以掠射角度进行光照的平面，其中第一行左侧展示了一个不正确的圆形高光，而第一行中间展示的是微表面 BRDF 所特有的延伸高光，这个效果是与现实相符的，如右边的照片所示。高光形状的差异在下面两个渲染球体上则要不易察觉得多，因为在这种球面的情况下，表面曲率才是是高光形状的主要影响因素。

各项同性法线分布函数

在渲染中所使用的大多数 NDF 都是各向同性的（isotropic），即关于宏表面法线 $\mathbf{n}$ 旋转对称。在这种情况下，NDF 的变量只有一个，即宏表面法线 $\mathbf{n}$ 与微表面法线 $\mathbf{m}$ 之间的夹角 θ_m 。在理想情况下，NDF 可以写成一个关于 $\cos\theta_m$ 的表达式，可以通过计算 $\mathbf{n}$ 和 $\mathbf{m}$ 之间的点乘来快速获得 $\cos\theta_m$ 。

Beckmann NDF [124]是光学界开发的、第一个微表面模型中所使用的法线分布函数，它至今仍被该社区广泛使用。它也是 Cook–Torrance BRDF 所选择使用的 NDF [285, 286]。归一化 Beckmann 分布的数学形式如下：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})}{\pi\alpha_b^2(\mathbf{n}\cdot\mathbf{m})^4}\exp\left(\frac{(\mathbf{n}\cdot\mathbf{m})^2-1}{\alpha_b^2(\mathbf{n}\cdot\mathbf{m})^2}\right) \tag{9.35}$$

其中的 $\chi^+(\mathbf{n}\cdot\mathbf{m})$ 项确保了所有指向宏表面下方的微表面法线的 NDF 值为 0，这个性质告诉我们，与我们将在本小节所讨论的其他所有 NDF 一样，这个 NDF 描述了一个高度场微表面。方程中的参数 α_b 控制了表面的粗糙度，它与微观几何表面的均方根（root mean square，RMS）斜率成正比，即 $\alpha_b = 0$ 代表了一个完全光滑的表面。

为了推导出 Beckmann NDF 的 Smith G_2 函数，我们需要相应的 Λ 函数，然后将其代入方程 9.24（使用可分离形式的 G_2 函数），或者方程 9.31（使用高度相关形式的 G_2 函数），或者方程 9.32（使用方向相关和高度相关的 G_2 函数）。

Beckmann NDF 是形状不变的（shape-invariant），这简化了 Λ 函数的推导。根据 Heitz [708]的定义，如果其粗糙度参数的影响相当于对微表面进行缩放（或者拉伸）的话，那么这个各向同性 NDF 就是形状不变的，形状不变的 NDF 可以写成如下形式：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})}{\alpha^2(\mathbf{n}\cdot\mathbf{m})^4} g\left(\frac{\sqrt{1-(\mathbf{n}\cdot\mathbf{m})^2}}{\alpha(\mathbf{n}\cdot\mathbf{m})}\right) \tag{9.36}$$

其中的 g 代表了一个任意的单变量函数。对于一个任意的各向同性 NDF，Λ 函数取决于两个变量：第一个变量是粗糙度 α，第二个变量是计算 Λ 的向量入射角（$\mathbf{v}$ 或者 $\mathbf{l}$）。然而，对于形状不变的 NDF 而言，Λ 函数只依赖于变量 a：

$$a = \frac{\mathbf{n}\cdot\mathbf{s}}{\alpha\sqrt{1-(\mathbf{n}\cdot\mathbf{s})^2}} \tag{9.37}$$

其中的向量 $\mathbf{s}$ 代表了观察方向 $\mathbf{v}$ 或者光线方向 $\mathbf{l}$，在这种情况下，Λ 函数只依赖于一个变量，这十分利于实现，因为单变量函数可以更加容易地使用近似曲线来进行拟合，并且可以在一维数组中打表。

Beckmann NDF 的 Λ 函数为：

$$\Lambda(a) = \frac{\mathrm{erf}(a)-1}{2} + \frac{1}{2a\sqrt{\pi}}\exp\left(-a^2\right) \tag{9.38}$$

计算方程 9.38 的开销很高，因为它包含了一个误差函数 erf。因此，通常会使用近似值[1833]来进行代替：

$$\Lambda(a) \approx \begin{cases} \frac{1-1.259a+0.396a^2}{3.535a+2.181a^2}, & \text{where } a < 1.6, \\ 0, & \text{where } a \geq 1.6. \end{cases} \tag{9.39}$$

下一个我们要讨论的 NDF 是 Blinn–Phong NDF，它在过去被广泛应用于计算机图形学中，尽管在最近，它在很大程度上已经被其他的法线分布所取代了。Blinn–Phong NDF 仍然被用于一些算力受限的情况（例如移动设备上），因为相比于本小节所讨论的其他 NDF，Blinn–Phong NDF 具有更低的计算成本。

Blinn–Phong NDF 是由 Blinn [159]提出的，作为对 Phong 着色模型（非基于物理的）的一个修改[1414]：

$$D(\mathbf{m}) = \chi^+(\mathbf{n} \cdot \mathbf{m}) \frac{\alpha_p + 2}{2\pi} (\mathbf{n} \cdot \mathbf{m})^{\alpha_p} \tag{9.40}$$

方程中的幂次 α_p 是 Phong NDF 的粗糙度参数，较高的 α_p 代表了较为光滑的表面，而较低的 α_p 则代表较为粗糙的表面。我们可以将 α_p 的值设置得任意高，从而创建一个极其光滑的表面，而一个完美的镜面则需要令 $\alpha_p = \infty$ 。通过将 α_p 设置为 0，可以得到一个随机程度最大的曲面（即均匀 NDF）。参数 α_p 不便于直接进行操作，因为它的视觉效果是高度非均匀的：当 α_p 较小时，它对数值变化十分敏感，一点点很小的数值变化，就会在视觉效果上产生很大的变化；当 α_p 较大时，它对数值变化则不太敏感，一个很大的数值变化，才能让视觉效果发生一点点变化。因此，α_p 通常是通过一个非线性映射，从用户控制的参数推导出来的，例如 $\alpha_p = m^s$ ，其中 s 是一个介于 0–1 之间的参数， m 是 α_p 在给定应用程序中的最大上界。许多游戏都使用了这种映射方式，例如《使命召唤：黑色行动》，其中的 m 被设置为 8192 [998]。

当 BRDF 参数的行为在感知上不均匀时，这种“接口映射（interface mappings）”的方式通常是十分有用的。用户通过滑动条（slider）进行设置，或者是直接在纹理中绘制参数，这些映射可以用于对这些用户参数进行解释。

通过等价变换 $\alpha_p = 2\alpha_b^{-2} - 2$ [1833]，可以找到 Beckmann 和 Blinn–Phong 粗糙度参数的等效值。当参数通过这种方式进行匹配时，这两个分布会非常接近，尤其是对于相对光滑的表面而言，如图 9.36 左上角所示。

Blinn–Phong NDF 并不是形状不变的，其 Λ 函数不存在解析形式。Walter 等人[1833]建议将 Beckmann Λ 函数与等价变换 $\alpha_p = 2\alpha_b^{-2} - 2$ 结合起来使用。

在 1977 年的同一篇论文中[159]，通过将 Phong 着色函数应用到一个微表面 NDF 中，Blinn 提出了另外两种 NDF。在这三种法线分布中，Blinn 建议使用 Trowbridge 和 Reitz [1788]所推导出来的一种分布，但是这个建议并没有得到广泛的重视，直到 30 年后，Trowbridge–Reitz 分布被 Walter 等人[1833]独立地重新发现，并将其命名为 GGX 分布。这一次，种子终于生根发芽了。在几年内，GGX 分布开始在电影行业[214, 1133]和游戏行业[861, 960]中快速普及，而如今它可能是这两个行业中最常用的法线分布，Blinn 的建议似乎超前了 30 年。虽然严格来说，“Trowbridge–Reitz 分布”才是这个分布正确的名称，但是我们在本书中仍然会使用 GGX 这个名称，因为它已经十分成熟并且广为流传了。

GGX 分布的数学表达形式如下：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})\alpha_g^2}{\pi\left(1+(\mathbf{n}\cdot\mathbf{m})^2\left(\alpha_g^2-1\right)\right)^2} \tag{9.41}$$

参数 α_g 提供了与 Beckmann 分布中参数 α_b 类似的粗糙度控制。在 Disney 原则着色模型中，Burley [214]通过 $\alpha_g = r^2$ ，将粗糙度控制暴露给用户，其中 r 是用户可以控制的粗糙度参数，其取值范围在 0–1 之间。将 r 作为一个滑动条值暴露出来，意味着其视觉效果的变化更加线性，大多数使用 GGX 分布的应用都采用了这种映射方式。

GGX 分布是形状不变的，其 Λ 函数相对比较简单：

$$\Lambda(a) = \frac{-1+\sqrt{1+\frac{1}{a^2}}}{2} \tag{9.42}$$

在方程 9.42 中，变量 a 只会以 a^2 的形式出现，这是十分便于计算的，因为这样我们避免了方程 9.37 中的平方根运算。

由于 GGX 分布和 Smith masking–shadowing 函数的流行，因此人们一直致力于优化这两者的组合。Lagarde 观察到[960]，在 GGX 中使用高度相关的 Smith G_2 函数（方程 9.31），在与镜面微表面 BRDF 的分母（方程 9.34）结合时，有几项会被消掉，合并后的项可以简化为如下形式：

$$\frac{G_2(\mathbf{l},\mathbf{v})}{4|\mathbf{n}\cdot\mathbf{l}||\mathbf{n}\cdot\mathbf{v}|} = \frac{0.5}{\mu_o\sqrt{\alpha^2+\mu_i\left(\mu_i-\alpha^2\mu_i\right)}+\mu_i\sqrt{\alpha^2+\mu_o\left(\mu_o-\alpha^2\mu_o\right)}} \tag{9.43}$$

为了简化表达，这个方程使用了以下两个变量替换： $\mu_i = (\mathbf{n} \cdot \mathbf{l})^+$ 和 $\mu_o = (\mathbf{n} \cdot \mathbf{v})^+$ 。Karis [861]提出了在 GGX 中使用的 Smith G_1 函数的近似形式：

$$G_1(\mathbf{s}) \approx \frac{2(\mathbf{n} \cdot \mathbf{s})}{(\mathbf{n} \cdot \mathbf{s})(2 - \alpha) + \alpha}, \tag{9.44}$$

方程 9.44 中的 $\mathbf{s}$ ，可以用向量 $\mathbf{l}$ 或者向量 $\mathbf{v}$ 进行替换。Hammon 表明[657]，这种近似形式的 G_1 函数，可以对由高度相关的 Smith G_2 函数和镜面微表面 BRDF 分母组成的组合项进行有效地近似：

$$\frac{G_2(\mathbf{l}, \mathbf{v})}{4|\mathbf{n} \cdot \mathbf{l}||\mathbf{n} \cdot \mathbf{v}|} \approx \frac{0.5}{\mathrm{lerp}(2|\mathbf{n} \cdot \mathbf{l}||\mathbf{n} \cdot \mathbf{v}|, |\mathbf{n} \cdot \mathbf{l}| + |\mathbf{n} \cdot \mathbf{v}|, \alpha)}, \tag{9.45}$$

方程 9.45 使用了线性插值运算符，即 $\mathrm{lerp}(x, y, s) = x(1 - s) + ys$ 。

当比较图 9.36 中的 GGX 分布和 Beckmann 分布时，很明显二者具有完全不同的形状。GGX 的峰值比 Beckmann 更窄，而围绕这些峰值的“尾巴”则更长。在图 9.36 底部的渲染图中，我们可以看到，GGX 较长的拖尾在高光的核心区域周围，产生了一个模糊（haze）或者微光（glow）外观。

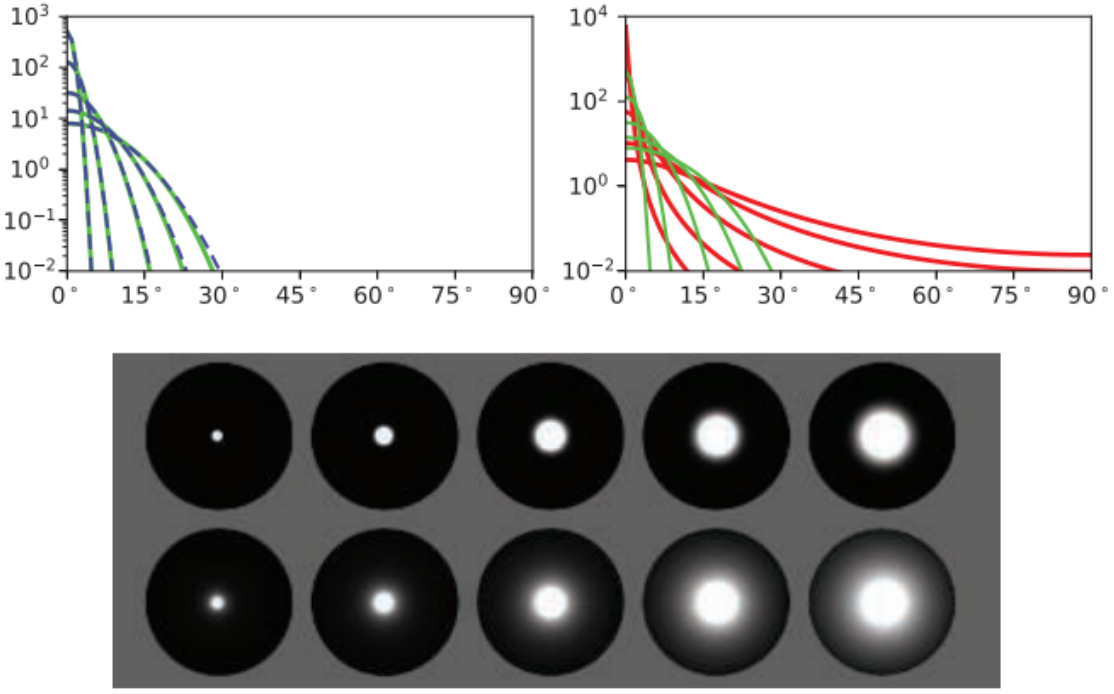


图 9.36：左上角：Blinn–Phong 分布（蓝色虚线）和 Beckmann 分布（绿色实线）的对比图，参数 α_b 在 0.025–0.2 范围内，并使用了参数关系 $\alpha_p = 2\alpha_b^{-2} - 2$ 进行粗糙度的转换。右上角：GGX 分布（红色实线）和 Beckmann 分布（绿色实线）的对比图，其中参数 α_b 的值与左图中的完全相同。通过人眼调节参数 α_g

的值，来匹配高光大小。底部渲染图中的球体也使用了相同的数值，其中第一行使用了 Beckmann NDF，第二行则使用了 GGX 分布。

真实世界中的许多材质都具有类似的模糊高光，其尾部通常还要比 GGX 分布的尾部更长 [214]，详见图 9.37。这种基于真实世界的观察认知对 GGX 分布的日益流行，以及探索能够更加准确地模拟被测材质的新分布，都做出了很大的贡献。

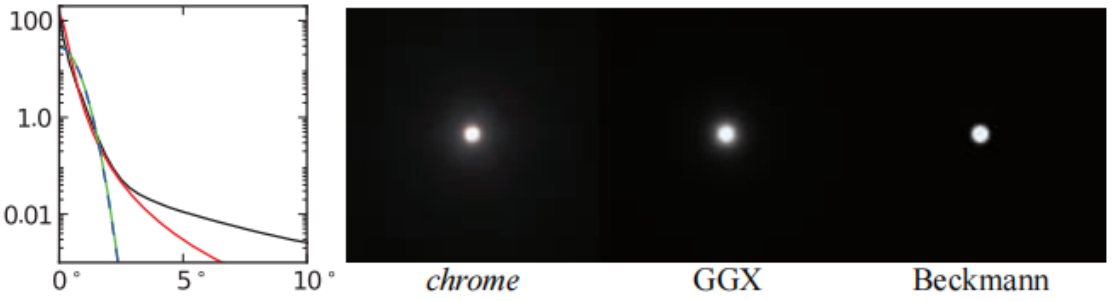


图 9.37：MERL 数据库中各种 NDF 对金属铬的匹配程度。左侧图中展示了相对于夹角 θ_m 的镜面峰值，三条曲线分别代表金属铬（黑色）、GGX 分布（红色，其中 $\alpha_g = 0.006$ ），Beckmann 分布（绿色，其中 $\alpha_g = 0.013$ ），Blinn–Phong 分布（蓝色虚线，其中 $n = 12000$ ）。右侧展示了金属铬、GGX 分布，Beckmann 分布的高光渲染结果。[214]

Burley [214]提出了广义的 Trowbridge–Reitz （GTR） NDF，其目标是对 NDF 的形状进行更多的控制，尤其是分布的拖尾，其数学表达如下：

$$D(\mathbf{m}) = \frac{k(\alpha, \gamma)}{\pi \left(1 + (\mathbf{n} \cdot \mathbf{m})^2 \left(\alpha_g^2 - 1\right)\right)^{\gamma}} \tag{9.46}$$

参数 γ 控制了尾部的形状：当 $\gamma = 2$ 时，GTR 分布与 GGX 分布完全相同；当 γ 值减小时，分布的尾部会变长；当 γ 值增大时，分布的尾部会变短。当 γ 值较大时，GTR 分布类似于 Beckmann 分布。方程中的 $k(\alpha, \gamma)$ 项是归一化系数，由于它要比其他 NDF 的归一化系数更加复杂，因此我们单独给出它的数学表达式：

$$k(\alpha, \gamma) = \begin{cases} \frac{(\gamma - 1)\left(\alpha^2 - 1\right)}{\left(1 - (\alpha^2)^{(1-\gamma)}\right)}, & \text{where } \gamma \neq 1 \text{ and } \alpha \neq 1 \\ \frac{\left(\alpha^2 - 1\right)}{\ln(\alpha^2)}, & \text{where } \gamma = 1 \text{ and } \alpha \neq 1 \\ 1, & \text{where } \alpha = 1. \end{cases} \tag{9.47}$$

GTR 分布并不是形状不变的，这使得寻找它的 Smith G_2 函数变得相当复杂。在这个 NDF 发表 3 年后， G_2 函数的解决方案才得以发表[355]；这个 G_2 函数的解相当

复杂，它只给出了某些 γ 值对应的解析解表（而其他中间值则必须使用插值获得）。GTR 的另一个问题是参数 α 和 γ，会以一种很不直观的方式影响人们所感知到的粗糙度和“微光”效果。

Student t 分布（Student’s t–distribution，STD）[1491]和指数幂分布（exponential power distribution，EPD）[763]，这两个 NDF 中包含了形状控制参数。与 GTR 相比，这些函数相对于它们的粗糙度参数是形状不变的。在撰写本文的时候，它们才刚刚发布，因此尚不清楚它们在应用程序中的实用性。

> 译者注：关于 Student t 分布，其中 Student 是英国统计学家 W. S. Gosset 在 1908 年发布论文时所用的笔名。

为了更好地对被测材质进行模拟，一种替代方案是使用多个镜面波瓣，而不是增加单个 NDF 的复杂性，这个想法是由 Cook 和 Torrance 提出的[285, 286]。Ngan [1271]对其进行了实验测试，他发现对于许多材质，添加第二个波瓣确实可以显著改善匹配度。皮克斯的 PxrSurface 材质[732]中包含一个“粗糙镜面（roughspecular）”波瓣，它便是用于这个目的（与主镜面波瓣结合在一起）。这个额外的波瓣是一个具有所有相关参数和构成项的镜面微表面 BRDF。Imageworks 采用了一种更加精确的方法[947]，它使用了两个 GGX 分布的混合，并将它们作为一个扩展的 NDF 暴露给用户，而不是作为一整个单独的镜面 BRDF 项。在这种情况下，只需要两个额外参数即可，分别是第二个粗糙度值和二者之间的混合量（用于控制插值）。

各项异性法线分布函数

虽然大多数材质的表面都是各向同性的，但有一些材料的微观结构具有显著的各向异性，这大大地影响了它们的材质外观，例如图 9.26 所示。为了准确地渲染这些材质，我们需要具有各向异性 NDF 的 BRDF。

与各向同性的 NDF 不同，各向异性的 NDF 不能仅仅使用单独的夹角 θ_m 进行描述，它还需要额外的方向信息。一般情况下，我们需要将微表面法线 $\mathbf{m}$ 变换到由法线 $\mathbf{n}$、切线（tangent）向量 $\mathbf{t}$、副切线（bitangent）向量 $\mathbf{b}$ 所定义的局部坐标系或者切线空间中，详见图 6.32。在实践中，这个变换通常被表示为三个独立的点积：$\mathbf{m} \cdot \mathbf{n}$，$\mathbf{m} \cdot \mathbf{t}$ 和 $\mathbf{m} \cdot \mathbf{b}$。

当我们将法线映射与各向异性 BRDF 相结合时，最重要的一点是，确保法线映射在变换法线的同时，也会对切线和副切线进行变换。通常来说，这个过程是通过对扰动法

线 $\mathbf{n}$，以及顶点插值而来的切线 $\mathbf{t}_0$ 和副切线 $\mathbf{b}_0$，应用改进的 Gram–Schmidt 过程来完成的。其数学表达如下（假设 $\mathbf{n}$ 已经被事先归一化了）：

$$\left.\begin{aligned}\mathbf{t}' &= \mathbf{t}_0 - (\mathbf{t}_0 \cdot \mathbf{n})\,\mathbf{n} \quad \Longrightarrow \mathbf{t} = \frac{\mathbf{t}'}{\|\mathbf{t}'\|},\\ \mathbf{b}' &= \mathbf{b}_0 - (\mathbf{b}_0 \cdot \mathbf{n})\,\mathbf{n},\\ \mathbf{b}'' &= \mathbf{b}' - (\mathbf{b}' \cdot \mathbf{t})\,\mathbf{t}\end{aligned}\right\} \Rightarrow \mathbf{b} = \frac{\mathbf{b}''}{\|\mathbf{b}''\|} \tag{9.48}$$

或者也在第一行之后，通过 $\mathbf{n}$ 和 $\mathbf{t}$ 的叉乘来得到正交向量 $\mathbf{n}$。

在实现拉丝金属和卷发等效果的时候，需要对切线方向进行逐像素的修改，一般是通过一张切线贴图（tangent map）来实现的。这个纹理贴图存储了逐像素的切线信息，类似于法线贴图中存储的逐像素法线信息。切线贴图通常存储的是切线向量在垂直于法线的平面上的二维投影。这种数据表示方法可以很好地与纹理过滤进行工作，并且也可以像法线贴图一样进行压缩。有些应用程序存储的是一个标量旋转角度，用于将切线向量围绕法线 $\mathbf{n}$ 进行旋转。虽然这种数据表示方式更加紧凑，但是当旋转角度突然从 360° 切换到 0° 的时候，很容易产生纹理滤波的瑕疵。

创建各向异性 NDF 的一种常见方法是，将现有的各向同性 NDF 进行推广，这种一般性的方法可以应用于任何形状不变的各向同性 NDF [708]，这也是形状不变的 NDF 更加实用的另一个原因。这里我们回顾一下，形状不变的各向同性 NDF 可以写成下面的形式：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})}{\alpha^2(\mathbf{n}\cdot\mathbf{m})^4} g\left(\frac{\sqrt{1-(\mathbf{n}\cdot\mathbf{m})^2}}{\alpha(\mathbf{n}\cdot\mathbf{m})}\right) \tag{9.49}$$

方程中的函数 g 是一个代表 NDF 形状的一维函数。方程 9.49 的其各向异性版本是：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})}{\alpha_x\alpha_y(\mathbf{n}\cdot\mathbf{m})^4} g\left(\frac{\sqrt{\frac{(\mathbf{t}\cdot\mathbf{m})^2}{\alpha_x^2} + \frac{(\mathbf{b}\cdot\mathbf{m})^2}{\alpha_y^2}}}{(\mathbf{n}\cdot\mathbf{m})}\right) \tag{9.50}$$

方程中的参数 α_x 和 α_y，分别代表了沿 $\mathbf{t}$ 方向和 $\mathbf{b}$ 方向的粗糙度。如果 $\alpha_x = \alpha_y$ 的话，方程 9.50 还可以退化为各向同性的形式。

各向异性 NDF 的 G_2 masking–shadowing 函数，与各向同性中的基本相同，除了变量 a（用于传递给 Λ 函数）的计算有所不同：

$$a = \frac{\mathbf{n} \cdot \mathbf{s}}{\sqrt{\alpha_x^2 (\mathbf{t} \cdot \mathbf{s})^2 + \alpha_y^2 (\mathbf{b} \cdot \mathbf{s})^2}} \tag{9.51}$$

其中方程 9.51 中的向量 $\mathbf{s}$，代表了相机的观察方向 $\mathbf{v}$ 或者光线的入射方向 $\mathbf{l}$。

利用这种方法，我们可以推导出各向异性版本的 Beckmann NDF：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n} \cdot \mathbf{m})}{\pi \alpha_x \alpha_y (\mathbf{n} \cdot \mathbf{m})^4} \exp\left(-\frac{\frac{(\mathbf{t} \cdot \mathbf{m})^2}{\alpha_x^2} + \frac{(\mathbf{b} \cdot \mathbf{m})^2}{\alpha_y^2}}{(\mathbf{n} \cdot \mathbf{m})^2} \right) \tag{9.52}$$

以及各向异性版本的 GGX NDF，二者的效果如图 9.38 所示：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n} \cdot \mathbf{m})}{\pi \alpha_x \alpha_y \left(\frac{(\mathbf{t} \cdot \mathbf{m})^2}{\alpha_x^2} + \frac{(\mathbf{b} \cdot \mathbf{m})^2}{\alpha_y^2} + (\mathbf{n} \cdot \mathbf{m})^2 \right)^2} \tag{9.53}$$

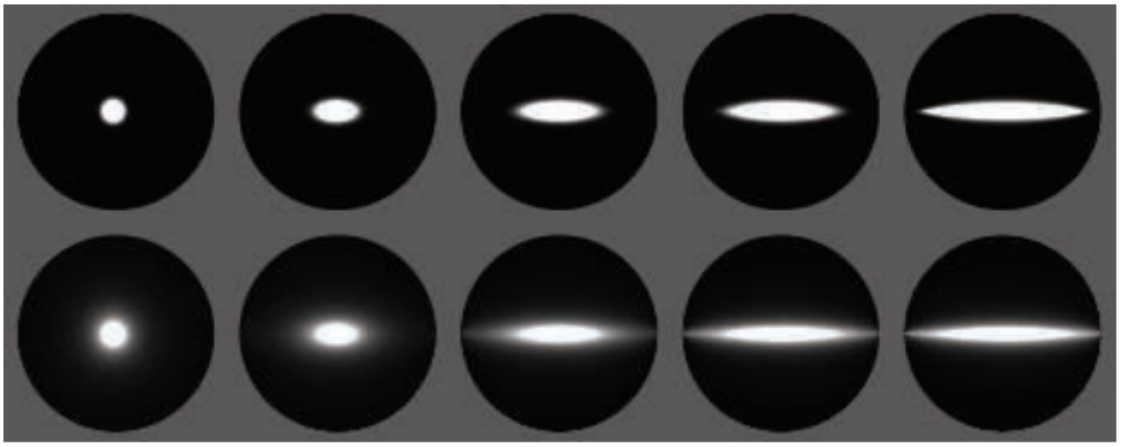


图 9.38：利用各向异性 NDF 渲染的球体，其中第一行使用的是 Beckmann NDF，第二行使用的是 GGX NDF。在这两行中，粗糙度参数 α_y 保持不变，α_x 从左到右递增。

虽然将各向异性 NDF 参数化的最直接方法，就是将各向同性的粗糙度参数化表示使用两次，一次用于 α_x，一次用于 α_y，但有时也会使用其他形式的参数化表示。在 Disney 原则着色模型中[214]，各向同性的粗糙度参数 r 会与另一个范围为 $[0, 1]$ 的标量参数 k_{aniso} 相结合，然后通过以下方式来计算出 α_x 和 α_y 的值：

$$
\begin{aligned}
k_{\text{aspect}} &= \sqrt{1 - 0.9 k_{\text{aniso}}}\,, \\
\alpha_x &= \frac{r^2}{k_{\text{aspect}}}, \\
\alpha_y &= r^2 k_{\text{aspect}}\,.
\end{aligned}
\tag{9.54}
$$

方程中的 0.9 将长宽比限制为 $10:1$ 。

Imageworks [947]使用了另一种不同形式的参数化方法，它允许构建任意程度的各向异性：

$$
\begin{aligned}
\alpha_x &= r^2 \left(1 + k_{\text{aniso}}\right), \\
\alpha_y &= r^2 \left(1 - k_{\text{aniso}}\right).
\end{aligned}
\tag{9.55}
$$

9.8.2 多次反弹的表面反射

正如章节 9.7 中所提到的，微表面 BRDF 理论并没有考虑到在微表面间进行多次反弹（bounce）的那部分光线。这种简化会导致一些能量损失和表面变暗，尤其是对于那些具有粗糙表面的金属而言[712]。

Imageworks [947]所使用的技术，将之前工作进行结合[811, 878]，创建了一个可以被添加到 BRDF 中的项，从而对多次反弹的表面反射进行模拟：

$$
f_{\text{ms}}(\mathbf{l}, \mathbf{v}) = \frac{\bar{F}\overline{R_{\text{sF1}}}}{\pi\left(1 - \overline{R_{\text{sF1}}}\right)\left(1 - \bar{F}\left(1 - \overline{R_{\text{sF1}}}\right)\right)}\left(1 - R_{\text{sF1}}(\mathbf{l})\right)\left(1 - R_{\text{sF1}}(\mathbf{v})\right)
\tag{9.56}
$$

其中 R_{sF1} 是指 f_{sF1} 上的定向反照率（directional albedo）（详见章节 9.3），也就是当 $F_0 = 1$ 时的镜面 BRDF 项。函数 R_{sF1} 依赖于粗糙度 α 和仰角 θ，这个函数是相对平滑的，因此可以对它进行预计算（使用方程 9.8 或者方程 9.9），并将结果存储在一个尺寸很小的二维纹理中，Imageworks 发现使用 32×32 的分辨率就足够了。

函数 $\overline{R_{\text{sF1}}}$ 是 R_{sF1} 在半球上的余弦加权平均值，这消除了仰角 θ 带来的变化，使得 $\overline{R_{\text{sF1}}}$ 只依赖于粗糙度 α，因此可以将其存储在一个一维纹理中，或者是使用一个便于计算的曲线来对其进行近似。由于 R_{sF1} 是关于表面法线 $\mathbf{n}$ 旋转对称的，因此可以用一个一维积分来计算 $\overline{R_{\text{sF1}}}$，这里我们还使用了换元法 $\mu = \cos\theta$ 来进行推导（详见方程 9.6）：

$$\begin{aligned}\overline{R_{\mathrm{sF1}}} &= \frac{\int_{\mathbf{s}\in\Omega} R_{\mathrm{sF1}}(\mathbf{s})(\mathbf{n}\cdot\mathbf{s})d\mathbf{s}}{\int_{\mathbf{s}\in\Omega}(\mathbf{n}\cdot\mathbf{s})d\mathbf{s}} = \frac{1}{\pi}\int_{\phi=0}^{2\pi}\int_{\mu=0}^{1} R_{\mathrm{sF1}}(\mu)\mu d\mu d\phi \\ &= 2\int_{\mu=0}^{1} R_{\mathrm{sF1}}(\mu)\mu d\mu.\end{aligned} \tag{9.57}$$

最后，方程 9.56 中的 $\overline{F}$ 代表了菲涅尔项的余弦加权平均值，可以用同样的方法来计算它：

$$\bar{F} = 2\int_{\mu=0}^{1} F(\mu)\mu d\mu \tag{9.58}$$

Imageworks 提供了一个方程 9.58 的封闭解，即 F 的广义 Schlick 形式（方程 9.18）：

$$\bar{F} = \frac{2p^2 F_{90} + (3p+1)F_0}{2p^2 + 3p + 1} \tag{9.59}$$

如果使用原始形式的 Schlick 近似（方程 9.16），那么这个解可以简化为以下形式：

$$\bar{F} = \frac{20}{21}F_0 + \frac{1}{21} \tag{9.60}$$

在各向异性的情况下，Imageworks 使用了介于 α_x 和 α_y 之间的中间粗糙度来计算 f_{ms}。这种近似策略可以避免增加 R_{sF1} 查找表的维度，同时引入的误差很小。

Imageworks 多次反弹的高光项结果如图 9.39 所示。

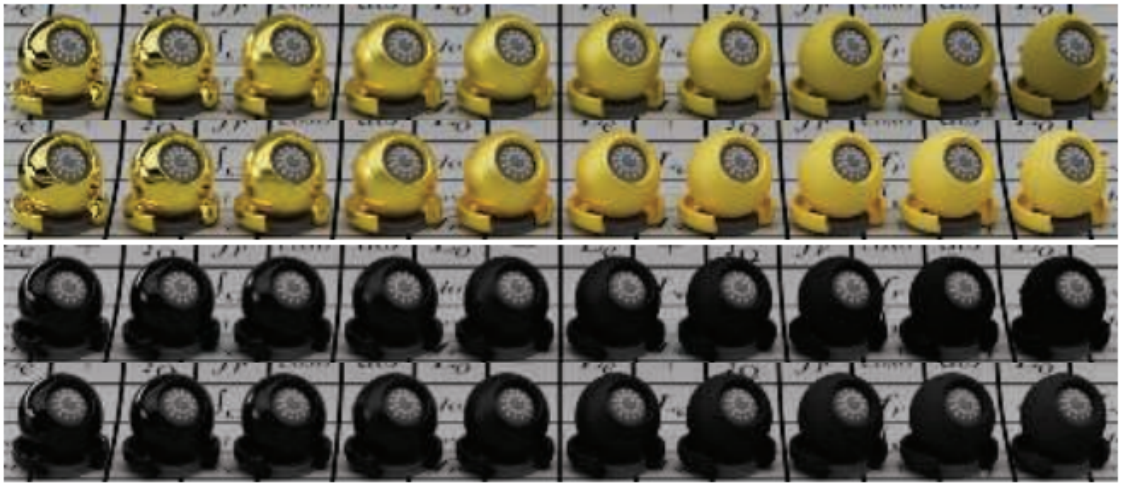


图 9.39：每一行的表面粗糙度从左到右递增。上面两行展示的是一种金色的金属材质，其中第一行在渲染的时候并没有使用 Imageworks 多次反弹项，而第二行在渲染的时候则使用了多次反弹项。二者之间的差异，对于表面粗糙度较高的球体而言最为明显。接下来的两行是黑色的电介质材质，其中第三行在渲染的时候没有使用多次反弹项，而第四行则使用了多次反弹项。第三行和第四行的差异更加细微，因为它们的镜面反射率要低得多。[947]

9.9 次表面散射的 BRDF 模型

在上一小节中，我们讨论了表面反射（surface reflection），或者叫做镜面反射（specular reflection）；而在这一小节中，我们将讨论反射问题的另一方面，即在表面以下的折射光线会发生什么。正如我们在章节 9.1.4 中讨论的那样，这些折射光线经历了某种散射作用（scatter）和吸收作用（absorption）的组合，其中一部分光线最终会被重新发射回原始表面。在本小节中，我们将重点讨论不透明电介质中的局部次表面散射，以及它们的 BRDF 模型。这里我们并不会讨论有关金属材质的话题，因为金属材质并没有任何明显的次表面光线作用（因为金属的对光线的吸收作用很强）。有关透明电介质材质，以及表现出全局次表面散射的电介质材质的话题，将在第 14 章进行讨论。

我们将首先从漫反射模型开始讨论，介绍漫反射颜色的相关特性，以及真实世界中漫反射材质可能具有的颜色值。而在接下来的几个小节中，我们会解释表面粗糙度是如何对漫反射着色产生影响的；以及对于给定的材质表面，到底是选择使用光滑表面还是粗糙表面的着色模型，以及这个选择的标准。在最后两个小节中，我们会主要关注光滑表面模型和粗糙表面模型本身。

9.9.1 次表面反照率

不透明电介质的次表面反照率（subsurface albedo） ρ_{ss} 指的是从表面逃逸的光线能量，与进入物质内部的光线能量之比。 ρ_{ss} 的值在 0（光线被全部吸收）到 1（没有任何光线被吸收）之间，并且与波长有关，因此 ρ_{ss} 在渲染中被表示为一个 RGB 向量。对于艺术家创作而言， ρ_{ss} 通常会被称为表面的漫反射颜色（diffuse color），就像是法线入射的菲涅尔反射率 F_0，通常会被称为镜面颜色（specular color）一样。次表面反照率与章节 14.1 中所讨论的散射反照率密切相关。

对于电介质而言，由于大部分光线都会透射进物质内部，而不是在表面上直接被反射出去，因此这些材质的次表面反照率 ρ_{ss} 通常会更加明亮，在视觉上要比高光颜色

F_0 更加重要。与表面的菲涅尔反射不同，光线在物质内部的吸收是一个独特的物理过程，因此 ρ_{ss} 通常具有与 F_0 不同的光谱分布（即不同的 RGB 颜色）。例如：彩色塑料是由透明的基板和嵌入在其内部的颜料颗粒组成的，光线照射到彩色塑料表面上，所产生的镜面反射光线是无色的，而漫反射光线有一部分（特定波长）会被颜料颗粒吸收，因此剩余被反射出来的光线会具有某种颜色。比如我们能够看到在一个红色的塑料球上看到一个白色的高光。

次表面反照率可以被认为是光线吸收与光线散射之间“相互竞争”所产生的结果，即光线到底是先被散射出物体，还是先被吸收？这也就是为什么液体上的泡沫，要比液体本身亮得多的原因：起泡过程并不会改变液体对光线的吸收率，但是所产生的大量气液界面会大大增加光线的散射量，这使得大部分入射光线在被吸收之前，会先被散射出去，最终导致了较高的次表面反照率和明亮的材质外观。新鲜的积雪是高反照率物质的另一个例子：新鲜的积雪十分蓬松，内部充满了空气，雪花颗粒与空气之间界面的散射量很大，吸收量很少，从而导致次表面反照率在可见光谱上可以达到 0.8 或者更大。白色油漆的次表面反照率要稍少小一些，约为 0.7。日常生活中遇到的许多物质，例如混凝土、石头、土壤等，反照率大约在 0.15 ~ 0.4 之间。而煤炭是一种次表面反照率极低的物质，接近 0.0。

和液体泡沫的例子相反，许多物质在潮湿之后反而会变得更暗。如果一个材料表面是多孔的，那么水就会渗入到原本充满空气的空间。相比于空气的折射率，电介质材料的折射率要更加接近于水。表面两侧相对折射率的下降，会降低物体内部的散射率，这使得光线在逃逸出物质之前，需要传播的平均距离更长。这种变化会导致更多的光线被吸收，从而使得次表面反照率变得更暗[821]。

有一个常见的误解，即真实感材质的 ρ_{ss} 永远不应该低于 0.015–0.03 的下限（在 8 bit 非线性 sRGB 编码中为 30–50），这个误解甚至出现在了一些广受好评的材质创作指南中[1163]。然而，这个下限所基于的颜色测量，包含了表面反射（specular）和次表面反射（diffuse）两部分，因此是过高的。实际材质可以有更低的 ρ_{ss} 值。例如：联邦规范中的“OSHA Black”涂料标准 [524]，其 Y 值为 0.35（最大为 100）。考虑到测量条件和表面光泽效应，这个 Y 所对应的 ρ_{ss} 值约为 0.0035（在 8 bit 非线性 sRGB 编码中为 11）。

当从真实世界的表面材质获取点值或者纹理时，从中分离镜面反射是十分重要的，这种提取操作可以通过精确控制光照和偏振滤光器[251, 952]来完成。为获得更加准确的颜色数据，还应当进行校准[1153]。

并不是每个 RGB 三元组都代表一个合理（或者是物理上可能存在的）的 ρ_{ss} 值。反射光谱的功率分布所受到的限制，要比发射光谱严格得多：对于任何波长的光线而

言，它们的值永远都不能超过 1，并且这些值的分布通常是相当平滑的。这些限制在颜色空间中定义了一个区域，其中包含了所有可能的 ρ_{ss} 所对应的 RGB 值。即使是相对较小的 sRGB 色域，也包含了其中所有的颜色，因此在设置 ρ_{ss} 的值时需要格外小心，避免指定那些不自然的、饱和且明亮的颜色。除了会降低画面的真实感之外，在预计算全局光照的时候，这些不合理的颜色可能会导致过于明亮的二次反射（详见章节 11.5.1）。Meng 等人[1199]于 2015 年的论文，是一个对于本主题很好的参考资料。

9.9.2 次表面散射和粗糙的尺度

一些用于局部次表面散射的 BRDF 模型，考虑了表面粗糙度所带来的影响（通常使用了带有漫反射 micro-BRDF f_μ 的微表面理论），而有些则没有这些影响。具体使用哪种类型的模型，并不是简单取决于表面有多粗糙（虽然这是一个常见的误解），正确的决定因素应当是表面不规则性与次表面散射距离的相对大小。

如图 9.40 所示，如果微观几何的不规则性大于次表面散射的距离（图 9.40 左上），那么次表面散射将会表现出一些与微观几何有关的效果（microgeometry-related effects），例如逆反射（详见图 9.29）。对于这种类型的表面，应当使用一个粗糙表面的漫反射模型。如上所述，这类模型一般都基于微表面理论，它将次表面散射现象视为每个微表面的局部散射（即将此表面散射视为微表面的局部现象），即只会对 micro-BRDF f_μ 产生影响。

如果次表面散射的距离都大于表面的不规则性（图 9.40 右上），那么在模拟次表面散射现象的时候，应该认为表面是平坦的，不会出现逆反射等效应。此时次表面散射对于微表面而言，并不是一个局部现象，因此无法使用微表面理论来进行描述。在这种情况下，应当使用一个光滑表面的漫反射模型。

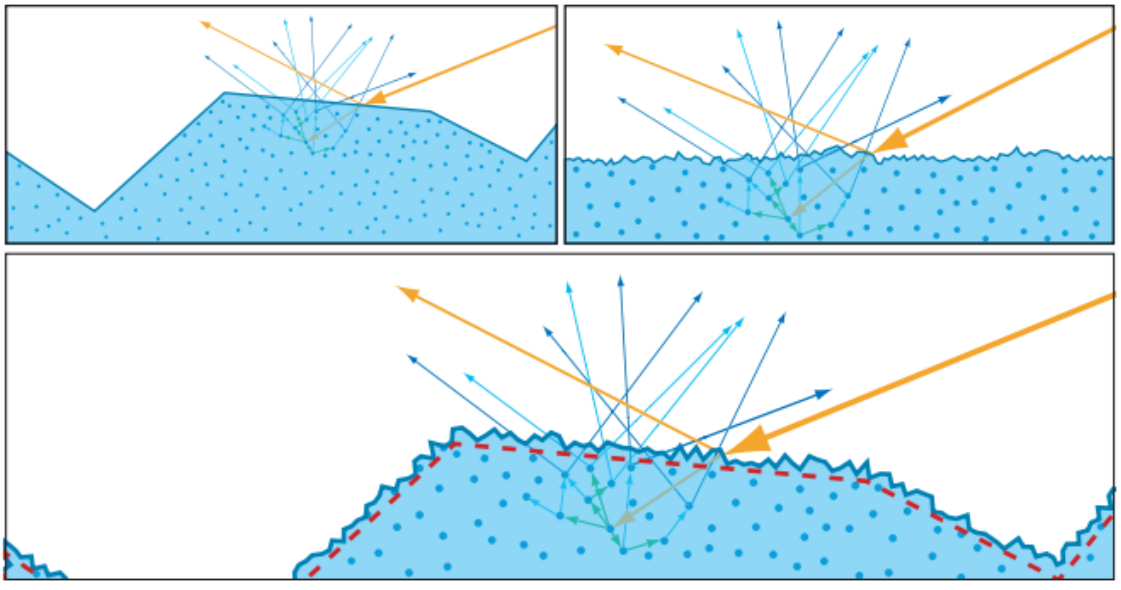


图 9.40：具有相似 NDF，但是微观几何尺度与次表面散射距离相对大小不同的三个表面。左上角：次表面散射距离小于表面不规则性；右上角：次表面散射距离大于表面不规则性。下方：展示了一个具有多种不同尺度粗糙度的微表面，图中红色虚线代表了其有效表面，它仅包含大于次表面散射距离的微观结构。

而在其他中间情况下，即表面在比散射距离更大或者更小的尺度上都具有一定的粗糙度，那么这时应当使用一个粗糙表面的漫反射模型，但是其有效表面（effective surface）只包含那些比散射距离更大的不规则性。在这种情况下，漫反射和镜面反射都可以使用微表面理论来进行建模，但是二者的粗糙度值是不同的：其中高光项将会使用一个基于实际表面粗糙度的值；而漫反射项将会使用一个较低的值，它基于的是有效表面的粗糙度。

观察尺度也与此相关，因为它决定了“微观几何”的具体含义。例如：一个应当使用粗糙表面漫反射模型的经典例子是月球，因为它表现出了明显的逆反射效应。当我们从地球上观察月球的时候，观察尺度非常大，以至于月球表面上一个五英尺高的巨石都可以算是“微观几何”。因此，我们能够观察到粗糙表面的漫反射效应（例如后反射）也就并不奇怪了。

9.9.3 光滑表面的次表面模型

本小节我们将讨论光滑表面的次表面模型（smooth-surface subsurface model）。这些方法适用于对表面不规则性小于次表面散射距离的材质进行建模，这些材质的表面粗糙度并不会直接影响它们的漫反射着色，但是如果漫反射项和高光项是耦合的（本小节中一些模型是这样的），那么表面粗糙度可能会对漫反射着色产生间接影响。

如章节 9.3 中所提到的，实时渲染程序通常会使用 Lambertian 项来对局部次表面散射现象进行建模，在这种情况下，BRDF 的漫反射项是次表面反照率 ρ_{ss} 除以 π：

$$f_{\mathrm{diff}}(\mathbf{l},\mathbf{v}) = \frac{\rho_{\mathrm{ss}}}{\pi} \tag{9.61}$$

Lambertian 模型并没有考虑到这样一个事实，即表面反射的光线并不能用于次表面散射。为了对这个模型进行改进，应当在表面反射项（镜面反射项）和次表面反射项（漫反射项）之间进行能量权衡。菲涅尔效应表明，这种表面-次表面之间的能量权衡，应当随着入射光角 θ_i 的变化而变化。随着光线入射角度（尤其是靠近掠射角度时）的增大，漫反射率会减小，镜面反射率会增大。实现这种平衡的最基本方法是，

将漫反射项乘以 1 减去镜面项的菲涅尔部分（F 项）[1626]。如果某个高光项是由平坦镜面所产生的话，那么对应产生的漫反射项为：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = (1 - F(\mathbf{n},\mathbf{l}))\frac{\rho_{\text{ss}}}{\pi} \tag{9.62}$$

如果镜面反射项是一个微表面 BRDF 项的话，那么对应产生的漫反射项为：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = (1 - F(\mathbf{h},\mathbf{l}))\frac{\rho_{\text{ss}}}{\pi} \tag{9.63}$$

方程 9.62 最终会生成均匀分布的出射光线，因为这个 BRDF 的结果并不依赖于出射方向 $\mathbf{v}$ 。这种光线行为是有一定道理的，因为光线在重新发射出表面之前，通常都会经历多次散射事件，因此其最终的出射方向将会是随机的。然而，存在有两个理由使得我们怀疑其实出射光线的分布并不是完全均匀的。首先，由于方程 9.62 中的漫反射 BRDF 项会随光线入射方向的变化而变化，Helmholtz 互易性意味着，它也会随着出射方向的变化而变化。第二，光线在最终射出表面的时候会发生折射，根据光线折射的性质，这些最终射出的光线会具有一定的方向性，即会偏向于某些方向。

Shirley 等人提出了一种用于平坦表面的耦合漫反射项，它解决了菲涅尔效应与表面–次表面反射之间的权衡问题，同时它还满足能量守恒和 Helmholtz 互易性[1627]。推导中假设菲涅尔反射使用了 Schlick 近似方法[1568]（方程 9.16），其数学形式如下：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = \frac{21}{20\pi}(1 - F_0)\,\rho_{\text{ss}}\left(1 - \left(1 - (\mathbf{n}\cdot\mathbf{l})^{+}\right)^{5}\right)\left(1 - \left(1 - (\mathbf{n}\cdot\mathbf{v})^{+}\right)^{5}\right) \tag{9.64}$$

方程 9.64 只适用于镜面反射为完美菲涅尔镜面的表面。Ashikhmin 和 Shirley [77]提出了一个广义的版本，它可以用来计算一个可逆的（reciprocal）、低计算开销的漫反射项，并且它可以与任何高光项结合使用。Kelemen 和 Szirmay–Kalos [878]对其进行了进一步完善：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = \rho_{\text{ss}}\frac{(1 - R_{\text{spec}}(\mathbf{l}))\,(1 - R_{\text{spec}}(\mathbf{v}))}{\pi\left(1 - \overline{R_{\text{spec}}}\right)} \tag{9.65}$$

方程中的 R_{spec} 是高光项的定向反照率（directional albedo，详见章节 9.3），而 $\overline{R_{\text{spec}}}$ 则是它在半球上的余弦加权平均值。值 R_{spec} 可以使用方程 9.8 或者方程 9.9

预计算出来，并将结果存储在一个查找表中。平均值 $\overline{R_{\text{spec}}}$ 的计算方法与我们之前所遇到的 $\overline{R_{\text{sF1}}}$ 相同（方程 9.57）。

方程 9.65 在形式上与方程 9.56 有一些明显的相似之处，这并不奇怪，因为 Imageworks 的多次弹射高光项，就是从 Kelemen–Szirmay–Kalos 耦合漫反射项中推导出来的。然而，二者之间有一个重要的区别：在这里我们使用的是 R_{spec} 而不是 R_{sF1}，即使用了包含菲涅尔项、多次弹射高光项 f_{ms}（如果使用了的话）在内的、完整的镜面反射 BRDF 项的定向反照率。这种差异增加了 R_{spec} 查找表的维度，因为它不仅取决于粗糙度 α 和仰角 θ，还取决于菲涅尔反射率。

在 Imageworks 对 Kelemen–Szirmay–Kalos 耦合漫反射项的实现中，他们使用了一个三维查找表，其中的第三个维度是折射率[947]。他们发现，在积分中使用多次弹射项，可以使 R_{spec} 变得比 R_{sF1} 更加平滑，因此 $16 \times 16 \times 16$ 查找表很足够了，图 9.41 展示了相应的结果。

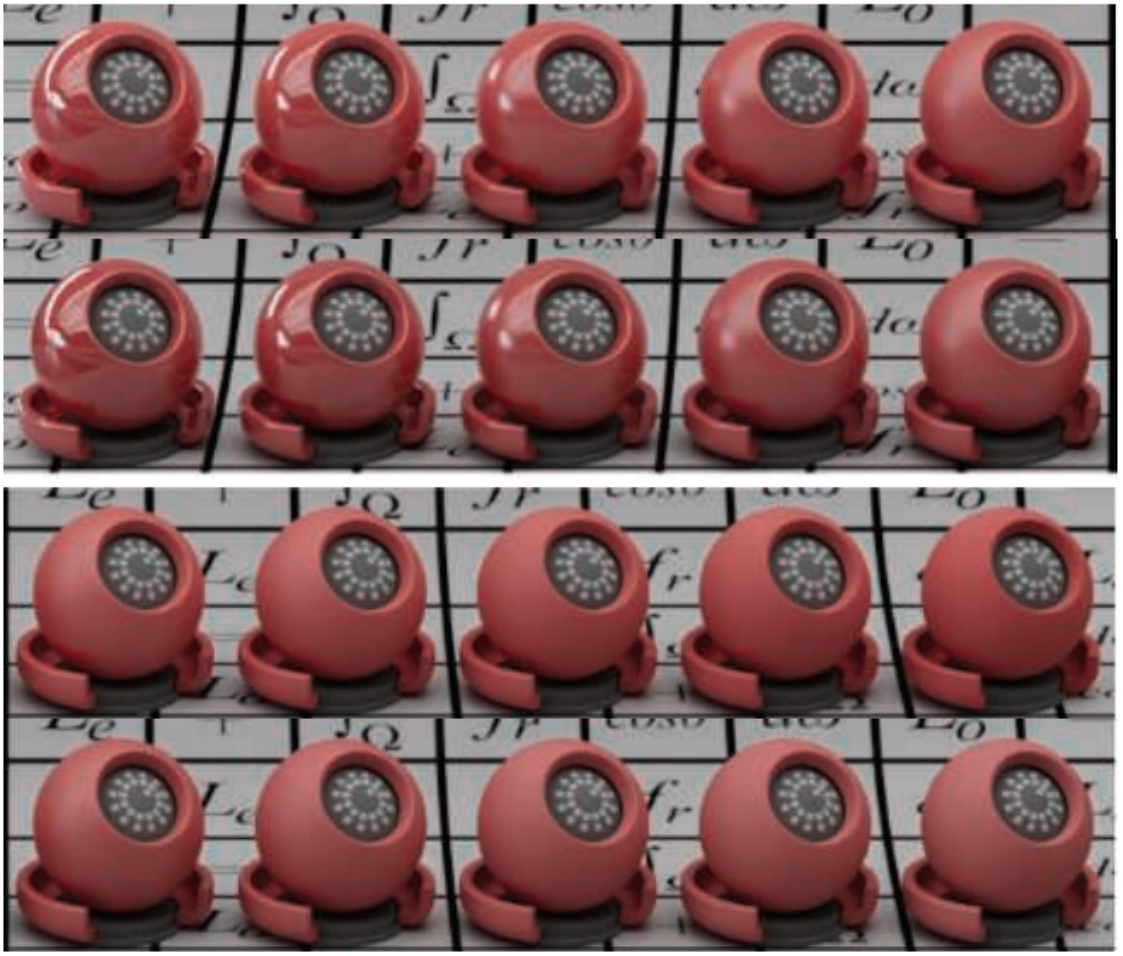


图 9.41：第一行和第三行图片展示了在一个 Lambertian 项中添加高光项的渲染结果。第二行和第四行图片使用了 Kelemen–Szirmay–Kalos 耦合漫反射项，并添加了一个相同的高光项。上面两行的粗糙度值要比下面两行低。在每一行中，粗糙度从左到右递增。[947]

如果 BRDF 使用 Schlick Fresnel 近似方法，但是不包含多次反弹高光项的话，那么 F_0 的值可以从积分中提取出来。这样我们就可以只使用一个二维的 R_{spec} 查找表，而不是 Karis [861]所讨论的三维表。此外，Lazarov [999]提出了一个用于拟合 R_{spec} 的解析函数，我们同样将系数 F_0 从积分中提取出来，从而对拟合函数进行简化。

Karis 和 Lazarov 都将镜面定向反照率 R_{spec} 用在了其他的地方，这与基于图像的照明（image-based lighting，IBL）有关。有关 IBL 的技术细节，我们将在章节 10.5.2 中进行讨论。如果一个应用程序同时实现了这两种技术，那么则可以使用同一个查找表，从而提高效率。

上述这些模型，是通过考虑表面反射（高光项）和次表面反射（漫反射项）之间的能量守恒而建立的；而其他的模型则是根据物理原理发展而来的，其中许多模型都依赖于 Subrahmanyan Chandrasekhar（1910–1995）的工作，他提出了一个半无限的、各向同性的散射体 BRDF 模型。Kulla 和 Conty [861]证明了，如果平均自由程（mean free path，译者注：一个物理概念，指某个分子连续两次与其它分子发生相撞的平均距离）足够短，那么这个 BRDF 模型可以完美匹配任意形状的散射体积（scattering volume）。有关 Chandrasekhar BRDF 的内容，可以在他的书中找到 [253]，不过在 Dupuy 等人[397]的一篇论文中，其中的方程 30 和方程 31 也对其进行了描述，它使用了我们熟悉的渲染符号，更加容易理解。

由于 Chandrasekhar BRDF 并不包含折射项，因此它只能用于建模折射率匹配的表面，这里的折射率匹配，指的是表面两侧的折射率完全相同，如图 9.11 所示。为了对折射率不匹配的表面进行建模，必须对 BRDF 进行一定的修改，将光线进入和离开表面的折射现象考虑在内。这种修改正是 Hanrahan 和 Krueger [662]以及 Wolff [1898]的核心工作。

9.9.4 粗糙表面的次表面模型

作为迪士尼原则着色模型的一部分，Burley [214]引入了一个漫反射 BRDF 项，用于匹配被测材质的粗糙度效果：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = \chi^+(\mathbf{n}\cdot\mathbf{l})\chi^+(\mathbf{n}\cdot\mathbf{v})\frac{\rho_{\text{ss}}}{\pi}\left((1-k_{\text{ss}})\,f_{\text{d}} + 1.25k_{\text{ss}}f_{\text{ss}}\right) \quad (9.66)$$

其中：

$$
\begin{aligned}
f_{\mathrm{d}} &= \left(1+\left(F_{\mathrm{D90}}-1\right)\left(1-\mathbf{n}\cdot\mathbf{l}\right)^5\right)\left(1+\left(F_{\mathrm{D90}}-1\right)\left(1-\mathbf{n}\cdot\mathbf{v}\right)^5\right), \\
F_{\mathrm{D90}} &= 0.5+2\sqrt{\alpha}(\mathbf{h}\cdot\mathbf{l})^2, \\
f_{\mathrm{Ss}} &= \left(\frac{1}{(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v})}-0.5\right)F_{\mathrm{SS}}+0.5, \\
F_{\mathrm{SS}} &= \left(1+\left(F_{\mathrm{SS90}}-1\right)\left(1-\mathbf{n}\cdot\mathbf{l}\right)^5\right)\left(1+\left(F_{\mathrm{SS90}}-1\right)\left(1-\mathbf{n}\cdot\mathbf{v}\right)^5\right), \\
F_{\mathrm{SS90}} &= \sqrt{\alpha}(\mathbf{h}\cdot\mathbf{l})^2,
\end{aligned}
\tag{9.67}
$$

方程 9.67 中的 α 是镜面粗糙度，在各向异性的情况下，则使用 α_x 和 α_y 之间的中间值。这个方程通常被称为迪斯尼漫反射模型（Disney diffuse model）。

次表面项 f_{ss} 的灵感来自于 Hanrahan–Krueger BRDF [662]，用于作为对远处物体的全局次表面散射现象的廉价近似。这个漫反射模型会基于用户控制的参数 k_{ss}，将 f_{ss} 与粗糙漫反射项 f_d 混合在一起。

迪士尼漫反射模型被广泛应用于电影[214]和游戏[214]（尽管没有次表面项）中。完整的迪斯尼漫反射 BRDF 还包含一个光泽（sheen）项，这主要是为了对纤维织物进行建模，同时也有助于补偿因缺乏多次弹射高光项而造成的能量损失。有关迪士尼光泽项的内容将在章节 9.10 中进行讨论。几年后，Burley 提出了一个更新后的模型[215]，旨在对全局次表面散射的渲染技术进行整合。

由于迪斯尼漫反射模型使用了与镜面 BRDF 相同的粗糙度参数，因此在对某些材质进行建模的时候，可能会遇到困难，详见图 9.40。然而，可以通过一个很小的修改，从而使用一个单独的漫反射粗糙度值。

其他大多数粗糙表面的漫反射 BRDF 都是基于微表面理论开发的，它们具有不同的 NDF、micro–BRDF f_μ 以及 masking–shadowing 函数 G_2。其中最著名的模型是由 Oren 和 Nayar 提出的[1337]。Oren–Nayar BRDF 使用了 Lambertian 的 micro–BRDF、球面高斯 NDF 以及 Torrance–Sparrow“V–cavity”的 masking–shadowing 函数。完整形式的 BRDF 模型了对二次反弹也进行了建模。Oren 和 Nayar 还在他们的论文中引入了一个简化的“定性”模型。多年来，人们提出了一些对 Oaren–Nayar 模型的改进方法[573]，包括对“定性”模型进行优化调整，使其在不增加额外成本的前提下，尽可能地接近完整模型[504]；以及将 micro–BRDF 更改为一个更加精确的光滑表面漫反射模型[574, 1899]。

Oren–Nayar 模型假设了一个全新的微表面，与当前镜面模型所使用的微表面相比，它具有完全不同的法线分布和 masking–shadowing 函数。利用各向同性的 GGX

NDF 和高度相关的 Smith masking–shadowing 函数，可以推导出两种不同的漫反射微表面模型。第一个模型是由 Gotanda [574]提出的，他使用了方程 9.64 所描述的镜面耦合漫反射项来作为 micro–BRDF，再对通用的微表面方程（方程 9.26）进行数值积分，最后使用一个解析函数来对数值积分的数据进行拟合。Gotanda 提出的 BRDF 并没有考虑微表面之间的相互反射，并且拟合函数也相对复杂。

使用与 Gotanda BRDF 相同的 NDF、masking–shadowing 函数和 micro–BRDF，Hammon [657]对 BRDF 进行了数值模拟，同时将相互反射考虑在内。他指出，相互反射对于这种微表面结构而言十分重要，对于粗糙表面来说，相互反射大约占据了总反射中的一半。Hammon 发现第二次反弹几乎包含了所有丢失的能量（即考虑更多的反弹次数，对最终结果的贡献并不大），因此他使用了两次反弹模拟的数据。 此外，可能是因为添加了相互反射，使得数据变得更加平滑，因此 Hammon 能够使用一个相当简单的函数，来对模拟结果进行拟合，最终的数学表达如下：

$$f_{\mathrm{diff}}(\mathbf{l},\mathbf{v}) = \chi^+(\mathbf{n}\cdot\mathbf{l})\chi^+(\mathbf{n}\cdot\mathbf{v})\frac{\rho_{\mathrm{ss}}}{\pi}\left((1-\alpha_g)\,f_{\mathrm{smooth}} + \alpha_g f_{\mathrm{rough}} + \rho_{\mathrm{ss}} f_{\mathrm{multi}}\right) \tag{9.68}$$

其中：

$$\begin{aligned} f_{\mathrm{smooth}} &= \frac{21}{20}\left(1-F_0\right)\left(1-(1-\mathbf{n}\cdot\mathbf{l})^5\right)\left(1-(1-\mathbf{n}\cdot\mathbf{v})^5\right), \\ f_{\mathrm{rough}} &= k_{\mathrm{facing}}\left(0.9-0.4k_{\mathrm{facing}}\right)\left(\frac{0.5+\mathbf{n}\cdot\mathbf{h}}{\mathbf{n}\cdot\mathbf{h}}\right), \\ k_{\mathrm{facing}} &= 0.5+0.5(\mathbf{l}\cdot\mathbf{v}), \\ f_{\mathrm{multi}} &= 0.3641\alpha_g, \end{aligned} \tag{9.69}$$

方程中的 α_g 是 GGX 镜面粗糙度。为了清楚起见，这里所使用术语与 Hammon 的陈述略有不同。请注意，方程 9.69 中的 f_{smooth} ，其实就是方程 9.64 中不含 ρ_{ss}/π 因子的耦合漫反射 BRDF，因为这个因子被包含在了方程 9.68 中。Hammon 讨论了一些“混合”BRDF，即使用其他光滑表面的漫反射 BRDF 来替代 f_{smooth} ，从而提高性能表现，或者改进旧着色模型下的资产兼容性。

总的来说，尽管 Hammon 并没有给出与测量数据的比较，但是他提出漫反射 BRDF 的计算开销很低，并且建立在可靠的理论原理之上。需要注意的是，表面不规则性大于散射距离的这个假设，是 BRDF 推导的基础，这可能会对它能够准确建模的材质类型有所限制，详见图 9.40。

方程 9.61 中展示的简单 Lambertian 项仍然被许多实时渲染程序所采用，除了较低的计算成本之外，与其他的漫反射模型相比，它可以更加容易地和间接烘焙光照一起使用；与更加复杂的着色模型相比，二者之间的视觉差异通常是不易察觉的[251, 861]。然而，对于真实感的持续追求，正在推动精确模型的广泛使用。

9.10 布料的 BRDF 模型

布料往往具有不同于其他类型材质的微观几何结构，根据织物类型的不同，它可能还会具有高度重复的编织微观结构、从表面垂直突出的圆柱体（线），又或是二者都有。由于布料表面独特的特征外观，因此通常需要使用专门的着色模型来进行渲染，这些独特外观包括：各向异性的镜面高光，粗糙散射[919]（光线在突出的、半透明的纤维中发生散射，从而导致边缘明亮的效果）， 甚至颜色也会随着观察方向的变化而变化（这是由织物中不同颜色的线所引起的）。

除了 BRDF 之外，大多数织物还具有高频空间变化，这也是生成令人信服的布料材质的关键因素[825]，如图 9.42 所示。

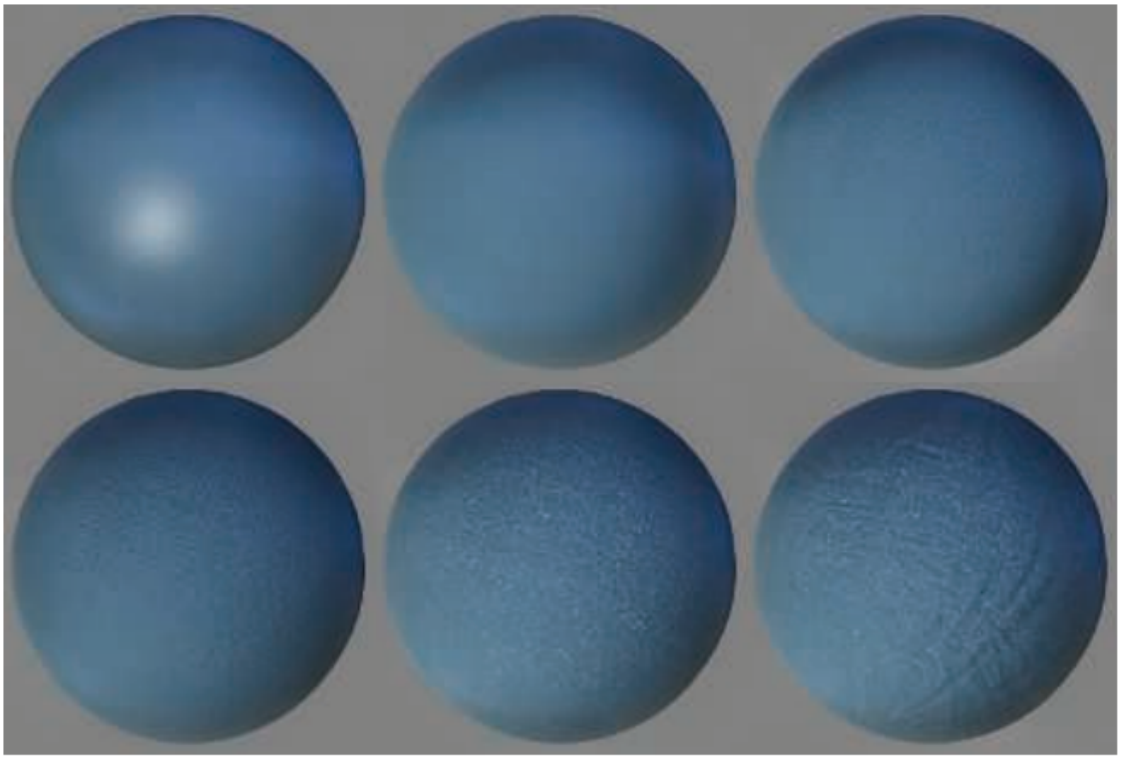


图 9.42：一种使用《神秘海域 4》布料系统的材质。左上方的球体使用了标准的 BRDF，它具有 GGX 微表面镜面反射和 Lambertian 漫反射。上部中间的球体使用了纤维织物的 BRDF。其他每个球体上都添加了不同类型的逐像素变化，从左上到右下分别是：织物编织细节，织物老化程度，细节瑕疵和小皱纹。

布料的 BRDF 模型主要分为三大类：根据观察建立的经验模型、基于微表面理论的模型、微圆柱体模型。我们将从每个类别中分别举出一些值得注意的例子。

9.10.1 经验布料模型

在游戏《神秘海域 2》[631]中，布料表面使用了以下的漫反射 BRDF 项：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi}\left(k_{\text{rim}}\left((\mathbf{v}\cdot\mathbf{n})^{+}\right)^{\alpha_{\text{rim}}} + k_{\text{inner}}\left(1-(\mathbf{v}\cdot\mathbf{n})^{+}\right)^{\alpha_{\text{inner}}} + k_{\text{diff}}\right) \tag{9.70}$$

其中 k_{rim} 是边缘光项（rim）的比例系数， k_{inner} 是正面（内部，inner）亮度项的比例系数， k_{diff} 是 Lambertian 项的比例系数。此外， α_{rim} 和 α_{inner} 分别控制了边缘项和内部项的衰减。这种行为并不是物理正确的，因为这些效果只与观察方向有关，与入射光线的方向无关。

相比之下，《神秘海洋 4》[825]中的布料材质使用了微表面模型或者微圆柱体模型，具体使用哪一种模型，取决于布料的高光项（会在之后的两个小节中进行详细描述），并且使用了一种叫做“环绕光照（wrap lighting）”的经验次表面散射方法，来对漫反射项进行近似：

$$f_{\text{diff}}(\mathbf{l},\mathbf{v})(\mathbf{n}\cdot\mathbf{l})^{+} \Rightarrow \frac{\rho_{\text{ss}}}{\pi}\left(\mathbf{c}_{\text{scatter}} + (\mathbf{n}\cdot\mathbf{l})^{+}\right)^{\mp}\frac{(\mathbf{n}\cdot\mathbf{l}+w)^{\mp}}{1+w} \tag{9.71}$$

这里我们使用了章节 1.2 中介绍过的符号 $(x)^{\mp}$ ，它表示 0–1 之间的限制操作。左侧这个奇怪的符号 $f_{\text{diff}}(\mathbf{l},\mathbf{v})(\mathbf{n}\cdot\mathbf{l})^{+} \Rightarrow \ldots$ 代表了这个模型会影响光照和 BRDF，并使用箭头右边的项来取代左边的项。用户指定的参数 $c_{scatter}$ 是一种散射颜色；值 w 的范围为 $[0,1]$ ，用于控制环绕光照的宽度。

对于布料材质的建模，迪士尼将他们的漫反射 BRDF 项 [214]（详见章节 9.9.4）和一个光泽项相结合，从而对粗糙散射（asperity scattering）进行模拟：

$$f_{\text{sheen}}(\mathbf{l},\mathbf{v}) = k_{\text{sheen}}\,\mathbf{c}_{\text{sheen}}\left(1-(\mathbf{h}\cdot\mathbf{l})^{+}\right)^{5} \tag{9.72}$$

其中 k_{sheen} 是用于调节光泽项强度的用户参数。光泽颜色 c_{hsheen} 会在白色和亮度归一化的 ρ_{ss} 之间进行插值（由另一个用户参数进行控制）。这里的亮度归一化是指用 ρ_{ss} 除以其亮度值，从而分离它的色相（hue）和饱和度（saturation）。

9.10.2 微表面布料模型

Ashikhmin 等人[78]提出使用一个逆高斯 NDF 来模拟天鹅绒（velvet）材质。这个 NDF 在后续工作中[81]又进行了一些微调，同时还提出了一种用于模拟一般材质的微表面 BRDF 变体，这个变体没有 masking–shadowing 项，也没有对分母进行修改。

在游戏《教团：1886》[1266]所使用的布料 BRDF 中，结合了改进的微表面 BRDF、Ashikhmin 和 Premoze 在后续报告[81]中提到的天鹅绒 NDF 的一般化形式、以及方程 9.63 中的漫反射项。天鹅绒 NDF 的一般化形式如下：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})}{\pi\left(1+k_{\text{amp}}\alpha^2\right)}\left(1+\frac{k_{\text{amp}}\exp\left(\frac{(\mathbf{n}\cdot\mathbf{m})^2}{\alpha^2((\mathbf{n}\cdot\mathbf{m})^2-1)}\right)}{\left(1-(\mathbf{n}\cdot\mathbf{m})^2\right)^2}\right) \tag{9.73}$$

其中， α 控制了逆高斯分布的宽度， k_{amp} 控制了逆高斯分布的振幅。完整形式的布料 BRDF 如下：

$$f(\mathbf{l},\mathbf{v}) = (1-F(\mathbf{h},\mathbf{l}))\frac{\rho_{\text{ss}}}{\pi} + \frac{F(\mathbf{h},\mathbf{l})D(\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{l}+\mathbf{n}\cdot\mathbf{v}-(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v}))} \tag{9.74}$$

在游戏《神秘海域 4》[825]中使用了这种 BRDF 的变体，用于模拟羊毛和棉花等粗糙纤维织物。

Imageworks [947]在光泽项中使用了一个不同的逆 NDF，同时这个光泽项可以被添加到任何 BRDF 中，这个 NDF 的形式如下：

$$D(\mathbf{m}) = \frac{\chi^+(\mathbf{n}\cdot\mathbf{m})\left(2+\frac{1}{\alpha}\right)\left(1-(\mathbf{n}\cdot\mathbf{m})^2\right)^{\frac{1}{2\alpha}}}{2\pi} \tag{9.75}$$

尽管对于这个 NDF 而言，并不存在 Smith masking–shadowing 函数的闭式解，但是 Imageworks 使用了一个解析函数来逼近其数值解。Estevez 和 Kulla [442]讨论了 masking–shadowing 函数的细节，以及光泽项和 BRDF 其余部分之间的能量守恒。图 9.43 展示了一些使用 Imageworks 光泽项进行渲染的样例。

图 9.43：将 Imageworks 光泽高光项添加到一个红色漫反射项中。从左到右的光泽粗糙度值分别为 α = 0.15，0.25，0.40，0.65，1.0。[442]

到目前为止，我们所看到的各种布料模型，都仅限于对特定类型的布料进行模拟，而在下一节中讨论的模型则试图以一种更加一般的方式来对布料进行建模。

9.10.3 微圆柱体布料模型

用于布料材质的微圆柱体模型（micro-cylinder model），与用于头发材质的微圆柱体模型非常相似，因此章节 14.7.2 中有关头发模型的讨论，可以帮助你更好的理解这类模型。这类模型背后的思想是假设表面被很多一维线段所覆盖。Kajiya 和 Kay 针对该假设建立了一个简单的 BRDF 模型[847]，Banks 也为该模型提供了坚实的理论基础[98]，因此它也被称为 Kajiya-Kay BRDF 或者 Banks BRDF。这个理念基于了这样的一个观察：一个由一维直线所组成的曲面，它在任何给定位置上都具有无限多的法线，这些法线是由垂直于该位置切向量 $\mathbf{t}$ 的法线平面（normal plane）所定义的。虽然从这个理论框架中开发出了许多新的微圆柱体模型，但是由于其简单性，因此原来的 Kajiya-Kay 模型仍然有一些用途。例如：在游戏《神秘海域 4》[825]中，Kajiya-Kay BRDF 用于表示丝绸和天鹅绒等闪亮织物的高光项。

Dreamworks [348, 1937]使用了一种相对简单的、艺术家可控的微圆柱体模型来模拟布料材质，这个模型允许使用纹理用来改变粗糙度、颜色和线的方向，其中线的方向还可以指向表面以外，从而模拟天鹅绒以及其他类似的织物。这个模型还可以对经纬线（横向和竖向的线）设置不同的参数，从而模拟更加复杂的变色织物，例如闪光丝绸（shot silk）。

Sadeghi 等人[1526]提出了一种基于织物样品和单根纤维测量的微圆柱体模型，这个模型还考虑了线和线之间的 masking 和 shadowing 效应。

在某些情况下，头发的 BSDF 模型（章节 14.7）也可以用于布料模拟。RenderMan 的 PxrSurface 材质[732]有一个“模糊”波瓣，它使用了来自 Marschner 等人[1128]提出的头发模型中的 R 项（章节 14.7）。Wu 和 Yuksel [1924, 1926]在实时布料渲染系统中所实现的模型之一，就是源自迪士尼动画电影[1525]中所使用的头发模型。

9.11 波动光学的 BRDF 模型

我们在前几个节中所讨论的模型都依赖于几何光学，它把光的传播看作成一种射线而不是波。几何光学基于了这样的假设：任何表面的不规则性要么小于一个波长，要么

大于 100 倍波长（大约）。

而现实世界中的表面并没有那么理想，它们在所有尺度上都具有不规则性，包括 1–100 倍波长范围内。我们将 1–100 倍波长范围内的不规则性称为纳米几何（nanogeometry），来与前几节所讨论的微观几何区分开来，前文中所讨论的微观几何，虽然它们的尺寸很小，无法单独进行绘制，但是却都大于 100 倍波长。纳米几何对于反射率的影响无法使用几何光学来进行模拟。因为这些影响取决于光的波动性，需要使用波动光学（wave optics，也称为物理光学，physical optics）来对它们进行建模。

厚度接近于光波长的表面或者薄膜，会产生与光的波动性有关的现象。

在本小节中，我们将涉及诸如衍射（diffraction）和薄膜干涉（thinfilm interference）之类的波动光学现象，并讨论它们在真实感渲染中的重要性（有时效果会令人十分惊讶），虽然这些材质可能看起来十分普通。

9.11.1 衍射模型

纳米几何会引起一种叫做衍射（diffraction）的现象，为了解释这种现象，我们使用了 Huygens–Fresnel 原理，它假设波前（wavefront，指具有相同波相位的一组点）上的每一个点都可以看作是新球面波的源，如图 9.44 所示。

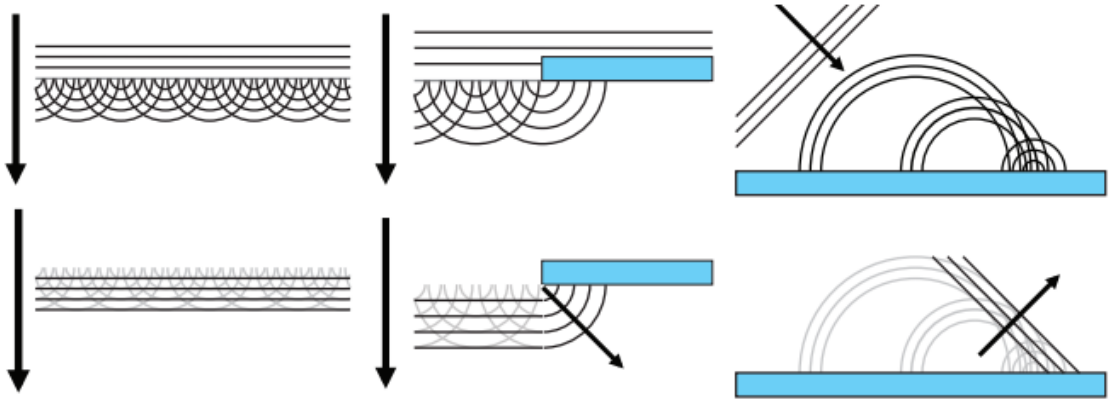


图 9.44：左图：我们可以看到一个平面波前在真空中传播。如果将波前上的每个点都视为新的球形波源，则新生成的波会在除正方向之外的所有方向上发生相消干涉，从而再次产生平面波前。中间：展示了波在遇到障碍物时发生的情况。对于障碍物边缘的球形波，在它的右侧并没有波会与它发生相消干涉，因此有些波会在边缘附近发生衍射或者“泄漏”。右图：一个平面波前被平面反射回来。平面波前会先遇到左侧的表面点，先遇到表面就会先被反射，因此从左侧表面点发射出的球面波会有更多的时间进行传播，

最终会变得更大。不同尺寸的球面波前，在沿着反射平面的边缘处会发生相长干涉，而在其他方向则会发生相消干涉。

当波遇到障碍物的时候，根据 Huygens–Fresnel 原理，它们会在拐角处轻微弯曲，这就是衍射的一个例子，这种现象是无法使用几何光学来预测的。当光线打到平面上时，几何光学确实可以正确预测光线在单一方向上的反射；也就是说，Huygens–Fresnel 原理提供了更多的见解和认识（insight）。这个原理表明，当表面上的球面波恰好排列成一条直线时，会形成反射波前，所有其他方向的波都会通过相消干涉被抵消。当我们在观察具有纳米级不规则性的表面时，这种认识会变得非常重要。由于各个表面点的高度是不同的，因此表面上的球面波不会排列得那么整齐，如图 9.45 所示。

图 9.45：左图：我们可以看到入射到粗糙纳米几何表面的平面波前。中间：我们可以看到表面上的球面波是根据 Huygens–Fresnel 原理形成的。右图：我们可以看到在发生相长干涉和相消干涉之后，产生的一些波（红色部分）形成了一个平面反射波。剩下的部分（紫色部分）则发生了衍射，每个方向上光的传播量都不同，这取决于光的波长。

正如上述两张图片中展示的那样，光会被散射到不同的方向上，其中一部分光会发生镜面反射，即在反射方向上叠加成一个平面波前。而剩余的光则会以一种定向模式衍射出来，这取决于纳米几何结构的具体特性。镜面反射光和衍射光的划分取决于纳米几何凸起的高度，或者更加准确地说，取决于高度分布的方差。衍射光在镜面反射方向上的扩散角度，取决于光的波长与纳米几何凸起的相对大小。然而有点违背直觉的是，更大尺度的不规则性反而会导致更小的扩散角度，如果这个不规则性大于 100 倍波长的话，那么衍射光和镜面反射光之间的角度会很小，几乎可以忽略不计。 不规则性的尺寸减小会导致衍射光扩散到更大的范围内，直到表面的不规则性小于光的波长，此时该点上也就不会有衍射现象发生了。

在具有周期性纳米几何结构的表面上，衍射现象是最清晰可见的，因为重复的表面结构通过相长干涉会增强衍射光，从而产生彩虹颜色，这种现象可以在 CD、DVD 光盘以及一些某些昆虫中观察到。虽然衍射现象也可以发生在非周期性表面上，但是多年来计算机图形学界一直认为这种情况下的衍射现象是十分轻微的。出于这个原因，除

了少数情况之外[89, 366, 686, 1688]，多年来计算机图形学文献大多都忽略了衍射现象。

然而，在最近 Holzschuch 和 Pacanowski [762]对测量材质的分析表明，在许多材质中存在着显著的衍射效应，这也许可以解释为什么当前的模型一直很难和这些材质与相匹配。Holzschuch 和 Pacanowski 的后续工作[763]提出了一种结合微表面理论和衍射理论的模型，他们在通用的微表面 BRDF（方程 9.26）中，使用了一个考虑衍射现象的 micro-BRDF。同时，Toisoul 和 Ghosh [1772, 1773]提出了一种用于捕捉由周期性纳米几何结构所产生的彩虹色衍射效果的方法，并且可以使用点光源和基于图像的照明来实时渲染这些效果。

9.11.2 薄膜干涉模型

薄膜干涉（Thin-film interference）是一种波动光学现象，当一个薄电介质层的顶部反射和底部反射的光路相互干扰时，就会发生这种现象，如图 9.46 所示。

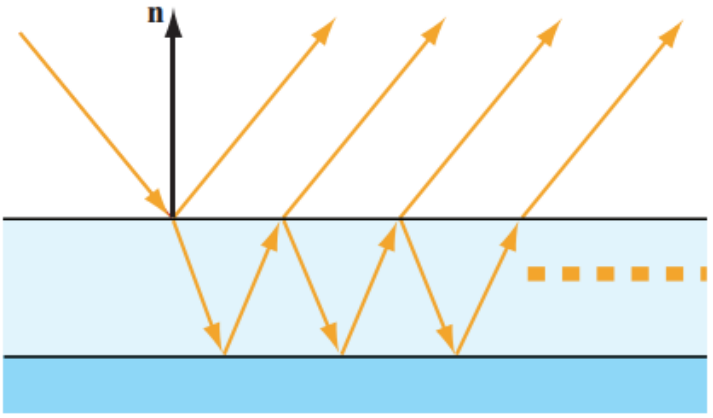


图 9.46：入射到一个反射基板的薄膜上的光线。除了主反射外，还有多条折射光路，这些光线要么在基板上被反射，要么在顶部的薄膜表面被内反射，要么直接折射穿过薄膜。这些光路都是同一个光波所产生的，但是由于光路长度的差异，它们具有一些很小的相位延迟，因此它们之间会发生干涉。

不同波长的光会发生相长干涉或者相消干涉，具体取决于波长和路径长度差之间的关系。由于路径之间的长度差异会随着角度的变化而变化，因此最终产生的结果是偏移了的彩虹颜色，因为不同波长的光会在相长干涉和相消干涉之间不断切换。

薄膜之所以需要很薄才能产生这种效果，其原因与相干长度（coherence length）的概念有关。这个长度是指光波的副本在移动之后，仍然可以与原始光波发生干涉的最大距离。这个最大长度与光的带宽（bandwidth）成反比，光的带宽是指其光谱功率分布（SPD）延申的波长范围。激光具有极窄的带宽，因此其相干长度也极长；根据

激光类型的不同，相干长度甚至可以达到几英里。这种关系是有道理的，因为一个很简单的正弦波在被许多不同波长的波取代之后，仍然会和原始波发生干涉。如果一个激光是完全单色的话（只包含一种波长的光），那么它就会具有一个无限大的相干长度，但是现实中的激光总会有一个不为 0 的带宽。相反，带宽极大的光将具有非常混乱无序的波形，具有这样一个波形的光波副本，只需要移动一段很短的距离，就会停止与原始波的干涉。

在理论上，理想的白光是由所有波长的光混合叠加而来的，其相干长度应当为 0。然而，对于可见光而言，人类视觉系统的带宽（仅能感知波长在 400 纳米–700 纳米范围内的光）最终决定了相干长度，大约为 1 微米。所以在大多数情况下，如果有人问你：“一层薄膜要多厚才不会引起可见的干涉现象？”这个问题的答案是：“大约 1 微米。”

与衍射现象类似，多年来，薄膜干涉被认为是一种特殊效应，只会发生在肥皂泡和油渍等表面。然而，Akin 指出[27]，薄膜干扰确实会给许多常见的表面带来一些微妙的色彩变化，模拟这种效果可以大大增加真实感，如图 9.47 所示。他的论文引起了人们对基于物理的薄膜干涉的极大兴趣，包括 RenderMan 的 PxrSurface 材质[732]以及 Imageworks 的着色模型[947]在内的各种着色模型，现在都已经支持了这种效果。

图 9.47：左侧的皮革材质没有薄膜干涉，右侧的皮革材质有薄膜干涉。由薄膜干涉所引起的高光变色，增强了图像的真实感。[27]

适合于实时渲染的薄膜干涉技术已经存在一段时间了。Smits 和 Meyer [1667]提出了一种有效的方法，来考虑一阶光路和二阶光路之间的薄膜干涉。他们观察到，最终产生的颜色主要是一个与路径长度差异有关的函数，这可以通过薄膜厚度、观察角度和

折射率计算出来。他们的实现需要一个具有 RGB 颜色的一维查找表，这个表中的数据可以使用密集光谱采样（dense spectral sampling）预先计算，并将其转换为 RGB 颜色进行存储，这使得该技术运行起来效率很高。在游戏《使命召唤：无限战争》中，使用了一种不同的快速薄膜近似方法，它被作为分层材质系统[386]的其中一部分。这些技术无法模拟光在薄膜中的多次反射，也不能模拟其他的物理现象。Belcour 和 Barla [129]提出了一种更加精确的技术，虽然计算成本变高了，但是仍然可以用于实时渲染。

9.12 分层材质

在现实生活中，表面的材质通常是相互重叠的。这些表面可能被灰尘、水、冰、雪所覆盖；也可能是出于装饰或者保护的原因，表面会被涂上漆或者其他涂层；又或是它可能包含了多层基本结构，例如许多生物材料。

其中最简单和最具视觉重要性的分层案例之一就是透明涂层（clear coat，也叫做清漆材质），它指的是涂在其他材质基底上的光滑透明层。例如：在一个粗糙的木材表面涂上一层光滑的清漆。迪士尼原则着色模型[214]便包含了一个透明涂层项，虚幻引擎[1802]、RenderMan 的 PxrSurface 材质[732]、以及 Dreamworks Animation [1937]和 Imageworks [947]所使用的着色模型，都包含这个透明涂层项。

透明涂层最显著的视觉效果，就是由光从透明涂层处反射与从底层基板处反射而产生的二次反射。当基底是金属材质的时候，这种二次反射的效果最为显著，因为此时透明电介质涂层和基底之间的折射率差异最大。当基底是电介质的时候，其折射率与透明涂层的折射率相近，因此第二次反射会相对较弱，这种效果类似于水下的材质，详见表 9.4。

透明涂层也可以有颜色。从物理角度来看，这种着色是光线被部分吸收的结果。根据 Beer–Lambert 定律（章节 14.1.2），光线被的吸收数量取决于穿过透明涂层的光路长度。而这个路径的长度则取决于观察视角、入射光的角度、以及材质的折射率。一些更加简单的透明涂层的实现，例如迪士尼原则模型和虚幻引擎，则没有对这种视角依赖性进行建模；而其他的实现则考虑到了这一点，例如 PxrSurface 材质，以及 Imageworks 和 Dreamworks 所使用的着色模型。Imageworks 的模型还允许任意数量的、不同类型的层叠加在一起。

一般情况来说，不同的层（透明涂层和基板）可以有不同的表面法线。例如水流流过平坦的路面、凹凸不平的土壤表面上铺上一层光滑的冰、或者在纸板箱上盖上皱巴巴的保鲜膜等。电影行业中所使用的大多数分层模型，都支持为每一层设置单独的法

线，而这种做法在实时图形应用中则并不常见，但是也有一些例外，例如虚幻引擎中的透明图层材质在实现上，将逐层的独立法线作为一个可选的特性。

Weidlich 和 Wilkie [1862, 1863]提出了一种分层的微表面模型，它假设表面层的厚度要比微表面的的尺寸小。他们所提出的模型支持任意数量的涂层相互叠加在一起，并且可以追踪从顶层到底层，再到顶层的反射事件和折射事件。它对于实时应用来说足够简单[420, 573]，计算成本也不是很高，但是这个模型并没有考虑层与层之间的多次反弹。Jakob 等人[811, 812]提出了一个全面而准确的框架模型用来模拟分层材质，这个框架还支持层与层之间的多次反弹；虽然它并不适合用于实时渲染，但是作为 ground-truth 而言还是十分好用的，而且它所使用的思想理念可能会对未来的实时分层材质有所启发。

游戏《使命召唤：无限战争》中所使用的分层材质系统[386]尤其引人注目，它允许用户将任意数量的材质层叠加在一起。该分层材质系统还支持折射、散射以、基于路径长度的层间吸收、以及逐层的表面法线。结合高效的实现方式，这个系统能够实现前所未有的复杂实时材质，尤其是对一个运行在 60 Hz 的游戏而言，如图 9.48 所示。

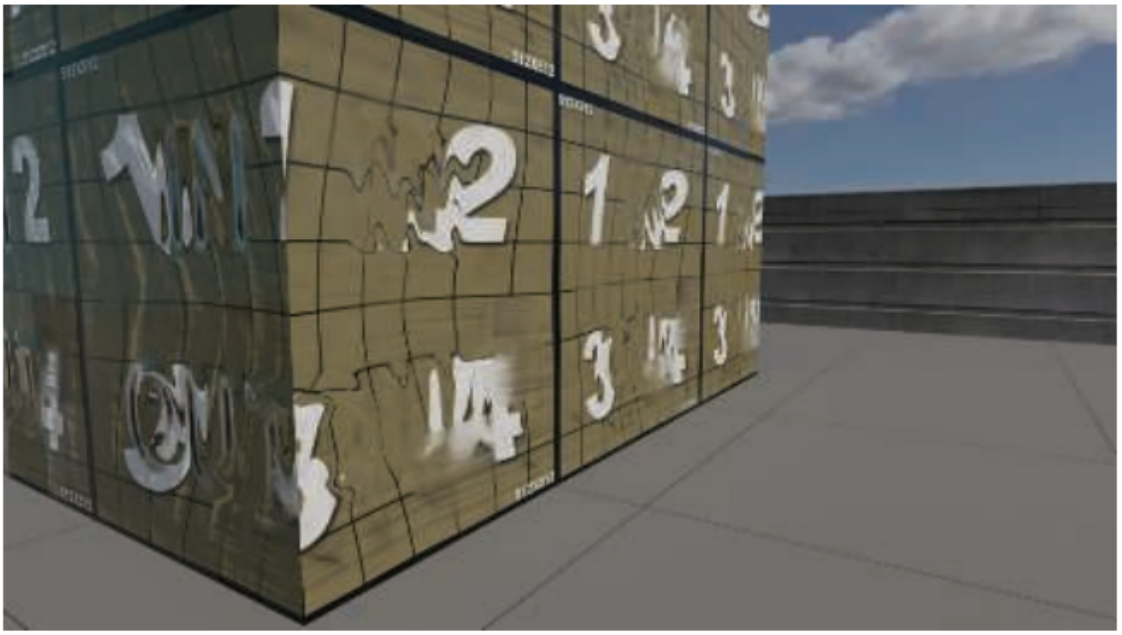


图 9.48：图中的测试表面展示了《使命召唤：无限战争》中多层材质系统的各种功能，虽然每立方体的表面仅由两个三角形构成，但是依然可以模拟出具有扭曲和散射的复杂几何表面。

9.13 混合和过滤材质

材质混合（material blending）是指将多种材质的属性（即 BRDF 参数）结合在一起的过程。例如：为了模拟一个带有锈迹的金属材质，我们可以绘制一个蒙版纹理来控制铁锈出现的位置，并使用它来混合铁锈和金属的材质属性（高光颜色 F_0，漫反射颜色 ρ_{ss} 和粗糙度 α）。材质的参数都存储在纹理中，每种材质的属性也都可以在空间上发生变化。可以通过提前创建新的纹理来实现材质混合（这个过程通常称为“烘焙”），或者也可以在着色器中进行实时处理。虽然表面法线 $\mathbf{n}$ 从技术上来说并不是一个 BRDF 参数，但是表面法线的空间变化对于材质外观的影响是非常重要的，因此材质混合通常也包括法线贴图的混合。

材质混合是很多实时渲染应用的关键。例如：游戏《教团：1886》就有一个十分复杂的材质混合系统[1266, 1267, 1410]，它允许美术人员从扩展库中获取素材，并创建任意深度的材质堆栈，并使用各种空间蒙版来进行控制。大多数材质混合都是离线预处理完成的，但是其中一些合成操作可以根据需要，延迟到运行时完成，这种运行时处理通常用于环境材质，用于为平铺纹理添加一些独特变化。广泛流行的材质创作工具 Substance Painter 和 Substance Designer 便使用了类似的方法来进行材质合成，Mari 纹理绘画工具也是如此。

将纹理元素动态混合在一起，可以实现很多种效果，同时节省内存开销，游戏应用基于各种目的使用了材质混合技术，例如：

- 展示建筑物、载具和生物（或不死生物）的动态伤害效果。[201, 603, 1488, 1778, 1822]
- 允许用户自定义游戏中的装备和服装。[604, 1748]
- 增加角色[603, 1488]和环境[39, 656, 1038]的视觉多样性，如图 20.5 所示。

有时，材质会以一种半透明的方式（即该层的不透明度小于 100%）混合在另一种的材质上；但即使是完全不透明的混合材质，在蒙版边界上也存在一些像素（或者纹素 texel，如果烘焙成纹理的话）需要进行部分混合。在任何一种情况下，正确的方法都是计算每种材质的着色模型，并将着色结果进行混合。但是这样需要计算两次着色过程，速度是很慢的，更快的方法是直接将 BRDF 的参数混合在一起，然后只计算一次着色。在材质属性与最终着色颜色之间具有线性关系或者近似线性关系的情况下（例如漫反射颜色和高光颜色），这种插值所引入的误差很小，或者根本没有误差。在很多情况下，即使材质参数与最终的着色颜色之间是高度非线性的（例如镜面粗糙度），在蒙版边界处所引入的误差也是可以接受的。

法线贴图的混合需要一些特殊考虑。通常可以从法线贴图中派生出高度贴图（height map），然后对高度贴图进行混合，从而获得不错的效果[1086, 1087]。在其他情况

下，例如在一个基础表面上叠加细节法线贴图（detail normal map），也可以使用其他形式的混合方法[106]。

材质过滤（material filtering）是一个与材质混合密切相关的话题。材质的属性通常都会存储在纹理贴图中，并通过 GPU 双线性滤波和 mipmap 等机制进行过滤。然而，这些机制都基于了这样一个假设，即被过滤的参数（着色方程的输入）与最终的颜色（着色方程的输出）之间存在着线性关系。但是这种线性关系只适用于某些参数，并不适用于所有的情况。在法线贴图上使用线性 mipmap 方法，或者是在包含非线性 BRDF 参数的纹理上（例如粗糙度）使用线性 mipmap 方法，都会产生瑕疵。这些瑕疵可能会表现为高光锯齿（闪烁高光）；或者是由于表面与相机之间的距离发生变化，从而导致表面光泽度和亮度的意外变化。在这两种瑕疵中，高光锯齿是更加明显的，消除这些瑕疵的技术通常被称为高光抗锯齿（specular antialiasing）技术，下面我们将讨论其中的几个方法。

9.13.1 过滤法线与法线分布

大部分材质过滤的瑕疵（主要来自高光锯齿），以及最常用的解决方案，都与法线和法线分布函数的过滤有关。由于这部分十分重要，因此我们将对这个话题进行一些深入讨论。

为了理解这些瑕疵发生的原因以及解决它们的方法，这里我们回顾一下 NDF，它是指亚像素表面结构的统计学描述。当相机与表面之间的距离增加时，之前在图像上覆盖多个像素的表面结构，现在只能覆盖若干个亚像素，即从凹凸贴图的控制领域转移到了 NDF 的控制领域中。这个转换过程与 mipmap 密切相关，mipmap 可以将纹理细节压缩到亚像素尺度上。

现在让我们思考一下物体是如何进行建模和渲染的，例如图 9.49 中左侧的圆柱体。对材质外观的建模总会预设一定的观察尺度：宏观尺度（macroscale）下的几何图形会被建模为三角形网格，细观尺度（mesoscale）下的几何图形会被建模为纹理，而小于单个像素的微观尺度下的（microscale）几何图形则会通过 BRDF 进行建模。

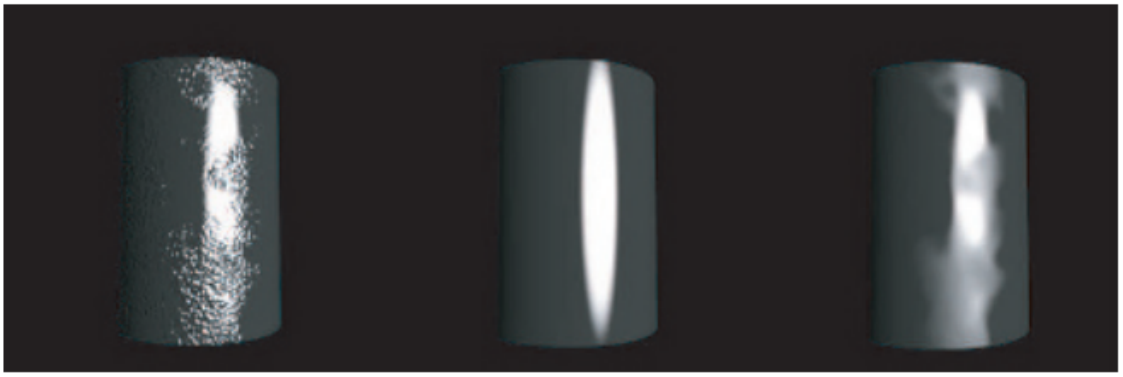


图 9.49：左侧：圆柱体使用原始的法线贴图进行渲染。中间：使用了一个分辨率低得多的法线贴图进行渲染，这个法线贴图中包含了平均化和重新归一化的法线，如图 9.50 左下角所示。右侧：圆柱体以相同的低分辨率法线贴图进行渲染，但同时使用了法线和光泽度值合适的理想 NDF，如图 9.50 右下角所示。右图明显更好地再现了原貌，而且这个表面在低分辨率下渲染时也不容易出现瑕疵。

根据图 9.49 所给定的观察尺度，将圆柱体建模为光滑网格（宏观尺度），并使用法线贴图（细观尺度）来表示凸起是较为合适的，同时使用了具有固定粗糙度 α_b 的 Beckmann NDF 来模拟微观尺度上的法线分布。这种组合表示方法在这个尺度上对圆柱体的外观进行了很好地模拟。但是，当观察尺度发生改变时会发生什么呢?

如图 9.50 所示，第一行展示了表面的其中一小部分，这部分表面被四个法线贴图的纹素所覆盖。假设我们按照这样的一个比例来渲染表面，使得法线贴图中的每个纹素刚好被一个画面像素所覆盖。对于法线贴图中的每个纹素而言，图中的红色箭头代表了法线（法线分布的平均值），它被 Beckmann NDF（黑色箭头）所包围。法线和 NDF 隐式表达了表面的微观几何结构，如图 9.50 第一行表面的横截面所示。中间的主要突起代表了法线贴图所定义的凸起，而周围的小摆动则代表了微观尺度的表面结构。法线贴图中的每个纹素，结合对应的粗糙度，可以被看作是，在这个纹素所覆盖的表面上收集到的法线分布。

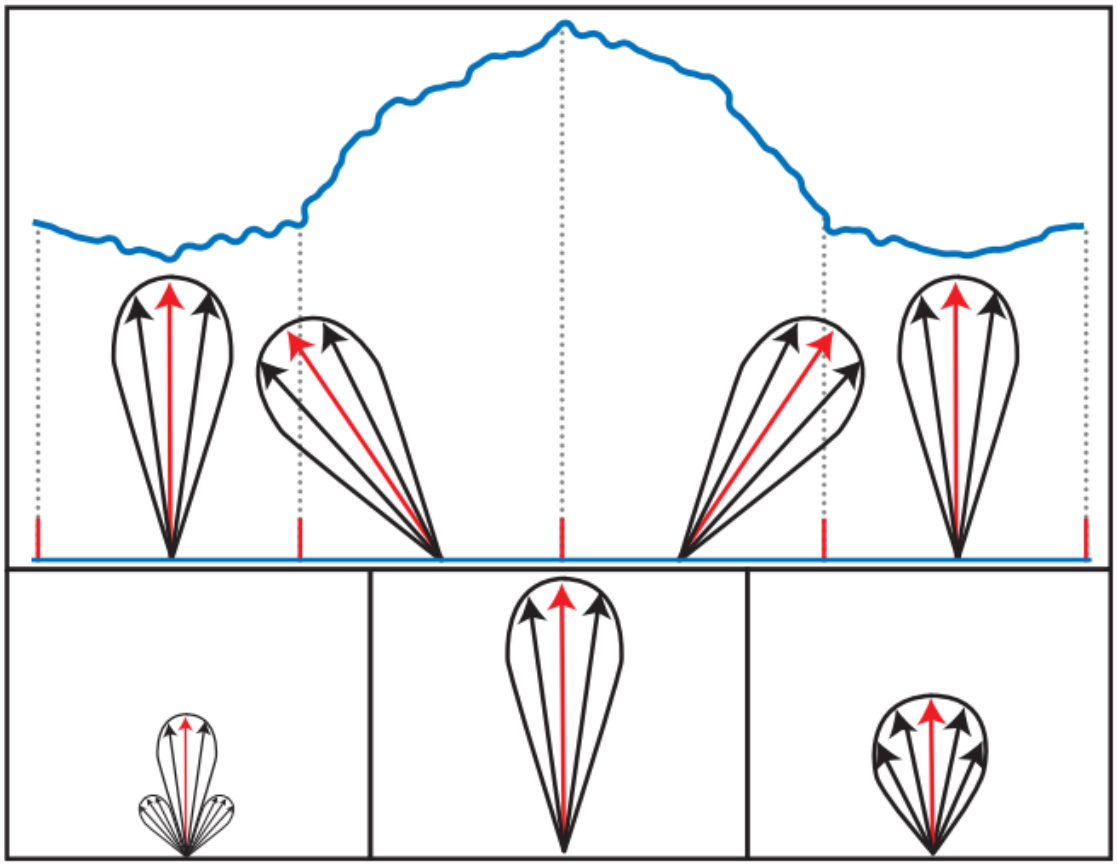


图 9.50：展示了图 9.49 中的部分表面。第一行显示了法线分布（其中的红色箭头代表了平均法线）和隐含的微观几何结构。第二行展示了将上述四个 NDF 平均为一个 NDF 的三种 mipmap 方法。其中左边的是 ground truth（将法线分布进行平均）；中间是对均值（法线）和方差（粗糙度）进行分别平均后的结果；右边使用了一个平均 NDF 来对 NDF 进行拟合。

现在我们假设相机到物体的距离变远了，此时屏幕上的一个像素只能覆盖四个法线贴图的纹素。在这个分辨率下，一个理想的表面表达方式应当是：能够准确表示该区域内所有法线的分布。这个分布可以通过计算上层 mipmap 中的，四个对应纹素的 NDF 的平均值来获得。图 9.50 左下角展示了这种理想的法线分布。这个平均后的法线结果如果用于渲染的话，可以最准确地代表在较低分辨率下的表面外观。

图 9.50 底部中间的图展示了分别对法线（法线分布的平均值）和粗糙度（对应了法线分布的扩散角度）进行平均的结果。最终的结果具有正确的平均法线分布（红色箭头），但是法线分布太窄了，这个误差会使得表面看起来过于光滑。更加严重的是，由于 NDF 太窄，因此将很容易造成瑕疵（闪烁高光）。

我们无法直接使用 Beckmann NDF 来表示理想的法线分布。但是如果我们额外使用一个粗糙度图的话，那么 Beckmann 粗糙度 α_a 就可以在不同的纹素之间发生变化。想象一下，对于每个理想的 NDF，我们都要找到与其最匹配的定向 Beckmann 波

瓣，无论是在方向上还是总体扩散角度上。我们将 Beckmann 波瓣的中心方向（即法线分布的平均值）存储在法线贴图中，将粗糙度值存储在粗糙度贴图中。最终结果如图 9.50 右下角所示，这个 NDF 更加接近理想中的情况。与简单的平均方法（图 9.50 底部中间的图）相比，使用这种方法可以更加真实地表示圆柱体的外观，如图 9.49 所示。

为了获得最佳的效果，应当对法线分布应用过滤操作（例如 mipmap），而不是对法线或者粗糙度值单独应用过滤操作，这样做意味着，需要使用一种稍微不同的方式来思考 NDF 和法线之间的关系。通常来说，NDF 定义在由法线贴图的逐像素法线所确定的局部切线空间中，然而，在过滤不同法线的 NDF 时，可以将法线贴图和粗糙度贴图的组合，看作是定义在底层几何表面的切线空间中的倾斜 NDF，这样来进行思考是十分有帮助的。

早期解决 NDF 过滤问题的尝试[91, 284, 658]是通过使用数值优化方法，来将一个或者多个 NDF 波瓣拟合到平均分布中。这种方法存在健壮性和速度上的问题，目前已经使用得不多了。相反，目前所使用的大部分技术，都是通过计算法线分布的方差来实现的。Toksvig [1774]观察到了这样一个事实：如果对法线进行平均化的而不是重新归一化的话，那么法线的平均长度与法线分布的宽度成反比。也就是说，原始法线分布的扩散角度越大，平均后得到的法线就越短。据此，他提出了一种基于法线长度来修改 NDF 粗糙度参数的方法，并使用修正后的粗糙度参数来重新计算 BRDF，从而对过滤后的法线扩散效果进行近似。

Toksvig 的原始方程是用于 Blinn–Phong NDF 的，其数学形式如下：

$$\alpha'_p = \frac{\|\overline{\mathbf{n}}\|\alpha_p}{\|\overline{\mathbf{n}}\| + \alpha_p(1 - \|\overline{\mathbf{n}}\|)} \tag{9.76}$$

其中 α_p 是原始的粗糙度值，α'_p 是修正后的粗糙度值；$\|\overline{\mathbf{n}}\|$ 是平均法线的长度。这个方程也可以使用等效替换 $\alpha_p = 2\alpha_b^{-2} - 2$（来自 Walter 等人[1833]）来得到 Beckmann NDF，因为这两个 NDF 的形状十分类似。使用 GGX 分布的方法就不太直接了，因为 GGX 和 Blinn–Phong（以及 Beckmann）之间并没有明确的等价关系。一种方法是强行让 α_g 和 α_b 相等，这样做的结果是，虽然高光中心的值是相同的，但是高光的外观却大不相同。更加麻烦的是，GGX 分布中并没有定义法线分布的方差，因此在与 GGX 分布一起使用的时候，这种基于方差的技术并没有扎实的理论基础。尽管有这些理论上的困难，但是在 GGX 分布中使用方程 9.76 的情况是相当普遍的，通常是令 $\alpha_p = 2\alpha_g^{-2} - 2$，这样做在实践中还是相当有效的。

Toksvig 方法的优点是在 GPU 纹理过滤中引入了法线方差，它也适用于最简单的法线 mipmap 方案（即对法线进行不归一化的线性平均）。这个特性对于那些动态生成的法线贴图（例如水面波纹）而言特别有用，因为对于这些法线贴图而言，它们的 mipmap 必须在运行过程中生成。但是这个方法对于静态的法线贴图而言并不是很好，因为它无法与常见的法线贴图压缩算法一起使用，原因在于这些压缩算法都要求法线是单位长度的。而 Toksvig 的方法依赖于平均法线的长度，这个长度是会发生变化的，因此使用这种方法的法线贴图必须要保持未压缩状态。但是即使这样，存储变短的法线也会导致精度问题。

Olano 与 Baker 的 LEAN 映射技术[1320]基于的是法线分布协方差矩阵的映射，与 Toksvig 的方法一样，它可以很好地兼容 GPU 纹理过滤和线性 mipmap，同时它还支持各向异性的法线分布。与 Toksvig 的方法类似，LEAN 映射技术可以很好地处理动态生成的法线，但是为了避免精度不足的问题，因此在处理静态法线的时候需要大量的存储空间。Hery 等人[731, 732]独立提出了一种类似的技术，并在皮克斯动画电影中被用于渲染亚像素级别的细节，例如金属薄片（metal flakes）和小划痕等。CLEAN 映射[93]是 LEAN 映射的一个简化变体，它不支持各向异性的法线，但好处是降低了存储空间。LEADR 映射[395, 396]对 LEAN 映射进行了扩展，同时也考虑了位移映射（displacement mapping）的可见性影响。

实时图形应用中所使用的大多数法线贴图都是静态的，并不是动态生成的，对于这种贴图而言，通常会使用方差映射技术。这些技术会在生成法线贴图 mipmap 的时候，会计算法线平均化所带来的方差损失。Hill 指出[739]，Toksvig 技术、LEAN 映射和 CLEAN 映射等技术，都可以使用这种方法来预先计算方差，从而消除这些技术在原始形式下的许多缺点。在某些情况下，预计算出来的方差值会单独存储在方差纹理以及它的 mipmap 中。通常，这些值会用于修改现有粗糙度贴图的 mipmap。例如：游戏《使命召唤：黑色行动》[998]中的方差映射技术便采用了这种方法。修正后的粗糙度值是通过将原始粗糙度值转换为方差，并加入法线贴图的方差，最终转换回粗糙度计算而来的。在游戏《教团：1886》中，Neubelt 和 Pettineo [1266, 1267]以一种类似方法使用了 Han [658]提出的技术，他们将法线贴图中存储的 NDF 与 BRDF 镜面项的 NDF 进行卷积，并将结果转换为粗糙度，最终将其存储在粗糙度贴图中。

还可以通过使用一些额外的存储空间来进一步改善结果，即在纹理空间的 xy 方向上分别计算方差，并将其存储在一个各向异性的粗糙度贴图中[384, 740, 1823]。这项技术本身局限于轴向的各向异性，这在一些人造表面中是很常见的，但是在自然形成表面中就不太常见了。还可以通过再多存储一个值，从而支持定向的各向异性（oriented anisotropy）[740]。

与原始形式的 Toksvig、LEAN 映射和 CLEAN 映射不同，方差映射技术并没有考虑由于 GPU 纹理过滤所引入的方差。为了弥补这一点，在方差映射的实现中通常会使用一个较小的滤波器[740, 998]，对法线贴图的顶级 mip 进行卷积。当组合多个法线贴图的时候（例如细节法线贴图）[106]，需要注意对法线贴图的方差进行正确组合[740, 960]。

模型的高曲率几何结构（高频信息）或者法线贴图会引入法线方差，前面我们所讨论的这些技术并不能缓解这种由方差所带来的的画面瑕疵，但是也有一些方法可以解决这些问题。如果在几何体上存在着唯一的纹理映射的话（通常是角色模型，较少与环境有关），那么模型的几何曲率可以被“烘焙”到粗糙度贴图中[740]。曲率还可以使用像素着色器中的导数指令来实时计算得出[740, 857, 1229, 1589, 1775, 1823]。如果可以使用法线缓冲的话，那么这种计算可以在几何体的渲染阶段中进行，或者是在处理后阶段中进行。

到目前为止，我们所讨论的方法主要集中在镜面反射上，但是法线方差同样也会影响到漫反射的着色。考虑法线方差对 $\mathbf{n} \cdot \mathbf{l}$ 项的影响，可以帮助提高漫反射着色和镜面反射着色结果的准确性，因为二者在反射率积分中都会乘上这一项[740]。

方差映射技术将法线分布近似为光滑的高斯波瓣，如果每个像素中包含数十万个凸起的话，那么这种近似是合理的，因为它们可以被很平滑的平均。但是在大部分情况下，一个像素只能覆盖几百个或者几千个凸起，这会导致表面具有“闪闪发光（sparkly）”的材质外观，我们可以在图 9.25 中看到这样的例子，它包含了一系列图像，不同图像中球体上的凸起会逐渐减小。其中右下角图像中球体的凸起非常小，它们被平均后形成了十分平滑的高光；但是对于左下角和底部中心的图像而言，虽然球体的凸起小于一个像素，但是还不足以被很平滑地平均，不足以产生平滑的高光。如果这些小球动起来的话，那么充满噪声的高光会表现为一帧一帧的不断闪烁。

如果我们将这样一个曲面的 NDF 渲染出来的话，那么它看起来会像图 9.51 左侧那样。当球体运动起来的时候，半向量 $\mathbf{h}$ 会在 NDF 上不断移动，穿过明亮和黑暗的区域，从而导致“闪闪发光”的外观。如果我们在这样的表面上使用方差映射技术，那么它会用一个类似于图 9.51 右侧的平滑 NDF，来对左侧的 NDF 进行近似，从而失去闪烁细节。

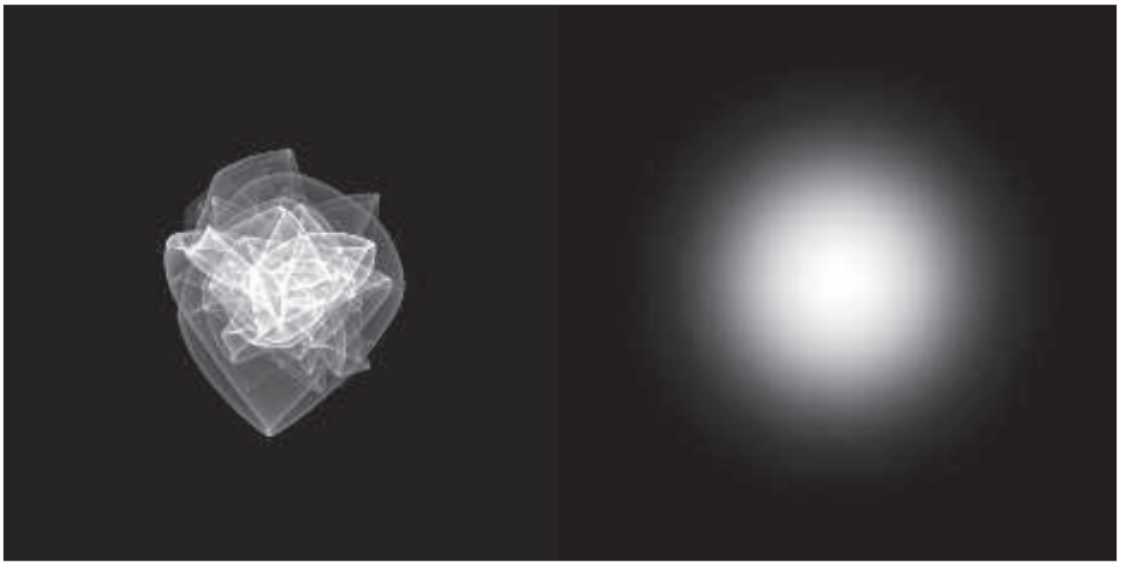


图 9.51：左侧是随机凹凸表面上的一小块区域（包含几十个凸起）的 NDF，右边则是一个大致具有相同宽度的 Beckmann NDF 波瓣。

在电影行业中，这个问题通常会使用大幅的超采样来解决，但是这在实时渲染应用中是不可行的，甚至在离线渲染中也是不可取的。但是已经出现了一些用于解决这个问题的技术，其中有一些并不适合实时渲染使用，但是却为未来可能出现的技术提供了研究方向[84, 810, 1941, 1942]。在这些方法中，有两种实现是用于实时渲染的。Wang 和 Bowles [187, 1837]提出了一种技术，用于在游戏《迪士尼无限 3.0（Disney Infinity 3.0）》中渲染闪闪发光的雪花。这个技术的目的是产生一种看似闪亮的外观，而不是真的去模拟特定的 NDF，它适用于具有稀疏闪光的材质，例如雪花等。Zirr 和 Kaplanyan [1974]的技术模拟了多个尺度上的法线分布，它可以用于更加广泛的材质外观，并且在空间上和时间上都是稳定的，

在本小节中，我们没有足够的篇幅来囊括所有有关材质过滤的文献，因此在这里我们将着重提及一些值得注意的参考文献。Bruneton 等人[204]提出了一种用于渲染海洋表面的技术，它支持环境光照，其中方差处理的尺度可以从几何到 BRDF。Schilling [1565]讨论了一种类似于方差映射的技术，该技术支持环境贴图的各向异性着色。Bruneton 和 Neyret [205]则对该领域的早期工作进行了全面概述。

补充阅读和资源

McGuire 的《Graphics Codex》[1188]和 Glassner 的《数字图像合成原理》[543, 544]对于本章节中所涉及的许多主题而言，都是一个非常好的参考资料。虽然 Dutre 的《Global Illumination Compendium》[399]中的一些部分有点过时（尤其是

BRDF 模型部分），但是它是渲染数学非常好的参考资料（例如球形积分和半球积分）。其中 Glassner 和 Dutre 的参考资料都可以在网上免费获取。

对于想要更多了解光和物质相互作用的读者，我们非常推荐 Feynman 的讲座[469]（可以在网上找到），其中的内容对于对于我们在撰写这一章的物理部分而言十分宝贵。另一个十分有用的参考资料是 Fowles 的《Introduction to Modern Optics》[492]，这是一个是简短易懂的入门文章。Born 和 Wolf [177]的《Principles of Optics》是一本更“重”（字面意思上也是）书，它包含了对光学的深入讲解。Nassau [1262]的《Physics and Chemistry of Color》描述了物体颜色背后的物理现象，非常彻底且详细。

Chapter 10 Local Illumination 局部光照

> Andrew Glassner——“Light makes right.”

> 安德鲁·格拉斯纳——“有了光，世界才变得美好。”（光明即正义。）（计算机图形专家）

在第 9 章中，我们讨论了基于物理的材质的相关理论，以及如何使用精确光源来计算它们。有了这些前置内容，我们就可以通过模拟光线与表面之间的相互作用，来进行着色计算，以便计算这个着色点在给定方向上，向虚拟相机发出了多少 radiance。这个光谱 radiance 是场景参考的像素颜色，它最终会被转换为图像中给定像素的显示参考颜色，这个过程详见章节 8.2。

实际上，我们需要考虑的交互作用从来都不是发生在某一个点上的。在章节 9.13.1 中，我们已经看到，为了正确计算着色效果，我们必须要对整个像素足迹（pixel footprint，即屏幕像素区域在表面上的投影面积）上的表面 BRDF 进行积分。这个积分的过程也可以被认为是一种抗锯齿方法。着色函数一般都是一个连续函数，并在一定区域范围内进行积分，我们并不会使用一个无上限的采样频率来对着色函数进行采样，相反我们会对其进行预积分。

到目前为止，我们只描述了点光源和方向光的效果，在这种光照条件下，表面只能接收来自少数离散方向上的光线，这种光照条件是不完整的。但是实际上，表面可以接收来自任意入射方向的光线。户外场景也不仅仅只是由太阳光（方向光）照亮的，如果只由太阳光照亮的话，那么所有处于阴影区域或者背对太阳的表面将会是纯黑的。因为太阳光会在大气层中发生散射，因此天空同样也是一个十分重要的光源。从月球的照片就可以看出天光的重要性，由于月球没有大气层，因此不存在天光，如图 10.1 所示。

图 10.1：在月球上拍摄的图像，由于缺乏散射太阳光的大气层，因此月球上并没有天光。这张图片展示了一个场景只被直接光源照亮时的样子，请注意在背对太阳的表面处，这些表面具有漆黑的阴影，同时没有任何表面细节。这张照片展示的是在阿波罗 15 号执行任务期间，宇航员 Astronaut James B. Irwin 站在月球漫游车旁边，图片前景区域的阴影来自于登月舱。这张照片由宇航员 David R. Scott 拍摄，他是此次任务的指挥官。

在阴天，黄昏或者黎明时分，室外的光照基本都来自于天光。即使是在晴朗的日子里，从地球上看向太阳，太阳也具有一定的面积，其所占据的视野区域是一个圆锥体，因此太阳并不是一个无穷小的光源。奇怪的是，虽然太阳和月球的尺寸差异巨大，太阳半径要比月球半径大两个数量级，但是从地球上看，它们基本具有相似的大小，二者直径所占据的视野角度大约都为半度。

在现实中，光源从来都不是精确的，但是在某些情况下，使用无穷小的实体来作为廉价近似，或者作为完整模型中的一部分是十分有用的。为了构建一个更加真实的光照模型，我们需要在表面着色点的半球范围内，对来自入射方向的 BRDF 进行积分。在实时渲染中，我们更喜欢通过找到方程的封闭形式解或者近似值，来求解渲染方程

（章节 11.1）所必须的积分。我们通常都会避免对多个样本（光线）的结果进行平均（蒙特卡洛方法），因为这种方法往往会很慢，如图 10.2 所示。

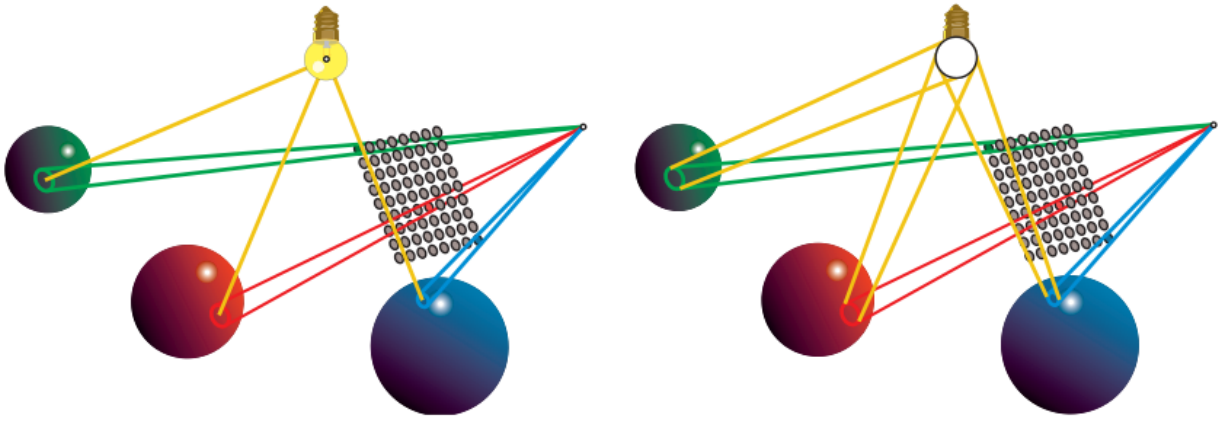


图 10.2：左图中展示的是我们在第 9 章中所看到的，对表面区域和定点光源进行积分。右图中展示的则是本章节的目标，我们将对着色方程和数学表达进行扩展，从而将光源表面上的积分考虑在内。

本章节将对这些解决方案进行探索和讨论，尤其是我们想通过计算各种非精确光源的 BRDF，来对之前所介绍的着色模型进行一些扩展。通常来说，为了找到一种廉价的解决方案（或者是任意的解决方案），我们需要对光源或者 BRDF 进行近似，有时候也会对二者同时进行近似。在一个感知框架中来评估最终的着色结果是十分重要的，这要求我们了解最终图像中哪些视觉元素更加重要，从而来为这些元素分配更多的精力和计算资源。

在本章节的最开始部分，我们将对面光源的公式进行解析积分。面光源将会成为场景中的主要光源，它负责大部分的直接光照强度，因此在计算面光源效果的时候，我们需要保留之前所有选择的材质属性。我们还会计算面光源的阴影，因为漏光会导致明显的视觉瑕疵。然后我们会对更加一般的光照环境的表示方法进行研究，这些光照环境由入射半球上的任意分布组成。在这些情况下，我们通常会使用一些更加近似的解决方案。环境光照通常用于表示大型、复杂但是不太强烈的光源，例如：包括来自天空和云的散射光；场景中大型物体反射的间接光（indirect light）；以及一些很暗的面光源的直接光照。这样的近似光源对于正确的图像白平衡十分重要，不然画面会显得太暗。即使我们在这里考虑了间接光照对场景照明的影响，但是我们仍然没有进入全局光照的领域（第 11 章），全局光照还依赖于场景中其他表面的显式知识和信息。

10.1 面光源

在第 9 章中，我们描述了理想的无穷小光源：精确光源和方向光。图 10.3 展示了一个表面着色点上的入射半球，以及无穷小光源和非零面光源（area light source）之间的差异。其中左侧光源使用了章节 9.4 中讨论的定义，它从单一方向 $\mathbf{l}_c$ 照射到表面上，其亮度由颜色 $\mathbf{c}_{light}$ 进行表示，这个颜色代表了从一个正对光源的白色 Lambertian 表面，所反射出的 radiance。点光源或者方向光在方向 $\mathbf{v}$ 上，对出射 radiance $L_o(\mathbf{v})$ 的贡献为 $\pi f\left(\mathbf{l}_c, \mathbf{v}\right) \mathbf{c}_{\text{light}} \left(\mathbf{n} \cdot \mathbf{l}_c\right)^+$，其中符号 x^+ 代表了将负数限制到 0，详见章节 1.2。相对地，面光源的亮度由其 radiance L_l 来进行表示，并且面光源与表面着色点位置之间会形成一个立体角，记为 ω_l，此时这个面光源在方向 $\mathbf{v}$ 上，对出射 radiance 的贡献是 $f(\mathbf{l}, \mathbf{v}) L_l (\mathbf{n} \cdot \mathbf{l})^+$ 在立体角 ω_l 上的积分。

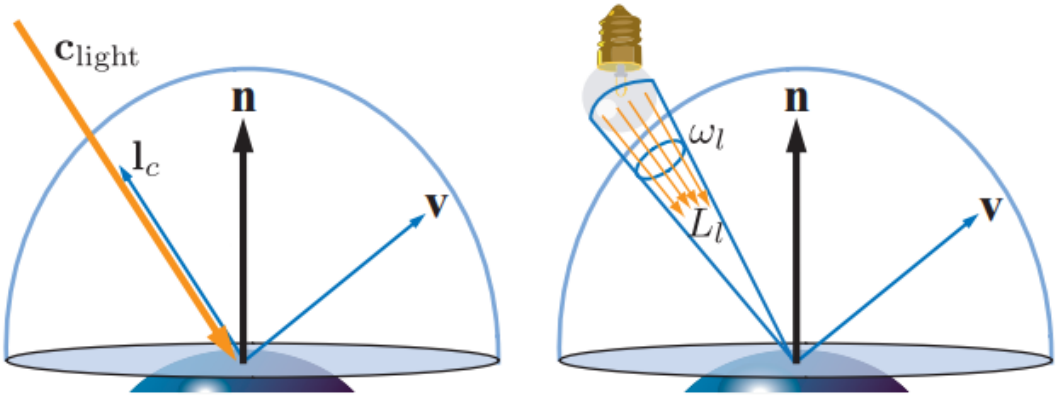


图 10.3：被光源照亮的表面，光源的入射方向位于由表面法线 $\mathbf{n}$ 所定义的半球范围内。左图中的光源是无穷小的；右图中的光源则被建模为一个面光源。

对无穷小光源的基本近似可以表示为如下形式：

$$L_o(\mathbf{v}) = \int_{\mathbf{l} \in \omega_l} f(\mathbf{l}, \mathbf{v}) L_l (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \approx \pi f\left(\mathbf{l}_c, \mathbf{v}\right) \mathbf{c}_{\text{light}} \left(\mathbf{n} \cdot \mathbf{l}_c\right)^+ \quad (10.1)$$

而面光源对表面着色点的照明贡献量，与 radiance（L_l）和从该位置所看到的光源尺寸（ω_l）有关。正如我们在章节 9.4 中所看到的，点光源和方向光是一种抽象近似，它在现实中并不存在，因为它们的可见立体角为 0，这意味着在单位面积上的 radiance 无穷大。理解这种由近似所引入的视觉误差，将有助于知道何时能够使用这种近似方法，以及在不能使用这种近似方法的时候，我们可以采取什么方法来进行替代。这些近似误差主要取决于两个因素：第一个是光源的尺寸有多大，即从着色点看向光源，该光源会覆盖多大的立体角；第二个则是表面的光泽程度。

图 10.4 展示了一个表面上的高光大小和高光形状，是如何受到表面材质的粗糙度和光源大小的影响的。对于一个很小的光源，它相对于相机视野只占据了一个很小的立体角，因此产生的误差也很小。相对于光泽表面来说，粗糙表面更加能够显示出光源

尺寸对高光的影响。一般来说，面向表面点的面光源，其发光情况与表面 BRDF 的镜面波瓣都是球面函数。如果考虑哪些方向集合对这两个函数的贡献最显著，我们可以得到两个立体角。误差的决定因素与面光源发射角和 BRDF 镜面高光立体角的相对大小成正比。

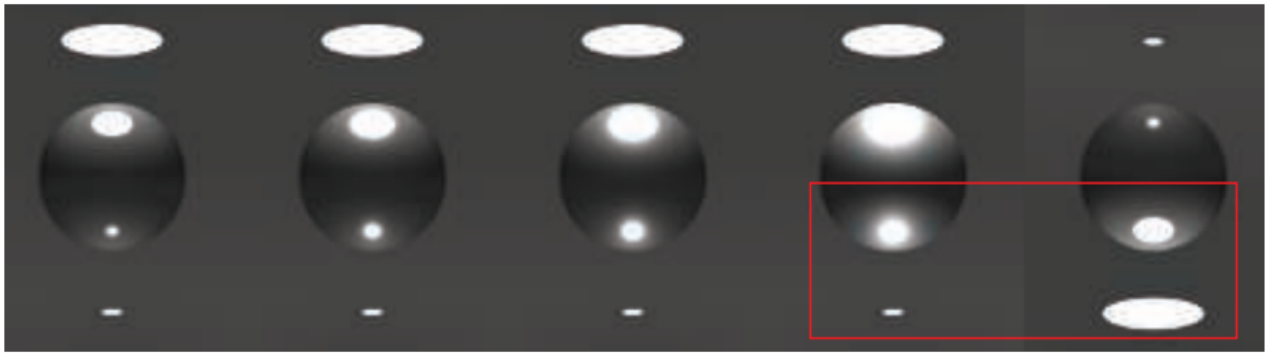


图 10.4：球体使用了 GGX BRDF 来进行渲染，从左到右，球体材质的表面粗糙度递增。最右侧图像和最左侧图像是一样的，只是将其垂直翻转了过来。请注意，在低粗糙度的材质上，由大圆盘灯引起的高光和着色效果，与较小光源在高粗糙度材质上所引起的高光效果，在视觉上面十分相似。

最后请注意观察，面光源的高光效果，可以通过使用精确光源并增加表面材质的粗糙度来进行近似。这个观察结果对于推导面光源积分的低成本近似十分有用，这也解释了为什么在实践中，许多实时渲染系统只使用精确光源就能够产生合理的结果：因为艺术家可以通过手动调整材质，来对误差进行一些补偿（compensate）。但是这样做肯定是有坏处的，因为它将材质属性与特定的光照设置耦合在了一起，当场景中的光照条件发生改变的时候，以这种方式生成的光影和画面看起来就不太正确了。

对于 Lambertian 表面这种特殊情况，直接使用点光源来表示面光源是很精确的。对于这样的表面，其出射的 radiance 与 irradiance 成正比：

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} E \tag{10.2}$$

方程中的 ρ_{ss} 是表面的次表面反照率（subsurface albedo），或者叫做漫反射颜色（diffuse color）。有了这个关系，我们可以对方程 10.1 进行等价变换，从而计算 irradiance，这样要简单得多：

$$E = \int_{\mathbf{l} \in \omega_l} L_l (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \approx \pi \mathbf{c}_{\text{light}} \ (\mathbf{n} \cdot \mathbf{l}_c)^+ \tag{10.3}$$

向量 irradiance（vector irradiance）的概念对于当面光源存在时，理解 irradiance 的行为十分有用。向量 irradiance 的概念由 Gershun [526]提出，他将其称之为光源向量（light vector），Arvo [73]将这个概念进一步推广。利用向量 irradiance，可以将任意大小和任意形状的面光源，精确地转换为点光源或者方向光。

想象一个分布为 L_i 的 radiance 进入到了空间中的点 $\mathbf{p}$，如图 10.5 所示。现在我们假设 L_i 与光线的波长无关，因此可以将其表示为一个标量。对于每个以入射方向 $\mathbf{l}$ 为中心的、无限小的立体角 $d\mathbf{l}$，我们都构建一个与入射方向 $\mathbf{l}$ 平行的向量，其长度等于从该方向入射的 radiance（标量）与 $d\mathbf{l}$ 的乘积。再对所有构建的向量进行求和（积分），最终可以得到向量 irradiance $\mathbf{e}$，这个过程的数学形式如下：

$$\mathbf{e}(\mathbf{p}) = \int_{\mathbf{l}\in\Theta} L_i(\mathbf{p},\mathbf{l})\mathbf{l}d\mathbf{l} \tag{10.4}$$

其中的 Θ 表示在整个球面方向上进行积分。

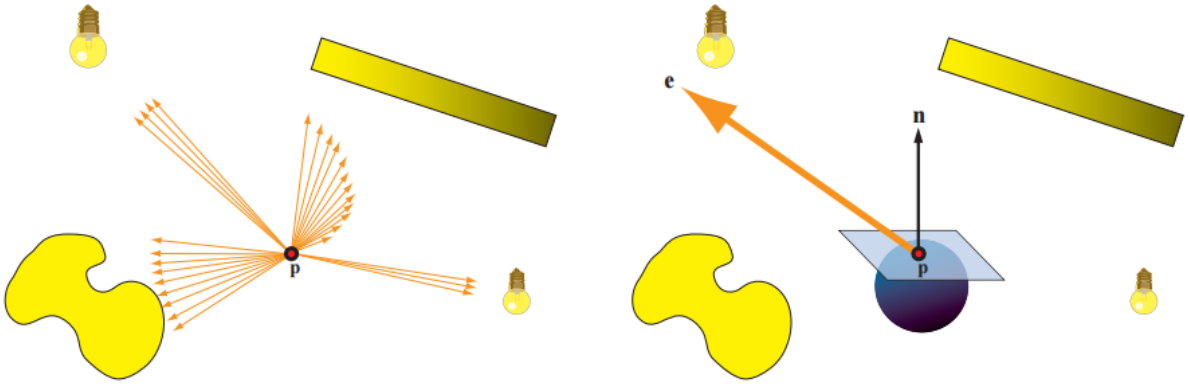


图 10.5：向量 irradiance 的计算过程。左图：点 $\mathbf{p}$ 被各种具有形状、大小、radiance 分布的光源所包围，其中黄色的明亮程度代表了光源发射出的 radiance 数量。以点 $\mathbf{p}$ 为起点的橙色箭头代表了向量，它指向任何存在入射 radiance 的方向，每个向量的长度，等于来自该方向上的 radiance 乘以箭头所覆盖的无限小的立体角。原则上应当存在无穷多个箭头。右图：向量 irradiance（橙色大箭头）是左图中所有橙色向量的总和。向量 irradiance 可以用来计算任意平面在点 $\mathbf{p}$ 处的净 irradiance。

向量 irradiance $\mathbf{e}$ 可以通过与任意平面的表面法线进行点乘运算，从而获得点 $\mathbf{p}$ 处的净 irradiance，其数学表达如下：

$$E(\mathbf{p},\mathbf{n}) - E(\mathbf{p},-\mathbf{n}) = \mathbf{n}\cdot\mathbf{e}(\mathbf{p}) \tag{10.5}$$

其中 $\mathbf{n}$ 是平面的表面法线。通过平面的净 irradiance，是指流经平面“正面”（表面法线 $\mathbf{n}$ 所指向的方向）和流经平面“背面”的 irradiance 之差。虽然说净 irradiance 本

身对于着色计算来说没有什么用处，但是如果说没有任何 radiance 被发射通过平面的“背面”的话（换句话说，对于所分析的光线分布，光线方向 $\mathbf{l}$ 和表面法线 $\mathbf{n}$ 之间的夹角都不会超过 90°，所有的入射光线都来自于着色点的正半球方向），即 $E(\mathbf{p}, -\mathbf{n}) = 0$，那么此时有：

$$E(\mathbf{p}, \mathbf{n}) = \mathbf{n} \cdot \mathbf{e}(\mathbf{p}) \tag{10.6}$$

单个面光源的向量 irradiance 可以实用方程 10.6 中的方法，来照亮具有任意法线 $\mathbf{n}$ 的 Lambertian 表面，只要这个面光源整体位于表面的正面方向即可。也就是说面光源上任意一点与着色点之间的连线，和表面法线之间的夹角都不超过 90°，如图 10.6 所示。

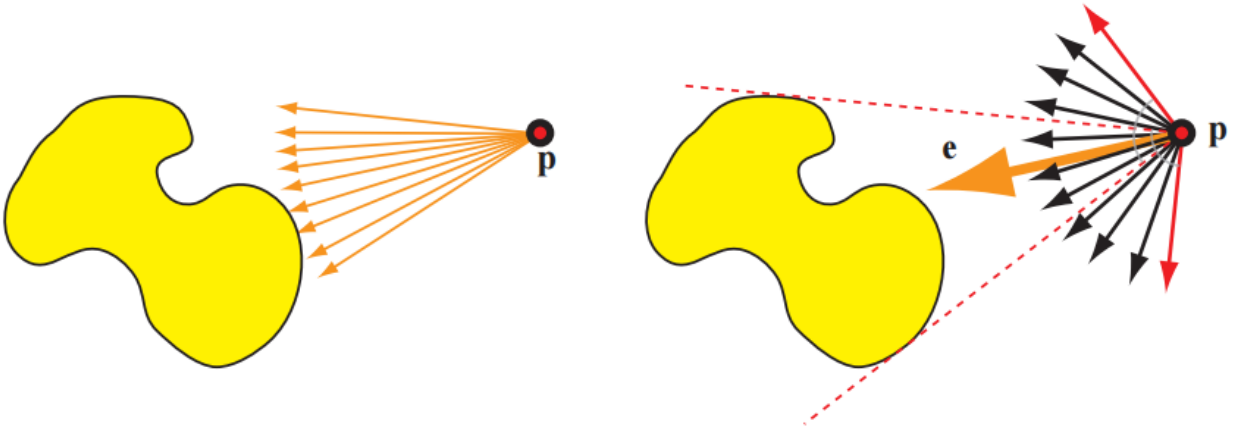


图 10.6：单个面光源的向量 irradiance。左图中的箭头代表用于计算向量 irradiance 的单个向量。右图中的橙色大箭头代表的就是向量 irradiance $\mathbf{e}$；红色虚线代表了光源的范围；红色向量（每个红色向量都垂直于一条红色虚线）定义了表面法线的极限范围，位于这个范围外的表面法线，将会与面光源的部分区域呈大于 90° 的夹角，这样的法线无法使用 $\mathbf{e}$ 来正确计算它们的 irradiance。

如果入射 radiance L_i 与波长无关的假设不成立的话，那么在一般情况下，我们就不能再定义单个向量 irradiance $\mathbf{e}$ 了。然而，彩色光通常在所有点上都具有相同的相对光谱分布，这意味着我们可以将 L_i 分解为颜色 $\mathbf{c}'$，以及与波长无关的 radiance 分布 L_i'。在这种情况下，我们可以计算 L_i' 的向量 irradiance $\mathbf{e}$，并对方程 10.6 进行一些扩展，将 $\mathbf{n} \cdot \mathbf{e}$ 乘上颜色 $\mathbf{c}'$。这样做的结果与计算方向光 irradiance 的方程相同，只是做了以下替换：

$$\begin{aligned} \mathbf{l}_c &= \frac{\mathbf{e}(\mathbf{p})}{\|\mathbf{e}(\mathbf{p})\|}, \\ \mathbf{c}_{\text{light}} &= \mathbf{c}' \frac{\|\mathbf{e}(\mathbf{p})\|}{\pi}. \end{aligned} \tag{10.7}$$

到此为止，我们已经可以将任意形状和任意大小的面光源转换为方向光，同时不会引入任何误差。

对于一些简单情况，用于求取向量 irradiance 的方程 10.4 可以求出解析解。例如：想象现在有一个以 $\mathbf{p}_l$ 为中心、半径为 r_l 的球形光源。球面上的每一点都会向各个方向发出具有恒定 radiance L_l 的光线。对于这种光源，将方程 10.4 和方程 10.7 联立，可以获得如下结果：

$$\begin{aligned} \mathbf{l}_c &= \frac{\mathbf{p}_l - \mathbf{p}}{\|\mathbf{p}_l - \mathbf{p}\|}, \\ \mathbf{c}_{\text{light}} &= \frac{r_l^2}{\|\mathbf{p}_l - \mathbf{p}\|^2} L_l. \end{aligned} \tag{10.8}$$

上述方程，与具有 $\mathbf{c}_{\text{light}_0} = L_l, r_0 = r_l$ 和标准距离平方反比衰减函数的泛光灯（章节 5.2.2）完全相同。可以对这个衰减函数进行一些修正，使得光线从球体表面才开始发生衰减，并在光源的最大影响距离处衰减到 0，有关这些调整的更多细节，详见章节 5.2.2。

> 请注意，对于球形光源而言，确实会采取通常的距离平方反比函数来控制光线强度的衰减，但是这种方式通常并不适用于所有形状的面光源。值得注意的是，圆盘光的衰减率与 $1/(d^2+1)$ 成正比。

在没有来自“背面”irradiance 的情况下，上述所描述的一切都是正确的。另一种思考方式是，面光源的任何部分都不能“位于地平线以下”，或者是被表面遮挡。我们可以将这个说法推广，对于表面而言，面光源和点光源之间的所有差异，都是由于遮挡差异所造成的。对于光线没有被遮挡的任意着色点，点光源的 irradiance 服从余弦衰减定律。Snyder 推导出了一个考虑遮挡情况的球形光源的解析表达式 [1671]，这个表达相当复杂。然而，由于这个表达式只和两个量有关（r/r_l，表面法线 $\mathbf{n}$ 和 $\mathbf{l}_c$ 之间的夹角 θ_i），因此可以对其进行预计算，并存储在一个二维纹理中。Snyder 还给出了两个适用于实时渲染的函数近似。

在图 10.4 中我们可以看到，对于较为粗糙的表面，面光源的光照效应其实不太明显。基于这一观察结果，我们可以使用一种不太物理正确，但是仍然有效的方法，来对 Lambertian 表面上的面光源效果进行模拟，这种方法叫做环绕光照（wrap lighting）。在这种方法中，首先会对 $\mathbf{n} \cdot \mathbf{l}$ 的结果进行一些简单的修改，然后再将其限制到 0。Forsyth [487]给出了环绕光照的一种形式：

$$E = \pi \mathbf{c}_{\text{light}} \left(\frac{(\mathbf{n} \cdot \mathbf{l}) + k_{\text{wrap}}}{1 + k_{\text{wrap}}} \right)^{+} \tag{10.9}$$

其中 k_{wrap} 的取值范围为 $[0, 1]$，对于点光源而言，$k_{wrap} = 0$；对于覆盖整个半球范围的面光源而言，$k_{wrap} = 1$。Valve 采用了另一种形式的环绕光照[1222]，来模仿大尺寸的面光源效果：

$$E = \pi \mathbf{c}_{\text{light}} \left(\frac{(\mathbf{n} \cdot \mathbf{l}) + 1}{2} \right)^{2} \tag{10.10}$$

一般来说，如果我们要计算面光源的话，我们还应当对阴影计算进行一定的修改，如果我们不这样做，一些面光源的视觉效果可能会被强烈的阴影所抵消[193]。正如我们在第 7 章中所讨论的，柔和的软阴影可能是面光源最明显的视觉效果。

10.1.1 光泽材质

面光源对非 Lambertian 表面的影响要更为复杂。Snyder 推导出了一个球形光源的解[1671]，但是这个解仅适用于原始的反射向量 Phong 材质模型，并且极其复杂，在实际应用中需要对其进行近似。

面光源在光泽（glossy）表面上的主要视觉效果是高光，如图 10.4 所示。这个高光的大小和形状类似于产生该效果的面光源，高光的边缘则会根据该表面的粗糙度，被一定程度的模糊。基于这一观察结果，研究者提出了几个针对该效应的经验近似值，这些近似方法在实践中是相当令人信服的。例如：我们可以对高光计算的结果进行修改，将其加上一个截止阈值，从而生成一个大而平坦的高光区域[606]。这可以有效地产生一个球面光源镜面反射的视错觉，如图 10.7 所示。

图 10.7：面光源在光滑物体上强烈反射所产生的高光，高光的形状与面光源的形状相类似。左图中，通过限制 Blinn–Phong 着色器的高光阈值，来对这种视觉现象进行近似。右图中，使用未修改的 Blinn–Phong 着色器对物体进行渲染，以便进行比较。

在实时渲染中，大多数对面光源光照效果的实用近似，都是基于了这样的一个想法：为每个着色点都寻找一个等效的精确光源，从而模拟非无穷小光源的效果。这种方法经常被用于实时渲染中，以解决各种各样的问题。这与我们在第 9 章中，在表面像素足迹所覆盖的区域内，对 BRDF 进行积分的原理相同。这种方式所产生的近似值，其计算成本通常会很低，因为我们只需要修改着色方程的输入即可，不会引入任何其他额外的复杂性。由于背后的数学运算并没有发生改变，因此我们通常可以保证，在某些条件下，我们可以恢复对原始着色结果的计算，并保留其所有的属性。由于大多数渲染系统的着色代码都是基于精确光源的，因此用它们来实现面光源只会引入局部的代码变化。

第一个被开发出来的近似方法是 Mittring 在虚幻引擎的“Elemental demo”[1229]中所使用的粗糙度修正（roughness modification）。这个想法首先会找到一个圆锥体，其中包了含入射到表面半球范围内的大部分光源 irradiance。然后我们在镜面波瓣的附近放置一个类似的圆锥体，它会包含“大部分”的 BRDF，如图 10.8 所示。这两个圆锥体是半球上函数的替代品，它们各自包含了一组方向，在这组方向上，这两个函数的输出值都会大于给定的任意阈值。这样做之后，我们可以寻找一个新的 BRDF 波瓣，来近似光源和材质 BRDF 之间的卷积，这个新的 BRDF 波瓣具有不同的粗糙度，它同样有一个对应的圆锥体，这个圆锥体的立体角等于光源波瓣立体角和材质波瓣立体角的和。

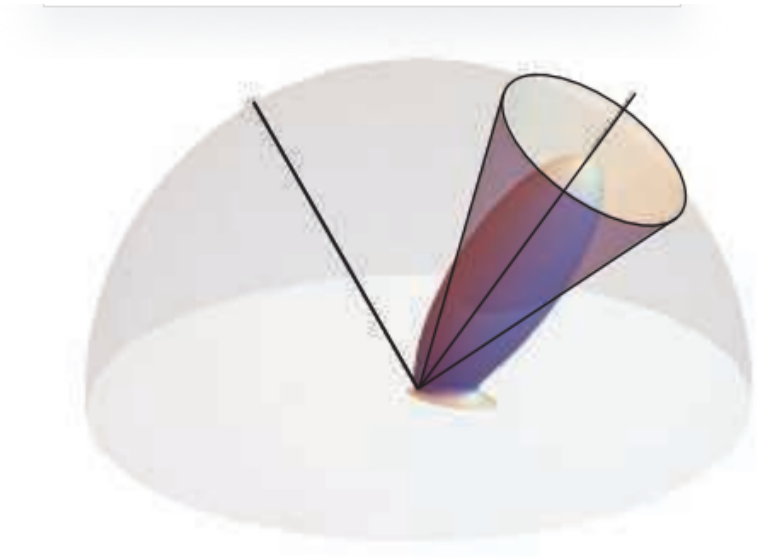


图 10.8：GGX BRDF 和一个圆锥体，这个圆锥体包含了一组方向，其中镜面波瓣会在这组方向上反射大部分光源的入射 radiance。

Karis [861]将 Mittring 所提出原理应用在了 GGX/Trowbridge–Gereitz BRDF（章节 9.8.1）和一个球面光源中，并对 GGX 的粗糙度参数 α_g 进行了简单的修改：

$$\alpha'_g = \left(\alpha_g + \frac{r_l}{2\left\|\mathbf{p}_l - \mathbf{p}\right\|}\right)^{\mp}$$

注意这里使用了章节 1.2 中介绍的符号 $x^{\mp}$，它表示将输入值限制到 0–1 之间。对于具有一定光泽的表面而言，这种近似方法的效果相当好，而且计算成本很低；但是对于闪亮的、几乎像镜子一样的材质来说就会失效。这种失效的原因是因为镜面波瓣总是光滑的，它无法模拟由面光源在表面上因尖锐反射所产生的锐利高光。此外，大多数微表面 BRDF 模型的波瓣并不是那么“紧凑”（即反射分布的扩散角度较大），同时会表现出一个较宽的衰减（高光拖尾），这使得粗糙度重映射的方法不太有效，如图 10.9 所示。

图 10.9：球形光照。从左至右分别是：使用数值积分方法计算出来参考结果、粗糙度修正技术、代表性点技术。

除了改变材质的粗糙度之外，还有另一个思路：根据被着色的表面点来改变的光线方向，从而来表示面光源的照明效果。这种方法被称为代表性点技术（most representative point），它通过对光线向量进行修改，使其在面光源表面方向上，能够对着色表面产生最大的能量贡献，如图 10.9 最右侧所示。Picott [1415]使用了光源表面上与反射光线夹角最小的点。Karis [861]对 Picott 的公式进行了改进，他将这个形成最小角度的点，近似为面光源到反射光线最近的点，从而进一步提高效率。他还提出了一个廉价的公式来对光线强度进行缩放，从而尽可能保持整体发射的能量，如图 10.10 所示。大多数代表性点的解都是很容易推导的，并且这种方法适用于各种几何形状的光源，因此对它们背后的理论进行一些了解是很重要的。这些方法的核心思路，类似于蒙特卡洛积分中重要性采样的思想，蒙特卡洛积分是指，我们通过在积分域上对样本结果进行平均，从而对定积分的值进行数值计算。为了更高效地做到这一点（使用较少的样本），我们可以尝试优先考虑那些对总体平均值具有较大贡献的样本。

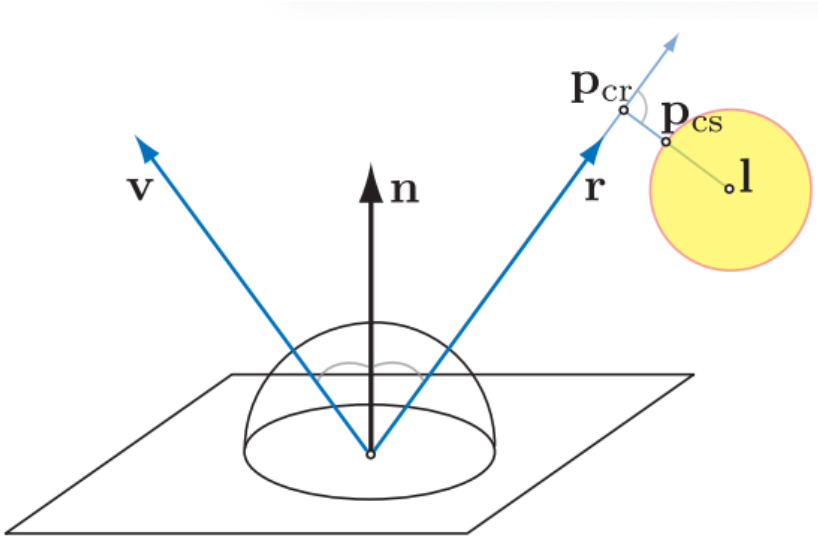


图 10.10：球的 Karis 代表性点近似。首先，计算反射光线上最接近球体中心 $\mathbf{l}$ 的点：$\mathbf{p}_{cr} = (\mathbf{l} \cdot \mathbf{r})\mathbf{r} - \mathbf{l}$。然后再计算球面上距离点 $\mathbf{p}_{cr}$ 最近的点：$\mathbf{p}_{cs} = \mathbf{l} + \mathbf{p}_{cr} \cdot \min\left(1, \frac{\text{radius}}{\|\mathbf{p}_{cr}\|}\right)$

定积分的中值定理可以更加严格地证明其有效性的，它允许我们用函数的单次求值来代替该函数的积分，其数学表达如下：

$$\int_D f(x)dx = f(c)\int_D 1 \tag{10.11}$$

如果 $f(x)$ 在区域 D 上是连续的，则 $\int_D 1$ 代表了该区域的面积，点 $c \in D$ 位于区域 D 中函数最小值与最大值之间的连线上。对于光照，我们考虑的是 BRDF 与光源 irradiance 的乘积，这个乘积在被光照覆盖的半球面上的积分。通常我们会认为光源是均匀照射的，因此我们只需要考虑光线随距离的衰减即可，并且大多数近似方法还会假设区域 D 在着色点位置上是完全可见的。即使有了上述这些假设，确定点 c 和归一化系数 $\int_D 1$ 的计算成本仍然可能会很高，因此我们需要对其进行进一步的近似。

代表性点的解也可以通过它们对高光形状的影响来限定。在部分的表面着色点上，代表性点可能不会发生改变，因为反射向量位于面光源所覆盖的方向锥之外，对于这些区域，我们可以使用一个点光源来进行高效照明，因为高光的形状只取决于镜面波瓣的底层形状。而对于那些反射向量指向面光源的表面着色点，其代表性点将会不断变化，以便能够指向能量贡献最大的方向。这样做有效地扩展了镜面波瓣的峰值，使其“变宽”，其效果类似于图 10.7 左侧的硬阈值。

这个较宽的、恒定的高亮峰值，也是近似值中剩余的误差来源之一。在较为粗糙的表面上，面光源反射看起来要比真实值（ground-truth，即通过蒙特卡罗积分获得）更加“尖锐”，这是一种与粗糙修正技术过度模糊相反的视觉瑕疵。为了解决这个问题，Iwanicki 和 Pesce [807]将 BRDF 波瓣、软阈值、代表性点参数与缩放因子，拟合到通过数值积分计算而来的球形区域光照结果中，从而得到了近似结果。这些拟合的函数生成了一个参数表，这个参数表通过材质粗糙度、球形光源的半径、光源中心与表面法线之间的夹角、观察向量来进行索引。由于在着色器中直接使用这种多维查找表的开销很大，因此他们还提供了封闭形式的近似。最近，de Carpentier [231]推导出了一种改进的方程，这个方程对于基于微表面的 BRDF，可以更好地保留掠射角度下球形面光源所生成的高光形状。该方法的原理是找到一个使得 $\mathbf{n} \cdot \mathbf{h}$ 最大化的代表性点，而不是原始方程中的 $\mathbf{n} \cdot \mathbf{r}$ （这是通过 Phong BRDF 推导出的），这里的 $\mathbf{n} \cdot \mathbf{h}$ 是表面法线和半向量（光线方向-观察方向）之间的点积。

10.1.2 一般光源形状

到目前为止，我们介绍了几种方法，它们可以从均匀辐射的球形光源和任意的光泽 BRDF 中计算着色效果。其中的大多数都采用了各种近似策略，从而能够获得可以快速实时计算的数学方程，但是与 ground-truth 的方法相比，它们都具有不同程度的误差。然而，即使我们的计算能力足够强，能够推导出完全精确的解决方案，我们仍然会犯一个很大的错误，这个错误来源于我们在光照模型中的假设。因为现实世界的光源通常并不是球形的，而且它们向外辐射的能量也并不是完美均匀的，如图 10.11

所示。但是球形光源在实践中仍然非常有用，因为它们提供了一种最简单的方法，来打破精确光源所引入的光照和表面粗糙度之间的错误关联。然而，只有在光源相对较小的情况下，球形光源才能对现实中的大多数光源进行良好的近似。

图 10.11：常用面光源的形状。从左到右依次：球形、矩形（卡牌光源）、管状（线光源）、管状的聚焦发射光源（辐射能量在半球上的分布不均匀，集中在光源的法线方向上）。请注意它们所生成的、不同形状的高光。

基于物理的实时渲染，其目标是生成令人信服的、可信的图像，因此我们只能通过将自己限制在一个理想化的场景中来实现这个目标。这就是在计算机图形学中反复出现的 trade-off（权衡），通常我们可以选择为简化假设的简单问题生成精确的解，或者是为更加一般化的问题推导出一个近似的解决方案，从而更好的对现实进行模拟。即选择简单问题的精确解，还是选择复杂问题的近似解。

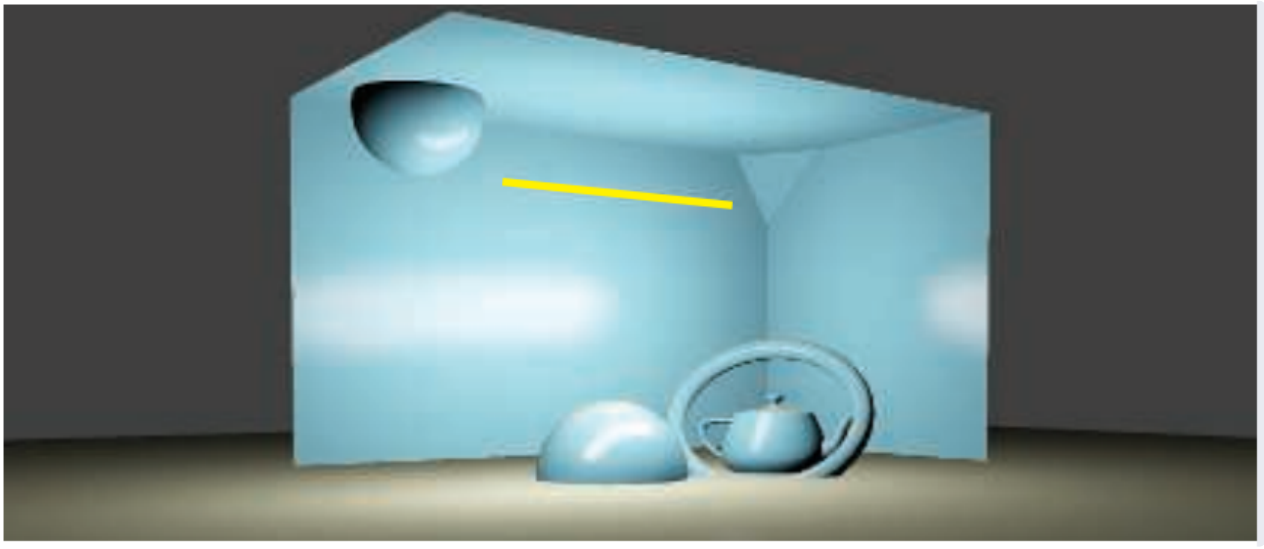


图 10.12：一个管状光源。使用了代表性点方法来计算图像中的光照效果。[807]

管状光源（也叫做胶囊体光源）是对球形光源的最简单扩展之一，它可以用来表示现实世界中的荧光灯管，如图 10.12 所示。对于 Lambertian BRDF，Picott [1415]给出了一个封闭形式的光照积分方程，它相当于在线性光源分段（light segment）的极值处，使用适当的衰减函数来评估两个点光源的照明效果，其数学形式如下：

$$\int_{\mathbf{p}_0}^{\mathbf{p}_1}\left(\mathbf{n}\cdot\frac{\mathbf{x}}{\|\mathbf{x}\|}\right)\frac{1}{\|\mathbf{x}\|^2}d\mathbf{x}=\frac{\frac{\mathbf{n}\cdot\mathbf{p}_0}{\|\mathbf{p}_0\|^2}+\frac{\mathbf{n}\cdot\mathbf{p}_1}{\|\mathbf{p}_1\|^2}}{\|\mathbf{p}_0\|\,\|\mathbf{p}_1\|+(\mathbf{p}_0\cdot\mathbf{p}_1)} \tag{10.12}$$

其中 $\mathbf{p}_0$ 和 $\mathbf{p}_1$ 是线性光源的两个端点， $\mathbf{n}$ 是表面法线。Picott 还推导了一个 Phong 镜面 BRDF 积分的代表性点解，来将其作为位于光源分段位置上点光源照明的近似，将这个代表性点与表面着色点相连接，可以与反射向量形成最小的夹角。这个代表性点的解可以动态地将线性光源转换为一个点光源，因此我们可以使用任意球形光源的近似，通过“叠加”的方式来构建一个胶囊光源。

在球形光源的情况下，Karis [861]在 Picott 原始解决方案的基础上，提出了一个更加高效（但是不太准确）的变体，该方法使用了与反射向量距离最短的光源表面点（而不是最小的夹角），并提出了一个缩放公式，试图恢复光照中的能量守恒。

许多其他光源形状的代表性点近似也可以很容易地获得，例如圆环光源和 Bezier 光源，但是通常我们并不希望着色器的分支过多。一个良好的光源形状，应当能够表示许多现实世界中的光源。其中最具表现力的一类形状是平面光源（planar area light），它位于一个平面上，被给定的几何形状所约束，例如：矩形光源（在这种情况下，它们也称为卡片光源），圆盘光源或者更加普遍的多边形光源。这些平面光源可以用于发出光线的面板（例如广告牌和电视屏幕），可以代替常用的摄影光源（例如柔光箱和反光板），可以用于模拟复杂照明装置的光圈，又或者是用于表示场景中墙壁和其他较大表面所反射出的光线。

最早的卡片光源（和圆盘光源）的实用近似，是由 Drobot [380]推导出来的，它也是一个代表性点的解决方案，但是它特别值得注意，因为将这种方法扩展到平面上的二维区域十分复杂，而且对于求解的整体方法来说也是如此。Drobot 从中值定理出发，他认为一个良好的光源计算候选点，应该位于光照积分的全局最大值附近。

对于一个 Lambert BRDF 而言，这个积分是：

$$L_l\int_{\mathbf{l}\in\omega_l}(\mathbf{n}\cdot\mathbf{l})^+\frac{1}{r_1^2}d\mathbf{l} \tag{10.13}$$

其中 L_l 是光源发出的恒定 radiance； ω_l 是光源几何形状所对应的立体角； r_1 是沿光照方向 $\mathbf{l}$，从着色点到光源平面的射线长度； $(\mathbf{n}\cdot\mathbf{l})^+$ 是 Lambertian 点积的限制结果。从着色点出发，沿法线方向发射一条射线，将这条射线与光源平面的交点记为 $\mathbf{p}'$ ；光源边界上距离点 $\mathbf{p}'$ 最近的点记作 $\mathbf{p}_c$ ，将点 $\mathbf{p}_c$ 与着色点相连接，在这个方向上可以获得 $(\mathbf{n}\cdot\mathbf{l})^+$ 的最大值。类似的，我们将光源平面上距离着色点最近的点记

为 $\mathbf{p}''$，点 $\mathbf{p}_r$ 是光源边界上距离点 $\mathbf{p}''$ 最近的点，在点 $\mathbf{p}''$ 处，我们可以获得 $1/r_1^2$ 的最大值，如图 10.13 所示。这个被积函数的全局最大值位于点 $\mathbf{p}_r$ 和点 $\mathbf{p}_c$ 之间的某处，即：$\mathbf{p}_{\max} = t_m \mathbf{p}_c + (1 - t_m)\,\mathbf{p}_r$，其中 $t_m \in [0, 1]$。Drobot 使用数值积分方法来找到不同配置下的最佳代表性点，然后再找到一个平均表现最好的参数 t_m。

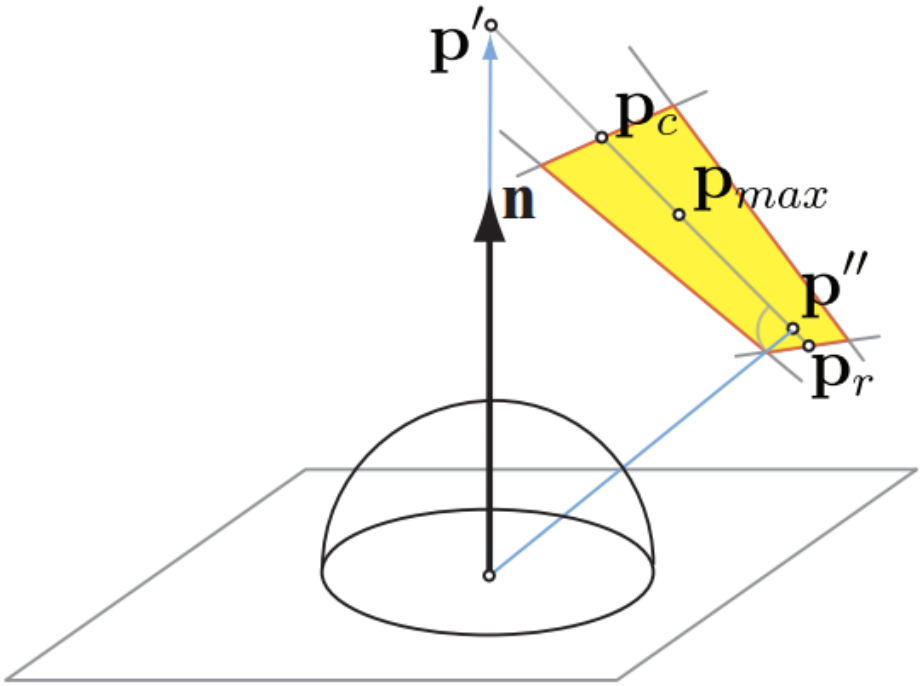


图 10.13：Drobot 的矩形面光源。上图展示了代表性点近似的几何构造过程。

在 Drobot 的最终解决方案中，对漫反射光照和镜面光照的进行了进一步的近似，所有这些都是通过与数值上的 ground-truth 解决方案进行对比来实现的。他还为纹理卡片光源（textured card light）推导出了一种算法，这种光源在矩形区域上的光线发射是不均匀的，具体的发光强度由一个纹理进行控制。这个过程是使用一个三维查找表来实现的，这个表中包含了自发光纹理在不同半径的圆形区域上的预积分。Mittring [1228]采用了类似的方法来进行光泽反射，它将反射光线与一个纹理化的矩形广告牌相交，并根据光线相交的距离，来检索预先计算的、模糊版本的纹理。这项工作要早于 Drobot 所提出的方法，但这个方法是一种更加经验主义的，不那么有原则性的方法，它确实试图与 ground-truth 的积分解决方案相匹配。

对于更加一般的平面多边形面光源（polygonal area light），Lambert [967]最早给出了完美漫反射表面的一个精确封闭形式的解。Arvo 对该方法进行了改进，将光泽材质建模为 Phong 镜面波瓣。Arvo[74]通过将向量 irradiance 的概念扩展到高维 irradiance 张量，并利用 Stoke 定理，将面积积分转换为沿积分域轮廓的简单积分，从而实现了这一点。Arvo 的方法仅有一个假设：光源对于表面着色点是完全可见的

（这是一个常见的假设，可以对与表面相切的光源多边形进行裁剪，来绕过这个假设），并且 BRDF 是一个径向对称的余弦波瓣。不幸的是，在实践中，Arvo 的解析方法对于实时渲染而言十分昂贵，因为它需要计算一个方程，其时间复杂度与 Phong 波瓣中所使用的指数线性相关，同时还与每个面光源多边形的边缘数量有关。最近，Lecocq [1004]找到了轮廓积分函数的一个 $O(1)$ 近似，并将这个解扩展到了一般的、基于半向量的 BRDF 中，这使得该方法更加实用。

到目前为止，我们所描述的所有实用的实时面光源光照方法，都采用了某些简化的假设，从而允许对解析结构的推导，以及对结果积分的近似处理。Heitz 等人[711]采用了一种不同的线性变换余弦（linearly transformed cosine，LTC）方法，据此产生了一种实用、准确且通用的技术。在他们的方法中，首先会在球体上设计一组具有高度表现力的函数（即这些函数可以具有多种不同的形状），这些函数可以很容易地在任意球面多边形上进行积分，如图 10.14 所示。

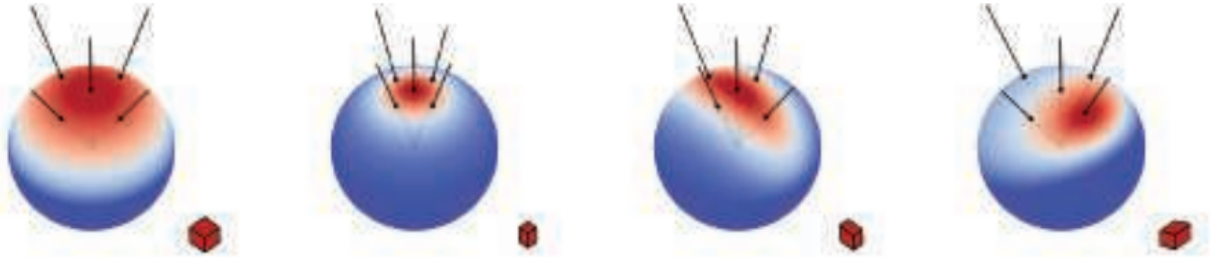


图 10.14：线性变换余弦技术背后的关键思想是：一个简单的余弦波瓣（最左侧图像）可以很容易地进行缩放、拉伸，并通过使用一个 3×3 的变换矩阵进行倾斜。这使得余弦波瓣在球面上可以有很多形状。

LTC 只使用了一个由 3×3 矩阵变换的余弦波瓣，因此它们可以在半球上调整大小、拉伸、旋转，从而适应各种形状。一个简单的余弦波瓣（与 Blinn–Phong 不同，它不包含进行指数运算）与球面多边形的积分已经很成熟了，它可以追溯到 Lambert 的工作[74, 967]。Heitz 等人观察到，在波瓣上使用变换矩阵来对积分进行扩展，并不会改变它的复杂性，我们可以通过逆矩阵来对多边形定义域进行变换，并消去位于积分内部的矩阵，最终的被积函数仅仅是一个简单的余弦波瓣，如图 10.15 所示。

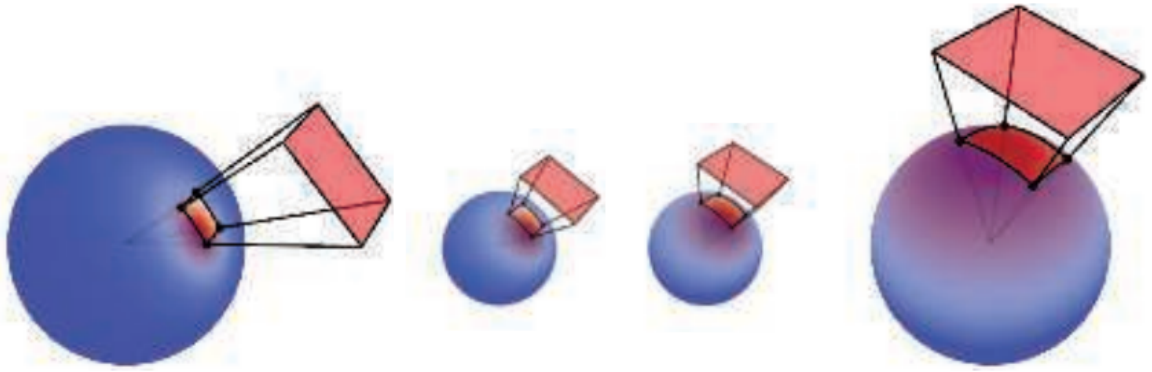


图 10.15：给定一个 LTC 和一个球面多边形定义域（最左侧图像），我们可以通过 LTC 矩阵的逆来对其进行变换，从而获得一个简单的余弦波瓣和一个新的定义域（最右侧图像）。新的余弦波瓣在变换后的定义域上的积分，就等于 LTC 在原定义域上的积分。

对于一般的 BRDF 和面光源形状，剩下的唯一工作，就是找到将球体上的 BRDF 函数表示为一个或者多个 LTC 的方法（或者近似值），这些工作可以离线完成并构建一个查找数组，通过 BRDF 参数（粗糙度、入射角等）来进行索引。基于余弦的线性变换方法既适用于一般的纹理多边形面光源，也适用于专门的、更易于计算的光源形状，例如卡片光源、圆盘光源和管状光源。LTC 方法可能要比代表性点方法更加昂贵，但是相应地也更加准确。

10.2 环境光照

原则上，反射（方程 9.3）并不会区分是从光源发出的直接光，还是从天空或者场景其他物体散射过来的间接光，在着色点半球内的所有入射方向上都存在 radiance，反射方程会对所有方向进行积分。然而在实践中，通常我们会认为直接光具有较高的 radiance 和相对较小的立体角，而间接光往往会以中等或者较低的 radiance，来覆盖半球方向上的其余部分。基于这种划分方式和理由，我们可以将二者分开进行处理。

到目前为止，我们讨论了面光源的技术，它对从光源形状发出的恒定 radiance 进行了积分。这样做为每个表面着色点都创建了一组方向，在这些方向上都具有恒定的非零入射 radiance。现在我们将要研究的是，在所有可能的入射方向上，对由不同函数定义的 radiance 进行积分的方法，如图 10.16 所示。

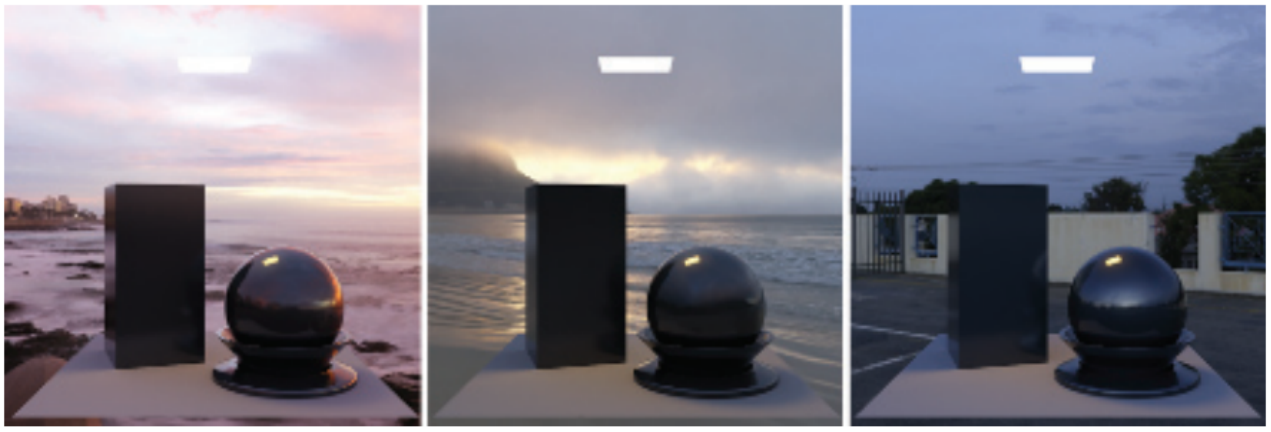


图 10.16：同一个场景在不同的环境光照下的渲染结果。

虽然我们将在这里对有关间接光照和“环境”光照的内容进行讨论，但是我们所讨论的并不是全局光照算法。这里最关键的区别在于，在本小节所讨论的方法中，所有的着色数学表达式都不依赖于场景中其他表面的知识，而是仅仅依赖于一组少量的光源图元。因此，虽然我们可以使用一个面光源来模拟光线在墙壁上的反射效果，而且它确实也是一个全局效果，但是其中的着色算法并不需要知道这个墙壁的存在；它所拥有的全部信息都来自于这个光源，并且所有的着色计算都是在局部完成的。全局照明技术（第 11 章）通常会与本章中的概念紧密相关，因为全局光照的许多解决方案，都可以被视为计算正确的局部光源图元集合的方法，并将这些局部光源作用于每个物体或者每个表面位置上，从而模拟光线在场景中相互反弹的作用效果。

环境光（ambient light）是最简单的环境光照模型，环境光的 radiance 不会随着方向发生变化，具有恒定的值 L_A 。但是即使是这样一个最基本的环境光照模型，也可以显著提升视觉质量。一个不考虑光线在物体之间反弹的场景会显得很不真实，在这样的场景中，处于阴影中的物体或者背对光源的物体将会是纯黑的，这与现实中我们所看到场景都不同。图 10.1 中所展示的月球表面很接近我们刚才描述的场景，但是即使在这样的场景中，也会有一些间接光线被附近的物体所反射。

环境光的确切影响将取决于 BRDF。对于 Lambertian 表面而言，无论表面法线 $\mathbf{n}$ 或者观察方向 $\mathbf{v}$ 的具体情况如何，环境光固定的 radiance L_A 对于出射 radiance 的贡献都是恒定的：

$$L_o(\mathbf{v}) = \frac{\rho_{ss}}{\pi} L_A \int_{\mathbf{l} \in \Omega} (\mathbf{n} \cdot \mathbf{l}) d\mathbf{l} = \rho_{ss} L_A \tag{10.14}$$

在进行着色计算的时候，这种恒定的出射 radiance 贡献会被添加到直接光源的贡献中。对于任意的 BRDF，这个等价方程为：

$$L_o(\mathbf{v}) = L_A \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v})(\mathbf{n} \cdot \mathbf{l}) d\mathbf{l} \tag{10.15}$$

方程 10.15 中的积分部分与定向反照率 $R(\mathbf{v})$ 相同（章节 9.3 中的方程 9.9），因此这个方程等价于 $L_o(\mathbf{v}) = L_A R(\mathbf{v})$ 。在一些较老的实时渲染应用中，有时会假设 $R(\mathbf{v})$ 是一个恒定的值，它被称为环境颜色 $\mathbf{c}_{amb}$ ，因此可以进一步将方程简化为 $L_o(\mathbf{v}) = \mathbf{c}_{\text{amb}} L_A$ 。

这里的反射方程忽略了遮挡情况，也就是说，在实际情况中，表面着色点的某些入射方向，会被其他物体或者同一物体的其他部分所遮挡。这种简化通常会降低画面的真实感，对于环境光照来说尤其明显；在忽略遮挡情况的时候，环境光照会显得非常均匀平坦。我们将在章节 11.3 中（环境光遮蔽）详细讨论解决该问题的方法，尤其是在章节 11.3.4 中。

10.3 球面函数和半球函数

上一小节中我们所讨论的环境光照还只是一个常数，为了将环境光照扩展到常数项之外，我们需要一种数学方法，来表示从任何方向照射到物体上的入射 radiance。首先，我们会将 radiance 考虑为一个仅对方向进行积分的函数，而不是针对表面位置进行积分。之所以这样做，是因为我们假设光照环境是无限远的。

到达给定着色点上的 radiance，在每个入射方向上可能都是不同的。从左边入射的光线可能是红色的，从右边入射的光线可能是绿色的；从顶部入射的光线会被遮挡，而从侧面入射的光线则不会被遮挡。这种类型的量可以使用球面函数（spherical function）来进行表示，这些函数定义在单位球体的表面上，或者定义在 $\mathbb{R}^3$ 的方向空间中，我们将这个定义域记为 S 。最终生成单个值还是多个值并不会对这些函数的运行产生影响，例如：通过为每个颜色通道存储单独的标量函数，这些相同的标量函数同样也可以用于对颜色值进行编码。

假设现在我们有一个 Lambertian 表面，球面函数可以通过存储预先计算的 irradiance 函数来计算环境光照，例如：对于每个可能的表面法线方向，计算 radiance 与余弦波瓣的卷积。更加先进（sophisticated）的方式是存储 radiance，并在运行过程中，计算每个表面着色点的 BRDF 积分。球面函数也广泛应用于全局光照算法中（第 11 章）。

与球面函数相关的是半球函数（hemisphere function），即只有一半方向上的值是有定义的，这些函数用于描述那些没有光线会从下方照射的表面时的入射 radiance。

我们将这些表示函数称为球面基底（spherical base），因为它们都是定义在球面函数向量空间中的基底。虽然环境/高光/方向形式（AHD，章节 10.3.3）在技术上来说，并不是数学意义上的基底，但是我们也将使用基底这个术语来指代它们。将一个函数转换为给定表示形式的操作被称为投影（projection），从给定表示形式中计算函数值的过程被称为重建（reconstruction）。

每种基底表示方法都有自己的一套权衡方式，我们将会在给定的基底中寻找如下属性，这些属性是我们想要基底能够具备的：

- 高效的编码（投影）和解码（查找）。
- 使用较少的系数和较低的重建误差来表示任意球面函数的能力。
- 投影的旋转不变性（rotational invariance），也就是将一个函数的投影结果进行旋转，与旋转这个函数，然后再进行投影所得到的结果是一样的。这个旋转不变性意味着使用近似函数（球谐函数）在旋转的时候，并不会改变所得到的结果。
- 易于计算编码函数的和与编码函数的乘积。
- 易于计算球面积分和球面卷积。

10.3.1 简单表格形式

想要表示球面函数或者半球函数，最直接方法就是直接选择几个方向，并为每个方向都存储一个值。在使用的时候，在所求方向周围找到一定数量的样本，并用某种形式的插值来对函数值进行重建。

虽然这种表示方式很简单，但是它的表征能力很强。将这些球面函数相加或者相乘十分简单，就像直接将它们所对应的表项相加或者相乘一样。我们可以对许多不同的球面函数进行编码，并使用更多的样本来降低重建误差。

想要以一种允许高效检索，同时又能相对平均地表示所有方向的方式，来将样本分布在一个球体上（如图 10.17 所示）并不是一件容易的事情。最常用的方法是，首先将球面展开为一个矩形区域（类似于世界地图），然后在该矩形区域内，使用点状网格来对其进行采样。由于一个二维纹理刚好可以和矩形区域内的网格相对应，因此我们可以使用纹素来作为样本值的底层存储方式。这样做可以让我们利用 GPU 加速的双线性纹理过滤，来进行快速查找（重建）。在章节 10.5 中我们将讨论有关环境贴图的内容，它就是这种形式的球面函数，在该小节中我们还会讨论将展开球面的不同方式。

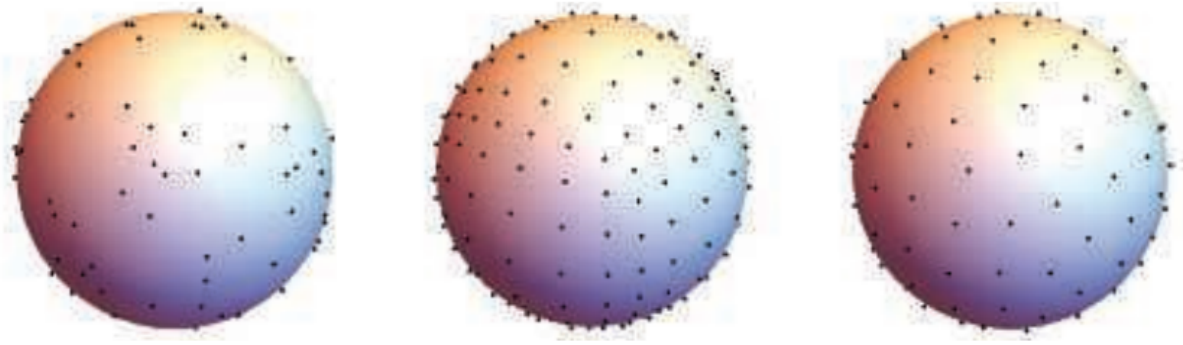


图 10.17：在球面上分布采样点的几种不同方法。从左到右分别是：随机分布，立方体网格点，球面 t 型设计。

这种简单表格形式当然也存在缺点。如果我们使用较低分辨率的贴图来存储这个表格，那么硬件过滤所提供的质量通常是不可接受的。卷积计算是处理光照时的常见操作，其计算复杂度与样本数量成正比，这可能会导致非常高的计算开销。此外，使用表格形式存储球面函数不具备投影不变性，这在某些应用中可能会出现一些问题，例如：想象一下，当光线从一组方向照射到物体表面上，此时我们已经有了对其 radiance 的编码；但是如果此时物体发生了旋转的话，那么编码结果可能会以不同的方式进行重建，这可能会导致编码的辐射能量发生变化，从而导致在场景动画的过程中，可能会出现脉冲瑕疵（pulsating artifact）。通过在投影和重建过程中，仔细构造与每个样本相关的核函数，可以缓解这些问题。不过更加常见的情况是，仅仅增加样本数量就足以掩盖这些问题。

通常来说，当我们需要存储复杂的高频函数，并且这些函数需要对许多数据点进行编码，从而保持较低误差的时候，就会使用表格形式。如果我们需要以一种更加紧凑的形式来对球面函数进行编码，并且这些函数只有少数几个参数的时候，我们可以选择使用更加复杂的基底。

作为一种常见的基底选择，环境立方体（ambient cube，AC）是最简单的表格形式之一，它由六个沿着主轴方向的平方余弦波瓣构成[1193]。之所以它会被称为环境“立方体”，是因为这种方法相当于在立方体的六个表面上存储数据，并在我们从一个方向移动到另一个方向的时候进行插值。对于任何给定的方向，只有三个波瓣是与之相关的，因此我们并不需要从内存中获取另外三个波瓣的参数[766]。在数学上，环境立方体可以被定义为：

$$F_{AC}(\mathbf{d}) = \mathbf{d}d \cdot \mathrm{sel}_{+}\left(\mathbf{c}_{+}, \mathbf{c}_{-}, \mathbf{d}\right) \tag{10.16}$$

其中 $\mathbf{c}_+$ 和 $\mathbf{c}_-$ 包含了立方体六个表面上的值； $\mathrm{sel}_+\left(\mathbf{c}_+, \mathbf{c}_-, \mathbf{d}\right)$ 是一个向量函数，它将一个方向向量作为输入，并会输出一个向量；对于输入参数 $\mathbf{d}$ 中的每个分量，会根据其正负性来选择取 $\mathbf{c}_+$ 还是 $\mathbf{c}_-$ 。

环境立方体和立方体贴图（详见章节 10.4）很类似，二者的不同之处在于，环境立方体的每个表面上只会存储一个纹素。在某些渲染系统中，针对这种特殊情况，在软件中执行重建过程可能要比在立方体贴图上使用 GPU 的双线性过滤更快。Sloan [1656]推导了一个简单的方程，可以在环境立方体和球谐函数基底之间进行转换（详见章节 10.3.2）。

使用环境立方体进行重建的质量相当低。通过对 8 个值（而不是 6 个）进行存储和插值，可以获得稍微好一点的结果，这 8 个值对应了立方体上的 8 个顶点。最近，Iwanicki 和 Sloan [808]提出了另一种被称为环境骰子（ambient dice ，AD）的方法，它的基底由沿二十面体顶点方向的平方和四次余弦波瓣组成。在重建的时候，需要使用 12 个存储值中的 6 个，确定检索哪 6 个值的逻辑要比环境立方体的稍微复杂一些，但是这种方法的质量要高得多。

10.3.2 球面基底

有无数种方法可以将函数投影（编码）到使用固定数量值（系数）的表示方法上。我们所需要的是一个跨球面的数学表达式，它具有一些可以改变的参数。然后，我们可以通过拟合，来对任意给定的函数进行近似，即找到一组参数，使得表达式与给定函数之间的误差最小。

最简单的选择是使用 一个常量：

$$F_c(\theta, \phi) = c \cdot 1.$$

我们可以通过将给定函数 f 在球面上进行求平均，从而将其投影到这个基底中，即 $c = \frac{1}{4\pi} \int_\Omega f(\theta, \phi)$ 。一个周期函数的平均值 c 也被称为 DC 分量（直流分量）。这种基底的构建十分简单，甚至也符合我们正在寻找的一些性质（易于重构、加法、乘积、旋转不变性）。然而对于大多数球面函数而言，这种方法的表达能力并不好，因为这种方法只是使用了函数的平均值来代替这些函数。我们可以使用两个系数 a 和 b 来构造一个稍微复杂的近似：

$$F_{\text{hemi}}\left(\theta, \phi\right) = a + \frac{\cos(\theta) + 1}{2}(b - a),$$

这种表示方法可以对位于球体两极的值进行精确编码，并可以在球体表面上对其进行插值。这个表示方法的表现能力更强，但是投影过程变得更加复杂了，并且不是所有旋转都具有旋转不变性。事实上，这个基底可以看作是一个包含两个样本的表格形式，这两个样本分别位于球体的两个极点上。

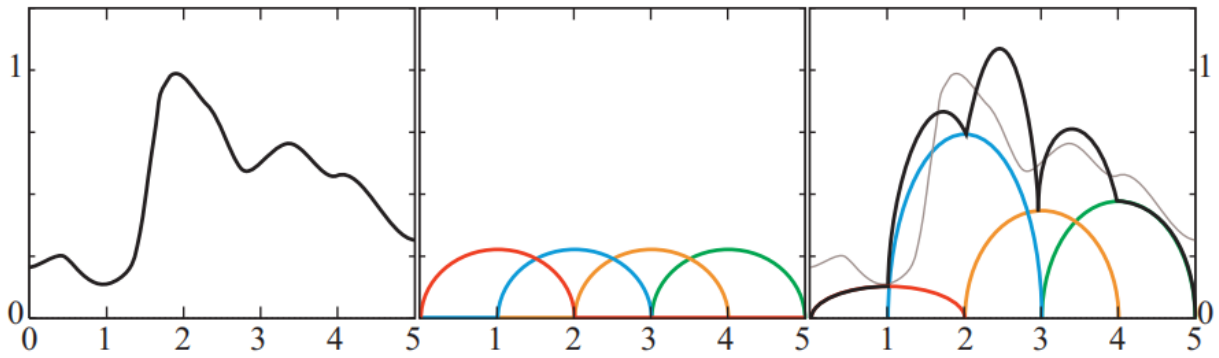


图 10.18：基函数的一个例子。在这个例子中，输入值为 0–5 之间的一个数，函数会返回 0–1 之间一个值，左侧的图展示了这样一个函数。中间的图展示了一组基函数（每种颜色都代表了一个不同的基函数）。右图则展示了使用基函数来对目标函数的近似，通过将每个基函数乘以一个权重并将它们相加来完成这个近似。右图中的每个基函数都按照各自的权重进行了缩放，图中的黑色线条代表了基函数求和之后的结果，这是对原始函数的近似结果；图中灰色线条代表了原始函数，用于和近似函数进行比较。

一般来说，当我们讨论函数空间中的一组基底时，我们的意思是：存在这样的一组函数，它们的线性组合（加权和求和操作）可以用来表示给定域中的其他函数。图 10.18 展示了这个概念的一个例子。本小节的剩余部分，将讨论一些可以用于近似球面函数的基底选择。

球面径向基函数

使用 GPU 硬件过滤的表格方法，其重建质量较低，这种较低的重建质量，在一定程度上是由插值样本时所使用的双线性函数造成的。可以使用其他函数来对样本进行加权重建，这样的函数可以产生比双线性滤波更高质量的结果，而且它们可能还会具有其他的一些优点。一类经常用于此目的的函数就是球面径向基函数（spherical radial basis function，SRBF），这些函数都是径向对称的（沿轴旋转对称），因此这些函数都只有一个输入参数，即函数所指方向与计算方向之间的夹角。基底就是由一组这样的函数所组成的，每个函数都被称为一个波瓣（lobe），它们分布在整个球面上。每个函数都由波瓣的一组参数进行表示，这个参数集合可以包含它们的方向，但是这样会使得投影过程变得更加困难（需要非线性的全局优化）。因此，我们通常会假定波瓣的方向是固定的，它们均匀地分布在整个球体上；并使用一些其他的参

数，例如每个波瓣的大小或者每个波霸所覆盖的角度（分布）。通过在给定方向上，对所有波瓣进行计算，并将结果进行求和，从而完成重建过程。

球面高斯函数

球面高斯分布（spherical Gaussian，SG）是一种十分常见的 SRBF 波瓣，在方向统计（directional statistic）中也被称为 von–Mises–Fisher 分布。需要注意的是，von–Mises–Fisher 分布通常都会包含一个归一化常数，我们在方程中会避免使用这个常数。单个波瓣可以定义为：

$$G(\mathbf{v}, \mathbf{d}, \lambda) = e^{\lambda(\mathbf{v} \cdot \mathbf{d} - 1)} \tag{10.17}$$

其中 $\mathbf{v}$ 是需要计算的方向，它是一个单位向量； $\mathbf{d}$ 是波瓣的方向轴，即分布的平均方向，同样也是一个单位向量；其中的 $\lambda \geq 0$ ，它代表了波瓣的尖锐程度，即控制了波瓣的角宽度（angular width），也被称为集中参数（concentration parameter）或者扩散参数（spread）[1838]。

为了构造球面基底，我们会使用给定数量的球面高斯函数的线性组合：

$$F_G(\mathbf{v}) = \sum_k w_k G\left(\mathbf{v}, \mathbf{d}_k, \lambda_k\right) \tag{10.18}$$

将一个球面函数投影（编码）到这种表示方法中，需要找到一组参数集合 $\{w_k, \mathbf{d}_k \text{，} \lambda_k\}$，来使得重建误差最小化。这个过程通常会通过数值优化的方法来完成，通常会使用一个非线性的最小二乘优化算法（例如 Levenberg–Marquardt 法）。需要注意的是，如果我们允许在优化过程中改变整个参数集合，那么我们就不会使用函数的线性组合来构造基底，此时方程 10.18 并不代表一组基底。只有当我们选择一组固定的波瓣（固定的方向和固定的扩散角度）时，才能获得一个适当的基底，从而很好地覆盖整个定义域[1127]，并且只需要对权重 w_k 进行拟合就可以完成投影过程。这样做也大大简化了优化问题，因为现在它可以使用普通的最小二乘法来进行优化。如果我们需要在不同的数据集（投影函数）之间进行插值，这也是一个很好的解决方案，对于这种情况而言，允许波瓣方向和波瓣锐度能够发生变化的坏处是很大的，因为这些参数是高度非线性的，很难进行拟合。

这种表示方法的一个优点是，在 SG 上进行的许多操作都有简单的解析形式，两个球面高斯函数的相乘，会产生另一个球面高斯函数[1838]：

$$G_1 G_2 = G\left(\mathbf{v}, \frac{\mathbf{d}'}{\|\mathbf{d}'\|}, \lambda'\right),$$

其中：

$$\mathbf{d}' = \frac{\lambda_1 \mathbf{d}_1 + \lambda_2 \mathbf{d}_2}{\lambda_1 + \lambda_2}, \quad \lambda' = (\lambda_1 + \lambda_2)\,\|\mathbf{d}'\|\,.$$

球面高斯函数在球面上的积分，也可以使用解析方法来进行计算：

$$\int_{\Omega} G(\mathbf{v}) d\mathbf{v} = 2\pi \frac{1 - e^{2\lambda}}{\lambda}$$

这意味着对两个球面高斯函数的乘积进行积分，也有一个很简单的方程。

如果我们能够将光线的 radiance 表示为一个球面高斯函数，那么可以将其与一个 BRDF（以相同的形式进行编码）相乘，然后再对乘积进行积分，从而进行光照计算[1408, 1838]。由于这些原因，SG 在许多研究项目[582, 1838]和工业应用中得到了应用[582, 1838]。（译者注：3D Gaussian Splatting）

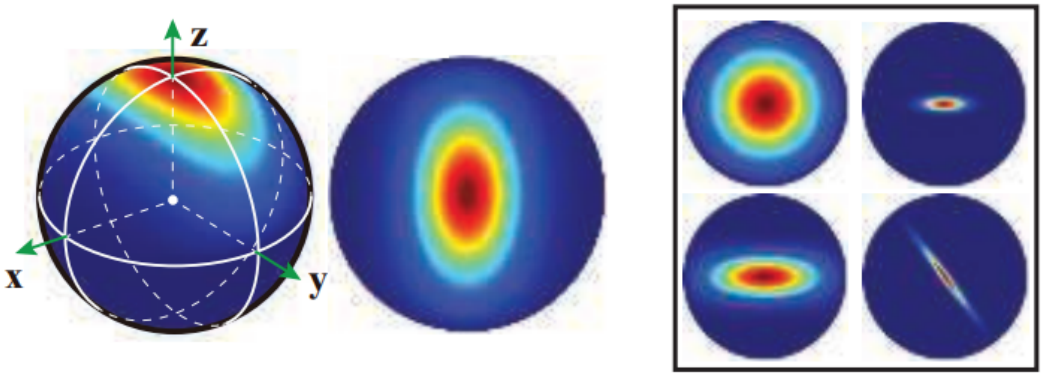


图 10.19：各向异性球面高斯。左图展示了一个球体上的 ASG，以及对应的俯视图。右图展示了 ASG 的其他四个例子，用于显示公式的表达能力。

对于平面上的高斯分布而言，可以将 von Mises–Fisher 分布进行推广，从而允许各向异性。Xu 等人[1940]引入了各向异性的球面高斯函数（anisotropic spherical Gaussians，ASG），如图 10.19 所示，它是通过在单个方向 $\mathbf{d}$ 上，增加两个补充轴 $\mathbf{t}$ 和 $\mathbf{b}$ 来实现的，它们共同形成一个正交的切线坐标系：

$$G(\mathbf{v}, [\mathbf{d}, \mathbf{t}, \mathbf{b}], [\lambda, \mu]) = S(\mathbf{v}, \mathbf{d}) e^{-\lambda(\mathbf{v}\cdot\mathbf{t})^2 - \mu(\mathbf{v}\cdot\mathbf{b})^2} \tag{10.19}$$

其中的 $\mu, \lambda \geq 0$，它们控制了波瓣沿切线坐标系（$\mathbf{t}$ 和 $\mathbf{b}$）的扩散程度；其中 $S(\mathbf{v}, \mathbf{d}) = (\mathbf{v} \cdot \mathbf{d})^+$ 是一个平滑项，这个平滑项是方向统计中的 Fisher–Bingham 分布与计算机图形学中所使用的 ASG 的主要区别。Xu 等人还提供了积分算子、乘积算子和卷积算子的解析近似。

虽然 SG 具有许多令人满意的特性，但是它们具有这样的一个缺点，与表格形式和具有有限范围（带宽）的一般内核不同，SG 具有全局支持（global support）的特性，即每个波瓣都会对球面上的任意一个方向产生影响。即使每个波瓣的衰减速度都很快，但是球体上的每个波瓣都是非零的。这个全局范围意味着，如果我们使用 N 个波瓣来表示一个函数，那么对于任意一个需要重建的计算方向，我们都需要对所有的 N 个波瓣进行计算，才能获得最终的重建结果。

球谐函数

> 我们在这里所讨论的基函数，更恰当的名称应当是“实数球谐函数”，因为它们表示了复数球谐函数的实部。

球谐函数（spherical harmonic，SH）是球面上的一组正交的基函数。一组基函数的正交集（orthogonal set）具有如下特点：任意两个不同基函数之间的内积（inner product）为零。内积是一个类似于点积的概念。两个向量之间的内积就是它们的点积，即各个分量对应相乘再相加的结果。我们可以类似地推导出两个函数的内积的定义，即将这两个函数相乘再进行积分，其数学表达如下：

$$\langle f_i(x), f_j(x) \rangle \equiv \int f_i(x) f_j(x) dx \tag{10.20}$$

上述方程是在相关定义域上进行积分的，即 x 轴上。对于图 10.18 所示的函数，其相关的定义域位于 x 轴上的 0–5 之间（请注意，图 10.18 所展示的基函数并不是正交的）。对于球面函数而言，定义域有所不同，但是基本的形式和概念都是相同的：

$$\langle f_i(\mathbf{n}), f_j(\mathbf{n}) \rangle \equiv \int_{\mathbf{n} \in \Theta} f_i(\mathbf{n}) f_j(\mathbf{n}) d\mathbf{n} \tag{10.21}$$

其中 $\mathbf{n} \in \Theta$ 代表了在单位球面方向上进行积分。

标准正交集（orthonormal set）也是一个正交集，其附加条件是该集合中的任意一个函数与自身的内积都为 1。更加正式的表述方式为，一组函数 $\{f_j()\}$ 是标准正交的条件是：

$$\langle f_i(), f_j() \rangle = \begin{cases} 0, & \text{where } i \neq j, \\ 1, & \text{where } i = j. \end{cases} \tag{10.22}$$

图 10.20 展示了一个类似于图 10.18 的例子，不同之处在于，其中的基函数都是标准正交的。请注意，图 10.20 中的所展示的标准正交基函数都是互不重叠的，这个条件对于非负函数的标准正交集合是十分必要的，因为任何的重叠都意味着内积非零。而对于那些在部分范围内为负的基函数，则可以发生重叠，它们仍然可以形成标准正交集。这种重叠通常会得到更好的近似结果，因为它允许使用更加平滑的基底。而那些互不重叠的基函数，往往会导致近似结果的不连续性。

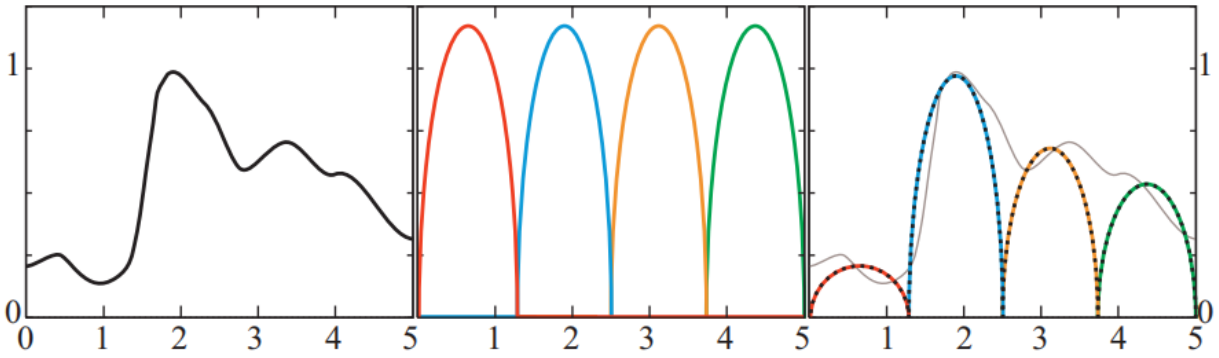


图 10.20：标准正交基函数。这个例子中使用空间和目标函数，与图 10.18 所使用的完全相同，不同之处在于，这个例子中的基函数都是正交的。左图展示了目标函数，中间展示了基函数的标准正交集，右图展示了缩放后的基函数。为了方便对比，最终得到的近似函数使用黑色虚线进行表示，原始函数则使用灰色实线进行表示。

标准正交基的优点在于，想要找到最接近目标函数的近似值十分简单。为了完成投影操作，每个基函数的权重系数都是目标函数 $f_{\text{target}}()$ 与相应基函数的内积：

$$\begin{aligned} k_j &= \langle f_{\text{target}}(), f_j() \rangle, \\ f_{\text{target}}() &\approx \textstyle\sum_{j=1}^{n} k_j f_j(). \end{aligned} \tag{10.23}$$

在实践中，这个积分只能通过数值方法来进行计算，通常会使用蒙特卡罗采样，对平均分布在球面上的 n 个方向进行计算，然后再取平均值。

标准正交基在概念上类似于章节 4.2.4 中所介绍的三维向量的“标准基底（standard basis）”。不同之处在于，标准基的目标并不是函数，而是一个点的位置；标准基由三个向量组成（每个维度一个向量），而不是一组函数。根据方程 10.22 中使用的定义，标准基也是标准正交的。将一个点投影到标准基上的方法也是一样的，每个权重系数都是位置向量和基向量点积的结果。这里有一个十分重要的区别，那就是标准基

可以精确地再现每个点，而一组有限的基函数只能对目标函数进行近似。由于标准基使用了三个基向量来表示三维空间，而函数空间所具有的维度是无限的，因此有限数量的基函数永远无法完美地表示函数空间中的信息。也就说，对于有限数量的基函数，其近似结果永远都不可能是精确的。

球谐函数函数都是正交或者标准正交的，它们还具有其他一些优点。球谐函数是旋转不变的，并且 SH 基函数的计算成本也不高，它们都是单位向量在 $x-$ 、 $y-$ 、 $z-$ 坐标中的简单多项式。然而，与球面高斯函数一样，它们都具有全局支持的特点，因此在重建的过程中，需要对所有的基函数都进行计算。有关基函数的表示方法可以在一些参考文献中找到，包括 Sloan[1656]的一篇演讲。他的演讲十分值得关注，因为他讨论了许多使用球谐函数的实用技巧，包括方程以及一些着色器代码。最近，Sloan 还提出了一些方法，来对高效完成 SH 的重建过程[1657]。

SH 基函数可以按照频带（frequency band）进行排列。第一个基函数是一个常数；接下来的三个基函数是线性函数，它们会在球面上缓慢变化；再接下来的五个基函数是变化稍快的二次函数，如图 10.21 所示。频率较低的函数（即在球面上变化缓慢的函数），例如 irradiance 等，可以用相对较少的 SH 系数来精确地进行表示（我们将在章节 10.6.1 中看到）。

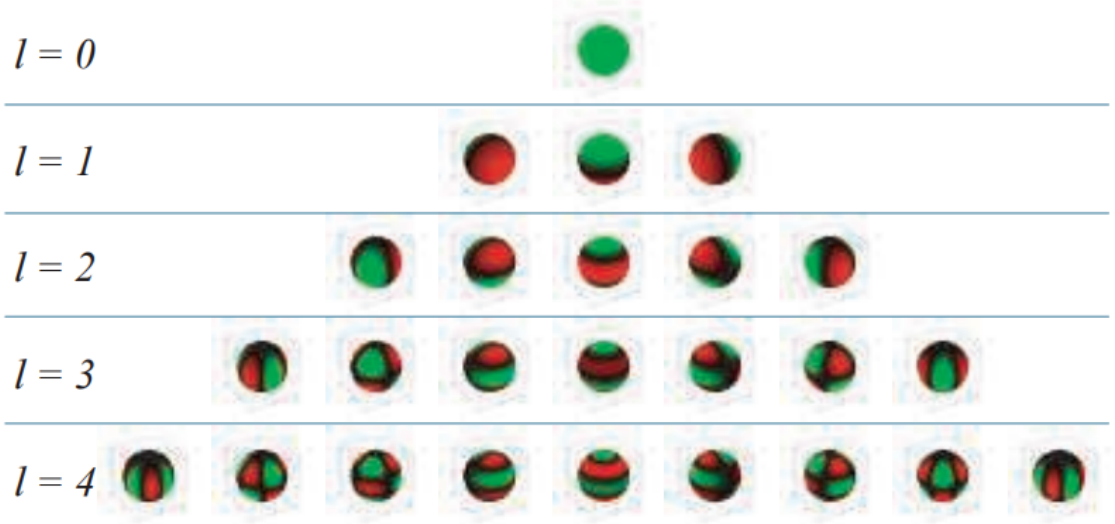


图 10.21：球谐函数的前五个频带。每个球谐函数都有正值区域（绿色）和负值区域（红色），当函数值接近零时，会逐渐变黑。

当投影到球谐函数的时候，所获得的系数代表了投影函数在各个频率上的振幅（amplitude），即其频谱（frequency spectrum），它代表了信号或者波形中各个频率成分中的强度分布。在这个光谱域（spectral domain）中，有这样一个基本的性质：两个函数乘积的积分等于函数投影系数的点积。这个特性能够使我们能够高效地计算光照积分。

球谐函数的许多运算在概念上十分简单，可以将其归结为对系数向量的矩阵变换[1657]。在这些操作中，有几个操作是很重要的，它们分别是：计算投影到球谐函数上的两个函数的乘积、旋转投影函数、计算卷积。在实际操作中，SH 中的矩阵变换意味着，这些操作的复杂度与所使用的系数数量呈二次关系，这可能会是一个很大的复杂度。但幸运的是，这些矩阵通常都具有一些特殊结构，可以利用这些结构设计出效率更高的算法。Kautz 等人[869]提出了一种优化旋转计算的方法，它将旋转分解为关于 x 轴和关于 z 轴的两次旋转。Hable [633]给出了一种快速旋转低阶 SH 投影的常用方法。Green 的综述[583]讨论了如何利用旋转矩阵的块结构，来实现更快的计算。目前，最先进的技术（the state of the art）是由 Nowrouzezahrai 等人[1290]提出的，他将 SH 分解为球带谐波（zonal harmonic）。

光谱变换（例如球谐函数和 H-basis）的一个常见问题是，它们会表现出一种叫做 ringing 的视觉瑕疵（也称为 Gibbs 现象）。如果原始信号中包含了无法使用带限（band-limited）近似来表示的快速变化，那么在重建的时候就会出现振荡瑕疵。在极端情况下，这个重建出来的函数甚至可能会产生负值。当然这个问题是有解决方法的，可以使用各种预过滤方法来解决这个问题[1656, 1659]。

其他球面表示

还有许多其他的表示方法，也可以使用有限数量的系数来对球面函数进行编码。线性变换余弦（章节 10.1.2）就是其中一个例子，它可以有效地对 BRDF 函数进行近似，同时易于在球面的多边形截面上进行积分。

球面小波（spherical wavelet）[1270, 1579, 1841]是一种具有平衡空间局部性（具有紧支撑，compact support）和频率局部性（平滑性）的基底，它可以用于高频函数的压缩表示。球面分段常数基函数（spherical piecewise constant basis function）[1939]将球面划分为常数值的区域；依赖于矩阵分解的双聚类近似（biclustering approximation）[1723]也被用于环境光照。

10.3.3 半球基底

尽管上面所介绍的基底都可以用来表示半球函数，但是它们太过浪费，因为总有一半的信号为零。在这些情况下，我们通常会更加倾向于使用直接在半球域上构造的表示方法，这对那些定义在表面上的函数而言特别相关，常见的例子包括：BRDF、入射 radiance、到达物体某一点的 irradiance 等。这些函数天然局限于以给定着色点为中心，并与表面法线对齐的半球范围内，它们都不会指向物体内部。

AHD 基底

沿着这个思路，一种最简单的表示方法就是将一个常数函数，与半球面上信号最强的单一方向结合起来。它通常被称为环境/高光/方向（ambient/highlight/direction，AHD）基底，它最多的用途就是用来存储 irradiance。AHD 的名字体现了各个分量的含义：A 代表一个恒定的环境光；H 代表一个方向光，它用于近似“高光”方向上的 irradiance；D 代表大部分入射光所集中的方向。AHD 基底通常需要存储 8 个参数，其中两个参数代表了角度，用于表示方向向量；另外六个参数（两个 RGB 颜色）用于环境光强度和方向光强度。它的首次显著应用是在游戏《雷神之锤 3》中，动态物体的体积光照就是以这种方式进行存储的。从那时起，它便被广泛应用于许多游戏中，例如《使命召唤》系列。

想要投影到这种表示方法上有些棘手，因为它是非线性的，对于给定的输入，想要找到其最优的近似参数，具有很高的计算成本。在实践中，通常会使用一些启发式方法。该信号首先会被投射到球谐函数上，并使用最优的线性方向来确定余弦波瓣的方向。有了这个方向，可以使用最小二乘法来计算环境光和高光。Iwanicki 和 Sloan [809]展示了如何在保证非负性的同时，来执行这个投影操作。

辐射法向映射/《半条命 2 》基底

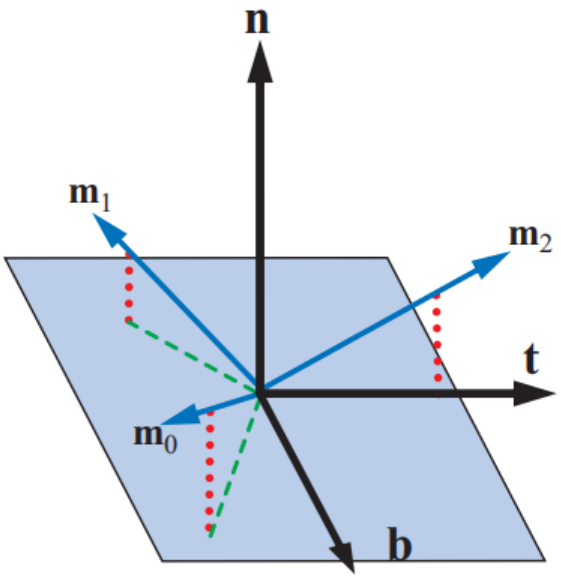


图 10.22：《半条命 2》光照基底。三个基向量在切平面上的仰角约 26°，它们在该平面上的投影会以 120° 的间隔均匀分布在法线周围。这三个基向量都是单位长度，其中每一个都垂直于另外两个。

Valve 在《半条命 2》系列游戏中[1193, 1222]使用了一种十分新颖的表示方式，用于表示定向 irradiance，它被称为辐射法向映射（radiosity normal mapping）。其最初的设计目的是为了存储预计算的漫反射光照，同时允许使用法线映射，它现在通常

被称为《半条命 2》基底。它通过在切线空间中采样三个方向来表示表面上的半球函数，如图 10.22 所示。三个相互垂直的基向量在切线空间中的坐标分别为：

$$\mathbf{m}_0 = \left(\frac{-1}{\sqrt{6}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}}\right), \quad \mathbf{m}_1 = \left(\frac{-1}{\sqrt{6}}, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{3}}\right), \quad \mathbf{m}_2 = \left(\frac{\sqrt{2}}{\sqrt{3}}, 0, \frac{1}{\sqrt{3}}\right). \tag{10.24}$$

在重建的时候，对于一个给定的切线空间方向 $\mathbf{d}$，我们可以沿着基向量的方向，对 E_0，E_1，E_2 进行插值，其数学表达如下：

$$E(\mathbf{n}) = \frac{\sum_{k=0}^{2} \max\left(\mathbf{m}_k \cdot \mathbf{n}, 0\right)^2 E_k}{\sum_{k=0}^{2} \max\left(\mathbf{m}_k \cdot \mathbf{n}, 0\right)^2}. \tag{10.25}$$

> GDC 2004 中给出的表达形式是不正确的，方程 10.25 中的形式来自于 SIGGRAPH 2007 的演示文稿[579]。

Green 指出[579]，如果在切线空间方向 $\mathbf{d}$ 上预先计算以下三个值，则可以显著降低方程 10.25 的计算量：

$$d_k = \frac{\max\left(\mathbf{m}_k \cdot \mathbf{n}, 0\right)^2}{\sum_{k=0}^{2} \max\left(\mathbf{m}_k \cdot \mathbf{n}, 0\right)^2}, \tag{10.26}$$

其中 $k = 0, 1, 2$，因此方程 10.25 可以简化为如下形式：

$$E(\mathbf{n}) = \sum_{k=0}^{2} d_k E_k \tag{10.27}$$

Green 总结了这种表示方法的其他几个优点，其中一些将在章节 11.4 中进行讨论。

《半条命 2》基底可以很好地用于表示定向 irradiance。Sloan [1654]发现这种表示所产生的结果要优于低阶的球谐函数。

半球谐波/H-Basis

Gautron 等人[518]将球谐函数特化到半球域上，他们称之为半球谐波（hemispherical harmonic，HSH）。有很多种方法可以实现这种特化。

例如：Zernike 多项式是类似于球谐函数一样的正交函数，但是它是定义在单位圆盘上的。与 SH 一样，这些函数也可以用于对频域（频谱）函数进行变换，这会产生许多很方便的性质。由于我们可以将一个单位半球转换为一个圆盘，因此我们可以用 Zernike 多项式来表示半球函数 [918]。然而，使用这些数据来进行重建的计算成本很高。Gautron 等人的解决方案具有较低的计算开销，同时还允许对系数向量使用矩阵乘法，来进行相对较快的旋转操作。

然而，HSH 基底的计算开销仍然要比球谐函数更高，因为它是通过将球体的负极移动到半球的外边缘来构建的。这种移位操作使得基函数不再是一个多项式，在计算过程中涉及除法和平方根运算，这在 GPU 硬件上通常会很慢。此外，在半球边缘处的基底总是不变的，因为它在移位之前会被映射到球体上的一个极点。在边缘附近的近似误差可能会很大，尤其是当只使用少数几个系数（球谐函数的频带）时。

Habel [627]引入了 H-Basis，它在经度参数化中使用了部分球谐函数基底，在纬度参数化中使用了部分 HSH 基底。这个基底混合了移位版本的 SH 和非移位版本的 SH，它仍然是正交的，同时计算效率更高。

10.4 环境映射

将一个球面函数记录在一个或者多个图像中的做法，被称为环境映射（environment mapping），因为我们通常会使用纹理映射来实现对表格的查找。这种表现形式是目前最强大且最流行的环境光照形式之一。与其他球面表示方法相比，虽然它占据了更多的内存，但是这种方法的实时解码过程十分简单、速度很快。此外，它可以表达任意高频的球面信号（通过增加纹理分辨率即可），并且能够准确捕捉任何范围内的环境 radiance（通过增加每个通道的 bit 即可）。但是这种准确性是有代价的，与存储在其他常用纹理中的颜色和着色器属性不同，存储在环境贴图中的 radiance 值通常都具有高动态范围（HDR）。每个纹素会使用更多的 bit，这意味着环境贴图往往要比其他纹理占用更多的存储空间，并且其访问速度要更慢。

对于任何的全局球面函数，我们都有这样的一个基本假设：场景中所有物体的入射 radiance L_i 都只依赖于方向这一个参数。这个假设要求物体与被反射的光线位于很远的地方，并且不会发生自反射现象（将自身反射的光线再次反射）。

依赖于环境映射的着色技术，其典型特征并不是它们表现环境光照的能力，而是我们如何将它们与给定的材质整合在一起。也就是说，为了进行积分，我们必须对 BRDF 采用哪种方式的近似和假设？反射映射（reflection mapping）是环境映射中最基本的一种情况，在这种情况下，我们会假设 BRDF 是一个完美的镜面。一个光学平面或

者镜面会将入射光线反射到光线的反射方向 $\mathbf{r}_i$ 上（章节 9.5）。类似地，出射 radiance 只来源于一个方向上的入射 radiance，即反射的观察向量 $\mathbf{r}$，该向量的计算方法与 $\mathbf{r}_i$ 相同（方程 9.15）：

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v} \tag{10.28}$$

这样可以大大简化镜面的反射方程：

$$L_o(\mathbf{v}) = F(\mathbf{n}, \mathbf{r})L_i(\mathbf{r}) \tag{10.29}$$

其中 F 是菲涅尔项（详见章节 9.5）。请注意，与基于半向量的 BRDF 中的菲涅尔项（使用半向量 $\mathbf{h}$ 与 $\mathbf{l}$ 或者 $\mathbf{v}$ 之间的夹角）不同，方程 10.29 中的菲涅尔项使用了表面法线 $\mathbf{n}$ 与反射向量 $\mathbf{r}$ 之间的夹角（与表面法线 $\mathbf{n}$ 与观察向量 $\mathbf{v}$ 之间的夹角相同）。

由于入射 radiance L_i 只与方向有关，因此可以将其存储在一个二维表格中。这种表示方法能够让我们对一个具有任意入射 radiance 分布和一个具有任何形状的镜面进行高效照明，我们通过计算每个着色点所对应的反射向量 $\mathbf{r}$，并在表中查找对应 radiance 来实现这个操作。Blinn 和 Newell [158]引入了这个表格，并将其称为环境贴图（environment map），如图 10.23 所示。

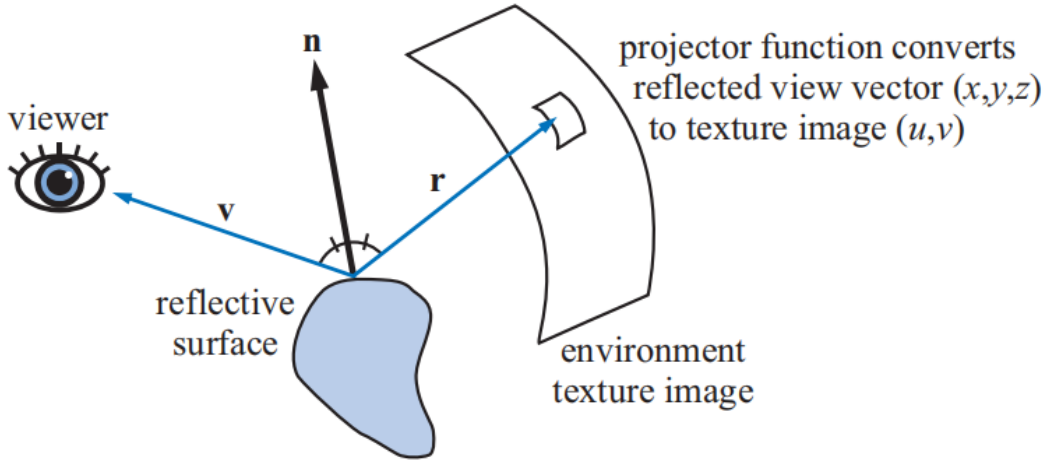


图 10.23：反射映射。当相机看到一个物体的时候，使用表面法线 $\mathbf{n}$ 与观察向量 $\mathbf{v}$ 来计算反射观察向量 $\mathbf{r}$，并使用这个反射观察向量 $\mathbf{r}$ 来访问环境贴图。通过使用一些投影函数，来将反射观察向量 (x, y, z) 转换为纹理坐标，从而检索环境贴图所存储的 radiance。

反射映射算法的步骤如下：

- 生成或者加载表示环境的纹理贴图。

- 对于包含反射物的每个像素，计算物体表面位置上的法线。
- 根据观察向量和表面法线，计算反射的观察向量。
- 使用反射观察向量来计算环境贴图的索引，通过这个索引来获取入射 radiance 值。
- 使用来自环境贴图的纹素数据，来作为方程 10.29 中的入射 radiance。

环境映射有一个潜在障碍值得一提，在使用环境映射的时候，平坦表面通常无法很好地工作。这是因为平面所反射出来的光线差异通常不会超过几度，这些反射方向紧密的聚集在一起，会导致环境贴图中的一小部分被映射到一个相对较大的表面上。使用章节 11.6.1 中所讨论的那些利用 radiance 发射位置信息的技术，可以获得更好的结果。此外，如果我们已知一个表面是完全平坦的（例如地板），我们完全可以使用实时的平面反射技术来进行实现（章节 11.6.2）。

使用纹理数据来对场景进行照明的想法，也称为基于图像的光照（image-based lighting，IBL），这种环境贴图通常是从现实世界中的场景获得的，一般会使用相机来捕获 360 度全景的、高动态范围的图像[332, 1479]。

将环境映射与法线映射组合使用特别有效，可以产生丰富的视觉效果，如图 10.24 所示。这些特性的组合在历史上也十分重要（译者注：可能体现在 PS3 时代游戏画面上的油腻感）。一种受限的凹凸环境映射首次在消费级图形硬件中使用了依赖纹理读取（章节 6.2），这使得这种能力成为像素着色器的一部分。

图 10.24：一个位于相机处的光源，以及凹凸映射与环境映射相结合的渲染效果。从左到右分别是：没有环境映射、没有凹凸映射、相机处没有光源、所有这三个特性组合在一起。

有各种各样的投影函数可以将反射观察向量存储到一个或者多个纹理中。我们将在本小节中讨论一些比较流行的映射方法，并指出每种映射的优缺点。[218]

10.4.1 经纬度映射

1976 年，Blinn 和 Newell [158]实现了第一个环境映射算法。他们所使用的映射方式与地球上所使用的纬度/经度系统相类似，这也就是为什么这种技术通常会被称为纬度–经度映射（latitude–longitude mapping）或者 lat–long 映射。他们的方案并不像是一个从外面进行观察的地球仪，而更像是夜空中的星座地图。就像地球仪上的信息可以被平展到 Mercator 投影或者其他投影地图上一样，空间中一个点周围的环境也可以被映射到纹理上。在计算特定表面位置上的反射观察向量时，这个向量会被转换为球坐标 (ρ, ϕ) 。这里的 ϕ 相当于经度，其弧度值在 $[-\pi, \pi]$ 之间变化； ρ 相当于纬度，其弧度值在 $[0, \pi]$ 之间变化。球坐标 (ρ, ϕ) 可以由方程 10.30 计算而来，其中 $\mathbf{r} = (r_x, r_y, r_z)$ 是归一化的反射观察向量， z 轴正方向指向上方：

$$\rho = \arccos(r_z) \quad and \quad \phi = \operatorname{atan2}(r_y, r_x). \tag{10.30}$$

有关 $\operatorname{atan2}$ 的描述，详见章节 1.2。在获得对应的球坐标之后，使用这些值来访问环境贴图，并检索在反射观察方向上所看到的颜色。请注意，这里所描述的经纬度映射与 Mercator 投影并不相同，经纬度映射会保持纬度线之间的距离恒定，而 Mercator 投影的纬度线距离会在两极处趋于无穷大。

在将一个球体展开为一个平面的时候，总是会发生一些变形，尤其是在不允许多次切割的情况下，而且每种投影方法在保持面积、距离和局部角度之间都有自己的权衡。这种映射方法的一个问题，球面上的信息密度是非常不均匀的。在图 10.25 顶部和底部的极端拉伸中可以看到这种情况，相较于靠近赤道的区域，在靠近两极的区域上会有更多数量的纹素。这种扭曲会带来一定的问题，首先它无法产生最高效的编码，其次在使用硬件进行纹理滤波的时候也会产生瑕疵，尤其是在两极处。滤波核不会随着纹理一起进行拉伸，因此会在具有较高纹素密度的区域过度收缩。还需要注意的是，虽然这个投影操作在数学上很简单，但是实际执行起来的效率可能会很低，因为诸如反余弦（arccosine）这样的超越函数（transcendental function），在 GPU 上的计算成本会很高。这里的超越函数是指：一种在数学中无法用有限次基本代数运算（加、减、乘、除和开方）进行表示的函数。

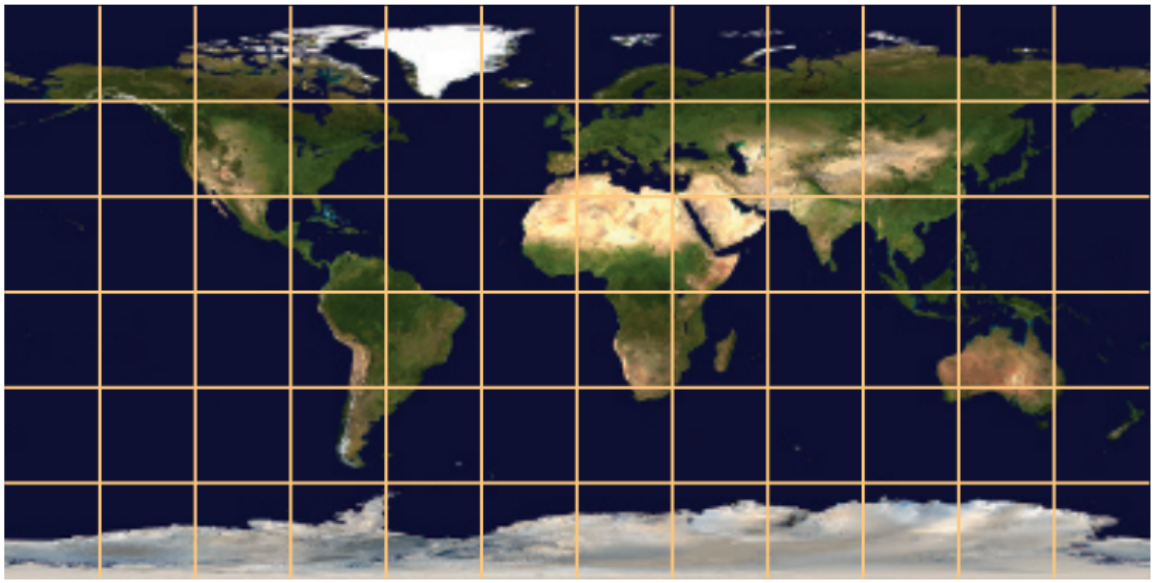


图 10.25：地球上的经纬度线是等距的，这与与传统的 Mercator 投影不同。

10.4.2 球面映射

球面映射（sphere mapping）是第一个在商业图形硬件中所使用的环境映射技术，它最初是由 Williams [1889]提出的，并由 Miller 和 Hoffman [1212]独立开发而来的。这个纹理图像来自于环境的外观，就像是在一个完全反射的球体中，以正交投影的方式来进行观察的那样，所以这个纹理也被称为球面贴图（sphere map）。一种制作真实环境球面贴图的方法是，对着一个闪亮的球体（例如圣诞树装饰品）进行拍摄，如图 10.26 所示。

图 10.26：球面贴图（左）和经纬度映射格式的等效贴图（右）。

由此产生的圆形图像也称为光照探针（light probe），因为它捕获了球体所在位置的光照情况。即使我们在运行过程中使用其他的编码方法，拍摄球形探针也是一种捕捉基于图像照明的有效方法。我们总是能够在球面投影和其他形式之间进行转换，例如

稍后我们所讨论的立方体映射（章节 10.4.3），前提是捕获的图像有足够的分辨率来克服不同方法之间的失真差异。

反射球只会在球的正面显示整个环境。它会将每个反射观察方向映射到球面二维图像上的一个点。假设我们看向另一个方向，即球面贴图上给定的一个点，我们需要计算反射的观察方向。为了实现这一点，我们首先会获取该点上的球面法线，然后再生成反射的观察方向。在检索的时候我们会反转这个过程，通过反射的观察方向来获得球面上的具体位置，因此我们需要推导球面上的表面法线，然后才能得到访问球面贴图所需的 (u, v) 参数。

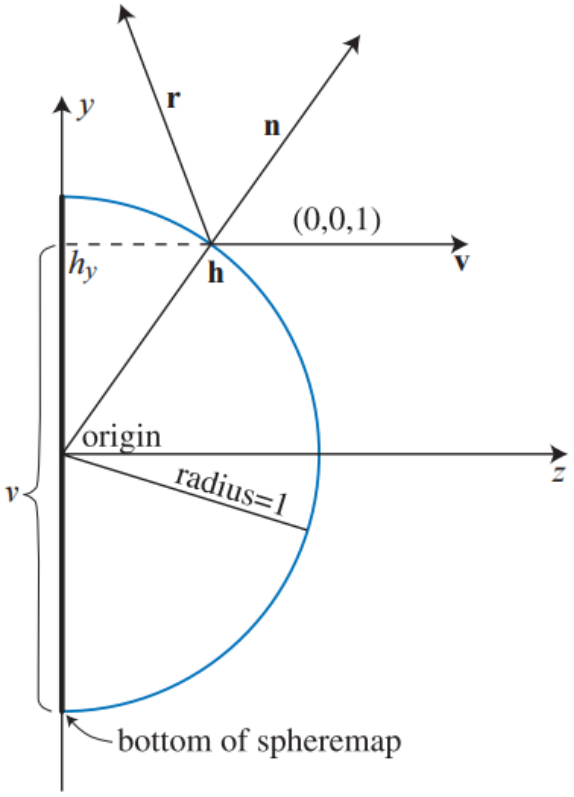


图 10.27：在给定的球面贴图空间中，原始观察向量 $\mathbf{v}$ 是恒定的，球面贴图的法线 $\mathbf{n}$ 介于原始观察向量 $\mathbf{v}$ 与反射观察向量 $\mathbf{r}$ 之间。对于位于原点的单位球体而言，图中的交点 $\mathbf{h}$ 与单位法线 $\mathbf{n}$ 具有相同的坐标。同时，图中还展示了为什么 h_y （从原点开始测量）与球体贴图的纹理坐标 v （不要与观察向量 $\mathbf{v}$ 混淆）有关。

球面上的法线，是反射观察向量 $\mathbf{r}$ 与原始观察向量 $\mathbf{v}$ 之间的半角向量，这里的原始观察向量 $\mathbf{v}$ 在球面贴图空间中为 $(0, 0, 1)$ ，如图 10.27 所示。因此这里的球面法线 $\mathbf{n}$ 是原始观察向量 $\mathbf{v}$ 和反射观察向量 $\mathbf{r}$ 的和，即 $(r_x, r_y, r_z + 1)$ ，将这个向量归一化，就可以得到单位球面法线：

$$\mathbf{n} = \left(\frac{r_x}{m}, \frac{r_y}{m}, \frac{r_z + 1}{m} \right), \quad \text{where} \quad m = \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} \tag{10.31}$$

如果这个球体位于原点处，并且半径为 1，那么单位法线的坐标实际上就是这个法线在球面上的位置 $\mathbf{h}$ 。我们并不需要知道 h_z ，因为 (h_x, h_y) 已经能够描述球面图像上的一个点了，其中每个值都在 $[-1, 1]$ 的范围内。为了能够访问球面贴图，我们需要将坐标映射到 $[0, 1)$ 范围内，因此要将坐标分别除以 2 并再加上 0.5，即：

$$
\begin{aligned}
m &= \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}, \\
u &= \frac{r_x}{2m} + 0.5, \\
v &= \frac{r_y}{2m} + 0.5
\end{aligned}
\tag{10.32}
$$

与经纬度映射相比，球面映射的计算过程要简单得多，并且只会有一个位于图像圆边缘的奇异点。球面映射的缺点是，球面贴图纹理所捕获的环境视图仅对单一的观察方向有效。这个纹理确实捕获了整个环境中的光照信息，因此可以根据新的观察方向，重新计算一个纹理坐标，但是这样做可能会导致一些视觉瑕疵，因为球面贴图的一小部分会因为新的观察方向而被放大，同时边缘周围的奇点也会变得更加明显。在实践中，我们通常假设球面贴图会跟随相机一起运动，并在观察空间中进行操作。

由于球面贴图被定义为固定的观察方向，原则上球面贴图上的每个点不仅定义了反射方向，还定义了表面法线的方向，如图 10.27 所示。对于任意各向同性的 BRDF 而言，可以求解其反射方程，并将结果存储在球面贴图中，这种 BRDF 可以包括漫反射、镜面反射、逆反射和其他项。只要光照和观察方向是固定的，那么球面贴图就是正确的。只要球体的 BRDF 是均匀且各向同性的，那么还可以使用在实际照明下，真实球体的摄影图像来作为球面贴图。

还可以索引两个球面贴图，一个存储反射向量，另一个存储表面法线，从而模拟镜面反射和漫反射环境效果。如果我们根据表面材质的颜色和粗糙度，来调整存储在球面贴图中的值，我们就获得了一种廉价的技术，它可以生成令人信服的材质效果，尽管这个材质效果与观察方向无关。这种方法被雕塑软件 Pixologic ZBrush 推广为“MatCap”着色，如图 10.28 所示。

图 10.28：MatCap 渲染的例子。左侧物体使用了右侧两个球面贴图进行着色处理。其中右上方的贴图使用了观察空间中的法线来进行索引；而右下角的贴图则使用了观察空间中的反射向量来进行索引，并将二者的值相加。最终生成的效果是相当令人信服的，但是移动观察点，图像中的光照效果并不会发生改变。

10.4.3 立方体映射

1986 年，Greene [590]引入了立方体环境贴图（cubic environment map），但是通常会将其称为立方体贴图（cube map），它是目前最流行的方法，其投影操作可以直接在现代 GPU 硬件上实现。通过将环境投影到以相机位置为中心的立方体侧面，来创建一个立方体贴图，然后将立方体表面上的图像用作环境贴图，如图 10.29 和图 10.20 所示。立方体贴图通常会以“交叉”图的形式呈现，即将立方体展开到一个平面上。然而在硬件实现的时候，立方体贴图会被存储为 6 个正方形纹理，而不是将立方体展开后的平面直接存储为一个大的矩形纹理，因此并不会浪费存储空间。

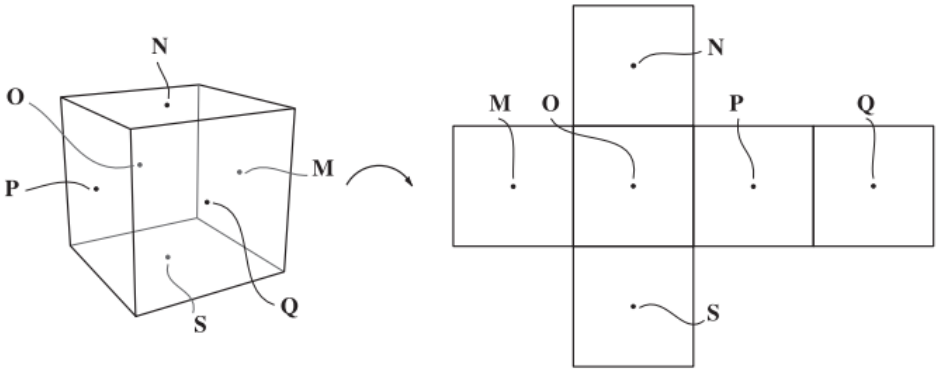


图 10.29：Greene 的环境贴图。图中展示了对应的关键点，左侧的立方体会被展开到右侧的环境贴图中。

图 10.30：与图 10.26 中所使用的环境贴图相同，不同之处在于这里将其转换为了立方体贴图的格式。

可以将相机放置在立方体的中心，然后以 90° 的视场角（FOV）看向每个立方体表面，依次进行渲染从而创建一个立方体贴图，如图 10.31 所示。为了能够从真实世界中生成立方体贴图，通常会将专用相机拍摄而来的（或者拼接而来的）球形全景图，投影到立方体贴图的坐标系中。

图 10.31：《极限竞速 7》中环境贴图的光照，会随着赛车位置的改变而更新。

与球面映射不同，立体体环境映射是与视角无关的。立方体贴图具有比经纬度映射更加均匀的采样特征，经纬度映射往往会在两极处过度采样（相对于赤道地区而言）。

Wan 等人[1835, 1836]提出了一种被称为等立方体（isocube）的映射方法，它比立方体映射具有更低的采样率差异，同时仍然利用立方体映射的纹理硬件来提高性能表现。

访问立方体贴图的方法非常简单，可以直接将输入向量作为一个三分量的纹理坐标，在其所指的方向上获取数据。因此，对于反射而言，我们可以直接将反射方向 $\mathbf{r}$ 传递给 GPU，甚至不需要将其归一化。在一些老旧的 GPU 上，双线性滤波可能会凸显立方体边缘处的缝隙，因为那时的纹理硬件无法在不同的立方体表面之间进行正确地过滤（执行这个操作的开销很大）。为了避免这个问题，研究人员开发了一些技术，例如让视图投影更宽一些，这样单个立方体表面就可以包含这些相邻的纹素了。但是，所有现代的 GPU 都可以正确地执行这种过滤操作，因此这些方法不再是必须的了。

10.4.4 其他投影方法

如今，立方体贴图是最流行的环境光照表格表示方法，原因是其通用性（versatility）、再现高频细节的准确性、以及在 GPU 上的执行速度很快。但是，还有一些其他的投影方法值得一提。

Heidrich 和 Seidel [702, 704]提出使用两个纹理来进行双抛物面环境映射（dual paraboloid environment mapping）。这个想法类似于球面映射，但是它并不是通过记录环境在球面上的反射信息来生成纹理的，而是使用了两个抛物线投影。每个抛物面都会创建一个类似于球面贴图的圆形纹理，每个圆形纹理都会覆盖一个环境半球。

与球面映射一样，反射观察向量是在贴图的基底上计算的，即在贴图的参考坐标系中进行计算的。反射观察向量的 z 分量的符号，用于决定访问具体哪一个纹理。这个访问函数为：

$$u = \frac{r_x}{2\left(1 + r_z\right)} + 0.5, \quad v = \frac{r_y}{2\left(1 + r_z\right)} + 0.5 \tag{10.33}$$

方程 10.33 用于正面图像；同理，当 r_z 的符号取反时，则用于背面图像。

与球面贴图相比，抛物线贴图具有更加均匀的环境采样，甚至均匀性可以与立方体贴图相比。但是，当在两个投影之间的接缝处进行采样和插值的时候，必须十分小心，这使得访问双抛物面环境贴图的成本很高。

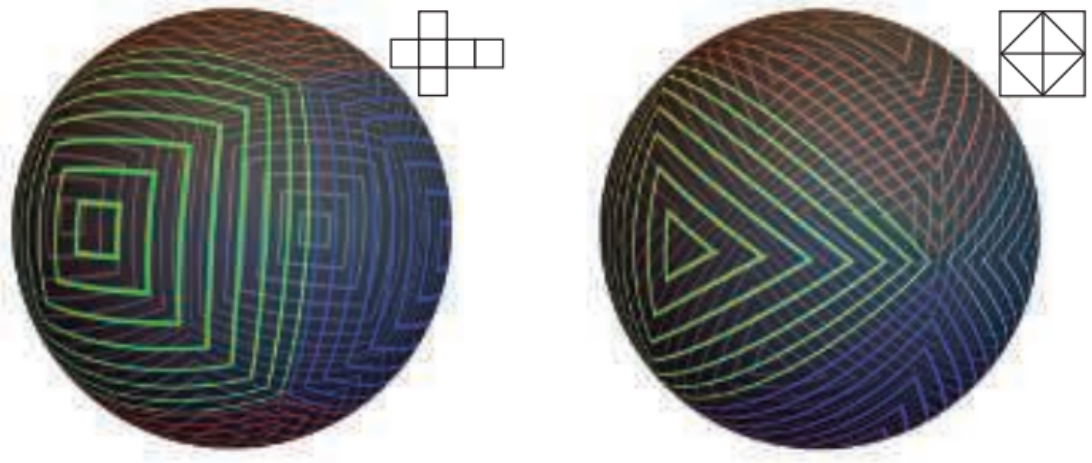


图 10.32：球体的立方体展开和八面体展开。

八面体映射（octahedral mapping）[434]是另一个值得注意的投影方法。它没有将周围的球面环境映射到一个立方体中，而是映射到了一个八面体中（如图 10.32 所示）。为了将这个八面体展开平铺成一个纹理，它的八个三角形面会被切开并排列在一个平面上，在理论上，将其排列成正方形结构或者矩形结构都是可以的。如果我们将其排列成正方形结构，那么访问八面体贴图的效率将会很高。对于给定的反射方向 $\mathbf{r}$，我们使用 L_1 范数的绝对值来将其归一化：

$$\mathbf{r}' = \frac{\mathbf{r}}{|r_x| + |r_y| + |r_z|}$$

如果 r'_y 为正，那么我们可以使用下面的坐标来索引正方形的纹理贴图：

$$u = r'_x \cdot 0.5 + 0.5, \quad v = r'_y \cdot 0.5 + 0.5 \tag{10.34}$$

如果 r'_y 为负，那么我们则需要通过变换，来将八面体的后半部分向外“折叠”：

$$u = (1 - |r'_z|) \cdot \text{sign}\,(r'_x) \cdot 0.5 + 0.5, \quad v = (1 - |r'_x|) \cdot \text{sign}\,(r'_z) \cdot 0.5 + 0.5 \tag{10.35}$$

与双抛物面环境映射不同，八面体映射并不会受到过滤问题的影响，因为参数化的接缝与所用纹理的边缘相对应。纹理的“wrap”采样模式可以自动从另一边访问纹素，并进行正确的插值。虽然八面体映射投影操作的数学计算要稍微复杂一些，但是在实践中的表现会更好。八面体贴图所引入的变形量与立方体贴图差不多，所以当立方体贴图纹理硬件不存在或者不支持的时候，八面体贴图是一个不错的选择。另外一个值

得注意的用法是，仅使用两个坐标来表示三维的归一化方向，从而能够对纹理进行一定程度的压缩（详见章节 16.6）。

有一种特殊情况是，环境贴图会围绕一个轴径向对称。对于这种情况，Stone [1705] 提出了一种简单的分解方法，使用一个一维纹理，存储来自对称轴子午线（meridian line）上的 radiance 值。后来他将这个方案扩展到二维纹理，在每一行中都存储一个环境贴图，并与不同的 Phong 波瓣进行预卷积。这种编码方式可以模拟大量材质，并被应用于编码从晴朗天空所发出的 radiance。

10.5 基于图像的高光照明

环境映射最初是作为一种镜面渲染技术发展起来的，但是也可以将它扩展到光泽（glossy）反射中。当用于模拟无限远处光源的镜面效果时，这样的环境贴图也被称为高光探针（specular light probe）。之所以使用这样一个术语，是因为它捕获（探测）了场景中给定点的各个方向上的 radiance，并使用这个信息来计算一般的 BRDF——而不仅仅针对于纯镜面或者 Lambertian 表面的特殊情况。可以将环境光照存储在立方体贴图中，并使用这些立方体贴图来控制光泽材质的反射效果，这样的贴图也被叫做高光立方体贴图（specular cube map）。

为了模拟表面的粗糙度，可以对纹理中的环境光照信息进行预滤波（prefiltered）[590]。通过对环境贴图进行模糊处理，我们可以表现出比完美镜面反射稍微粗糙一些的镜面反射效果。这种模糊处理应当以一种非线性的方式进行，即不同部分的纹理应当以不同的方式进行模糊。这种非线性处理是有必要的，因为环境贴图的纹理代表了一种到理想球面方向空间的非线性映射，两个相邻纹素中心之间的角距离是不固定的，单个纹素所覆盖的立体角也是不固定的。一些专门用于预处理立方体贴图的工具，例如 AMD 的 CubeMapGen（现在是开源的），在对立方体贴图进行过滤的时候会考虑到这些因素。来自其他面的邻近样本会被用来创建 mipmap，并且每个纹素所覆盖的角度范围也会被考虑在内。图 10.33 展示了这样的一个例子。

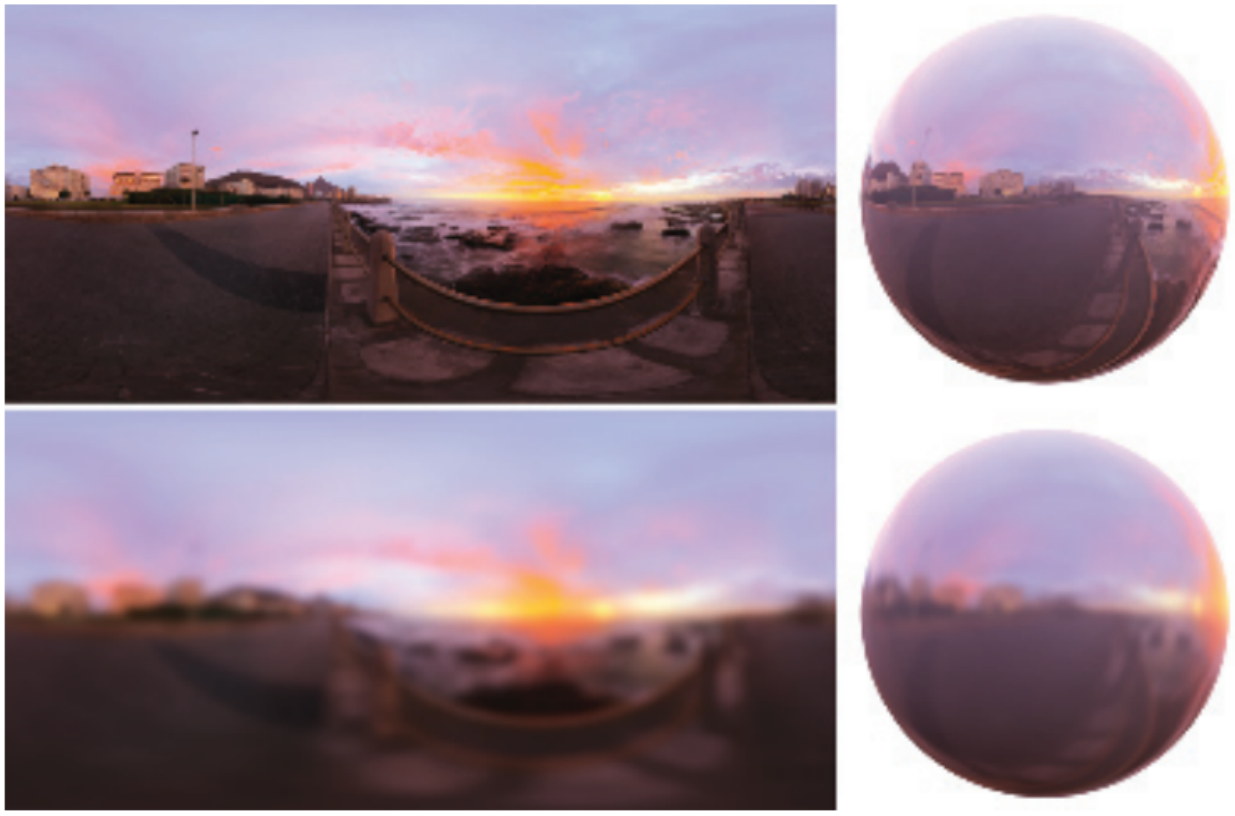


图 10.33：在第一行中，左侧是原始环境贴图，右侧是将其应用到一个球体上的着色结果。在第二行中，使用高斯滤波器对相同的环境贴图进行模糊处理，模拟了粗糙材质的外观。

这种通过对环境贴图进行模糊，同时根据经验来近似粗糙表面的外观，这个过程与实际的 BRDF 没有任何联系。一个更有原则的方法是，当给定表面法线和观察方向的时候，考虑 BRDF 函数在球面上的形状，然后我们再使用这个分布来对环境贴图进行过滤，如图 10.34 所示。使用一个镜面波瓣来对环境贴图进行过滤，这并不是一件简单的事情，因为 BRDF 可以表现出任意的形状，这取决于其粗糙度参数、观察向量和法线方向；也就是说，至少有五个维度的输入值（粗糙度和两组极坐标角度，分别用于表示观察方向和法线方向）控制了最终的波瓣形状。为其中的每个参数组合存储多张环境贴图是不现实的。

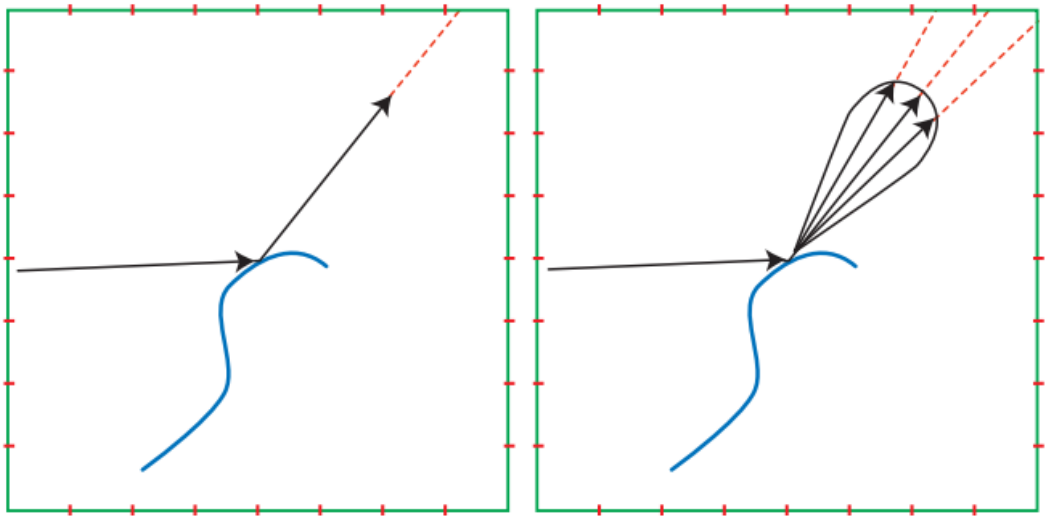


图 10.34：左侧展示了一个被物体反射的光线，它被反射到了完美镜面反射方向上，使用这个方向来从环境纹理（在本例中是立方体贴图）中检索对应的值。右图展示了反射光线的镜面波瓣，它用于对环境纹理进行采样。绿色方块表示立方体贴图的横截面，红色的虚线代表了纹素之间的边界。

10.5.1 预过滤环境映射

在实际实现中，应用于光泽材质的预过滤环境光照需要使用 BRDF 的近似值，以便生成的纹理能够独立于观察方向和表面法线。如果我们将 BRDF 的形状变化限制在材质光泽度上，即只有材质的光泽度会对 BRDF 的形状产生影响，那么我们就可以根据不同的粗糙度参数，预计算并存储对应的环境贴图，并在运行过程中，选择一个合适的环境贴图进行使用。在实践中，这意味着我们不需要使用模糊滤波核，也可以对波瓣形状进行控制，使得这个波瓣在反射向量周围径向对称。

想象从一个给定的反射观察方向射入的一组光线，直接来自这个反射观察方向上的光线将提供最大的贡献值，随着入射光线方向与反射观察方向之间的夹角越来越大，这些光线的贡献也会越来越小。环境贴图的纹素面积与纹素的 BRDF 贡献值相乘，给出了该纹素的相对影响力。这个加权贡献乘上环境贴图的纹素颜色，并将结果进行求和可以获得 $\mathbf{q}$，同时也会计算加权贡献的总和 s。最后的结果 $\mathbf{q}/s$，就是在反射观察方向的波瓣上整体的颜色，将其存储在生成的反射贴图中。

如果我们使用 Phong 材质模型，那么这种径向对称的假设自然而然是成立的，我们几乎可以准确地计算环境光照。Phong [1414]是根据经验来推导出这个模型的，与我们在章节 9.8 中所看到的 BRDF 相比，这个模型并没有物理依据。Phong 模型和我们在章节 9.8.1 中讨论的 Blinn–Phong BRDF [159]都是幂次的余弦波瓣，但是在 Phong 模型中，这个余弦是由反射向量（详见方程 9.15）和观察向量的点积而来

的；而在 Blinn–Phong BRDF 中，这个余弦是由半向量（详见方程 9.33）和表面法线点积而来的。这使得反射波瓣是旋转对称的，详见图 9.35 所示。

对于一个径向对称的镜面波瓣，我们仍然无法接受的唯一效果是地平线裁剪（horizon clipping），因为它会使得镜面波瓣的形状依赖于观察方向。想象现在我们有一个闪亮的球体（其表面并不是镜面），此时如果我们观察球体表面的中心区域，就会得到一个对称的 Phong 波瓣。而如果我们观察球体轮廓的表面区域时，实际上波瓣的一部分必须要被切割掉，因为来自地平线以下的光线无法到达相机，如图 10.35 所示。这和我们之前讨论面光源近似值（章节 10.1）时所看到的问题是一样的，但是在实际应用中，这种现象经常会被实时方法所忽略。忽略这个效应，会使得在掠射角度下形成过于明亮的着色效果。

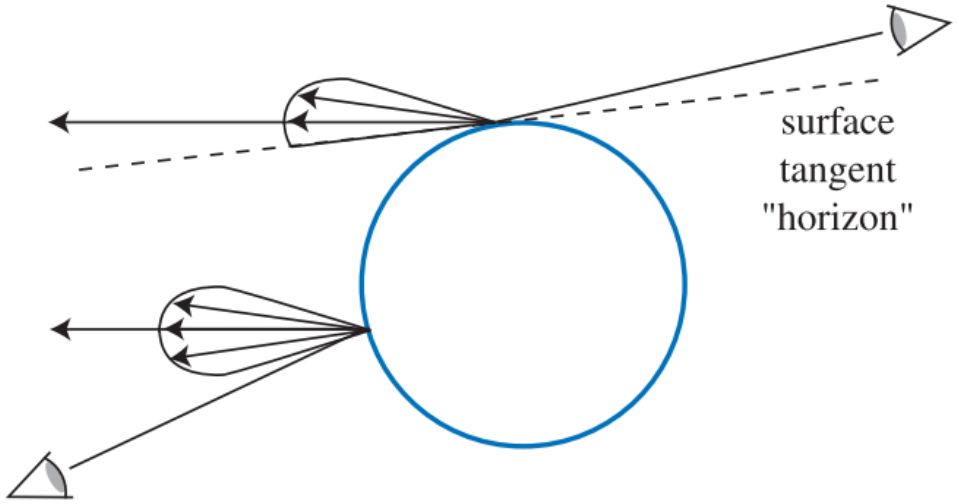


图 10.35：两个观察者看向同一个闪亮的球体。球体上两个不同的位置对于两个观察者产生了相同的反射方向。左侧观察者所看到的表面反射是一个对称的波瓣。右边观察者所看到的反射波瓣则会被表面本身的视界所截断，因为光线并不会被视界（地平线）以下的表面所反射。

Heidrich 和 Seidel [704]以这种方式，使用单个反射贴图来模拟表面的模糊性。为了适应不同的粗糙度水平，通常会使用环境立方体贴图的 mipmap（章节 6.2.2）。mipmap 的每个级别都存储了入射 radiance 的模糊版本，较高级别的 mipmap 用于存储较为粗糙的表面，即更宽的 Phong 波瓣 [80, 582, 1150, 1151]。在运行过程中，我们通过使用反射向量来索引立方体贴图，并根据所需的 Phong 指数（材质粗糙度）来强制选择给定的 mipmap 层级，如图 10.36 所示。

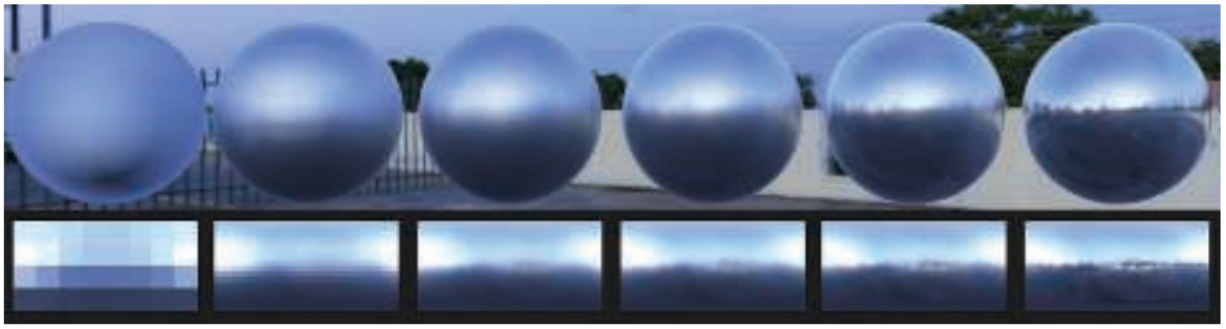


图 10.36：环境贴图预过滤。将立方体贴图与不同粗糙度的 GGX 波瓣进行卷积，并将结果存储在纹理的 mipmap 中。粗糙度从左到右递减，下面一行展示了生成的对应 mipmap，上面一行展示了在反射方向上渲染的球体。

对于比较粗糙的材质，会使用较宽的滤波核来去除高频信息，因此需要使用较低的分辨率才能得到充分模糊的结果，这正好与 mipmap 结构完美对应。此外，通过使用 GPU 硬件的三线性插值，可以在预滤波的 mipmap 层级之间进行采样，从而模拟我们没有精确表示的粗糙度值。当与菲涅尔项结合时，这种反射贴图对于光泽表面的效果要更好。

出于性能和抗锯齿的原因，具体选择要使用哪一个 mipmap 层级，不仅要考虑着色点处的材质粗糙度，还要考虑屏幕像素足迹所覆盖的表面区域内的法线变化和粗糙度变化。Ashikhmin 和 Ghosh [80]指出，为了获得最佳效果，应当对两个候选 mipmap 层级进行比较（由纹理硬件计算而来的最小化层级与当前滤波器宽度所对应的层级），并在二者中选择使用分辨率较低的那个 mipmap 层级。为了使得结果更加准确，还应当考虑表面方差的扩大效应，并使用一个新的粗糙度水平，这个粗糙度水平与一个 BRDF 波瓣项有关，这个 BRDF 波瓣可以视为像素足迹内的平均波瓣。这个问题与 BRDF 抗锯齿（章节 9.13.1）完全相同，并且适用相同的解决方案。

前面提出的过滤方案都有着这样的一个假设，即对于给定的反射方向，所有的波瓣都具有相同的形状和高度。这个假设也意味着波瓣必须是径向对称的。除了上述我们提到地平线裁剪问题之外，大多数 BRDF 波瓣在所有角度也上并不都是均匀的、径向对称的，例如：在掠射角度下，波瓣通常会变得更尖更薄；此外，波瓣的长度通常也会随着仰角的变化而变化。

上述的这些效应对于弯曲表面而言通常是很难感知到的，然而对于一些平坦的表面而言（例如地板），径向对称的滤波器可能会引入明显的误差。（详见图 9.35）

卷积环境贴图

想要生成预过滤的环境贴图意味着，要对与方向 $\mathbf{v}$ 相关的每个纹素上，计算环境 radiance 与镜面波瓣 D 的积分：

$$\int_{\Omega} D(\mathbf{l}, \mathbf{v}) L_i(\mathbf{l}) d\mathbf{l}.$$

这个积分是一个球面卷积（spherical convolution），由于其中的 L_i 只能通过环境贴图的表格形式获得，因此这个积分通常无法解析求解，只能通过数值方法进行求解。一种流行的数值方法是采用蒙特卡罗方法：

$$\int_{\Omega} D(\mathbf{l}, \mathbf{v}) L_i(\mathbf{l}) d\mathbf{l} \approx \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \frac{D\left(\mathbf{l}_k, \mathbf{v}\right) L_i\left(\mathbf{l}_k\right)}{p\left(\mathbf{l}_k, \mathbf{v}\right)} \tag{10.36}$$

其中 $\mathbf{l}_k$ （ $k = 1, 2, \ldots, N$ ）是单位球面上的离散样本（方向）； $p\left(\mathbf{l}_k, \mathbf{v}\right)$ 是在方向 $\mathbf{l}_k$ 上生成样本（采样）的概率密度函数，如果我们对球面进行均匀采样，则总有 $p\left(\mathbf{l}_k, \mathbf{v}\right) = 1$ 。虽然这个总和对于我们想要积分的每个方向 $\mathbf{v}$ 而言都是正确的，但是当结果存储在环境贴图中时，我们还必须考虑投影所带来的失真扭曲，因此需要对每个纹素，按照其所占据的立体角进行加权（详见 Driscoll [376]）。

虽然蒙特卡洛方法是简单且正确的（无偏），但是它需要大量的样本才可能会收敛到积分的数值，即使对于离线过程而言也是很慢的。这种情况对于 mipmap 的第一级而言尤其明显，在第一级中，我们会对较浅的高光波瓣进行编码（在 Blinn–Phong 模型中是高指数，在 Cook–Torrance 模型中是指低粗糙度）。在这种情况下，我们不仅需要对更多的像素进行计算（因为我们需要更高的分辨率来存储高频细节），而且对于那些不接近完美反射的方向而言，其波瓣可能会接近于零。当 $D\left(\mathbf{l}_k, \mathbf{v}\right) \approx 0$ 时，大多数样本都是无效的，这样的样本只会浪费计算量。

为了避免这种现象，我们可以使用重要性采样（importance sampling），在这种情况下，我们会使用一个概率分布来生成采样方向，这个概率分布会尽量贴合镜面波瓣的形状。这是蒙特卡罗积分中一种常见的方差降低技术，并且对于大多数常用的波瓣而言，都存在使用重要性采样的策略[279, 878, 1833]。为了获得更加有效的采样方案，还可以将环境贴图中的 radiance 分布与镜面波瓣的形状结合起来考虑 [270, 819]。然而，所有依赖于点采样的技术，通常只能用于离线渲染和生成 ground–truth 中，因为这个过程通常需要使用数百个样本，需要耗费的时间过长。

为了进一步降低采样方差（即噪声），我们还可以对样本之间的距离进行估计，并使用锥形方向上的和，而不是单一方向来进行积分。使用锥形对环境贴图进行采样，可以通过对其中一个 mipmap 层级进行点采样来近似，直到某个层级的纹素与锥形占据大致相同的立体角[280]。虽然这样做会引入偏差，但是它可以大大降低实现无噪声

结果所需的样本数量，这种类型的采样可以在 GPU 的帮助下，以交互式速率进行执行。

McGuire 等人[1175]开发了一种同样使用区域样本的技术，它旨在对镜面波瓣的卷积结果进行实时近似，并且不需要任何预计算。这个过程十分巧妙，它对非预过滤环境立方体贴图的多个 mipmap 层级进行了混合，以便能够重建 Phong 波瓣的形状。类似地，Hensley 等人[718, 719, 720]使用了 SAT（详见章节 6.2.2）来快速实现近似。McGuire 等人和 Hensley 等人的技术严格来说还是存在预计算的，因为在渲染环境贴图之后，它们仍然需要生成 mipmap 层级或者前缀和（prefix sum，用于 SAT），而对于这两种常见操作，都存在一些高效算法，因此所需的预计算量要比执行整个镜面波瓣卷积少得多。这两种技术的速度都很快，甚至可以用于环境光照的实时表面着色，但是缺点在于，不如其他依赖于预过滤的方法准确。

Kautz 等人[868]提出了另一种变体方法，一种快速生成滤波抛物面反射贴图的分层技术。最近，Manson 和 Sloan [1120]使用了一种高效的二次 B 样条滤波方案，来生成环境贴图的 mipmap，这显著提高了 SOTA 的水平。这些经过特殊计算的 B 样条滤波 mipmap，之后会通过组合少量样本进行使用，这与 McGuire 等人和 Kautz 等人的技术相类似，它可以生成快速且准确的近似结果。使用这种方法可以实时生成结果，而且与使用重要性采样的蒙特卡罗方法所计算出的 ground-truth 相比，二者的质量相当。

当我们想要过滤的环境贴图是动态渲染而来的时候，快速卷积技术允许我们对预过滤立方体贴图进行实时更新。使用环境贴图通常会使得物体难以在不同的光照情况下进行移动，例如：从一个房间移动到另一个房间。我们可以逐帧生成立方体环境贴图，或者每隔几帧重新生成一次；如果我们采用了一些比较高效的滤波方案，那么替换新的镜面反射贴图也是相对高效的。

另一种重新生成整个环境贴图的方法是，将动态光源的高光效果叠加到基础的静态环境贴图上，所添加的高光可以是预先过滤的“亮斑”，这些亮斑会被添加到预过滤的基础环境贴图中，这样做可以避免在运行时执行过滤操作。这种思路也有一些局限性，这个局限性来自于环境贴图的基本假设：光源和反射物体都位于远处，因此不会随着观察物体位置的改变而改变。这个基本假设意味着我们不能轻易使用局部光源。

如果场景中的几何物体是静态的，但是一些光源（例如太阳）是动态的，此时有一种十分廉价的技术，它不需要将场景动态渲染到立方体贴图中，而是将表面属性（位置、法线、材质）存储在在一个 G-buffer 环境贴图中。有关 G-buffer 的内容将在章节 20.1 中进行详细讨论。然后，我们使用这些属性计算表面的出射 radiance，并

将其记录在环境贴图中。《使命召唤：无限战争》[384]、《巫师 3》[1778]和《孤岛惊魂 4》[1154]等游戏都使用了这种技术。

10.5.2 微表面 BRDF 的分裂积分近似

环境光照十分有用，因此出现了许多技术来减少立方体贴图预过滤中固有的 BRDF 近似问题。

到目前为止，我们假设所使用的 BRDF 是一个 Phong 波瓣，然后再将其乘以一个完美镜面的菲涅尔项，从而构建了如下形式的近似：

$$\int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \approx F(\mathbf{n},\mathbf{v})\int_{\mathbf{l}\in\Omega} D_{\text{Phong}}(\mathbf{r})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \tag{10.37}$$

其中 $\int_\Omega D_{\text{Phong}}(\mathbf{r})$ 是环境贴图中每个反射方向 $\mathbf{r}$ 的预计算结果。如果此时我们考虑方程 9.34 中所使用的镜面微表面 BRDF f_{smf}，这里为了方便理解，我们再次给出这个 BRDF 的方程：

$$f_{\text{smf}}(\mathbf{l},\mathbf{v}) = \frac{F(\mathbf{h},\mathbf{l})G_2(\mathbf{l},\mathbf{v},\mathbf{h})D(\mathbf{h})}{4|\mathbf{n}\cdot\mathbf{l}||\mathbf{n}\cdot\mathbf{v}|} \tag{10.38}$$

我们可以注意到，即使 $D(\mathbf{h}) \approx D_{\text{Phong}}(\mathbf{r})$ 是有效的，我们在方程 10.37 中积分也去掉了 BRDF 最重要的几个部分：shadowing–masking 项 $G_2(\mathbf{l},\mathbf{v},\mathbf{h})$ 和半向量菲涅尔项 $F(\mathbf{h},\mathbf{l})$。并且将菲涅尔项放到积分外部是没有理论基础的。Lazarov 指出[998]，与完全不使用菲涅尔项相比，使用依赖于 $\mathbf{n}\cdot\mathbf{v}$ 的完美镜面的菲涅尔项，而不是微表面 BRDF 中的 $\mathbf{n}\cdot\mathbf{h}$，会产生更大的误差。Goanda[573]， Lazarov[999]和 Karis[861]都独立推导出了一个类似的分裂积分近似（split–integral approximation）：

$$\int_{\mathbf{l}\in\Omega} f_{\text{smf}}(\mathbf{l},\mathbf{v})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \approx \int_{\mathbf{l}\in\Omega} D(\mathbf{r})L_i(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l}\int_{\mathbf{l}\in\Omega} f_{\text{smf}}(\mathbf{l},\mathbf{v})(\mathbf{n}\cdot\mathbf{l})d\mathbf{l} \tag{10.39}$$

请注意，尽管这个解决方案通常被称为“分裂积分（split integral）”，但是我们实际上并没有将积分分解为两个拆开的项，因为那并不是一个很好的近似。需要牢记的是，f_{smf} 中包含了镜面波瓣 D，同时我们注意到 D 和 $\mathbf{n}\cdot\mathbf{l}$ 项都出现在了右侧的第一个被积函数中。在上面的分裂积分近似中，我们在两个被积函数里都包含了环境贴图中围绕反射向量对称的所有项。Karis 将他的推导称为分裂求和（split–sum），因

为它是在预计算中所使用的重要性采样数值积分器（方程 10.36）上完成的，但实际上它是完全相同的解。

由此得到的两个积分都可以进行高效地预计算。其中第一种方法假设波瓣 D 是径向对称的，该方法仅依赖于表面的粗糙度和反射向量，而在实践中，我们可以使用任意的波瓣，只要让 $\mathbf{n} = \mathbf{v} = \mathbf{r}$ 即可。这个积分可以像之前一样进行预计算，并存储在立方体贴图的 mipmap 中。当将半向量 BRDF 转换为绕反射向量的波瓣时，为了在环境光照和解析光源（analytic light）之间获得类似效果的高光，径向对称的波瓣应当使用修正的粗糙度。例如：要将一个基于 Phong 的反射向量镜面项，转换为一个基于半角的 Blinn–Phong BRDF，可以将指数除以 4，来获得较好的拟合结果[472, 957]。

第二个积分是镜面项的半球定向反射率 $R_{\text{spec}}(\mathbf{v})$（详见章节 9.3）。函数 R_{spec} 取决于仰角 θ，粗糙度 α 和菲涅尔项 F。通常菲涅尔项 F 是使用 Schlick 近似（方程 9.16）实现的，它只有 F_0 一个参数，因此函数 R_{spec} 是一个拥有三个输入参数的函数。Gotanda 对函数 R_{spec} 进行了数值预计算，并将结果存储在一个三维查找表中。Karis 和 Lazarov 注意到，可以将 F_0 从 R_{spec} 中提取出来，这会产生两个因子，每个因子都取决于这两个参数：仰角 θ 和粗糙度 α。基于这种观察和发现，Karis 将函数 R_{spec} 的预计算查找表压缩成了一个可以存储在双通道纹理中的二维表格，而 Lazarov 则通过函数拟合，对这两个因子分别进行了解析近似。后来，Iwanicki 和 Pesce [807]推导出了一种更加精确且更加简单的解析近似。请注意，R_{spec} 还可以用于提高漫反射 BRDF 模型的准确度（详见方程 9.65）。如果应用程序中同时实现了这两种技术，那么 R_{spec} 的实现可以同时用于这两种技术中，从而提高效率。

对于一个恒定的环境贴图，分裂积分的解是精确的。立方体贴图部分提供了与镜面反射率成比例的光照强度，这是均匀光照下的正确 BRDF 积分。根据经验，Karis 和 Lazarov 都观察到，这种近似也适用于一般的环境贴图，尤其是当内容频率相对较低的时候（低频信息），这种情况在一些户外场景中很常见，如图 10.37 所示。与 ground–truth 相比，该技术最大的误差来源是，它假设预过滤环境立方体贴图是径向对称的、没有被裁剪的镜面波瓣（如图 10.35 所示）。Lagarde 建议[960]，可以根据表面粗糙度，将用于检索预过滤环境贴图的向量，从反射方向往法线方向稍稍倾斜，因为从经验上来看，与 ground–truth 相比，这样做可以减少误差。这样做其实是有理论依据的，因为它对没有被表面入射 radiance 半球所剪切的波瓣进行了部分补偿。

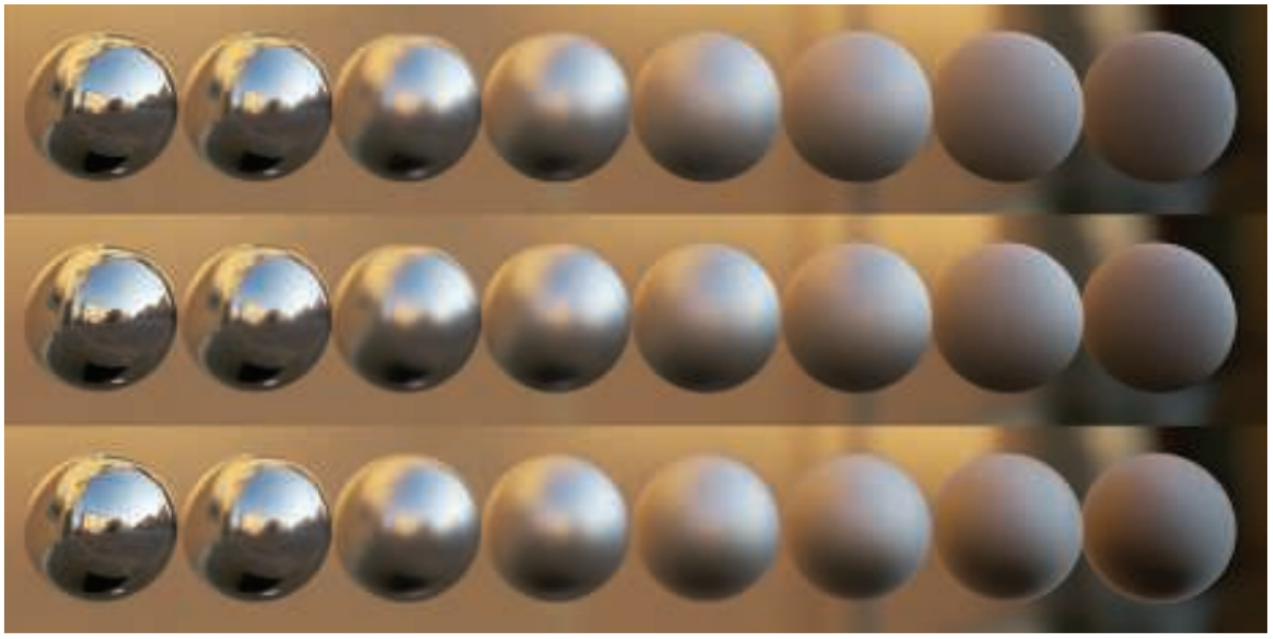


图 10.37：Karis 的“分裂求和”近似。从左到右，材质的粗糙度越来越高。第一行：参考解决方案。第二行：分裂积分近似。第三行：在镜面波瓣上加入所需径向对称（$\mathbf{n} = \mathbf{v} = \mathbf{r}$）的分裂积分。最后一行会引入最多的误差。

10.5.3 不对称和各向异性波瓣

到目前为止，我们所看到的解决方案都局限于各向同性的镜面波瓣，这意味着当入射方向和出射方向围绕表面法线旋转的时候，这些波瓣的形状和大小不会发生改变（详见章节 9.3），并且仍然会围绕反射向量径向对称。微表面 BRDF 波瓣是围绕着半向量 $\mathbf{h} = (\mathbf{l} + \mathbf{v})/\|\mathbf{l} + \mathbf{v}\|$ （方程 9.33）进行定义的，因此即使是在各向同性的情况下，也不会出现我们所需要的对称性。半向量依赖于光线方向 $\mathbf{l}$，对于环境光照而言，光线方向并不是唯一的。因此，与 Karis [861]的做法一样，对于这样的 BRDF，我们会让 $\mathbf{n} = \mathbf{v} = \mathbf{r}$，并推导出一个恒定的粗糙度修正因子，来让镜面高光的尺寸与原始的半向量方程相匹配。这些假设都是相当大的误差来源，如图 10.38 所示。

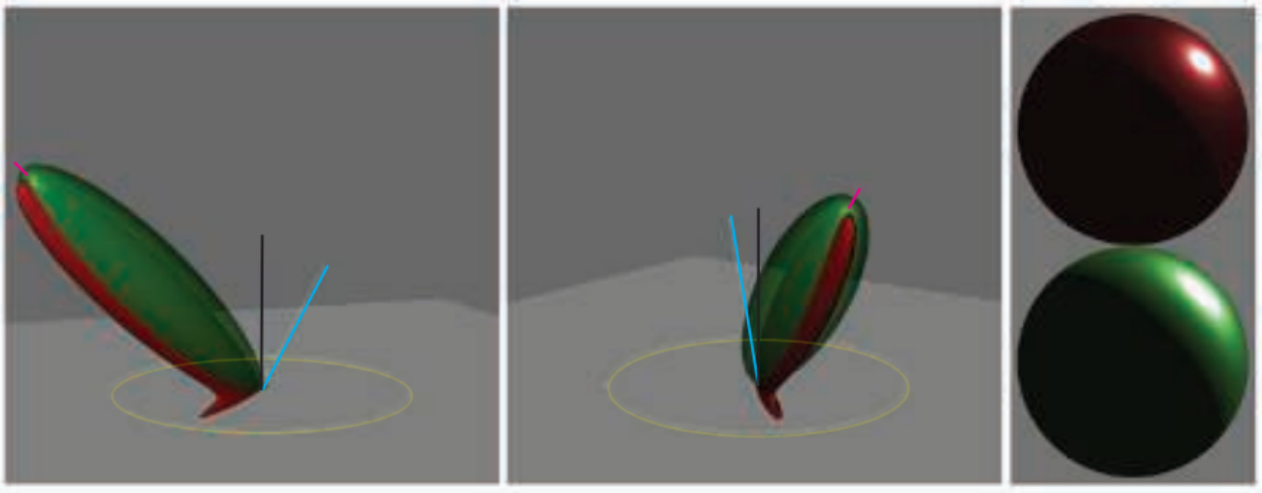


图 10.38：红色代表了 GGX BRDF，绿色代表了围绕反射向量径向对称的 GGX NDF 波瓣。后者进行了缩放修正，从而与 GGX 镜面波瓣的峰值相匹配，但是请注意，它无法捕获基于半向量 BRDF 的各向异性形状。在最右边，使用这两个波瓣各自渲染了一个球体，请注意二者之间高光的差异。

我们在章节 10.5.1 中所提到的一些方法（例如来自 Luksch 等人[1093]、Colbert 和 Krivanek [1093]的方法），可以在交互速率下计算具有任意 BRDF 的环境光照。然而，由于这些方法往往需要几十个样本来收敛，因此很少用于物体的实时表面着色，它们可以被视为蒙特卡洛积分中的重要性采样技术。

通过在镜面波瓣上施加径向对称来创建预过滤的环境贴图，使用这个环境贴图的逻辑也很简单直接，即访问与当前表面镜面粗糙度相适应的预过滤波瓣即可。我们的结果只有在直视表面（$\mathbf{n} = \mathbf{v}$）的情况下才能保证是正确，而在其他任何其他情况下我们都不能保证正确；并且在掠射视角下，无论 BRDF 波瓣的形状如何，都会产生一些误差，因为我们忽略了真实的波瓣不能低于着色表面点视界这一事实（地平线裁剪）。一般来说，在使用这种方法的情况下，镜面反射方向上所对应的数据很可能与实际情况不那么匹配。

Kautz 和 McCool 通过对存储在预过滤环境贴图中的径向对称波瓣使用更好的采样方案，从而对原生的预积分进行了改进[867]。他们提出了两种方法：第一种方法使用单个样本，它试图找到一个最佳波瓣，来对当前观察方向上的 BRDF 进行近似，而不是依赖一个恒定的修正因子。第二种方法则是对来自不同波瓣的几个样本进行平均。第一种方法可以更好地模拟掠射角度下的表面。他们还推导出了一个修正因子，以修正使用径向对称波瓣近似方法与原始 BRDF 相比，反射出的总能量差异。第二种方法还对结果进行了扩展，包括典型半向量模型的拉伸高光。在这两种情况下，都使用了优化技术来计算驱动预过滤波瓣采样的参数表。Kautz 和 McCool 的技术还使用了贪婪拟合算法和抛物线环境贴图。

最近，Iwanicki 和 Pesce [807]推导了一种 GGX BRDF 与环境立方体贴图的类似近似，他们使用了一种称为 Nelder–Mead 最小化的方法。他们还分析了利用现代 GPU 硬件的各向异性过滤能力来加速采样的想法。

使用预过滤立方体贴图中的单个样本，但是将其位置调整到一个更加复杂的镜面 BRDF 峰值的想法，被 Revie [1489]用于与延迟着色相结合的毛皮渲染中（章节 20.1）。在这种情况下，方法的限制并不是直接来自于环境贴图，而是需要在 G-buffer 中编码数量尽可能少的参数。McAuley [1154]对这个想法进行了扩展，将这种技术用于延迟渲染系统中的所有表面。

McAllister 等人[1150, 1151]开发了一种技术，该技术通过利用 Lafortune BRDF 的特性，可以实现各种渲染效果，包括各向异性和逆反射。这个 BRDF [954]本身就是一种基于物理渲染的近似，它由多个 Phong 波瓣组成，这些波瓣会在反射方向周围进行扰动。Lafortune 通过将这些波瓣拟合到 He–Torrance 模型[686]中，并通过一个叫做 goniorereflectometer 的装置，来对真实材质进行测量，证明了这种 BRDF 表达复杂材质的能力。McAllister 的技术基于这样一种观察：由于 Lafortune 波瓣是广义的 Phong 波瓣，因此可以使用传统的预过滤环境贴图，利用其 mipmap 层级来对不同的 Phong 指数进行编码。Green 等人[582]提出了一种类似的方法，该方法使用了高斯波瓣来代替 Phong 波瓣；此外，他们的方法还可以进行扩展，从而支持环境贴图的定向阴影（章节 11.4）。

10.6 irradiance 环境映射

上一小节中我们讨论了如何使用预过滤环境贴图来实现光泽反射，这些贴图同样也可以用于实现漫反射[590, 1212]。镜面反射所使用的环境贴图具有一些共同的属性，无论它们是用于镜面反射的未过滤贴图，还是用于光泽反射的过滤贴图。在这两种情况下，高光环境贴图都是使用反射观察向量来进行索引的，并且它们都包含了 radiance 值，其中未过滤的环境贴图包含了入射的 radiance，而过滤的环境贴图则包含了出射的 radiance。

相反，漫反射环境贴图只需要使用表面法线 $\mathbf{n}$ 来进行索引，并且它们所包含的是 irradiance 值，因此它们也被称为 irradiance 环境贴图[1458]。从图 10.35 中可以看出，使用环境贴图的光泽反射，由于其固有的模糊性（波瓣扩散角度），因此在某些条件下会产生误差（地平线裁剪），相同的反射观察向量可能会对应不同的反射情况。这些问题并不会发生在 irradiance 环境贴图中，因为表面法线包含了漫反射所有的相关信息。与原始光照相比，irradiance 贴图是极其模糊的，因此可以使用较低的分辨率来进行存储。通常会使用预过滤镜面环境贴图的最低 mipmap 层级来存储 irradiance 数据。此外，与我们上一小节研究的光泽反射不同，我们不再针对 BRDF 波瓣进行积分，BRDF 波瓣会被裁剪到表面法线周围的半球范围内。环境光照与 clamped 余弦波瓣（即 clamp 到 0 过后的余弦函数）的卷积是一个精确值，而不是一个近似值。

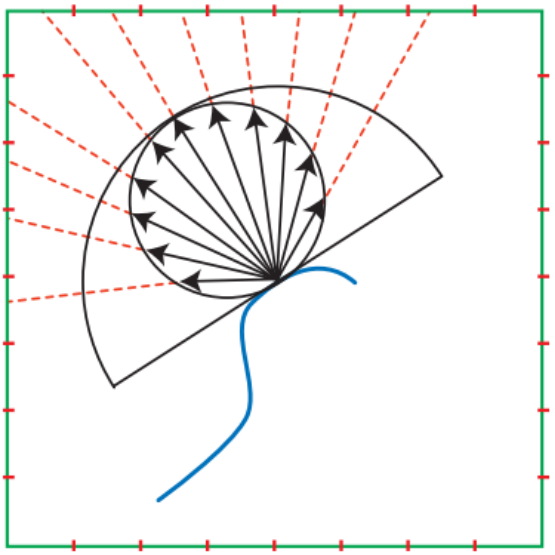


图 10.39：计算 irradiance 环境贴图的过程。在原始环境纹理（本例中是一个立方体贴图）中对表面法线周围的余弦加权半球进行采样，并对采样结果进行求和，从而获得与视图无关的 irradiance。图中绿色的方块代表了立方体贴图的横截面，红色的虚线代表了纹素之间的边界。这里使用的是一个立方体贴图，但实际上任何环境表示都是可以使用的。

对于贴图中的每个纹素，我们需要将影响给定的法线方向表面的所有光照，将其贡献按照余弦进行加权并求和。irradiance 环境贴图是通过对原始环境贴图应用一个 far-reaching（意味影响深远的，波及广泛的）的滤波器创建的，这个滤波器会覆盖整个可见的半球范围，同时该滤波器还包含了余弦因子，如图 10.39 所示。图 10.26 中所展示的球面贴图与图 10.40 所展示的 irradiance 贴图相对应。图 10.41 则展示了一个实际使用中的 irradiance 贴图。

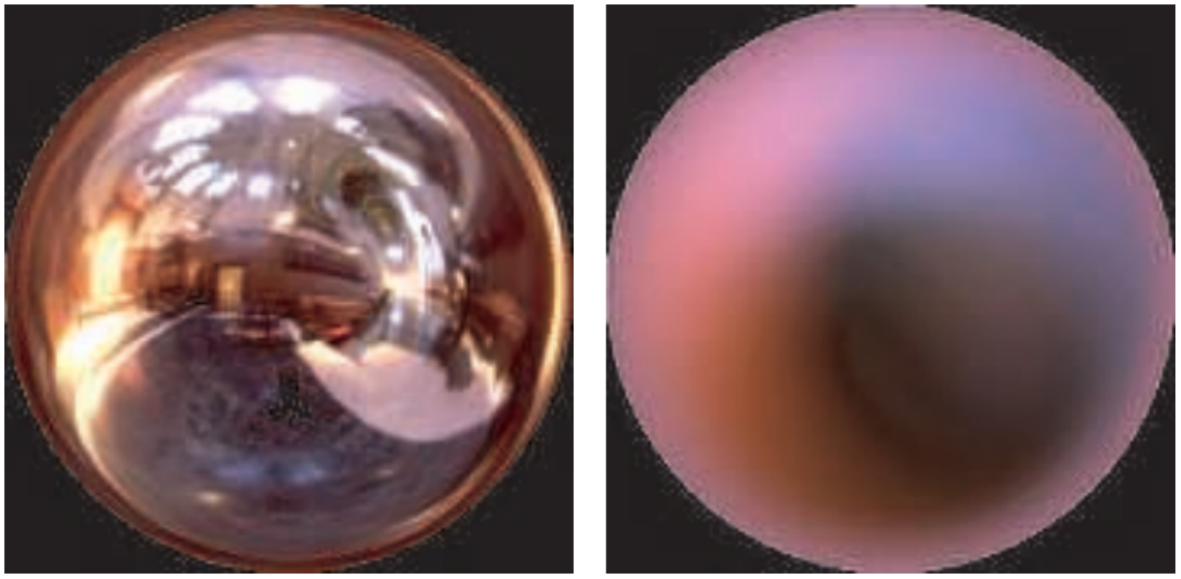


图 10.40：从 Grace 大教堂球面贴图生成的 irradiance 贴图。左侧图像是原始的球面贴图。右侧图像则是通过将每个像素上半球范围内的颜色进行加权求和而得到的。

图 10.41：使用 irradiance 贴图生成的人物光照，来自《死或生 3》。

irradiance 环境贴图的访问和存储通常是与高光环境贴图或者高光反射贴图分开的，其访问和存储方式通常是一种视图无关的表示形式，例如立方体贴图，如图 10.42 所示。在访问这个立方体贴图的时候，会使用表面法线来检索 irradiance 值，而不是之前的反射观察向量。从 irradiance 环境贴图中获得的值会和漫反射率相乘，从高光环境贴图中获得的值会和镜面反射率相乘。通过在掠射角度下增加镜面反射率（可能也会降低漫反射率），还可以模拟菲涅尔效应[704, 960]。

图 10.42：立方体贴图（左）及其过滤后的 irradiance 贴图（右）。

由于生成 irradiance 环境贴图使用了非常宽的滤波器，因此很难通过采样来在运行时高效地创建它们。King [897]讨论了如何在 GPU 上使用卷积操作来创建 irradiance

贴图，他通过将环境贴图转换到频域（frequency domain），能够在 2004 年的硬件上，以超过 300 FPS 的速率来生成 irradiance 贴图。

漫反射表面或者粗糙表面的过滤环境贴图可以以较低的分辨率进行存储，有时候也可以从相对较小的反射贴图中直接生成，例如一个每个面具有 64×64 纹素的立方体贴图。这种方法存在一个问题，将面光源渲染到如此小的纹理中，光线可能会“落在纹素之间的区域”，这会导致光线闪烁甚至完全消失。为了避免这个问题，Wiley 和 Scheuermann [1886]建议在渲染动态环境贴图的时候，使用较大的“卡片（即有纹理的矩形）”来表示这样的面光源。

在光泽反射的情况下，动态光源也可以被添加到预过滤的 irradiance 环境贴图中。Brennan [195]给出了一种廉价的方法，想象现在有一个单一光源的 irradiance 贴图，在光源的方向上，当光线直射到表面上的时候，radiance 会达到最大值。给定表面法线（即给定纹素）的 radiance 会随着与光线夹角的余弦值而下降，并且在表面背对光源的地方达到 0。GPU 可以通过渲染一个半球，这个半球代表了余弦波瓣，以观察者为中心，半球的极点指向光源，从而快速地将这个光源的贡献添加到现有的 irradiance 贴图中。

10.6.1 球谐 irradiance

上文中，我们讨论了仅使用贴图（例如立方体贴图）来表示 irradiance 环境贴图，还有一些其他表示方法也可以用来存储环境 irradiance 信息，如章节 10.3 所述。尤其是球谐函数，使用它来表示 irradiance 环境贴图是相当流行的，因为环境光照中的 irradiance 是平滑的。将 radiance 与余弦波瓣进行卷积，可以从环境贴图中去除所有高频信息。

Ramamoorthi 和 Hanrahan 表明[1458]，仅使用前 9 个 SH 系数（每个系数都是一个 RGB 向量，因此我们需要存储 27 个浮点数）就可以以 1% 的误差来表示 irradiance 环境贴图。这样一来，任何 irradiance 环境贴图都可以表示为一个球面函数 $E(\mathbf{n})$，并使用方程 10.21 和方程 10.23 将其投影到 9 个 RGB 系数上。相比立方体贴图和抛物线贴图而言，这种表示形式更加紧凑；并且在渲染的过程中，可以通过一些简单的多项式计算来重建 irradiance 信息，而不是去读取和检索纹理。一半来说，如果 irradiance 环境贴图表示的是间接照明，那么只需要较少的精度即可，这在交互式应用中十分常见。在这种情况下，我们只使用常量基函数和三个线性基函数，即一共四个系数（四个 RGB 向量，共 12 个浮点数）就可以产生很好的结果了，因为间接光照本身往往就是低频的，它随着角度的变化很缓慢。

Ramamoorthi 和 Hanrahan 还指出[1458]，通过将每个系数乘上一个常数，可以将入射 radiance 函数 $L(\mathbf{l})$ 的 SH 系数，转换为 irradiance 函数 $E(\mathbf{n})$ 的 SH 系数。这样做可以生成一种将环境贴图快速过滤为 irradiance 环境贴图的方法，通过将环境贴图投影到 SH 基底上，然后再将每个 SH 系数乘上一个常数即可完成。例如：King [897]的快速 irradiance 滤波就是这样实现的，其核心思想是：从 radiance 中计算 irradiance，实际上相当于在入射 radiance 函数 $L(\mathbf{l})$ 与 clamped 余弦函数 $\cos(\theta_i)^+$ 之间进行球面卷积。由于 clamped 余弦函数围绕球体的 z 轴是旋转对称的，因此它在 SH 中具有一种特殊的形式：它的投影在每个频带中只有一个非零系数。而这个非零系数则对应了图 10.21 中间一列的基函数，它们也被称为球带谐波（zonal harmonic）。

将一个一般的球面函数与旋转对称函数（例如 clamp 过的余弦函数）之间进行球面卷积，所产生结果是球面上的另一个函数，这种卷积操作可以在函数的 SH 系数上高效进行。卷积结果的 SH 系数等于两个函数 SH 系数的点积（乘法），再通过 $\sqrt{4\pi/(2l+1)}$ 进行缩放，其中 l 是频带指数（frequency band index）。irradiance 函数 $E(\mathbf{n})$ 的 SH 系数，等于 radiance 函数 $L(\mathbf{l})$ 的系数乘以 clamped 余弦函数 $\cos(\theta_i)^+$ 的系数，再按照频带常数进行缩放。clamped 余弦函数的前九个系数值很小，这也就解释了为什么只使用九个系数就足以表示 irradiance 函数 $E(\mathbf{n})$。SH irradiance 环境贴图可以使用这种方式来进行快速计算，Sloan [1656]描述了一种高效的 GPU 实现。

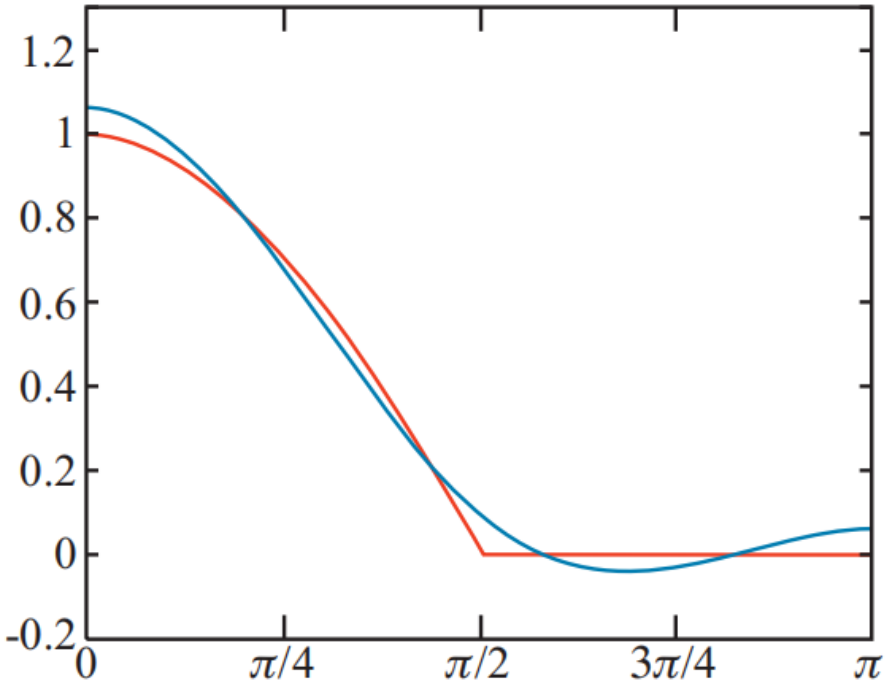


图 10.43：clamped 余弦函数（红色）与包含九个系数的球谐函数近似（蓝色）。近似结果十分接近。请注意在 $\pi/2$ 到 π 之间，蓝色曲线有轻微的下降和上升。

这里还存在一个固有的近似值，因为尽管函数 $E(\mathbf{n})$ 的高阶系数值很小，但是它们并不为零，如图 10.43 所示。虽然 $\pi/2$ 到 π 之间的曲线会发生“摆动”，这在信号处理中被称为振铃（ringing），但是这种近似总体还是非常接近的。这种振铃现象通常会发生在用少量基函数来对具有高频部分的函数进行近似的时候，详见章节 10.3.2。在 $\pi/2$ 处将函数 clamp 为 0，会带来一个急剧的变化，这意味着我们的 clamped 余弦函数具有无限大的信号频率。振铃现象在大多数情况下都是不明显的，但是在一些极端的光照条件下仍然可以看到这种现象，例如颜色的剧烈变化或者位于物体阴影附近的明亮斑点。如果 irradiance 环境贴图只是用来存储间接光照的话（通常会这么做），那么振铃现象就不太可能成为一个问题。还有一些预过滤方法可以最小化这个问题[1656, 1659]，如图 10.44 所示。

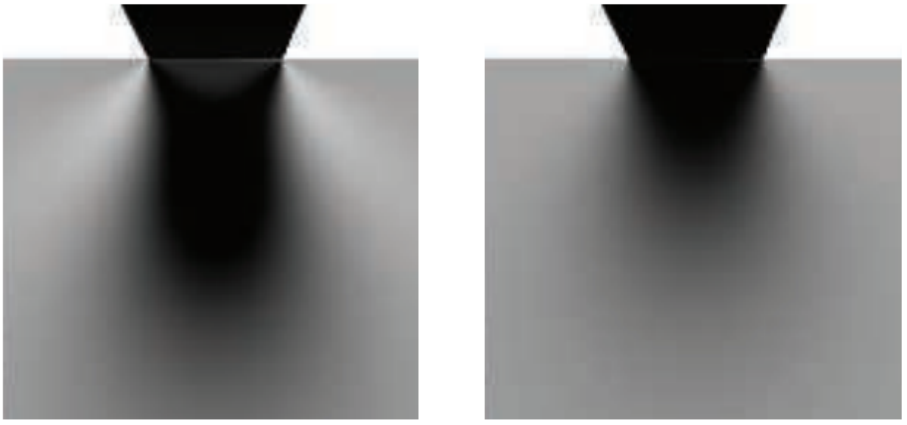


图 10.44：左图：由振铃现象所引起的视觉瑕疵。右图：一种可能的解决方案是让原始函数变得更加平滑，这样它就可以避免振铃现象，这个过程被称为“窗口化（windowing）”。

图 10.45 展示了直接导出的 irradiance 贴图，与使用包含九个系数的球谐函数的对比结果。这种 SH 表示可以在渲染的过程中，使用当前表面的法线 $\mathbf{n}$ 进行计算[1458]，也可以使用 SH 来快速创建一个立方体贴图或者抛物线贴图供之后进行使用。对于漫反射而言，这样的光照效果虽然很廉价，但是可以提供不错的视觉效果。

动态渲染的立方体环境贴图可以被投影到 SH 基底上[871, 897, 1458]。由于立方体环境贴图是入射 radiance 函数的离散表示，因此方程 10.21 中对球面的积分就变成了对立方体贴图纹素值的求和：

$$k_{Lj} = \sum_t f_j(\mathbf{r}[t])L[t]d\omega[t] \tag{10.40}$$

其中 t 是当前立方体贴图纹素的索引，$\mathbf{r}[t]$ 为指向当前纹素的方向向量，$f_j(\mathbf{r}[t])$ 是第 j 个 SH 基函数在 $\mathbf{r}[t]$ 处的值，$L[t]$ 是存储在纹素中的 radiance 值，$d\omega[t]$ 是该

纹素所对应的立体角。Kautz [871]、King [897]和 Sloan [1656]描述了如何计算这个立体角 $d\omega[t]$ 。

为了将 radiance 系数 k_{Lj} 转换为 irradiance 系数，需要将其乘以 clamped 余弦函数的缩放系数：

$$k_{Ej} = k'_{\cos^+ j} k_{Lj} = k'_{\cos^+ j} \sum_t f_j(\mathbf{r}[t]) L[t] d\omega[t] \tag{10.41}$$

其中 k_{Ej} 是 irradiance 函数 $E(\mathbf{n})$ 的第 j 个系数， k_{Lj} 是入射 radiance 函数 $L(\mathbf{l})$ 的第 j 个系数， $k'_{\cos^+ j}$ 是 clamped 余弦函数 $\cos(\theta_i)^+$ 按 $\sqrt{4\pi/(2l+1)}$ （其中 l 是频带系数）进行缩放的第 j 个系数。

给定 t 和立方体贴图的分辨率，对于每个基函数 $f_j()$ ，系数 $k'_{\cos^+ j} f_j(\mathbf{r}[t]) d\omega[t]$ 都是恒定的。这些基函数的系数可以离线预计算，并将其存储在立方体贴图中，它应当与将要渲染的动态环境贴图具有相同的分辨率。可以通过在每个颜色通道中打包一个单独的基函数系数，来减少所要使用的纹素数量。为了计算动态立方体贴图的 irradiance 系数，需要将相应基函数系数贴图中的纹素，与动态立方体贴图中的纹素相乘，并对结果进行求和。除了关于动态 irradiance 立方体贴图的相关信息，King [897]还提供了在 GPU 上实现 SH 投影的细节。

动态光源还可以被添加到现有的 SH irradiance 环境贴图中。这种合并是通过计算光线 irradiance 贡献的 SH 系数，并将其添加到现有系数中来实现的，这样做可以避免对整个 irradiance 环境贴图进行重新计算。这是一个十分简单的过程，因为点光源、圆盘光源和球形光源的系数都存在简单的解析表达式[583, 871, 1656, 1690]，对这些系数求和的效果与对 irradiance 求和的效果相同。对于那些与 z 轴对齐的光源，通常这些表示都是以球带谐波（zonal harmonic）的方式给出的，然后可以通过旋转操作，来将光源定位到任意的方向上。球带谐波的旋转是 SH 旋转（详见章节 10.3.2）的一种特殊情况，它的效率更高，只需要进行一个点积操作即可，而不需要执行一个完整的矩阵变换。对于具有复杂形状的光源，可以先将它们绘制到一个图像中，然后再将图像数值投影到 SH 基底上[1690]，从而获得这些光源所对应的系数。对于物理天空模型这个特殊情况，Habel [626]展示了 Preetham 天光在球谐函数中的一种直接扩展。

能够将常见的分析光源很容易地投影到 SH 中，这一点也是很重要的，因为通常环境光照会被用来代替远处或者较弱的光源，其中一个重要的例子就是补光灯（fill light）。在渲染中，这些光源会被放置在场景的特殊位置上，用于模拟场景中的间接光照，即被表面反弹的光线。这些补光灯通常不会计算高光效果，尤其是当这些光源

在物理尺寸上相对于着色物体而言很大，并且相对于场景中的其他光源而言相对昏暗的时候，这些因素会使得它们的高光变得更加分散，在场景中不那么明显。这种类型的光源与在现实世界中的电影光照和视频光照有相似之处，在现实世界中，会使用一些补光灯来在阴影中添加一些照明，使得阴影看起来没有那么黑。

在球谐函数空间中，也可以简单地进行相反的推导，即从投影在 SH 上的 radiance 中，提取出解析光源。在 Sloan [1656]对 SH 技术的研究中，给定一个已知轴的方向光，他展示了如何从 SH irradiance 表示中计算出光源应当具有的强度，来使得其自身与编码 irradiance 之间的误差最小化。

在他之前的工作中[1653]，Sloan 展示了如何在仅使用第一个（线性）频段系数的情况下，来选择一个接近最优的方向，该综述中还包含了一种用于提取多个方向光的方法。这项工作表明，对应光照求和而言，球谐函数是一个实用的基底。我们可以将多个光源同时投影到 SH 中，并提取出少量的方向光，这些光源可以对原来的光源集合进行较好地近似。lightcuts 框架提供了一种聚合次要光源的一般性方法[1832]。

虽然 SH 投影最常用于 irradiance 中，但是它也可以用来模拟有光泽的、视角相关的 BRDF 光照。Ramamoorthi 和 Hanrahan [1459]描述了这样一种技术，他们在立方体贴图中没有存储单一的颜色，而是存储了使用视角相关的环境贴图进行编码的球谐函数投影系数。然而在实践中，这种技术要比我们之前所介绍的预过滤环境贴图方法占据更多的存储空间。Kautz 等人 [869]推导出了一种更加经济的解决方案，该方法利用了 SH 系数的二维表格，但是这种方法只能用于相当低频的光照信息。

10.6.2 其他表示方法

立方体贴图和球谐函数是 irradiance 环境贴图最常用的表示方法，但是还有一些其他的表示方法，如图 10.45 所示。

图 10.45：irradiance 编码的各种方式。从左到右分别是：使用环境贴图和对 irradiance 进行蒙特卡罗积分计算的漫反射光照；使用环境立方体（ambient cube）编码的 irradiance；球谐

函数；球面高斯函数；H-basis（它只能表示一个半球方向，因此指向下方的法线不会被着色）。

许多 irradiance 环境贴图有两种主要颜色：顶部的天空颜色和底部的地面颜色。受到这一观察现象的启发，Parker 等人[1356]提出了一种只使用两种颜色的半球光照（hemisphere lighting）模型，该方法假设上半球会发出一个均匀的 radiance L_{sky}，下半球则会发出另一个均匀的 radiance L_{ground}，在这种情况下，irradiance 的积分是：

$$E = \begin{cases} \pi \left(\left(1 - \frac{1}{2}\sin\theta\right) L_{\text{sky}} + \frac{1}{2}\sin\theta L_{\text{ground}} \right), & \text{where } \theta < 90^\circ \\ \pi \left(\frac{1}{2}\sin\theta L_{\text{sky}} + \left(1 - \frac{1}{2}\sin\theta\right) L_{\text{ground}} \right), & \text{where } \theta \geq 90^\circ \end{cases} \tag{10.42}$$

其中方程中的 θ 是表面法线与天空半球轴之间的夹角。Baker 和 Boyd 提出了一个计算速度更快的近似方程，Taylor [1752]对其进行了描述：

$$E = \pi \left(\frac{1 + \cos\theta}{2} L_{\text{sky}} + \frac{1 - \cos\theta}{2} L_{\text{ground}} \right) \tag{10.43}$$

方程 10.43 在天空和地面之间进行了线性插值，使用 $(\cos\theta + 1)/2$ 作为插值因子。其中 $\cos\theta$ 通常可以通过向量点积来进行快速计算，并且当天空半球轴与某个基本轴（例如 y 轴或者 z 轴）重合的时候，$\cos\theta$ 实际上根本不需要计算，因为它就等于法线 $\mathbf{n}$ 的某个世界空间的坐标分量。这个近似结果与原结果相当接近，而且速度要快得多，因此对于大多数应用而言，它比原本完整的表达式更加好用。

Forsyth [487]提出了一种廉价且灵活的光照模型，被称为三重光源（trilight），其中包括了定向光照、双向光照、半球光照，还有作为特殊情况的环绕光照（wrap lighting）。

Valve 最初使用了环境立方体（章节 10.3.1）来表示 irradiance。一般来说，我们在章节 10.3 中所看到的所有球面函数表示方法，都可以用于预计算的 irradiance。对于 irradiance 函数所表示的低频信号，我们知道 SH 是一个很好的近似。我们倾向于使用特殊的方法来简化或者使用比球谐函数更少的存储空间。

如果我们想计算遮挡和其他全局光照效果，或者想整合光泽反射，就需要能够表达高频信号的复杂的表示方法（章节 10.1.1）。将所有的相互作用考虑在内，并对光照结果进行预计算，这个思想被称为预计算 radiance 传输（precomputed radiance transport，PRT），我们将在章节 11.5.3 中进行详细讨论。能够捕获光泽照明的高频

信号的光照方法，也称为全频率（all-frequency）光照，在这种情况下经常会使用小波（wavelet）表示来作为压缩环境贴图的手段[1059]，并以一种与之前所介绍的球谐函数相类似的方式，来设计高效的运算方法。Ng 等人[1269, 1270]展示了使用 Haar 小波来对 irradiance 环境映射进行推广，从而对自阴影效应进行建模。他们在小波基底中存储环境贴图以及在物体表面上变化的 shadowing 函数。这种表示十分值得注意，因为它相当于对环境立方体贴图进行了变换，在每个立方体表面上执行二维的小波投影，因此它也可以被视为一种立方体贴图的压缩技术。

10.7 误差来源

为了正确地进行着色计算，我们必须在非定点光源上计算积分。在实践中，这一要求意味着，我们可以根据所计算光源的属性，采用许多不同的技术。实时引擎通常会以解析方式来对一些重要的光源进行建模，对光源照亮区域计算近似积分，并通过阴影贴图来计算遮挡信息。而所有其他光源（远距离光源、天光、补光和表面反弹光源），通常都会使用环境立方体贴图来表示高光分量，使用球面基底来表示漫反射 irradiance。

使用这种混合照明技术意味着，我们永远不会直接使用给定的 BRDF 模型，而是使用具有不同程度误差的近似值。有时候 BRDF 近似是显式的，因为我们对中间模型进行了拟合，从而计算光照积分，LTC 就是这样一个例子。但是在其他时候，我们所构建的近似值只有在某些条件（通常很少）下，对于给定的 BRDF 才是精确的，在大多数情况下都会出现误差，预过滤立方体贴图就属于这一类。

在开发实时着色模型的时候，需要考虑一个重要因素，即确保不同形式光照之间的差异不明显。从视觉上看，来自不同表示的连贯光照效果，甚至可能要比每种表示的绝对近似误差更加重要。

遮挡对于真实感渲染而言也很重要，因为在没有光线的地方出现“漏光”现象，在视觉上通常要比在应当有光的地方没有光更加明显。大多数面光源表示对于阴影而言都是很难处理的，在如今，即使使用了各种阴影“软化”技术（章节 7.6），也没有任何一种实时阴影技术能够准确考虑光源的形状。我们计算了一个标量因子，当物体投射阴影的时候，我们将其相乘以减少给定光源的贡献值，这实际上是错误的，在使用 BRDF 进行积分的时候，我们应当考虑到这种遮挡所带来的影响。这对于环境光照而言尤其困难，因为我们没有一个明确的、主要的光源方向，因此无法使用任何精确光源的阴影技术。

图 10.46：电影行业中的人工照明。

即使我们已经看到了一些相当先进的光照模型，但是请记住，它们都不是对现实世界光照来源的精确表示，例如：在环境光照中，我们会假设 radiance 来自于无限远处，但是现实中这是不可能的。而且，我们看到的所有解析光源都基于了这样一个更加强势的假设：光源表面上的每个点，都会均匀地向半球方向上发出 radiance。但是在实践中，这种假设也是一种可能的误差来源，因为真实世界中的光源往往具有很强的方向性。在摄影照明和电影照明中，被称为镜头挡光板（gobo）、剪影遮光片（cuculori）或者 cookies 的特制蒙版和滤镜，经常被用来实现某些艺术效果。如图 10.46 所示，摄影师 Gregory Crewdson 所拍摄的照片，展示了电影光照的复杂情况。为了限制光源的照明角度，同时保持大面积的光线发射，可以在大型发光面板（即所谓的柔光箱）前面添加栅格化的黑色屏蔽材料，这种装置被称作 honeycombs（蜂窝）。光源的外壳处也会使用复杂的镜面和反光板，例如：室内照明、汽车前灯和手电筒等。如图 10.47 所示，这些光学系统会远离其物理中心的地方，生成一个或者多个发射光线的虚拟发射器，在进行光源衰减计算的时候，也应当考虑到这个偏移。

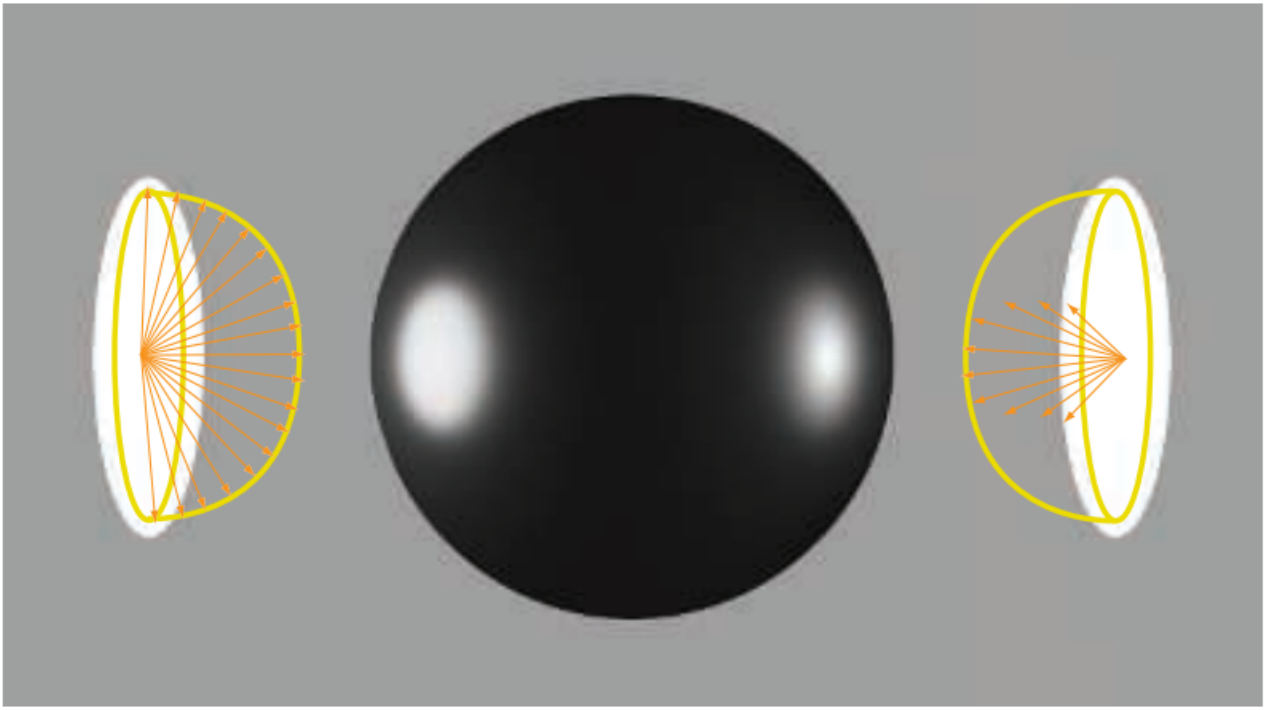


图 10.47：相同的圆盘光源有着两种不同的发射方式。左边：圆盘上的每一点都均匀地向外半球发射光线。右边：radiance 集中在圆盘法线周围的波瓣上。

请注意，我们始终应当在一个面向感知的、面向结果的框架中对这些误差进行评估，除非我们的目标是进行预测渲染，即精确可靠地模拟真实世界的表面外观。在艺术家手中，即使某些简化并不真实，但是仍然可以生成有用并且富有表现力的光照效果。当这些基于物理的模型能够使艺术家更容易创建视觉可信的图像时，那么它们就是有用的，因为我们的最终目标是渲染出令人信服的图像，而不是这些物理模型本身。

补充阅读和资源

Hunter [791]所撰写的《Light Science and Magic: An Introduction to Photographic Lighting》一书，是一个理解现实世界摄影光源良好参考。对于电影照明而言，《Set Lighting Technician's Handbook: Film Lighting Equipment, Practice, and Electrical Distribution》[188]是一个很好的介绍。

Debevec 在基于图像的照明领域做出了开创性工作，这对于任何需要从现实场景中捕捉环境贴图的人而言都十分具有吸引力。这项工作的大部分内容都涵盖在 SIGGRAPH 2003 的课程中[333]，以及由 Reinhard 等人[1479]编写《High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting》一书中。

光源描述文件（light profile）是一个可以帮助模拟光源的资源。照明工程学会（Illuminating Engineering Society，IES）出版了一本用于照明测量的手册和文件格式标准[960, 961]。这种格式的数据通常可以从相关设备制造商那里获得。IES 标准仅限于对光源的发射角度（angular emission profile）进行了描述，它没有完全模拟由于光学系统所带来的衰减影响，也没有对光源表面区域上的发射情况进行模拟。

Szirmay-Kalos [1732]关于镜面效果的最新报告中，包括了许多有关环境映射技术的参考文献。

Chapter 11 Global Illumination 全局光照

> Jeremy Birn——“If it looks like computer graphics，it is not good computer graphics.”

> 杰里米·伯恩——“如果它看起来像是计算机图形学生成的，那它就不是一个好的计算机图形学。”（皮克斯动画公司的光照技术总监）

渲染过程最终计算的是 radiance，到目前为止，我们一直在使用反射方程（reflectance equation）来其进行计算：

$$L_o(\mathbf{p},\mathbf{v}) = \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{p},\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \tag{11.1}$$

其中 $L_o(\mathbf{p},\mathbf{v})$ 是表面位置 $\mathbf{p}$ 在观察方向 $\mathbf{v}$ 上的出射 radiance； Ω 是表面位置 $\mathbf{p}$ 的上半球范围； $f(\mathbf{l},\mathbf{v})$ 是观察方向 $\mathbf{v}$ 和当前光线入射方向 $\mathbf{l}$ 上的 BRDF； $L_i(\mathbf{p},\mathbf{l})$ 是从光线方向 $\mathbf{l}$ 到达表面位置 $\mathbf{p}$ 的入射 radiance； $(\mathbf{n}\cdot\mathbf{l})^+$ 是光线方向 $\mathbf{l}$ 和表面法线 $\mathbf{n}$ 之间的点积，并将负数结果 clamp 到 0，即将来自表面下方的光线过滤掉。

11.1 渲染方程

反射方程是完整渲染方程的一种特殊情况，它由 Kajiya 在 1986 年提出[846]。渲染方程具有各种不同的表达形式，我们将使用以下这个版本：

$$L_o(\mathbf{p},\mathbf{v}) = L_e(\mathbf{p},\mathbf{v}) + \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_o(r(\mathbf{p},\mathbf{l}),-\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \tag{11.2}$$

其中多出来的一项为 $L_e(\mathbf{p},\mathbf{v})$ ，它表示了从表面位置 $\mathbf{p}$ 向观察方向 $\mathbf{v}$ 发射的 radiance，用于描述自发光表面的出射 radiance。被积函数中有一项做了如下替换：

$$L_i(\mathbf{p},\mathbf{l}) = L_o(r(\mathbf{p},\mathbf{l}),-\mathbf{l}) \tag{11.3}$$

这一项意味着，从方向 $\mathbf{l}$ 进入表面位置 $\mathbf{p}$ 的入射 radiance，等于另一个表面位置向相反方向 $-\mathbf{l}$ 的出射 radiance。在这种情况下，这里的“另一个表面位置”由光线投射函数（ray casting function） $r(\mathbf{p},\mathbf{l})$ 所定义的，这个函数会从表面位置 $\mathbf{p}$ 向方向 $\mathbf{l}$ 上发射一条光线，并返回所击中的第一个表面位置，如图 11.1 所示。

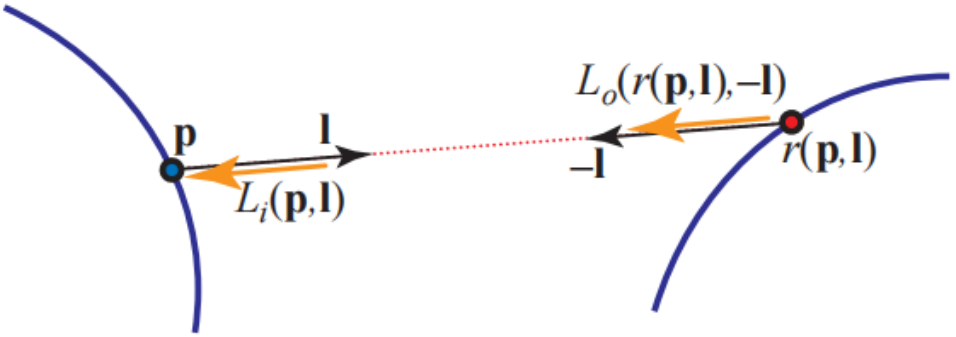


图 11.1：图中展示了表面着色点 $\mathbf{p}$，光线方向 $\mathbf{l}$，光线投射函数 $r(\mathbf{p},\mathbf{l})$，着色点 $\mathbf{p}$ 的入射 radiance $L_i(\mathbf{p},\mathbf{l})$，以及表面点 $r(\mathbf{p},\mathbf{l})$ 的出射 radiance $L_o(r(\mathbf{p},\mathbf{l}),-\mathbf{l})$。

渲染方程的含义很简单。为了对表面位置 $\mathbf{p}$ 进行渲染，我们需要知道在观察方向 $\mathbf{v}$ 上，离开表面位置 $\mathbf{p}$ 的出射 radiance L_o，它等于该点自身发射的 radiance L_e，再加上反射出的 radiance。有关光源发射和反射率的内容，在前面几章我们已经讨论过了。甚至这里的光线投射操作好像看起来也不是那么陌生，例如：z-buffer 实际上就计算了从相机投射到场景中的光线。

这里我们所遇到的唯一的新项是 $L_o(r(\mathbf{p},\mathbf{l}),-\mathbf{l})$，这一项明确指出，入射到某一点上的 radiance，一定是从另一点发出的。不幸的是，这是一个递归项，也就是说，如果我们想要计算点 $r(\mathbf{p},\mathbf{l})$ 在方向上 $-\mathbf{l}$ 上的出射 radiance，那么我们首先还要知道来自表面位置 $r\left(r(\mathbf{p},\mathbf{l}),\mathbf{l}'\right)$ 的出射 radiance，接下来还需要计算来自表面位置 $r\left(r\left(r(\mathbf{p},\mathbf{l}),\mathbf{l}'\right),\mathbf{l}''\right)$ 的出射 radiance，直到无穷。令人十分惊讶的是，如此复杂的计算量，现实世界居然可以对其进行实时计算。

我们凭借直觉可以知道，光源照亮了一个场景，它所发出的光子（photon）在场景中四处反弹，每次与表面发生碰撞的时候，都会以各种方式被吸收、反射或者折射。渲染方程十分重要，因为它在一个简单的方程中总结了所有可能的光线路径。

渲染方程有一个重要的属性，即它与所发射出的光线呈线性关系。如果我们使光源的强度翻倍，那么最终的着色结果也会加倍变亮。同时，材质对于每种光源的响应也是相互独立的，也就是说，一种光源的存在并不会影响另一种光源与材质之间的相互作用。

在实时渲染中，只使用局部光照模型也是很常见的，我们只需要对可见点的表面数据进行光照计算即可，而这正是 GPU 最擅长的。传入 GPU 的各种图元被独立处理和光栅化，然后它们就会被丢弃，我们在点 **b** 执行光照计算时，无法访问点 **a** 的光照计算结果。诸如透明、反射和阴影效果，都是全局光照算法的范畴，它们利用了来自其他物体的信息，而不仅仅是被光源所照亮的物体。这些效果大大增强了渲染图像的真实感，并提供了视觉暗示（cues）来帮助观察者理解空间中的位置关系。同时，这些效果模拟起来也十分复杂，可能需要进行预计算或者渲染多个 pass 来计算一些必须的中间信息。

有一种思考光照问题的方法，即通过光子的传播路径来理解光照。在局部光照模型中，光子从光源出发，传播到表面上（忽略中间的物体），然后到达眼睛。阴影算法考虑了这些中间物体的直接遮挡效果。环境贴图可以捕捉从远处光源到达物体表面的光线，然后将其应用到局部的光泽物体上，这些物体会以镜面反射的方式，将这些光线反射到眼睛中。irradiance 贴图还可以捕捉到光源对遥远物体的影响，并在半球范围的方向上进行积分，被这些物体所反射的光线会进行加权求和，从而计算出一个表面的光照效果，最终被眼睛所看到。

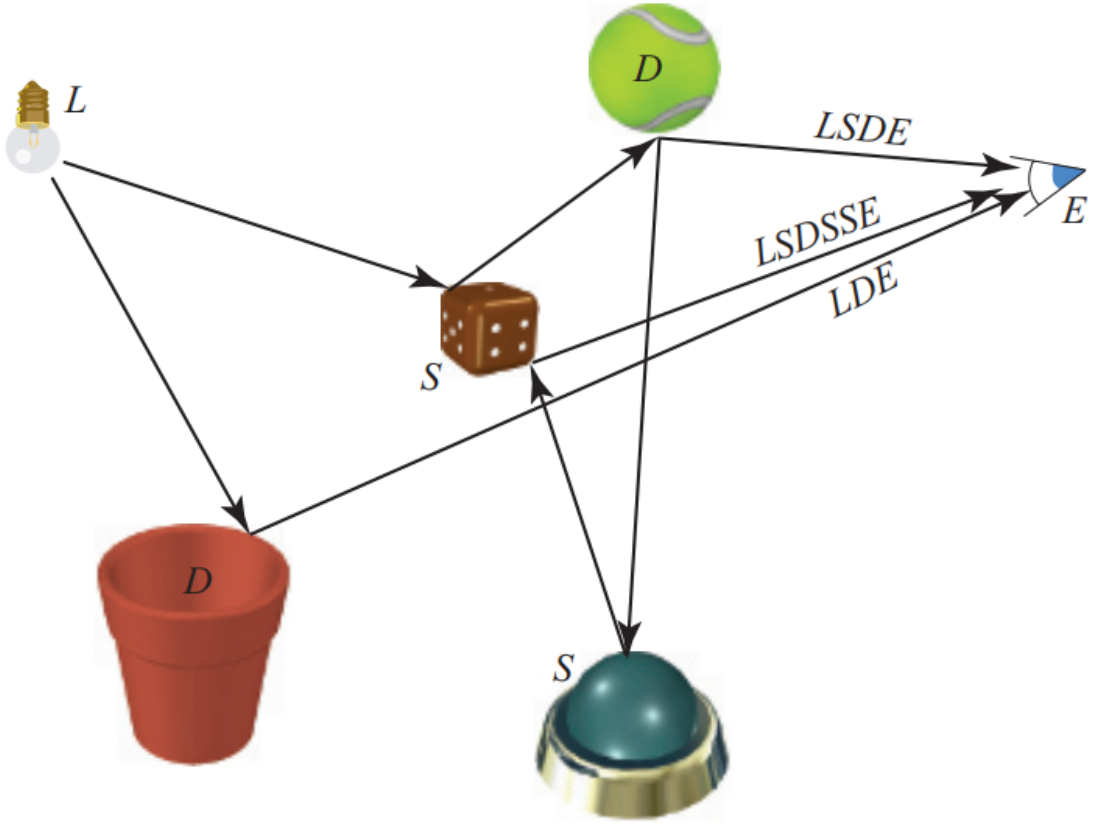


图 11.2：图中展示了一些路径及其到达眼睛时的等效符号。注意，图中展示了两条从网球开始的连续路径，分别是 LSDE 和 LSDSSE。

以一种更加正式的方式来思考光线传输路径的不同类型和不同组合，有助于理解现有的各种算法。Heckbert [693]提出了一个符号方案，它用于描述由某种技术所模拟的光线路径。光子从光源（L）到眼睛（E）的每次相互作用，都可以标记为漫反射（D）或者镜面反射（S），还可以通过添加其他表面类型来进一步分类，例如“有光泽的（glossy）”，它代表了有光泽，但是又不像镜子的表面，如图 11.2 所示。可以使用正则表达式来简单地概括这些算法，从而展示它们所模拟的交互类型。表 11.1 对基本符号进行了总结。

Operator	Description	Example	Explanation
$*$	zero or more	$S*$	zero or more specular bounces
$+$	one or more	$D+$	one or more diffuse bounces
$?$	zero or one	$S?$	zero or one specular bounces
$\|$	either/or	$D\|SS$	either a diffuse or two specular bounces
$()$	group	$(D\|S)*$	zero or more of diffuse or specular

表 11.1：正则表达式符号。

从光源出发的光子可以通过各种路径最终到眼睛。最简单的路径是 LE，光源被眼睛直接看到。一个基本的 z-buffer 是 $L(D|S)E$，或者写成其等价形式 $LDE|LSE$。光子离开光源，到达一个漫反射表面或者一个镜面，然后再到达眼睛。请注意，在一个基础的渲染系统中，点光源没有对应的物理表示，它不会被眼睛直接观察到。对于一个具有几何形状的光源而言，将会产生这样一个路径 $L(D|S)?E$，除了照射到表面之外，从光源发出的光线也可以直接进入眼睛。

如果将环境映射添加到渲染器中，那么这个表达式就不再那么简单了。虽然 Heckbert 的表示法是从光源出发最终到达眼睛，但是对于渲染而言，从相反方向来构建表达式通常要更加容易。眼睛将首先看到一个镜面或者一个漫反射表面，即 $(S|D)E$，如果这个表面是一个镜面，那么它也可以选择反射到一个环境贴图中的镜面，或者是一个漫反射表面上。因此，存在一条额外的可能路径：$((S|D)?S|D)E$。同时再加上眼睛直接看到光源的路径，那么这个表达式最终会变为：$L((S|D)?S|D)?E$。

可以将这个表达式展开：$LE|LSE|LDE|LSSE|LDSE$，它代表了所有可能存在的路径，或者简写为：$L(D|S)S?E$。每一种表示方法在理解关系和限制方面都有各自的优势。这种符号表示法的部分用途是表达算法的效果，并能够以此为基础进行

构建，例如： $L(S|D)$ 是生成环境贴图时所编码的内容，而 SE 则代表了随后访问该贴图的过程。

渲染方程本身也可以用简单的表达式 $L(D|S) * E$ 来进行概括，即来自光源的光子在到达眼睛之前，可以与 0 到几乎无限数量的漫反射表面或者镜面发生相互作用。

对于全局光照的研究，主要集中在计算光线在这些路径上传播的方法。当将其应用于实时渲染时，我们通常愿意牺牲一些质量或者正确性，来换取更快的计算速度。最常见的两种策略就是简化和预计算。例如：我们可以假设所有反射到眼睛中的光线都是漫反射的，这种简化在某些环境和场景中表现很好。我们还可以离线环境中，对一些物体之间效果的相关信息进行预计算，例如生成记录表面光照水平的纹理，然后在运行过程中，根据这些存储的信息进行一些基本计算，从而获得全局光照效果。本章节将展示如何使用这些策略，来实时实现各种全局光照效果。

11.2 通用全局光照

我们在前面几章中，着重介绍了求解反射方程的各种方法。我们假设入射 radiance L_i 具有一定的分布，并分析了它是如何影响着色计算的。而在本章节中，我们将介绍用于求解完整渲染方程的算法。二者之间的区别在于，前者忽略了 radiance 的来源，它假设是直接给出的；而后者则明确地说明了这一点：到达某一点的 radiance 是从其他点发射或者反射而来的。

图 11.3：路径追踪可以生成照片级逼真的图像，但是其计算成本较高。上面图像中的每个像素都使用了超过 2000 条路径（2000spp），每个路径长达 64 段（深度，即反弹次数）。它花费了两个多小时来进行渲染，但是仍然会表现出一些轻微的噪声。[149]

能够求解完整渲染方程的算法，可以生成令人惊叹的、照片级逼真的图像，如图 11.3 所示。然而对于实时应用来说，这些方法的计算成本都太高了，那么为什么我们还要讨论它们呢？第一个原因是：在静态或者部分静态的场景中，这样的算法可以在预处理阶段执行，并将计算结果存储下来，以供稍后在实时渲染期间使用。这在游戏中是一种十分常见的方法，稍后我们将对这类系统的不同方面进行讨论。

第二个原因是：全局光照算法都建立在严格的理论基础上，它们是直接从渲染方程中推导出来的，它们所做的任何近似都是经过仔细分析的。在设计实时解决方案的时候，可以且应该应用类似的推理思路；即使我们走了某些捷径，使用了一些技巧，但是我们也应当知道这么做的后果是什么，什么才应该是正确的方法。随着图形硬件变得越发强大，我们将能够做出更少的妥协和近似，并且能够创建出更加接近正确物理结果的实时渲染图像。

求解渲染方程的两种常用方法是有限元法（finite element）和蒙特卡罗法（Monte Carlo）。其中辐射度算法（radiosity）基于了第一种方法，而不同形式的光线追踪算法（ray tracing）则使用了第二种方法。在这两种不同思路的算法中，光线追踪要更加流行。这主要是因为它可以在同一个算法框架内，对一般的光线传输效果进行有效处理，包括体积散射等效果。而且光线追踪算法也更加容易扩展和并行化。

我们将简要介绍这两种方法，有兴趣的读者还应该参考其他优秀的书籍，它们涵盖了在非实时情况下求解渲染方程的细节[400, 1413]。

11.2.1 辐射度

辐射度算法（Radiosity）[566]是第一种用于模拟漫反射表面之间光线反弹的计算机图形技术，其名字来源于该算法所计算的物理量。在经典的算法形式中，辐射度算法可以计算相互反射以及面光源所产生软阴影。辐射度算法的基本思想相对简单，并且已经有了完整的书籍对这个算法进行介绍[76, 275, 1642]。光线会在环境中发生弹射，当我们打开一盏灯时，房间内的照明会很快达到平衡，在这种稳定状态下，每个表面都可以被看作是一个光源。基本的辐射度算法作出了一种简化的假设，即所有场景中的间接光都来自于漫反射表面。对于具有抛光大理石地板或者墙上有巨大镜面的场景而言，这个假设是不成立的，但是对于现实中的许多建筑而言，这是一个相对合

理的近似。辐射度算法可以对无限数量的有效漫反射进行追踪。如果使用本章节开头所介绍的符号表示法，那么可以将它的光线传输路径写为是 $LD*E$ 。

辐射度算法假设每个物体表面都由一定数量的面片（patch）组成。对于每个较小的区域（面片），辐射度算法都会计算一个平均辐射度值（radiosity value），因此这些面片的尺寸需要足够小，才能够捕捉所有的照明细节（例如阴影边缘）。这些面片不需要和底层表面的三角形一一匹配，甚至面片的大小尺寸也可以不一样。

从渲染方程出发，我们可以推导出第 i 个面片的辐射度为：

$$B_i = B_i^e + \rho_{ss} \sum_j F_{ij} B_j \tag{11.4}$$

其中 B_i 代表了面片 i 的辐射度； B_i^e 为面片 i 的辐射出度（radiant exitance），即面片 i 所发出的辐射度； ρ_{ss} 是次表面反照率（详见章节 9.3）。只有光源的辐射出度才不为 0。 F_{ij} 是面片 i 和面片 j 之间的形状因子（form factor），这个形状因子的定义为：

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} V(\mathbf{i},\mathbf{j}) \frac{\cos\theta_i \cos\theta_j}{\pi d_{ij}^2} da_i da_j \tag{11.5}$$

其中 A_i 是面片 i 的面积； $V(\mathbf{i},\mathbf{j})$ 是点 i 与点 j 之间的可见性函数，如果它们之间没有物体遮挡光线，则该项为 1，否则为 0。角度值 θ_i 和 θ_j 分别是两个面片的法线，与点 i 和点 j 的之间连线的夹角。最后， d_{ij} 是点 i 和点 j 的之间距离。如图 11.4 所示。

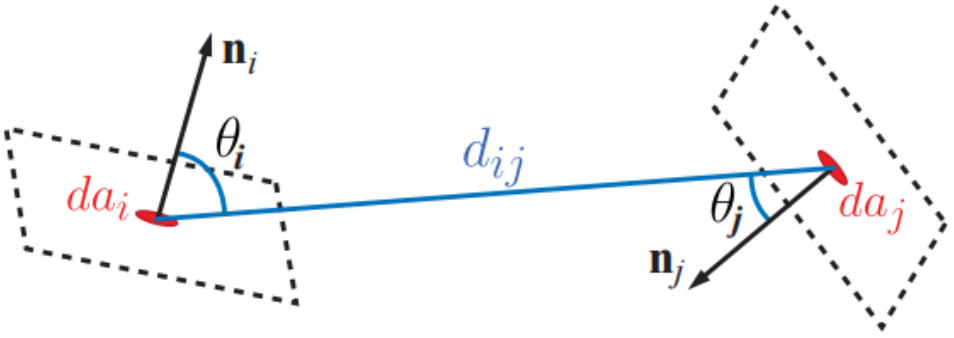


图 11.4：两个表面点之间的形状因子。

形状因子是一个纯粹的几何项，它描述了离开面片 i 的均匀漫反射辐射能量，有多少能够入射到面片 j 上[399]。两个面片的面积、距离、相对朝向、以及它们之间存在的任何表面，都会对它们的形状因子产生影响。想象现在有一个面片，假设它代表了

一个计算机显示器；房间里的其他每个面片，都会直接接收到这个显示器所发出的部分光线。位于显示器背面或者看不见显示器的表面，它们无法接收到显示器发射出来的光线，对于这些表面而言，这个比例因子为零。这些比例因子的和为 1。辐射度算法的一个重要部分，就是准确确定场景中面片之间的形状因子。

在计算出形状因子之后，所有面片的方程（方程 11.4）会被组合为一个的线性系统（一个由线性方程组成的数学模型）。然后对这个线性系统进行求解，从而得到每个面片的辐射度值。随着面片数量的不断增加，会导致计算复杂度变得很高，求解这样一个矩阵的成本是相当大的。

由于这个算法的扩展性很差，并且存在一些其他的限制，因此经典的辐射度算法很少用于作为光照解决方案。然而，在现代实时全局光照系统中，预先计算形状因子，并在运行过程中使用它们来执行某种形式的光线传播，这个思想仍然很流行。我们将在章节 11.5.3 中来讨论这些方法。

11.2.2 光线追踪

光线投射（ray casting）是指从某个表面位置上发射一根光线，从而确定在特定方向上存在哪些物体的过程。光线追踪（ray tracing）使用光线来确定不同场景元素之间的光线传输。在其最基本的形式中，光线会从相机所在的位置出发，穿过像素网格进入到场景中。对于每条光线，都找到第一个与光线相交的物体。然后，通过从交点向各个光源发射光线，并查找交点与光源之间是否存在遮挡物体，来检查这个交点是否位于阴影中。不透明的物体会遮挡光线，透明的物体会减弱光线。还可以在交点处发射其他光线，如果表面具有光泽，则可以在反射方向上生成光线。这条光线会获取第一个相交物体的颜色，然后再对相交点进行阴影测试。也可以在透明固体的折射方向上生成光线，然后再进行递归计算。光线追踪的基本机制非常简单，以致于最基础的光线追踪渲染器的代码，甚至可以写在一张名片的背面[696]。

经典的光线追踪算法只能提供有限的效果：尖锐的反射和折射，以及硬阴影。然而，光线追踪的基本思想可以用于求解完整的渲染方程。Kajiya 认识到[846]，可以利用光线的发射机制以及评估它们所携带的光线能量，从而计算方程 11.2 中的积分项。方程 11.2 是递归的，这意味着对于每条光线，我们需要在不同的位置重新计算积分。幸运的是，我们已经有了处理这个问题的坚实数学基础。在曼哈顿计划（Manhattan Project）期间，为物理实验而开发的蒙特卡罗（Monte Carlo）方法就是专门为处理这类问题而设计的。我们并不是直接按照积分规则来计算每个着色点上的积分值，而是通过在积分域上采样一定数量的随机点，从而获得被积函数的具体数值。然后我们使用这些被积函数值来计算积分的估计值，采样点越多，最终的精度就越高。这种方

法最重要的性质是，我们只需要对被积函数上的点进行求值计算，给定足够的时间，我们就可以以任意的精度来计算积分。在渲染领域中，这正是光线追踪所能提供的，当我们投射光线的时候，就相当于对方程 11.2 中的被积函数进行了点采样。尽管在交点处我们还需要递归计算另一个积分，但是我们不需要计算它的最终精确值，我们可以再次对这个积分进行点采样。当光线在场景中反弹的时候，就形象的建立了一条路径（path），我们沿每条光线路径，对被积函数进行一次计算，这个过程被称为路径追踪（path tracing），如图 11.5 所示。

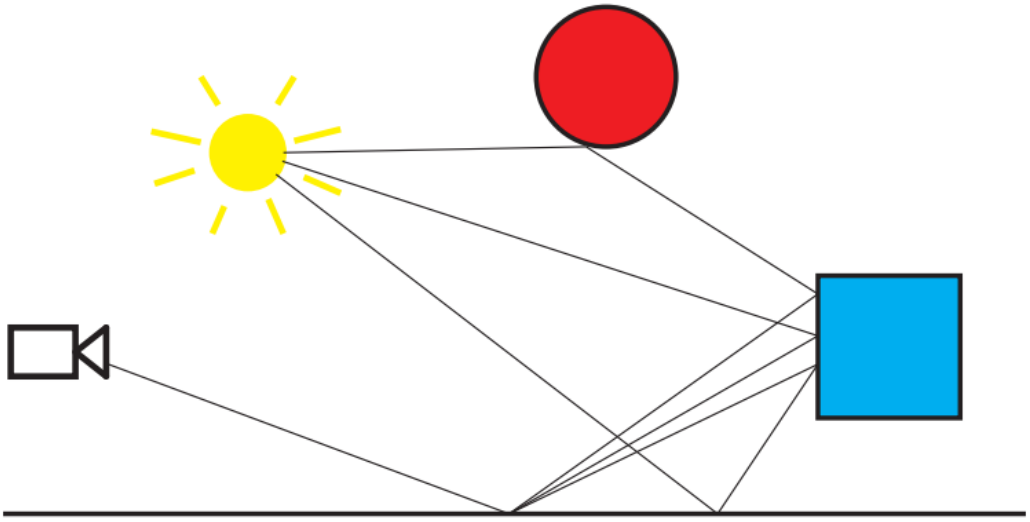


图 11.5：路径追踪算法所生成的示例路径。图中所展示的三条路径通过了成像平面中的同一个像素，并用于估计它的亮度值。图片底部的地板具有一定的光泽，因此会在一个较小的立体角范围内反射光线。蓝色盒子和红色球体都是漫反射表面，因此光线在交点处，会绕法线周围进行均匀地散射。

对路径进行追踪是一个非常强大的概念。这些路径可以用于渲染光泽材质或者漫反射材质。使用它们，我们还可以生成软阴影、渲染透明物体以及焦散效果。通过对路径追踪进行扩展，我们还可以对体积内的点进行采样，而不仅仅是对物体表面进行采样，这样我们就可以处理雾和次表面散射等效果。

路径追踪的唯一缺点是，想要实现高视觉保真度的画面，所需要的计算复杂度是很高的。对于电影级的图像，我们可能需要对数十亿条路径进行追踪，因为我们计算的只是积分的估计值，而不是真实值。如果使用的路径太少，那么这种近似将会是不精确的，在一些特殊情况下会及其不精确，从而产生大量噪声。此外，即使是两个相邻的点，最终的着色结果也可能会差异很大，这与我们的期望不太相符，我们总是希望相邻点具有相似的光照结果。我们将这样的结果称为具有高方差（high variance），从视觉上看，方差会体现为图像中的噪声（如图 11.6 所示）。现在已经有了很多方法，可以在不增加额外追踪路径的前提下，消除或者减弱这种噪声所带来的影响。其中一

种流行的技术是重要性采样（importance sampling），这个方法的思路是，通过向光线来源的主要方向发射更多的光线，从而大大降低方差。

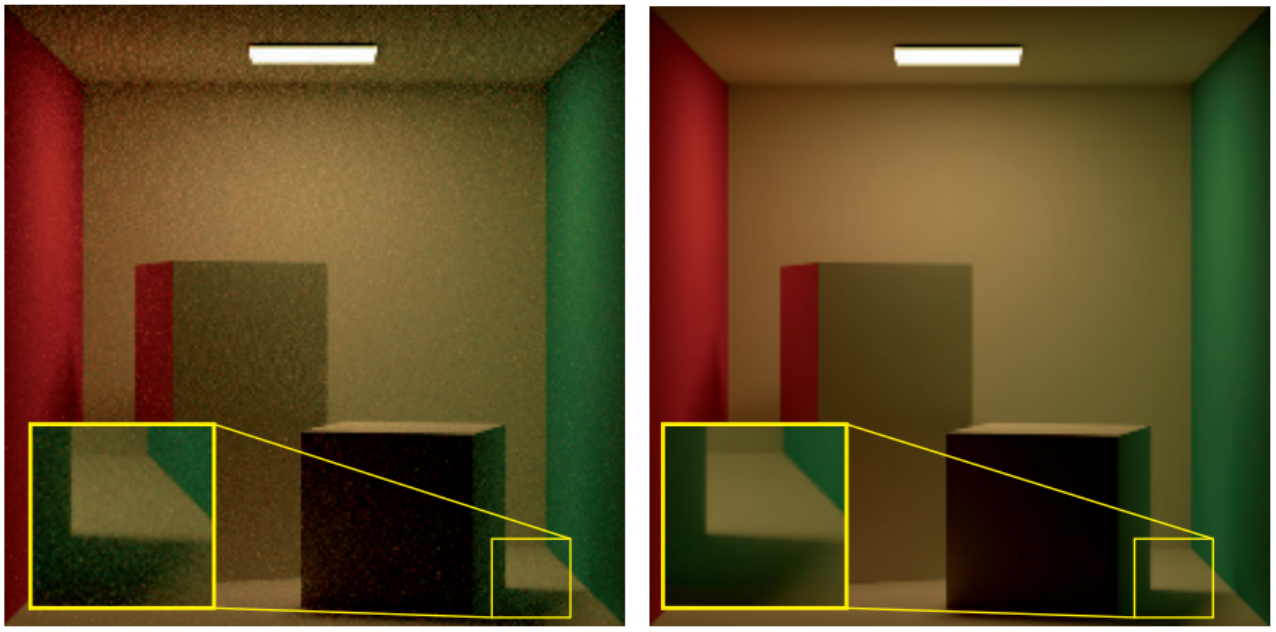


图 11.6：使用蒙特卡罗路径追踪时，由于样本数量不足而产生的噪声。左侧图像为每像素 8 个路径（8 spp），右侧图像为每像素 1024 个路径（1024 spp）。[149]

许多已经出版了的论文和书籍对路径追踪及其相关方法进行了详细讨论。Pharr 等人[846]为现代离线光线追踪技术提供了一个很好的介绍。Veach [846]为光线传输算法的现代推理奠定了数学基础。我们将在本章最后的章节 11.7 中，讨论交互式的光线追踪和路径追踪。

11.3 环境光遮蔽

上一小节中所介绍的通用全局光照算法，它们的计算成本都很高。虽然它们可以产生各种复杂的效果，但是生成一幅图像往往需要好几个小时。我们将首先介绍一些最简单的，但是在视觉上很有说服力的方法，并在本章节逐步探索实时替代方案，逐步构建更加复杂的效果。

一种基本的全局光照效果是环境光遮蔽（ambient occlusion，AO）。这项技术是在 21 世纪初，由工业光魔的 Landis [974]所开发的，当时是用于提高电影《珍珠港》中，由计算机生成的飞机的环境光照质量。尽管这种效应的物理基础进行了相当程度的简化，但是最终的结果看起来却令人惊讶地可信。当光照缺乏方向变化，无法展现物体细节时，这种廉价方法可以提供对于物体形状的视觉暗示。

11.3.1 环境光遮蔽理论

环境光遮蔽的理论背景可以直接从反射方程中推导出来。这里为了简单起见，我们将首先关注 Lambertian 表面，该表面的出射 radiance L_o 与表面 irradiance E 成正比。irradiance 是入射 radiance 的余弦加权积分，一般来说，它取决于表面位置 $\mathbf{p}$ 和表面法线 $\mathbf{n}$ 。同样，为了简单起见，我们将假设来自所有方向 $\mathbf{l}$ 上的入射 radiance 都是恒定的，即 $L_i(\mathbf{l}) = L_A$ 。基于上述假设，此时计算 irradiance 的方程如下所示：

$$E(\mathbf{p}, \mathbf{n}) = \int_{\mathbf{l} \in \Omega} L_A (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} = \pi L_A \tag{11.6}$$

在这里，我们将对半球 Ω 的所有可能入射方向进行积分。在恒定均匀光照的假设下，irradiance（以及由此产生的出射 radiance）与表面位置和表面法线无关，并且在整个物体上都是恒定的。这会生成一个平坦均匀的外观。

方程 11.6 并没有考虑任何的可见性。着色点半球范围内的某些方向，可能会被自身物体的其他部分或者是场景中的其他物体所遮挡。在这些方向上将会具有不同的入射 radiance，而不是恒定的 L_A 。为了简单起见，我们假设来自这些遮挡方向上的入射 radiance 为零。虽然这个假设忽略了场景中可能会被其他物体反弹，并最终从这些遮挡方向到达点 $\mathbf{p}$ 的光线，但是它极大地简化了推理过程。基于上述假设，我们可以得到以下方程，它由 Cook 和 Torrance 首次提出[285, 286]：

$$E(\mathbf{p}, \mathbf{n}) = L_A \int_{\mathbf{l} \in \Omega} v(\mathbf{p}, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{11.7}$$

其中 $v(\mathbf{p}, \mathbf{l})$ 是一个可见性函数，如果从点 $\mathbf{p}$ 向方向 $\mathbf{l}$ 投射的光线会被物体遮挡，则该函数值为 0，反之为 1。

将可见性函数进行余弦加权积分，然后再进行归一化，最终的结果被称为环境遮挡系数：

$$k_A(\mathbf{p}) = \frac{1}{\pi} \int_{\mathbf{l} \in \Omega} v(\mathbf{p}, \mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}. \tag{11.8}$$

这个系数代表了未被遮挡的半球的余弦加权百分比，它的范围位于 $[0, 1]$ 内，对于完全被遮挡的着色点，它的值为 0；对于没有任何遮挡的着色点，它的值为 1。值得注意的是，凸面（convex）物体，例如球体或者立方体，不会对自身造成遮挡。因此当

场景中不存在其他物体时，凸面物体的环境遮挡值将为 1。如果物体表面存在凹陷区域，则这些区域的遮挡值将小于 1。

在定义 k_A 之后，考虑遮挡情况的环境 irradiance 方程为：

$$E(\mathbf{p}, \mathbf{n}) = k_A(\mathbf{p})\pi L_A \tag{11.9}$$

注意，在方程 11.9 中，irradiance 会随着表面位置的变化而变化，因为 k_A 确实是由表面位置所决定的，这样所得到的结果会更加真实，如图 11.7 所示。由于尖锐折痕处的 k_A 值较低，因此这里的表面位置会显得较暗。

图 11.7：仅使用恒定环境光照（左）和使用环境光遮蔽（右）渲染的物体。即使光照是恒定均匀的，环境光遮蔽也可以表现出物体的细节。[149]

比较图 11.8 中的表面位置 $\mathbf{p}_0$ 和 $\mathbf{p}_1$，可以发现表面朝向也会对 k_A 有影响，因为可见性函数 $v(\mathbf{p}, \mathbf{l})$ 在积分的时候会被余弦因子加权。比较图 11.8 左侧的表面位置 $\mathbf{p}_1$ 和 $\mathbf{p}_2$，虽然二者具有一个大小相同的未遮挡立体角，但是表面位置 $\mathbf{p}_1$ 的大部分未遮挡区域都位于其表面法线附近，该位置上的余弦因子相对较大，从箭头的亮度就可以看出。相比之下，表面位置 $\mathbf{p}_2$ 的大部分未遮挡区域都位于其表面法线的一侧，因此该位置的余弦因子相对较小，也就是说，在 $\mathbf{p}_2$ 处的 k_A 较低。从这里开始，为简单起见，我们将不再显式说明遮挡系数对表面位置 $\mathbf{p}$ 的依赖。

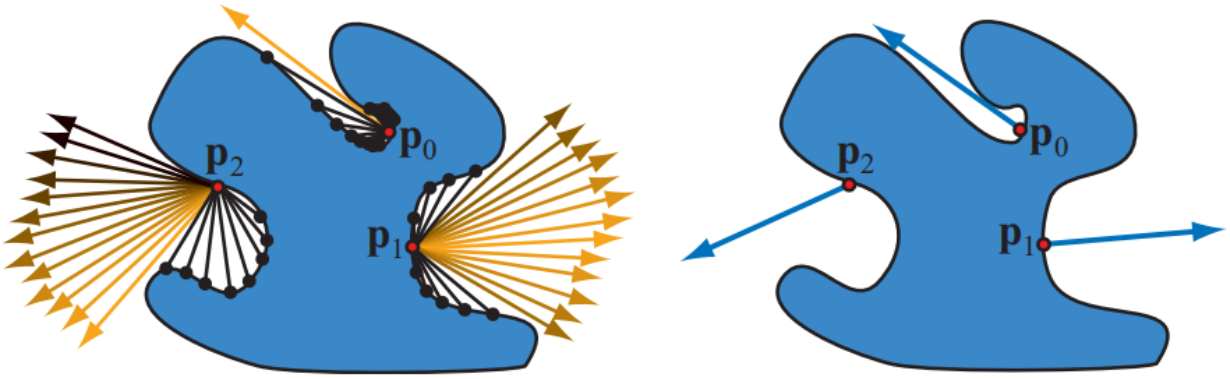


图 11.8：环境光照下的物体，图中展示了三个点，分别是 $\mathbf{p}_0$ 、 $\mathbf{p}_1$ 和 $\mathbf{p}_2$ 。在左侧，以相交点（黑色圆点）为端点的射线代表了被遮挡的方向；以箭头为端点的射线代表了未被遮挡的方向，并根据余弦因子的大小进行着色，因此更加靠近表面法线方向的箭头颜色会较浅。在右侧，蓝色箭头代表了平均的未被遮挡方向，或者叫做环境法线（bent normal）。

除了 k_A ，Landis [974]还计算了一个平均的未遮挡方向，它称为环境法线（bent normal），这个方向向量是未遮挡方向的余弦加权平均值：

$$\mathbf{n}_{\text{bent}} = \frac{\int_{\mathbf{l}\in\Omega} \mathbf{l} v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}}{\left\| \int_{\mathbf{l}\in\Omega} \mathbf{l} v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \right\|} \tag{11.10}$$

其中符号 $\|\mathbf{x}\|$ 代表了向量 $\mathbf{x}$ 的长度。积分的结果再除以它自身的长度，可以得到归一化的结果，如图 11.8 右侧所示。这样产生的向量可以在着色期间代替几何法线，从而提供更加准确的结果，同时不需要额外的性能开销（详见章节 11.3.7）。

11.3.2 可见性和 obscurance

用于计算环境遮挡因子 k_A （方程 11.8）的可见性函数 $v(\mathbf{l})$ 需要仔细定义。例如：对于一个物体，比如人物或者车辆，定义这个函数 $v(\mathbf{l})$ 是很简单的，我们只需要从表面位置向方向 $\mathbf{l}$ 投射光线，然后检查光线是否会与同一物体的任何其他部分相交即可。然而，这种方法并没有考虑到附近其他物体的遮挡情况。通常，为了进行光照，可以假设物体被放置在一个平面上，通过将该平面加入到可见性计算中，可以实现更加真实的遮挡效果。这样做的另一个好处是，物体对地面的遮挡可以模拟接触阴影的效果[974]。

不幸的是，这种可见性函数方法对于封闭的几何体是不起作用的。想象现在有一个场景，它是一个包含各种物品的封闭房间。在这种情况下，所有表面的 k_A 值都会为 0，因为来自表面的所有射线都会击中某个物体（墙壁或者物体）。对于这类场景而言，经验方法会更加合适，它会试图重现环境遮挡的外观，但是不一定会对物理可见

性进行模拟。其中的一些方法的灵感来自于 Miller 的可访问性着色（accessibility shading）[1211]，该方法对在表面角落和缝隙处如何捕获污垢或者腐蚀进行了建模。

Zhukov 等人[1970]引入了 obscurance 的思想，它通过使用距离映射函数 $\rho(\mathbf{l})$ 来代替可见性函数 $v(\mathbf{l})$，从而对环境光遮蔽的计算进行了修改：

$$k_A = \frac{1}{\pi}\int_{\mathbf{l}\in\Omega} \rho(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \tag{11.11}$$

可见性函数 $v(\mathbf{l})$ 只有两个有效值，其中 1 代表没有相交，0 表示有相交，而距离映射函数 $\rho(\mathbf{l})$ 则是一个连续函数，其返回值取决于射线与表面相交之前所传播的距离。当相交距离为 0 时，函数 $\rho(\mathbf{l})$ 的值为 0；当相交距离大于设定的最大距离 d_{max} 时，或者根本没有相交时，函数 $\rho(\mathbf{l})$ 的值为 0。对于相交距离大于 d_{max} 的交点，实际上不需要进行任何测试，这样可以大大加快 k_A 的计算速度。图 11.9 展示了环境光 occlusion 和环境光 obscurance 之间的区别。请注意，使用环境光 occlusion 进行渲染的图像要暗得多，这是因为即使在很远的距离也会对相交情况进行检测，这样会减小 k_A 的值。

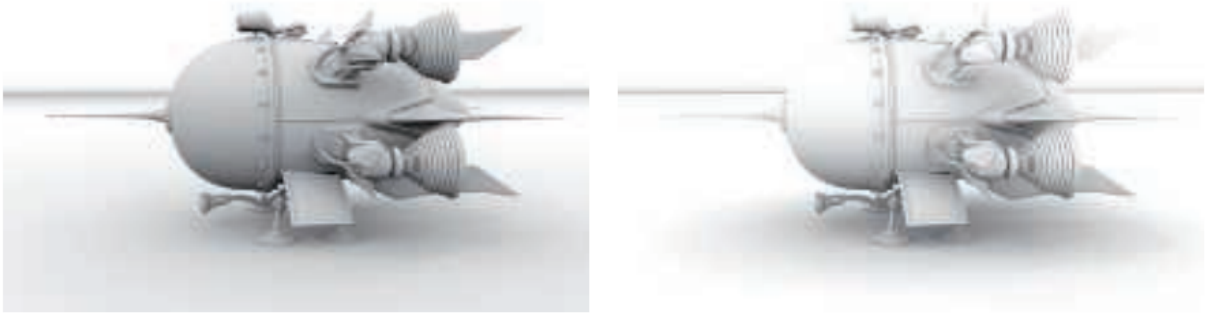


图 11.9：环境光 occlusion 和环境光 obscurance 之间的区别。左侧图像的遮挡是使用无限长的射线进行计算的。右侧图像则使用了有限长度的射线。[149]

尽管人们试图从物理角度为其辩护，但实际上 obscurance 在物理上是不正确的。然而，obscurance 通常可以给出符合观众期望的合理结果。obscurance 方法的一个缺点在于，这个最大相交距离 d_{max} 的值需要进行手动调整，才能达到令人满意的效果。这种类型的妥协会在计算机图形学中经常出现，一些技术可能并没有直接的物理基础，但是“在感知上却令人信服”。计算机图形学的目标通常是渲染一张可信的图像，因此这些技术当然是可以使用的。也就是说，基于物理理论的方法具有这样一些优点，它们可以自动进行工作，并且可以通过推理现实世界中的工作原理，来对其进一步改进。

11.3.3 考虑相互反射

尽管环境光遮蔽所产生的结果在视觉上是令人信服的，但是与完整全局光照模拟产生的结果相比，环境光遮蔽的结果要更暗一些，如图 11.10 中图像的对比。

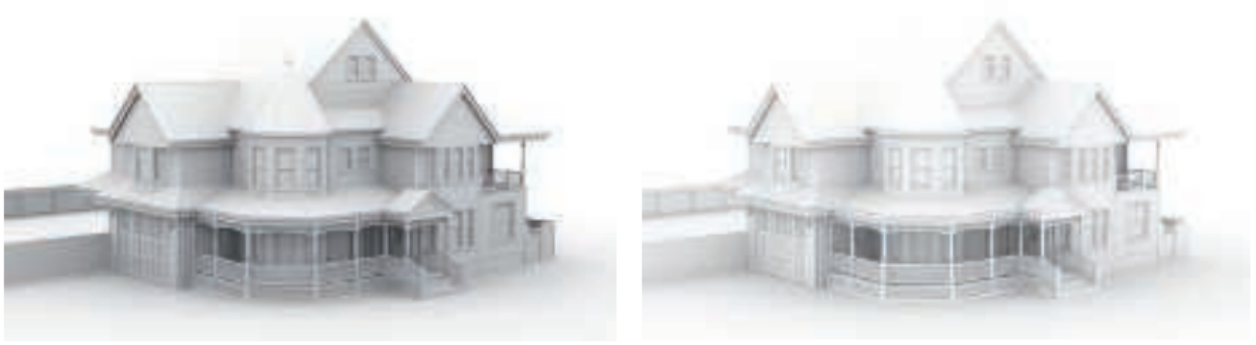


图 11.10：不具有相互反射和具有相互反射的环境光遮蔽之间的区别。左侧图像只使用了可见性信息，右侧图像使用了一次反弹的间接照明。[149]

环境光遮蔽与完整全局光照之间的一个重要区别是相互反射（interreflection）。方程 11.8 假设被遮挡方向上的 radiance 为零，但是实际上相互反射会为这些方向引入一个非零的 radiance。如图 11.10 所示，与右侧模型相比，左侧模型的折痕处和凹陷处会更暗。这种差异可以通过适当增加 k_A 的值来解决。使用 obscurance 距离映射函数来代替可见性函数（章节 11.3.2）也可以缓解这个问题，因为 obscurance 函数的值通常会大于零。

以一种更加精确的方式来追踪相互反射是很昂贵的，因为它需要求解一个递归问题。想要给一个点进行着色，必须首先对其他点进行着色，以此类推。虽然计算 k_A 的值相比于执行一个完整的全局光照计算而言要便宜得多，但是我们还是希望能够以某种形式来包含这部分丢失的光线，从而避免过度暗化。Stewart 和 Langer [1699]提出了一种廉价、但是却惊人准确的方法来近似相互反射。它基于在漫反射光照下对 Lambertian 场景的观察，即从一个给定位置能够看见的表面，往往会具有相似的 radiance。我们假设遮挡方向的 radiance L_i，等于当前着色点的出射 radiance L_o，从而打破了递归，可以得到这样一个解析表达式：

$$E = \frac{\pi k_A}{1 - \rho_{ss}\left(1 - k_A\right)} L_i \tag{11.12}$$

其中 ρ_{ss} 是次表面反照率，或者叫做漫反射率。方程 11.12 相当于使用一个新的环境遮挡因子 k'_A 来代替之前的 k_A：

$$k'_A = \frac{k_A}{1 - \rho_{ss}(1 - k_A)} \tag{11.13}$$

方程 11.13 倾向于让环境遮挡因子变得更大（更亮），从而使得它在视觉上更加接近一个完整全局光照所产生的结果，它在一定程度上模拟了相互反射效应。这种效应高度依赖于 ρ_{ss} 的值，其隐含的假设是：在着色点附近的表面颜色是相同的，这样可以产生有点像类似颜色渗透（color bleeding）的效果。Hoffman 和 Mitchell [755]使用了这种方法，从而可以实用天光来照亮地形。

Jimenez 等人[835]提出了一种不同的解决方案。他们对许多场景执行了完整的离线路径追踪，每个场景都使用均匀的、白色的、无限远的环境贴图来进行照亮，从而获得考虑相互反射的适当遮挡值。在此基础上，他们拟合了一个三次多项式，来对环境遮挡因子 k_A 和次表面反照率 ρ_{ss} 映射到遮挡值 k'_A 的函数 f 进行近似，这个新的遮挡值会相互反射的光线照亮。他们的方法同样假设反照率是局部恒定的，并且可以根据给定点的反照率，推导出入射反弹光的颜色。

11.3.4 预计算环境光遮蔽

环境遮挡因子的计算可能会很耗时，通常都是在渲染之前离线计算的。预计算任何与光照相关的信息（包括环境光遮蔽），这个过程通常被称为烘焙（baking）。

预计算环境光遮蔽最常见的方法就是蒙特卡罗方法。发射光线并检查光线与场景的交点，然后对方程 11.8 进行数值计算，例如：我们在法线 $\mathbf{n}$ 的半球方向上均匀随机选择 N 个方向 $\mathbf{l}$，然后沿着这些方向发射光线并进行追踪。基于光线的相交结果，我们对可见性函数 v 进行计算，这种计算环境光遮蔽的方法，可以表达成如下方程：

$$k_A = \frac{1}{N}\sum_{i}^{N} v(\mathbf{l}_i)(\mathbf{n} \cdot \mathbf{l}_i)^+ . \tag{11.14}$$

在计算环境光 obscurance 时，可以将投射的光线限制在一个最大距离内，通过最大距离内的相交距离来计算 v 的值。

环境光 occlusion 或者环境光 obscurance 的环境遮挡因子，其计算过程都包含一个余弦加权因子。虽然我们可以像方程 11.14 那样将其直接纳入计算，但更加有效的方法是通过重要性采样。现在我们对光线发射的分布进行调整，将其修改为按余弦加权进行发射光线，而不是先在半球范围内均匀投射光线，然后再对结果进行余弦加权。

换句话说，光线会更有可能被投射到接近表面法线的方向上，因为来自这些方向的结果具有更大的贡献值，它们是更加重要的样本。这种抽样方案被称为 Malley 方法。

环境光遮蔽的预计算可以在 CPU 或者 GPU 上进行。在这两种计算环境下，都有一些针对复杂几何图形的光线投射加速库。其中最受欢迎的两个分别是：用于 CPU 的 Embree [1829]和用于 GPU 的 OptiX [951]。在过去，来自 GPU 管线的生成结果，例如深度图[1412]或者遮挡查询[493]等，也会被用于计算环境光遮蔽，但是随着更加通用的光线投射解决方案在 GPU 上的日益普及，它们在今天不太常用了。大多数商业建模和渲染软件都会提供预计算环境光遮蔽的选项。

遮挡数据对于物体上的每个顶点都是唯一的。它们通常会存储在纹理、体积或者网格顶点中。而无论存储的信号类型如何，不同存储方法都具有类似的特点和问题。同样的方法还可以用于存储环境光遮蔽、定向遮蔽或者预计算光照等，详见章节 11.5.4。

预计算数据也可以用来模拟物体之间的环境光遮蔽效果。Kontkanen 和 Laine [924, 925]将物体对其周围的环境光遮蔽效应存储在一张立方体贴图中，并将其称为环境光遮蔽场（ambient occlusion field）。他们使用了一个二次多项式的倒数，来模拟环境光遮蔽随物体之间距离的变化情况。这个多项式的系数存储在一张立方体贴图中，以模拟遮挡的方向性变化。在运行过程中，利用遮挡物的距离和相对位置来获取合适的系数，从而对遮挡值进行重建。

Malmer 等人[1111]将环境遮挡因子和环境法线（可选）存储一个三维网格中，从而对结果进行了改进，他们将这个三维网格称为环境光遮蔽体（ambient occlusion volume）。这种方法的计算成本较低，因为可以从纹理中直接读取出环境遮挡因子，不用实时计算。与 Kontkanen 和 Laine 的方法相比，这种方法需要存储的标量值要更少一些，两种方法的纹理分辨率都比较低，总体的存储需求是类似的。Hill [737]和 Reed [1469]描述了 Malmer 等人的方法在商业游戏引擎中的实现，他们对算法的各个实现方面以及一些有用的优化方法进行了讨论。这两种方法都适用于刚体物体，而且可以扩展到具有少量运动部件的铰接物体上，其中每个运动部件都会被视为一个单独的物体。

无论我们选择哪种方法来存储环境光遮蔽值，我们都需要明确，我们正在处理的是一个连续信号。当我们从空间中的特定点发射光线的时候，我们进行的是采样（sample）；当我们在着色之前对这些数值插值的时候，我们进行的是重建（reconstruct）。信号处理领域中的所有相关工具都可以用来提高这个采样–重建过程的质量。Kavan 等人[875]提出了一种方法，他们将其称之为最小二乘烘焙（least–squares baking）。在这个方法中，遮挡信号会在网格上进行均匀采样，然后推导出顶点对应的值，从而可以在最小二乘法中，将插值和采样顶点之间的总差异

最小化。他们还专门讨论了在顶点存储数据的方法，同样的推理方法也可以用于导出存储在纹理或者体积中的值。

《命运》是一款广受好评的游戏，它使用了预计算的环境光遮蔽作为基础的间接光照解决方案（如图 11.11 所示）。这款游戏是在两代主机硬件之间的过渡时期发行的，因此它需要一个解决方案，来平衡新平台上预期的高质量画面与老平台上性能和内存使用限制。这个游戏的其中一个特点是，一天中的光照效果会随着时间动态变化，因此任何预计算的解决方案都必须正确地考虑到这一点。开发者选择使用环境光遮蔽，是因为它在有着较低开销的同时，可以提供可信的外观表现。同时，由于环境光遮蔽将可见性计算与光照渲染过程相解耦，因此可以在一天中的任何时间，使用相同的预计算数据。Sloan 等人[1658]完整介绍了这个系统，包括基于 GPU 的烘焙管线。

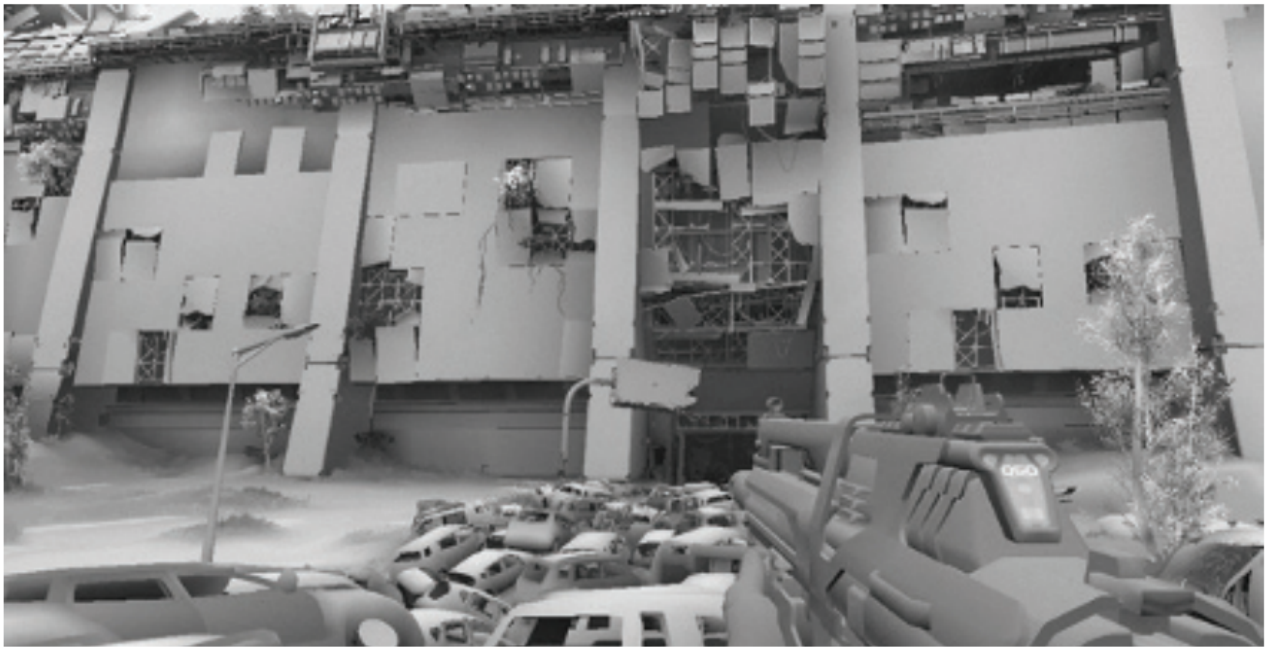


图 11.11：《命运》在间接光照计算中使用基于预计算的环境光遮蔽。该方案同时运行在了两种不同的硬件世代，兼顾了高质量和高性能。

育碧的《刺客信条》[1692]和《孤岛惊魂》[1154]系列，也使用了一种预计算的环境光遮蔽，来增强他们的间接光照解决方案。他们以自上而下的视角来渲染世界，并对产生的深度图进行处理，从而计算大范围的遮挡信息。根据相邻深度样本的分布情况，采用了多种启发式算法来对遮挡值进行估计。通过将世界空间位置投影到纹理空间中，所产生的世界空间 AO 贴图可以应用于所有物体。他们将这种方法称为世界 AO（World AO）。Swoboda [1728]也描述了类似的方法。

11.3.5 环境光遮蔽的动态计算

对于静态场景，可以预先计算环境遮挡因子 k_A 和环境法线 $\mathbf{n}_{\text{bent}}$ 。但是，对于存在移动或者形状改变物体的场景，必须要通过实时计算这些参数才能获得更好的结果。执行这个操作的方法按照空间可以划分为两类：在物体空间中执行的方法；在屏幕空间中执行的方法。

计算环境光遮蔽的离线方法，通常会从每个表面点向场景中投射大量光线（数十到数百条），并对交点进行检查。这是一个成本很高的操作，而实时渲染中则重点关注如何进行近似，或者如何避免大部分的计算。

Bunnell [210]通过将表面建模为放置在网格顶点处的圆盘元素，从而计算环境遮挡因子 k_A 和环境法线 $\mathbf{n}_{\text{bent}}$ 。这里选择圆盘的原因是，圆盘之间的遮挡情况可以通过解析计算获得，不需要单独投射光线。简单地将一个圆盘与所有其他圆盘的遮挡因子加起来，会产生双重阴影从而导致表面过暗。也就是说，如果一个圆盘位于另一个圆盘的后面，那么这两个圆盘都将被视为遮挡表面，但是实际上只有最近的圆盘才应当是。Bunnell 使用了一种巧妙的两 pass 方法来避免这个问题：第一个 pass 正常计算环境光遮蔽效果，包括错误的双重阴影在内。在第二个 pass 中，会根据第一个 pass 中的遮挡情况，来减少每个圆盘的贡献值。实际上这只是一个近似值，但在实践中，它能够产生令人信服的结果。

计算每对元素之间的遮挡情况具有 $O(n^2)$ 的复杂度，除非场景构成十分简单，否则这个复杂度对于实时渲染而言太高了。对于远距离的表面，可以使用一些简化的表示，从而降低部分计算开销。Bunnell 构建了一个分层的元素树，其中每个节点都是一个圆盘，它代表了其子树圆盘的聚合。在进行圆盘之间的遮挡计算时，对于较远的表面会使用较高层级的的节点。这可以将时间复杂度降低到 $O(n \log n)$ ，这是一个更加合理的复杂度。Bunnell 的技术很高效，并且能够产生高质量的结果，该技术被应用在了加勒比海盗电影（Pirates of the Caribbean）的最终渲染中[265]。

Hoberock [751]对 Bunnell 的算法进行了几项修改，使用更高的计算成本进一步提高了质量。他还提出了一种距离衰减因子，其结果与 Zhukov 等人[1970]所提出的 obscurance 因子相类似。

Evans [444]描述了一种基于符号距离场（signed distance field，SDF）的动态环境光遮蔽近似方法。在这种表示方法中，物体会被嵌入到一个三维网格中。网格中的每个位置都会存储到最近物体表面的距离。对于在物体内部的点，这个值为负；对于在物体外部的点，这个值为正。Evans 在体积纹理中创建并存储场景的 SDF。为了估计物体上某个表面位置的遮挡情况，他使用了一种启发式方法，该方法会沿着表面法线进行点采样，这些点会距离表面越来越远。Quilez 指出[1450]，当 SDF 以解析方式

进行表示（章节 17.3），而不是存储在三维纹理中的时候，也可以使用相同的方法进行处理。虽然这种方法是非物理的，但是生成的结果在视觉上令人满意。

Wright [1910]进一步扩展了使用符号距离场来进行环境光遮蔽的方法。Wright 并没有使用启发式方法来生成遮挡值，而是进行了锥形追踪（cone tracing）。这个圆锥的顶点位于着色点，并对编码在距离场中的场景表示进行相交测试。锥形追踪会沿轴执行一组步进操作，在每一次步进之后都会使用一个更大半径的球，来与 SDF 进行相交测试。如果此时距离最近的遮挡物距离（从 SDF 中采样的值）小于球体的半径，那么圆锥的这部分就会被遮挡（如图 11.12 所示）。如果仅仅追踪一个锥形区域，那么结果将是很不精确的，并且无法包含余弦项，出于这个原因，Wright 追踪了一组覆盖整个半球的圆锥，从而来估计环境光遮蔽。为了提高视觉保真度（visual fidelity），他的解决方案不仅使用了场景的全局 SDF，还使用局部的 SDF，这个局部 SDF 用于代表单个物体或者在逻辑上相连接的物体集合。

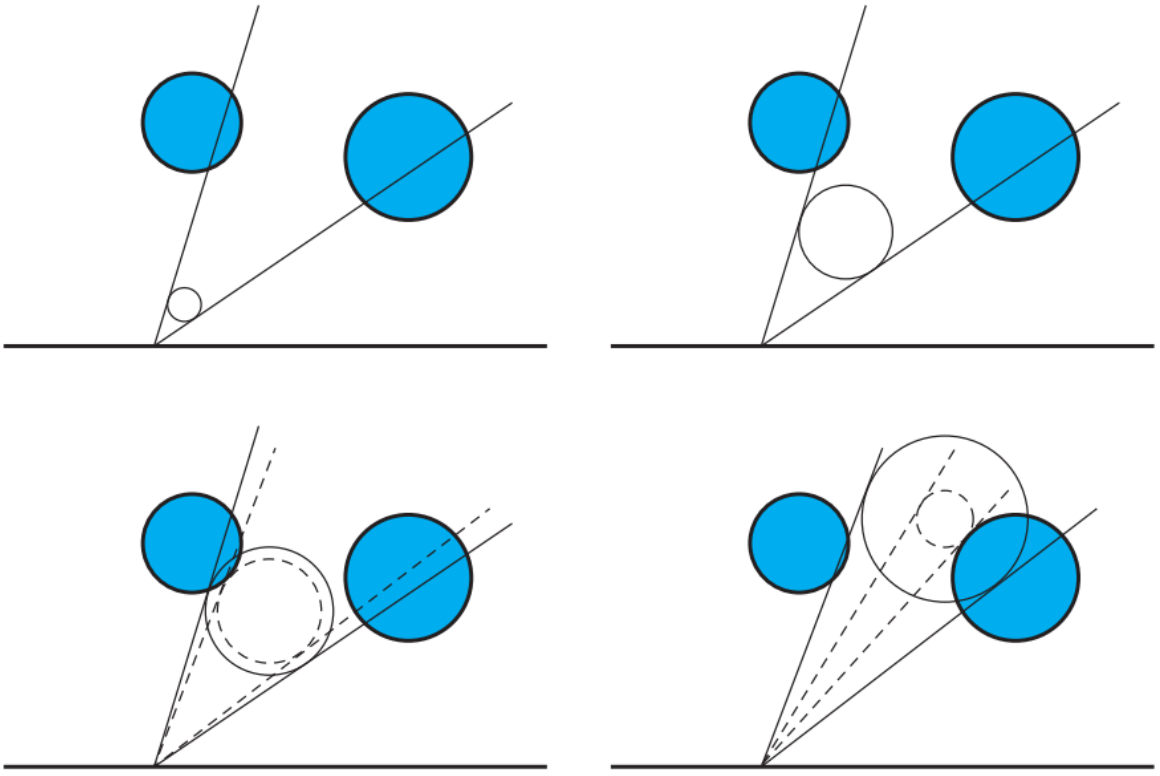


图 11.12：锥形跟踪通过在场景几何与半径不断增大的球体之间，进行一系列的相交测试来近似遮挡情况。测试球体与圆锥的侧面相接，距离顶点越远，球体的半径就越大。在每一次步进中，锥体的角度都会因为相交遮挡而减小，以考虑场景几何形状的遮挡情况。最终的遮挡因子为裁剪过后的圆锥立体角与原始圆锥立体角之比，这是一个估计值。

Crassin 等人[305]在场景的体素表示中描述了一种类似的方法。他们使用稀疏体素八叉树（章节 13.10）来存储场景的体素化信息。他们用于计算环境光遮蔽的算法，实

际上是一种通用完整全局光照算法的特例（详见章节 11.5.7）。

Ren 等人[1482]则将遮挡物近似为球体，如图 11.13 所示，并使用球谐函数来表示表面点被单个球体遮挡的可见性函数，这样一组球体聚合起来的可见性函数，就是单个球体可见性函数的乘积。但不幸的是，计算球谐函数的乘积是一个成本很高的操作。他们的核心思想是：对单个球谐可见性函数的对数进行求和，然后再对结果取指数。这样所产生的结果与可见性函数相乘的结果相同，但是球谐函数的求和操作，其计算成本明显要比乘法小。这篇论文表明，在正确的近似方法下，可以通过执行快速的对数运算和指数运算，从而获得整体加速效果。

这种方法计算出的不仅仅是环境遮挡因子，而是一个完整的球面可见性函数，它使用了球谐函数来进行表示（详见章节 10.3.2）。其中，球谐函数的第一个系数（0 阶）可以作为环境遮挡因子 k_A，后面三个系数（1 阶）可以用于计算环境法线 $\mathbf{n}_{\text{bent}}$。更高阶的系数可以用于阴影环境贴图或者圆形光源。由于这种方法将几何体近似为包围球，因此无法对来自折痕或者其他小细节的遮挡情况进行建模。

图 11.13：这种方法生成的环境光遮蔽效果是模糊的，无法显示遮挡细节。可以使用更简单的几何表示来计算 AO，这样仍然可以实现合理的效果。上图将一个犰狳模型（左）近似为一组球体（右）。在这两个例子中，模型在背后墙上投下的遮挡阴影几乎一样。

Sloan 等人[1655]在屏幕空间中，对 Ren 所描述的可见性函数进行了求和。对于每个遮挡物，他们都会考虑一组像素，这组像素距离着色点的距离，小于所规定的世界空间距离。这个操作可以通过渲染一个球体，并在着色器中执行距离测试或者使用模板测试来实现。对于所有受到影响的屏幕区域，会将一个适当的球谐函数值添加到一个离屏缓冲区中。在获得所有遮挡物的可见性之后，会对缓冲区中的值进行求幂运算，最终获得每个屏幕像素上的组合可见性函数。Hill [737]使用了相同的方法，但是他将球谐可见性函数限制到二阶系数。在这种假设下，球谐函数的乘积运算只涉及到少量的标量乘法，甚至可以通过 GPU 的固定功能混合硬件来完成。这使得我们可以在性

能有限的主机硬件上使用这种方法。由于该方法只使用了低阶的球谐函数，因此无法生成具有清晰边界的硬阴影，只能生成无方向的遮挡。

11.3.6 屏幕空间方法

基于模型空间的方法，其开销与场景的复杂度成正比。然而，我们完全可以从屏幕空间中已有的数据出发，推导出一些有关遮挡的信息，例如深度和法线。这种基于屏幕空间的算法，具有恒定的开销，其复杂度与与场景的细节程度无关，只与渲染时所使用的画面分辨率有关。

> 在实践中，屏幕空间算法的执行时间，还取决于数据在深度缓冲或者法线缓冲中的分布，因为这种数据分散效应，在进行遮挡计算的时候，会降低 GPU 缓存的命中率，从而延长算法的执行时间。

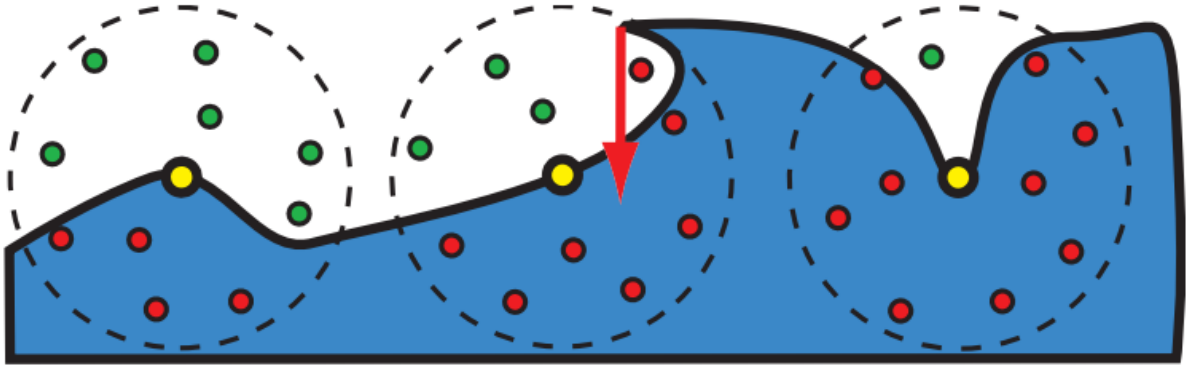


图 11.14：Crytek 的环境光遮蔽方法，被应在了图中的三个表面点（黄色圆圈）上。这里为了清晰起见，使用了二维形式来展示该算法的流程，相机位于图像内容的正上方（未显示在图中）。在这个例子中，有 10 个样本分布在了围绕表面点的圆盘上（实际上它们是分布在一个球体内部）。未通过深度测试的样本点使用红色进行表示，即该样本所对应的深度，超过了 z-buffer 中对应位置的深度；通过的样本则使用绿色进行表示。环境遮挡因子 k_A 的值是通过测试的样本数与总样本数的加权比值。为了简单起见，这里我们先忽略了可变的样本权重，假设所有的样本都具有相同的权重。对于左边的像素点，总共 10 个样本，其中有 6 个通过，因此 $k_A = 0.6$ 。对于中间的像素点，只有 3 个样本通过了测试。还有一个样本虽然在物体外部，但是没有通过深度测试，如图中红色箭头所示，最终计算出的 $k_A = 0.3$ 。对于右边的像素点，只有 1 个样本通过了测试，因此 $k_A = 0.1$ 。

Crytek [1227]开发了一种动态的屏幕空间环境光遮蔽（screen-space ambient occlusion，SSAO）算法，并用在了《孤岛危机》中。他们使用 z-buffer 作为唯一的输入，在一个全屏 pass 中计算来环境光遮蔽效果。每个像素都有一个环境遮挡因子 k_A ，它会在该像素周围的球形范围内采样一组点，并将样本与 z-buffer 进行深度

测试，然后来估计 k_A 。 k_A 的值与 z-buffer 中位于像素点深度前面的测试样本有关，通过的样本数量越少， k_A 的值就越低，如图 11.14 所示。与 obscurance 因子相类似[1970]，这些样本的权重会随着到像素距离的增大而减小，即距离像素越远，该样本的权重就越小。需要注意的是，由于这些样本并没有被余弦因子 $(\mathbf{n} \cdot \mathbf{l})^+$ 加权，因此所产生的环境光遮蔽效果是不正确的。该方法会将球形范围内的所有样本都考虑在内，而不是只考虑表面上半球范围内的样本。这种简化意味着会对表面以下的样本进行计数，但是实际上我们是不应当对它们进行计数的。这样做会导致表面变暗（因为环境遮挡因子 k_A 变大了），同时边缘会比周围环境更亮。尽管如此，最终产生结果在视觉上令人十分满意，如图 11.15 所示。

图 11.15：左上角：展示了屏幕空间环境光遮蔽的效果。右上角：展示了没有环境光遮蔽的反照率（漫反射颜色）。左下角：将上述两者进行了合并。右下角：最终的渲染图像，添加了高光着色和阴影效果。

Shanmugam 和 Arikan [1970]同时提出了一种类似的方法。在他们的论文中，他们描述了两种方法，其中一种可以从附近的小细节中生成良好的环境光遮蔽效果；另一中可以从较大的物体中生成较为粗略的环境光遮蔽效果。将二者的结果结合起来，就可以生成最终的环境遮挡因子。其中，他们的精细尺度环境光遮蔽方法使用了一个全屏 pass，在这个 pass 中，不仅访问了 z-buffer，还访问了可见像素表面的法线缓冲。对于每个着色像素，会从 z-buffer 中对附近的像素进行采样，被采样的像素分布在球体内部，会根据其法线信息来计算着色像素的遮挡项。这种方法并没有将双重

阴影考虑在内，因此结果会显得有点暗。他们的粗略遮挡方法，与 Ren 等人的物体空间方法相类似（我们在上文中讨论过），它同样将遮挡几何体近似为球体。然而不同的是，Shanmugam 和 Arikan 的方法是在屏幕空间进行遮挡计算的，并使用了与屏幕对齐的广告牌，来覆盖每个遮挡球体的“效果区域”。与 Ren 等人[1482]的方法不同，这里的粗略遮挡方法也没有考虑双重阴影。

由于这两种方法极其简洁，因此很快引起了工业界和学术界的注意，并催生了大量的后续工作。许多方法，例如 Filion 等人[471]在游戏《星际争霸 II》中所使用的方法，以及 McGuire 等人[471]所使用的可扩展环境光 obscurance 方法，都使用了这种特别启发式方法（ad hoc heuristics）来生成遮挡因子。这类方法具有良好的性能表现，并暴露出了一些参数，可以通过手动调整参数来达到预期的艺术效果。

其他的一些方法旨在提供更有原则和理论基础的遮挡计算方法。Loos 和 Sloan [1072]注意到，Crytek 的方法可以被解释为蒙特卡洛积分。他们将计算出来的值称为体积 obscurance，并将其定义为：

$$v_A = \int_{\mathbf{x} \in X} \rho(d(\mathbf{x}))o(\mathbf{x})d\mathbf{x} \tag{11.15}$$

其中 X 是围绕该像素点的一个三维球形邻域； ρ 是距离映射函数，与方程 11.11 所描述的相类似； d 是距离函数； $o(\mathbf{x})$ 是占用函数（occupancy function），如果 $\mathbf{x}$ 未被占用，则 $o(\mathbf{x})$ 等于 0，否则等于 1。他们注意到， $\rho(d)$ 函数对于最终视觉质量的影响很小，因此可以使用常数函数。在这个假设下，体积 obscurance 是对占用函数在像素点邻域上的积分。Crytek 的方法是在三维邻域内进行随机采样从而计算积分，而 Loos 和 Sloan 则通过对像素的屏幕空间邻域随机采样，在 xy 维度上进行积分，对 z 轴的积分过程则是解析的。如果该点的球面邻域中不包含任何几何图形，则积分值等于射线与球体 X 相交的长度。如果该点的球面邻域中存在几何图形，则会使用深度缓冲来作为占用函数的近似值，并且仅会在每个线段的未占用部分上进行积分，如图 11.16 左侧所示。该方法最终生成的结果，其质量与 Crytek 的方法相当，但是使用的样本数量较少，因为在其中一个维度上（ z 轴）的积分是精确的。如果可以使用表面法线的话，还可以对这个方法进一步扩展，从而获得更好的结果。在这个考虑法线的版本中，线积分会被限制在由像素点法线所定义的平面上。

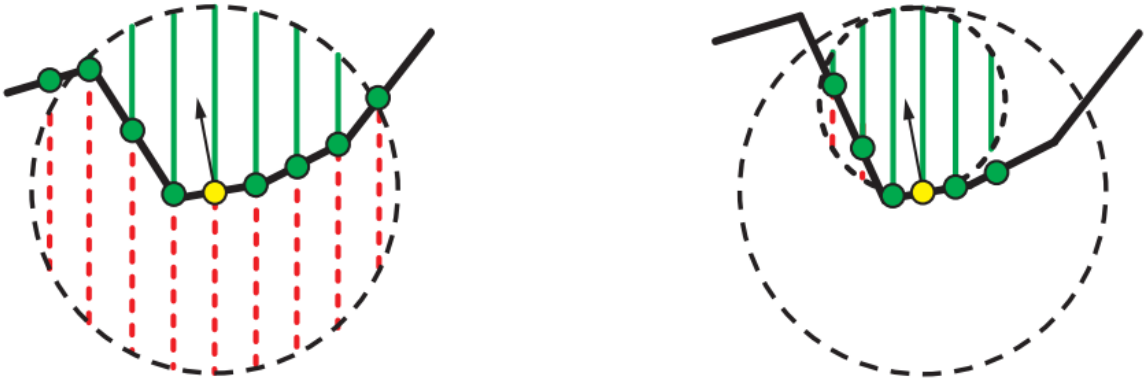


图 11.16：体积 obscurance（左）使用了线积分，来计算像素点周围的未占用体积的积分。体积环境光遮蔽（右）同样也使用了线积分，不同的是，它计算了与着色点相切球体的占用率，这对反射方程中的余弦项进行了模拟。在这两种情况下，积分的结果都是球体的未占用体积（绿色实线）与球体总体积（未占用体积与占用体积之和，其中占用体积使用红色虚线进行表示）的比值。对于这两幅图像，相机都是从上往下观察的，其中绿色点代表了从深度缓冲中读取的样本，黄色点代表了正在计算遮挡情况的样本。

Szirmay-Kalos 等人[1733]提出了另一种使用法线信息的屏幕空间方法，它被称为体积环境光遮蔽（volumetric ambient occlusion）。方程 11.6 描述了在法线半球上进行的积分，这个积分还包含了余弦项。他们提出，这种类型的积分，可以将被积函数中的余弦项移除，并使用余弦分布来限制积分范围，从而对余弦因子进行近似。这样做可以将积分转换到一个球面上，而不是在一个半球上；这个球体的半径为半球的一半，并且会沿着法线移动一个球体半径的距离，最终这个球体会与半球内接，被半球完全包裹。其中未被占用部分的体积，其计算方法与 Loos 和 Sloan 的方法一样，都是通过在像素邻域上进行随机采样，并在 z 轴上对占用函数进行解析积分，如图 11.16 右侧所示。

Bavoil 等人[119]提出了一种不同的方法，用于解决估计局部可见性的问题，他们从 Max [1145]的视界映射（horizon mapping）中获得了灵感。他们的方法被称为基于视界的环境光遮蔽（horizon-based ambient occlusion，HBAO），它假设 z-buffer 中的数据表示了一个连续的高度场。通过确定视界角（horizon angle），可以对像素点的可见性进行估计，这里的视界角，指的是切面上方被邻域遮挡的最大角度。也就是说，对于某个点上的给定方向，我们会记录最高的可见物体所对应的角度。如果我们忽略积分中的余弦项，那么环境遮挡因子可以被计算为视界上未被遮挡部分的积分，或者是 1 减去视界下被遮挡部分的积分：

$$k_A = 1 - \frac{1}{2\pi} \int_{\phi=-\pi}^{\pi} \int_{\alpha=t(\phi)}^{h(\phi)} W(\omega) \cos(\theta) d\theta d\phi \qquad (11.16)$$

其中 $h(\phi)$ 是切平面的视界角； $t(\phi)$ 是切平面与观察向量的切角（tangent angle）； $W(\omega)$ 是衰减函数，如图 11.17 所示。积分前面的 $1/2\pi$ 是归一化系数，它将积分的结果归一化到 $[0,1]$ 之间。

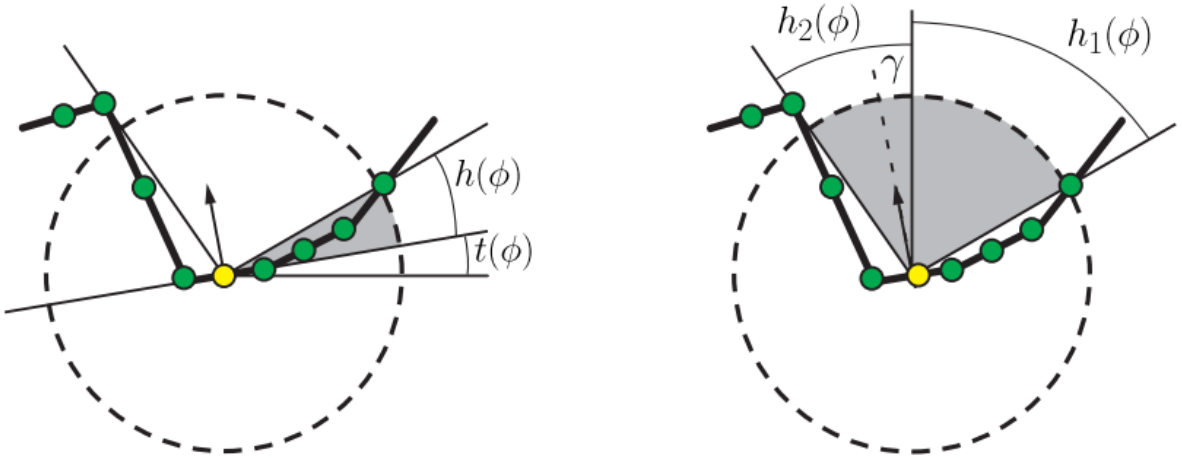


图 11.17：HBAO（左）通过找到切平面上方的视界角 h ，并对视界角之间的未遮挡角度进行积分，从而计算环境遮挡因子。切平面和观察向量之间的角度记为 t 。GTAO 使用了相同的视界角度 h_1 和 h_2 ，同时还使用了法线和观察向量之间的角度 γ ，并将余弦项添加到计算中。在上述两幅图中，相机都是从上往下观察场景的，图中显示的是场景横截面，其中视界角是角度 ϕ 的函数， ϕ 是一个相对于观察方向的角度。图中绿色的点代表了从深度缓冲中读取的样本。黄色点代表了正在进行遮挡计算的样本。

对于定义视界的角度 ϕ ，我们利用角度的线性衰减，可以解析地计算内部的积分：

$$k_A = 1 - \frac{1}{2\pi}\int_{\phi=-\pi}^{\pi}(\sin(h(\phi)) - \sin(t(\phi)))W(\phi)d\phi \qquad (11.17)$$

这个剩余的积分，是通过对一些方向进行采样，来找到视界角度，从而进行数值计算的。

Jimenez 等人[835]也使用了这种基于视界的方法，他们称之为真实环境光遮蔽（ground-truth ambient occlusion ，GTAO）。他们的目标是实现 ground-truth 的结果，并能够与光线跟踪的结果相匹配，该方法所使用的唯一信息，就是由 z-buffer 构建的高度场。HBAO 在计算遮挡的时候并不包括余弦项，并且它还增加了一个特殊的衰减（没有出现在方程 11.8 中），因此它的结果最多只能与光线追踪相接近，但是始终还是不一样的。GTAO 引入了缺失的余弦因子，去除了这个特殊的衰减函数，并在绕观察向量的参考系中给出了遮挡积分，该方法的遮挡因子定义如下：

$$k_A = \frac{1}{\pi}\int_0^{\pi}\int_{h_1(\phi)}^{h_2(\phi)} \cos(\theta-\gamma)^+ |\sin(\theta)| d\theta d\phi \qquad (11.18)$$

其中 $h_1(\phi)$ 和 $h_2(\phi)$ 为给定 ϕ 的左右视界角； γ 为表面法线与观察方向之间的夹角。这里积分的归一化项为 $1/\pi$ ，这与 HBAO 中的不同，因为 GTAO 包含了余弦项，这使得开放半球的积分结果为 π ，如果方程中不包含余弦项，则开放半球的积分结果为 2π 。在给定高度场假设的情况下，方程 11.18 与与方程 11.8 完全匹配，如图 11.17 所示。这里的内部积分仍然可以进行解析求解，因此只需要对外部积分进行数值计算即可，这个积分过程与 HBAO 中的积分过程基本相同，都是对给定像素周围的多个方向上进行采样。

在这些基于视界的方法中，成本最高的操作就是沿着屏幕空间的线段对深度缓冲进行采样，从而确定视界角度。Timonen [1771]提出了一种方法，专门用于提高这一步的性能表现。他指出，用于估计给定方向上视界角度的样本，可以在屏幕空间中沿直线排列的像素之间进行大量重用。他将遮挡计算分为两步，首先，他会在整个 z-buffer 中执行线段追踪。在追踪的每一步中，他都会根据所规定的最大影响距离，在沿着线段移动的时候更新视界角度，并将这个信息写入一个缓冲区中。在视界映射（horizon mapping）中，每个屏幕空间方向上都会创建一个这样的缓冲区。这些缓冲区的大小不需要与原始的深度缓冲区相同，而是取决于线段之间的间距，以及沿着线段的步长，在选择这些参数的时候有一定的灵活性。不同的设置会对最终的质量产生影响。

第二步是根据存储在缓冲区中的视界角度信息来计算遮挡因子。Timonen 使用 HBAO（方程 11.17）所定义的遮挡因子，但是也可以使用其他遮挡估计方法，例如 GTAO 中的遮挡因子（方程 11.18）。

深度缓冲并不是一个完美的场景表示，因为在一个给定的方向上，只有最近的物体会被记录下来，我们实际上并不知道它背后发生了什么。有许多技术使用了启发式方法，来尝试推断可见物体的厚度信息，这些近似值在许多情况下都表现良好，人眼对于稍微不准确的结果是很宽容的。虽然有一些方法使用了多层深度来缓解这个问题，但是由于将其集成到渲染引擎中太过复杂，并且这类方法的运行时成本很高，因此它们从未流行过。

屏幕空间中的方法依赖于对 z-buffer 进行反复采样，从而在给定点周围构建一些简化的几何模型。实验表明，想要获得较高的视觉质量，可能需要多达几百个样本，这个级别的样本数量太多了，想要用于交互式渲染，每个像素最多只能采样 10–20 个样本，甚至更少。Jimenez 等人[835]报告提到，为了适应 60 FPS 的性能预算，他

们只能在每个像素上使用 1 个样本！为了弥合理论和实践之间的差距，屏幕空间方法通常会采用某种形式的空间抖动。在最常见的形式中，每个屏幕像素都会使用略有不同的随机样本集合，然后进行旋转或者径向移动。并在 AO 计算的主要阶段之后，执行一次全屏的滤波 pass。联合双边滤波（章节 12.1.1）可以避免在表面的不连续处进行过滤，从而保持尖锐的边缘。它可以利用可用的深度信息或者法线信息来对过滤进行限制，即它只会对属于同一表面的样本进行过滤。还有一些方法使用了随机变化的采样模式，以及经过实验选择的滤波核；另一些方法则使用了固定大小的屏幕空间采样模式（例如 4×4 像素），以及一个限制在该邻域上的滤波核。

环境光遮蔽的计算也可以在时域上进行超采样[835, 1660, 1916]。通常会在每一帧中应用不同的采样模式，并对计算出来的遮挡因子进行指数平均从而实现这个目的。使用上一帧的 z-buffer、相机变换和动态物体的运动信息，来将上一帧的数据重新投影到当前视图中，然后再将其与当前帧的结果进行混合。还会使用一些基于深度、法线、速度的启发式方法，来检测上一帧数据的可靠性，对于不可靠的数据需要丢弃（例如：由于一些新物体进入了视野中，因此上一帧中的数据与当前帧存在差异）。章节 5.4.2 在更一般的情况下，介绍了时域超采样和时域抗锯齿技术。时域过滤的成本较小，并且很容易实现，虽然它并不总是完全可靠的，但是在实践中出现的大多数问题都不太明显。这主要是因为环境光遮蔽不会直接单独显示在画面上，它只是光照计算的输入之一。在将这种环境光遮蔽效果与法线贴图、反照率纹理以及直接光照相结合之后，任何微小的瑕疵都会被掩盖掉，人眼一般很难观察到这些瑕疵。

11.3.7 使用环境光遮蔽进行着色

虽然我们是在恒定、遥远光照环境中推导出的环境光遮蔽值，但是我们也可以将其应用于更复杂的光照场景中。再次回顾一些反射方程：

$$L_o(\mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}. \tag{11.19}$$

如章节 11.3.1 中所介绍的，方程 11.19 中包含了可见性函数 $v(\mathbf{l})$ 。

假如我们现在正在处理一个漫反射表面，我们可以使用 Lambertian BRDF 来代替方程 11.19 中的 $f(\mathbf{l}, \mathbf{v})$ ，这个 BRDF 等于次表面反照率 ρ_{ss} 除以 π ，将其带入方程 11.19，可得：

$$L_o = \int_{\mathbf{l} \in \Omega} \frac{\rho_{ss}}{\pi} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} = \frac{\rho_{ss}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{11.20}$$

我们对方程 11.20 进行一些化简整理，可得：

$$\begin{aligned} L_o &= \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \\ &= \frac{\rho_{\text{ss}}}{\pi} \frac{\int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}}{\int_{\mathbf{l} \in \Omega} v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}} \int_{\mathbf{l} \in \Omega} v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \\ &= \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) \frac{v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+}{\int_{\mathbf{l} \in \Omega} v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}} d\mathbf{l} \int_{\mathbf{l} \in \Omega} v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}. \end{aligned} \tag{11.21}$$

如果我们使用方程 11.8 中所定义的环境光遮蔽，则方程 11.21 可以简化为：

$$L_o = k_A \rho_{\text{ss}} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) K(\mathbf{n}, \mathbf{l}) d\mathbf{l} \tag{11.22}$$

其中：

$$K(\mathbf{n}, \mathbf{l}) = \frac{v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+}{\int_{\mathbf{l} \in \Omega} v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}} \tag{11.23}$$

上述形式为我们提供了一个全新的视角来看待这个过程。方程 11.22 中的积分，可以认为是对入射 radiance L_i 应用了一个方向性的滤波核 K 。滤波器 K 以一种复杂的方式在空间和方向上同时变化，但它具有两个重要的属性。首先，由于对点积进行了 clamp 操作，因此它最多只能覆盖点 $\mathbf{p}$ 法线周围的半球范围。其次，由于分母中包含归一化因子，因此它在整个半球上的积分等于 1。

为了进行着色，我们需要计算两个函数乘积的积分，即入射 radiance L_i 和滤波器函数 K 乘积的积分。在某些情况下，我们可以使用一种简化的方式来描述这个滤波器，并以很低的成本来计算这个二重积分，例如当 L_i 和 K 都使用球谐函数来进行表示的时候（章节 10.3.2）。降低这个方程复杂度的另一种方法是，使用一个具有类似特性，但是更简单的滤波器来对其近似。最常见的选择就是归一化的余弦核函数 H ：

$$H(\mathbf{n}, \mathbf{l}) = \frac{(\mathbf{n} \cdot \mathbf{l})^+}{\int_{\mathbf{l} \in \Omega} (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}} \tag{11.24}$$

在没有入射光线被阻挡的时候，这种近似是十分准确的。它还涵盖了与原本滤波器相同的角度范围。虽然它完全忽略了可见性函数，但是方程 11.22 中仍然包含了环境光遮蔽 k_A，因此在被着色的表面上会有一些与可见性相关的暗化。

选择了这个近似滤波核，那么方程 11.22 就变成了：

$$L_o = k_A \rho_{ss} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) \frac{(\mathbf{n} \cdot \mathbf{l})^+}{\int_{\mathbf{l} \in \Omega} (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l}} d\mathbf{l} = \frac{k_A}{\pi} \rho_{ss} E. \qquad (11.25)$$

这意味着，在最简单的形式中，可以通过计算 irradiance，并将其乘上环境光遮蔽值来完成环境光遮蔽的效果着色。这里的 irradiance 可以来自任何来源，例如：它可以从 irradiance 环境贴图（章节 10.6）中进行采样。这种方法的准确性，取决于这个近似滤波器有多大能力能够表现正确滤波器。对于在球面上平滑变化的光照，这种近似方式能够给出合理的结果。如果 L_i 在所有可能的方向上都是恒定的，就好像场景是由全白的环境贴图所照亮的那样，在这种情况下，它是完全准确的。

这个方程还让我们了解到，为什么环境光遮蔽对于精确光源或者很小的面光源而言是一个很差的可见性近似，因为这些光源在表面上只占据了很小的一个立体角（对于精确光源而言是无穷小的），而可见性函数对光照积分会产生重要影响。它几乎是以二进制的方式来控制光源的贡献，也就是说，它要么完全启用，要么完全禁用。忽略可见性（正如我们在方程 11.25 中所做的那样）是一个影响很大的近似操作，这样做通常不会产生符合预期的结果。在这种近似情况下，所产生的阴影缺乏清晰度，并且没有任何预期的方向性，也就是说，它们看起来并不像是由特定光源产生的。对这种光源的可见性进行建模，环境光遮蔽并不是一个好的选择，应当使用一些其他的方法，例如阴影贴图等。然而，值得注意的是，有时候我们会使用较小的局部光源来模拟间接光照的效果，在这种情况下，使用环境光遮蔽值来调整它们的贡献是合理的。

到目前为止，我们都是假设在 Lambertian 表面上进行着色的。在处理更加复杂的、非常数的 BRDF 时，这一项无法从积分中提出来（就像我们在方程 11.20 中所做的那样）。对于镜面材质而言，K 不仅取决于可见性和法线，还取决于观察方向。对于一个典型的微表面 BRDF 而言，其波瓣会在整个区域上发生显著改变；使用单一的、预先确定的形状来对其近似会显得过于粗糙，无法产生可信的结果。这也就是为什么在漫反射 BRDF 中，使用环境光遮蔽进行着色最有意义的原因。我们会在接下来的若干小节中讨论一些其他方法，它们更加适合复杂的材质模型。

使用环境法线（详见方程 11.10）可以更加精确地近似滤波器 K。虽然滤波器中仍然没有包含可见性项，但是其最大值与未被遮挡的平均方向相匹配，这使得它在总体上

可以更好地逼近方程 11.23。当几何法线和环境法线不匹配的时候，使用环境法线将会给出更加准确的结果。Landis [974]不仅将它用在环境贴图的着色中，还用在了一些直接光照的着色中，来代替常规的阴影技术。

对于环境贴图的着色，Pharr [1412]提出了一种替代方案，该方法使用 GPU 的纹理过滤硬件来动态执行滤波操作。滤波器 K 的形状是动态确定的，其滤波中心位于环境法线的方向上，其大小取决于 k_A 的值，这样可以更加精确地与方程 11.23 中的原始滤波器相匹配。

11.4 定向遮蔽

尽管单独使用环境光遮蔽可以极大地提高图像的视觉质量，但它毕竟是一个大大简化了的模型。在处理大面积光源的时候，它所给出的可见性近似很差，更不用说较小的光源或者精确光源了。它也无法正确处理光滑的 BRDF 或者更加复杂的光照环境。想象现在有一个表面，它被远处的圆形顶灯所照亮，这个圆形顶灯的颜色从红色渐变为绿色。这个圆形顶灯可能会用来代表来自天空的光线，又或者是来自某个遥远的星球的光线，如图 11.18 所示。即使环境光遮蔽会让点 $\mathbf{a}$ 和点 $\mathbf{b}$ 的光照变暗，但是它们仍然会被红色和绿色的天空所照亮。使用环境法线可以缓解这种效果，但是这样做也不是完美的。我们之前所提出的简单模型不够灵活，无法处理这种特殊情况，其中一种解决方案是，使用一些更具表现力的方式来描述可见性。

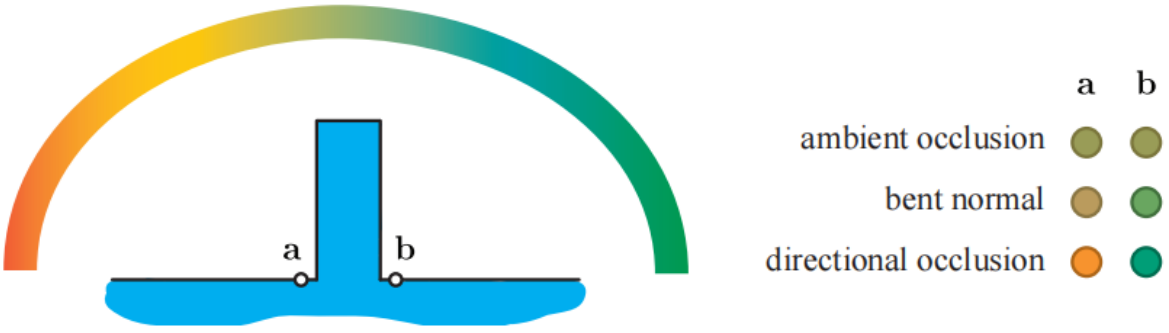


图 11.18：图中展示了在复杂的光照条件下，点 $\mathbf{a}$ 和点 $\mathbf{b}$ irradiance 的近似颜色。环境光遮蔽无法模拟任何方向性，因此这两个点上的颜色是相同的。使用环境法线可以有效地将余弦波瓣移向天空的未遮挡部分，但是由于积分范围没有受到任何限制，因此所提供的结果还不够准确。定向遮蔽方法能够正确地消除来自天空中被遮挡部分的光线。

我们将专注于编码整个球面可见性或者半球可见性的方法，即描述哪些方向会阻挡入射 radiance 的方法。这些信息可以用来为精确光源产生阴影，但这并不是它的主要目的。针对这些特定类型光源的专用方法（详见第 7 章），能够生成质量更好的阴影，因为它们只需要对光源的某个位置或者某个方向进行可见性编码即可。

这里我们所要描述的解决方案，主要是用于为大面积光源或者环境照明提供遮挡效果，这些方法可以生成柔和的阴影，并且由近似可见性所引起的瑕疵也不是很明显。此外，这些方法还可以在常规阴影技术无法运行的时候，提供一些遮挡效果，例如凹凸贴图细节所产生的自阴影，以及超大场景的阴影，导致阴影贴图没有足够的分辨率。

11.4.1 预计算定向遮蔽

Max [1145]引入了视界映射（horizon mapping）的概念来描述高度场表面的自遮挡现象。在视界映射中，对于表面上的每个点，会根据一组方位角方向来确定视界角度，例如 8 个方向：北、东北、东、东南、以此类推。

我们可以不存储在给定方位上的视界角，而是将未遮挡的三维方向集合作为一个整体，将其建模为椭圆[705, 866]或者圆形[1306, 1307]孔径，其中后一种技术被称为环境光圈照明（ambient aperture lighting，如图 11.19 所示）。这些技术对存储的要求比视界映射低，但是当未遮挡的方向不像椭圆或者圆的时候，可能会产生错误的阴影效果。例如在一个平面上，以规则间隔突出的山峰，应该设置一个星形的未遮挡方向，这与上述椭圆方案和圆形方案不匹配。

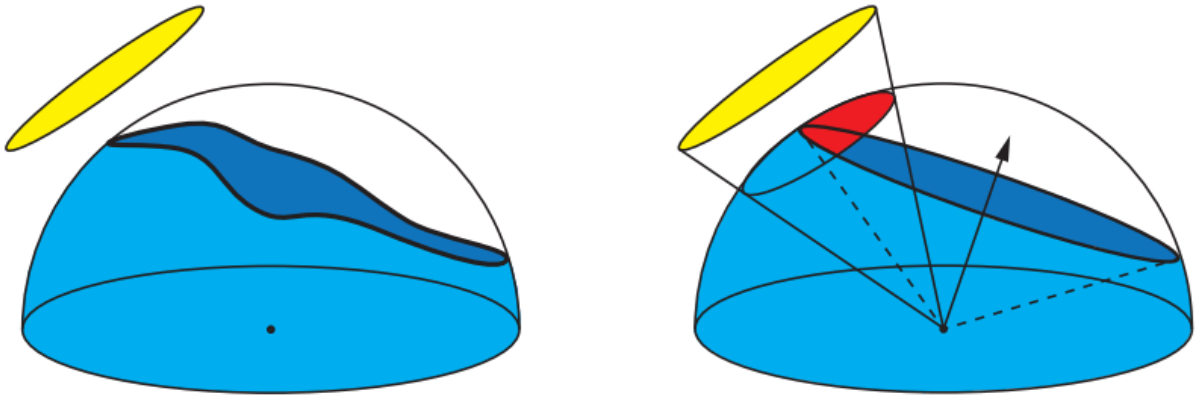


图 11.19：环境光圈照明使用了一个圆锥体，来对着色点上方未遮挡区域的实际形状进行近似。左图中，面光源实用黄色进行表示，表面位置的可见视界使用蓝色进行表示。右图中，视界范围被简化为一个圆形，它是一个从表面位置向右上方突出的圆锥，圆锥的边缘使用虚线进行表示。然后通过将面光源的圆锥，与代表未遮挡区域的圆锥相交，相交区域使用红色进行表示。

遮挡技术有许多变种。Wang 等人[1838]使用了球形符号距离函数（spherical signed distance function，SSDF）来表示可见性。它将一个到被遮挡区域边界的符号距离编码到球体上。章节 10.3 节中所讨论的任何球面基底或者半球基底，都可以用来对可见性进行编码[582, 632, 805, 1267]。就像环境光遮蔽一样，这些定向可见性信息可以存储在纹理、网格顶点或者体积中[1969]。

11.4.2 定向遮蔽的动态计算

许多用于生成环境光遮蔽的方法，同样也可以用于生成定向的可见性信息。Ren 等人[1482]提出的球谐函数指数方法，以及 Sloan 等人[1655]提出的屏幕空间变体，以球谐向量的形式来生成可见性。如果使用多个 SH 频带，这些方法本身就可以提供方向信息，同时使用更多的频带可以更加精确的对可见性进行编码。

锥形追踪方法，例如 Crassin 等人[305]和 Wright [1910]所提出的方法，它们为每个追踪区域都提供了一个遮挡值。出于质量原因，即使是对环境光遮蔽进行估计，也是会进行多次锥形追踪，这些可用的信息本身就已经具有方向性了。如果还需要某些特定方向的可见性，我们还可以在该方向上执行较少次数的锥形追踪。

Iwanicki [806]也使用了锥形追踪，但他将其限制在了一个方向上。该方法将动态角色近似为一组球体，追踪的结果用于生成投射到静态几何体上的软阴影，这与 Ren 等人[1482]和 Sloan 等人[1655]的方法相类似。在这个解决方案中，静态几何物体的照明使用 AHD 进行编码存储（详见章节 10.3.3）。环境光遮蔽和定向遮蔽这两部分的可见性可以独立进行处理，其中在对于定向可见性而言，会对指定方向进行一次锥形追踪，并与球体进行相交，从而计算其衰减因子。

许多屏幕空间中的方法也可以进行扩展，从而提供定向的遮挡信息。Klehm 等人[904]使用 z-buffer 数据来计算屏幕空间中的环境圆锥（screen-space bent cone），这些圆锥实际上就是圆形孔径，这与 Oat 和 Sander[1307]离线预计算的圆锥非常相似（章节 11.4.1）。当对像素的邻域进行采样的时候，它会将未遮挡的方向相加，最终的结果向量，其长度可以用来估计可见性锥（visibility cone）的顶角大小，其方向则定义了可见锥的轴。Jimenez 等人[835]使用视界角度来估计圆锥的轴方向，并使用环境遮挡因子来推导出圆锥的顶角大小。

11.4.3 使用定向遮蔽进行着色

由于编码定向遮蔽的方式实在太多，因此我们无法提供一个标准通用的着色方案，具体所使用的解决方案将取决于我们想要达到的特定效果。

让我们再次回顾反射方程，在这个版本的方程中，我们将入射 radiance 拆分为远处的照明 L_i 及其可见性 v：

$$L_o(\mathbf{v}) = \int_{\mathbf{l}\in\Omega} f(\mathbf{l},\mathbf{v})L_i(\mathbf{l})v(\mathbf{l})(\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \tag{11.26}$$

我们能够做的最简单的操作，就是使用可见形信号来遮挡精确光源。由于大多数编码可见性的方法都很简单，其结果的质量往往也不太令人满意，但是这样的方法可以让我们在一个基本的例子中进行推理。这种方法同样也可以用于传统阴影方法由于分辨率不足而失效的情况，在这种情况下，生成的结果精度没有那么重要，总比没有任何形式的遮挡要好得多。这种情况包括：面积非常大的地形模型，使用凹凸贴图表示的表面微小细节等。

根据章节 9.4 中的讨论，当处理精确光源的时候，方程 11.26 会变为：

$$L_o(\mathbf{v}) = \pi f\left(\mathbf{l}_c, \mathbf{v}\right) \mathbf{c}_{\text{light}}\, v\left(\mathbf{l}_c\right) \left(\mathbf{n} \cdot \mathbf{l}_c\right)^+ \tag{11.27}$$

其中 $\mathbf{c}_{\text{light}}$ 是一个纯白的 Lambertian 表面正对光源时所反射出的颜色，$\mathbf{l}_c$ 是指向光源的颜色。

我们可以把上面的方程解释为，首先计算材质对未遮挡光源的响应结果，再将结果乘以可见性函数的值。如果光线方向位于视界以下（当使用视界映射时）、或者位于可见锥之外（当使用环境光圈照明时）、或者位于 SSDF 的负区域，那么可见性函数的值为零，因此不需要考虑来自光源的任何贡献。值得一提的是，尽管可见性函数被定义为一个二进制函数，但是许多表示方式都可以返回整个范围内的值，即 $[0, 1]$，而不仅仅是非 0 即 1，位于范围内的非整数值代表部分遮挡的情况。

> 至少在大多数情况下是这样。在某些情况下，我们希望可见性函数取 0 和 1 以外的值，但是仍然位于 $[0, 1]$ 范围内。例如：当对由半透明材质所引起的遮挡进行编码时，我们可能会希望使用小数遮挡值。

由于振铃效应，球谐函数或者 H-basis 甚至可能会重建出负值，这些行为是我们不想要的，它只是编码方式的固有属性。

我们可以对面光源进行类似的推理。在这种情况下，位于光源所对应的立体角内，L_i 等于光源发出的 radiance；位于光源所对应的立体角外，L_i 为零。我们将其记作 L_l，并假设它在光源立体角上是恒定的。此时我们可以将对整个球体 Ω 的积分，转换为对光源立体角 Ω_l 的积分，即：

$$L_o(\mathbf{v}) = L_l \int_{\mathbf{l} \in \Omega_l} v(\mathbf{l}) f(\mathbf{l}, \mathbf{v}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{11.28}$$

如果我们假设方程中的 BRDF 也是常数，例如 Lambertian 表面，那么它也可以从积分中提出来，即：

$$L_o(\mathbf{v}) = \frac{\rho_{ss}}{\pi} L_l \int_{\mathbf{l} \in \Omega_l} v(\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{11.29}$$

为了确定被遮挡的光照，我们需要计算可见性函数乘上余弦项，在光源所对应的立体角上的积分。在某些情况下，这个积分可以通过解析计算出来。Lambert [967]推导了一个方程，用于计算一个球面多边形上的余弦积分。如果我们的面光源是多边形的，并且我们可以根据可见性表示来对其进行剪裁的话，那么我们只需要使用这个 Lambert 方程就可以得到精确的结果，如图 11.20 所示。例如：当我们选择视界角作为可见性表示的时候，就可以这么做。然而，如果出于某种原因，我们选择了其他的编码方式，例如环境圆锥（bent cone），此时再对光源进行裁剪将会产生圆形片段，因此我们将无法再使用 Lambert 方程。如果我们的面光源是非多边形的话，上述原则同样适用。

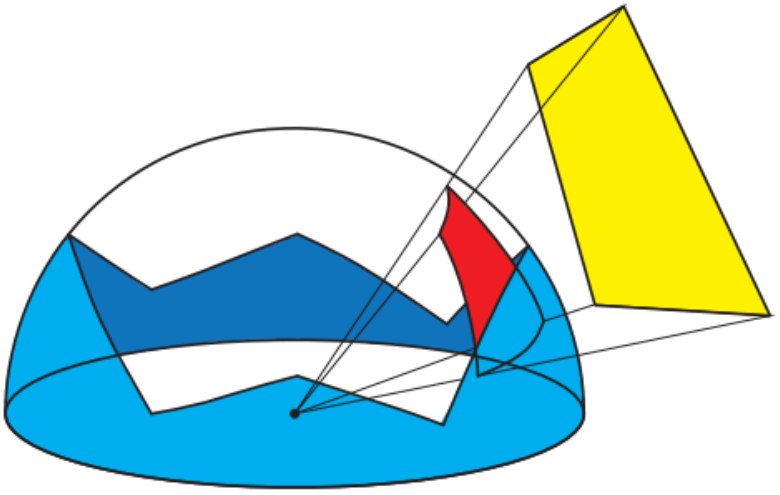


图 11.20：将一个黄色的多边形光源，投影到着色点的单位半球上，形成了一个球面多边形。如果使用视界映射来描述可见性的话，则可以对这个球面多边形进行裁剪，裁剪的结果使用红色进行表示。红色多边形的余弦加权积分可以使用 Lambert 方程进行解析计算。

还有另外一种可能的假设方法，即假设余弦项的值在整个积分域中是个常数。如果面光源的尺寸很小的话，那么这种近似是相当精确的。简单起见，我们可以使用面光源中心方向所对应的余弦值。这时，我们只需要计算可见性函数在光源立体角上的积分即可。下一步的操作，还是取决于我们所选择的可见性表示方法和面光源类型。如果我们使用球形光源，并且使用环境圆锥来表示可见性的话，那么积分的值就是可见性圆锥与光源圆锥相交部分所对应的立体角。这部分是可以解析计算的，Oat 和 Sander [1307]推导出了一种计算方法，虽然精确求解的方程相当复杂，但是好在他

们还提供了一个近似解，这个近似解在实践中十分有效。如果使用球谐函数来编码可见性的话，那么这个积分同样也可以解析计算。

对于环境光照而言，我们无法限制积分的范围，因为光照是来自四面八方的。我们需要找到一种方法来计算方程 11.26 中的完整积分。为了简单起见，让我们首先考虑 Lambertian BRDF：

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{11.30}$$

这个方程中的积分叫做三重乘积积分（triple product integral）。如果其中的单个函数可以使用特定的方式来进行表示的话（例如球谐函数或者小波），那么它是可以通过解析计算出来的。但不幸的是，这对于通常的实时应用程序而言太昂贵了，尽管这样的解决方案已经被证明，可以在简单的环境设置中以交互式帧率来运行[1270]。

不过，我们的这个例子稍微简单一些，因为其中一个函数是余弦函数。我们可以将方程 11.30 改写为：

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} \overline{L_i}(\mathbf{l}) v(\mathbf{l}) d\mathbf{l} \tag{11.31}$$

或者：

$$L_o(\mathbf{v}) = \frac{\rho_{\text{ss}}}{\pi} \int_{\mathbf{l} \in \Omega} L_i(\mathbf{l}) \bar{v}(\mathbf{l}) d\mathbf{l} \tag{11.32}$$

其中：

$$\begin{aligned} \overline{L_i}(\mathbf{l}) &= L_i(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+, \\ \bar{v}(\mathbf{l}) &= v(\mathbf{l}) (\mathbf{n} \cdot \mathbf{l})^+. \end{aligned}$$

与 $L_i(\mathbf{l})$ 和 $v(\mathbf{l})$ 一样，$\overline{L_i}(\mathbf{l})$ 和 $\bar{v}(\mathbf{l})$ 都是球面函数。我们没有尝试直接去计算这个三重乘积积分，而是首先将余弦项乘以 L_i （方程 11.31）或者 v_i （方程 11.32），这样做使得被积函数变成两个函数的乘积。虽然这看起来只是一个数学技巧，但是它可以极大地简化计算。如果被积函数中的乘积因子，使用了标准正交基（例如球谐函数）来进行表示，那么这个二重乘积积分可以很简单的计算出来，积分的结果就是它们系数向量的点积（章节 10.3.2）。

但是我们仍然需要计算 $\overline{L_i}(\mathbf{l})$ 或者 $\overline{v}(\mathbf{l})$，由于它们都包含了余弦项，因此要比完全一般的情况稍微简单一些。如果我们使用球谐函数来表示这些函数，那么余弦函数将会被投影到球带谐波（zonal harmonics，ZH）上。球带谐波是球谐函数的一个子集，其中每个频带只有一个系数是非零的（详见章节 10.3.2）。这个投影的系数有一个很简单的解析公方程[1656]。SH 和 ZH 的乘积计算效率，要比 SH 和 SH 的乘积高得多。

如果我们决定先将余弦项乘以 v（方程 11.32），那么我们可以在离线环境中对其进行预计算，同时只需要存储可见性即可。正如 Sloan 等人[1651]所描述的那样（章节 11.5.3），这是一种形式的预计算 radiance 传输（precomputed radiance transfer）。然而，在这种形式下，我们无法对法线进行任何精细的修改，因为由法线控制的余弦项已经和可见性函数融合在一起了。如果我们想要模拟精细尺度的法线细节，则可以先用 L_i 乘以余弦项（方程 11.31）。由于我们事先并不知道法线的具体方向，因此可以预先计算出不同法线所对应的乘积[805]，或者是在运行过程中动态执行乘法操作[809]。离线预计算 L_i 和余弦项的乘积意味着，我们对光照的任何修改都会受到限制，并且允许光照在空间上发生变化会消耗大量的内存。另一方面，在运行时计算这个乘积的开销也很高。Iwanicki 和 Sloan [809]描述了如何降低这一操作的成本，在他们的例子中，这个乘积可以在更低的粒度（顶点）上进行计算。乘积的结果与余弦项进行卷积，再投影到一个更简单的表示方法（AHD）上，然后再使用逐像素的法线进行插值和重建。这种方法允许他们在 60 FPS 的游戏中，使用 L_i 乘以余弦项的策略。

Klehm 等人[904]提出了一种使用环境贴图表示光照，并使用锥形编码可见性的解决方案。他们使用了不同大小的滤波核来对环境贴图进行过滤，这些滤波核代表了不同锥形开口的可见性与光照乘积的积分。他们按照锥形开口角度大小的增加，将结果存储在纹理的 mipmap 中。这样做是合理的，因为较大锥形开口的预过滤结果在球体上会平滑变化，因此不需要使用较高分辨率来进行存储。在预过滤的过程中，他们假设可见性锥的方向与法线是对齐的，这是一个近似假设，但是在实践中可以给出较为可信的结果。他们还分析了这种近似是如何对最终质量产生影响的。

如果我们需要处理光泽 BRDF 和环境光照，那么情况就要更加复杂了。此时我们无法再将 BRDF 从积分中提取出来，因为它并不是一个常数。为了解决这个问题，Green 等人[582]建议用一组球面高斯函数（spherical Gaussian）来对 BRDF 本身进行近似。这些球面高斯函数都是径向对称的，它们可以使用三个参数来进行表示（十分紧凑）：方向（或者平均值） $\mathbf{d}$，标准差 μ 和振幅 w。这个近似 BRDF 可以定义为球面高斯函数的和：

$$f(\mathbf{l},\mathbf{v}) \approx \sum_k w_k(\mathbf{v}) G\left(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}\right) \tag{11.33}$$

其中 $G(\mathbf{d},\mu,\mathbf{l})$ 是一个球面高斯波瓣，它指向方向 $\mathbf{d}$，锐度为 μ（详见章节 10.3.2）；w_k 是第 k 个波瓣的振幅。对于各向同性的 BRDF 而言，其波瓣的形状仅仅取决于法线方向和观察方向之间的夹角。我们可以将一组近似值存储在一维查找表中，并在运行时进行插值重建。

有了这个 BRDF 近似，我们可以将方程 11.26 改写成：

$$\begin{aligned} L_o(\mathbf{v}) &\approx \int_{\mathbf{l}\in\Omega} \sum_k w_k(\mathbf{v}) G\left(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}\right) L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \\ &= \sum_k w_k(\mathbf{v}) \int_{\mathbf{l}\in\Omega} G\left(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}\right) L_i(\mathbf{l}) v(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l}. \end{aligned} \tag{11.34}$$

Green 等人还假设可见性函数在每个球面高斯的范围内都是恒定的，这使得他们可以将可见性项从积分中提取出来。他们在波瓣的中心方向上计算了可见性函数，最终的方程形式如下：

$$L_o(\mathbf{v}) \approx \sum_k w_k(\mathbf{v}) v_k\left(\mathbf{d}_k(\mathbf{v})\right) \int_{\mathbf{l}\in\Omega} G\left(\mathbf{d}_k(\mathbf{v}), \mu_k(\mathbf{v}), \mathbf{l}\right) L_i(\mathbf{l}) (\mathbf{n}\cdot\mathbf{l})^+ d\mathbf{l} \tag{11.35}$$

剩余的积分代表了入射光线与球面高斯进行卷积，这个球面高斯是给定方向和给定标准差的。这个卷积的结果可以进行预先计算，并存储在一个环境贴图中，其中较大的 μ 所对应的卷积结果存储在较低的 mipmap 层级中。这里的可见性可以使用较低阶的球谐函数进行编码，或者是任何其他的表示方法，因为这里只需要进行点计算即可。

Wang 等人[1838]以类似的方式来对 BRDF 进行近似，不同的是他们以一种更加精确的方式来处理可见性。他们的表示方法允许在可见性函数的范围内，计算单个球面高斯函数的积分。他们使用这个积分值来引入一个新的球面高斯函数，它具有相同的方向和标准差，但是振幅不同，他们会在实际的光照计算中使用这个新的球面高斯函数。

对于某些应用程序而言，这种方法可能会过于昂贵。因为它需要从预过滤的环境贴图中进行多次采样，而纹理采样往往会成为渲染过程中的瓶颈。Jimenez 等人[835]和

El Garawany [414]给出了更简单的近似方法，为了计算遮挡因子，他们使用一个圆锥来表示整个 BRDF 波瓣，忽略了 BRDF 波瓣对观察角度的依赖，只考虑材质粗糙度等参数，如图 11.21 所示。它们将可见性近似为一个圆锥体，并计算可见性圆锥与 BRDF 圆锥相交部分的立体角，就像环境光圈照明所做的那样。这个计算出来的标量结果会用于对光照的衰减，虽然这是一个重大的简化，但是最终的结果看起来是可信的。

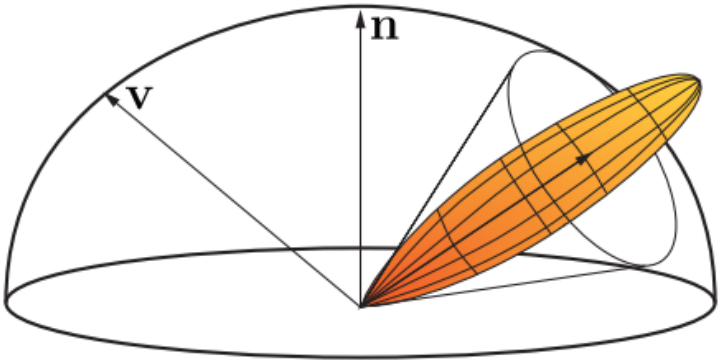


图 11.21：为了计算遮挡信息，光泽材料的镜面波瓣可以表示为一个圆锥。如果将可见性近似为另一个圆锥，那么遮挡因子可以通过这两个圆锥相交部分的立体角计算得来，这个方法与环境光圈照明相同（详见图 11.19）。这张图片展示了使用一个圆锥来表示 BRDF 波瓣的一般原理，但这仅仅是为了进行说明。在实践中，为了产生更加合理的遮挡效果，这个圆锥需要更宽。

11.5 漫反射全局光照

接下来几个小节将介绍的各种方法，它们不仅可以实时模拟遮挡效果，还可以模拟完整的光线弹射。它们可以大致分为两种算法，它们各自拥有不同的假设，即光线在到达眼睛之前，从一个漫反射表面反射回来，还是从一个镜面反射回来。相应的光线路径可以表示为 $L(D|S) * DE$ 或者 $L(D|S) * SE$，其中有许多方法都对早期的反弹类型进行了限制。第一组解决方案假设光线的入射方向在着色点上半球范围内平滑变化，或者直接忽略这种变化。第二组解决方案则假设光线的入射方向具有很高的变化率，这个假设依赖于这样的一个事实，即光线只会从一个相对较小的立体角中照射到着色点上。由于这两种约束条件的差别很大，因此将它们分开处理是有益的。我们在本小节中介绍漫反射全局照明的方法，在下一节中介绍镜面全局光照的方法，然后在最后一节中介绍未来很可能会应用的统一方法。

11.5.1 表面预照明（Surface Prelighting）

辐射度算法和路径追踪算法都是为离线使用而设计的。虽然已经有了一些在实时环境中使用它们的尝试，但是结果仍然太不成熟，无法用于实际的产品中。目前最为常见的做法是使用它们来预先计算与光照相关的信息。这个昂贵的离线过程是预计算的，计算出来的结果会被存储起来，然后在显示期间使用，从而提供高质量的光照效果。正如章节 11.3.4 中所述，以这种方式对静态场景进行预计算的过程被称为烘焙（baking）。

这种做法有一定的限制。如果我们提前进行光照计算，那么我们将无法在运行过程中更改场景的设置。场景中的所有几何体、灯光和材质都需要保持不变，我们无法改变一天中的时间，也不能在墙上炸一个洞。但是在许多情况下，这种限制是一种可以接受的权衡，例如：建筑可视化的相关应用可以假设用户只在虚拟环境中走动；游戏同样也会限制玩家的行动等。在这样的应用中，我们可以将几何物体分为静态物体（static）和动态物体（dynamic）。在预计算过程中会使用静态物体来计算光照，让它们与光源充分进行交互作用。比如静态的墙壁会投下阴影，静态的红地毯反射出红光。动态物体只会充当接收者，它们不会遮挡光线，也不会产生间接的光照效果。在这样的场景中，动态几何物体的尺寸通常会被限制得相对较小，这样可以忽略它们对光照的影响，或者是使用其他技术来进行建模，从而最小化光照质量的损失。例如：动态几何物体可以使用屏幕空间中的一些方法来生成遮挡效果。常见的动态物体包括角色、装饰性的几何体以及车辆等。

可以进行预计算的、最简单的照明信息就是 irradiance。对于一个平坦的 Lambertian 表面，irradiance 和表面颜色一起，完整描述了材质对于光照的反应。因为光源的照明效果是彼此独立的，所以动态光源可以被添加在预计算的 irradiance 之上，如图 11.22 所示。

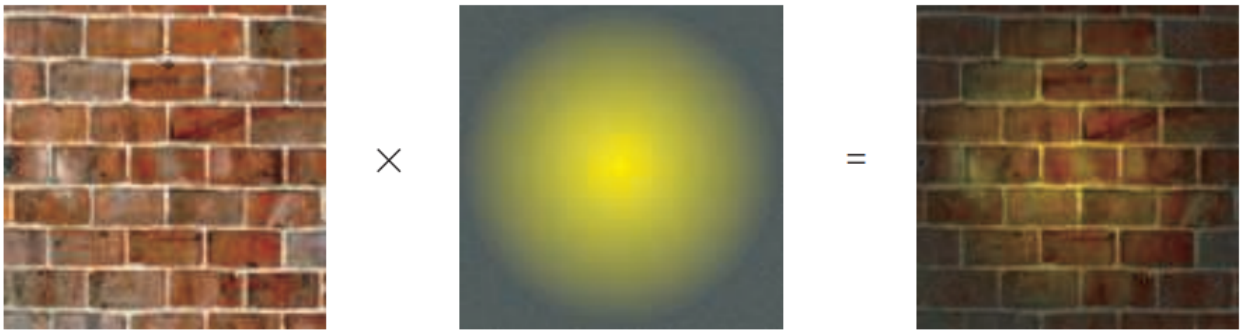


图 11.22：对于一个法线已知的 Lambertian 表面，其 irradiance 可以预先计算出来。在运行过程中，将这个值乘以实际的表面颜色（例如纹理颜色），从而获得反射的 radiance。根据表面颜色的确切形式，可能还需要额外除以 π 来确保能量守恒。

1996 年的《雷神之锤》和 1997 年的《雷神之锤 2》是第一个使用预计算 irradiance 的商业交互式应用程序。Quake 预先计算了静态光源的直接贡献，这主要是作为一种提高性能的方法。《雷神之锤 2》还计算了一个间接分量，使其成为第一款使用全局光照算法来生成更加真实光照的游戏。它使用了一种基于辐射度的算法，因为这种技术非常适合用于计算 Lambertian 环境中的 irradiance。此外，由于内存的时间限制，光照的分辨率相对较低，这与辐射度解决方案中典型的模糊、低频阴影匹配得很好。

预计算的 irradiance 通常会和漫反射颜色或者 albedo 贴图相乘，并单独存储在一个纹理集合中。虽然在理论上可以预先计算出辐射出度（exitance，等于 irradiance 乘以漫反射颜色），并将其存储在一组纹理中，但是在大多数实际情况下，许多应用都没有采用这个做法。因为颜色贴图的使用频率通常会很高，它们利用了各种类型的分块平铺，并且其中的部分区域经常会在模型之间进行重复使用，所有的这些操作都是为了保持合理的内存使用。而 irradiance 的使用频率则要低得多，重复使用的情况也很少。因此将光照信息和表面颜色分开存储，可以消耗更少的内存空间。

除了限制最为严格的硬件平台之外，如今已经很少使用预计算 irradiance 的方法了。因为根据定义，irradiance 是针对给定的法线方向进行计算的，这意味着我们无法对物体的表面法线进行修改，我们无法使用法线映射来提供高频的表面细节。这也意味着只能对平面进行预计算 irradiance。如果我们需要在动态几何物体上使用烘焙光照，我们就需要其他的方法来存储这些光照信息。这些限制条件促使人们寻找一种方法，来存储带有方向分量的预计算光照。

11.5.2 定向表面预照明

为了在 Lambertian 表面上使用预照明和法线映射，我们需要一种方法来表示 irradiance 随表面法线的变化。为了给动态几何物体提供间接光照，我们还需要在每个可能的表面方向上进行计算。幸运的是，我们已经有了各种工具可以用于表示这样函数。在章节 10.3 中，我们描述了根据法线方向确定光照的各种方法。这些方法中包括了针对半球函数域的专门解决方案，就像不透明表面的情况一样，球体下半部分的值是无关紧要的。

最常用的方法是存储完整的球面 irradiance 信息，例如使用球谐函数来进行存储。该方案首先是由 Good 和 Taylor [564]在加速光子映射（photon mapping）的背景下提出的，并被 Shopf 等人[1637]在应用在了实时场景中。在这两种情况下，定向的 irradiance 都会存储在纹理中。如果采用 9 个球谐系数（即三阶 SH），可以获得较好的质量，但是存储和带宽的成本较高。如果只使用四个系数（即二阶 SH）的话，

虽然成本较低，但是会丢失许多细节信息，光线的对比度较低，法线贴图也不太明显。

Chen [257]使用了《光环 3》方法的一种变体，这种方法的目的是以较低的成本来实现三阶 SH 的质量。他从球面信号中提取出最主要的光照，并将其分离存储为一个颜色和一个方向。剩余的基底则使用二阶 SH 来进行编码，使用这种方法，可以将系数的数量从 27 个减少到 18 个，而且质量损失很小。Hu [Lightmap Compression in Halo 3", Yaohua Hu, GDC 2008]描述了如何对这些数据进行进一步地压缩。Chen 和 Tatarchuk [258]在生产环境中使用了基于 GPU 的烘焙管线，他们提供了进一步的信息。

Habel 等人[627]所提出的 H-basis 是另一种可选的解决方案。由于它只对半球面上的信号进行编码，因此可以使用较少的系数提供与球谐函数相同的精度。仅仅使用六个系数就可以获得与三阶 SH 相当的质量。因为 H-basis 只会针对一个半球进行定义，所以我们需要表面上的一些局部坐标系来正确地确定它的朝向。通常，由 uv 参数化所产生的切线坐标系可以用于此目的。如果想要在纹理中存储 H-basis 的分量，那么纹理的分辨率应当足够高，从而适应底层切线空间的变化。如果不同切线空间中的多个三角形覆盖了同一个纹素，那么重建出的信号将会是不精确的。

球谐函数和 H-basis 的一个问题是，它们都会出现振铃现象（章节 10.6.1）。虽然预过滤可以减轻这种现象，但它也会使光照变得更加平滑，这可能并不总是我们想要的。此外，即使是成本较低的变体方法，在存储和计算方面仍然具有相对较高的成本。在一些限制更加严格的情况下，例如在低端平台或者虚拟现实平台上，这种开销可能会令人望而却步。

成本是那些简单替代方案最重要的流行原因。《半条命 2》使用了一个自定义的半球基底（章节 10.3.3），每个样本存储了三个颜色值，总共有九个系数。尽管 AHD（章节 10.3.3）很简单，但它也是一个较为流行的选择，它被用在了许多游戏中，例如使命召唤系列[809, 998]和《最后生还者》[806]，如图 11.23 所示。

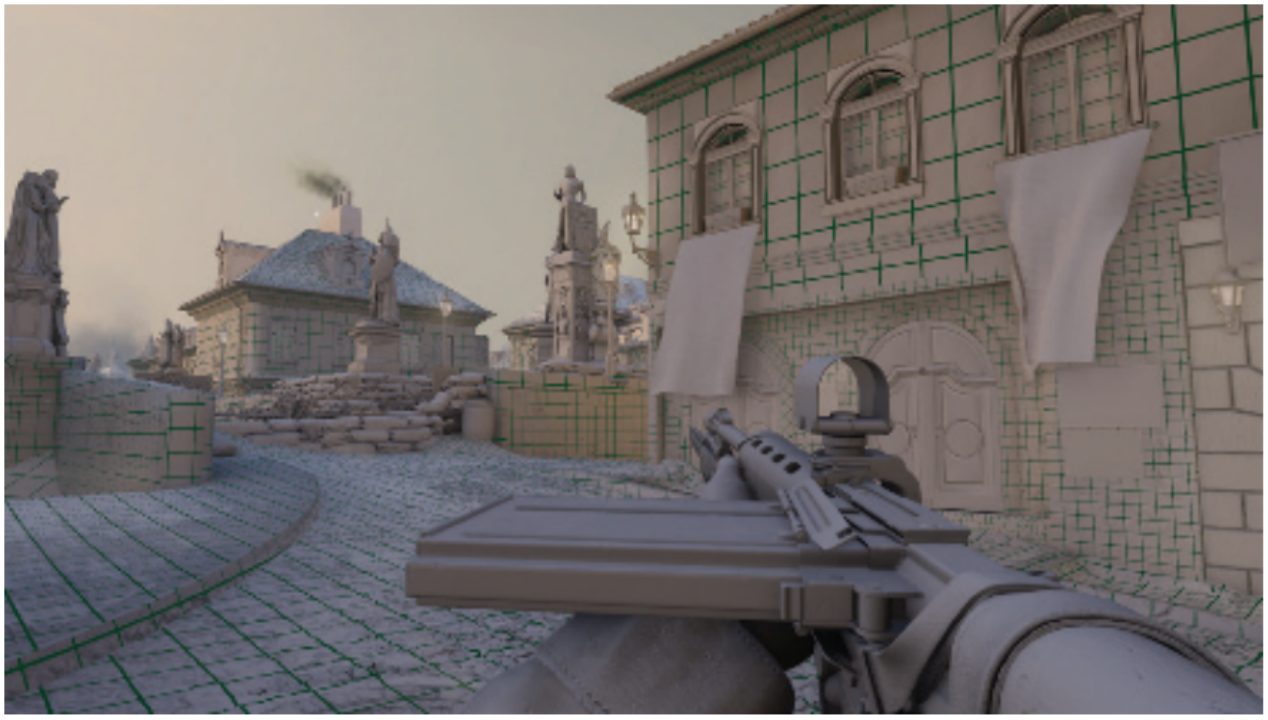


图 11.23：《使命召唤：二战》使用 AHD 表示方法来编码光照贴图中光照的方向变化。图中展示的绿色网格用于在调试（debug）模式下可视化光照贴图的密度，其中的每个方块都对应了一个单独的光照贴图纹素。

Crytek 在游戏《孤岛惊魂》[806]中使用了一个变体。这种 Crytek 表示方法包含了切线空间中的平均光线方向、平均光线颜色和一个标量的方向因子。其中最后一个值用于混合环境（ambient）项和定向（directional）项，它们都使用了相同的颜色。这样可以将每个样本的存储空间减少到 6 个系数：3 个系数用于颜色，2 个系数用于方向，1 个用于方向因子。Unity 引擎在它的其中一个模式中也使用了类似的方法[315]。

这种类型的表示方法是非线性的，这意味着，在技术上而言，对单个组件进行线性插值（无论是在纹素之间还是顶点之间）在数学上是不正确的。如果主要光源的方向变化很快，例如在阴影边界上变化很快，那么在阴影中很可能会出现视觉瑕疵。尽管有这些不准确的地方，但是最终的结果在视觉上还是令人满意的。由于环境光照和定向光照区域之间具有较高的对比度，法线贴图的效果会被增强，这通常是我们想要发生的。此外，定向光照的分量还可以用于计算 BRDF 的高光响应，这可以为低光泽材质的环境贴图提供一种低成本的替代方案。

在这类算法谱系的另一端，是为高质量视觉表现而设计的方法。Neubelt 和 Pettineo [1268]在游戏《教团：1886》中，使用纹理贴图来存储球面高斯函数的系数，如图 11.24 所示。他们存储的是入射 radiance，而不是 irradiance，radiance 会被投影到一组高斯波瓣上（章节 10.3.2），它被定义在一个切线坐标系中。根据具体

场景中光照的复杂程度，他们会使用 5 到 9 个波瓣。为了产生漫反射响应效果，球面高斯函数会与沿表面法线方向的余弦波瓣进行卷积。通过将高斯函数与镜面 BRDF 波瓣进行卷积，这种表示方法也足够精确，可以提供低光泽的高光效果。Pettineo 详细描述了整个系统[1408]，他还提供了一个应用程序的源代码，这个应用程序能够烘焙和渲染不同的光照表示方法。

图 11.24：《教团：1886》在光照贴图中存储了投影到一组球面高斯波瓣上的入射 radiance。在运行过程中，使用 radiance 与余弦波瓣进行卷积来计算漫反射响应（左图），与适当形状的各向异性球面高斯进行卷积来生成高光响应（右图）。

如果我们需要着色点任意方向上的光照信息，而不仅仅是在着色点是半球范围内的光照信息（例如：为动态几何物体提供间接照明），那么我们可以使用一些编码完整球面信号的方法。这里自然而然会提到球谐函数。当不太关心内存开销的时候，三阶 SH（每个颜色通道有 9 个系数）是最流行的选择；否则也可以使用二阶 SH（每个颜色通道有 4 个系数，这与 RGBA 纹理的通道数量相匹配，因此一个贴图可以存储一个颜色通道的 SH 系数）。球面高斯函数也适用于完整球体的情况，因为波瓣可以分布在整个球体上，或者也可以只分布在法线周围的半球上。然而，由于需要被波瓣覆盖的立体角是球面技术的两倍，因此可能需要使用更多的波瓣来保持相同的质量。

如果我们想避免处理振铃问题，同时又负担不起使用大量波瓣所带来的开销，那么环境立方体（章节 10.3.1）是一个可行的选择[1193]。它由六个 clamped cos^2 波瓣组成，它们都沿着主轴方向。每个余弦波瓣只覆盖一个半球，即它们具有局部性（local support），这意味着它们只在其球面域的一个子集上具有非零值。因此，在重建过程中只需要使用 6 个存储值中的 3 个可见波瓣即可，这限制了光照计算的带宽成本。其重建质量与二阶球谐函数相类似。

环境骰子[808]（章节 10.3.1）可以生产比环境立方体更高质量的结果。该方案采用了 12 个沿二十面体顶点方向的波瓣，这些波瓣是 cos^2 和 cos^4 波瓣的线性组合。在重建期间会使用 12 个存储值中的 6 个，其重建质量可以与三阶球谐函数相媲美。这些表示方法和其他的类似表示方法（例如：由三个 cos^2 波瓣和一个 cos 波瓣所组成

的基底，它们被扭曲从而覆盖整个球面）已经在许多商业成功的游戏中进行了使用，例如《半条命 2》[1193]，使命召唤系列[766, 808]，《孤岛惊魂 3》[533]，《全境封锁》[1694]和《刺客信条 4：黑旗》[1911]等。

11.5.3 预计算传输

虽然上述的预计算光照看起来很惊艳，但是它本质上还是静态的。任何几何物体或者光照的改变都会使整个解决方案失效。就像在现实世界中一样，拉开窗帘（场景中几何物体的局部变化）可能会让整个房间充满光线（光照的全局变化）。人们花费了大量的研究工作来寻找能够允许某些类型变化的解决方案。

如果我们假设场景中的几何物体没有发生变化，只有光照发生了变化，那么我们可以对光线与模型的相互作用进行预计算。物体之间的影响（例如相互反射或者次表面散射），可以预先进行一定程度的分析，并将结果存储下来，而不需要对实际的 radiance 进行操作。接收入射光线，并将其转换为整个场景的 radiance 分布，这个函数被称为传输函数（transfer function）。这样的方法被称为预计算传输（precomputed transfer）或者预计算 radiance 传输（precomputed radiance transfer，PRT）。

与之前所介绍的完全离线的烘焙光照不同，这类技术确实具有明显的运行时间开销。当在屏幕上显示场景时，我们需要计算特定光照环境中的 radiance。为了实现这一点，我们需要将一定数量的直接光源“注入”到系统中，然后应用传输函数来将其传播到整个场景中。有些方法会假设这种直接光照来自于环境贴图，还有一些其他的解决方案允许任意的光照设置，并且能够以灵活的方式来进行改变。

Sloan 等人[1651]将预计算 radiance 传输的概念引入了图形学，他们使用球谐函数来描述它，但是这个方法其实不必使用球谐函数。该方法的基本思想很简单，如果我们使用一定数量（最好是数量较少）的“构件（building block）”光源来描述直接光照，那么我们就可以对场景如何被这些光源单独照亮进行预计算。想象一下，现在房间里有三台电脑显示器，每个显示器只能显示一种颜色，但是其亮度可以发生变化。我们将每个显示器的最大亮度设为 1，即归一化的“单位”亮度。我们可以独立地预计算每个显示器对房间照明的影响，这个过程可以使用章节 11.2 中介绍的方法来完成。因为光线的传输是线性的，所以三个显示器同时照亮场景的结果，就相当于每个显示器直接或者间接发出的光线总和。并且显示器的光照彼此相互独立，互不影响，因此如果我们将其中一个显示器设置为最大亮度的一半，那么这样做只会改变这个显示器对总照明的贡献，并不会影响其他的显示器。

这样做允许我们快速计算整个房间内的全部反弹光线。我们将每个预计算的光源解决方案，乘上显示器的实际亮度，然后再对这些结果进行求和。我们可以打开或者关闭显示器，让它们变得更亮或者更暗，甚至是改变它们的颜色，想要获得最终的光照效果，我们只需要做这些乘法和加法即可，如图 11.25 所示。

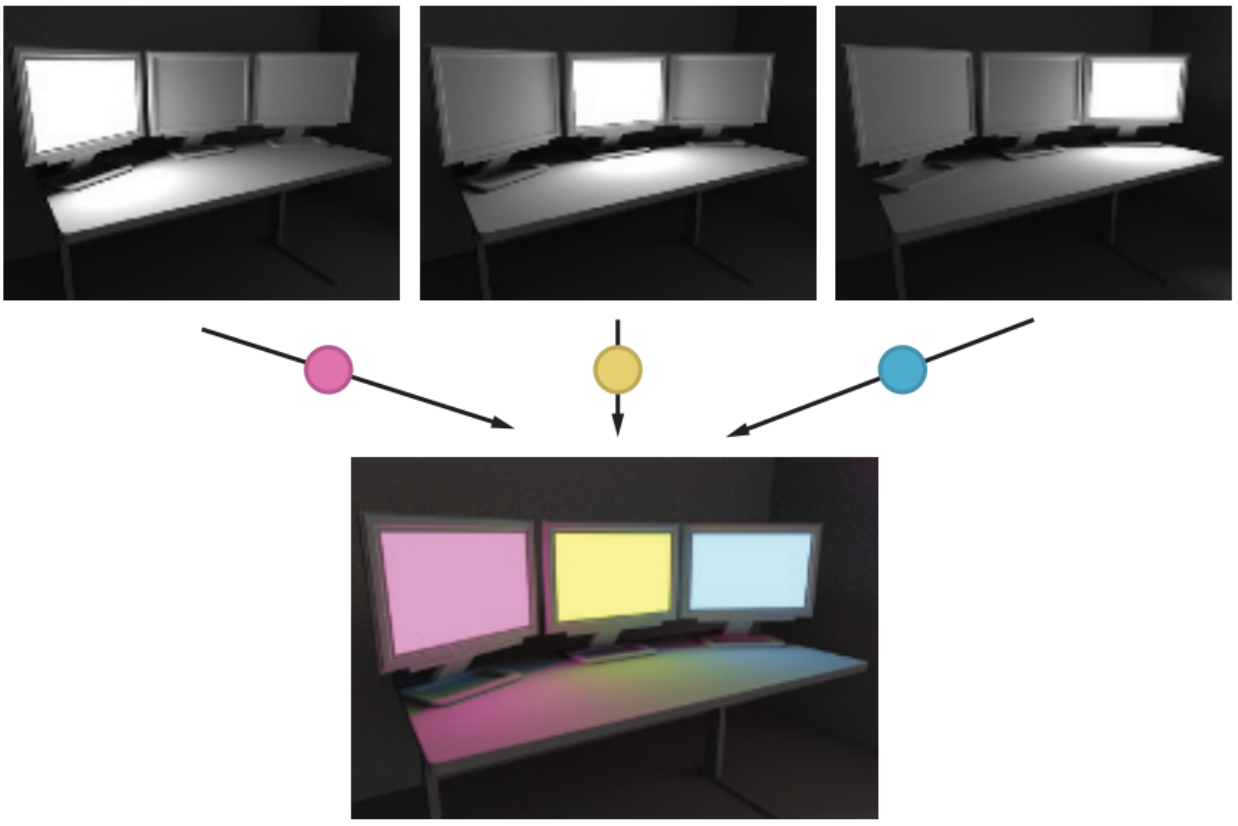


图 11.25：使用预计算 radiance 传输的渲染示例。会预先计算三个显示器的完整光照传输，分别获得一个归一化的“单位”响应。由于光线传输的线性叠加特点，这些单独的解可以分别乘以显示器的颜色（本例中是粉色、黄色和蓝色），从而获得最终的光照效果。

上述过程可以写出如下数学形式：

$$L(\mathbf{p}) = \sum_i L_i(\mathbf{p}) \mathbf{w}_i \tag{11.36}$$

其中 $L(\mathbf{p})$ 是点 $\mathbf{p}$ 的最终 radiance； $L_i(\mathbf{p})$ 是来自显示器 i 的预计算单位（归一化）贡献； $\mathbf{w}_i$ 是该显示器的当前亮度。这个方程在数学意义上定义了一个向量空间（vector space）， L_i 是这个空间中的基向量。任何可能的光照效果，都可以通过这些光源贡献的线性组合来生成。

Sloan 等人[1651]的原始 PRT 论文使用了与上文相同的推理过程，但是具体的背景有所不同，他们使用球谐函数来表示的无限远的环境光照。同时，他们没有存储场景对

显示器的响应，而是存储场景对于周围光线的响应，并使用球谐函数来定义周围光线的分布。通过对一些 SH 波段进行这样的操作，他们可以渲染一个被任意光照环境照亮的场景。他们将这种光照投影到球谐函数上，将每个结果系数乘以其各自的归一化“单位”贡献，然后再将结果加在一起，就像是我们对显示器所做的那样。

请注意，用于将光线“注入”到场景中的基底表示，与用于表达最终光照的表示是独立的。例如：我们可以使用球谐函数来描述场景是如何被照亮的，但是选择另一种基底来存储到达任意给定点上的 radiance。假设我们使用一个环境立方体来进行存储，我们会计算有多少 radiance 会从顶部到达着色点，有多少 radiance 会从两侧到达着色点。每个方向上的传输都会单独进行存储，而不是作为表示总传输的单个标量值。

Sloan 等人[1651]的 PRT 论文分析了两个案例。第一种是当接收基底只是表面上的一个标量 irradiance 时，此时接收物需要是一个完全漫反射的表面，并且需要具有预先定义好的法线，这意味着它无法使用法线贴图来获取精细尺度的细节。传输函数的形式是：输入光照的 SH 投影与预计算传输向量之间的点积。其中后者可以在整个场景中进行空间变化。

如果我们需要渲染非 Lambertian 材质，或者要使用法线映射，那么我们可以使用第二种变体。在这种情况下，周围光照的 SH 投影会被转换为给定点入射 radiance 的 SH 投影。因为这个操作为我们提供了整个球面上（或者半球，如果我们处理的是静态不透明物体的话）的完整 radiance 分布，我们可以将其与任何 BRDF 进行正确地卷积。此时的传输函数会将 SH 向量映射到其他的 SH 向量上，它具有一个矩阵乘法的形式，但是这种乘法操作的成本是很高的，无论是计算量还是内存开销。如果我们对源基底和接收基底都使用三阶 SH，那么我们需要为场景中的每个点都存储一个 9×9 的矩阵，并且这些数据仅仅用于黑白（monochrome）传输。如果我们想要实现彩色效果，那么我们就需要 3 个这样的矩阵，这样每个点需要的内存量就十分惊人了。

一年以后，Sloan 等人[1652]解决了这个问题。他们没有直接存储传输向量或者传输矩阵，而是使用主成分分析（principal component analysis，PCA）技术对整个集合进行了分析。这里的传输系数可以被认为是多维空间中的点（例如： 9×9 矩阵意味着空间是 81 维），但是这些点在该空间中并不是均匀分布的。它们会形成维数较低的簇，这种聚类就像是沿着直线分布的三维点一样，实际上它们都位于三维空间的一维子空间中。PCA 可以有效地检测出这种统计意义上的关系。一旦 PCA 发现了子空间，就可以使用更少的坐标来表示这些点，因为我们可以使用更少的维度来存储子空间中的位置。用刚才的直线例子来类比，我们不需要使用三个坐标来存储一个点的完整位置信息，我们只要存储该点沿直线的距离即可。Sloan 等人使用这种方法，将传

输矩阵的维度从 625 维（25×25 传输矩阵）降低到 256 维。虽然这个维度对于常见的实时应用程序而言还是太高了，但是它为后续方法拓展了思路，许多后来的光线传输算法均采用了 PCA 来作为数据压缩的一种方式。

这种降维存储本质上是有损压缩的。在极少数情况下，数据点会形成完美的子空间；但是大多数情况下它只能对原始数据进行近似，因此将原始数据投影到子空间中的数据上会导致一些退化。为了提高质量，Sloan 等人将一组传输矩阵划分为若干个簇，分别对每个簇进行 PCA 操作。这个过程还包括了一个优化步骤，以确保聚类边界上不会出现不连续现象。他们还提出了一种允许物体发生有限形变的扩展变体，被称为局部可变形预计算 radiance 传输（local deformable precomputed radiance transfer，LDPRT）[1653]。

PRT 已经在一些游戏中以各种形式进行了使用。PRT 在玩法侧重于户外区域的游戏中尤其受欢迎，因为这些区域的时间和天气条件都是动态变化的。《孤岛惊魂 3》和《孤岛惊魂 4》都使用了 PRT，其中源基底是二阶 SH，接收基底是一个自定义的四方向基底[533, 1154]。《刺客信条 4：黑旗》使用一个基底函数作为来源（太阳颜色），在一天中的不同时间中对传输进行了预计算。这种表示方式可以理解为在时间维度上来定义源基底函数，而不是在方向维度上。《刺客信条 4：黑旗》中的接收基底与《孤岛惊魂》系列中所使用的相同。

SIGGRAPH 2005 关于预计算 radiance 传输的课程[870]，对这个领域的研究进行了很好地综述。Lehtinen [1019, 1020]给出了一个数学框架，这个框架可以用来分析各种算法之间的差异，并据此开发新的算法。

原始 PRT 方法假设周围的光照是无限远的。虽然这个模型可以很好的模拟室外场景的光照效果，但是它对室内环境的限制太大了。然而，正如我们之前所提到的，这里的核心概念是：光照的初始来源是完全不可知的。Kristensen 等人[941]描述了一种方法，该方法对一组分散在整个场景中的光源进行了 PRT 计算。这对应了存在大量的“源”基底函数，然后这些光源会被组合成聚类，接收光照的几何物体也会被划分到若干个区域中，每个区域中的物体都会受到不同光源子集的影响。这个过程会显著压缩传输的数据。在运行过程中，会通过从预计算集合中对最近的光源进行插值，从而来近似计算放置在任意位置上的光源所产生的光照效果。Gilabert 和 Stefanov [533]在游戏《孤岛惊魂 3》中使用了这种方法来生产间接光照效果。但是这种方法的基本形式只能处理点光源，无法处理其他类型的光源。虽然这类方法也可以进行扩展，从而支持其他类型的光源，但是其开销会随着每个光源的自由度成指数级增长。

到目前为止所讨论的 PRT 技术预计算了来自一些元素之间的传输函数（向量和矩阵），然后会将其用于模拟光源。另一类流行的方法是对表面之间的传输进行预计

算，在这种类型的系统中，光照的实际来源变得无关紧要。可以使用任何类型、任意位置的光源，因为这类方法的输入是来自某些表面集合的出射 radiance（或者其他相关的物理量，例如 irradiance，如果方法假设只存在漫反射表面的话）。这些直接光照的计算，可以使用阴影（第 7 章）、irradiance 环境贴图（章节 10.6），或者本章前面所讨论的环境光遮蔽和定向遮蔽等方法。场景中的任何表面，可以通过设置其出射 radiance，来将其转换为一个面光源。

根据这些原则设计实现的系统，其中最流行的就是由 Geomerics 开发的 Enlighten，如图 11.26 所示。虽然该算法的确切细节从未完全公开过（不开源），但是许多演讲和演示都准确描述了该系统的原理[315, 1101, 1131, 1435]。这个系统应用于早期版本的 Unity 引擎中。

图 11.26：由 Geomerics 实现 Enlighten 系统的可以实时生成全局光照效果。这张图片展示了它与 Unity 引擎集成的一个例子。用户可以自由地改变一天中的时间，打开或者关闭光源。所有的间接光照都会实时更新。

为了实现光线传输的目的，我们假设场景中的表面都是 Lambertian 的。使用 Heckbert 的符号表示法，这里我们处理的路径集合是 $LD*(D|S)E$，因为眼睛所看到的最后一个表面不需要是纯漫反射的，只是在计算光线传输的时候，光线会在场景中的漫反射表面上进行弹射预计算。系统定义了一组“源”元素和另一组“接收”元素。源元素存在于表面上，并共享表面上的一些属性，例如漫反射颜色和表面法线。预处理步骤会计算光线在源元素和接收元素之间的传输情况和传输信息。这种信息的确切形式取决于源元素具体是什么，以及用于在接收器上收集光照的基底是什么。在最简单的形式中，源元素可以是点，然后我们会在接收位置处生成 irradiance；在这

种情况下，传输系数就是源和接收物之间的相互可见性。在运行过程中，会将所有源元素的出射 radiance 提供给系统，根据这些信息，我们可以利用预计算的可见性，以及已知的源和接收物的位置、方向等信息，来对反射方程（方程 11.1）进行数值积分。使用这种方法，就完成了光线的一次弹射，由于大多数间接光照效果都来源于第一次弹射，因此仅仅执行一次弹射就足以提供合理的光照效果了。然而，我们可以使用这个光线，再次运行传播步骤来生成第二次反弹的光线。这个过程通常是在几帧中完成的，其中上一帧的输出会作为下一帧的输入。

使用点作为源元素会产生大量的连接（connection）。为了提高性能表现，法线和颜色相似区域的聚类（簇）也可以用作源集合。在这种情况下，传输系数与辐射度算法中所看到的形状因子相同（章节 11.2.1）。请注意，尽管二者有相似之处，但是该算法与经典的辐射度算法是不同的，因为它每次只会计算一次弹射的光线，并且不涉及求解线性方程组的问题。该算法借鉴了渐进辐射度（progressive radiosity）的思想[275, 1642]。在这个系统的一次迭代过程中，一个 patch 可以确定它能够从其他 patch 接收到多少能量。将 radiance 传输到接收位置的过程被称为聚集（gathering）。

接收元素处的 radiance 可以使用不同的形式进行聚集。向接收元素的传输过程，可以使用我们之前所描述过的任何定向基底。在这种情况下，原来的单个系数会变成一个向量，其维数等于接收基底中的函数数量。当使用定向表示方法来执行聚集操作的时候，生成的结果与章节 11.5.2 中所描述的离线解决方案相同，因此它可以与法线映射方法一起使用，也可以提供低光泽材质的高光响应。

在许多变体中都使用了这个思想。为了节省内存，Sugden 和 Iwanicki [1721]使用了 SH 传输系数，对它们进行了量化，并将它们间接地存储为调色板中某个记录（entry）的索引（index）。Jendersie 等人[820]构建了一个包含源 patch 的层次结构，当子 patch 所覆盖的立体角太小时，会将高层元素的引用存储在这个树中。Stefanov [1694]引入了一个中间步骤，其中表面元素的 radiance 首先会传播到场景的体素化表示中，然后再作为传输的源。

（在某种意义上）将表面划分为源 patch 的理想分割方式，取决于接收物的位置。对于距离较远的元素而言，将它们作为独立的实体会产生不必要的存储成本，但是当近距离观察它们的时候，则应当将其单独对待。层次化的源 patch 在一定程度上缓解了这个问题，但是并不能完全解决它。能够为特定接收物进行组合的 patch，它们可能要相距足够远才能防止这种合并。Silvennoinen 和 Lehtinen [1644]提出了一种解决该问题的新方法。该方法没有显式地创建源 patch，而是为每个接收位置生成一组不同的 patch。物体会被渲染到散布在场景周围的一组稀疏环境贴图中。每个贴图都会

被投影到球谐函数上，而这个低频版本则会“虚拟（virtually）”投影回环境中。接收点会记录它们能够看到多大的投影，并且这个过程会针对每个发送者的 SH 基函数分别完成。这样做会根据环境探针（probe）和接收点的可见性信息，为每个接收物创建一组不同的源元素。

由于源基底是由投影到 SH 的一个环境贴图构成的，因此它很自然地结合了更远的表面。为了选择要进行使用的探针，接收物会使用一种倾向于附近的探针的启发式方法，这使得接收物可以以相似的尺度来“观察”环境。为了限制必须存储的数据量，会使用聚类 PCA 对传输信息进行压缩。

Lehtinen 等人[1021]描述了另一种形式的预计算传输方法。在这种方法中，源元素和接收元素都不位于网格上，而是位于体积中，因此可以在三维空间中的任何位置上进行查询。这种形式可以很方便地在静态物体和动态物体之间提供一致的光照效果，但是其计算量相当大。

Loos 等人[1073]预计算了具有不同侧壁（side wall）配置的、模块化的、单元格内的传输。然后将多个这样的单元格缝合和扭曲，从而对场景的几何形状进行近似。radiance 首先会传播到单元格边界处，然后使用预计算模型来将其传播到邻近的单元格中。这种方法的计算速度很快，即使是在移动平台上也可以有效运行，但是其结果质量较低，可能无法满足要求更高的应用程序。

11.5.4 存储方法

无论我们是想使用完全预计算的光照，还是对传输信息进行预计算，从而允许一些光照的变化，其生成的结果数据都必须要以某种形式进行存储，同时这种形式必须是 GPU 友好的。

光照贴图（light map）是存储预计算光照最常用的方法之一，它们是存储了预计算信息的纹理。虽然有时像 irradiance 贴图这样的术语，会用来表示存储特定类型的数据，但是术语光照贴图可以对这些数据和纹理进行统称。在运行过程中，会使用 GPU 内置的纹理机制，获取到的值通常都是双线性过滤的，这在某些情况下可能并不是完全正确的。例如，当我们使用 AHD 表示方法时，经过滤波后的 D（方向）分量在插值之后将不再是单位长度，因此需要重新对其进行归一化。使用插值也意味着 A（环境）和 H（高亮）与我们在直接在采样点计算它们所获得的结果相比，也并不是完全相同的。但是，即使表示方法是非线性的，但是结果通常看起来也还能接受。

在大多数情况下，光照贴图都不会使用 mipmap，通常而言都是没有必要的，因为与常见的 albedo 贴图或者法线贴图相比，光照贴图的分辨率都很小。即使在一些高质

量的应用程序中，光照贴图中单个纹素所覆盖的面积至少也有 20×20 厘米，甚至是更多。对于这种尺寸的纹素而言，几乎不需要添加额外的 mipmap 层级。

为了在纹理中存储光照信息，常见中的模型物体需要提供一个唯一的参数化（unique parameterization）。在将一个漫反射颜色纹理映射到一个模型上的时候，对于网格的不同部分使用相同的纹理区域，这样做通常是比较好的，尤其是当模型被一个包含重复图案的贴图纹理化时。但是想要重复使用光线贴图是非常困难的，网格上每个点的光照情况都是唯一的，因此场景中的每个三角形，都需要在光照贴图上占据一块属于自己的唯一区域。创建一个参数化的过程，最开始是将网格分割为更小的块，这可以使用一些启发式方法来自动完成[1036]，也可以在创作工具中手动完成。通常情况下，其他纹理映射中已经包含了一部分的分割信息，这部分信息也会被使用。接下来，每个块都会被独立地参数化，从而确保它在纹理空间中不会发生重叠[1057, 1617]。在纹理空间中产生的元素会称为图表（chart）或者壳（shell）。最后，所有生成的 chart 都会被打包到同一个纹理中，如图 11.27 所示。

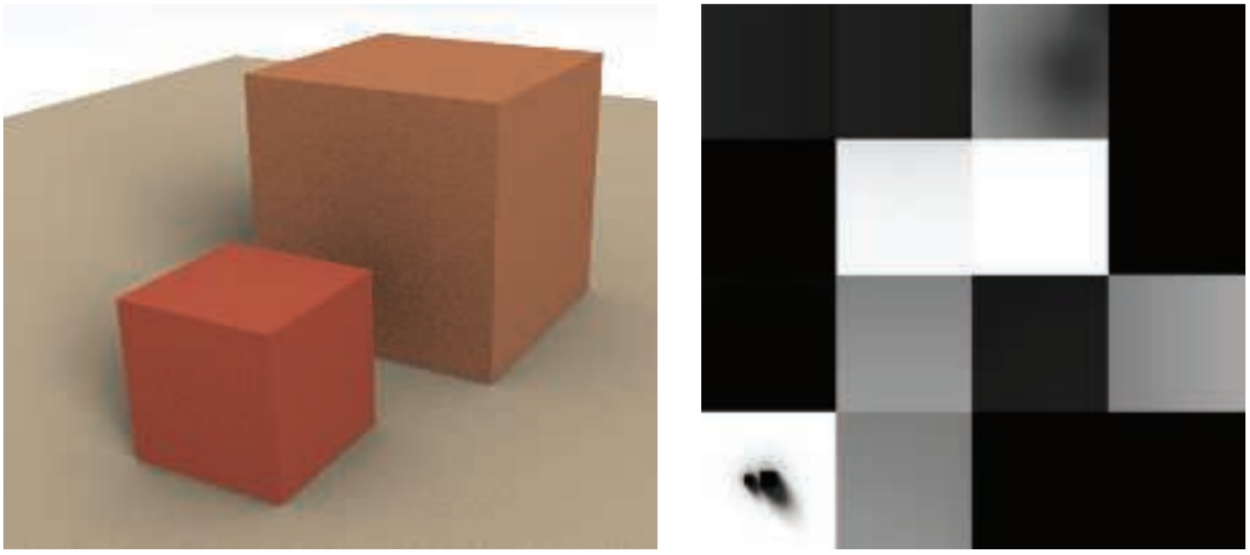


图 11.27：光照信息被烘焙到一个场景中，将光照贴图应用到物体表面上从而实现光照。光照贴图使用了一个唯一的参数化。场景会被划分成多个元素，这些元素被展开并打包成一个共同的纹理。例如：右图左下角的小块对应了地面，它展示了两个立方体的阴影。[218]

必须要小心确保 chart 之间不会发生重叠，并且 chart 之间的过滤占用空间（filtering footprint）也必须保持相互独立。当渲染一个给定 chart 的时候（双线性过滤会访问四个相邻的纹素），其他所有可以被访问的纹素都应该被标记为已使用，这样就不会有其他 chart 与它们发生重叠。否则，chart 之间可能会出现颜色溢出现象，即其中一个 chart 的光照可能会出现在另一个 chart 中。对于光照贴图系统来说，提供一个用户可以控制的“排水沟（gutter）”量，用于调整光线贴图 chart 之间的间距，虽然这种做法十分常见，但是这种 chart 的分离是没有必要的。一个 chart

正确的过滤占用空间，可以通过使用一套特殊的规则，在光照贴图空间中通过光栅化来自动确定，如图 11.28 所示。如果以这种方式光栅化的 shell 不会发生重叠，那么我们就可以保证不会发生颜色溢出现象。

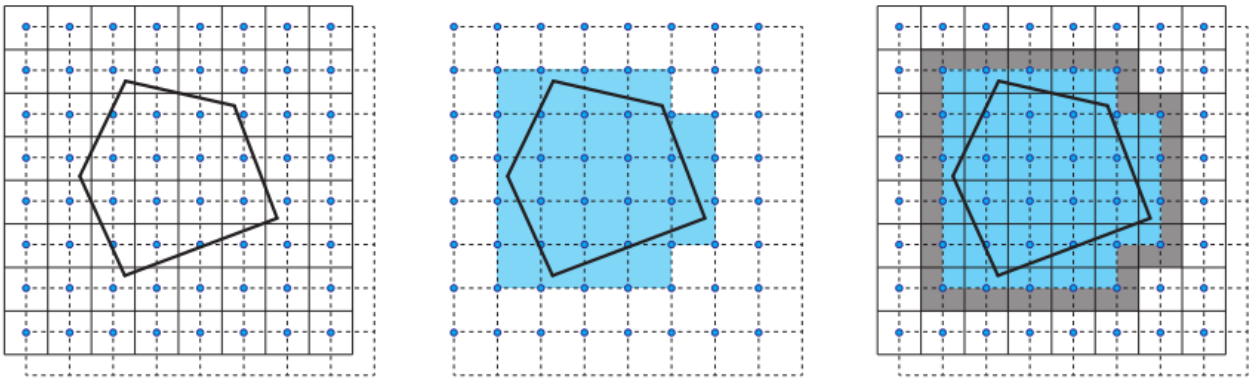


图 11.28：为了准确确定 chart 的过滤占用空间，我们需要找到在渲染期间会进行访问的所有纹素。四个相邻纹素的中心点相连接会构成一个正方形，如果一个 chart 与这个正方形相交，那么在双线性过滤期间，将会使用到这四个纹素。上图中的纹素网格使用实线进行标记，纹素中心使用蓝点进行标记，chart 使用粗实线进行光栅化（左）。首先，我们将 chart 保守光栅化到一个网格中，再将其偏移半个纹素大小，偏移后的网格使用虚线进行标记（中）。任何触碰到标记单元格的纹素，都会被认为是该 chart 所占用的（右）。

避免颜色溢出是光照贴图很少使用 mipmap 的另一个原因。chart 的过滤占用空间需要在所有 mipmap 层级上都保持独立，这将会导致 shell 之间的间距过大。

将 chart 打包到纹理中的最优方法是一个 NP-完全问题，这意味着没有任何已知的算法能够产生一个具有多项式级别复杂度的解。在实时应用程序中，单个纹理可能就会包含数十万个 chart，所有现实世界的解决方案，都使用了微调的启发式方法和精心优化的代码来快速进行打包[183, 233, 1036]。如果这些光照贴图稍后会进行分块压缩（章节 6.2.6），那么为了提高压缩质量，还可以向打包器添加一些额外的约束，从而确保单个块中只包含类似的值。

光照贴图的一个常见问题是接缝（seam，如图 11.29 所示）。因为模型网格被分割成了不同的 chart，并且每个 chart 都是独立进行参数化的，所以不可能确保沿分割边缘两侧的光照效果是完全相同的，这种情况会表现为视觉上的不连续性。如果模型网格是手动分割（参数化）的，可以通过将它们的接缝设置在不可见的区域，从而来避免这个问题。然而，这样做是一个费时费力的过程，并且无法应用在自动生成参数化表示的过程中。Iwanicki [806]对最终生成的光照贴图进行后处理，对沿着分割边缘的纹素进行修改，从而最小化两侧插值结果的差异。Liu 和 Ferguson 等人[1058]通过等式约束（equality constraint）来让插值结果与边缘强制匹配，并求解出最能保持两侧平滑的纹素值。另一种方法则是在创建参数化和打包 chart 的时候考虑这个约

束。Ray 等人[1467]展示了如何使用保持网格的参数化（grid-preserving parameterization）来创建不受接缝瑕疵影响的光照贴图。

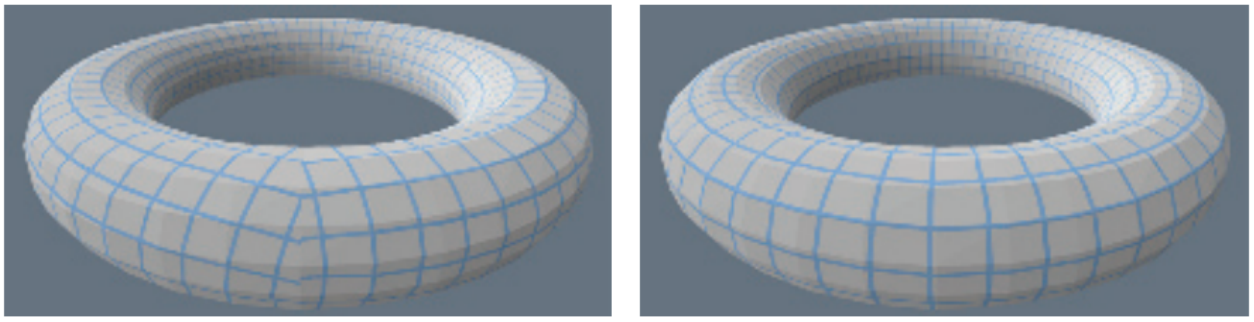


图 11.29：为一个圆环体创建一个唯一的参数化表示，需要将其切割并展开。左边的圆环体使用了一个简单的映射方法，它在创建的时候并没有考虑在纹理空间中的接缝位置。上图中的一个蓝色网格就代表了一个纹素，请注意左侧纹素网格的不连续性。使用一些更加高级的算法，我们可以创建一个唯一的参数化表示，同时确保纹素网格在三维网格上保持连续，如右图所示。这种展开方法对于光照贴图而言是完美的，因为最终生成的光照效果不会显示出任何的不连续性。

预计算的光照信息也可以存储在网格的顶点上。这样做的缺点是光照质量取决于网格细分的精细程度。因为这个决定通常是在模型创作的早期阶段就做出的，因此很难确保网格上有足够的顶点，使得在所有预期的光照情况下看起来都表现很好。此外，这个网格细分的操作成本可能是很高的。如果网格被细分得很精细，那么光照信号将会被过采样。如果使用定向的光照存储方法，则需要通过 GPU 在顶点之间对整个光照表示进行插值，并将其传递到像素着色器阶段，从而执行光照计算。在顶点和像素着色器之间传递这么多参数的情况是相当罕见的，并且会产生现代 GPU 未经优化的工作负载，这会导致效率和性能的低下。由于这些原因，因此很少会在顶点上存储预计算的光照信息。

虽然我们需要表面上的入射 radiance 信息（第 14 章会讨论的体渲染除外），但是我们可以通过体积的方式对其进行预计算和存储。这样做可以在空间中的任意位置上查询光照效果，并为预计算阶段不存在于场景中的物体提供照明。但是请注意，这些物体并不会正确地反射光线或者遮挡光线。

Greger 等人[594]提出了 irradiance 体积的概念，它代表了对 irradiance 环境贴图进行稀疏空间采样的五维（三个空间和两个方向）irradiance 函数。即空间中存在一个三维网格，每个网格点上都是一个 irradiance 环境贴图。动态物体会从最近的贴图中插值出 irradiance 值。Greger 等人使用了一个两级的自适应网格来进行空间采样，但是也可以使用其他一些体积数据结构，例如八叉树[1304, 1305]等。

在原始 irradiance 体积中，Greger 等人将每个样本点的 irradiance 存储在一个小纹理中，但这种表示方法无法在 GPU 上进行高效过滤。如今，体积光照数据通常会存储在三维纹理中，因此对体积进行采样也可以使用 GPU 的硬件加速过滤。样本点处的 irradiance 函数包含以下常见表示方法：

- 二阶和三阶的球谐函数（SH），其中二阶球谐函数要更为常见，因为其单个颜色通道需要使用四个系数，可以很方便地打包成常见纹理格式的四个通道（RGBA）。
- 球面高斯函数。
- 环境立方体或者环境骰子。

AHD 编码方法，虽然它在技术上能够表示球面上的 irradiance 信息，但同时也会产生视觉可见的、分散观众注意力的瑕疵。如果使用 SH 的话，还可以使用球谐梯度（harmonic gradient）来进一步提高质量[54]。上述这些表示方法在许多游戏中都进行了成功的应用[766, 808, 1193, 1268, 1643]。

Evans [444]描述了一个应用在《小小大星球》中计算 irradiance 体积的技巧，它没有存储完整的 irradiance 贴图，而是在每个点上存储平均 irradiance。根据 irradiance 场的梯度信息，即 irradiance 变化最迅速的方向，来计算近似的方向因子。这个梯度并不是显式计算出来的，而是通过在 irradiance 场中取两个样本，其中一个样本位于表面点 $\mathbf{p}$ 处，另一个样本位于沿方向 $\mathbf{n}$ 上稍微偏移的点上，并让一个样本减去另一个样本，从而计算梯度与表面法线 $\mathbf{n}$ 之间的点积。这种近似表示方法的动机是，《小小大星球》中的 irradiance 体积是动态计算的。

irradiance 体积也可用于为静态物体表面提供照明效果。这样做的好处是不需要为光照贴图提供单独的参数化，因此该技术也不会产生接缝瑕疵。静态物体和动态物体都可以使用相同的光照表示方法，这样两种不同类型的几何物体之间可以得到一致的光照效果。在延迟渲染（章节 20.1）中使用这种体积表示方法是很方便的，因为所有光照计算都可以在一个 pass 中完成。这种方法的主要缺点是内存开销过大，光照贴图所使用的内存量与分辨率的平方成正比；而对于规则的体积结构而言，它所使用的内存量则与分辨率的立方成正比。由于这个原因，网格体积表示方法使用了相当低的分辨率。自适应的、分层的光照体积具有更好的特性，但是它们仍然要比光照贴图存储更多的数据。与规则间距的网格体积相比，它们的执行速度要更慢，因为额外的间接表示会在着色器代码中创建加载依赖，这可能会导致停滞阻塞以及执行速度的变慢。

在体积结构中存储表面照明有些棘手。因为具有截然不同光照特征的多个表面，有时可能会占据相同的体素，我们不确定应当存储哪些数据。当从这样的体素中进行采样

时，所获得的光照效果通常是不正确的。这种情况经常会发生在明暗交界处，例如明亮室外和黑暗室内之间的墙壁附近，最终会导致室外出现的黑暗面片或者室内出现的明亮面片。解决方法也很直接，让体素的尺寸足够小即可，使得一个体素永远不会跨越这样的边界，但是这样做通常是不切实际的，因为需要的数据量实在是太大了。处理这个问题最常见的方法是：将采样位置沿着法线进行一些移动，或者是插值过程中调整所使用的三线性混合权重。这些做法通常也是不完美的，可能还需要对几何形状进行手动调整来掩盖问题。Hooker [766]在 irradiance 体积中添加了额外的裁剪平面，从而将它们的影响限制在凸多面体的内部。Kontkanen 和 Laine [766]讨论了减少颜色溢出的各种策略。

图 11.30：Unity 引擎使用了一个四面体网格，来从一组探针中插值出光照信息。

存储光照信息的体积结构不一定要是规则均匀的。一种流行的做法是将光照数据存储在不规则的点云中，然后再将这些点连接起来构成 Delaunay 四面体（如图 11.30 所示），Cupisz [316]将这种方法进行了推广。为了检索光照信息，我们首先需要找到采样位置所在的四面体，这是一个迭代过程，其开销可能会有点高。我们对网格进行遍历，并在相邻的单元之间进行移动。会使用查找点（采样位置）相对于当前四面体顶点的重心坐标（barycentric coordinate），来选择下一步要进行访问的邻居四面体（如图 11.31 所示）。在一个常见的场景中，可能会包含数千个存储光照信息的点云位置，因此这个检索过程可能会很耗时。为了对它进行加速，我们可以在前一帧中记录一个用于查找的四面体（如果可能的话），或者使用一个简单的体积数据结构，来为场景中的任意采样点提供一个良好的“起始检索四面体”。

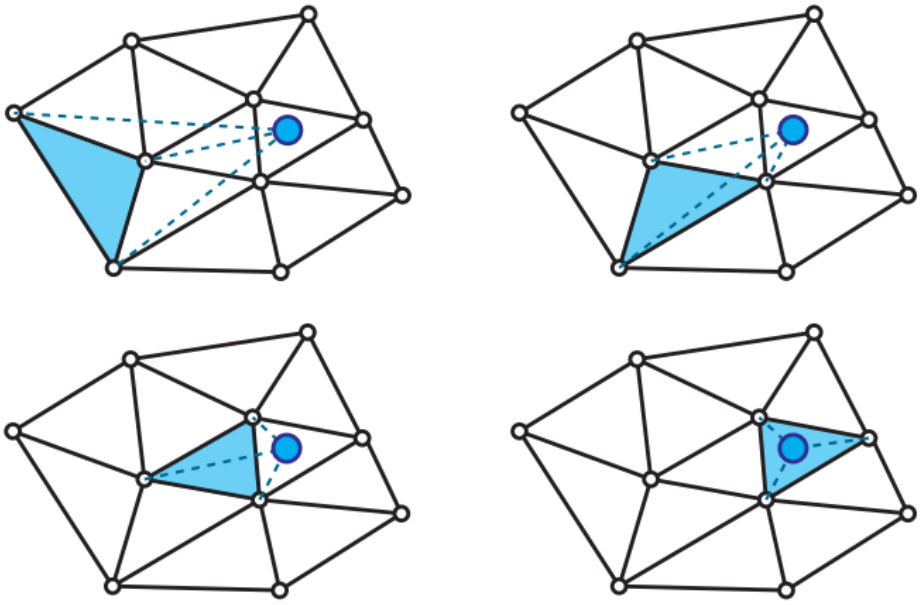


图 11.31：在二维四面体网格中的查找过程。步骤顺序从左到右，从上到下。对于给定的起始单元格（使用蓝色标记），我们会计算查找点（蓝点）相对于单元格顶点的重心坐标。在下一步中，我们会选择重心坐标中负数绝对值最大的那个顶点，并向其对边的邻居四面体进行移动。

一旦检索到了合适的四面体，就会使用重心坐标来对存储在四面体顶点的光照信息进行插值。这个插值操作并不会被 GPU 加速，它只需要 4 个值进行插值，而不是网格上三线性插值所需要的 8 个值。

这些预计算和存储光照信息的位置可以手动放置[134, 316]，也可以自动放置[809, 1812]。它们通常被称为光照探针（lighting probe 或者 light probe），因为它们对光照信号进行了探测（采样）。这个术语需要和章节 10.4.2 中的“光照探针”区分开来，后者是记录在环境贴图中的远距离光照。

从四面体网格中采样获得的光照质量，高度依赖于网格的结构，而不仅仅是探针的总体密度。如果光照探针分布不均匀的话，那么生成的网格中可能会包含一些细长的四面体，从而产生视觉上的瑕疵。如果这些探针是手动放置的，那么这些问题可以很容易地进行纠正，但是这毕竟是一个手动过程，费时费力。这个四面体网格的结构与场景的几何结构无关，因此如果处理不当的话，光照效果在插值的时候会跨越墙壁的两侧，从而产生漏光瑕疵，就像是上文中的 irradiance 体积一样。在手动放置探针的情况下，开发者可以通过插入额外的探针来避免发生这种情况。在自动放置探针的情况下，可以向探针或者四面体中添加某种形式的可见性信息，从而将单个四面体的影响范围限制在相关区域内[809, 1184, 1812]。

对于静态和动态的几何物体，通常会使用不同的光照存储方法。例如：静态物体可以使用光照贴图，而动态物体则可以从体积结构中获得光照信息。虽然这样做很流行，

但是这种方案可能会导致不同类型的几何物体之间产生不一致的外观表现。其中一些差异可以通过正则化（regularization）来消除，即在这些表示方法中对光照信息进行平均。

当烘焙光照信息的时候，需要注意的是，只需要在它们真正有效且合法的地方来计算光照信息即可。生成的网格通常是不完美的，一些顶点可能会被放置在几何体内部，或者网格的部分区域可能会产生自相交现象。如果我们在这些有缺陷的位置上计算入射 radiance，那么结果将是不正确的。它们可能会导致我们不希望出现的暗化，或者是无阴影光照的颜色溢出等现象。Kontkanen 和 Laine[926]，Iwanicki 和 Sloan [809]讨论了不同的启发式方法，这些方法可以用于丢弃无效样本。

环境光遮蔽和定向遮蔽信号与漫反射光照共享许多空间特性，如章节 11.3.4 所述，上述所有的方法都可用于存储它们。

11.5.5 动态漫反射全局光照

尽管预计算光照可以产生令人印象深刻的效果，但是它的主要优点同样也是它的主要缺点，即这种方法需要进行预计算。这个离线预计算的过程可能会很长，在常见的游戏关卡中，可能需要花费数个小时来进行光照烘焙，这种情况并不少见。由于光照计算需要花费很长时间，因此艺术家们被迫在多个层次上同时工作，从而避免在等待烘焙完成的时候无所事事。反过来，这通常又会导致渲染资源的过度负载，从而导致烘焙时间变得更长。这种烘焙–调整–再烘焙的循环，会严重影响工作效率并导致挫败感（frustration）。同时在某些情况下，可能无法使用预计算光照，因为场景中的几何物体在运行过程中会不断发生改变，或者在某种程度上，场景中的几何物体是由用户控制创建的。

为了模拟动态环境中的全局光照效果，已经有好几种方法被开发了出来。它们要么不需要任何预处理过程，要么算法的准备阶段足够得快，可以每帧执行。

在完全动态环境中模拟全局光照的最早方法之一是基于“即时辐射度（Instant Radiosity）”[879]。尽管这个方法名为辐射度算法，但是它与辐射度算法几乎没有共同之处。在这种方法中，会从光源向外投射光线，对于这些光线照射到的每个位置，都会放置一个新的光源，用于代表来自该表面元素的间接照明，这些光源被称为虚拟点光源（virtual point light，VPL）。基于这个思路，Tabellion 和 Lamorlette [1734]开发了一种在《怪物史莱克 2》制作过程中所使用的方法，该方法会对场景表面执行一次直接光照 pass，并将结果存储在纹理中。然后，在渲染过程中，该方法会对光线进行追踪，并使用缓存下来的光照数据来创建一次弹射的间接光照效果。Tabellion 和 Lamorlette 的研究表明，在很多情况下，一次弹射就足以产生令人信服

的结果。虽然这是一种离线方法，但是它启发了 Dachsbacher 和 Stamminger [321]，他们提出一种名为反射阴影贴图（reflective shadow maps，RSM）的方法。

与常规的阴影贴图（章节 7.4）类似，反射阴影贴图是从光源的视角来进行渲染的。除了深度信息之外，它们还会存储有关可见表面的其他信息，例如反照率 albedo、法线、直接光照（通量 flux）。在进行最后着色的时候，RSM 的纹素会被视为虚拟点光源，从而提供单次弹射的间接照明效果。由于一个典型的 RSM 中可能会包含数十万个像素，将这像素都作为点光源明显是不现实的，因此需要使用重要性驱动（importance-driven）的启发式方法，来选择其中的一个子集。Dachsbacher 和 Stamminger [322]后来展示了如何通过逆转这个过程来对该方法进行优化。该方法会基于整个 RSM 来创建一些虚拟光源，并将其放置（splatted）在屏幕空间中（章节 13.9），而不是为每个着色点都从 RSM 中选择相关的纹素。

该方法的主要缺点是，它无法为间接光照提供遮挡效果。虽然这样做是一个很显著的近似，但是该方法生成的结果看起来还是合理的，并且对于许多应用程序而言也是可以接受的。

为了获得高质量的结果，并在光线运动过程中保持时域稳定性，因此需要创建大量的间接光源。如果创建的间接光源数量太少，当重新生成 RSM 的时候，它们的位置往往会迅速发生改变，从而导致闪烁瑕疵的出现。另一方面，从性能的角度来看，场景中存在太多的间接光源是十分具有挑战性的。Xu [1938]描述了游戏《神秘海域 4》是如何应用这种方法的。为了保证性能要求，他在每个像素上只使用了少量的间接光源（16 个），但是会在几帧之间循环使用不同的光源集合，并且会对结果进行时域过滤，如图 11.32 所示。

图 11.32：游戏《神秘海域 4》使用了反射阴影贴图来提供来自玩家手电筒的间接光照。左边的图像展示了没有间接光照的场景，右边的图片中则启用了间接光照。右侧小插图则展示了未启用时域过滤（上），以及启用了时域过滤（下）的特写画面。它用于增加每个图像像素所使用 VPL 的有效数量。

针对缺乏间接遮挡的问题，人们提出了不同的解决方法。Laine 等人[962]使用了双抛物面阴影贴图来作为间接光源，但是会逐步将它们添加到场景中，因此在一帧中只有少量阴影贴图会被渲染。Ritschel 等人[1498]使用简化的、基于点的场景表示，来绘制大量不完美的阴影贴图（imperfect shadow maps）。这样的贴图很小，并且在直接使用的时候还会包含许多缺陷，但是在经过简单的过滤之后，能够提供足够的保真度，从而为间接光照提供适当的遮挡效果。

有些游戏则使用了与这些解决方案相关的方法。其中《Dust 514》渲染了一个自上而下的世界视图，并且在需要的时候可以拥有多达 4 个独立的图层[1110]。这些生成的纹理会用于间接光照的收集，这很像 Tabellion 和 Lamorlette 的方法。在风筝 demo 中，虚幻引擎使用了类似的方法来提供地形的间接光照效果[60]。

11.5.6 光照传播体积

辐射传输理论（radiative transfer theory）是一种模拟电磁辐射如何在介质中传播的一般方法，它包括了散射（scattering）、发射（emission）和吸收（absorption）。尽管实时图形学力求显示所有的这些效果，但是除了最简单的情况之外，用于模拟这些效果的方法都具有很高的成本，无法直接应用于渲染中。然而，该领域中所使用的一些技术，在实时图形应用中被证明是很有用的。

由 Kaplanyan [854]提出了光照传播体积（light propagation volumes, LPV），其灵感来源于辐射传输中的离散坐标法（discrete ordinate methods）。在他的方法中，场景被离散成一个规则的三维网格，每个单元格内都会维护一个穿过它的定向 radiance 分布，他使用二阶球谐函数来处理这些信息。在第一步中，光照会被注入到包含直接光照表面的单元格中。可以使用反射阴影贴图来找到这些单元格，也可以使用任何的其他方法。注入这些单元格的光照信息，是该表面反射出的 radiance，它在表面法线附近构成了一个分布，指向远离表面的方向，并且会从材质的颜色中获得自身的颜色。接下来会对光照进行传播，每个单元格都会对其邻居单元格的 radiance 场进行分析，并据此修改自身的 radiance 分布，从而考虑来自各个方向的 radiance。在一个步骤中，radiance 只在一个单元格的距离上进行传播，因此为了让 radiance 进行充分传播，需要进行多次迭代，如图 11.33 所示。

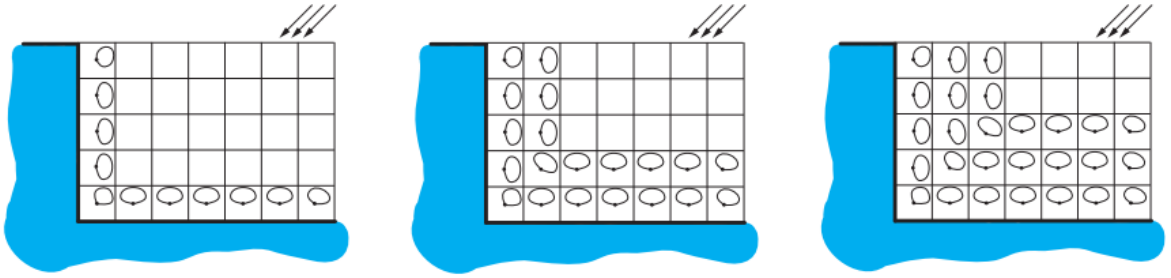


图 11.33：光照分布通过一个体积网格进行传播的三个步骤。左侧图像显示了由一个方向光照亮几何体所产生的反射光照分布。请注意，只有与几何物体直接相邻的单元格才具有非零的光照分布。在随后的步骤中，来自相邻单元格的光照会被收集并通过网格进行传播。

这种方法的重要优点在于，它会为每个单元格生成完整的 radiance 场，这意味着我们可以使用任意的 BRDF 来进行着色，尽管在使用二阶球谐函数的时候，光泽 BRDF 的反射质量会很低。Kaplanyan 展示了漫反射表面和镜面的例子。

为了允许光照在更大的距离上进行传播，增加体积所覆盖的区域面积，同时保持合理的内存开销，Kaplanyan 和 Dachsbacher [855]开发了该方法的一种级联变体。他们不再使用与单元格大小相同的体积，而是使用一组逐渐变大的单元格，这些单元格彼此之间能够嵌套。光照会被注入到所有的层级中并独立地进行传播。而在查找过程中，会为给定位置选择最详细且可用的层级单元格来计算光照。

在最初的实现中，他们没有考虑间接照明的任何遮挡。修改后的方法使用了来自反射阴影贴图的深度信息，以及来自相机位置的深度缓冲，从而向这些体积块中添加了有关光线遮挡物的信息。这些信息是不完整的，但是场景也可以在预处理期间进行体素化，从而使用更加精确的表示方法。

该方法与其他体积方法存在相同的问题，其中最大的问题是颜色溢出。不幸的是，在 LPV 方法中单纯地增加网格分辨率来解决这个问题，还会导致出现其他问题。当使用较小尺寸的单元格时，就需要更多的迭代步骤，来在相同的世界空间距离上进行光线传播，这会使得该方法的成本明显上升。在网格分辨率和性能之间找到一个平衡并非易事。同时该方法还存在锯齿问题，网格的有限分辨率，加上 radiance 的粗糙定向表示（二阶球谐函数），会导致光照信号在相邻单元格之间移动时发生退化。例如对角条纹等空间瑕疵，可能会在多次迭代后出现。其中一些问题可以通过在执行传播 pass 之后，再执行空域过滤来进行消除。

11.5.7 基于体素的方法

Crassin [304]提出了体素锥形追踪全局光照（voxel cone tracing global illumination，VXGI），它也是基于一种体素化的场景表示。几何物体本身使用稀疏体素八叉树（sparse voxel octree）的形式进行存储，我们将在章节 13.10 中进行介绍。这种结构提供了一种类似于 mipmap 的场景表示，因此可以对体积空间进行快速的遮挡测试等操作。每个体素块还包含了它们所代表的几何物体所反射出的光线量等信息，它以一种定向形式进行存储，因为 radiance 会在六个主要方向上发生反射。首先会使用反射阴影贴图，将直接照明注入到八叉树的最低层节点中，然后再根据层次结构向上进行传播。

这个八叉树结构用于估计入射 radiance。在理想情况下，我们将会追踪一条射线，从而计算来自特定方向上的 radiance 估计。然而，这样做需要追踪许多射线才能获得理想结果，因此我们会将整个光束近似于一个圆锥，这个圆锥位于它们的平均方向上，我们对这个圆锥进行追踪，最后只会返回一个值。想要精确测试圆锥与八叉树的交点并不是一件容易的事情，因此这个操作会被进一步近似，我们会沿着圆锥的中心轴，对八叉树结构进行一系列的查找。每次查找都会对八叉树的某个层次进行读取，该层次上的节点大小，应当与给定点处的锥形截面相对应。查找提供了在圆锥原点方向上反射的滤波 radiance，以及几何物体占据查找空间的百分比，这个百分比信息会用于减弱来自后续点的光照强度，这有点类似于 alpha 混合。整个锥体的遮挡信息也会被追踪，在每个步骤中，它会被减少到几何物体占当前样本的百分比。在累积 radiance 的时候，首先会将其乘以合并后的遮挡因子（如图 11.34 所示）。虽然这种策略无法检测到由多个部分遮挡组合而成的完整遮挡，但是其结果仍然是可信的。

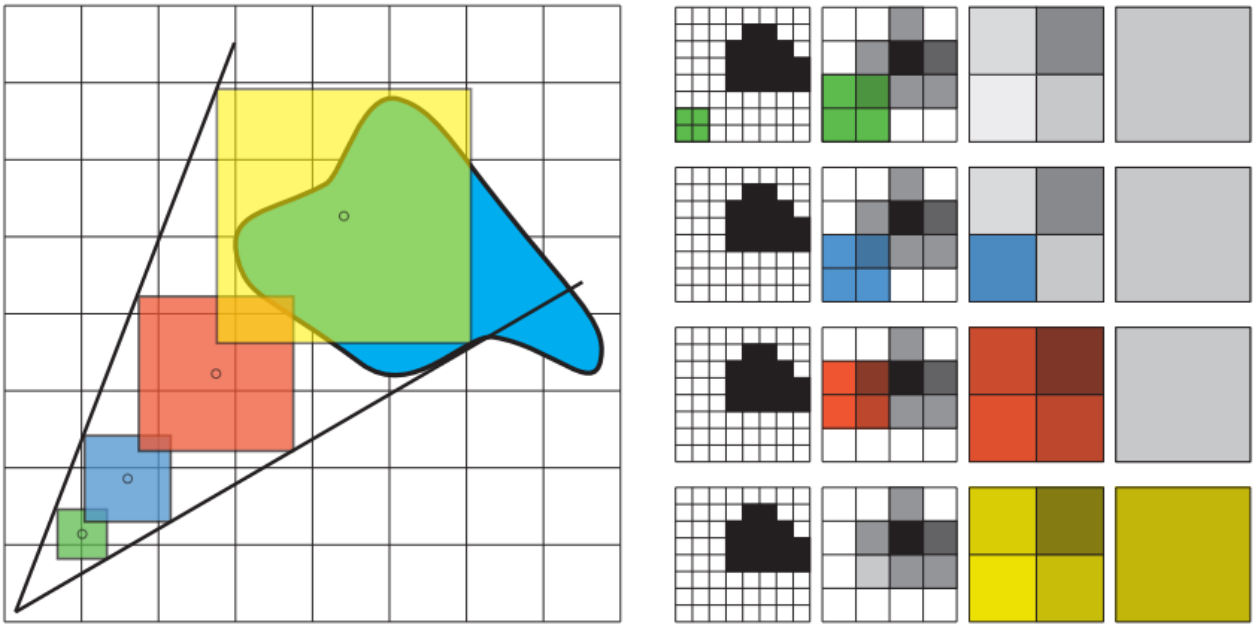


图 11.34：体素锥形追踪使用一系列体素树中的过滤查找，来对一个精确的锥形追踪进行近似。左图显示的是三维轨迹的二维模拟。右图展示了体素化几何的分层表示，从左到右每一列

所展示的体素树，其层次越来越粗糙。在右图每一行中，展示了用于为给定样本提供覆盖率的层次结构节点。选择合适的级别进行使用，从而使得较粗级别节点的大小大于当前查找的大小，较细级别节点的大小小于当前查找的大小。会使用一个类似于三线性滤波的过程，来在这两个选定的级别之间进行插值。

为了计算漫反射光照，我们需要跟踪若干个圆锥，具体生成和追踪的圆锥数量，取决于性能和精度之间的权衡。追踪更多的圆锥可以提供更高质量的结果，其代价是需要花费更多的时间。我们假设余弦项在整个圆锥上都是恒定的，因此这一项可以从反射方程的积分中提取出来。这样做可以使得漫反射光照的计算变得很简单，只需要计算锥形追踪的返回值，并计算其加权和即可。

正如 Mittring [1229]所描述的，这个方法的原型版本是在虚幻引擎中实现的。他给出了一些开发人员需要进行的优化，从而可以使其作为完整渲染管线的一部分进行使用。这些改进包括以较低的分辨率来执行追踪，并在空间散布圆锥。这样做的目的是为了让每个像素只跟踪一个圆锥。并在屏幕空间中对结果进行过滤，从而获得漫反射响应的完整 radiance。

使用稀疏八叉树存储光照信息，一个主要问题就是查找成本较高。找到包含给定位置的叶子节点，需要进行一系列的内存查找，中间还穿插着一些简单的逻辑来确定要遍历哪一个子树。一次典型的内存读取可能需要耗费几百个时钟周期。GPU 试图通过并行执行多组着色线程（warp 或者 wavefront）来隐藏这种延迟（第 3 章）。即在任何给定时间内，只有一组着色线程会执行 ALU 操作，当它需要等待内存读取的时候，另一组着色线程会取而代之。能够同时激活的 warp 数量由不同的因素所决定，但所有的这些因素，都与单个组所使用的资源数量有关（章节 23.3）。在遍历分层数据结构的时候，大部分时间都花在内存读取上，会等待从内存中获取下一个节点的数据。然而，在等待期间执行的其他 warp 中，很可能也会进行内存读取。与内存访问的次数相比，ALU 其实工作得很少，并且由于实际运行的 warp 总数是有限的，因此经常会出现所有分组都在等待内存返回数据，都没有实际工作执行的情况。

大量的 warp 停滞会导致性能表现不佳，人们已经开发了一些方法来缓解这些低效问题。McLaren [1190]使用一组级联的三维纹理来代替八叉树结构，这种方法很像级联的光照传播体积[855]（章节 11.5.6）。它们具有相同的尺寸，但是所覆盖的区域面积越来越大。通过这种方式，只需进行一次常规的纹理查找即可完成数据的读取，而不需要进行额外的依赖读取。存储在纹理中的数据与存储在稀疏体素八叉树中的数据相同，它们都包含六个方向上的反照率、占用率和反弹光照信息。因为级联的位置会随着相机的移动而发生变化，因此物体可能会不断地进出高分辨率区域。由于内存的限制，我们不可能在内存中一直维护这些体素化内容，因此它们会在需要的时候才进行

体素化。McLaren 还介绍了一些优化方法，使得这种技术能够用于 30 FPS 的游戏，例如《明日之子（The Tomorrow Children）》，如图 11.25 所示。

图 11.35：游戏《明日之子》使用了体素锥形追踪来渲染间接光照效果。

11.5.8 屏幕空间方法

与屏幕空间环境光遮蔽（章节 11.3.6）一样，可以只使用存储在屏幕位置上的表面信息[1499]，来模拟一些漫反射全局光照效果 。这些方法并不像 SSAO 那样流行，主要是因为屏幕空间中可用的数据量十分有限，因此会导致更加明显的瑕疵。诸如颜色溢出（color bleeding）这样的效果，通常是由于强烈的直射光线照亮具有相对恒定颜色的大面积区域而产生的。像这样的表面通常不可能完全适应视图，即可能只有部分会出现在画面中。这种情况使得反射光线的数量强烈依赖于当前帧，并且会随着相机的移动而发生波动。出于这个原因，屏幕空间中的方法仅适用于在精细尺度上对其他解决方案进行扩展补充，这种精细尺度超出了主要算法所能够达到的分辨率。这类系统在游戏《量子破碎》[1643]中进行了使用，在这个游戏中，使用了 irradiance 体积来模拟大规模全局光照的效果，使用屏幕空间中的解决方案来提供有限距离的弹射光线。

11.5.9 其他方法

Bunnell 用于计算环境光遮蔽的方法 [210]（章节 11.3.5），也可以用于动态计算全局光照效果。通过存储每个圆盘的反射 radiance 信息，来对基于点的场景表示方法

（章节 11.3.5）进行增强。在收集步骤中，可以在每个收集位置上构建一个完整的入射 radiance 函数，而不仅仅是收集遮挡信息。就像环境光遮蔽一样，必须要执行一些后续步骤，来消除那些来自于被遮挡圆盘的光照。

11.6 镜面全局光照

上一小节中所介绍的方法，主要是为了模拟漫反射全局光照效果，而在本小节中，我们将会介绍各种用于渲染视图依赖（view-dependent）效果的方法。对于光泽材质而言，其镜面波瓣要比漫反射光照中所使用的余弦波瓣紧密得多，其扩散范围小得多。如果我们想要渲染一种极有光泽的材质，这种材质具有很薄的镜面波瓣，我们需要一种能够传递这种高频细节的 radiance 表示方法。反过来，这些条件也意味着，反射方程只需要计算从有限立体角入射的光线即可，而不是像 Lambertian BRDF 那样，需要反射来自整个半球的入射光线，这与漫反射材质的要求完全不同。这些特性解释了想要实时渲染这样的效果，为什么需要进行完全不同的权衡考虑的原因。

存储入射 radiance 的方法可以用于提供粗略的视图依赖效果。当使用 AHD 编码或者 HL2 基底时，我们是可以计算镜面反射的，就好像光照来自于编码方向（在使用 HL2 基底时，是三个方向）的方向光一样。这种方法的确可以通过间接照明提供一些高光效果，但是它们相当不准确。在使用 AHD 编码时，这种方法尤其成问题，因为方向分量可能会在很小的距离内发生剧烈变化，这种方差会导致高光以不自然的方式发生变形。可以通过在空间方向上进行滤波来减少这种瑕疵[806]。在使用 HL2 基底时，如果相邻三角形之间的切线空间变化很快，同样也会出现类似的问题。

可以通过使用更高的精度来表示入射光线，从而减少瑕疵的出现。Neubelt 和 Pettineo 在游戏《教团：1886》[1268]中使用球形高斯波瓣来表示入射 radiance。为了渲染高光效果，他们使用了 Xu 等人[1940]提出的一种方法，该方法包含了一种典型微表面 BRDF 高光响应（章节 9.8）的有效近似。如果使用一组球面高斯函数表示光照，并且假设菲涅尔项和 masking-shadowing 函数在其范围内为常数，则反射方程可以被近似为：

$$L_o(\mathbf{v}) \approx \sum_k \left(M\left(\mathbf{l}_k, \mathbf{v}\right) \left(\mathbf{n} \cdot \mathbf{l}_k\right)^+ \int_{\mathbf{l} \in \Omega} D(\mathbf{l}, \mathbf{v}) L_k(\mathbf{l}) d\mathbf{l} \right) \quad (11.37)$$

其中 L_k 为第 k 个球面高斯所表示的入射 radiance，M 是结合了菲涅尔项和 masking-shadowing 函数的组合因子，D 为 NDF 项。Xu 等人引入了一种各向异性的球面高斯（anisotropic spherical Gaussian，ASG），他们使用 ASG 来对 NDF

进行建模。他们还为计算 SG 和 ASG 乘积的积分提供了一种有效的近似，如方程 11.37 所示。

Neubelt 和 Pettineo 使用了 9–12 个高斯波瓣来表示光照，这使得他们只能模拟中等光泽的材质。他们能够使用这种方法来表现大部分的游戏光照效果，因为游戏《教团：1886》发生在 19 世纪的伦敦，而那时具有高度抛光的材质，玻璃和反射表面是十分罕见的。

11.6.1 局部环境贴图

到目前为止我们所讨论的方法，还不足以渲染令人信服的抛光材质。对于这些技术而言，它们所能描述的 radiance 场太过粗糙，无法精确编码入射 radiance 的细节，这使得反射看起来很暗淡。如果在同一材质上进行使用的话，所产生的结果也与分析光源的镜面高光不一致。一种解决方案是使用更多的球面高斯函数或者更高阶的 SH 来获得我们所需要的细节。这样做是可行的，但是我们很快就会面临一个性能问题：SH 和 SG 都有全局支持（global support）特点。即每个基函数在整个球面上都是非零的，这意味着我们需要将所有的基函数都计算一遍，才能获得给定方向上的光照信息。这样做的计算成本会变得很高，因为想要渲染尖锐的反射效果，我们可能需要数千个基函数。而且也不可能在漫反射光照的分辨率下，存储那么多的数据。

在实时环境中为全局光照提供高光分量，其中最流行的解决方案是局部环境贴图（localized environment map），它可以解决我们之前遇到的两个问题。首先，入射 radiance 会被表示为一个环境贴图，因此只需要几个值就可以获得所需的 radiance。其次，这些局部环境贴图稀疏地分布在整个场景中，因此如果我们想要增加入射 radiance 的空间精度，只要增加这些局部环境贴图的角分辨率（angular resolution）即可。这种在场景中特定点进行渲染的环境贴图，通常会被称为反射探针（reflection probe）。图 11.36 展示了这样一个例子。

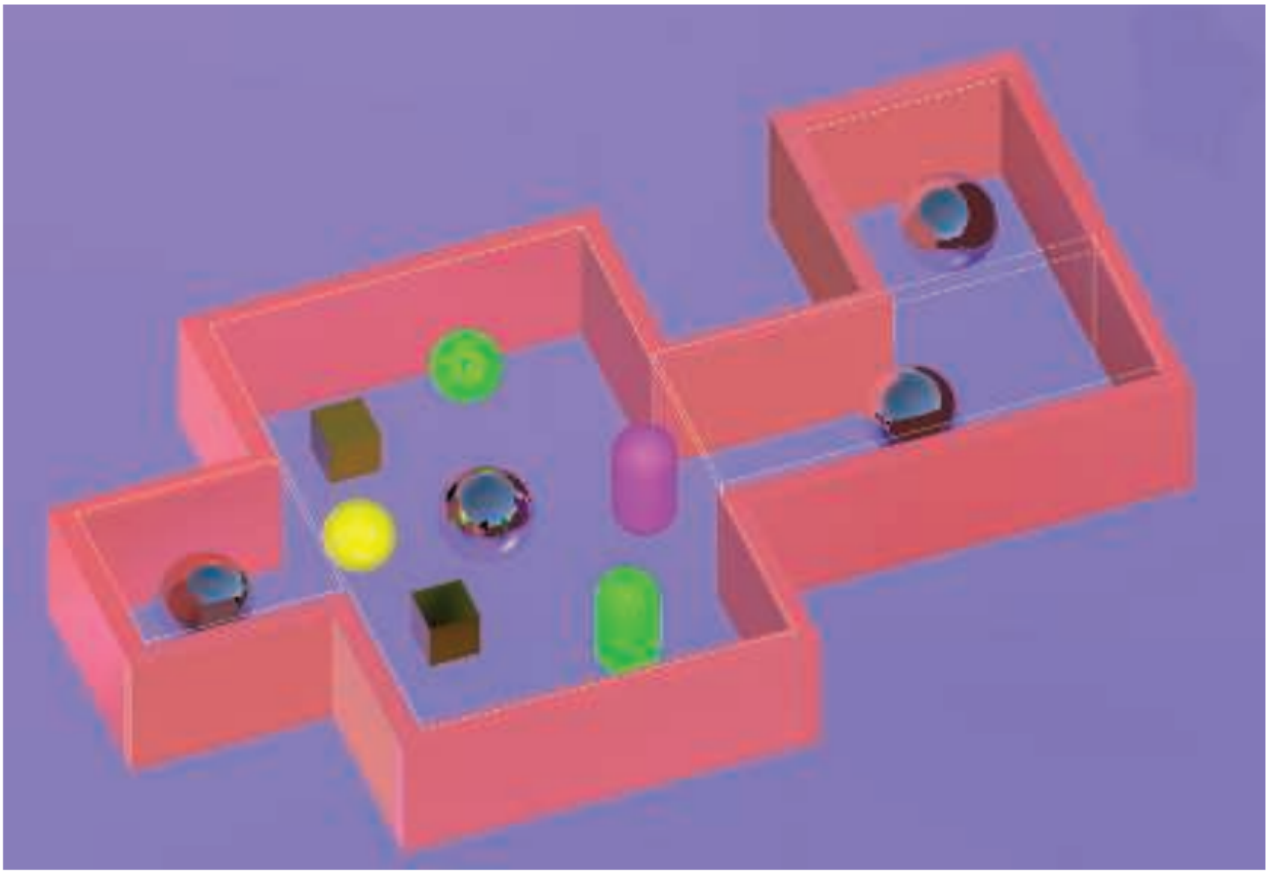


图 11.36：一个简单的场景与局部反射探针。图中的反射球代表了探针的位置，淡淡的黄色线条代表了长方体形状的反射代理。请注意，代理的形状与场景的整体形状相近似。

环境贴图非常适合用于渲染完美的反射效果，即镜面的间接照明。已经有很多方法可以利用纹理来实现各种各样的高光效果了（章节 10.5）。所有这些方法都可以与局部环境贴图一起使用，以渲染间接光照的镜面响应效果。

最早将环境贴图与空间中特定点相绑定的游戏之一是《半条命 2》[1193, 1222]，在他们的系统中，会由艺术家首先在整个场景中放置采样位置。在预处理阶段中，会在每个位置上渲染一个立方体贴图。在进行高光计算的时候，物体会使用最近位置上的结果来作为入射 radiance 的表示。相邻的物体可能会使用不同的环境贴图，这将会导致视觉效果的不匹配，但是艺术家可以手动调整立方体贴图所覆盖的范围。

如果一个物体很小，并且环境贴图就是从其中心进行渲染的（在隐藏该物体之后，它就不会出现在纹理中），那么所生成的结果是相当精确的。不幸的是，这种情况十分少见；在大多数情况下，一个反射探针会同时用于多个物体，有时候还会具有明显的空间范围。高光表面的位置距离环境贴图的中心越远，其结果与现实的差异就越大。

Brennan [194]和 Bjorke [155]提出了一种解决这个问题的方法。他们并没有将入射光照看作是来自一个无限远的包围球体，而是假设这些入射光照来自一个有限大小的球体，该球体的半径是用户进行定义的。在检索入射 radiance 的时候，输入的方向

不会直接用于索引环境贴图，而是将其视为来自评估表面发射出的射线，并与该球体相交。然后会计算一个新的方向，即从环境贴图中心位置指向球面交点位置的方向，这个方向向量会用作查找方向，如图 11.37 所示。这个过程具有在空间中“固定”环境贴图的效果，这个做法通常被称为视差校正（parallax correction）。同样的方法也可以用于其他的基本形状类型，例如 box [958]。用于与光线相交的形状通常会被称为反射代理（reflection proxy）。所使用的代理物体应当能够表示渲染到环境贴图中的几何物体的一般形状和大小。虽然通常而言这是不可能的，但是如果反射代理能够与几何体完全匹配（例如用一个 box 代表一个矩形房间），那么这种方法可以提供完美的局部反射效果。

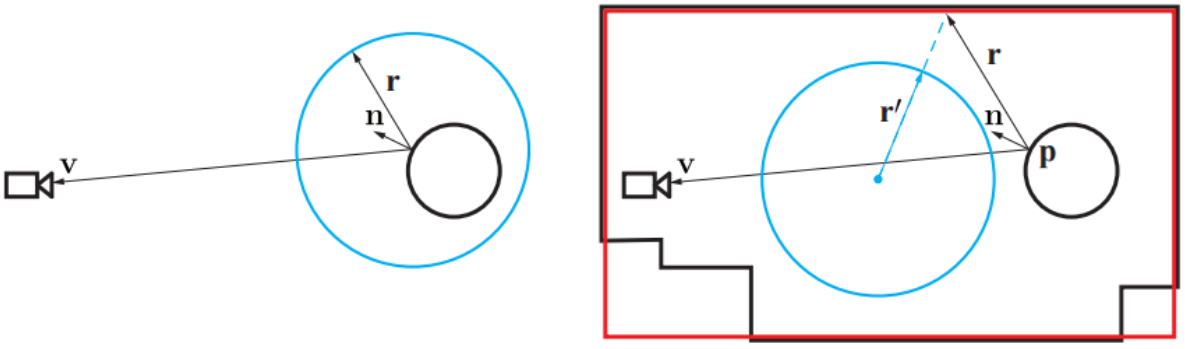


图 11.37：使用反射代理对环境贴图（EM）进行空间局部化的效果。在上图中所展示的两种情况下，我们都希望在黑色圆的表面上渲染环境的反射效果。左边是常规的环境映射，它使用蓝色圆圈进行表示（它也可以是任何表示形式，例如立方体贴图）。左图中的效果是通过使用反射观察方向 $\mathbf{r}$ 访问环境贴图来确定的。仅仅使用这个方向作为参数，蓝色圆圈 EM 会被视为半径无限大且遥远的。对于黑色圆表面上的任何点，EM 好像都以该点为中心。右图中，我们希望 EM 能够把周围的黑色房间表示为本地的，而不是无限远的。蓝色圆圈 EM 是在房间的中心处生成的。要像访问房间一样访问这个 EM，会从位置 $\mathbf{p}$ 处，沿着反射观察方向发射一根反射光线，这个光线会在着色器中与一个简单的代理物体（房间周围的红色框）相交。这个交点与 EM 的中心形成一个新的方向 $\mathbf{r}'$，然后会像常规的环境映射一样，使用这个方向 $\mathbf{r}'$ 来访问 EM。通过求解 $\mathbf{r}'$，这个过程会将 EM 视为具有一个实际的物理形状，即图中的红框。这个红色代理框的假设会在房间的左下角和右下角失效，因为代理形状与实际房间的几何形状并不匹配。

这种技术在游戏中非常流行，它易于实现，运行速度快，可以应用于前向渲染和延迟渲染中。美术人员还可以直接控制其外观与内存开销。如果某些区域需要更加精确的照明效果，他们可以放置更多的反射探针，同时让代理物体更好地适应场景几何形状。如果用于存储环境贴图的内存过多，那么从场景中删除一些探针也是很容易的。当使用光泽材质的时候，着色点与反射代理交点之间的距离，可以用来决定使用哪个

级别的预过滤环境贴图，如图 11.38 所示。这样做可以模拟在我们远离着色点时，BRDF 波瓣不断增长的覆盖区域。

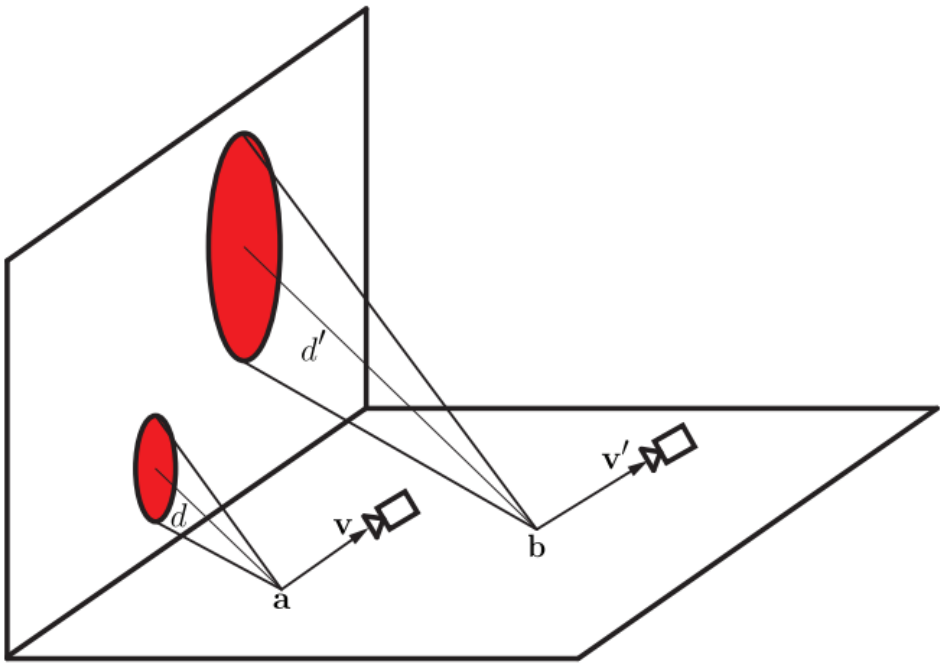


图 11.38：点 $\mathbf{a}$ 和点 $\mathbf{b}$ 处的 BRDF 是相同的，观察向量 $\mathbf{v}$ 和 $\mathbf{v}'$ 相等。由于点 $\mathbf{a}$ 到反射代理的距离 d，要比点 $\mathbf{b}$ 到反射代理的距离 d' 短，因此其 BRDF 波瓣在反射代理一侧的占用比较小（用红色标记）。当对预过滤环境贴图进行采样时，这个距离参数可以与反射点的粗糙度一起使用，来决定所使用的 mipmap 层级。

当多个探针覆盖同一区域时，可以建立如何组合它们的直观规则。例如：探针可以有一个用户设置的优先级参数，具有较高优先级参数的探针，其优先级会高于其他优先级较低的探针，或者可以在它们之间进行平滑地插值融合。

不幸的是，由于这种方法过于简单，因此会导致各种各样的瑕疵。反射代理的几何形状很少能够与底层几何结构完全匹配。这会使得某些区域上的反射效果被不自然地拉伸，这个问题主要会发生在高度反射、抛光的材质上。此外，渲染到环境贴图中的反射物体会根据贴图的位置来计算它们的 BRDF。访问环境贴图的表面位置，不会与这些物体具有完全相同的视图，因此纹理中存储的结果不是完全正确的。

反射代理也会导致漏光问题（有时会很严重）。通常而言，查找过程会从环境贴图的明亮区域返回结果值，因为这个简化的光线投射会错过应当引起遮挡的局部几何形状。这个问题有时候可以通过使用定向遮蔽方法（章节 11.4）来缓解。另一个缓解这个问题的流行策略，是使用预计算漫反射光照，它通常会以更高的分辨率进行存储。环境贴图中的反射值首先会除以渲染位置上的平均漫反射光照。这样做可以有效地从环境贴图中去除平滑、扩散的贡献值，从而只留下较高频率的成分。在进行着色时，

反射值再乘以着色位置上的漫反射光照。这样做可以部分缓解反射探针空间精度不足的问题[384, 999]。

有一些方法可以使用反射探针来捕获更加复杂的几何表示。Szirmay-Kalos 等人[1730]为每个反射探针都存储了一个深度贴图，并在查找时对使用它执行一次光线追踪，这样可以产生更加准确的结果，但是需要花费一些额外的开销。McGuire 等人[1184]提出了一种更加有效的方法，它根据探针的深度缓冲来追踪光线。他们的系统会存储多个探针，如果最初选择的探针没有包含足够的信息来可靠地确定命中位置，则会选择使用备用探针，并继续使用新的深度数据来进行光线追踪。

当使用光泽 BRDF 的时候，环境贴图通常是预过滤的，并且每个 mipmap 层级所存储的入射 radiance 都会与一个逐渐增大的滤波核进行卷积。预过滤步骤会假设这个滤波核是径向对称的（详见章节 10.5）。然而，当使用视差校正的时候，BRDF 波瓣在反射代理形状上所占据的空间，会根据着色点位置而发生变化，这样做会使得预过滤过程变得稍微不正确。Pesce 和 Iwanicki 对这个问题的不同方面进行了分析，并讨论了潜在的解决方案[807, 1395]。

所使用的反射代理形状，也不必是封闭的、凸的。也可以使用简单的平面矩形，也可以使用包含高质量细节的 box 或者球形代理[1228, 1640]。

11.6.2 环境贴图的动态更新

使用局部反射探针需要对每个环境贴图进行渲染和过滤，这项工作通常是离线完成的，但是在某些情况下可能也需要在运行时完成。在开放世界游戏中，一天中的时间会不断发生变化，世界场景中几何物体是动态生成的，因此将这些贴图都进行离线处理会花费太长时间，影响开发效率。在某些极端情况下，如果需要许多变体环境贴图时，我们甚至无法将它们全部都存储在磁盘上。

实际上，有些游戏会在运行过程中实时渲染反射探针，这种类型的系统需要进行仔细调整，以免对性能产生重大影响。除了一些很简单的情况之外，我们不可能在每一帧中都重新渲染所有可见的探针，因为对于现代游戏而言，每帧通常会使用数十个甚至数百个探针。幸运的是，我们也不需要这样做的。我们很少会要求反射探针在任何时候都能够准确地描述它们周围的所有几何形状。大多数情况下，我们确实希望反射探针能够对一天中某个时刻的变化做出正确反应，但是我们可以通过其他的一些方法来对动态几何物体的反射进行近似，例如我们后面要介绍的屏幕空间方法（章节 11.6.5）。这些假设允许我们在加载阶段提前渲染一些探针，而后续的探针则会在它们进入相机视野时逐个进行渲染。

即使我们希望在反射探针中渲染动态几何物体，我们也只能以一个较低的帧率来对其进行更新。我们可以定义渲染反射探针所需要的帧时间，并且在每帧中更新固定数量的反射探针。基于每个探针到相机的距离、距离上次更新的时间，以及其他类似因素的启发式方法可以帮助我们确定反射探针的更新顺序。在时间预算特别紧张的情况下，我们甚至可以将单个环境贴图的渲染拆分到多个帧中进行。例如：我们可以在一帧中只渲染立方体贴图的其中一个面，在 6 帧中渲染一个完整的立方体贴图。

在离线执行卷积操作的时候，通常会使用高质量的滤波，这种滤波涉及对输入纹理的多次采样，这在要求高帧率的游戏中是不可能实现的。Colbert 和 Krivanek [279]开发了一种方法，该方法使用重要性采样，能够以相对较低的样本数量（约为 64）来实现质量相当的过滤。为了消除大部分噪声，他们从具有完整 mipmap 链的立方体贴图中进行采样，并使用启发式方法来确定每个样本应该读取哪个 mipmap 层级。他们的方法是一种对环境贴图进行快速、运行时预过滤的流行选择[960, 1154]。Manson 和 Sloan [1120]在基函数中构造了所需的滤波核，构造一个特定滤波核的精确系数必须要在优化过程中获得，但是对于一个给定的形状，这个过程只会发生一次。卷积分为两个阶段进行：首先，使用一个简单的滤波核来对环境贴图同时进行下采样和过滤。接下来，将得到的 mipmap 链中的样本组合起来，构建最终的环境贴图。

为了限制光照 pass 的带宽开销以及内存开销，可以对最终生成的纹理进行压缩。Narkowicz [1259]描述了一种将高动态范围反射探针压缩为 BC6H 格式（章节 6.2.6）的有效方法，该格式能够存储半精度的浮点值。

想要渲染复杂的场景，即使一次只渲染立方体贴图一个面，这对于 CPU 而言开销仍然可能会过大。一种解决方案是在离线过程中为环境贴图生成 G-buffer，在运行时只需要计算光照和卷积即可[384, 1154]，这大大降低了 CPU 的负载。如果需要的话，我们甚至可以在预生成的 G-buffer 上渲染动态的几何物体。

11.6.3 基于体素的方法

在大多数性能受限的情况下，局部环境贴图是一个很好的解决方案，然而，其质量往往不能令人满意。在实践中，必须使用一些变通方法来掩盖由于探针空间密度不足，或者反射代理对实际几何形状过于粗糙的近似而导致的问题。如果每帧有更多的可用时间，则可以使用一些更加精细的方法。

体素锥形追踪，无论是使用稀疏八叉树进行存储[307]，还是其级联版本（章节 11.5.7）[1190]，同样可以用于渲染高光效果。该方法会将场景表示存储在稀疏体素八叉树中，并在这个体素数据结构中进行锥形跟踪。一次锥形追踪只会返回一个值，这

个值表示了来自该圆锥所对应立体角的平均 radiance。对于漫反射光照而言，我们需要对多个方向上的圆锥进行追踪，因为只用一个圆锥是不准确的。

对于光泽材质而言，使用锥形追踪的效率要高得多。在镜面光照的情况下，BRDF 的波瓣会很狭窄，只需要考虑一个来自较小立体角的 radiance，因此我们不再需要同时追踪多个圆锥区域，在大多数情况下，一个着色点只需要追踪一个圆锥就足够了。只有较为粗糙材质上的高光效果，才可能需要追踪多个圆锥，但是又因为这样的反射效果十分模糊，在这种情况下，我们只需要使用局部反射探针即可，根本不需要执行锥形追踪。

与之相反的是高度抛光的材质，它们的反射效果几乎像镜子一样，这会使得所进行追踪的圆锥区域变的十分狭窄，就像一条射线一样。有了这样一个精确的追踪，底层场景表示的体素本质可能会在反射中被注意到，所产生的反射效果将会表现出体素化的立方体外观，而不是多边形几何。但是这个瑕疵在实践中很少会成为一个问题，因为反射效果很少会被人们直接观察到，这个反射效果会叠在某个纹理表面上，其最终的贡献值会被纹理进行修正，这个过程通常会掩盖任何缺陷和瑕疵。当需要完美的镜面反射效果时，还可以使用其他方法，从而以更低的成本来实现这个效果。

11.6.4 平面反射

另一种方法是重复使用常规的场景表示，对其进行重新渲染从而创建一个反射图像。如果反射表面的数量是有限的，并且它们都是平面的，我们就可以使用常规的 GPU 渲染管线，来创建从这些表面上所反射的场景图像。这些图像不仅可以提供精确的镜面反射效果，而且还可以通过对图像进行一些额外处理，从而渲染令人信服的光泽效果。

理想的反射面遵循反射定律（law of reflection），即入射角等于反射角。也就是说，入射光线与表面法线之间的夹角，等于反射光线与表面法线之间的夹角，如图 11.39 所示。图 11.39 还展示了一个反射物体的“图像”，根据反射定律，物体的反射图像实际上就是该物体经过平面的物理反射。也就是说，我们沿着入射光线继续前进（注意这里不是反射光线），穿过反射平面，最终可以到达反射物体上相同的位置。

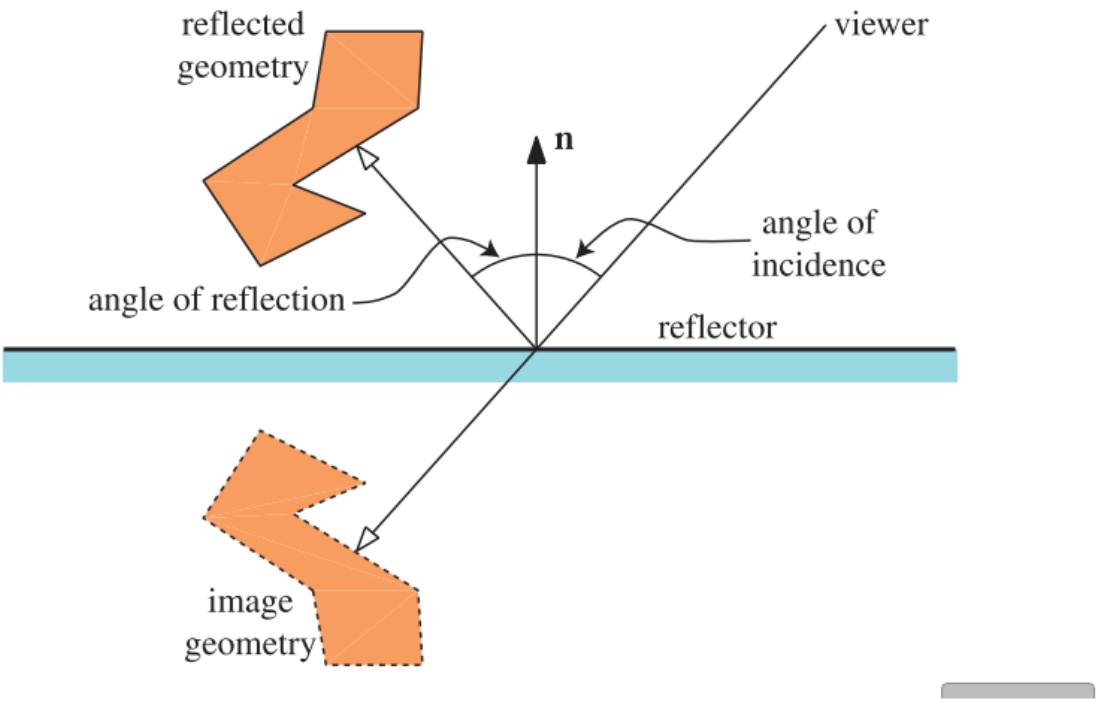


图 11.39：平面上的反射，上图展示了入射角和反射角、反射的几何形状以及反射平面（reflector）。

这就引出了一个原理：可以通过创建物体的一个副本，将其转换到反射位置上，然后再从那里渲染这个副本物体，从而实现反射效果。为了实现正确的光照效果，光源也必须要在平面上进行反射，包括光源的位置和光照的方向[1314]。一种等效的方法是保持场景表示不变，通过镜子将观察者的位置和观察方向反射到反射平面的另一侧。这种反射操作可以通过对投影矩阵进行简单地修改来实现。

位于反射平面背面的物体不应该被反射。这个问题可以使用反射平面的平面方程来解决，最简单的方法就是在像素着色器中定义一个裁剪平面。让这个裁剪平面与反射平面相重合 [654]，在渲染反射场景的时候，使用这个裁剪平面将与观察位置位于同一侧（即原本位于镜子背面的物体）的所有反射几何物体都裁剪掉。

11.6.5 屏幕空间方法

就像环境光遮蔽与漫反射全局光照一样，一些高光效果也可以在屏幕空间中进行计算。由于镜面波瓣比较尖锐，因此这样做要比漫反射情况稍微精确一些。我们只需要在绕反射观察向量的有限立体角范围内，即可获得有关 radiance 的信息，而不需要在整个半球范围内进行计算，因此屏幕空间中的数据会更有可能包含这个信息。这种类型的方法最早由 Sousa 等人[1678]提出，同时也被其他开发人员所发现，整个系列的方法被称为屏幕空间反射（screen-space reflections，SSR）。

给定着色点的位置，观察向量和法线，我们可以沿着法线反射的观察向量来追踪一根光线，并测试其与深度缓冲的交点。这个测试是通过沿着光线进行迭代移动，每次步进一定的距离，并将光线此时的位置投影到屏幕空间中，再从 z-buffer 深度中检索该位置的深度信息来完成的。如果此时光线上的点到相机的距离，要比深度缓冲中对应位置的几何物体的深度更远，这意味着光源位于几何物体的内部，此时我们就可以认为光线与场景相交。然后我们可以从颜色缓冲中的对应位置处，读取到相应的颜色值，从而获得追踪方向入射到着色点的 radiance。这种方法假设光线照射到的表面是 Lambertian 表面，但是这个条件只是许多方法的一种近似，在实践中当然可以使用其他 BRDF。光线可以在世界空间中以均匀的步长进行追踪，但是这样做所获得的交点信息十分粗糙，因此当检测到光线与场景相交时，可以执行一个细化检索的 pass，在有限的距离内可以使用二分查找来精确定位交点的位置。

McGuire 和 Mara 指出[1179]，由于透视投影的原因，在世界空间中以均匀间隔进行步进，所产生的采样点分布在屏幕空间中是不均匀的。在靠近相机的光线部分会采样不足，因此可能会错过一些光线与场景相交的位置；而距离相机较远的光线部分则会被过采样，因此相同的深度缓存像素会被多次读取，从而产生不必要的内存流量和冗余计算。他们建议使用一种数值微分法（digital differential analyzer，DDA）来在屏幕空间中执行射线步进，DDA 是一种可以用于光栅化线条的方法。

首先，将待追踪光线的起点和终点都投影到屏幕空间中，沿着这条线段依次检查每个像素，以保证均匀的采样精度。使用这种方法的一个结果是，在执行相交测试的时候，不需要对每个像素的观察空间（view-space）深度进行重建。观察空间中深度值的倒数，即在常规透视投影下存储在 z-buffer 中的值，这个值在屏幕空间中呈线性变化。这意味着我们可以在实际追踪之前，计算该像素对屏幕空间 x 坐标和 y 坐标的导数（斜率），然后再使用简单的线性插值来获得屏幕空间线段上任何位置的值。使用这种方法计算出来的值，可以直接与深度缓冲中的数据进行比较。

基本形式的屏幕空间反射只对一条光线进行追踪，因此只能提供镜面反射效果。然而，完美的镜面是相当罕见的，在现代基于物理的渲染管线中，光泽反射是更加常见也是更加需要的，SSR 同样也可以用于渲染这些效果。

在简单的临时方法中[1589, 1812]，反射仍然沿着反射方向使用单一的光线追踪，并将结果存储在离屏缓冲区中，在后续步骤中进行处理。通过使用一系列的滤波核，通常还会与缓冲区的下采样操作相结合，从而创建一组具有不同模糊程度的反射缓冲区。在计算光照的时候，BRDF 波瓣的宽度决定了哪个反射缓冲区会被采样。虽然通常会选择与 BRDF 波瓣形状相匹配的滤波核，但是这样做（模糊）只是一个粗略的近似，因为在进行屏幕空间过滤时，并不会考虑不连续性、表面朝向以及其他对结果精

度至关重要的因素。最后会添加自定义的启发式方法，使得屏幕空间中的光泽反射，在视觉上与其他来源的镜面反射相匹配。尽管这只是一个近似值，但是最终生成的结果还是令人信服的。

Stachowiak [1684]以一种更有原则的方式来处理这个问题。计算屏幕空间反射是光线追踪的一种形式，就像常规的光线追踪一样，它可以用于执行适当的蒙特卡洛积分。他不仅使用了反射观察方向，还使用了对 BRDF 的重要性采样以及光线的随机发射。由于性能的限制，光线追踪是在屏幕半分辨率下完成的，每个像素上只会追踪少量光线（1 到 4 根）。由于所使用的光线太少，会产生有噪声的图像，因此相交结果会在相邻像素之间进行共享。对于一定范围内的像素，假设它们的局部可见性是相同的。如果从点 $\mathbf{p}_0$ 向方向 $\mathbf{d}_0$ 发出的光线与场景在点 $\mathbf{i}_0$ 处相交，那么我们可以假设，如果从点 $\mathbf{p}_1$ 向方向 $\mathbf{d}_1$ 发出一条光线，它也会和场景相交于点 $\mathbf{i}_1$ ，并且在点 $\mathbf{i}_1$ 之前不会与其他任何表面相交。这样我们可以直接重复使用光线数据，不需要真的对其进行追踪，只需要适当地修改这次追踪对邻域积分的贡献值即可。从形式上讲，在计算当前像素 BRDF 的概率分布函数（pdf）时，从相邻像素发出光线的方向将具有不同的概率分布。

为了进一步增加光线的有效数量，还需要对结果进行时域过滤。通过离线计算与场景无关的部分积分项，并将其存储在由 BRDF 参数索引的查找表中，还可以进一步降低最终积分结果的方差。如果反射光线的所有信息都可以在屏幕空间中找到，那么以上的这些策略可以让我们获得精确的、无噪声的结果，这个结果接近于离线路径追踪所获得的 ground–truth 图像，如图 11.40 所示。

图 11.40：这幅图像中的所有高光效果，都是使用随机屏幕空间反射（stochastic screen-space reflflection）算法渲染的[1684]。请注意反射效果的垂直拉伸，这是微表面模型反射的特点。

在屏幕空间中进行光线追踪操作的成本通常是很高的。因为它包含了对深度缓冲的重复采样（可能会有多次），并且还会对查找结果执行某些额外的操作。由于这个读取过程是相当不连贯的，因此缓存的利用率可能会很差，从而导致着色器在执行期间为了等待内存数据的返回，而发生长时间的停滞。因此在具体的实现过程中需要格外注意，尽可能得优化执行效率。屏幕空间反射通常会在一个降低的分辨率下进行计算[1684, 1812]，并使用时域过滤来弥补因为追踪分辨率下降而带来的质量下降。

Uludag [1798]描述了一种使用分层深度缓冲（Hi-Z，章节 19.7.2）来加速光线追踪的优化方法。首先需要创建一个层次结构，深度缓冲会逐步进行下采样操作，每一步的下采样系数在每个方向上均为 2。较高层级上的像素会存储较低层级上的四个对应像素中的最小深度值。接下来会使用这个层次结构来执行光线追踪。如果在给定的步骤中，光线穿过了单元格，但是没有击中单元格中存储的几何图形，那么则将光线推进到单元格的边界，并在下一次步进中使用更低分辨率的缓冲，更低分辨率的缓冲区意味着更大的步长。如果光线在当前单元格中发生了相交，则将光线推进到相交位置，并在下一次步进中使用更高分辨率的缓冲，更高分辨率的缓冲区意味着更小的步

长。在命中最高分辨率的缓冲区时，追踪过程会被终止，此时认为光线与场景相交。这个动态步进过程如图 11.41 所示。

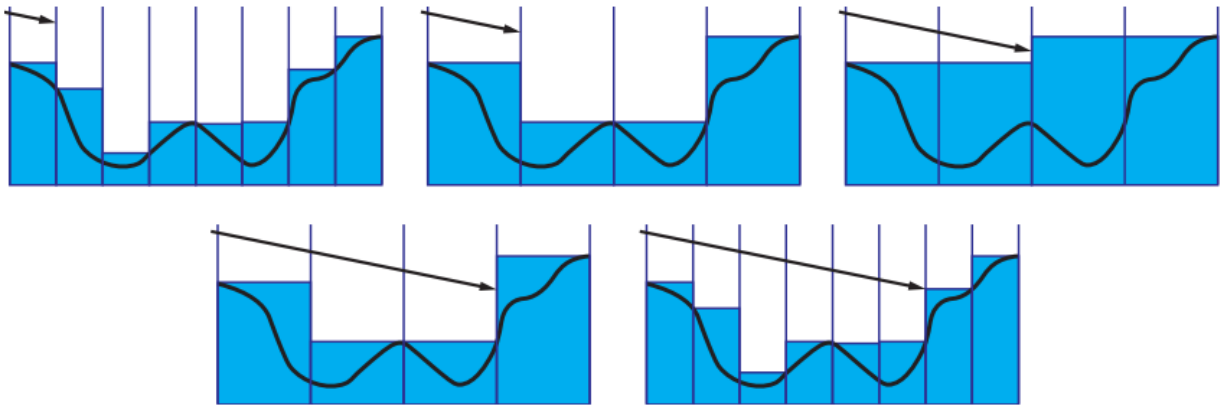


图 11.41：通过分层深度缓冲来进行光线追踪。如果光线在穿过像素时没有击中几何图形，则在下一次步进中会使用较低分辨率的缓冲。如果发生了相交，则接下来的步进将会使用更高分辨率的缓冲。这个过程允许光线以较大的步进长度遍历空白区域，从而提供更高的性能表现。

这个方案特别适用于较长距离的追踪，因为首先它确保了不会遗漏交点，同时还允许光线以较大的步长进行步进。它还可以很好地访问缓存，因为深度缓冲不是在随机的、遥远的位置上进行读取的，而是在本地邻近区域上读取的，这样大大提高了缓存效率。Grenier [599]还给出了实现这个方法的许多实用技巧。

其他人则完全避免使用光线追踪。Drobot [384]通过反射代理重用交点的位置，并从那里查找屏幕空间中的 radiance。Cichocki [266]假设了平面反射器，他没有使用光线追踪，相反他执行了一个全屏 pass，其中每个像素会将自身的像素值写入对应的反射位置中。

与其他屏幕空间的方法一样，由于屏幕空间中的信息是有限的，据此生成的反射效果也会受到有限数据所造成的瑕疵影响。对于反射光线而言，在没有与场景几何相交的情况下就离开屏幕区域，或者是击中场景几何的背面，这些情况是很常见的，在这些情况下，我们无法获得可用的光照信息。这种情况需要进行优雅（gracefully）地处理，因为即使是相邻像素，光线追踪的有效性也会经常不同。可用使用一些空间滤波器来部分填充追踪缓冲区中的空白区域[1812, 1913]。

SSR 的另一个问题是缺乏关于深度缓冲中物体厚度的信息。因为深度缓冲中只存储了一个深度值，所以当光线到达由深度信息所描述的表面背后时，我们无法判断光线是否击中了场景中的其他物体。Cupisz [315]讨论了各种低成本的方法，来减轻由于不知道深度缓冲中物体的厚度而产生的瑕疵。Mara 等人[1123]描述了深度 G-buffer，它存储了多层数据，因此包含了更多有关表面和环境的信息。

屏幕空间反射是一个很好的工具，它可以提供一组特定的效果，例如在近乎平坦的平面上，渲染邻近物体的局部反射效果。它能够大大提高实时高光照明的质量，但是它们并没有提供一个完整的解决方案。本章节中所介绍的各种方法通常会叠加在一起使用，从而构建一个完整而健壮的系统。通常会将屏幕空间反射作为第一层方案，如果它无法提供准确的结果，则使用局部反射探针作为备用。如果给定区域中没有探针，则使用全局的默认探针[1812]。这种类型的设置思路，可以提供一种一致且健壮的方式，来获得令人信服的间接镜面反射效果，这对于生成可信外观而言尤其重要。

11.7 统一方法

到目前为止我们所介绍的方法，它们可以组合成一个能够渲染漂亮图像的完整系统。然而，这些系统交错在一起，缺乏路径追踪的优雅性和概念简洁性。渲染方程的不同方面都会以不同的方式进行处理，在每个方面都做出了不同程度的妥协。尽管最终的生成图像看起来很逼真，但是在很多情况下，这些方法依然会失败，导致视错觉的中断。由于上述的这些原因，实时路径追踪一直是研究工作的重点。

通过路径追踪来渲染可接受质量的图像，其所需的计算量远远超过了 CPU 的能力，即使是最快的 CPU 也不行，因此通常会使用 GPU 来进行计算。GPU 极快的计算速度和计算单元的灵活性，使得它们成为这项任务的良好候选者。实时路径追踪的应用包括建筑可视化以及电影渲染预览等。对于这些情况而言，较低且可变的帧率是可以接受的。可以使用渐进收敛（progressive refinement）（章节 13.2）等技术来提高相机静止时的图像质量。高端系统则可以同时使用多个 GPU。

相比之下，游戏需要以最终的质量要求来渲染每一帧，并且需要能够在预算时间内稳定运行。GPU 可能还需要执行一些其他任务，而不仅仅是渲染。例如：诸如粒子模拟之类的系统，通常会放到 GPU 上进行执行，从而释放一些 CPU 的处理能力。所有这些因素结合在一起，使得路径追踪在如今的游戏渲染中变得不切实际。

在图形学界有一种说法:“光线追踪是未来的技术，并且将永远是！”这句讽刺暗示了这个问题的复杂程度，即使硬件速度和算法都有了巨大的进步，也总会有更加高效的方法来处理渲染管线中的特定部分。使用额外的开销，并且只使用光线投射和主要的可见性（深度缓冲），可能很难证明是合理的，目前有相当多的事实可以佐证它，因为 GPU 从未被设计用于执行高效的光线追踪，它们的主要目标一直是光栅化三角形，并且 GPU 已经在这项任务上已经变得非常擅长了。虽然光线跟踪的过程可以被映射到 GPU 中进行，但是目前的解决方案还没有任何固定功能的硬件对其进行直接

支持。想要使用运行在 GPU 计算单元上的软件解决方案，来击败硬件光栅化是很困难的。

> 译者注：硬件方面出现了 RT Core，专门用于构建加速结构和光线求交；软件方面出现了 UE5 的 Lumen。

更加合理、但不那么纯粹的方法是，使用路径跟踪方法来处理光栅化渲染框架内难以实现的效果。我们对相机可见的三角形进行光栅化，但是在计算反射效果的时候，我们不再依赖近似的反射代理或者不完整的屏幕空间信息，而是通过路径追踪来计算。我们不再尝试模拟具有模糊效果的面光源阴影，而是直接通过向光源追踪光线并计算正确的遮挡信息。我们要发挥 GPU 的优势，对于硬件无法有效处理的元素，会使用更加通用的解决方案来进行处理。虽然这样的系统仍然是一个拼凑起来的系统，并且还是缺乏路径追踪的简洁性，但是实时渲染总是包含了各种妥协。如果要为了节省几毫秒而不得不放弃一些优雅简洁性，那它就是正确的选择，因为帧率是没有商量余地的。

虽然我们可能永远无法声称实时渲染是一个“已解决的问题”，但是更多地使用路径追踪将有助于将理论和实践更紧密地结合在一起。随着 GPU 的计算速度越来越快，在不久的将来，这种混合式解决方案应该可以适用于大多数应用程序，甚至可能会适用于最苛刻的应用程序。已经出现了一些基于这些原则构建的初始系统 [1548]。

> 译者注：上述思路即混合渲染管线（Hybrid Rendering Pipeline）。

光线追踪系统依赖于加速方案的使用，例如层次包围结构（bounding volume hierarchy，BVH），这个加速结构用于对可见性测试进行加速。有关这个话题的更多信息，详见章节 19.1.1。一个原生的、简单的 BVH 实现其实并不能很好地映射到 GPU 上。如第 3 章所述，GPU 会原生执行若干个线程组，这些线程组称为 warp 或者 wavefront。一个 warp 是通过锁步（lock-step）进行处理的，即一个 warp 中的每个线程都会执行相同的操作。如果 warp 中的某些线程不执行代码的特定部分（例如分支），那么它们会被暂时禁用。出于这个原因，GPU 中的代码应该以一种特殊方式进行编写，使得一个 wavefront 中各个线程之间控制流的发散最小化。假设每个线程只处理一根光线，那么这种方案通常会导致线程之间产生较大的分歧和发散。不同的光线将执行遍历代码的不同发散分支，并在这个过程中与不同的边界体积相交，其中的有些光线会比其他光线更早完成树结构的遍历。这种行为偏离了 GPU 的理想使用状态，即所有线程都在使用 GPU 的计算能力。为了消除这些低效问题，人们开发了一些遍历方法，来最小化线程分歧并重新使用提前结束的线程[15, 16, 1947]。

为了生成高质量的图像，可能需要对每个像素追踪成百上千条光线。即使是使用最优的 BVH、最高效的树遍历算法和最快速的 GPU，目前也只能在最简单的场景中实时做到这一点，而在稍微复杂一点的场景中则根本无法实现。在可用的性能限制下，我们所生成的图像会具有非常多的噪点，根本无法用于显示。然而幸运的是，这些充满噪声的图像可以使用降噪算法来进行处理，从而产生基本无噪声的图像，如图 11.42 和图 24.2 所示。最近在实时光追降噪领域取得了令人印象深刻的进展，并且开发出了一些算法，可以在每像素仅追踪一根光线的情况下（1spp），创建视觉上接近高质量的、路径追踪生成的图像 [95, 200, 247, 1124, 1563]。

图 11.42：时空方差引导滤波（spatiotemporal variance-guided filtering，SVGF）可以对每像素仅使用一个样本（1spp）的路径追踪图像（左）进行降噪处理，从而创建平滑的无瑕疵图像（中）。其质量与每像素使用了 2048 个样本的参考图像（右）相当。

2014 年，PowerVR 发布了他们的 Wizard GPU [1158]。除了常规的功能之外，其硬件中还包含了构建和遍历加速结构的特殊单元（详见章节 23.11）。该系统证明了使用固定功能的硬件单元来加速光线投射的能力和吸引力。见证未来可能会发生什么将是十分令人兴奋的！

补充阅读和资源

Pharr 等人的《Physically Based Rendering》[1413]一书，是非交互式全局光照算法的优秀指南，他们的工作特别具有价值，这在于他们深入地描述了他们所发现的有用方法。Glassner 的书《Principles of Digital Image Synthesis》[543, 544]（现在是免费的），在物理方面讨论了光线与物质的相互作用。Dutre 等人[400]的《Advanced Global Illumination》为辐射度量学和求解 Kajiya 渲染方程（主要是离线求解）提供了基础。McGuire [1188]的《Graphics Codex》是一本电子参考书，

其中包含了大量与计算机图形学相关的方程和算法。Dutre 撰写的《Global Illumination Compendium》[399]所参考的工作是相当古老的，但是它是免费的。Shirley 的一系列短书《Ray Tracing in One Weekend》[1628]是一个廉价且快速学习光线追踪的方法。

Chapter 12 Image-Space Effects 图像空间特效

> Ernest Hemingway——“The world was not wheeling anymore. It was just very clear and bright and inclined to blur at the edges.”

> 欧内斯特·海明威——“世界不再转动，它变得清晰而明亮，边缘似乎变得有些模糊。”（美国著名作家；1899—1961）

制作一副图像要比简单地描绘（portray）物体复杂得多，想要让图像看起来具有真实感，其中一方面是让它看起来像一张照片。就像摄影师对他们的摄影作品进行后期调整一样，我们也希望能够对渲染出的图像进行修改调整，例如色彩平衡等。在渲染图像中添加胶片颗粒（film grain）、暗角（vignetting）以及其他细节变化，可以使其看起来更有说服力。此外，一些更加引人注目（dramatic）的效果，例如镜头光晕（lens flare）和泛光（bloom）可以增加艺术感。塑造景深（depth of field）和运动模糊（motion blur）等效果还可以增加真实感，也可以用于生成一些艺术效果。

利用 GPU 可以高效地对图像进行采样和处理。本章节将首先讨论如何使用图像处理技术（image processing technique）来修改渲染后的图像。还可以使用深度、法线等额外数据来对这些操作进行增强，例如：对噪声区域进行平滑的同时，仍然保留尖锐的边缘。重投影（reprojection）方法可用于节省着色的计算，或者快速创建缺失的帧。最后，我们会介绍各种基于样本的技术，来产生镜头光晕、泛光、景深、运动模糊以及其他效果。

12.1 图像处理

图形加速器通常用于从几何描述和着色描述中创建人工场景，而图像处理则不同，我们输入一副图像，并使用各种方式对其进行它修改。有了可编程的着色器，以及使用输出图像作为输入纹理的能力，我们将二者相结合，为使用 GPU 进行各种各样的图像处理操作开辟了道路。这种效果可以与图像合成（image synthesis）相结合。通常，我们会生成一个图像，然后对其执行一个或者多个图像处理操作。在渲染之后对图像进行修改的过程叫做后处理（post-processing）。在渲染一帧的时候，可以执

行大量访问图像、深度以及其他缓冲区的 pass [46, 1918]，例如：游戏《战地 4》拥有超过 50 种不同类型的渲染 pass，尽管在一帧中不会同时执行所有的这些 pass [1313]。

使用 GPU 进行后处理操作有几个关键技术。场景以某种形式渲染到离屏缓冲区中，例如颜色图像和 z-buffer 等。这些渲染得到的图像会被视为纹理，这些纹理会被应用到填充屏幕的四边形上。后处理实际上就是通过渲染这个四边形来执行的，会为每个像素调用像素着色器程序。大多数图像处理效果，都依赖于在相应的屏幕像素处检索图像纹素的信息。根据系统限制和具体算法，可以通过从 GPU 检索像素位置，或者将 $[0, 1]$ 范围内的纹理坐标分配给这个屏幕四边形，并按照传入图像的大小进行缩放来实现这个过程。

实际上，使用一个填充屏幕的三角形，可能要比四边形更加高效。例如：当使用单个屏幕三角形而不是两个三角形组成的屏幕四边形时，由于更好的缓存一致性（cache coherency）[381]，在 AMD GCN 架构上的图像处理速度提高了近 10%。这个三角形被设计得足够大，可以填满整个屏幕[146]，如图 12.1 所示。无论使用什么基本几何物体，其目的都是一样的：对屏幕上的每个像素都运行一遍像素着色器。这种类型的渲染称为全屏 pass（full screen pass）。如果可以的话，我们还可以使用计算着色器来执行这些图像处理操作，这样做有几个优点，我们会在之后进行介绍。

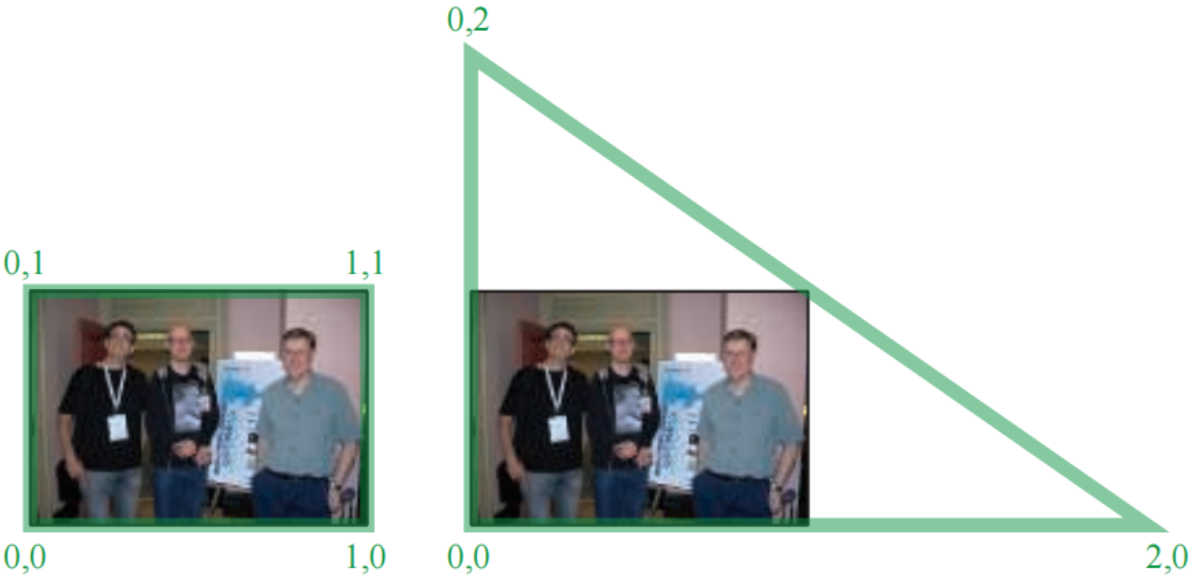


图 12.1：左边使用了一个屏幕填充的四边形，并展示了对应的 (u, v) 纹理坐标。右边则使用了一个三角形来填满整个屏幕，并对其纹理坐标进行了适当的调整，从而提供与左图相同的映射效果。

使用传统的渲染管线，像素着色器阶段可以访问图像数据，检索所有相关的相邻样本并对其进行操作，相邻样本的贡献度会根据它与待评估像素的相对位置来进行加权。

有些操作，例如边缘检测（edge detection），具有一个固定大小的邻域（例如 3×3 像素），邻域内每个像素都有不同的权重（有时也可能是负的）。每个纹素值乘上其相应的权重，并将邻域内的结果相加，从而获得最终结果。

如章节 5.4.1 所述，可以使用各种各样的滤波核来重建信号，使用类似的方式，还可以使用滤波核对图像进行模糊操作。一个旋转不变的滤波核（rotation-invariant filter kernel）是指，它在为纹素分配权重的时候不依赖于径向夹角。也就是说，这样的滤波核，其权重完全取决于纹素到滤波中心的距离。方程 5.22 所展示的 $sinc$ 滤波器就是这样的一个例子。高斯滤波器是一种常用的滤波核，其形状是著名的钟形曲线：

$$\text{Gaussian}(x) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right) e^{-\frac{r^2}{2\sigma^2}} \tag{12.1}$$

其中 r 是到纹素中心的距离， σ 为标准差（standard deviation）， σ^2 是方差（variance）。标准差 σ 越大，所形成的钟形曲线就越宽。一个粗略的经验法则是，使用大小为 3σ 或者更宽的滤波核（ 3σ 原则）[1795]。更大的滤波范围会使图像变得更加模糊，相应也会带来更多的内存访问和开销。

e 前面的系数项使得连续曲线下的面积为 1。然而，在构建离散滤波核的时候，这一项是不相关的。将滤波核内的每个纹素权重相加，然后再让权重除以这个和，就可以让最终的权重之和为 1。由于这个归一化过程，因此方程前面的常数项是没有任何作用的，它通常也不会出现在滤波核的数学描述中。图 12.2 展示了二维高斯滤波器和一维高斯滤波器的形成过程。

(a)

0.0030	0.0133	0.0219	0.0133	0.0030
0.0133	0.0596	0.0983	0.0596	0.0133
0.0219	0.0983	0.1621	0.0983	0.0219
0.0133	0.0596	0.0983	0.0596	0.0133
0.0030	0.0133	0.0219	0.0133	0.0030

(b)

0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545

(c)

0.0545	0.0545	0.0545	0.0545	0.0545
0.2442	0.2442	0.2442	0.2442	0.2442
0.4026	0.4026	0.4026	0.4026	0.4026
0.2442	0.2442	0.2442	0.2442	0.2442
0.0545	0.0545	0.0545	0.0545	0.0545

图 12.2：执行高斯模糊的一种方法是，对一个 5×5 的区域进行采样，并对每个贡献加权求和。上图的（a）部分展示了 $\sigma = 1$ 时，滤波核中的各个权重。第二种方法则是使用分离的滤波器。连续执行两个一维高斯模糊（b）和（c），可以得到相同的结果。第一次 pass，会使用图（b）中所展示的一行权重，在每行中使用 5 个样本来对像素进行水平模糊。第二个 pass，将第一个 pass 中所生成的图像作为输入，会使用图（c）中所展示的一列权重，在每列中使用 5 个样本来对像素进行垂直模糊，从而获得最终结果。将（b）中的权重乘上（c）中的权重，可以得到与（a）相同的权重，这表明两个滤波器实际上是等效的，因此是可分离的。使用（a）对一个像素进行滤波需要 25 个样本，而单独使用（b）和（c）对一个像素进行滤波分别只要 5 个样本，总共 10 个样本，这大大降低了计算量。

$sinc$ 滤波器和高斯滤波器的一个问题是，这两个函数会无限延申。一种变通方法是将这样的滤波器 clamp 到一个特定的直径或者正方形区域内，超出范围的函数部分直接将其视为 0。不同的滤波核针对不同的特性来进行设计，例如易于控制、平滑或者易于计算等。Bjorke [156]和 Mitchell 等人[1218]提供了一些常见的旋转不变滤波器，以及在 GPU 上进行图像处理的其他信息。

任何全屏滤波操作都会试图从图像边界以外的地方来采样像素，例如：如果我们正在为屏幕左上角像素收集 3×3 的滤波样本，那么实际上我们正在尝试检索一些不存在的纹素。一个基本的解决方案是将纹理采样器设置为 clamp 到屏幕边缘，当请求屏幕外不存在的纹素时，将会检索距离最近的边缘纹素。这会导致图像边缘的出现一些滤波错误，但是这些错误通常都是不明显的。另一种解决方案是以一个略高于显示区域的分辨率来生成要过滤的图像，从而使得这些屏幕外像素存在，在进行滤波的时候，只对屏幕内的纹素执行滤波操作。

使用 GPU 的一个优点在于，可以使用内置的插值硬件和 mipmap 硬件来帮助最小化要访问的纹素数量。例如：假设我们使用的是一个 box 滤波器，即对给定纹素周围 3×3 网格内的 9 个纹素求平均值，并显示这个模糊后结果。像素着色器会对这 9 个纹素样本进行加权求和，并将模糊后的结果输出到对应像素上。

然而，其实我们并不需要显式地访问这 9 个样本。通过对纹理进行双线性插值，一次纹理访问可以检索最多四个相邻纹素的加权和[1638]。使用这个想法， 3×3 的滤波网格实际上只需要进行四次纹理访问就可以完成采样，如图 12.3 所示。对于一个权重相等的 box 滤波器，可以将单个样本可以放在四个纹素的中间，从而获得这四个纹素的平均值。对于像高斯滤波器这样权重不同的滤波器，使用四个样本之间的双线性插值是不准确的，可以将每个样本仍然放置在两个纹素之间，但是会更加偏向其中的某一个纹素。例如：假设其中一个纹素的权重为 0.01 ，其相邻纹素的权重为 0.04 。那么样本可以这样放置，使其与第一个纹素的距离为 0.8 ，与第二个纹素的距离为 0.2 ，从而使得每个纹素都具有适当的比例。这样一来，这个样本的权重将是两个纹素权重之和，即 0.05 。或者，还可以对每四个纹素使用双线性插值样本，并找到最接近理想权重的偏移量，从而对高斯函数进行近似。

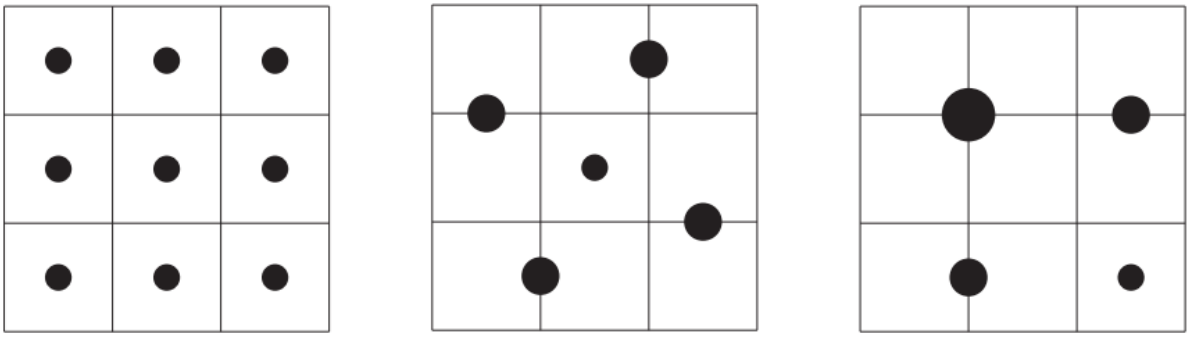


图 12.3：左侧：通过执行 9 次纹理采样，并将贡献值进行求和平均，从而构建一个 box 滤波器。中间：使用了 5 个样本的对称模式，其中位于外部的样本分别代表了 2 个纹素，因此每个样本的权重都是中心样本的两倍。这种类型的模式也可以用于其他滤波核，其中通过在两个纹素之间移动外部样本，使得这些外部样本更加靠近或者远离其中一个纹素，从而改变每个纹素

的相对贡献。右侧：使用了更加高效的四样本模式。其中左上角的样本用于在四个纹素之间进行插值。右上角和左下角的样本各自插值了两个纹素。每个样本被赋予的权重大小，与它所代表的纹素数量成比例。

有些滤波核是可分离的（separable）。最典型的两个例子是高斯滤波器和 box 滤波器，这意味着它们可以使用两个独立的一维滤波操作来代替一次二维滤波操作。这样做可以使得总体上需要访问的纹素大大减少。时间复杂度从 d^2 降低到 $2d$，其中 d 是滤波核直径或者支持范围[815, 1218, 1289]。例如：假设 box 滤波器将应用在图像中每个像素周围的 5×5 区域中。首先，可以对图像进行水平过滤：每个像素左侧的两个相邻纹素与右侧的两个相邻纹素，以及滤波像素本身，都具有相等的权重 0.2，然后再将这五个纹素的值进行加权求和。然后，再将得到的图像在垂直方向上进行模糊，使用上面两个相邻纹素和下面两个相邻纹素，再加上滤波像素本身，进行加权求和。最终，我们不是在一次 pass 中访问 25 个纹素，而是在两次 pass 中总共访问了 10 个纹素，如图 12.2 所示。宽度越宽的滤波核，所节省的开销就越多，因为节省了 $d^2 - 2d$。

圆形的圆盘滤波器，对于背景虚化（bokeh）效果很有用（章节 12.4），但是其计算成本通常会很高，因为该滤波器在实数域中是不可分离的。然而，可以通过使用复数来解决这个问题，复数的使用打开了一个广泛的函数家族。Wronski [1923]对这类可分离滤波器的实现细节进行了讨论。

计算着色器很适合进行过滤操作，与像素着色器相比，滤波核的尺寸越大，在计算着色器中进行滤波操作的性能就越好[1102, 1710]。例如：线程组的内存可以在不同像素的滤波计算之间共享图像访问，从而减少带宽压力[1971]。通过使用计算着色器的分散写入（scattered write）操作，可以以常量开销实现任意半径的 box 滤波器。对于水平和垂直的滤波 pass，可以提前计算好一行或者一列中第一个像素的滤波核值。后续的每个像素，通过在滤波核的前端添加下一个样本，并在减去在远端留下的样本，从而确定当前像素的滤波值。这种“移动平均（moving average）”技术可以用于在常数时间内近似任意大小的高斯模糊[531, 588, 817]。

下采样是另一种在模糊处理时常用的 GPU 技术。下采样是指要为当前操作的图像生成一个较小分辨率的版本，例如：沿着两个轴将分辨率减半，创建四分之一屏幕的图像。根据输入数据和算法的需要，原始图像可以被滤波为一个较低的分辨率，或者也可以简单地以这个较低的分辨率来创建一个图像。当这个下采样的图像被混合到最终的完整分辨率图像中时，纹理的放大操作将会使用双线性插值来在样本之间进行混合，这会产生进一步的模糊效果。在原始图像的低分辨率版本上执行相关操作，会大大减少需要访问的纹素数量。此外，任何应用于这个低分辨率图像的滤波核，都会增

加它的相对大小，例如：在低分辨率图像上应用宽度为 5 的滤波核（即中心像素的每侧分别还有两个纹素），其效果类似于在原始图像上应用宽度为 9 的滤波核。这样做的质量会稍微低一些，但是对于大面积相似颜色的模糊而言（这种情况经常出现在光晕效果和其他现象中），大多数瑕疵都很小，难以察觉[815]。还可以通过减少每个像素所存储的 bit 数，来降低内存访问的开销。下采样操作还可以用于其他缓慢变化的现象，例如：许多粒子系统可以以屏幕半分辨率进行渲染[1391]。这种下采样的思想可以扩展到创建图像 mipmap，并从多个层进行采样，从而提高模糊过程的计算速度[937, 1120]。

12.1.1 双边滤波

通过使用某种形式的双边滤波[378, 1355]，可以改善上采样的结果以及其他图像处理操作。其核心思想是丢弃或者降低与中心样本表面无关的样本贡献值，它可以用于保持边缘。想象一下，我们将相机对准一个红色物体，其背后还有一个距离很远的蓝色物体，背景是灰色的。那么这个蓝色物体应当是模糊的，而红色物体应当是清晰的。一个简单的双边滤波器会对像素的颜色进行检查，如果是红色，则不会出现模糊，物体会保持清晰；否则，这个像素就会被模糊处理，所有非红色的样本都将用于模糊像素。如图 12.4 所示。

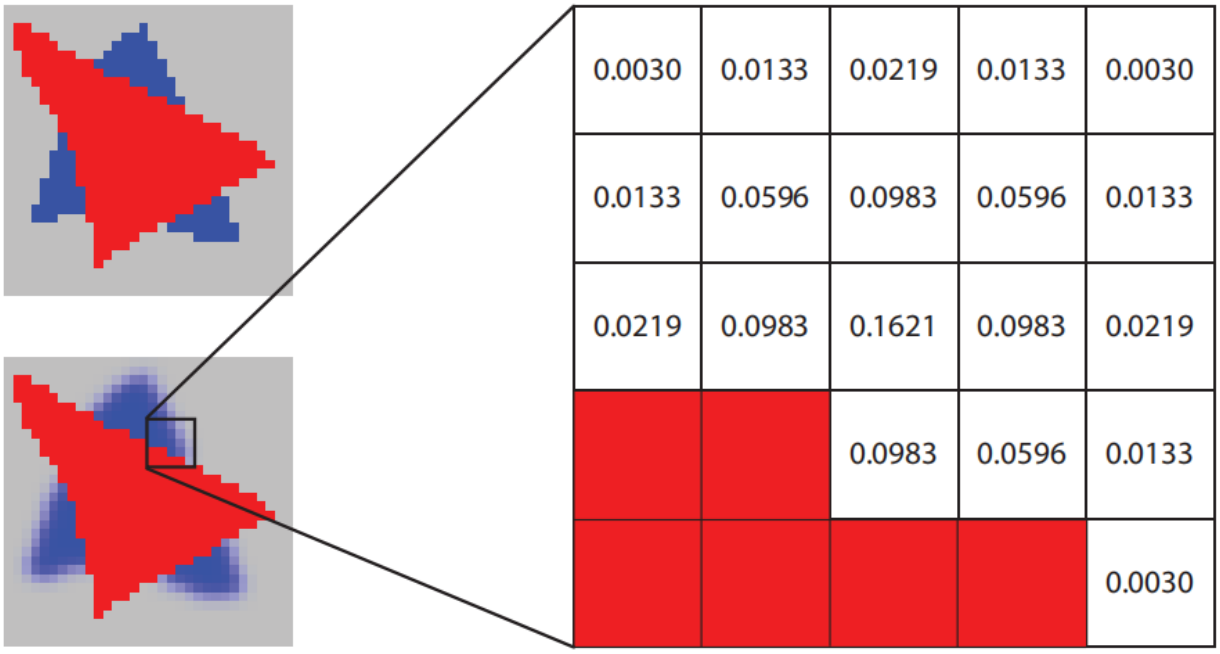


图 12.4：双边滤波。左上角是原始图像。在左下角，我们会对非红色的样本进行模糊，并且在模糊的过程中也仅使用非红色的样本。右侧展示了一个像素的滤波核，在计算高斯模糊的时候，滤波核中的红色像素会被忽略。剩余像素的颜色会乘以相应的滤波器权重并进行求和，然

后再计算这些滤波器权重的和。在这个例子中，权重的总和为 0.8755，因此计算得到的颜色还要除以这个值。

在这个例子中，我们可以通过检查像素颜色来决定忽略哪些像素。联合双边滤波（joint bilateral filter）或者交叉双边滤波（cross bilateral filter）会使用一些额外的信息，例如深度、法线、识别值、速度或者其他数据，来确定是否使用邻近的样本。例如：Ownby 等人[1343]展示了仅使用少量样本进行阴影映射时，对生成的结果进行模糊会使它看起来会变得更好。然而，一个物体上的阴影不应当对另一个不相关的物体产生影响，这种模糊处理会使得物体的阴影从边缘溢出。他们使用了一个双边滤波器，通过比较给定像素的深度与其邻域像素的深度，来丢弃属于不同表面上的样本。以这种方式来减少区域内的可变性被称为降噪（denoising），通常会被用于屏幕空间环境光遮蔽等算法中（章节 11.3.6）[1971]。

接近使用到相机的距离来寻找边缘通常来说是不够的，例如：一个软阴影跨越了两个立方体表面所形成的边缘，这个阴影可能只会落在一个表面上，另一个表面则背对光源。仅仅使用深度信息，可能会导致阴影在模糊时从一个表面溢出到另一个表面上，因为这个边缘并不会被检测到。我们可以在进行模糊处理的时候，只使用那些深度和表面法线与中心样本相似的邻居样本，从而解决这个问题，使得阴影不会溢出到其他表面上。这样做对跨越共享边缘的样本进行了限制，因此这种双边滤波器也称为边缘保持滤波器（edge-preserving filters）。是否削弱或者忽略邻近样本的影响，以及在多大程度上削弱或者忽略邻近样本的影响，这取决于开发人员的决定，还取决于模型、渲染算法和观察条件等因素。

除了花费在检查邻居样本和对权重求和上的额外时间，双边滤波还有一些其他的性能开销。一些常见的滤波优化操作，例如将滤波操作变成两 pass 的分离滤波，或者使用双线性插值进行加权采样，这些优化操作是难以在双边滤波中使用的。由于我们事先无法知道应该忽略哪些样本或者削弱哪些样本的影响，因此我们无法使用 GPU 在一次“tap”中收集多个图像纹素的技术。也就是说，由于可分离的两 pass 滤波器具有明显的速度优势，因此导致了人们致力于开发双边滤波的近似方法[1396, 1971]。

Paris 等人[1355]讨论了双边滤波器的许多其他应用。在必须保留边缘信息但可以重复使用样本从而减少噪声的地方，可以应用双边滤波器。它还可以用于将着色频率与几何图形的渲染频率相解耦。例如：Yang 等人[1944]在较低的分辨率下进行着色处理，然后通过使用法线信息和深度信息，在上采样期间进行双边滤波从而形成最终帧。另一种方法是使用最近深度滤波（nearest-depth filtering），该方法会检索低分辨率图像中的四个样本，并使用深度最接近高分辨率图像的那个样本[816]。Hennessy [717]和 Pesce [1396]对这些方法和其他图像上采样方法进行了对比和比

较。低分辨率渲染的一个问题是，可能会丢失部分精细尺度的细节。Herzog 等人[733]通过利用时间一致性和重投影技术进一步提高了图像质量。需要注意的是，双边滤波器是不可分离的，因为每个像素所采样的样本数量会发生变化。Green [589]则指出，将双边滤波视为可分离滤波器所带来的瑕疵，能够被其他着色效果所隐藏。

实现后处理管线的一种常见方法是使用乒乓缓冲区（ping-pong buffer）[1303]。这是在两个离屏缓冲区之间应用操作的想法，其中每个缓冲区都用于保存中间结果或者最终结果。对于第一个 pass 而言，第一个缓冲区是输入纹理，第二个缓冲区则是纹理输出的地方。在下一个 pass 中，这两个缓冲区的角色会被颠倒，第二个缓冲区现在会作为输入纹理，而第一个缓冲区则被用于输出。在第二个 pass 中，第一个缓冲区中的原始内容会被覆盖，这个过程是暂时的，它被作为一个处理 pass 的临时存储区域。对临时资源的管理和重用，是设计现代渲染系统的关键组成部分[1313]。从架构的角度来看，让每个独立的 pass 执行特定的效果是很方便的；然而，为了提高效率和性能表现，最好在一个 pass 中结合尽可能多的后处理效果[1918]。

在前面的章节中，访问其邻居的像素着色器会被用于形态学抗锯齿（morphological antialiasing）、软阴影、屏幕空间环境光遮蔽和其他技术。后处理效果一般会针对最终图像进行操作，它可以模拟热成像效果（thermal imaging）[734]，重建胶片颗粒（film grain）[1273]和色差（chromatic aberration）[539]，进行边缘检测[156, 506, 1218]，生成热浪（heat shimmer）[1273]和波纹效果（ripple）[58]，对图像进行色调分离（posterize）[58]，帮助渲染云雾效果[90]，以及执行大量的其他操作[156, 539, 814, 1216, 1217, 1289]。章节 15.2.3 中介绍了一些用于非真实感渲染（non-photorealistic rendering）的图像处理技术，图 12.5 则给出了一些例子，它们都使用彩色图像作为唯一的输入。

图 12.5：使用像素着色器进行图像处理操作。其中左上角为原始图像，在其基础上进行了各种处理。右上角为高斯差分运算（Gaussian difference operation），左下角为边缘检测，右下角为边缘检测与原始图像混合后的合成图像。

在本章节结束的时候，我们会介绍一些使用广告牌和图像处理技术进行实现的效果，不再去赘述所有可能的算法。

12.2 重投影技术

重投影技术基于对前一帧计算样本进行重用的思想。顾名思义，这些样本可以从新的观察位置和观察方向尽可能地进行重复使用。重投影方法的一个目标是将渲染成本分摊到多个帧上，即利用时间一致性（temporal coherence）。因此，这也与章节 5.4.2 中所介绍的时域抗锯齿有关。重投影方法的另一个目标是，如果应用程序未能及时完成当前帧的渲染，则构建一个近似结果。这种方法在虚拟现实应用中尤其重要，可以避免模拟器眩晕症（simulator sickness）（章节 21.4.1）。

重投影方法分为反向重投影（reverse reprojection）和正向重投影（forward reprojection）。反向重投影[1264, 1556]的基本思想如图 12.6 所示。当在 t 时刻渲染一个三角形时，会计算当前帧（t）和前一帧（$t-1$）的顶点位置。使用顶点着色中的 z 和 w 分量，像素着色器可以为 t 和 $t-1$ 时刻计算插值 z/w，如果两帧之间的距离足够近，那么可以在前一帧的颜色缓冲中来对位置 $\mathbf{p}_i^{t-1}$ 的颜色进行双线性查找，并使用这个颜色值来代替当前帧的颜色值，而不是重新计算一个新的着色值。对于前一帧中被遮挡的区域（例如图 12.6 中的深绿色区域），这些区域没有相应的着色像素可以使用，这种现象被称为缓存未命中（cache miss）。发生这种事件的时候，我们需要计算新的像素着色值来填补这些空洞。由于着色值的重用假定它们与任何类型的运动（物体、相机、光源）都无关，因此明智的做法是不要在太多帧中重用这些着色值。Nehab 等人[1264]建议，在重复使用几帧之后，应当进行自动刷新。一种方法是将屏幕区域划分为 n 个组，其中每组是伪随机选择的 2×2 像素区域。每一帧都会更新一个组，避免重复使用某些像素值的时间过长。反向重投影的另一种变体是存储一个速度缓冲区（velocity buffer），并在屏幕空间中执行所有测试，这样做可以避免顶点的双重变换。

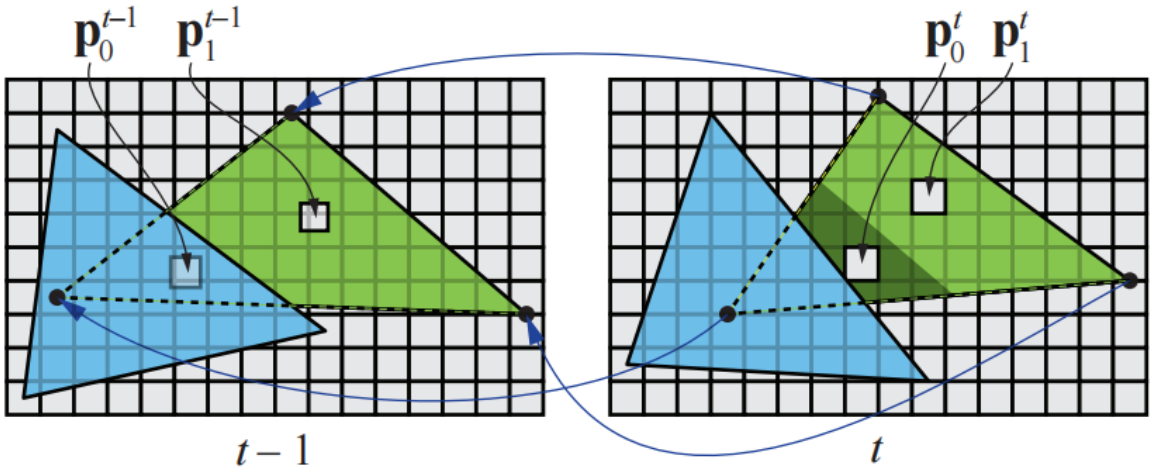


图 12.6：左图是 $t-1$ 时刻的绿色三角形和蓝色三角形；右图是 t 时刻的绿色三角形和蓝色三角形。右图中的三维点 $\mathbf{p}_i^t$ 位于绿色三角形上两个像素的中心，它们连同像素面积一起，被反向重投影到了左图中的点 $\mathbf{p}_i^{t-1}$。从左图中我们可以看到，点 $\mathbf{p}_0^{t-1}$ 被遮挡，而 $\mathbf{p}_0^t$ 则是可见的，在这种情况下，没有着色结果可以被重复使用。然而，点 $\mathbf{p}_1$ 在 $t-1$ 时刻和 t 时刻都是可见的，因此可以对该点重用着色结果。[1264]

为了获得更好的质量，还可以使用一个运行时平均滤波器（running-average filter）[1264, 1556]，它会逐步淘汰旧的着色值。它特别适用于空间抗锯齿（spatial antialiasing）、软阴影和全局光照。这个滤波器的描述如下：

$$\mathbf{c}_f\left(\mathbf{p}^t\right) = \alpha \mathbf{c}\left(\mathbf{p}^t\right) + (1-\alpha)\mathbf{c}\left(\mathbf{p}^{t-1}\right) \tag{12.2}$$

其中 $\mathbf{c}\left(\mathbf{p}^t\right)$ 是点 $\mathbf{p}^t$ 上的新着色值， $\mathbf{c}\left(\mathbf{p}^{t-1}\right)$ 是前一帧中的重投影颜色， $\mathbf{c}_f\left(\mathbf{p}^t\right)$ 是应用滤波器之后的最终颜色。Nehab 等人在某些用例中使用 $\alpha = 3/5$ ，但是他建议根据具体的渲染内容来尝试使用不同的值。

正向重投影则是从第 $t-1$ 帧的像素开始，并将它们投影到第 t 帧中，因此不需要进行两次顶点着色。这意味着来自第 $t-1$ 帧的像素会被分散到第 t 帧中，而反向重投影方法则会收集从第 $t-1$ 帧到第 t 帧的像素值。反向重投影方法还需要处理那些变得可见的遮挡区域，通常会通过一些启发式的空洞填充方法来完成，即使用周围像素的信息来推断出缺失区域的值。Yu 等人[1952]使用正向重投影方法，以一种廉价的方式来计算景深效果。Didyk 等人[350]基于运动向量（motion vector），在第 $t-1$ 帧中自适应地生成网格来避免空洞，而不是使用经典的空洞填充方法。这个网格是通过深度测试来进行渲染的，然后会将其投影到第 t 帧中，这意味着遮挡问题和折叠问题，是作为带深度测试的自适应三角形网格光栅化的一部分来进行处理的。Didyk 等人将他们的方法从左眼重投影到右眼，从而为虚拟现实生成一对立体图像，这两个图像之间的相关性通常会很高。后来，Didyk 等人[351]提出了一种感知驱动的方法（perceptually motivated method）来执行时域采样，例如：将帧率从 40 Hz 增加到 120 Hz。

Yang 和 Bowles [185, 1945, 1946]提出了从 t 和 $t+1$ 两帧中，投影到这两个帧之间 $t+\delta t$ 帧的方法，即 $\delta t \in [0,1]$ 。这些方法有更大的机会能够更好地处理遮挡情况，因为它们使用了两帧而不是一帧。在游戏中使用这种方法，能够将帧率从 30 FPS 提高到 60 FPS，这种效果是可能的，因为该方法的运行时间不到 1 毫秒。我们推荐 Yang 和 Bowles 的课程笔记[1946]，以及 Scherzer 等人[1559]对时域相关方法的广泛调查。Valent [1812]在《杀戮区域：暗影坠落》中也使用了重投影方法来加速渲染。有关使用重投影来实现时域抗锯齿的细节，请参阅章节 5.4.2 末尾的大量参考资料。

12.3 镜头光晕和泛光

镜头光晕（lens flare）是由光线通过间接反射，或者其他非预期途径穿过晶状体系统或者眼睛所引起的现象。耀斑（flare）可以分为几种现象，其中最明显的是光晕（halo）和绒毛状光环（ciliary corona）。光晕是由透镜晶体结构的径向纤维所引起的，它看起来像一个围绕着光源的圆环，其外缘是红色的，而内部则是紫色的。光晕的视觉尺寸是恒定的，与到光源的距离无关。绒毛状光环来自于透镜的密度波动，它表现为从某一点辐射出的射线，这些射线可以延伸到光晕以外的地方[1683]。

当镜头的某些部分在内部反射光线或者折射光线时，镜头也可以产生一些其他的次要效果。例如：由于相机的光圈叶片，可能会产生多边形图案。由于玻璃上的细小凹槽，可能在挡风玻璃上看到条状光线[1303]。泛光是由晶状体和眼睛其他部分的散射所引起的，它会在光源周围产生辉光效果，并降低场景其他地方的对比度。数码相机会通过电荷耦合器件（charge-coupled device，CCD）来将光子转换为电荷，从而实现图像的捕获。CCD 中的一个电荷位置饱和并溢出到邻近的位置时，就会发生泛光现象。光晕、日冕和泛光，这类效果被称为炫光效果（glare effect）。

实际上，随着相机技术的不断进步，大多数这样的人工瑕疵也越来越少。更好的设计、镜头罩和抗反射涂层，可以减少或者消除这些错误的鬼影瑕疵[598, 786]。然而，现在通常会使用一些数字方式，来将这些效果添加到真实照片上。因为计算机显示器所能够产生的光线强度是有限的，因此我们会通过在图像中添加这种效果，来给人眼一种场景或者物体变亮的印象[1951]。在照片、电影和交互式计算机图形程序中，由于这些效果的普遍使用，使得泛光效果和镜头光晕几乎是陈词滥调（cliches）的。然而，如果使用得当，这些效果可以给观众提供强烈的视觉暗示。

为了提供令人信服的效果，镜头光晕应当随着光源位置的变化而变化。King [899]创建了一组具有不同纹理的正方形方块来表示镜头光晕，它们的中心位于从屏幕上光源位置到屏幕中心的直线上。当光源远离屏幕中心时，这些正方形方块会变小且更加透明；随着光源向内移动，它们会变大且更加不透明。Maughan [1140]通过使用 GPU 来计算屏幕面光源的遮挡，从而改变镜头光晕的亮度。他生成了一个单像素的强度纹理，然后使用它来对效果的亮度进行衰减。Sekulic [1600]将光源渲染为单个多边形，使用遮挡查询硬件来给出可见区域的像素计数（详见章节 19.7.1）。为了避免在等待查询值返回给 CPU 时的 GPU 停滞，查询到的结果会在下一帧中进行使用，然后来确定衰减量。由于这些炫光现象的强度可能会以相当连续和可预测的方式进行变化，因此单帧的延迟几乎不会引起感知上的混乱。Gjøl 和 Svendsen [539]首先生成一个深度缓冲（这个深度缓冲也将其用于其他效果），并在镜头光晕将会出现的区域以螺旋模式来采样 32 次，并使用采样结果来对光晕纹理进行衰减。在渲染光晕几何图形时，可见性采样是在顶点着色器中完成的，这样可以避免硬件遮挡查询所造成的延迟。

场景中明亮物体或者光源的条纹，可以通过绘制半透明广告牌，或者对明亮像素本身进行后处理过滤来实现。《侠盗猎车手 5》等游戏使用了一组应用在广告牌上的纹理，来实现这些效果[293]。

Oat [1303]讨论了使用 steerable 滤波器来产生条纹效果的方法。这种类型的滤波器并不是在一个区域上进行对称滤波，而是会在一个给定方向上进行滤波。沿着这个给

定方向的纹素值会被相加，从而产生一个条纹效果。该方法使用了降采样到四分之一宽度和四分之一高度的图像，并使用乒乓缓冲区执行两次 pass，给出了令人信服的条纹效果。图 12.7 展示了这种技术的一个例子。

图 12.7：镜头光晕、星状眩光和泛光效果，以及景深和动态模糊。请注意一些正在移动的球，其表面出现了频闪的现象，这是由于对独立的图像进行了累积。[1208]

除了广告牌之外，还有许多其他的变体和技术，这些技术的效果要更好。Mittring [1229]使用一些图像处理技术来隔离图像中的明亮部分，对它们进行下采样，并在一些纹理中对它们进行模糊。然后再通过复制、缩放、镜像和着色等操作，将它们再次合成到最终图像上。使用这种方法，艺术家无法单独控制每个光晕的外观表现，因为所有的光晕都会使用这个相同的生成过程。然而，图像上任何明亮的部分都可能产生镜头光晕，例如镜面反射、表面的自发光部分、明亮的火花粒子等。Wronski [1919]描述了变形镜头光晕，这是 20 世纪 50 年代使用的电影摄像设备所带来的副产品。Hullin 等人[598, 786]为各种鬼影瑕疵提供了一个物理模型，并通过追踪一组光线来计算其效果。它给出了基于镜头系统设计的合理结果，并在精度和性能之间进行了权衡。Lee 和 Eisemann [1012]在这项工作的基础上，使用了一个线性模型，从而避免

了昂贵的预处理操作，Hennessy [716]则给出了实现细节。图 12.8 中展示了实际产品中所使用的典型镜头光晕系统。

图 12.8：游戏《巫师 3》中生成太阳光晕的过程。首先，对输入图像应用一个高对比度的校正曲线，从而分离太阳未被遮挡的部分。接下来，对图像应用以太阳为中心的径向模糊。如左侧图像所示，模糊是以一系列方式进行执行的，每次模糊都是基于前一次模糊的输出结果。这样做可以创建一个平滑的、高质量的模糊，同时可以在每个 pass 中使用有限数量的样本来提高效率。所有的模糊操作都是在半分辨率下进行的，从而减少时间开销。最终生成的光晕图像会与原始场景渲染的结果进行叠加。

泛光效果，即一个非常明亮的区域，其亮度溢出到相邻的像素上，它是通过结合前文中已经提出的几种技术来实现的。其主要的想法是，创建一个仅由“过曝”明亮物体所组成的泛光图像，对其进行模糊处理，然后再将其合成到正常的图像中。所使用的模糊通常是高斯模糊[832]，但是最近与参考镜头的匹配表明，这个模糊分布应当更倾向于尖峰形状[512]。制作这种图像的一种常见方法是 bright-pass 滤波器：保留任何明亮的像素，将所有暗淡的像素都变成黑色，通常会在过渡点进行一些混合或者缩放[1616, 1674]。对于只有几个小物体的泛光效果，可以通过计算屏幕包围盒，来限制后期处理中模糊 pass 和合成 pass 的范围[1859]。

中间过程的泛光图像可以使用低分辨率进行渲染，例如：将分辨率降低到原始宽度和原始高度的二分之一到八分之一。这样做一方面可以节省时间开销，另一方面有助于提高过滤效果。这幅低分辨率的图像被模糊之后，会与原始图像相结合。这种降低分辨率的方法会被用于许多后处理效果的实现中，就像是压缩颜色分辨率或者降低颜色

分辨率的技术一样[1877]。泛光图像可以被下采样若干次，并从生成的图像中进行重新采样，在最小化采样成本的同时，还可以提供更加广泛的模糊效果[832, 1391, 1918]。例如：在屏幕上移动的单个明亮像素可能会导致闪烁，因为它可能在某些帧中并没有被采样。

由于泛光效果的目标是让图像在明亮的地方看起来过度曝光，因此图像的颜色还会按需要进行缩放，并添加到原始图像中。这种加法混合方式会使得颜色更加饱和，并最终变成白色，这通常正是我们想要的，图 12.9 给出了一个示例。Alpha 混合可以用于更加艺术化的控制[1859]。对于高动态范围（HDR）图像，可以通过过滤（而不是阈值化）来获得更好的结果[512, 832]。LDR 和 HDR 的泛光效果可以分开计算，然后再进行组合，从而以一种更加令人信服的方式来捕获不同的现象[539]。当然还可以有其他的变体，例如：前一帧的结果也可以添加到当前帧中，为动画物体提供一个条纹发光的效果[815]。

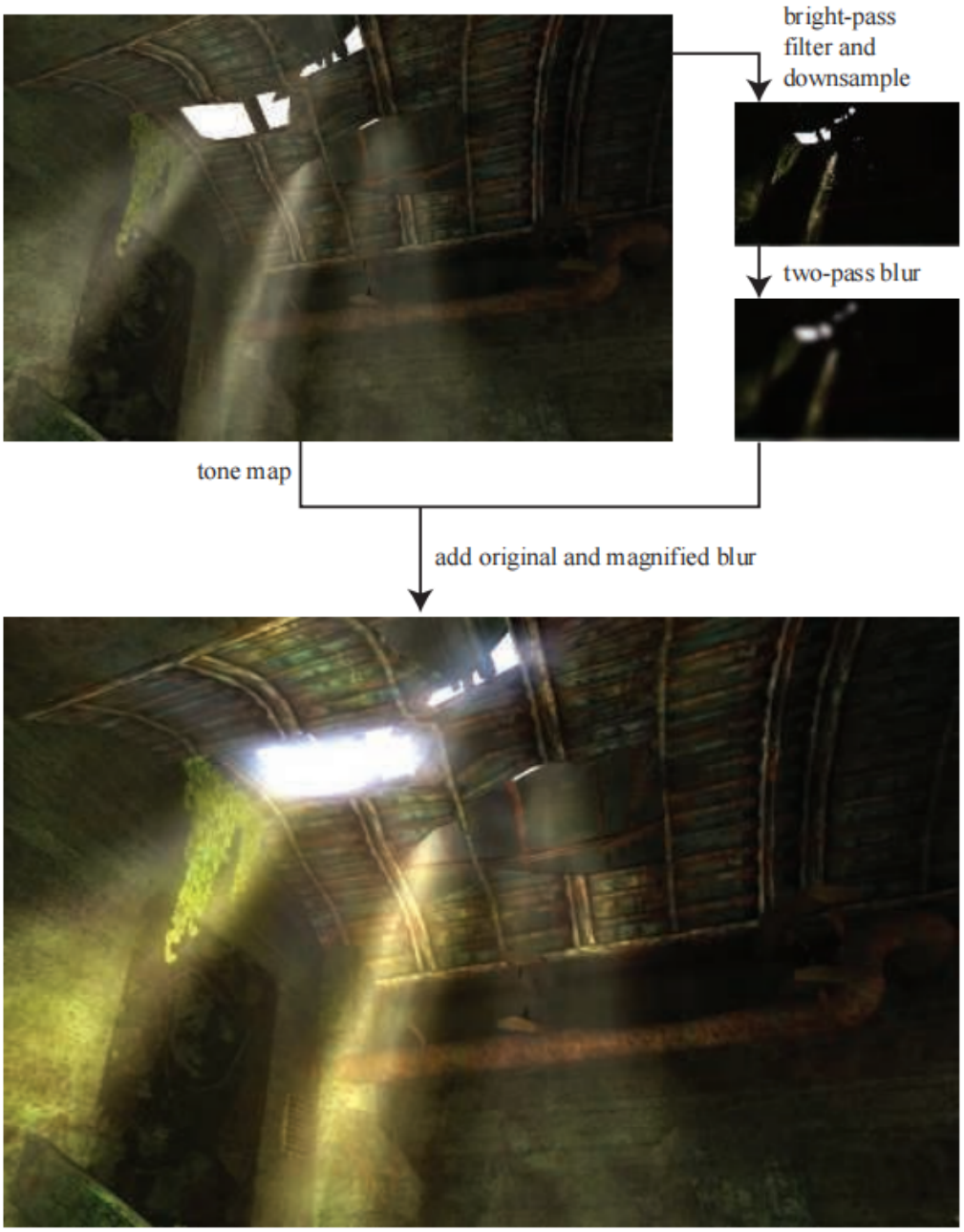


图 12.9：HDR 的色调映射和泛光效果。下面的图像是通过在原始图像上使用色调映射，并在原始图像上添加后处理泛光效果而生成的。[1869]

12.4 景深

对于一个给定设置的相机镜头，它有一个物体聚焦的范围，即它的景深（depth of field）。任何超出这个范围的物体都是模糊的，距离越远越模糊。在摄影中，这种模糊效果的强弱与光圈大小和焦距有关。减小光圈的尺寸可以增加景深，也就是说：减小光圈尺寸，可以增加聚焦的深度范围，但是形成图像的光亮也会相应减少（详见章节 9.2）。在户外白天拍摄的照片通常具有较大的景深，因为光量充足，使用一个较小的光圈也能保证足够的进光量，在最理想情况下，光圈可以足够小，即一个针孔相

机。而在一个光线昏暗的房间里，景深则会明显缩小。因此，控制景深效果的一种方法是将其与色调映射联系起来，使得失焦（out-of-focus）物体随着光照水平的降低而变得更加模糊。另一种方法则是允许手动进行艺术控制，根据需要改变焦点和增加景深，从而达到理想的艺术效果，如图 12.10 所示。

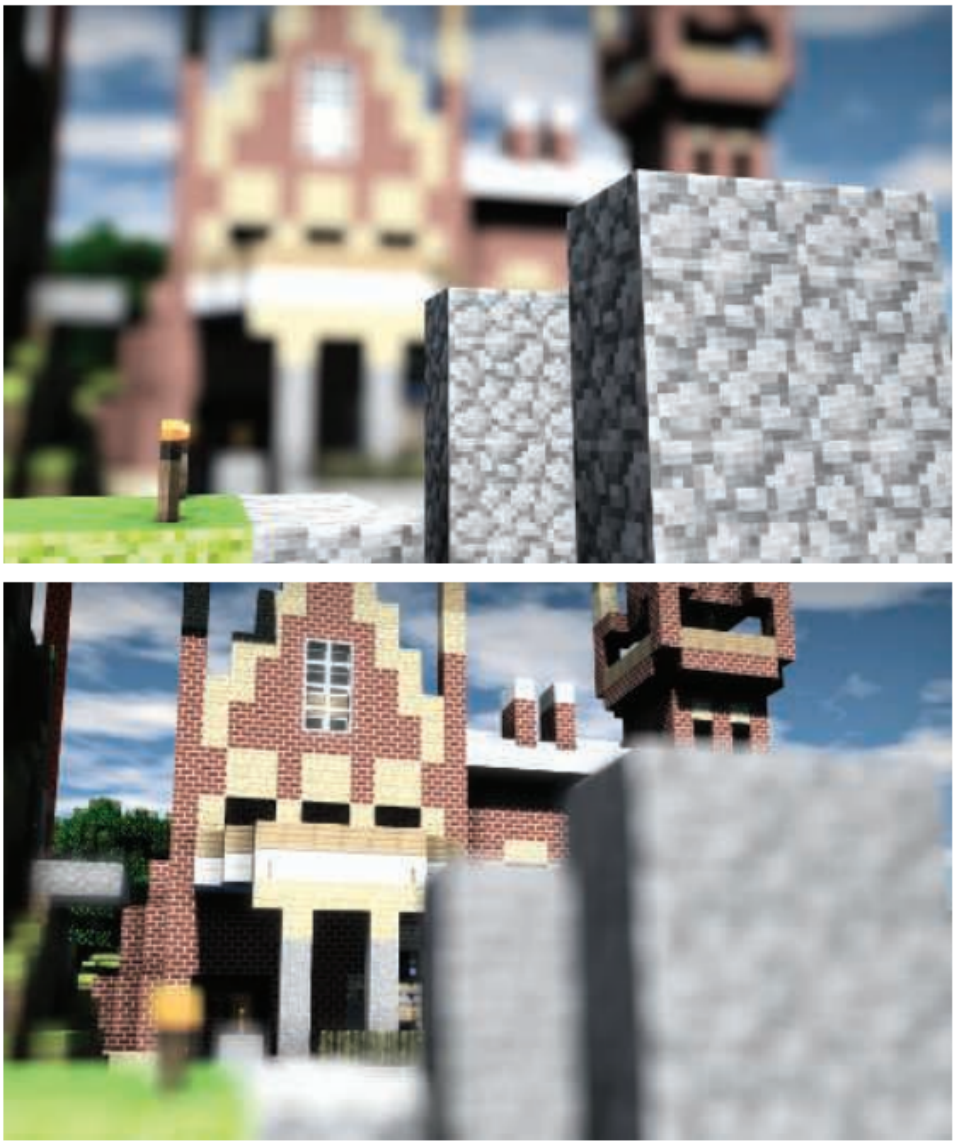


图 12.10：景深效果取决于相机的焦距。[209, 1178]

可以使用一个累积缓冲区（accumulation buffer）来模拟景深效果[637]，详见图 12.11。通过改变镜头上的观察位置并保持焦点固定，物体将会相对于它们与焦点的距离从而被渲染得更加模糊。然而，与其他累积效果一样，这种方法的成本很高，需要对每张图像进行多次渲染。也就是说，它确实可以收敛到正确的 ground-truth 图像，这个图像可以用于测试其他景深算法的质量。通过改变观察光线在光圈上的位置（初始发射位置），光线追踪也可以收敛到物理正确的结果。为了提高渲染效率，有许多方法会在失焦的物体上使用较低层次的 LOD。

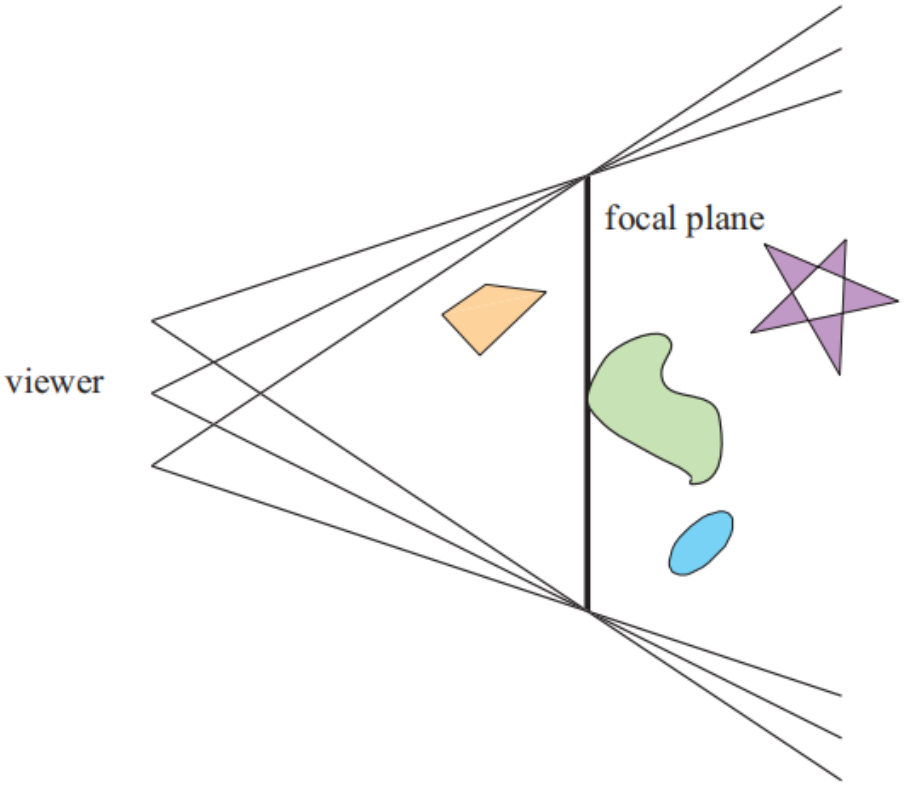


图 12.11：通过累积缓冲区形成的景深效果。观看者所在的位置稍微移动了一点点，同时保持观察方向指向焦点。将每个渲染出的图像相加在一起，然后再显示所有图像的平均值，即可获得景深效果。

虽然这种方法对于交互式应用而言不切实际，但是这种在镜头上移动观察位置，再将结果累积平均的技术，提供了一种合理的方式来考虑每个像素中应当记录什么信息。场景中的表面按照距离可以分为三个区域：聚焦在焦点附近的区域（focus field，焦点场或者中场）；远处的区域（远场）；较近的区域（近场）。对于位于焦距处的表面，每个像素都是清晰聚焦的，因为所有累积的图像都具有大致相同的渲染结果。焦点场是一个深度范围，在这个深度范围内的物体只会发生轻微的失焦，例如少于半个像素[209, 1178]。这个深度范围就是摄影师们所说的景深。而在交互式计算机图形学中，我们会默认使用一个具有完美焦距的针孔相机，因此景深指的是对远场内容和近场内容进行模糊的效果。最终平均图像中的每个像素，都是在不同观察位置下所看到的表面位置的混合结果，由于这些表面位置可能会有很大的差异，因此会使得失焦区域（out-of-focus）变得模糊。

这个问题有一个有限的解决方案，即创建不同的图层来渲染不同区域中的内容。即对焦点物体渲染一副图像，对距离大于焦距的物体（远场）渲染一副图像，对距离小于焦距的物体（近场）渲染一副图像。这个操作可以通过改变相机的远近裁剪平面来实现。将远场图像和近场图像进行模糊处理，然后再将这三幅图像按照前后顺序组合在

一起[1294]。这种方法被称为 2.5 维方法，之所以这么叫，是因为这些二维图像被赋予了深度并据此组合在一起，这种方法在某些情况下可以提供合理的结果。但是当物体跨越多个图像时，这种方法就会失效，因为物体会突然从模糊变为聚焦。此外，所有过滤后的物体都具有均匀一致的模糊程度，这个模糊程度不会随着距离而发生任何变化，这样是不合理的[343]。

观察这个过程的另一种方法是：考虑景深如何对表面上的单个位置产生影响。现在让我们想象表面上的一个小点，当表面聚焦时，我们可以通过单个像素看到这个小点；如果表面失焦，小点则将出现在附近的像素中，这取决于不同的观察视图。在极限情况下，这个小点将在像素网格上定义一个实心圆。这个实心圆被称为弥散圆（circle of confusion）。

在摄影中，焦点场之外的审美质量被称为散景（bokeh），这是一个来自日语的单词，其意思是“模糊（blur）”。(bokeh 读作“bow-ke”，其中“bow”是“bow and arrow”中的“bow”，“ke”是“kettle”中的“ke”）。通过光圈的光线通常是均匀分布的，而不是符合某种高斯分布[1681]。模糊区域的形状与光圈叶片的数量、形状以及尺寸有关。一些廉价的相机会产生五边形的模糊，而不是完美的圆形；而目前大多数新相机都有 7 个叶片，高端机型有 9 个或者更多叶片。还有一些更好的相机会使用圆形叶片，从而使得散景效果呈圆形[1915]。在夜间拍摄时会使用更大尺寸的光圈，从而可以产生一个更圆的图案。与镜头光晕（lens flare）和泛光（bloom）被放大的效果相类似，我们有时会刻意渲染一个六边形的弥散圆，从而暗示我们正在用一个物理相机进行拍摄。正如 Barre-Brisebois 所解释的那样[107]，在可分离的两 pass 后处理模糊中，六边形是一种特别容易产生的形状，因此被广泛应用在许多游戏中。

计算景深效果的一种方法是，获取每个像素在表面上的位置，并将其着色值散射（scatter）到这个弥散圆或者弥散多边形内部的相邻像素中，如图 12.12 左侧所示。但是这个散射的概念不能很好地映射到像素着色器的功能上。像素着色器可以进行高效的并行操作，其中一个原因就是因为它们不会将结果传播给相邻像素。一种解决方案是为每个远近场像素都渲染一个 sprite（详见章节 13.5）[1228, 1677, 1915]。每个 sprite 都会被渲染到一个单独的图层中，sprite 的大小由弥散圆的半径所决定。每一个图层都存储了所有重叠 sprite 的平均混合和，然后这些图层会被一个接一个地组合起来。这种方法有时也会被称为前向映射（forward mapping）技术[343]。这种方法有一些致命的缺点，首先是性能问题，即使使用图像降采样技术，这种方法也可能会很慢，更加糟糕的是，这种方法的时间开销是不稳定的，尤其是当焦点场很浅时（焦距很小）[1517, 1681]。性能的可变性意味着我们很难去管理每帧用于渲染的预

算，即分配给执行所有渲染操作的时间。这种不可预测性会导致帧丢失，以及用户体验上的不均衡，不平滑。

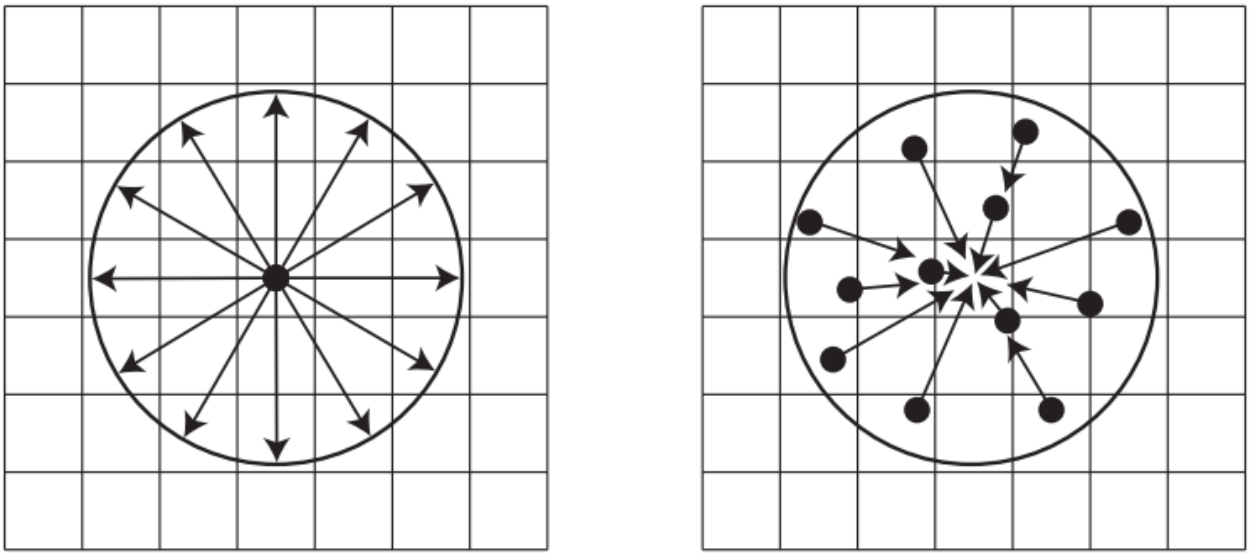


图 12.12：散射（scatter）操作是获取像素的着色值，并将其扩散到邻近区域的像素上，例如通过渲染一个圆形的 sprite。在聚集操作中，相邻区域的值被采样并用于影响某一个像素。GPU 的像素着色器经过优化，可以通过纹理采样来执行聚集操作。

另一种思考弥散圆的方法是：假设像素周围的局部邻域都具有相同的深度。有了这个想法之后，就可以执行一个聚集（gather）操作了，如图 12.12 右侧所示。对于聚集前置渲染 pass 结果这个操作，像素着色器是经过专门优化的。因此，我们可以根据每个像素的深度，来对表面进行不同程度的模糊，从而实现景深效果。像素的深度信息定义了一个弥散圆（circle of confusion）[1672]，即应当被采样聚集的区域有多宽。这种聚集方法被称为反向映射方法（backward mapping，reverse mapping）。

大多数实用的景深算法都是从一个视角的初始图像开始，这意味着从一开始我们就缺失了一些信息。这个单一视角所能看到的场景信息是不完整的，场景的其他视图可能还会看到一些额外的表面。正如 Pesce 所指出的那样，我们应当尽可能地利用我们拥有的可见样本[1390]。

这类聚集技术在多年来不断发展，每一次都在之前工作的基础上有所改进。这里我们将主要介绍 Bukowski 等人[209, 1178]所提出的方法及其遇到的问题，以及这些问题相应的解决方案。他们的方案基于深度信息，为每个像素生成一个代表弥散圆半径的带符号值。这个半径可以从相机设置和特性中获得，但是艺术家们通常更喜欢手动控制效果，因为近场、焦点场和远场的范围都可以手动调整。半径的符号指定了像素是位于近场还是远场，其中 $-0.5 < r < 0.5$ 处于焦点场，即半个像素的模糊会被认为处于焦点场中。

这个包含弥散圆半径的缓冲区会用于将图像分成两个图像，即近场图像和其他图像，每个图像都会在两个 pass 中，使用可分离的滤波器进行下采样和模糊操作。这里的图像分离是为了解决一个关键问题，即近场中的物体应当具有模糊的边缘。如果我们根据弥散圆的半径对每个像素进行模糊，并输出到单个图像中，那么最终结果中的前景物体可能会是模糊的，但仍然会具有清晰的边缘。例如：从前景物体穿过轮廓边缘到达聚焦物体时，样本半径将会降为零，因为聚焦物体不需要进行模糊处理。这将会导致前景物体对其周围像素的影响突然下降，从而产生一个尖锐的边缘，如图 12.13 所示。

图 12.13：近场模糊。左边是没有景深效果的原始图像。在中间，近场像素会被模糊，但是在靠近焦点场的区域会有一个清晰的物体边缘。右图展示了使用单独的近场图像，并将其叠加在较远内容上的效果，近场像素不仅会被模糊，同时也不会出现清晰的物体边缘。[209, 1178]

我们想要的效果是，让位于近场中的物体被平滑地模糊，并产生超出其边界的模糊效果。这是通过在单独的图像中写入并模糊近场像素来实现的。此外，这个近场图像中的每个像素都会被赋予一个 alpha 值，来代表它的混合因子，这个混合因子同样也会被模糊处理。在创建两个分离图像的时候，还可以使用联合双边滤波和其他的一些测试；详情请参阅论文[209, 1178]，以及其中提供的详细代码。这些测试有这样的几个功能，例如：对于远场模糊，对于那些比采样像素远得多的邻近物体，会选择将其丢弃。

在根据弥散圆半径进行分离和模糊处理之后，我们会将这两个分离图像再次合成。我们会使用弥散圆的半径，来对原始聚焦图像和远场图像之间进行线性插值。弥散圆的半径越大，所使用的模糊远场结果就越多。然后再使用近场图像中的 alpha 覆盖率

值，来将近场图像与刚才的插值结果相融合。通过这种方式，可以让近场的模糊内容适当地扩散到后面的场景上，如图 12.10 和图 12.14 所示。

图 12.14：《巫师 3》中的景深效果。近场和远场模糊令人信服地与焦点场混合在一起。

这个算法有几个简化和调整，可以使其看起来更加合理。还可以使用一些其他的方法来更好地处理粒子，并且透明度也可能会导致一些问题，因为这些现象都涉及了每个像素多个 z 深度的问题。尽管如此，由于这种方法只需要输入颜色缓冲区和深度缓冲区，并且只使用了三个后处理 pass，因此这个方法简单且相对健壮。基于弥散圆的采样，以及将远场、近场分离为单独的图像（或者图像集合）的思想，是许多景深模拟算法的共同主题。下面我们将讨论在电子游戏中所使用的一些新方法（就像前面提到的方法一样），这些方法需要高效、健壮，同时还具有可预测的开销 [832, 1390]。

这里我们所介绍的第一种方法使用了运动模糊，有关运动模糊的概念和方法，我们将在下一小节中重新进行讨论。回到最初弥散圆的概念中，想象一下，将图像中的每个像素都转换为对应的弥散圆，其强度与圆的面积成反比。按顺序依次来渲染这组圆会给带来我们最好的结果，但是这样就把我们带回到了之前提到的散射（scatter）概念，因此这样做通常是不切实际的。但是这种思维模式在这里会很有价值，给定一个像素，我们想要确定所有重叠在该位置上的弥散圆，并将它们按照顺序混合在一起，如图 12.15 所示。使用场景定义的最大弥散圆半径，对于每个像素，我们可以检查位于该半径范围内的每个邻居像素，并查找其弥散圆是否包含当前给定像素的位置。然后将所有影响该像素的重叠相邻样本进行排序和混合。

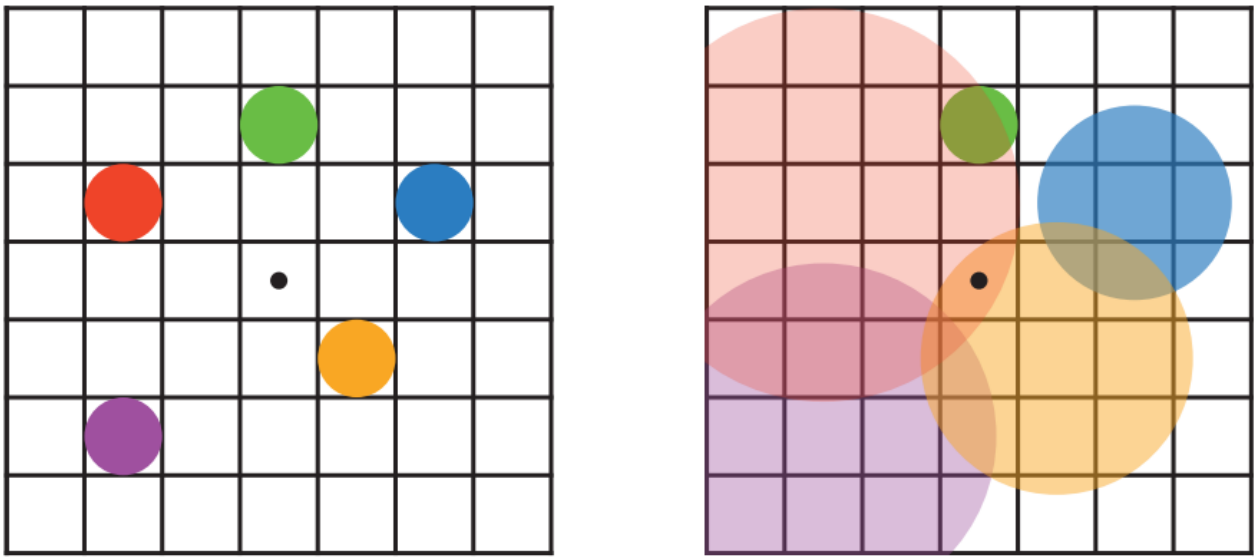


图 12.15：重叠的弥散圆。左侧是一个包含五个点的场景，这些点都是聚焦的。假设其中的红色点位于近场中，它最接近观察者，其次是橙色点；绿色点位于焦点场内；蓝色点和紫色点位于远场中，紫色点距离最远。右图展示了由于应用景深而产生的弥散圆，较大的圆圈对于每个像素的影响较小。绿色点是不变的，因为它是聚焦的。最中心的像素只有红色和橙色的圆圈相重叠，因此它们会混合在一起，红色覆盖橙色，最终会形成该像素的颜色。

上述这种方法是理想的，但是在 GPU 上，将找到的片元进行排序，这个操作的开销太大了。相反，我们使用了一种被称为“聚集时散射（scatter as you gather）”的方法，我们通过查找哪些邻居像素将会散射到当前像素位置来进行聚集操作。选择具有最小 z 深度（最近距离）的重叠邻居来表示较近的图像。任何其他在 z 深度上与该深度接近的重叠邻居，都会将其 alpha 混合的贡献添加进来，然后再取平均值，并将颜色和 alpha 值存储在“前景”层中。而这种类型的混合则不需要进行排序操作。而所有其他剩余的重叠邻居，同样会进行类似的求和平均，并将结果放在一个单独的“背景”层中。这里的前景层和背景层并不对应于之前我们所提到的近场和远场，它们只是在每个像素区域中碰巧所找到的任何东西。然后前景图像会被合成到背景图像上，产生近场模糊的效果。虽然这种方法听起来很复杂，但是通过应用各种采样和过滤技术，可以使其十分高效。你可以参考 Jimenez [832]、Sousa [1681]、Sterna [1698] 和 Courreges [293, 294]的演示来了解一些不同形式的实现，并查看图 12.16 中所展示的例子。

图 12.16：近景深和远景深。在前景中，明亮的反射杆上存在五边形的散景。[1408]

在一些较早的电子游戏中使用的是另一种方法，该方法基于计算热扩散（heat diffusion）的想法。图像被视为向外扩散的热分布，而每个弥散圆则代表了该像素的热传导系数（thermal conductivity）；而焦点区域则是一个完美的绝缘体，不会发生热扩散。Kass 等人[864]描述了如何将一维的热扩散系统视为一个三对角矩阵（tridiagonal matrix），每个样本都可以在常数时间内进行求解。计算着色器可以很好地存储和求解这种类型的矩阵，因此相关从业者开发了几种实现，其中图像会沿着每个轴分解为这样的一维系统[612, 615, 1102, 1476]。弥散圆的可见性问题仍然存在，通常会基于深度信息，生成并合成单独的图层来进行解决。这种技术无法很好地处理弥散圆中的不连续性，因此如今已经很少使用了。

一种特殊的景深效果是由明亮的光源或者画面中的反射所引起的。光源或者镜面反射的弥散圆，即使这种效果会在区域中变得暗淡，但是可能还是要比其画面中的相邻物体亮得多。虽然将每个模糊像素都渲染为一个 sprite 是十分昂贵的，但是这些明亮的光源往往具有更高的对比度，因此可以更加清楚地显示出光圈的形状。而其余像素之间的差异则比较小，因此形状不那么重要。有时，“散景（bokeh）”一词也被（这是错误的）用来指代这些明亮的区域。检测高对比度区域，并只将这些较亮像素渲染为 sprite，同时对其余像素使用聚集技术，从而可以获得一个具有定义的散景，同时也更加高效[1229, 1400, 1517]，如图 12.16 所示。计算着色器同样也可以发挥作用，它可以为聚集景深、高效散景溅射（splatting）创建高质量的 SAT 表[764]。

我们只介绍了渲染景深和明亮散景效果的一小部分方法，并描述了一些用于提高过程效率的技术。研究人员还探索了随机光栅化（stochastic rasterization）、光场处理等方法。Vaidyanathan 等人[1806]的文章对之前的工作进行了总结，McGuire [1178] 给出了一些实现上的总结。

12.5 运动模糊

为了渲染出令人信服的图像序列，稳定且足够高的帧率是很重要的。平滑且连续的运动当然是更好的，但是过低的帧率会使得运动变得不稳定。电影会以每秒 24 帧的速度进行放映，但是电影院通常是黑暗的，在昏暗的光线下，眼睛的瞬时响应（temporal response）对昏暗光线的闪烁不太敏感。此外，电影放映机以 24 FPS 的速度改变图像，但是在显示下一幅图像之前，会将当前图像重新显示 2–4 次，来减少人眼感知到的闪烁。当然，也许最重要的是，电影的每一帧画面，通常都是运动模糊过的图像；而在默认情况下，交互式图形程序所生成的图像没有运动模糊的。运动模糊（motion blur），也可以叫做动态模糊。

在电影中，运动模糊来源于画面中物体在屏幕上的移动，或者来自于摄像机本身的运动。当相机快门打开 1/40 秒–1/60 秒时，就会出现这种效果，而电影中拍摄一帧画面的时间是 1/24 秒。我们习惯于在电影中看到这种模糊效果，并且会认为这是很正常的，因此我们也希望在电子游戏中能看到它们。当快门的打开时间为 1/500 秒或者更短时，可以产生一种剧烈运动（hyperkinetic）的效果，这种效果最早出现在《角斗士》和《拯救大兵瑞恩》等电影中。

对于快速移动的物体，在没有运动模糊的情况下会显得很不稳定，这些物体在帧与帧之间“跳过”了许多像素。这也可以被认为是一种走样，类似于锯齿，但其本质上是时间上的锯齿而不是空间上的。运动模糊可以被认为是一种时域上的抗锯齿。正如提高显示分辨率可以减少锯齿但是无法消除它们一样，提高帧率也不能完全消除对运动模糊的需求。电子游戏的一大特点就是相机和物体的快速运动，因此应用运动模糊可以显著提高它们的视觉效果。事实上，30 FPS 加上运动模糊的效果，通常要比 60 FPS 但没有运动模糊的效果好[51, 437, 584]。

运动模糊依赖于相对运动。如果一个物体在屏幕上从左向右移动，那么它在屏幕的水平方向会被模糊。如果相机正在跟随一个移动的物体，那么物体不会被模糊，而发生相对运动的背景则会被模糊，如图 12.17 所示。这就是现实世界的相机的工作原理，一个优秀的导演知道如何拍摄一个镜头，并使得所感兴趣的区域是聚焦且不模糊的。

图 12.17：在左边，相机是固定的，汽车是模糊的。在右边，相机跟随汽车，背景是模糊的。

与景深效果类似，累积一系列图像提供了一种创建运动模糊的方法[637]。当相机快门打开的时候，一帧画面存在一个持续时间。场景会在这段时间内的不同时间进行渲染，每次渲染前都会对相机和物体进行重新定位。生成的若干张图像会被混合在一起，最终得到一个图像，其中相对于相机视野发生运动的物体是模糊。但是对于实时渲染而言，这样的过程通常是适得其反的，因为它会大大降低帧率。此外，如果物体发生快速运动，并且单个独立图像变得可感知时，就会看到瑕疵，图 12.7 中也展示了这个问题。随机光栅化可以避免多幅图像混合时出现的鬼影现象，但是相对应的，它会产生噪声现象[621, 832]。

如果我们想要的是运动模糊的画面效果，而不是完全真实的物理模拟，那么这个累积的概念可以被巧妙地运用。想象一些，我们现在已经生成了运动模型的 8 帧画面，并将其求和到一个高精度缓冲区中，然后对其求平均值并显示在屏幕上。在第 9 帧中，模型会被再次渲染并累加，但是同时，我们会再次执行第 1 帧的渲染，并从求和结果中减去第 1 帧渲染的结果。现在缓冲区中还是有 8 帧模糊的运动模型，即第 2 帧到第 9 帧。而在下一帧中，我们将会减去第 2 帧，然后加上第 10 帧，同样得到第 3 到第 10 帧，共计 8 帧的累积总和。这样就产生了高度模糊的艺术效果，但是代价是每一帧会渲染两遍场景[1192]。

对于实时图形程序而言，需要一些比上述多次渲染方法更快的技术。景深和运动模糊都可以通过对一组视野进行平均来渲染，这表明了两种现象之间的相似性。为了高效渲染这些效果，这两种效果都需要将它们的样本分散到邻近的像素上，但是通常我们都会执行聚集操作。它们还需要处理不同模糊程度的多个层级，并在给定起始帧内容的情况下重建被遮挡的区域。

运动模糊有几种不同的来源，每种都有各自适用的方法。这些来源可以分为相机方向变化、相机位置变化、物体位置变化、物体方向变化，这里大致按照复杂性递增的顺序进行列举的。如果相机保持位置不变，那么整个世界可以被认为是一个围绕着观察者的天空盒（详见章节 13.3）。此时改变相机方向会使得图像整体上出现方向性的模糊。给定此时相机旋转的方向和速度，我们沿着这个方向对每个像素进行采样，这个速度决定了滤波器的宽度。这种定向模糊被称为线积分卷积（line integral convolution，LIC）[219, 703]，它也被用于流体流动的可视化。Mitchell [1221]讨论了在给定运动方向下的运动模糊的立方体环境贴图（motion-blurring cubic environment map）。如果相机沿着其观察方向轴进行旋转，那么使用一个圆形模糊，每个像素的方向和速度会相对于旋转中心而发生变化[1821]。

如果相机的位置发生变化，那么视差（parallax）就会起作用，例如：远处物体的移动速度相对较慢，模糊也相对较少。当相机沿着观察方向向前移动的时候，这里的视差可以被忽略。仅仅使用一个径向模糊就可能足够了，这种方法能够实现夸张的戏剧效果，图 12.18 展示了这样一个例子。

图 12.18：放射状的径向模糊可以增强运动感。

为了增加真实感，例如在一个竞速游戏中，我们需要适当计算每个物体运动的模糊效果。如果在向前看的同时进行侧向移动，这在计算机图形学中被称为平移（pan），深度缓冲区会告诉我们每个物体应该被模糊多少。

> 在电影摄影中，平移（pan）指的是在不改变位置的情况下左右旋转相机。其他的一些术语是，横向移动为“truck”，垂直移动是“pedestal”。

物体越近，模糊程度就越大。如果相机向前移动的话，运动量就更加复杂了。Rosado [1509]描述了如何使用前一帧的相机观察矩阵，来实时计算画面的速度。这里的想法是将像素的屏幕位置和深度转换回世界空间中，然后再使用前一帧的相机观察矩阵，来将这个世界空间坐标转换为屏幕上的位置。前后屏幕空间位置之间的差异

就是速度向量，它可以用于对该像素进行模糊。复合物体可以渲染为原始屏幕大小的四分之一，这样既节省了像素处理的开销，又过滤掉了采样的噪声[1428]。

如果物体之间相互独立移动，那么情况会变得更加复杂。一种直接但有局限性的方法是对模糊本身进行建模和渲染。这就是使用线段来代表移动粒子的基本原理。这个概念可以扩展到其他物体上，想象一把剑划过天空，在刀片的前后，沿其边缘添加了两个多边形。这些可以在运行时动态建模或者动态生成。这些多边形的每个顶点都使用了 alpha 不透明度，即多边形与剑相交的地方是完全不透明的，而在多边形的外边缘则是完全透明的。这个想法的核心思路是，模型在运动方向上具有透明度，这模拟了剑只在快门打开的部分时间里（假想的）才会覆盖这些像素的效果。

这种方法可以用于一些简单的模型，例如摆动的剑刃，但是诸如纹理、高光以及其他特征也需要进行模糊。每个移动的表面可以被视为单个的样本，我们想要对这些样本进行散射操作（早期的运动模糊方法就是这样做的），可以通过在运动方向上对几何形状进行扩展从而实现这个目的[584, 1681]。这种几何操作是十分昂贵的，因此开发了“边聚集边散射（scatter-as-you-gather）”的方法。对于景深效果，我们会将每个样本扩展到其弥散圆的半径范围内。对于运动样本，我们将会沿着一帧中移动的路径对每个样本进行拉伸，类似于 LIC。快速移动的样本会覆盖更多的区域，因此它在单个位置的影响就比较小。理论上而言，我们可以获取场景中的所有样本，并按照顺序将其绘制为半透明的线段，图 12.19 展示了可视化的结果。随着采样次数的增加，模糊效果的前后边缘都会出现一个平滑透明的渐变，就像是我们在剑的例子中所做的那样。

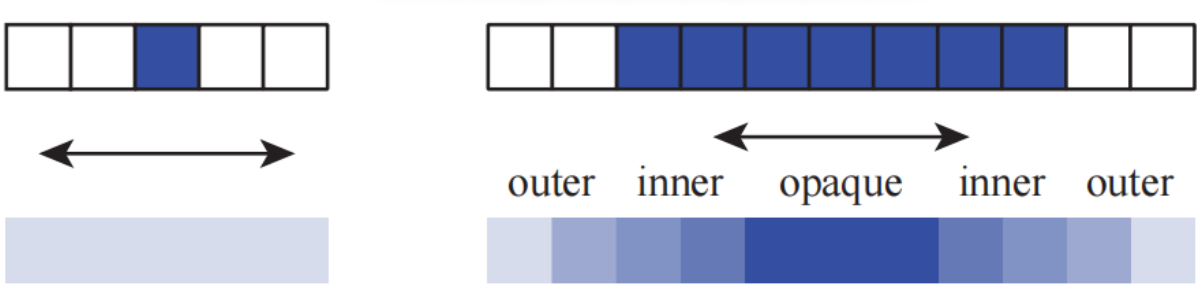


图 12.19：在左边，单个样本水平移动给出了一个透明的结果。在右边，由于覆盖外部区域的样本数量比较少，7 个样本产生了一个锥形的效果。最中间的区域是不透明的，因为在整个一帧画面中，它总是会被一些样本所覆盖。[832]

为了应用这个想法，我们需要知道每个像素表面的速度。一个被广泛采用的工具是速度缓冲区（velocity buffer）[584]。想要创建这个速度缓冲区，需要在模型的每个顶点上插值出屏幕空间速度。可以通过在模型上应用两个模型矩阵来计算出这个速度，其中一个矩阵用于前一帧，另一个矩阵用于当前帧。顶点着色器会计算前后位置的差

异，并将这个差异向量转换为屏幕空间下的相对坐标，图 12.20 展示了速度缓冲区的可视化结果。Wronski [1912]讨论了如何推导速度缓冲区，以及如何将运动模糊与时域抗锯齿相结合。Courreges [294]简要说明了《毁灭战士（2016）》是如何实现这种组合的，并对结果进行了比较。

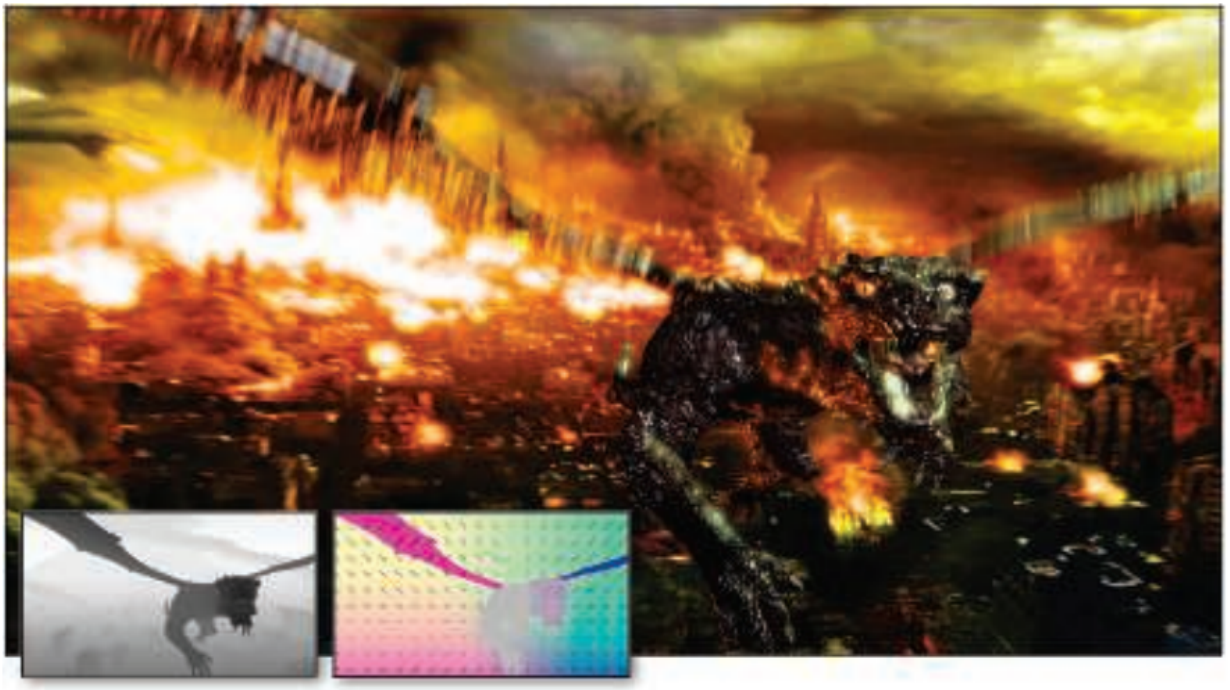


图 12.20：运动模糊源自于物体和相机的相对运动。右下角展示了深度缓冲区和速度缓冲区的可视化结果。[1173]

一旦获得了速度缓冲区，每个物体在各个像素处的速度也就已知了；此时，未被模糊的图像也被渲染完成了。请注意，我们在处理景深时遇到了一个类似的问题，即计算效果所需的全部信息并不是从一张单一图像中获得的，还有一些当前视图看不见的信息。对于景深而言，理想情况是将多个视图平均在一起，其中一些视图包括了在其他视图中看不到的物体表面。而对于交互式运动模糊而言，我们会从时间序列中取出一帧，并将其用作代表性图像。我们将尽可能地使用这些数据，同时需要意识到，单一视图无法提供运动模糊效果所需的全部数据，这个因素有时候可能会产生一些瑕疵。

给定这一帧和速度缓冲区，我们可以使用一个用于运动模糊的“边聚集边散射（scatter-as-you-gather）”系统，来重建影响每个像素的物体。我们从 McGuire 等人[208, 1173]所描述的方法开始，这个方法被 Sousa [1681]和 Jimenez [832]进一步发展了，（Pettineo [1408]提供了源代码）。在第一个 pass 中，计算屏幕上每个部分的最大速度，例如：将每 8×8 像素称为一个瓦片（tile，章节 23.1），找到这个瓦片内的最大速度。该 pass 的结果是一个包含各个瓦片最大速度（一个包含大小和方向的矢量）的缓冲区。在第二个 pass 中，在瓦片结果缓冲区中，会为每个瓦片检索的 3×3 的区域，以找到区域内的最大速度。这个 pass 确保了一个瓦片中快速

移动的物体，其运动模糊的影响将会被相邻瓦片计算在内。也就是说，初始静态的场景视图将会变成物体模糊的图像。这些模糊效果可以重叠到相邻瓦片中，因此这些瓦片必须对一个足够宽的区域进行检查，才能找到这些快速移动的物体。

最后，在第三个 pass 中计算运动模糊的图像。与景深类似，检查每个像素的邻域是否存在快速移动并且与该像素重叠的样本。二者不同之处在于，运动模糊中的每个样本都有不同的运动方向和运动速度。相关学者和从业人员已经开发了不同的方法，来对这些相关的样本进行过滤和混合。其中一种方法是使用瓦片内最大速度来确定滤波核的方向和宽度。如果这个速度小于半个像素，则不需要进行运动模糊[1173]；否则，将会沿着最大速度的方向进行图像采样。需要注意的是，遮挡在这里十分重要，因为它与景深效果有关。在一个静态物体后面快速移动的模型，其模糊效果不应该溢出到这个静态物体上。如果发现邻近样本的距离与该像素 z 深度足够接近，则认为这个邻近样本是可见的。我们将这些样本混合在一起，从而形成前景的贡献。

在图 12.19 中，运动模糊物体有三个区域。其中不透明区域会被前景物体完全覆盖，因此不需要进行混合处理。在原始图像中（最上面的 7 个蓝色像素），外部模糊区域（outer）在对应的像素位置上有一个背景颜色，前景可以与之混合。然而，内部模糊区域（inner）则不包含背景颜色，因为原始图像只包含前景信息。对于这些像素，通过对采样后不在前景中的相邻像素进行过滤来估计背景颜色，这么做的理由是对背景的任何估计，都比什么都没有要好。图 12.20 给出了一个示例。

有几种采样和过滤方法可以用来对这种方法所生成的外观进行改进。为了避免鬼影，采样位置会随机抖动（jitter）半个像素[1173]。在外部模糊区域中，我们拥有正确的背景，但是在内部模糊区域中，我们只有对背景的估计值，在这里进行一点模糊处理，可以避免由于背景估计所带来的不连续性[832]。像素内物体的移动方向和移动速度，可能会与 3×3 瓦片集合内的主导速度不相同，因此在这种情况下可以使用一个不同的滤波方法[621]。Bukowski 等人[208]提供了其他的实现细节，并讨论了如何在不同平台上对该方法进行扩展。

这种方法非常适合动态模糊，但是其他的一些方法和系统当然也是可行的，这些方法在质量和性能之间进行了权衡。例如：Andreev [51]使用速度缓冲区和运动模糊，在 30 FPS 渲染的帧之间进行插值，从而有效地提供了 60 FPS 的帧率。另一个概念则是将运动模糊和景深这两个效果结合到同一个系统中。其关键思想是将速度向量和弥散圆相结合，从而得到统一的模糊滤波核 [1390, 1391, 1679, 1681]。

还有一些其他的方法被进行了研究，随着 GPU 功能和性能的不断提高，这些研究仍将继续。例如，Munkberg 等人[1247]使用随机和交错采样，从而在低采样率下来渲

染景深和运动模糊效果。而在随后的 pass 中，他们使用一种快速重建技术[682]，来减少采样瑕疵，并恢复运动模糊和景深的平滑属性。

在电子游戏中，玩家的体验通常并不像观看电影一样，电子游戏在他们的直接控制下，视角会以一种不可预测的方式进行变化。在这种情况下，如果纯粹基于一种相机的方式，运动模糊有时会表现得很差。例如：在第一人称射击游戏中，有些用户发现视角旋转所带来的模糊效果会分散注意力，甚至导致运动眩晕症（motion sickness，晕 3D）。在《使命召唤：高级战争》中，有一个选项可以移除由于摄像机旋转所产生的运动模糊效果，从而使得运动模糊只会发生在运动的物体上。其美术团队在游戏过程中移除了旋转模糊，但是在一些过场动画中则打开了它。平移运动模糊仍然被开启，因为它有助于传达人物奔跑时的速度感。此外，一些艺术指导也可以用来修改运动模糊的物体，这是物理的电影摄像机无法做到的。假设一艘飞船进入了用户的视野，但是摄像机没有跟踪它，也就是说，玩家没有转头。通过使用标准的运动模糊，这个飞船将会变得模糊，即使此时玩家的眼睛正在追踪它。如果我们假设玩家将要追踪一个物体，我们可以相应地调整算法，使得玩家的眼睛在跟随这个物体的时候，背景是模糊的，而物体仍然保持清晰。

眼动追踪设备（eye tracking device）和更高的帧率可能有助于改善运动模糊的应用，或者完全消除它。然而，运动模糊效果可以使人产生一种电影般的感觉，因此它可能会继续以这种方式或者其他原因进行使用，例如暗示疾病或者头晕等。运动模糊可能会继续进行使用，如何应用它可以是一门艺术，也可以是一门科学。

补充阅读和资源

一些教科书致力于传统的图像处理，例如 Gonzalez 和 Woods 等人[559]的书籍。我们特别要注意的是 Szeliski 所撰写的《Computer Vision: Algorithms and Applications》[1729]一书，因为它讨论了图像处理和其他许多主题，以及它们是如何与合成渲染（synthetic rendering）相关的。这本书的电子版可以免费下载，详见我们网站 realtimerendering.com 上的链接。Paris 等人[1355]的课程讲义提供了有关双边滤波器的正式介绍，也给出了许多应用它们的例子。

McGuire 等人[208, 1173]和 Guertin 等人[621]的文章清晰地阐述了他们各自在运动模糊方面的工作，他们也提供了具体的实现代码。Navarro 等人[1263]为交互式和批处理应用程序提供了一份有关运动模糊的全面报告。Jimenez [832]详细阐述了滤波和采样的问题，以及散景、运动模糊、泛光和其他电影效果的解决方案。Wronski [1918]讨论了如何重组复杂的后处理管线，从而提高效率。有关一系列模拟光学透镜效果的更多信息，请参阅由 Gotanda [575]组织的 SIGGRAPH 课程中的相关讲座。

Chapter 13 Beyond Polygons 超越多边形

> Lennart Anderson——“Landscape painting is really just a box of air with little marks in it telling you how far back in that air things are.”

> 莱纳特·安德森——“山水画实际上只是一盒空气，上面有一些小标记，来告诉你事物在空气中有多远。”（美国画家；1928–2015）

使用三角形对表面进行建模，这通常是解决场景中物体描述的问题的最直接方法。然而，三角形表示只在一定程度上是好的。用图像表示物体的一个很大优点是，其渲染成本与要渲染的像素数量成正比，而不是与几何模型中的顶点数量成正比。因此，基于图像的渲染，其中一个用途就是作为一种渲染模型的更高效方法。然而，图像采样技术的用途要广泛得多。许多物体，例如云和动物毛皮，很难使用三角形来进行表示。而分层半透明图像（layered semitransparent image）则可以用来显示这种复杂的表面。

在本章节中，我们首先将基于图像的渲染方法与传统的三角形渲染方法进行了比较，并对现有算法进行了综述。然后我们会介绍一些常用的技术，例如：精灵图（sprite），告示牌（billboard），impostor，粒子（particle），点云（point cloud）和体素（voxel），以及更多的实验性方法。

13.1 渲染频谱

渲染的目标是在屏幕上描绘一个物体，而如何实现这个目标则取决于我们的选择。渲染场景没有唯一正确的方法，每一种渲染方法都是一种对于现实的近似值，如果我们的目标是真实感渲染的话。

三角形具有从任何角度都能够合理地表示物体的优点。随着相机的移动，物体的表示方法不一定需要发生改变。然而，为了提高渲染质量，当观察者距离物体较近时，我们可能希望使用一个更加详细的模型；而当观察者距离物体较远时，我们可能希望使用模型的简化形式。这叫做细节层次技术（level of detail techniques，LOD），详见章节 19.9，其主要目的是使得场景的渲染速度更快。

当场景物体逐渐远离观察者时，其他渲染和建模技术可以发挥作用。使用图像代替三角形来表示物体可以提高渲染速度，使用可以快速发送到屏幕上的单个图像来表示物体，通常会具有更低的成本。Lengyel [1029]提出了一种表示渲染连续性的方法，如图 13.1 所示。我们首先将从频谱的左侧开始，然后逐渐回到我们更加熟悉的右侧区域。

图 13.1：渲染频谱。[1029]

13.2 固定视图效果

对于复杂的几何模型和着色模型，以交互速率来重新渲染整个场景可能是十分昂贵的。通过限制观察者的移动能力，可以实现各种形式的加速。最严格的限制情况就是相机根本不移动，在这种情况下，很多渲染可以只进行一次。

例如：想象现在有这样一个场景，其中的静态部分是一个带有栅栏的牧场，一匹动态的马正在穿过它。牧场和栅栏会被渲染一次，然后将颜色和 z-buffer 存储起来。每一帧渲染中，这些缓冲区会被用于初始化颜色缓冲和 z-buffer。然后，剩余所需要渲染的全部内容就剩下了这个动态马本身。如果马位于栅栏后面，那么存储和复制的 z-depth 值将会遮挡马。需要注意的是，在这种情况下，马不会投下阴影，因为整个场景是不变的。我们还可以进一步优化这个过程，例如确定马匹影子的影响范围，然后只需要在存储缓冲区上重新计算静态场景的一小块区域即可。最关键的是，在图像中设置每个像素的颜色的时间或者方式没有任何限制，因此对于固定的视图，可以通过将复杂的几何模型转换为一组简单的缓冲区，这些缓冲区可以在许多帧中重复使用，从而节省了大量的重复渲染时间。

在计算机辅助设计（CAD）应用中，通常所有建模的物体都是静态的，并且在用户执行各种操作的时候，视图也不会发生变化（相机不移动）。一旦用户移动到想要的视图，颜色缓冲区和 z-buffer 就可以被存储起来，以便之后重复使用，然后每帧重新绘制用户界面和突出显示的元素即可。这允许用户进行快速标记、度量或者以其他方式与复杂的静态模型进行交互。通过在缓冲区中存储额外的附加信息，还可以执行一

些其他操作。例如：可以通过存储给定视图的物体 ID、法线和纹理坐标，并将用户的交互转换为对纹理本身的修改来实现一个三维绘制程序。

与静态场景相关的一个概念是 golden thread，也被称为自适应细化（adaptive refinement）或者渐进细化（progressive refinement）。其想法是：当观察者（相机）和场景都是静态的时候，随着时间的推移，计算机可以产生质量越来越好的图像。场景中的物体看起来会越来越真实。这种高质量的渲染结果可以进行突然切换，也可以在一系列帧中进行混合。这种技术在 CAD 和可视化应用中特别有用。在这个过程中，可以进行许多不同类型的细化操作。随着时间的推移，可以在每个像素内的不同位置生成更多的采样样本，并在整个过程中显示平均后的结果，从而提供抗锯齿效果[1234]。这个方法同样适用于景深效果，其中样本会在镜头和像素上随机分层[637]。可以使用高质量的阴影技术来创建更好的图像。我们还可以使用一些更加复杂的技术，例如光线追踪或者路径追踪，然后在新图像中淡入（fade in）。

有一些应用程序将固定视角和静态几何物体的想法更进一步，以便在电影级质量的图像中实现光照的交互式编辑。这种技术被称为“重光照（relighting）”，其想法是用户在场景中选择一个视图，然后使用此时的数据进行离线处理，进而生成一组缓冲区或者其他更加复杂的数据结构，然后使用这些新生成的数据来作为场景表示。例如：Ragan-Kelley 等人[1454]将着色样本与最终像素相分离，这种方法允许他们执行运动模糊、透明效果和抗锯齿。他们还使用了自适应细化，随着时间的推移来不断提高图像质量。Pellacini 等人[1366]扩展了基本的重光照，使其包括了间接全局光照。这些技术与延迟渲染方法中所使用的技术非常相似（将在章节 20.1 中进行详细描述）。二者主要区别在于，这里所介绍的技术用于分摊多帧渲染的昂贵成本，而延迟渲染则使用类似技术来加速一帧内的渲染。

13.3 天空盒

环境贴图（environment map，章节 10.4）代表了局部空间内的入射 radiance。虽然这种贴图通常用于模拟反射效果，但是它们也可以直接用于表示周围的环境，图 13.2 给出了这样的一个例子。任何具体类型的环境贴图表示方法，例如全景图（panorama）或者立方体贴图（cube map），都可以用于实现这个目的。其网格足够大，足以包含场景中的其余物体，这个网格就被称为天空盒（skybox）。

图 13.2：Mission Dolores（一个位于旧金山的教堂）的全景图，底部的三张图片是根据这个全景图生成的。请注意这些视图本身是如何看起来不失真的。

拿起一本纸质书，从书的左边缘或者右边缘往外看；先用右眼看，然后再用左眼看。书的边缘相对于背后物体边缘的偏移叫做视差（parallax）。这种效应对于附近的物体而言十分重要，它有助于我们在移动时感知相对深度。然而，如果一个物体或者一组物体距离观察者足够远，并且彼此又足够接近，那么当观察者改变观察位置时，就几乎感知不到视差效应。例如：如果我们移动 1 米，甚至是 1000 米，远处的山通常看起来并不会有什么明显的不同。当我们移动的时候，这座山可能会被附近的物体所遮挡，但如果移除这些遮挡物体，这座山及其周围环境看起来还是一样的。

天空盒的网格通常会以观察者为中心，并跟随观察者一起移动。天空盒网格不需要特别特别大，因为通过保持天空盒与观察者的相对位置，它看起来形状就不会发生改变。对于图 13.2 这样的场景，观众可能只移动了一小段距离，就会发现其实自己相对于周围的建筑物并没有真正的发生移动。对于更大规模的场景内容而言，例如星空或者远处的风景，用户通常并不会移动得非常远或者非常快，因此缺乏物体大小、形状或者视差的变化，并不会打破这种错觉。

天空盒通常会以立方体贴图的形式渲染在 box 网格上，因为每个面上的纹素密度大致相同。为了让天空盒看起来足够好看，因此立方体贴图的纹理分辨率必须足够大，即

满足每个屏幕像素包含一个立方体贴图纹素[1613]。立方体贴图所需分辨率的公式可以近似为：

$$\text{texture resolution} = \frac{\text{screen resolution}}{\tan(\text{ fov }/2)} \tag{13.1}$$

其中 fov 是相机的视场角。较小的视场角意味着立方体贴图必须具有较高的分辨率，因为此时立方体表面的较小部分会占据相同的屏幕尺寸。这个方程可以这样推导而来，当视场角为 90°（水平和垂直）时，立方体贴图其中一个表面的纹理必须覆盖整个屏幕。

除了 box 之外，还可以使用其他形状来表示周围世界。例如：Gehling [520]描述了一种系统，该系统使用一个扁平的圆顶来代表天空。这种圆顶几何形状被认为最适合用于模拟在头顶移动的云层。这里的云层本身是通过组合和动画各种二维噪声纹理来实现的。

由于我们知道，天空盒位于所有其他物体的后面，因此我们可以进行一些微小但有价值的优化。天空盒不需要写入 z-buffer，因为它永远不会遮挡任何东西。如果先绘制天空盒，我们也不需要从 z-buffer 读取数据；同时天空盒的网格可以是任何大小的，因为这里天空盒网格的深度信息是无关紧要的。然而，按照“不透明物体-天空盒-透明物体”的顺序进行绘制有一定的好处，这样做的好处在于，当我们绘制天空盒的时候，场景中的不透明物体已经覆盖了屏幕上的一些像素，从而减少了渲染天空盒时需要调用的像素着色器数量[1433, 1882]。

13.4 光场渲染

radiance 可以在不同的位置和方向、不同的时间和不同的光照条件下进行捕获。在现实世界中，计算摄影领域探索了从这些数据中提取各种结果的方法[1462]。物体也可以使用纯粹基于图像的方法进行表示。例如“Lumigraph [567]和光场渲染技术（light-field rendering）[1034]，这些技术试图从一组观察点中捕获单个物体。对于给定的一个新视角，这些技术可以在存储下来的视图之间进行插值，从而创建新视角下的视图。这是一个复杂的问题，需要大量数据来存储所需的全部视图。这个概念有点类似于全息摄影（holography），它使用一个二维视图数组来表示物体。这种渲染形式的诱人之处在于，它能够捕获一个真实的物体，并且能够从任何角度来重新显示它。任何物体，无论其表面和光照如何复杂，都可以以一个几乎恒定的速率进行显示。有关这个主题的更多信息，请参阅 Szeliski 的书[1729]。近年来，人们对光场渲染重新产生了研究兴趣，因为它可以让眼睛利用虚拟现实显示器来适当地调整焦点

[976, 1875]。这些技术目前在交互式渲染中的应用十分有限，但是它们在计算机图形学领域开辟了新的领域。

> 译者注：有关新视角合成的内容，请了解 NeRF 的原理及最新进展。

13.5 Sprite 和图层

sprite（也叫做精灵图）是最简单的基于图像的渲染图元之一[519]。sprite 是指在屏幕上移动的图像，例如鼠标光标等。sprite 不一定非要是矩形的，因为 sprite 中的一些像素可以是透明的。对于简单的 sprite 而言，其中存储的每个像素都将被复制到屏幕上的一个像素中。还可以通过显示一系列不同的 sprite 来生成动画。

一种更加通用的 sprite 类型是将图像纹理渲染到始终面向观众的多边形上，通过这种方式，允许 sprite 被调整大小和拉伸。纹理图像的 alpha 通道可以为 sprite 的各个像素提供完全或者部分的透明度，因此同样也在边缘处提供了抗锯齿效果（章节 5.5）。这种类型的 sprite 也可以有深度信息，也就是 sprite 本身在场景中的位置。

图 13.3：动画《Chicken Crossing》的一帧画面，使用 Talisman 模拟器进行渲染。在这个场景中，使用了 80 层 sprite，其中一些被突出显示在左侧。由于鸡翅一部分位于后挡板的前面，一部分位于后挡板的后面，因此二者被放置在了独立的 sprite。

我们可以将场景看作是一系列的图层，这在二维的序列帧（cel）动画中十分常见。例如：在图 13.3 中，后挡板位于鸡的前面，鸡在卡车驾驶室的前面，而驾驶室又在道路和树木的前面。这种分层方法适用于大量的观察点集合。每个 sprite 层都有一个与之相关的深度。通过从后往前的顺序进行渲染，即画家算法（painter' s algorithm），我们可以在不需要 z-buffer 的情况下来构建场景。此时相机向前移动，只会让物体变大，这很容易使用相同的 sprite 或者相关的 mipmap 来进行处理。移动相机实际上还会改变前景和背景的相对覆盖范围，这可以通过修改每个 sprite 层的覆盖范围和位置来进行处理。当观察者水平或者垂直移动时，这些图层可以基于各自的深度进行相应的移动。

可以用一组 sprite 来表示一个物体，其中每个单独的 sprite 都代表不同的视图。如果物体在屏幕上足够小，存储大量视图集合（即使是动画物体）也是一个可行的策略[361]。视角的微小变化也可以通过扭曲 sprite 的形状来进行处理，尽管这种近似最终会崩溃，还是需要生成一个新的 sprite。具有不同表面的物体在一个很小的旋转之后也可能会发生很大的变化，因为会有新的多边形变得可见，而其他的一些多边形则会被遮挡。

这种图层和图像的扭曲过程，是微软在 20 世纪 90 年代后期支持的 Talisman 硬件架构的基础[1672, 1776]。虽然这个特定的系统由于一些原因已经逐渐消失了，但是这种通过一个或者多个基于图像的表示来表现模型的想法，已经被发现是富有成效的。使用具有不同功能的图像，可以很好地映射到 GPU 的优势上，并且基于图像的技术可以与基于三角形的渲染相结合。在下面的几个小节中，我们将讨论 imposter（顶替者）、深度 sprite（depth sprite），以及其他使用图像来代替多边形内容的方法。

13.6 广告牌技术

基于观察方向来修改纹理矩形朝向的技术被称为广告牌技术（billboarding），这个矩形被称为广告牌（billboard）[1192]。随着视图的改变，矩形的朝向也会随之改变。广告牌结合 alpha 纹理和动画，可以表现许多不光滑实体表面的视觉现象。草、烟、火、雾、爆炸、能量盾、蒸汽轨迹和云等，这些只是广告牌技术可以表示的物体中的一小部分[1192, 1871]，如图 13.4 所示。

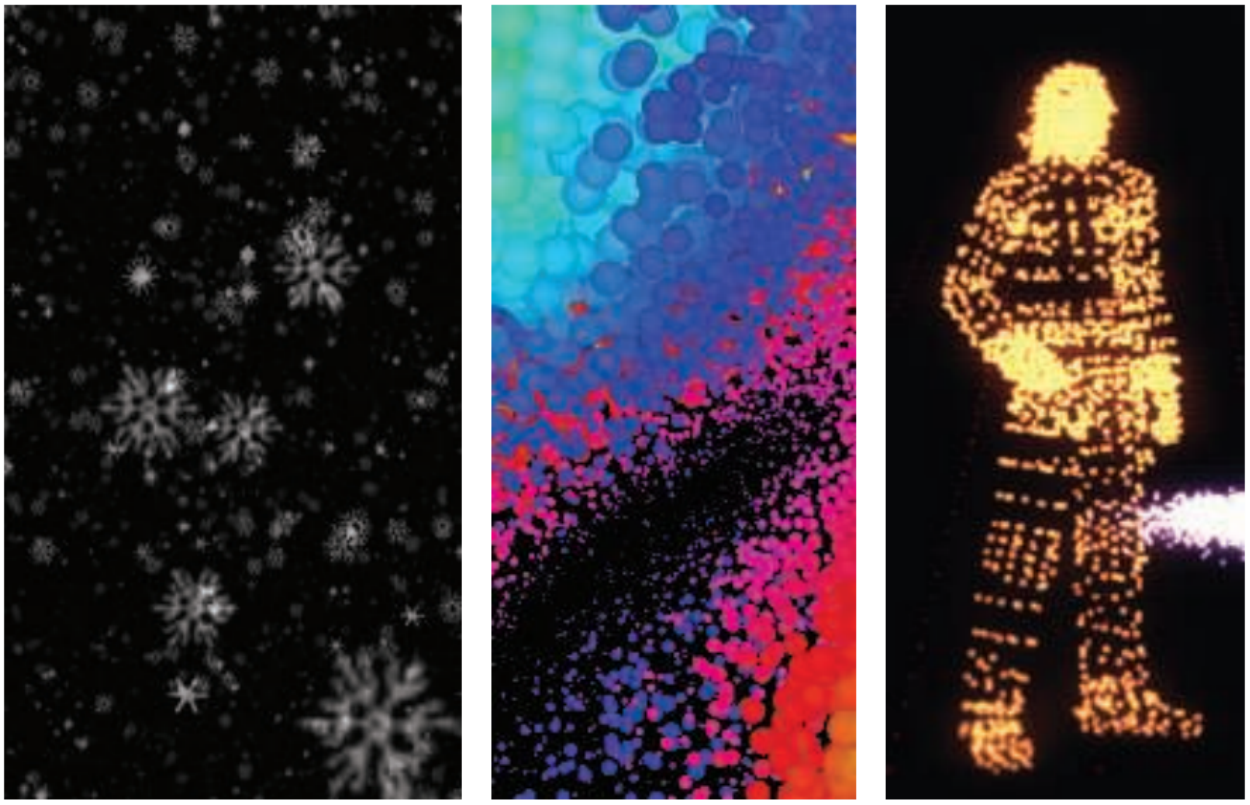


图 13.4：表示雪、地面和人物角色的小广告牌。

本小节将会介绍几种十分流行的广告牌形式。在这些形式中，会使用一个表面法线和一个指向上方的方向（up 向量）来定位四边形。这两个向量可以创建广告牌表面的标准正交基。换句话说，这两个向量描述了将四边形旋转到最终方向所需要的旋转矩阵（章节 4.2.4）。然后使用四边形上的锚点位置（anchor location）（例如四边形中心），来确定其在空间中的具体位置。

通常来说，表面法线 $\mathbf{n}$ 和 up 向量 $\mathbf{u}$ 并不是垂直的。在所有的广告牌技术中，这两个向量中的其中一个会被确定为固定向量，即必须保持在给定的方向上。而使得另一个向量垂直于这个固定向量的过程总是相同的。首先，创建一个 right 向量 $\mathbf{r}$，它指向四边形的右侧边界，这是通过计算 up 向量 $\mathbf{u}$ 和表面法线 $\mathbf{n}$ 的叉乘获得的。将这个向量 $\mathbf{r}$ 归一化，因为它将被用作旋转矩阵标准正交基的其中一个轴。如果向量 $\mathbf{r}$ 的长度为 0，那么说明 $\mathbf{u}$ 和 $\mathbf{n}$ 是平行的，可以使用章节 4.2.4 中所描述的相关技术来进行后续处理[784]。如果 $\mathbf{r}$ 的长度不为零，但几乎为零，那么说明 $\mathbf{u}$ 和 $\mathbf{n}$ 几乎平行，此时会出现精度误差。

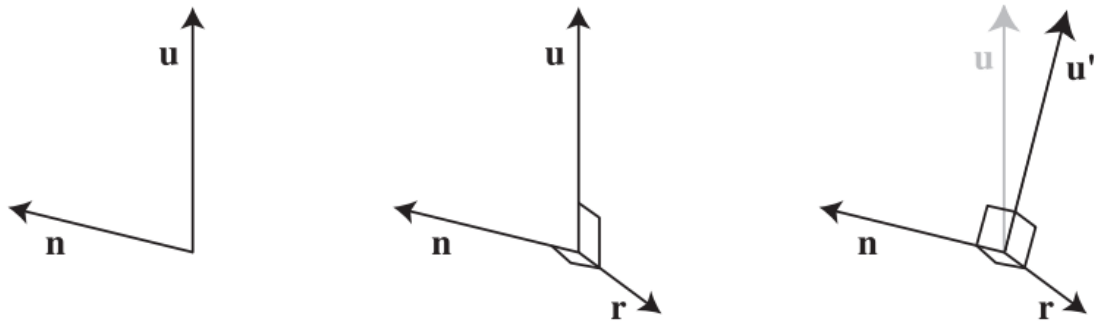


图 13.5：对于一个具有表面法线 $\mathbf{n}$ 和近似 up 向量方向 $\mathbf{u}$ 的广告牌，我们想要创建一个由三个相互垂直的向量组成的集合（标准正交基）来确定广告牌的方向。在中间的图中，right 向量 $\mathbf{r}$ 是通过计算向量 $\mathbf{u}$ 和 $\mathbf{n}$ 的叉乘得到的，所以它垂直于向量 $\mathbf{u}$ 和 $\mathbf{n}$ 。在右图中，固定向量 $\mathbf{n}$ 与 $\mathbf{r}$ 叉乘，得到新的 up 向量 $\mathbf{u}'$ ，这个新向量 $\mathbf{u}'$ 与向量 $\mathbf{n}$ 与 $\mathbf{r}$ 相互垂直。

从（非平行的）向量 $\mathbf{n}$ 和向量 $\mathbf{u}$ 中计算向量 $\mathbf{r}$ 和第三个新向量的过程如图 13.5 所示。如果法线 $\mathbf{n}$ 保持不变，就像大多数广告牌技术所做的那样一样，那么新的 up 向量 $\mathbf{u}'$ 是：

$$\mathbf{u}' = \mathbf{n} \times \mathbf{r} \tag{13.2}$$

相反，如果 up 方向是固定的（适用于轴向对齐的广告牌，例如水平地面上树木），那么此时新的法线 $\mathbf{n}'$ 是：

$$\mathbf{n}' = \mathbf{r} \times \mathbf{u} \tag{13.3}$$

然后将新向量进行归一化，并使用这三个向量构建一个旋转矩阵。例如：对于固定的表面法线 $\mathbf{n}$ 和调整后的 up 向量 $\mathbf{u}'$ ，这个旋转矩阵是：

$$\mathbf{M} = (\mathbf{r}, \mathbf{u}', \mathbf{n}) \tag{13.4}$$

该矩阵会将 xy 平面中的一个四边形，以 $+y$ 方向指向其上边缘，并以其锚点位置为中心，变换到正确的方向。然后再应用一个平移矩阵，来将四边形的锚点移动到目标位置上。

有了这些初步的准备，剩下的主要任务就是决定用什么表面法线和 up 向量来定义广告牌的方向。下面的几个小节中，我们将讨论几种构造这些向量的不同方法。

13.6.1 屏幕对齐（screen-aligned）的广告牌

最简单的广告牌形式就是与屏幕对齐的广告牌（screen-aligned billboard）。这种广告牌形式与二维 sprite 相同，因为图像总是会与屏幕平行，并且具有固定的 up 向量。摄像机将场景渲染到与远近裁剪平面平行的视平面（view plane）上。我们经常会在近裁剪平面的位置上来想象这个假想平面。对于这种类型的广告牌而言，其所需的表面法线是视平面法线的负值，其中视平面的法线 $\mathbf{v}_n$ 指向远离观察点的方向，也就是说，这里广告牌的表面法线指向观察点。up 向量 $\mathbf{u}$ 来自于相机本身，它是视平面上的一个向量，定义了相机的向上方向。这两个向量本身就是垂直的，因此我们只需要创建 right 向量 $\mathbf{r}$，就可以构建出这个广告牌的旋转矩阵。因为广告牌表面法线 $\mathbf{n}$ 和 up 向量 $\mathbf{u}$ 都是相机的常数，因此这个旋转矩阵对于所有这种类型的广告牌都是相同的。

除了粒子效果外，屏幕对齐的广告牌对于注释文本和地图位置标记等信息也十分有用，因为文本总是与屏幕本身对齐的，即“广告牌”这个名字本身的含义。需要注意的是，在使用文本注释的时候，物体通常会在屏幕上保持一个固定的大小。这意味着如果用户前进或者离开广告牌的位置时，广告牌的世界空间大小将会增加。因此，这里广告牌的尺寸是视图相关的，这会使得视锥裁剪等方案变得更加复杂。

13.6.2 面向世界（world oriented）的广告牌

我们希望屏幕对齐的广告牌能够展示玩家的身份或者地点的名称。但是当摄像机发生倾斜的时候，例如在飞行模拟中进入一段曲线轨迹，我们同样希望这些广告牌也能够发生相应地倾斜。如果一个 sprite 代表了一个物理物体，那么它通常基于的是世界空间中的 up 方向，而不是摄像机的 up 方向。圆形的 sprite 不会受到摄像机倾斜的影响，但是其他形状的广告牌则会受到影响。我们可能希望这些广告牌始终保持面向观众，但同时也要沿着它们的观察轴进行旋转，以保持世界空间中的朝向。

对于这样的 sprite，渲染它们的一种方法是使用世界空间中的 up 向量来生成旋转矩阵。在这种情况下，广告牌的表面法线仍然是视平面法线的负值，这是一个恒定的向量，并且与该向量垂直的新 up 向量是从世界空间中的 up 向量衍生出来的，如前所述。与屏幕对齐的广告牌一样，这个矩阵可以用于所有的 sprite，因为这些向量（视平面法线、世界空间 up 向量）在渲染场景时不会发生改变。

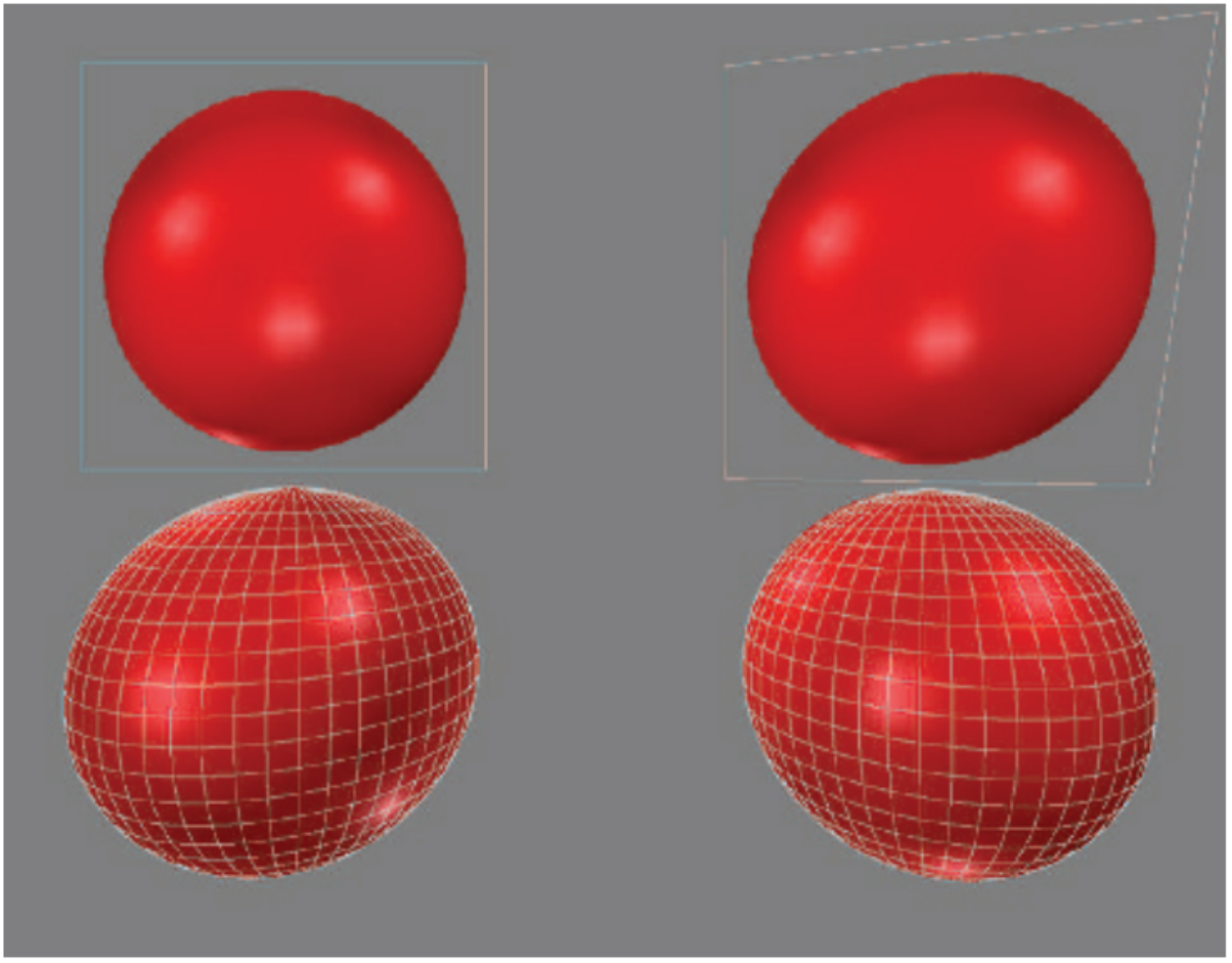


图 13.6：使用一个较宽的视场角来观察四个球体。左上角是一个球体的广告牌纹理，使用了视平面对齐方式。右上角的广告牌是面向视点的。下面一行则展示两个场景中的真实球体。

对所有的 sprite 都使用相同的旋转矩阵是有风险的。由于透视投影的特性，到观察轴有一定距离的物体会被扭曲拉伸。如图 13.6 中底部的两个球体，由于是在平面上进行投影，因此球体的投影变成了椭圆形。这种现象并不是错误的，如果观众的眼睛与屏幕之间的距离和位置比较合适的话，这种现象看起来并不会有什么问题。也就是说，如果虚拟相机的几何视场（geometric field of view）与人眼的显示视场（display field of view）相匹配，那么这些球体看起来就没有被扭曲拉伸。视场中 10%–20% 的轻微不匹配，并不会被观众注意到[1695]。然而，通常的做法是为虚拟相机提供更大的视场角，以便向用户呈现更宽广的世界。此外，只有当观察者在显示器面前以给定的距离居中时，这种视场的匹配才有效。几个世纪以来，艺术家们已经意识到这个问题，并进行了必要的补偿。人们预期是圆形的物体（例如月亮），无论它们在画布上的位置如何，都会被画成圆形[639]。

当视场角很小或者 sprite 很小的时候，这种扭曲效果可以被忽略，并且可以使用一个与视平面对齐的单一方向来作为广告牌的法线。否则，广告牌的法线需要为从广告牌中心指向观察者位置的向量。这就是我们所说的面向视点的广告牌（viewpoint–

oriented），如图 13.7 所示。图 13.6 展示了使用不同对齐方式的广告牌效果。从图中可以看出，无论广告牌出现在屏幕上的什么位置，视平面对齐（屏幕对齐）都可以使广告牌不会出现扭曲。而面向视点的对齐方式会扭曲球体的图像，其扭曲形式与将场景投影到平面上时真实球体的扭曲形式相同。

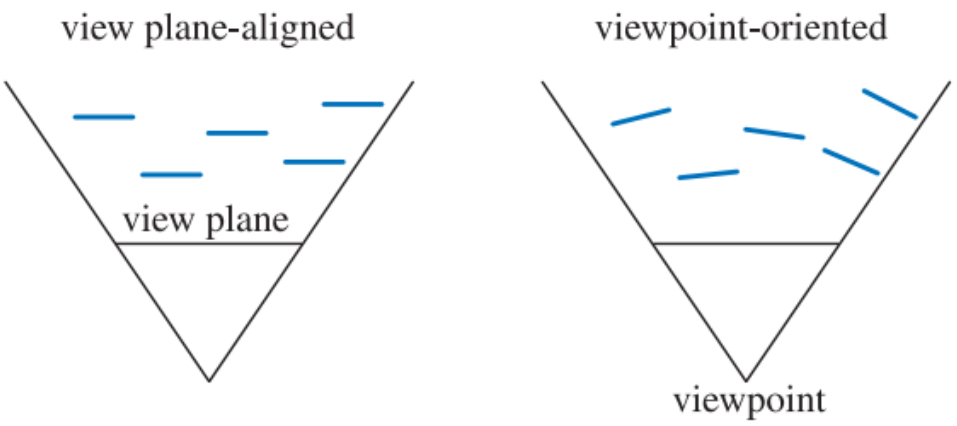


图 13.7：两种不同对齐方式的广告牌的俯视图。根据不同的对齐方法，五块广告牌的朝向也不同。

面向世界的广告牌对于渲染许多不同的现象都十分有用。Guymon [624]和 Nguyen [1273]都讨论了如何生成令人信服的火焰、烟雾和爆炸等效果。一种技术是以随机和混乱的方式，将动画 sprite 聚集和重叠在一起。这样做有助于隐藏动画序列的循环模式，同时也避免了每个火焰或者爆炸效果看起来都一样。

镂空纹理（cutout texture）中的透明纹素对最终图像没有影响，但是它必须由 GPU 进行处理，并在光栅化管线的后期阶段中被丢弃，因为其 alpha 值为零。一组动画的镂空纹理，通常都会具有特别大的透明纹素边缘区域。我们通常会将这些纹理应用于矩形图元。Persson 指出，一个更加紧凑的多边形与调整过的纹理坐标可以更快地渲染这些 sprite，因为需要进行处理的纹素大大减少了[439, 1379, 1382]，如图 13.8 所示。他发现，使用具有 4 个顶点的新多边形就可以带来显著的性能提升，但使用超过 8 个顶点的新多边形的所带来的性能提升会达到收益递减点。例如：虚幻 4 引擎中有一个“粒子切割（particle cutout）”工具，可以用来寻找这样的多边形[512]。

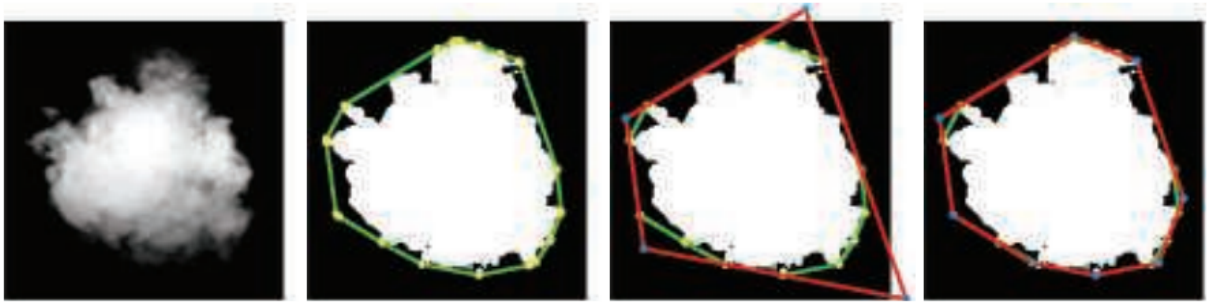


图 13.8：这个云 sprite 中包含了一个很大的透明边缘。使用凸包（绿色），更紧密的四顶点和八顶点多边形（红色），可以包含更少的透明纹素。与最左侧的原始方形的粒子 sprite 相比，这样做的总面积分别减少了 40% 和 48%。[1382]

广告牌的一个常见用途是渲染云层。Dobashi 等人[358]使用广告牌来模拟并渲染云，并通过渲染同心半透明外壳来创建光线束效果（shafts of light）。Harris 和 Lastra [670]也使用 impostor 来模拟云，如图 13.9 所示。

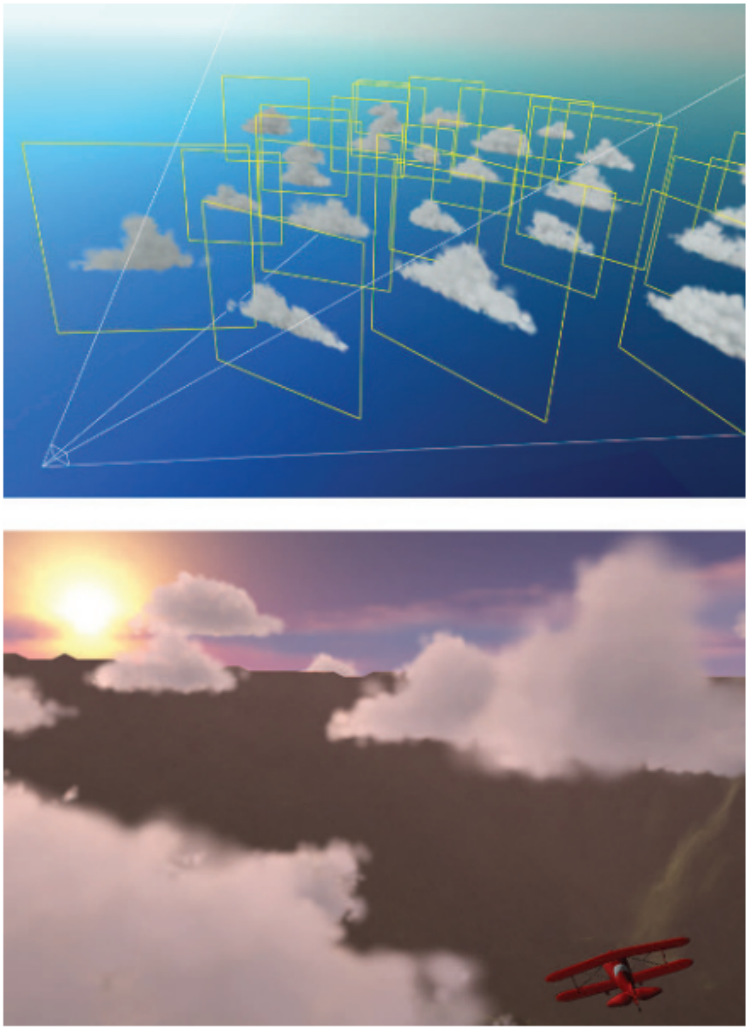


图 13.9：使用一组面向世界的 impostor 来创建云朵。

Wang [1839, 1840]详细介绍了微软飞行模拟器产品中所使用的云建模和渲染技术。每片云都由 5 到 400 块广告牌组成，但是只需要 16 种不同的基本 sprite 纹理即可实现各种各样的云，因为这些基本的 sprite 纹理可以使用非均匀缩放和旋转来进行修改，从而组合形成各种各样类型的云。还会根据距离云中心的距离来修改该点的透明度，从而模拟云的形成和消散。为了节省处理时间，远处的云都被渲染为一组环绕场景的 8 个全景纹理，类似于天空盒。

平面广告牌并不是唯一可能的云层渲染技术。例如：Elinas 和 Stuerzlinger [421]通过渲染一系列嵌套的椭球体来生成云，这些椭球体在轮廓边缘处会变得更加透明。Bahnassi 和 Bahnassi [90]会渲染一个他们称之为“巨型粒子（mega-particle）”的椭球体，然后再使用模糊处理和屏幕空间中的扰动纹理，来给出一个令人信服的类云外观。Pallister [1347]讨论了如何程序化生成云层图像，并在头顶的天空网格上对这些图像进行动画处理。Wenzel [1871]在观察者上方，使用了一系列的平面来模拟远处的云层。在本小节中，我们将专注于广告牌及其他图元的渲染和混合方法。有关云层广告牌的着色方面将在章节 14.4.2 中进行讨论，有关真实体积云的方法也将在章节 14.4.2 中进行讨论。

正如章节 5.5 和章节 6.6 中所描述的，为了正确地进行合成，重叠的半透明广告牌应当按前后顺序来进行渲染。烟雾广告牌在与固体物体相交时会产生瑕疵，如图 13.10 所示。此时这种视错觉会被打破，因为原本是一个体积的物体此时被看作成了一组图层。一个解决方案是，让像素着色器在处理每个广告牌的时候，检查底层物体的 z-depth。广告牌在渲染时会对这个深度进行测试，但是不会使用自己的深度来替换它，也就是说，这个过程不会写入 z-depth。如果某个像素处的底层物体十分接近广告牌的深度，那么此时广告牌的片元将会变得更加透明。通过这种方式，使得广告牌更像是一个体积，这种层状瑕疵也就消失了。当达到最大衰减距离的时候，随着深度进行线性衰减会导致不连续性。S 曲线淡出函数可以避免这个问题，Persson 指出 [1379]，观察者与粒子之间的距离将会影响这个最好的淡出范围。Lorach [1075, 1300]提供更多信息和实现细节。以这种方式修改其透明度的广告牌被称为软粒子（soft particle）。

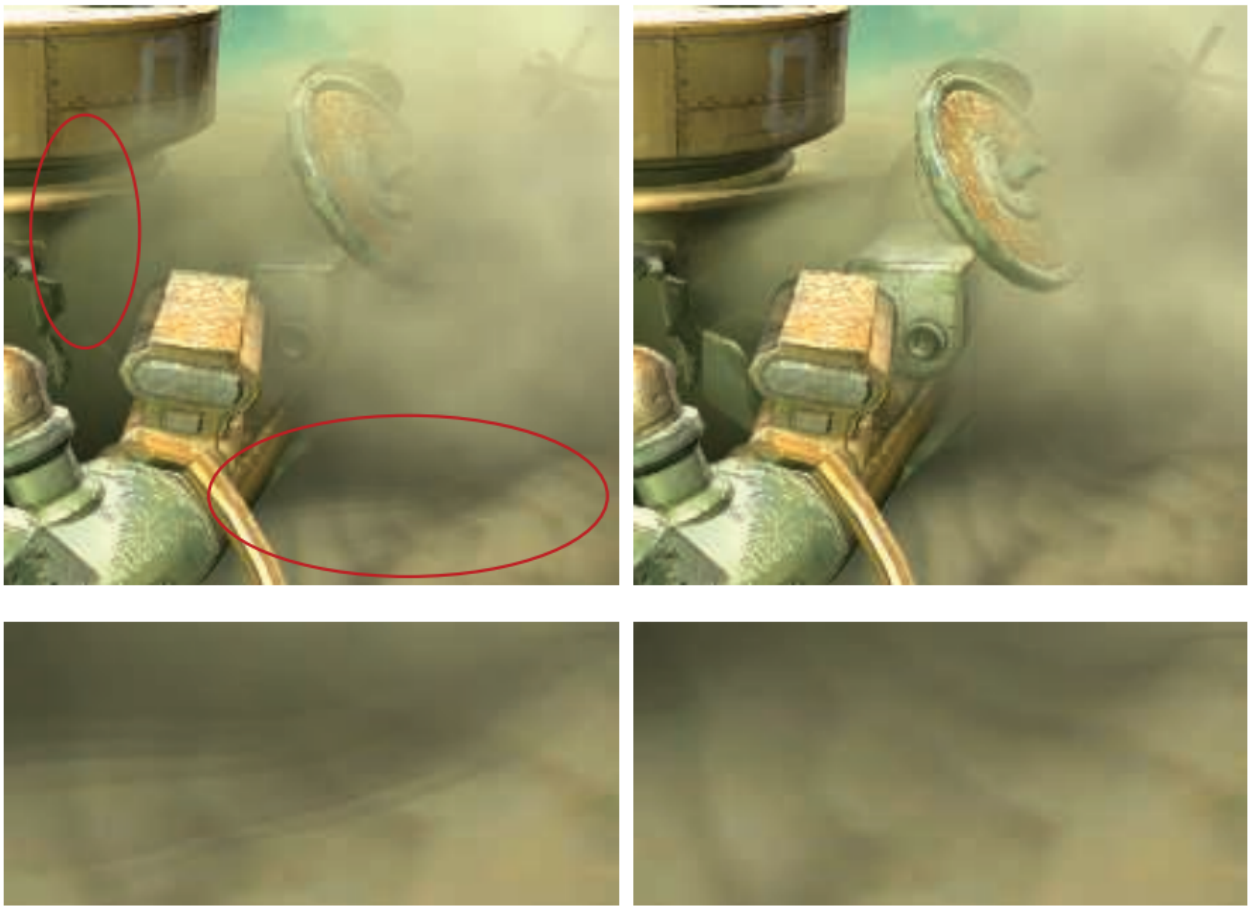


图 13.10：在左边，由于尘埃云广告牌与物体相交，圆圈区域内会显示出清晰的边缘和条带。在右边，广告牌会在物体附近逐渐消失，从而避免了这个问题。在底部，下方圆圈区域的内容被放大以供对比参考。[1300]

使用软粒子的淡出解决了广告牌与固体物体相交的问题，如图 13.10 所示。当爆炸效果在场景中移动或者观察者穿过云层时，也可能会发生其他瑕疵。在前一种情况下，广告牌会在动画过程从物体的后面移动到物体的前面。如果广告牌从完全不可见变为完全可见，这会引起一个明显的“pop”效果。同样地，当观察者穿过广告牌的时候，广告牌会移动到近裁剪平面的前面，此时广告牌会完全消失，从而导致一个可以被观察到的突然变化。一个快速的解决方法是让广告牌在靠近观察者时变得更加透明，随着广告牌越来越近，它们会逐渐淡出，从而避免这种突兀的“pop”效果。

当然还有一些更加写实的解决方案。Umenhoffer 等人[1799, 1800]提出了球形广告牌的概念。这里的广告牌物体被认为定义了空间中的一个球形体积。广告牌本身在渲染时会忽略 z-depth 的读取；广告牌的目的纯粹是让像素着色器在球体可能出现的位置上执行。像素着色器会计算这个球体上的入口位置和出口位置，并根据需要使用固体物体来改变出口深度，并使用近裁剪平面来改变入口深度。通过这种方式，每个广告牌的球体可以通过增加透明度来适当地淡出，我们根据来自摄像机的光线在这个剪切球体内传播的距离，来对球体透明度进行修改。

在《孤岛危机》中使用了一种稍微不同的技术[1227, 1870]，它使用 box 体积来代替球体，从而减少了像素着色器的计算成本。另一个优化是让广告牌代表体积的前面，而不是代表体积的后面。这允许使用 z-buffer 进行深度测试，从而跳过固体物体背后的部分空间。这种优化只有在已知体积完全位于观察者前面时才可行，因为此时广告牌不会被近裁剪平面裁剪。

13.6.3 轴向广告牌

最后一种常见类型的广告牌被称为轴向广告牌（axial billboarding）。在这个方案中，被纹理化的物体通常并不会直接面对观察者。相反，它可以围绕一些固定的世界空间轴进行旋转，并在这个范围内尽可能多地面向观察者。这种广告牌技术可用于展示远处的树木。这里我们并不是使用一个实体的表面来表示树木，甚至不是章节 6.6 中所描述的，使用一对树木轮廓来表示树，我们仅仅使用的是单个树木广告牌。世界空间中的 up 向量会被设置为沿树干的轴。当观察者进行移动的时候，树木会面向观察者，如图 13.11 所示。这仅仅是一个面向摄像机的广告牌，与图 6.28 中的十字树（cross-tree）并不相同。对于这种形式的广告牌，世界空间的 up 向量是固定的，观察方向会作为第二个可调整的向量。一旦这个旋转矩阵构建完成，树木就会被平移到它所在的位置。

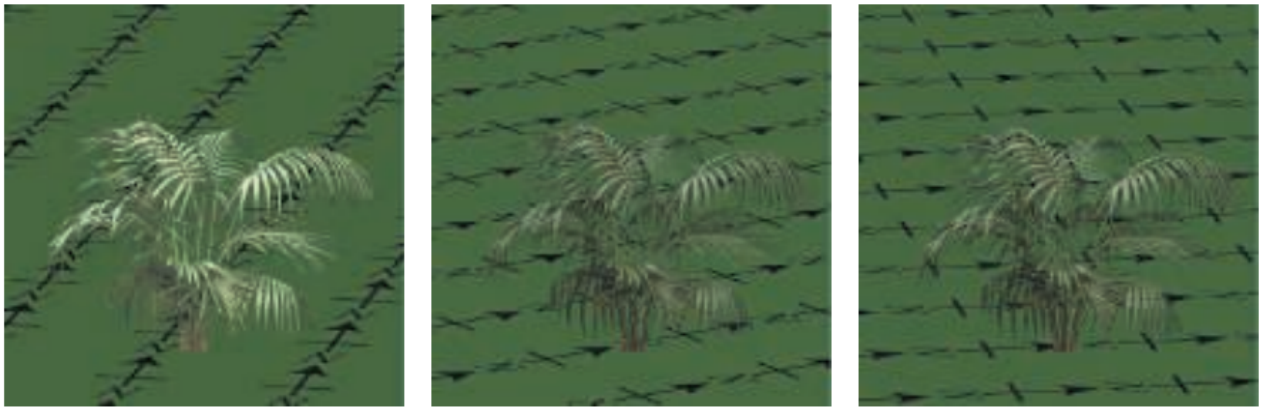


图 13.11：当观察者在场景中移动时，灌木广告牌会发生旋转以面向前方。在这个例子中，灌木从南面被照亮，因此不断变化的视野会使得整体着色随着旋转而发生变化。

这种形式的广告牌与面向世界的广告牌的不同之处在于，具体什么向量是固定的，什么向量又是允许旋转的。对于面向世界的广告牌而言，广告牌会直接朝向观察者，并可以沿着这个观察轴进行旋转；此时广告牌会被旋转，使其 up 方向尽可能与世界空间中的 up 方向对齐。而对于轴向广告牌而言，世界空间中的 up 方向定义了固定的旋转轴，广告牌只能围绕这个轴进行旋转，使其尽可能地朝向观察者。例如：如果广告牌都位于水平地面上，而观察者位于广告牌的上方，那么面向世界的广告牌将会完全朝向观察者，而轴向广告牌则仍会保持直立，更加贴合场景。

由于这种行为模式，因此轴向广告牌存在的一个问题是，如果观察者从树木上方飞过并向下俯瞰，这种视错觉就会被破坏，因为树木看起来几乎只有一片，就像是被切割出来的一样。一种解决方法是添加一个树木的水平横截面纹理（不需要使用广告牌），这样可以帮助改善这个问题[908]。

另一种方法是使用细节层次技术（LOD），将基于图像的模型转换为基于网格的模型[908]。将树木模型从三角形网格转换为广告牌集合的自动化方法将在章节 13.6.5 中进行讨论。Kharlamov 等人[887]提出了相关的树木渲染技术，Klint [908]解释了如何进行大量植被的数据管理和表示。图 19.31 展示了在商业 SpeedTree 包中，用于渲染远处树木的轴向广告牌技术。

与屏幕对齐的广告牌适合表示对称的球形物体一样，轴向广告牌也适合表示具有圆柱形对称性的物体。例如：激光束效果可以用轴向广告牌来进行渲染，因为它们的外观在对称轴周围的任何角度上看起来都是一样的。图 13.12 展示了这种广告牌和其他广告牌的一个例子。图 20.15 则给出了更多的示例。

图 13.12：使用广告牌的例子。图中平视显示（heads-up display，HUD）图形和星形弹道（projectile）都是屏幕对齐的广告牌。右图中巨大的泪滴状爆炸是一个面向视点的广告牌。曲线光柱是由一组相连四边形组合而成的轴向广告牌。为了创建一个连续的光柱，这些四边形会在它们的角上彼此连接，因此它们不再是严格的矩形。

这些类型的技术说明了这些算法及其后续算法的一个重要思想，即像素着色器的目的是评估真正的几何形状，丢弃在物体边界之外发现的片元。对于广告牌而言，当图像纹理完全透明时，就会发现这样的片元。正如我们将看到的，可以通过计算更加复杂的像素着色器，来找到模型存在的位置。而几何形状在这些方法中的作用是使得像素着色器被调用，并给出一些粗略的 z-depth 估计，当然这个估计可能会被像素着色器改进。我们希望避免在模型外部的像素上浪费计算时间，但是我们也不希望几何结构

过于复杂，以致于在每个三角形外部都要进行顶点处理和不必要的像素着色器调用（因为会沿其边缘生成 2×2 的四边形，详见章节 18.2.3），这会增加额外的成本。

13.6.4 Impostor

impostor 是一个广告牌，通过将当前观察点中的复杂物体渲染成图像纹理，并将该图像纹理映射到一个广告牌上，从而创建一个 imposter。impostor 可以用于同一个物体的若干实例，或者是在几帧中进行重复使用，从而分摊它的创建成本。本小节将介绍更新 impostor 的不同策略。Maciel 和 Shirley [1097]早在 1995 年就发现了几种不同类型的 impostor，包括本小节所介绍的这一种。从那时开始，impostor 的定义范围已经缩小到我们在这里所使用的定义[482]。

在物体存在的地方，impostor 图像是不透明的；而在其他任何地方都是完全透明的。imposter 可以通过几种方法来代替几何网格，例如：imposter 图像可以表示由小型静态物体所组成的杂乱东西（clutter）[482, 1109]。由于可以将复杂的模型简化为单个图像，因此 imposter 对于快速渲染远处的物体十分有用。另一种方法是使用最小 LOD 的模型（详见章节 19.9）；然而，这种简化的模型往往会丢失物体的形状信息和颜色信息。而 imposter 则没有这个缺点，因为可以使生成的图像分辨率与显示器的分辨率近似匹配[30, 1892]。另一种使用 impostor 的情况是，某些靠近观察者的物体在移动时，只会将同一面暴露给观察者，也就是说，观察者只能看到这个物体的同一个侧面[1549]。

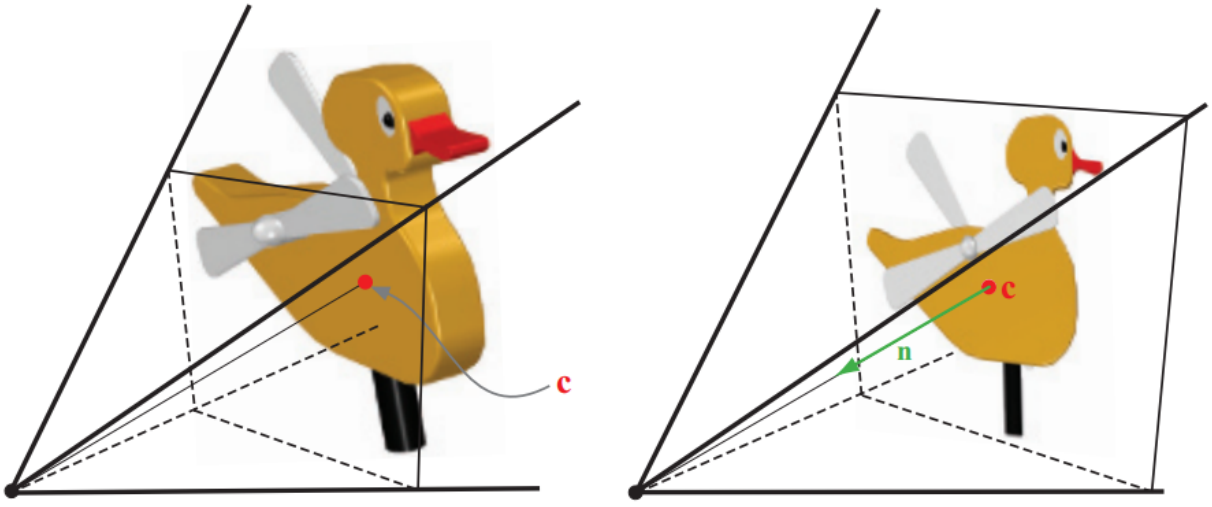


图 13.13：在左边，通过视锥体从侧面观察物体，从而创建了一个 impostor。观察方向朝向物体的中心点 $\mathbf{c}$，使用这个相机设置渲染一副图像，并将其用作 impostor 纹理。如右侧所示，将 imposter 纹理应用于一个四边形上。impostor 的中心等于原始物体的中心，法线（从 imposter 中心发出）直接指向视点。

在渲染物体以创建 impostor 图像之前，我们要将观察者设置到物体包围盒的中心，并选择 impostor 矩形，使其直接指向观察点（如图 13.13 左侧）。impostor 所对应的四边形，其大小为包含物体投影包围盒的最小矩形。impostor 图像的 alpha 值一开始会被清除为 0，而在物体渲染的地方，alpha 值则被设置为 1.0。然后将生成的图像用作面向视点的广告牌，如图 13.13 的右侧所示。当相机或者 impostor 物体发生移动时，纹理的分辨率可能会被放大，这有可能会打破错觉。Schaufler 和 Sturzlinger [1549]提出了一些确定 impostor 图像何时需要更新的启发式方法。

Forsyth [482]给出了许多在游戏中使用 impostor 的实用技巧。例如：对于那些靠近观察者或者鼠标光标的物体，更加频繁地进行更新可以提高感知质量。当动态物体使用 impostor 时，他描述了一种预处理技术，可以确定在整个动画中任意顶点移动的最大距离 d 。这个距离除以动画中的时间步数，即 $\Delta = d/\text{frames}$ 。如果这个 impostor 已经被使用了 n 帧而没有进行更新，则 $\Delta * n$ 会被投影到图像平面上。如果这个距离大于用户设置的阈值，则会更新 impostor。

将纹理映射到面向观察者的矩形上，并不总是能产生令人信服的效果。这个问题在于 impostor 本身并没有厚度，所以当 imposter 与真实的几何图形相结合时，可能会出现一些问题，如图 13.16 所示。Forsyth 建议将纹理沿着观察方向投影到物体的包围框上[482]，这样做至少给了 impostor 一点几何存在感，有了一点几何效果。

通常而言，最好是在物体移动时渲染几何图形，而在物体静止时则切换到 impostor[482]。Kavan 等人[874]引入了 polypostor，其中一个人的模型由一组 impostor 进行表示，人的四肢和躯干各一个。这个系统试图在纯粹的 impostor 和纯粹的几何之间取得平衡。Beacco 等人[122]描述了 polypostor 和大量其他与 impostor 相关的人群渲染技术，并详细对比了每种技术的优势和局限性。图 13.14 给出了一个使用 polypostor 渲染人的例子。

图 13.14：一种 impostor 技术，其中每个单独的动画元素都由一组图像进行表示。这是在一系列遮挡和合成操作中进行渲染的，将这些操作结合起来，能够在给定视图下构建一个令人信服的模型。[122]

13.6.5 广告牌表示

impostor 的一个问题是，渲染出来的图像必须持续面向观察者。如果远处的物体正在改变其朝向，那么就必须重新计算 impostor。Decoret 等人[338]提出了广告牌云（billboard cloud）的概念，可以使得远处的物体更像它们所代表的三角网格。一个复杂的模型通常可以用一小组重叠的镂空广告牌来进行表示。可以将一些额外的信息应用到他们的表面上，例如法线贴图或者位移贴图，以及不同的材质等，这样可以使得模型更有说服力。

这种寻找一组平面的想法要比镂空纹理的类比更加普遍。广告牌之间可以相互交叉，镂空纹理也可以任意复杂。例如：一些研究人员将广告牌与树木模型进行匹配[128, 503, 513, 950]。从具有数万个三角形的模型中，他们可以创建令人信服的广告牌云，而这个广告牌云则由少于 100 个具有纹理的四边形组成，如图 13.15 所示。

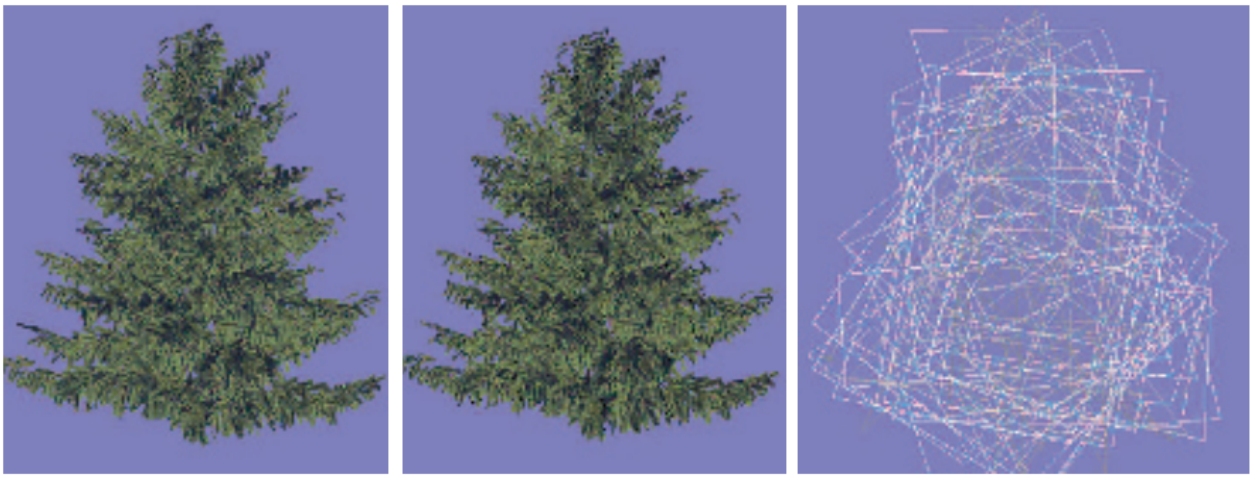


图 13.15：左边是一个由 20610 个三角形组成的树木模型。在中间，这个树木则是由 78 块广告牌组合而成。重叠的广告牌轮廓显示在右边。[950]

使用广告牌云可能会导致相当数量的过度绘制（overdraw），这可能是十分昂贵的。最终的质量也可能会受到影响，因为交叉的镂空纹理可能意味着无法实现严格从后到前的绘制顺序。Alpha 覆盖（详见章节 6.6）可以帮助渲染复杂的 alpha 纹理集合[887]。为了避免过度绘制，SpeedTree 等专业软件包表示并简化了由树叶和四肢所组成的大型网格模型。虽然几何处理需要更多的时间，但是要远远小于过度绘制的成本。图 19.31 给出了这样的一个示例。另一种方法是使用体积纹理来表示这样的物体，并将其渲染为一系列垂直于眼睛观察方向的图层[337]，这部分内容将在章节 14.3 中进行介绍。

13.7 位移技术

如果一个 impostor 的纹理使用了深度信息进行增强的话，这样就定义了一个渲染图元，它称为深度 sprite 或者钉板（nailboard）[1550]。纹理图像是一个 RGB 图像，为每个像素增加一个 Δ 参数，便形成一个 $\mathrm{RGB}\Delta$ 纹理。Δ 存储了从深度 sprite 矩形到深度 sprite 所代表的正确几何深度之间的偏差。这个 Δ 通道是观察空间中的高度场。因为深度 sprite 包含了深度信息，因此它们与 impostor 相比，能够更好地与周围物体融合在一起。当深度 sprite 的矩形穿透附近几何物体的时候，这一点尤其明显，如图 13.16 所示。像素着色器可以通过改变每个像素的 z-depth 来执行这个算法。

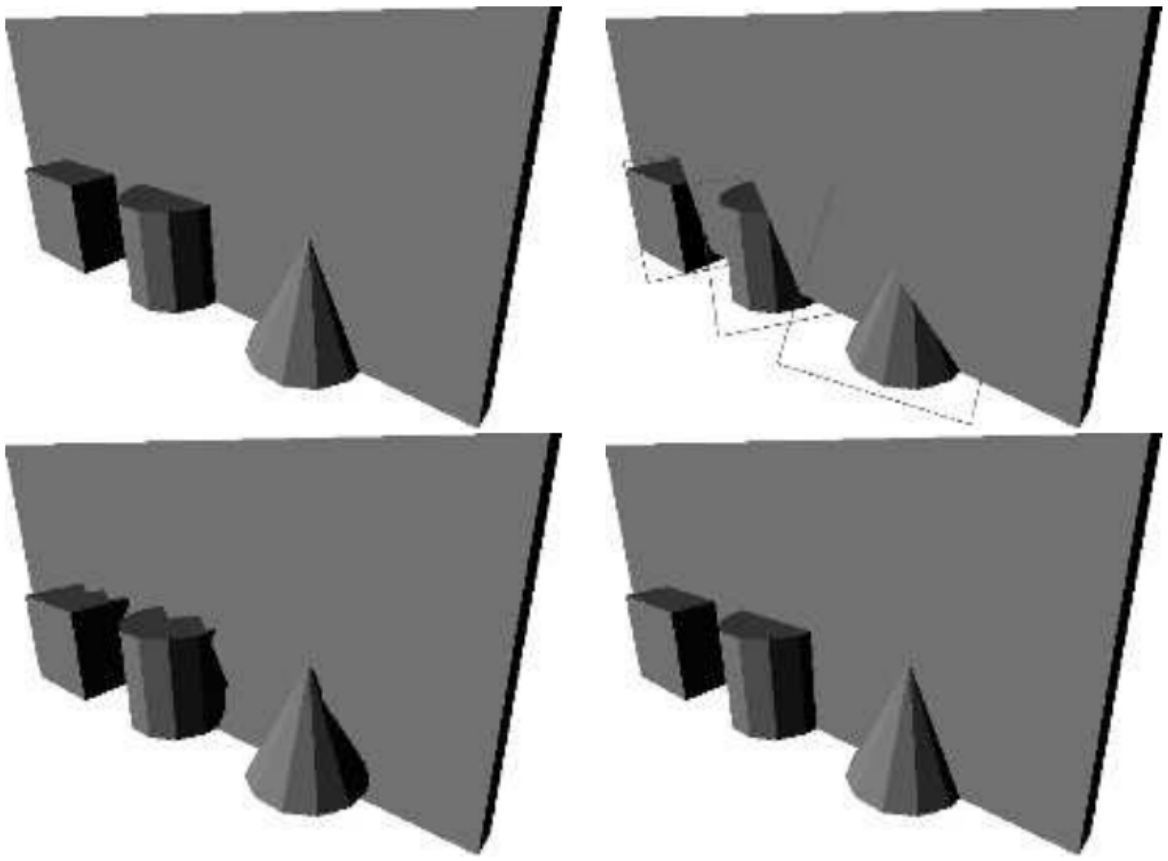


图 13.16：左上角的图像展示了一个简单的场景，该场景使用几何图形进行渲染。右上角的图像创建了对应的 impostor，并将其用于立方体、圆柱体和锥体的渲染，该图像展示了这样做会发生什么。下面两幅图像则展示了使用深度 sprite 时的效果。其中左下角中的深度 sprite 使用 2 bit 的深度偏差，而右下角中则使用了 8 bit 的深度偏差。[1550]

Shade 等人[1611]也描述了一个深度 sprite 的图元，他们使用扭曲（warping）来找到（account for）新的视点。他们引入了一种被称为分层深度图像（layered depth image）的图元，其中的每个像素都包含若干个深度信息。设置多个深度的原因是为了避免在变形扭曲的过程中，由于去遮挡（deocclusion，即让隐藏区域变得可见）而产生的间隙。Schaufler [1551]、Meyer 和 Neyret [1203]也提出了相关的技术。为了控制采样率，Chang 等人[255]提出了一种称为 LDI 树的分层表示方法。

与深度 sprite 相关的是 Oliveira 等人[1324]所引入的浮雕纹理映射（relief texture mapping）。这里的浮雕纹理是一个具有高度场的图像，它用于表示表面的真实位置。与深度 sprite 不同，这个图像并不是渲染在广告牌上的，而是渲染在世界空间中的四边形上。物体可以通过一组在接缝处相匹配的浮雕纹理来进行定义。利用 GPU，高度场可以被映射到表面上，并可以使用光线步进来对其进行渲染，如章节 6.8.1 中所述。浮雕纹理映射与一种被称为光栅化层次包围体（rasterized bounding volume hierarchy，rasterized BVH）的技术相类似[1288]。

Policarpo 和 Oliveira [1425]在单个四边形上使用一组纹理来保存高度场信息，并将每个纹理依次进行渲染。打个简单的比方，在注塑机（injection molding machine）中形成的任何物体，都可以使用两个高度场来进行描述构建。其中每个高度场都代表了该模具的一半。更加精细复杂的模型可以通过使用额外的高度场来进行创建。给定模型的一个特定视图，所需要的高度场数量等于与任何像素发生重叠的表面的最大数量。就像球形广告牌一样，每个底层四边形的主要目的就是让像素着色器对高度场纹理进行评估计算。这个方法也可用于创建表面上复杂的几何细节，如图 13.17 所示。

图 13.17：图中的编织表面使用了 4 个高度场纹理进行建模，并使用了浮雕映射来进行渲染。[1425]

Beacco 等人[122]在人群场景中使用了浮雕 impostor（relief impostor）。在这种表示方法中，会生成模型的颜色、法线和高度场纹理，同时这些纹理与 box 的每个面相关联。当一个面被渲染时，会执行光线步进来找到每个像素上的可见的表面（如果有的话）。这个 box 与模型的刚性部分（“骨头”）相关联，因此它可以执行动画。这个模型并没有蒙皮（skinning），因为我们假定使用这种方法所渲染的角色距离相机很远。纹理化提供了一种简单的方法，来降低原始模型的细节水平，如图 13.18 所示。

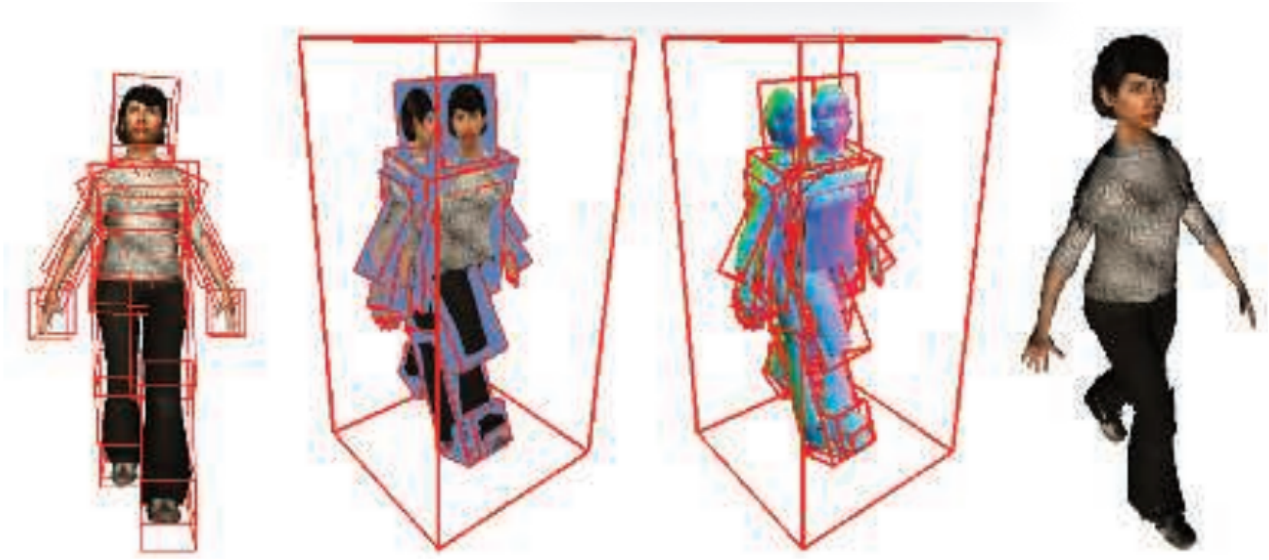


图 13.18：浮雕 impostor。角色的表面模型被分割成若干个 box，然后再使用表面模型来为每个 box 的面创建高度场、颜色和法线纹理。最后再使用浮雕映射方法来进行渲染。[122]

Gu 等人[616]引入了几何图像（geometry image）。其思想是将不规则网格转换为包含位置值的正方形图像。这个图像本身就代表了一个规则的网格，即网格位置中隐式包含了所形成的三角形信息。也就是说，图像中四个相邻的纹素构成了两个三角形。形成这种图像的过程是困难且相当复杂的，这里我们感兴趣的是编码模型的最终图像。这个最终图像可以用于生成网格。其关键特征是，这个几何图像可以进行 mipmap 处理，mipmap 金字塔的不同层级构建了模型的更简化版本。这种顶点数据和纹素数据之间，网格和图像之间的模糊界限，是一种令人着迷且诱人的建模思考方式。几何图像也被用于具有特征保持贴图的地形，以模拟悬垂结构（overhang）[852]。

在这里，我们讲不再讨论如何使用图像来表示整个多边形物体，而是将讨论如何在粒子系统和点云中使用不相连的单个样本。

13.8 粒子系统

粒子系统 [1474]是指一组由独立微小物体所构成的集合，并使用一些算法来控制这些物体的运动。粒子系统的应用十分广泛，包括模拟火、烟、爆炸、水流、旋转星系和其他现象等。因此，一个粒子系统在控制物体渲染的同时，还会控制物体的动画。在粒子的生命周期内创建、移动、修改和删除粒子的相关控制同样也是粒子系统的一部分。

与本小节内容相关的是这些粒子的建模方式和渲染方式。其中每个粒子既可以是单个的像素，也可以是从粒子先前位置到当前位置绘制的轨迹线段，它们通常会使用广告

牌来进行表示。如章节 13.6.2 中所述，如果这些粒子是圆形的，那么 up 向量的选取将会与它们的显示不相关。换句话说，此时我们只需要知道粒子的位置就可以确定其方位。图 13.19 展示了一些粒子系统的例子。每个粒子的广告牌可以通过调用几何着色器（geometry shader）来进行生成，但在实践中，使用顶点着色器来生成 sprite 可能会更快[146]。除了表示粒子的图像纹理之外，还可以包含其他的纹理，例如法线贴图等。轴向广告牌可以表现较粗的线条，图 14.18 展示了一个使用线段来绘制雨滴的例子。

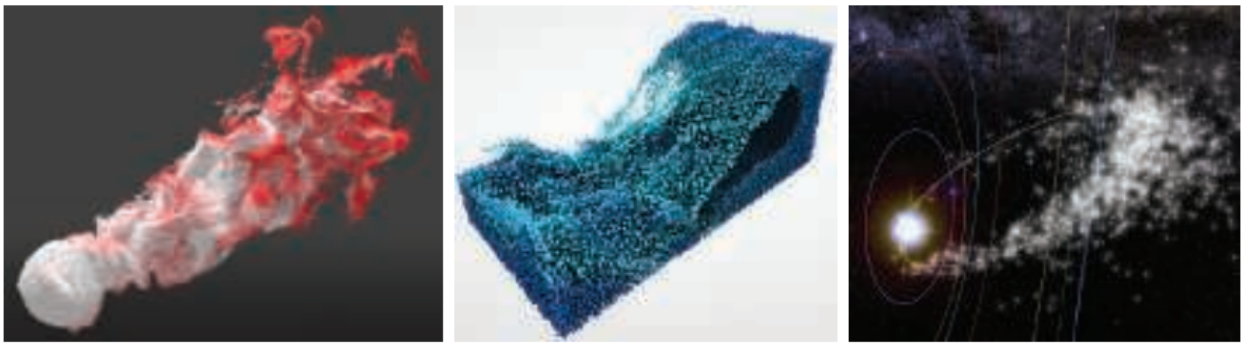


图 13.19：粒子系统：烟雾模拟（左），流体模拟（中），以及银河系天空盒上的流星路径（右）。

如果使用半透明广告牌粒子来表示烟雾等现象，那么就必须解决如何正确渲染透明物体的问题。我们可能需要对这些半透明广告牌进行从后到前的排序操作，但是这样做的代价可能会很高。Ericson [439]提供了一系列高效渲染粒子的建议，我们在这里列出了其中的一些，以及相关的论文：

- 从浓密（thick）的镂空纹理中构建烟雾效果；避免半透明意味着不需要进行排序和混合操作。
- 如果需要使用半透明效果，则优先考虑叠加混合或者减法混合，因为这两种混合方式不需要进行排序[987, 1971]。
- 使用少量的动画粒子，相比于使用大量的静态粒子，可以提供类似的质量和更好的性能表现。
- 为了保持帧率稳定，在渲染粒子的数量上使用一个动态上限值。
- 让不同的粒子系统使用相同的着色器，从而避免状态切换所带来的额外成本（章节 18.4.2）[987, 1747]。
- 使用一个包含所有粒子图像的纹理图集或者纹理数组，从而避免调用纹理变更所带来的开销[986]。

- 在一个较低分辨率的缓冲区中平滑地绘制不同的粒子（例如烟雾）[1503]，并在 MSAA 完成之后再进行合并或者绘制。

Tatarchuk 等人[1747]对最后一个观点进行了进一步的阐述。他们将烟雾渲染到一个相当小的缓冲区中，只有屏幕分辨率的十六分之一大小，并使用一个方差深度贴图来帮助计算粒子效果的累积分布函数。详情请参阅他们的展示和介绍。

如果存在大量需要渲染的粒子，那么一次完整的排序操作可能会十分昂贵。美术指导可以手动指示渲染顺序，对不同效果进行正确地分层处理，从而改善排序问题。而对于较小的粒子或者对比度较低的粒子，可能不需要进行排序处理。粒子有时候也可以以某种排序的顺序来进行发射[987]。如果粒子是相当透明的，那么可以使用不需要排序的加权混合透明度技术[394, 1180]。更加复杂的、顺序无关的透明系统也是可能的，例如：Kohler [920]简要介绍了将粒子渲染到一个存储在纹理数组中的九层深度缓冲区中，然后再使用计算着色器来进行排序处理。

13.8.1 粒子着色

具体的着色操作取决于粒子的类型。例如火花（spark）之类的发射器（emitter）并不需要进行着色，为了简单起见，通常会使用叠加混合（additive blending）。Green [589]描述了如何将流体系统作为球形粒子渲染到深度图像中，然后再对深度图像进行模糊处理，并从中导出法线数据，最终将结果与场景进行合并。对于诸如灰尘或者烟雾的微小颗粒，可以使用逐图元（per-primitive）或者逐顶点（per-vertex）的着色方式[44]。然而这样的光照方式会使得具有不同表面的粒子看起来很平。可以为粒子提供一个法线贴图，从而给予它们适当的表面法线来将其照亮，但是这样会带来纹理访问的额外开销。对于圆形粒子而言，在粒子的四个角上使用四条发散的法线向量可能就足够了[987, 1650]。烟雾粒子系统可以有更加精细的光线散射模型。可以使用辐射度法线映射（章节 11.5.2）或者球谐函数来照亮这些粒子[1190, 1503]。在更大的粒子上也可以使用曲面细分操作，使用域着色器（domain shader）来在每个顶点位置上累积光照效果[225, 816, 1388, 1590]。

可以在每个顶点位置上计算光照效果，并在粒子四边形上进行插值[44]。这个操作的速度很快，但是对较大的粒子而言，这样生成的质量较低，因为远处的顶点很可能会错过小范围光源的光照贡献。一种解决方案是在每个像素的基底上对粒子进行着色处理，但是所使用的分辨率要比最终图像所使用的分辨率低。为此，每个可见粒子会在光照贴图纹理（light-map texture）中分配一个 tile（瓦片）[384, 1682]。每个 tile 的分辨率可以根据屏幕上的粒子大小进行调整，例如：根据屏幕上的投影面积，将 tile 的分辨率在 1×1 和 32×32 之间进行调整。一旦分配好了 tile，那么每个 tile

所对应的粒子就会进行渲染，并将像素所对应的世界位置写入一个二级纹理中。然后使用一个计算着色器来计算这个二级纹理中到达每个位置上的 radiance。radiance 是通过对场景中的光源进行采样来收集的，并使用一个加速结构来针对潜在的贡献光源进行计算评估，这部分内容将在第 20 章中进行介绍。由此生成的 radiance 可以直接作为颜色信息或者球谐函数来写入到光照贴图纹理中。当每个粒子最终被渲染到屏幕上时，通过在粒子四边形上映射每个 tile，并使用一次纹理读取来获得每个像素所对应的 radiance，从而将光照应用到粒子上。

也可以采用相同的原则，为每个发射器都分配一个 tile [1538]。在这种情况下，使用一个深度光照贴图纹理（deep light-map texture），将会给拥有许多粒子的光照效果添加一种体积的感觉。值得注意的是，由于粒子的平坦特性并且粒子通常会朝向观察者，当观察点围绕任何粒子发射器进行旋转的时候，本小节中所介绍的各个粒子光照模型都会产生可见的闪烁瑕疵。

除了粒子的光照之外，还需要特别注意粒子的体积阴影和自阴影现象。为了接收来自其他遮挡物的阴影，较小的粒子通常可以在它们的顶点位置上对阴影贴图进行深度测试，而不是在每个像素位置上。由于这些粒子是分散的点，并且会被渲染为面向摄像机的简单四边形，因此我们无法在阴影贴图上使用光线步进来实现对其他物体的阴影投射。不过，可以使用飞溅（splatting）方法（章节 13.9）来实现这一点。为了在其他场景元素上投射来自太阳的阴影，我们可以将粒子飞溅到一个纹理中，并在一个缓冲区（一开始会被清除为 1）中乘以其逐像素的透光率 $T_r = 1 - \alpha$ 。这个纹理可以由用于灰度的单个通道或者由用于彩色透射的三个通道组成。这些纹理与阴影级联层级一起，通过将透光率乘以规则不透明的阴影级联所产生的可见性（详见章节 7.4），来将其应用到场景中。这种技术有效地提供了一层透明的阴影[44]。这种技术唯一的缺点是，粒子会错误地将阴影投射到粒子与太阳之间的不透明元素上。可以通过谨慎的关卡设计来避免这种情况的发生。

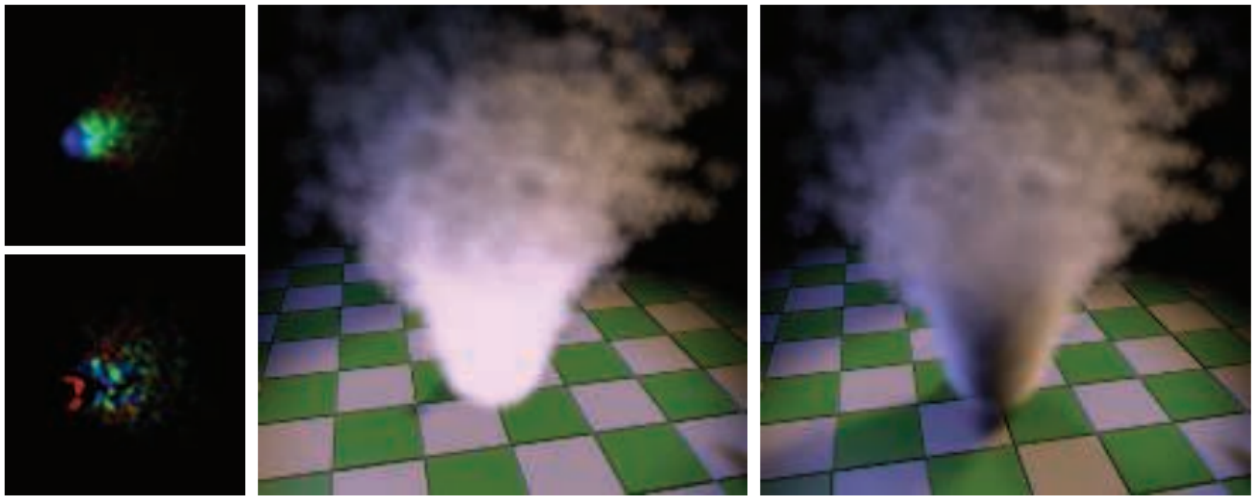


图 13.20：粒子使用傅里叶不透明度映射的投射出的体积阴影。在左边展示了傅里叶不透明度贴图，它包含了来自一个聚光灯视图的函数系数。在中间，渲染后的粒子没有阴影效果。在右边，所产生的体积阴影会投射在粒子上，以及场景的其他不透明表面上。[816]

为了实现粒子的自阴影效果，必须使用一些更加先进的技术，例如傅里叶不透明度映射（Fourier opacity mapping，FOM）[816]，如图 13.20 所示。首先会从光源的视角来渲染粒子，并将将它们的贡献高效地添加到透光率函数（transmittance function）中，这个函数会被表示为存储在不透明度贴图中的傅里叶系数。当从这个角度渲染粒子的时候，可以通过对不透明度贴图进行采样，获得傅里叶系数来重建透光率信号。这种表示方法可以很好地表达平滑的透光率函数。然而，为了满足纹理存储的需求，该方法仅使用了有限个数的傅里叶基底，因此当透光率变化较大时，它会受到振铃效应（ringing）的影响。这可能会导致在粒子四边形上出现不正确的亮区或者暗区。FOM 方法非常适合粒子渲染，但是也可以使用一些其他的方法，这些方法具有各自不同的优缺点。这些方法包括章节 14.3.2 中将会介绍的自适应体积阴影贴图（adaptive volumetric shadow map）[1531]，该方法类似于深度阴影贴图（deep shadow map）[1066]；GPU 优化的粒子阴影映射[120]，它类似于不透明度阴影贴图[894]，但是这种方法只适用于面向相机的粒子，因此它不适用于带状或者运动拉伸的粒子；以及透光率函数映射[341]，它和 FOM 相类似。

另一种方法是在包含消光系数（extinction coefficient） σ_t 的体积中，对粒子进行体素化[742]。这些体积可以像 clipmap 一样被放置在相机周围[1739]。该方法可以同时对粒子和参与介质的体积阴影进行统一评估计算，因为它们都可以在这些共享体积中进行体素化。我们会生成一个单独的深度阴影贴图[894]，存储来自这些“消光体积（extinction volumes）”中的逐体素透光率 T_r ，这将会自动产生从两个来源（粒子、参与介质）投射出的体积阴影。由此产生的相互作用还有很多：粒子和参与介质之间可以相互投射阴影，以及产生自阴影效果，如图 14.21 所示。最终结果的质量与体素的大小有关，为了能够实现实时运行，所设置的体素大小可能会很大。这将导致较为粗糙，但是视觉上较为柔和的体积阴影。更多细节详见章节 14.3.2。

13.8.2 粒子模拟

如何使用粒子来高效且令人信服地模拟一些物理过程，这是一个十分广泛的主题，超出了本书的讨论范围，因此我们将向你推荐一些参考资源。GPU 可以为 sprite 生成动画路径，甚至是执行碰撞检测。流式输出可以控制粒子的创建和销毁，这是通过将结果存储在一个顶点缓冲区中，并在 GPU 上每帧更新这个缓冲区来实现的[522, 700]。如果可以使用无序访问视图缓冲区（unordered access view buffer，UAV）

的话，那么这个粒子系统可以完全基于 GPU，并由顶点着色器进行控制[146, 1503, 1911]。

图 13.21：游戏《命运 2》中粒子系统的例子。

Van der Burg 的论文[211]和 Latta 的综述[986, 987]，对粒子模拟的基础知识进行了快速介绍。Bridson 关于计算机图形学流体模拟的书对理论进行了深入讨论[197]，包括基于物理模拟各种形式的水、烟雾和火焰的技术。一些实践者对交互式渲染器中的粒子系统进行了讨论。Whitley [1879]详细介绍了为《命运 2》所开发的粒子系统，图 13.21 展示了一个示例图像。Evans 和 Kirczenow [445]讨论了他们对于 Bridson 论文中的流体算法的实现。Mittring [1229]给出了关于如何在虚幻 4 引擎中控制粒子的简要细节。Vainio [1808]深入研究了游戏《声名狼藉：私生子》中粒子效果的设计和渲染。Wronski [1911]提出了一种高效生成和渲染雨水的系统。Gjøl 和 Svendsen [539]讨论了烟雾和火焰效果的实现，以及许多其他基于采样的技术。Thomas [1767]实现了一个基于计算着色器的粒子模拟系统，该系统包括碰撞检测，透明排序和基于 tile 的高效渲染。Xiao 等人[1936]提出了一种交互式的物理流体模拟器，也可以计算需要显示的等值面。Skillman 和 Demoreuille [1650]通过他们的粒子系统和其他基于图像的效果，将游戏《野兽传奇》的效果发挥到极限（turn the volume up to eleven）。

13.9 点渲染

1985 年，Levoy 和 Whitted 撰写了一份开创性的技术报告[1033]，他们建议使用点来作为一种新的图元来渲染一切物体。其总体思路是使用大量的点来表示一个表面，并对这些点进行渲染。而在随后的 pass 中，执行高斯过滤从而填充渲染点之间的空白间隙。高斯滤波器的半径取决于表面上点的密度，以及投影在屏幕上的密度。Levoy 和 Whitted 在 VAX-11/780 上实现了这个系统。

然而，直到大约 15 年后，基于点的渲染才再次引起人们的兴趣。这个渲染思路的再次兴起有两个主要原因：第一个原因是那时的计算能力达到了可以以一个交互速率来进行基于点的渲染；第二原因是使用激光测距扫描仪（laser range scanner）可以获得非常详细的模型[1035]。从那时起，大量的 RGB-D（深度）设备已经达到了实用水平，例如：用于地形测绘的航空 LIDAR（光探测和测距，LIght Detection And Ranging，又叫做激光雷达）仪器[779]；用于短程数据捕获的微软 Kinect 传感器；iPhone 的 TrueDepth 相机；谷歌的 Tango 设备。自动驾驶汽车上的激光雷达系统，每秒可以记录数百万个点。通过摄影测量或者其他计算摄影技术处理过的二维图像也可以用于提供点云数据集。这些技术的原始输出是一组带有额外数据（通常是强度或者颜色）的三维点。还可以提供一些额外的分类数据，例如：某个点是否来自于建筑物或者路面[37]。这些点云（point cloud）可以通过多种方式进行操作和渲染。

这样的模型最初会表示为不相连的三维点。读者可以阅读 Berger 等人[137]的论文，深入了解点云过滤技术和将点云转化为网格的方法。Kotis 和 Cozzi [930]提出了一种以交互速率处理、体素化和渲染这些体素的方法。而本小节中将会讨论直接对点云数据进行渲染的技术。

QSplat [1519]是一个很有影响力的、基于点的渲染器，它首次发布于 2000 年。它使用了一个球体的层次结构树来描述模型。这个树中的节点会被压缩，从而允许渲染由数亿个点所组成的场景。一个点会被渲染为一个具有半径的形状，这个形状被称为 splat。还可以使用具有不同形状的 splat，例如正方形、不透明圆和模糊圆。换句话说，splat 就是粒子，尽管我们最终的渲染目的是表示一个连续的表面，图 13.22 展示了这样的一个例子。渲染过程可以在树的任何层次上停止，该级别的节点会被渲染为与节点球体半径相同的 splat。因此，这个包围球层次结构的构造能够使得在任何层次上都不会出现空洞。由于树的遍历（渲染）可以在任何级别停止，因此可以通过在可用渲染时间耗尽时终止遍历，从而获得交互式的帧率。当用户停止移动时，可以对渲染质量进行反复优化，直到达到层次结构的叶子节点。

图 13.22：这些模型使用了基于点的渲染方法，并使用圆形 splat 进行渲染。左图展示的是天使露西的完整模型，它包含 1000 万个顶点。然而，在实际渲染中只使用了大约 300 万个 splat。中间和右边的图片放大了头部细节。中间的图像在渲染期间使用了大约 4 万个 splat。当观察者停止移动时，结果会逐渐收敛成右侧的图像，它包含 60 万个 splat。

大约在同一时间，Pfister 等人[1409]提出了面元（surfel）的概念，即一个表面元素（surface element）。它也是一个基于点的图元，用于表示物体表面的一部分，因此面元中总是会包含一个法线。会使用一个八叉树（章节 19.1.3）来存储采样的面元：位置，法线和过滤的纹理。在渲染过程中，面元会被投影到屏幕上，然后会使用一个可见性飞溅算法（visibility splatting algorithm）来填充产生的空洞。QSplat 和面元的论文确定并解决了点云系统中的一些关键问题：管理数据集大小和从给定的点集中渲染出令人信服的表面。

QSplat 使用一个层次结构，但是这个层级结构会被细分到单个点的级别，层次结构内部的父节点是包围球体，每个包围球体都包含一个点，这个点代表了子节点的平均值。Gobbetti 和 Marton [546]引入了分层点云（layered point cloud），这是一种能够更好地映射到 GPU 上的分层结构，并且它不会产生人为的“平均”数据点。其中每个内部节点和子节点都包含了大致相同数量的点（记作 n），这些点会在一次 API 调用中作为一个集合来进行渲染。我们从整个集合中取出 n 个点来形成根节点，将其作为模型的一个粗略表示。选择一个点与点之间距离大致相等的集合要比随机选择的结果更好[1583]。法线或者颜色之间的差异也可以用于聚类选择[570]。剩余的点在空间上会被划分为两个子节点，在每个节点上都重复上述的过程，即选择 n 个具有代表性的点，并将其余的点划分为两个子集。这种选择和细分过程会一直持续下去，直到每个子节点包含的点数量小于等于 n，如图 13.23 所示。Botsch 等人[180]的工作是 SOTA 的一个很好例子，这是一种使用延迟渲染（章节 20.1）和高质量滤波的 GPU 加速技术。在显示期间，可见的节点会被加载并进行渲染，直到满足某些限制

条件。节点所占据的相对屏幕大小可以用来确定该点集的重要程度，并可以提供一个渲染广告牌的大小估计。由于这个方法不会为父节点引入新的点，因此内存使用量与存储的点数量成正比。这种方案的一个缺点是，当放大单个子节点的时候，所有的父节点都必须被发送到管线中，即使每个父节点中只有几个点是可见的。

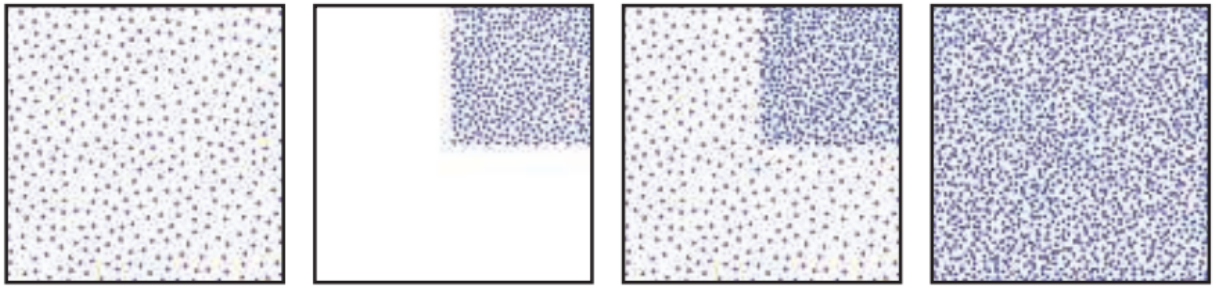


图 13.23：分层点云。第一幅图展示的根节点，它包含了从子节点数据中提取的稀疏子集。第二幅图展示了一个子节点中所包含的点；第三幅图是根节点与该子节点一起显示的结果，请注意子节点区域是如何被填充的。第四幅图则是完整的点云，它包含了根节点和所有的子节点。[1583, 12]

在当前的点云渲染系统中，数据集可能会非常巨大，由数千亿个点组成。由于这些集合无法完全加载到内存中，更不用说以交互速率来进行渲染显示了，因此几乎每个点云渲染系统都会使用分层结构来进行加载和显示。实际使用的分层方案可能会受到具体数据的影响，例如：对于地形数据而言，四叉树通常要比八叉树更加适合。关于点云数据结构的高效创建和高效遍历已经有了相当多的研究。Scheiblauer [1553]对这一领域中的研究，以及表面重建技术和其他算法进行了总结。Adorjan [12]则对这几个系统进行了综述，他所讨论的重点是摄影测量生成的建筑点云。

理论上来说，splat 可以提供单独的法线和半径来定义一个表面。但是在实践中，这样的数据所占用的内存太大，并且需要经过大量的预处理工作才能获得这样的数据，因此通常会使用固定半径的广告牌。由于排序和混合的成本，正确渲染半透明 splat 广告牌的开销可能会很大，同时还可能会充满瑕疵。因此通常会使用不透明的广告牌（正方形或者镂空的圆形）来兼顾交互性和渲染质量，如图 13.24 所示。

图 13.24：选择其中的 500 万个点来渲染一个包含 1.45 亿个点的小镇数据集。通过检测深度差异来增强边缘。当点云数据稀疏或者广告牌半径太小时，就会出现空洞。底部一行是图像预算分别为 50 万个点、100 万个点、500 万个点时所对应的渲染结果。[1583]

如果点云数据并不包含法线信息，那么则可以使用各种技术来提供着色效果。一种基于图像的方法是计算某种形式的屏幕空间环境光遮蔽（章节 11.3.6）。通常的做法是，先将所有的点渲染到一个深度缓冲区中，并使用足够宽的半径来形成一个连续的表面。在随后的渲染 pass 中，每个点的着色效果都会按照靠近观察者的相邻像素数量，来成比例的变暗。眼穹光照（eye-dome lighting，EDL）可以进一步突出表面细节[1583]。EDL 的工作原理是检查相邻像素的屏幕深度，并找到比当前像素更加接近观察者的那些像素。对于每个这样的相邻像素，会计算它与当前像素之间的深度差并进行求和。然后将这些差值的平均值与强度因子相乘，来作为指数函数 exp 的输入，最终对着色效果进行修正，如图 13.25 所示。

图 13.25：左侧：具有法线数据的点在一个单独的 pass 中进行渲染。中间：一个没有法线数据的点云，使用屏幕空间环境光遮蔽进行渲染。右边：同样是没有法线数据的点云，使用 EDL 进行渲染。后面两种方法都需要额外执行一个 pass，从而建立图像中的深度缓冲。

如果点云中的每个点都具有颜色信息或者强度信息，那么实际上光照效果本身已经被嵌入其中了，因此可以直接进行显示，尽管光泽物体或者反光物体并不会对相机视图的变化做出响应。而其他一些非图形的属性信息（例如物体类型或者高度等）也可以用来显示这些点。这里我们只涉及了管理点云和渲染点云的基础知识。Schuetz [1583]对各种渲染技术进行了讨论，并提供了相应的实现细节，以及一个高质量的开源系统。

点云数据还可以与其他数据源相结合。例如：Cesium 程序可以将点云与高分辨率地形、图像、矢量贴图数据和摄影测量生成的模型结合起来。另一种与扫描相关的技术是从天空盒的视角来捕捉环境信息，并将颜色信息和深度信息保存下来，从而使得捕捉的场景具有实际存在感。例如：用户可以将数字合成的模型添加到场景中，并使它们与这种类型的天空盒进行正确合并，因为周围图像中的每个点都具有深度信息，如图 13.26 所示。

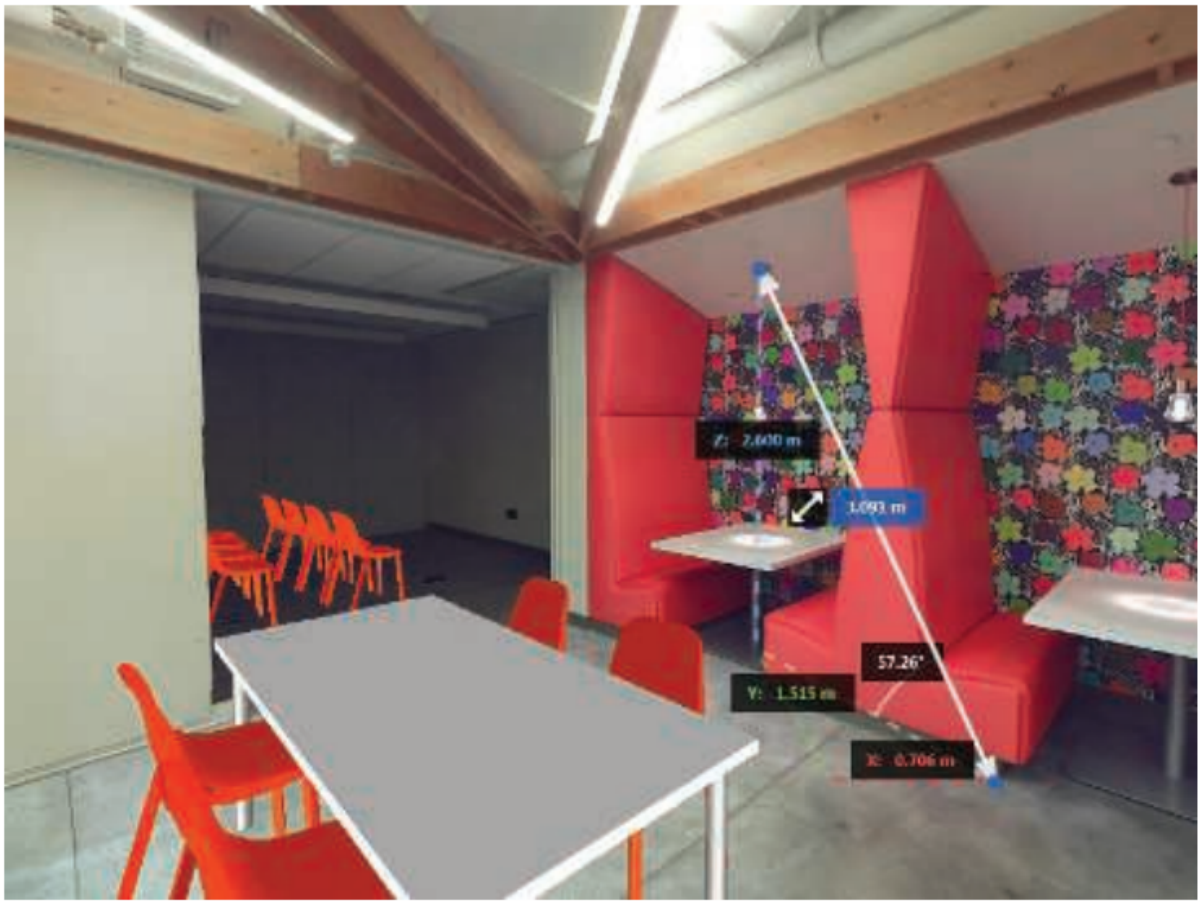


图 13.16：场景中的每个像素都有深度信息。对于固定的视图位置（相机位置固定，方向可以不固定），用户可以在正确处理遮挡的情况下，测量世界空间位置和定位虚拟物体。

目前的 SOTA 已经取得了相当大的进展，并且这些技术在数据捕获和数据显示之外的领域也得到了应用。作为一个例子，我们简要总结了 Evans [446]为游戏《Dreams》所提出的基于点的实验性渲染系统。每个模型由若干聚类的层次包围盒结构（bounding volume hierarchy，BVH）进行表示，其中每个聚类包含 256 个点。这些点是由符号距离函数（signed distance function）生成的（章节 17.3）。为了支持 LOD，会为每个 LOD 层次生成单独的 BVH、聚类和点。为了从高层次细节过渡到低层次细节，会将高密度子簇中的点数量随机减少到原来的 25%，然后再将低层次细节中的父聚类交换进来。该渲染器基于一个计算着色器进行构建，它使用原子操作来将点划分到一个帧缓冲中，从而避免冲突。它实现了若干种技术，例如随机透明度，景深（使用基于弥散圆的抖动 splat），环境光遮蔽和不完美的阴影贴图[1498]等。为了对瑕疵进行平滑处理，还会进行时域抗锯齿（章节 5.4.2）。

点云代表了空间中的任意位置，因此点云的渲染是具有挑战性的，因为我们往往不知道或者很难获得点与点之间的间隙信息。Kobbelt 和 Botsch [916]对这个问题以及其他与点云相关的研究领域进行了综述。在本章节的最后，我们将转向一种非多边形的

表示方法，在这种表示方法中，样本与其相邻样本之间的距离总是相同的，即体素数据。

13.10 体素

像素（pixel）指的是“图像元素（picture element）”，纹素（texel）指的是“纹理元素（texture element）”，相应的，体素（voxel）指的是“体积元素（volume element）”。每个体素都代表了均匀三维网格中的一个空间体积，通常是一个立方体。体素是存储体积数据的传统方法，可以表示从烟雾到 3D 打印模型，从扫描骨骼到地形表示的许多物体。体素中可以只存储一个 bit 的数据，用来表示体素中心是位于物体内部还是物体外部。对于医疗应用而言，可以存储密度或者不透明度，还可能会存储体积流量（volumetric flow）。还可以存储颜色、法线、符号距离或者其他数据，从而便于渲染。这里我们不需要知道每个体素的具体位置信息，因为网格中的索引便决定了它的位置。

13.10.1 应用

模型的体素表示方法可以用于许多不同的目的。一个规则的数据网格适合进行与整个物体有关的各种操作，而不仅仅是对其表面进行操作。例如：用体素表示的一个物体，该物体的体积就是其内部体素的总和。网格的规则结构和体素明确定义的局部邻域，意味着可以通过元胞自动机（cellular automata）或者其他算法来模拟烟雾、侵蚀或者云层形成等现象。有限元分析（finite element analysis）方法也可以利用体素，来确定物体的抗拉强度。对模型进行雕塑或者雕刻，实际上就变成了一个减去体素的问题。相反，可以通过将多边形模型放置到体素网格中，确定该模型与哪些体素重叠来构建一个更加精细的模型。传统多边形工作流必须处理奇异点（singularity）和精度问题，而体素这种构造实体几何建模的操作是更加高效的、更加可预测的，并且保证有效。基于体素的系统，例如 OpenVDB [1249, 1336]和 NVIDIA GVDB Voxels [752, 753]，被用于电影制作、科学和医疗可视化、3D 打印以及其他应用领域，如图 13.27 所示。

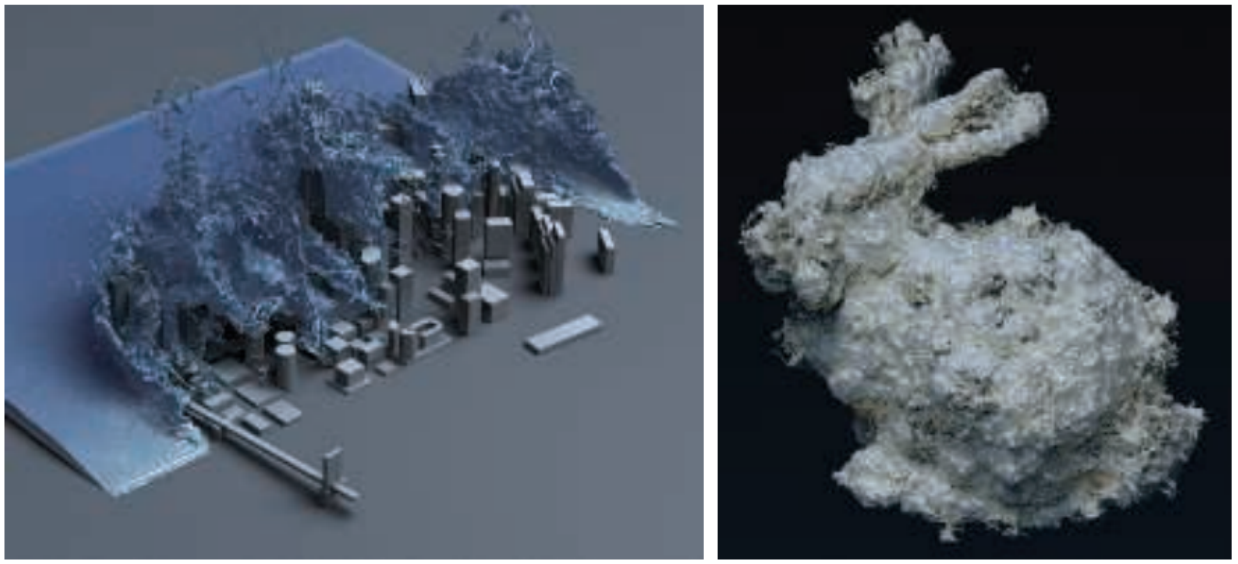


图 13.27：体素应用程序。左侧：流体模拟是直接在稀疏体素网格上进行计算的，并将其渲染为一个体积。右侧：一个多边形的兔子模型被体素化成一个符号距离场，然后再使用噪声函数对其进行扰动，并对一个等值面进行渲染。[1925]

13.10.2 体素存储

体素数据对于存储空间的需求很大，随着体素分辨率的不断提高，数据量的增长速度为 $O(n^3)$ 。例如：假设每个维度上的分辨率为 1000，那么一个三维的体素网格将会产生 10 亿个位置。基于体素的游戏例如《我的世界》等，可以拥有巨大的世界。在这款游戏中，地形数据以 $16 \times 16 \times 256$ 体素块的形式进行流式加载，并以每个玩家为中心，对一定半径范围内的数据进行渲染。每个体素都会存储一个标识符、额外的方向或者样式数据。每种类型的方块都有自己对应的多边形表示，无论是使用立方体进行表现的实心石头，使用 alpha 纹理进行表现的半透明窗户，还是使用一对广告牌进行表现的草。详见图 12.10 和图 19.19。

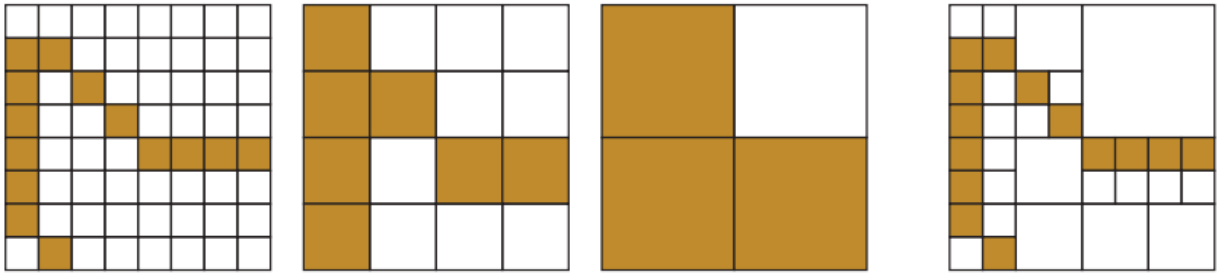


图 13.28：二维形式的稀疏体素八叉树（SVO）。给定最左边的一组体素，我们可以观察到树上的哪些父节点中是包含有体素的。最右边是最终八叉树的可视化，它展示了每个网格位置上所存储的最深节点。[963]

存储在体素网格中的数据通常具有很强的一致性（coherence），即相邻位置上很可能会具有相同或者相似的值。根据数据来源的不同，体素网格中的绝大部分可能都是空的，这被称为稀疏体积（sparse volume）。这里的一致性和稀疏性，都会驱动我们使用一种更加紧凑的表示方法来存储体素数据。例如：可以在网格上施加一个八叉树（章节 19.1.3）。在最低的八叉树层次上，每个 $2 \times 2 \times 2$ 的体素样本可能会包含相同的数据，我们可以在八叉树中进行记录，并丢弃冗余的体素。在树结构的上层也可能会检测到这样的相似性，同样地，我们可以丢弃这些相同的子八叉树节点，只有数据不同的地方才需要对它们进行存储。这种稀疏体素八叉树（sparse voxel octree，SVO）的表示方法[87, 304, 308, 706]，产生了一种天然的 LOD 表示方法，相当于三维体积上的 mipmap，如图 13.28 和图 13.29 所示。Laine 和 Karras [963]为 SVO 数据结构提供了丰富的实现细节和各种扩展方法。

图 13.29：不同 LOD 下的体素光线追踪结果。沿着包含模型的体素网格边缘，从左到右的分辨率分别为 256，512 和 1024。[753]

13.10.3 体素的生成

体素模型的输入可以有各种来源。例如：许多扫描设备在任意位置上生成数据点。GPU 可以加速体素化（voxelization）过程，这是一个将点云[930]、多边形网格或者其他表示方法转换为一组体素的过程。对于多边形网格数据而言，Karabassi 等人[859]提出了一种快速而粗略的体素化方法，即从 6 个正交视图（顶部、底部和四个侧面）来渲染物体。每个视图上的渲染都会生成一个深度缓冲区，每个像素都保存了该方向上第一个可见体素的位置。如果一个体素的位置超出了存储在 6 个缓冲区中的对应深度，那么它就是不可见的，因此会被标记为位于物体内部。这种方法有一些瑕疵，它会错过一些在这六个视图中无法看到的特征，从而导致一些体素被错误地标记在物体内部。不过，对于简单的模型而言，这种方法就已经足够了。

受视觉外壳（visual hull）的启发[1139]，Loop 等人[1071]使用了一个更简单的系统，来创建现实世界中人的体素化数据。该系统会捕获一组人的图像，并提取出图像中人的轮廓（silhouette）。每个轮廓都会在给定的相机位置上切割出一组体素，即只有在你能够看到人的像素位置上，才会生成与它们相关的体素。

体素网格也可以从一组图像集合中进行创建，例如使用医学影像设备来生成局部切片，然后再将这些切片堆叠起来。沿着同样的思路，网格模型可以进行切片渲染，在模型内部找到的体素会被适时地记录下来。通过调整相机的近裁剪平面和远裁剪平面，从而单独渲染模型的一个切片，并对生成的体素内容进行检查。Eisemann 和 Decoret [409]引入了切片贴图（slicemap）的概念，其中 32 bit 的渲染目标被认为是 32 个独立的深度，每个深度都有一个位标记。渲染到这个体素网格中的三角形的深度会被转换为对应的位标记并进行存储。然后可以在一个渲染 pass 中渲染这 32 层数据，如果使用更宽通道的图像格式和多个渲染目标，则可以为该 pass 提供更多的体素层。Forest 等人[480]给出了这个算法的实现细节，他同时指出，在现代 GPU 上，一个 pass 可以渲染多达 1024 层体素。请注意，这个切片算法只能识别模型的表面数据，即模型的边界表示。而之前我们所提到的六正交视图算法还可以识别（尽管有时会识别错误）完全在模型内部的体素。图 13.30 给出了三种常见的体素化类型。Laine [964]对相关术语、各种体素化类型，以及生成和使用它们所涉及的问题进行了全面彻底的处理。

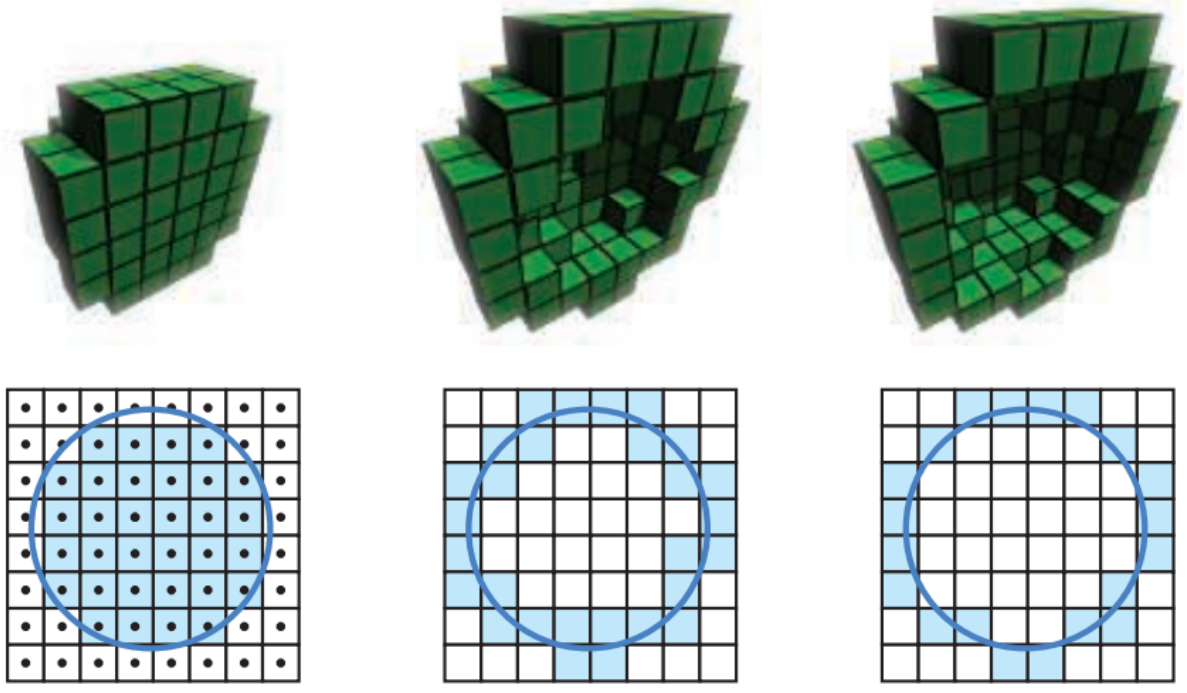


图 13.30：一个球体以三种不同的方式进行了体素化，并展示了其截面。左边是一个实体的体素化，通过测试每个体素相对于球体中心的距离来进行确定。中间则是一个保守的体素化，其中任何接触到球体表面的体素都会被选择并存储下来。这个表面被称为 26-分离体素化（26-

separating voxelization），在其 $3 \times 3 \times 3$ 的邻域内，没有内部体素与外部体素相邻。换句话说，内部体素和外部体素永远不会共享同一个面、边或者顶点。右边是 6-分离体素化（6-separating voxelization），其中边缘和角可以在内部体素和外部体素之间进行共享。[1594]

随着现代 GPU 提供的新功能，使得更加有效的体素化方法成为可能。Schwarz、Seidel [1594, 1595]以及 Pantaleoni [1350]介绍了使用计算着色器进行构建的体素化系统，该系统提供了直接构建 SVO 的能力。Crassin 和 Green [306, 307]描述了他们用于规则网格体素化的开源系统，该系统利用了从 OpenGL 4.2 开始提供的图像加载/存储操作。这些操作允许对纹理内存进行随机读取和随机写入。通过使用保守光栅化（章节 23.1.2）来确定与一个体素重叠的所有三角形，他们的算法可以高效地计算体素的占用率，以及平均颜色和法线等信息。他们也可以用这种方法来自上而下地构建一个 SVO，并且在向下构建的过程中，只对非空节点进行体素化，然后再使用自下而上的过滤 mipmap 来对结构进行填充。Schwarz [1595]给出了光栅化和计算内核体素化系统的实现细节，并解释了各个系统的特点。Rauwendaal 和 Bailey [1466]构建了一个混合系统，并提供了源代码。他们提供了并行体素化方案的性能分析，并详细说明了如何正确使用保守光栅化从而避免误报。如果可以接受少量误差的话，Takeshige [1737]讨论了 MSAA 如何能够成为保守光栅化的一种可行替代方案。Baert 等人[87]提出了一种创建 SVO 的算法，该算法可以高效地基于核外运行（out-of-core），即可以在不需要将整个模型驻留在内存的情况下，以一个较高的精度来对场景进行体素化。

由于场景体素化需要进行大量的处理，因此动态物体（移动或者动画）对于基于体素的系统而言是一个很大的挑战。Gaitatzes 和 Papaioannou [510]通过对场景的体素表示进行渐进式（progressively）更新来解决这一任务。他们使用场景相机的渲染结果和生成的任何阴影贴图，来对体素数据进行清空和设置。体素会根据深度缓冲进行测试，从而清除那些比已记录的 z-depth 更加接近的体素。然后将缓冲区中的深度位置视为一组点，并将其转换到世界空间中。如果之前没有标记的话，则确定并设置这些点所对应的体素。这种清除和设置的过程依赖于相机的观察视角，这意味着当前相机没有看到的场景部分实际上是未知的，这可能会导致一些错误的发生。然而，这种快速的近似方法，使得计算基于体素的全局光照效果（详见章节 11.5.6）能够在动态环境中以交互速率进行执行。

13.10.4 体素的渲染

体素数据存储在一个三维数组中，它也可以被认为是存储在一个三维纹理中。这些体素数据可以使用多种方式进行渲染。在下一章中，我们将讨论半透明体素数据的可视

化方法（例如雾），或者是用于检查数据集的切片平面（例如超声图像）。而在本小节中，我们将专注于渲染代表实体物体的体素数据。

想象一下最简单的体积表示方法，其中每个体素都包含了一个 bit，用于记录该体素是位于物体内部还是位于物体外部。有一些常见的方法可以显示这些数据[1094]。一种方法是直接对体素进行光线投射[752, 753, 1908]，从而确定每个立方体最近的碰撞表面。另一种技术则是将体素立方体转换为一组多边形。虽然使用网格来进行渲染的速度会很快，但是这会在体素化期间会产生额外的成本，并且这种方法最适合于静态的体素。如果每个体素的立方体都是不透明的话，那么我们可以删除两个立方体之间相邻的正方形面，因为立方体之间的共享正方形面是不可见的。执行完这个过程之后，会给我们留下一个方形的外壳，但是外壳内部则是空心的。还可以使用一些简化技术（章节 16.5）来进一步减少多边形的数量。这个过程如图 13.31 所示。

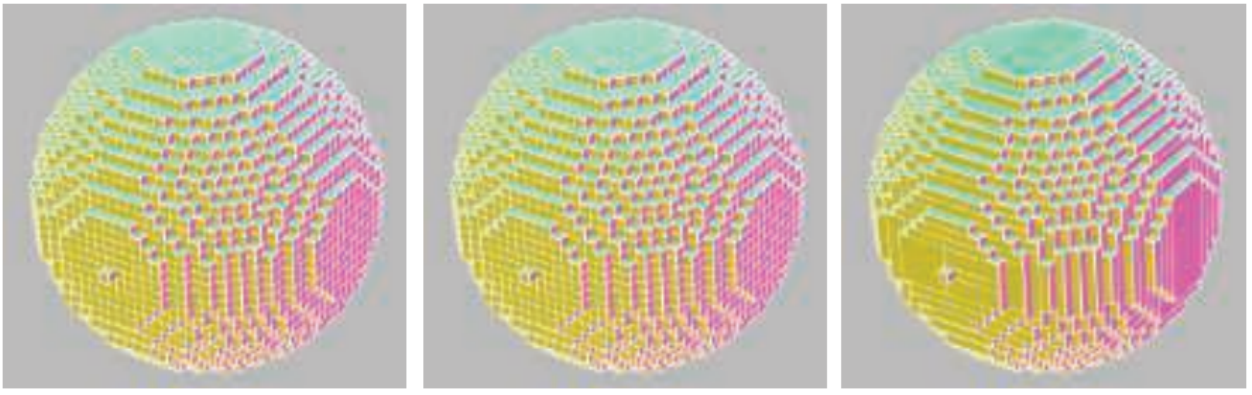


图 13.31：立方体剔除。左侧：这个包含 17074 个体素的实心球体，由 102444 个四边形面片组成，其中每个体素包含 6 个四边形面片。中间：相邻实体体素之间的共享的两个四边形面会被移除，使得四边形面片的数量减少到 4770 个。二者的外观是完全一样的，因为构成外壳的四边形面片没有受到影响。右边：使用一个快速贪心算法，来将较小的四边形面片合并成更大的四边形面片，最终使用了 2100 个四边形来表现这个球体。[1094]

对于表示曲面的体素而言，这组立方体面的着色是不太令人信服的。对立方体进行着色的一个常见替代方案是，使用诸如移动立方体（marching cube）之类的算法来创建一个更加平滑的网格表面[558, 1077]。这个过程被称为表面提取（surface extraction）或者多边形化（polygonalization，简写为 polygonization）。在算法中，我们不再将每个体素都视为一个 box，而是将其视为一个点样本。然后我们可以使用 8 个相邻的样本，以 $2 \times 2 \times 2$ 的模式来构建角，从而形成一个立方体。这 8 个角的状态可以定义一个穿过立方体的表面。例如：如果立方体的顶部四个角位于物体外部，而底部四个角则位于物体内部，那么我们可以进行一个合理的猜测，这个表面很可能就是一个将立方体分成两半的水平正方形。又例如：一个角位于物体外部，而其余的角位于物体内部，这会形成一个三角形表面，这个三角形的顶点分别位于连接

外部角的三条立方体边的中点处，如图 13.32 所示。这个将一组立方体角，转换为相应多边形网格的过程是十分高效的，因为 8 个角的 bit 可以被转换为一个 0–255 的索引，使用这个索引来访问一个表格，这个表格指定了每种可能配置模式的三角形数量和位置。

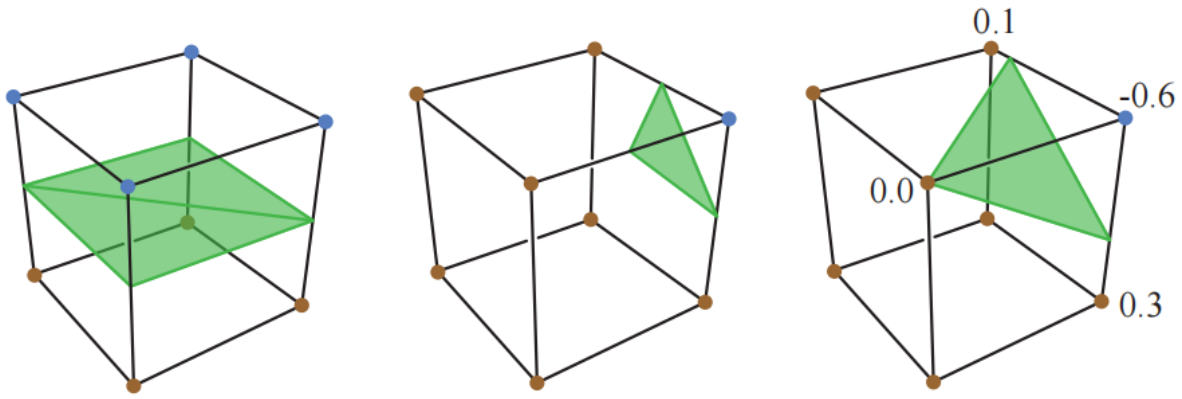


图 13.32：移动立方体算法。左侧：底部四个角是物体内部的体素中心，因此在底部四个角和顶部四个角之间，形成了一个包含两个三角形的水平正方形。中间：一个角位于物体外部，因此形成了一个三角形。右侧：如果角上存储了符号距离值的话，那么我们可以对三角形的顶点进行插值，使该三角形的顶点位于每条边的 0.0 处。请注意，共享同一条边的立方体，将在这条边的相同位置上创建一个顶点，从而确保表面不会出现裂缝。

其他渲染体素的方法，例如水平集（level set）[636]，则更加适合平滑的曲面。想象一下，每个体素中都存储了该空间位置到物体表面的距离，正值代表物体内部，负值表示物体外部。我们可以使用这些数据来调整网格顶点的位置，从而更加精确地表示曲面，如图 13.32 的右侧所示。或者我们可以直接对等值为 0 的水平集进行光线追踪，这种技术被称为水平集渲染（level-set rendering）[1249]。这种算法尤其擅长表现曲面模型的表面和法线，而不需要任何额外的体素属性。

表示密度差异的体素数据，可以通过决定构成表面的方式来以不同的方式进行可视化。例如：某些给定的密度可以很好地代表肾脏组织，而另一些密度则可以代表肾结石。选择其中一个密度值就定义了一个等值面（isosurface），即一组具有相同值的位置。这个值应当能够进行修改，这对于科学可视化尤其有用。对任何等值面直接进行光线追踪，这实际上就是一种广义形式的水平集光线追踪，其中的目标值为零。或者还可以将等值面提取出来，并将其转换为多边形网格模型。

2008 年，Olick [1323]进行了一个具有影响力的演讲，这个演讲的主题是如何使用光线投射来直接渲染稀疏体素数据，这个演讲启发了进一步的研究工作。针对规则体素的射线检测非常适合 GPU 进行实现，并且可以在交互帧率下完成。有许多研究人员对这个渲染领域进行了探索。有关这个主题的介绍，你可以从 Crassin 的博士论文

[304]和 SIGGRAPH 演讲[308]开始，其中涵盖了各种基于体素方法的优点。Crassin 通过使用锥形追踪（cone tracing）来利用体素数据类似于 mipmap 的性质，其一般想法是利用体素数据的规则性和定义良好的局部性属性，来定义几何属性和着色属性的预过滤方案，从而允许使用线性的滤波器。在场景中追踪一根光线，这个光线可以从其起始点，在一个锥形区域内来收集一个近似的可见性。当光线在空间中移动的时候，其兴趣半径会逐渐增加，这意味着会在更高的层次上对体素结构进行采样，这个过程类似于当单个像素包含多个纹素时，mipmap 会在更高层级上进行采样。这种类型的采样方法可以快速计算软阴影效果和景深效果，因为这些效果可以被分解为锥形追踪问题。这种区域采样的思想在其他处理过程中也很有价值，例如抗锯齿和正确过滤可变的表面法线。Heitz 和 Neyret [706]对之前的工作进行了介绍，并提出了一种用于锥形追踪的新数据结构，它可以改善可见性计算的结果。Kasyan [865]对面光源使用了体素锥形追踪，并对误差来源进行了讨论，图 13.33 中给出了一个对比，最终效果详见图 7.33。使用锥形追踪来计算全局光照效果的方法，我们已经在章节 11.5.7 中进行了讨论。

图 13.33：锥形追踪所生成的阴影。上图：在 Maya 中对球形面光源进行 20 秒光线追踪的渲染结果。下图：同一场景的体素化和锥形追踪耗时约 20 毫秒，其中模型使用多边形进行渲染，而其体素化版本则用于阴影计算。[865]

最近的研究趋势是在 GPU 上探索八叉树以外的数据结构。八叉树的一个主要缺点在于，在其上进行光线追踪等操作，需要进行大量的树结构遍历操作，因此需要存储大量的中间节点。Hoetzlein [752]指出，基于 VDB 树（一种网格层次结构）的 GPU 光线追踪，可以比八叉树获得更加显著的性能提升，并且更加适合体素数据的动态变化。Fogal 等人[477]证明，通过使用索引表而不是八叉树，可以使用一个两 pass 的方法来实时渲染一个较大的空间。其中第一个 pass 用于识别可见的子区域（块），以及以流式方式获取来自磁盘中对应区域中的数据。第二次 pass 对当前驻留在内存中的区域进行渲染。Beyer 等人[138]对大规模体积渲染技术进行了全面的综述。

13.10.5 其他主题

表面提取技术通常用于隐式表面（章节 17.3）的可视化。有不同形式的基本算法可以用来形成网格，并且在这些算法在形成网格的时候有一些微妙之处。例如：如果发现一个立方体的所有角都位于物体内部，那么这些角应当在多边形网格中被连接在一起，还是彼此保持独立？你可以阅读 de Araujo 等人[67]的文章来了解有关隐式曲面的多边形化技术。Austin [85]研究了各种通用多边形化方案的优缺点，并发现方形的移动正方形（cubical marching square）技术具有最理想的特性。

在使用光线投射进行渲染的时候，除了将体素进行完全多边形化之外，还有一些其他的解决方案。例如：Laine 和 Karras [963]将一组平行平面附加到每个近似表面的体素上，然后使用后处理模糊来掩盖体素之间的不连续性。Heitz 和 Neyret [706]以一种线性可滤波的表示方法来访问符号距离，这样允许重建出平面方程，并可以确定任意空间位置和分辨率在给定方向上的覆盖范围。

Eisemann 和 Decoret [409]展示了如何使用体素表示方法来执行深度阴影映射（章节 7.8），并将其用于半透明重叠表面的阴影投射。正如 Kampe，Sintorn 等人[850, 1647]所表明的那样，与为每个光源都生成一个阴影贴图相比，体素化场景的另一个优点在于，所有光源的阴影光线都可以使用这同一个表示来进行测试。与直接可见的表面渲染效果相比，人眼对于阴影和间接光照等次要效果中的小错误更加包容，并且这些任务所需要的体素数据要少得多。当只追踪一个体素的占用情况时，许多稀疏体素节点之间可能存在极高的自相似性（self-similarity）[849, 1817]。例如：一堵墙会包含一组体素，这些体素在若干个级别上都是相同的。这意味着树中的各种节点和整个子树都是相同的，因此我们可以使用单个实例来代表这些节点，并将它们存储在一个有向无环图（directed acyclic graph）中（章节 19.1.5）。这样做可以大大减少每个体素结构所占据的存储空间。

补充阅读和资源

在 Szeliski 的《Computer Vision: Algorithms and Applications》一书[1729]中讨论了基于图像的渲染、光场、计算摄影和许多其他主题。请访问我们的网站 realtimerendering.com，来获取这本值得阅读的书，网站上有免费的电子版链接。Weier 等人[1864]在他们的 SOTA 报告中，讨论了利用视觉感知系统的局限性的各种加速技术。Dietrich 等人[352]在他们关于大规模模型渲染报告的侧边栏中，对基于图像的技术进行了综述。

在本章节中，我们只涉及了一些用于模拟自然现象的图像、粒子和其他非多边形方法。有关更多的示例和细节，请参阅相关参考文献。其中一些文章讨论了各种各样的技术，Beacco 等人[122]对人群渲染技术进行了调查，并讨论了基于 impostor、LOD 方法和其他许多技术的变种方法。Gjøl 和 Svendsen 的演示[539]展示了基于图像的采样和过滤技术，这些技术可以用于广泛的效果中，包括泛光、镜头光晕、水体效果、反射、雾、火和烟等。

Chapter 14 Volumetric and Translucency Rendering 体积与半透明渲染

> Leonardo Da Vinci——“Those you wish should look farthest away you must make proportionately bluer；thus，if one is to be fifive times as distant，make it fifive times bluer.”

> 列奥纳多·达·芬奇——“那些你希望看得最远的事物，你必须按比例把它画得更蓝；因此，如果一个人比你远五倍，就要把他画得蓝五倍。”（意大利文艺复兴时期画家、自然科学家；1452—1519）

参与介质（participating media）是一个术语，它用来描述充满颗粒的体积。同时顾名思义，它们是参与光线传输的介质，换句话说，它们通过散射作用或者吸收作用，来影响穿过它们的光线。在渲染虚拟世界的时候，我们通常关注的是简单或者复杂的实体表面，这些表面看起来是不透明的，因为它们是密度很大的参与介质，会对光线进行反射，例如通常使用 BRDF 进行建模的电介质或者金属。生活中充满了密度较小的参与介质，例如水、雾、蒸汽，甚至是由稀疏分子所组成的空气。根据组成成分的不同，介质与光线的相互作用也会有所不同，光线穿过介质并被介质中的粒子所反射，这类事件通常会被称为光线散射（light scattering）。介质中粒子的密度可以是同质的（homogeneous，均匀的），例如空气或者水；也可以是异质的（heterogeneous，不均匀的，随空间位置而变化），例如云或者蒸汽。一些致密的材料通常呈现为固体表面，表现出较高水平的光线散射，例如皮肤或者蜡烛的蜡。如章节 9.1 中所介绍的那样，漫反射表面着色模型是光线在微观层面发生散射的结果。任何事物都会发生散射（Everything is scattering）。

14.1 光线散射理论

在本小节中，我们将介绍光线在参与介质中的模拟和渲染。我们曾在章节 9.1.1 和章节 9.1.2 中讨论过光线的散射和吸收，而本小节将讨论的是对这些物理现象的定量处理。许多作者在多重散射路径追踪的背景下描述了辐射的传输方程[479, 743, 818, 1413]。而在本小节中，我们将专注于单次散射（single scattering），并对其工作原理建立起一个良好直觉。单次散射只会考虑光线在粒子（构成参与介质的粒子）上的

一次反射；而多重散射则会对每条光路上的多次弹射进行追踪，因此要复杂得多[243, 479]。使用多重散射和不使用多重散射的对比结果，可以在图 14.51 中看到。表 14.1 给出了用于表示散射方程中参与介质属性的符号和单位。请注意，本章节中的许多物理量，例如 $\sigma_a, \sigma_s, \sigma_t, p, \rho, v, T_r$ ，都与光线的波长有关，这意味着在实践中它们都是 RGB 量。

Symbol	Description	Unit
σ_a	Absorption coefficient	m^{-1}
σ_s	Scattering coefficient	m^{-1}
σ_t	Extinction coefficient	m^{-1}
ρ	Albedo	unitless
p	Phase function	sr^{-1}

表 14.1：用于散射和参与介质的符号。这些参数都可以依赖于波长（即 RGB），从而实现彩色光线的吸收或者散射。其中相位函数 p 的单位是立体弧度（steradians，简写为 sr）的逆（详见章节 8.1.1）。

14.1.1 参与介质材质

有四种类型的事件可以影响沿着光线进行传播并穿过介质的 radiance。图 14.1 中给出了这些函数的图示说明：

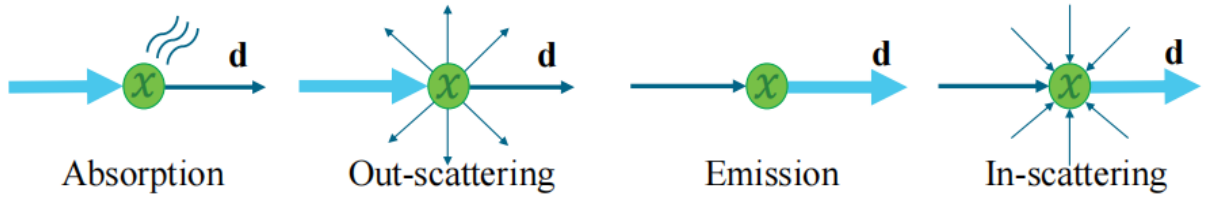


图 14.1：不同类型的相互作用事件，这些事件会改变参与介质中沿方向 $\mathbf{d}$ 进行传播的 radiance。

可以将其总结为：

- **吸收 Absorption**（σ_a 的函数）——光子被介质吸收并转化为热量或者其他形式的能量。
- **外散射 Out-scattering**（σ_s 的函数）——光子被介质中的粒子反弹，并从介质中散射出去。该事件的发生概率，取决于描述光线反射方向分布的相位函数 p 。

- **发射 Emission**——当介质达到较高温度时，例如火焰的黑体辐射（black-body radiation），可以从介质中发射光线。有关发射的更多细节，详见 Fong 等人的课程讲义 [479]。
- **内散射 In-scattering**（σ_s 的函数）——来自任何方向的光子，在被介质粒子反弹之后都可以散射到当前的光路中，并对最终的 radiance 产生一定的贡献。在给定方向上内散射进来的光量，也取决于该光线方向的相位函数 p 。

综上所述，在一个路径中加入光子是内散射 σ_s 和发射的函数；而移除光子则是消光系数（extinction） $\sigma_t = \sigma_a + \sigma_s$ 的函数，它代表了吸收和外散射。由辐射传输方程（radiative transfer equation）可知，系数集合代表了位置 $\mathbf{x}$ 和方向 $\mathbf{v}$ 上的 radiance 相对于 $L(\mathbf{x}, \mathbf{v})$ 的导数。这也就是为什么这些系数的值都在范围 $[0, +\infty]$ 内，详见 Fong 等人[479]的说明。其中的散射系数和吸收系数决定了该介质的反照率 ρ ，其定义如下：

$$\rho = \frac{\sigma_s}{\sigma_s + \sigma_a} = \frac{\sigma_s}{\sigma_t} \tag{14.1}$$

反照率 ρ 代表了在可见光谱范围内，介质中散射作用相对于吸收作用的重要程度，即介质的整体反射率。 ρ 的值位于 $[0, 1]$ 范围内，接近 0 意味着大部分光线都会被吸收，这会产生一种浑浊的介质，例如黑色的排烟。接近 1 意味着大部分光线都会被散射而不是被吸收，这会产生一种更加明亮的介质，例如空气、云或者地球的大气层。

正如章节 9.1.2 中所描述的，介质的外观表现是其散射特性和吸收特性的组合结果。现实世界中很多参与介质的系数，都已经进行了测量和发布[1258]。例如：牛奶具有很高的散射值，因此会产生浑浊且不透明的外观；并且由于牛奶具有较高的反照率 $\rho > 0.999$ ，因此牛奶看起来也是白色的。 另外一个例子：红酒的特点是几乎没有散射作用，但是具有较高的吸收作用，这使得它具有半透明且彩色的外观。如图 14.2 中渲染的液体，并将其与图 9.8 中拍摄的真实液体进行比较。

图 14.2：葡萄酒和牛奶在不同浓度下具有不同的吸收特性和散射特性，图中展示了不同浓度下葡萄酒和牛奶的渲染结果。[1258]

这些性质和事件都与光线的波长相关。这种相关性意味着，在给定的介质中，不同频率的光线可能会以不同的概率被吸收或者倍散射。从理论上来说，为了从根本上解决这个问题，我们应当在渲染过程中使用光谱值。但是为了效率起见，因此在实时渲染中（在离线渲染中也有少数例外情况[660]）我们依然会使用 RGB 值。在可能的情况下，应当使用颜色匹配函数（color-matching function）（详见章节 8.1.3）来从光谱数据中预先计算出诸如 σ_a 和 σ_s 等物理量所对应的 RGB 值。

在之前的章节中，由于并不存在参与介质，因此我们可以假设进入相机的 radiance 与离开最近表面的 radiance 相同。更准确地说，我们假设 $L_i(\mathbf{c}, -\mathbf{v}) = L_o(\mathbf{p}, \mathbf{v})$，其中 $\mathbf{c}$ 是相机位置，$\mathbf{p}$ 是与最近表面与观察光线的交点，$\mathbf{v}$ 是指向从点 $\mathbf{p}$ 指向点 $\mathbf{c}$ 的单位观察向量。

一旦引入了参与介质，上述假设就不再成立了，我们需要考虑沿着观察光线的 radiance 变化。作为一个例子，我们现在将描述在评估来自精确光源（由单个无穷小点所表示的光源，详见章节 9.4）的散射光线时所涉及的计算：

$$L_i(\mathbf{c}, -\mathbf{v}) = T_r(\mathbf{c}, \mathbf{p}) L_o(\mathbf{p}, \mathbf{v}) + \int_{t=0}^{\|\mathbf{p}-\mathbf{c}\|} T_r(\mathbf{c}, \mathbf{c}-\mathbf{v}t) L_{\text{scat}}(\mathbf{c}-\mathbf{v}t, \mathbf{v}) \sigma_s \, dt \tag{14.2}$$

其中 $T_r(\mathbf{c}, \mathbf{x})$ 是给定点 $\mathbf{x}$ 与相机位置 $\mathbf{c}$ 之间的透光率（章节 14.1.2）；$L_{\text{scat}}(\mathbf{x}, \mathbf{v})$ 是沿着观察射线，在给定点 $\mathbf{x}$ 处散射的光线（章节 14.1.3）。这个方程中的各个计算

部分如图 14.3 所示，我们将在接下来的几个小节中对其进行解释。关于如何从辐射传输方程中推导出方程 14.2 的更多细节，可以在 Fong 等人[479]的课程说明中找到。

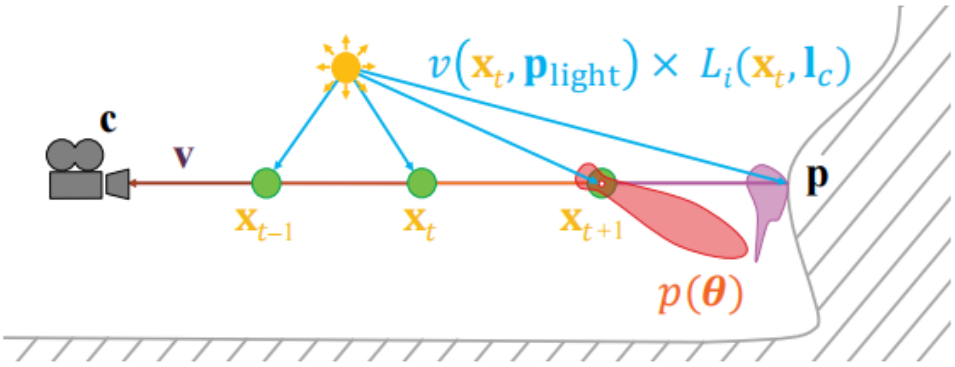


图 14.3：来自精确光源的单次散射积分的示意图。沿着观察射线的采样点被显示为绿色，采样点的相位函数被显示为红色，不透明表面 S 的 BRDF 被显示为紫色。其中 $\mathbf{l}_c$ 是指向精确光源中心的方向向量，$\mathbf{p}_{light}$ 是光源的位置，p 是相位函数，函数 v 是可见性项。

14.1.2 透光率

透光率 T_r 代表了光线在一定距离内能够通过介质的比例，其数学定义为：

$$T_r\left(\mathbf{x}_a, \mathbf{x}_b\right) = e^{-\tau}, \quad \tau = \int_{\mathbf{x}=\mathbf{x}_a}^{\mathbf{x}_b} \sigma_t(\mathbf{x}) \|d\mathbf{x}\| \tag{14.3}$$

这种关系也被称为 Beer–Lambert 定律。方程中的光学深度 τ 是没有单位的，它代表了光线的衰减量。消光（extinction）系数或者传播距离越大，光学深度 τ 也就越大，反过来则说明通过介质的光线就越少。光学深度 $\tau = 1$ 将会移除大约 60% 的光线。例如：如果在 RGB 中 $\sigma_t = (0.5, 1, 2)$，光线能通过的深度 d 为 1 米，则透光率 $T_r = e^{-d\sigma_t} \approx (0.61, 0.37, 0.14)$，如图 14.4 所示。需要对 (i) 不透明表面的 radiance $L_o(\mathbf{p}, \mathbf{v})$；$(ii)$ 内散射事件所产生的 radiance $L_{\text{scat}}(\mathbf{x}, \mathbf{v})$；以及 (iii) 从散射事件到光源的每条路径都应用这个透光率 T_r。从视觉上看，(i) 将导致一些雾状的表面遮挡；(ii) 将导致散射光线的遮挡，从而提供一种有关介质厚度的视觉暗示（如图 14.6 所示）；(iii) 将导致参与介质的体积自阴影现象（如图 14.5 所示）。由于 $\sigma_t = \sigma_a + \sigma_s$，因此透光率可能会同时受到吸收和外散射的影响

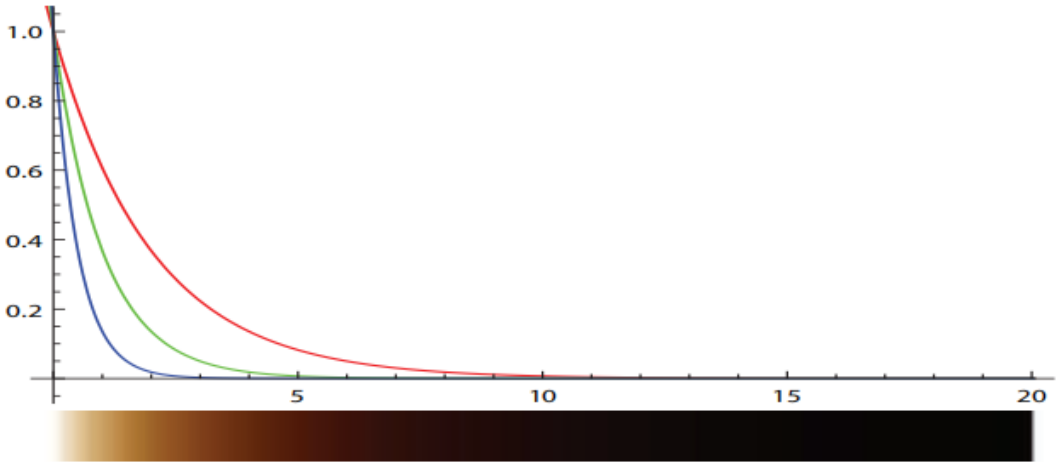


图 14.4：当 $\sigma_t = (0.5, 1.0, 2.0)$ 不变时，透光率是一个关于深度的函数。正如预期的那样，红色分量的消光系数越低，因此最终透射出的红色也就越多。

14.1.3 散射事件

对于场景中给定位置 $\mathbf{x}$ 和方向 $\mathbf{v}$ 的精确光源，对其内散射事件进行积分可以这样做：

$$L_{\text{scat}}(\mathbf{x}, \mathbf{v}) = \pi \sum_{i=1}^{n} p(\mathbf{v}, \mathbf{l}_{c_i})\, v\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right) c_{\text{light}_i}\left(\left\|\mathbf{x} - \mathbf{p}_{\text{light}_i}\right\|\right) \quad (14.4)$$

方程中 n 是光源的数量，$p()$ 是相位函数，$v()$ 是可见度函数，$\mathbf{l}_{c_i}$ 是第 i 个光源的方向向量，$\mathbf{p}_{\text{light}_i}$ 是第 i 个光源的位置。此外 c_{light_i} 是第 i 个光源的 radiance 到位置 $\mathbf{x}$ 距离的函数，这里使用了章节 9.4 中所定义，以及章节 5.2.2 中的平方反比衰减函数。可见性函数 $v\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right)$ 代表了从光源位置 $\mathbf{p}_{\text{light}_i}$ 处发出的光线最终到达位置 $\mathbf{x}$ 的比例，其数学形式如下：

$$v\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right) = \text{shadowMap}\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right) \cdot \text{vol Shad}\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right) \quad (14.5)$$

其中方程中的 $\text{volShad}\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right) = T_r\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right)$。在实时渲染中，阴影是由两种类型的遮挡所产生的：不透明遮挡和体积遮挡。计算来自不透明物体的阴影（shadowMap）通常会使用阴影映射或者第 7 章中所介绍的其他技术。

方程 14.5 中的体积阴影项 $\text{volShad}\left(\mathbf{x}, \mathbf{p}_{\text{light}_i}\right)$ 代表了从光源位置 $\mathbf{p}_{\text{light}_i}$ 到采样点 $\mathbf{x}$ 的透光率，其取值范围为 $[0, 1]$。由体积所产生的遮挡效果是体渲染中的重要组成部分，其中体积元素可以产生自阴影，或者是在其他场景元素上投射阴影，如图 14.5 所示。这个结果通常是这样获得的：沿着从眼睛出发，到达第一个表面的主光线

（primary ray）进行光线步进；然后再沿着从各个样本点到每个光源的二次光线（secondary ray）进行光线步进。这里所提到的“光线步进（ray marching）”是指，使用 n 个样本对两点之间的路径进行采样，从而对沿途的散射光线和透光率进行积分。有关这种采样方法的更多细节，详见章节 6.8.1，在该章节所使用的光线步进是用来渲染高度场的。光线步进的思想与三维体积有点类似，每条光线会一步一步地前进，并在沿途的每个样本点上对体积材质或者光照进行采样。如图 14.3 所示，主光线使用红色进行表示，二次阴影光线使用蓝色进行表示，而绿色则代表了主光线上的样本点。有许多其他出版物也对光线步进方法进行了详细描述 [479, 1450, 1908]。

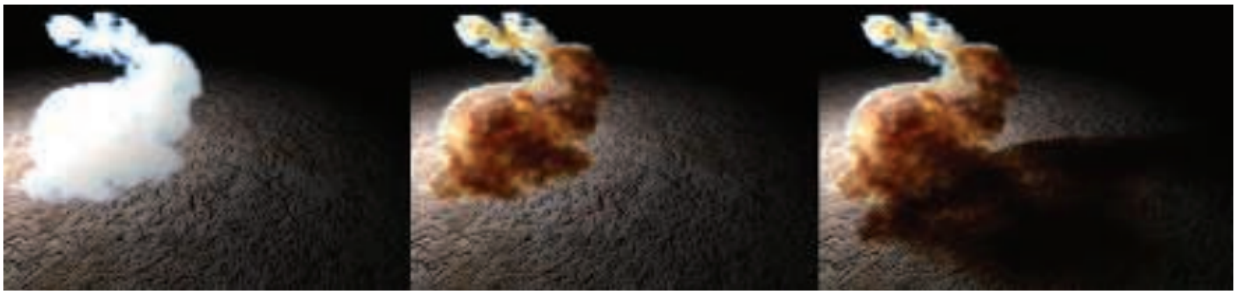


图 14.5：体积阴影示例，其中的 Stanford 兔子由参与介质所构成[744]。左：没有体积自阴影；中：有体积自阴影；右：将阴影投射在其他场景元素上。

光线步进的时间复杂度为 $O(n^2)$，其中 n 是每条路径上的样本数量，因此光线步进的计算开销上涨得会很快。为了在质量和性能之间的进行权衡（trade-off），可以使用一些特定的体积阴影表示技术来存储光线出射方向上的透射率。这些技术将在本章节剩余小节中的适当部分进行解释。

为了直观地了解光线在参与介质中的散射（scattering）行为和消光（extinction）行为，我们假设 $\sigma_s = (0.5, 1, 2)$ 和 $\sigma_a = (0, 0, 0)$。对于介质中的较短光路，内散射事件将会占据主导地位（相对于消光事件）；而在我们所假设的这个例子下，外散射事件将会占据主导地位，由于深度较小，因此 $T_r \approx 1$。这个材质将会呈现蓝色，因为蓝色通道的 σ_s 值最高。光线在介质中穿透得越深，就会由于消光作用，最终通过的光子数量就越少。在这种情况下，来自消光作用的透射颜色将开始占主导地位。这可以用 $\sigma_t = \sigma_s$ 来进行解释，因为 $\sigma_a = (0, 0, 0)$。因此，与散射光线随光学深度 $d\sigma_s$ 的线性积分（方程 14.2）相比，透光率 $T_r = e^{-d\sigma_t}$ 的下降速度要快得多。对应在本例中，由于红色通道在通过介质时被消光的概率较小（因为该通道的 σ_t 值最小），因此红光将会占据主导地位。这种现象如图 14.6 所示，这也正是大气层和天空中所发生的情况。当太阳角度较高时（例如：直穿大气层的短光路，该光路垂直于地面），蓝色光线会发生更多的散射，从而使得天空呈现自然的蓝色。然而，当太阳

位于地平线角度时，太阳光线在地球大气中具有一条很长的光线路径，更多的红光会被透射，因此天空会显得更红。而在这两种情况之间，便是我们都知道的日出和日落的过渡。有关大气材质组成的更多细节，详见章节 14.4.1。有关这个效果的另一个例子，详见图 9.6 右侧的乳白色玻璃。

图 14.6：Stanford 龙，从左到右的介质密度越来越大，分别为 0.1、1.0 和 10.0，其中 $\sigma_s = (0.5, 1.0, 2.0)$ 。

14.1.4 相位函数

参与介质由半径不同的粒子所组成。相对于光线的向前传播方向，这些粒子大小的分布将会影响光线在给定方向上发生散射的概率。章节 9.1 对这种行为背后的物理原理进行了解释。

在评估内散射事件的时候，可以使用一个相位函数，来在宏观层面上描述散射方向的概率和分布，如方程 14.4 所示。图 14.7 对此进行了图示，其中相位函数使用红色进行表示，相位函数有一个参数 θ ，它代表光线的向前传播路径（蓝色）与外散射方向 $\mathbf{v}$ （绿色）之间的夹角。注意在这个相位函数的例子中存在两个主要的波瓣：一个与光路方向相反的、较小的后向散射波瓣，一个与光路方向相同的、较大的前向散射波瓣。相机 B 位于较大的前向散射波瓣的方向上，因此与相机 A 相比，它将会接收到更多的散射 radiance。为了保证能量守恒，相位函数在单位球体上的积分必须为 1。

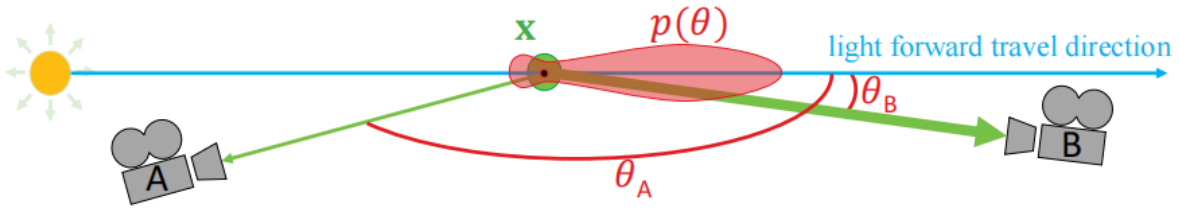


图 14.7：相位函数（红色），以及不同参数 θ 对散射光线（绿色）的影响。

相位函数会根据到达该点的定向 radiance 信息，来相应改变该点上的内散射。最简单的相位函数是各向同性的：即光线将在各个方向上发生均匀散射。这种完美但不符合实际的行为可以使用下面的方程进行描述：

$$p(\theta) = \frac{1}{4\pi} \tag{14.6}$$

其中参数 θ 是入射光线与外散射方向之间的夹角， 4π 是单位球的表面积大小。

基于物理的相位函数依赖于粒子的相对尺寸 s_p ，即：

$$s_p = \frac{2\pi r}{\lambda} \tag{14.7}$$

其中 r 是粒子的半径， λ 的光线是波长[743]。根据相对尺寸 s_p 大小的不同，会发生以下不同类型的散射：

- 当 $s_p \ll 1$ 时，发生瑞利散射（Rayleigh scattering），例如空气。
- 当 $s_p \approx 1$ 时，发生米氏散射（Mie scattering）。
- 当 $s_p \gg 1$ 时，发生几何散射（geometric scattering）。

瑞利散射

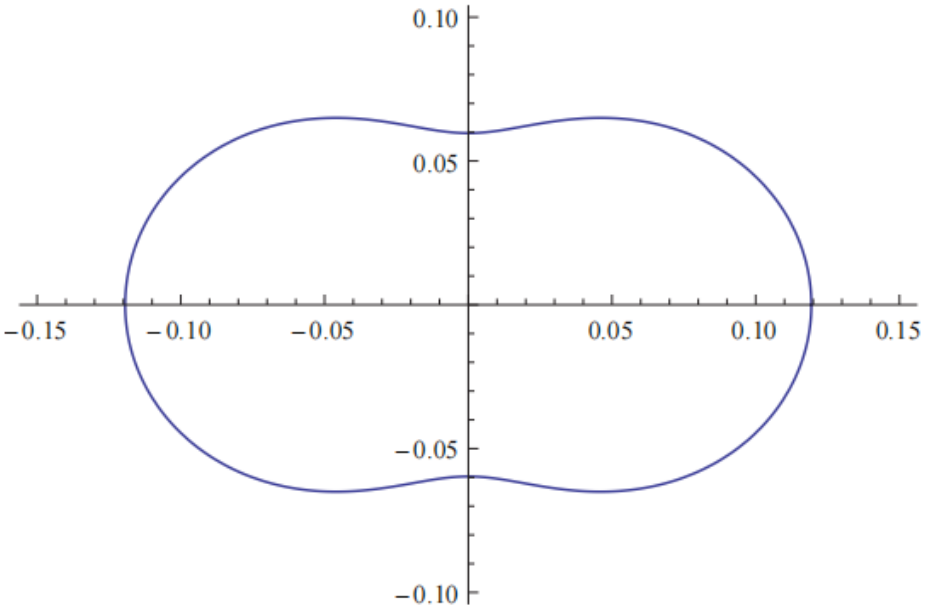


图 14.8：以 θ 为参数的瑞利相位函数的极坐标图。光线从左侧水平入射，图中展示了相对于角度 θ 的相对强度，从 x 轴逆时针方向进行测量。在瑞利散射中，前向散射和

后向散射的概率是一样的。

瑞利（Lord Rayleigh, 1842–1919）提出了空气分子散射光线的术语。在很多应用领域中，这些表达式被用于描述地球大气中的光线散射现象。这个相位函数包含两个波瓣（如图 14.8 所示），即相对于光线方向的后向散射（backward scattering）与前向散射（forward scattering）。这个函数只有一个参数 θ，θ 是入射光线和外散射方向之间的夹角。该相位函数的数学形式如下：

$$p(\theta) = \frac{3}{16\pi}\left(1 + \cos^2\theta\right) \tag{14.8}$$

瑞利散射与光线的波长高度相关。如果将瑞利散射视作波长 λ 的函数的话，那么瑞利散射的散射系数 σ_s 将与波长 λ 的四次方成反比：

$$\sigma_s(\lambda) \propto \frac{1}{\lambda^4} \tag{14.9}$$

这种关系意味着，相较于波长较长的红光而言，波长较短的蓝光或者紫光会更容易被散射。方程 14.9 的光谱分布可以通过光谱颜色匹配函数（章节 8.1.3）来将其转换为 RGB 值，即 $\sigma_s = (0.490, 1.017, 2.339)$。这个值被归一化为亮度 1，并且应当根据所需要的散射强度来对其进行缩放。有关蓝光在大气中被更多地散射，从而产生的视觉效果将在章节 14.4.1 中进行解释。

米氏散射

米氏散射[776]是一种当粒子大小与光线波长大致相同时可以使用的模型。这种类型的散射与光线的波长无关。软件 MiePlot 可以用来模拟这种现象[996]。特定粒子尺寸的米氏相位函数通常是一个十分复杂的分布，它具有强烈且尖锐的定向波瓣，也就是说，相对于光子的运动方向，光子在某些特定方向上的散射概率会很高。计算这种用于体积着色的相位函数具有很高的计算成本，但幸运的是我们很少会使用到它。介质通常都会具有连续的粒子尺寸分布。对所有这些不同尺寸的米氏相位函数求平均值，便得到整个介质的平滑平均相位函数。因此，可以使用一些相对光滑的相位函数来表示米氏散射。

常用于描述米氏散射的一种相位函数是 Henyey–Greenstein （HG）相位函数，它最初的目的是用来模拟星际尘埃中的光线散射[721]。这个函数无法精确捕捉真实世界中每个散射的复杂行为，但是它可以很好地匹配相位函数的其中一个波瓣[1967]，即主散射方向上的波瓣。它可以用来表示任何烟、雾或者灰尘状的参与介质，这类介质通

常会表现出强烈的后向散射或者前向散射，从而在光源附近产生巨大的视觉光晕。常见的例子包括：雾中的聚光灯；太阳方向上的云，其边缘会呈现出强烈的银边效应（silver-lining effect）。

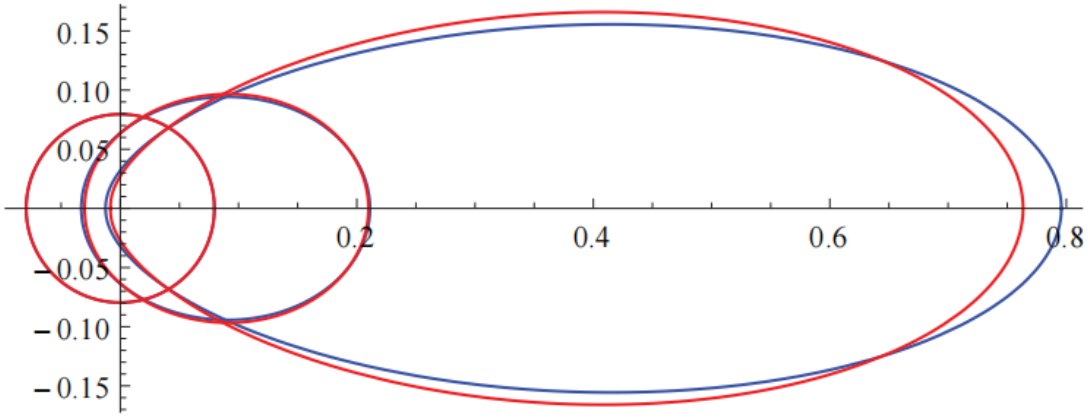


图 14.9：以 θ 为参数，Henyey-Greenstein 相位函数（蓝色）和 Schlick 近似相位函数（红色）的极坐标图。光线从左侧水平射入。参数 g 从 0 增加到 0.3 和 0.6，从而导致了一个较强的向右波瓣，这意味着光线将更多地散射到其前进路径上，即从左向右散射。

HG 相位函数可以表示比瑞利散射更加复杂的行为，其数学形式如下：

$$p_{hg}(\theta, g) = \frac{1 - g^2}{4\pi \left(1 + g^2 - 2g\cos\theta\right)^{1.5}} \tag{14.10}$$

这个相位函数可以生成各种形状，如图 14.9 所示。方程中参数 g 可以用来表示后向散射（$g < 0$）、各向同性散射（$g = 0$）或者前向散射（$g > 0$），其中参数 g 的范围在 $[-1, 1]$ 范围内。使用 HG 相位函数的散射结果如图 14.10 所示。

一种可以获得与 HG 相位函数相似结果，但是速度更快的方法是使用 Blasi 等人[157]提出的近似相位函数，这个相位函数通常会以第三作者的名字进行命名，即 Schlick 相位函数：

$$p(\theta, k) = \frac{1 - k^2}{4\pi(1 + k\cos\theta)^2}, \quad where \quad k \approx 1.55g - 0.55g^3 \tag{14.11}$$

方程 14.11 中不包含任何复杂的幂函数，仅仅只有一个平方函数，因此计算起来要快得多。为了将这个函数映射到原始 HG 相位函数上，需要从参数 g 中计算参数 k。而对于具有恒定 g 值的参与介质，只需要进行一次计算即可。在实际应用中，Schlick 相位函数是一个很好的节能近似（计算量更少，更环保），如图 14.9 所示。

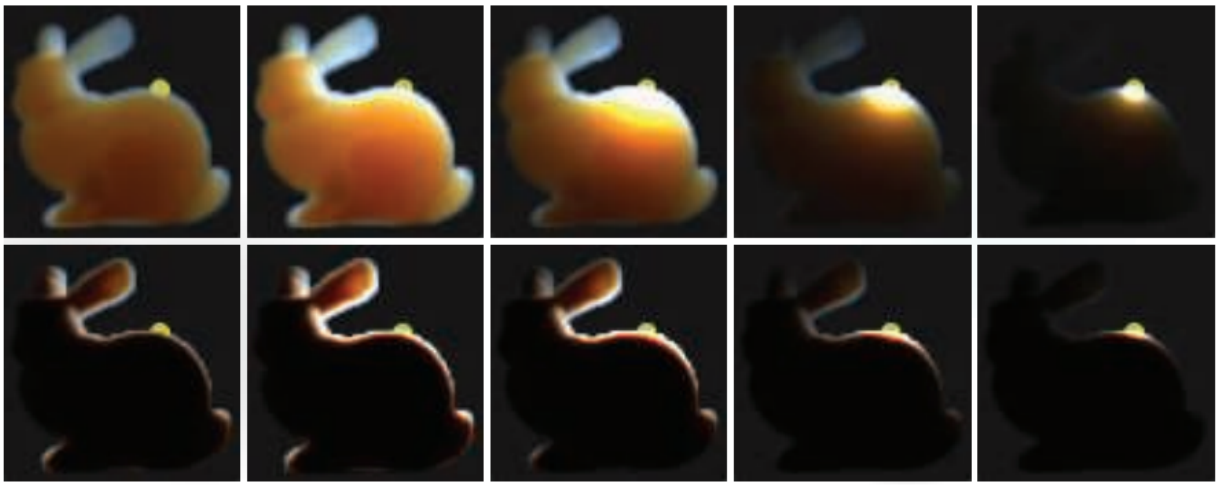


图 14.10：Stanford 兔子由参与介质构成，以表现 HG 相位函数所带来的影响，其中参数 g 从各向同性逐渐变化为强烈的前向散射。从左到右的参数 g 分别是：$g = 0.0, 0.5, 0.9, 0.99, 0.999$。第二行中所使用的参与介质密度，是第一行的 10 倍。

为了表达更加复杂的一般相位函数范围，也可以将多个 HG 或者 Schlick 相位函数进行混合。这种方法使得我们能够表达这样一个相位函数[743]，它同时具有强烈的前向散射波瓣和后向散射波瓣，类似于云的散射行为，详见章节 14.4.2 中的描述和说明。

几何散射

当粒子尺寸明显大于光线波长的时候，就会发生几何散射（geometric scattering）。在这种情况下，光线可以在每个粒子上发生折射和反射。这种行为可能需要一个更加复杂的散射相位函数才能在宏观层面上对其进行模拟。光线的偏振（polarization）也会影响到这种类型的散射现象，例如：现实生活中的一个常见例子就是彩虹效果，彩虹是由空气中水分粒子内部的光线反射所引起的，使得太阳光线在一个小视角（大约 3 度）内的反向散射中被分解成可见光谱。这种复杂的相位函数可以使用 MiePlot 软件进行模拟[996]。在章节 14.4.2 中就描述了这样一个相位函数。

14.2 特殊的体渲染

在本小节中，会介绍一些基本的、有限的体渲染算法。有些人甚至会说，这些算法都是老一套的把戏，它们往往会依赖于特定的模型。之所以会在这里依然介绍和使用它们，是因为这些算法仍然工作良好且表现良好。

14.2.1 大规模雾

雾可以近似地表示为一种基于深度的效果。其最基本的形式是，根据与相机之间的距离来对场景顶部的雾进行 alpha 混合，这种雾通常被称为深度雾（depth fog）。这种类型的效果是一种对观察者的视觉暗示。第一，它可以增加真实感和戏剧感，如图 14.11 所示。

图 14.11：使用雾来突出一种氛围。

第二，它是一个重要的深度暗示，可以帮助场景中的观察者来确定物体的位置大概有多远，如图 14.12 所示。第三，它可以作为遮挡剔除的一种形式，如果物体位于太远的位置时，它可能会被雾完全挡住，此时可以安全地跳过这些物体的渲染，从而提高应用程序的性能表现。

图 14.12：在游戏《战地 1》（DICE 的一款游戏）的关卡图像中，使用雾来展示游戏区域的复杂性。深度雾用于展示大尺度的自然风景。在右侧地面上还可以看到高度雾，展示了从山谷中升起的大量建筑。

一种表示雾量（amount of fog）的方法，是使用范围在 $[0,1]$ 中的 f 来表示透光率，当 $f=0.1$ 时，则代表有 10% 的背景表面是可见的。假设背景表面的输入颜色为 $\mathbf{c}_i$，雾的颜色为 $\mathbf{c}_f$，则最终颜色 $\mathbf{c}$ 为：

$$\mathbf{c}=f\mathbf{c}_i+(1-f)\mathbf{c}_f \tag{14.12}$$

可以使用许多不同的方法来评估 f 的值。随着深度线性增长的雾可以使用如下方程：

$$f=\frac{z_{\text{end}}-z_s}{z_{\text{end}}-z_{\text{start}}} \tag{14.13}$$

其中 z_{start} 和 z_{end} 是用户控制的参数，它们决定了雾在哪里开始，以及在哪里结束（这里的结束是指变得完全雾蒙蒙）；z_s 是从观察者到待计算雾表面的线性深度。关于如何计算雾的透光率，一种物理正确的方法是使其随距离呈指数级增长，即遵循透光率的 Beer–Lambert 定律（章节 14.1.2）。这种效果可以通过使用以下方程来进行实现：

$$f=e^{-d_f z_s} \tag{14.14}$$

其中标量 d_f 是控制雾密度的用户参数。这种传统的、大尺度的雾，是对光线在大气中的散射和吸收的一种粗略近似（章节 14.4.1），但是它仍然被如今的游戏所应用，并且效果很好，如图 14.12 所示。

这就是硬件雾在 OpenGL 和 DirectX API 中的暴露方式。对于移动设备等硬件上的简单情景而言，考虑使用这些模型来实现雾效是仍然具有价值的。如今的许多游戏都依赖于更加高级的后处理来实现大气效果，例如雾和光线散射。透视视图中的雾具有一个问题，即深度缓冲中的深度值是以一种非线性的方式进行计算的（章节 23.7）。我们需要使用逆投影矩阵，来将非线性的深度缓冲值转换会线性深度 z_s [1377]。然后可以使用像素着色器来将雾应用为一个全屏 pass，从而实现一些更加高级的结果，例如高度相关的雾或者水下着色等。

高度雾（height fog）使用一个参数化的高度和厚度，来表示一个由参与介质构成的简单平板（slab）。对于屏幕上的每个像素，会使用一个将观察射线到达表面前穿过 slab 的距离作为参数的函数，来评估雾的密度和散射光线。Wenzel [1871]提出了一个封闭形式的解，用于评估 slab 内部参与介质的指数衰减。这样做可以在 slab 边缘附近产生一个平滑的过渡雾效，这在图 14.12 左侧的背景雾中是很明显的。

深度雾和高度雾有很多种可能的变体。雾的颜色 $\mathbf{c}_f$ 可以是一个单一的颜色，可以使用观察向量来对立方体贴图进行采样，甚至也可以是复杂大气散射的结果，即使用一个逐像素的相位函数，来对颜色进行方向性的修改[743]。还可以使用 $f = f_d f_h$，来将深度雾 f_d 和高度雾 f_h 的透光率结合起来，并在场景中将这两种类型的雾效交织在一起。

深度雾和高度雾是大范围的雾效。人们可能想要渲染更多的局部现象，例如分离的雾区，又例如：在洞穴或者墓地中的几个坟墓的周围渲染一些雾。我们可以使用椭球或者 box 等形状来在需要的地方添加一些局部雾效[1871]。这些雾元素使用各自的包围盒，按照从后往前的顺序进行渲染。在像素着色器中计算每个形状与观察向量的前后交点，其中前交点记为 d_f，后交点记为 d_b。使用体积深度 $d = \max(0, \min(z_s, d_b) - d_f)$ 来计算透光率 T_r（章节 14.1.2），覆盖率则为 $\alpha = 1.0 - T_r$，其中 z_s 代表了最近的不透明表面的线性深度。在场景顶部添加的散射光量 $\mathbf{c}_f$ 可以用 $\alpha \mathbf{c}_f$ 进行表示。为了允许从网格模型中评估更多不同的雾效形状，Oat 和 Scheuermann [1308]给出了一种巧妙的单 pass 方法，来计算体积中最近的入口点和最远的出口点。他们将表面距离 d_s 保存在表面的一个通道中，将 $1 - d_s$ 保存在另一个通道中。通过设置 alpha 的混合模式来保存找到的最小值，在体积渲染之后，第一个通道种将具有最近的值 d_f，第二个通道将具有最远的值 d_b，以 $1 - d$ 的方式进行编码，我们可以从这个编码中恢复 d 的值。

水体是一种参与介质，因此会表现出相同类型的、基于深度的颜色衰减。沿海水域的透光率约为 $(0.3, 0.73, 0.63)$ 每米[261]，因此利用方程 14.23，我们可以计算出 $\sigma_t = (1.2, 0.31, 0.46)$。当使用一个不透明表面来渲染黑暗的水体时，可以在相机位于水面以下的时候启用雾效渲染，而在相机位于水面以上时关闭雾效渲染。Wenzel [1871]提出了一种更加先进的解决方案。当相机位于水下时，散射和透光率会被积分在一起，直到击中一个固体或者水面。如果相机位于水面以上，那么仅仅会对从水面到海底固体几何物体之间的距离进行积分。

14.2.2 简单的体积光

参与介质中的光线散射可能会很难评估。但值得庆幸的是，有许多有效的技术，可以在各种情况下对这种散射进行令人信服地近似。

想要获得体积效果，最简单的方法就是在帧缓冲上直接渲染透明网格并进行混合。我们将其称为溅射（splatting）方法（章节 13.9）。而想要渲染穿过窗户、茂密森林或者来自聚光灯的光束，一种解决方案是使用相机对齐的粒子，每个粒子上都带有一个

纹理。每个带有纹理的四边形，都会在光束的方向上被拉伸，同时始终保持面向相机（圆柱体约束）。

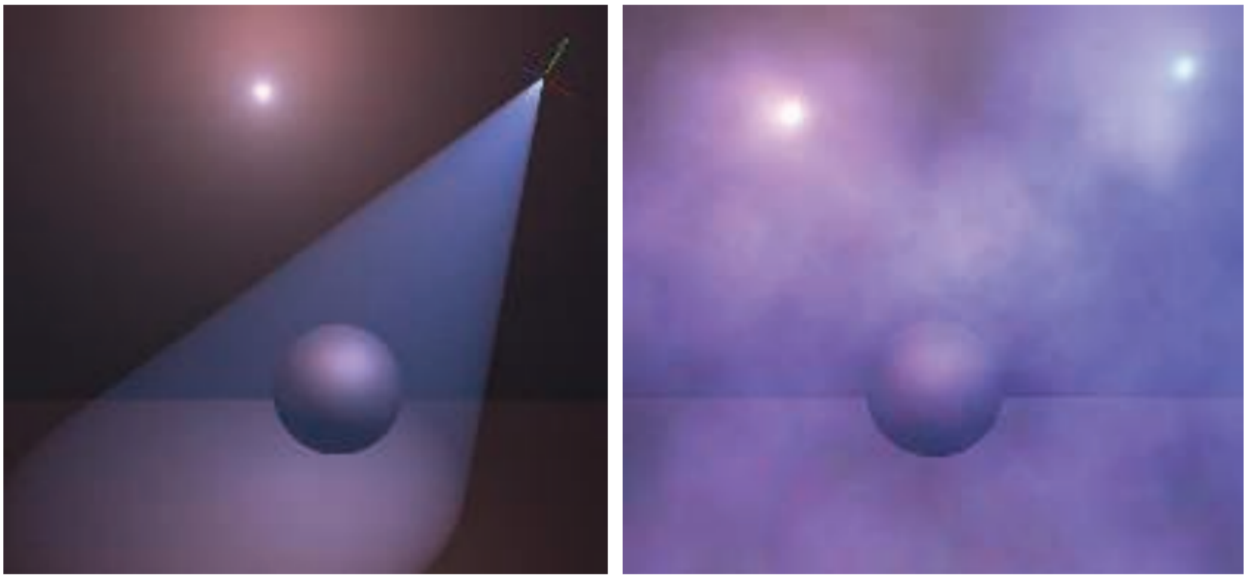


图 14.13：利用下面 GLSL 代码片段的解析积分，对光源的体积光线散射进行评估计算。它可以作为一种后处理效果应用在均匀介质（左）上，或者应用在粒子上（右），其中每个粒子都是一个具有深度的体积。[1098]

网格溅射方法的缺点在于，累积大量的透明网格将会增加所需要的内存带宽，这可能会导致性能瓶颈，并且这些面向相机的纹理化四边形有时候可能会被观察者注意到。为了解决这个问题，人们提出了利用光线单次散射的封闭形式解来进行后处理的方法。我们假设一个同质的、球形的均匀相位函数，并假设路径上具有一个恒定的参与介质，我们可以对散射光线和正确的透光率进行积分，结果如图 14.13 所示。下面的 GLSL 着色器代码片段，展示了这种技术的一个示例实现[1098]：

GLSL

```
float inScattering (vec3 rayStart , vec3 rayDir ,vec3 lightPos , float
{
    // Calculate coefficients .
    vec3 q = rayStart - lightPos ;
    float b = dot( rayDir , q);
    float c = dot(q, q);
    float s = 1.0f / sqrt (c - b*b);
    // Factorize some components .
    float x = s * rayDistance ;
    float y = s * b;
    return s * atan ( (x) / (1.0 + (x + y) * y));
}
```

其中 rayStart 是光线的起始位置，rayDir 是光线的归一化方向，rayDistance 是沿着光线的积分距离，lightPos 是光源的位置。Sun 等人[1722]的解还额外考虑了散射系数 σ_s ，这个解还描述了漫反射 radiance 和镜面 radiance 在 Lambertian 表面上和 Phong 表面上的反弹还会受到这样一个事实的影响：即光线在到达任何表面之前，都会在非直线的路径上进行散射。为了考虑透光率和相位函数，可以使用一个更加 ALU-heavy 的解决方案[1364]。所有这些模型在它们所做的事情上都是有效的，但是它们并没有考虑来自深度贴图或者非同质参与介质的阴影。

依靠一种被称为泛光的技术，可以近似地计算屏幕空间中的光线散射[539, 1741]。对帧缓冲进行模糊处理，并以一个较小的百分比再将模糊后的结果叠加到帧缓冲上，从而使得每个明亮的物体会将自身 radiance 泄漏到附近范围内[44]。这种技术通常用于对相机镜头中的缺陷进行近似，但是在某些环境中，它也是一种适用于较短距离的、非遮挡散射的良好近似。章节 12.3 中对泛光效果进行了详细描述。

Dobashi 等人[359]提出了一种方法，通过使用一系列平面来对体积进行采样，从而渲染大规模的大气效果。这些采样平面垂直于观察方向，并按照从后向前的顺序进行渲染。Mitchell [1219]也提出了同样的方法来渲染聚光灯的效果，并使用阴影贴图投射来自不透明物体的体积阴影。章节 14.3.1 详细描述了使用溅射切片（splatting slice）来渲染体积的方法。

Mitchell [1225]、Rohleder 和 Jamrozik [1507]提出了一种在屏幕空间中运行的替代方法，如图 14.14 所示，它可以用来渲染来自遥远光线（例如太阳）的光束。首先，该方法会将缓冲区清除为黑色，在远平面上渲染一个围绕太阳虚假明亮物体，并使用一个深度缓冲进行深度测试，从而接收未被遮挡的像素。其次，在图像上应用定向模糊，以便从太阳处向外泄漏之前积累下来的 radiance。可以使用一个两 pass 的可分离滤波技术（章节 12.1），其中每个 pass 都使用了 n 个样本，从而获得与 n^2 个样本相同的模糊结果，并且具有更快的渲染速度[1681]。最后，将最终的模糊缓冲区添加到场景缓冲区中。这个技术是有效的，尽管只有屏幕上能够看见的光源才可以投射出光束，但是它以一个较小的成本却能够提供显著的视觉效果。

图 14.14：使用屏幕空间后处理渲染的光束。[1225]

14.3 通用的体渲染

在本小节中，我们将介绍更多基于物理的体渲染技术，即尝试表示介质与光源之间的相互作用（章节 14.1.1）。通用的体渲染技术会涉及到空间变化的参与介质，这类参与介质通常会使用体素来进行表达（章节 13.10），体积光的相互作用会在视觉上产生复杂的散射现象和阴影现象。通用的体渲染方案还必须考虑体积与其他场景元素的正确组合，例如不透明表面或者透明表面。这种空间变化的介质属性，在游戏环境中可能会体现为烟雾效果和火焰效果，以及体积光和阴影的相互作用等。此外，我们可能还希望将固体材质表示为半透明体积，用在诸如医学可视化等应用中。

14.3.1 体积数据可视化

体积数据可视化（volume data visualization）是一种用于显示体积数据和分析体积数据的工具，这些体积数据通常都是标量字段。计算机断层扫描（computer tomography，CT）和磁共振图像（magnetic resonance image，MRI）技术可以用来创建人体内部结构的临床诊断图像。一个数据集中可能会包含很多体素（例如 256^3 个体素），每个体素位置上都包含一个或者多个值。这些体素数据可以用来构成一个三维图像。体素渲染可以展示一个实体模型，或者使各种材质（例如皮肤和颅骨）来表现部分透明或者完全透明的效果。切割面可以用于显示源数据的其中一个子空间或者部分数据。体渲染除了用于医学和石油勘探等不同领域的可视化之外，还可以生成照片级逼真的图像。

有许多的体素渲染技术[842]。在复杂的光照环境下，可以使用常规的路径追踪或者光子映射（photon mapping）来对体积数据进行可视化。人们已经提出了几种开销较低的方法，来达到实时级别的性能表现。

对于实体物体，可以使用隐式曲面技术（详见章节 17.3）来将体素表示转换为多边形曲面。对于半透明现象，体积数据集可以通过垂直于观察方向的一组等间距切片来进行采样，图 14.15 展示了其工作原理。也可以使用这种方法来渲染不透明表面[797]。在这种情况下，当体素密度大于给定阈值时，我们就认为该体素是固体体积，并且可以使用法线 $\mathbf{n}$ 来评估密度场的三维梯度。

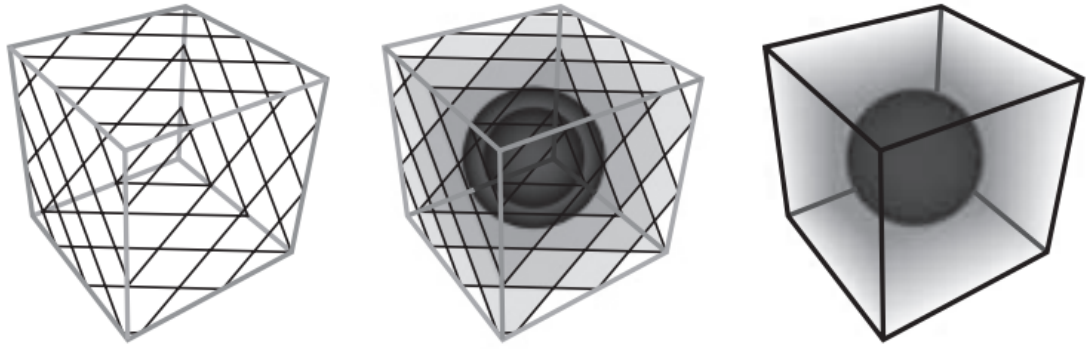


图 14.15：这个体积是由一系列平行于观察平面的切片进行渲染的。左侧展示了一些切片与体积的交点。中间展示了这些切片的渲染结果。右边展示了大量切片被渲染和混合后的结果。

对于半透明数据，可以在每个体素内存储各自的颜色和不透明度。为了减少内存占用，并使得用户能够控制可视化，因此提出了传输函数（transfer function）。第一个解决方案是使用一个一维的传输纹理，来将体素的标量密度值映射到颜色和不透明度。然而，这种方法并不能识别特定的材质转变，例如：从人类鼻窦骨到空气，或者骨骼组织到软组织，对这些不同的组织独立使用不同的颜色进行着色。为了解决这个问题，Kniss 等人[912]建议使用基于密度 d 和密度场梯度长度 $\|\nabla d\|$ 来进行索引的二维传输函数。发生变化的区域会具有较高的梯度大小时，这种方法使得基于密度转换的着色更有意义，如图 14.16 所示。

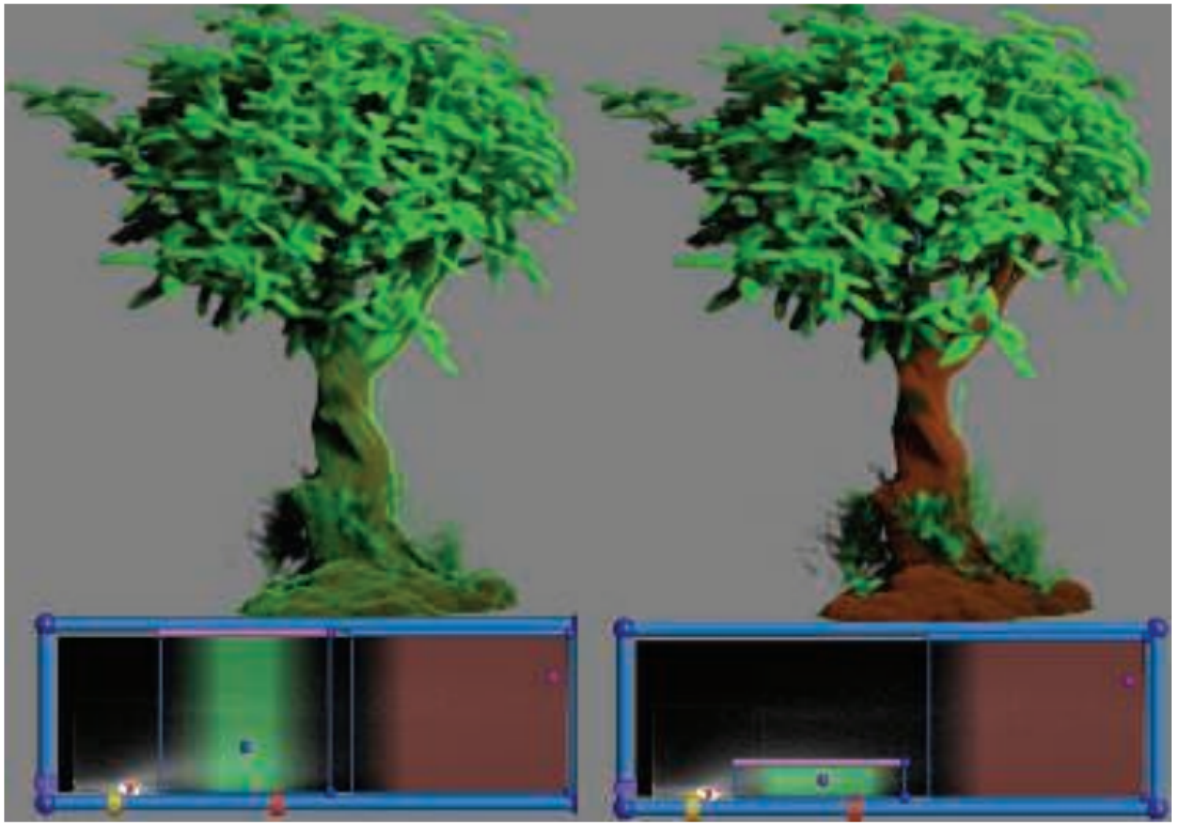


图 14.16：使用一维（左）传输函数和二维（右）传输函数来计算体积的材质和不透明度[912]。在第二种情况下，可以保持树干的棕色，不让树干被叶子密度较轻的绿色所覆盖。图像底部为对应使用的传输函数，x 轴代表密度，y 轴代表密度场的梯度长度 $||\nabla d||$ 。[912]

Ikits 等人[797]对这种技术和相关问题进行了深入讨论。Kniss 等人[913]则对这种方法进行了扩展，该方法使用半角（half-angle）来进行切片。切片仍然是从后往前进行渲染的，但是这些切片的朝向会介于光线方向和观察方向之间。使用这种方法，可以从光源方向来渲染 radiance 和遮挡信息，并在观察空间中累积每个切片。切片纹理可以作为渲染下一个切片时的输入，使用光源方向的遮挡信息来评估体积阴影，并使用 radiance 来估计多次散射，即光线在到达眼睛之前在介质中的多次反弹。由于之前的切片是根据圆盘内的多个样本进行采样的，因此这个技术只能模拟锥体范围内前向散射所引起的次表面现象。这种方法最终生成的图像质量很高，如图 14.17 所示。Schott 等人[1577, 1578]对这种半角方法进行了扩展，以计算环境光遮蔽和景深模糊等效果，从而提高了用户在查看体素数据时对深度和体积的感知。

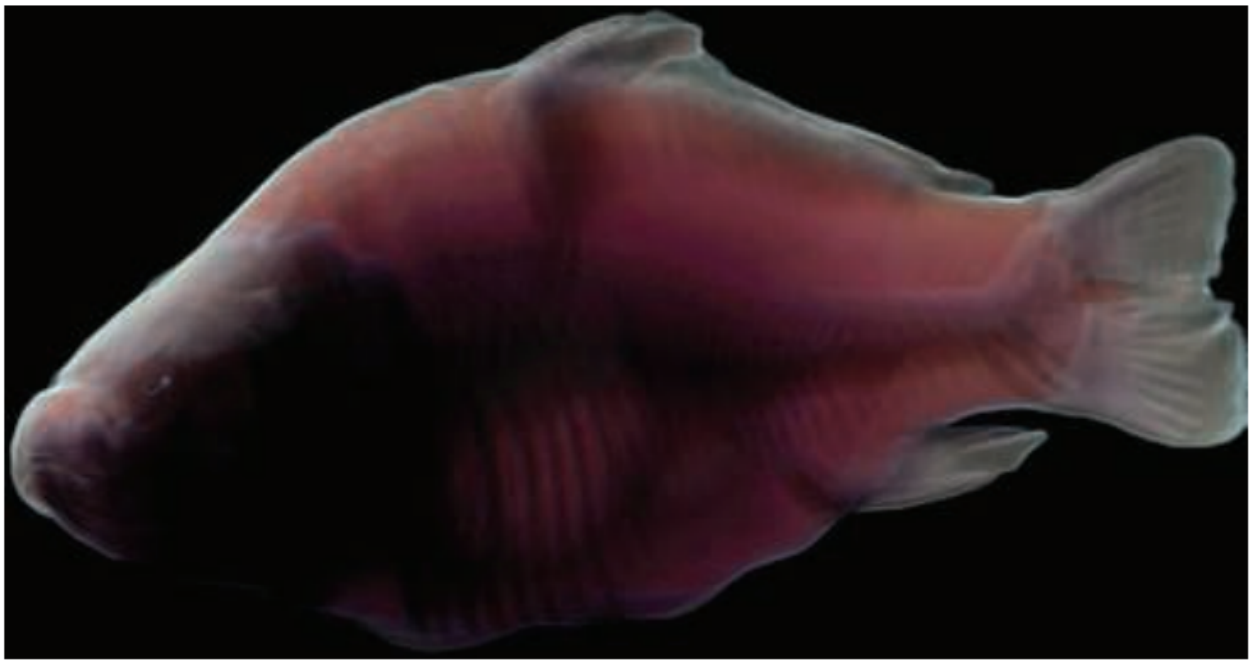


图 14.17：基于半角切片的前向次表面散射体渲染。[797]

如图 14.17 所示，半角切片方法可以渲染高质量的次表面散射现象。然而，由于需要对每个切片进行光栅化，因此会带来很高的内存带宽成本。Tatarchuk 和 Shopf [1744]在着色器中使用光线步进来进行医学成像，因此只需要进行一次光栅化即可，大大降低了带宽成本。有关光照和阴影的内容，可以按照下一小节中所描述的方式进行实现。

14.3.2 参与介质渲染

实时图形应用程序可以通过渲染参与介质，来描绘更加丰富的场景。当涉及到时间、天气或者环境变化（例如建筑物破坏）的时候，这些效果的渲染要求会变得更高。例如：森林中的雾在中午时分或者黄昏时分看起来会有所不同；在树木之间的光束，应当与太阳变化的方向和颜色相适应。光束也应当根据树木的运动来进行相应的动画，例如：爆炸会移除场景中的一些树木，从而导致该区域内的散射光线发生变化，因为此时的遮挡物变少了，并且产生了大量灰尘。篝火、手电筒和其他光源也会在空气中产生散射现象。在本小节中，我们将讨论能够实时模拟这些动态视觉现象效果的技术。

一些技术专注于渲染单一来源的大规模阴影散射。Yusov [1958]对其中一种方法进行了深入的描述，它基于对沿极线（epipolar line，核线）的内散射进行采样，这里的极线指的是投射到相机图像平面单一直线上的光线。使用从光源视角进行渲染的深度贴图来确定一个样本是否位于阴影中。这个算法会从相机位置开始执行光线步进，同时会使用一个沿着光线的最小/最大（min/max）层次结构来跳过空白的空间，并且仅在深度不连续的地方进行光线步进，即只在真正需要准确计算体积阴影的地方进行光线步进。通过渲染由光源空间深度贴图所生成的网格，也可以在观察空间中进行采样，而不是沿着极线来对这些不连续性进行采样[765]。在观察空间中，只需要计算正面和背面之间的体积就可以对最终的散射 radiance 进行计算。为此，计算内散射的方法是将正面散射的 radiance 添加到视图中，然后再减去背面散射的 radiance。

这两种方法都能有效地再现由于不透明表面遮挡而产生单次散射事件，并能够生成阴影[765, 1958]。然而，这两种方法都无法表示异质（heterogeneous）的参与介质，因为它们都假设参与介质是恒定均质的。另外，这些技术无法生成来自透明表面的体积阴影，例如：来自参与介质的自阴影或者来自粒子的透明阴影（章节 13.8）。但是这些方法仍然在游戏中发挥着巨大的作用，因为它们可以在较高的分辨率下进行渲染，而且渲染速度很快，这要归功于在光线步进的过程种跳过了空白区域[1958]。

研究人员提出了溅射方法（splatting），用于处理更加一般的异质介质，这种方法会沿着光线来对体积材质进行采样。Crane 等人[303]在不考虑任何输入照明的情况下，使用溅射方法来渲染烟雾、火焰和水，这些都是流体模拟的结果。在烟雾和火焰的情况下，会在每个像素处生成一条光线，这条光线会在体积内进行步进，以固定的间隔来收集材质的颜色信息和遮挡信息。在渲染水体的时候，一旦光线与水面相交，便会终止体积采样；在每个样本位置处，会使用密度场梯度来评估表面法线。为了保证水面的平滑，会使用三次插值来对密度值进行过滤。图 14.18 展示了这些技术的使用案例。

图 14.18：在 GPU 上使用体渲染与流体模拟相结合的技术来渲染烟雾和水体。[303]

将太阳光（平行光）、点光源和聚光灯考虑在内，Valient [1812]将每个光源的散射包围体渲染到一个半分辨率的缓冲区中。每个光照体积都会使用光线步进来进行采样，并在光线步进的开始位置应用一个逐像素的随机偏移量。这样做会增加一些噪点，但是它的优点在于可以消除由均匀步进而产生的带状瑕疵。这种在每一帧中使用不同的噪声值，是一种隐藏瑕疵的常用手段。在对前一帧进行重投影，并与当前帧进行混合之后，这些噪点将会被平均，从而消失。通过将平面粒子体素化成一个三维纹理，再以八分之一的屏幕分辨率将这个三维纹理映射到相机的视锥体上，从而来渲染异质介质。这个体积数据会在光线步进期间作为材质的密度信息。半分辨率的散射结果可以通过使用双边高斯模糊，然后再使用双边上采样滤波器，并考虑像素之间的深度差异，来将其叠加合成到全分辨率的主缓冲上[816]。当深度差异与中心像素相比过高时，这个样本将会被丢弃。这个高斯模糊在数学上是不可分离的（章节 12.1），但是在实践中的效果很好。这个算法的复杂度取决于溅射在屏幕上的光照体积数量，因为这个复杂度与其像素覆盖率有关。

这种方法可以通过使用蓝噪声（blue noise）来进行扩展，蓝噪声更擅长在一帧中的像素上产生均匀分布的随机值[539]。这样做的结果是，使用双边滤波器进行上采样以及在空间中混合样本的时候，能够产生更加平滑的视觉效果。半分辨率缓冲的上采样操作也可以通过将四个随机样本混合在一起来进行实现。这样生成的结果仍然会有一些噪点，但是由于它给出了逐像素的全分辨率噪声，因此可以很容易地通过一个时域抗锯齿的后处理（章节 5.4）来进行解决。

所有这些方法的缺点在于，有序深度的体积元素与其他任何透明表面混合在一起，将无法给出一个视觉上正确的渲染顺序。这里的透明表面可以是大型非凸的透明网格，也可以是大规模的粒子效果。当涉及到在透明表面上应用体积光照时，所有的这些算法都需要进行一些特殊的处理，例如在一个体素数据中包含内散射和透光率的体积

[1812]。那么，为什么我们不从一开始就使用基于体素的表示方法呢？这样不仅可以表示空间变化的参与介质属性，还可以表示由光线散射和透光率所引起的 radiance 分布。实际上，这样的技术在电影工业中使用已久[1812]。

Wronski [1917]提出了一种方法，该方法会将场景中来自太阳和光源的散射 radiance，体素化为一个映射在裁剪空间中的三维体积纹理 V_0 。在每个体素中心的世界空间位置上来计算散射 radiance，其中体积的 x 轴和 y 轴与屏幕坐标相对应，而 z 坐标则映射到相机视锥体的深度。这个体积纹理的分辨率要比最终图像的分辨率低得多。这种技术的一种典型实现使用了这样的一个体素分辨率，即 x 轴和 y 轴的分辨率为屏幕分辨率的八分之一。而 z 坐标的细分则取决于质量和性能的权衡，其中 64 个切片是一个典型的选择。这个纹理包含了存储在 RGB 通道的内散射 radiance $L_{\text{scat in}}$ ，以及存储 alpha 通道的消光系数 σ_t 。根据这个输入数据，按照从近到远的顺序迭代每个切片，从而生成最终的散射体积 V_f ：

$$V_f[x,y,z] = (L'_{\text{scat}} + T'_r L_{\text{scat in}}\, d_s, T_{r_{\text{slice}}} T'_r) \tag{14.15}$$

其中：

$$\begin{aligned} L'_{\text{scat}} &= V_0[x,y,z-1]_{rgb} \\ T'_r &= V_0[x,y,z-1]_a \\ T_{r_{\text{slice}}} &= e^{-\sigma_t d_s} \end{aligned}$$

这将在世界空间的切片深度 d_s 上，从前一个切片 $z-1$ 的数据中来更新切片 z 的数据。这样做将会导致散射体积 V_f 包含到达观察者的散射 radiance，以及背景中每个体素的透光率。在方程 14.15 中，我们可以注意到 $L_{\text{scat in}}$ 仅会受到前一个切片透光率 T_r 的影响。这种行为实际上是不正确的，因为 $L_{\text{scat in}}$ 也应当受到当前切片内 σ_t 所产生的透光率的影响。

Hillaire [742, 743]对这个问题进行了讨论。对于在给定深度上具有恒定消光系数 σ_t 的情况，他提出了 $L_{\text{scat in}}$ 积分的一个解析解：

$$V_f[x,y,z] = \left(L'_{\text{scat}} + \frac{L_{\text{scat in}} - L_{\text{scat in}} T_{r_{\text{slice}}}}{\sigma_t}, T_{r_{\text{slice}}} T'_r\right) \tag{14.16}$$

一个 radiance 为 L_s 的不透明表面，其最终像素的 radiance L_o 将由 V_f 中的散射 radiance L_{scat} 和透光率 T_r 进行修正，并使用裁剪空间的坐标进行采样，即 $L_o =$

$T_r L_s + L_{\text{scat}}$ 。由于 V_f 是比较的粗糙的，因此它可能会受到相机运动、高频强光或者阴影的影响，从而产生一些瑕疵。可以将前一帧中的 V_f 进行重投影，并使用一个指数移动平均（exponential moving average，EMA），来将其与新的 V_f 结合起来[742]。

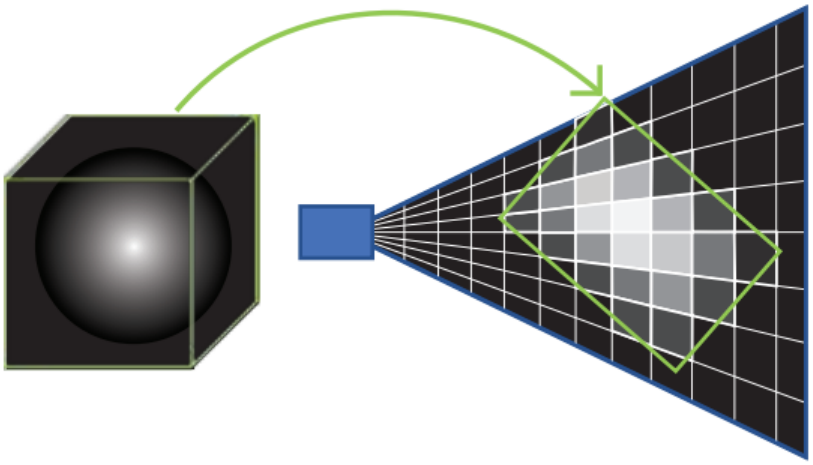


图 14.19：艺术家将一个参与介质的体积放置在关卡中，将这个参与介质体素化到相机的视锥体空间中[742, 1917]。左侧是一个三维纹理，在这个例子中该三维纹理代表了一个球体形状，这个球体形状被映射到了体积中。这个纹理定义了体积的外观，类似于三角形上的纹理。在右侧，通过考虑它的世界变换，这个体积被体素化到了相机的视锥体中。使用一个计算着色器来将贡献累积到该体积所包含的每个体素中。最终得到的材质可以用来评估每个体素中的光线散射作用[742]。请注意，在映射到相机的裁剪空间时，每个体素会被压成截锥体的形状，它称为 froxel。

在此框架的基础上，Hillaire [742]提出了一种基于物理的方法来定义参与介质的材质，即：散射 σ_s ，吸收 σ_a ，相位函数参数 g 和发射的 radiance L_e 。这个材质会被映射到相机的视锥体中，并存储到一个参与介质的材质体积纹理 V_{pm} 中，这是一个存储不透明表面材质的、三维版本的 G-buffer（章节 20.1）。Hillaire 表示，如果只考虑单次散射的话，尽管进行了离散的体素化操作，使用这种基于物理的材质表示能够产生接近于路径追踪的视觉效果。与网格类似，位于世界中的参与介质体积也会被体素化为 V_{pm} （如图 14.19 所示）。在每个体积中都定义了一种材质，并添加了一些变化，得益于从三维纹理中采样获得的密度，从而可以生成异质的参与介质，生成的结果如图 14.20 所示。同样的方法也在虚幻引擎中进行了实现[1802]，但是不同之处在于，虚幻引擎并没有使用 box 体积来作为参与介质的来源，而是使用了粒子，即它假设了一个球形的体积而不是一个 box 形状的体积。也可以使用一个稀疏结构来表示材质体积纹理[1191]，即让某些体素为空，或者是让某些体素指向一个包含更细粒度材质数据的体积。

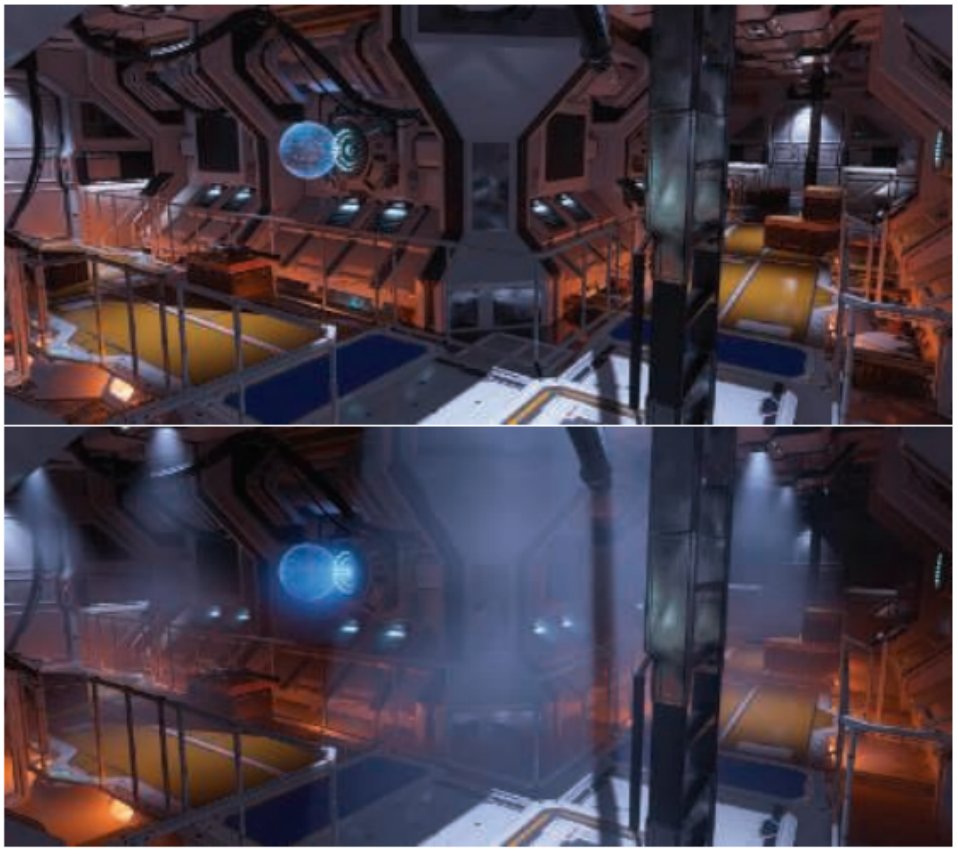


图 14.20：上面的场景没有体积光和体积阴影，下面的场景有体积光和体积阴影。场景中的每一个光源都与参与介质进行了相互作用。每个光源的 radiance、IES 配置（IES 描述文件，一种描述光源的光照分布的文件格式）和阴影贴图都会用于累积其散射光线的贡献[742]。

这个基于摄像机视锥体的方法，其唯一的缺点在于：为了在性能较弱的平台上达到可接受的性能（并使用合理的内存开销），需要使用较低的屏幕空间分辨率[742, 1917]。这就是前面所介绍的溅射方法的优点，因为它们可以生成清晰的视觉细节。前面我们提到，溅射方法需要更多的内存带宽，并且无法提供统一的解决方案，例如：溅射方法很难在其他透明表面上进行应用，因为会产生排序问题；溅射方法也无法让参与介质在自身上投射体积阴影。

不仅仅是直射光，那些经过反射或者散射的光线也可以进一步被参与介质散射。与 Wronski [1917]的方法类似，虚幻引擎可以烘焙体积光照贴图（volume light map），来将 irradiance 存储在体积中，并在观察空间进行体素化的时候将其散射回介质中[1802]。为了在参与介质中实现动态全局光照，也可以使用光照传播体积（light propagation volume，LPV）[143]。

体积阴影是一个重要的特性。如果没有体积阴影的话，浓雾场景的最终图像可能会看起来过于明亮和平坦[742]。此外，阴影也是一种重要的视觉暗示，它们有助于观察者对深度和体积的感知[1846]，从而产生更加逼真的图像，带来更好的沉浸感。

Hillaire [742]提出了一种实现体积阴影的统一解决方案。根据 clipmap 分布方案，参与介质的体积和粒子会被体素化成围绕相机的三个级联体积，它们被称为消光体积（extinction volume）[1777]。这些体积包含了用于计算 T_r 的消光系数 σ_a，并代表了一个统一的数据源，以便使用不透明的阴影贴图来实现体积阴影[742, 894]，如图 14.21 所示。这样的解决方案使得粒子和参与介质能够产生自阴影，能够相互投射阴影，并能够在场景中其他不透明和透明的元素上投射阴影。

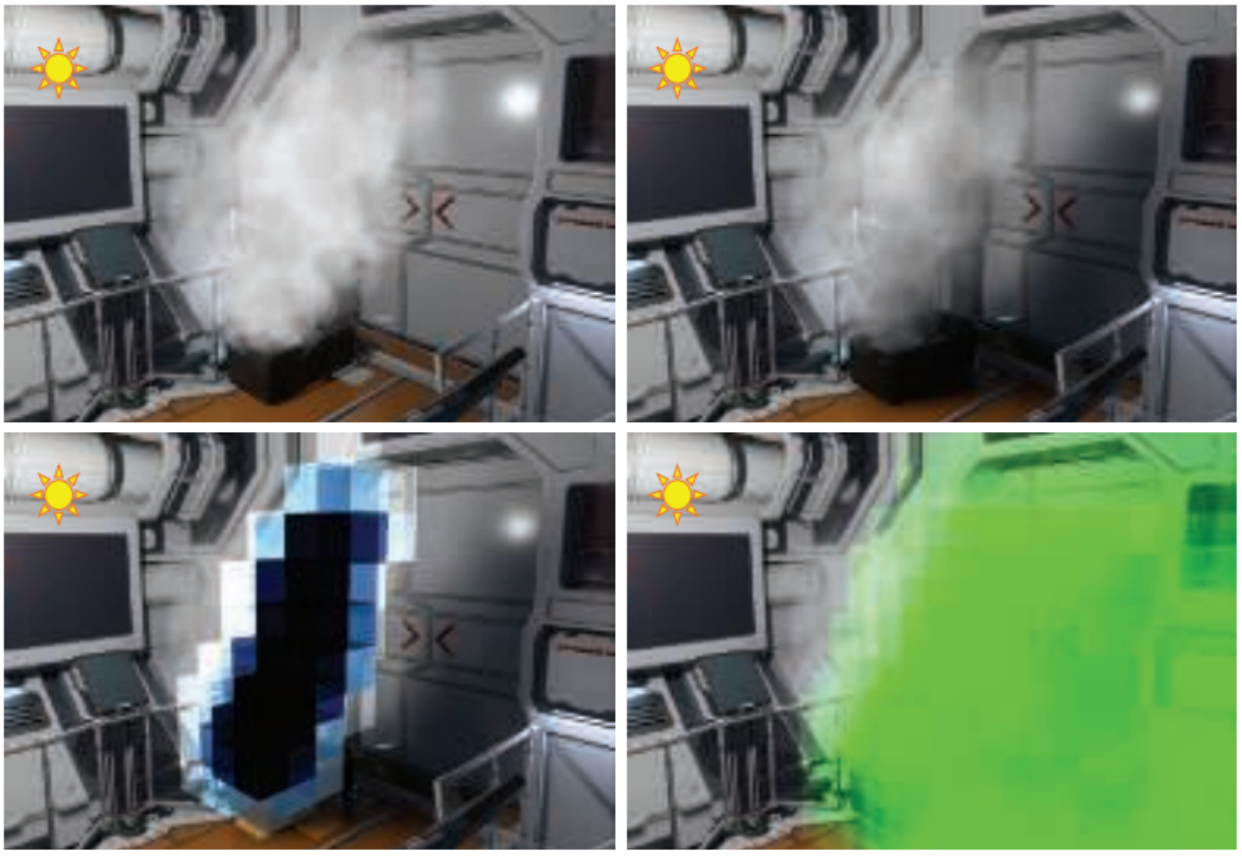


图 14.21：左上角的场景没有体积阴影，右上角的场景有体积阴影。左下角是体素化粒子消光的调试视图，右下角是体素化体积阴影的调试视图。其中绿色越深，意味着透光率越少[742]。

体积阴影可以使用不透明阴影贴图来进行表示。然而，如果需要较高分辨率来捕捉细节的话，使用体积纹理可能很快就会成为限制。因此，为了能够更加有效地表示 T_r，研究人员提出了一些替代的表示方法，例如使用函数的正交基，如傅里叶变换（Fourier transform）[816]或者离散余弦变换（discrete cosine transform）[341]，详见章节 7.8。

14.4 天空渲染

想要渲染一个真实效果的世界，当然需要渲染行星的天空、大气效果和云。地球上的蓝天，实际上就是太阳光在大气参与介质中散射的结果。至于为什么白天的天空是蓝

色的，而太阳在地平线上时天空是红色的，这在章节 14.1.3 中进行了解释。大气也是一个十分关键的视觉暗示，因为大气的颜色与太阳的方向有关，而太阳的方向与一天中的时间有关。同时，有时候大气是雾蒙蒙，这个外观有助于观众感知场景中元素的相对距离、相对位置和相对大小。准确渲染这些效果组件是非常重要的，因为越来越多的游戏和其他应用程序用到了这些组件，这些应用程序通常具有一天内的动态时间、不断变化的天气（影响云的形状），以及可供探索、驾驶甚至飞行的大型开放世界。

14.4.1 天空和空气透视

为了渲染大气效果，我们需要考虑两个主要的组成部分，如图 14.22 所示。第一，我们需要模拟太阳光线与空气粒子之间的相互作用，产生与波长相关的瑞利散射。这将会产生天空的颜色和天空中的薄雾，这也称为空气透视（aerial perspective）。第二，我们需要研究集中在地面附近的大颗粒对太阳光线所产生的影响，这些大颗粒的浓度则取决于天气条件和空气污染水平等因素。这些大颗粒会引起与波长无关的米氏散射，这种现象会在太阳周围形成明亮的光晕，尤其是在粒子浓度较高的情况下。

图 14.22：两种不同类型的大气光线散射：第一行图片只有瑞利散射；第二行图片既有常规的瑞利散射，也有米氏散射。从左到右分别是：密度为 0，常规密度[203]，夸张密度。[743]

第一个基于物理的大气模型[1285]，是从太空来渲染地球以及地球的大气层，它模拟了太阳光线的单次散射。使用 O 'Neil 提出的方法也可以得到类似的结果[1333]。这个方法可以在一个单 pass 的着色器中，使用从地面到太空的光线步进来渲染地球。在渲染天空穹顶（sky dome）的时候，需要使用昂贵的逐顶点光线步进来整合米氏散射和瑞利散射。这个视觉上的高频相位函数是在像素着色器中进行计算的，这使得地球大气的外观较为平滑，避免了由于插值而暴露天空的实际几何形状。也可以将散

射结果存储在纹理中，并将计算分散在几帧中进行，这样也实现相同的结果，在带来一定更新延迟的情况下，可以获得更好的性能表现[1871]。

分析技术使用了拟合的数学模型，这个模型基于对真实天空 radiance 的测量[1443]，或者基于大气光线散射路径追踪所生成的参考图像[778]。与参与介质的材质参数相比，这个输入参数的集合通常是有限的。例如：使用浑浊度（turbidity）来代表粒子对米氏散射的贡献，而不是参与介质的系数 σ_s 和 σ_t 。Preetham 等人[1443]所提出的这种模型利用浑浊度和太阳高度，来在任何方向上对天空 radiance 进行评估计算。通过增加对光谱输出的支持、对太阳周围散射 radiance 更好的方向性以及一个新的地面反照率输入参数，可以对这个方法进行改进[778]。这种分析天空模型的计算速度很快。然而，这种方法仅限于在地面上的视角，同时我们无法对大气参数进行修改，也就是说，这个方法无法对外星行星进行模拟，也无法实现特定的、艺术驱动（art-driven）的视觉效果。

渲染天空的另一种方法是假设地球是完美的球形，在其周围有一层由异质参与介质所组成的大气层。Bruneton 和 Neyret [203]，以及 Hillaire [743]对大气的组成做了进行了广泛且详细的描述。利用这些物理事实，可以根据当前的视图高度 r 、观察向量相对于天顶的余弦值 μ_v 、太阳方向相对于天顶的余弦值 μ_s 、以及观察向量相对于方位平面上太阳方向的余弦值，来在预计算表格中存储透光率和散射率。例如：从视点到大气边界的透光率可以使用两个参数 r 和 μ_v 来进行参数化。在预计算步骤中，透光率可以在大气中进行积分，并存储在一个二维的查找表（LUT）纹理贴图中，并在运行过程中使用相同的参数化来对其进行采样。这个纹理可以将大气透光率应用在各种天空元素上，例如太阳，星星，或者其他天体。

通过将散射现象考虑在内，Bruneton Neyret [203]描述了一种方法，这种方法可以将散射结果存储在一个由上述所有参数进行参数化的四维 LUT S_{lut} 中。他还提供了一种可以通过 n 次迭代来评估 n 阶多次散射的方法：(i) 计算单次散射表 S_{lut} ，(ii) 使用 S_{lut}^{n-1} 来计算 S_{lut}^{n} ，(iii) 将结果添加到 S_{lut} 中。重复执行 $n-1$ 次的步骤 (ii) 和 (iii) 。Bruneton 和 Neyret 提供了有关该过程的更多细节和源代码[203]，图 14.23 展示了这种方法生成的结果。Bruneton 和 Neyret 的参数化方法有时会在地平线上表现出一些视觉瑕疵。Yusov [1957]提出了一种改进的转换方法，也可以通过忽略参数 ν [419]，只使用一个三维的 LUT；使用这种方法的话，地球将不会在大气中投下阴影，这是一种可以接受的权衡。这种方法的优点在于，这个三维的 LUT 会小得多，更新和采样的成本也会更低。

图 14.23：使用查找表方法，分别从地面（左）和太空（右）来实时渲染地球大气。[203]

EA 的寒霜引擎使用了最后一种的三维 LUT 方法，并被许多即时游戏所采用，例如《极品飞车》、《镜之边缘：催化剂》和《FIFA》[743]。在这种情况下，艺术家可以对这些基于物理的大气参数进行调整，从而达到目标天空的视觉效果，甚至可以模拟地球外的大气效果，如图 14.24 所示。当大气参数发生变化的时候，这个 LUT 必须重新进行计算。为了能够更加高效地更新这些 LUT，也可以使用一个函数来对大气中的材质积分进行近似，而不是使用光线步进[1587]。通过将 LUT 和多次散射的计算分摊在多帧中，可以将 LUT 的更新代价平摊到原来的 6%。对于给定的散射阶 n，我们可以只更新 S^n_{lut} 中的一部分，并对最后两个求解的 LUT 进行插值，从而实现将计算分摊在几帧中进行，这样做的代价也只是会有几帧的延迟而已。为了避免对每个像素的不同 LUT 进行多次采样，米氏散射和瑞利散射会被烘焙到相机视锥体映射（camera–frustum–mapped）的低分辨率体积纹理的体素中，这是另外一种优化方法。为了在太阳周围生成平滑的散射光晕，可以在像素着色器中来计算视觉上的高频相位函数。使用这种类型的体积纹理，允许我们在场景中的任何透明物体上应用逐顶点的空气透视。

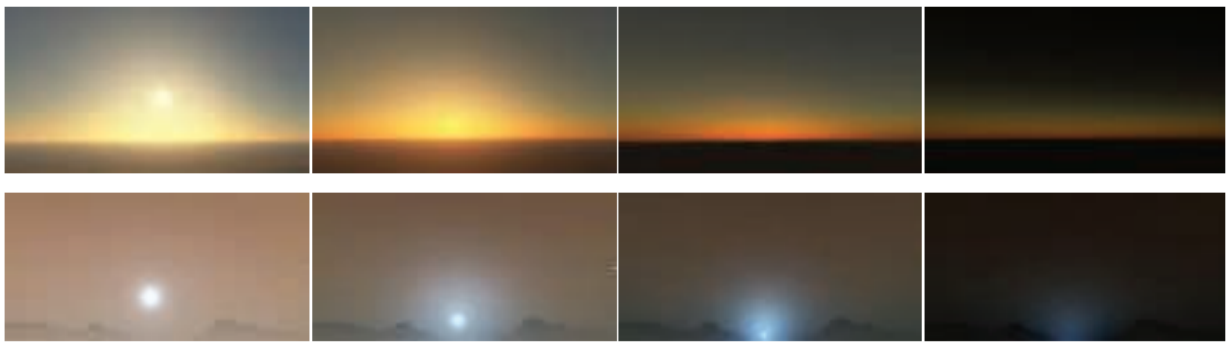


图 14.24：使用全参数化模型的实时渲染，可以对地球大气层（上）[203]和其他行星的大气层进行模拟，例如火星上的蓝色日落（下）[743]。

14.4.2 云

云是天空中的复杂元素。当代表即将到来的风暴时，天空中的云看起来会很有威胁性；云还可以会是小巧的、史诗般的、单薄的或者厚重的。云的变化速度很慢，其大尺度的形状和小尺度的细节都会随着时间的推移而逐渐发生变化。具有天气和时间变化的大型开放世界游戏要更加复杂，需要动态的云渲染解决方案。根据目标性能和视觉质量的不同，可以使用不同的技术来渲染天空中的云。

云是由水滴组成的，它具有较高的散射系数以及复杂的相位函数，这些物理特点使得云有着特定的外观。如章节 14.1 所述，云通常会被表示为参与介质，其材质具有较高的单次散射反照率 $\rho = 1$，消光系数 σ_t 在层云（stratus，底层的、水平的云层）中的范围为 $[0.04, 0.06]$，在积云（cumulus，孤立的、低层的、像棉花一样蓬松的云）的范围内为 $[0.05, 0.12]$ [743]，如图 14.25 所示。鉴于 ρ 接近于 1，因此我们可以假设 $\sigma_s = \sigma_t$ 。

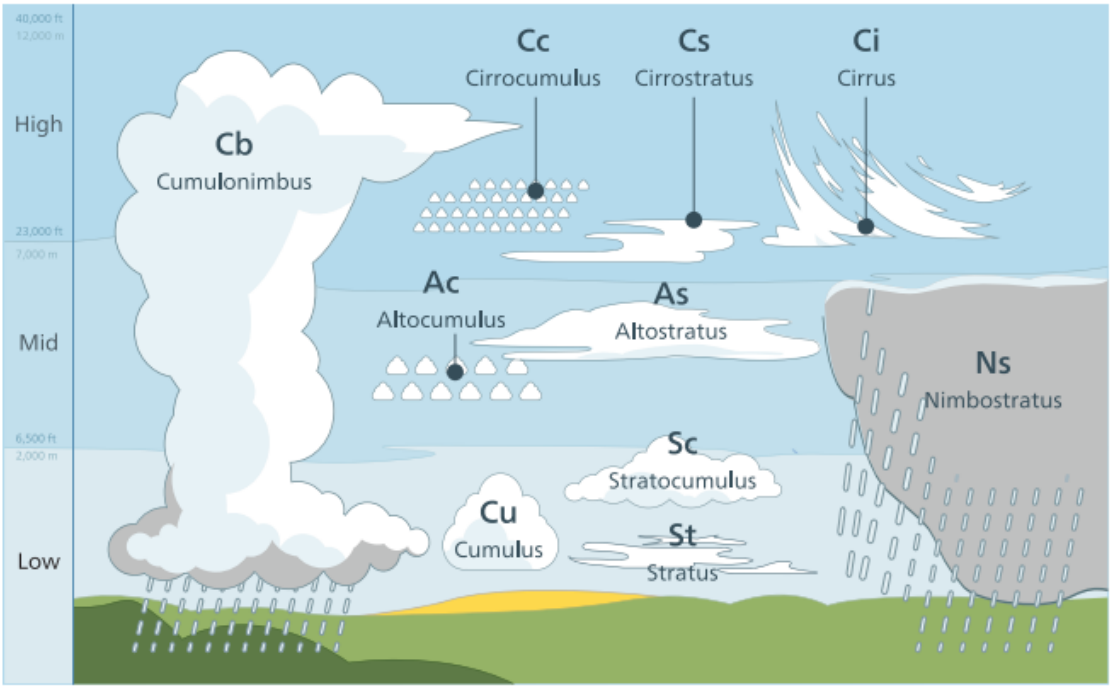


图 14.25：地球上不同类型的云。

云渲染的一个经典方法是，使用 alpha 混合来在天空上叠加一个全景纹理（panoramic texture），这种方法在渲染静态天空的时候很方便。Guerrette [620] 提出了一种视觉上流动的技术，这种方法可以给人一种错觉，就好像天空中的云受到了一个全局的风向影响，看起来就像是云在天空中运动一样。这是一种十分有效的方法，它对静态的全景云纹理方法进行了改进。但是，这种方法无法表达云的形状变化和光照变化。

将云作为粒子

Harris 将云作为粒子和 impostor 体积来进行渲染[670]，详见章节 13.6.2 和图 13.9。

Yusov [1959]提出了另一种基于粒子的云渲染方法。他所使用的渲染图元被称为体积粒子（volume particle）。每个体积粒子都被表示为一个四维 LUT，每个体积粒子都是一个面向视图的四边形粒子，该方法允许对粒子上的散射光线和透光率进行检索，这是一个与太阳光线方向和观察方向有关的函数，如图 14.26 所示。这种方法非常适合渲染层积云（stratocumulus cloud），层积云如图 14.25 所示。

图 14.26：将云作为粒子体积来进行渲染。[1959]

将云作为粒子来进行渲染，有时可以看到一些离散化现象和突然出现的瑕疵，尤其是在围绕云进行视角旋转的时候。这些问题可以通过使用体积感知混合（volume-aware blending）来进行避免。这种能力是通过使用一种被称为光栅器有序视图（rasterizer order view，ROV，章节 3.8）的 GPU 功能来进行实现的。体积感知混合可以使得每个图元的像素着色器操作在资源上进行同步，同时允许确定性的自定义混合操作。最近的 n 个粒子的深度层会被保存在一个缓冲区中，这个缓冲区与我们的渲染目标具有相同的分辨率。这个缓冲区在后续步骤中会进行读取，并通过考虑交叉深度来对当前渲染的粒子进行混合，然后再次写入下一个要进行渲染的粒子。该方法的渲染结果如图 14.27 所示。

图 14.27：左边：云粒子以常规的方式进行渲染。右边：使用体积感知混合来进行粒子渲染。[1959]

将云作为参与介质

Bouthors 等人[184]将云视为孤立的元素，使用两个组成部分来表示云：一个网格，用于表现云的整体形状；一个超纹理（hypertexture）[1371]，用于在网格表面到云内部一定深度的范围内添加高频细节。使用这种表示方式，云的边缘可以使用精细的光线步进来收集细节，而云的内部则可以被认为是同质的。光线在对云的内部结构进行光线步进时，会对 radiance 进行积分，同时会根据散射顺序，使用不同的算法来收集散射 radiance。利用章节 14.1 中所描述的解析方法，可以对单次散射的光线进行积分。利用位于云表面的圆盘状光线收集器，可以对传输函数表进行离线预计算，从而加速多重散射的计算速度。最终生成的结果具有很高的视觉质量，如图 14.28 所示。

图 14.28：使用网格和超纹理进行渲染的云。[184]

除了将云作为孤立的元素进行渲染，还可以将它们建模为大气中的一层参与介质。Schneider 和 Vos 基于光线步进，提出了一种高效的云渲染方法[1572]。这种方法只需要几个参数，就可以在动态光照下渲染复杂的、动画化的、详细的云形状，如图 14.29 所示。这个参与介质的云层是使用两级程序化噪声（procedural noise）来构建的。其中第一层给出了云的基本形状。第二层通过对这个形状进行腐蚀来添加细节。在这种情况下，Perlin 噪声[1373]和 Worley 噪声[1907]的混合噪声，被认为可以很好的表现菜花状（cauliflower-like）的积云和以及类似形状的云。生成这种噪声纹理的源代码和相关工具都是开源的[743, 1572]。通过使用在云层中沿着观察射线分布的样本，对来自太阳的散射光线进行积分，从而实现了光照效果。

图 14.29：使用 Perlin-Worley 噪声和光线步进进行渲染的云层，它具有动态的体积光照和体积阴影。[1572]

体积阴影可以通过对云层内若干样本的透光率进行评估，并朝向太阳进行二次光线步进来进行实现[743, 1572]。对于这些阴影样本的噪声纹理，可以对较低层级的 mipmap 进行采样，从而实现更好的性能表现，同时还可以在样本数量较少时，消除因样本不足所带来的瑕疵。通过每一帧在纹理中对来自太阳的透光率曲线进行编码（有许多技术可以实现这一点，详见章节 13.8），可以避免对每个样本进行第二次的光线步进。例如：在游戏《最终幻想 15》[416]中使用的透光率函数映射（transmittance function mapping） [341]。

如果我们想要捕捉到云的每个微小细节，那么可以在高分辨率下使用光线步进来渲染云，其开销可能会很大。为了获得更好的性能表现，可以在一个较低的分辨率下来渲

染云。一种方法是只对每个 4×4 块中的单个像素进行更新，并通过重投影前一帧中的数据来填充其余的部分像素[1572]。Hillaire [743]提出了一种变体方法，该方法总是会以固定的低分辨率来进行渲染，并在观察射线步进的起始位置上添加噪声扰动。并将前一帧的结果进行重投影，然后使用指数移动平均（EMA）的方法来与当前帧进行结合[862]。这种方法的渲染分辨率较低，但是收敛速度更快。

云的相位函数是十分复杂的[184]。这里我们将介绍两种可以实时计算它们的方法。可以将函数编码为一个纹理，并基于 θ 来对其进行采样。如果这样做会占据太多内存带宽的话，可以将章节 14.1.4 中的两个 Henyey–Greenstein 相位函数结合在一起[743]，来对该函数进行近似，其数学形式如下：

$$p_{\text{dual}}\left(\theta, g_0, g_1, w\right) = p_{\text{dual}_0} + w\left(p_{\text{dual}_1} - p_{\text{dual}_0}\right) \tag{14.17}$$

其中两个最主要的参数，散射离心率（scattering eccentricity） g_0 和 g_1 ，以及混合因子 w ，都可以由艺术家进行控制。这些参数在表现前向散射和后向散射时很重要，可以在远离光源或者朝向光源（例如太阳或者月亮）时展现云中的细节，如图 14.30 所示。

图 14.30：使用光线步进渲染的云层，并使用 Hillaire 所描述的[743]参与介质的物理表示方法，来实现动态光照和动态阴影。

有许多方法可以在云中对环境光照的散射光线进行近似。一个最直接的解决方案就是使用单一的 radiance 输入，从天空渲染一个立方体贴图纹理。还可以使用一个自下而上的、从暗到亮的渐变，来对环境光照进行缩放，从而近似云层本身的遮挡效果。也可以将这个输入的 radiance 分为底部和顶部两部分，例如地面和天空[416]。然后，我们假设云层内的介质密度是恒定的，再对两种贡献的环境光散射进行解析积分[1149]。

多重散射的近似

云看起来是明亮的和白色的，这种外观是光线在其内部发生多重散射的结果。如果没有多重散射的话，那么一个很厚的云，只会在其体积的边缘处被照亮，而在云的其他地方将会显得十分黑暗。之所以云看起来与烟雾、浓雾不同，其中多重散射是一个关键的组成部分。使用路径追踪来计算多重散射效果是十分昂贵的。Wrenninge [1909] 提出了一种在光线步进时，对多重散射现象进行近似的方法。它对散射的 o 个八度（octave）进行了积分，并将它们进行求和：

$$L_{\text{multiscat}}\left(\mathbf{x}, \mathbf{v}\right) = \sum_{n=0}^{o-1} L_{\text{scat}}\left(\mathbf{x}, \mathbf{v}\right) \tag{14.18}$$

当在计算 L_{scat} 的时候，进行了以下替换（例如使用 σ_s' 来替换 σ_s ）：

$$\sigma_s' = \sigma_s a^n, \quad \sigma_e' = \sigma_e b^n, \quad p'(\theta) = p\left(\theta c^n\right)$$

其中 a 、b 、c 的范围位于 $[0, 1]$ 中，它们是用户可以控制的参数，这些参数将会使得光线穿透参与介质。当这些参数接近 0 时，云看起来会更加柔和。在计算 $L_{\text{multiscat}}\left(\mathbf{x}, \mathbf{v}\right)$ 的时候，为了确保这种方法是能量守恒的，我们必须保证 $a \leq b$ ，不然会导致更多的光线被散射，此时方程 $\sigma_t = \sigma_a + \sigma_s$ 将不再成立，因为 σ_s 可能会大于 σ_t 。这种解决方案的优点在于，它可以在光线步进的同时，对每个不同八度的散射光线进行实时积分。图 14.31 展示了改进后的视觉效果。这种解决方案的缺点在于，当光线可以向任何方向进行散射时，这种方法不能很好的模拟复杂的多重散射行为。但是使用这种方法，云的外观质量的确得到了改善，这种方法还允许光照艺术家们可以通过几个参数来轻松地控制云的视觉效果，这得益于该方法能够实现广泛的渲染效果。通过使用这种方法，光线可以穿透介质，展现更多内部细节。

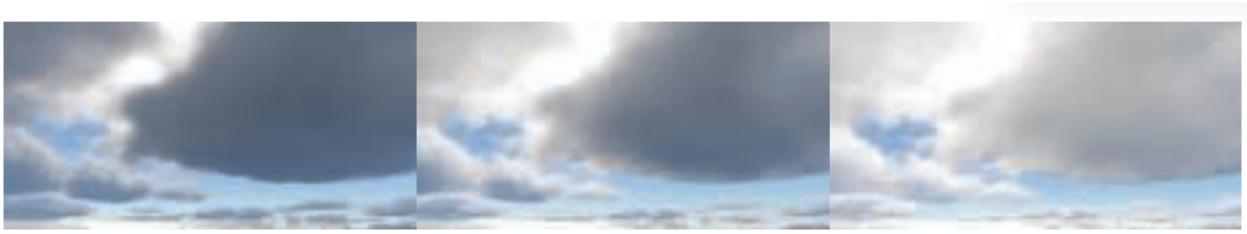


图 14.31：使用方程 14.18 对多重散射进行近似，并据此渲染的云。从左到右，n 分别为 1、2、3。这使得太阳光能够以一种可信的方式穿透云层。[743]

云与大气的相互作用

在渲染有云的场景时，为了保证视觉上的一致性，考虑云与大气散射之间的相互作用是十分重要的，如图 14.32 所示。

图 14.32：对于完全覆盖天空的云，在渲染的时候需要考虑大气散射的影响[743]。左：没有应用在云上的大气散射，这样会导致不连贯的视觉效果。中：有大气散射，但是没有阴影，这样会使得环境显得太亮。右：云层遮挡了天空，影响了光线在大气中的散射，从而生成连贯的视觉效果。[743]

由于云本身是一种大范围的场景元素，因此应当对其应用大气散射。通过在云层中采集的每个样本，可以对章节 14.4.1 中所述的大气散射进行计算，但是这样做的开销会迅速变大。相反，可以根据一个代表云层平均深度和平均透光率的单一深度，来将大气散射应用在云上[743]。

如果想要通过增加云的覆盖面积来模拟阴雨天，则应当减少云层下大气的太阳光线散射。只有穿过云层散射的光线，才会在云层下的大气中发生散射。可以通过减少天空对空气透视的照明贡献，并增加散射回大气中的光线，来对光照效果进行修改[743]。图 14.32 展示了改善后的视觉效果。

总而言之，可以通过基于物理的材质表示方式和光照来实现先进的云渲染效果。利用程序化噪声可以生成逼真的云层形状和云层细节。最后，正如本小节中所描述的，为

了获得连贯一致的视觉效果，还需要牢记一些全局的相互作用，例如云与天空之间的交互。

14.5 半透明表面

半透明（translucent）表面通常指的是具有高吸收系数和低散射系数的材质。这些材质包括玻璃杯、水或者图 14.2 中的葡萄酒。此外，本小节还将讨论具有粗糙表面的半透明玻璃（毛玻璃）材质。在许多出版物中也详细介绍了这些主题[1182, 1185, 1413]。

译者注：下面对一些容易混淆的、光学领域中的透明概念进行一些解释，来自于 Transparency and translucency – Wikipedia：

- **Transmittance**（透光率，n）：透射出的光谱能量与入射的光谱能量之比。材质的透光率与光线波长有关，不同波长的光线具有不同的透光率，因此使用 RGB 数据进行表示，包含三个分量。
- **Transparency**（透明度，n），**Transparent**（透明的，adj）：允许光线通过材质，但是不产生明显的散射现象，整体外观通常只有一种颜色。材质内部由均匀折射率的物质组成。在宏观上遵循 Snell 定律。
- **Translucency**（半透明度，n），**Translucent**（半透明的，adj）：允许光线通过材质，但是会产生明显的散射现象，整体外观通常会有颜色变化。材质内部由非均匀折射率的物质组成。在宏观上不一定遵循 Snell 定律。
- **Opacity**（不透明度，n），**Opaque**（不透明的，adj）：不会发生光线投射现象，光线无法穿过这种材质。

14.5.1 覆盖率和透光率

正如章节 5.5 中所讨论的，半透明表面可以被视为一个具有覆盖率（coverage）α 的覆盖物，例如：不透明的织物或者纤维组织，可以视为以一定的百分比隐藏了背后的物体。而对于玻璃和其他材质，我们想要计算它们的半透明度，对于不同波长的光线，它允许一定百分比的光线穿过自身的固体体积，就像是背景上的过滤器一样，这个透明度是一个与透光率（transmittance）T_r（章节 14.1.2）有关的函数。假设输出颜色为 $\mathbf{c}_o$，表面 radiance 为 $\mathbf{c}_s$，背景颜色为 $\mathbf{c}_b$，此时将表面的透明度视为覆盖率，其混合操作为：

$$\mathbf{c}_o = \alpha \mathbf{c}_s + (1 - \alpha)\mathbf{c}_b \tag{14.19}$$

而在半透明表面的情况下，混合操作将是：

$$\mathbf{c}_o = \mathbf{c}_s + \mathbf{T}_r \mathbf{c}_b \tag{14.20}$$

其中 $\mathbf{c}_s$ 包含了固体表面的镜面反射，即玻璃或者凝胶（gel）。需要注意的是，T_r 是一个包含三个分量的透光率颜色向量。为了实现彩色的半透明效果，可以使用任何现代图形 API 中的双源（dual-source）颜色混合功能，以便指定这两种输出颜色与目标缓冲中的颜色 $\mathbf{c}_b$ 进行混合。根据给定表面的反射和透光率是否为彩色，Drobot [386]提供了不同的混合操作。

在一般情况下，可以为同时指定了覆盖率和半透明的表面使用一个共同的混合操作 [1185]。在本例中所使用的混合函数为：

$$\mathbf{c}_o = \alpha \left(\mathbf{c}_s + \mathbf{T}_r \mathbf{c}_b \right) + (1 - \alpha) \mathbf{c}_b \tag{14.21}$$

当物体的厚度发生变化时，可以使用方程 14.3 来计算透射光量，该方程可以简化为：

$$\mathbf{T}_r = e^{-\sigma_t d} \tag{14.22}$$

其中 d 是光线穿过材质体积的距离。物理消光参数 σ_t 代表了光线在穿过介质时衰减的比例。为了方便艺术家们的直观创作，Bavoil [115]将目标颜色 $\mathbf{t}_c$ 设置为在给定距离 d 处的透光率大小。这样我们可以使用下列方程来重建消光系数 σ_t：

$$\boldsymbol{\sigma}_t = \frac{-\log\left(\mathbf{t}_c\right)}{d} \tag{14.23}$$

例如：当目标透光率颜色 $\mathbf{t}_c = (0.3, 0.7, 0.1)$，距离 $d = 4.0$ 米时，我们可以使用方程 14.23 计算消光系数 σ_t：

$$\boldsymbol{\sigma}_t = \frac{1}{4}(-\log 0.3, -\log 0.7, -\log 0.1) = (0.3010, 0.0892, 0.5756) \tag{14.24}$$

请注意，当透光率为 0 时，需要作为一个特殊情况来进行处理。一种解决方案是从 T_r 的每个分量中减去一个很小的，接近 0 数（epsilon），例如 0.000001。图 14.33 展示了颜色过滤的效果。

图 14.33：一个网格具有多层不同的吸收因子，所形成的半透明效果[115]。

对于一个空壳网格，其表面由一层薄薄的半透明材质组成，背景颜色会被这个半透明材质所遮挡，具体的遮挡效果是一个与光线在介质中传播路径长度 d 有关的函数。因此，沿着表面的法线方向或者切线方向进行观察，会导致不同的背景遮挡效果，因此光线的路径长度 d 是一个与表面厚度 t 和入射角度有关的函数，因为光线路径的长度也会随着入射角度而发生变化。Drobot [386]提出了这样的一种方法，其中透光率 T_r 使用以下方程进行计算：

$$\mathbf{T}_r = e^{-\sigma_t d}, \quad where \quad d = \frac{t}{\max(0.001, \mathbf{n} \cdot \mathbf{v})} \tag{14.25}$$

图 14.34 显示了渲染结果。有关薄膜表面和多层表面的更多细节，详见章节 9.11.2。

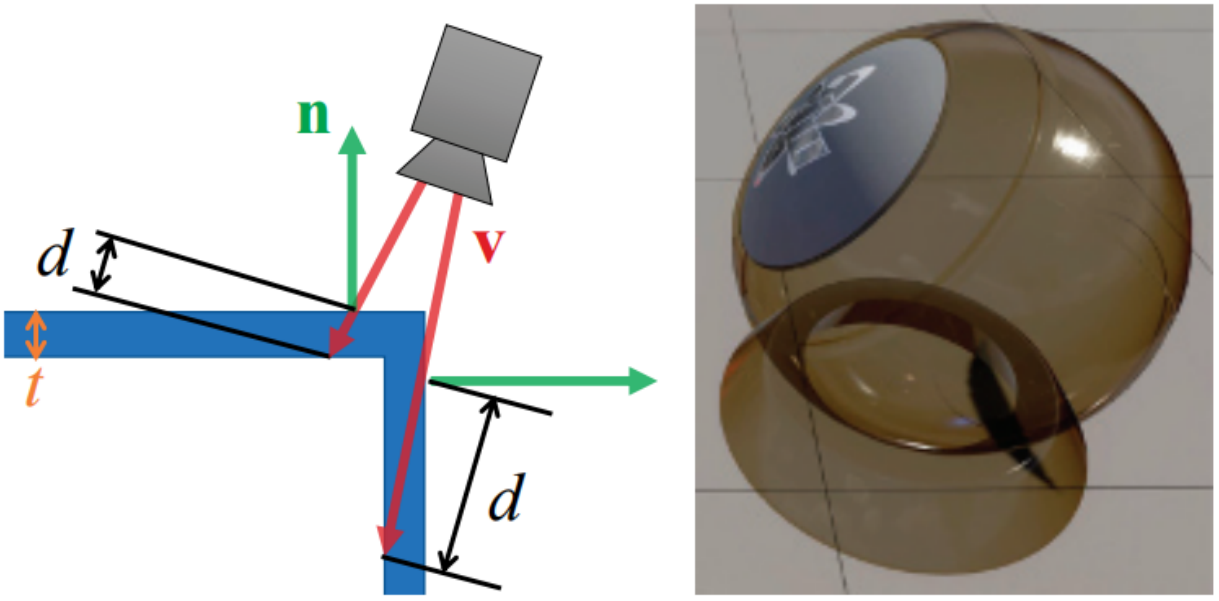


图 14.34：根据观察射线 $\mathbf{v}$ 在厚度为 t 的透明表面内传播的距离 d，来计算彩色的透光率

对于半透明的实体网格，有很多种方法可以计算光线穿过透射介质的实际距离。一个常见的方法是，首先渲染观察射线离开体积的表面。这个表面可以是水晶球的背面，也可以是海底（即水体的尽头），同时会将该表面的深度信息或者位置信息存储下来。然后再正常渲染这个体积的表面。在着色器中，访问刚才存储下来的出口深度，并计算它与当前像素表面之间的距离，然后使用这个距离来计算应用在背景上的透光率。

如果能够保证这个体积是封闭的且凸面的话，即每个像素一定会有一个入口点和一个出口点，就像水晶球一样，那么这种方法就是有效的。我们刚才提到的海床例子也是有效的，因为一旦我们离开水体，就会遇到一个不透明的表面，而在不透明表面中不会发生透射现象。对于其他更加复杂精细的模型（例如玻璃雕塑或者其他带有凹面的物体），会存在两个或者多个独立的界面跨度（span），他们都可能会吸收入射光线。使用章节 5.5 中所讨论的深度剥离（depth peeling）技术，我们可以精确地按照从后到前的顺序来渲染体积的表面。当每个正面表面被渲染的时候，我们会计算此时穿过体积的距离，并使用这个距离来计算透光率。按照这种思路，交替使用这两种方法便可以得到适当的最终透光率。请注意，如果所有的体积都是由相同浓度的相同材质所构成，同时如果表面没有反射分量的话，那么这个透光率只需要在最后使用求和距离计算一次就可以了。在最近的 GPU 上，直接在单个 pass 中存储物体片元的 A-buffer 或者 K-buffer 方法，也可以用于进一步提高效率[115, 230]。图 14.33 展示了多层透光率的例子。

对于场景中的大规模海水效果，可以直接使用场景的深度缓冲来表示底部的海床。同时在渲染透明表面的时候，必须要考虑菲涅尔效应，有关菲涅尔效应的话题，我们在章节 9.5 中进行了详细讨论。由于大多数传输介质的折射率都要明显高于空气的折射率，因此在掠射角度下，所有的入射光线都会从界面反射出来，不会有光线发生透射。图 14.35 展示了这种效果，当我们直接看向水中的时候，可以看到水下的物体；但是当我们以掠射角度看向更远地方的时候，水面基本上会隐藏波浪以下的物体。有几篇文章解释了如何处理大型水体的反射、吸收和折射现象[261, 977]。

图 14.35：在渲染水体时候，需要考虑透光率和反射率所带来的影响。当我们往下看时，由于透光率较高且呈现蓝色，因此可以看到浅蓝色的水。而在远处的地平线附近，由于透光率较低（光线需要在水中传播很远的距离），并且由于菲涅尔效应（以牺牲透射为代价而增加反射），因此海底变得不太可见。

14.5.2 折射

对于上一小节中所描述的透光率，我们会假设入射光线都来自于网格体积之外，并且沿着直线穿过这个网格体积。当网格的前后表面是平行的，并且厚度不是很大的时候（例如：窗户玻璃），这个假设是相对合理的。而对于其他透明介质而言，折射率（index of refraction）起着十分重要的作用。我们在章节 9.5 中描述了 Snell 定律（折射定律），它描述了光线在遇到网格表面时，其方向会如何发生改变。

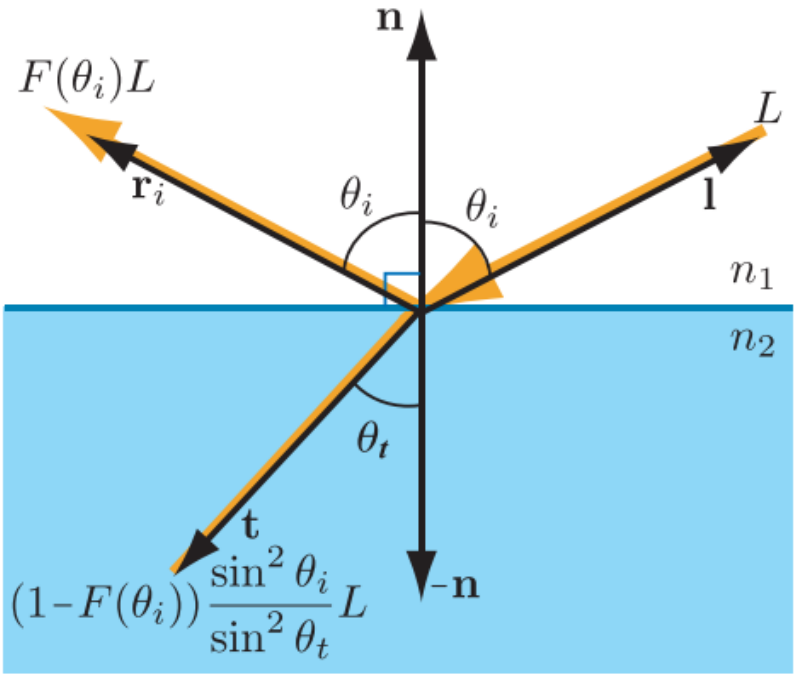


图 14.36：折射和透射的 radiance，是一个与入射角 θ_i 和透射角 θ_t 有关的函数。

由于能量守恒，任何未被反射的光线都将被透射，即透射通量（transmitted flux）与入射通量（incoming flux）的比例为 $1-f$，其中 f 是反射光量。透射 radiance 与入射 radiance 的比例是不同的，由于入射光线和透射光线的投影面积和立体角不同，因此二者的 radiance 关系为：

$$L_t = (1 - F(\theta_i)) \frac{\sin^2\theta_i}{\sin^2\theta_t} L_i \tag{14.26}$$

图 14.36 展示了这种行为。Snell 定律与方程 14.26 相结合，可以得到透射 radiance 的另一种表达形式：

$$L_t = (1 - F(\theta_i)) \frac{n_2^2}{n_1^2} L_i \tag{14.27}$$

Bec [123]提出了一种计算折射向量的有效方法。为了便于阅读（因为在 Snell 方程中，通常会使用 n 来表示折射率），这里我们定义 $\mathbf{N}$ 为表面法线，$\mathbf{l}$ 为指向光源的方向：

$$\mathbf{t} = (w - k)\mathbf{N} - n\mathbf{l} \tag{14.28}$$

其中 $n = n_1/n_2$ 为相对折射率， w 和 k 分别代表：

$$\begin{aligned} w &= n(\mathbf{l} \cdot \mathbf{N}), \\ k &= \sqrt{1 + (w - n)(w + n)}. \end{aligned} \tag{14.29}$$

最终计算得到的折射向量 $\mathbf{t}$ 会被归一化。水的折射率约为 1.33，玻璃的折射率一般在 1.5 左右，而空气的折射率则约为 1.0。

折射率会随着波长的变化而变化。透明介质会以不同的角度来弯曲各种颜色的光线，即不同颜色的光线在通过同一种透明介质的时候，各发生折射的角度各不相同。这种现象被称为光的色散（dispersion），它解释了为什么棱镜可以将白光扩散成彩虹色的光锥，以及为什么会出现彩虹现象。色散会导致透镜出现色差（chromatic aberration）问题。在摄影领域，这种现象会被称为紫边（purple fringing），在日光下沿着高对比度的边缘，这种现象会变得尤其明显。而在计算机图形学中，我们通常会忽略这种影响，因为通常而言，紫边是一种需要避免的瑕疵。为了正确地模拟这种效果，我们还需要进行一些额外的计算，因为进入透明表面的每一根光线，在色散之后都会产生一组光线，我们必须对这些光线进行追踪。因此，通常我们只会使用一根折射光线。值得注意的是，一些虚拟现实渲染器会应用一个逆色差变换（inverse chromatic aberration transform），来对头戴显示器的棱镜进行补偿[1423, 1823]。

想要模拟折射现象，一般方法是在折射物体的位置上生成一个立方环境贴图（cubic environment map，EM）。当在渲染这个物体的时候，可以通过计算正面表面的折射方向来访问这个 EM，图 14.37 展示了这样一个例子。Sousa [1675]提出了一种基于屏幕空间的方法，该方法并没有使用 EM。第一，场景会像往常一样被渲染到一个场景纹理 $\mathbf{s}$ 中，其中不包含任何的折射物体。第二，将场景纹理 $\mathbf{s}$ 的 alpha 通道清除为 1，然后再将折射物体渲染到 $\mathbf{s}$ 的 alpha 通道中；如果某个像素通过了深度测试，则向对应位置的 alpha 通道写入 0。第三，完整渲染折射物体，并在像素着色器中，根据像素在屏幕上的位置，并添加一个扰动偏移，来对场景纹理 $\mathbf{s}$ 进行采样，从而对折射现象进行模拟，例如：对表面法向切线的 xy 分量进行扰动缩放。在这种情况下，只有当 $\alpha = 0$ 时，我们才会考虑对样本的颜色进行扰动。这个测试目的是为了避免使用来自折射物体前方的样本，否则将会导致错误的顺序，即将这些样本的颜色扭曲进来，就像是它们位于折射物体后面一样。需要注意的是，也可以不用将 α 设置为 0，可以使用场景的深度贴图，来比较当前像素着色器与扰动样本的深度差异[294]。如果距离中心像素比较远的话，那么样本偏移的距离就会比较近；也可以忽略这种扰动，直接使用常规的场景样本，就好像没有发生折射一样。

图 14.37：左图：使用立方体环境贴图来模拟玻璃天使的折射现象，这个贴图本身也被用作天空盒的背景[218]。右图：玻璃球的反射现象和折射现象，具有色差效应[1696]。

这些技术可以给人一种折射的错觉，但是实际上它们与现实世界中的折射现象差距非常大。在我们刚才所描述的技术中，当光线进入透明固体的时候，光线的方向会发生改变；而当光线离开透明固体的时候，此时它并不会发生第二次弯曲，而现实世界中的光线则会再次发生弯曲。即在这些技术中，这个出口界面永远不会起到作用。这个缺陷有时候并不重要，因为人眼具有很强的包容性，可以忍受这些不太真实的现象[1185]。

许多游戏都会通过一个单独的图层来表现折射现象。对于粗糙的折射表面，会根据材质的粗糙度来对背景进行模糊处理，从而模拟由微观几何法线分布所引起的折射光线方向的扩散。在游戏《毁灭战士（2016）》中[1682]，场景首先会按照正常的方式进行渲染。然后将其降采样到一半的分辨率，并进一步降低到 4 个不同的 mipmap 层级。每个 mipmap 层级都会根据模仿 GGX BRDF 波瓣的高斯模糊来进行将采样。最后一步，在全分辨率的场景上渲染折射物体[294]。通过采样场景的 mipmap 纹理，并将材质粗糙度映射到对应的 mipmap 级别，从而将背景合成在折射表面的后面 。折射表面越粗糙，背景就越模糊。Drobot [386]也提出了相同的方法，他使用了一般的材质表示方法。在 McGuire 和 Mara [1185]的统一透明框架中也使用了类似的技术。在这种情况下，可以使用一个高斯点扩散函数（Gaussian point-spread function），在单个 pass 中对背景进行采样，如图 14.38 所示。

图 14.38：图像底部的透明玻璃具有基于粗糙度的背景散射。玻璃后面的元素看起来或多或少会有些模糊，这对折射光线的扩散现象进行了模拟。

也可以通过多个图层来处理更加复杂的折射情况。其中的每一层都可以使用存储在纹理中的深度信息和法线信息来进行渲染。然后可以使用一个基于浮雕映射原则（章节 6.8.1）的程序，来对穿过图层的光线进行追踪。存储在纹理中的深度会被视为一个高度场，每条光线都会进行光线步进，直到找到一个交点。Oliveira 和 Brauwers [1326]提出了这样的一个框架，来处理穿过网格背面的折射。此外，附近的不透明物体还可以被转换为颜色贴图和深度贴图，提供最后的一个不透明层[1927]。所有这些基于图像空间的折射方案，它们都具有一个共同的限制，即屏幕边界以外的内容无法被折射。

14.5.3 焦散和阴影

计算由折射光线和衰减光线所产生的阴影现象和焦散现象，是一项十分复杂的任务。在离线渲染环境中，有许多方法可以实现这一目标，例如双向路径追踪（bidirectional path tracing）或者光子映射（photon mapping）[822, 1413]。幸运的是，有许多方法提供了这种现象的实时近似值。

焦散（caustic）是指光线偏离其直线路径的视觉现象，例如光线通过玻璃表面或者水面之后的发散。光线偏离并发散的结果是，光线会在某些区域内散焦（defocused），从而产生阴影区域；并在其他一些区域内聚焦（focused），使得光线路径会变得更加密集，从而产生更强的入射光线。具体的光线路径和焦散现象，取决于光线所遇到的曲面。一个经典的反射焦散例子是在咖啡杯内部能够看到的心形焦散。而折射焦散的现象则更加明显，例如：光线通过水晶饰品、透镜或者一杯水，会发生聚焦和散焦，从而形成明显的明暗区域，如图 14.39 所示。光线在被弯曲的水面反射和折射之后，会在水面上和水面下都产生焦散现象。当光线汇聚时，光线将集

中在不透明表面上，并产生焦散。当在水面以下时，汇聚的光路将在水中变得可见。并且由于水体粒子对光子的散射，还会导致光束的出现。焦散是一个单独的现象因素，来自体积边界的菲涅尔效应与穿过体积时的透光率会导致光线减少，而焦散现象与这种光线减少无关。

图 14.39：真实世界中的的反射焦散（左）和折射焦散（右）。

为了在水面上生成焦散效果，可以离线生成焦散的动画纹理，并将其作为一个光照贴图应用到水面上。一般来说，这个焦散光照贴图会叠加在其他的光照贴图之上。许多游戏都采用了这种方法来模拟水面焦散效果，例如基于 CryEngine 开发的《孤岛危机 3》[1591]。一个关卡中的水面区域，是通过水体（water volume）来进行控制的。这个水体的顶部表面可以使用凹凸贴图的纹理动画或者物理模拟来生成动态的水面效果。可以利用凹凸贴图生成的法线信息，当在水面上或者水面下进行垂直投影的时候，从可以将其法线方向映射到一个 radiance 贡献上，从而生成焦散效果。使用艺术家创作的、基于高度的最大影响距离来控制因传播距离而发生的衰减。还可以对水面进行模拟，使得水面能够对世界中的物体运动做出反应，从而生成与环境情况相匹配的焦散事件。图 14.40 展示了一个水中的焦散例子。

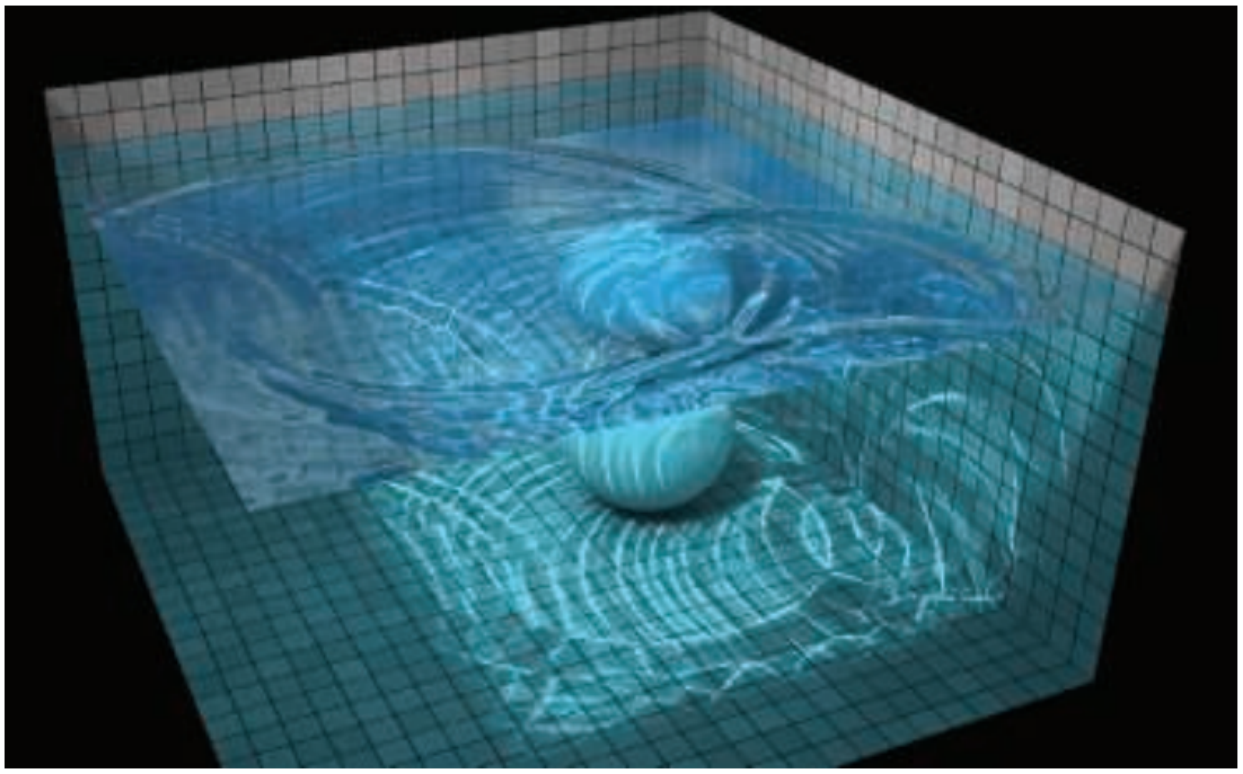


图 14.40：展示水中焦散效果的 Demo。[1831]

在水下环境中，同样的动画水面技术也可以用于模拟水体介质中的焦散效果。Lanza [977]提出了一种生成光束的两步方法。首先，从光源的视角进行渲染，将光线的位置和折射方向存储到一个纹理中。从水面的位置开始，对线条进行光栅化，在视图中沿着折射方向进行延伸。通过叠加混合将这些线条累积起来，最后使用一个后处理模糊来对结果进行模糊处理，从而掩盖因线条数量不足所产生的瑕疵。

Wyman [1928, 1929]提出了一种用于渲染焦散的图像空间技术。该方法的工作原理是，首先通过透明物体的正面和背面，来计算折射后的光子位置和入射方向。这可以通过使用章节 14.5.2 中介绍的背景折射技术来实现[1927]。然而，由于菲涅尔效应的存在，因此会将场景的交点位置、折射后的入射方向和透光率存储在纹理中，而不是在纹理中存储折射后的 radiance。纹理中的每个纹素都会存储一个光子，然后可以将这些光子以正确的强度溅射回视图中。为了实现这一目标，有两种可能的方法：在观察空间中或者在光源空间中，以高斯衰减的方式，将光子作为四边形进行溅射（splat），结果如图 14.41 所示。McGuire 和 Mara [1185]提出了一种更加简单方法来处理焦散状的阴影，即根据透明表面的法线来对透光率进行修正，由于菲涅尔效应的存在，如果垂直于入射表面，那么会具有更高的透过率，反之透过率会更低。章节 7.8 中还介绍了其他的体积阴影技术。

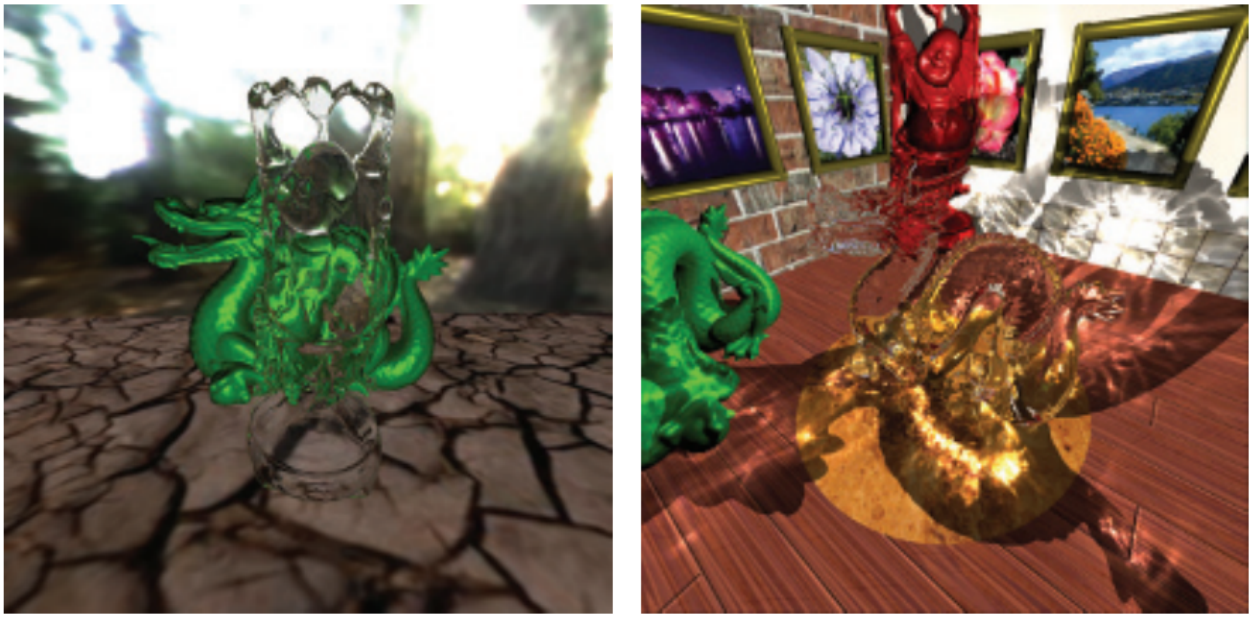


图 14.41：在左边，佛像（Buddha）折射了附近的物体和周围的天空盒[1927]。在右边，通过类似于阴影贴图的分层贴图来生成的焦散效果[1929]。

14.6 次表面散射

次表面散射（subsurface scattering，SSS）是一种复杂的物理现象，它存在于具有较高散射系数的固体材质中（详见章节 9.1.4）。这类材质包括蜡、人体皮肤和牛奶等，如图 14.2 所示。

章节 14.1 中对一般的光线散射理论进行了介绍。在某些情况下，对于那些具有较高光学深度（optical depth）的介质，光线在其内部发生散射的尺度相对较小，例如人体皮肤。散射光线会从其原始入射点附近的表面处重新发射出来。这种位置上的偏移意味着我们无法使用一个 BRDF（章节 9.9）来对次表面散射进行建模。也就是说，当散射尺度大于一个像素距离的时候，它将具有更加全局的性质。必须使用一些特殊的方法来渲染这种效果。

图 14.42 展示了光线在物体中被散射的情况。这种散射现象将会导致入射光线具有许多不同的路径。单独模拟每个光子的行为是不现实的（即使对于离线渲染而言也是如此），因此必须通过对可能的光线路径进行积分，或者对这样的积分进行近似，才可能解决这个问题。除了散射现象之外，光线在穿过材质的时候还会被部分吸收。

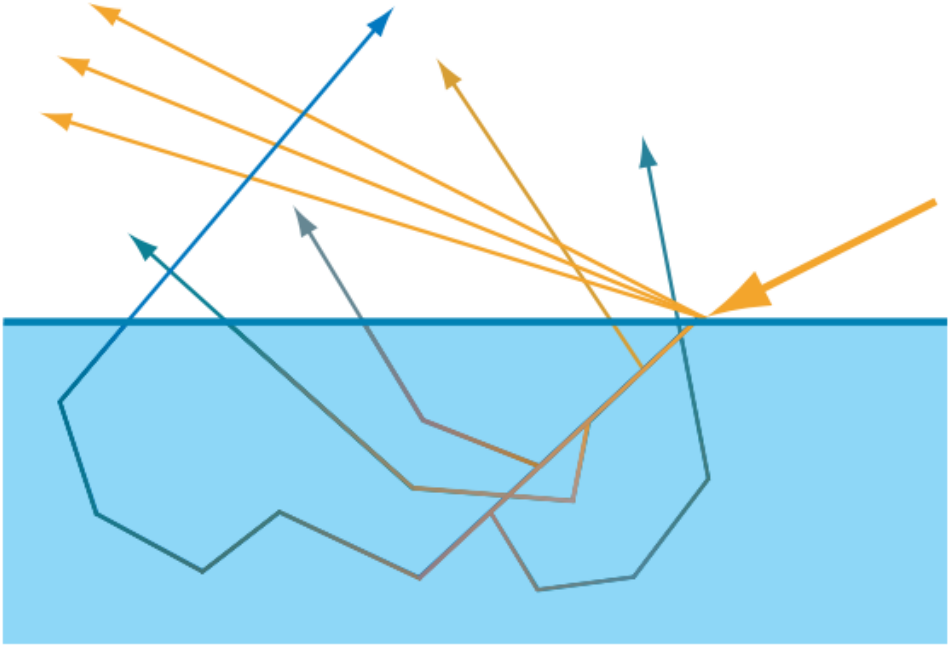


图 14.42：光线穿过物体时发生的散射现象。一开始，进入物体的光线会沿着折射方向进行传播，但是散射作用会使光线的方向发生反复改变，直到最终离开材质（或是被材质吸收）。光线通过材质的路径长度，决定了光线被吸收的百分比。

有一个重要因素可以用于区分图 14.42 中所示的各种光路，即散射事件发生的数量。对于某些光线路径而言，光线在一次散射后便可以离开材质；而对于另一些光线路径而言，光线则被散射了两次、三次或者更多次。我们通常会将散射路径划分为单次散射（single scattering）和多次散射（multiple scattering）。对于不同类型的光线路径，会使用不同的渲染技术来进行模拟。对于某些材质而言，单次散射在总体效果中所占据的比重较少，即多次散射占据了主导地位，例如皮肤。由于这些原因的存在，因此许多次表面散射的渲染技术都侧重于对多次散射进行模拟。在本小节中，我们将介绍几种次表面散射的近似技术。

14.6.1 环绕光照

环绕光照（wrap lighting）[193]可能是最简单的次表面散射方法。我们在章节 10.1 中介绍过这个技术，当时是为面光源的一种近似。当环绕光照用于近似次表面散射的时候，我们可以添加一个颜色偏移[586]，这个颜色偏移代表了光线在通过材质的时候，部分波长的光线会被材质吸收。例如：在渲染皮肤的时候，可以使用一个红色的偏移。

当以这种方式进行使用的时候，环绕照明试图模拟多重散射对曲面着色的影响。光线从相邻点“泄露”到当前的着色点上，软化了从亮到暗的过渡区域，其中较暗的区域意

味着该处表面会向着远离光源的方向进行弯曲。Kolchin 指出[922]，这种效应取决于表面的曲率（curvature），并且他推导出了一个基于物理的版本。虽然他推导出的表达式具有很高的计算成本，但是其背后的思想却是十分有用的。

14.6.2 法线模糊

Stam 指出[1686]，多次散射可以被建模为一个扩散（diffusion）过程。Jensen 等人[823]进一步发展了这一思想，并推导出了一个解析的双向表面散射分布函数（bidirectional surface scattering distribution function，BSSRDF）模型。BSSRDF 是 BRDF 在全局次表面散射情况下的推广版本[1277]。这个扩散过程对出射的 radiance 具有空间模糊效应。

这种模糊只适用于漫反射，因为镜面反射发生在材质的表面处，它不受次表面散射的影响。由于法线贴图经常用于编码小尺度上的表面变化，因此对于次表面散射而言，一个十分有用的技巧是只对镜面反射应用法线贴图[569]。而漫反射则采用光滑的、无扰动的法线。由于这种技巧并不会带来额外的计算开销，因此在使用其他次表面散射方法的时候，同样可以使用这个技巧。

对于许多材质而言，多次散射通常只会发生在相对较小的距离内。皮肤是其中的一个重要例子，对于皮肤这样的材质，大多数次表面散射都会发生在距离表面几毫米的范围内。对于这样的材质，使用无扰动的漫反射着色法线就足够了。Ma 等人[1095]基于实际测量的数据，对这种方法进行了扩展。他们求解了来自散射物体的反射光线，并发现镜面反射是基于几何表面法线的，而由于次表面散射现象的存在，会使得漫反射看起来像是使用了模糊过的表面法线一样。此外，在可见光谱的范围内，不同波长光线的模糊程度会有所不同。他们提出了一种实时的着色技术，在这个技术中，对于镜面反射项以及漫反射项的 R, G, B 通道，都会使用独立获取的法线贴图[245]。为每个通道都使用不同的法线贴图将会导致颜色溢出现象。由于这些漫反射法线贴图通常类似于镜面法线贴图的模糊版本，因此可以对这个技术进行一些修改，即使用单个法线贴图，同时调整该法线贴图的 mipmap 层级，使用模糊版本的法线贴图来用于漫反射着色，这样做的代价是失去了颜色偏移效果，因为现在每个通道所使用的法线都是相同的。

14.6.3 预积分皮肤着色

Penner [1369]将环绕光照（wrap lighting）和法线模糊（normal blurring）的思路结合起来，提出了一种预积分的皮肤着色解决方案。

材质的散射率和透光率会被预先积分，并存储在一个二维查找表中。这个 LUT 的第一个轴是基于 $\mathbf{n} \cdot \mathbf{l}$ 进行索引的；第二个轴是基于 $1/r = \|\partial n/\partial p\|$ 进行索引的，它代表了表面的曲率。表面的曲率越高，对透射颜色和散射颜色的影响就越大。由于网格上每个三角形的曲率都是恒定的，因此这些表面曲率值必须进行离线烘焙和平滑处理。

为了处理次表面散射对小尺度表面细节的影响，Penner 对 Ma 等人提出的技术进行了修改，Ma 等人[1095]所提出的技术在上一小节中进行了讨论。Penner 并没有为漫反射的 R, G, B 通道使用单独的法线贴图，而是根据每个颜色通道中次表面材质的扩散配置，来对原始的法线贴图进行模糊，从而生成所需的独立法线贴图。由于使用了四个独立的法线贴图，因此该方法是内存密集型的（memory intensive），为了对其进行一些优化处理，Penner 使用了一个单一的平滑法线贴图，并与每个颜色通道的顶点法线相混合。

这种技术会忽略光线在阴影边界上的扩散，因为在默认情况下，它只会依赖于曲率来对着色进行调整。为了使得散射配置（scattering profile）能够跨越阴影边界，可以使用阴影的半影配置来对 LUT 坐标进行偏移。通过应用上述的改进，这种十分快速的技术在质量上能够近似于下一小节中将介绍的高质量方法[345]。

14.6.4 纹理空间扩散

对漫反射法线进行模糊处理可以实现多重散射的一些视觉效果，但是无法实现其他的效果，例如软化的阴影边缘等。可以使用纹理空间扩散（texture-space diffusion）的相关技术来解决这些限制。这个想法最早是由 Lensch 等人[1032]提出的，当时是作为另一种技术的其中一部分，但是后续 Borshukov 和 Lewis [178, 179]所提出的版本所具有的影响力最大。他们将多重散射的想法形式化为一个模糊过程。首先，将表面的 irradiance（漫反射光照）渲染到一个纹理中，这是通过在光栅化阶段使用纹理坐标来作为渲染位置实现的。真实的位置会单独进行插值，从而可以在着色阶段使用。这个纹理会被进行模糊处理（滤波），并在后续渲染时用于计算漫反射着色。模糊处理中所使用的滤波器，其形状和大小取决于材质和波长。例如：对于皮肤材质而言，R 通道所使用的滤波器要比 G 通道或者 B 通道更宽，这会使得在靠近阴影边缘的附近变红。对于大多数材质而言，用于模拟次表面散射的适当滤波器通常都具有一个较窄的中心尖峰，以及一个宽而浅的底部。这项技术最初被应用于离线渲染中，但是 NVIDIA [345, 586]和 ATI [568, 569, 803, 1541]的研究人员，很快就提出了基于 GPU 的实时实现。

在 d'Eon 和 Luebke 的演讲展示中[345]，他们介绍了此类技术中最完整的解决方案（之一），该方法支持复杂的滤波器，可以用于模拟多层次表面结构的效果。Donner 和 Jensen 表明[369]，这样的结构可以生成最为真实的皮肤渲染效果。这个完整的 NVIDIA 皮肤渲染系统由 d'Eon 和 Luebke 提出，它可以生成出色的效果（如图 14.43 所示），但是其计算开销很高，需要大量的模糊 pass。但是，该系统可以很容易地缩减计算规模，从而提高性能表现。

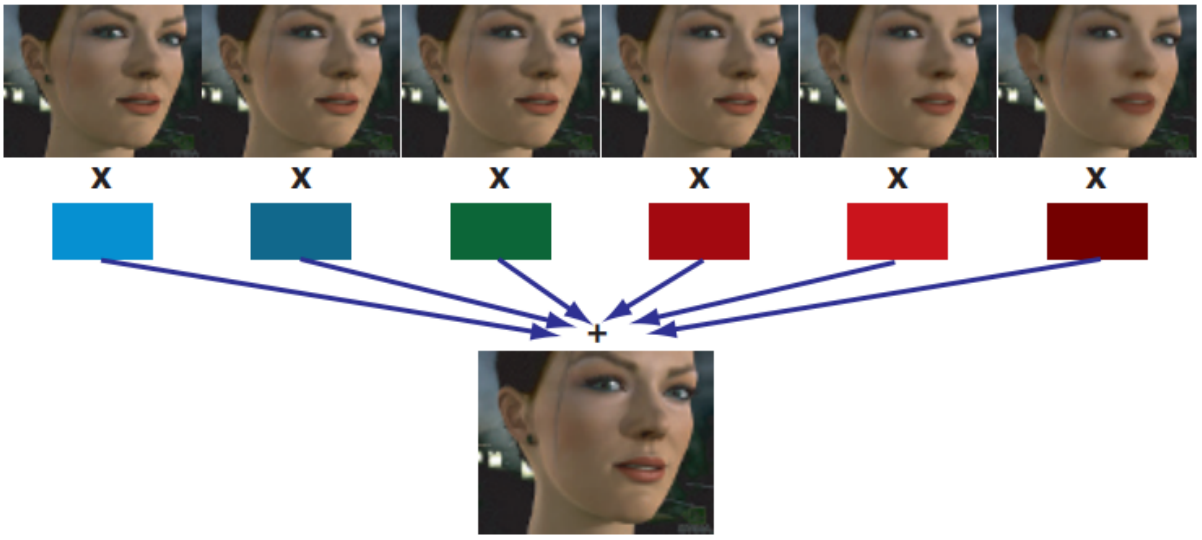


图 14.43：纹理空间的多层扩散。利用 RGB 权重来对 6 种不同的模糊图像进行组合。最终的图像会在这个线性组合的基础上，再加上一个高光项。[345]

Hable [631]提出了一个包含 12 个样本的滤波核，用于取代多个高斯 pass。这个滤波器可以作为一种预处理应用于纹理空间中；也可以在进行网格光栅化的时候，应用于像素着色器中。这种方法可以使得人脸渲染的速度变得更快，但是代价是会损失一些真实感。当靠近人脸进行观察的时候，较低次数的采样会产生肉眼可见的色带。但是在中等距离上进行观察的时候，这种方法所带来的质量损失可以忽略不计。

14.6.5 屏幕空间扩散

对场景中的所有网格渲染一个光照贴图，并对其进行模糊处理，这样做的开销会很大，无论是计算开销还是内存开销。此外，在一帧的渲染中，场景中的网格需要进行两次渲染，一次是在光照贴图中，另一次则是在相机视图中。同时这个光照贴图还需要具有一个合理的分辨率，从而能够在小尺度的细节中表现次表面散射现象。

为了解决这些问题，Jimenez 提出了一种基于屏幕空间的方法[831]。

1. 像往常一样渲染一遍场景，而场景中需要表现次表面散射现象的网格（例如人脸），将会在模板缓冲区（stencil buffer）中进行标注。

2. 对于存储下来的 radiance，应用一个两 pass 的屏幕空间处理，来模拟次表面散射现象，并通过使用模板测试，从而只在需要进行次表面散射的地方（包含半透明材质的像素）应用这个开销很大的算法。

在额外的两个 pass 中，会在水平方向上和垂直方向上应用两个一维的双边模糊滤波核。彩色的模糊滤波核是可分离的，但是由于以下两个原因，它无法以一种完全分离的方式来进行应用。

1. 必须要考虑线性的观察深度，从而根据表面距离来将模糊核拉伸到正确的宽度。
2. 双边滤波避免了不同深度材质中的光线泄漏，即光线不应当在不发生相互作用的表面之间进行泄露。

此外，为了使得这个模糊滤波器不仅能够应用于屏幕空间中，而且还能够应用于表面的切线空间中，它必须还要考虑表面的法线方向。最后，虽然由于以上这些原因，使得模糊滤波器的可分离性成为一种近似，但是最终的结果仍然是高质量的。后来提出了一种改进的可分离滤波器[833]。由于该算法依赖于屏幕上的材质区域面积，因此对于渲染人脸特写而言开销很大。但是这种成本在某种意义上来说是合理的，因为我们期望的就是这些区域内可以有很高质量的次表面渲染效果。当场景包含多个角色的时候，这个算法尤其有价值，因为场景中的人物角色会被同时处理，如图 14.44 所示。

图 14.44：高质量渲染的扫描人脸模型。屏幕空间中的次表面散射可以在单个后处理中，对多个角色进行处理，渲染逼真的人体皮肤材质。[831]

为了进一步优化处理过程，可以将线性深度存储在场景纹理的 alpha 通道中。由于这个一维模糊只依赖于少量的样本，因此会在近距离的人脸特写上看到欠采样的现象。为了避免这个问题，可以将这个滤波核进行逐像素的旋转，这样做能够隐藏之前的鬼

影瑕疵，代价是会出现一些噪声[833]。可以使用时域抗锯齿（章节 5.4.2）来显著降低这个噪声。

在实现屏幕空间扩散的时候，我们需要格外注意，只对 irradiance 进行模糊处理，而不是漫反射反照率（albedo）或者镜面光照（高光）。一种实现这个目标的方法是，将 irradiance 和镜面光照渲染到独立的屏幕空间缓冲区中。如果使用延迟渲染（章节 20.1）技术的话，那么 G-buffe 中自然就包含了漫反射反照率。为了减少内存带宽的压力，Gallagher 和 Mittring [512]建议使用棋盘格模式（checkerboard pattern），来将 irradiance 和镜面光照存储在单个缓冲区中。在 irradiance 被模糊之后，通过将漫反射反照率与模糊的 irradiance 相乘起来，并在上面叠加镜面光照，从而合成最终图像。

在这个屏幕空间的算法框架内，还可以渲染大尺度的次表面散射现象，例如：穿过鼻子或者耳朵的光线。在渲染网格的漫反射照明时，Jimenez 等人[827]所提出的技术还通过利用反向的表面法线 $-\mathbf{n}$，对来自背面的入射光线进行采样，从而增加了来自背面次表面传输的贡献。最终的计算结果会通过透光率来进行修正，这个透光率是通过对一个传统的阴影贴图（从光源的视角进行渲染）进行采样来计算的，这类似于 Dachsbacher 和 Stamminger [320]的方法，我们将在下一小节中对其进行介绍。为了表现光线在一个圆锥体范围内的前向散射，可以对阴影贴图进行多次采样。为了能够使用一个较低的像素样本数来降低渲染成本，可以使用两个逐像素随机偏移或者随机旋转的阴影样本。这样做将会产生许多不必要的视觉噪声。但是值得庆幸的是，我们会在屏幕空间中使用次表面模糊滤波核来实现半透明的次表面光线扩散，而噪声会在这个过程中被自动过滤掉。因此，在每个光源只有一个额外深度贴图样本的情况下，可以模拟光线通过面部较薄的部分时，在圆锥体中的前向散射，从而实现高质量的半透明效果。

14.6.6 深度贴图技术

到目前为止我们所讨论的技术，都是在相对较小的距离上来模拟光线散射，例如皮肤。对于表现出大尺度光线散射的材质，我们还需要其他技术来进行模拟，例如：光线在手中的传播。其中的许多技术都着重关注单次散射的情况，因为这要比多次散射更加容易建模。

图 14.45 左侧展示了大尺度单次散射的理想模拟结果。由于折射现象的存在，因此光线在进入和离开物体时都会改变传播方向。我们需要将所有路径的效果叠加起来，才能为单个表面点进行着色。同时还需要考虑光线的吸收效应，每条路径的光线吸收量取决于该路径在材质内部的长度。即使对于离线渲染器而言，对单个着色点追踪所有

的折射光线也是非常昂贵的，因此我们通常会忽略进入材质时的折射现象，只考虑光线在离开材质时的方向变化[823]。由于我们总是会在入射光线的方向上来投射射线，因此 Hery 指出[729, 730]，我们可以直接使用光源空间的深度贴图（通常用于渲染阴影）来找到这个入射光线的方向，而不需要进行真正的光线投射操作，如图 14.45 的中间部分所示。对于根据相位函数来对光线进行散射的介质，光线的散射角度同样也会影响散射光线的数量。

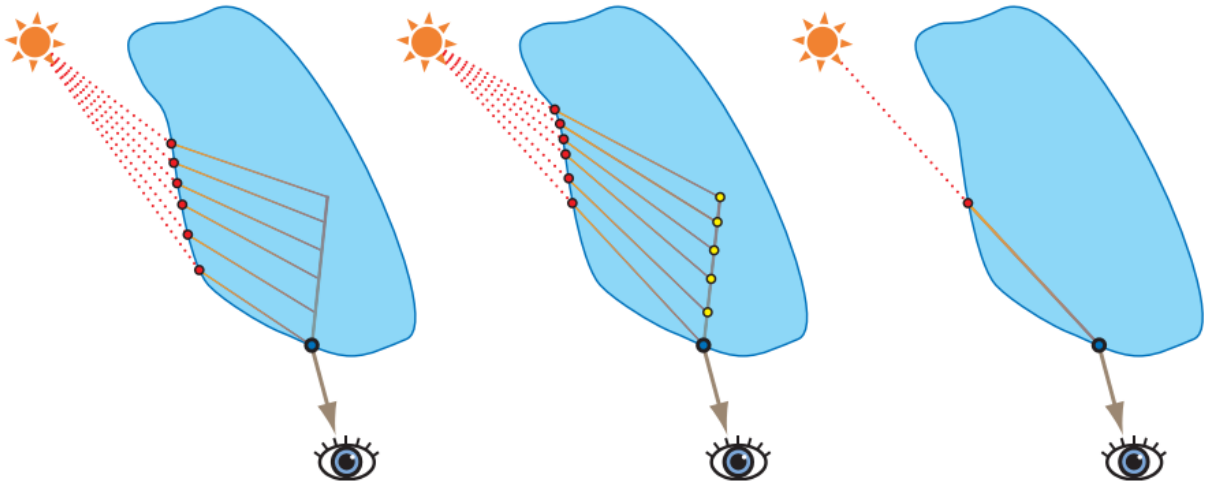


图 14.45：左图展示了最理想的情况，光线会在进入物体和离开物体的时候发生折射；通过材质内的光线步进，对离开物体时光线折射的散射贡献进行正确收集。同时在计算消光系数 σ_t 的时候，考虑每条光线路径的长度。这可以通过路径追踪或者一些实时近似方法来进行实现[320]。中间的图像则展示了一种计算起来更加简单的情况，即光线只会在离开物体的时候发生折射。这是实时渲染中常用的近似方法，因为想要从折射的采样点（黄色）找到对应的光线入口点（红色）是一件很困难的事情。右图展示了一个更加简化的近似方法，其计算速度更快，这种近似方法只考虑了一条光线，并没有考虑沿着折射光线进行多次采样[586]。

执行深度贴图的查找操作，其速度要比光线投射更快，但是由于 Hery 的方法需要检索多个样本，因此对于大多数的实时渲染应用程序而言，这个方法的速度还是太慢了。Green [586]提出了一个更快的近似方法，如图 14.45 右侧所示。虽然这种方法并不是基于物理的，但是生成的结果还是令人信服的。该方法的一个问题在于，物体背面的细节会被显示出来，因为物体的厚度变化会直接影响着色结果的颜色。尽管如此，Green 的近似方法仍然足够有效，皮克斯将其应用在了《美食总动员》等电影中[609]。皮克斯将这种技术称为软糖光源（gummi light）。Hery 方法的另一个问题是，深度贴图无法包含多个物体，也无法包含高度非凸（non-convex）的物体。这是因为我们假设着色点（蓝色）和物体交点（红色）之间的整个光线路径都位于物体内部。皮克斯通过使用一种深度阴影贴图（deep shadow map）的方法来解决这个问题[1066]。

由于每个表面点都可能会受到来自其他表面点的光线影响，因此想要实时模拟大尺度的多重散射是相当困难的。Dachsbacher 和 Stamminger [320]提出了一种阴影映射的扩展方法来模拟多重散射现象，这个方法被称为半透明阴影映射（translucent shadow mapping）。该方法会将一些额外信息存储在光源空间的贴图中，例如 irradiance 和表面法线等。然后从这些纹理（包括深度贴图）中提取几个样本，并将其结合起来，从而形成对散射 radiance 的估计。在 NVIDIA 的皮肤渲染系统中 [345]，使用了这种技术的一种改进版本。Mertens 等人[1201]提出了类似的方法，不同之处在于，该方法使用了屏幕空间中的纹理，而不是光源空间中的纹理。

树叶也表现出强烈的次表面散射效应，当光线从树叶背面入射进来的时候，树叶会呈现出明亮的绿色。除了反照率纹理和法线纹理之外，还可以将存储树叶体积透光率 T_r 的纹理映射到树叶表面上[1676]。然后，可以使用一个特殊（ad hoc）的模型来近似光线的额外次表面贡献。由于树叶是比较薄的场景元素，因此可以使用负的表面法线，来近似树叶背面的法线 $\mathbf{n}$ 。此时背面光线的贡献可以计算为： $(\mathbf{l} \cdot -\mathbf{n})^+ \cdot (-\mathbf{v} \cdot \mathbf{l})^+$ ，其中 $\mathbf{l}$ 是光线方向， $\mathbf{v}$ 是观察方向。然后将其与表面反照率相乘，并叠加到直接光照的贡献上。

以类似的方式，Barre-Brisebois 和 Bouchard [105]提出了一种廉价的特定近似方法，该方法可以网格上模拟大尺度的次表面散射效果。首先，对于每个网格，他们会生成一个存储平均局部厚度（averaged local thickness）的灰度纹理（grayscale texture），这个平均局部厚度的值，等于 1 减去使用内法线 $-\mathbf{n}$ 计算的环境光遮蔽值。这个纹理被称为 t_{ss} ，可以将其认为是一种透光率的近似值，可以将其应用在从表面背面入射来的光线上。最后将次表面散射添加到常规的表面光照中，这里的次表面散射可以使用如下方程进行计算：

$$t_{\text{ss}}\mathbf{c}_{\text{ss}}\left((\mathbf{v} \cdot -\mathbf{l})^+\right)^p \tag{14.30}$$

方程中的 $\mathbf{l}$ 是归一化的光线向量， $\mathbf{v}$ 是归一化的观察向量， p 为用于近似相位函数的指数（如图 14.10 所示）， $\mathbf{c}_{\text{ss}}$ 为次表面反照率。然后将这个表达式与光线的颜色、光线的强度和随距离的衰减相乘在一起。这个近似模型并不是基于物理的，也不是能量守恒的，但是它能够在一个 pass 中，快速渲染看起来较为合理的（plausible）次表面光照效果，如图 14.46 所示。

图 14.46：左侧展示了为 Hebe 雕像生成的局部厚度纹理。中间展示了使用这个局部厚度纹理来生成的次表面光线散射效果。右边展示了另一个半透明的立方体场景，这些立方体使用了相同的次表面散射技术进行渲染。[105]

14.7 毛发和皮毛

毛发（hair）是指从哺乳动物的真皮层生长出来的蛋白质细丝。就人类而言，毛发分散在身体的不同部位上，包括头顶、胡须（beard）、眉毛（eyebrow）和睫毛（eyelash）等。而其他哺乳动物通常都会被皮毛（fur，浓密的、有限长度的毛发）所覆盖，动物身上不同部位的皮毛往往会具有不同的属性。头发可以是直的（straight）、波浪的（wavy）、或者卷曲的（curly），每一种头发都有着不同的强度和粗糙度。天然的头发可以是黑色、棕色、红色、金色、灰色或者白色，还可以被人工染成彩虹中的所有颜色（成功率不同）。

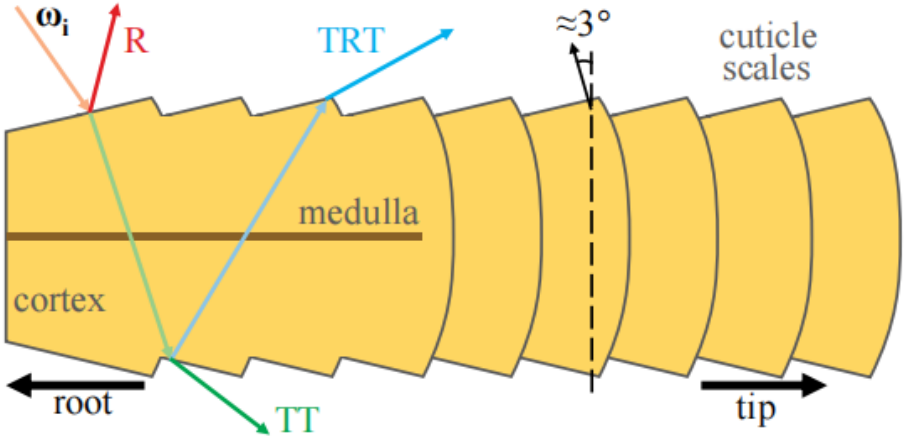


图 14.47：一根毛发的纵向剖面，展示了组成它的不同材质，以及沿方向 ω_i 的入射光线所产生的光照分量。

> 译者注：这里引用了两篇论文，其中论文 1052 是闫令琪老师的。

毛发（hair）和皮毛（fur）的结构基本上是一样的，它由 3 层组成[1052, 1128]，如图 14.47 所示：

- 最外面是角质层（cuticle），它代表纤维的表面。这个表面是粗糙的，它由重叠的鳞片所组成，与毛发的方向相比，倾斜角大约为 $\alpha = 3^\circ$，这使得其法线会向着毛发的根部倾斜。
- 中间的是皮层（cortex），它含有黑色素，为毛发纤维赋予了颜色[346]。其中一种色素是真黑素（eumelanin），它负责产生棕色，$\sigma_{a,e} = (0.419, 0.697, 1.37)$；另一种色素是棕黑色素（pheomelanin），它负责产生红色，$\sigma_{a,p} = (0.187, 0.4, 1.05)$。
- 最内层是髓质（medulla）。髓质很小，在建模人类毛发时通常会被忽略[1128]。但是髓质在动物皮毛中的占比更大，具有更大的意义，在建模动物皮毛的时候不能忽略[1052]。

我们可以将毛发纤维视为类似于粒子的东西，即一个体积的离散化表示，不同之处在于这里我们使用曲线代替点。利用双向散射分布函数（bidirectional scattering distribution function，BSDF），我们可以描述毛发纤维与光线之间的相互作用。BSDF 与 BRDF 相对应，不同之处在于，BSDF 会将光线在球面区域内进行积分，而不仅仅是在半球范围内。BSDF 通过不同的层，聚集了毛发纤维内发生的所有相互作用，它在章节 14.7.2 中进行了详细介绍。光线会在纤维中发生散射，同时也会被其他纤维所反射，在这个多重散射现象中会产生复杂的彩色 radiance。此外，由于纤维会吸收光线，这是一个与其材质和色素有关的函数，因此在毛发内部表现体积自阴影现象也很重要。在本小节中，我们对最近的技术进行介绍，这些技术使得我们能够渲染较短的毛发，例如胡须，头毛和皮毛等。

14.7.1 几何和 Alpha

毛发可以被渲染为一个压扁的毛发四边形（hair quad），围绕由艺术家绘制的毛发引导曲线（hair guide curves），会在顶点着色器中生成一个四边形的丝带（ribbon）。每条四边形丝带都会沿着对应的毛发引导曲线，从而代表一束毛发，并按照特定的方向跟随皮肤一起运动[863, 1228, 1560]。这种方法非常适合模拟胡须，或者是模拟长度较短、且大部分都是静态的毛发。这种方法的效率也很高，因为较大的四边形会占据更多视野面积，因此只需要少量的丝带就足以覆盖整个头部，这反过来又提高了性能表现。如果需要表现更多的细节，则可以渲染数千条更薄更细的四边

形丝带，例如：通过物理模拟生成又细又长的头发。在这种情况下，最好是使用沿着毛发曲线切线的圆柱体约束，来使得生成的四边形面向观察方向[36]。即使只使用很少毛发引导线来模拟毛发，也可以通过对周围毛发引导线的属性进行插值，来实例化新的头发[1954]。

所有这些四边形丝带元素，都可以被渲染为 alpha 混合的几何图形。如果使用 alpha 混合的话，就需要保证正确的毛发渲染顺序，不然会出现透明度瑕疵（章节 5.5）。为了缓解这个问题，可以使用预先排序的索引缓冲区（index buffer），首先渲染那些靠近头部的毛发，然后再那些渲染远离头部的毛发。这种策略适用于较短的、静态的毛发，不适合用于渲染长的、相互交错的、动态的毛发。可以使用 alpha 测试和深度测试来解决这个排序问题；但是这样可能会导致高频几何图形和纹理出现严重的锯齿问题。可以使用 MSAA 来对每个样本进行 alpha 测试[1228]，相应的代价是额外的样本和更大的内存带宽。另外，也可以使用任何顺序无关的透明度方法，例如章节 5.5 中所介绍的方法。例如：TressFX 存储了 $k = 8$ 个最近的片元[36]，并在像素着色器中进行更新，来保证前 7 层是有序的，从而实现多层 alpha 混合[1532]。

另一个问题是来自 mipma 过程中的 alpha 缩小（章节 6.6），这会导致 alpha 测试出现瑕疵。有两种方法可以解决这个问题，第一种是执行更加智能的 mipmap 生成，从而保证 alpha 不出现错误。第二种是使用一个更加高级的哈希 alpha 测试[1933]。在渲染细长毛发的时候，也可以根据毛发的像素覆盖率来对不透明度进行修改[36]。

诸如胡须、睫毛和眉毛这样的细小毛发，要比头发更加容易渲染。睫毛和眉毛甚至可以被处理成几何皮肤，拥有真实的网格，从而配合头部和眼睑的运动。这些小元素上的毛发表面可以使用一个不透明的 BRDF 材质来进行光照计算。当然也可以使用一个 BSDF 来对毛发进行着色，我们将在下一小节中进行介绍。

14.7.2 毛发

Kajiya 和 Kay [847]开发了一个 BRDF 模型，用于渲染一个由有组织的、无限小的圆柱形纤维所组成的体积。我们在章节 9.10.3 中讨论过这个模型，它最初是用于在一个代表表面密度的体积纹理上，通过光线步进来渲染毛茸茸的元素。这个 BRDF 代表了体积对镜面光照和漫反射光照的响应，它也可以用于毛发的渲染。

Marschner 等人[1128]进行了一项开创性的工作，他们测量了人类毛发纤维中的光线散射，并根据这些观察结果提出了一个模型。他们在一根毛发中观察到了不同的散射分量，如图 14.47 所示。

1. 首先， R 分量代表了光线在角质层（空气/纤维界面）上的反射，这会导致一个向毛发根部偏移的白色镜面峰值。
2. 其次， TT 分量代表了光线通过毛发纤维的传播路径，第一次从空气折射到毛发材质中，第二次从毛发材质折射回空气中。
3. 最后， TRT 分量代表了光线首先经过折射，在毛发纤维中进行传播，然后被纤维的另一面所反射，最后再次经过折射，在毛发材质外进行传播。

这些变量名中的“ R ”代表一次内反射（internal reflection），“ T ”代表一次透射。 TRT 分量可以被认为是一个次级镜面高光，它与 R 相比发生了位移，并且由于光线在穿过纤维材质时被部分吸收，因此具有了颜色。

在视觉上， R 分量看起来是毛发上的无色镜面反射。当大量毛发从背后被照亮时， TT 分量看起来是一个明亮的亮点。而 TRT 分量对于真实的毛发渲染而言至关重要，因为它会导致毛发上的偏心闪烁，也就是说，在现实生活中，毛发的横截面并不是一个完美的圆形，而是一个椭圆形。这种闪烁对于真实感而言非常重要，因为它们可以使得毛发看起来没有那么均匀，如图 14.48 所示。

图 14.48：左侧金色头发的参考图像是使用路径追踪进行渲染的。右侧棕色头发是实时渲染的，由于纤维的偏心而呈现出镜面闪烁效果。[346]

Marschner 等人[1128]提出了一组函数，它将 R 、 TT 和 TRT 分量建模为毛发 BSDF 的一部分，用于表示毛发纤维对光线的响应。这个模型还恰当地考虑了透射和反射时的菲涅尔效应，但它忽略了 $TRRT$ 、 $TRRRT$ 以及其他更长、更复杂的光线路径。

然而，这个原始模型并不是能量守恒的，d'Eon 等人[346]对其进行了研究和修正。通过更好地考虑粗糙度和高光锥形范围的收缩，对 BSDF 进行了重新设计，并保证了能量守恒。同时对这些分量也进行了扩展，可以包含更长的光线路径，例如 $TR*T$ 。透射率也可以通过测量获得的黑色素（melanin）消光系数来进行控制。与

Marschner 等人[1128]的工作类似，他们的模型能够很好地表现偏心线段上的闪烁。Chiang 等人[262]提出了另一种能量守恒的模型。这个模型给出了参数化的粗糙度和多重散射颜色，这对艺术家来说更加直观，可以很方便地控制毛发的视觉表现，而不是让他们去调整高斯分布的方差或者黑色素的浓度系数。

艺术家们可能想要为角色的毛发创建一个特殊的镜面效果，例如通过改变粗糙度参数等。使用一个基于物理的、能量守恒的模型，毛发体积深处的散射光线也会发生一些变化。为了提供更多的艺术控制选项，可以将前几个散射路径（R，TT，TRT）和多重散射项分开进行处理[1525]。这可以通过维护第二组 BSDF 参数来进行实现，这组参数仅用于多重散射路径。此外，BSDF 的 R，TT 和 TRT 分量可以使用简单的数学形状来进行表示，这些数学形状可以被艺术家们理解，艺术家们通过调整这些形状，来对毛发的外观进行直观的控制。根据入射方向和出射方向来对 BSDF 进行归一化处理，使得整个系统仍然能够保持能量守恒。

上面所给出的每个 BSDF 模型都很复杂，计算成本也很高，并且它们主要用于电影制作中，一般使用路径追踪来进行渲染。值得庆幸的是，也存在一些可以实时计算的 BSDF 模型。Scheuermann 提出了一种特别的 BSDF 模型[1560]，这个模型易于实现，渲染速度很快，并且在将毛发渲染为较大的四边形丝带的时候，看起来令人信服。更进一步，通过将 BSDF 存储在一个 LUT 纹理中，并使用输入方向和输出方向作为参数进行索引[1274]，可以在实时环境中使用 Marschner 的模型[1128]。然而，这种方法很难渲染具有空间变化的毛发外观。为了避免这个问题，最近有一个基于物理的实时模型[863]，它使用简化的数学表达来对之前工作中的分量进行近似，从而获得了令人信服的渲染结果，如图 14.49 所示。然而，与离线渲染的结果相比，所有这些实时的毛发渲染模型，在质量上还存在一定的差距。这些简化的实时算法通常不具备高级的体积阴影或者多重散射。而这两种效果对于吸收率低的毛发而言尤其重要，例如金色头发。

图 14.49：实时渲染的毛发，R，TT，TRT 以及多重散射分量。[863, 1802]

在考虑体积阴影的情况下，最近的一些解决方案[36, 863]依赖于一个透光率值，根据恒定的吸收率 σ_a，使用 d 来作为沿着光线方向上从遇到的第一根毛发到当前纤维的距离，来计算这个透光率值。这种方法十分实用且直接，因为它仅仅依赖于引擎中任何可用的阴影贴图即可。然而，这种方法无法表达由毛发团块所引起的局部密度变化，这个密度变化对于明亮的毛发而言尤其重要，如图 14.50 所示。为了解决这个问题，可以使用一个体积阴影的表示方法（详见章节 7.8）。

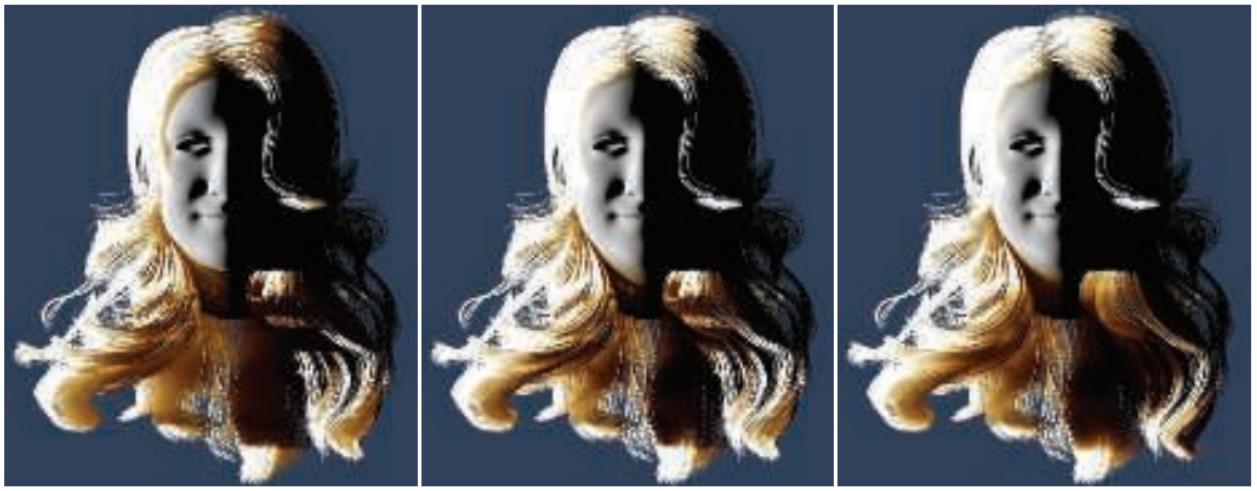


图 14.50：左：使用与第一个遮挡物不同的深度以及恒定的消光系数，会导致过于平滑的体积阴影。中间：使用深度阴影贴图[1953]可以实现更多的透光率变化，并使这个变化的透光率与毛发体积内的聚集程度相匹配。右：基于到第一个遮挡物的距离，将深度阴影贴图与 PCSS 相结合，可以实现更加平滑的体积阴影效果（更多细节详见章节 7.6）。[781]

在渲染毛发的时候，多重散射效果是计算开销很大的一项，适合用于实时实现的解决方案并不多。Karis [863]提出了一种近似多重散射的方法，这个特别的模型使用了假法线（类似于环境法线 bent normal）与环绕漫反射光照，并且在将毛发的基本颜色与光照相乘之前，将其提高到一个与深度相关的幂次，从而对光线散射通过许多股头发后的颜色饱和度进行近似。

Zinke 等人[1972]提出了一种更加先进的双重散射（dual-scattering）技术，其渲染结果如图 14.51 所示。这种方法之所以是双重（dual）的，是因为它会根据两个因素来计算散射光线的数量。第一，在着色像素与光源位置之间，将每根毛发束的 BSDF 结合起来，来计算一个全局的透光率因子 Ψ^G，这个因子 Ψ^G 给出了着色位置上入射 radiance 的透光率。可以在 GPU 上统计毛发的数量，以及计算光线路径上的平均发束方向来计算这个值 Ψ^G，后者（平均发束方向）会对 BSDF 产生影响，因此它也对透光率产生影响。可以使用深度不透明度映射[1953]，或者占用率贴图[1646]来对这些数据进行积累统计。第二，局部散射分量 Ψ^L 对这样一个事实进行了近似，即着色位置处的透射 radiance 将在当前位置周围的毛发纤维中发生散射，并对其

radiance 产生贡献。这两项都以 $\Psi^G + \Psi^G\Psi^L$ 的形式进行添加，并通过像素链的 BSDF 来累积光源贡献。这种技术的开销更大，但它是一个对毛发体积中的多重散射现象的精确且实时的近似。它同样也可以与本章中所介绍的任何 BSDF 一起使用。

图 14.51：前两幅图像是使用路径追踪进行渲染的毛发，来作为三个毛发散射分量（ R ， TT ， TRT ）的参考，第二幅图像还添加了多重散射效果。最后两幅图像展示了使用双重散射近似的渲染结果：第三幅是路径跟踪的渲染结果，第四幅是在 GPU 上的实时渲染结果。[1953]

对于动画的半透明材质，环境光照是另一个难以计算的输入项。从球谐函数中来采样 irradiance 是很常见的操作。还可以在从毛发的静止位置上，计算非定向的、预积分的环境光遮蔽，来对光照进行加权修正[1560]。使用与多重散射相同的假法线方法，Karis 提出了一个用于处理环境光照的特殊模型[863]。

想要了解更多有关实时毛发渲染的信息，Yuksel 和 Tariq [1954]提供了一个全面的在线课程。在阅读相关研究论文和学习更多细节之前，这个在线课程将教给你有关毛发渲染的许多领域，例如模拟、碰撞、几何、BSDF、多重散射和体积阴影等内容。在如今的实时渲染应用中，头发看起来已经相当逼真了，但是仍然需要更多的研究，来更好地近似基于物理的环境光照和毛发中的多重散射。

14.7.3 皮毛

与毛发（hair）相反，皮毛（fur）在动物身上十分常见，它指的是那些较短的、半有组织（semi-organized）的发束。与用于体渲染的多层纹理相关的一个概念是体积纹理（volumetric texture），它是由多层二维半透明纹理进行表示的体积描述方法[1203]。

例如：Lengyel 等人[1031]使用一组包含 8 个纹理的集合来表示表面上的皮毛。每个纹理都代表了在距离表面一定距离处的毛发切片。这个模型会被渲染 8 次，使用一个顶点着色器程序，每次都会沿着顶点法线将每个三角形稍微向外移动一点。通过这种方式，每个连续的模型都描绘了表面上的一个不同高度。以这种方式创建的嵌套模型

被称为壳（shell）。这种渲染方法会在沿着物体轮廓边缘的地方失效崩溃，因为毛发会随着图层的展开而分解成点。为了隐藏这个瑕疵，还会通过在轮廓边缘处生成的鳍片（fin）上，应用不同的毛发纹理来表示皮毛，如图 14.52 和图 19.28 所示。这种沿着轮廓挤压出鳍片的想法，还可以为其他类型的模型创建一些视觉复杂性。例如：Kharlamov 等人[887]使用鳍片和浮雕映射，来生成具有复杂轮廓的树木网格。

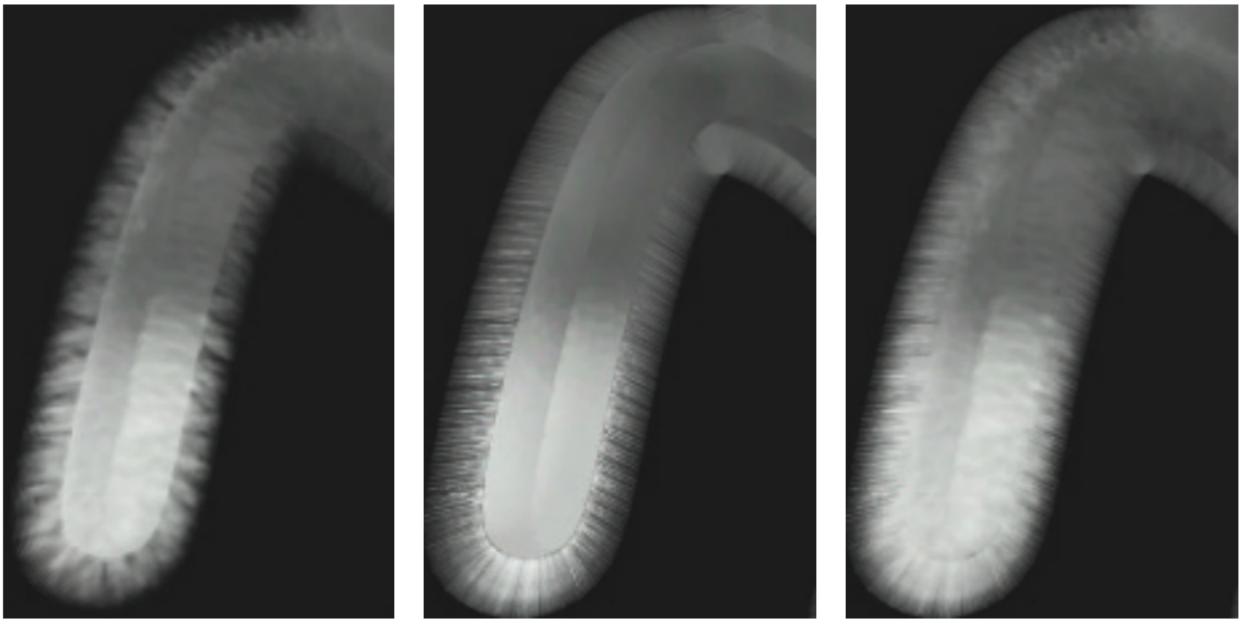


图 14.52：使用体积纹理渲染的毛皮。这个模型被渲染了 8 次，每个 pass 都会将表面都向外扩展一点。左边是八个 pass 的渲染结果。注意轮廓上断裂的毛发。中间展示了鳍片渲染的结果。右边是最终的渲染结果，同时使用了鳍片和壳。[1300]

几何着色器的引入，使得被皮毛覆盖表面可以挤出真正的折线毛发（polyline hair）。《失落的星球（Lost Planet）》[1428]便使用了这种技术。渲染一个表面，并将每个像素的值保存下来：皮毛颜色、长度和角度。然后使用几何着色器对这个图像进行处理，使得图像上的每个像素都变成一个半透明的折线。通过为每个像素都创建一个覆盖该像素的毛发，因此可以自动维护 LOD。皮毛通过两个 pass 进行渲染。在屏幕空间中指向下方的皮毛将首先被渲染，并按照从屏幕底部到屏幕顶部的顺序进行渲染。通过这种方式，可以将渲染的皮毛按照从后到前的正确顺序进行混合。在第二个 pass 中，剩余的皮毛此时都指向上方，并按照从上到下的顺序进行渲染，同样也可以被正确混合。随着 GPU 的不断发展，使得更多的新技术成为可能，并且效果很好。

也可以使用前面几小节中所介绍的技术。可以从皮肤表面挤出一些特定的几何形状，来作为一束毛发的渲染替代，例如游戏《星球大战：前线》中的 Chewbacca，或者是 TressFX 的 Rat demo [36]。在将毛发渲染为细丝的时候，Ling-Qi（闫令琪）等人[1052]已经证明了，仅仅是将毛发建模成均匀的圆柱体是不够的。对于动物皮毛而

言，其最内层的髓质要比毛发更大、更暗，它会减少光线散射的影响。因此，他们提出了一种双圆柱体的纤维 BSDF 模型，可以模拟更多类型的毛发和皮毛[1052]。这个 BSDF 模型考虑了更加详细的光线路径，例如 $TttT$ ， $TrRrT$ ， $TttRttT$ 等，其中小写字母 t 和 r ，代表了光线与髓质之间的相互作用。这种复杂的方法能够生成更加逼真的视觉效果，尤其是对于粗糙皮毛和精细散射效果的模拟效果非常好。这种皮毛渲染技术涉及对大量毛发束实例的光栅化，因此任何能够帮助减少渲染时间的技术都是十分受欢迎的。Ryu 提出[1523]，可以根据运动幅度和距离来减少毛发束实例的数量，从而作为一种 LOD 控制。这种方法被用于离线的电影渲染中，并且看起来很容易应用到实时应用程序中。

14.8 统一方法

随着硬件和软件的发展，我们现在已经能够在实时应用中实现体渲染了。那么未来还可能会实现什么新特性呢?

在本章的开头我们曾说过：“万物都在散射（everything is scattering）”。对于参与介质的材质，我们可以使用一个很高的散射系数 σ_s 来实现不透明介质。这与定义漫反射响应和镜面响应的复杂各向异性相位函数一起，将会产生一个不透明的表面材质。鉴于此，是否有一种表示方法可以统一实体材质和体积材质呢?

到目前为止，体积渲染和不透明材质的渲染是分开的，因为当前 GPU 的计算能力有限，这迫使我们只能针对不同情景使用特定的方法。我们使用网格来表示不透明表面，使用 alpha 混合的网格来表示透明材质，使用粒子广告牌来表示烟雾体积，并使用光线步进来模拟参与介质内部的体积光照效果。

正如 Dupuy 等人[397]所暗示的那样，使用统一的表示方法来同时表示固体和参与介质是可能的。一种可能的表示方法是使用对称 GGX（symmetrical GGX，SGGX）[710]，我们在章节 9.8.1 中介绍过 GGX 法线分布函数，SGGX 是对 GGX 的扩展。在这种情况下，在一个体积内表示定向片状颗粒（oriented flake particle）的微片理论（microflake theory），取代了用于表示表面法线分布的微表面理论（microfacet theory）。从某种意义上来说，与网格表示方法相比，这种微片表示中的 LOD 将会变得更加实用，因为这里的 LOD 就是对材质属性的体积过滤。这将会导致更加连贯的光照效果，以及更加连贯的大型世界，同时还能够保持应用在背景上的光照、形状、遮挡、或者透光率等。例如：如图 14.53 所示，使用体积过滤的树木表示方法来渲染一片森林，我们将再也看不到树木网格的 LOD 切换过程；这种表示方法可以对较薄的几何物体提供平滑的过滤效果，能够避免由树枝所引起的锯齿，同时还可以考虑到每个体素内的底层树木几何，从而提供正确的遮挡值。

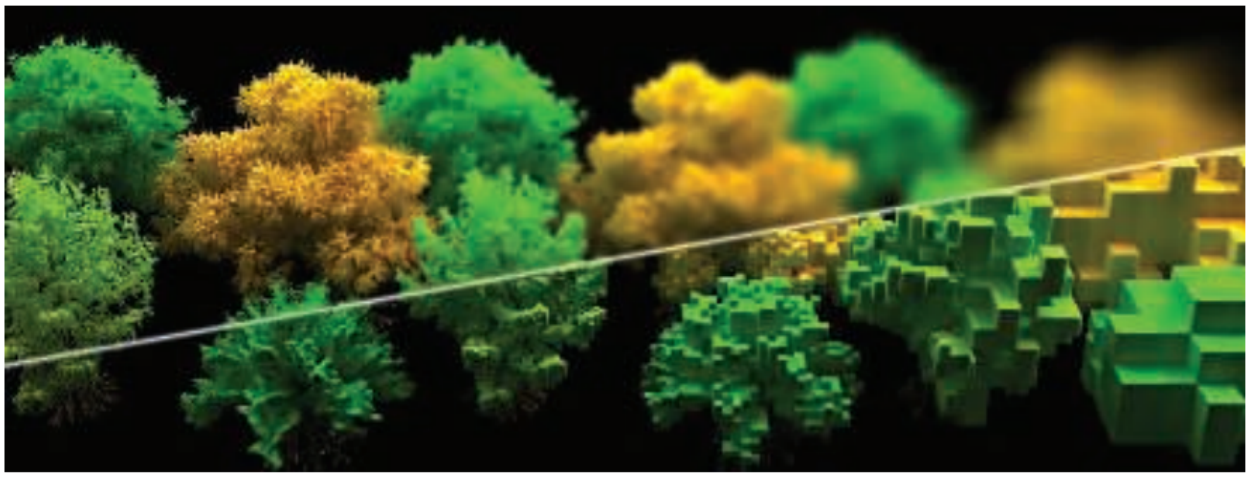


图 14.53：上面是使用 SGGX 进行渲染的森林，从左到右的 LOD 依次递减。下面显示的是未经过滤的原始体素。[710]

补充阅读和资源

进一步阅读的补充资源在整个章节中都有提及，但是在这里还是值得特别强调一下。在 Fong 等人[479]的课程讲义中解释了通用的体渲染技术，并提供了大量的背景理论、优化细节和电影制作中所使用的解决方案。对于天空和云的渲染，本章节内容建立在 Hillaire 的课程讲义[743]之上，其中还有非常多的细节内容，我们无法在这里面面俱到。有关体积材质的动画技术并不在本书的讨论范围之内，我们建议读者阅读一些有关实时模拟的论文[303, 464, 1689]，尤其是 Bridson 的书[197]。McGuire 的演讲[1182]，以及 McGuire 和 Mara 的论文[1185]，让人们对于透明度相关的效果，以及可以用于各种元素的策略和算法，有了更加广泛深刻的理解。对于毛发和皮毛的渲染和模拟，我们再次向读者推荐 Yuksel 和 Tariq 的课程讲义[1954]。

Chapter 15 Non-Photorealistic Rendering 非真实感渲染

> Stanislaw Ulam——“Using a term like ‘nonlinear science’ is like referring to the bulk of zoology as ‘the study of nonelephant animals.”

> 斯塔尼斯拉夫·乌拉姆——“使用‘非线性科学’这样的术语，就像是把大部分动物学称为“对非大象动物的研究”一样。”（美国籍数学家，曾参与曼哈顿计划；1909—1984）

真实感渲染（photorealistic rendering）的目标是让渲染图像与现实照片之间难以区分。而非真实感绘制（non-photorealistic rendering，NPR），又称风格化绘制（stylized rendering），则有着广泛的目标。某些形式的 NPR，其目标是生成类似于技术插图的图像，即只有那些与特定应用程序目标相关的细节才应当被显示和渲染出来。例如：在向客户销售汽车的时候，一张闪亮的法拉利引擎照片可能会有用；但是在维修引擎的时候，一张突出显示相关部件的简化线条图可能会更有意义（并且打印成本更低）。

图 15.1：应用于咖啡研磨机上的各种非真实感渲染风格。

NPR 的另一个领域是对绘画风格和自然媒介的模拟，例如钢笔、墨水、木炭和水彩等。这是一个十分广阔的领域，它为各种各样的算法提供了机会，这些算法都试图去捕捉各种绘画介质的感觉，图 15.1 展示了一些例子。有两本比较旧的书涵盖了绘画 NPR 的算法和技术[563, 1719]。鉴于 NPR 领域的广度，因此在本书中，我们将专注于渲染笔画和线条的技术。我们的目标是介绍一些用于 NPR 的实时算法。本章节首先会详细讨论实现卡通渲染风格（cartoon rendering style）的方法，然后再讨论 NPR 领域内的其他主题。在本章节的最后，还会介绍各种用于线条渲染的技术。

15.1 卡通着色

正如不同类型的字体会给文本带来不同的感觉一样，不同风格的渲染也有自己独特的情绪、含义和表现形式。人们对 NPR 中的一种特殊形式给予了大量关注，即卡通渲染（toon rendering），或者叫做 cel rendering（这里 cel 的全称是 Celluloid，即赛璐璐风格）。由于这一渲染风格与动画片（cartoon）有关，因此它具有幻想和童年的表现含义。在最简单的卡通渲染中，我们使用纯色来填充物体，并使用实线来绘制物体的轮廓，从而将不同的实色区域分隔开来。关于这种风格流行的一个原因，McCloud 在其经典著作《理解漫画（Understanding Comics）》中[1157]提到，“通过简化来进行增强（amplification through simplification）”。通过对画面内容进行简化并剔除杂乱的信息，可以增强与演出相关信息的效果。对于卡通角色而言，会有更多的观众认同那些绘画风格简单的角色。

卡通渲染风格在计算机图形学中已经使用几十年了，主要是将三维的模型与二维卡通动画结合在一起。与其他 NPR 风格相比，这种风格在定义上较为简单，因此便于使用计算机来进行自动生成。这种风格在许多电子游戏中都得到了很好地应用[250, 1224, 1761]，如图 15.2 所示。

图 15.2：一个来自于游戏《大神（Okami）》的实时 NPR 渲染示例。

在这种卡通渲染风格中，物体的轮廓通常会被渲染为黑色线条，从而增强卡通外观。有关如何寻找和渲染这些轮廓的内容将在下一小节中进行讨论。有几种不同的方法可以渲染这些卡通表面。其中有两种最为常见的方法，第一种是使用一个纯色（不接受光照）来填充网格区域；第二种则是使用一个双色（two-tone）方法，其中一种颜色代表明亮区域，另一种颜色代表阴影区域。这里所说的双色方法，有时候也会被称为硬着色（hard shading），这种着色方法在像素着色器中执行起来十分简单，当着色法线与光源方向之间的点积大于某个值的时候，就使用较浅的颜色；反之如果点积小于某个值的时候，就使用较暗的颜色。当光照环境比较复杂时，还可以使用另一种方法，即对最终图像本身进行量化。这个过程也被称为色调分离（posterization），它指的是一个将连续范围内的值转换为几个离散颜色的过程，每个颜色之间具有较大的差异，如图 15.3 所示。对 RGB 值进行量化可能会导致一些令人不快的色调偏移（hue shift），因为每个单独颜色通道中的变化方式，与其他颜色通道中的变化方式并没有密切的联系。一种更好的选择是使用 HSV、HSL 或者 YCbCr 等色调保持（hue-preserving）的色彩空间。或者还可以定义一个一维函数，或者使用一个一维纹理（ramp 纹理），来将不同强度等级的颜色重新映射到特定的色调或者颜色上。

还可以使用量化或者其他滤波器来对纹理进行预处理操作。图 15.16 还展示了一个具有更多颜色级别的例子。

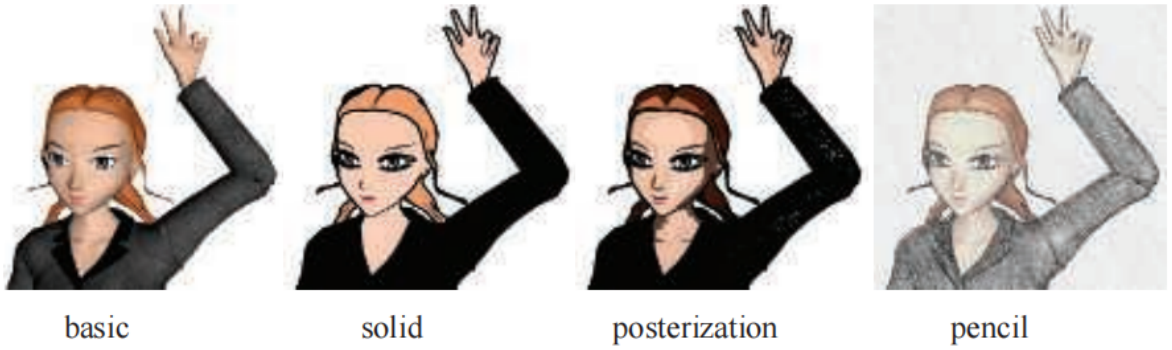


图 15.3：最左边的是基本渲染，之后三幅图片依次使用了纯色填充、色调分离和铅笔着色。[1449]

Barla 等人[104]通过使用一个二维贴图来代替之前的一维着色纹理，从而添加与视角相关的效果。这个二维贴图的第二个维度，是通过表面的深度或者方向进行索引的。这会使得物体在较远距离处，或者是快速移动的时候，能够在对比度上实现平滑的软化效果。在游戏《军团要塞 2》中，该算法与其他各种着色方程和绘制纹理相结合，从而实现了一种卡通风格和现实风格相融合的效果[1224]。卡通着色器还有很多很多其他的变体，用于实现不同的目的，例如在对表面特征或者地形特征进行可视化的时候，可以用于增强对比效果[1520]。

15.2 轮廓渲染

这些用于卡通边缘渲染的算法，是 NPR 中的一些主要课题和重要技术。本小节我们的目标是对一些轮廓渲染算法进行介绍，让大家对该领域有个大致的了解。我们所使用的方法可以大致分为基于表面着色（surface shading）、程序化几何（procedural geometry）、图像处理（image processing）、几何边缘检测（geometric edge detection），或者是这些方法的混合方法。

在卡通渲染中，可以使用以下几种不同类型的边缘：

- 不被两个三角形所共享的边界（boundary）或者边界边缘（border edge）。例如一张纸的边缘。实心物体通常没有边界边缘。
- 折痕边缘（crease edge）、硬边（hard edge）或者特征边缘（feature edge），是指被两个三角形所共享的边界，其中这两个三角形之间的夹角（叫做二面角，dihedral angle）大于某个预设的阈值。一个比较好的默认折痕夹角为

60 度[972]。例如：立方体中便含有折痕边缘。折痕边缘可以进一步细分为脊边（ridge edge）和谷边（valley edge）。

- 当共享某个边界的两个三角形具有不同的材质、或者产生着色变化的时候，就会出现材质边缘（material edge）。它也可以是艺术家希望进行展示的一些边缘，例如：人脸额头上的线条，或者一个分隔相同颜色裤子和衬衫的线条。
- contour 边缘（contour edge）指的是，与某些方向向量（通常是来自眼睛的观察向量）相比，两个相邻三角形的朝向有所不同。
- silhouette 边缘（silhouette edge）指的是，沿着物体的轮廓边界，这个边界可以将物体从图像的背景中分离出来。

> 译者注：contour 可以译为轮廓，是物体广义的轮廓边缘。silhouette 可以译为剪影，通常包含一种颜色，旨在表现物体的整体形状而不是其他的视觉细节。例如人面向太阳拍照，所形成的黑色剪影，就可以叫做 silhouette 。这里为了更加准确清晰，就不翻译 contour 和 silhouette 了。

上述边缘的分类如图 15.4 所示。这种分类方法和名称来源于相关文献和论文中的常用用法，但是也有一些不同的地方，例如：上面我们提到的折痕边缘（crease edge）和材质边缘（material edge），在其他地方有时也会被称为边界边缘（boundary edge）。

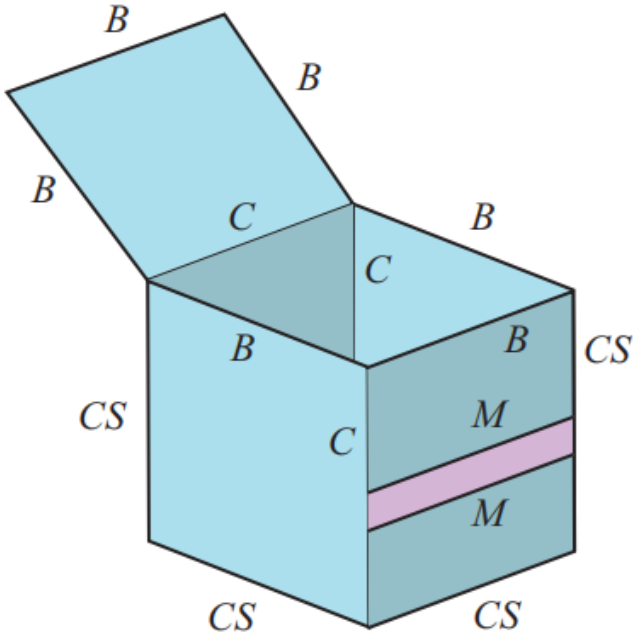


图 15.4：一个顶部打开的盒子，前面还有一个粉色条纹。图中展示了边界边缘（B）、折痕边缘（C）、材质边缘（M）以及 silhouette 边缘（S）。根据上述所给出的定义，图中的边界边缘（C）都不属于 silhouette 边缘，因为这些边缘只有一个相邻的多边形。

在这里我们要区分一下 contour 边缘和 silhouette 边缘。二者都有轮廓边缘的含义，其中表面的一部分会面向观察者，而另一部分则面向其他方向。silhouette 边缘是 contour 边缘的一个子集，silhouette 边缘可以将物体与另一个物体或者背景分离开来。例如：在头部的侧视图中，耳朵会形成 contour 边缘，即使它们会出现在头部 silhouette 轮廓的内部。图 15.3 中还展示了一些其他这样的例子，包括鼻子、两根弯曲的手指、以头发部分。在一些早期的文献中，contour 边缘也被称为 silhouette 边缘，但它通常指的就是完整的 contour 边缘。此外，contour 一词还具有等高线的含义，这里我们所提及的 contour，不能与地形图上所使用的等高线概念相混淆。

图 15.5：从左到右分别是：silhouette 边缘，contour 边缘，contour 边缘 + 暗示性 contour 边缘。

需要注意的是，边界边缘与 contour 边缘或者 silhouette 边缘是不同的。contour 边缘和 silhouette 边缘都与观察方向有关，而边界边缘则与观察方向无关。暗示性 contour 边缘（suggestive contour 边缘）[335]是从原始观察位置，去观察那些几乎是 contour 的位置所形成的。它们可以提供额外的边缘信息，来帮助传达物体的形状，如图 15.5 所示。虽然本小节的重点主要是对 contour 边缘的检测和渲染，但是也会涉及一些其他类型的笔画工作[281, 1014, 1521]。我们还会着重关注如何在多边形模型中寻找这样的边缘。Benard 等人[132]讨论了如何在细分曲面中、或者是在其他高阶定义所构成的模型中，寻找 contour 边缘的方法。

15.2.1 基于法线的 contour 边缘着色

与章节 15.1 中的表面着色器相类似，着色法线和观察方向之间的点积可以用来寻找 contour 边缘[562]。如果这个点积接近于零，即着色法线几乎垂直于观察方向，那么说明这个表面几乎是侧对着眼睛的，因此这个位置很可能会接近 contour 边缘。可以将这些区域涂成黑色，并随着点积的增大而逐渐变为白色，如图 15.6 所示。在可编程着色器出现之前，这个算法是使用一个带有黑色圆环的球形环境贴图来进行实现

的，或者是将 mipmap 金字塔纹理的最顶层涂成黑色，也可以实现这一效果[448]。而在如今，这种类型的着色是直接在像素着色器中实现的，当屏幕法线垂直于观察方向时，则将该片元涂成黑色。

图 15.6：当着色法线几乎垂直于观察方向时，可以使得该处的表面变暗，从而对 contour 边缘进行着色。通过增大衰减的角度，可以显示一个更厚的边缘。

这种着色方法在某种意义上来说与边缘光照刚好相反，在边缘光照中，光线将会照亮物体的轮廓；而在这种边缘变黑的着色方法中，就好像相机位置处有一个光源一样，整个场景都是被这个相机光源所照亮的，并且这个光照效果具有很强的衰减，从而使得边缘变得很暗。它也可以被认为是一种图像处理中的阈值滤波器（thresholding filter），当某个表面低于一定强度的时候，会将图像上的对应位置变为黑色，否则变为白色。

这种方法具有一个特点，或者也可以算是一个缺点，即 contour 边缘的宽度是可变的，具体的宽度取决于表面的曲率。由于在尖锐折痕的地方，表面曲率会发生剧烈变化，几乎没有像素表面的法线会垂直于观察方向；因此这种方法适用于没有折痕边缘的曲面模型，因为这种模型在沿着 silhouette 的区域内，其表面法线通常都会几乎垂直于观察方向，因此存在一些可以被涂成黑色的像素。这个算法无法应用在立方体等模型上，因为折痕边缘附近的表面区域不具这样的属性。但是即使在曲面模型上，这个算法有时候也会失效，看起来会很糟糕；因为当物体距离相机较远时，在 contour 边缘附近采样的表面法线，可能并不会与观察方向几乎垂直。Goodwin 等人[448]指出了这个基本概念元素作为视觉暗示的有效性，并讨论了如何结合光照、表面曲率和相机距离来确定笔画的厚度。

15.2.2 程序化的几何 Silhouette

Rossignac 和 van Emmerik [1510]提出了一个实时渲染 contour 边缘的技术，这也是该类技术中最早的之一，该算法后来被 Raskar 和 Cohen [1460]进行了改进。这种技术的通用思路是，先正常渲染一遍模型正面，然后再渲染一遍模型背面，从而使得模型的 contour 边缘变得可见。渲染模型背面的方法有很多种，每种方法都有自己的优缺点。但是这些方法都会先渲染一遍模型正面；然后打开正面剔除，关闭背面剔除，从而只渲染模型的背面。

渲染 contour 边缘的一种方法是，只渲染模型背面的边缘，而不是渲染模型背面的三角面。使用偏移（biasing）或者其他技术（章节 15.4），来保证其中的一些线段正好能够画在模型正面的正前方。这样的话，就只有模型正面与模型背面相交的边缘是可见的了[969, 1510]。

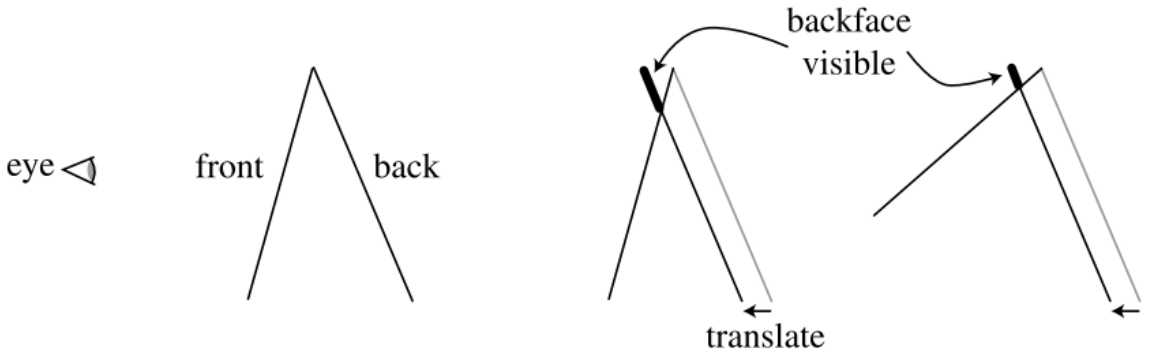


图 15.7： silhouette 的深度偏移方法，通过将背面向前平移来进行实现。如果正面位于一个不同的角度（如右侧所示），将会看到不同大小的背面区域。[1460]

想要让这些线条变宽，其中一种方法是将模型背面渲染为黑色，并向前进行偏移。Raskar 和 Cohen 给出了几种偏移方法，例如按照固定的量进行平移，或者是对具有非线性性质的 z-depth 进行补偿，又或者是使用一个基于深度-斜率的偏移（depth-slope bias），例如 OpenGL 中的 glPolygonOffset 。Lengyel [1022]讨论了如何通过对透视矩阵进行修改，来提供更加精细的深度控制。所有这些方法都存在一个问题，即它们无法创建的均匀宽度的描边线条。为了对这一点进行改进，向前偏移的量不仅要取决于背面，还要取决于相邻的正面，如图 15.7 所示。可以使用背面三角形的斜率来调整多边形向前的偏移量，同时线条的粗细也将取决于正面三角形的斜率。

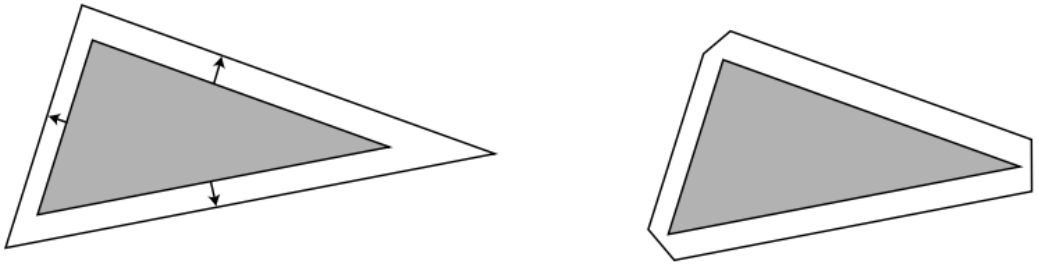


图 15.8：三角形扩展技术（Triangle fattening）。左侧：一个背面三角形按照其平面进行扩展，其中三角形的每个边缘在世界空间中的偏移量都不同，从而使得最终生成的边缘在屏幕空间中具有相同的厚度。然而对于细长的三角形而言，这个技巧将会失效，因为其中一个角会被过分拉长。右侧：将三角形的边缘进行扩展，并将其连接形成斜切角，从而避免这个问题。

Raskar 和 Cohen [1460, 1461]通过将每个背面三角形沿其边缘进行加粗，从而解决这个相邻依赖的问题，能够实现粗细一致的线条。也就是说，三角形的斜率与到观察者的距离共同决定了三角形向外扩展的程度。其中一种扩展方法是，将三角形的三个顶点沿其平面向外进行扩展。一种更加安全保守的渲染方法是，将三角形的三条边向外进行扩展，并将这些边重新连接起来。这样做可以避免调整后的顶点远离原始三角形，如图 15.8 所示。需要注意的是，使用这种方法的话，就不需要对模型背面进行偏移了，因为此时背面自动就会扩展到正面边缘的外部。图 15.9 中展示了这三种方法的执行结果。这种描边技术更加可控且更加均匀，它被许多电子游戏所采用，用于绘制游戏中的角色轮廓，例如《波斯王子（Prince of Persia）》[1138]和《爆炸头武士（Afro Samurai）》[250]。

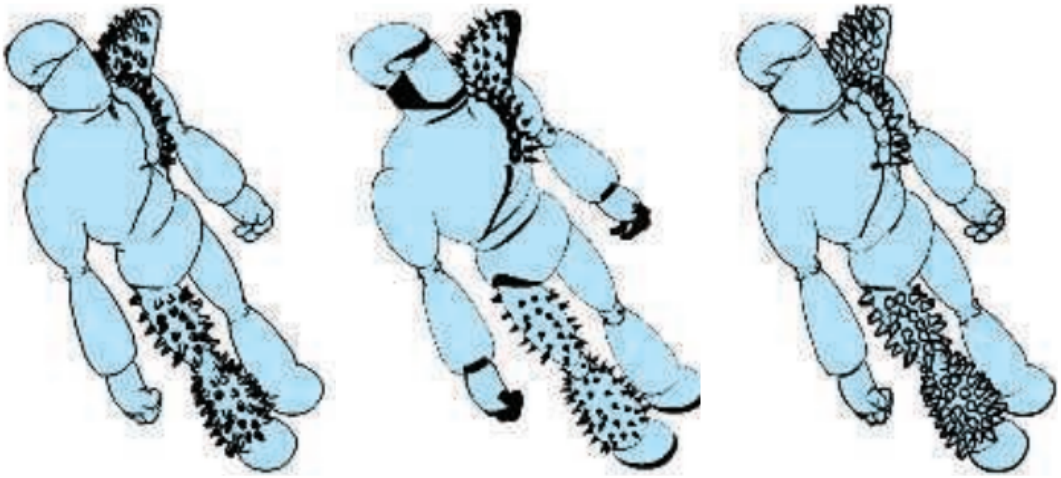


图 15.9：使用边缘加粗线条、 z 偏移和扩展三角形算法进行渲染的背面 contour 边缘。由于细小特征的偏移问题，因此背面边缘技术往往会导致线条之间的连接性不佳，并且线条的粗细也不均匀。由于 z 偏移算法依赖于正面的倾斜角度，因此它所产生的边缘宽度并不均匀。[1460]

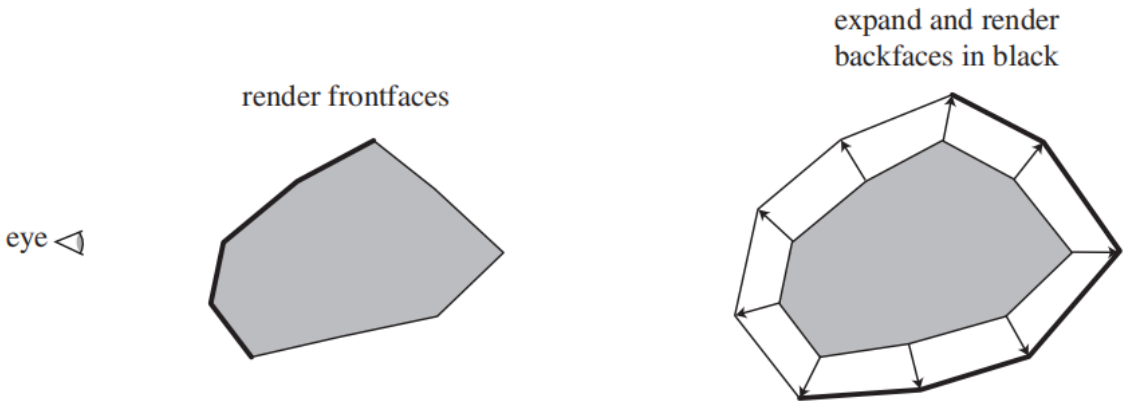


图 15.10：三角壳技术通过沿着顶点法线对表面进行偏移，从而创建第二个表面。

在上述给出的方法中，背面三角形沿着其原始平面进行展开。另一种方法则是沿着共享的顶点法线，来将三角形的顶点向外移动，具体移动的距离与顶点到眼睛的 z 距离成正比[671]。这种方法被称为外壳方法（shell）或者光晕方法（halo），这是因为位移后的背面会在原始物体周围形成一个外壳。想象现在有一个球体，我们先正常渲染一遍球体，然后以相对于球体中心 5 像素宽的半径来对球体进行扩展。如果说球体中心移动一个像素，相当于它在世界空间移动 3 毫米的话，那么此时扩展球体的半径将增大 15 毫米。我们将这个扩展版本的背面渲染为黑色，此时 contour 边缘的宽度将会是 5 个像素，如图 15.10 所示。这种沿着顶点法线将顶点向外移动的任务，非常适合顶点着色器来进行执行。这种类型的扩展方法有时也会称为壳映射（shell mapping）。这种方法实现简单，效率较好，健壮性好，性能稳定，如图 15.11 所示。可以对背面进一步扩展，并根据角度来对背面进行着色，从而实现一种力场效果或者光晕效果。

图 15.11：这是来自游戏《Cel Damage》中的实时卡通风格渲染，它使用了背面外壳扩展来渲染 contour 边缘，以及显式的折痕边缘渲染。

这种壳技术有几个潜在的缺陷。例如：想象一下我们直视一个立方体，此时只有正面是可见的。形成 contour 边缘的四个背面，每个都会向着其相应立方体表面的方向进行移动，这样会导致在立方体的角落处产生空隙。这种现象产生的原因是，虽然每个角上都有一个顶点，但是每个立方体表面都有着不同的顶点法线。扩展后的立方体并没有真正形成一个外壳，因为每个立方体的顶点都在向着不同的方向进行扩展。一种解决方案是，让处于同一位置的若干顶点共享使用一个新生成的平均顶点法线。另一种解决方案是，在折痕处创建退化（degenerate）的几何图形，然后再根据面积将其扩展成三角形。Lira 等人[1053]使用了一个额外的阈值纹理。来控制每个顶点的移动量。

壳技术和扩展技术会浪费其中一部分的填充图形，因为所有的背面三角形都会被送进管线中。所有这些技术都还有一些其他的局限性，例如无法精确控制边缘的外观，以及很难正确渲染透明表面等，当然这取决于具体使用的透明算法。

这类几何技术有一个值得称道的特点，即在渲染过程中不需要任何连通信息或者边缘列表。其中的每个三角形都是独立于其他三角形来进行处理的，因此这种技术非常适合 GPU 进行实现[1461]。

这类算法只能渲染 contour 边缘。Raskar [1461]给出了一个巧妙的解决方案，这个方法不需要创建和访问边缘的连通数据结构，就可以在变形模型上渲染脊状的折痕边缘（ridge crease edge），其核心思路是沿着要进行渲染的三角形边缘生成一个额外的多边形。会由开发人员定义一个临界二面角（critical dihedral angle），它决定了这个折痕是否应当可见，这些边缘多边形，会根据这个临界二面角来从三角形平面进行弯曲。如果两个相邻三角形的折痕角大于这个临界折痕角，那么边缘多边形将会变得可见，否则它们将会被三角形所遮挡隐藏，如图 15.12 所示。谷边（valley edge）也可以使用类似的技术来进行渲染，但是需要一个额外的模板缓冲和多个 pass 来进行处理。

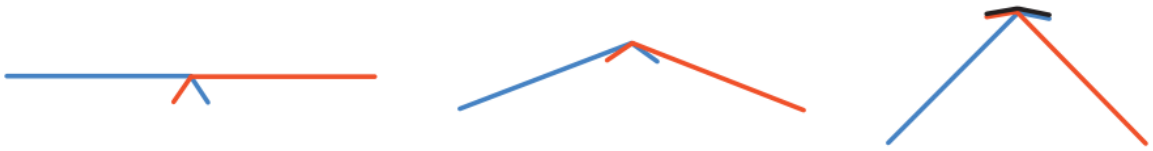


图 15.12：两个相连三角形的侧视图，每个三角形边都有一个附着的小“鳍”。当两个三角形沿着这个边缘发生弯曲的时候，附着的小鳍会在某个时刻变得可见。在右侧，两个小鳍露出来了，并被涂成了黑色，这看起来像是山脊的边缘。

15.2.3 基于图像处理的边缘检测

上一小节中的算法，有时也会被归类为基于图像的算法，因为它们会受到屏幕分辨率的影响。另一种类型的算法更加直接，它直接基于图像信息来进行处理，这类算法只会对存储在图像缓冲区中的数据进行操作，不会对场景中的几何形状进行修改，甚至不需要知道场景中的几何信息。

Saito 和 Takahashi [1528]首先引入了 G-buffer 的概念，它也被用在了延迟渲染中（章节 20.1）。Decaudin [336]对 G -buffer 的使用进行了扩展，从而来实现卡通渲染，其基本思想很简单：可以通过在各种缓冲区上执行图像处理算法来实现 NPR。通过查找相邻 z-buffer 中的不连续点，可以找到许多 contour 边缘的位置。相邻表面的不连续法线通常也会标识 contour 边缘和边界边缘的位置。使用环境颜色来渲染场景，或者是使用对象标记值，可以用于检测材质边缘、边界边缘和真正的 silhouette 边缘。

对这些边缘进行检测和渲染包括两部分。首先，渲染场景中的几何图形，使用像素着色器来保存深度、法线、对象 ID 或者其他任何渲染目标（render target）所需要的数据。然后以类似于章节 12.1 中所描述的方式来执行后处理。后处理 pass 会对每个像素周围的邻域进行采样，并基于这些采样输出一个结果。例如：假设我们对场景中

的每个物体都有一个唯一的标识值。这样的话，我们可以在每个像素上对这个对象 ID 进行采样，并将其与周围四个相邻像素的 ID 值进行比较。如果其中的任何一个 ID 值都与测试像素的 ID 值不相同，则输出黑色；反之输出白色。对周围 8 个相邻像素进行采样会更加万无一失，但是相应的，采样代价也会更高。这种简单的测试可以用来绘制大多数物体的边界边缘和轮廓（即真正的 silhouette 边沿）。材质 ID 可以用来查找材质边缘。

contour 边缘可以通过在法线缓冲和深度缓冲上使用各种滤波器来找到。例如：如果相邻像素之间的深度差超过某个阈值，则可能会存在 contour 边缘，因此这些像素会被设置为黑色。在实践中，我们需要一些更加精细的边缘检测算子，而不是简单地判断相邻像素是否和我们的样本相匹配。在这里我们不会讨论各种边缘检测滤波器的优缺点，例如 Roberts cross、Sobel 和 Scharr 等滤波器，因为有很多图像处理文献都对这些滤波器进行了广泛的介绍[559, 1729]。由于这些算子的结果并不一定是布尔值，因此我们可以对它们的阈值进行调整，或者是在某些区域内进行黑白渐变。需要注意的是，法线缓冲也可以用于检测折痕边缘，因为相邻像素之间较大的法线差异，也暗示了 contour 边缘或者折痕边缘的存在。Thibault 和 Cavanaugh [1761]讨论了他们是如何将这种技术应用在游戏《无主之地》的深度缓冲中的。在其他的一些技术中，他们对 Sobel 滤波器进行了修改，从而使其能够创建单像素宽度的轮廓，并使用深度计算来提高检测精度，如图 15.13 所示。或者也可以反过来，只在阴影周围添加轮廓，忽略那些具有较大深度差异的相邻边缘[1138]。

图 15.13：对 Sobel 边缘检测进行了修改，并应用在了游戏《无主之地》中。最终发布的版本（并没有显示在这里）通过掩盖前景中草的边缘，来进一步改善外观表现。[1761]

膨胀算子（dilation operator）是一种形态学算子（morphological operator），用于对检测到的边缘进行加粗[226, 1217]。在生成边缘图像之后，会应用一个单独的 pass 来进行处理。在每个像素位置上，会对该像素及其周围一定半径内像素进行检查。找到该范围内最暗的像素值，并将其作为输出值返回。这样操作下来，原本一条很细的黑线就会按照搜索区域的直径进行加粗。还可以使用多个 pass 来进一步加粗线条，这里存在一个 trade-off，即执行额外的 pass 会带来额外的成本，但是这样做每个 pass 所需要的样本数量就会大大减少。不同的结果可能会具有不同的粗细程度，例如：silhouette 边缘可能要比其他的 contour 边缘更粗。与此相关的一些腐蚀算子（erosion operator）可以用于对线条进行细化，或者是实现其他效果。图 15.14 展示了一些结果。

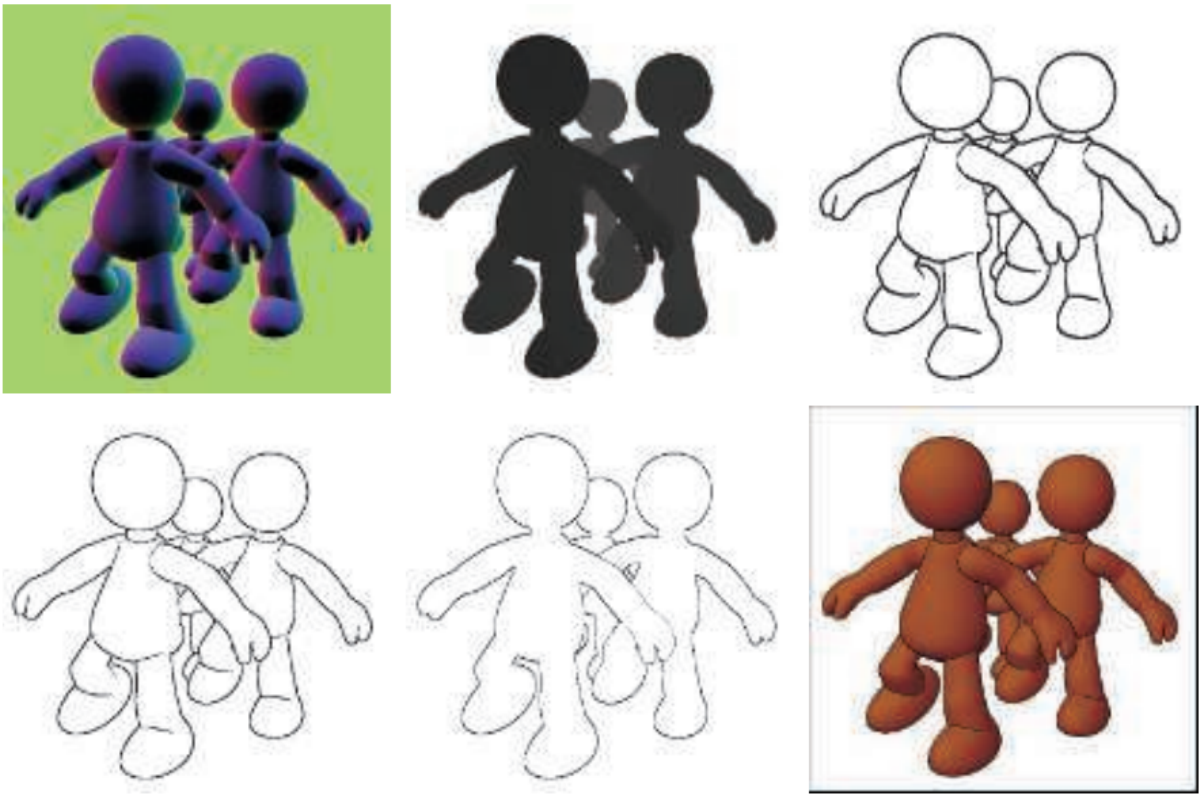


图 15.14：对法线贴图（左上）和深度贴图（中上）中的值进行了 Sobel 边缘检测，生成的结果分别展示在左下和中下。右上的图片是经过膨胀处理的加粗结果。右下角的最终渲染结果使用了 Gooch 着色，并将生成的边缘图片叠加在其顶部。

这种类型的算法有以下几个优点。第一，与其他大多数技术不同，这种类型的算法可以处理所有类型的表面，无论是平面还是曲面。第二，由于这个方法是基于图像的，因此不要求网格相互连通，也不要求网格的一致性。

这种技术的缺陷相对较少。对于靠近边缘的表面，比较 z-depth 的滤波器可能会错误地检测到表面上的 contour 边缘像素。比较 z-depth 的另一个问题在于，如果像素之间的深度差异很小的话，那么 contour 边缘可能会被遗漏。例如：桌子上的一张纸，使用这种方法可能会遗漏掉它的边缘，因为这个纸张和桌面的深度差异很小。类似地，基于法线贴图的滤波器也将会遗漏这张纸的边缘，因为纸张的法线和桌面的法线几乎是相同的。再比如，将一张纸折叠起来，可能会无法检测到重叠处的边缘 [725]。同时也会产生阶梯状的锯齿线条，章节 5.4.2 中所介绍的各种形态学抗锯齿技术，可以很好地处理这种高对比度的输出结果，同时还可以使用色调分离（posterization）等技术来提高边缘的视觉质量。

这种边缘检测也可能会以相反的方式失效，即在不应该存在边缘的地方产生了边缘。确定具体哪些位置属于边缘并不是一个简单的操作，例如：想象现在有一个玫瑰的茎秆，即一个细长的圆柱体。在靠近观察的时候，样本像素附近的茎秆法线变化不大，

因此不会检测到边缘。当我们逐渐远离这个玫瑰的时候，像素之间的法线会发生快速变化，直到在某些点上，由于这些法线差异的存在，可能会在边缘附近出现错误的边缘检测。在深度图中检测边缘有时也会出现类似的问题，透视效应对于深度的影响是另一个需要进行补偿的因素。Decaudin [336]给出了一种改进的方法，该方法通过处理法线贴图和深度贴图中的梯度来寻找变化，而不仅仅是数值本身。而各种像素之间的差异具体要如何转化为颜色上的变化，这是一个需要根据画面具体内容来进行调整的过程[250, 1761]，如图 15.15 所示。

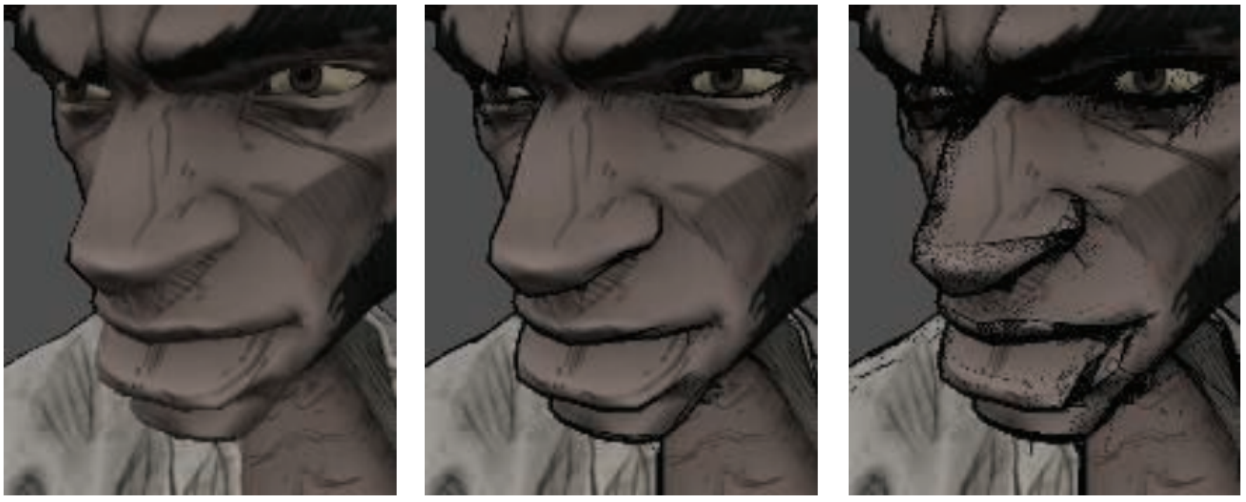


图 15.15：各种边缘方法。图中的褶皱等特征边缘属于纹理的一部分，它们是由艺术家事先添加的。人物角色的 silhouette 边缘则是通过背面扩展而产生的。contour 边缘是利用图像处理中的边缘检测生成的，它可以设置不同的权重。左图中 contour 边缘的生成权重很小，因此这些边缘显得很模糊。中间则展示了一些轮廓，尤其是鼻子处和嘴唇处的 contour 边缘。右图中所使用的权重过大，因此产生了一些瑕疵[250]。

一旦我们生成了一些笔画（stroke），就可以根据需要对其进行进一步的图像处理。由于这些笔画可以在单独的缓冲区中进行创建，因此它们也可以进行独立地修改，然后再叠加到表面上。例如：噪声函数可以用于对线条和表面进行磨损和摆动，或者是在二者之间创建细小的间隙，从而提供一种手绘外观。纸张的高度场同样可以用来影响渲染效果，例如：木炭等固体材料会沉积在纸张凸起的顶部；而水彩画则会聚集在纸张的凹陷处，图 15.16 展示了这样的一个例子。

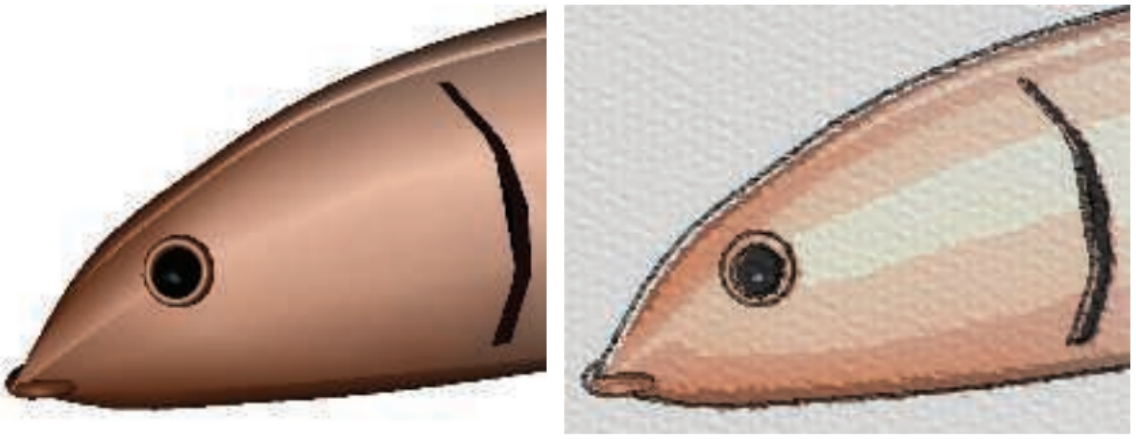


图 15.16：左边展示了这个鱼模型的原始渲染图，通过边缘检测、色调分离、噪声扰动、模糊处理和混合操作，将生成的结果叠加在左图上，最终产生了右图中的效果。

我们在这里专注于使用几何信息或者其他非图形信息（例如法线、深度和对象 ID 等）来检测边缘。当然，图像处理技术是针对图像进行开发的，这种边缘检测技术当然可以应用在颜色缓冲中。其中一种方法被称为高斯差分（difference of Gaussians，DoG），即使用两个不同的高斯滤波器来对图像进行两次处理，并从其中一个结果中减去另一个结果。这种边缘检测方法应用在 NPR 中可以产生特别令人愉悦的结果，它被用于生成各种艺术风格的图像，例如铅笔画和蜡笔画[949, 1896, 1966]。

NPR 技术中的一些图像后处理算子，着重于对艺术介质的模拟，例如水彩（watercolor）和丙烯颜料（acrylic paint）等。这一领域中已经有了相当多的研究，对于交互式应用程序而言，最大的挑战是如何使用最少的纹理样本来做最多的事情。可以在 GPU 上使用双边滤波器、均值偏移滤波器（mean-shift）和 Kuwahara 滤波器，从而在保留边缘信息的同时，对绘画区域进行平滑处理[58, 948]。Kyprianidis 等人[949]对该领域中的图像处理效果进行了全面的回顾和分类。Montesdeoca 等人[1237]的作品是一个非常好的案例，他将许多简单直接的技术结合在一起，从而实现了交互式的水彩画效果。图 15.17 展示了一个使用水彩风格进行渲染的模型。

图 15.17：左边是一个标准的真实感渲染结果。在右边，通过使用均值偏移颜色匹配来软化水彩纹理，并增加了对比度和饱和度，以及其他的一些技术，最终实现了水彩风格的渲染效果。

15.2.4 几何 contour 边缘检测

到目前为止，我们所介绍的方法都存在一个问题，即边缘的风格化特征是受限的，我们无法实现丰富的边缘外观。例如：想让线条看起来是虚线是很困难的，更不用说让线条看起来像手绘的或者笔触的了。对于这种类型的操作，我们需要找到 contour 边缘，并直接对它们进行渲染。如果我们拥有了独立的边缘实体，我们就可以实现各种额外的效果，例如：我们可以让网格保持冻结的同时，让 contour 轮廓发生跳动。

contour 边缘是指，两个相邻三角形中的其中一个面向观察者，而另一个面则指向其他方向。可以使用如下数学公式进行判断：

$$(\mathbf{n}_0 \cdot \mathbf{v})(\mathbf{n}_1 \cdot \mathbf{v}) < 0 \tag{15.1}$$

其中 $\mathbf{n}_0$ 和 $\mathbf{n}_1$ 是两个三角形的表面法线， $\mathbf{v}$ 是从眼睛到边缘（即到任意端点）的观察方向。为了使得这个测试能够正确生效，三角形的表面必须保持一致的朝向（章节 16.3）。

在模型中查找 contour 边缘的暴力方法，可以直接遍历边缘的列表并依次执行这个测试[1130]。Lander [972]提出了一个有价值的优化方法，即识别出位于平面多边形内部的边缘，并将其忽略。也就是说，给定一个连通的三角形网格，如果某条边上的两

个相邻三角形位于同一平面上，那么这条边不可能是 contour 边缘。在一个简单的时钟模型上实现这个优化过的测试方法，可以将检测的边数从 444 条减少到 256 条。此外，如果模型是一个实体物体，那么模型上的凹边永远不可能是 contour 边缘。Buchanan 和 Sousa [207]通过对每个单独的面重复使用点积测试结果，从而避免了对每个边缘重新进行单独的点积测试。

如果每一帧都对 contour 边缘进行重复检测，这样做的计算开销是很高的。如果相机视图和场景物体在每帧之间的变化程度很小，那么可以合理地假设前一帧中的 contour 边缘在当前帧中可能仍然是有效的。Aila 和 Miettinen [13]将每条边缘都与一个有效距离相关联。这个有效距离是观察者可以进行移动的距离，在这个距离内，仍然能够维持 contour 边缘的状态。在任何实体模型中，每个单独的 contour 边缘总是由单个封闭曲线构成的，这称为 silhouette 环（silhouette loop）；或者更恰当地说，是一个 contour 环（contour loop）。对于物体边界内的 contour 边缘而言，这个环的其中一部分可能会被遮挡。即使是真正的 silhouette 也可能是由几个环所构成的，其中一部分环位于轮廓内部，或者是被其他表面所遮挡。由此可知，每个顶点上必须有偶数条 contour 边缘[23]，如图 15.18 所示。需要注意的是，在沿着网格边缘的时候，这个环在三维空间中通常是锯齿状的，因为这个环的 z-depth 会发生明显的变化。如果我们希望边缘能够形成更加光滑的曲线，例如随着距离的变化发生相应的粗细改变[565]，那么我们还需要进行一些额外的处理，即在三角形的法线之间进行插值，从而对三角形内部的真实 contour 边缘进行近似[725, 726]。

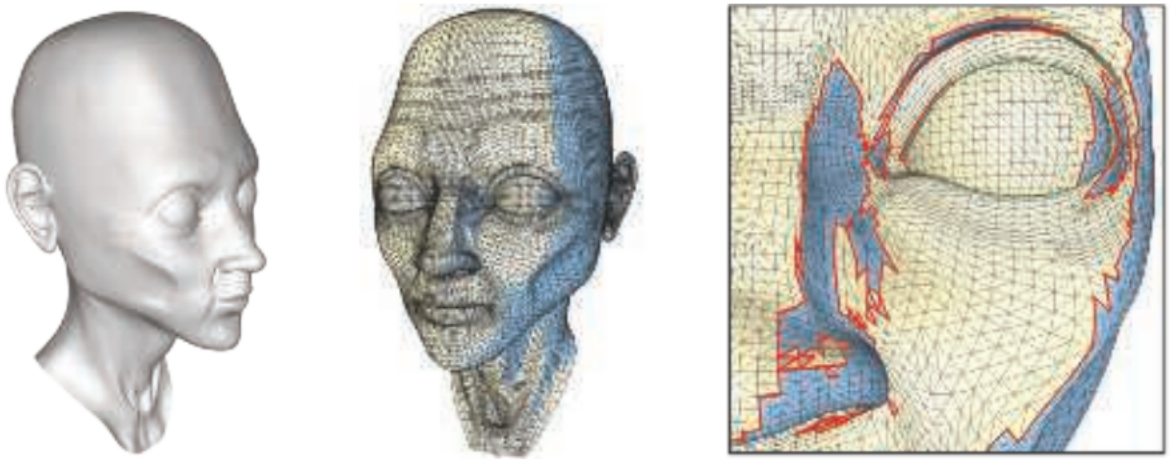


图 15.18：contour 环。左边是模型的相机视图。在中间，使用蓝色来代表那些背对着相机的三角形。右图是一个面部区域的特写。请注意图中的复杂性，以及一些 contour 环是如何隐藏在鼻子后面的。[132]

对这些环的位置进行逐帧追踪，要比从头开始创建循环快得多。Markosian 等人[1125]从一组环开始，并随着相机的移动，使用随机搜索算法来对这个环集合进行更

新。当模型方向发生改变的时候，contour 环也会被销毁并重新创建。Kalnins 等人[848]指出，当两个环发生合并的时候需要采取一些纠正措施，否则会在下一帧中出现明显的突变。他们使用了一个像素搜索和“投票”算法，来保持帧与帧之间的 contour 一致性。

这些技术可以显著提高性能表现，但是可能会不那么精确。一些线性方法是精确的，但是开销太大。使用相机来访问 contour 边缘的层次化方法同时兼顾了速度和精度。对于非动画模型的正交视图而言，Gooch 等人[562]使用一个层次化的高斯图来确定 contour 边缘。Sander 等人[1539]使用了 n 元法线锥（normal cone，章节 19.3）的树结构。Hertzmann 和 Zorin [726]使用了一种模型的双重空间（dual-space）表示，这允许他们在模型的边缘处添加层次结构。

所有这些显式的边缘检测方法都是 CPU 密集型的，并且具有较差的缓存一致性，因为构成 contour 的边缘会分散在整个边缘列表中。为了避免这些开销，可以使用顶点着色器来检测和渲染这些 contour 边缘[226]。这个想法是将模型中的每条边作为两个三角形（构成一个退化四边形）来发送到管线中，通过相邻三角形的顶点来构建三角形的法线。当发现某条边是 contour 边缘的一部分时，就会移动四边形的顶点，使其不再退化（即使其可见）。然后再渲染出这个很薄的四边形鳍片（fin）。我们在章节 7.3 中介绍了如何为阴影体的创建来寻找 contour 边缘，二者的思想实际上是相同的。如果管线中可以使用几何着色器的话，那么这些额外的鳍片四边形就不需要进行存储了，它们可以在运行过程中动态生成[282, 299]。一个比较简单的实现方法可能会在鳍片之间留下裂缝（chink）和缝隙（gap），这个瑕疵可以通过修改鳍片的形状来进行纠正[723, 1169, 1492]。

15.2.5 隐藏线移除

一旦我们找到了这些 contour 边缘，就可以对线条进行渲染。明确地找到这些边缘有一个好处，那就是我们可以将其作为钢笔笔触（stroke）、颜料笔触或者其他任何我们想要的媒介，来进行风格化处理。这些笔触可以是基本线条、纹理 impostor（章节 13.6.4）、基本图元、或者是任何我们想要尝试的东西。

使用几何边缘会带来另一个复杂的问题，即并不是所有的边缘都是可见的。通过在表面渲染的过程中生成 z-buffer，我们可以隐藏那些被遮挡的几何边缘，这对于虚线等一些简单的样式来说可能已经足够了。Cole 和 Finkelstein [282]对表示直线的四边形进行了扩展，他们沿着直线本身的中线（spine）来采样 z 深度。然而，即使是使用了这些方法，沿着线条的每个点都是独立进行渲染的，因此我们事前无法知道这个线条的开始位置和结束位置。对于 contour 环或者其他边缘而言，线段意味着笔触或

者其他的连续物体，我们需要知道每个笔触首次出现的时间和消失的时间。这种确定每个线段的可见性，称为隐藏线渲染（hidden line rendering）。在隐藏线渲染中，我们首先会对一组线段进行处理，然后返回一组较短的线段（可能会被裁剪）。

Northrup 和 Markosian [1287]通过渲染所有物体的三角形和 contour 边缘，并为每个三角形和 contour 边缘都分配不同的 ID 号码来解决这个问题。这个 ID 缓冲区会被重新读取，并从中确定可见的 contour 边缘。然后检查这些可见区域是否存在重叠部分，并将它们连接在一起，从而形成平滑的笔触路径。如果屏幕上的线段很短，那么这种方法是有效的，但是它并不会包含线段本身的裁剪部分。然后我们会沿着这些重建出来的路径来渲染风格化的笔触。这些笔触本身可以以许多不同的方式来进行风格化，包括逐渐变窄（taper）、光晕（flare）、波浪线（wiggle）、过冲（overshoot）和淡出（fading）等效果，以及一些深度暗示和距离暗示等。图 15.19 展示了这样一个例子。

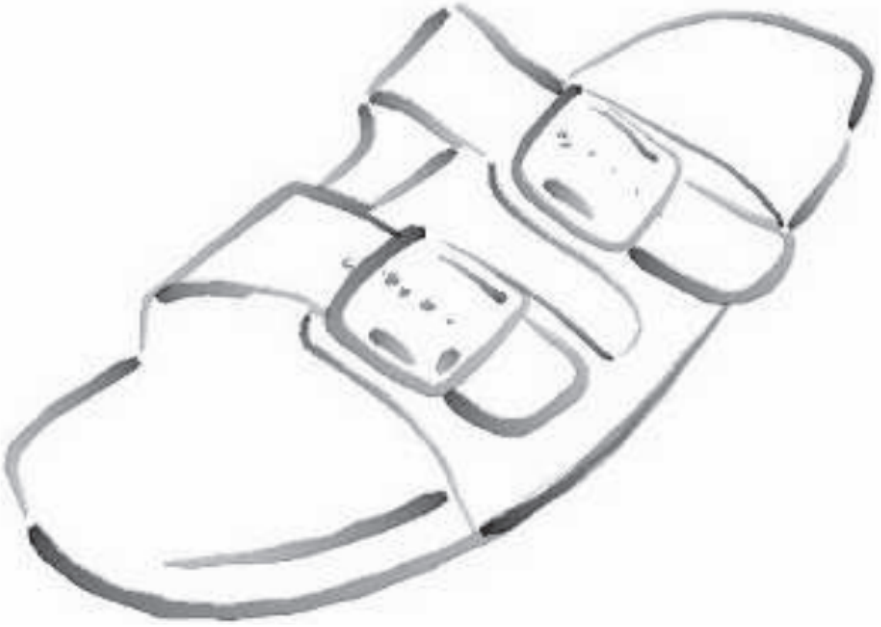


图 15.19：使用 Northrup 和 Markosian 的混合技术来生成的图像。找到 contour 边缘，将其构建成链，并渲染为笔触。

Cole 和 Finkelstein [282]提出了一种计算边缘集合可见性的方法。他们将每个线段存储为两个世界空间中的坐标值。然后会使用一系列的 pass，在整个片段集合上运行一个像素着色器，裁剪并确定每个片段的像素长度，为每个潜在的像素位置都创建一个图集并确定其可见性，最后再使用这个图集来创建可见的笔触。虽然这个算法比较复杂，但是在 GPU 上的处理速度相对较快，并且能够提供一系列可见的笔触，并且这些笔触的开始位置和结束位置都是已知的。

这种风格化处理，通常包括将一个或者多个预先制作的纹理应用于线条四边形上。Rougier [1516]讨论了一种不同的方法，可以程序化地渲染虚线图案。每个线段在渲染的时候都会访问一个纹理，这个纹理存储了所需的所有虚线图案。每个图案都会被编码为一组命令，它指定了虚线的图案以及所使用的结束（endcap）类型和连接（join）类型。通过使用四边形的纹理坐标，每个图案都会在着色器中进行一系列测试，从而确定线条在四边形中的每个点具体覆盖了多少像素。

确定 contour 边缘，将它们连接成连续的链（chain），然后确定每个链的可见性并形成笔触，这个过程是很难完全并行化处理的。在生成高质量风格化线条的时候，还存在另一个问题，即下一帧的每个笔触都将被重新绘制，这些笔触可能会发生长度的改变，或者是首次出现。Benard 等人[130]对各种渲染方法进行了调研，这些方法为沿边缘的笔触以及表面上的图案提供了时域一致性。有关边缘和笔触的问题，目前还没有被完全解决，并且其中可能会涉及到一些计算，因此对其的研究仍在继续[131]。

15.3 笔触表面风格化

虽然卡通渲染是一种十分流行的画面风格，但是还有很多很多种（理论上无限多种）其他的风格可以应用在表面上。这些效果的范围十分广泛，从对真实感纹理进行修改[905, 969, 973]，再到让算法程序化地生成帧与帧之间的几何装饰（geometric ornamentation）[853, 1126]。在本小节中，我们将简要介绍与实时渲染相关的技术。

Lake 等人[966]讨论了使用漫反射着色项来选择在表面上使用的纹理。当漫反射项变暗时，则使用具有深色效果的纹理。这个纹理可以使用屏幕空间坐标进行应用，从而提供一种手绘风格的效果。为了进一步增强这种素描（sketched）外观，还可以将一个纸张纹理应用于屏幕空间的所有表面上，如图 15.20 所示。这类算法存在一个主要问题，即淋浴门效应（shower door effect），这种效应指的是在动画过程中，场景中的物体看起来就好像是通过压花玻璃（patterned glass）进行观察的一样，会给人一种物体在纹理中游泳一样的感觉。Breslav 等人[196]通过确定哪些图像变换与底层模型的位置运动最为匹配，从而来维护一个纹理的二维查找。这样做可以保持与基于屏幕填充模式的联系，同时与场景中的物体建立更强的联系。

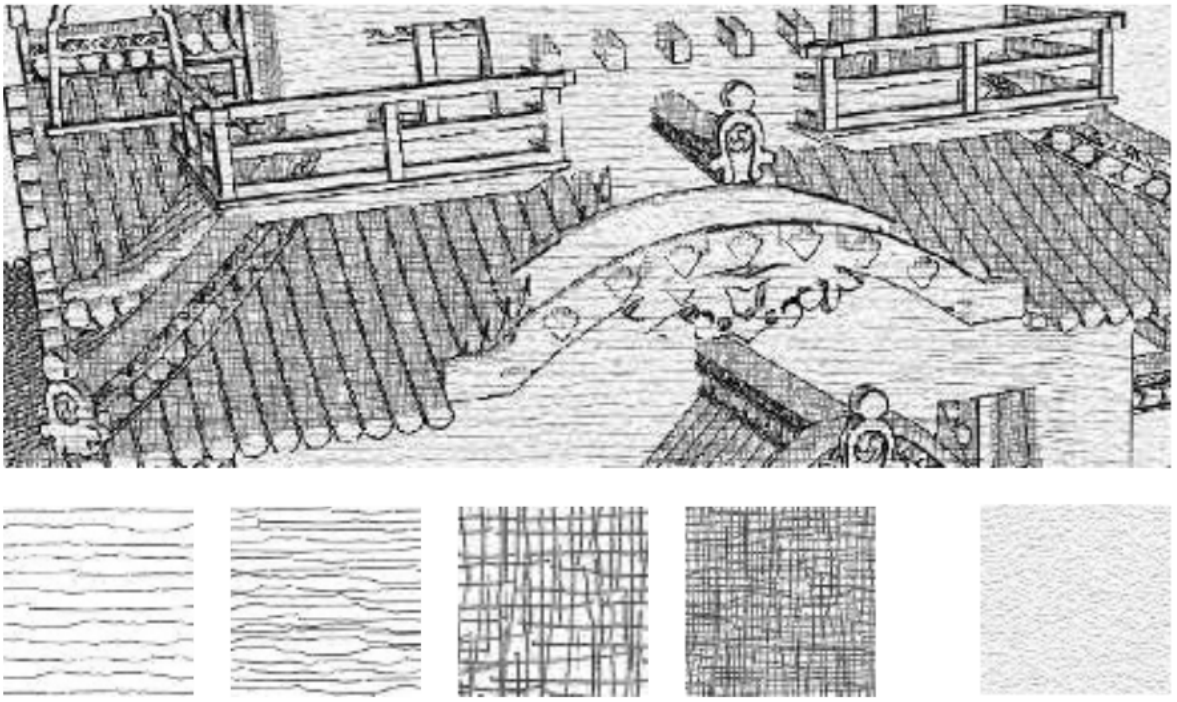


图 15.20：使用纹理调色板、纸张纹理和 contour 边缘进行渲染的图像。

一个显而易见的解决方案是：直接在物体表面上应用这些纹理。这样做的挑战在于，这些基于笔触的纹理（stroke-based texture）需要保持相对均匀的笔触粗细和笔触密度，才能看起来令人信服。如果这个纹理被放大了，那么笔触就会显得太粗；如果这个纹理被缩小了，那么笔触要么会变得很模糊，要么会变得很细并且会出现噪声（这取决于是否使用了 mipmap）。Praun 等人[1442]提出了一种实时生成笔触纹理 mipmap 的方法，并以一种平滑的方式将其应用在表面上。这样做可以在物体距离发生变化的时候，依然保持屏幕上的笔触密度。

图 15.21：色调艺术贴图（TAM）。这些笔触被绘制到不同的 mipmap 层级中。每个 mipmap 层级中都包含其左侧和上方所有纹理中的笔触。通过这种方式，在 mipmap 层级和相邻纹理之间可以进行平滑的插值。

这个方法的第一步是构建这些需要使用的纹理，它们被称为色调艺术贴图（tonal art map，TAM）。这是通过在 mipmap 层次中绘制笔触来实现的，如图 15.21 所示。Klein 等人[905]在他们的“艺术贴图（art map）”中使用了一个类似的想法，来维持 NPR 纹理的笔触大小。有了这些纹理贴图，模型就可以通过在每个顶点上，对所需的色调贴图之间进行插值，从而完成笔触渲染。这种技术所生成的图像，更加具有手绘的感觉[1441]，如图 15.22 所示。

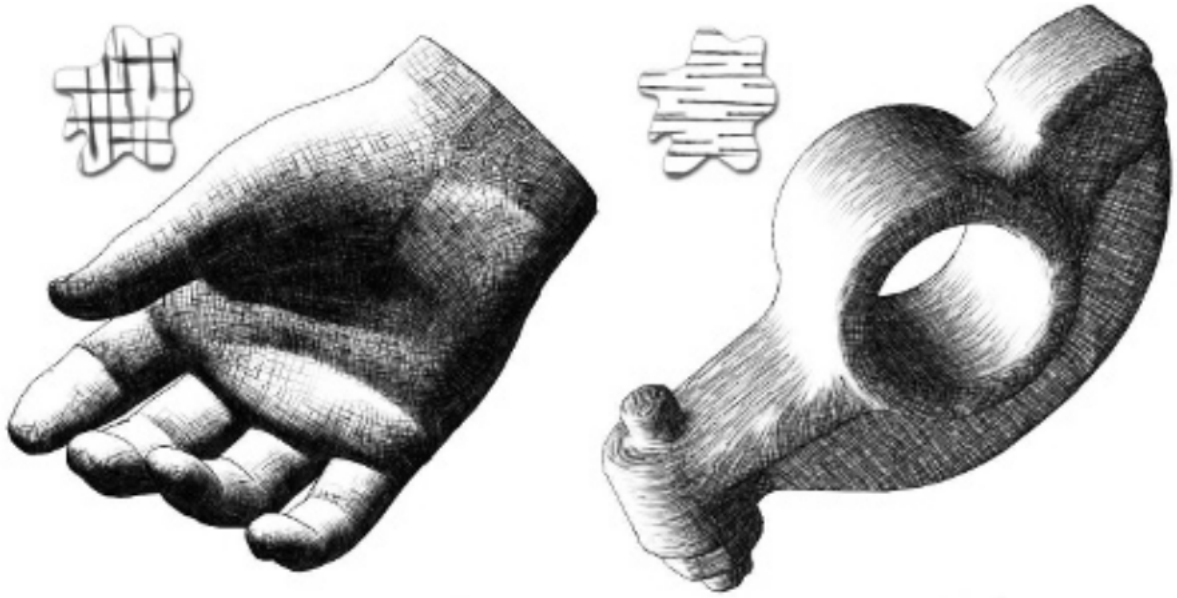


图 15.22：使用色调艺术贴图（TAM）进行渲染的两个模型。左上角的样本展示了用于渲染该模型所使用的重叠纹理图案。

Webb 等人[1858]提出了两种对 TAM 进行扩展的方法，它们可以提供更好的结果。第一种使用了一个体积纹理，并且允许使用颜色；第二种使用了一个阈值方案，可以改善抗锯齿效果。Nuebel [1291]给出了一种相关的木炭渲染方法，他所使用的噪声纹理也会沿着其中一个轴从暗变到亮。在渲染过程中，会使用强度值来沿着这个轴对纹理进行访问。Lee 等人[1009]使用 TAM 和其他技术，生成了令人印象深刻的图像，这些图像看起来就像是由铅笔绘制的一样。

关于笔触渲染，除了上面已经讨论过的技术之外，还有许多其他可能的技术。为了获得素描效果，可以让边缘发生抖动[317, 972, 1009]，或者超过它们的原始位置，如图 15.1 中右上部分和中下部分所示。

Girshick 等人[538]讨论了沿曲面上的主曲线方向线（principal curve direction line）来绘制笔触。这里的主曲线方向线，指的是从表面上的任何一个点，都存在一个指向最大曲率的第一主方向（first principal direction）切向量。第二主方向（second principal direction）则是一个垂直于第一主方向的切向量，这个方向是指

向了曲面曲率最小的方向。这些方向线对于曲面的感知而言十分很重要。它们还具有这样一个优点，即只需要为静态模型生成一次即可，因为这种类型的笔触与光照和着色无关。Hertzmann 和 Zorin [726]讨论了如何对主方向进行整理（clean up）和平滑。已经有大量的研究和开发工作，对如何使用这些方向以及其他应用在任意表面上的纹理数据进行了探索，例如驾驶模拟动画和其他的一些应用。相关读者可以参考 Vaxman 等人[1814]的报告。

嫁接（graftal）是指，可以将几何形状或者贴花纹理根据需要添加到表面上，从而产生特定的效果[372, 853, 1126]。它们可以通过所需的细节水平、表面相对于眼睛的方向，或者其他的因素来进行控制。这些方法也可以用来模拟钢笔或者画笔的笔触。图 15.23 展示了这样的一个例子。几何嫁接是程序化建模的一种形式[407]。

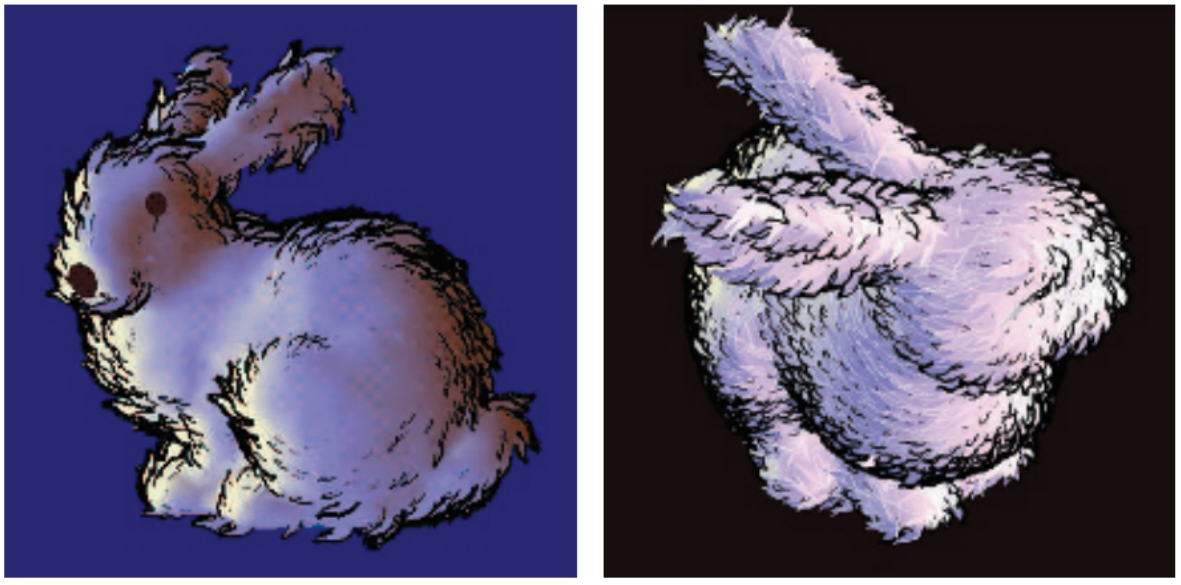


图 15.23：使用两种不同的嫁接风格来渲染 Stanford 兔子。

本章节中只涉及了 NPR 研究中的几个方向。有关更多更详细的信息，请参阅本章节最后的“补充阅读和参考资料”部分。NPR 领域中通常并不存在潜在的物理正确答案，因此我们也没有一个 ground-truth 的结果来进行参照。这既是一个问题，同时也是一种解放。各种技术在执行速度、渲染质量以及实现成本之间进行权衡。在交互渲染速率的严格时间限制下，大多数解决方案都会在一定条件下发生扭曲（bend）和失效（break）。根据应用程序中的实际需求来确定具体使用哪种表现良好的方案，是使得该领域成为一个迷人挑战的原因之一。

我们将大部分的注意力都集中在了一个特定的主题上，即 contour 边缘的检测和渲染。最后，我们将把注意力转向线条和文本的渲染。这两种非真实感的图元会被经常使用，同时它们在实现上也有着一些自己的独特挑战，因此值得单独进行介绍。

15.4 线条

渲染简单的实体“硬”线条，通常被认为是相对无聊的。然而，它们在 CAD 等领域中十分重要，可以用于查看底层模型的面片、或者用于识别物体的形状。它们在高亮显示选中物体和技术说明等领域中也非常有用。此外，其中所涉及的一些技术也同样适用于解决其他问题。

15.4.1 渲染三角形边缘

在实心三角形的上面正确渲染三角形的边缘，要比直接渲染这些三角形本身困难得多。如果一条边缘线条的位置与三角形的位置完全相同，那么我们如何确保这条线总是会在三角形前面被渲染呢？一个简单的解决方案是在渲染线条的时候，使用一个固定的偏移量[724]。也就是说，每条线在被渲染的时候，都要比实际距离更近一些，这样它就会位于表面前方。如果这个固定的偏移量过大，那么本来应当被隐藏的部分边缘可能会出现在表面前方，从而破坏了我们想要的效果。但是如果这个偏移量太小，那么靠近这条边缘的三角形曲面可能会遮挡部分甚至是全部的线条。如章节 15.2.2 中所述，调用诸如 OpenGL 的 glPolygonOffset API，可以根据线条的斜率，将表面向后进行移动。这种方法相当有效，但它并不是完美的。

Herrell 等人[724]的解决方案完全避免了使用偏移量。该方法使用了一系列步骤来对一个模板缓冲区进行标记和清除，以便能够在三角形上正确地渲染边缘。但是这种方法只适用于最小的三角形集合，因为每个三角形都必须单独进行绘制，并且对于每个三角形，都要清除模板缓冲区，这使得处理过程非常耗时。

Bærentzen 等人[86, 1295]提出了一种方法，它可以很好地映射到 GPU 上。该方法使用了一个像素着色器，使用三角形的重心坐标（barycentric coordinate）来确定到最近边缘的距离。如果该像素接近一个边缘，那么会使用边缘颜色来对其进行着色。以这种方式渲染的边缘，其粗细程度可以是任何想要的值，可以让其受到距离的影响，也可以让其保持恒定，如图 15.24 所示。这种方法最主要的缺点在于，contour 边缘的粗细程度只有内部线条的一半，因为每个三角形都只会负责渲染一半宽度的线条，因此内部线条会更粗。但是在实践中，这种不匹配现象通常不会太明显。

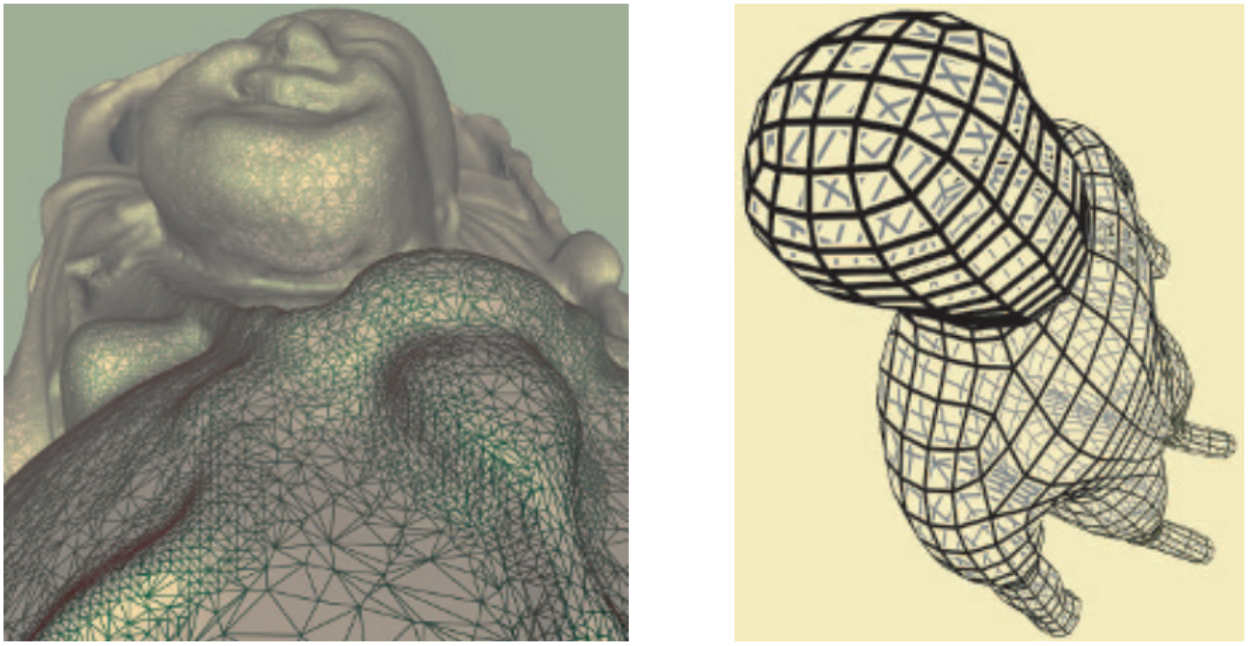


图 15.24：使用像素着色器生成的线条。左边是抗锯齿后的单像素宽度边缘；右边是带有光晕的可变粗细线条。

Celes 和 Abraham [242]对这一想法进行了扩展和简化，并对之前的工作进行了全面的总结。他们的想法是为每个三角形边缘都使用一个一维的纹理坐标集合，其中 1.0 代表定义边缘的两个顶点， 0.0 代表三角形的另一个顶点。他们使用纹理映射和 mipmap 链来提供恒定宽度的边缘。这种方法易于编码，并且能够提供一些实用的控制手段。例如：可以设置线条的最大密度，这使得网格密集的区域不会完全被边缘填充，因此不会变成纯色。

15.4.2 渲染遮挡线条

在正常的线框图中，我们并不会渲染三角形表面，模型中所有的边缘都是可见的。为了避免渲染那些被表面遮挡的线条，会将所有实心三角形都渲染到 z-buffer 中，然后再正常地绘制边缘[1192]。如果我们无法在渲染所有线条之前，首先渲染所有表面，那么也有一个成本稍高的解决方案，即使用与背景颜色相匹配的纯色来渲染这些表面。

在渲染这些线条的时候，也可以对其进行部分遮挡模糊（obscured），而不是完全将它们隐藏。例如：这些被遮挡的线条可以渲染为浅灰色，而不是根本不渲染。可以通过适当设置 z-buffer 的状态来实现这一点。首先像之前一样，将实心三角形渲染到 z-buffer 中，然后正常渲染一遍线条，最后再反转 z-buffer 的意义，即只绘制那些超出当前像素 z-depth 的线条，同时关闭对 z-buffer 的修改，这样渲染出来的线条就不会修改任何已有的深度值。然后再使用一种遮挡模糊的风格来重新绘制这些被

遮挡的线条。对于风格化版本的线条而言，则可以使用一个完全移除遮挡线条的过程[282]。

15.4.3 光晕

当两条线相交的时候，通常的惯例是擦掉一部分较远的线条，从而使得线条之间的前后顺序更加明显。这个操作可以相对容易地完成，只想要把每条线绘制两次即可，并在其中一次绘制一个光晕（holo）[1192]。这个方法可以通过绘制背景颜色来消除线条之间的重叠部分。首先，将所有的线条都渲染到 z-buffer 中，并将每条线表示为一个代表光晕的粗四边形。可以使用一个几何着色器来帮助创建这样的四边形。然后，使用正常的颜色来绘制出每个线条。被 z-buffer 遮挡的区域将会隐藏在其背后绘制的线条。同时必须使用一个偏移量或者其他方法，来确保每条细黑线都位于这个粗 z-buffer 四边形的上面。

相交于顶点的线条可能会被相互竞争的光晕部分隐藏。可以缩小这些代表光晕的四边形，但是可能会导致其他一些瑕疵。Bærentzen 等人[86, 1295]的线条渲染技术也可以用于实现光晕，如图 15.24 所示。光晕是逐三角形生成的，因此不会出现相互干扰的问题。另一种方法则是使用图像后处理（章节 15.2.3）来检测和绘制光晕。

有关我们这里所讨论的几种不同的线条渲染方法，结果在图 15.25 中进行了展示。

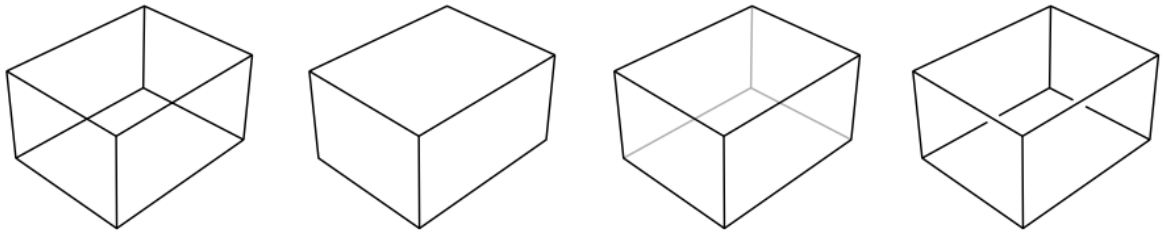


图 15.25：四线不同的线条渲染风格。从左到右分别是：线框、隐藏线条、模糊线条，光晕线条。

15.5 文本渲染

鉴于文本阅读对于文明而言具有重要的意义，因此人们花费了大量的精力来对其进行渲染。与许多其他对象不同的是，文本中单个像素的变化就可以产生显著的差异，例如将“l”变成“1”。本小节对用于文本渲染的主要算法进行了总结。

人眼对于亮度差异的感知，要比对颜色差异的感知更加敏感。这一事实至少从 Apple II 时代起[527]，就被用于提高感知的空间分辨率。基于这个想法的一个应用程序是微

软的 ClearType 技术，该技术建立在液晶显示器（liquid-crystal display，LCD）的一个特性之上。LCD 显示器上的每个像素都由三个垂直的彩色矩形所组成，即红色、绿色和蓝色。在一个 LCD 显示器上使用放大镜进行观察，我们可以看到这些微小的彩色像素矩形。在不考虑这些亚像素矩形颜色的前提下，这种配置方式所能提供的水平分辨率是屏幕水平像素的三倍。使用不同的着色来填充不同的亚像素，这种技术因此被称为亚像素渲染（subpixel rendering）。人眼会将这些颜色混合在一起，使得红色条纹和蓝色条纹变得无法察觉，如图 15.26 所示。这项技术于 1998 年首次公布，该技术对大型的、低 DPI 的 LCD 显示器具有很大的帮助。微软在 Word 2013 中停止使用了 ClearType 技术，显然是因为混合不同背景颜色的文本存在一些问题。Excel、各种网络浏览器、Adobe 的 CoolType、苹果的 Quartz 2D 以及 FreeType 和 SubLCD 等库，都使用了这项技术。Shemanarev 的一篇古老文章[1618]对这种方法的各种微妙之处和存在的问题进行了全面介绍。

图 15.26：同一个单词，分别使用了灰度抗锯齿（grayscale antialiasing）和亚像素抗锯齿（subpixel antialiasing）。当一个彩色像素显示在 LCD 屏幕上的时候，构成该像素的彩色、垂直亚像素矩形将会被点亮。这样做可以提供额外的水平空间分辨率。

这项技术是一个很好的例子，说明了人们在清晰地渲染文本上花费了多少精力。字体（font）中的字符（character）被称为字形（glyph），它通常是由一系列线段以及二次或者三次 Bezier 曲线进行描述的，图 17.9 展示了这样的一个例子。所有字体渲染系统的工作原理，都是确定这些字形会对其重叠的像素产生怎样的影响。FreeType 和 Anti-Grain Geometry 等库的工作原理，是通过为每个字形都生成一个很小的纹理，并在需要的时候重复使用它们。对于不同大小的字体，不同强调的字体（emphasis，即斜体或粗体）都会生成相应的纹理。

这些系统会假设每个纹理都是像素对齐的，即每个像素都只包含一个纹素，就像通常的文档一样。但是当文本应用于三维表面的时候，这个假设可能就不再成立了。使用一个包含一组字形的纹理是一种简单而流行的方法，但是这种方法也有一些潜在的缺点。虽然应用程序可能仍然会对齐文本，使其面对观察者，但是缩放和旋转将会使得每个像素占据一个纹素的假设不再成立。即使是进行了屏幕对齐，字体微调（font hinting）也可能不会被考虑在内。这里的微调是指对字形轮廓进行调整，使其与像素单元格相匹配的过程。例如，字符“I”的纵轴宽度为一个纹素，它最好被渲染为覆盖一列像素，而不是覆盖相邻两列像素的一半宽度，如图 15.27 所示。所有这些因素都

意味着，使用一个光栅纹理可能会导致显示模糊或者锯齿问题。Rougier [1515]全面介绍了纹理生成算法所涉及到的问题，并展示了如何在基于 OpenGL 的字形渲染系统中使用来自 FreeType 的微调。

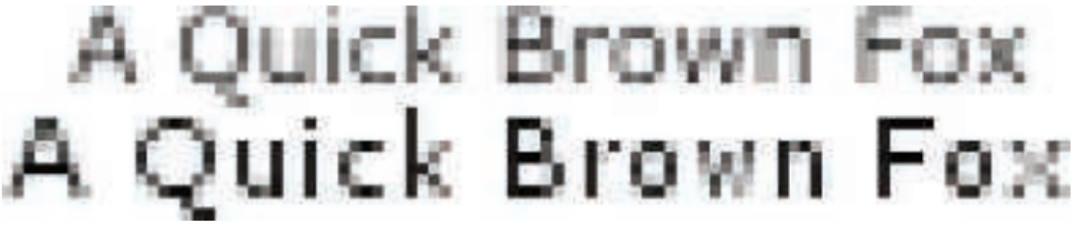


图 15.27：Verdana 字体的渲染结果，第一行没有使用微调，第二行使用了微调。[1515]

Pathfinder 库[1834]是一个使用 GPU 来生成字形的最近成果。它具有较低的设置时间和最小的内存开销，其性能要优于竞争对手基于 CPU 的字体渲染引擎。它使用曲面细分和计算着色器来生成和汇总每个像素上的曲线效果，并在性能较差的 GPU 上退回到使用几何着色器和 OpenCL。与 FreeType 一样，这些字形也会被缓存和重复使用。它具有高质量的抗锯齿效果，再加上如今高分辨率显示器的广泛使用，使得微调技术几乎要过时了。

将文本应用于不同大小和方向的任意表面，可以在不需要复杂 GPU 支持的情况下完成，同时仍然能够提供较为合理的抗锯齿效果。Green [580]提出了这样的一个系统，即 Valve 首次在《军团要塞 2》中所使用的系统。该算法使用了 Frisken 等人[495]提出的采样距离场（sampled distance field）数据结构。即纹理中的每个纹素，都保存了到字形最近边缘的带符号距离。这个距离场试图在纹理描述中对每个字形的精确边界进行编码。然后，使用双线性插值来给出该字母在每个样本上对 alpha 覆盖率的一个良好近似。图 15.28 展示了这样的一个例子。字符中的尖角在被双线性插值的时候可能会被平滑，这个问题可以通过在四个独立通道中编码更多的距离值来进行解决[263]。这种方法有一个局限性，即创建这些符号距离纹理是非常耗时的，因此需要进行预先计算并将结果存储下来。尽管如此，不少字体渲染库都是基于这种技术进行构建的[1440]，并且它们能够很好地适应移动设备[3]。Reshetov 和 Luebke [1485]总结了这些方面的工作，并给出了他们自己的方案，即在放大过程中调整样本的纹理坐标。

图 15.28：矢量纹理。左侧：字母“g”使用符号距离场来进行表达[3]。右侧：这个“no trespassing”的标志根据距离场来进行渲染的。通过将特定的距离范围映射到轮廓颜色上，可以在文本周围添加轮廓效果[580]。

即使没有缩放问题和旋转问题，使用中文字符的字体也可能需要数千个甚至更多的字形。一个高质量的大号字符需要使用更大的纹理来进行表示。如果以一定的角度来观察字形，则可能需要对纹理进行各向异性过滤。直接从边缘描述和曲线描述中来渲染字形，可以避免使用一个巨大的纹理，同时还可以避免来自采样网格的瑕疵。Loop-Blinn 方法[1068, 1069]使用一个像素着色器来直接计算 Bezier 曲线，我们将在章节 17.1.2 中对此进行讨论。这种技术需要一个曲面细分步骤，这在加载的时候可能会有很大的开销。Dobbie [360]通过为每个字符的包围盒都绘制一个矩形，并在一个 pass 中计算所有字形的轮廓来避免这个问题。Lengyel [1028]提供了一个健壮的评估器，它可以判断一个点是否位于字形内部，这对于避免瑕疵而言至关重要。他还讨论了一些有关计算的优化，以及一些效果的实现，例如：发光（glow）、阴影和多种颜色（这可以用于表情符号 emojy）。

补充阅读和资源

有关非真实感渲染和卡通渲染的灵感，请阅读斯 Scott McCloud 的《Understanding Comics》一书[1157]。从研究人员的角度来看，阅读 Hertzmann 有关使用 NPR 技术的论文[728]，有助于建立关于艺术和插图的科学理论。

《Advanced Graphics Programming Using OpenGL》这本书[1192]，尽管是在固定功能硬件的时代撰写的，但是在广泛的技术插图和科学可视化技术上，具有值得一读的章节。虽然也有些过时，但是 Gooches [563]和 Strothotte [1719]的书，都是

NPR 算法的良好起点。Isenberg 等人[799]和 Hertzmann [727]提供了有关 contour 边缘和笔触渲染技术的调研。Rusinkiewicz 等人[1521]在 SIGGRAPH 2008 的课程讲座中，也详细研究了笔触渲染技术，其中包含了一些比较新的工作。Benard 等人[130]对帧与帧之间的一致性算法进行了调研。对于一些艺术化的图像处理效果，我们建议感兴趣的读者参考 Kyprianidis 等人[949]的综述。国际非真实感动画与渲染学术会议（International Symposium on Non-Photorealistic Animation and Rendering，NPAR）的论文集，对该领域中的研究进行了综述。

Mitchell 等人[1223]提供了案例研究，其中关于工程师和艺术家如何进行合作，来为游戏《军团要塞 2》赋予独特的画面风格。Shodhan 和 Willmott [1632]讨论了游戏《孢子》中的后处理系统，包括油画效果、旧胶片效果以及其他一些效果的像素着色器实现。SIGGRAPH 2010 课程“Stylized Rendering in Games”是另一个十分具有价值的实际例子。特别地，Thibault 和 Cavanaugh [1761]展示了《无主之地》艺术风格的演变，并对这一过程中所面临的技术挑战进行了介绍和描述。Evans 的演讲[446]是一个迷人的探索，其中包含了用于实现特定介质风格的广泛渲染技术和建模技术。

Pranckevicius [1440]提供了一个有关文本渲染加速技术的调研，并提供了额外的资源链接。

Chapter 16 Polygonal Techniques 多边形技术

> Leopold Crelle——“It is indeed wonderful that so simple a fifigure as the triangle is so inexhaustible.”

> 利奥波德·克雷尔——“像三角形这样简单的图形竟如此取之不尽，这的确令人惊奇。”（德国数学家；1780—1855）

到目前为止，我们在进行渲染之前，都会默认假设这些等待渲染的模型正好是我们所需要的格式，并且具有正确的细节。而在现实实践中我们很少会这么幸运，各种建模软件和数据捕获设备都有着自己独特的怪癖（quirk）和局限性，可能会导致数据集和渲染结果存在歧义和错误。不仅如此，通常我们还需要在存储大小、渲染效率和结果质量之间进行权衡。在本章节中，我们将讨论多边形数据集中可能会遇到的各种问题，以及对这些问题的修复和解决方法。然后我们会介绍一些高效渲染和存储多边形模型的技术。

在交互式计算机图形学中，这种多边形表示方法的首要目标就是视觉精度和渲染速度。其中的“精度（accuracy）”是一个取决于上下文的术语。例如：一名工程师希望能够以交互速率来检修机器部件，并要求在任何时候都能看到物体上的每个斜面和倒角。相比之下，如果游戏的帧率足够高，那么某一帧中出现一些小错误或者不准确的地方是完全允许的，因为这些小错误可能并不会发生在当前玩家注意力集中的地方，同时这些小错误可能在下一帧中就消失了。在交互式图形学的相关工作中，了解这些待解决问题的边界条件是十分重要的，因为这些边界决定了可以使用什么类型的技术。

本章节中所涉及的内容包括曲面细分（tessellation）、整合（consolidation）、优化（optimization）、简化（simplification）和压缩（compression）。多边形可以有许多不同的表现形式，它们通常都会分割成更加容易处理的图元，例如三角形或者四边形。这个过程被称为三角形划分（triangulation），或者更一般地说，被称为曲面细分（tessellation）。

> 单词“Tessellation”中包含两个“l”，它可能是计算机图形学中最常拼错的单词；紧接着的是“frustum”。

整合（consolidation）是计算机图形学中的术语，它包括将单独的多边形合并到一个网格结构中，以及派生出新的数据来用于表面着色，例如法线等。优化（optimization）是指对网格中的多边形数据进行排序，从而使渲染速度变得更快。简化（simplification）是指删除一个网格中的不重要特征。压缩（compression）则关心的是用于描述网格信息的各种元素所需要的存储空间，并致力于最小化这个存储空间。

三角形划分确保了给定的网格描述能够被正确显示。合并进一步改善了数据的显示，并且可以通过允许共享计算和减少内存开销来提高速度。优化技术可以进一步提高处理速度。简化可以通过删除不需要的三角形来提高速度。压缩可用于进一步降低整体的内存占用，进而通过减少内存开销和总线带宽来提高速度。

16.1 三维数据的来源

通常有以下几种方法可以创建或者生成多边形模型：

- 直接将信息写入几何描述文件。
- 编写一些程序来生成这样的数据，这称为程序化建模（procedural modeling）。
- 将其他形式的数据转换为表面数据或者体积数据。例如：将蛋白质数据转换为一组球体和圆柱体。
- 使用建模软件构建或者雕刻出一个物体模型。
- 对某个物体进行拍照，使用一张或者多张照片来重建表面信息，这被称为摄影测量（photogrammetry）。
- 使用三维扫描仪、数字化仪（digitizer）或者其他传感设备，在不同的点上对真实世界中的模型进行采样。
- 在某些空间体积中生成表示相同值的等值面，例如来自 CAT 或者 MRI 的医学扫描数据，或者是在大气中测得的压力数据和温度数据。
- 对上述这些技术进行组合使用，从而生成多边形模型数据。

在建模软件的领域中，有两种主要类型的建模软件：基于实体的和基于表面的。基于实体的建模软件通常出现在计算机辅助设计（CAD）领域中，并且通常会着重于构建与实际加工过程相对应的建模工具，例如切割、钻孔和刨削等。在这些建模软件内部有一个计算引擎，用于严格操作物体的底层拓扑边界。为了进行显示和分析，这些建模软件都具有 faceter，faceter 是一种软件，它可以将内部的模型表示转换为可以显示的三角形。例如：数据库中的球体可以通过一个中心点和一个半径进行表示，而 faceter 则可以将其转换为任意数量的三角形或者四边形表示。有时候，最好的渲染加速方法其实就是最简单的：在使用 faceter 的时候，适当降低所需的视觉精度，可以减少需要生成的三角形数量，从而提高渲染速度并节省存储空间。

在 CAD 工作中有一个重要的考虑因素，即所使用的 faceter 是否是为图形渲染而设计的。例如：用于有限元方法（fifinite element method，FEM）的 faceter，其目的是将表面分割成几乎具有相同面积的三角形。这样的细分过程很适合进行三角形简化，因为它们包含了许多在图形中用不到的数据。类似地，有一些 faceter 所生成的三角形集合非常适合用于 3D 打印，从而生成一个现实世界中的物体，但是这种三角形数据缺少顶点法线，因此不适合用于快速的图形显示。

类似 Blender 或者 Maya 这样的建模软件，并不是基于实体构建的。相反，物体会使用各自的表面进行定义。与基于实体的建模软件一样，这些基于表面的系统可以使用一些内部表示和 faceter 来显示物体，例如样条线或者细分表面（subdivision surface，详见第 17 章）。这些建模软件允许用户直接对表面进行操作，例如添加或者删除三角形、顶点等；同时，用户还可以手动降低模型的三角形数量。

还有一些其他类型的建模软件，例如基于隐式曲面的建模系统（包括“blobby”metaball）[67, 558]，它们还会使用一些诸如混合、权重和场（field）等概念。这些建模软件可以根据某个函数的解来生成表面数据，例如 $f(x, y, z) = 0$。然后再使用诸如移动立方体（marching cube）之类的多边形化技术，来创建用于显示的三角形集合（章节 17.3）。

点云数据是简化技术的有力候选者，即点云数据可以通过应用简化技术来进行优化。点云中的数据往往会以固定的间隔进行采样，因此其中的很多样本点对于形成表面的视觉感知作用很小，这些样本的影响可以忽略不计。相关的研究人员花了几十年的时间来研究对缺陷数据的过滤，以及从点云数据中重建出多边形网格的技术[137]。章节 13.9 中包含了这个话题的更多内容。

对于从扫描数据中生成的网格，可以执行任意数量的清洗（cleanup）操作或者是其他高阶操作（higher-order operation）。例如：分割技术（segmentation

technique）尝试对一个多边形模型进行分析，并试图识别出单独的、可分离的部分[1612]。这样做有助于创建动画、应用纹理贴图、匹配形状以及其他操作等。

还有许多其他方法可以生成用于表面的多边形数据。这个过程中最关键的是要理解数据是如何进行创建的，以及是为了实现什么目的而创建的。通常，这些数据并不是专门为了高效的图形显示而生成的。此外，还存在许多不同的三维数据文件格式，任意两种格式之间的转换通常并不是无损的。理解这些输入数据可能遇到的限制和问题，是本章节中的重要主题。

16.2 曲面细分和三角形划分

曲面细分（tessellation）是将一个表面分割为一组多边形的过程。这里我们将专注于多边形表面的细分，有关曲面的细分将在章节 17.6 中进行讨论。进行多边形曲面细分的原因有很多，其中最常见的理由就是，所有的图形 API 和图形硬件都针对三角形进行了特殊优化，三角形的效率很高。三角形就像是现实世界中的原子一样，它们可以表示任何表面并进行渲染。将一个复杂的多边形转换为三角形的过程被称为三角形划分（triangulation）。

在对多边形进行细分的时候，有几个可能的细分目标，例如：我们想要使用的一种算法，它只能处理凸多边形，这种曲面细分被称为凸分割（convex partitioning）。还有一些其他的目标，比如我们想要使用一些全局光照算法，该算法要求在模型顶点上存储阴影效果或者相互反射效果，这个时候我们可能需要对表面进行细分（网格化）[400]。图 16.1 中展示了不同类型的曲面细分。还有一些非图形的曲面细分原因，例如我们要求每个三角形的面积不能大于某个给定的值，或者是三角形的内角不能太小，要大于某个最小角度。一种被称为 Delaunay 的三角形划分方式，要求在每个三角形顶点上都放置一个具有一定半径的圆，这个圆内部不能包含任何其他的任何顶点，这样做可以使得三角形的最小内角最大化。虽然这些限制通常是针对于非图形应用的（例如有限元分析），但是它们也可以用于改进表面的外观。在渲染中，我们也需要尽量避免出现细长的三角形，因为在对远处顶点进行插值的时候，它们可能会造成瑕疵。同时，这种细长三角形的光栅化效率也很低[530]。

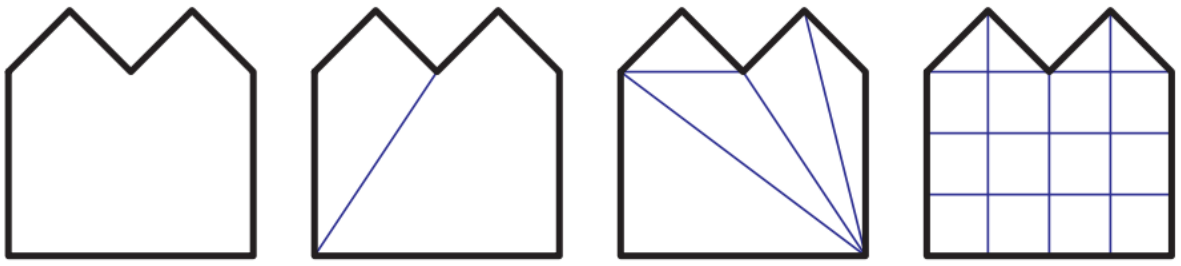


图 16.1：各种类型的曲面细分。第一个多边形没有进行细分；第二个多边形被划分为凸区域；第三个多边形被三角化；第四个多边形则被均匀网格化。

大多数曲面细分算法都在二维平面中进行工作，它们会假设多边形中的所有点都在位于同一个平面上。然而，一些模型生成系统可能会生成严重扭曲的非平面多边形，这个问题有一个十分常见的例子，即从侧面观察的弯曲四边形，这可能会形成所谓的沙漏四边形（hourglass）或者领结四边形（bowtie），如图 16.2 所示。虽然这种特定的多边形可以通过创建一条对角线来实现三角形化，但是一些更加复杂的弯曲多边形就不那么容易进行控制了。

图 16.2：从侧面观察弯曲的四边形，它形成了一个不那么明确的领结形状或者沙漏形状，它包含了两个可能的三角形。

当可能出现扭曲多边形的时候，一个快速的纠正操作是将顶点投影到一个平面上，这个平面垂直于多边形的近似法线。这个平面的法线通常是通过计算三个正交平面（ xy ， xz ， yz ）上的投影面积得到的。也就是说，直接去掉 x 坐标，得到的多边形在 yz 平面上的面积，就是新的 x 分量；同理，在 xz 平面上的面积是 y 分量，在 xy 平面上的面积是 z 分量。这种计算平均法线的方法被称为 Newell 公式[1505, 1738]。

投影到这个平面上的多边形可能仍然会出现自相交问题，即该多边形的两条边或者多条边会相互交叉。这时候就需要一些更加复杂、成本更高的计算方法了。Zou 等人[1978]对之前基于最小化表面积、或者最小化二面角的曲面细分工作进行了讨论，并提出了在一个集合中优化若干非平面多边形的算法。

Schneider 和 Eberly [1574]、Held [714]、O'Rourke [1339]和 de Berg 等人[135]，都对各种三角形划分方法进行了综述。其中最基本的三角形划分算法，是对多边形上任意两点之间的线段进行检查，看看它们是否会与多边形的任何边相交或者重叠。如果这条线段会发生相交或者重叠的话，那么它就不能用来对多边形进行分割，此时我们需要检查下一个可能的点对。如果这条线段满足要求的话，则使用它来将多边形一分为二；并重复使用相同的方法，对新生成的多边形继续进行三角形划分。这种方法的效率很低，其时间复杂度为 $O(n^3)$ 。

一种更加高效的方法是耳切法（ear clipping），其时间复杂度为 $O(n^2)$ 。首先，使用一个 pass 来对多边形进行遍历，从而找到耳结构，即查看所有顶点索引为 i ，$(i+1)$ ，$(i+2)$ （模 n ）的三角形，并检查线段 $i,(i+2)$ 是否满足与任何多边形边都不相交。如果确实不相交的话，则三角形 $(i+1)$ 就构成了一个耳结构，如图 16.3 所示。每个符合要求的耳结构会依次从多边形中移除，并对顶点 i 和 $(i+2)$ 处的三角形进行重新检查，看看它们现在是否构成了耳结构。最终，多边形中的所有耳结构都会被移除，这个多边形也就被成功三角形化了。还有一些更加复杂的三角形划分方法，其时间复杂度为 $O(n \log n)$ ，甚至某些方法在一些特殊情况下，时间复杂度可以达到 $O(n)$ 。Schneider 和 Eberly 给出了耳切法[1574]以及其他快速三角形划分方法的伪代码。

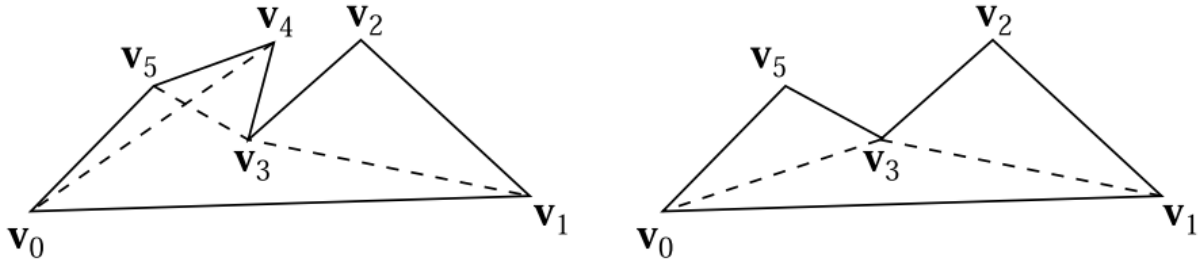


图 16.3：耳切法。左侧是一个多边形，其中的顶点 $\mathbf{v}_2$ ， $\mathbf{v}_4$ ， $\mathbf{v}_5$ 都构成了潜在的耳结构。在右侧，位于 $\mathbf{v}_4$ 的耳结构被移除，然后对相邻的顶点 $\mathbf{v}_3$ 和 $\mathbf{v}_5$ 进行重新检查，看看它们现在是否形成了耳结构。重新检查可以发现， $\mathbf{v}_5$ 处此时重新形成了耳结构。

与三角形划分相比，将多边形划分为凸区域（convex region）在存储和进一步的计算成本方面都更加高效。Schorn 和 Fisher [1576]给出了一个代码的健壮性测试。凸多边形（convex polygon）可以很容易地实用扇形三角形或者条状三角形进行表示，这个话题将在章节 16.4 中进行讨论。还有一些凹多边形（concave polygon）也可以被表示为扇形，这种多边形被称为星形（star-shaped），但是想要检测这种类型的多边形需要进行更多的工作[1339, 1444]。Schneider 和 Eberly [1574]给出了两种凸多边形划分方法，其中一种是快速但 dirty 的方法，另一种则是最优方法。

多边形并不总是由单一的轮廓所构成，图 16.4 展示了一个由三个轮廓（outline，也称为 loop 或者 contour）所构成的多边形。这种形式的多边形总是可以通过在环之间生成连接边（也称为锁孔边 keyholed 或者桥接边 bridge edge），来将其转换为一个单轮廓的多边形。Eberly [403]讨论了如何找到定义这些边缘的可见顶点。这个转换过程也可以反过来，在多边形中检索单独的环结构。

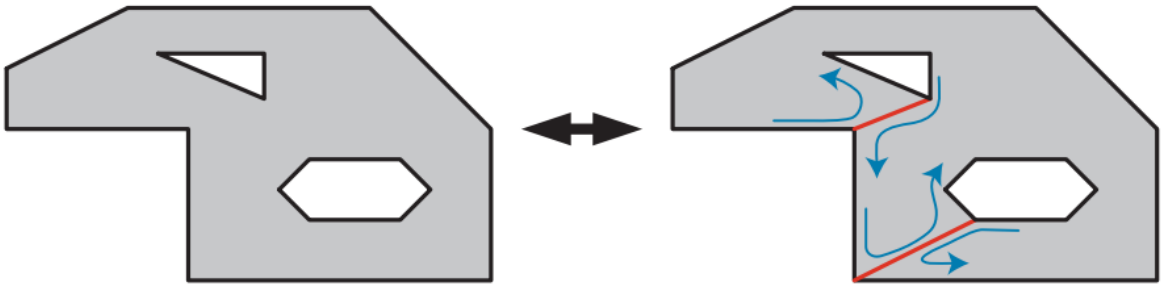


图 16.4：将由三个轮廓所组成的多边形转换为单个轮廓的多边形。右侧的连接边使用红色进行表示，多边形内部的蓝色箭头，代表了形成单个环的顶点访问顺序。

编写一个健壮且通用的三角形划分器（triangulator）是一项艰巨的任务。总是会出现各种各样微妙的 bug、病态的特例、以及一些精度问题，这导致很难编写万无一失的代码。解决三角形划分问题的一种方法是，直接使用图形加速器本身来渲染这些复杂的多边形。这个多边形被作为一个三角形扇（triangle fan）渲染到模板缓冲区中。通过这样做，需要被填充的区域会被绘制奇数次，而凹区域和孔洞则会被绘制偶数次。通过对模板缓冲区使用反转模式，在第一个 pass 结束的时候仅会对填充区域进行标记，如图 16.5 所示。而在第二个 pass 中，这个三角形扇形会被再次渲染，并通过使用模板缓冲区，只允许对填充区域进行绘制。通过对每个环形成的三角形进行绘制，这个方法甚至可以用于渲染具有多个轮廓的多边形。这个巧妙方法主要缺点在于，每个多边形都必须使用两个 pass 来进行渲染，并且每一帧都要清空模板缓冲区，而且无法直接使用深度缓冲。这个技术可以用于显示某些用户的交互，例如用户选择了一块复杂的区域，可以使用这个方法来动态绘制这个复杂区域的内部。

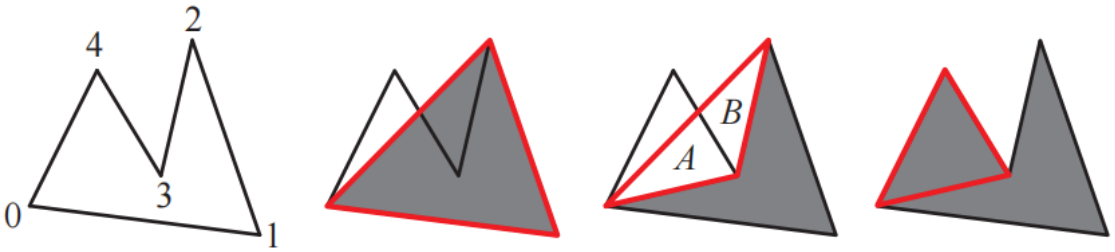


图 16.5：通过光栅化来完成三角形划分，使用奇偶校验（odd/even parity）判断哪些区域是可见的。最左边的多边形，从顶点 0 开始，会被作为三个三角形扇绘制到模板缓冲区中。在第二幅图中，第一个三角形 $[0, 1, 2]$ 填充其覆盖区域，这个区域内还包括了位于多边形外部的空间。在第三幅图中，三角形 $[0, 2, 3]$ 填充其覆盖区域，此时将区域 A 和区域 B 的奇偶性更改为偶数次绘制，从而使其变为空，即擦除。在第四幅图中，三角形 $[0, 3, 4]$ 填充了多边形的剩余部分。

16.2.1 着色问题

有时网格数据会以四边形的形式给出，为了正确显示这些四边形网格，必须要先将其转换为三角形。偶尔会出现一些凹四边形，在这种情况下，只有一种方法可以对其进行三角形化。而在其他凸四边形的情况下，可以选择两条对角线中的任意一条来对四边形进行分割。稍微花一点时间来选择一个更好的对角线，有时候可以获得更好的视觉效果。

有几种不同的方法来决定如何对一个四边形进行分割，其中最关键的思想是最小化新边顶点之间的差异。对于那些顶点上没有额外数据的平面四边形，最好是选择较短的那条对角线。如果使用了一些简单的烘焙全局光照解决方案，会在顶点上存储一些颜色数据，那么此时应当选择颜色差异较小的那条对角线[17]，如图 16.6 所示。这种将两个差异最小的对角连接起来的做法（会通过使用某种启发式方法进行确定），对于最小化画面瑕疵有一定的帮助。

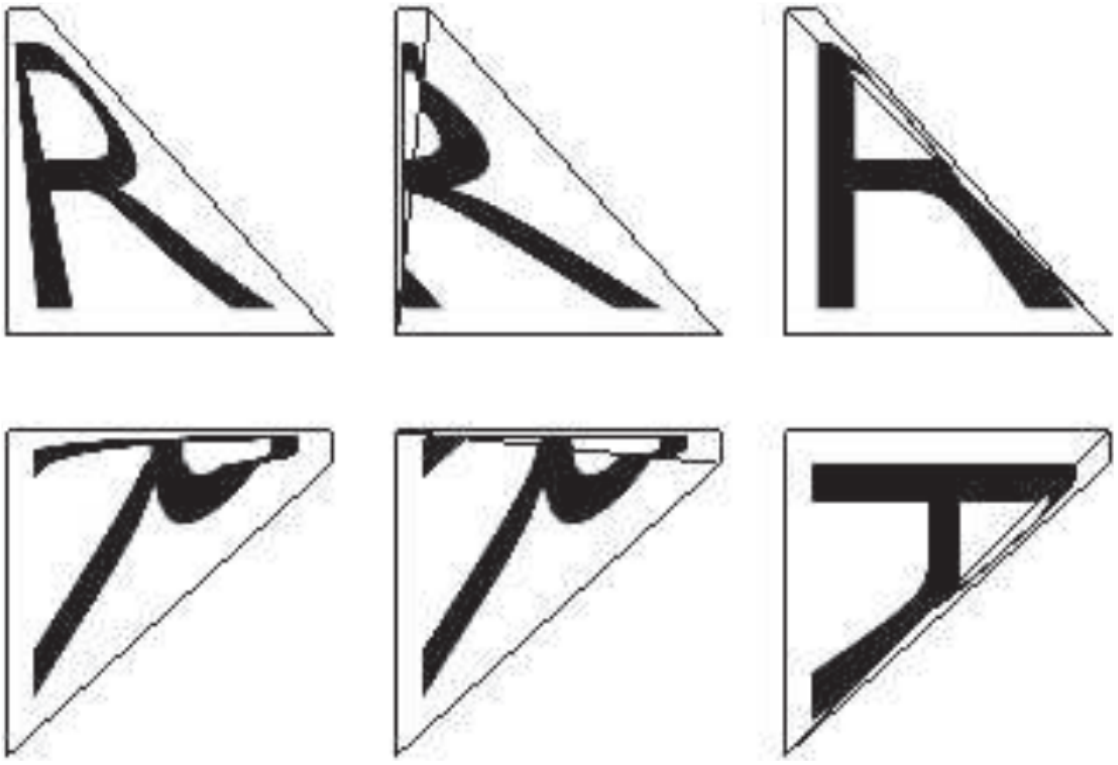


图 16.7：左上角展示了设计者的原本意图，一个带有字母“R”的方形纹理贴图，被应用在了一个扭曲的四边形上。第一行中间和右边的两张图片，展示了两种不同的三角形划分方法，以及它们之间的区别。第二行对多边形进行了旋转操作，其中最左侧的非三角形化的四边形，其外观表现会发生改变。

有些时候，三角形可能无法很好地表达设计师的意图（intent）。此时如果将纹理应用到一个扭曲的四边形上，那么无论使用哪种对角线分割方法，都无法保留设计师原本的意图。也就是说，在非三角化的四边形上进行简单的水平插值，即从左侧边缘到

右侧边缘进行插值，也会失败，图 16.7 展示了这个问题。之所以会出现这个问题，是因为应用在表面上的图像在显示的时候会被扭曲。一个三角形只有三个纹理坐标，因此它可以建立一个仿射变换（affine transformation），但是无法进行扭曲操作。在三角形上，一个基本的 (u, v) 纹理最多只能被剪切（shear），而无法被扭曲（warp）。Woo 等人[1901]对这个问题进行了更深入的讨论。针对这一问题，有以下几种可能的解决方案：

- 提前对纹理进行扭曲操作，并使用新的纹理坐标来将这个纹理重新应用到表面上。
- 将表面细分为更加精细的网格，但是这样只能缓解问题，无法解决问题。
- 使用投影纹理（projective texturing），在运行过程中对纹理进行动态扭曲[691, 1470]。但是这样会使得表面上的纹理间距不均匀。
- 使用一个双线性映射方案[691]。这可以通过在每个顶点上的增加额外的数据来进行实现。

虽然这种纹理扭曲现象听起来像是一种病态情况（pathological case），但是在某种程度上，每当要应用的纹理数据与底层四边形的比例不匹配时，就会发生这种情况；也就是说，几乎在任何弯曲表面上都会发生这种纹理扭曲现象。在圆锥体上会发生一个极端的情况，当一个圆锥体被纹理化和细分化（faceted）的时候，位于圆锥顶端的三角形顶点将会具有不同的法线。这些顶点法线并不会被相邻的三角形所共享，因此会出现着色不连续的情况[647]。

16.2.2 边界开裂和 T 顶点

我们将在第 17 章中对曲面话题进行详细讨论，这些曲面通常会被细分成多边形网格进行渲染。这种曲面细分是通过沿着定义表面的样条曲线进行步进来完成的，会在步进的过程中计算顶点的位置信息和相应的法线信息。当我们使用一个简单的步进方法时，在样条曲面相交的地方可能会出现问题。在一条共享边界上，两侧曲面上的点需要相互重合，取决于不同模型的性质，有时候自然而然就会发生这种我们想要的情况；但是在通常情况下，如果没有进行足够的注意，在一条样条曲线上生成的顶点并不会与其相邻曲线上的点相匹配。这种效应被称为边缘开裂（Edge Cracking），当观察者通过这些表面裂缝进行观察的时候，会导致令人烦恼的视觉瑕疵。即使观察者无法直接通过裂缝进行观察，但是由于着色插值方式的不同，也会经常看见边界之间的接缝。

修复这些裂缝的过程称为边缘缝合（edge stitching），我们的目标是确保沿着共享边界的所有顶点都会被两侧的样条曲面所共享，这样就不会出现裂缝了，如图 16.8 所示。章节 17.6.2 讨论了如何使用自适应曲面细分来避免样条表面的开裂。

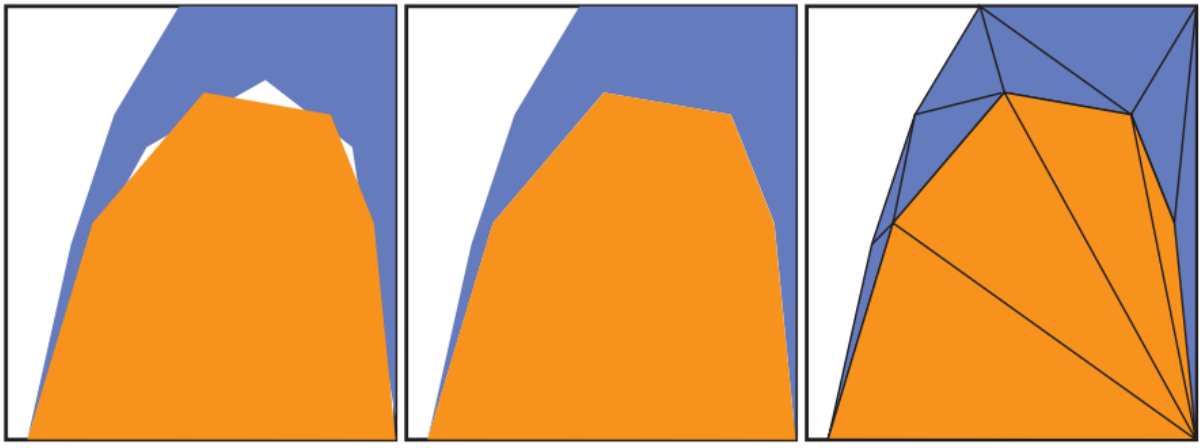


图 16.8：左侧展示了两个表面相遇处的裂缝。中间则是通过匹配边缘顶点进行修复的裂缝。右图展示的是修正后的网格。

在连接平面的时候会遇到一个类似的问题，即 T 顶点（T-vertices）。当两个模型的边界相交，但是没有共享沿着边界的所有顶点时，就会出现这类问题。即使这些边界在理论上完美相交，但是如果渲染器没有足够的精度来表示屏幕上的顶点位置时，也有可能会出现裂缝。现代的图形硬件会使用亚像素寻址（subpixel addressing）来帮助避免这个问题[985]。

更加明显的是，即使没有精度问题，也可能会出现着色瑕疵[114]，图 16.9 展示了这个问题。可以通过找到这样的边界，并确保它们与相邻面共享相同的顶点来解决这个问题。另外一个可能出现的问题是，使用简单的扇形算法可能会产生退化的三角形（degenerate，即面积为 0 的三角形）。例如在图 16.9 中，假设右上角的四边形 $\mathbf{abcd}$ 被三角化为两个三角形，即三角形 $\mathbf{abc}$ 和三角形 $\mathbf{acd}$。其中三角形 $\mathbf{abc}$ 是一个退化的三角形，因此点 $\mathbf{b}$ 是一个 T 顶点。Lengyel [1023]讨论了如何找到这样的顶点，并提供了将凸多边形重新进行正确三角形划分的代码。Cignoni 等人[267]描述了一种方法，即当 T 顶点的位置已知时，可以避免产生退化的三角形。该算法的时间复杂度为 $O(n)$，并且能够保证最多只生成一个条状三角形和扇状三角形。

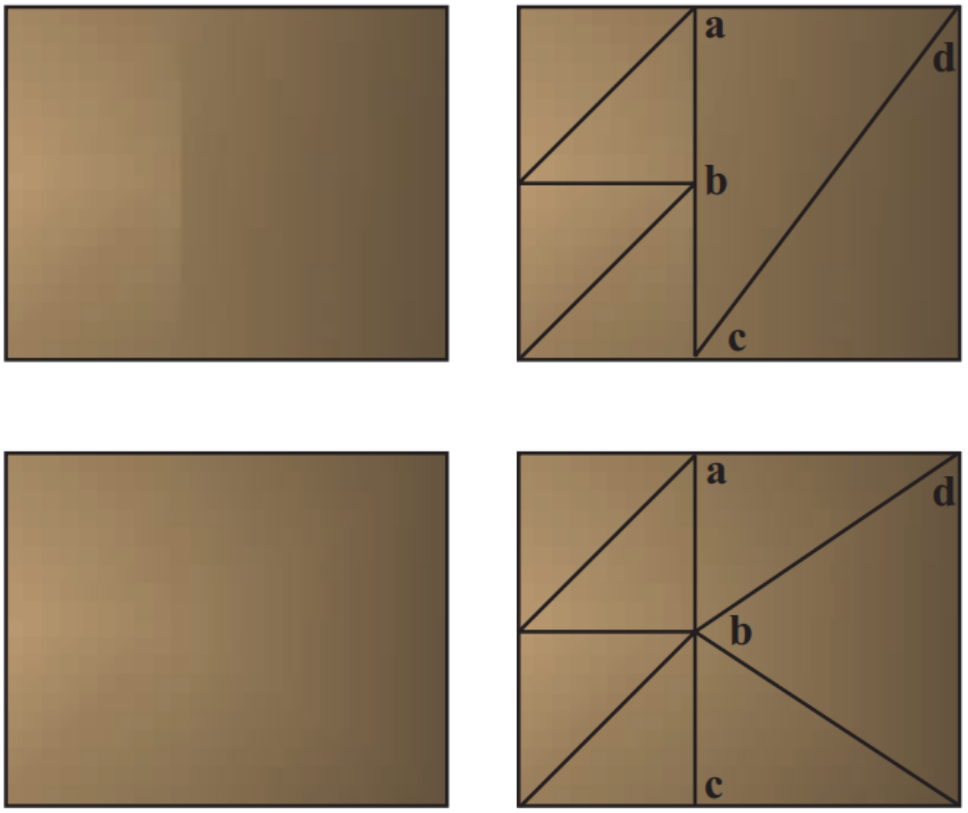


图 16.9：在第一行中，表面的底层网格显示出了着色不连续的现象。顶点 $\mathbf{b}$ 是一个 T 顶点，因为它属于左侧的三角形，而并不属于三角形 $\mathbf{acd}$ 中的一部分。一种解决方案是将这个 T 顶点添加到这个三角形中，并创建两个新的三角形 $\mathbf{abd}$ 和 $\mathbf{bcd}$ （图中未显示）。细长的三角形会容易引起其他的着色问题，因此对其进行重新三角形划分通常是一个更好的解决方案，如第二行所示。

16.3 整合

一旦模型通过了所需要的任何曲面细分算法，我们就有了一组表示模型的多边形网格。这里有一些操作可能会对数据的显示有所帮助。其中最简单的操作就是检查这些多边形本身是否是正确构建的，是否至少包含 3 个不同的顶点位置，以及这些顶点是否不共线。例如：如果三角形中的任意两个顶点重合，那么这个三角形的面积就为 0 了，因此可以丢弃这个三角形。请注意，在本小节中，我们所针对的是多边形，而不仅仅是三角形。根据我们所实现的具体目标，存储多边形数据可能会更加高效，而不是立即将其转换为三角形来进行显示。三角形划分会创建更多的边缘，这反过来又为接下来的操作制造了更多的工作。

对多边形而言，一个常用的过程是合并（merging），它指的是在面片之间找到共享的顶点。另一个操作被称为定向（orientation），即构成同一个表面的所有多边形应当面向相同的方向。网格的定向对于之后的一些算法而言十分重要，例如背面剔除、

折痕边缘检测、正确的碰撞检测和碰撞响应等。与定向相关的操作是顶点法线生成（vertex normal generation），它会使得表面看起来很光滑。我们将这类技术统称为整合算法（consolidation algorithm）。

16.3.1 合并

有些数据会以不相连多边形的形式出现，它通常被称为多边形 soup 或者三角形 soup（译者注：这里的 soup 是汤的意思，代表了集合中元素混乱无序的状态）。存储单独的多边形会浪费内存，而渲染单独的多边形则会带来极低的效率；由于这些原因以及其他的一些原因，通常会将单个多边形合并为一整个多边形网格（polygon mesh）。从最简单的角度来说，一个网格由一组顶点和一组轮廓所组成。其中每个顶点上都包含一个位置数据，以及一些其他的可选数据，例如着色法线、纹理坐标、切向量和颜色等。而每个多边形轮廓都包含了一个整型的索引列表。每个索引都是一个从 0 到 $n-1$ 的数字，其中 n 是顶点的数量，每个索引都指向顶点列表中的一个顶点。通过这样的存储方式，每个顶点就只需要存储一次即可，它可以被任意数量的多边形所共享。而我们所说的三角形网格，其实是一个只包含三角形的多边形网格。我们将在章节 16.4.5 中，对网格数据的存储方案进行深入讨论。

给定一组不相连的多边形，可以有几种方法对其进行合并。其中一种方法是使用哈希[542, 1135]。我们首先将顶点计数器初始化为 0。对于每个多边形，我们都尝试将该多边形中的每个顶点依次添加到一个哈希表（hash table）中，哈希表会根据顶点的数值进行散列。如果发现某个顶点并不在这个哈希表中，则将该顶点和相应的顶点计数器值一起存储在表中，然后将顶点计数器的值加 1，同时将这个顶点数据存储在最终的顶点列表中。如果发现某个顶点已经在哈希表中存在了，则检索该顶点的存储索引，通过存储指向该顶点的索引来对多边形进行保存。在处理完所有的多边形之后，顶点列表（vertex buffer）和索引列表（index buffer）也就构建完成了。

模型数据有时会出现这样一种情况，即分属于不同多边形的两个顶点，位置十分接近但是又并不完全相同。合并这类顶点的过程被称为焊接（welding）。可以通过使用排序以及一个更加宽松的位置相等函数，来完成高效的顶点焊接操作[1135]。

16.3.2 定向

对于模型数据，一个与质量相关的问题是面片的朝向。有一些模型数据的朝向本身就是正确的，其表面法线显式或者隐式地指向正确的方向。例如：在一些 CAD 任务中有这样一个标准，即正面观察一个多边形面片的时候，多边形轮廓上的顶点按照逆时针方向进行排序。这被称为缠绕方向（winding direction），在三角形中则使用右手

定则（right-hand rule）。想象一下，我们右手的手指以逆时针的顺序对多边形的顶点进行环绕，此时大拇指所指向的方向就是多边形的法线方向。这个方向与观察坐标系或者世界坐标系中所使用的左手系或者右手系无关，这个方向完全依赖于正面观察三角形时，世界中的顶点顺序。也就是说，如果将一个反射矩阵应用于在这个定向网格上，那么多边形中每个三角形的法线将会与其缠绕方向相反。

给定一个合理的多边形网格模型，下面是一种对网格进行定向的方法：

1. 为所有的多边形构建边-面结构。
2. 对这些边进行排序或者哈希，从而找出那些匹配的边。
3. 找出相互接触的多边形组。
4. 对于每一组多边形，按需对其进行翻转，从而保证朝向的一致性。

第一步是创建一组半边（half-edge）对象，这里的半边指的是多边形的一条边，它具有一个指向其关联面（多边形）的指针。由于一条边通常会被两个多边形所共享，因此通常会将这种数据结构称为半边。在创建各个半边对象的时候需要使用一些排序算法，使得半边的第一个顶点存储在第二个顶点之前。如果一个顶点的 x 坐标值较小，那么将其排在另一个顶点之前。如果两个顶点的 x 坐标是相等的，那么则使用 y 坐标进行比较；如果仍然相等，则使用 z 坐标进行比较。例如：顶点 $(-3, 5, 2)$ 会排在顶点 $(-3, 6, -8)$ 的前面，因为 x 坐标都为 -3，因此比较 y 坐标，发现 $5 < 6$ 。

这一步的目标是寻找哪些边是相同的。由于在存储每条边的时候，第一个顶点总是会小于第二个顶点，因此在比较边是否相同的时候，实际上就是将第一个顶点与第一个顶点相比较，第二个顶点与第二个顶点相比较，而不需要比较某一条边的第一个顶点与另一条边的第二个顶点这样的排列方式。这里可以使用一个哈希表来查找相互匹配的边[19, 542]，如果多边形中的顶点之前已经被合并过了，那么这些半边也可以使用相同的顶点索引来进行表示；可以将每条半边放入一个临时的列表中，与第一个顶点索引相关联，并使用这种方式来进行匹配。一个顶点平均有 6 条附着的边，这使得分组后的边缘匹配效率很高[1487]。

一旦这些半边完成了匹配，那么就可以知道相邻多边形之间的连接关系了，它们会构成一个邻接图（adjacency graph）。对于三角形网格而言，它可以表示为一个列表，其中每个三角形都有（最多）三个相邻的三角形面；而那些没有两个相邻多边形的边就是边界边缘。这些由边连接起来的多边形集合会形成一个连续的组，例如：一个茶壶模型包含两个组，分别是壶身和盖子。

下一步是赋予网格朝向的一致性，例如：我们通常希望所有多边形的轮廓都是逆时针方向的。对于一组内的连续多边形，可以选择任意一个多边形作为起始多边形。检查其每个相邻的每个多边形，并确定朝向是否一致。如果在遍历两个多边形的时候，发现边的遍历方向是相同的，那么就需要翻转这个相邻的多边形，如图 16.10 所示。递归检查这些相邻多边形的相邻多边形，直到这个连续多边形组中的所有多边形都检查过一次。

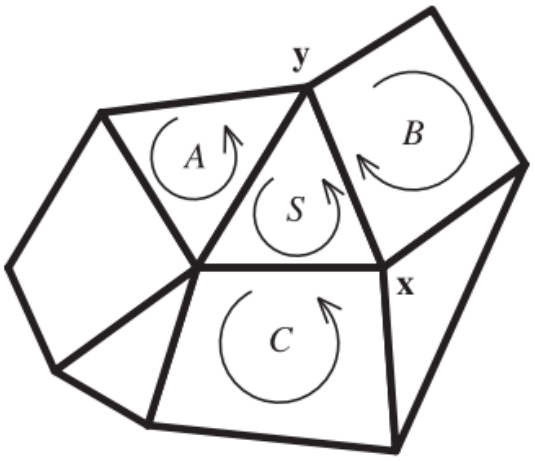


图 16.10：选择一个起始多边形 S，并检查它的邻居多边形。多边形 S 和多边形 B 共享的这条边，由于其上的顶点是以相同的顺序进行遍历的（从顶点 x 到顶点 y），因此 B 的轮廓需要颠倒过来才能满足右手法则。

虽然此时所有的多边形面的朝向都是正确的，但是它们可能会同时朝向物体内部。在大多数情况下，我们都希望这些多边形面朝模型外部。想要判断是否应当翻转所有面，有一个快速的测试方法，即计算该多边形组的带符号体积，并对最终的符号正负性进行检查。如果它是负数，则翻转顶点顺序和顶点法线。可以通过计算每个三角形的带符号标量三重积（译者注：其绝对值等于平面六面体的体积），并进行求和，从而来计算这个带符号体积。你可以在网站 realtimerendering.com 的线性代数附录中，了解这个体积计算方法。

这种方法适用于实体物体，但它并非是万无一失的。例如：如果物体是一个构成房间的 box，用户可能希望它的法线指向内部，从而朝向相机。如果物体并不是一个实体，而是一个表面描述的话，那么想要自动完成每个表面的定向是一件很棘手的事情。例如：如果两个立方体沿着一条边相连在一起，并且它们属于同一个网格，那么这条边将被四个多边形所共享，这会使得确定朝向变得更加困难。类似莫比乌斯环（Mobius）这样的单面物体，永远也不能完全定向，因为它没有内外的概念。即使对于那些表现良好的表面网格，也很难确定到底哪一面应当朝外。Takayama 等人

[1736]对之前的工作进行了讨论，并提出了自己的解决方案，该方法会从每个面上随机投射光线，并确定哪个朝向从外部更加可见。

16.3.3 实体性

不严谨地说，如果一个网格被确定了朝向，并且从外部可见的所有多边形都具有相同的朝向，那么我们可以说这个网格构成了一个实体（solid）。换句话说，只有网格的其中一侧是可见的，这种多边形网格被称为封闭的（closed）或者水密的（watertight）。

在知道了一个物体是实体之后，意味着我们可以使用一些背面剔除算法来提高渲染效率，详见章节 19.2。同时，这里的实体概念也是投射阴影体（章节 7.3）和其他一些算法的关键属性。例如：3D 打印机要求它所打印的网格是实体的。

最简单的实体性（solidity）测试就是检查网格中的每个多边形边缘，是否恰好被两个多边形所共享。对于大多数数据集而言，这个测试已经足够了。这样的表面可以被粗略地称为流形（manifold），具体来说是二维流形（two-manifold）。从技术上讲，流形表面是一个没有任何拓扑不一致性（topological inconsistency）的表面，这里的拓扑不一致性，指的是有三个或者更多个多边形共享同一条边缘，或者有两个及以上的顶角相互接触。构成实体的连续表面应当是一个没有边界边缘的流形。

16.3.4 法线平滑和折痕边缘

有一些多边形网格会形成曲面，但是这些多边形的顶点上并没有法线数据，因此无法使用曲率错觉（illusion of curvature）来进行渲染，如图 16.11。

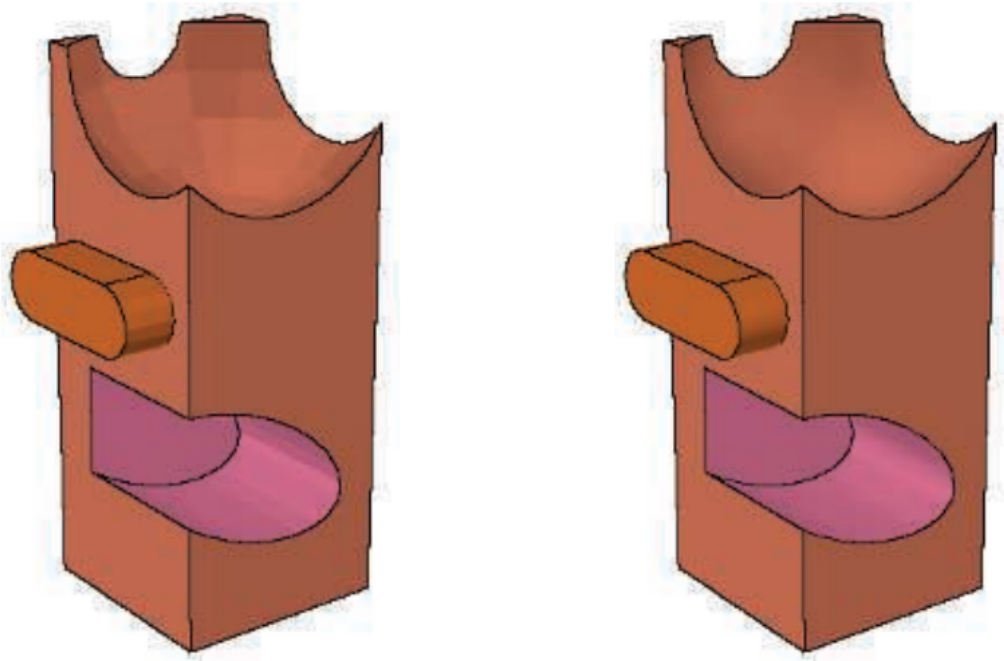


图 16.11：左边的物体没有逐顶点的法线；右边物体有顶点的法线。

有许多模型格式不会提供有关表面边缘的信息，章节 15.2 中我们介绍了不同类型的边缘。这些边缘是十分重要的，有以下几个原因。它们可以突出显示模型中某个区域，这个区域内包含了一组多边形；或者有助于非真实感渲染。由于这些边缘提供了重要的视觉暗示，因此渐进式的网格算法（章节 16.5）通常会避免对这些边缘进行简化。

在具有朝向的网格中，通常可以推导出合理的折痕边缘和顶点法线。一旦保证了朝向的一致性，并建立了邻接图，就可以使用一些平滑技术（smoothing technique）来生成顶点法线。有些模型格式可以通过指定多边形网格的平滑组（smoothing group）来提供帮助。这个平滑组值用于显式定义哪些多边形属于一组，这组多边形一起组成了一个弯曲表面。不同平滑组之间的边缘会被认为是尖锐的。

另一种对多边形网格进行平滑的方法是指定一个折痕角度。这个折痕角度值会与二面角（dihedral angle）进行比较，二面角是指两个多边形平面法线之间的夹角。这个折痕角度的数值通常设定在 20–50 度之间。如果我们发现两个相邻多边形之间的二面角小于指定的折痕角度，那么就认为这两个多边形属于同一个平滑组。这种技术有时也被称为边缘保持（edge preservation）。

这种使用折痕角度的方法有时候会得到不恰当的平滑效果，例如可能会对那些应当保持折痕的角度进行平滑，反之亦然。通常情况下，都需要进行一些实验，因为对于一个多边形网格而言，使用一个单一的角度肯定是不完美的。这种平滑组方法也有一定的局限性，一个经典的例子是：当你从中间捏着一张纸，这个薄薄的纸片可以被认为

是一个单一的平滑组，但是实际上纸张的内部是存在折痕的，这些折痕会被平滑组平滑掉。建模人员需要使用多个相互重叠的平滑组，或者是直接在网格上定义折痕边缘。另一个例子是由三角形面片所构成的圆锥体，对这个圆锥体的整个表面进行平滑会得到一个奇特的结果，即圆锥顶端有一条指向外部的法线，这个法线与圆锥体的轴线相重合。这里的圆锥顶端是一个奇点，为了能够对插值法线进行完美地表示，其中的每个三角形都需要像四边形一样，在这个顶端位置上存在两个法线[647]。

幸运的是，这种情况通常是很罕见的。我们在找到一个平滑组之后，就可以为平滑组内的共享顶点计算新的顶点法线。计算顶点法线的标准教科书做法，是对共享顶点上的多边形表面法线进行平均[541, 542]。然而，这种方法可能会导致不一致且权重较差的结果。Theurmer 和 Wuthrich [1770]提出了另一种方法，在该方法中，每个多边形法线的贡献大小会根据它在顶点处所形成的角度来进行加权。这种方法具有一个理想的特性，即无论共享这个顶点的多边形是否为三角形，都能得到相同的结果。假设此时一个细分多边形变成了两个共享顶点的三角形，如果仍然使用前面提到的平均法线方法，将会不正确地对这两个新三角形施加两倍于原始多边形的影响，如图 16.12 所示。

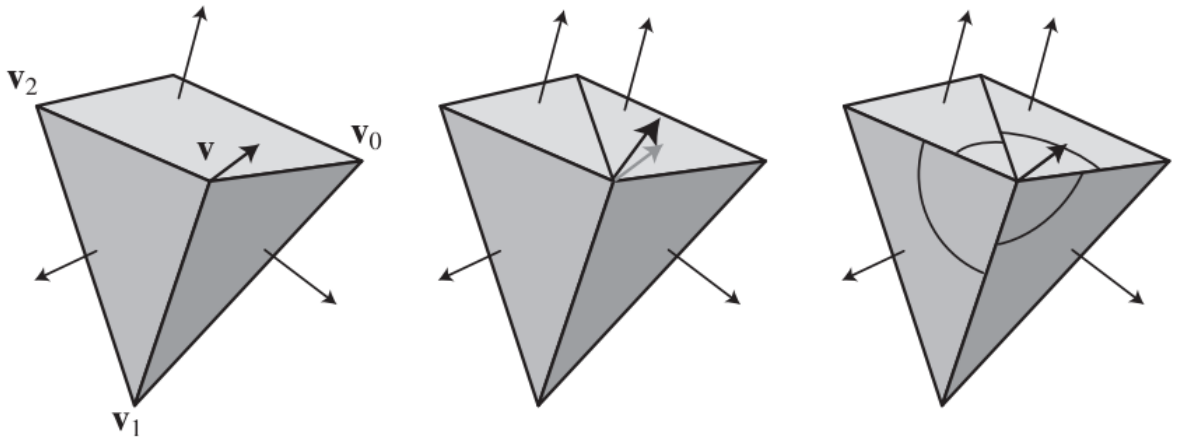


图 16.12：左侧：对一个四边形和两个三角形的表面法线取平均值，从而得到一个顶点法线。中间：这个四边形被三角形化了。由于每个多边形的法线权重是相等的，因此最终会导致这个平均法线发生偏移。右侧：使用了 Thurmer 和 Wuthrich 的方法，该方法会根据形成多边形顶角的角度，来对每个法线的贡献进行加权，这样即使多边形进行了三角形划分，最终形成的顶点法线也不会发生偏移。

Max [1146]给出了一种不同的加权方法，该方法基于了这样一个假设：长边所形成的多边形对于法线的影响较小。当使用一些简化技术的时候，这种平滑方法可能会更好，因为由简化技术所形成的较大的多边形，将不太可能会遵循表面的曲率。

Jin 等人[837]对以上这些方法和其他的方法进行了全面的调研，他得出的结论是：在各种条件下，角度加权方法要么的最好的，要么就是最好的之一。Cignoni [268]在 Meshlab 中实现了一些方法，并对它们进行了注释。他还警告说，不要使用三角形的面积来对法线的贡献进行加权。

对于高度场数据，Shankel [1614]展示了如何使用沿轴的相邻高度差，来对使用角度加权法的平滑结果进行快速近似。对于一个给定点 $\mathbf{p}$ 和四个相邻点，其中 x 轴上的高度场相邻点为 $\mathbf{p}^{x-1}$ 和 $\mathbf{p}^{x+1}$ ， y 轴上的高度场相邻点为 $\mathbf{p}^{y-1}$ 和 $\mathbf{p}^{y+1}$ ，那么点 $\mathbf{p}$ 上的法线近似值（未归一化）为：

$$\mathbf{n} = \left(p_x^{x+1} - p_x^{x-1}, p_y^{y+1} - p_y^{y-1}, 2\right) \tag{16.1}$$

16.4 三角形扇，三角形带和三角形网格

使用一个三角形列表来存储和显示三角形集合，这是最简单的一种方式，但是通常它的效率也是最低的。在这种方法中，每个三角形的顶点数据会被一个接一个地放入这个列表中。每个三角形都有自己独立的 3 个顶点，并且三角形之间也不会共享顶点数据。一种提高图形性能的标准方法是将一组共享顶点的三角形发送到图形管线中，共享顶点意味着调用顶点着色器的次数会变少，因此需要进行变换操作的顶点和法线也就会更少。在本小节中，我们将对各种共享顶点信息的数据结构进行介绍，从三角形扇（triangle fan）和三角形带（triangle strip）开始，再逐步发展到更加精细、更加高效的表面渲染形式。

16.4.1 三角形扇

图 16.13 展示了一个三角形扇（triangle fan）。这个数据结构展示了一种构建三角形的方式，并且能够使得每个三角形的存储成本少于 3 个顶点。这个被所有三角形共享的顶点称为中心顶点（center vertex），在图中对应顶点 $\mathbf{v}_0$ 。对于起始三角形 T_0 ，会依次发送顶点 $\mathbf{v}_0$ 、 $\mathbf{v}_1$ 、 $\mathbf{v}_2$ （按此顺序）。对于后续的每个三角形，这个中心顶点总是会与之前发送的顶点，以及当前正在发送的顶点一起进行使用。三角形 T_1 是通过发送顶点 $\mathbf{v}_3$ 来构建的，因此创建了一个由顶点 $\mathbf{v}_0$ （总是包含）、 $\mathbf{v}_2$ （之前发送的顶点）和 $\mathbf{v}_3$ （当前正在发送的顶点）所定义的三角形。三角形 T_2 通过发送顶点 $\mathbf{v}_4$ 来构建的，以此类推。请注意，一个一般化的凸多边形，可以很容易的表示为一个三角形扇，因为这个凸多边形的任意一个顶点都可以作为起始的中心顶点。

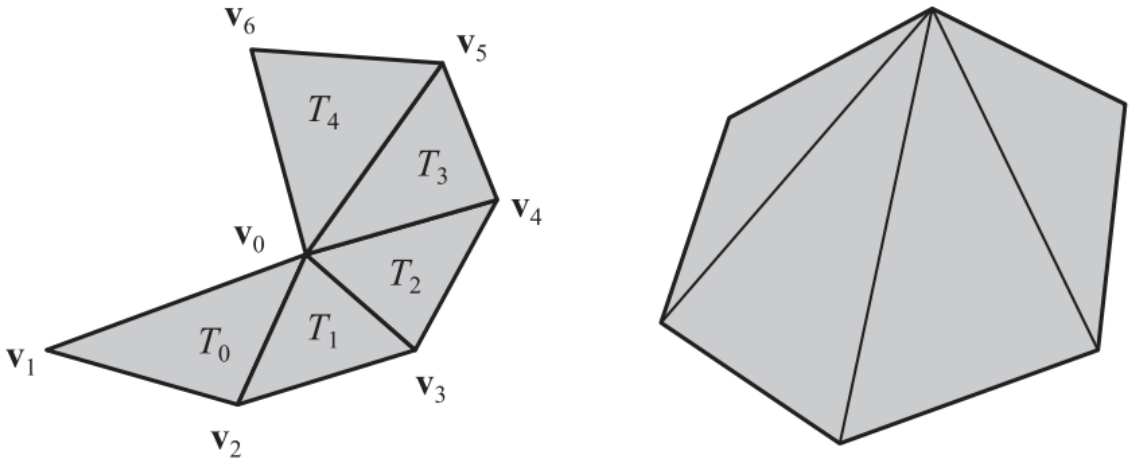


图 16.13：左侧说明了三角形扇的概念。三角形 T_0 发送顶点 $\mathbf{v}_0$ （中心顶点）、 $\mathbf{v}_1$ 和 $\mathbf{v}_2$ 。后续的三角形 $T_i (i > 0)$ ，只需要发送顶点 $\mathbf{v}_{i+2}$ 。右侧展示了一个凸多边形，它总是可以变成一个三角形扇。

包含 n 个顶点的三角形扇，可以被定义为一个有序的顶点列表：

$$\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\} \tag{16.2}$$

其中 $\mathbf{v}_0$ 是中心顶点，同时会在这个列表上添加一个结构，来表明第 i （ $0 \leq i < n-2$ ）三角形是：

$$\triangle \mathbf{v}_0 \mathbf{v}_{i+1} \mathbf{v}_{i+2} \tag{16.3}$$

如果一个三角形扇由 m 个三角形组成，那么第一个三角形将会发送三个顶点，然后剩下的 $m-1$ 个三角形各自会发送一个顶点。这意味着，长度为 m 的连续三角形扇，所发送的平均顶点数 v_a 可以表示为：

$$v_a = \frac{3 + (m-1)}{m} = 1 + \frac{2}{m} \tag{16.4}$$

从方程 16.4 中我们可以很容易看出，当 $m \to \infty$ 时， $v_a \to 1$ 。这似乎与现实世界中的情况没有太大的相关性，因为我们并不会发送一个无限大的三角形扇，我们可以考虑一个更加合理的数值。如果 $m = 5$ ，那么 $v_a = 1.4$ ，这意味着平均每个三角形只需要发送 1.4 个顶点。

16.4.1 三角形带

三角形带（triangle strip） 和三角扇比较类似，也会重复使用之前三角形中的顶点。在三角形扇中，我们会使用一个中心点与上一次发送的顶点来帮助构建当前的三角形；而在三角形带中，我们会使用前一个三角形中的两个顶点，来帮助构建当前的三角形，如图 16.14 所示。

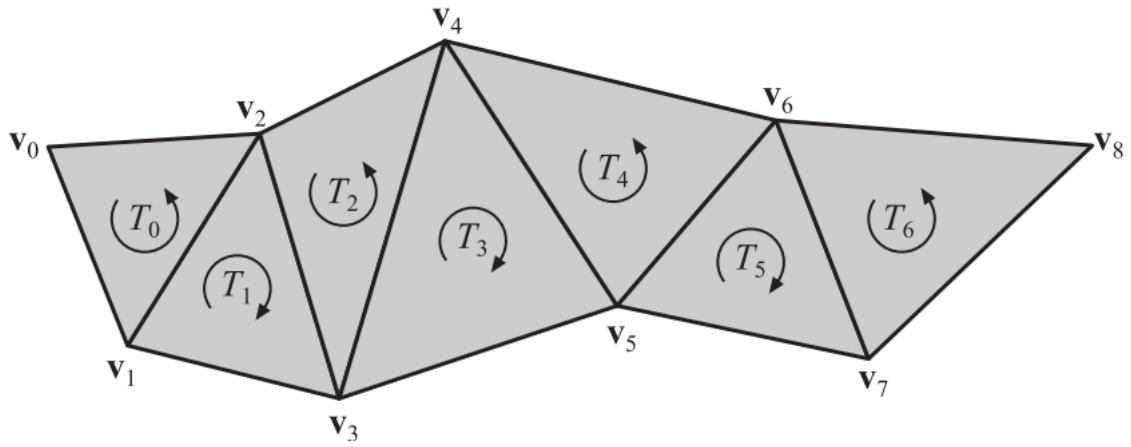


图 16.14：图中的一系列三角形可以表示为一个三角形带。请注意，条带中的每个三角形方向都是不同的，条带中的第一个三角形决定了所有三角形的方向。在每个三角形内部，通过遍历顶点 $[0,1,2]$ ， $[1,3,2]$ ， $[2,3,4]$ ， $[3,5,4]$ 来保证一致的逆时针顺序，以此类推。

如果将这些三角形视为一个条带，那么可以使用一种更加紧凑的方式来将它们发送到渲染管线中。对于第一个三角形（记为 T_0 ），它的三个顶点（记为 $\mathbf{v}_0$ 、 $\mathbf{v}_1$ 和 $\mathbf{v}_2$ ）会按照顺序进行发送。而对于这个条带中的后续三角形，只需要发送一个顶点即可，因为之前的三角形已经发送了另外两个顶点。例如：在发送三角形 T_1 的时候，只需要发送顶点 $\mathbf{v}_3$ ，并使用三角形 T_0 中的顶点 $\mathbf{v}_1$ 和 $\mathbf{v}_2$ ，来构建这个三角形 T_1 。对于三角形 T_2 ，只需要发送顶点 $\mathbf{v}_4$ ，以此类推。

一个包含 n 个顶点的连续三角形带，可以被定义为一个有序的顶点列表：

$$\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\} \tag{16.5}$$

以及一个表示第 i （ $0 \leq i < n-2$ ）个三角形的结构：

$$\triangle \mathbf{v}_i \mathbf{v}_{i+1} \mathbf{v}_{i+2} \tag{16.6}$$

之所以这种条带被称为是连续的（sequential），因为其中的顶点是按照给定的顺序进行发送的。这个定义意味着包含 n 个顶点的连续三角形带中有 $n-2$ 个三角形。

这里我们也对长度为 m 的三角形带（即由 m 个三角形组成）的平均顶点数进行一下分析，这个平均顶点数也记为 v_a ，这个 v_a 的方程与三角扇的完全相同（方程

16.4），因为二者具有相同的启动阶段，后续每个新的三角形也只会发送一个顶点。类似地，当 $m \to \infty$ 时，三角形带的 v_a 也会趋向于每个三角形只发送一个顶点。当 $m = 20$ 时，$v_a = 1.1$，这要比 3 好得多，也比较接近 1.0 的极限。与三角形扇一样，第一个三角形的启动成本（总是会发送 3 个顶点）会分摊到后续的三角形上。

三角形带的吸引力源于这样一个事实。根据渲染管线中性能瓶颈的位置不同，相比于使用简单的三角形列表，使用三角形带可以节省多达三分之二的渲染时间。这种加速是由于避免了冗余操作，例如将每个顶点发送两次到图形硬件中，然后再对每个顶点执行矩阵变换、裁剪以及其他操作。三角形带对于诸如草叶、或者其他顶点无法被重复使用的物体而言十分有用。由于这种简单性，因此当几何着色器在输出多个三角形的时候，就会使用三角形带。

三角形带还有几种变体，例如对三角形的顺序不进行强制要求；通过使用双重顶点，或者重新设置索引值，从而在单个缓冲区中存储多个不连续的三角形带。曾经有相当多的研究，关注于如何将一个任意三角形网格分解成若干个三角形带[1076]。这样的努力和研究已经失去意义了，因为带索引的三角形网格（indexed triangle mesh）的引入，实现了更好的顶点数据重用，从而导致了更快的显示速度和更少的整体内存开销（通常）。

16.4.3 三角形网格

三角形扇和三角形带仍然有其用途，但是在所有的现代 GPU 上，对于复杂模型而言，通常都是使用一个具有单一索引列表（章节 16.3.1）的三角形网格[1135]。三角形扇和三角形带允许一些数据共享，但是网格存储允许更多的数据共享。在网格中，会使用一个额外的索引数组来记录哪些顶点构成了三角形。通过这种方式，单个顶点就可以与多个三角形相关联。

连通平面图（connected planar graph）的 Euler–Poincare（欧拉–庞加莱）公式[135]可以帮助我们确定构成闭合网格的平均顶点数量：

$$v - e + f + 2g = 2 \tag{16.7}$$

其中 v 是顶点的数量，e 是边的数量，f 是面的数量，g 是 genus（字面意思是：类、种）。genus 代表了物体上的孔洞数量。例如：一个球体的 genus 为 0，一个圆环（torus）的 genus 为 1，假设每个面都有一个环。如果一个面可以有多个环的话，那么方程 16.7 可以改写为：

$$v - e + 2f - l + 2g = 2 \tag{16.8}$$

其中 l 是环的数量。

对于一个封闭（实体）模型，每条边都有两个面，每个面都至少有三条边，因此 $2e \geq 3f$ 。如果一个网格中都是三角形（就像 GPU 所要求的那样），那么 $2e = 3f$ 。假设 genus 为 0，将 $1.5f$ 替换方程中的 e ，可以得到 $f \leq 2v - 4$ 。如果网格中的所有面都是三角形，那么则有 $f = 2v - 4$ 。

对于大型封闭的三角形网格，一个经验法则是三角形的数量大约等于顶点数量的两倍。类似地，我们发现每个顶点平均连接了大约 6 个三角形（即一个顶点对应 6 条边）。与一个顶点相连的边的数量，叫做这个顶点的价（valence）。需要注意的是，网格的网状结构并不会影响结果，只有三角形的数量会对结果产生影响。由于三角形带中每个三角形的平均顶点数接近于 1，而顶点数量大约是三角形数量的两倍，因此如果使用一个三角形带来表示一个大型网格，那么每个顶点平均需要发送两次。而在极限情况下，三角形网格可以做到每个三角形只发送 0.5 个顶点。

请注意，上述这种分析过程只适用于光滑的、封闭的网格。网格中只要出现了边界边缘（boundary edge，只属于一个多边形的边），顶点与三角形的比例就会增加。Euler–Poincare 公式仍然成立，但是网格的外边界（outer boundary）必须被视为与所有外部边缘（exterior edge）都相邻的独立（未使用）面片。类似地，模型中的每个平滑组实际上都算是独立的网格，因为 GPU 在两个平滑组相交的尖锐边缘处需要单独的顶点记录，这些顶点具有不同的法线。例如：一个立方体的角上有三条不同的法线，那么这个位置上会存储三个顶点记录。纹理或者其他顶点数据的变化，也会导致顶点记录的数量增加。

根据理论预测，每个三角形大约需要处理 0.5 个顶点。在实践中，顶点在被 GPU 变换之后，会放入一个先进先出（first–in, first–out，FIFO）缓存中，或者放入一个近似于最近最少使用（least recently used，LRU）的系统中[858]。这个缓存（cache）保存了每个顶点经过顶点着色器变换后的结果。如果传入的某个顶点已经存在于这个缓存中了，那么就可以直接使用这个缓存中的变换结果，而不需要再次调用顶点着色器，这样做可以带来显著的性能提升。但是如果三角形网格中的三角形是按照随机顺序进行发送的，那么这种缓存机制可能就没什么用。三角形带算法的优化缓存大小为两个，即使用最后的两个顶点。Deering 和 Nelson [340]首先探索了将顶点数据存储在更大的 FIFO 缓存中的想法，并通过使用一种算法来确定将顶点添加到缓存中的顺序。

FIFO 缓存的大小是有限的。例如：PS3（PLAYSTATION 3）系统中的 FIFO 缓存拥有大约 24 个顶点，这也取决于每个顶点所占据的字节数。在较新的 GPU 上，这个

FIFO 缓存的大小也没有显著增加，32 个顶点是一个经典的最大值。

Hoppe [771]引入了一个用于衡量缓存重用的重要指标，即平均缓存未命中率（average cache miss ratio，ACMR）。这是每个三角形需要处理的平均顶点数量。ACMR 的范围可以从 3（三角形的每个顶点，每次都必须进行重新处理）到 0.5（一个大型封闭网格被完美重用；没有任何顶点会被重新处理）。如果缓存大小与网格本身一样大时，那么 ACMR 就等于顶点与三角形的理论比值。对于给定的缓存大小和网格排序方式，可以精确计算 ACMR 的值，因此可以描述任何给定方法在该缓存大小下的效率。

16.4.4 缓存无关的网格布局

网格中三角形的理想顺序是能够最大化地利用顶点缓存。Hoppe [771]提出了一种可以最小化网格 ACMR 的算法，但是必须事先知道缓存的大小。如果算法假设的缓存大小要大于实际的缓存大小，那么使用该算法生成的网格所带来的性能优势会明显减少。为不同大小的缓存进行求解，可能会产生不同的最佳排序方式。当目标缓存的大小未知时，使用缓存无关（cache-oblivious）的网格布局算法可以生成运行良好的排序方式，无论鸡腿的缓存大小是多少。这种排序方式有时候也被称为通用索引序列（universal index sequence）。

Forsyth [485]、Lin 和 Yu [1047]使用类似的原理，提供了一些快速贪婪算法。根据顶点在缓存中的位置，以及附着在它们上面的未处理三角形数量来给它们进行打分，接下来会处理顶点得分最高的那个三角形。该算法通过将最近使用过的三个顶点的得分稍微降低一点，从而避免创建三角形带，而是会创建一个类似于希尔伯特曲线（Hilbert curve）的图案。对于那些连接较少数量三角形的顶点，通过赋予它们更高的分数，该算法会倾向于避免留下孤立的三角形。这个算法所能够达到的平均缓存未命中率，与那些更加复杂且成本更高的算法相当。Lin 和 Yu 的方法要稍微复杂一些，但是使用了类似的思想。当缓存大小为 12 时，30 个未优化模型的平均 ACMR 为 1.522；在经过算法优化之后，根据缓存大小的不同，ACMR 平均会下降到 0.664 或者更低。

Sander 等人[1544]对之前的工作进行了概述，并提出了他们自己的方法，这种方法的速度更快（尽管该方法并不是缓存大小无关的），被称为 Tipsify。此外，他们还尽量会将最外层的三角形放在列表的前面，从而最大限度地减少过度绘制（章节 18.4.5）。例如；想象现在有一个咖啡杯，首先渲染那些构成杯子外部的三角形，那么后续的内部三角形很可能就会被遮挡，不会出现在视野中。

Storsjo [1708]对 Forsyth 和 Sander 的方法进行了对比分析，并提供了二者方法的具体实现。他的结论是：这些方法所提供的布局已经接近了理论的极限。在 Kapoulkine [858]的一项最新研究中，在 3 家硬件供应商的 GPU 上，对 4 种缓存已知（cache-aware）的顶点排序算法进行了对比。他的结论是：Intel 使用了一个包含 128 个 entry 的 FIFO 缓存，每个顶点使用三个或者更多个 entry；而 AMD 和 NVIDIA 的系统则近似于 16 个 entry 的 LRU 缓存。这种架构上的差异会显著影响算法的性能表现。他发现 Tipsify 算法[1544]和 Forsyth 的算法[485]（稍微差一些）在这些平台上的表现相对较好。

综上所述，对三角形进行离线预处理能够显著提高三角形网格的顶点缓存性能；并且如果顶点处理阶段是整个管线中的性能瓶颈时，这种预处理可以提高整体的帧率。这种预处理方法的执行速度快，实际的算法复杂度为 $O(n)$，还有一些可用的开源版本[485]。考虑到这些算法能够自动应用于网格，并且这种优化并不会带来额外的存储成本，也不会对工具链中的其他工具产生影响，因此这些方法通常都是成熟开发系统中的一部分。例如：Forsyth 的算法似乎是 PLAYSTATION 网格处理工具链中的一部分。虽然现代 GPU 都采用了统一的着色器架构，顶点变换后的缓存问题已经得到了发展，但是避免缓存未命中仍然是一个十分重要的问题[530]。

16.4.5 顶点和索引缓冲区/数组

一种方法为现代图形加速器提供模型数据的方法，是使用一些特殊的存储对象，在 DirectX 中被称为顶点缓冲区（vertex buffer），在 OpenGL 中被称为顶点缓冲区对象（vertex buffer objects，VBO）。本小节将会使用 DirectX 中的术语，这些概念在 OpenGL 中也有相对应的概念。

顶点缓冲区的思想是将模型数据存储在一个连续的内存块中。顶点缓冲区是一个特定格式的顶点数组，这个格式指定了存储在内部的顶点是否包含法线、纹理坐标、颜色或者其他的特殊信息。每个顶点都会将自身的数据存储在一个组中，一个顶点接着一个顶点地进行存储。顶点数据的大小（以字节为单位）被称为该顶点的步幅（stride）。这种类型的存储方式被称为交错缓冲区（interleaved buffer）。或者也可以使用一组顶点流（vertex stream）。例如：使用一个流来保存位置数组 $\{\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\ldots\}$，使用另一个流来保存法线数组 $\{\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2\ldots\}$。在实践中，一个包含了顶点所有数据的单个缓冲区对于 GPU 而言会更加高效，但是这种高效也没有高到要避免使用多个流[66, 1494]。多个流的主要开销在于额外的 API 调用，如果应用程序的 CPU 资源很紧张（译者注：即常说的吃 CPU），但是在其他方面的开销并不是很大，那么可以避免这种使用多个流所带来的额外开销[443]。

Wihlidal [1884]讨论了使用多个流来帮助渲染系统提升性能的不同方式，包括 API、缓存和 CPU 处理等。例如：在 CPU 上进行向量处理的 SSE 和 AVX 会更加容易地应用在单独的流上。使用多个流的另一个原因在于，可以对网格数据进行更加高效地更新。例如：如果一个网格中只有顶点的位置流会随着时间发生变化，那么单独更新这一个属性缓冲区的开销，要比构建并发送整个交错流的开销小得多[1609]。

具体要如何访问顶点缓冲区，取决于设备的 $\mathrm{DrawPrimitive}$ 方法。这些数据可以被处理为：

1. 一个包含独立顶点的列表。
2. 一个包含未连接线段的列表，即顶点对。
3. 一条简单的折线。
4. 一个三角形列表，其中每三个顶点构成一个三角形，例如：顶点 $[0, 1, 2]$ 构成一个三角形，顶点 $[3, 4, 5]$ 构成下一个三角形，以此类推。
5. 一个三角形扇，其中第一个顶点与每个连续的顶点对都会构成一个三角形，例如： $[0, 1, 2]$ ， $[0, 2, 3]$ ， $[0, 3, 4]$ 分别构成了一个三角形。
6. 一个三角形带，其中三个相邻的顶点构成一个三角形，例如： $[0, 1, 2]$ ， $[1, 2, 3]$ ， $[2, 3, 4]$ 分别构成了一个三角形。

在 DirectX 10 中，三角形列表和三角形带也可以包含相邻的三角形顶点，用于提供给几何着色器（章节 3.7）进行使用。

顶点缓冲区可以直接进行使用，也可以被索引缓冲区引用。索引缓冲区中的索引保存了顶点缓冲区中顶点的位置。这个索引是一个 16 bit 的无符号整数（uint16）；如果模型网格很大，并且 GPU 和 API（章节 16.6）也支持的话，还可以使用 32 bit 的无符号整数（uint32）。索引缓冲区和顶点缓冲区组合起来，所显示的绘制图元与“原始”的顶点缓冲区完全相同。两种方式的不同之处在于，在索引/顶点缓冲区的组合中，每个顶点只需要在顶点缓冲区中存储一次即可；而对于没有索引的顶点缓冲区，所存储的顶点数据可以发生重复。

三角形网格的结构由索引缓冲区来进行表达。存储在索引缓冲区中的前 3 个索引指定了第一个三角形，接下来的 3 个索引指定了第二个三角形，以此类推。这种数据组织形式被称为带索引的三角形列表（indexed triangle list），实际上这里的索引本身就构成了一个三角形列表。OpenGL 将索引缓冲区、顶点缓冲区与顶点格式信息绑定在一个顶点数组对象（VAO）中。这里的索引也可以按三角形带的顺序进行排列，这样可以节省索引缓冲区所占据的空间。这种带索引的三角形带格式在实践中很少会用

到，因为想要为一个大型网格创建这样的三角形带集合，需要进行一些额外的努力，并且也要求处理几何图形的工具也需要支持这种格式。图 16.15 展示了顶点缓冲区和索引缓冲区的结构。

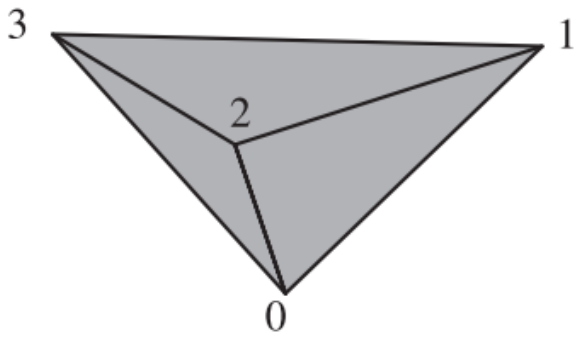


图 16.15：定义图元的不同方式，按照从上到下（内存开销从最多到最少）的粗略顺序分别是：单独的三角形；一个顶点三角形列表；使用一个或者两个数据流的三角形带；以独立三角形或者按照三角形带列出的索引缓冲区。

（由于图 16.15 太大，并且包含大量解说性文字，因此从这里开始进行描述）

图中展示了三个三角形，分别由顶点 $\mathbf{p}_0$ 到 $\mathbf{p}_1$ 、法线 $\mathbf{n}_0$ 到 $\mathbf{n}_1$ 所构成。可以通过一系列独立的调用来渲染出这些三角形，分别是：

- $\mathbf{p}_0$ ， $\mathbf{n}_0$ ， $\mathbf{p}_1$ ， $\mathbf{n}_1$ ， $\mathbf{p}_2$ ， $\mathbf{n}_2$
- $\mathbf{p}_1$ ， $\mathbf{n}_1$ ， $\mathbf{p}_3$ ， $\mathbf{n}_3$ ， $\mathbf{p}_2$ ， $\mathbf{n}_2$
- $\mathbf{p}_2$ ， $\mathbf{n}_2$ ， $\mathbf{p}_3$ ， $\mathbf{n}_3$ ， $\mathbf{p}_0$ ， $\mathbf{n}_0$

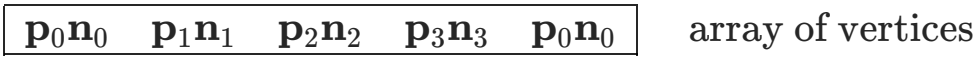
可以将位置信息和法线信息分别放入两个独立的列表中，这两个列表包含了三角形的组织顺序，每三个一组（不重复使用）会被作为一个三角形：

$\mathbf{p}_0$	$\mathbf{p}_1$	$\mathbf{p}_2$	$\mathbf{p}_1$	$\mathbf{p}_3$	$\mathbf{p}_2$	$\mathbf{p}_2$	$\mathbf{p}_3$	$\mathbf{p}_0$	array of positions
$\mathbf{n}_0$	$\mathbf{n}_1$	$\mathbf{n}_2$	$\mathbf{n}_1$	$\mathbf{n}_3$	$\mathbf{n}_2$	$\mathbf{n}_2$	$\mathbf{n}_3$	$\mathbf{n}_0$	array of normals

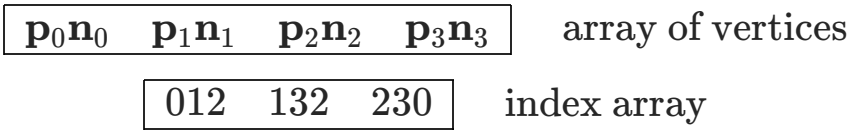
将位置信息和法线信息分别放入两个独立的列表中，每三个一组（重复使用）会被作为一个三角形：

$\mathbf{p}_0$	$\mathbf{p}_1$	$\mathbf{p}_2$	$\mathbf{p}_3$	$\mathbf{p}_0$	array of positions
$\mathbf{n}_0$	$\mathbf{n}_1$	$\mathbf{n}_2$	$\mathbf{n}_3$	$\mathbf{n}_0$	array of normals

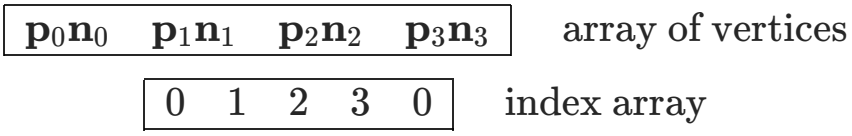
将顶点信息放入一个交错数组（同一个顶点的属性会被放在一起）中，下面是一个表示三角形带的数组：



将顶点信息放入一个交错数组中，并使用一个额外的索引列表来给出三角形的组织关系：



将顶点信息放入一个交错数组中，并使用一个额外的索引列表来表示一个三角形带：



（图 16.15 到此结束）

具体提使用哪种结构由图元和程序所决定。想要显示一个矩形很容易，只需要一个顶点缓冲区，使用四个顶点来表示两个三角形所组成的三角形带或者三角形扇即可。如前所述，使用索引缓冲区的一个优点在于数据可以进行共享。它的另外一个优点就是简单，因为三角形可以有任意的顺序和配置方式，并且没有三角形带的锁步需求（lock-step，指相邻三角形顶点顺序翻转）。最后，在使用索引缓冲区的时候，需要在 GPU 上传输和存储的数据量要更小。通过共享顶点所节省的内存，远远超过了多存储一个索引数组所带来的额外开销。

索引缓冲区和一个或者多个顶点缓冲区提供了一种描述多边形网格的方法。然而，数据存储的目的通常是尽可能提高 GPU 的渲染效率，而不一定是最紧凑的存储方式。例如：想要存储一个立方体，一种方法是将它的 8 个角位置存储在一个数组中，再将 6 个不同的法线方向存储在另一个数组中，以及 6 个定义立方体面的环，每个环代表一个正方形，每个正方形由 4 个索引组成。每个顶点位置由两个索引进行描述，一个用于顶点列表，另一个用于法线列表。纹理坐标则由另一个数组和第三个索引进行表示。这种紧凑的表示方法被用于许多模型文件格式中，例如 Wavefront OBJ。而在 GPU 上，由于只能使用一个索引缓冲区，因此我们要在一个顶点缓冲区中存储 24 个不同的顶点，因为每个角点位置都有三个不同的法线，其中每个法线各自对应着一个相邻的面。而对应的索引缓冲区中将存储构成表面的 12 个三角形索引。

Masserann [1135]讨论了如何高效地将这类文件描述，转换为紧凑而高效的索引/顶点缓冲区，而不是转换为没有共享顶点的无索引三角形列表。还有一些更加紧凑的方案，例如将网格存储在纹理贴图或者缓冲区纹理中，并使用顶点着色器的纹理拾取（texture fetch）或者拉取机制（pulling mechanism）来进行实现，但是这些方法无法使用变换后的顶点缓存，这会带来一些性能损失[223, 1457]。

为了获得最大效率，顶点缓冲区中的顶点顺序应当与索引缓冲区中的顶点访问顺序相匹配。也就是说，索引缓冲区中第一个三角形所引用的 3 个顶点，应当刚好是顶点缓冲区中的前 3 个顶点。当索引缓冲区遇到一个新顶点的时候，它应当刚好是顶点缓冲区中的下一个顶点。使用这样的排列顺序，可以最小化变换前顶点缓存（pre-transform vertex cache）的缓存未命中（cache miss）；我们在章节 16.4.4 中讨论过，变换后的顶点缓存和变换前的顶点缓存是分开的。对顶点缓存中的数据进行重新排序是一个十分简单的操作，但是对于性能来说，它与在变换后的顶点缓存中找到一个有效的三角形顺序一样重要[485]。

还有一些更高层次（higher-level）的方法可以对顶点缓冲区和索引缓冲区进行分配和使用，从而进一步提高效率。例如：可以在 GPU 上存储一个不变的缓冲区，以供每一帧中进行使用；并且可以从同一个缓冲区中生成相同物体的多个实例和变体。我们将在章节 18.4.2 中深入讨论此类技术。

使用管线的流式输出（stream output）功能（章节 3.7.1），我们可以将处理过的顶点发送到一个新的缓冲区中，这允许在 GPU 上对顶点缓冲区进行处理，但是并不实际渲染它们。例如：一个描述三角形网格的顶点缓冲区可以被视为初始 pass 中的简单点集，顶点着色器可以按照需要执行逐顶点的计算，并使用流式输出来将结果发送到一个新的顶点缓冲区中。在后续的 pass 中，这个新的顶点缓冲区可以与描述网格连通性的原始索引缓冲区进行配对，从而对生成的网格进行进一步的处理和显示。

16.5 简化

网格简化（mesh simplification），也称为数据简化（data reduction）或者数据抽取（data decimation），是指将一个详细的多边形模型，降低它的三角形面数，同时尽量保持其原有外观的过程。对于实时渲染而言，这个过程是为了减少需要存储和发送到管线中的顶点数量。这对于提高应用程序的可伸缩性（scalable）十分重要，因为一些性能较弱的设备只能显示较少数量的三角形。另外，我们所接收的模型数据可能也会包含一些不合理、不必要的曲面细分。图 16.16 展示了如何通过数据简化技术，来降低存储的三角形数量。

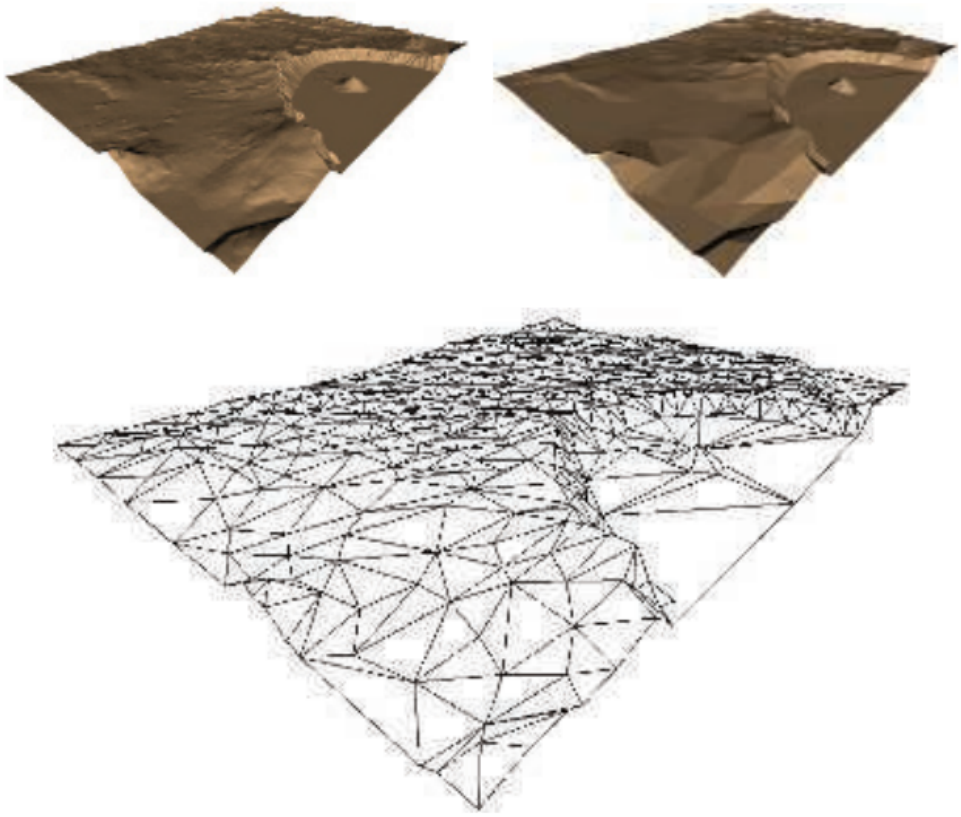


图 16.16：左上角是一个火山口湖的高度场，它由 20 万个三角形渲染而成。右上角的图展示了该模型在不规则三角网（triangulated irregular network，TIN）中简化为 1000 个三角形。底部展示了简化之后的网格。

Luebke [1091, 1092]定义了三种类型的网格简化：静态（static）、动态（dynamic）和视图相关（view-dependent）。静态简化的思想是：在渲染开始之前创建单独的细节层次（LOD）模型，并在渲染的过程中动态选择具体要进行使用的模型，我们将在章节 19.9 中进行详细介绍。离线的简化对于其他任务也很有用，例如为细分表面提供较为粗糙的网格来进行细化[1006, 1007]。动态简化提供了 LOD 模型的连续版本，而不是几个离散的模型，因此这种方法也被称为连续细节层次（continuous level of detail，CLOD）算法。视图相关的技术适用于模型中细节层次不同的地方。具体来说，地形渲染是一种常见情况，在地形渲染中，附近的区域需要进行详细的表示，而远处的区域则处于较低的细节水平。本小节将讨论后面两种简化技术，即动态简化和视图相关的简化。

16.5.1 动态简化

减少三角形数量的一种方法是使用一个边坍缩（edge collapse）操作，它通过移动一条边上的两个顶点，使其重合来移除这条边。图 16.17 展示了这个操作的一个例子。对于实体模型而言，一次边坍缩总共会移除两个三角形、三个边和一个顶点。因

此，一个具有 3000 个三角形的封闭模型，需要使用 1500 次边坍缩，才能将面数减少到 0。之前我们提到了一个经验法则，即一个包含 v 个顶点的封闭三角形网格，大约会有 $2v$ 个面和 $3v$ 条边。这个法则可以使用 Euler–Poincare 公式（章节 16.4.3）推导出来，即固体表面的 $f - e + v = 2$ 。

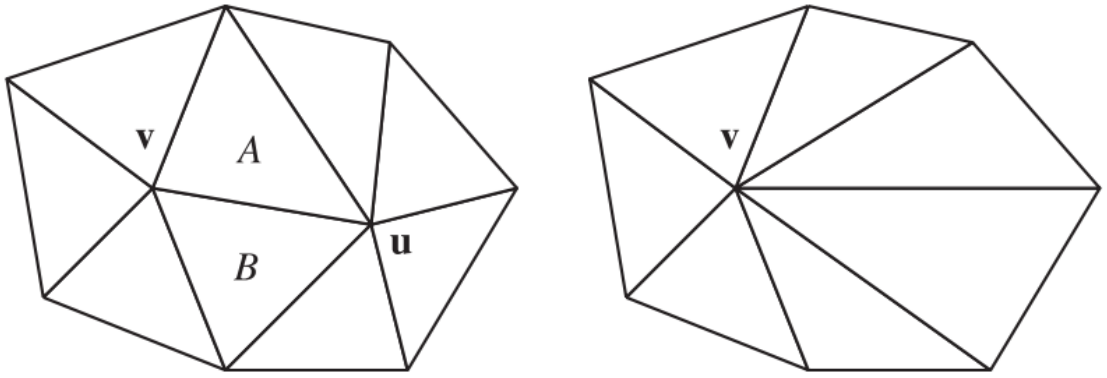


图 16.17：左边是 $\mathbf{uv}$ 边坍缩之前的样子，右图展示了点 $\mathbf{u}$ 坍缩成点 $\mathbf{v}$ 的情况，这个操作移除了三角形 A 、三角形 B 和边 $\mathbf{uv}$ 。

边坍缩的过程是可逆的。可以按照顺序来存储边坍缩的具体过程，这使得我们可以从一个简化模型出发，并从中重建出一个复杂模型。这个特性对于模型的网络传输而言十分有用，因为数据库中存储的是模型的边坍缩版本，它可以通过这种高效压缩的形式进行发送，并在接收模型的时候，逐步重建出复杂模型并进行显示[768, 1751]。由于这个特点，这种简化过程通常也被称为视图无关的渐进式网格划分（view–independent progressive meshing，VIPM）。

在图 16.17 中，顶点 $\mathbf{u}$ 被坍缩到了顶点 $\mathbf{v}$ 的位置上，但是顶点 $\mathbf{v}$ 同样也可以被坍缩到顶点 $\mathbf{u}$ 的位置上。简化系统中会使用一个子集放置策略（subset placement strategy）来对这两种可能性进行限制。这种策略的一个优点在于，如果我们限制了可能性，那么我们可以对所做的选择进行隐式编码[516, 768]。这种策略的效率更快，因为我们需要进行评估的可能性变少了，但是这种策略也可能会产生质量较低的近似结果，因为我们所检查的解空间也变得更小了。

当使用一个最优放置策略（optimal placement strategy）的时候，我们可以检查一个更大范围内的可能性。在这个最优放置策略中，我们不是将一个顶点坍缩成另一个顶点，而是将一条边上的两个顶点坍缩到一个新位置上。Hoppe [768]研究了点 $\mathbf{u}$ 和点 $\mathbf{v}$ 都移动到边上某个新位置时的情况。他指出：为了对最终数据表示的压缩进行改进，可以将搜索局限于只对中点进行检查。Garland 和 Heckbert [516]更进一步，通过求解一个二次方程来找到最优坍缩位置，这个最优坍缩位置可能会位于边缘以外的

地方。最优放置策略的优点在于，它可以提供更高质量的网格。它的缺点是需要额外的处理、代码和内存来记录更大范围的可能位置。

为了确定最佳的坍缩点，我们可以对局部邻域进行分析。这种局部性是一个重要且有用的特性，原因有以下几个。如果边坍缩操作的代价只取决于几个局部变量（例如：边的长度、边附近的表面法线），那么这个代价函数是很容易进行计算的，并且每次坍缩操作只会对它的几个邻居顶点产生影响。例如：假设一个模型在开始计算的时候，有 3000 种可能的边塌陷选择。此时我们执行成本函数值最小的那个边坍缩操作，因为它只会对附近几个三角形及其边缘产生影响，因此只有那些成本函数会受到影响的边坍缩选择才需要重新进行计算（例如：需要对 10 个成本函数进行重新计算，而不是 3000 个），并且这个列表只需要进行少量的重新排序即可。由于一次边坍缩操作只会影响少数几个边坍缩的代价值，因此可以使用堆或者其他优先级队列来维护这个代价值列表[1649]。

有些时候，无论一次边坍缩的成本有多低，都必须避免进行这样的边坍缩操作，如图 16.18 所示。这种现象可以通过对相邻三角形进行检查，看看它们是否会由于坍缩操作而发送法线翻转现象。

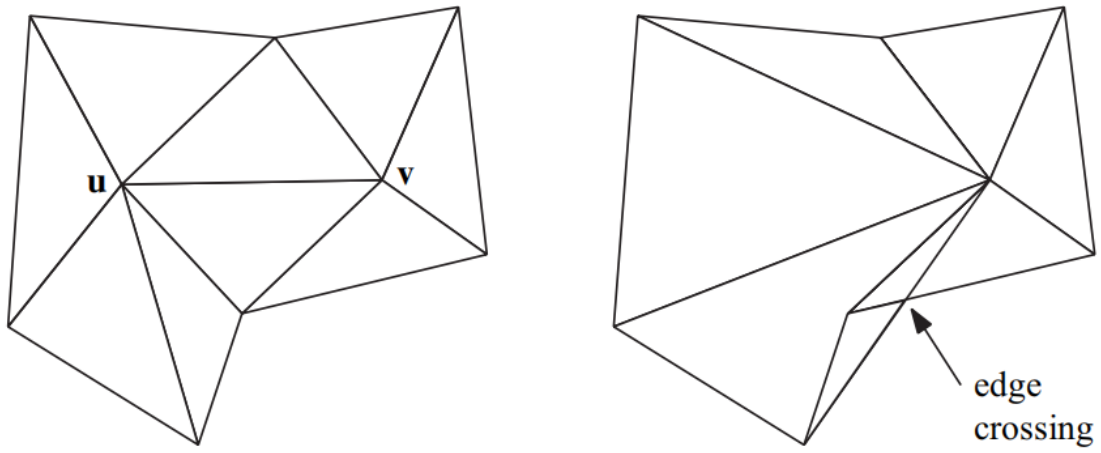


图 16.18：一次糟糕的边坍缩。左边是点 $\mathbf{u}$ 坍缩成点 $\mathbf{v}$ 前的网格。右边是坍缩之后的网格情况，会出现边缘交叉的现象。

这个坍缩操作本身是一种对模型数据库的编辑，用于存储这些坍缩操作的数据结构可以有很好的文档记录[481, 770, 1196, 1726]。在每次执行边坍缩操作之前，都需要使用代价函数来进行分析，然后执行代价值最小的那个边坍缩操作。这个最佳成本函数会随着模型类型和其他一些因素而发生变化。根据要解决的问题，这个成本函数可能会在速度、质量、健壮性和简单性之间做出权衡[1092]。它也可以进行按需定制

（tailor），以保持表面边界、材质位置、光照效果、沿轴对称、纹理放置、体积或者其他的一些约束条件。

为了让读者能够了解这些成本函数是如何运作的，这里我们将介绍 Garland 和 Heckbert 的二次误差度量（quadric error metric，QEM）成本函数[515, 516]。这个成本函数在很多情况下都是通用的。相比之下，在一些早期的研究中，Garland 和 Heckbert [514]发现，地形的简化最适合使用 Hausdorff 距离，其他人也证实了这一点[1496]。这个函数实际上就是简化网格中的一个顶点到原始网格的最长距离。图 16.16 展示了使用这个指标的结果。

对于一个给定的顶点，有一组共享该顶点的三角形，每个三角形都有一个平面方程。移动这个顶点的 QEM 代价函数是这些平面与新位置之间距离的平方和。使用数学形式进行描述，即：

$$c(\mathbf{v}) = \sum_{i=1}^{m} (\mathbf{n}_i \cdot \mathbf{v} + d_i)^2$$

这就是新位置 $\mathbf{v}$ 和 m 个平面的代价函数，其中 $\mathbf{n}_i$ 是第 i 个平面的法线， d_i 是第 i 个平面到原始位置的距离。

图 16.19 展示了同一条边的两种可能坍缩方式。假设这个立方体宽两个单位。将点 $\mathbf{e}$ 坍缩成点 $\mathbf{c}$ （ $\mathbf{e} \rightarrow \mathbf{c}$ ）的代价函数将为 0，因为点 $\mathbf{e}$ 在移动到点 $\mathbf{c}$ 时，并不会离开它所共享的平面。 $\mathbf{c} \rightarrow \mathbf{e}$ 的代价函数将为 1，因为点 $\mathbf{c}$ 从立方体的右侧平面，移动了 1 个单位的平方距离。这里我们会执行代价较低的边坍缩操作，因此 $\mathbf{e} \rightarrow \mathbf{c}$ 的坍缩要比 $\mathbf{c} \rightarrow \mathbf{e}$ 更好。

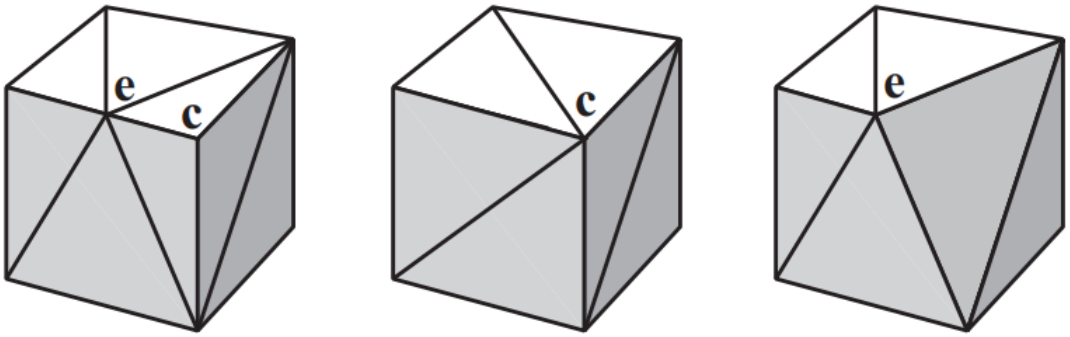


图 16.19：左侧：展示了一个立方体，其中一条边上存在一个额外的顶点 $\mathbf{e}$ 。中间：展示了点 $\mathbf{e}$ 坍缩到点 $\mathbf{c}$ 时会发生什么。右边：展示了点 $\mathbf{c}$ 坍缩到点 $\mathbf{e}$ 时会发生什么。

这个成本函数可以通过各种方式进行修改。想象有两个三角形共用同一条边，并且形成了一个锋利的边缘，他们可能是一个鱼鳍或者一个涡轮叶片。在这条边上坍缩顶点的代价函数很低，因为沿着一个三角形滑动的点并不会远离另一个三角形所在的平面。一个基本的成本函数，其数值与所删除特征的体积变化有关，但是这种体积变化并不能很好地反映它的视觉重要性。想要保留一条具有尖锐折痕的边缘，其中一种方法是添加一个额外的平面，该平面包含这条边并具有一个法线，这个法线相当于两个三角形法线的平均值。此时，远离这条边的顶点，将会具有更高的代价函数[517]。另一种变体方法是，通过三角形面积的变化来对成本函数进行加权。

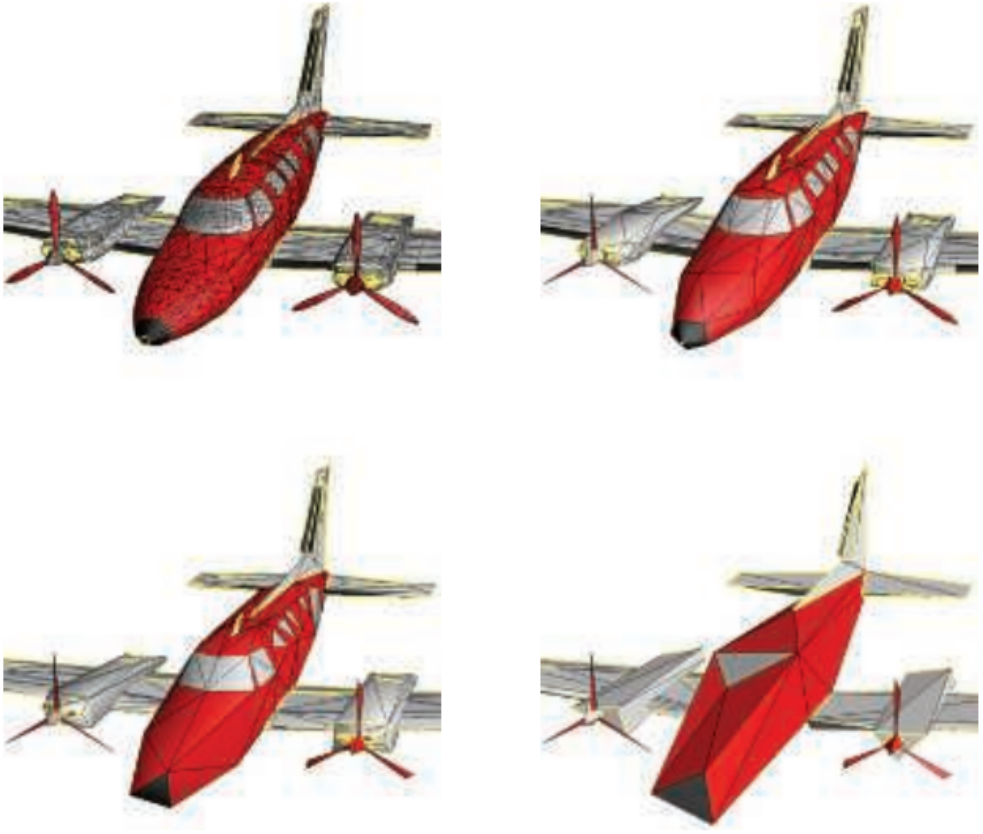


图 16.20：网格简化。左上角是包含 13546 个面的原始网格，右上角被简化为 1000 个面，左下角为 500 个面，右下角为 150 个面[770]。

还有一些其他的扩展类型，使用了一些维护其他表面特征的代价函数。例如：模型的折痕边缘和边界边缘在描绘模型外观的时候十分重要，我们需要尽量保留这些边缘特征，使得这些边缘不太可能被修改，如图 16.20 所示。其他值得保留的表面特征，包含存在材质变化、纹理贴图边缘、以及逐顶点颜色变化的位置[772]，如图 16.21 所示。

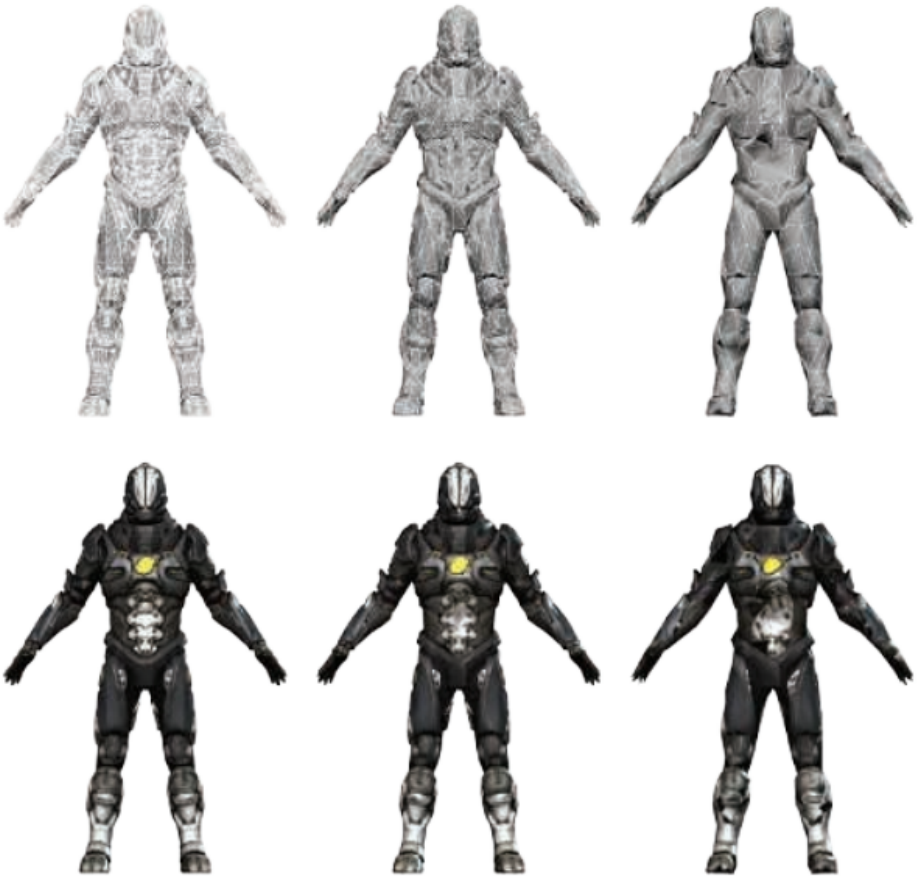


图 16.21：网格简化。第一行：网格和简单的灰色材质。第二行：使用了纹理。每一行从左到右，模型分别包含 51123、6389 和 1596 个三角形。模型上的纹理被尽可能地保持不变，但是随着三角形数量的减少，还是会发生一些失真扭曲。

大多数简化算法都会出现的一个严重的问题，即纹理经常会明显偏离其原始外观[1092]。当边缘发生坍缩的时候，纹理到表面的底层映射会发生扭曲。同理，纹理坐标的值可以在边界处进行匹配，但是会分属于应用纹理的不同区域，例如：沿着镜像模型的中心边缘。Caillaud 等人[220]对之前的各种方法进行了调研，并提出了他们自己用于处理纹理接缝的算法。

速度可能是另一个我们所关注的问题。在那些用户自己创建内容的系统中，例如在一个 CAD 系统中，可能会需要动态创建 LOD 模型。使用 GPU 来执行网格简化目前已经取得了一些成功进展[1008]。另一种方法则是使用一些更加简单的简化算法，例如顶点聚类（vertex clustering）[1088, 1511]。该方法的核心思想是：使用三维的体素网格或者类似结构来覆盖整个模型，体素中的任何顶点都会被移动到该单元格内的“最佳”顶点位置。这样做可能会移除一些三角形，当三角形的两个顶点或者三个顶点位于同一个体素内时，这个三角形会发生退化。这个算法健壮性很好，因为它不需要网格的连通性，并且可以很容易地将多个分离网格合并为一个网格。然而，这些基本的顶点聚类算法，很少能够达到完整 QEM 算法的效果。Willmott [1890]讨论了他

的团队是如何将这种聚类方法，以一种健壮且高效的方式应用在游戏《孢子》的用户内容创建模块中的。

还有一种与简化类似的思想，是将表面的原始几何形状转换为一个用于凹凸映射（bump mapping）的法线贴图。对于一些比较小的特征（例如按钮或者皱纹），当然可以使用纹理进行表示，而且几乎不会损失保真度。Sander 等人[1540]对之前这一领域的工作进行了讨论，并提供了一个解决方案。这种算法通常用于为交互式应用程序的生成模型，并将一个高质量的模型信息烘焙到一个纹理化的表示中[59]。

使用简化技术能够从一个复杂模型中生成大量的细节层次（LOD）模型。在使用 LOD 模型时会遇到一个问题，如果一个模型在两帧之间进行突然切换，有时候玩家可以看到这个切换过程 [508]，这个问题被称为“popping（突然跳出）”。一种解决方案是使用地貌（geomorphs）[768]来增加或者减少 LOD。由于我们事前知道这个复杂模型中的顶点是如何映射到简单模型中的，因此我们可以创建平滑的过渡过程。更多细节详见章节 19.9.1。

使用视图无关的渐进式网格划分（view-independent progressive meshing）具有一个优点，即我们可以只创建一次顶点缓冲区，并在不同 LOD 的相同模型副本之间进行共享[1726]。然而，在基本方案下，我们还需要为每个模型副本都创建一个单独的索引缓冲区。另一个问题是效率，由于顶点坍缩的顺序决定了三角形的显示顺序，因此顶点缓存的一致性比较差。Forsyth [481]讨论了在构建和共享索引缓冲区时，几种可以提高效率的实用方法。

网格简化的技术是十分有用的，但是完全自动化的简化系统有时候也不是万能的（panacea）。图 16.22 展示了保持对称性的问题。一个出色的模型师可以创造出比自动化程序质量更好的、低三角形数量（low-triangle-count）的模型。例如：人的眼睛和嘴巴是脸部最重要的部分，而一个朴素（naive）的简化算法可能会将这些重要特征当作无关紧要的东西平滑掉。重拓扑（retopology）指的是在建模、平滑或者使用简化技术的时候，向模型中添加边缘以保持各种特征分离的过程。与网格简化相关的算法仍在继续研究和开发中，并尽可能地尝试自动化。

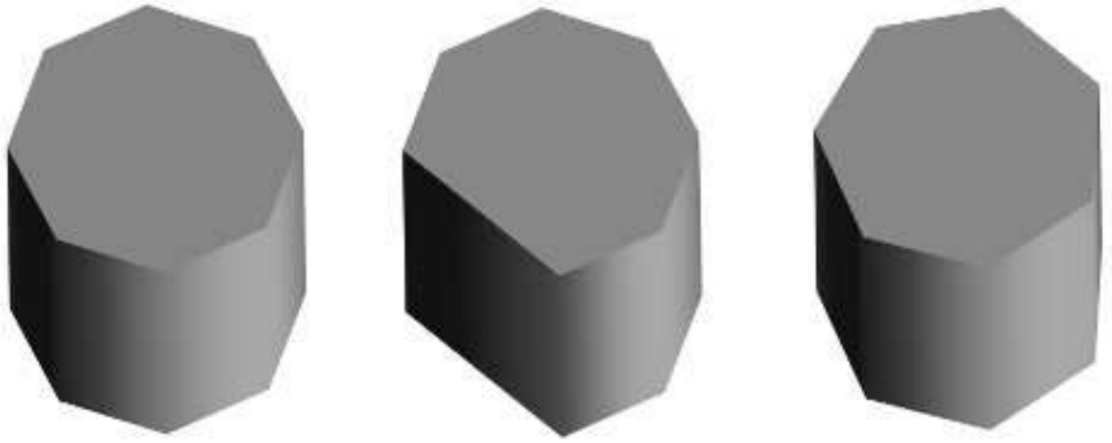


图 16.22：对称问题。左侧的圆柱体包含 10 个平面（包括顶部和底部）。中间的圆柱体包含 9 个平面，其中有 1 个平面被自动简化移除了。右侧的圆柱体仍包含 9 个平面，并由建模软件的 faceter 重新生成。

16.6 压缩和精度

三角形网格数据可以通过各种方式进行压缩，并且能够获得相似的好处。正如 PNG 和 JPEG 等图像文件格式会对纹理进行无损压缩和有损压缩一样，针对三角形网格数据的压缩目前已经开发出了各种压缩算法和文件格式。

压缩的代价是更多的编码时间和解码时间，压缩的目标是最大限度地降低数据存储空间。传输一种更小的数据表示形式，其所节省的时间必须要超过解压数据所花费的额外时间。在互联网上进行模型文件传输，较慢的下载速度意味着我们可以使用一些更加复杂的算法。MPEG–4 中所采用的 TFAN 算法[1116]可以对网格连通性（mesh connectivity）进行压缩和高效解码。与仅使用 gzip 压缩相比，Open3DGC、OpenCTM 和 Draco 等编码器所创建的模型文件，其大小仅为 gzip 压缩的四分之一或者更小[1335]。使用这些方案的解压都是一次性操作，其速度相对较慢（大约每秒几百万个三角形），但是可以大大节省传输数据所花费的时间。Maglo 等人[1099]对相关算法进行了全面的回顾。而这里我们所关注的是直接涉及 GPU 本身的压缩技术。

本小节主要讨论的是各种最小化三角形网格存储空间的方法，这样做的主要目的就是为了进一步提高渲染效率。对多个三角形中的顶点数据进行重用（而不是重复）（reuse vs repeate），可以减少缓存未命中的现象。同时，删除那些对视觉影响不大的三角形，可以节省顶点处理的开销和内存占用。较小的存储占用可以降低带宽成本，以及更好地利用缓存。GPU 在显存中所存储的内容也有一些限制，因此使用一些数据简化技术可以让更多的三角形被显示出来。

顶点数据可以使用一些固定压缩比的压缩方法（fixed-rate compression），其原因与纹理压缩（章节 6.2.6）相类似。所谓固定压缩比的压缩方法，指的是在压缩开始之前，我们就已经知道了最终压缩过后的数据大小。每个顶点都使用独立的（self-contained）压缩形式，意味着解码过程可以在 GPU 上进行。Calver [221]提出了使用顶点着色器进行解压的各种方法。Zarge [1961]指出，这种数据压缩还可以帮助将顶点格式与缓存行（cache line）进行对齐。Purnomo 等人[1448]将简化技术和顶点量化技术相结合，并使用图像空间指标（image-space metric）来对给定目标尺寸的网格进行优化。

在索引缓冲区的格式中，可以找到一种简单的压缩形式。索引缓冲区由一个无符号整数（unsigned integer）数组所组成，这些无符号整数给出了顶点缓冲区中顶点的数组位置。如果在顶点缓冲区中的顶点数量少于或者等于 216 个，那么索引缓冲区就可以使用无符号短整型（unsigned short）进行表示，而不是无符号长整型（unsigned long）。对于顶点数量少于 2^8 个的网格，有些 API 还支持使用无符号字节（unsigned byte），但是这样做可能会导致开销更大的对齐问题，因此通常都会避免使用。值得注意的是，OpenGL ES 2.0、无扩展的 WebGL 1.0 以及一些老旧的台式机 GPU 和笔记本电脑 GPU 都有一个限制，即它们不支持 unsigned long 类型的索引缓冲区，因此只能使用 unsigned short 类型。

另一个压缩机会是直接对三角形网格数据本身进行压缩。举一个基本的例子：一些三角形网格会在每个顶点上存储一个或者多个颜色来表示烘焙光照、模拟结果或者其他的一些信息。在一个常见的显示器上，红色、绿色、蓝色各自占据 8 个 bit，因此这些存储在顶点上的数据可以使用三个 unsigned byte 来进行表示，而不是三个 float。GPU 的顶点着色器可以将该字段转换为单独的值，然后在三角形遍历期间对其进行插值。但是在许多架构上都需要进行额外的注意，例如：Apple 建议在 iOS 上将 3 个字节的数据字段填充到 4 个字节，以避免额外的处理[66]，详见图 16.23 中间的插图。

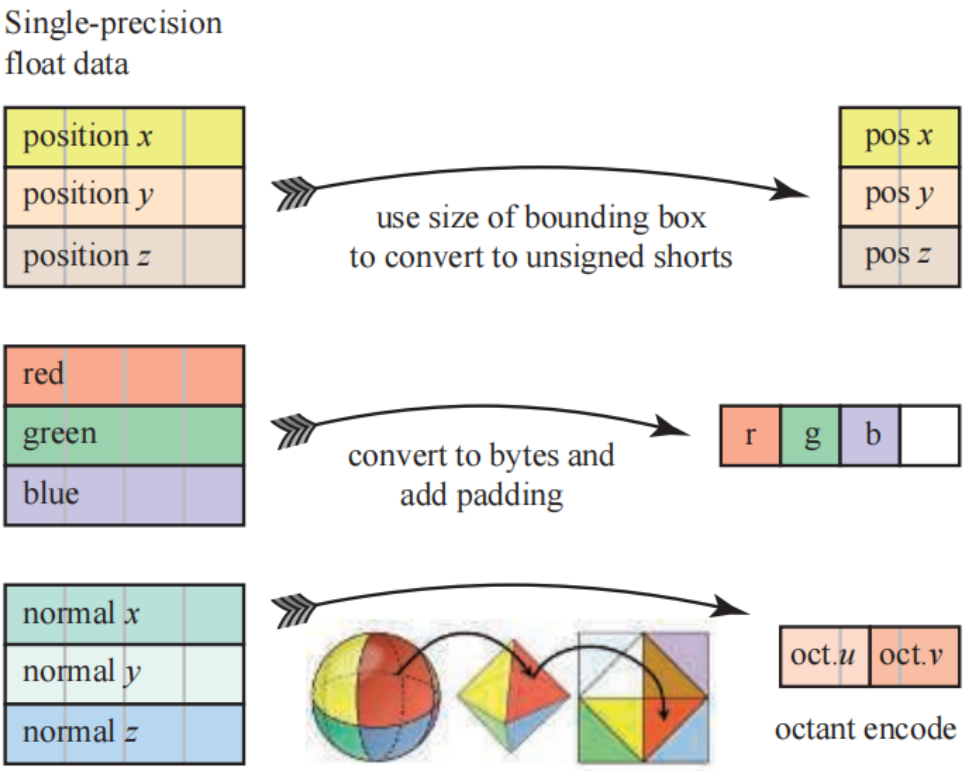


图 16.23：针对顶点数据，一些常见的固定压缩比的压缩方法。[269]

另一种压缩方法则是压根不存储任何颜色。例如：假设这个颜色数据代表的是一些温度结果，而温度值本身就可以存储为单个数字，我们可以将这温度数字转换为对一维颜色纹理的索引。更进一步，如果我们不需要使用这个温度值本身，而是只需要对应的颜色，那么可以使用单个 unsigned byte 来引用这个颜色纹理。

即使有时候我们需要实际存储这个温度值，也可能只需要小数点后几位即可。浮点数（float）的总精度是 24 bit，它比 7 位小数要大一点。而 16 bit 的浮点数就几乎提供了 5 位小数的精度。实际中的温度值范围很小，可能并不需要浮点数格式中的指数部分。通过将最小值作为偏移因子（offset），将最大值减去最小值作为比例因子（scale），剩余值可以在有限范围内均匀分布。例如：如果数值的范围是从 28.51 到 197.12，那么一个 unsigned short 想要转换为一个温度数值，首先需要除以 $2^{16}-1$，然后再乘上比例因子（$197.12-28.51$），最后再加上偏移量 28.51 即可。通过存储数据集的比例因子（scale factor）和偏移因子（offset factor），并将它们传递给顶点着色器程序，那么这个数据集本身只需要占用一半存储空间即可。这种类型的变换被称为标量量化（scalar quantization）[1099]。

顶点的位置数据非常适合这样的简化方法。通常单个网格只会横跨空间中的一小片区域，通过使用一个比例向量和偏移向量（或者一个 4×4 的矩阵）来表示所有的顶点位置，可以为整个场景节省相当大的存储空间，而且并不会带来严重的精度损失。对于某些场景而言，可以为每个模型都生成一个比例因子和偏移因子，从而提高这些模

型的精度。然而，这样做也可能会导致在分离网格相接触的地方出现一些裂缝[1381]。这些顶点原本可能位于世界空间中的相同位置，但是在不同的模型中进行了缩放和偏移，在经过变换之后可能会来到一个略有不同的位置。当场景中的所有模型与场景整体相比来说比较小时，一种解决方案是对所有模型都使用相同的比例因子，并将偏移因子对齐，这样可以提供更多的精度[1010]。

有些时候，即使顶点数据都使用浮点数进行存储，也可能会遇到精度问题。一个经典的例子是：在地球外太空渲染航天飞机。这个航天飞机的模型本身可以精确到毫米尺度，但是地球表面却在 10 万多米之外，二者在尺度上的差距可以达到 8 个小数点之多。在计算航天飞机相对于地球的世界空间位置时，生成的顶点位置需要使用更高的精度。如果不采取一些纠正措施的话，当观察者靠近航天飞机进行观察时，航天飞机会因为精度不足而在屏幕内发生抖动现象。虽然这个航天飞机的例子是这个问题的极端情况，但是如果始终使用单个坐标系统，一些大型多人的开放世界可能会遭受相同的问题。即位于世界边缘处的物体将会失去足够的精度，使得精度问题变得很明显：动画物体将会四处抖动；个别顶点将会在不同的时间发生断裂；阴影贴图中的纹素将会随着轻微的摄像机移动而发生跳变。一种解决方案是重做（redo）变换管线，对于每个以世界原点为中心的物体，世界和相机的平移（translation）首先会被连接（concatenate）在一起，这样大部分精度问题就被抵消了[1379, 1381]。另一种方法是对整个世界进行分割处理，并将原点重新定义在每个部分的中心处，这种方法的挑战性在于，如何处理从一个世界分块移动到另一个世界分块的过度。Ohlarik [1316]、Cozzi 和 Ring [299]对这些问题和解决方案进行了深入讨论。

对于其他的一些顶点数据，还可以有与之关联的特定压缩技术。我们知道纹理坐标通常会被限制在 $[0.0, 1.0]$ 的范围内，因此可以使用 unsigned short 类型来安全地表示纹理坐标，隐式的偏移因子为 0，比例因子为 $2^{16} - 1$。纹理坐标通常都是成对出现，它们可以很好地放入两个 unsigned short [1381]中，甚至是放入 3 个字节[88]中，这取决于具体的精度要求。

与其他坐标集合不同，法线通常是归一化的，即所有归一化的法线集合构成了一个球体。因此研究人员专门研究了从球面到平面的变换，从而对法线数据进行有效压缩。Cigolle 等人[269]对各种算法的优势和权衡进行了分析，并提供了相应的示例代码。他们得出结论的结论是：八象限投影（octant projection）和球面投影（spherical projection）是最实用的，在高效编解码的同时能够最小化误差。Pranckevi [1432] 和 Pesce [1394]讨论了在延迟渲染生成 G-buffer 的时候，如何对法线压缩进行压缩（章节 20.1）。

还有一些其他数据，这些数据具有一些特殊属性，这些特殊属性可以用于减少存储空间。例如：法线、切线和副切线（bitangent）通常会用于法线映射。如果这三个向量相互垂直，并且左右手性一致的话，那么只需要存储两个向量即可，第三个向量可以使用向量叉乘计算得出。再比如：一个 4 byte 的四元数和 1 bit 的手性标识符，再加上一个 7 bit 的 w 分量（一共 5 个 byte），可以表示一个由基底构成的旋转矩阵[494, 1114, 1154, 1381, 1639]。为了进一步提高精度，可以省略 4 个四元数中最大的那个分量，并将另外 3 个分量各存储 10 bit，并使用剩下的 2 个 bit 来标识具体是哪一个值没有被存储。这是由于四元数的平方和为 1，因此我们可以从其他三个值中推导出第四个值[498]。Doghramachi 等人[363]使用切线/副切线/法线的方案来存储轴和角度，这个方案同样也是 4 个 byte，与四元数存储相比，这个解码过程只需要大约一半的着色器指令。

图 16.23 对一些固定压缩比的压缩方法进行了总结。

补充阅读和资源

Meshlab 是一个开源的网格可视化和网格操作系统，它实现了大量的算法，包括网格清理（mesh cleanup）、法线推导、网格简化等。Assimp 是一个开源库，它可以对各种三维文件格式进行读写。有关更多的软件推荐，详见本书的在线网站 realtimerendering.com。

Schneider 和 Eberly [1574]提出了各种关于多边形和三角形的算法，以及相应的伪代码。

虽然 Luebke [1091]的实际调研已经很古老了，但是它仍然是一个对网格简化算法的良好介绍。《Level of Detail for 3D Graphics》[1092]一书涵盖了有关简化和相关主题的内容。

Chapter 17 Curves and Curved Surfaces 曲线和曲面

> Johannes Kepler——“Where there is matter, there is geometry.”

> 约翰内斯·开普勒——“哪里有物质，哪里就有几何学。”（德国天文学家，数学家；1571—1630）

三角形是一个基本的原子渲染图元，三角形会被图形硬件快速转换为着色片元，并放入帧缓冲区中。然而，在建模系统中创建的一些物体和动画路径，可能会具有许多不同的底层几何描述方法。曲线（curve）和曲面（curved surface）可以使用方程进行精确地描述。对这些方程进行计算，并创建一组三角形，然后再将其发送到管线中进行渲染。

曲线和曲面的美妙之处至少有以下四点：

1. 它们的表示方式要比一组三角形更加紧凑。
2. 它们可以提供具有可伸缩性的（scalable）几何图元。
3. 它们所提供的图元，要比直线与平面三角形更加平滑、更加连续。
4. 使用它们进行动画和碰撞检测，会变得更加简单，同时也更快。

这种紧凑的曲线表示方法，可以为实时渲染提供几个优势。首先，可以节省用于存储模型的内存（因此也可以提高内存缓存的效率）。这对于游戏主机而言尤其有用，因为这些游戏主机的内存空间通常没有 PC 那么大。相较于对网格表面进行变换操作，对曲面进行变换通常只需要较少的矩阵乘法即可。如果图形硬件可以直接接收这样的曲面描述，那么 CPU 发送给图形硬件的数据量，通常要比发送三角形网格少得多。

诸如 PN 三角形、细分曲面等曲面模型的描述方法具有这样一个重要的属性，即一个具有较少多边形的模型看起来会更加逼真，更加令人信服。单个多边形会被视为曲面，因此会在表面上创建更多数量的顶点。较高的顶点密度，其结果就是表面和 silhouette 边缘的光照质量会更高，如图 17.1 所示。

图 17.1：《使命召唤：高级战争》中的一个场景，其中角色 Ilona 的面部是使用 Catmull–Clark 细分表面和自适应四叉树算法（章节 17.6.3）进行渲染的。

曲面的另一个主要优点在于它们是可伸缩的。一个曲面描述可以变成 2 个三角形或者 2000 个三角形。曲面是动态 LOD 建模的一种天然形式：当距离曲面物体很近时，可以对其进行更加密集地采样分析，并生成更多的三角形。对于动画来说，曲面的优点在于，需要进行动画的顶点数量要少得多。这些特征点可以用来形成一个曲面，然后再根据这个曲面生成一个更加光滑的细分曲面。此外，碰撞检测也可以变得更加高效和准确[939, 940]。

曲线和曲面的主题贯穿了整本书[458, 777, 1242, 1504, 1847]。我们的目标是对实时渲染中常用的曲线和曲面进行全面介绍。

17.1 参数化曲线

在本小节中，我们将介绍参数化曲线（parametric curve）。参数化曲线会在许多不同的环境中进行使用，并且会使用许多不同的方法来进行实现。对于实时图形程序而言，参数化曲线通常会用于沿着预定义的路径，对相机或者某些物体进行移动。这可能会同时涉及到位置和方向的改变。然而在本章节中，我们只考虑那些改变位置的参数化路径。有关方向插值的内容，详见章节 4.3.2。参数化曲线的另一个用途是毛发渲染，如图 17.2 所示。

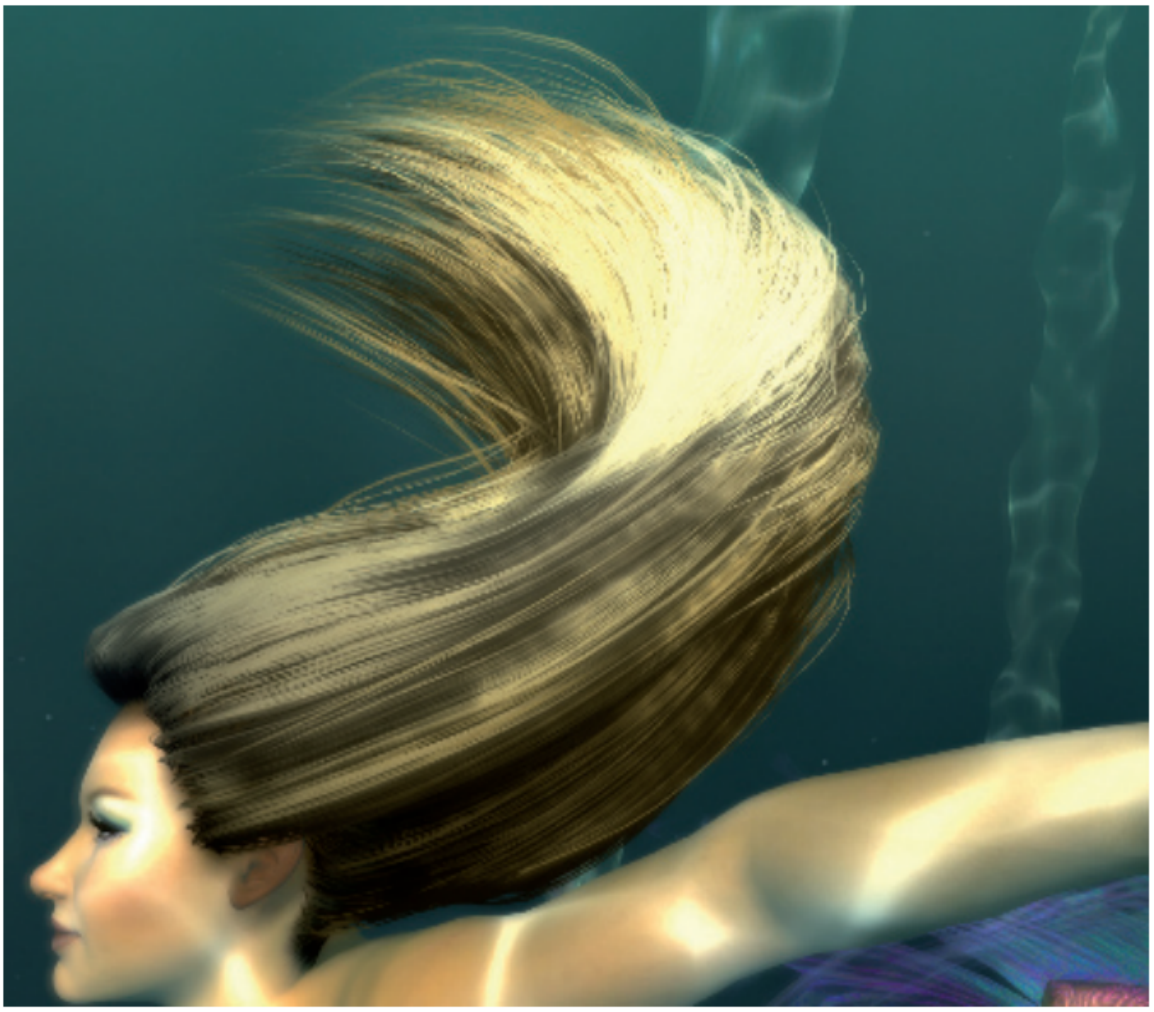


图 17.2：使用细分立方曲线来渲染头发。[1274]

假设我们想在一定的时间内，将相机从一个点移动到另一个点，并且这个执行时间和执行速度与底层硬件的性能无关。举个例子：假设相机应当在一秒内完成这次移动，而渲染一帧需要 50 ms。这意味着我们可以在这一秒中渲染 20 帧画面。而在一个性能更强的计算机上，渲染一帧画面可能只需要 25 ms，相当于每秒 40 帧画面，因此我们希望将相机在这一秒内，移动到 40 个不同的位置上。使用参数化曲线可以很轻松地找到这一组位置。

一条参数化曲线可以使用某种参数 t 的函数，从而对这些点进行描述。在数学上，我们将其写成 $\mathbf{p}(t)$ ，这意味着该函数会为每个 t 值都返回一个点坐标。这个参数 t 可能会属于某个区间，这个区间被称为定义域（domain），例如 $t \in [a, b]$ 。这个参数化曲线生成的点坐标是连续的，即当 $\epsilon \to 0$ 时，有 $\mathbf{p}(t+\epsilon) \to \mathbf{p}(t)$ 。粗略地说，如果 ϵ 是一个非常小的数，那么点 $\mathbf{p}(t)$ 和点 $\mathbf{p}(t+\epsilon)$ 会非常靠近。

在下一小节中，我们将从 Bezier 曲线的直观描述和几何描述开始（Bezier 曲线是一种常见形式的参数化曲线），然后再使用数学语言对其进行精确描述。再然后，我们

会讨论如何使用分段 Bezier 曲线，并介绍曲线的连续性概念。在章节 17.1.4 和章节 17.1.5 中，我们将介绍另外两条十分有用的曲线，即三次 Hermite 样条（cubic Hermite spline）和 Kochanek–Bartels 样条。最后，我们将在章节 17.1.2 中介绍如何使用 GPU 来渲染 Bezier 曲线。

17.1.1 Bezier 曲线

线性插值（linear interpolation）可以在点 $\mathbf{p}_0$ 和点 $\mathbf{p}_1$ 之间画出一条直线，这是很简单的，如图 17.3 左侧的插图所示。给定两个端点，我们可以使用下面的函数来描述一个线性插值点 $\mathbf{p}(t)$，其中 t 是曲线参数，$t \in [0, 1]$：

$$\mathbf{p}(t) = \mathbf{p}_0 + t\,(\mathbf{p}_1 - \mathbf{p}_0) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \tag{17.1}$$

其中的参数 t 控制了点 $\mathbf{p}(t)$ 在直线上的具体位置；$\mathbf{p}(0) = \mathbf{p}_0$，$\mathbf{p}(1) = \mathbf{p}_1$；而 $0 < t < 1$ 则给出了点 $\mathbf{p}_0$ 和点 $\mathbf{p}_1$ 之间线段上的一点。这样一来，如果我们想要在一秒内，将相机以 20 步从点 $\mathbf{p}_0$ 线性移动到点 $\mathbf{p}_1$，那么我们可以令 $t_i = i/(20-1)$，其中 i 代表了第几帧画面（i 从 0 开始，并在 19 结束）。

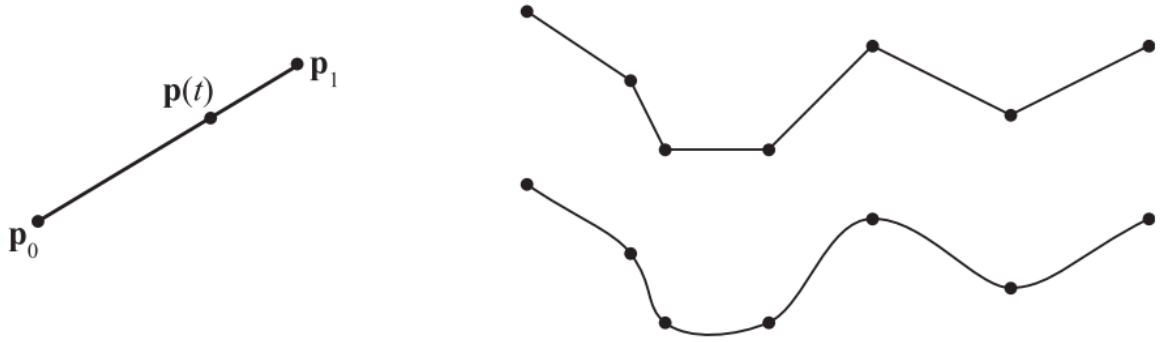


图 17.3：两点之间的线性插值会形成一条直线路径（左）。右侧展示了包含 7 个点的路径，其中右上方展示的是线性插值的结果，右下方展示的是一种更加平滑的插值结果。使用线性插值最令人反感的地方在于，线段之间的连接处会出现不连续变化（即突然的抖动）。

如果我们只需要在两个点之间进行插值，那么线性插值可能就足够了，但是如果路径上存在更多的点，那么线性插值通常就不太行了。例如：在对多个点进行线性插值的时候，会形成一条折线，在连接两个线段的点（也称为关节 joint）上会出现突然变化，这通常是很难接受的，如图 17.3 的右侧所示。

为了解决这个问题，我们将线性插值的方法向前推进了一步，即进行多次线性插值。这样做我们就得到了 Bezier 曲线（读作贝塞尔）的几何结构。这里插播一个历史趣闻，Bezier 曲线是由 Paul de Casteljau 和 Pierre Bezier 独立开发的，并应用于法国

的汽车工业。这个曲线之所以被称为 Bezier 曲线，因为 Bezier 在 de Casteljau 之前就公开了他的研究工作，尽管 de Casteljau 在 Bezier 之前就已经写下了他的技术报告[458]。

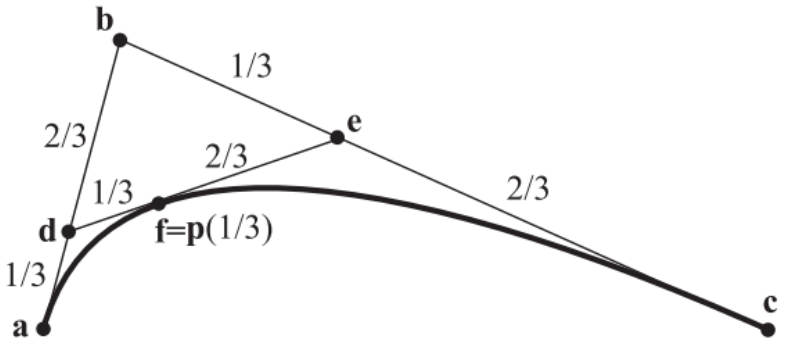


图 17.4：多次线性插值可以得到一条 Bezier 曲线。这条曲线由三个控制点 $\mathbf{a}$ 、 $\mathbf{b}$ 、 $\mathbf{c}$ 进行定义。假设我们想要找到参数 $t = 1/3$ 时曲线上的点，我们首先在点 $\mathbf{a}$ 和点 $\mathbf{b}$ 之间进行线性插值得到点 $\mathbf{d}$ 。接下来，从点 $\mathbf{b}$ 和点 $\mathbf{c}$ 中插值点 $\mathbf{e}$ 。最后在点 $\mathbf{e}$ 和点 $\mathbf{d}$ 之间再次进行线性插值，可以得到最终想要的点 $\mathbf{p}(1/3) = \mathbf{f}$ 。

首先，为了能够重复进行插值，我们必须添加更多的点。例如：可以使用三个点 $\mathbf{a}$ 、 $\mathbf{b}$ 、 $\mathbf{c}$ ，它们被称为控制点（control point）。假设我们想找到点 $\mathbf{p}(1/3)$ ，也就是 $t = 1/3$ 时曲线上的点。我们使用 $t = 1/3$ ，对 $\mathbf{a}\&\mathbf{b}$ 和 $\mathbf{b}\&\mathbf{c}$ 分别进行线性插值，并计算出两个新的顶点 $\mathbf{d}$ 和 $\mathbf{e}$ ，如图 17.4 所示。最后，我们再次使用 $t = 1/3$ ，对点 $\mathbf{d}$ 和点 $\mathbf{e}$ 进行线性插值来计算点 $\mathbf{f}$ 。这里我们定义 $\mathbf{p}(t) = \mathbf{f}$ ，使用这种方法，我们可以得到以下数学关系：

$$\begin{aligned}\mathbf{p}(t) &= (1-t)\mathbf{d} + t\mathbf{e} \\ &= (1-t)[(1-t)\mathbf{a} + t\mathbf{b}] + t[(1-t)\mathbf{b} + t\mathbf{c}] \\ &= (1-t)^2\mathbf{a} + 2(1-t)t\mathbf{b} + t^2\mathbf{c},\end{aligned} \tag{17.2}$$

这是一条抛物线，因为参数 t 的最大次数为 2。事实上，给定 $n+1$ 个控制点，则曲线的自由度即为 n 。这意味着控制点的数量越多，曲线的自由度就越大。一次曲线是一条直线（称为 linear），二次曲线被称为 quadratic，三次曲线被称为 cubic，四次曲线被称为 quartic，等等。

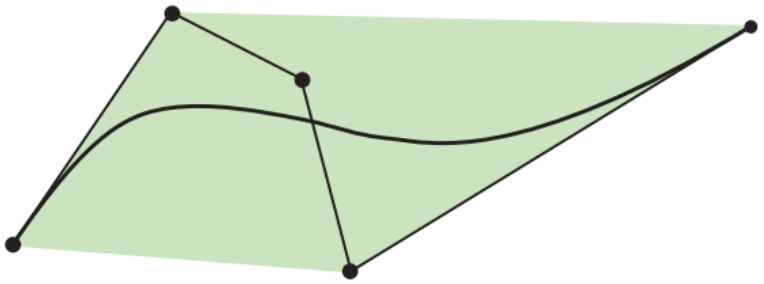


图 17.5：在五个点之间重复进行线性插值，最终会得到一个四次 Bezier 曲线。曲线的控制点使用黑色圆点来进行表示，整个曲线位于控制点所形成的凸包（绿色区域）内部。同时，曲线上的起始点（第一个点），与第一个控制点和第二个控制点之间的直线相切。曲线的另一端（结束点）也是如此。

这种重复或者递归的线性插值，通常被称为 de Casteljau 算法[458, 777]。图 17.5 中展示了使用 5 个控制点时的效果。为了进行一般化的表示，这里并没有使用点 $\mathbf{a}-\mathbf{f}$，而是使用下面的表示法，即将第 i 个控制点记为 $\mathbf{p}_i$，因此在图 17.4 的例子中，$\mathbf{p}_0=\mathbf{a}$，$\mathbf{p}_1=\mathbf{b}$，$\mathbf{p}_2=\mathbf{c}$。同时，在经过 k 次线性插值之后，可以得到中间控制点 $\mathbf{p}_i^k$，因此在图 17.4 的例子中 $\mathbf{p}_0^1=\mathbf{d}$，$\mathbf{p}_1^1=\mathbf{e}$，$\mathbf{p}_0^2=\mathbf{f}$。

包含 $n+1$ 个控制点的 Bezier 曲线可以使用如下的递归公式进行描述，其中 $\mathbf{p}_i^0=\mathbf{p}_i$ 为初始控制点：

$$\mathbf{p}_i^k(t)=(1-t)\mathbf{p}_i^{k-1}(t)+t\mathbf{p}_{i+1}^{k-1}(t),\quad \begin{cases} k=1\ldots n, \\ i=0\ldots n-k \end{cases} \tag{17.3}$$

请注意，该曲线上的一个点使用 $\mathbf{p}(t)=\mathbf{p}_0^n(t)$ 来进行描述，这并不像它看起来那样复杂。再次思考一下，当我们从三个点 $\mathbf{p}_0$，$\mathbf{p}_1$，$\mathbf{p}_2$ 来构造 Bezier 曲线时会发生什么，这三个点实际上就等价于 $\mathbf{p}_0^0$，$\mathbf{p}_1^0$ 和 $\mathbf{p}_2^0$。现在我们有 3 个控制点，这意味着 $n=2$。为了对公式简化表示，有时候我们会把“(t)”从“$\mathbf{p}$”中去掉。在第一步中 $k=1$，我们可以得到：

$$\begin{aligned}\mathbf{p}_0^1&=(1-t)\mathbf{p}_0+t\mathbf{p}_1\\ \mathbf{p}_1^1&=(1-t)\mathbf{p}_1+t\mathbf{p}_2\end{aligned}$$

最后，当 $k=2$ 时，我们可以得到：

$$\mathbf{p}_0^2=(1-t)\mathbf{p}_0^1+t\mathbf{p}_1^1$$

这与直接求 $\mathbf{p}(t)$ 的结果是相同。图 17.6 展示了它的运作原理。

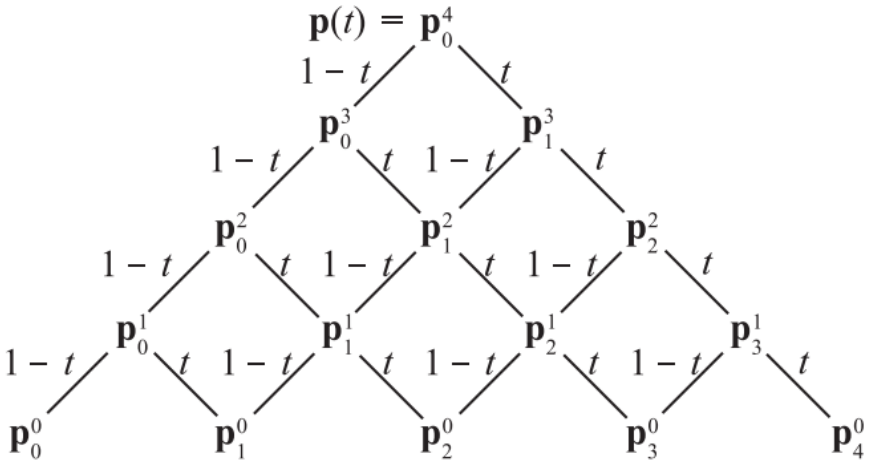


图 17.6：这个图说明了 Bezier 曲线是如何使用重复线性插值来运行的。在这个例子中，展示了一个四次曲线的插值过程。四次曲线意味着存在 5 个控制点，即 $\mathbf{p}_i^0$，其中 $i = 0, 1, 2, 3, 4$，这 5 个控制点位于金字塔的最底部。这个图应当从下往上看，即点 $\mathbf{p}_0^0$ 的权重为 $1-t$，点 $\mathbf{p}_1^0$ 的权重为 t，两个点之间进行线性插值，从而形成点 $\mathbf{p}_0^1$。这个过程会不断重复，直到形成最顶部的点 $\mathbf{p}(t)$。[551]

现在我们已经掌握了 Bezier 曲线是如何运行的基础知识，现在我们可以看看对 Bezier 曲线更加数学的描述。

使用 Bernstein 多项式的 Bezier 曲线

如方程 17.2 所示，二次 Bezier 曲线可以使用一个代数公式来进行描述。事实证明，每条 Bezier 曲线都可以使用这样一个代数公式来进行描述，这意味着我们不需要真的执行这个重复插值的过程。方程 17.4 中展示了这个公式，它可以产生与方程 17.3 相同的曲线。Bezier 曲线的这种描述方法，被称为 Bernstein 形式：

$$\mathbf{p}(t) = \sum_{i=0}^{n} B_i^n(t)\mathbf{p}_i \tag{17.4}$$

方程 17.4 中包含了一个 Bernstein 多项式，它有时也被称为 Bezier 基函数，这个多项式的数学形式如下：

$$B_i^n(t) = \binom{n}{i} t^i(1-t)^{n-i} = \frac{n!}{i!(n-i)!}t^i(1-t)^{n-i} \tag{17.5}$$

方程 17.5 中的第一项 $\binom{n}{i}$，被称为二项式系数（binomial coefficient），在第 1 章的方程 1.6 中进行了定义。Bernstein 多项式的有如下两个基本性质：

$$B_i^n(t) \in [0,1], \quad \text{when} \quad t \in [0,1],$$
$$\sum_{i=0}^{n} B_i^n(t) = 1 \tag{17.6}$$

方程 17.6 中的第一个公式意味着，当 t 的范围为 $[0,1]$ 时，Bernstein 多项式的结果也在范围 $[0,1]$ 内。第二个公式意味着，无论方程 17.4 中的 Bezier 曲线次数为多少，Bernstein 多项式的求和结果均为 1（如图 17.7 所示）。粗略地说，这个性质意味着最终的 Bezier 曲线，将保持“靠近”控制点 $\mathbf{p}_i$。事实上，根据方程 17.4 和方程 17.6，整个 Bezier 曲线都位于控制点所形成的凸包中（convex hull，详见在线网站的线性代数附录）。这个性质在计算曲线的包围面积或者包围体积时，是一个十分有用的属性。图 17.5 展示了这样的一个例子。

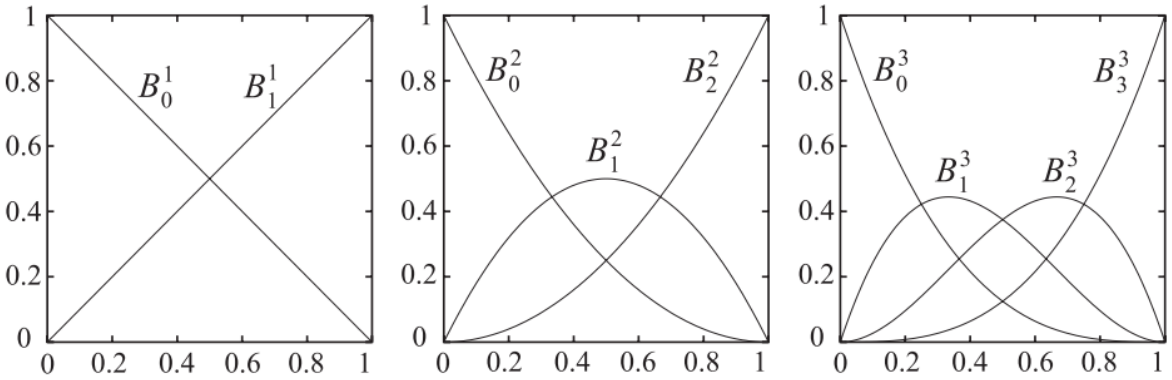


图 17.7：从左到右分别是 $n=1$，$n=2$，$n=3$ 时的 Bernstein 多项式。左侧是线性插值，中间是二次插值，右边是三次插值。这些是 Bernstein 形式的 Bezier 曲线中所使用的混合函数。因此，想要计算 t 值处的二次曲线值（中），只需在 x 轴上找到这个 t 值，然后进行垂直移动，依次与三条曲线相遇，这三个交点的 y 坐标就是这三个控制点所对应的权重。注意当 $t \in [0,1]$ 时，才有 $B_i^n(t) \geq 0$；同时，这些混合函数都具有对称性，即 $B_i^n(t) = B_{n-i}^n(1-t)$。

图 17.7 中展示了 $n=1$、$n=2$、$n=3$ 时的 Bernstein 多项式。它们也称为混合函数（blending function）。当 $n=1$（线性插值）时，情况是显而易见的，它给出了 $y=1-t$ 和 $y=t$ 这两条直线。这意味着当参数 $t=0$ 时，$\mathbf{p}(0)=\mathbf{p}_0$；当参数 t 逐渐增加时，点 $\mathbf{p}_0$ 的混合权重将会降低，而点 $\mathbf{p}_1$ 的混合权重将会增加，并保持二者的权重之和为 1。最后，当参数 $t=1$ 时，$\mathbf{p}(1)=\mathbf{p}_1$。一般来说，对于所

有的 Bezier 曲线，$\mathbf{p}(0) = \mathbf{p}_0$ 和 $\mathbf{p}(1) = \mathbf{p}_n$ 都是成立的，即端点也会被插值（即在曲线上）。同样，在 $t = 0$ 时，Bezier 曲线会与向量 $\mathbf{p}_1 - \mathbf{p}_0$ 相切；在 $t = 1$ 时，Bezier 曲线会与向量 $\mathbf{p}_n - \mathbf{p}_{n-1}$ 相切。另一个十分有用的特性是，我们在对 Bezier 曲线进行旋转操作的时候，不需要先计算 Bezier 曲线上的点，然后再旋转曲线；而是先旋转形成 Bezier 曲线的控制点，然后再直接计算曲线上的点即可。曲线上的控制点通常要比生成的点少，因此先对控制点进行变换的效率会更高。

这里我们举一个例子，来了解 Bernstein 版本的 Bezier 曲线是如何运行的。这里我们假设 $n = 2$，即一个二次 Bezier 曲线。此时方程 17.4 为：

$$\begin{aligned}\mathbf{p}(t) &= B_0^2\mathbf{p}_0 + B_1^2\mathbf{p}_1 + B_2^2\mathbf{p}_2 \\ &= \begin{pmatrix} 2 \\ 0 \end{pmatrix} t^0(1-t)^2\mathbf{p}_0 + \begin{pmatrix} 2 \\ 1 \end{pmatrix} t^1(1-t)^1\mathbf{p}_1 + \begin{pmatrix} 2 \\ 2 \end{pmatrix} t^2(1-t)^0\mathbf{p}_2 \\ &= (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2,\end{aligned} \tag{17.7}$$

方程 17.7 和方程 17.2 实际上是一样的。请注意方程 17.7 中的混合函数 $(1-t)^2$、$2t(1-t)$ 和 t^2，它们实际上就是图 17.7 中间所展示的函数。以同样的方式，一个三次 Bezier 曲线被化简为：

$$\mathbf{p}(t) = (1-t)^3\mathbf{p}_0 + 3t(1-t)^2\mathbf{p}_1 + 3t^2(1-t)\mathbf{p}_2 + t^3\mathbf{p}_3 \tag{17.8}$$

方程 17.8 可以写成矩阵形式，有时对于数学化简十分有用：

$$\mathbf{p}(t) = \begin{pmatrix} 1 & t & t^2 & t^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix} \tag{17.9}$$

通过收集方程 17.4 中构成 t^k 的项，我们可以看出，每条 Bezier 曲线都可以写成如下的形式，它被称为幂形式（power form），其中 $\mathbf{c}_i$ 是通过收集这些项而得到的点：

$$\mathbf{p}(t) = \sum_{i=0}^{n} t^i\mathbf{c}_i \tag{17.10}$$

为了得到 Bezier 曲线的导数，我们需要对方程 17.4 进行求导，这个求导过程是很简单的。对推导过程进行重新整理和化简之后，可以得到如下的结果[458]：

$$\frac{d}{dt}\mathbf{p}(t) = n\sum_{i=0}^{n-1} B_i^{n-1}(t)\left(\mathbf{p}_{i+1} - \mathbf{p}_i\right) \tag{17.11}$$

实际上，这个导数同样也是一条 Bezier 曲线，但是要比原本的 $\mathbf{p}(t)$ 低一阶。

Bezier 曲线的一个潜在的缺点是，这条曲线并不会经过所有的控制点（除了两侧的端点之外）。另一个问题在于，随着控制点数量的增加，方程的次数也在增加，从而使得计算过程越来越昂贵。一个解决这个问题的方法是，在每对控制点之间使用一条简单的低阶曲线，并确保这种分段插值具有足够高的连续性，这是章节 17.1.3 到章节 17.1.5 的主题。

有理 Bezier 曲线

虽然 Bezier 曲线对许多事情都十分有用，但是实际上 Bezier 曲线的自由度并不是很高，因为只有控制点的位置可以进行自由控制。而且，并不是所有的曲线都可以使用 Bezier 曲线来进行描述的。例如：一个圆形通常被认为是一个十分简单的形状，但是这个简单的圆形无法使用一条或者一组 Bezier 曲线来进行定义。另一种选择是有理 Bezier 曲线（rational Bezier curve），该类曲线的描述方程如下所示：

$$\mathbf{p}(t) = \frac{\sum_{i=0}^{n} w_i B_i^n(t)\mathbf{p}_i}{\sum_{i=0}^{n} w_i B_i^n(t)} \tag{17.12}$$

其中方程中的分母是 Bernstein 多项式的加权和，而分子则是标准的 Bezier 曲线（方程 17.4）的加权版本。对于这种类型的曲线，用户可以使用权重 w_i 来添加额外的自由度。有关这些曲线的更多信息，详见 Hoschek 和 Lasser 所撰写的书[777]，以及 Farin 所撰写的书[458]。Farin 还描述了如何使用三条有理 Bezier 曲线来描述一个圆。

17.1.2 GPU 上的有界 Bezier 曲线

本小节将介绍一种在 GPU 上绘制 Bezier 曲线的方法[1068, 1069]。具体来说，目标的目标是“有界 Bezier 曲线（bounded Bezier curve）”，其中这个 Bezier 曲线本身与首尾控制点之间的直线构成了一个封闭区域，这个区域会被填充。有一种十分简单的方法可以实现这一点，那就是使用一个专门的像素着色器来渲染一个三角形。

这里我们使用一个二次曲线（quadratic curve），即二次 Bezier 曲线，相应的控制点为 $\mathbf{p}_0$，$\mathbf{p}_1$，$\mathbf{p}_2$。如果我们将这些顶点的纹理坐标设置为 $t_0 = (0,0)$、$t_1 =$

$(0.5, 0)$ 、$t_2 = (1, 1)$，那么在渲染三角形 $\Delta \mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$ 的时候，这些纹理坐标会像往常一样进行插值。我们还会对三角形内的每个像素都计算下面这个标量函数，其中 u 和 v 是插值出的纹理坐标：

$$f(u, v) = u^2 - v \tag{17.13}$$

然后像素着色器会根据函数 $f(u, v)$ 的正负性，来判断像素位于曲线内部（$f(u, v) < 0$），还是位于曲线外部（$f(u, v) > 0$），如图 17.8 所示。当使用这个像素着色器来渲染一个透视投影的三角形时，同样我们会得到一个相应的投影 Bezier 曲线。Loop 和 Blinn 对此给出了证明[1068, 1069]。

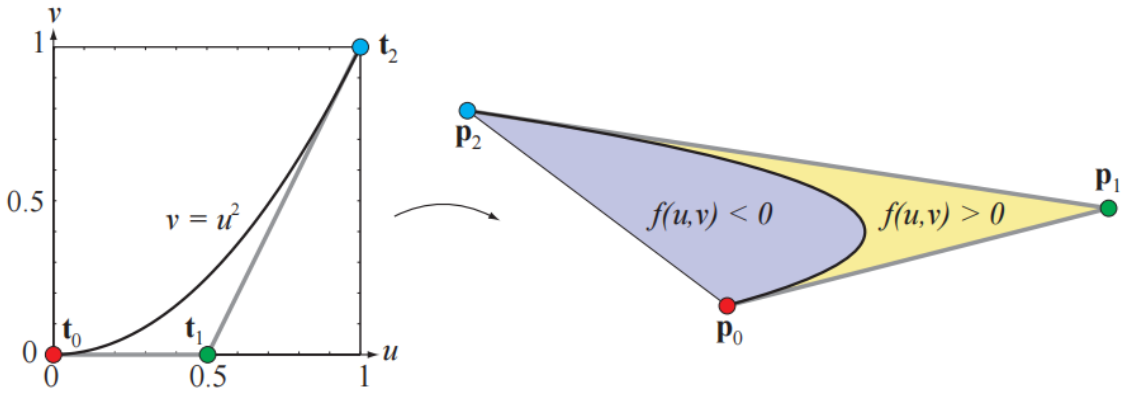


图 17.8：有界 Bezier 曲线的渲染。左边：在规范纹理空间中展示了曲线。右边：曲线在屏幕空间中进行渲染。如果使用条件 $f(u, v) \geq 0$ 来丢弃曲线外部的像素，则可以获得右图浅蓝色区域的渲染结果。

这种类型的技术可以用于渲染 TrueType 字体，如图 17.9 所示。Loop 和 Blinn 还展示了如何渲染有理二次曲线和有理三次曲线，以及如何使用这种表示方法进行抗抗锯齿处理。由于文本渲染的重要性，因此这一领域的研究工作一直在继续，相关文本算法详见章节 15.5。

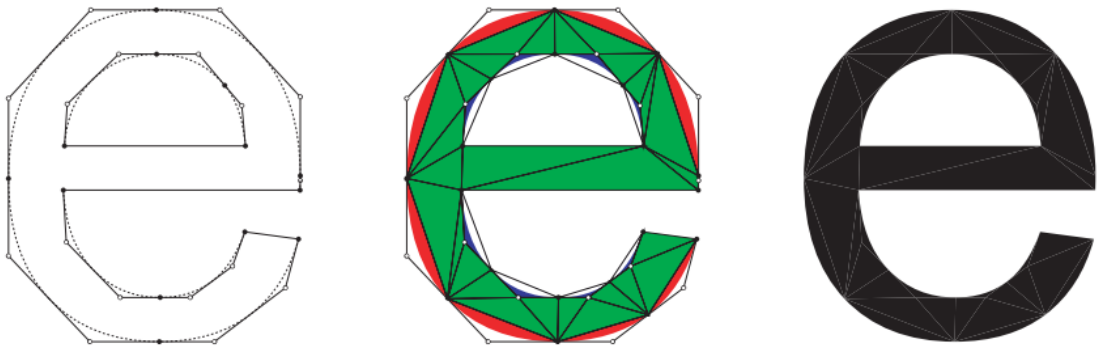


图 17.9：左边：字母 e 由几条直线和几条二次 Bezier 曲线进行表示。中间：这种表示被“细分”成若干个有界的 Bezier 曲线（外部的红色和内部的蓝色）以及若干个三角形（绿色）。右边：最终渲染出的字母效果。

17.1.3 曲线的连续性与分段 Bezier 曲线

假设现在我们有两条三次 Bezier 曲线，三次 Bezier 曲线也就意味着每条曲线由四个控制点进行定义。其中第一条曲线由控制点 $\mathbf{q}_i$ 进行定义，第二条曲线由控制点 $\mathbf{r}_i$ 进行定义，其中 $i = 0, 1, 2, 3$ 。为了连接这两条曲线，我们可以设定 $\mathbf{q}_3 = \mathbf{r}_0$ ，这个点被叫做关节（joint）。然而如图 17.10 所示，使用这种简单的连接技术，关节处不可能变得很光滑。由若干条曲线片段（在这个例子中为两条）所组成的复合曲线被称为分段 Bezier 曲线（piecewise Bezier curve），在这里记为 $\mathbf{p}(t)$ 。进一步地，假设我们希望 $\mathbf{p}(0) = \mathbf{q}_0$ ， $\mathbf{p}(1) = \mathbf{q}_3 = \mathbf{r}_0$ ， $\mathbf{p}(3) = \mathbf{r}_3$ 。因此，这个符合曲线到达点 $\mathbf{q}_0$ 、 $\mathbf{q}_3 = \mathbf{r}_0$ 、 $\mathbf{r}_3$ 的时间（参数）分别为 $t_0 = 0.0$ 、 $t_1 = 1.0$ 、 $t_2 = 3.0$ ，如图 17.10 中的标记。

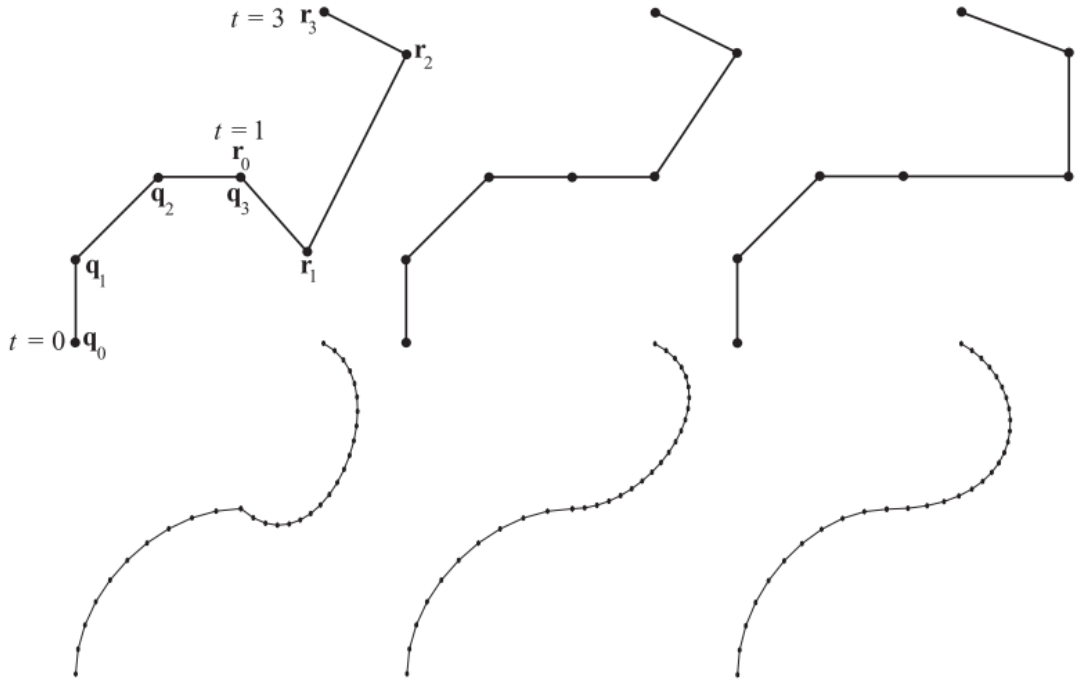


图 17.10：图中展示了两个三次 Bezier 曲线（每个曲线有四个控制点）之间不同的连续性，从左到右分别是 C^0 连续， G^1 连续， C^1 连续。第一行展示了控制点，第二行展示了曲线，其中左侧 $\mathbf{q}_i$ 段的曲线上有 10 个样本点，左侧 $\mathbf{r}_i$ 段的曲线上有 20 个样本点。在这个例子中，使用了一下几个时间点对： $(0.0, \mathbf{q}_0)$ ， $(1.0, \mathbf{q}_3)$ ， $(3.0, \mathbf{r}_3)$ 。对于 C^0 连续的情况，连接处会有一个突然的抖动（其中 $\mathbf{q}_3 = \mathbf{r}_0$ ）。对于 G^1 连续的情况，通过使得连接处的切线相互平行（且长度相等）来改善平滑性。然而，由于 $3.0 - 1.0 \neq 1.0 - 0.0$ ，因此仅仅是切线平行

还无法提供 C^1 连续。我们可以在连接处观察到（第二行中间），样本点的密度突然增大。为了得到 C^1 连续，要使得连接处右侧两个控制点的差向量，其长度必须是左侧两个控制点差向量的两倍，即与参数差相对应。

从上一小节中我们知道，一条 Bezier 曲线对于 $t \in [0,1]$ 才有定义，因此对于控制点 $\mathbf{q}_i$ 所定义的第一段曲线而言，这是符合要求的，因为 $\mathbf{q}_0$ 处的时间为 0.0， $\mathbf{q}_3$ 处的时间为 1.0。但是当 $1.0 < t \leq 3.0$ 时会发生什么呢？答案很简单：我们必须使用第二段曲线，然后将第二段曲线的参数区间从 $[t_1, t_2]$ 平移并缩放到 $[0,1]$ 范围内。这可以通过使用下面的公式来完成：

$$t' = \frac{t - t_1}{t_2 - t_1} \tag{17.14}$$

因此，对于由控制点 $\mathbf{r}_i$ 所定义的 Bezier 曲线， t' 才是真正的参数。使用这种方法，可以很容易地将好几个 Bezier 曲线拼接在一起。

现在我们构建的分段曲线在关节处的平滑性很差，一种连接曲线的更好方法基于了这样的一个事实：在 Bezier 曲线的第一个控制点处，曲线与直线 $\mathbf{q}_1 - \mathbf{q}_0$ 相切（章节 17.1.1）；同样地，在最后一个控制点处，三次曲线会与 $\mathbf{q}_3 - \mathbf{q}_2$ 相切，这种特性可以在图 17.5 中看到。因此，为了使得两条曲线在连接处相切，第一条曲线的切线应当与第二条曲线的切线在该点处平行。更正式地说，应当遵循以下方程：

$$(\mathbf{r}_1 - \mathbf{r}_0) = c\,(\mathbf{q}_3 - \mathbf{q}_2) \quad \text{for} \;\; c > 0 \tag{17.15}$$

方程 17.15 意味着，关节处的入射切线 $\mathbf{q}_3 - \mathbf{q}_2$ 应当与出射切线 $\mathbf{r}_1 - \mathbf{r}_0$ 的方向相同。

在方程 17.15 中使用由方程 17.16 所定义的系数 c，可以实现更好的连续性（ C^1 连续）[458]。

$$c = \frac{t_2 - t_1}{t_1 - t_0} \tag{17.16}$$

图 17.10 中也展示了这一点。如果我们设 $t_2 = 2.0$，那么有 $c = 1.0$；也就是说，当两侧曲线段上的时间间隔相等时，那么入射切向量和出射切向量的长度也应当是相同的。但是，当 $t_2 = 3.0$ 时，切向量的长度相等就不行了。虽然曲线看起来好像是一样的，但是 $\mathbf{p}(t)$ 在复合曲线上的移动速度（采样点的密度）并不是平滑的。使用方程 17.16 中的常数 c 可以解决这个问题。

使用分段曲线的优点在于，可以使用一些低阶曲线进行表示，并且最终得到的曲线将会经过一组控制点。在上面的例子中，每个曲线段都是一个三次 Bezier 曲线。通常都会使用一个三次 Bezier 曲线，因为它是可以描述一个 S 形曲线的最低次曲线（被称为 inflection）。最终得到的曲线 $\mathbf{p}(t)$ 会经过点 $\mathbf{q}_0$ ， $\mathbf{q}_3 = \mathbf{r}_0$ ， $\mathbf{r}_3$ 。

这里我们将通过一个实际例子来介绍两种重要的连续性指标，下面是曲线连续性概念的一种稍微数学化的表述。对于曲线而言，我们通常会使用符号 C^n 来区分关节处不同类型的连续性。 C^n 意味着整个曲线上的 n 阶导数都应当是连续且非零的。 C^0 连续意味着线段应当在同一点相连接，线性插值就可以满足这个条件了，本小节中所介绍的第一个例子就是这种情况。 C^1 连续意味着，如果我们在曲线上的任何一点（包括关节处）都进行一次求导操作，那么求导的结果（一阶导数）也应当是连续的。本小节中所介绍的第三个例子就是这样的，它使用了方程 17.16 来进行修正。

还有一个指标，记为 G^n ，这里我们以 G^1 连续（几何连续）为例。对于 G^1 连续的情况，在关节处相交的曲线段，两侧的切向量应当是平行的，并且方向相同，但是并没有对长度的要求。换句话说， G^1 连续要比 C^1 连续更弱， C^1 连续的曲线总是 G^1 连续的，除非两条曲线在连接点处的的速度（velocity ）趋近于 0，并且在连接点之前还具有不同的切线。几何连续性的概念可以推广到更高的维度，图 17.10 中间的插图展示了 G^1 连续的情况 。

17.1.4 三次 Hermite 插值

Bezier 曲线很好地描述了光滑曲线构造背后的理论，但是有时候它的控制性无法很好地进行预测。在小本节中，我们将介绍三次 Hermite 插值，这样的曲线往往会更加容易控制。其原因在于，一条三次 Bezier 曲线是通过使用四个控制点来进行描述的，而三次 Hermite 曲线则使用了起点 $\mathbf{p}_0$ 和终点 $\mathbf{p}_1$ ，以及起点切线 $\mathbf{m}_0$ 和终点切线 $\mathbf{m}_1$ 来进行定义的。这里我们同样将 Hermite 插值记为 $\mathbf{p}(t)$ ，其中 $t \in [0, 1]$ ，其数学定义如下：

$$\mathbf{p}(t) = \left(2t^3 - 3t^2 + 1\right)\mathbf{p}_0 + \left(t^3 - 2t^2 + t\right)\mathbf{m}_0 + \left(t^3 - t^2\right)\mathbf{m}_1 + \left(-2t^3 + 3t^2\right)\mathbf{p}_1 \quad (17.17)$$

我们也将 $\mathbf{p}(t)$ 称为一个 Hermite 曲线段或者一个三次样条段。这是一个三次插值，因为是方程 17.17 中混合函数的最高次数为 t^3 。这条曲线有以下性质：

$$\mathbf{p}(0) = \mathbf{p}_0, \quad \mathbf{p}(1) = \mathbf{p}_1, \quad \frac{\partial \mathbf{p}}{\partial t}(0) = \mathbf{m}_0, \quad \frac{\partial \mathbf{p}}{\partial t}(1) = \mathbf{m}_1 \quad (17.18)$$

这意味着 Hermite 曲线在起点 $\mathbf{p}_0$ 和终点 $\mathbf{p}_1$ 之间进行了插值，并且这两点处的切线为 $\mathbf{m}_0$ 和 $\mathbf{m}_1$ 。图 17.11 中展示了由方程 17.17 得到的混合函数，这些混合函数也可以由方程 17.4 和方程 17.18 中推导出。

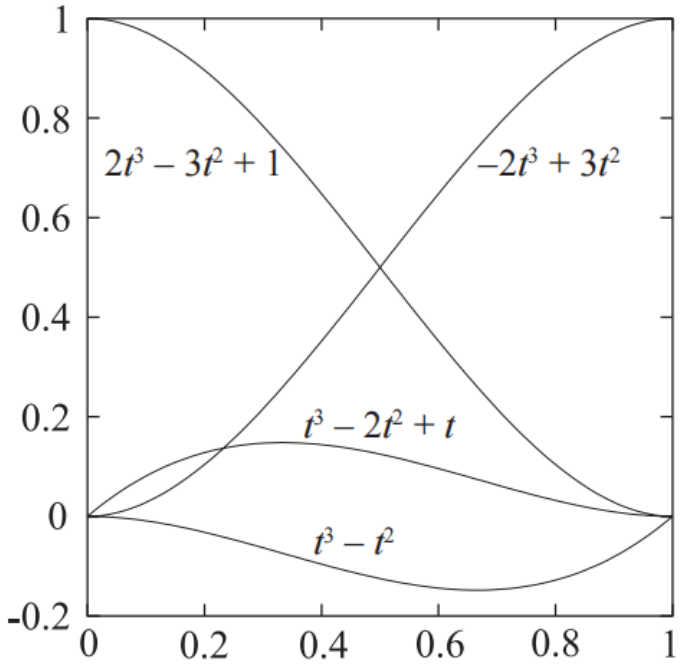


图 17.11：三次 Hermite 插值的混合函数。请注意切线混合函数的不对称性。将方程 17.17 中的混合函数 $t^3 - t^2$ 和 $\mathbf{m}_1$ 取负，可以得到一个对称的外观。

图 17.12 中展示了一些三次 Hermite 插值的例子，所有这些例子都对相同的起点和终点进行了插值，但是它们具有不同的切线。请注意图中切线的长度，不同长度的切线会给出不同的结果，更长的切线将会对整体形状产生更大的影响。

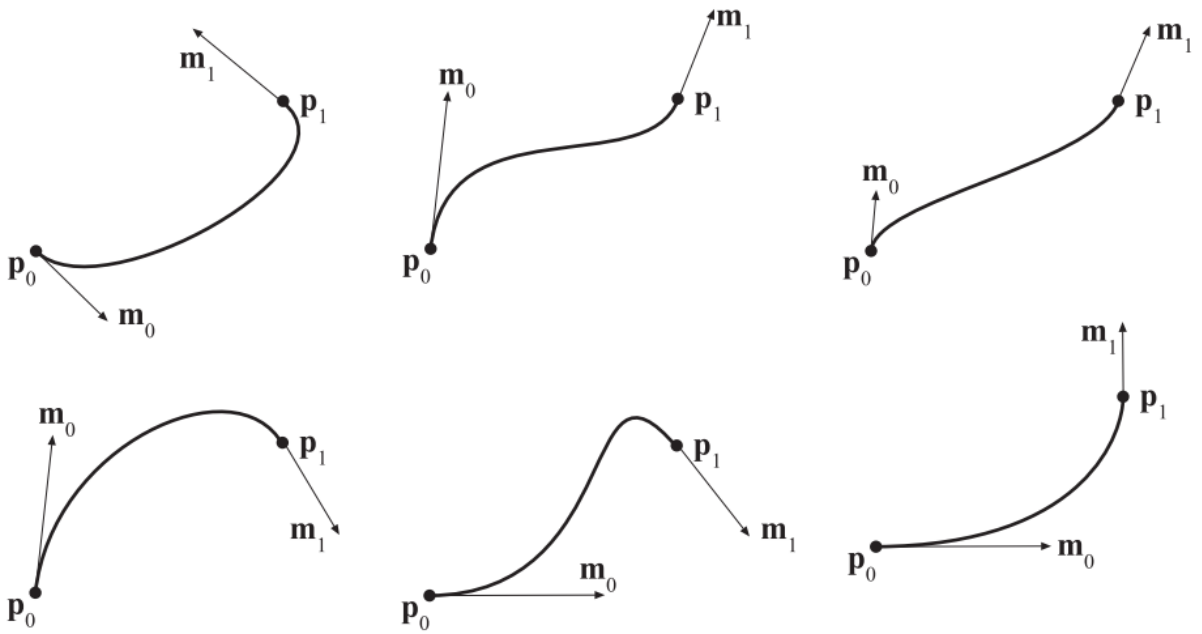


图 17.12：一些 Hermite 插值的例子。一条曲线由两个点和两条切点进行定义，分别是起点 $\mathbf{p}_0$ 和终点 $\mathbf{p}_1$ ，以及每个点处的切线 $\mathbf{m}_0$ 和 $\mathbf{m}_1$ 。

在 Nalu 的 Demo 中[1274]，使用了三次 Hermite 插值来渲染毛发，详见图 17.2。一个粗糙（coarse）控制的毛发会用于动画和碰撞检测，然后会计算切线，并对三次曲线进行细分和渲染。

17.1.5 Kochanek–Bartels 曲线

当在多个点之间进行插值的时候，可以将若干条 Hermite 曲线连接起来。然而在我们这样做的时候，在选择共享切线上具有一定的自由度，选择不同的切线会提供不同的曲线外观。在这里，我们将介绍一种计算这种切线的方法，它被称为 Kochanek–Bartels 曲线。假设现在我们有 n 个点，即 $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$ ，则需要插值 $n-1$ 个 Hermite 曲线段。这里我们假设在每个点上只会存在一条切线，现在我们可以观察一下这些“内部”切线，即 $\mathbf{m}_1, \ldots, \mathbf{m}_{n-2}$ 。点 $\mathbf{p}_i$ 处的切线可以使用两个弦（chord）的组合来进行计算[917]： $\mathbf{p}_i - \mathbf{p}_{i-1}$ 和 $\mathbf{p}_{i+1} - \mathbf{p}_i$ ，如图 17.13 左侧所示。

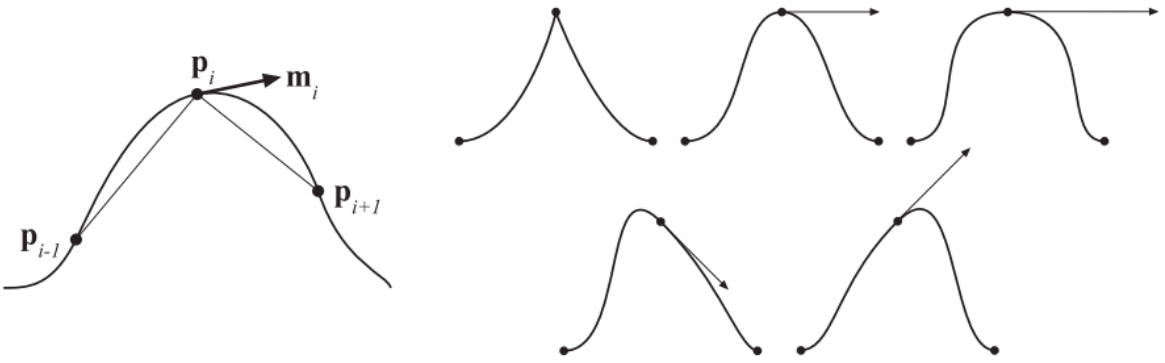


图 17.13：左：一种计算切线的方法是使用弦（chord）的组合。右边第一行中的三条曲线分别具有不同的张力参数（ a ）。其中第一条曲线的 $a \approx 1$ ，代表张力较高的情况；第二条曲线的 $a \approx 0$ ，代表默认张力的情况；第三条曲线的 $a \approx -1$ ，代表张力较低的情况。右边第二行中的两条曲线分别具有不同的偏移参数（ b ）。其中第一条曲线的偏移参数为负数，第二条曲线的偏移参数为正数。

首先，我们引入一个张力（tension）参数 a ，来对切向量的长度进行修正。它控制了关节处曲线的尖锐程度。切线的计算方法为：

$$\mathbf{m}_i = \frac{1-a}{2}\left((\mathbf{p}_i - \mathbf{p}_{i-1}) + (\mathbf{p}_{i+1} - \mathbf{p}_i)\right) \tag{17.19}$$

图 17.13 的右侧第一行，展示了不同的张力参数所带来的外观表现。这个张力参数的默认值是 $a = 0$；更高的张力参数可以带来更加尖锐的弯曲（如果 $a > 1$，则会在关节处形成一个环），一个负值会使得关节附近的曲线不那么紧绷（taut）。其次，我们引入一个偏移（bias）参数 b，它会影响切线的方向（同时间接影响切线的长度）。同时使用张力参数 a 和偏移参数 b，我们可以得到新的法线：

$$\mathbf{m}_i = \frac{(1-a)(1+b)}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{(1-a)(1-b)}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i) \tag{17.20}$$

其中偏移参数的默认值是 $b = 0$。一个正的偏移量会使得弯曲更倾向于弦 $\mathbf{p}_i - \mathbf{p}_{i-1}$；一个负的偏移量会使得弯曲更倾向于弦 $\mathbf{p}_{i+1} - \mathbf{p}_i$。如图 17.13 右侧第二行所示。用户可以自行设置张力参数和偏移参数，或者是让它们保持默认值，这通常会产生所谓的 Catmull–Rom 样条[236]。曲线段的第一个点和最后一个点的切线，也可以使用这些公式进行计算，直接让其中一个弦的长度为 0 即可。

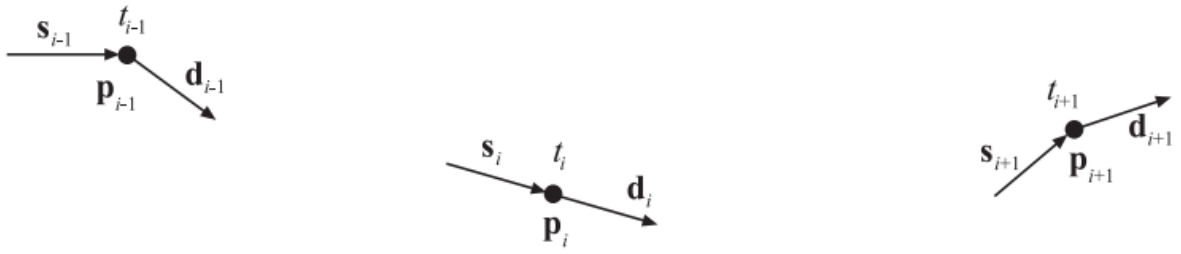


图 17.14：Kochanek–Bartels 曲线的入射切线和出射切线。在每个控制点 $\mathbf{p}_i$ 上，还显示了对应的参数 t_i，其中对所有的 i，都有 $t_i > t_{i-1}$。

另外一个控制关节处行为的参数（c）可以被合并到切线方程中[917]。然而，这需要在每个关节处引入两条切线，其中一条切线代表入射切线，记为 $\mathbf{s}_i$（source）；另外一条切线代表出射切线，记为 $\mathbf{d}_i$（destination），如图 17.14 所示。请注意，在 $\mathbf{p}_i$ 和 $\mathbf{p}_{i+1}$ 之间的曲线段上，使用了切线 $\mathbf{d}_i$ 和 $\mathbf{s}_{i+1}$。切线的计算方法如下，其中 c 是连续性（continuity）参数：

$$\begin{aligned}\mathbf{s}_i &= \frac{1-c}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{1+c}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i), \\ \mathbf{d}_i &= \frac{1+c}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{1-c}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i).\end{aligned} \tag{17.21}$$

同样地，这个连续性参数的默认值为 $c = 0$，即 $\mathbf{s}_i = \mathbf{d}_i$。如果 $c = -1$，我们会得到 $\mathbf{s}_i = \mathbf{p}_i - \mathbf{p}_{i-1}$，$\mathbf{d}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$，这会在关节处产生一个尖角，此时只满足 C^0 连续。不断增大 c 的值，会使得 $\mathbf{s}_i$ 和 $\mathbf{d}_i$ 越来越相似，当 $c = 0$，有 $\mathbf{s}_i = \mathbf{d}_i$。

当 $c = 1$ 时，我们会得到 $\mathbf{s}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$ ， $\mathbf{d}_i = \mathbf{p}_i - \mathbf{p}_{i-1}$ 。因此，这个连续性参数 c 可以给予用户更多的控制权，如果需要的话，使用这个参数可以在连接处获得一个尖角。

将张力参数、偏移参数和连续性参数组合在一起，其中默认的参数值为 $a = b = c = 0$：

$$\begin{aligned}\mathbf{s}_i &= \frac{(1-a)(1+b)(1-c)}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{(1-a)(1-b)(1+c)}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i \\ \mathbf{d}_i &= \frac{(1-a)(1+b)(1+c)}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{(1-a)(1-b)(1-c)}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i\end{aligned} \tag{17.22}$$

只有当所有的曲线段都使用相同长度的时间间隔时，方程 17.20 和方程 17.22 才有效。考虑到不同曲线段的时间长度往往会不同，因此可能还需要对切线进行调整，类似于章节 17.1.3 中所做的那样。将调整后的切线记为 $\mathbf{s}'_i$ 和 $\mathbf{d}'_i$ ，其数学表达如下，其中 $\Delta_i = t_{i+1} - t_i$：

$$\mathbf{s}'_i = \mathbf{s}_i \frac{2\Delta_i}{\Delta_{i-1} + \Delta_i} \quad \text{and} \quad \mathbf{d}'_i = \mathbf{d}_i \frac{2\Delta_{i-1}}{\Delta_{i-1} + \Delta_i} \tag{17.23}$$

17.1.6 B-样条

在这里，我们将对 B 样条（B-spline）的主题进行简要介绍，并特别关注三次均匀 B 样条。一般来说，B 样条和 Bezier 曲线十分相似，B 样条可以表示为一个 t （使用移位基函数）、 β_n （由控制点进行加权）和 c_k 的函数，例如：

$$s_n(t) = \sum_k c_k \beta_n(t-k) \tag{17.24}$$

在这种情况下，上述方程会构成一条曲线，其中参数 t 是 x 轴坐标， $s_n(t)$ 是 y 轴坐标，同时控制点只是均匀间隔的 y 值。想要了解更多内容，详见 the Killer B [111]、Farin [458]、Hoschek 和 Lasser [777]的文章。

在这里，我们将遵循 Rujters 等人[1518]的介绍方式，并给出均匀三次 B 样条的特殊情况。这个三次基函数 $\beta_3(t)$ 由三部分拼接而成：

$$\beta_3(t) = \begin{cases} 0, & |t| \geq 2 \\ \frac{1}{6}(2-|t|)^3, & 1 \leq |t| < 2 \\ \frac{2}{3} - \frac{1}{2}|t|^2(2-|t|), & |t| < 1 \end{cases} \tag{17.25}$$

图 17.15 展示了这个基函数的构造方式。这个函数每一处都具有 C^2 连续性，这意味着如果将几个 B 样条曲线拼接在一起的话，那么所形成的复合曲线也将具有 C^2 连续性。一条三次曲线具有 C^2 连续性，而一条 n 次曲线一般可以具有 C^{n-1} 连续性。一般来说，可以按照如下方式来创建一组基函数。$\beta_0(t)$ 是一个“方形（box）”函数，即：如果 $|t| < 0.5$，则 $\beta_0(t) = 1$；如果 $|t| = 0.5$，则 $\beta_0(t) = 0.5$；对于剩余的 t，有 $\beta_0(t) = 0$。下一个基函数 $\beta_1(t)$ 是通过使用 $\beta_0(t)$ 对 $\beta_0(t)$ 进行卷积得到的，它是一个“帐篷（tent）”函数。同样地，之后的基函数 $\beta_2(t)$ 是通过使用 $\beta_1(t)$ 对 $\beta_1(t)$ 进行卷积得到的，它是一个更加平滑的函数，也就是具有 C^1 连续性。重复这个过程还可以得到 C^2 连续性，以此类推。

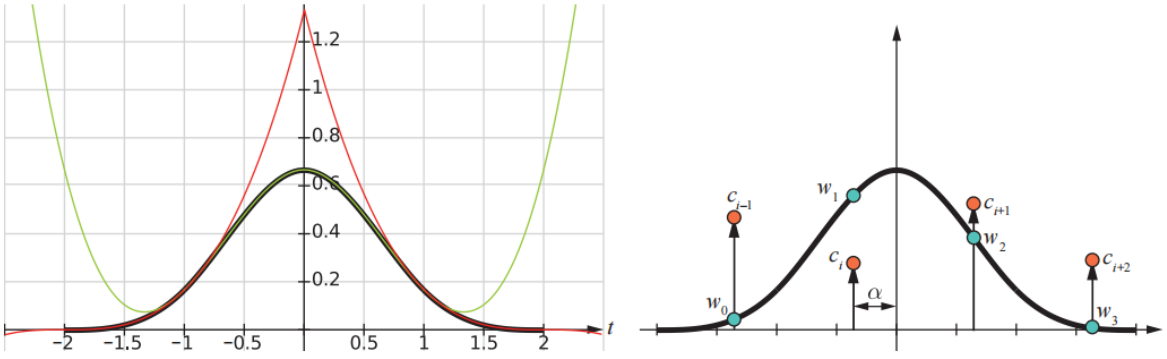


图 17.15：左：基函数 $\beta_3(t)$ 是一条很粗的黑色曲线，它由两条分段三次函数所构成（红色和绿色）。当 $|t| < 1$ 时使用绿色曲线，当 $1 \leq |t| < 2$ 时使用红色曲线，其他地方的曲线均为零。右：使用四个控制点 c_k（$k \in i-1, i, i+1, i+2$）来创建一段曲线，我们只会得到点 c_i 与 c_{i+1} 之间的那一段曲线。将 α 输入 w 函数来计算基函数，然后将这些值乘以相应的控制点，最后再将所有值加在一起，就可以得到曲线上的一个点。详见图 17.16。[1518]

曲线段的求值方法如图 17.15 右侧所示，其数学公式为：

$$s_3(i+\alpha) = w_0(\alpha)c_{i-1} + w_1(\alpha)c_i + w_2(\alpha)c_{i+1} + w_3(\alpha)c_{i+2} \tag{17.26}$$

请注意，方程 17.26 在任何时候都只会使用四个控制点，这意味着曲线具有局部支持性（local support），即只需要有限数量的控制点就可以定义一段曲线。方程 17.26 中的函数 $w_k(\alpha)$ 是使用三次基函数 $\beta_3(t)$ 进行定义的：

$$
\begin{aligned}
w_0(\alpha) &= \beta_3(-\alpha - 1), \quad & w_1(\alpha) &= \beta_3(-\alpha), \\
w_2(\alpha) &= \beta_3(1 - \alpha), & w_3(\alpha) &= \beta_3(2 - \alpha).
\end{aligned} \tag{17.27}
$$

Ruijters 等人[1518]表明，方程 17.27 可以重写为以下形式：

$$
\begin{aligned}
w_0(\alpha) &= \tfrac{1}{6}(1 - \alpha)^3, & w_1(\alpha) &= \tfrac{2}{3} - \tfrac{1}{2}\alpha^2(2 - \alpha), \\
w_2(\alpha) &= \tfrac{2}{3} - \tfrac{1}{2}(1 - \alpha)^2(1 + \alpha), & w_3(\alpha) &= \tfrac{1}{6}\alpha^3.
\end{aligned} \tag{17.28}
$$

在图 17.16 中，我们展示了将两条均匀三次 B 样条曲线拼接为一条曲线的结果。这样做的一个主要的优点在于，拼接后的曲线是连续的，它具有与基函数 $\beta(t)$ 相同的连续性，在三次 B 样条的情况下为 C^2 连续性。从图中我们可以看到，我们无法保证曲线会通过每个控制点。请注意，我们还可以为 x 坐标创建一个 B 样条，这将会在平面上给出一条一般化曲线（而不仅仅是函数）。由此产生的二维点为 $(s_3^x(i + \alpha), s_3^y(i + \alpha))$，这实际上是对方程 17.26 的两次求值，一次使用 x 进行计算，一次使用 y 进行计算。

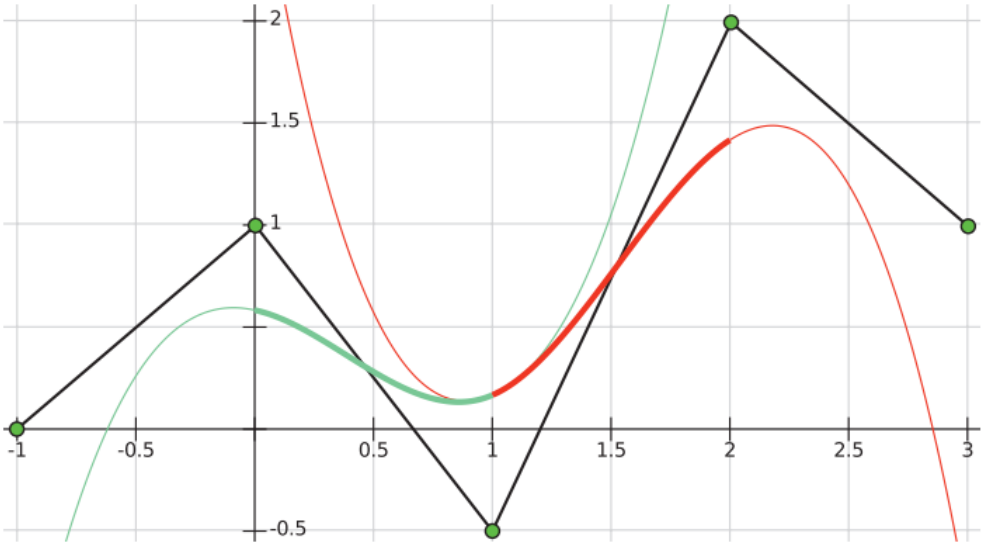


图 17.16：在这个例子中，使用 5 个控制点 c_k （绿色圆圈）定义了一个均匀三次样条曲线。其中两段加粗的曲线是分段 B 样条曲线的一部分。左侧曲线（绿色）由最左侧的四个控制点定义，右侧曲线（红色）由最右边的四个控制点定义。曲线在 $t = 1$ 处相交，具有 C^2 连续性。

我们仅仅展示了如何使用均匀的 B 样条，如果控制点之间的间距是不均匀的，那么方程将会变得更加复杂，同时控制效果也会更加灵活[111, 458, 777]。

17.2 参数化曲面

对参数化曲线的概念进行扩展，我们可以得到参数化曲面。打个比方，三角形和多边形是对线段的扩展，我们从一维的线进入到了二维的面。参数化表面可以用于对曲面物体进行建模，一个参数化表面是由少量控制点进行定义的。参数化表面的细分是一个在若干位置上计算表面表示的过程，并将它们连接起来形成三角形，从而对真实表面进行近似。这样做是因为图形硬件可以高效地渲染三角形。在运行过程中，参数化表面可以被细分成任意数量的三角形；因此，参数化曲面非常适合在质量和性能之间进行权衡，因为更多的细分三角形需要更多的时间来进行渲染，但同时能够提供更好的着色效果和轮廓外观。参数化表面的另一个优点在于，这些用于定义表面的控制点可以被动画化，然后再对表面进行细分。与直接对一个巨大的三角形网格进行动画相比，后者的开销会更大。

本小节首先会介绍 Bezier 面片（Bezier patch），它是一个具有矩形定义域的曲面，它们同样也被称为张量积 Bezier 曲面（tensor–product Bezier surface）。然后我们会介绍具有三角形定义域的 Bezier 三角形（Bezier triangle），并在章节 17.2.3 中讨论其连续性。在章节 17.2.4 和章节 17.2.5 中，我们会介绍两种方法，来将输入的三角形替换为 Bezier 三角形。这两个技术分别被称为 PN 三角形和 Phong 曲面细分。最后，在章节 17.2.6 中介绍了 B 样条面片。

17.2.1 Bezier 面片

我们在章节 17.1.1 中介绍了的 Bezier 曲线的概念，它只具有一个参数 t，实际上我们可以将其扩展到使用两个参数，这样形成的就不再是一条曲线了，而是一个曲面。我们首先会将线性插值（linear interpolation）扩展为双线性插值（bilinear interpolation）。现在，我们不再使用两个点进行插值，而是使用四个点进行插值，分别将其称为点 $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$，如图 17.17 所示。

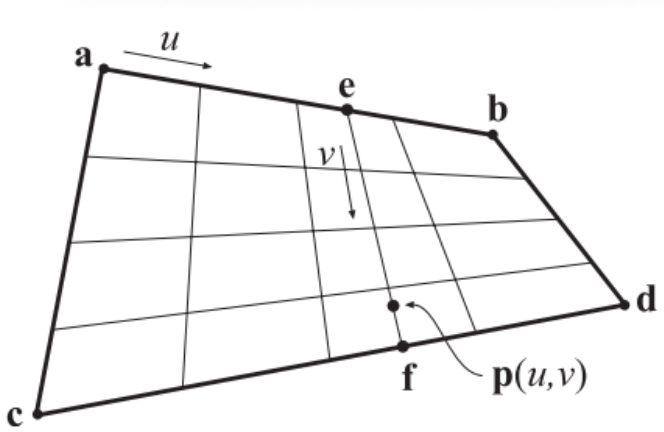


图 17.17：使用四个点进行双线性插值。

同时我们不再使用单个参数 t，而是使用两个参数 (u, v)。使用参数 u 对 $\mathbf{a}\&\mathbf{b}$，$\mathbf{c}\&\mathbf{d}$ 分别进行线性插值，从而得到点 $\mathbf{e}$ 和点 $\mathbf{f}$：

$$\mathbf{e} = (1-u)\mathbf{a} + u\mathbf{b}, \quad \mathbf{f} = (1-u)\mathbf{c} + u\mathbf{d} \tag{17.29}$$

同理，再使用参数 v 对点 $\mathbf{e}$ 和点 $\mathbf{f}$ 在另一个方向上进行线性插值，这样就得到了一个双线性插值的结果：

$$\begin{aligned}\mathbf{p}(u, v) &= (1-v)\mathbf{e} + v\mathbf{f} \\ &= (1-u)(1-v)\mathbf{a} + u(1-v)\mathbf{b} + (1-u)v\mathbf{c} + uv\mathbf{d}.\end{aligned} \tag{17.30}$$

请注意，这与用于纹理映射的双线性插值（方程 6.1）实际上是相同原理的。方程 17.30 描述了一个最简单的非平面参数化表面，使用不同的 (u, v) 参数值可以在表面上生成不同的点。其定义域（有效值集合）为 $(u, v) \in [0, 1] \times [0, 1]$，即参数 u 和 v 都位于 $[0, 1]$ 范围内。当区域为矩形时，这样得到的表面通常会称为面片（patch）。

为了从线性插值扩展到一条 Bezier 曲线，需要添加更多的控制点并进行重复线性插值，同样的策略可以用于生成面片。这里假设我们使用了 9 个点，这 9 个控制点排列在一个 3×3 的网格中，如图 17.18 所示，图中还进行了相应地标记。为了从这些控制点中构建一个双二次（biquadratic）Bezier 面片，我们首先需要进行四次双线性插值，从而创建出四个中间点，如图 17.18 所示。然后使用这四个中间点，再次进行双线性插值，从而得到最终的表面点。

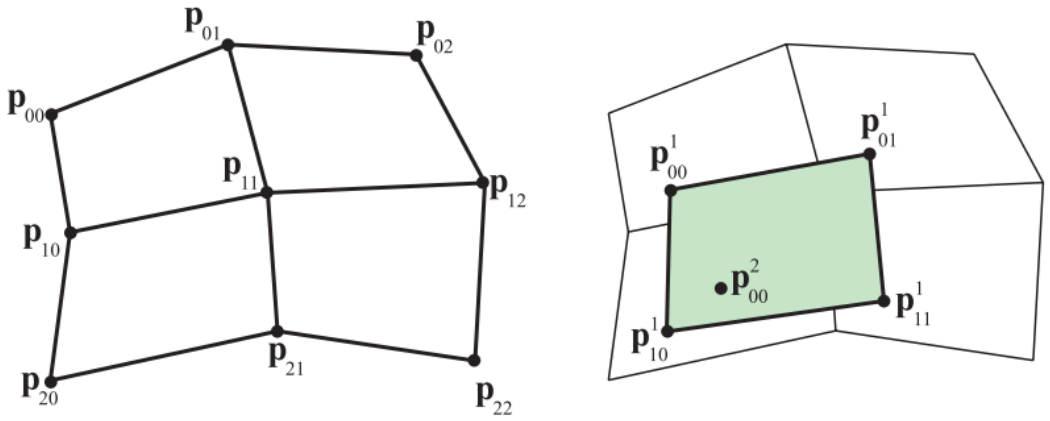


图 17.18：左：一个双二次 Bezier 表面，它由 9 个控制点 $\mathbf{p}_{ij}$ 进行定义。右：为了在 Bezier 表面上生成一个点，首先使用最近的控制点进行四次双线性插值，这样可以创建出四个点中间点 $\mathbf{p}^1_{ij}$。最后再次对这四个中间点进行双线性插值，得到最终的表面点 $\mathbf{p}(u, v) = \mathbf{p}^2_{00}$。

上面所描述的重复双线性插值，实际上是 de Casteljau 算法在面片上的扩展。这里我们需要对一些符号进行定义。表面的阶数（次数，自由度）为 n，控制点为 $\mathbf{p}_{i,j}$，其中 $i,j \in [0 \ldots n]$。因此，$(n+1)^2$ 个控制点可以构建出一个 n 次 Bezier 面片。请注意，这些原始控制点的上标应该是 0，即 $\mathbf{p}_{i,j}^0$，但是它通常会被省略；有时候我们会直接使用 $_{ij}$ 来表示下标，而不是使用 $_{i,j}$，这样可以避免混淆。使用 de Casteljau 算法的 Bezier 面片可以使用如下方程进行描述：

de Casteljau [面片]:

$$\begin{aligned}\mathbf{p}_{i,j}^k(u,v) &= (1-u)(1-v)\mathbf{p}_{i,j}^{k-1} + u(1-v)\mathbf{p}_{i,j+1}^{k-1} + (1-u)v\mathbf{p}_{i+1,j}^{k-1} + uv\mathbf{p}_{i+1,j+1}^{k-1} \\ k &= 1 \ldots n, \quad i = 0 \ldots n-k, \quad j = 0 \ldots n-k\end{aligned} \tag{17.31}$$

与 Bezier 曲线类似，Bezier 面片上 (u,v) 处的点为 $\mathbf{p}_{i,j}^n(u,v)$。Bezier 面片也可以使用 Bernstein 多项式进行描述，这被称为 Bernstein 形式，如方程 17.32 所示：

Bernstein [面片]:

$$\begin{aligned}\mathbf{p}(u,v) &= \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n} B_j^n(v)\mathbf{p}_{i,j} = \sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v)\mathbf{p}_{i,j}, \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \binom{m}{i}\binom{n}{j} u^i(1-u)^{m-i}v^j(1-v)^{n-j}\mathbf{p}_{i,j}.\end{aligned} \tag{17.32}$$

请注意，在方程 17.32 中，表面的自由度有两个参数 m 和 n。这个“复合”自由度有时候会表示为 $m \times n$。在大多数情况下都会进行一些简化，即 $m = n$。假设我们现在 $m > n$，其结果相当于是先进行 n 次双线性插值，然后再进行 $m-n$ 次线性插值，结果如图 17.19 所示。

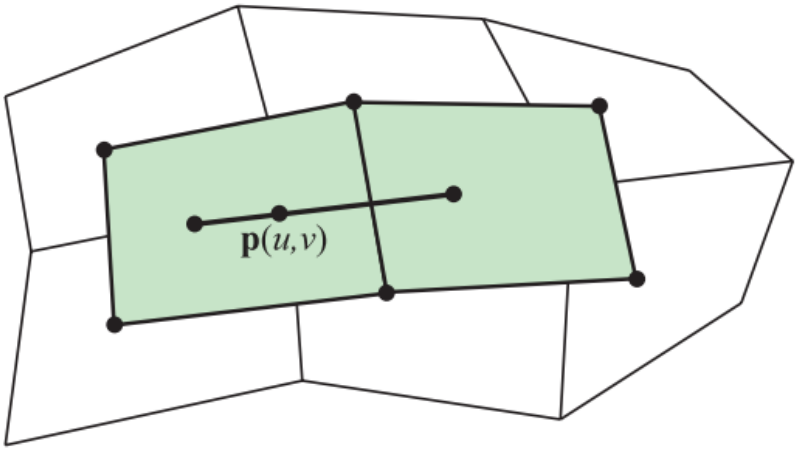


图 17.19：在不同方向上具有不同的自由度。

还可以对方程 17.32 进行改写，从而构建另一种理解方式：

$$\mathbf{p}(u,v) = \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n} B_j^n(v)\mathbf{p}_{i,j} = \sum_{i=0}^{m} B_i^m(u)\mathbf{q}_i(v) \quad (17.33)$$

其中：

$$\mathbf{q}_i(v) = \sum_{j=0}^{n} B_j^n(v)\mathbf{p}_{i,j}, \ \ i = 0 \ldots m$$

从这两个方程中我们可以看出，当我们固定一个 v 值不变时，这实际上就是一条 Bezier 曲线。我们假设 $v = 0.35$，则可以从 Bezier 曲线中计算出点 $\mathbf{q}_i(0.35)$，而方程 17.33 实际上就描述了当 $v = 0.35$ 时，这个 Bezier 表面上的一条 Bezier 曲线。

接下来，我们将介绍 Bezier 面片的一些有用属性。通过将参数 $(u,v) = (0,0)$，$(u,v) = (0,1)$，$(u,v) = (1,0)$，$(u,v) = (1,1)$ 带入到方程 17.32 中，可以很容易地证明 Bezier 面片会穿过这些拐角处的控制点，即点 $\mathbf{p}_{0,0}$，$\mathbf{p}_{0,n}$，$\mathbf{p}_{n,0}$ 和 $\mathbf{p}_{n,n}$。此外，这个面片的每条边界都由对应边界控制点形成的 n 次 Bezier 曲线所定义。因此，这些拐角控制点处的切线，同样也由这些边界 Bezier 曲线所定义。每个拐角控制点都有两条切线，分别位于 u 方向上和 v 方向上。与 Bezier 曲线的情况一样，这个 Bezier 面片同样也位于其控制点所形成的凸包内部，并且：

$$\sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v) = 1 \tag{17.34}$$

其中 $(u, v) \in [0, 1] \times [0, 1]$ 。最后，先对这些控制点进行旋转，然后再在面片上生成新的点；与先在面片上生成点，然后再对这些点进行旋转，二者在数学上是完全相同的，但是通常前者的操作会更快。

对方程 17.32 求偏导（偏微分，partially differentiate）[458]，可以得到：

Derivatives [面片]:

$$\begin{aligned}\frac{\partial \mathbf{p}(u,v)}{\partial u} &= m\sum_{j=0}^{n}\sum_{i=0}^{m-1} B_i^{m-1}(u)B_j^n(v)\left[\mathbf{p}_{i+1,j} - \mathbf{p}_{i,j}\right] \\ \frac{\partial \mathbf{p}(u,v)}{\partial v} &= n\sum_{i=0}^{m}\sum_{j=0}^{n-1} B_i^m(u)B_j^{n-1}(v)\left[\mathbf{p}_{i,j+1} - \mathbf{p}_{i,j}\right].\end{aligned} \tag{17.35}$$

从方程 17.35 中可以看出，面片的自由度在被微分的方向上减少了 1。根据方程 17.35 计算表面的偏导数，可以计算得到非归一化的表面法线：

$$\mathbf{n}(u, v) = \frac{\partial \mathbf{p}(u, v)}{\partial u} \times \frac{\partial \mathbf{p}(u, v)}{\partial v} \tag{17.36}$$

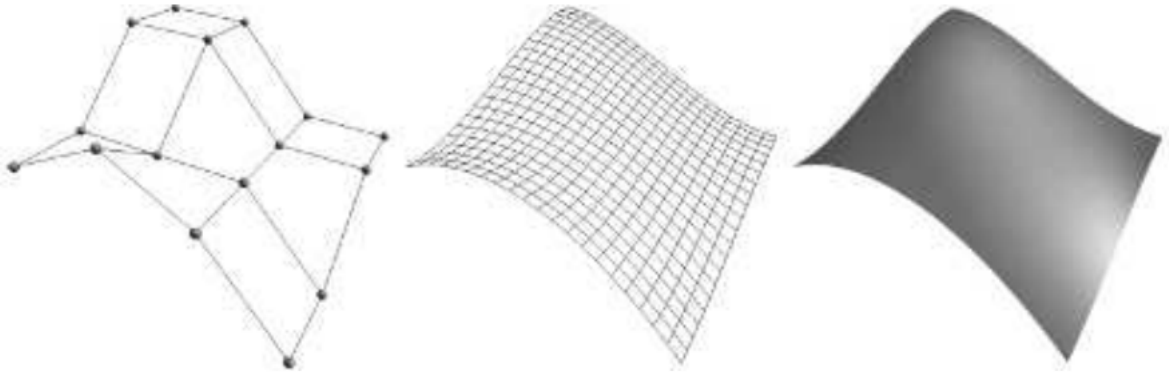


图 17.20：左：一个 4×4 的控制网格，可以构建一个 3×3 自由度的 Bezier 面片。中间：展示了在这个 Bezier 表面上生成的实际四边形网格。右：对这个 Bezier 面片进行着色。

图 17.20 展示了实际的 Bezier 面片与相应的控制网格。图 17.21 展示了移动控制点所产生的效果。

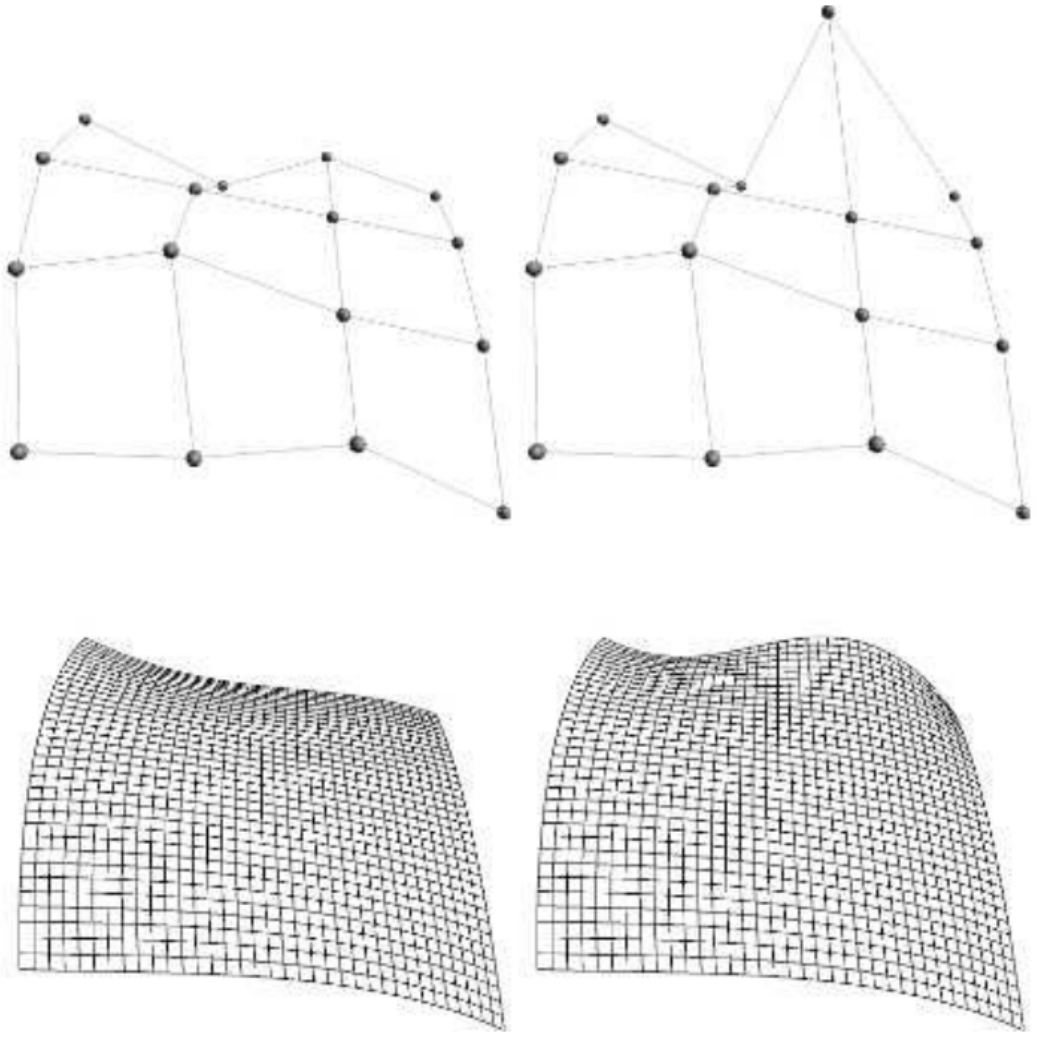


图 17.21：这组图像展示了在移动一个控制点的时候，Bezier 面片会发生什么变化。大部分变化都集中在被移动控制点的附近。

有理 Bezier 面片

一条 Bezier 曲线可以被扩展为一条有理 Bezier 曲线（章节 17.1.1），从而引入更多的自由度。同理，一个 Bezier 面片也可以被扩展为一个有理 Bezier 面片：

$$\mathbf{p}(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j} B_i^m(u) B_j^n(v) \mathbf{p}_{i,j}}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j} B_i^m(u) B_j^n(v)} \tag{17.37}$$

有关这种类型面片的更多信息，请参阅 Farin 的书籍[458]，以及 Hochek 和 Lasser 的书籍[777]。类似地，有理 Bezier 三角形是对 Bezier 三角形的扩展，我们将在下一小节中进行讨论。

17.2.2 Bezier 三角形

虽然通常三角形会被认为是比矩形更加简单的几何图元，但是当涉及到 Bezier 曲面的时候，情况却并非如此：Bezier 三角形并不像 Bezier 面片那么简单。但是这种类型的面片仍然是值得进行研究和表示的，因为它可以用于形成 PN 三角形和 Phong 曲面细分，这是两种快速且简单的算法。请注意，一些常见游戏引擎，例如虚幻引擎、Unity 引擎和 Lumberyard 引擎，都支持 Phong 曲面细分和 PN 三角形。

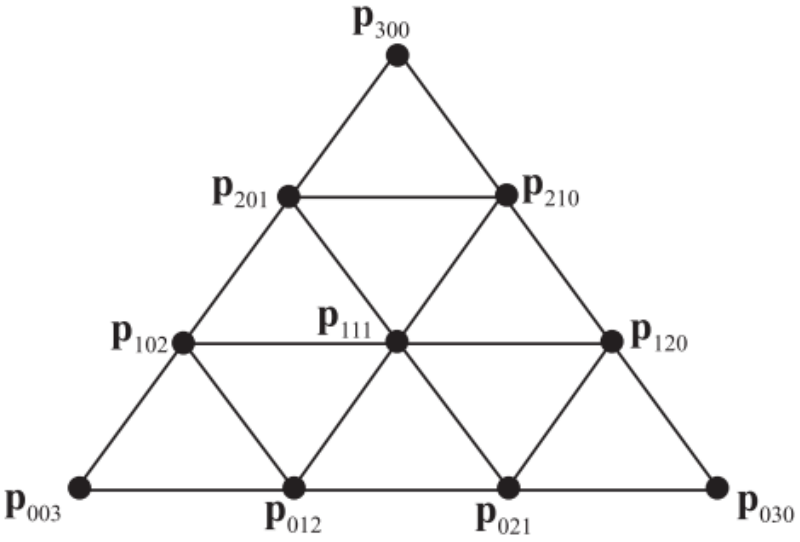


图 17.22：三次 Bezier 三角形的控制点网格。

如图 17.22 所示，Bezier 三角形的控制点位于一个三角形网格内部。Bezier 三角形的自由度为 n，这意味着每条边上都有 $n+1$ 个控制点。我们将这些控制点记为 $\mathbf{p}^0_{i,j,k}$，有时也缩写为 $\mathbf{p}^0_{ijk}$。请注意，$i+j+k=n$，并且所有控制点的下标都要满足 $i,j,k \geq 0$。因此，控制点的总数为：

$$\sum_{x=1}^{n+1} x = \frac{(n+1)(n+2)}{2} \tag{17.38}$$

毫无疑问，Bezier 三角形也是基于重复插值的。然而，由于其定义域的形状是一个三角形，这里必须使用重心坐标（章节 22.8）来进行插值。回想一下，现在有一个位于三角形 $\Delta\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$ 内部的一个点，它可以表示为：

$$\begin{aligned}\mathbf{p}(u,v) &= \mathbf{p}_0 + u\left(\mathbf{p}_1 - \mathbf{p}_0\right) + v\left(\mathbf{p}_2 - \mathbf{p}_0\right) \\ &= (1-u-v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2\end{aligned}$$

其中 (u,v) 就是重心坐标。对于三角形内的点，必须保证：

$$u \geq 0, v \geq 0$$
$$1-(u+v) \geq 0 \Leftrightarrow u+v \leq 1$$

在此基础上，Bezier 三角形的 de Casteljau 算法为：

de Casteljau [三角形]:

$$\begin{aligned} \mathbf{p}_{i,j,k}^{l}(u,v) &= u\mathbf{p}_{i+1,j,k}^{l-1} + v\mathbf{p}_{i,j+1,k}^{l-1} + (1-u-v)\mathbf{p}_{i,j,k+1}^{l-1}, \\ l &= 1 \ldots n, \quad i+j+k = n-l. \end{aligned} \tag{17.39}$$

Bezier 三角形在 (u,v) 处的最终点为 $\mathbf{p}_{000}^{n}(u,v)$ 。Bernstein 形式的 Bezier 三角形为：

Bernstein [三角形]:

$$\mathbf{p}(u,v) = \sum_{i+j+k=n} B_{ijk}^{n}(u,v)\mathbf{p}_{ijk} \tag{17.40}$$

现在 Bernstein 多项式的计算依赖于两个参数 u 和 v ，因此计算方式有所不同，如下所示：

$$B_{ijk}^{n}(u,v) = \frac{n!}{i!j!k!}u^{i}v^{j}(1-u-v)^{k}, \quad i+j+k=n \tag{17.41}$$

Bernstein 形式的 Bezier 三角形的偏导数为[475]：

Derivatives [三角形]:

$$\begin{aligned} \frac{\partial \mathbf{p}(u,v)}{\partial u} &= \textstyle\sum_{i+j+k=n-1} nB_{ijk}^{n-1}(u,v)\left(\mathbf{p}_{i+1,j,k} - \mathbf{p}_{i,j,k+1}\right), \\ \frac{\partial \mathbf{p}(u,v)}{\partial v} &= \textstyle\sum_{i+j+k=n-1} nB_{ijk}^{n-1}(u,v)\left(\mathbf{p}_{i,j+1,k} - \mathbf{p}_{i,j,k+1}\right). \end{aligned} \tag{17.42}$$

与 Bezier 面片一样，Bezier 三角形同样具有一些类似的特性，这并不令人惊讶，例如：Bezier 三角形会穿过三个拐角控制点；并且每条边界都是一条 Bezier 曲线，并由该边界上的控制点所定义；同时，该 Bezier 三角形位于控制点形成的凸包内部。图 17.23 展示了一个 Bezier 三角形。

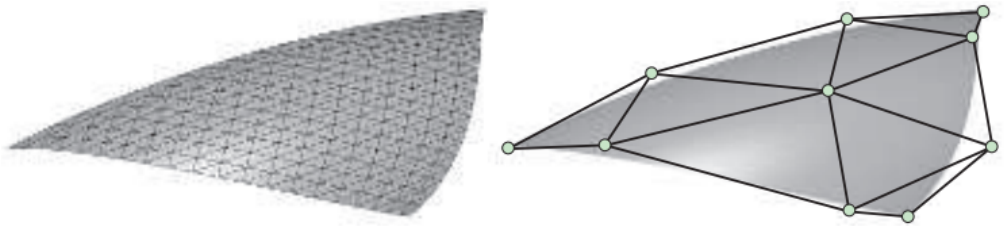


图 17.23：左：Bezier 三角形的线框图。右：Bezier 三角形的着色结果与控制点网格。

17.2.3 连续性

当使用 Bezier 曲面来构建一个复杂物体时，人们通常都想要将几个不同的 Bezier 曲面拼接在一起，从而形成一个复合表面。为了得到一个好看（平滑过度）的结果，我们必须要确保表面上能够获得合理的连续性。这一点与章节 17.1.3 中的曲线是一样的。

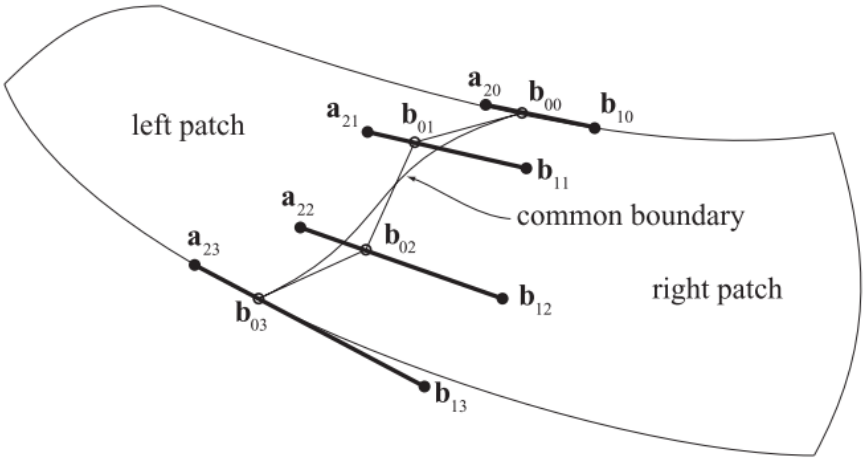


图 17.24：图中展示了如何将两个具有 C^1 连续的 Bezier 面片缝合在一起。其中，加粗线条上的所有控制点都必须共线，并且每对线段的长度之间都必须具有相同的比例。请注意，还需要满足条件 $\mathbf{a}_{3j} = \mathbf{b}_{0j}$，这样才能获得面片之间的共享边界，这一点也可以在图 17.25 中看到。

假设我们想要将两个双立方（bicubic）Bezier 面片应该拼接在一起。其中每个 Bezier 面片都有 4×4 个的控制点。如图 17.24 所示，其中左侧面片具有控制点 $\mathbf{a}_{ij}$，右侧具有控制点 $\mathbf{b}_{ij}$，其中 $0 \leq i, j \leq 3$。首先为了保证 C^0 连续性，每个面片在边界上必须共享相同的控制点，也就是说，两侧面片对应的控制点需要重合，即 $\mathbf{a}_{3j} = \mathbf{b}_{0j}$。

然而，仅仅保证 C^0 的连续性，这还不足以获得一个好看的复合表面。这里我们将介绍一种简单的技术，它可以保证 C^1 的连续性[458]。为了实现这一点，我们必须对

靠近共享边界的两行控制点的位置进行约束。这两行控制点分别是 $\mathbf{a}_{2j}$ 和 $\mathbf{b}_{1j}$，对于 $j \in [0,3]$，点 $\mathbf{a}_{2j}$、点 $\mathbf{b}_{0j}$、点 $\mathbf{b}_{1j}$ 必须要保持共线，也就是说它们会位于同一条直线上。并且，它们之间的长度比值也必须相同，即 $\|\mathbf{a}_{2j} - \mathbf{b}_{0j}\| = k\,\|\mathbf{b}_{0j} - \mathbf{b}_{1j}\|$。这里的比例系数 k 是一个常数，对所有 j 都必须相等。如图 17.24 和图 17.25 所示。

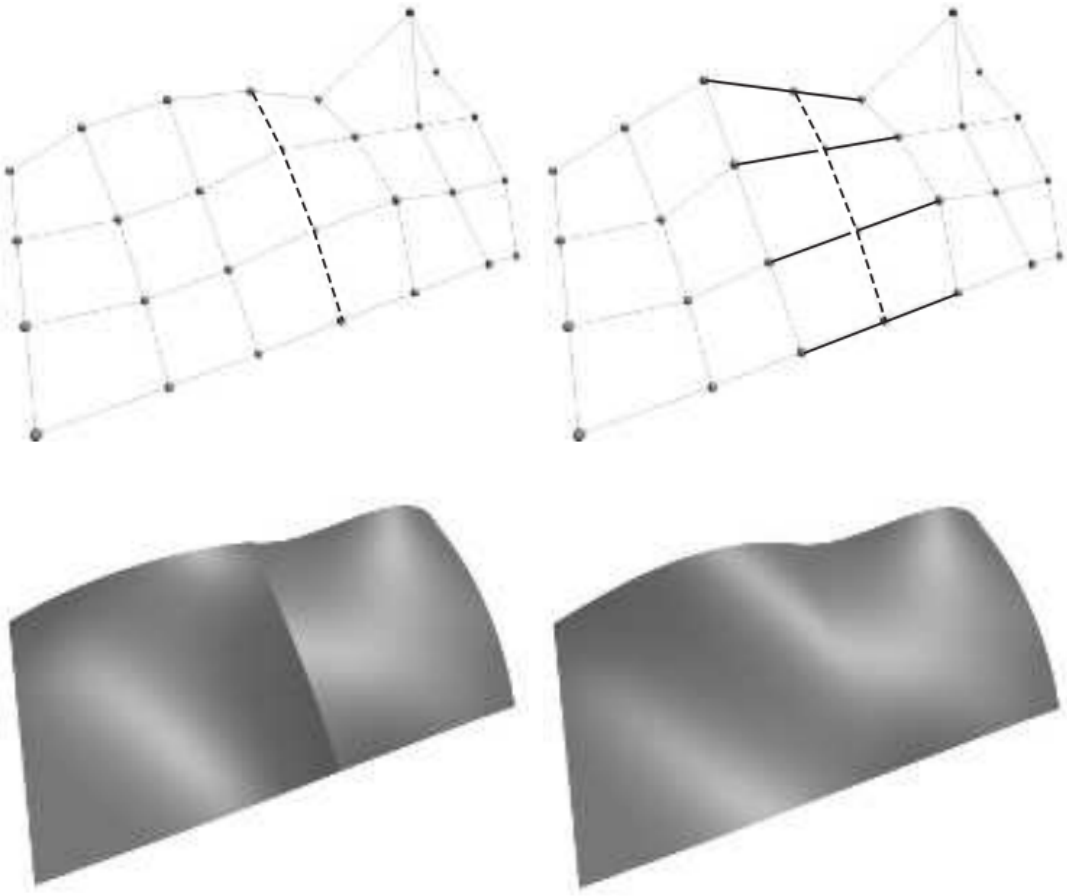


图 17.25：左侧展示了两个拼接的 Bezier 面片，它们之间只满足 C^0 连续。我们可以很明显地观察到，在这些面片之间存在着色不连续的情况。右边展示了一组相似的拼接面片，它们满足 C^1 连续性，看起来表现更好。在第一行中，虚线代表了两个拼接面片之间的共享边界。在右上角中，加粗黑线代表了拼接面片两侧的控制点，它们需要位于同一条直线上。

这种构造方式消耗了许多设置控制点的自由度，当我们将四个面片（共用一个拐角点）拼接在一起时，可以更加清楚地看到这一点。图 17.26 展示了这个构造过程，图的右侧展示了构造的结果，并展示了共享控制点（公共拐角点）周围的 8 个控制点位置。这 9 个点必须位于在同一个平面上，并且这 9 个点本身还必须能够形成一个双线性面片，如图 17.17 所示。如果想要让这个拐角点满足 G^1 连续性（只在这个拐角点处满足），则令这个 9 个控制点共面即可。这样不会损失那么多的自由度。

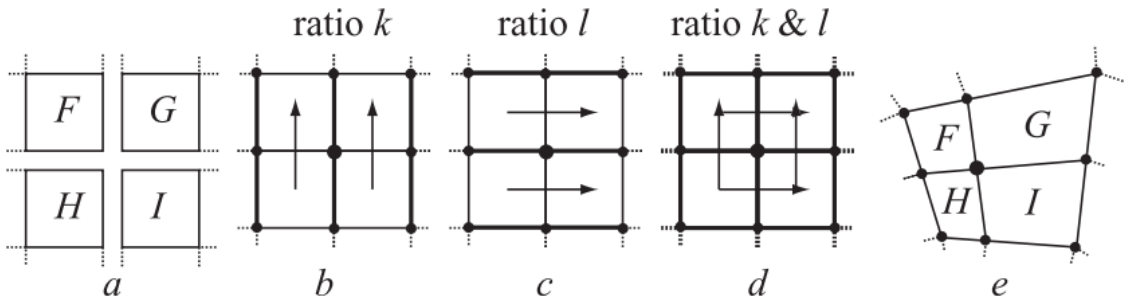


图 17.26：（a）将 F 、 G 、 H 、 I 四个面片拼接在一起，所有面片共用同一个拐角。（b）在竖直方向上，这三组点（位于加粗黑线上）必须具有相同的比例 k ；（b）中没有显示出这种关系，请观察最右边的图。（c）与（b）类似，在水平方向上，这两个面片上的每组控制点都必须具有相同的比例 l 。（d）将这四个面片缝合在一起时，它们在垂直方向上必须具有同一个比例 k ，在水平方向上必须具有同一个比例 l 。（e）展示了这样做之后的结果，其中最接近（包括）共享控制点的 9 个控制点，它们具有正确计算的比例。

Bezier 三角形的连续性通常要更加复杂，Bezier 面片和 Bezier 三角形的 G^1 连续性也十分复杂[458, 777]。当构造一个具有许多 Bezier 曲面的复杂对象时，通常很难保证它在所有边界上都获得合理的连续性。为了解决这个问题，一种方法是转向使用细分曲面，我们将在章节 17.5 中进行讨论。

请注意，想要获得一个良好的跨边界纹理外观， C^1 连续是必需的。对于反射和着色而言，在 G^1 连续的条件下就能够获得较为合理的结果。如果满足 C^1 或者更高的连续性，则可以获得更好的结果。图 17.25 展示了这样的一个例子。

在接下来的两个小节中，我们将会介绍两种方法，它们利用三角形的顶点法线，来将每个输入的三角形（平面）转换为一个 Bezier 三角形。

17.2.4 PN 三角形

给定一个具有逐顶点法线的三角形网格，Vlachos 等人[1819]提出了一个叫做 PN 三角形的方案，其目标是构建一个比仅仅使用三角形更加美观的表面。其中的字母“PN”是“Point and Normal”的缩写，因为这是生成曲面所需要使用到的关键数据是顶点位置和顶点法线，它们有时也称为 N-面片（N-patch）。这个方案试图通过为每个三角形都创建一个替代曲面，从而来改善三角形网格的着色效果和轮廓外观。相关的曲面细分硬件能够动态地生成每个表面，因为曲面细分是根据每个三角形的顶点和法线生成的，并不需要相邻图元的信息，图 17.27 展示了这样的一个例子。本文所介绍的算法基于 van Overveld 和 Wyvill 的工作[1341]。

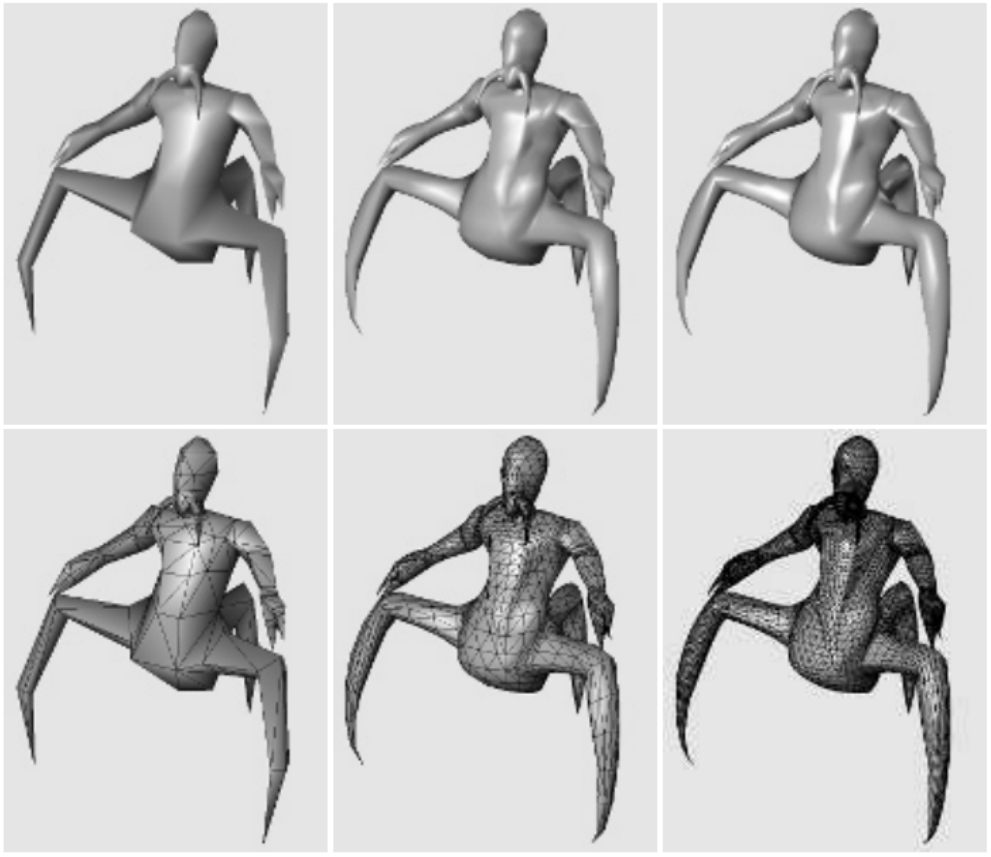


图 17.27：每一列都是同一模型的不同 LOD。左侧是原始的三角形数据，它由 414 个三角形所组成。中间的模型包含 3726 个三角形，右边的模型包含 20286 个三角形，它们都是使用本小节中介绍的算法生成的。请着重注意模型轮廓和着色效果是如何被改善的。第二行是对应的线框模型，每个原始三角形都会生成相同数量的子三角形。

假设现在我们有一个三角形，三个顶点分别是 $\mathbf{p}_{300}$ ， $\mathbf{p}_{030}$ ， $\mathbf{p}_{003}$ ，对应的三条法线分别是 $\mathbf{n}_{200}$ ， $\mathbf{n}_{020}$ ， $\mathbf{n}_{002}$ 。最基本思想是使用这些信息，为每个原始三角形都创建一个三次 Bezier 三角形，并根据这个 Bezier 三角形，从中生成任意数量的三角形。

为了简化表示，我们设 $w = 1 - u - v$ 。一个三次 Bezier 三角形的定义如下：

$$\begin{aligned}\mathbf{p}(u,v) &= \sum_{i+j+k=3} B^3_{ijk}(u,v)\mathbf{p}_{ijk} \\ &= u^3\mathbf{p}_{300} + v^3\mathbf{p}_{030} + w^3\mathbf{p}_{003} + 3u^2v\mathbf{p}_{210} + 3u^2w\mathbf{p}_{201} \\ &\quad + 3uv^2\mathbf{p}_{120} + 3v^2w\mathbf{p}_{021} + 3vw^2\mathbf{p}_{012} + 3uw^2\mathbf{p}_{102} + 6uvw\mathbf{p}_{111}.\end{aligned} \tag{17.43}$$

如图 17.22 所示。为了确保两个 PN 三角形边界处的 C^0 连续性，可以根据角控制点和角法线来确定边界上的控制点。（假设相邻三角形之间共享法线）。

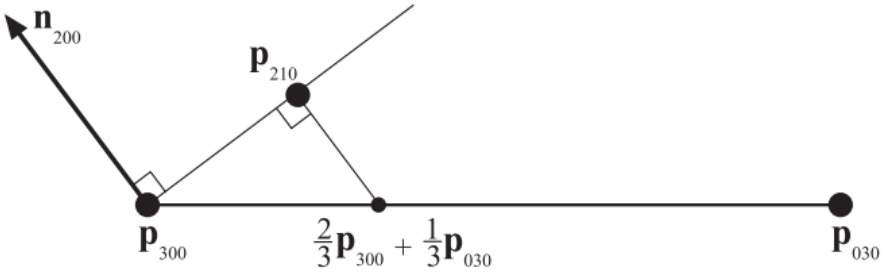


图 17.28：如何使用控制点 $\mathbf{p}_{300}$ 处法线 $\mathbf{n}_{200}$，以及两个角控制点 $\mathbf{p}_{300}$ 和 $\mathbf{p}_{030}$，来计算边界上的 Bezier 控制点 $\mathbf{p}_{210}$。

假设我们想要使用控制点 $\mathbf{p}_{300}$， $\mathbf{p}_{030}$ 以及点 $\mathbf{p}_{300}$ 处的法线 $\mathbf{n}_{200}$，来计算边界控制点 $\mathbf{p}_{210}$，如图 17.28 所示。简单地取点 $\frac{2}{3}\mathbf{p}_{300} + \frac{1}{3}\mathbf{p}_{030}$，并将其投影到由点 $\mathbf{p}_{300}$ 和法线 $\mathbf{n}_{200}$ 所定义的切平面上[457, 458, 1819]。假设这里采用都是归一化法线，那么点 $\mathbf{p}_{210}$ 的计算结果为：

$$\mathbf{p}_{210} = \frac{1}{3}\left(2\mathbf{p}_{300} + \mathbf{p}_{030} - \left(\mathbf{n}_{200} \cdot (\mathbf{p}_{030} - \mathbf{p}_{300})\right)\mathbf{n}_{200}\right) \tag{17.44}$$

其他的边界控制点都可以按照类似方法来进行计算。下面我们还需要计算内部的控制点 $\mathbf{p}_{111}$，它可以使用下面的方程进行计算，它遵循一个二次多项式[457, 458]：

$$\mathbf{p}_{111} = \frac{1}{4}\left(\mathbf{p}_{210} + \mathbf{p}_{120} + \mathbf{p}_{102} + \mathbf{p}_{201} + \mathbf{p}_{021} + \mathbf{p}_{012}\right) - \frac{1}{6}\left(\mathbf{p}_{300} + \mathbf{p}_{030} + \mathbf{p}_{003}\right) \tag{17.45}$$

我们可以通过方程 17.42 来计算表面上的两条切线，并根据切线计算表面法线。但是 Vlachos 等人[1819]没有这样做，而是选择对已有法线进行二次插值，如下所示：

$$\begin{aligned}\mathbf{n}(u,v) &= \sum_{i+j+k=2} B^2_{ijk}(u,v)\mathbf{n}_{ijk} \\ &= u^2\mathbf{n}_{200} + v^2\mathbf{n}_{020} + w^2\mathbf{n}_{002} + 2\left(uv\mathbf{n}_{110} + uw\mathbf{n}_{101} + vw\mathbf{n}_{011}\right).\end{aligned} \tag{17.46}$$

这可以被认为是一个二阶 Bezier 三角形，其中 6 个控制点就是 6 个不同的法线。在方程 17.46 中，所使用的次数是二次，这是很自然的，因为导数的次数要比实际的 Bezier 三角形低一次；同时简单的线性插值无法描述一个弯曲变化的法线，如图 17.29 所示。

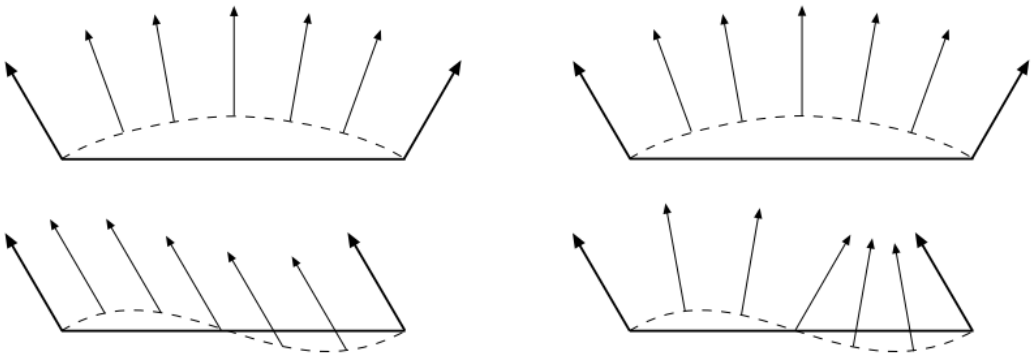


图 17.29：这幅图说明了为什么需要对法线进行二次插值，以及为什么仅仅使用线性插值是不够的。左边一列展示了使用线性插值的情况。当被插值的法线描述一个凸表面时（左上），线性插值表现良好；但是当表面出现一个弯曲变化时（左下），线性插值就失效了。右边一列展示了二次插值的情况。[1342]

为了能够使用方程 17.46，我们还需要计算法线控制点 $\mathbf{n}_{110}$ 、 $\mathbf{n}_{101}$ 和 $\mathbf{n}_{011}$ 。一种直观但有缺陷的解决方案是，直接使用 $\mathbf{n}_{200}$ 和 $\mathbf{n}_{020}$ （原始三角形顶点的法线）的平均值来计算 $\mathbf{n}_{110}$ 。然而，当 $\mathbf{n}_{200} = \mathbf{n}_{020}$ 的时候，就会遇到图 17.29 左下角所示的问题。正确的构造方法是，先获取法线 $\mathbf{n}_{200}$ 和 $\mathbf{n}_{020}$ 的平均值，然后在平面 π 上反射这个法线，如图 17.30 所示。平面 π 的法线与控制点 $\mathbf{p}_{300}$ 和 $\mathbf{p}_{030}$ 之间的差向量平行。由于这个法线只会在平面 π 上被反射，也就是说，法线与平面上的位置无关，因此我们可以假设这个平面穿过原点。另外请注意，每个法线都应当被归一化。使用数学方程进行描述，法线 $\mathbf{n}_{110}$ 的非归一化版本可以表示为[1819]：

$$\mathbf{n}'_{110} = \mathbf{n}_{200} + \mathbf{n}_{020} - 2\frac{(\mathbf{p}_{030} - \mathbf{p}_{300}) \cdot (\mathbf{n}_{200} + \mathbf{n}_{020})}{(\mathbf{p}_{030} - \mathbf{p}_{300}) \cdot (\mathbf{p}_{030} - \mathbf{p}_{300})} (\mathbf{p}_{030} - \mathbf{p}_{300}) \tag{17.47}$$

最初，van Overveld 和 Wyvill 使用系数 $3/2$ 来代替方程 17.47 中的 2 。从最终生成的图像上来看，很难判断到底使用哪个值比较好，但是使用系数 2 符合平面上的真实反射规律。

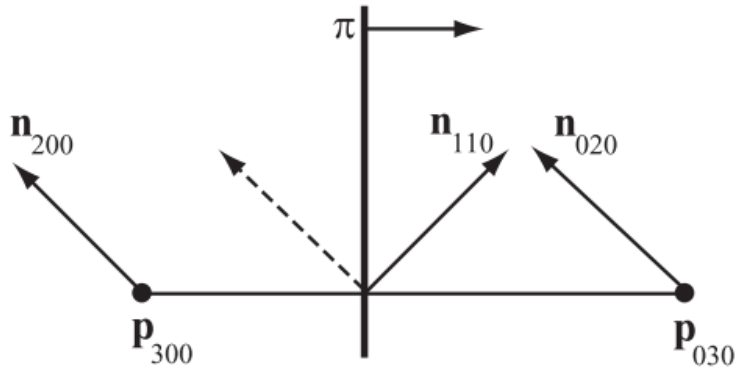


图 17.30：为 PN 三角形构造法线 $\mathbf{n}_{110}$ 。图中的虚线代表了 $\mathbf{n}_{200}$ 和 $\mathbf{n}_{020}$ 的平均法线，而正确构造的 $\mathbf{n}_{110}$ 则是这个平均法线在平面 π 上的反射结果。这个平面 π 的法线与 $\mathbf{p}_{030} - \mathbf{p}_{300}$ 平行。

至此，这个三次 Bezier 三角形的所有 Bezier 控制点，以及二次插值的所有法线都已经计算完成了。只需要在这个 Bezier 三角形上创建三角形，让它们可以被渲染即可。这种方法的优点在于，表面能够以一个相对较低的成本，获得更好的轮廓和形状。

下面是一种指定细节级别的方法。可以将原始的三角形数据认为是 LOD 0，随着三角形边界上新引入的顶点数量不断增加，LOD 数也会相应增长。因此可以这样认为，LOD 1 在三角形的每条边上引入了一个新顶点，从而在这个 Bezier 三角形上创建四个子三角形；而 LOD 2 则在每条边上引入两个新顶点，从而生成了九个子三角形。以此类推，LOD n 便生成 $(n+1)^2$ 个子三角形。为了防止 Bezier 三角形之间出现裂缝，网格中的每个三角形都必须使用相同的 LOD 级别进行细分。实际上这是一个严重的缺点，因为那些很小的三角形也会像大三角形一样被细分。可以使用自适应曲面细分（章节 17.6.2）和分数曲面细分（章节 17.6.1）等技术来避免这些问题。

PN 三角形的一个问题在于，难以控制折痕的生成，通常需要在折痕附近插入额外的三角形。虽然这些 Bezier 三角形之间的连续性只有 C^0 ，但是在许多情况下，它们看起来都还可以接受。这主要是因为三角形之间的法线是连续的，因此这一组 PN 三角形模拟了一个 G^1 连续的表面。Boubekeur 等人[181]提出了一种更好的解决方案，即一个顶点可以同时拥有两条法线，两个这样的顶点相连接，便可以生成一条折痕边缘。需要注意的是，如果想要获得表现良好的纹理效果，则三角形（或者面片）之间的边界需要满足 C^1 连续性。同样值得了解的是，如果两个相邻的三角形之间不共享相同的法线，那么就会出现裂缝。Grun [614]描述了一种方法，可以进一步提高 PN 三角形的连续性质量。Dyken 等人[401]提出了一种受到 PN 三角形启发的技术，在该技术中，只有被观察者所看到的轮廓（silhouette）才会被自适应细分，因此会变得

更加弯曲，这些 silhouette 曲线的推导方法与 PN 三角形曲线相似。为了获得平滑的过渡效果，他们在粗糙轮廓和细分轮廓之间进行了混合。为了对连续性进行改善，Funzig 等人[505]提出了 PNG1 三角形，它是对 PN 三角形的改进，可以保证处处都满足 G^1 连续性。McDonald 和 Kilgard [1164]提出了 PN 三角形的另一种扩展方法，它可以对相邻三角形上的不同法向量进行处理。

17.2.5 Phong 曲面细分

Boubekeur 和 Alexa [182]提出了一种叫做 Phong 曲面细分（Phong tessellation）的表面构造，它与 PN 三角形有许多相似之处，但是它的计算速度更快，实现更加简单。这里我们将基底三角形的顶点命名为 $\mathbf{p}_0$ ， $\mathbf{p}_1$ 和 $\mathbf{p}_2$ ，对应的归一化法线分别为 $\mathbf{n}_0$ ， $\mathbf{n}_1$ 和 $\mathbf{n}_2$ 。首先，回顾一下，三角形上的一个点，其重心坐标为 (u, v) ，那么它的实际坐标为：

$$\mathbf{p}(u, v) = (u, v, 1 - u - v) \cdot (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2) \tag{17.48}$$

在 Phong 着色中，法线会在整个平面三角形上进行插值，同样是使用上方程 17.48 完成的，不同之处在于使用法线来代替了顶点。Phong 曲面细分尝试使用重复插值，来创建一个 Phong 着色法线插值的几何版本，最终会生成一个 Bezier 三角形，图 17.31 展示了这个过程。

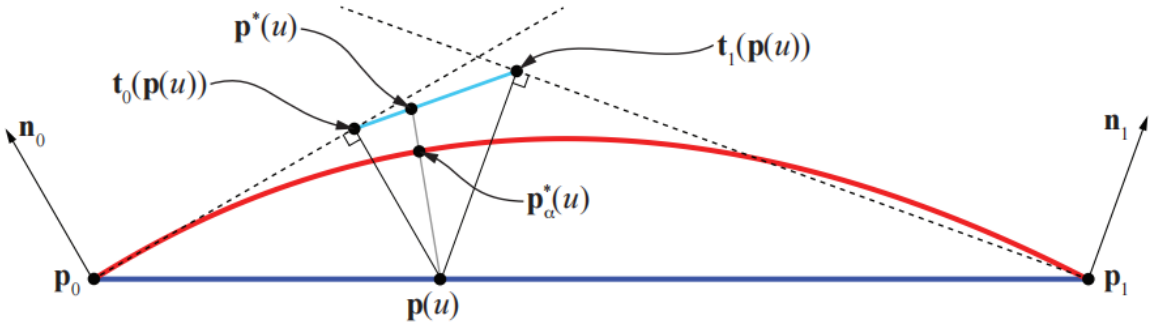


图 17.31：这里我们使用一条曲线（而不是曲面）来说明 Phong 曲面细分的结构，这也意味着 $\mathbf{p}(u)$ 只是 u 的函数，而不是 (u, v) 的函数，对于函数 $\mathbf{t}_i$ 也是如此。 $\mathbf{p}(u)$ 首先会被投影到两个切平面上，从而生成了点 $\mathbf{t}_0$ 和点 $\mathbf{t}_1$ 。然后再对点 $\mathbf{t}_0$ 和点 $\mathbf{t}_1$ 进行线性插值，从而生成 $\mathbf{p}^*(u)$ 。最后，使用一个形状因子 α ，来将基底三角形和 $\mathbf{p}^*(u)$ 进行混合。在这个例子中，我们使用了 $\alpha = 0.75$ 。

第一步是创建一个函数，来将基底三角形上的点 $\mathbf{q}$ 投影到一个切平面上，这个平面由一个顶点和一个法线所定义。这个函数的数学形式如下：

$$\mathbf{t}_i(\mathbf{q}) = \mathbf{q} - ((\mathbf{q} - \mathbf{p}_i) \cdot \mathbf{n}_i)\,\mathbf{n}_i \tag{17.49}$$

这里我们不再使用三角形的顶点来执行线性插值（方程 17.48），而是使用这个函数 $\mathbf{t}_i$ 来完成线性插值，其结果为：

$$\mathbf{p}^*(u,v) = (u, v, 1-u-v) \cdot (\mathbf{t}_0(u,v), \mathbf{t}_1(u,v), \mathbf{t}_2(u,v)) \tag{17.50}$$

为了增加一些控制灵活性，因此在基底三角形和方程 17.50 之间添加了一个形状因子 α，从而得到 Phong 曲面细分的最终公式：

$$\mathbf{p}^*_\alpha(u,v) = (1-\alpha)\mathbf{p}(u,v) + \alpha\mathbf{p}^*(u,v) \tag{17.51}$$

其中 $\alpha = 0.75$ 是一个比较推荐的值[182]。想要生成这样的一个表面，所需的唯一信息就是基底三角形的顶点和法线，以及用户提供的形状因子 α，这使得该表面的计算速度很快。最终得到的三角形路径是二次的，这个次数要低于 PN 三角形（三次）。其中三角形的表面法线是通过简单线性插值而生成的，就像是标准的 Phong 着色所做的那样。图 17.32 展示了 Phong 曲面细分应用到网格上的效果。

图 17.32：对这个怪物蛙模型使用了 Phong 曲面细分。从左到右分别是：基础网格的平面着色；基础网格与 Phong 着色；对基础网格应用 Phong 曲面细分。请注意轮廓的改进程度。在这个例子中，我们使用了 $\alpha = 0.6$ 。

17.2.6 B–样条曲面

我们在章节 17.1.6 中简要介绍了 B 样条曲线，这里我们将同样对 B 样条曲面进行介绍。我们对方程 17.24 进行推广，可以得到 B 样条面片：

$$\mathbf{s}_n(u,v) = \sum_k \sum_l \mathbf{c}_{k,l}\beta_n(u-k)\beta_n(v-l) \tag{17.52}$$

方程 17.52 与 Bezier 面片的方程 17.32 非常相似。请注意，$\mathbf{s}_n(u,v)$ 是表面上的一个三维顶点。如果将这个函数用于纹理过滤，那么方程 17.52 描述的就是一个高度场，即 $\mathbf{c}_{k,l}$ 代表的是一维高度。

对于双三次 B 样条面片，方程 17.25 中的 $\beta_3(t)$ 函数可以用于方程 17.52 中。即总共需要 4×4 个控制点 $\mathbf{c}_{k,l}$，方程 17.52 所描述的面片实际上位于最内层的 2×2 控制点范围内，如图 17.33 所示。同时，双三次 B 样条面片对于 Catmull–Clark 细分曲面算法也是必不可少的（章节 17.5.2）。关于 B 样条曲面，有很多非常好的参考书籍（详见书中连接）[111, 458, 777]。

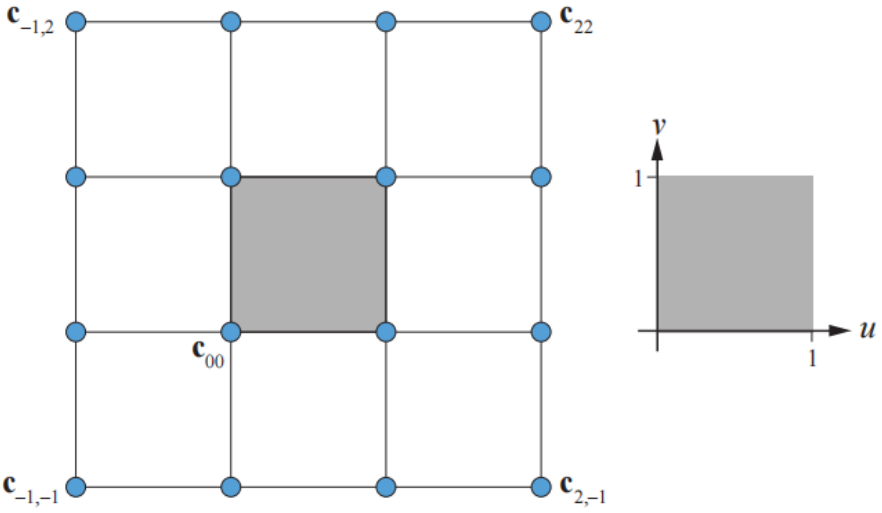


图 17.33：双三次 B 样条面片的构造模式，该面片具有 4×4 数量的控制点 $\mathbf{c}_{k,l}$。面片上 (u,v) 的定义域如右侧所示，它是一个单位正方形，位于最内部控制点所构成的范围内。

17.3 隐式表面

到目前为止，我们只讨论了参数化曲线和参数化曲面，而隐式表面（implicit surface）则是表示模型的另一个有效方法。隐式表面并不会使用一些参数（例如 u 和 v）来显式地描述表面上的一个顶点，而是会使用以下形式的函数来进行描述，它被称为隐式函数（implicit function）：

$$f(x,y,z) = f(\mathbf{p}) = 0 \tag{17.53}$$

对于隐式表面和隐式函数，可以这样进行理解：当我们将一个点 $\mathbf{p}$ 代入到隐式函数 f 中时，如果函数值为零，则说明点 $\mathbf{p}$ 位于这个隐式表面上。隐式表面通常会用于与射线的相交测试（章节 22.6—章节 22.9），因为它们要比相应的参数化表面（如果

有的话）更加容易求交。隐式表面的另一个优点是，一些构造实体几何（constructive solid geometry）算法可以很容易地应用在隐式表面上，也就是说，物体之间可以相减或者相加，在逻辑上是 AND:ed 或者 OR:ed 。此外，隐式表面对象还可以很容易地进行混合和变形。

下面是一些常见的隐式表面，它们都位于原点处：

$$\begin{aligned} f_s(\mathbf{p}, r) &= \|\mathbf{p}\| - r, && \text{sphere;} \\ f_{xz}(\mathbf{p}) &= p_y, && \text{plane in } xz \\ f_{rb}(\mathbf{p}, \mathbf{d}, r) &= \|\max(|\mathbf{p}| - \mathbf{d}, 0)\| - r, && \text{rounded box.} \end{aligned} \tag{17.54}$$

这些方程看起来都很生硬，需要对其进行一些解释。其中球面（sphere）就是点 $\mathbf{p}$ 到原点的距离，然后再减去半径。所以如果点 $\mathbf{p}$ 位于半径为 r 的球面上，那么 $f_s(\mathbf{p}, r) = 0$；否则，将会返回一个带符号的距离值，负数代表点 $\mathbf{p}$ 位于球体内部，正数代表点 $\mathbf{p}$ 位于球体外部。因此，这些函数有时也会被称为符号距离函数（signed distance function，SDF）。平面 $f_{xz}(\mathbf{p})$ 实际上就是点 $\mathbf{p}$ 的 y 坐标，即 y 轴正半轴的那一侧。对于圆角方框（rounded box）的表达式，我们假设向量的绝对值（ $|\mathbf{p}|$ ）和最大值是按照每个分量进行计算的。其中的 $\mathbf{d}$ 是方框的半边向量，如图 17.34 所示，图中还对这个公式进行了文字说明。如果想要获得一个非圆角方框（non-rounded box），只需设置 $r = 0$ 即可。

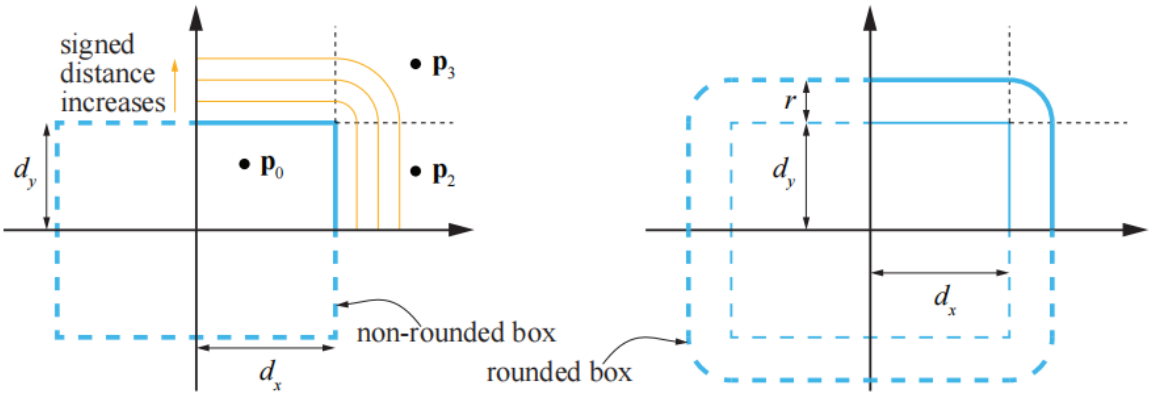


图 17.34：左：非圆角方框，其符号距离函数为 $\|\max(|\mathbf{p}| - \mathbf{d}, 0)\|$，其中点 $\mathbf{p}$ 为待测点，向量 $\mathbf{d}$ 的代表了 box 的半边长度。请注意，绝对值运算 $|\mathbf{p}|$ 使得其余的计算都发生在右上角象限中（这里使用 2D 进行说明）。$|\mathbf{p}| - \mathbf{d}$ 意味着，如果点 $\mathbf{p}$ 沿 x 轴方向上位于 box 内部，那么 $|p_x| - d_x$ 将会是一个负值，其他轴向上也是如此。这里只有正值会被保留，而负值会被 $\max()$ 限制为 0。因此，$\|\max(|\mathbf{p}| - \mathbf{d}, 0)\|$ 实际上计算了点 $\mathbf{p}$ 到 box 边缘的最近距离，这意味着如果在计算 $\max()$ 后有多个值为正数，那么 box 外的符号距离场将会变成圆

角。右：一个非圆角方框减去 r ，可以得到一个圆角方框，即让这个 box 向所有方向上都扩展半径 r 的长度。

隐式表面的法线由偏导数（partial derivative）进行描述，它被称为梯度（gradient），记为 ∇f ：

$$\nabla f(x,y,z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right) \tag{17.55}$$

为了能够精确计算，方程 17.55 中的隐式函数 f 必须是可微的（differentiable），因此也是连续的（continuous）。在实践中，人们经常使用一种被称为中心差分（central diffffference）的数值技术，它使用场景函数 f 进行采样[495]：

$$\nabla f_x \approx \frac{f\left(\mathbf{p} + \epsilon \mathbf{e}_x\right) - f\left(\mathbf{p} - \epsilon \mathbf{e}_x\right)}{2\epsilon} \tag{17.56}$$

同理， ∇f_y 与 ∇f_z 也可以这样计算出来。回顾一下，方程 17.56 中的 $\mathbf{e}_x = (1,0,0)$ ， $\mathbf{e}_y = (0,1,0)$ ， $\mathbf{e}_z = (0,0,1)$ ；而 ϵ 则是一个很小的数。

想要使用方程 17.54 中的基本类型来构建出一个复杂场景，需要使用并集运算符 $\cup$ （union operator）。例如：隐式表面 $f(\mathbf{p}) = f_s(\mathbf{p},1) \cup f_{xz}(\mathbf{p})$ ，它代表了由一个球面和一个平面所组成的场景。并集运算符有两个操作数，它通过取其中较小那个来实现，因为我们想要找到最接近点 $\mathbf{p}$ 的表面。如果想要对物体进行平移变换，可以在调用符号距离函数之前，先对点 $\mathbf{p}$ 进行平移，例如 $f_s(\mathbf{p}-\mathbf{t},1)$ 代表了一个被 $\mathbf{t}$ 平移的球面。旋转变换和其他类型变换也可以使用相同的方式来实现，即先对点 $\mathbf{p}$ 进行逆变换，再调用隐式函数。通过使用 $\mathbf{r} = \mathrm{mod}(\mathbf{p}, \mathbf{c}) - 0.5\mathbf{c}$ 来代替点 $\mathbf{p}$ ，将 $\mathbf{r}$ 作为符号距离函数的参数，还可以在整个空间中不断重复这个物体。

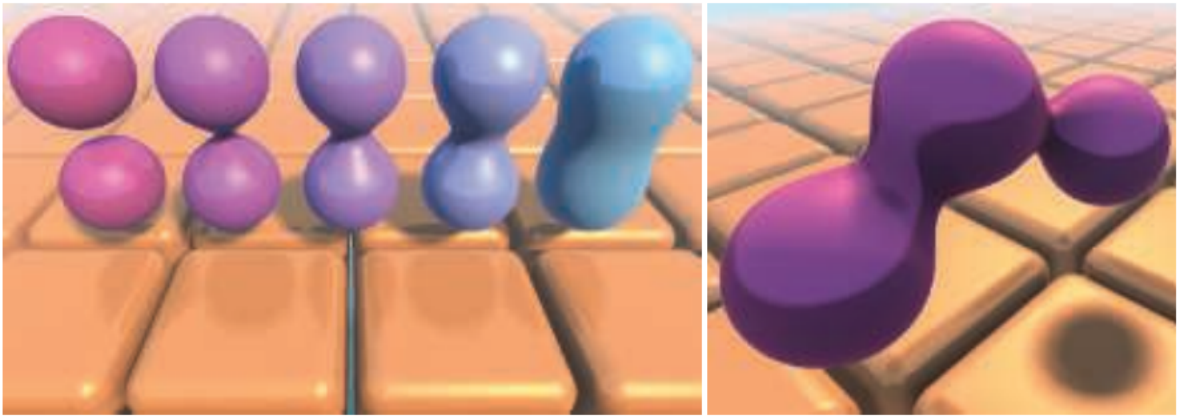


图 17.35：左：每对球体以不同的混合半径进行混合，混合半径从左到右递增；地面由重复的圆角方框所组成。右：将三个球体混合在一起。

隐式表面的混合是一个很好的特性，可以被用于 blobby 建模[161]、软体对象、或者元球（metaball）[67, 558]，图 17.35 展示了这样的一些例子。其基本思想是使用几个简单的几何图元（例如球体、椭球体或者其他一些可用形状），并将它们进行平滑地混合。其中每个物体都可以被看作是一个原子（atom），混合之后可以得到包含原子的分子（molecule）。混合的方法有很多种，其中一种常用的方法[1189, 1450]是对两个距离 d_1 和 d_2 进行混合，以及一个混合半径 r_b：

$$\begin{aligned} h &= \min\left(\max\left(0.5 + 0.5\left(d_2 - d_1\right)/r_b, 0.0\right), 1.0\right), \\ d &= (1-h)d_2 + hd_1 - r_b h(1-h), \end{aligned} \tag{17.57}$$

方程 17.57 中的 d 为混合距离。虽然这个函数只能对两个物体之间的最短距离进行混合，但是可以重复使用这个函数，来混合更多的物体（如图 17.35 的右侧部分所示）。

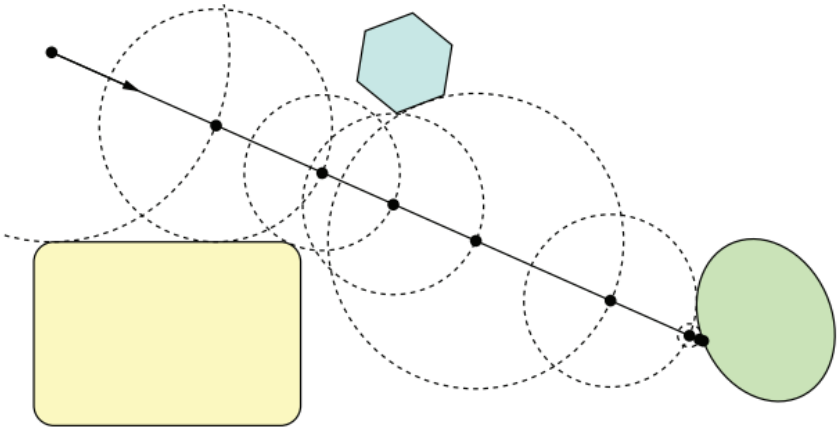


图 17.36：符号距离场中的光线步进。图中的虚线圈代表了从该位置到最近表面的距离。可以沿着当前的射线方向，直接步进到当前虚线圆的边界处，从而跳过其中的空白空间。

为了对一组隐式函数进行可视化，通常所使用的方法是光线步进[673]。一旦我们可以对一个场景进行光线步进，之后也就能够生成阴影、反射、环境光遮蔽以及其他的一些效果。图 17.36 展示了在一个符号距离场（signed distance field）内的光线步进。在光线的第一个点 $\mathbf{p}$ 处，我们计算点 $\mathbf{p}$ 到场景的最短距离 d。这个最短距离 d，可以理解成在点 $\mathbf{p}$ 处存在一个半径为 d 的球体，而在这个球体内部不存在其他任何物体。此时我们可以沿着射线方向向前步进 d 个单位，直到我们在某个设定的误差

范围（ϵ）内与表面相交；或者到达了预定义的最大步进次数，在这种情况下，我们可以认为光线击中了背景。图 17.37 展示了两个很好的例子。

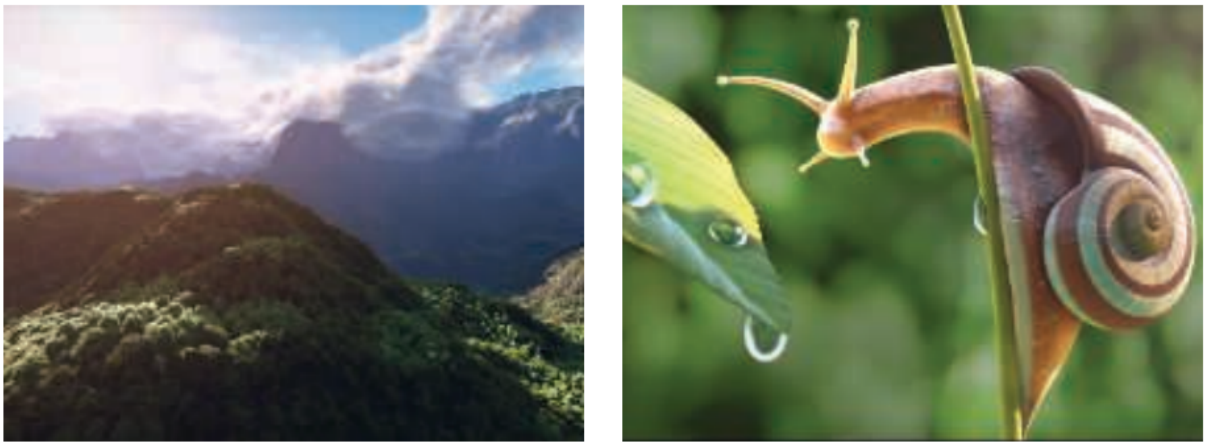


图 17.37：使用符号距离函数和光线步进来程序化生成热带雨林（左）和蜗牛（右）。热带雨林中的树，是使用带有程序化噪声的椭球体生成的。

每个隐式表面也可以转化为由三角形所组成的网格表面，有几种算法可以实现这一点[67, 558]。章节 13.10 中所介绍的移动立方体（marching cube）算法就是一个著名的例子。使用 Wyvill 和 Bloomenthal 算法来执行多边形化的代码可以在网络上获取到[171]；de Araujo 等人[67]对隐式表面多边形化的最新技术进行了调研。Tatarchuk 和 Shopf [1744]描述了一种技术，他们称之为移动四面体（marching tetrahedra），在该技术中，可以使用 GPU 来在一个三维数据集合中找到等值面。图 3.13 展示了一个使用几何着色器来提取等值面的例子。Xiao 等人[1936]提出了一个流体模拟系统，在该系统中，GPU 会对 10w 个粒子的位置进行计算，并使用这些粒子来渲染等值面，所有这些计算都是以交互式速率进行的。

17.4 细分曲线

细分技术（subdivision technique）可以用于创建光滑的曲线和表面。细分技术被应用于建模的其中一个原因是，它们在离散表面（三角形网格）和连续表面（例如一组 Bezier 面片）之间建立了联系，因此可以用于 LOD 技术（章节 19.9）。在章节 17.4 和章节 17.5 中，我们将首先描述细分曲线（subdivision curve）的工作原理，然后再去讨论更加流行的细分表面（subdivision surface）。

想要对细分曲线进行一些解释，最好的一个例子就是切角（corner cutting），如图 17.38 所示。图中左侧多边形的角会被切掉，并创建了一个新的多边形，这个新多边形的顶点数是原始多边形的两倍。

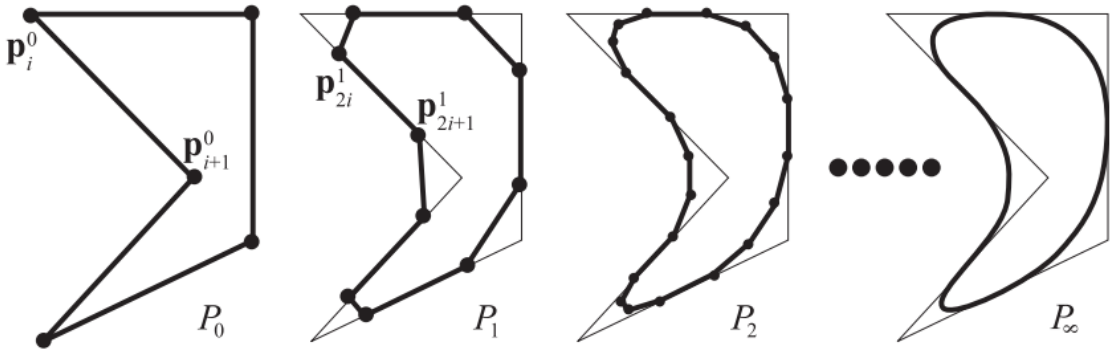


图 17.38：Chaikin 的细分方案。初始的控制多边形 P_0 被细分为多边形 P_1，然后再细分为多边形 P_2。可以看到，在这个细分过程中，每个多边形的尖角都会被截断。在经过无限次细分之后，可以得到一条极限曲线 P_∞。Chaikin 的细分方案是一个近似方案，因为最终生成的曲线并不会经过初始顶点。

然后，这个新多边形的角会被再次切掉，生成一个新的多边形，以此类推，可以一直切下去（或者更加实际地说，直到我们看不到任何差异为止）。最终所得到的曲线被称为极限曲线（limit curve）或者临界曲线，这是一个十分光滑的曲线，因为所有尖锐的角都被切掉了。这个过程也可以被认为是一个低通滤波器（low-pass filter），因为所有的尖角（高频信号）都被去除了。这个过程通常被写做 $P_0 \rightarrow P_1 \rightarrow P_2 \cdots \rightarrow P_\infty$，其中 P_0 为初始多边形，也被称控制多边形（control polygon），最后的 P_∞ 为极限曲线。

这种细分过程可以使用很多种不同的方式来完成，每种方式都有一个独特的细分方案。图 17.38 所展示的方案称为 Chaikin 方案[246]，其工作原理如下。假设多边形的 n 个初始顶点分别是 $P_0 = \left\{\mathbf{p}_0^0, \ldots, \mathbf{p}_{n-1}^0\right\}$，其中顶点的上标代表了其细分层次。Chaikin 的方案会在原始多边形的每对顶点之间创建两个新的顶点，记作 $\mathbf{p}_i^k$ 和 $\mathbf{p}_{i+1}^k$，这个过程的数学表达如下：

$$\mathbf{p}_{2i}^{k+1} = \frac{3}{4}\mathbf{p}_i^k + \frac{1}{4}\mathbf{p}_{i+1}^k \quad and \quad \mathbf{p}_{2i+1}^{k+1} = \frac{1}{4}\mathbf{p}_i^k + \frac{3}{4}\mathbf{p}_{i+1}^k \tag{17.58}$$

可以看到，方程 17.58 中的上标从 k 变为 $k+1$，这意味着我们从一个细分层级进入到了下一个细分层级，即 $P_k \rightarrow P_{k+1}$。在一次细分完成之后，原始顶点会被丢弃，新的顶点会被重新连接。图 17.38 中展示了这种行为，在距离原始顶点 $1/4$ 处创建了新的顶点。细分方案的美妙之处在于可以快速生成光滑的曲线，并且简单优雅。然而，我们并不会像章节 17.1 中那样，能够立即获得这条新曲线的参数化形式，虽然确实可以证明 Chaikin 算法生成了一个二次 B 样条[111, 458, 777, 1847]。同时，到目前为止，我们所提出的方案也只能处理多边形（封闭），但是大多数方案也都可以进

行一些扩展，从而可以对折线（开放）进行处理。对于 Chaikin 的方案，唯一的区别在于，折线的两个端点会每个细分步骤中保持不变（而不是被丢弃）。这使得最终生成的曲线会依次经过每个端点。

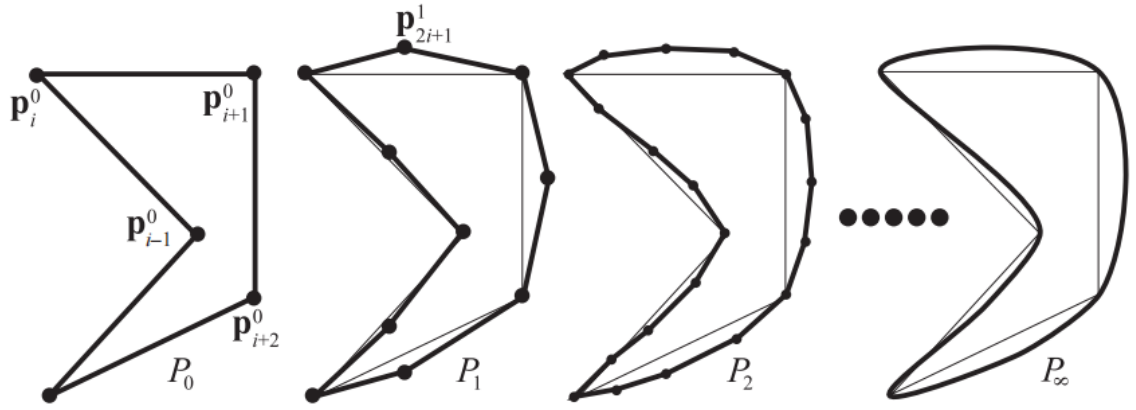


图 17.39：4 点细分方案的工作过程。这是一种曲线会经过初始顶点的插值方案，一般来说，曲线 P_{i+1} 依次经过曲线 P_i 上的所有点。这里我们使用了与图 17.38 中的相同控制多边形。

有两种不同的细分方案，分别是近似细分（approximating）和插值细分（interpolating）。Chaikin 的方案是一种近似细分，因为最终生成的极限曲线一般并不会经过初始多边形的顶点。这是因为在细分过程中，原始顶点都被丢弃了（或者被更新了）。相反，插值细分会保留之前细分步骤中的所有顶点，因此极限曲线 P_∞ 会依次经过 P_0 ， P_1 ， P_2 等的所有顶点，这意味着插值细分会对初始多边形进行插值。图 17.39 展示了一个插值细分的方案，其中所使用的多边形与图 17.38 中的完全相同。该方案使用最近的 4 个顶点来创建一个新的顶点[402]，这个过程的数学描述如下：

$$\begin{aligned}\mathbf{p}^{k+1}_{2i} &= \mathbf{p}^k_i, \\ \mathbf{p}^{k+1}_{2i+1} &= \left(\frac{1}{2}+w\right)\left(\mathbf{p}^k_i+\mathbf{p}^k_{i+1}\right)-w\left(\mathbf{p}^k_{i-1}+\mathbf{p}^k_{i+2}\right).\end{aligned} \tag{17.59}$$

方程 17.59 中的第一行，意味着我们会保留上一步中的顶点，并且不会改变它们（即不进行插值）；第二行则会在点 $\mathbf{p}^k_i$ 和点 $\mathbf{p}^k_{i+1}$ 之间创建一个新的顶点。其中的 w 被称为张力参数（tension parameter），当 $w=0$，结果是线性插值的；当 $w=1/16$ 时，我们可以得到如图 17.39 所示的行为。可以证明[402]，当 $0<w<1/8$ 时，所得到的曲线是 C^1 连续的。对于开放的折线（即首尾不相连），我们会在端点处遇到一些问题，因为我们需要在新顶点的两侧各有两个顶点，而折线情况下我们只有一个。我们可以将端点旁边的那个顶点，以端点为中心对称到另一边，从而解决这

个问题。即在折线的起点处，点 $\mathbf{p}_1$ 通过点 $\mathbf{p}_0$ ，反射得到点 $\mathbf{p}_{-1}$ ，然后在细分过程中使用这个顶点。图 17.40 展示了点 $\mathbf{p}_{-1}$ 的创建过程。

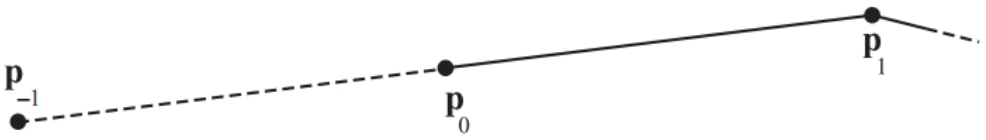


图 17.40：创建一个反射顶点 $\mathbf{p}_{-1}$ ，用于处理开放折线的端点情况。这个反射顶点的计算公式为： $\mathbf{p}_{-1} = \mathbf{p}_0 - (\mathbf{p}_1 - \mathbf{p}_0) = 2\mathbf{p}_0 - \mathbf{p}_1$ 。

另一种近似细分方案使用了以下的细分规则：

$$\begin{aligned} \mathbf{p}_{2i}^{k+1} &= \frac{3}{4}\mathbf{p}_i^k + \frac{1}{8}\left(\mathbf{p}_{i-1}^k + \mathbf{p}_{i+1}^k\right), \\ \mathbf{p}_{2i+1}^{k+1} &= \frac{1}{2}\left(\mathbf{p}_i^k + \mathbf{p}_{i+1}^k\right). \end{aligned} \tag{17.60}$$

其中方程 17.60 的第一行会对现有顶点进行更新；第二行会计算两个相邻顶点的中点。这个近似细分方案会生成一个三次 B 样条曲线（章节 17.1.6）。有关这些细分曲线的更多信息，请参阅 SIGGRAPH 的细分课程[1977]；the Killer B 的书[111]；Warren 和 Weimer 的细分书籍[1847]；或者 Farin 的 CAGD 书籍[458]。

给定一个点 $\mathbf{p}$ 及其相邻顶点，也可以直接将这个点“推（push）”到极限曲线上，即确定点 $\mathbf{p}$ 在曲线 P_∞ 上的坐标。同样，这对于切线也是可能的。Joy 在对这个主题进行了在线介绍[843]。

细分曲线中的许多概念同样也适用于细分曲面，我们将在下一小节中进行介绍。

17.5 细分表面

细分曲面是一种有效方法，它可以从具有任意拓扑结构的网格中，定义光滑、连续、无裂纹的表面。与本章节中的其他曲面一样，细分曲面也可以提供无限的 LOD。也就是说，我们可以生成任意数量的三角形或者多边形，同时保持紧凑的原始表面表示。图 17.41 展示了一个表面被细分的例子。细分表面的另一个优点在于，这些细分规则都很简单并且易于实现。细分表面的缺点在于，对表面连续性的分析常常会涉及到很多数学。然而，这种连续性分析，通常只有那些希望创建新细分方案的人才会兴

趣，并且这个话题也超出了本书的涵盖范围。有关这些细节，请参阅 Warren 和 Weimer 的书[1847]，以及 SIGGRAPH 有关细分的课程[1977]。

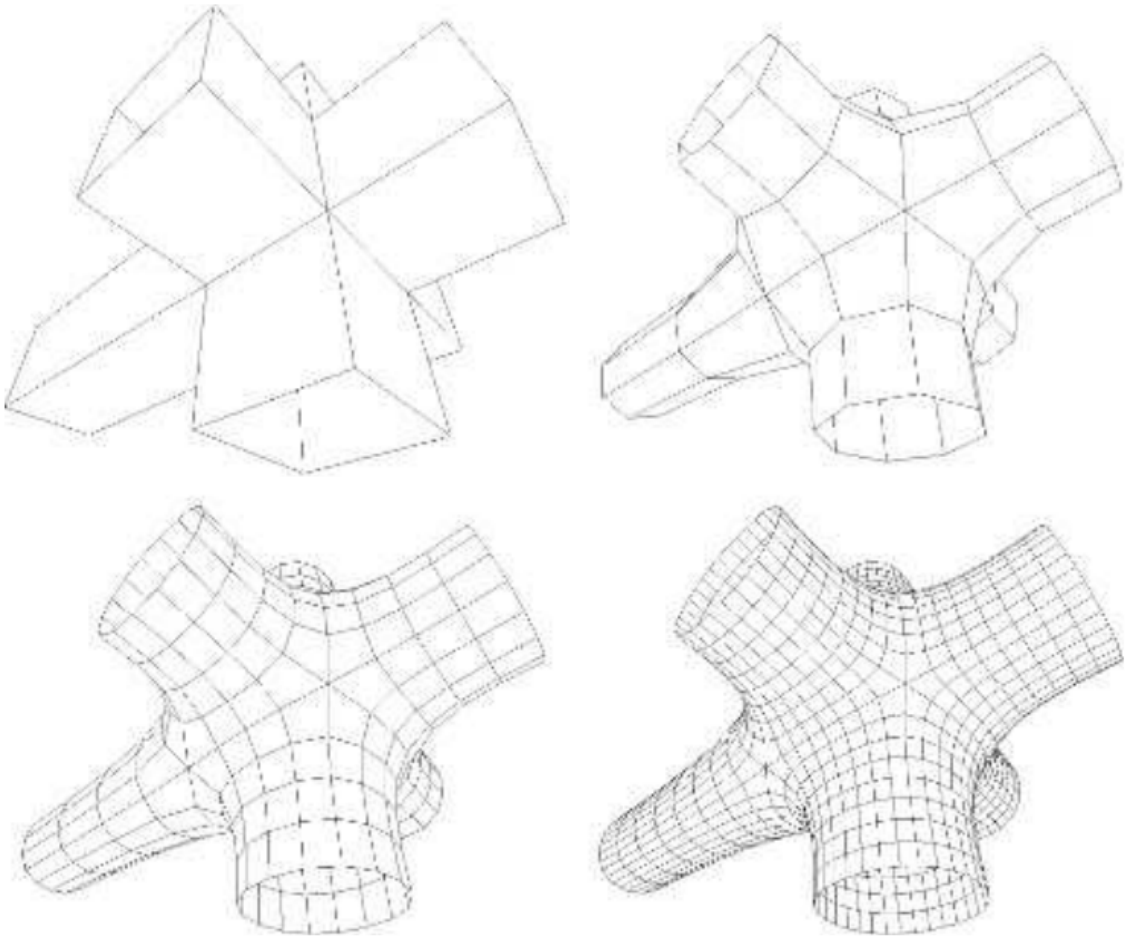


图 17.41：左上角展示了控制网格，即原始网格的样子，这是描述最终细分表面的唯一几何数据。其他三张图像各自被细分了 1、2、3 次。我们可以看到，随着细分次数的增加，所生成的多边形数量也越来越多，表面也越来越光滑。这里所使用的方案，是章节 17.5.2 中介绍的 Catmull–Clark 方案。

一般来说，曲面的细分（以及曲线的细分）可以被认为是一个两阶段的过程[915]。我们从一个多边形网格出发，这个初始的多边形网格被称为控制网格（control mesh）或者控制笼（control cage）。第一阶段称为细化阶段（refinement phase），该阶段会创建新的顶点，并将这些顶点重新连接，以构建新的、更小的三角形。第二个阶段称为平滑阶段（smoothing phase），通常会对网格中部分顶点或者全部顶点的新位置进行计算，如图 17.42 所示。在这两个阶段中的具体执行细节，便是一个细分方案的最主要特征。在第一阶段中，我们可以使用各种不同的方式来对多边形进行分割（split）；而在第二阶段中，具体细分规则的选择将会给出不同的网格特征，例如连续性水平、表面是近似细分还是插值细分等，这些属性在章节 17.4 中进行了描述。

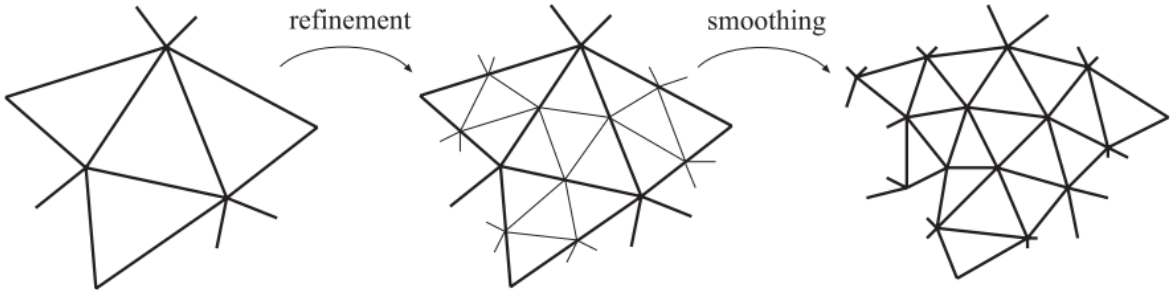


图 17.42：细分可以分为细化阶段和平滑阶段。细化阶段会创建新的顶点，并将这些顶点重新连接以创建新的三角形；而平滑阶段会对顶点的新位置进行计算。

细分方案（subdivision scheme）的特征可以被分为稳定的（stationary）或者不稳定的（non-stationary）；均匀的（uniform）或者非均匀的（nonuniform）；以及基于三角形的（triangle-based）还是基于多边形的（polygon-based）。一个稳定的方案会在每个细分步骤中都使用相同的细分规则；而一个不稳定的方案可能会根据当前正在处理的步骤来动态改变细分规则；我们下面所介绍的方案都是稳定的。一个均匀的方案会对每个顶点或者边都使用相同的规则；而非均匀的方案可能会对不同的顶点或者边使用不同的规则，例如：对于表面边界处的边，通常都会使用一组不同的细分规则。基于三角形的方案只能对三角形进行操作，同样也只能生成三角形；而基于多边形的方案则可以对任意多边形进行操作。

接下来我们将介绍几种不同的细分方法。在那之后，我们还会介绍两种使用细分曲面的扩展技术，以及对法线、纹理坐标和颜色进行细分的方法。最后会介绍一些实用的细分算法和渲染算法。

17.5.1 Loop 细分

Loop 的方法[767, 1067]是第一个针对三角形的细分方案，该方案类似于章节 17.4 中的最后一个方案，因为 Loop 细分是一种近似细分，它会对每个现有的顶点进行更新，并为每条边都创建一个新的顶点。这种方案的连通性如图 17.43 所示，从图中我们可以看到，每个三角形会被细分为 4 个新的三角形，因此在经过 n 次细分步骤之后，最初的一个三角形会被细分为 4^n 个三角形。

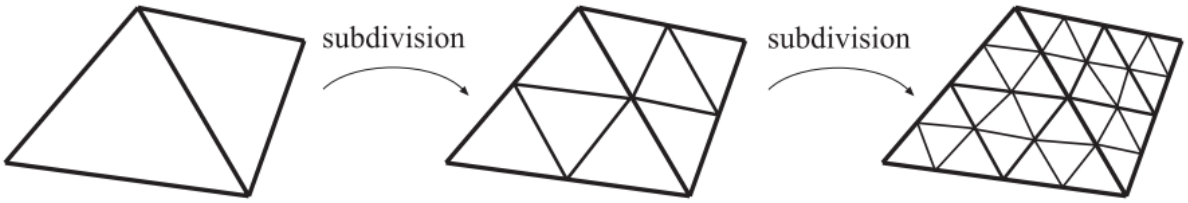


图 17.43：两步细分法的连通性，如 Loop 细分。每个三角形会生成 4 个新的三角形。

首先，在一个细分算法中，我们关注的是一个已存在的顶点 $\mathbf{p}^k$ ，其中 k 是细分步骤的数量。这也就意味着，点 $\mathbf{p}^0$ 其实就是原始控制网格中的顶点。

在经过一次细分之后，点 $\mathbf{p}^0$ 变成了点 $\mathbf{p}^1$ 。在一般情况下， $\mathbf{p}^0 \to \mathbf{p}^1 \to \mathbf{p}^2 \to \cdots \to \mathbf{p}^\infty$ ，其中 $\mathbf{p}^\infty$ 是极限点。如果点 $\mathbf{p}^k$ 存在 n 个相邻顶点，即 $\mathbf{p}_i^k, i \in \{0, 1, \ldots, n-1\}$ ，那么我们就说点 $\mathbf{p}^k$ 的价（valence）为 n 。图 17.44 中展示了上述这些符号和标记。另外，我们将一个 6 价的顶点称为规则顶点（regular）或者普通顶点（ordinary）；否则，它会被称为不规则顶点（irregular）或者异常顶点（extraordinary）。

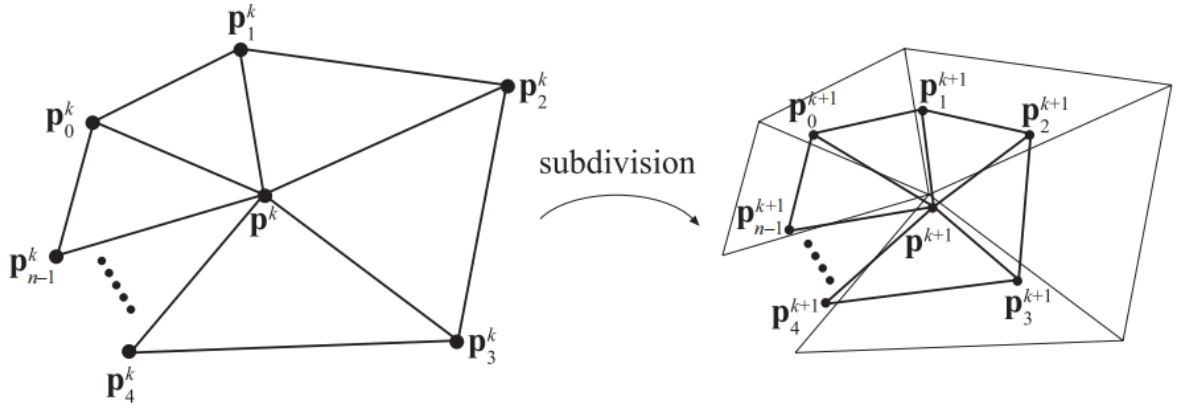


图 17.44：用于 Loop 细分方案的符号标记。左侧的邻域会被细分为右侧的邻域。对点 $\mathbf{p}^k$ 进行更新，并替换为点 $\mathbf{p}^{k+1}$ ；对于点 $\mathbf{p}^k$ 与点 $\mathbf{p}_i^k$ 之间的每一条边，都会生成一个新的顶点 $\mathbf{p}_i^{k+1}, i \in (1, \cdots, n)$ 。

下面我们将给出 Loop 方案的细分规则，其中第一个方程表示将一个已存在的顶点 $\mathbf{p}^k$ 更新为 $\mathbf{p}^{k+1}$ 的规则；第二个方程表示是在点 $\mathbf{p}^k$ 和每个相邻点 $\mathbf{p}_i^k$ 之间创建一个新的顶点 $\mathbf{p}_i^{k+1}$ 。同样， n 是点 $\mathbf{p}^k$ 的价。这个方程具体如下：

$$\begin{aligned} \mathbf{p}^{k+1} &= (1 - n\beta)\mathbf{p}^k + \beta\left(\mathbf{p}_0^k + \cdots + \mathbf{p}_{n-1}^k\right), \\ \mathbf{p}_i^{k+1} &= \frac{3\mathbf{p}^k + 3\mathbf{p}_i^k + \mathbf{p}_{i-1}^k + \mathbf{p}_{i+1}^k}{8}, i = 0 \ldots n-1. \end{aligned} \tag{17.61}$$

请注意，这里的下标 i 是对 n 取模计算的，也就是说：如果 $i = n - 1$，则让 $i + 1$，下标为 0；当 $i = 0$ 时，则让 $i - 1$，下标为 $n - 1$。这些细分规则可以很容易地被可视化为遮罩（mask），也称为模板（stencil），如图 17.45 所示（译者注：以下统一翻译为模板）。

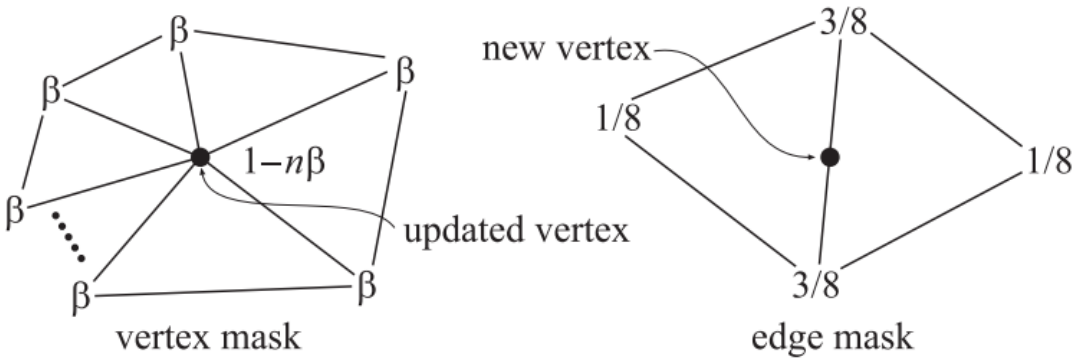


图 17.45：Loop 细分方案的模板（黑色圆圈代表更新或者生成的顶点）。这个模板展示了每个相关顶点的权重。例如：在对一个已经存在的顶点进行更新时，会对已经存在的顶点使用权重 $1 - n\beta$，对所有相邻的顶点使用权重 β，这些相邻顶点被称为 1 环顶点（1–ring）。

这些模板可视化的主要用途是，只需要使用一个简单的插图，就可以传达几乎整个细分方案。请注意，每个模板的权重之和为 1。这是一个适用于所有细分方案的特性，这样做的原因是，一个新的顶点应当位于加权点的邻域内部。在方程 17.61 中的常数 β，实际上是一个关于 n 的函数，其数学表达如下：

$$\beta(n) = \frac{1}{n}\left(\frac{5}{8} - \frac{(3 + 2\cos(2\pi/n))^2}{64}\right) \tag{17.62}$$

Loop 对于函数 β 的建议方案[1067]，可以在每个规则顶点（6 阶）上实现 C^2 连续性，在其他地方（所有不规则顶点上）实现 C^1 连续性[1976]。由于在细分过程中我们只会创建规则顶点，因此在原始控制网格中存在不规则顶点的地方，表面只有 C^1 连续性。图 17.46 展示了一个使用 Loop 方案对网格进行细分的例子。Warren 和 Weimer 给出了方程 17.62 的一个变体[1976]，该变体避免了使用三角函数：

$$\beta(n) = \frac{3}{n(n+2)} \tag{17.63}$$

使用方程 17.63，在规则顶点处具有 C^2 连续性，而在其他地方只有 C^1 连续性。由此生成的表面很难与常规的 Loop 表面区分开来。但是对于一个未封闭的网格，我们

就无法使用上述的细分规则了；相反，必须对这种边界使用一些特殊规则。对于 Loop 的方案，我们可以使用方程 17.60 中的反射规则，这也将在章节 17.5.3 中进行讨论。

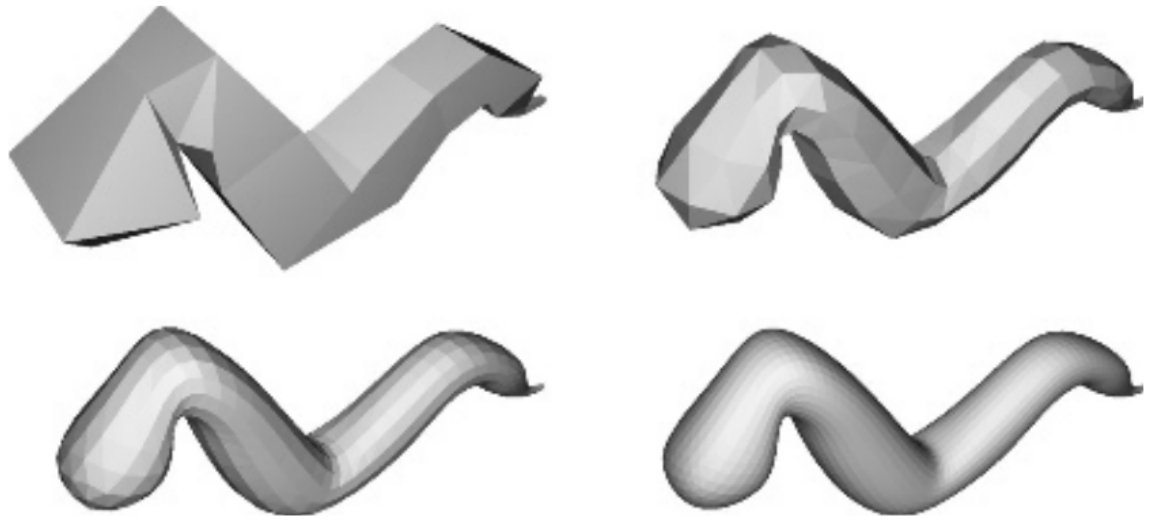


图 17.46：采用 Loop 细分方案对这个蠕虫进行三次细分。

经过无限次细分的表面称为极限表面（limit surface）。极限表面上的顶点和切线可以使用封闭形式的表达式进行计算。其中顶点的极限位置[767, 1977]可以使用方程 17.61 中的第一个公式进行计算，并将 $\beta(n)$ 替换为：

$$\gamma(n) = \frac{1}{n + \dfrac{3}{8\beta(n)}} \tag{17.64}$$

顶点 $\mathbf{p}^k$ 处的两条极限切线，可以通过对相邻的顶点（这些顶点被称为 1–环顶点或者 1–邻域顶点）进行加权来计算[767, 1067]，如下所示：

$$\mathbf{t}_u = \sum_{i=0}^{n-1} \cos(2\pi i/n)\mathbf{p}_i^k, \quad \mathbf{t}_v = \sum_{i=0}^{n-1} \sin(2\pi i/n)\mathbf{p}_i^k \tag{17.65}$$

有了某点上的两条切线，我们当然可以计算出法线，这里的法线是 $\mathbf{n} = \mathbf{t}_u \times \mathbf{t}_v$。需要注意的是，章节 16.3 中我们介绍过一种计算相邻三角形法线的方法，但是这里的方法开销更低[1977]。更重要的是，它可以给出该点的精确法线。

近似细分方案的一个主要优点在于，最终所得到的表面会趋于均匀（fair）。粗略地讲，这里的均匀与曲线或者表面弯曲的平滑程度有关[1239]，更高程度的均匀意味着

更加平滑的曲线或者表面。近似细分方案另一个优点在于，其收敛速度要比插值细分方案更快。然而，这也会意味着网格的形状通常会缩小。对于四面体这样的小型凸面网格而言，这一点尤其明显，如图 17.47 所示。一种减弱这种影响的方法是，在控制网格中使用更多数量的顶点，也就是说，在建模的时候必须要谨慎。Maillot 和 Stam 提出了一种结合多种细分方案的框架[1106]，从而可以控制这种收缩。还有一个特性可能会发挥巨大优势，即 Loop 细分表面会被包裹在原始控制点所形成的凸壳内部[1976]。

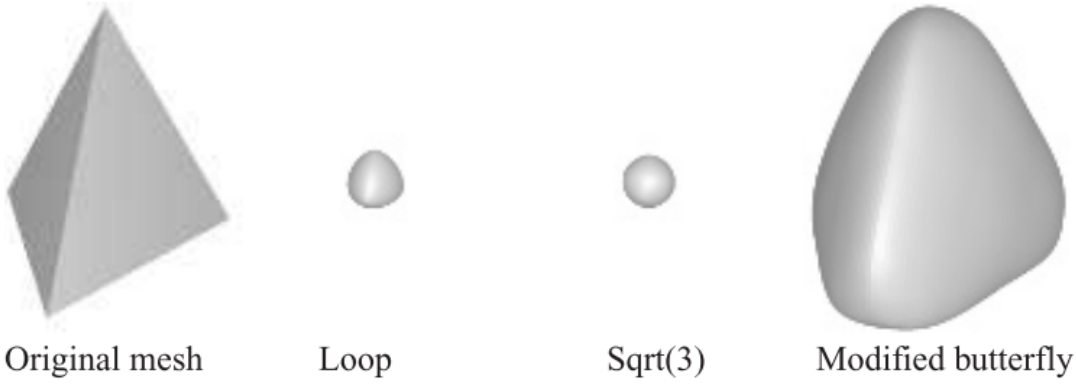


图 17.47：一个四面体被细分了 5 次，分别使用了 Loop 方案； $\sqrt{3}$ 方案；改进的 butterfly（modified butterfly，MB）方案[1975]。其中 Loop 方案和 $\sqrt{3}$ 方案[915]都是近似细分方案，而 MB 则是插值细分方案，插值细分意味着初始网格顶点会位于最终的表面上。本书中我们只会介绍近似细分方案，因为它们在游戏和离线渲染中十分流行。

Loop 细分方案生成了一种广义的三向四次 box 样条曲线（three-directional quartic box spline）。

> 这些样条曲面的内容超出了本书的讨论范围。请读者自行查阅 Warren 的书籍[1847]，SIGGRAPH 课程[1977]，以及 Loop 的论文[1067]。

因此，对于那些仅由规则顶点所构成的网格，我们实际上可以将其描述为一种样条表面。然而，对于存在不规则顶点的网格而言，这种样条描述方式是不可能的。能够从任意的网格顶点中生成光滑的曲面，这是细分方法的优点之一。在后续的章节 17.5.3 和章节 17.5.4 中，我们还会对使用 Loop 方案进行曲面细分的扩展方法进行介绍。

17.5.2 Catmull–Clark 细分

有很多细分方案可以处理多边形网格（而不仅仅是三角形网格），其中最著名的两个是 Catmull–Clark [239]和 Doo–Sabin [370]。

> 顺便说一句，这两篇文章发表在同一期刊的同一期上。

这里我们只对前者进行简要介绍。Catmull–Clark 表面被广泛应用于皮克斯的动画片中，包括动画短片《棋逢对手（Geri's Game）》[347]、《玩具总动员 2》以及之后皮克斯所有的动画故事片。这种细分方案也经常用于制作游戏模型，并且可能是其中最受欢迎的一种。DeRose 等人[347]指出，Catmull–Clark 方法倾向于生成更加对称的表面。例如：一个长方形的 box，会生成一个对称的椭球状表面，这与直觉一致。相比之下，基于三角形的细分方案会将立方体的每个表面都视为两个三角形，因此会根据正方形的三角形划分方式产生不同的结果。

图 17.48 展示了 Catmull–Clark 表面的基本思想，图 17.41 展示了使用 Catmull–Clark 细分的一个实际例子。从图中可以看出，该方案只会生成具有 4 个顶点的面。实际上，在完成第一步细分后，之后的每个细分步骤中只会生成 4 价的顶点，因此这样的顶点同样也被称为普通顶点或者规则顶点（在三角形面中则为 6 价）。

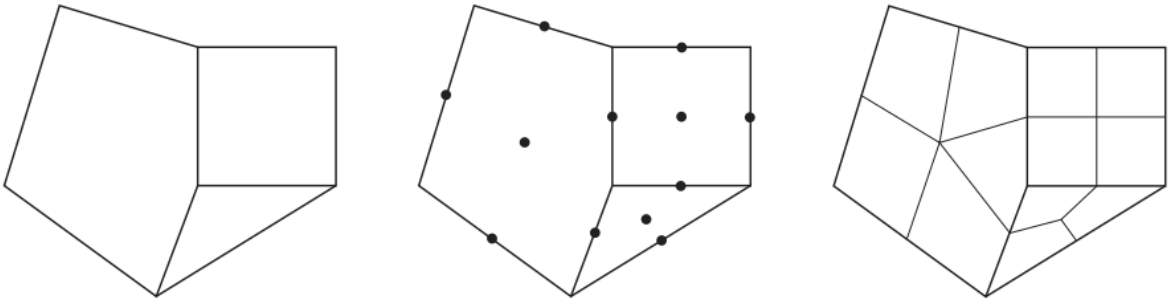


图 17.48：Catmull–Clark 细分的基本思想。其中每个多边形面都会生成一个新点，每条边上也会生成一个新点。然后再将它们连接起来，最右侧展示了一次细分的结果。这里并没有给出原始数据点的权重。

遵循 Halstead 等人[655]的符号表示，我们仅仅关注一已经存在的顶点 $\mathbf{v}^k$，它周围有 n 个边缘点 $\mathbf{e}_i^k$，其中 $i = 0 \cdots n-1$，如图 17.49 所示。现在，对于每个多边形面，我们都会计算一个新的面点 $\mathbf{f}^{k+1}$，并将其作为这个面的质心（centroid），即这个面上所有点的平均值。基于这些条件，具体的细分规则如下[239, 655, 1977]：

$$
\begin{aligned}
\mathbf{v}^{k+1} &= \frac{n-2}{n}\mathbf{v}^k + \frac{1}{n^2}\sum_{j=0}^{n-1}\mathbf{e}_j^k + \frac{1}{n^2}\sum_{j=0}^{n-1}\mathbf{f}_j^{k+1}, \\
\mathbf{e}_j^{k+1} &= \frac{\mathbf{v}^k + \mathbf{e}_j^k + \mathbf{f}_{j-1}^{k+1} + \mathbf{f}_j^{k+1}}{4}.
\end{aligned}
\tag{17.66}
$$

我们可以看到，顶点 $\mathbf{v}^{k+1}$ 在进行计算的时候，会考虑到原始顶点、边缘点的平均值，以及新创建的面点平均值。另一方面，新的边缘点 $\mathbf{e}_j^{k+1}$ 在计算的时候，会考虑原始顶点、原本的边缘点、以及两个新创建的面点，这两个面点分别是共享这条边的两个相邻面上的面点（平均值）。

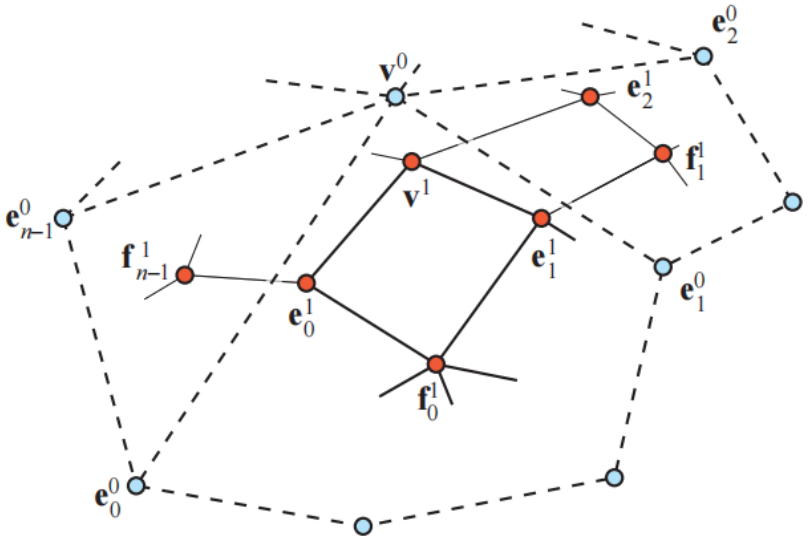


图 17.49：在这一步细分之前，我们有了蓝色的顶点以及相应的边面。在经过一步 Catmull–Clark 细分之后，我们得到了新的红色顶点，所有的新面都是四边形[655]。

Catmull–Clark 表面实际上描述了一个广义的双三次 B 样条表面。因此，对于那些仅由规则顶点（4 价）所构成的网格，我们实际上可以将表面描述为一个双三次 B 样条表面（章节 17.2.6）[1977]。然而，这对于包含不规则顶点的网格来说是不可能的，能够使用曲面细分来对这类网格进行处理是该方案的优势之一。同样顶点的极限位置和极限切线也可以进行计算，甚至可以使用显式方程来对任意的参数值进行计算[1687]。Halstead 等人[655]描述了一种计算极限顶点位置和极限顶点法线的不同方法。

章节 17.6.3 中介绍了一组高效技术，它可以使用 GPU 来渲染 Catmull–Clark 的细分表面。

17.5.3 分段平滑细分

从某种意义上说，曲面可能会被认为是十分无聊的，因为它们缺乏表面细节。有两种方法可以对曲面细节进行改进，分别是使用凹凸贴图或者位移贴图（章节 17.5.4）。在这里我们将介绍第三种方法，即分段平滑细分（piecewise smooth subdivision）方法，其基本思想是在表面上改变细分规则，从而允许出现褶皱（dart）、拐角（corner）和折痕（crease）。这样可以扩大曲面能够建模和表达的范围。Hoppe

等人[767]首先对 Loop 细分曲面应用了这种方法。图 17.50 展示了标准 Loop 细分曲面与分段光滑细分曲面的比较。

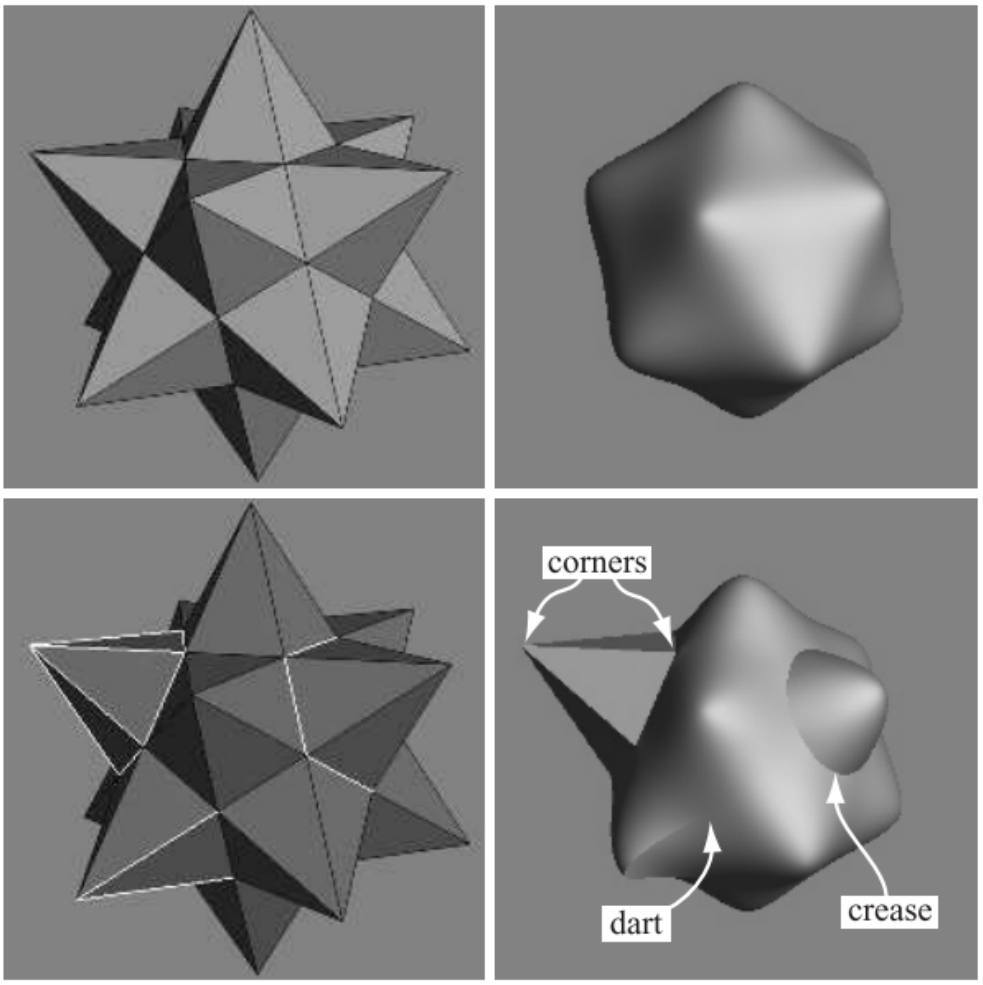


图 17.50：左上角展示了原始的控制网格，右上角展示了使用标准 Loop 细分方案生成的极限表面。第二行则展示了 Loop 方案的分段平滑细分。左下角展示了带有标记边缘（尖锐）的控制网格，其中的标记使用浅灰色进行显示。右下角展示了据此生成的表面，并对拐角、褶皱和折痕进行了标记。

为了能够在表面上实现这样的特征，我们需要知道哪些部分是尖锐的边缘，并将这部分进行标记，这样我们就知道在哪里进行细分了。我们将一个顶点上的尖锐边缘数量记为 s，然后我们将顶点划分为光滑顶点（$s = 0$）、褶皱顶点（$s = 1$）、折痕顶点（$s = 2$）以及拐角顶点（$s > 2$）。因此，折痕是表面上的一条曲线，该曲线的连续性为 C^0。褶皱是一个非边界（non-boundary）顶点，它指的是一个会平滑融入表面的折痕。最后，拐角是三个或者更多折痕聚集在一起的顶点。可以对每个边界进行相应地标记，从而定义边界。

在对各种顶点类型进行分类后，Hoppe 等人使用了一个表格，来确定对各种组合具体使用哪个模板。他们还展示了如何计算表面上的极限顶点和极限切线。Biermann

等人[142]提出了几种改进的细分规则。例如：当异常顶点出现在边界上时，之前的一些细分规则可能会导致出现间隙，而新的细分规则可以避免这种情况的发生。此外，这些规则使得在顶点处指定法线成为可能，并且生成的表面将会适应在该点处获得的法线。DeRose 等人[347]提出了一种创建软折痕（soft crease）的技术，该技术允许一条边首先被尖锐细分若干次（包括分数次），然后再使用标准细分。

17.5.4 位移（Displaced ）细分

凹凸映射（章节 6.7）是一种为光滑表面添加细节的方法。然而，这只是一个错觉技巧（illusionary trick），它仅仅改变了每个像素位置上的法线信息或者局部遮挡信息。无论有没有使用凹凸贴图，物体的轮廓看起来都是一样的。位移映射（displacement mapping）是对凹凸映射（bump mapping）的自然扩展[287]，在位移映射中，表面会被实际位移，这种位移通常是沿着法线方向进行的。因此，如果表面上有一个点 $\mathbf{p}$，其归一化法线为 $\mathbf{n}$，那么位移表面上的对应点是：

$$\mathbf{s}(u,v) = \mathbf{p}(u,v) + d(u,v)\mathbf{n}(u,v) \tag{17.67}$$

其中标量 d 是点 $\mathbf{p}$ 的位移距离，当然这个位移也可以是一个向量[938]。

在本小节中，我们将介绍位移细分表面（displaced subdivision surface）[1006]。这个移位表面的一般想法是，将一个粗糙的控制网格细分为一个光滑的表面，然后再沿着法线进行标量位移。在一个位移细分表面的描述中，方程 17.67 中的点 $\mathbf{p}$ 为（粗糙控制网格的）细分表面上的极限点， $\mathbf{n}$ 为点 $\mathbf{p}$ 处的归一化法线，其计算方式为：

$$\mathbf{n} = \frac{\mathbf{n}'}{\|\mathbf{n}'\|}, \text{where } \mathbf{n}' = \mathbf{p}_u \times \mathbf{p}_v \tag{17.68}$$

方程 17.68 中的 $\mathbf{p}_u$ 和 $\mathbf{p}_v$ 是细分表面的一阶导数，即它们是点 $\mathbf{p}$ 处的两条切线。Lee 等人[1006]对原始的粗糙控制网格使用了 Loop 细分曲面，其切线可以使用方程 17.65 进行计算。需要注意的是，这里的符号表示略有不同，方程 17.65 中使用了 $\mathbf{t}_u$ 和 $\mathbf{t}_v$ 来代表切线，这里我们则使用了 $\mathbf{p}_u$ 和 $\mathbf{p}_v$，它们的含义是一样的。方程 17.67 描述了结果表面中的位移位置，这里我们还需要该点处的法线 $\mathbf{n}_s$ 才能正确渲染这个位移细分表面。法线和位移细分表面的解析计算如下所示[1006]：

$$
\begin{aligned}
\mathbf{n}_s &= \mathbf{s}_u \times \mathbf{s}_v, \text{ where} \\
\mathbf{s}_u &= \frac{\partial \mathbf{s}}{\partial u} = \mathbf{p}_u + d_u\mathbf{n} + d\mathbf{n}_u \\
\mathbf{s}_v &= \frac{\partial \mathbf{s}}{\partial v} = \mathbf{p}_v + d_v\mathbf{n} + d\mathbf{n}_v
\end{aligned}
\tag{17.69}
$$

为了简化计算，Blinn [160]建议在位移量较小的情况下，可以忽略方程 17.69 中的第三项。否则，可以使用下列表达式来计算 $\mathbf{n}_v$ （同理还有 $\mathbf{n}_v$ ）[1006]：

$$
\begin{aligned}
\overline{\mathbf{n}}_u &= \mathbf{p}_{uu} \times \mathbf{p}_v + \mathbf{p}_u \times \mathbf{p}_{uv}, \\
\mathbf{n}_u &= \frac{\overline{\mathbf{n}}_u - (\overline{\mathbf{n}}_u \cdot \mathbf{n})\,\mathbf{n}}{\|\mathbf{n}'\|}.
\end{aligned}
\tag{17.70}
$$

请注意，方程 17.70 中的 $\overline{\mathbf{n}}_u$ 并不是什么新的符号标记，它只是计算过程中的一个“临时”变量。对于一个普通顶点（6 价）而言，其一阶导数和二阶导数的计算方法很简单。它们的计算模板如图 17.51 所示。对于一个异常顶点（非 6 价），方程 17.69 中第一行和第二行的第三项会被省略。图 17.52 展示了一个使用 Loop 细分方案的位移映射结果。

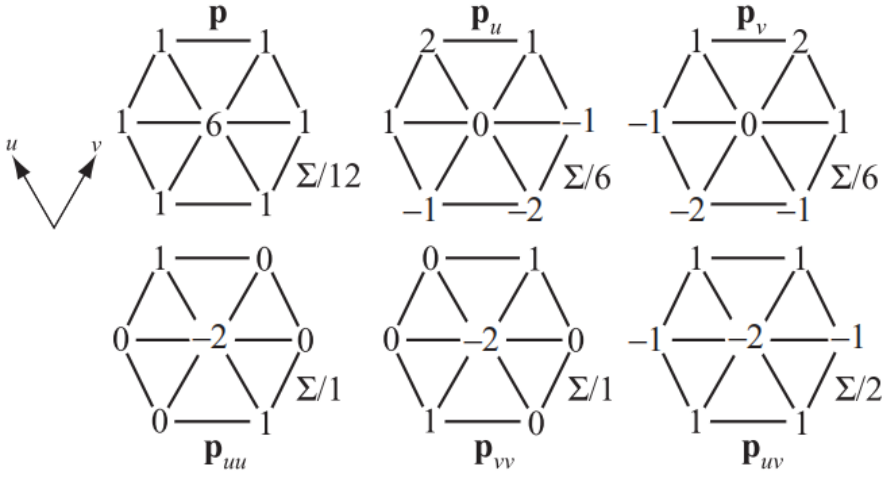


图 17.51：Loop 细分方案中普通顶点的模板。请注意，在使用这些模板之后，结果需要除以权重之和。[1006]

当一个位移表面距离观察者很远时，可以使用标准的凹凸映射来产生这种位移的错觉，这样做可以节省一些几何处理的开销。有一些凹凸映射方案，需要在顶点处使用切线空间坐标系，可以使用以下方法来产生一个切线空间基底 $(\mathbf{b}, \mathbf{t}, \mathbf{n})$ ，其中：

$$\mathbf{t} = \mathbf{p}_u / \|\mathbf{p}_u\|$$
$$\mathbf{b} = \mathbf{n} \times \mathbf{t}$$

Nießner 和 Loop 提出了一种方法[1281]，该方法与 Lee 等人所提出的方法相类似，不同之处在于他们使用了 Catmull–Clark 表面，并使用位移函数上的导数来直接进行求值，计算速度更快。他们还使用了基于硬件的曲面细分管线（章节 3.6）来进行快速曲面细分。

图 17.52：左边是一个粗糙的控制网格。中间采用了 Loop 细分方案进行细分。右边展示了位移细分表面。

17.5.5 法线、纹理和颜色插值

在本小节中，我们将介绍一些不同的策略，来分别处理法线、纹理坐标和逐顶点颜色的插值。

章节 17.5.1 中我们介绍了 Loop 细分方案，它可以显式地计算极限切线和极限法线，不过这些计算会涉及到三角函数，其计算开销可能会很高。Loop 和 Schaefer [1070] 提出了一种近似技术，该方法使用双三次 Bezier 曲面来近似 Catmull–Clark 曲面（章节 17.2.1）。对于法线计算，会导出两个切线面片，其中一个在 u 方向上，另一个在 v 方向上，使用这些向量的叉乘来计算法线。一般来说，可以使用方程 17.35 来计算一个 Bezier 面片的导数。然而，由于导出的 Bezier 面片近似于 Catmull–Clark 表面，因此这两个切线面片并不会形成连续的法线场。有关如何克服这些问题，详见 Loop 和 Schaefer 的论文[1070]。Alexa 和 Boubekeur [29]认为，就每次计算的质量而言，对法线进行细分可以更加高效，同时也会在着色中提供了更好的连续性表现。有关对法线进行细分的细节，请参考他们的论文（详见书中链接）。在 Ni 等人的 SIGGRAPH 课程[1275]中，也可以找到更多类型的近似方案。

假设网格中的每个顶点都有一个纹理坐标和一个颜色。为了能够将这些数据用于细分表面，我们还必须为每个新生成的顶点都创建相应的纹理坐标和颜色。其中最显而易见的方法就是，使用与多边形网格相同的细分方案。例如：我们可以将颜色视为一个四维向量（RGBA），并对这个四维向量进行细分，从而为新顶点创建颜色。这是一种合理的方法，因为这样生成的颜色将会有一个连续的导数（假设细分方案至少有 C^1 连续性），这样可以避免表面上出现突然的颜色变化。同样的方法也可以用于纹理坐标的生成[347]，但是当纹理空间中存在边界时，则需要小心处理，例如：假设现在我们有两个面片共享同一条边缘，但是沿着这条边缘具有不同的纹理坐标。几何网格会像往常一样使用表面规则进行细分，但是在这种情况下，纹理坐标应当使用边界规则来进行细分。

Piponi 和 Borshukov [1419]给出了一种复杂的纹理化细分曲面的方案。

17.6 高效曲面细分

为了能够在一个实时渲染环境中显示曲面，我们通常需要为曲面创建一个三角形网格，这个过程被称为曲面细分（tessellation）。其中最简单的曲面细分形式被称为均匀曲面细分（uniform tessellation）。假设现在我们有一个参数化的 Bezier 面片 $\mathbf{p}(u,v)$，详见方程 17.32 中的描述。我们想为每个面片的边都计算 11 个顶点，从而对这个面片进行细分，总共会得到 $10 \times 10 \times 2 = 200$ 个三角形。最简单的方法就是对 uv 空间进行均匀采样。因此，我们对所有的 $(u_k, v_l) = (0.1k, 0.1l)$，都计算一遍 $\mathbf{p}(u,v)$，其中 k 和 l 是 0–10 范围内的任何整数。这个操作可以通过两个嵌套的 for 循环来实现。每四个表面点 $\mathbf{p}(u_k, v_l)$，$\mathbf{p}(u_{k+1}, v_l)$，$\mathbf{p}(u_{k+1}, v_{l+1})$ 和 $\mathbf{p}(u_k, v_{l+1})$ 可以创建两个三角形。

这种方法虽然很简单很直接，但是还有一些更快的方法。相比于先将曲面细分成三角形网格，然后再将这个三角形网格通过总线从 CPU 发送到 GPU，将这个曲面表示直接发送到 GPU 中，并由 GPU 来处理这个数据扩展的过程要更有意义，效率会更高。我们在章节 3.6 中描述了管线的曲面细分阶段，这里我们快速回顾一下，详见图 17.53。

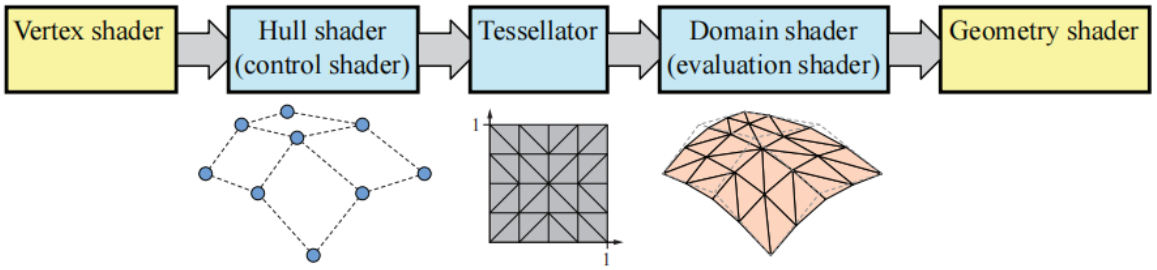


图 17.53：图形管线与硬件曲面细分，中间的三个方框（蓝色）展示了新的阶段。这里我们使用了 DirectX 中的命名约定，括号中有 OpenGL 中的对应名称。其中的壳着色器（hull shader）计算了控制点的新位置，并计算了曲面细分因子（tessellation factor），这个因子决定了后续步骤中具体应当生成多少个三角形。曲面细分器（tessellator）会在 uv 空间中生成顶点（在本例中则是一个单位正方形），并将这些顶点连接成三角形。最后，域着色器（domain shader）使用控制点来计算每个 uv 坐标所对应的位置。

镶嵌器中可以使用分数曲面细分（fractional tessellation）技术，我们将在下一小节中进行介绍。然后我们会介绍有关自适应曲面细分（adaptive tessellation）的部分。最后，我们会介绍如何使用曲面细分硬件来高效渲染 Catmull–Clark 表面和位移映射表面。

17.6.1 分数曲面细分

为了使得参数化表面获得更加光滑的 LOD，Moreton 引入了分数曲面细分因子（fractional tessellation factor）[1240]。这些分数因子使得有限形式的自适应曲面细分成为可能，因为可以在参数化表面的不同区域使用不同的曲面细分因子。下面我们将介绍这些技术的工作原理。

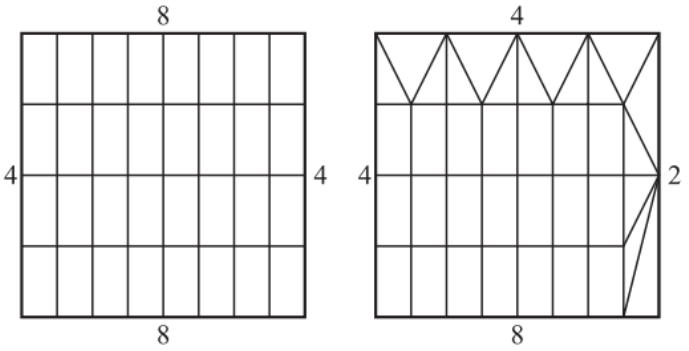


图 17.54：左：普通的曲面细分，在行方向上使用一个因子，在列方向上使用另一个因子。右：在所有的四个边上，都使用独立的曲面细分因子。[1240]

在图 17.54 中，左侧展示了每行每列都使用恒定的曲面细分因子，右侧展示了所有四条边界都使用的独立曲面细分因子。请注意，一条边上的曲面细分因子是这条边上生成顶点的数量减 1。在图 17.54 的右侧面片中，上下两边缘的内部都使用了顶部和底部细分因子中较大的那个；同样，左右两边缘的内部也使用了左边和右边细分因子中较大的那个。因此，面片内部的基本曲面细分率为 4×8，而对于细分因子较小的边界，则会沿着边界来填充三角形。Moreton [1240]更加详细地描述了这个过程。

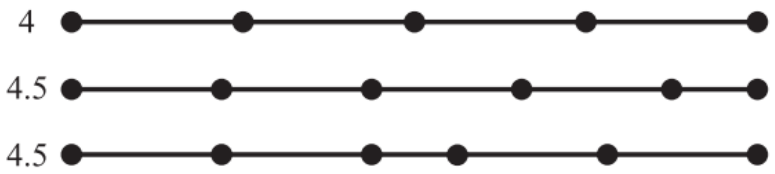


图 17.55：上：整数曲面细分。中：分数曲面细分，最右边会出现分数段细分。下：分数曲面细分，中间会出现分数段细分。这样可以避免相邻面片之间出现裂缝。。

分数曲面细分因子的概念如图 17.55 所示。对于整数曲面细分因子 n，会在 k/n 处生成 $n+1$ 个点，其中 $k = 0, \ldots, n$。对于分数曲面细分因子 r，会在 k/r 处生成 $\lceil r \rceil$ 个点，其中 $k = 0, \ldots, \lfloor r \rfloor$。其中运算符 $\lceil r \rceil$ 代表了 r 的上限（ceiling），即将 r 向上舍入；$\lfloor r \rfloor$ 代表了 r 的下限（floor），即将 r 向下舍入。这样，最右边的点会被“固定（snapped）”到最右侧的端点处。我们从图 17.55 的第二行插图可以看出，这种模式是非对称的。这可能会导致出现一些问题，因为相邻的面片可能会在另一个方向上生成顶点，从而在表面之间产生裂缝。Moreton 通过创建一个对称的点模式来解决这个问题，如图 17.55 第三行所示，图 17.56 也展示了一个例子。

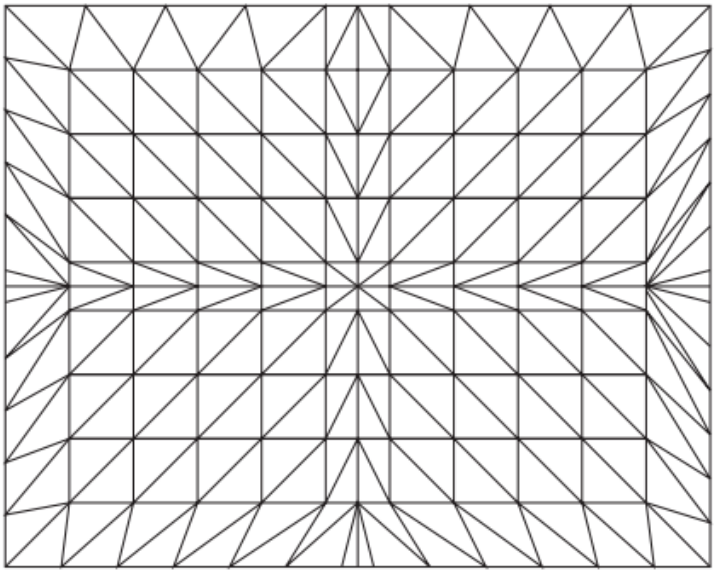


图 17.56：矩形区域的分数细分面片。[1240]

到目前为止，我们已经看到了对一个矩形区域（例如 Bezier 面片）进行曲面细分的方法。同样地，也可以对三角形进行分数曲面细分[1745]，如图 17.57 所示。就像上文中的四边形一样，也可以对每个三角形边分别指定独立的分数曲面细分率。使用这种方法可以实现自适应的曲面细分（章节 17.6.2），如图 17.58 所示，其中渲染了一个位移映射的地形。一旦我们创建了三角形或者四边形，就可以将它们转发到管线的下一个阶段中，这将在下一小节中进行介绍。

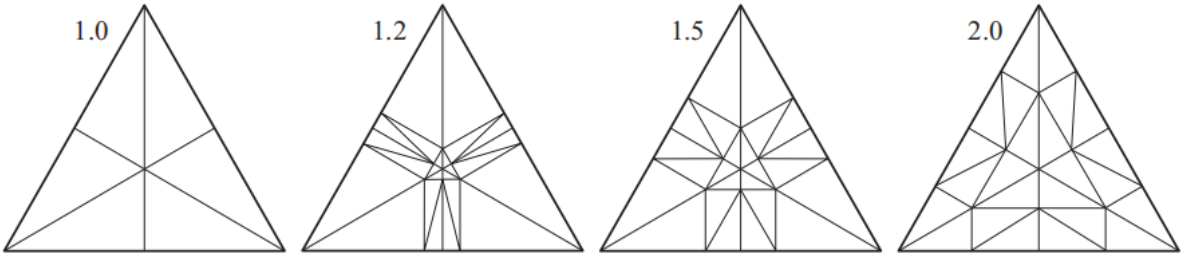


图 17.57：三角形的分数曲面细分，旁边显示了对应的曲面细分因子。请注意，这个曲面细分因子可能会与实际曲面细分硬件所产生的因子并不完全对应。[1745]

17.6.2 自适应曲面细分

如果使用足够高的采样率，那么均匀曲面细分就已经能够得到很好的结果了，然而，表面上的某些区域可能并不需要如此高程度的曲面细分。这可能是因为表面上的某些区域弯曲程度更大，因此需要更高的曲面细分来进行处理；而表面上的其他部分几乎都是平坦的或者遥远的，因此只需要几个三角形就可以很好地近似它们了。使用均匀曲面细分会生成很多不必要的三角形，其中一个解决方案是使用自适应曲面细分（adaptive tessellation），它指的是根据表面上的某些指标（例如曲率、三角形边的长度或者某些屏幕尺寸指标）来动态调整曲面细分率的算法。图 17.58 展示了一个使用自适应曲面细分的地形例子。

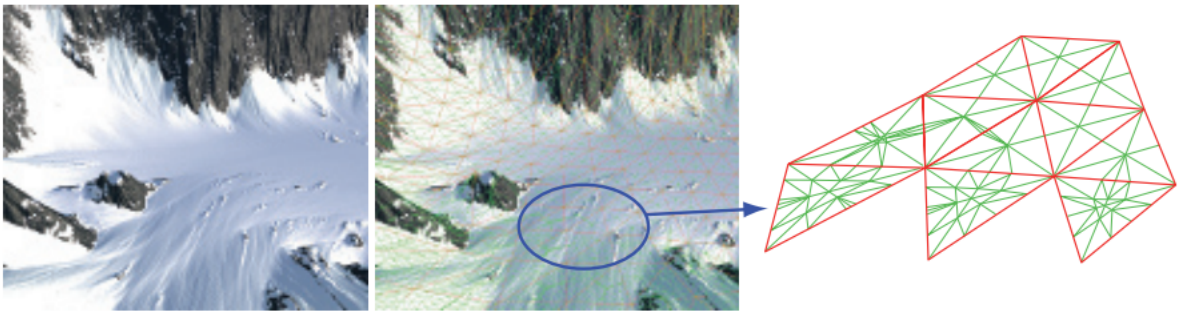


图 17.58：位移地形使用自适应分数曲面细分进行渲染。从右侧放大的网格中可以看到，红色三角形的边缘使用了独立的分数曲面细分因子，这实现了自适应的曲面细分。

在不同的细分区域之间，需要注意避免出现裂缝，如图 17.59 所示。当使用分数曲面细分的时候，边缘的曲面细分因子通常会基于边缘本身的信息，因为这个边缘是两个相连片元之间共享的所有数据。这是一个很好的开始，但是由于浮点数本身的不准确性，仍然可能会出现裂痕。Nießner 等人[1279]讨论了如何使得计算过程完全无懈可击（fully watertight），例如：确保对于一条边，无论从 $\mathbf{p}_0$ 到 $\mathbf{p}_1$ 是否进行曲面细分，返回的都是完全相同的点，反之亦然。

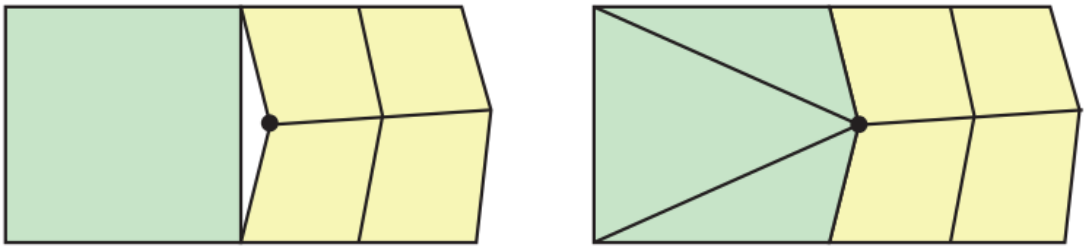


图 17.59：左边：可以看到两个区域之间存在一条裂缝，这是因为右侧的曲面细分率比左侧高。这个问题在于，右边的区域对存在黑点的表面进行了计算，而左边的区域则没有。右边：一个标准的解决方案。

在本小节中，我们将介绍一些通用技术，它们可以用来计算分数的曲面细分因子；或者决定什么时候停止进一步的曲面细分；以及什么时候将一个较大的面片细分成一组较小的面片。

终止自适应曲面细分

为了能够提供自适应的曲面细分效果，我们需要确定何时来停止曲面细分，即如何计算这个分数曲面细分因子。我们可以仅仅使用一条边的信息来确定是否应当终止曲面细分，同样也可以使用来自整个三角形或者其他组合的信息来决定是否终止。

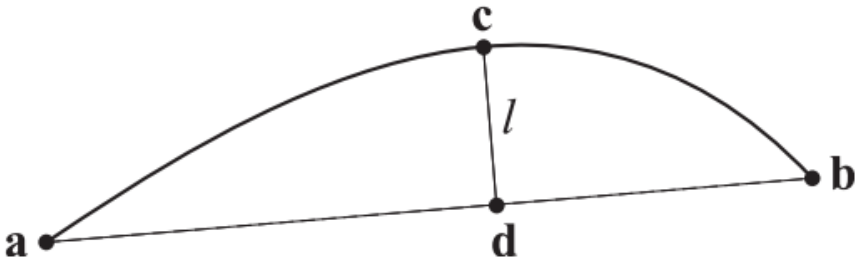


图 17.60：点 $\mathbf{a}$ 和点 $\mathbf{b}$ 已经在这个表面上生成了。现在的问题是：是否应该在表面上生成一个新的点 $\mathbf{c}$ 。

还应当注意的是，在使用自适应曲面细分的时候，如果某个边缘上的曲面细分因子在两帧之间变化过大，则可能会在两帧之间出现一些游动的（swimming）或者突变（popping）的瑕疵，这也是在计算曲面细分因子时要考虑到的一个因素。给定一条边($\mathbf{a}$, $\mathbf{b}$)和一条相关的曲线，即面片的边缘曲线，我们可以尝试估计点 $\mathbf{a}$ 和点 $\mathbf{b}$ 之间曲线的平坦程度，如图 17.60 所示。在参数化空间中找到点 $\mathbf{a}$ 和点 $\mathbf{b}$ 之间的中点，并计算其三维空间中的对应点 $\mathbf{c}$ 。然后将点 $\mathbf{c}$ 投影到 $\mathbf{a}$ ， $\mathbf{b}$ 所在的直线上，形成点 $\mathbf{d}$ ，并计算线段 $\mathbf{cd}$ 的长度 l 。这个长度 l 可以用于确定这段曲线是否足够平坦。如果这个 l 足够小，则可以认为这段曲线是平坦的。但是请注意，这种方法可能会将一段 S 形曲线错认为是平坦的。一种解决这个问题的方法是，可以对参数化样本点进行随机扰动[470]。与只使用投影线段长度 l 相比，另一种选择是使用一个比值 $l/\|\mathbf{a}-\mathbf{b}\|$ ，它可以给出一个相对度量[404]。这种判断曲线平坦程度的方法同样也可以扩展到三角形面片上，我们只需要计算三角形面片中间的表面点，并计算从该点到三角形平面的距离即可。为了确保这种算法能够终止，通常需要对细分的数量设定一个上限，当达到这个上限时就终止细分。对于分数曲面细分，可以将点 $\mathbf{c}$ 到点 $\mathbf{d}$ 的向量投影到屏幕上，并依据这个屏幕长度来决定曲面细分率的大小。

到目前为止，我们已经讨论了如何仅从表面的形状来确定一个合理的曲面细分率。还有一些其他用于动态曲面细分的因素，例如：顶点的局部邻域是否为[769, 1935]：

1. 位于视锥体内。
2. 位于模型的正面。
3. 占据屏幕空间中的一大片面积。
4. 靠近物体的轮廓。

在这里我们将依次讨论这些因素。对于视锥体剔除而言，可以放置一个球体来包围这条边，然后在视锥体上对这个球体进行测试。如果这个球体位于视锥体外部，那么我们就不再细分这条边了。

对于正面剔除（face culling），可以从表面描述中计算得到点 $\mathbf{a}$ ，点 $\mathbf{b}$ ，以及可能的点 $\mathbf{c}$ 处的法线。这三条法线和点 $\mathbf{a}$ 、 $\mathbf{b}$ 、 $\mathbf{c}$ 一起，定义了三个平面。如果这三个平面全部都是朝向后的，那么很可能不需要对该边缘进行细分。

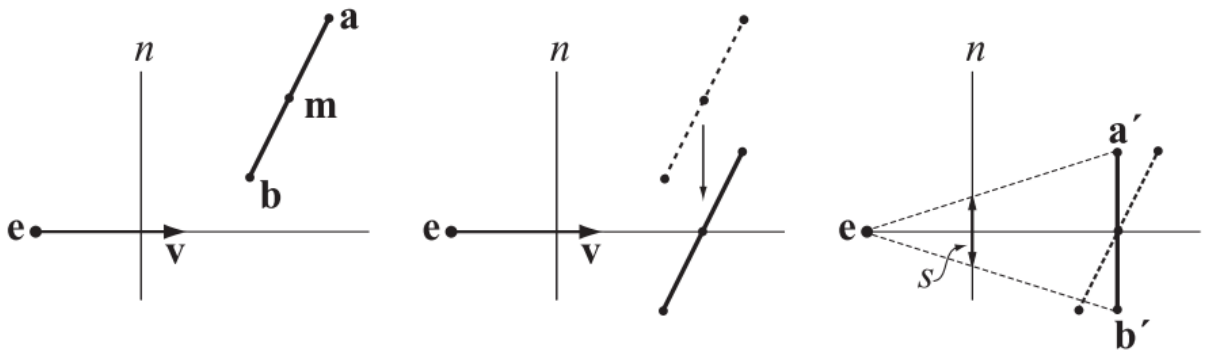


图 17.61：估计一条线段在屏幕空间中的投影 s 。

实现屏幕空间覆盖率（screen-space coverage）的方法有很多（详见章节 19.9.2）。所有的这些方法都是将一些简单的对象投影到屏幕上，并估计投影在屏幕空间中的长度或者面积。较大的面积或者较长的长度意味着需要进行曲面细分。图 17.61 展示了一条从点 $\mathbf{a}$ 到点 $\mathbf{b}$ 的线段，以及它在屏幕空间中投影的快速估计。首先，对这条线段进行平移，使其中点位于观察射线上。然后，假设这条线段平行于近裁剪平面 n ，并根据这条线段来计算屏幕空间中的投影 s 。使用图 17.61 右侧线段上的点 $\mathbf{a}'$ 和点 $\mathbf{b}'$ ，这个屏幕空间中投影 s 的计算方法如下：

$$s = \frac{\sqrt{(\mathbf{a}' - \mathbf{b}') \cdot (\mathbf{a}' - \mathbf{b}')}}{\mathbf{v} \cdot (\mathbf{a}' - \mathbf{e})} \tag{17.71}$$

方程 17.71 中的分子部分就是线段的长度，将其除以眼睛（点 $\mathbf{e}$ ）到线段中点的距离。然后将计算出的屏幕空间投影 s 与表示屏幕空间中最大边缘长度的阈值 t 进行比较。可以对方程 17.71 进行一些改写，从而避免计算平方根，如果满足以下条件，则继续进行曲面细分：

$$s > t \Longleftrightarrow (\mathbf{a}' - \mathbf{b}') \cdot (\mathbf{a}' - \mathbf{b}') > t^2 \left(\mathbf{v} \cdot (\mathbf{a}' - \mathbf{e})\right)^2 \tag{17.72}$$

方程中的 t^2 是一个常数，因此可以进行预先计算。对于分数曲面细分而言，可以使用方程 17.71 中的 s ，并对其进行一些缩放来作为实际使用的曲面细分率。另一种测量投影边缘长度的方法是，在这条边的中心处放置一个球体，使这个球体的半径为边缘长度的一半，然后再使用球体的投影作为这条边上的曲面细分因子[1283]。这个球体测试与面积成正比，而上面的测试则与边长成正比。

增加轮廓处的曲面细分率是很重要的，因为它们对于物体的感知质量起着主要作用。可以将点 $\mathbf{a}$ 处的法线与从眼睛指向点 $\mathbf{a}$ 的向量进行点积，并判断点积结果是否接近

于零，从而确定这个三角形是否靠近轮廓边缘。如果这个条件对点 $\mathbf{a}$ 、 $\mathbf{b}$ 或 $\mathbf{c}$ 中的任何一个都成立，则应当对这个三角形进行进一步的曲面细分。

对于位移细分，Nießner 和 Loop [1281]对每个基底网格顶点 $\mathbf{v}$ 都使用了下列因子中的其中一个，这个顶点 $\mathbf{v}$ 连接了 n 个边向量 $\mathbf{e}_i$ ，其中 $i \in \{0, 1, \ldots, n-1\}$ ，这些因子的计算方法为：

$$
\begin{aligned}
f_1 &= k_1 \cdot \|\mathbf{c} - \mathbf{v}\|, \\
f_2 &= k_2 \sqrt{\sum \mathbf{e}_i \times \mathbf{e}_{i+1}}, \\
f_3 &= k_3 \max\left(\|\mathbf{e}_0\|, \|\mathbf{e}_1\|, \ldots, \|\mathbf{e}_{n-1}\|\right),
\end{aligned}
\tag{17.73}
$$

其中 i 是循环索引，对连接到点 $\mathbf{v}$ 的所有 n 条边 $\mathbf{e}_i$ 进行了遍历；点 $\mathbf{c}$ 是相机所在的位置； k_i 是用户提供的常数。在方程 17.73 中，因子 f_1 仅仅基于了从相机到顶点的距离， f_2 计算连接到点 $\mathbf{v}$ 的四边形的面积， f_3 则使用了最大的边缘长度。然后会分别对边缘上的两个顶点，计算上述这些顶点的曲面细分因子，最终这两个顶点所使用的曲面细分因子为 6 个（ 3×2 ）曲面细分因子中的最大值。同时，可以使用对边的曲面细分因子中的最大值，来作为内部的曲面细分因子。这种方法可以与本小节中所介绍的任何边缘曲面细分因子方法一起使用。

值得注意的是，Nießner 等人[1279]建议对字符使用单个的全局曲面细分因子，并根据相机到字符的距离来计算这个因子。具体的细分数量为 $\lceil \log_2 f \rceil$ ，其中 f 是每个字符的曲面细分因子，可以使用上述任何一种方法进行计算。

很难说有哪一种方法适用于所有的应用程序，因此最好是对现有的几种启发式方法，以及它们的组合方法都进行实际测试。

分割和骰子方法

Cook 等人[289]引入了一种被称为分割和骰子（split and dice）的方法，该方法的目标是对曲面进行细分，并使得每个三角形的大小与像素的大小相同，从而避免几何锯齿。为了能够实时进行处理，应当增大这个曲面细分的阈值使得 GPU 能够进行实时处理。首先，每个面片会被递归地分割成一组子面片并同时进行估计，直到对某个子面片使用均匀曲面细分能够得到所需大小的三角形位置。因此，这也是一种自适应的曲面细分。

想象现在有一个较大的面片被用作景观表示。一般来说，分数曲面细分是无法适应这种情况的，例如：在靠近相机的地方应当有更高的曲面细分率，而在远离相机的地方应当有较低的曲面细分率。这种分割和骰子方法的核心思想对于实时渲染可能会很有

用，即使在现在的情况下（性能不够），我们的目标曲面细分率是创建比像素尺寸更大的三角形。

接下来，我们将介绍实时图形场景中分割和骰子的一般方法。这里我们假设使用的是矩形面片。然后在整个参数化域上，即从 $(0,0)$ 到 $(1,1)$ 的方块，开始执行一个递归程序。使用刚才描述的自适应终止准则来判断当前表面是否被足够细分，如果是，则终止曲面细分；如果不是，则将该定义域划分为四个大小相同的正方形，并对每个子正方形继续调用这个递归程序。持续执行这个过程，直到当前表面被充分细分，或者是达到预定的递归级别。该算法的本质是在曲面细分的过程中，递归地创建四叉树结构。然而，如果相邻的子方块被细分到不同的层次水平，很可能会产生裂缝。标准的解决方案是，确保两个相邻的子方块最多只相差一个细分级别，这种结构被称为受限四叉树（restricted quadtree）。然后使用图 17.59 中所展示的技术来填充这些裂缝。这种方法的缺点在于，会涉及较多的统计工作（bookkeeping）。

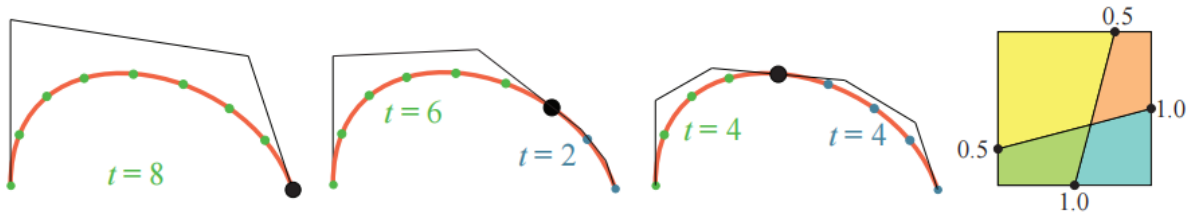


图 17.62：对一条三次 Bezier 曲线应用分数分割。每条曲线旁都显示了对应的曲面细分率 t。图中的黑色圆点是分裂点，它从曲线的右侧出发，向着曲线的中心进行移动。为了能够对三次曲线进行分数分割，这个分裂点会平滑地移向曲线的中心，并使用两段三次 Bezier 曲线来代替原始曲线，它们共同组成了原始曲线。在右侧，同样的概念被用于分割面片，它被分割成四个更小的子面片，其中 1.0 代表分割点位于边缘的中心点上，0.0 代表分割点位于面片的拐角处。[1044]

Liktor 等人[1044]提出了分割和骰子的变体方法，可以使用 GPU 来执行这个过程。当在执行过程中突然决定要再分裂一次，其关键在于要避免出现游动瑕疵和突变效果，例如：相机已经移动到了一个更加接近表面的位置。为了解决这个问题，他们使用了分数分割（fractional split）方法，其灵感来自分数曲面细分，如图 17.62 所示。由于这个分裂是从曲线的一侧平滑引入的，并逐渐过度到曲线的中心（或者面片的中心），因此可以避免出现游动瑕疵和突变瑕疵。当达到自适应曲面细分的终止标准时，GPU 也会使用分数曲面细分来对每个剩余的子面片进行细分。

17.6.3 快速 Catmull–Clark 曲面细分

Catmull–Clark 表面（章节 17.5.2）经常被用于建模软件和电影渲染中，因此，能够利用图形硬件来高效渲染这些表面是很有吸引力的。Catmull–Clark 曲面的快速曲面细分方法是近年来一个十分活跃的研究领域。这里我们将介绍其中的一些方法。

近似方法

Loop 和 Schaefer [1070]提出了一种技术，可以将 Catmull–Clark 曲面转换为一种表示形式，这种表示可以在域着色器中进行快速计算，并且不需要知道多边形的相邻信息。

在章节 17.5.2 中我们提到，当多边形网格中的所有顶点都是普通顶点的时候，Catmull–Clark 表面可以被描述为许多较小的 B 样条曲面。Loop 和 Schaefer 将原始 Catmull–Clark 细分网格中的四边形（quad）转换为一个双立方 Bezier 曲面（章节 17.2.1）。这种做法对于非四边形来说是不可能的，因此这里我们假设不存在这样的多边形（回顾一下，在第一步细分之后，Catmull–Clark 表面中的多边形都是四边形）。如果一个多边形网格中包含非 4 价的顶点，那么就无法创建一个与 Catmull–Clark 表面弯曲相同的双三次 Bezier 面片。因此，他们提出了一种近似表示方法，这个方法适用于 4 价顶点的四边形，并且在其他地方与 Catmull–Clark 表面十分接近。为了达到这个目的，我们会同时使用几何面片（geometry patch）和切线面片（tangent patch），下面将对此进行描述。

几何面片是一个简单的双三次 Bezier 面片，它具有 4×4 个控制点。这里我们将描述这些控制点是如何进行计算的。一旦我们有了这样一个双三次 Bezier 面片，我们就可以对这个面片进行细分，使用域着色器可以在任何参数化坐标 (u, v) 处对 Bezier 面片进行快速计算。因此，假设现在有一个仅由 4 价顶点四边形所组成的网格，我们想要计算网格中某个四边形所对应的 Bezier 面片控制点。为此，我们需要知道这个四边形的相邻四边形的信息。这一操作的标准方法如图 17.63 所示，其中给出了三种不同的模板。这些模板可以进行可以旋转和反射，从而创建所有的 16 个控制点。请注意，在实践中，模板的权重之和应该为 1，这里为了清楚起见，省略了这个归一化过程。

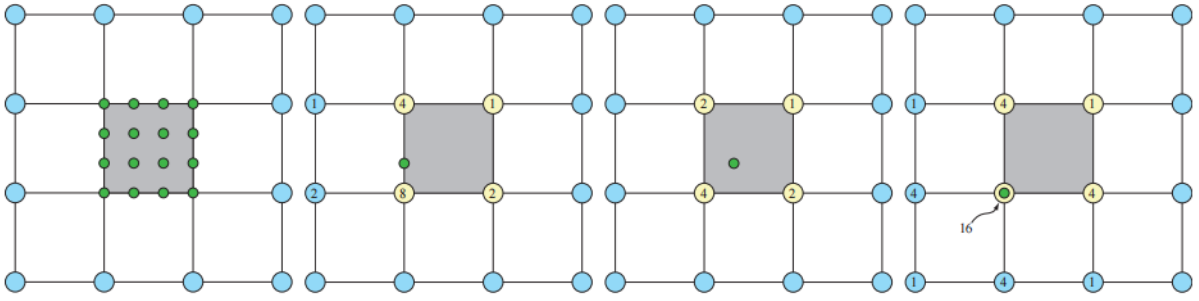


图 17.63：左：四边形网格的其中一部分，我们想为图中的灰色四边形计算一个 Bezier 面片。请注意，这个灰色四边形的顶点都是 4 价的。图中的蓝色顶点是周围相邻四边形上的顶点；图中的绿色圆圈是 Bezier 面片的控制点。之后三幅插图展示了用于计算绿色控制点的不同模板。例如：如果我们想要计算一个内部控制点，则会使用第三幅图中的模板，并根据模板中显示的权重来对四边形的顶点进行加权。

上述技术是计算一个 Bezier 面片的普通情况。当网格中存在一个异常顶点的时候，我们会计算一个异常面片[1070]。图 17.64 展示了这种情况下的模板，其中灰色四边形的左下角顶点就是一个特殊顶点。

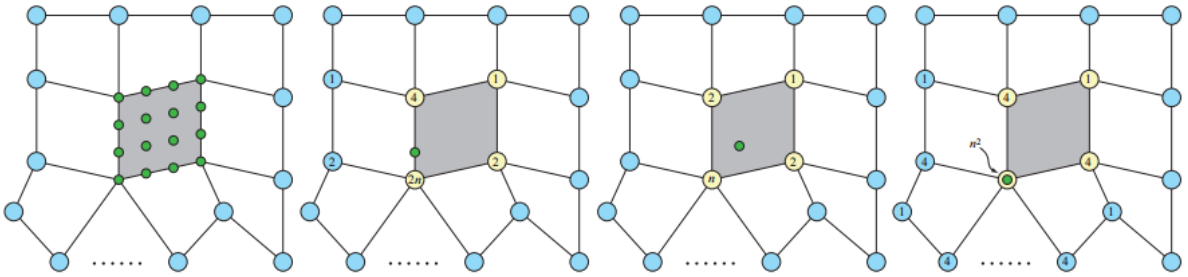


图 17.64：左：为网格中的灰色四边形生成一个 Bezier 面片。这个灰色四边形中的左下角顶点非常特别，因为它是 5 价的，并不是 4 价的。图中蓝色顶点是周围相邻四边形的顶点；绿色圆圈是 Bezier 面片的控制点。右边三幅插图展示了用于计算绿色控制点的不同模板。

请注意，这将产生一个近似于 Catmull–Clark 细分表面的面片，并且在特殊顶点的边缘处只有 C^0 连续性。当对这种表面进行着色的时候，出现的一些小瑕疵可能会分散观众的注意力，因此建议使用类似于 N 面片的技巧（章节 17.2.4）。然而，为了降低计算的复杂度，我们推导出了两个切线面片：一个在 u 方向上，另一个在 v 方向上。而表面法线就是这些向量的叉乘结果。一般来说，Bezier 面片的导数可以使用方程 17.35 进行计算。然而，由于生成的 Bezier 面片近似于 Catmull–Clark 表面，因此切线面片之间并不会形成连续的法线场。关于如何克服这些问题，详见 Loop 和 Schaefer 的论文[1070]。图 17.65 展示了一类可能会发生的瑕疵。

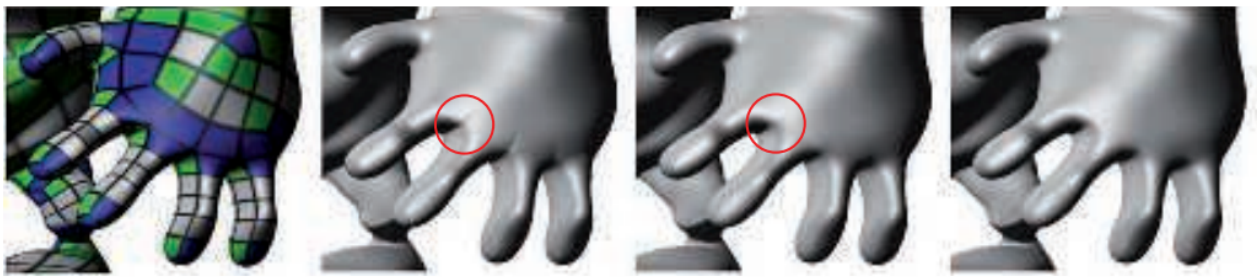


图 17.65：图片 1：展示了网格的四边形结构。其中白色四边形是普通四边形（顶点都是 4 价的），绿色四边形中包含一个特殊顶点，蓝色四边形则包含多个特殊顶点。图片 2：几何面片近似。图片 3：带有切线面片的几何面片。请注意，图中红色圆圈处的着色瑕疵消失了。图片 4：真正的 Catmull–Clark 表面。

Kovacs 等人[931 描述了如何对上述方法进行扩展，从而来处理折痕和拐角（章节 17.5.3），并在 Valve 的起源（Source）引擎中实现这些扩展方法。

特征自适应细分和 OpenSubdiv

皮克斯展示了一个名为 OpenSubdiv 的开源系统，它实现了一套被称为特征自适应细分（feature adaptive subdivision，FAS）的技术[1279, 1280, 1282]。其基本方法与前面所讨论的技术有很大不同。这项工作的基础在于，对于规则表面的细分相当于双三次 B 样条面片（章节 17.2.6），这里的规则表面是指网格中的四边形顶点都是规则的（4 价）。因此，只会对非规则表面进行递归细分，直到达到某个预设的最大细分层级为止，如图 17.66 左侧所示。FAS 还可以处理折痕和半光滑的折痕[347]，并且 FAS 算法也需要对这些折痕进行细分，如图 17.66 右侧所示。其中的双三次 B 样条面片可以使用曲面细分管线直接进行渲染。

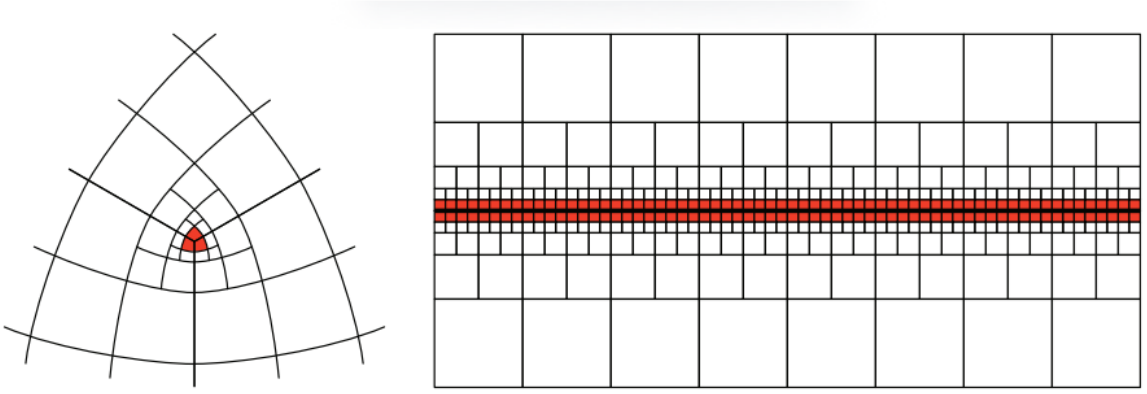


图 17.66：左：围绕着一个特殊顶点进行递归细分，这个位于最中间的特殊顶点具有三条边。随着细分递归的进行，它会留下一组规则面片（包含四个顶点，每个顶点都为 4 价）。右：中间的加粗线条代表了对这个平滑折痕周围的细分。[1279]

该方法首先使用 CPU 创建一个表格，这个表格将细分过程中需要访问的顶点索引编码到一个指定的级别中。由于索引与顶点位置无关，因此这个基本网格可以被动画化。一旦生成了一个双三次 B 样条面片，就不再需要递归了，这意味着这个表格通常会相对较小。这个基本网格、带有索引和附加顶点价信息的表格、以及折痕数据只需要上传到 GPU 中一次即可。

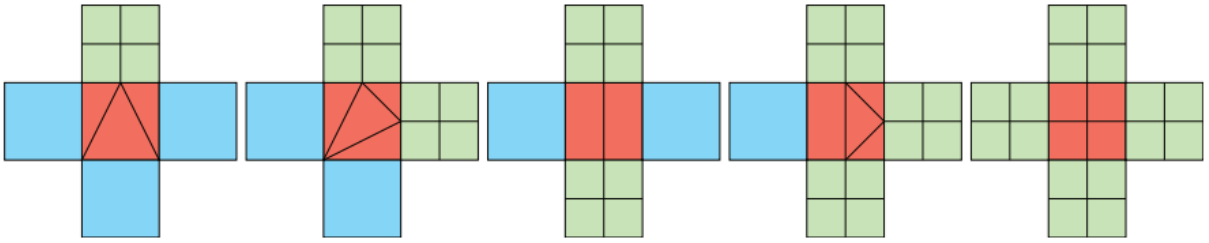


图 17.67：图中的红色方块是过渡区域，它具有四个相邻的区域，分别是蓝色区域（当前细分层）和绿色区域（下一个细分层）。这幅插图展示了可能发生的五种情况，以及它们是如何被拼接在一起的。[1279]

为了对网格进行一步细分，首先要计算新的表面点，然后计算新的边缘点，最后再对顶点进行更新，每种类型的点都使用一个计算着色器来执行。对于渲染而言，要对完整面片（full patch，FP）和过渡面片（transition patch，TP）进行区分。一个完整面片只与相同细分水平的面片共享边缘，同时一个规则的完整面片会直接使用 GPU 的曲面细分管线，将其渲染为一个双三次 B 样条面片，否则将会继续进行细分。这个自适应细分过程会确保相邻面片之间最多只有一个细分级别的差异。而一个过渡面片则对至少一个邻居具有细分级别上的差异。为了获得没有裂缝的渲染效果，每个过渡面片会被划分成若干个子面片，如图 17.67 所示。这样，细分顶点就会沿着每条边缘的两侧相匹配。每种类型的子面片都会使用不同的壳着色器和实现了插值变体的域着色器来进行渲染。例如：对于图 17.67 中最左边的情况，它会被渲染为三个三角形 B 样条补丁。而在异常顶点周围则会使用另一个域着色器，并使用 Halstead 等人[655]的方法来计算极限位置和极限法线。图 17.68 展示了一个使用 OpenSubdiv 的 Catmull–Clark 表面渲染的结果。

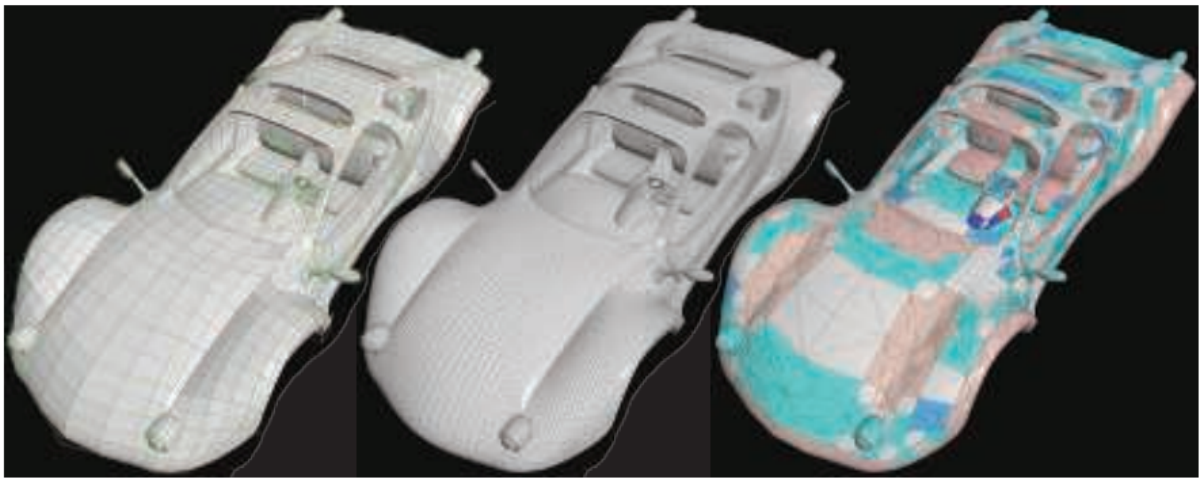


图 17.68：左：控制网格的线框使用绿色线条和红色线条进行表示，表面使用灰色进行表示（8k 个顶点），其中的红色线条是使用一个细分步骤生成的。中：网格进行了额外两个步骤的

细分（102k 个顶点）。右：使用自适应曲面细分生成的表面（28k 个顶点）。

FAS 算法可以处理折痕、半平滑折痕、分层细节以及自适应 LOD。我们推荐参考 FAS 的论文[1279]，以及 Nießner 的博士论文[1282]来了解更多细节。Schafer 等人[1547]提出了 FAS 的一种变体，被称为 DFAS，其速度更快。

自适应四叉树

Brainerd 等人[190]提出了一种被称为自适应四叉树（adaptive quadtree）的方法。它类似于 Loop 和 Schaefer [1070]的近似方案，即在原始基础网格的每个四边形上都提交一个细分图元。此外，该方法还预计算了一个细分计划，这是一个四叉树结构，它从一个输入表示来编码分层细分（类似于特征自适应细分），直到某个最大细分层级。这个细分计划同样还包含细分表面所需要的控制点模板列表。

在渲染过程中会遍历这个四叉树，这使得可以将 (u, v) 坐标映射到一个细分层次结构中的面片，这个面片可以直接进行计算。这个四叉树的叶子节点是原始表面域的一个子区域，该子区域中的表面可以使用模板中的控制点直接进行计算。会使用一个迭代循环来遍历域着色器中的四叉树，这个域着色器的输入是一个参数化的 (u, v) 坐标。需要持续进行遍历，直到到达 (u, v) 坐标所在的叶子节点。。根据所到达的四叉树节点类型会采取不同的操作，例如：当到达一个可以直接进行计算的子区域时，它所对应的双三次 B 样条面片的 16 个控制点会被检索，然后着色器继续对这个面片进行计算。

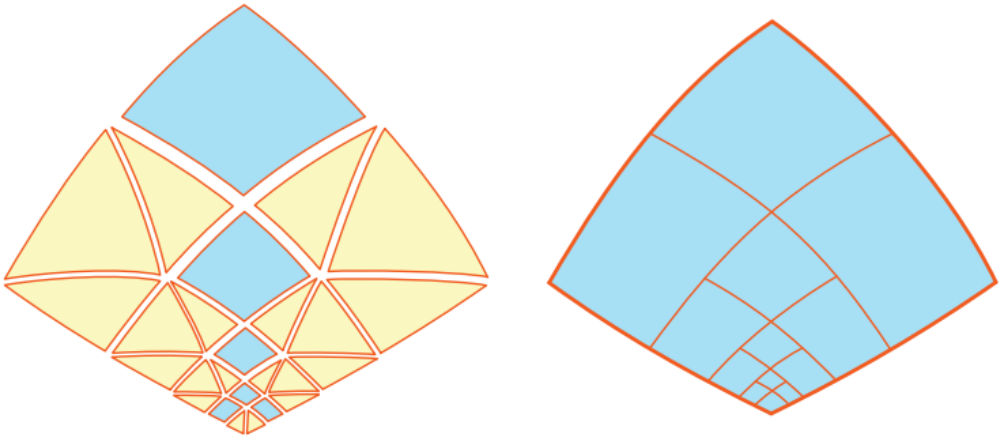


图 17.69：左：根据特征自适应细分（FAS）进行分层细分，其中每个三角形和每个四边形都会被渲染为单独的细分图元。右：使用自适应四叉树进行分层细分，其中整个四叉会被渲染为单个细分图元。[190]

图 17.1 展示了一个使用这种技术进行渲染的例子。该方法是迄今为止速度最快的，并且能够准确地渲染 Catmull–Clark 细分曲面的方法，同时还能够处理折痕以及其他拓

扑特征。图 17.69 展示了使用自适应四叉树相对于 FAS 的另一个优势，图 17.70 进一步说明了这一点。自适应四叉树还能够提供更加均匀的细分结果，因为每个提交的四叉树与细分图元之间具有一对一的映射关系。

图 17.70：使用了自适应四叉树的细分面片。每个面片都有一个对应的基础网格面片，在图中使用黑色曲线进行表示，每个面片内部的分层结构说明了细分步骤。我们可以看到，在中心有一个颜色均匀的面片（灰紫色）。这意味着它被渲染为一个双三次 B 样条面片，而其他（具有特殊顶点）的面片则清楚地展示了其潜在的自适应四叉树结果。

补充阅读和资源

曲线和曲面是一个巨大的主题，想要获得更多有用的信息，最好是参考一些专门讨论这个主题的书籍。Mortenson 的书籍[1242]很好地介绍了几何建模。Farin 的书籍[458, 460]，以及 Hoschek 和 Lasser 的书籍[777]都是概括性的，这些书籍讨论了计算机辅助几何设计（Computer Aided Geometric Design，CAGD）的许多方面。对于隐式表面，请参阅 Gomes 等人[558]的书籍，以及 de Araujo 等人[67]的最新论文。有关细分曲面的更多信息，请参阅 Warren 和 Heimer 的书籍[1847]，以及 Zorin 等人[1977]有关“建模和动画的细分（Subdivision for Modeling and Animation）”的 SIGGRAPH 课程说明。Ni 等人[1275]的关于细分曲面替代品（substitute）的课程也是一个十分有用的资源。Nießner 等人[1283]的调研，以及 Nießner 的博士论文[1282]，对于使用 GPU 实时渲染细分表面很有帮助。

对于样条插值，除了上述 Farin[458]、Hoschek 和 Lasser 的书籍[777]之外，我们还建议感兴趣的读者参考 the Killer B 的书籍[111]。对于曲线和曲面，Bernstein 多项式的许多性质都是由 Goldman [554]给出的。几乎所有你需要知道的，关于三角形

Bezier 曲面的知识，都可以在 Farin 的文章[457]中找到。另一类有理曲线和有理曲面是非均匀的有理 B 样条（nonuniform rational B-spline，NURBS）[459, 1416, 1506]，它常用于 CAD 领域中。

Chapter 18 Pipeline Optimization 管线优化

> Donald Knuth——“We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil.”

> 唐纳德·克努特——“我们应当在 97% 的时间中忘记优化：过早优化是万恶之源。”（美国计算机科学家，1974 年图灵奖获得者；1938—）

在本书中，算法都是在质量、内存、性能权衡的上下文中进行介绍的。而在本章节中，我们将讨论与特定算法无关的性能问题和优化机会。本章节的重点在于性能瓶颈的检测和优化，我们首先会介绍一些小的、局部的更改优化，直到将应用程序作为一个整体，来利用多处理功能。

我们在第 2 章中介绍了，渲染图像的过程是基于流水线架构的，主要有四个概念阶段：应用阶段、几何处理阶段、光栅化阶段和像素处理阶段。总有一个阶段会成为瓶颈，即流水线中最慢的那个步骤。换句话说，这个瓶颈阶段决定了整个管线的吞吐量极限（即总的渲染性能），因此性能瓶颈是优化（optimization）的主要候选项。

对渲染管线的性能进行优化，类似于流水线处理器（CPU）[715]的优化过程，因为它主要包括两个步骤。第一，我们需要对管线的性能瓶颈进行定位。第二，当这个阶段得到某种程度上的优化之后，如果还是没有达到优化目标，那么会再次重复第一步。需要注意的是，在优化步骤之后，瓶颈可能会出现在另外一个地方。最好只在瓶颈阶段投入足够的优化精力，这样瓶颈可能就会转移到另一个阶段中。在这个阶段再次成为性能瓶颈之前，可能还需要对其他几个阶段进行优化。因此，我们不应当把精力浪费在对某个单一阶段进行过度优化（over-optimizing）上。

性能瓶颈的位置可能会在一帧中，甚至在一次 draw call 中发生变化。在某个时刻，几何阶段可能会成为性能瓶颈，因为此时需要渲染许多微小的三角形。之后的像素处理阶段也可能会成为性能瓶颈，因为在每个像素上都会计算很繁重的程序化着色器。在像素着色器中，执行过程可能会因为纹理队列已满而发生停滞，或者在到达特定循环或者分支时花费更多的处理时间。所以，如果我们说此时应用程序阶段是性能瓶

颈，我们的意思是指，在这一帧中的大多数时间内，应用阶段都是性能瓶颈，但是性能瓶颈通常都会有很多个。

由于渲染管线是一种流水线结构，充分利用流水线结构的另一种方法是认识到，如果最慢的那个阶段已经无法再进一步优化时，可以适当调整其他阶段，让它们的速度与最慢的那个阶段相匹配。这样做并不会改变性能表现，因为流水线中的最慢阶段并没有改变效率，但是可以使用额外的计算处理，来在有限的性能下尽可能提高图像质量[1824]。例如：假设现在的瓶颈位于应用阶段，生成一帧画面需要 50 ms，而其他的每个阶段都只需要 25 ms。这意味着在不改变渲染管线速度的情况下（50 ms 的帧渲染时间，相当于每秒 20 帧），几何阶段和光栅化阶段在 50 ms 内完成它们的工作就可以了。例如：我们可以使用更加复杂的光照模型，或者是增加阴影和反射的真实感，当然这个前提是这些操作不会再增加应用阶段的工作量。

计算着色器的出现和使用，也改变了我们对于瓶颈和未使用资源的看法。例如：假设我们正在渲染一张阴影贴图，那么此时顶点着色器和像素着色器的任务十分简单，如果此时一些固定功能的管线阶段（例如光栅化或者像素合并）成为性能瓶颈，那么 GPU 的计算资源可能得不到最充分的利用。当出现这样的情况时，可以使用异步的计算着色器来重叠绘制（overlapping draw），从而使得着色器单元保持时刻忙碌[1884]。本章的最后一小节中，我们还会讨论基于任务的多处理（task-based multiprocessing）。

流水线优化是一个过程，在这个过程中，我们首先要最大化渲染速度（优化瓶颈），然后再允许那些非瓶颈阶段与瓶颈阶段消耗一样多的时间。也就是说，这这并不总是一个简单的过程，因为不同的 GPU 和驱动程序可能都会有自己的特性和快速路径。在阅读这一章的时候，你要牢记一句格言（了解你的体系结构）：

$$KNOW\ \ YOUR\ \ ARCHITECTURE$$

因为在不同的体系结构中，相应的优化技术会有很大差别。也就是说，要警惕基于特定的 GPU 硬件来对功能实现进行优化，因为硬件并不是一成不变的，它可能会随着时间的推移而变化[530]。另一个相关的格言是（测量，测量，再测量）：

$$MEASURE, MEASURE, MEASURE.$$

18.1 分析和调试工具

分析（profiling）工具和调试（debugging）工具对于发现代码中的性能问题十分有用，这些工具的能力各不相同，包括：

- 帧捕获与可视化。通常可以对某一帧画面进行分步回放，并显示相应的状态和资源使用显示情况。
- 分析 CPU 和 GPU 的运行时间，包括调用图形 API 所花费的时间。
- 着色器调试，以及可能的热编辑功能（hot editing），它允许在修改代码之后，立即查看对应的效果。
- 在应用程序中设置调试标记（debug marker），从而帮助开发人员识别代码区域。

这些分析和调试工具会因操作系统、图形 API 以及 GPU 硬件厂商而异。对于上述功能的大多数组合都有相应的工具，这也就是为什么上帝（God）创造了谷歌（Google）的原因（译者注：一个梗，类似于“内事不决问百度，外事不决问谷歌，房事不决问天涯”）。下面我们将介绍一些专门用于交互式图形程序的软件名称，以帮助您开始性能优化任务：

- RenderDoc 是一个高质量的 Windows，Linux，Android 调试器，它可以用于 DirectX，OpenGL 和 Vulkan。它最初是由 Crytek 开发的，现在已经开源了。
- GPU PerfStudio 是 AMD 为其图形硬件提供的专属工具套件，可在 Windows 和 Linux 上运行。它提供了一个值得注意的工具，即一个静态着色器分析器，它可以在不运行程序的情况下对性能进行评估。AMD 的 Radeon GPU Profiler 是一个独立的、类似的工具。
- NVIDIA 的 Nsight 是一个具有广泛功能的性能和调试系统。它集成在了 Windows 上的 Visual Studio 中，以及 Mac OS 和 Linux 上的 Eclipse 中。
- Microsoft 的 PIX 长期以来一直被 Xbox 开发人员所使用，并被带回到了 Windows 上的 DirectX 12。Visual Studio 中的 Graphics Diagnostics 功能可以用于早期版本的 DirectX。
- Microsoft 的 GPUView 使用了 ETW （Event Tracing for Windows），这是一个高效的事件日志系统。GPUView 是几个使用 ETW 会话的程序之一，它重点关注 CPU 和 GPU 之间的交互，并可以表明具体哪个是瓶颈[783]。
- Graphics Performance Analyzer （GPA）是英特尔的一个套件，它专注于性能和框架分析，同时并不仅局限于他们的图形芯片。

- OSX 上的 Xcode 提供了 Instruments，其中有一些用于计时、性能、网络、内存泄漏的工具。值得一提的是 OpenGL ES Analysis，它可以检测性能和正确性问题，并可以提出一些解决方案。还有 Metal System Trace，它可以提供来自应用程序、驱动程序和 GPU 中的追踪信息。

上述这些工具是一些主要使用的工具，并且已经存在好几年了。但是，有时候也可能会没有工具能够完成我们想实现的目的。大多数 API 都内置了计时器查询（timer query）工具，从而帮助分析 GPU 的性能。一些供应商也提供了相应的库，来访问 GPU 计数器和线程追踪等。

18.2 定位性能瓶颈

优化管线的第一步是找到最大性能瓶颈的所在处[1679]。寻找瓶颈的一种方法是设置多个测试，其中每个测试都会减少特定阶段所执行的工作量。如果其中一个测试会导致帧率上升，那么我们就发现了一个瓶颈阶段。还有一个类似的方法可以对某一个阶段进行测试，即减少其他阶段的工作量，同时保持被测阶段的工作量不变。如果此时的性能表现没有发生改变，那就说明当前被测阶段是性能瓶颈。一些性能工具可以提供详细的 API 调用开销，并能够指出哪些 API 调用的开销最大，但是它不一定能够准确指出管线中的哪个阶段是性能瓶颈。即使这些性能工具能够实现这一点，去理解每个测试背后的思想也是十分有用的。

下面我们对用于测试各个阶段的一些想法进行简要讨论，来说明具体如何进行这种测试。统一着色器架构（unified shader architecture）的出现是理解底层硬件重要性的一个完美例子。从 2006 年底开始，它逐渐成为了许多 GPU 的硬件基础，其核心思想是，顶点着色器、像素着色器以及其他的一些着色器，都使用相同的硬件功能单元进行实现。GPU 负责完成负载均衡（load balancing），并动态改变分配给顶点着色器和像素着色的计算单元比例。举个例子：如果我们现在渲染一个较大的四边形，那么只有少数几个着色器单元会分配给顶点变换，而大部分的着色器单元都会分配给片元处理任务。想要确定瓶颈位于顶点着色阶段还是在像素着色阶段就不那么容易了[1961]。无论是整个着色器处理，还是管线中的其他功能阶段，都将可能会成为性能瓶颈的所在处，因此我们会依次对每种可能性进行讨论。

18.2.1 应用阶段测试

如果我们所使用的平台可以提供用于测量处理器工作负载的实用程序，那么我们可以使用这个程序来查看我们的图形程序是否使用了 100%（或者接近 100%）的 CPU 处理能力。如果 CPU 的占用率一直很高，那么这个程序可能会是 CPU 瓶颈的（CPU–

limited）。这种方法也并不是万无一失的，因为有时应用程序可能会等待 GPU 渲染完一帧。我们说一个程序是 CPU 瓶颈或者 GPU 瓶颈，但是这个瓶颈可能会在一帧的生命周期中发生变化。

一种测试 CPU 瓶颈的巧妙方法是，只向 GPU 发送一些导致其很少工作或者根本不工作的数据。在某些系统中，这个操作可以直接通过使用一个空的驱动程序（一个接受调用，但是不执行任何操作的驱动程序）代替真正的驱动程序来进行实现。这实际上是为整个程序的运行速度设置了上限，因为我们既不使用图形硬件，也不调用驱动程序，也就是说，一个运行在 CPU 上的应用程序，始终是 CPU 瓶颈的。通过这种测试，我们可以了解到在应用程序阶段之外，剩余基于 GPU 的阶段还有多少改进优化的空间。同时需要注意，由于驱动程序还需要处理自身的一些任务，以及完成 CPU 和 GPU 之间的通信，因此使用一个空的驱动程序可能会隐藏这些由于驱动程序而造成的瓶颈。驱动程序通常是造成 CPU 端瓶颈的原因，我们将在后续内容中深入讨论这个话题。

如果可能的话，另一种更加直接的方法是降低 CPU 的时钟频率[240]，CPU 频率的降低意味着 CPU 性能的降低。如果此时应用程序的性能降低与 CPU 的频率降低成正比，那么这个应用程序至少在某种程度上是与 CPU 性能相绑定的。GPU 同样也可以采用类似的降频（underclock）方法。如果 GPU 频率降低会导致应用程序的性能成比例的降低，那么至少在某些时候，这个应用程序是与 GPU 性能相绑定的。这些降频方法可以帮助识别性能瓶颈，但是有时候也会导致之前不是瓶颈的阶段变成瓶颈。另一个相对的选择是超频（overclock），但是这里没有提及。

18.2.2 几何处理阶段测试

几何阶段是最难进行测试的阶段。这是因为如果几何阶段的工作负载发生了变化，那么其他一些阶段的工作负载也经常会发生相应的变化。为了避免这个问题，Cebenoyan [240]给出了一系列从光栅化阶段回归到管线的测试。

在几何阶段中，瓶颈主要出现会在这两个方面：顶点获取和顶点处理。如果想要知道瓶颈是否来自于模型的数据传输，我们可以增大顶点格式的大小。例如：我们可以在每个顶点上发送一些额外的纹理坐标来增大顶点数据，如果性能出现下降，则说明这个阶段就是性能瓶颈。

顶点处理由顶点着色器完成。对于顶点着色器的瓶颈测试，可以让着色器程序变得更长更复杂。但是要小心确保编译器不会优化掉这些额外的指令。

如果我们的管线中还使用了几何着色器，其性能会受到输出大小和程序长度的影响。如果我们使用了曲面细分着色器，其性能同样会受到程序长度的影响，以及还有曲面细分因子的影响。使用控制变量法，改变这些元素中的其中一个，同时尽量保证其他阶段的工作负载不变，可以我们帮助确定这些元素是否是导致性能瓶颈的原因。

18.2.3 光栅化阶段测试

这个阶段包括了三角形设置和三角形遍历两个子阶段。在生成阴影贴图的时候，我们会使用极其简单的像素着色器程序，因此可能会在光栅化阶段或者合并阶段出现瓶颈。还有一种情况可能会使三角形设置和光栅化阶段成为性能瓶颈，即渲染那些来自曲面细分或者类似物体（例如草叶和树叶）的微小三角形，当然这种情况通常很少会发生[1961]。并且，这些微小三角形也会增加顶点着色器和像素着色器的开销。在某个给定区域中添加更多数量的顶点，显然会增加顶点着色器的工作负载，同时也会增加像素着色器的工作负载，因为每个三角形都会以 2×2 的四边形为一组来进行光栅化，因此小三角形会增加每个三角形外部的辅助像素数量[59]，有时这也会被称为四边形过度渲染（quad overshading，章节 23.1）。为了确定光栅化是否真的为瓶颈，可以通过增大顶点着色器和像素着色器的程序大小，来延长它们的执行时间。如果每帧的渲染时间没有增加，那么就说明瓶颈位于光栅化阶段。

18.2.4 像素处理阶段测试

像素着色程序的影响可以通过改变屏幕分辨率来进行测试。如果以一个较低的屏幕分辨率来进行渲染会导致帧率明显上升，那么说明像素着色器很可能会是性能瓶颈，至少在某些时候是这样的。如果有一个设计良好的系统，那么就需要当心，因为较小的屏幕分辨率也可能会导致使用简化模型来进行渲染，从而降低几何处理阶段的负载。

直接降低显示分辨率可能还会影响三角形遍历、深度测试和混合、以及纹理访问等方面的开销。为了避免这些因素的影响，并对瓶颈进行隔离，可以使用一种与顶点着色程序测试相同的方法，即添加更多的指令来观察对执行速度的影响。同样地，需要确保这些额外的指令，不会被编译器优化掉。如果此时的帧渲染时间增加了，那么说明像素着色器可能就是瓶颈（或者至少在执行成本增加的时候成为了瓶颈）。另外，可以将像素着色器中的指令简化到最少程度，这一点在顶点着色器中通常是很难做到的。如果此时的帧渲染时间减少了，那么说明像素着色器可能就是瓶颈。纹理缓存未命中的开销也可能会很高，我们可以尝试使用一个 1×1 分辨率的纹理来代替原来的纹理，如果这样做可以提供更快的性能表现，那么纹理内存的访问就是一个瓶颈。

着色器是一些独立的程序，有着自己独特的优化技术。Persson [1383, 1385]介绍了几种低层级（low-level）着色器优化方法，以及关于图形硬件如何发生演变，最佳实践如何发生变化的相关细节。

18.2.5 合并阶段测试

在这个阶段中会进行深度测试和模板测试，还会进行混合操作，并将幸存下来的结果写入缓冲区中。改变这些缓冲区的输出 bit 深度是一种改变此阶段带宽成本的方法，这样可以帮助我们查看这个阶段是否会成为瓶颈。为不透明对象开启 alpha 混合，或者使用其他的混合模式，也会影响光栅化操作中执行内存访问和处理的数量。

这个阶段可能会成为后处理 pass、阴影、粒子系统渲染的瓶颈。在较小的程度上，也可能会成为毛发渲染和草叶渲染的瓶颈，因为在这些渲染过程中，顶点着色器和像素着色器都很简单，它们所做的工作比较少。

18.3 性能测量

为了实现优化目的，我们需要对性能表现进行测量。这里我们会讨论一些有关 GPU 速度的不同指标。图形硬件制造商过去常常会提供峰值速率（peak rate），例如每秒处理的顶点数（vertices per second）和每秒处理的像素数（pixels per second）等，这些指标在开发过程中是很难达到的。此外，由于我们处理的是一个流水线系统，因此真正的性能表现并不像所列出的这些数字一样简单。这是因为在运行过程中，瓶颈的位置可能会随着时间的变化而变化，并且管线中的不同阶段在执行期间会以不同的方式进行交互。由于这种过程的复杂性，因此 GPU 在市场上的部分卖点是其物理特性，例如：内核的数量和时钟频率、内存的大小、速度和带宽等。

尽管如此，如果使用得当的话，GPU 计数器和线程追踪仍然是重要的诊断工具。如果某个给定部分的峰值性能是已知的，但是它的计数较低，那么这个区域不太可能会成为瓶颈。有些硬件供应商会将计数器数据表示为每个阶段的利用率百分比。这些值是在一段时间内给出的，在这段时间内瓶颈可能会发生移动，因此这些参数并不完美，但是它们对于瓶颈定位仍有很大帮助。

这种衡量性能的指标越多越好，但是即使是一些看似简单的物理指标也很难进行精确比较。例如：同一款 GPU 芯片的时钟频率在不同 IHV 合作伙伴（译者注：例如华硕、微星等厂商）之间可能也会有所不同，因为每个 IHV 都有着自己的散热方案，因此会将各自出厂的 GPU 超频到它们所认为安全的范围内，因此不同厂家的同一款型号显卡，出厂频率都有所不同。即使是在单个系统上进行 FPS 基准比较

（benchmark），也并不总是像听起来那么简单。NVIDIA 的 GPU Boost [1666]和 AMD 的 PowerTune [31]技术就是一个很好的例子，也非常符合我们之前提到的格言“know your architecture”。NVIDIA 的 GPU Boost 之所以会出现，部分原因就是一些复合的基准测试，这些基准测试会同时运行 GPU 管线中的许多部分，从而将 GPU 的功耗推到极限，这意味着 NVIDIA 不得不降低 GPU 的基本频率以防止芯片过热。大多数应用程序不会同时对管线的所有部分都进行这样的操作，因此在这些应用中，GPU 可以安全地以更高频率来运行。GPU Boost 技术会追踪 GPU 的功耗和温度特征，并相应地调整运行频率。AMD 和 Intel 对他们的 GPU 也有类似的功耗/性能优化。这种多变性可能会导致相同的基准测试以不同的速度来运行，具体会取决于 GPU 的初始温度。为了避免这个问题，微软在 DirectX 12 中提供了一种方法来锁定 GPU 核心的频率，从而获得稳定的时序[121]。其他的一些 API 也可以检查 GPU 的功率状态，但是要更加复杂[354]。

在测量 CPU 性能的时候，整体的趋势是避免使用每秒指令数（instructions per second，IPS）、每秒浮点操作数（floating point operations per second，FLOPS）、GHz（gigahertz）以及简单的短基准测试。相反，首选的方法是对一系列不同的实际程序进行测试，然后再比较这些程序的运行时间[715]。遵循这一趋势，大多数独立的图形基准测试都是针对几个给定场景、各种不同的屏幕分辨率、不同的抗锯齿设置和画面质量设置等，来测量这些程序的实际帧率。许多图形密集型（graphics-heavy）的游戏中都包含了基准测试模式，或者是由第三方创建的基准测试（例如《CS:GO》的创意工坊地图 FPS Benchmark），这些基准测试通常用于比较 GPU 的性能。

FPS 是一种比较 GPU 运行基准的粗略表达，在对一系列帧率进行分析的时候应当避免使用 FPS。FPS 的问题在于它是一个倒数度量，它并不是线性的，因此可能会导致错误分析。例如：假设我们发现应用程序在不同时间的帧率分别为 50 FPS，50 FPS 和 20 FPS。我们将这些帧率进行平均之后，可以得到 40 FPS 的平均帧率。但是这个值是非常具有误导性的，并且是错误的。我们可以将这些帧率转换为帧渲染时间，即 20 ms、20 ms 和 50 ms，因此平均的帧渲染时间为 30 ms，对应在帧率上就是 33.3 FPS。类似地，在测量各个算法性能的时候，也需要以毫秒为单位进行评价。对于一个给定基准测试与给定性能的机器，我们可以说某些阴影算法或者后处理效果“消耗”了 7 FPS，并且基准测试的运行速度要慢得多。然而，我们无法将这句话进行推广，也没法用在其他性能的机器上，这是没有意义的，因为这个值还取决于处理其他内容所需的时间，而且我们也无法将不同技术的 FPS 指标相加，这是没有意义的，但是我们可以将各种算法花费的时间相加在一起[1378]。

为了能够看到管线优化所带来的潜在影响，我们需要在禁用双缓冲的情况下，对每帧的总渲染时间进行测量，即在单缓冲模式下关闭垂直同步（vertical synchronization）。这是因为在开启双缓冲的情况下，帧缓冲区的交换只能与监视器的频率同步，我们在章节 2.1 中介绍过。De Smedt [331]讨论了对帧生成时间的分析，可以通过这种分析，发现并修复由 CPU 工作负载峰值所引起的卡顿问题，以及一些可以优化性能的有用技巧。使用这种统计分析通常是有必要的，也可以使用 GPU 上的时间戳来了解一帧内发生的事情[1167, 1422]。

设备的原生速度固然很重要，但是对于移动设备来说，另一个目标是对功耗进行优化。适当降低帧率同时保持应用程序的交互性，可以显著延长电池寿命，同时对用户的体验影响不大[1200]。Akenine-Moller 和 Johnsson 指出[25, 840]，每瓦性能（performance per watt）和每秒帧数（frames per second），与 FPS 具有相同的缺点。他们认为更加有用的测量方法是每项任务的焦耳数（joules per task），例如每像素的焦耳。

18.4 优化

一旦我们找到性能瓶颈的所在位置，我们希望能够对这个阶段进行优化，从而提高性能表现。在本小节中，我们将介绍应用阶段、几何处理阶段、光栅化阶段和像素处理阶段中的优化技术。

18.4.1 应用阶段

可以通过使得程序代码的效率更高、程序的内存访问速度更快或者次数更少来对应用阶段进行优化。这里我们将讨论一些适用于 CPU 代码优化的通用元素。

对于代码优化而言，找到代码中花费时间最多的位置是至关重要的。一个好的代码分析器可以帮助快速找到这些代码热点，这里的代码热点是指花费的时间最多的代码片段，然后再针对这些地方进行优化。程序中的代码热点通常都是内部循环（inner loop），即每一帧会执行多次的代码片段。

优化的基本准则是尝试各种优化策略：重新检查算法、假设以及代码语法，并尽可能地尝试各种变体。CPU 的架构和编译器的性能，通常会限制开发人员形成如何编写高效率代码的直觉能力，因此我们需要经常对假设进行质疑，并保持开放的心态。

第一步是对编译器的优化标志（optimization flag）进行试验。通常有许多不同的优化标志可以进行尝试，对于具体要使用什么样的优化选项，不要做过多的假设。例如：将编译器设置为使用更加激进的循环优化，可能会导致代码变得更慢。此外，如

果可能的话，还可以尝试使用不同的编译器，因为不同的编译器会以不同的方式进行优化，其中一些要明显优于其他的编译器。可以使用分析器来了解这个更改所产生的影响。

内存问题

在很多年前，算术指令的数量是衡量算法效率的关键指标，而如今则是内存访问模式（memory access pattern）。处理器的速度在过去的很多年间快速增长，而 DRAM 的数据传输速度则增长有限，因为 DRAM 会受到引脚数量的限制。从 1980 年到 2005 年，CPU 的性能大约每两年翻一番，而 DRAM 的性能大约要每六年才能翻一番[1060]。这个问题被称为冯·诺依曼瓶颈（Von Neumann bottleneck）或者内存墙（memory wall）。而面向数据设计（data-oriented design）则将缓存一致性作为一种优化手段。

> 面向数据设计（data-oriented design）不能与数据驱动设计（data-driven design）混淆，数据驱动设计可以指很多东西，从 AWK 编程语言到 A/B 测试等。

在现代的 GPU 硬件上，最重要的是数据传输的距离，传输速度和功耗与这个距离成比例。不同的缓存访问模式可以造成数量级上的性能差异[1206]。这里的缓存（cache）是指一个较小的快速存储区域，它的存在原因是因为程序中通常会有很多一致性（coherence，也可以叫做相关性，连贯性），而缓存可以有效利用这些一致性。也就是说，内存中的相邻位置往往会被依次访问（空间局部性），并且代码通常也是按照顺序进行访问的。此外，同一个内存位置往往也会被重复访问（时间局部性），缓存也会利用这一点[389]。处理器高速缓存的访问速度非常快，速度仅次于寄存器（register）。许多快速算法的工作原理就是尽可能在本地来访问数据，并且尽可能少地访问内存数据。

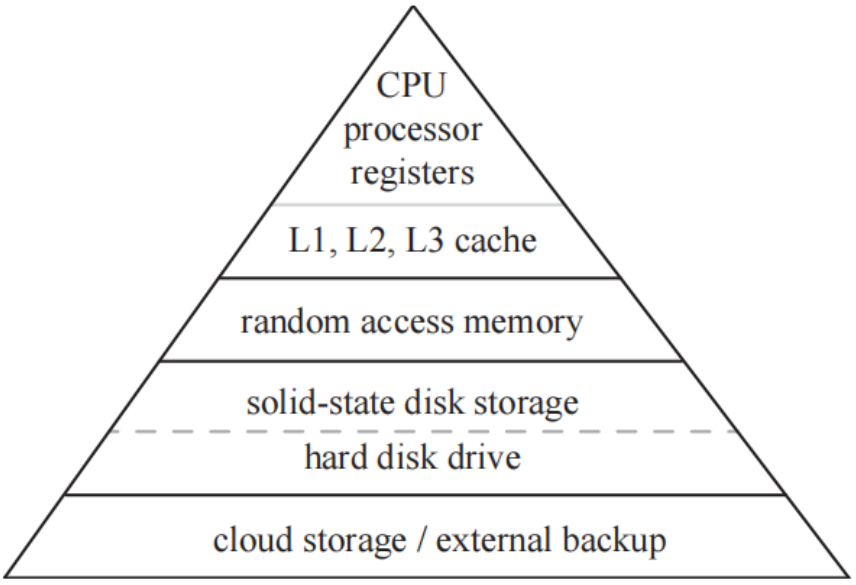


图 18.1：存储层次结构。随着金字塔层次的下降，速度和成本都在下降。

寄存器和本地高速缓存构成了存储层次结构中的顶端，并延伸到动态随机存取存储器（dynamic random access memory，DRAM），即内存；然后再到固态硬盘（SSD）和机械硬盘（HDD）中的存储。顶部是少量、高速但昂贵的内存，底部是大量、缓慢但廉价的存储。在每个层次之间都会出现明显的速度下降，如图 18.1 所示。例如：处理器中的寄存器通常可以在一个时钟周期内完成访问操作，而 L1 高速缓存则需要在几个周期内才能完成访问。存储级别的每一次变化都会以这种方式增加延迟。我们在章节 3.10 中讨论过，延迟有时可以被体系结构所隐藏，但是延迟是一个始终需要牢记的因素。

不良的内存访问模式很难在分析器中直接检测出来。需要在一开始的程序设计阶段，就内嵌一个良好的内存访问模式[1060]。下面的列表是在编程过程中应当考虑的一些因素：

- 代码中按顺序访问的数据，也应当按顺序存储在内存中。例如：在渲染一个三角形网格的时候，其数据顺序为：纹理坐标#0，法线#0，颜色#0，顶点#0，纹理坐标#1 和法线#1，如果按照这个顺序去依次访问它们，那么也要将它们依次存储在内存中。这在 GPU 上也很重要，就像变换后的顶点缓存一样（章节 16.4.4）。另外，请参阅章节 16.4.5，来了解为什么存储单独的数据流是有益的。
- 避免间接指针、跳转和函数调用（在代码的关键部分），因为这些操作可能会显著降低 CPU 的性能表现。当我们使用一个指针来指向另一个指针的时候，这样就得到了一个间接指针（pointer indirection），并且可以以此类推下去。现代

CPU 会尝试预测性地执行指令（分支预测）和获取内存（缓存预取），从而保持所有功能单元都时刻忙碌。当循环中的代码流程保持一致时，这些技术非常有效，但是在一些分支数据结构（例如二叉树、链表和图等）上就会失效；因此我们要尽可能使用数组。McVoy 和 Staelin [1194]给出了一个通过指针来追踪链表的代码示例，这样做导致前后数据的缓存未命中，并且在他们的示例中，CPU 停顿的时间要比追踪指针所需的时间长 100 倍以上（如果缓存可以提供指针地址的话）。Smits 指出[1668]，可以将基于指针的树结构扁平化为一个带有跳跃指针的列表，从而大大改进了层次结构的遍历效率。使用 van Emde Boas 布局是另一种避免缓存未命中的方法，详见章节 19.1.4。高分支的树结构通常要比二叉树更加可取，因为分支越多，树的深度就越低，从而减少了间接指针的数量。

- 将经常使用的数据结构与缓存行大小的倍数进行对齐，这样做可以显著提高整体性能。例如：64 字节的缓存行在 Intel 和 AMD 处理器[1206]上十分常见。编译器选项可以提供一些帮助，但是更加明智的做法是，在设计数据结构的时候考虑对齐问题，即填充（padding）操作。Windows 和 Linux 上的 VTune 和 CodeAnalyst、Mac 上的 Instruments，以及 Linux 上的开源 Valgrind 等工具，可以帮助识别缓存瓶颈。数据对齐也会影响 GPU 着色器的性能表现[331]。
- 尝试数据结构的不同组织方式。例如：Hecker [698]展示了如何对一个简单的矩阵乘法器，测试各种不同的矩阵结构来节省大量时间。对于一个结构体数组：

```C++
struct Vertex { float x,y,z ;};
Vertex myvertices [1000];
```

或者一个数组结构体：

```C++
struct VertexChunk {float x[1000], y[1000], z[1000];};
VertexChunk myvertices ;
```

不同的数据组织方式在不同架构上的性能表现也不同。第二种结构更加适合使用 SIMD 命令，但是随着顶点数量的增加，缓存未命中的概率也会增加。随着数组大小的增加，使用一个混合方案可能会是最佳选择：

```C++
struct Vertex4 { float x[4] ,y[4] ,z [4];};
Vertex4 myvertices [250];
```

- 最好是在启动的时候，为相同大小的对象分配一个较大的内存池，然后使用我们自行编写的分配程序和释放程序来管理这个内存池中的内存[113, 736]。Boost 等库提供了池分配功能。与那些创建的记录相比，一组连续的记录会更有可能在缓存中保持一致性。也就是说，对于具有垃圾回收功能的语言（例如 C#和 Java），内存池实际上会降低性能表现。

虽然与内存访问模式没有直接关系，但是在渲染循环中分配或者释放内存的操作是非常值得避免的。可以使用一些内存池，同时只进行一次的临时空间分配，让栈、数组和其他存储结构只增不减（使用变量或者标志来标记哪些元素应当被删除）。

18.4.2 API 调用

在本书中，我们会根据硬件的一般趋势，然后再给出相应的建议，例如：带索引的顶点缓冲区对象通常是为加速器提供几何数据的最快方法（章节 16.4.5）。本小节将介绍如何以最佳方式来调用图形 API。大多数图形 API 都有着类似的架构，并且有相当完善的方法来高效使用它们。

理解对象缓冲区（object buffer）的分配和存储是高效渲染的基础[1679]。对于具有 CPU 和独立 GPU（独立显卡）的台式机而言，CPU 和 GPU 通常都会有各自的专属内存。图形驱动程序可以控制内存对象的驻留位置，但是也可以给它一些提示，告诉它最佳的存储位置在哪里。一个常见的分类是静态缓冲区和动态缓冲区。如果一个缓冲区中的数据每一帧都在发生变化，则可以使用动态缓冲区（不需要 GPU 上的长期存储空间）。游戏主机、集成低功耗 GPU 的笔记本电脑、移动设备通常都有统一内存（unified memory），其中 GPU 和 CPU 会共享相同的物理内存。即使是在这些设备当中，在正确的资源池中分配资源也十分重要。正确地将资源标记为 CPU-only 或者 GPU-only 可以带来一些好处。一般来说，如果一个内存区域会被两个芯片访问，那么当其中一个芯片向内存写入数据的时候，另一个芯片必须要使其缓存失效（这是一个昂贵的操作），以确保不会得到过时的错误数据。

如果一个物体不会发生变形，或者这个变形过程可以完全由着色器程序来执行（例如：蒙皮），那么可以将这个物体的数据存储在 GPU 的内存中，这会带来一些性能收益。可以将其存储在静态缓冲区中，来表明该对象的不变性。通过这种方式让数据驻留在 GPU 的显存中，每次渲染不需要重新通过总线来发送数据，从而完全避免了管线在这一阶段中的瓶颈。一般情况下，GPU 的显存带宽要比 CPU 和 GPU 之间的总线带宽高得多。

状态改变

调用图形 API 有几个与之相关的开销。在应用程序端，无论这个调用实际做了些什么，更多的调用就会意味着需要花费更多的应用程序时间。这种开销可能是很小的，也可能是很明显的，可以使用一个空的驱动程序来帮助识别这种开销。依赖于 GPU 值的查询函数，可能会由于与 CPU 的同步而发生停滞，从而使得帧率降低一半[1167]。在这里，我们将深入研究并优化一个常见的图形操作，即对管线进行准备来绘制一个网格。这个操作可能涉及一些状态改变（state change），例如：设置着色器及其统一变量、附加的纹理、更改混合状态、更改所使用的颜色缓冲等。

提高应用程序性能的一个主要方法是，将具有相似渲染状态的对象进行分组处理，从而最小化状态更改所带来的开销。由于 GPU 是一种极其复杂的状态机，同时很可能是计算机科学中最为复杂的状态机，因此状态改变的成本可能会很高。其中一小部分的成本可能会涉及到 GPU，但是绝大部分的成本都来自在 CPU 上执行的驱动程序。如果 GPU 能够很好地映射到 API，那么状态改变的成本往往是可预测的，尽管仍然会很大。如果 GPU 有着严格的功率限制、或者有限的芯片面积（例如在某些移动设备上）、或者是有硬件错误需要解决，那么驱动程序将不得不执行一额外些处理，从而导致额外的高开销。状态改变的成本主要在 CPU 端的驱动程序中。

一个具体的例子是 PowerVR 架构中是如何支持混合操作的。在旧 API 中，混合操作是使用的固定功能的接口类型进行指定的。而 PowerVR 中的混合操作则是可编程的，这意味着其驱动程序必须将当前的混合状态打包到像素着色器中[699]。在这种情况下，一些更加高级的设计不能很好地映射到 API 中，因此在驱动程序中会产生很大的设置成本。在本章节中，我们会注意到不同硬件架构和运行软件对各种优化影响的重要性，尤其是对于状态改变的成本。甚至是特定的 GPU 类型和特定版本的驱动程序也可能会对性能表现产生影响。在阅读本章节的时候，请时刻牢记“your mileage may vary”。（译者注：这是一句俗语，简称 YMMV，字面意思是“您的历程可能有所不同”，代表过程中的变量会改变最终的结果）。

Everitt 和 McDonald [451]指出，不同类型的状态改变在成本上有着很大的差异，并给出了一些关于在 NVIDIA OpenGL 驱动程序上，每秒可以执行多少次状态改变的粗略估计。以下是他们在 2014 年的列表，开销从高到低依次递减：

- 渲染目标（render target）（framebuffer 对象），大约 60k/秒。
- 着色器程序，大约 300k/秒。
- 混合模式（ROP），例如透明度。
- 纹理绑定，1.5M/秒。

- 顶点格式。
- 统一缓冲区对象（uniform buffer object，UBO）绑定。
- 顶点绑定。
- 统一变量更新，大约 10M/秒。

这个大致的成本顺序也得到了其他人的证实[488, 511, 741]。一个更加昂贵的状态改变操作，是在 GPU 的渲染模式和计算着色器模式之间进行切换[1971]。想要避免发生状态改变，可以对要显示的对象按照着色器类型进行分组，然后按照使用的纹理进行分组，以此类推（按照成本顺序）进行排序分组。按照状态进行排序有时会被称为批处理（batching）。

另一种策略是重构对象的数据组织方式，从而实现更多的共享。减少纹理绑定改变的一种常见方法是，将多个纹理图像放入一个大纹理中，或者更好的是放入一个纹理数组（texture array）。如果 API 支持的话，使用无绑定纹理（bindless texture）是另一种避免状态改变的选择（章节 6.2.5）。与更新统一变量相比，对着色器程序进行修改的成本通常要高得多，因此同一类材质之间的变化可以使用使用“if”语句来进行切换，这要比使用单个着色器更好。我们也可以通过共享一个着色器来实现更大的批次[1609]。然而，让着色器变得更加复杂也会降低 GPU 的性能表现。可以通过实际实验测量，来看看那种方法更加高效，这是唯一万无一失的方法。

对图形 API 进行更少、更加高效的调用可以额外节省一些成本。例如：通常可以将多个统一变量打包成一组，绑定单个统一缓冲区对象的效率要高得多[944]。在 DirectX 中，这样的缓冲区被称为常量缓冲区（constant buffer）。正确使用这些方法，既可以节省每个函数的执行时间，也可以节省在每次 API 调用中进行错误检查的时间[331, 613]。

现代驱动程序通常会推迟状态设置，直到遇到第一个 draw call 为止。如果在此之前进行了几次冗余的 API 调用，那么驱动程序将会过滤掉这些调用，从而避免状态改变。通常会使用一个脏标记（dirty flag）来指出需要进行的状态改变，因此在每次 draw call 之后都返回一个基本状态可能会变得成本很高。例如：当我们要绘制一个对象时，我们可能会假设状态 X 默认是关闭的。实现这一目标的一种方法是“启用（X）；绘制（M_1）；禁用（X）”，然后再“启用（X）；绘制（M_2）；禁用（X）”，即在每次绘制操作之后都会恢复初始状态。然而，在两次 draw call 之间对状态进行再次设置很可能会浪费大量时间，即使它们之间并没有发生实际的状态改变。

通常来说，应用程序对于何时需要进行状态改变有着更高层次的了解，例如：将不透明表面的混合方式从“替换（replace）”改为透明表面的“叠加（over）”，通常在一帧中只需要完成一次即可，可以避免在这类物体渲染完成之前对混合模式进行改变。Galeano [511]展示了忽略掉这种过滤操作、以及发出不必要的状态调用，将会使得他们的 WebGL 应用程序花费近 2 毫秒/帧的时间。然而，如果驱动程序已经高效地进行了这种冗余过滤操作，那么在应用程序中仍然执行这种相同的测试可能会是另一种浪费。具体花费多少精力来过滤掉冗余的 API 调用，这主要取决于底层的驱动程序 [443, 488, 741]。

合并和实例化

高效调用 API 可以避免 CPU 成为瓶颈。API 调用的另一个问题是小批次问题，这是影响现代 API 性能的一个重要因素。简单地说，一个被三角形填充的网格，渲染起来要比大量小而简单的网格更加高效。这是因为无论这个图元的大小如何，每个 draw call 都有固定的成本开销（即处理图元的成本）。

早在 2003 年，Wloka [1897]就指出，每个批次仅仅绘制两个（尺寸相对较小的）三角形，其效率距离 GPU 的最大吞吐量还差 375 倍。对于一个 2.7 GHz 的 CPU 而言，此时每秒只能处理 40 万个三角形，而不是峰值的每秒 1.5 亿个三角形。对于那些由许多小而简单的物体所组成的场景，这些物体只包含很少的几个三角形，其渲染性能完全受 API 的 CPU 限制，GPU 的能力再强也没法增加渲染效率。也就是说，这些 draw call 在 CPU 上的处理时间，要大于 GPU 实际渲染网格所需的时间，即 GPU 没有被充分利用。

> Wloka 使用批次“patch”来代表通过一次 draw call 渲染的单个网格。随着时间的推移，这个术语的含义也越来越广，现在有时指的是要渲染一组具有相同状态的独立对象，因为这样也可以减少 API 的开销。

Wloka 的经验法则是“你每帧只能进行 X 个批次。”这是你每帧能够进行的 draw call 的最大数量，这纯粹是因为 CPU 的限制。在 2003 年，API 成为瓶颈的断点（breakpoint）是每个物体大约 130 个三角形。图 18.2 展示了这个断点在 2006 年是如何上升到每个网格 510 个三角形的。但是随着时代的发展，我们做了很多工作来改善这个 draw call 问题，同时 CPU 变得更快了。在 2003 年的建议是每帧进行 300 次 draw call；而在 2012 年，这个上限是每帧 16000 次 draw call [1381]。但是即使是这个相对庞大的数字，对于一些复杂的场景来说也是远远不够的。使用 DirectX 12、Vulkan 和 Metal 等现代图形 API，可以将驱动程序本身的成本最小化

——这也是它们的主要优势之一[946]。然而，GPU 在每个网格上还有一些固定的成本。

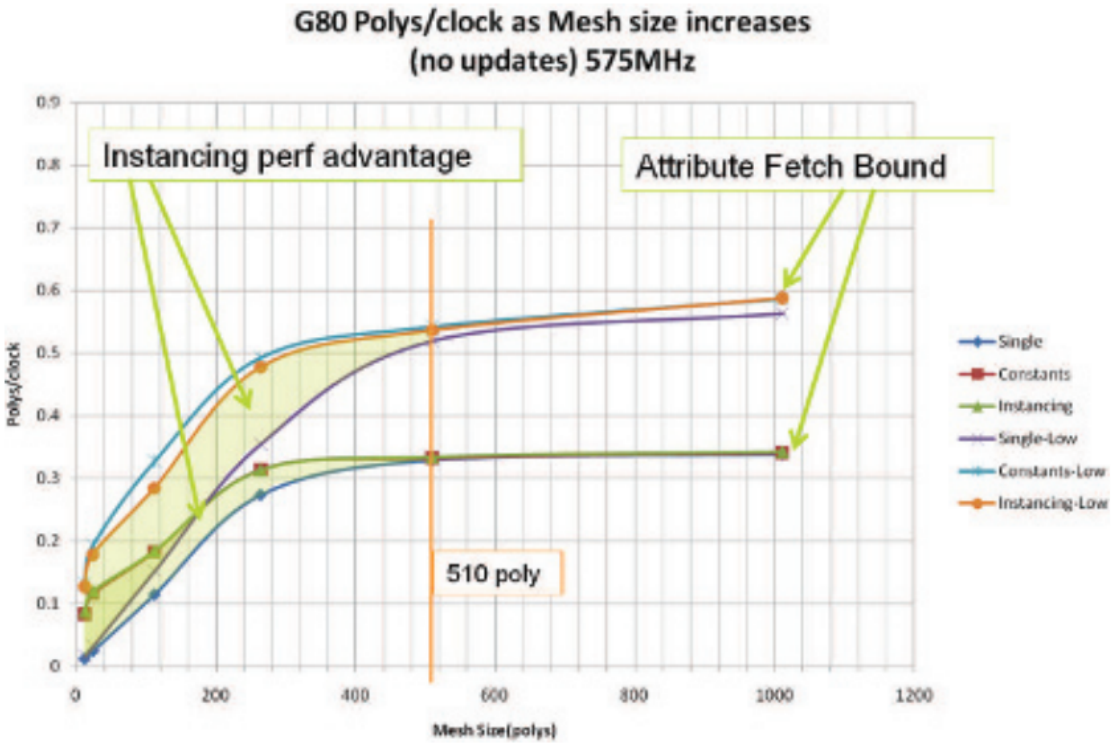


图 18.2：英特尔酷睿 2 双核 2.66 GHz CPU 的批次性能基准测试，使用了 NVIDIA G80 GPU，运行 DirectX 10。在不同的条件下，运行不同大小的批处理并进行计时。其中的“Low”条件是指，只有位置和恒定颜色的像素着色器的三角形；另一组测试则是合理的网格和着色效果。“Single”代表多次渲染同一个批次。“Instancing”代表重用网格数据，并将每个实例的数据都放在单独的数据流中。“Constants”是一个 DirectX 10 中的方法，其中实例数据会被放在常量内存中。从图中我们可以看出，小批次会降低所有方法的性能表现，但是实例化则会提高性能表现。当一个批次中包含几百个三角形时，性能会趋于稳定，因为此时性能瓶颈变成了从顶点缓冲区和缓存中检索顶点的速度。

减少 draw call 次数的一种方法是将多个物体合并到一个网格中，这样就只需要一次 draw call 来渲染该集合即可。对于使用相同状态并且是静态的几何物体而言（至少相对于其他物体来说是静态的），可以进行一次性合并，并且可以每帧中都重用这个批次[741, 1322]。之所以我们能够合并网格，是因为可以使用一个通用着色器以及纹理共享技术来避免状态改变。合并所节省的成本，不仅仅来自于避免 API 的过多 draw call，同时应用程序本身所处理的物体也变少了，这也可以节省一些开销。然而，如果批次数量远远大于所需要的数量，那么可能会使其他算法（例如视锥体剔除）的效率降低[1381]。一种做法是使用一个包围盒层次结构，来帮助查找和分组相邻的静态物体。与合并有关的另一个问题是物体选择，因为这些静态物体在同一个网

格中是没有区别的。一个典型的解决方案是，在网格的每个顶点中都存储一个对象标识符来进行标记。

另一种最小化应用程序处理和 API 成本的方法，是使用某种形式的实例化（instancing）技术[232, 741, 1382]。大多数 API 都支持在一次 draw call 中，对同一个物体进行多次绘制。

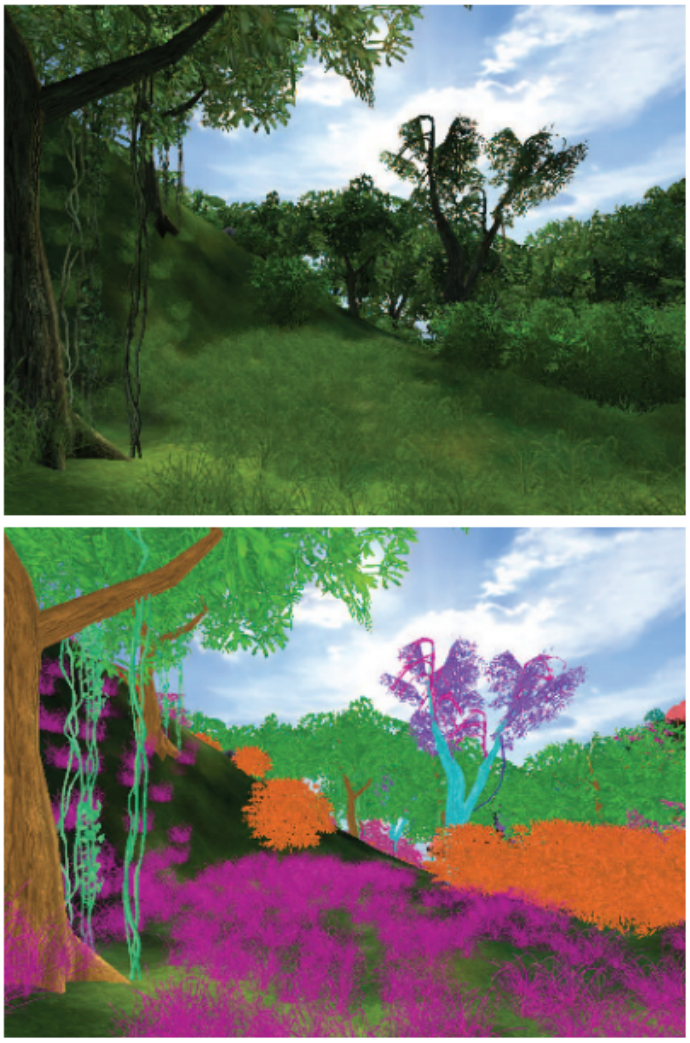


图 18.3：植被实例化。图中所有颜色相同的对象，都是在一次 draw call 中进行渲染的。[1869]

这通常是通过指定一个基础模型，并提供一个单独的数据结构来实现的，这个数据结构中包含了每个特定实例所需要的信息。除了位置和朝向之外，还可以为每个实例指定其他属性，例如树叶的颜色或者由风所引起的曲率变化，以及任何可以被着色器程序用来影响模型的东西。繁茂的丛林场景可以通过使用实例来进行创建，如图 18.3 所示。人群场景也是一个很好的例子，通过在一组选择中选择不同的身体部位，每个角色都可以显得与众不同。还可以添加随机着色和贴花效果来实现进一步的变化。实

例化也可以与 LOD 技术相结合[122, 1107, 1108]，图 18.4 中展示了这样的一个例子。

图 18.4：人群场景。使用实例化可以最小化所需的 draw call 数量。同时还使用了 LOD 技术，例如将远距离的模型渲染为 impostor。[1107, 1108]

将合并和实例化相结合的概念，被称为合并实例化（merge-instancing），其中合并网格中所包含的物体，可以依次被实例化[146, 1382]。

从理论上来说，几何着色器可以用于实现实例化操作，因为它可以根据输入网格创建出对应的重复数据。在实践中，如果需要大量实例的话，这种方法可能要比使用实例化 API 命令更慢。几何着色器的目的是执行局部的、小规模的数据放大[1827]。此外，对于某些体系结构而言，例如 Mali（一种 GPU 架构）的基于 tile 的渲染器（tile-based renderer），几何着色器是在软件上进行实现的，而不是基于硬件实现的。这里引用一句 Mali 的最佳实践指南[69]：“为你的问题找到一个更好的解决方案，几何着色器并不是你的解决方案。”

18.4.3 几何处理阶段

几何阶段负责完成顶点变换、逐顶点的光照计算、裁剪、投影和屏幕映射等操作。我们在其他章节中，讨论了一些降低通过管线数据量的方法。我们在第 16 章中所提到的，高效的三角形网格存储方式、模型简化和顶点数据压缩，都可以用于节省处理时间和内存开销。在第 19 章中所提到的，诸如截锥体剔除和遮挡剔除这样的技术，可

以避免将完整的图元本身发送到管线中。在 CPU 上使用这类大尺度的技术，可以完全改变应用程序的性能特征，因此非常值得在早期开发阶段进行尝试。而在 GPU 上，这种技术实际上不太常见。一个值得注意的例子是，可以使用计算着色器来执行各种类型的剔除[1883, 1884]。

光照的效果可以逐顶点计算，也可以逐像素计算（在像素处理阶段），或者两者同时进行计算。光照计算可以通过几种方式来进行优化。首先，应当考虑所使用的光源类型，是所有的三角形都需要进行照明吗？有时模型只需要进行纹理化，在顶点上使用纹理进行着色，或者只是在顶点上添加颜色即可。

如果光源相对于几何物体是静态的，那么漫反射光照和环境光照可以进行预计算，并存储为顶点上的颜色。这种做法通常被称为光照“烘焙”。一种更加精细的预照明形式是预先计算场景中的漫反射全局光照（章节 11.5.1）。这样的光照效果可以存储为顶点上的颜色或者强度，也可以存储为光照贴图。

对于前向渲染（forward rendering）系统，光源数量会影响几何处理阶段的性能表现。更多的光源意味着更多的计算量。减少工作量的一种常见方法是，禁用或者减少局部光照，并使用环境贴图（章节 10.5）来进行代替。

18.4.4 光栅化阶段

光栅化可以通过少数几种方式进行优化。对于那些封闭物体（实体物体）和永远不会显示其背面的物体（例如：房间墙壁的背面），应当开启背面剔除（章节 19.3）。这将会减少大约一半的三角形光栅化数量，从而降低了三角形遍历阶段的负载。此外，当像素着色的计算成本很高时，这样做可能会带来很大的益处，因为背景永远不会被着色。

18.4.5 像素处理阶段

对像素处理阶段进行优化通常是很有帮助的，因为需要进行着色的像素数量通常要比顶点多得多。但是也有明显的例外情况，因为顶点总是需要进行处理的，但是这次绘制最终可能并不会生成任何可见的像素。渲染引擎中的无效剔除可能导致顶点着色的成本超过像素着色的成本。尺寸太小的三角形不仅会导致更多的顶点着色计算，而且还会生成更多部分覆盖（partial-covered）的四边形，从而导致一些额外的工作。更重要的是，那些只覆盖几个像素的纹理网格通常会具有较低的线程占用率。正如章节 3.10 中所讨论的那样，对纹理进行采样需要花费大量的时间，GPU 会通过切换到其他片元上执行着色器程序，从而隐藏这个纹理采样的延迟，当纹理数据被获取之后再返回到之前的片元上继续计算。较低的线程占用率会导致较差的延迟隐藏效果。对

于一些使用大量寄存器的复杂着色器而言，同一时间内可用的线程数量会更少，这也会导致较低的线程占用率（章节 23.3），这种情况被称为高寄存器压力（register pressure），也叫做寄存器不足。还有一些其他的微妙之处，例如：频繁切换到其他 warp 上可能会导致更多的缓存未命中。Wronski [1911, 1914]对各种占用问题和解决方案进行了讨论。

首先，使用原生的纹理格式和像素格式，也就是说，最好使用图形加速器内部所使用的格式，从而避免从一种格式到另一种格式的转换，这个过程可能是十分昂贵的[278]。另外两种与纹理相关的技术是，第一：只加载需要的 mipmap 层级（章节 19.10.1）；第二：使用纹理压缩技术（章节 6.2.6）。通常而言，尺寸更小、数量更少的纹理，意味着更少的内存使用，这反过来又意味着更少的传输时间和访问时间。纹理压缩还可以提高缓存性能，因为现在相同数量的缓存会被更多的像素所占用。

还有一个 LOD 技术是，根据物体到观察者的距离来使用不同的像素着色器程序。例如：在一个场景中有三个飞碟模型，距离最近的那个飞碟可能会有一个复杂的凹凸贴图来描绘表面细节，而距离较远的那两个飞碟则不需要。此外，远处的飞碟还可以对高光效果进行简化甚至是完全移除高光，这一方面简化了计算，另一方面还可以减少“萤火虫”现象，即欠采样导致的闪光瑕疵。在简化模型上使用逐顶点的颜色可以带来一些额外的好处，因为不需要因为纹理变化而发生状态改变。

只有在三角形光栅化之后可见的片元，才会调用相应的像素着色器。GPU 的 early-z 测试（章节 23.7）会根据已有的 z-buffer，对片元的 z 深度进行检查。如果发现这个片元不可见，那么会丢弃这个片元，即不会调用任何像素着色器，从而节省大量时间。虽然可以通过像素着色器来对 z 深度进行修改，但是这样做意味着我们就无法执行 early-z 测试了。

为了理解程序的详细行为，尤其是像素处理阶段的工作负载，将深度复杂度（depth complexity）进行可视化是十分有用的，这里的深度复杂度代表了覆盖一个像素的表面数量，也可以理解为一个像素上进行深度测试的次数，图 18.5 展示了这样的一个例子。生成深度复杂度图像的一个简单方法是，使用一个类似于 OpenGL 中的 $\mathrm{glBlendFunc}(\mathrm{GL\_One}, \mathrm{GL\_One})$ 的函数调用，同时禁用 z-buffer。首先图像会被清除为黑色，然后将场景中的所有物体都渲染为颜色 $(1/255, 1/255, 1/255)$。这个混合函数设置之后的效果是，对于被渲染的每个图元，写入像素的值将会增加一个强度级别。一个深度复杂度为 0 的像素是黑色的，而一个深度复杂度为 255 的像素是纯白色的，即 $(255, 255, 255)$。

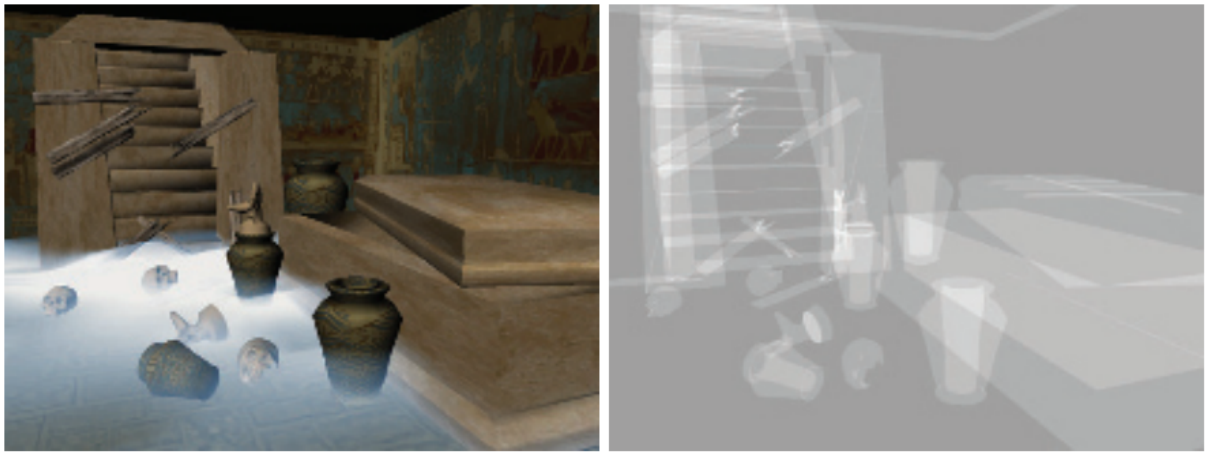


图 18.5：左边是场景的渲染图，右侧展示了该场景的深度复杂度。

像素过度绘制（pixel overdraw）的数量，与实际渲染的表面数量有关。像素着色器被调用的次数，可以通过再次渲染这个场景来找到，但是要启用 z-buffer。过度绘制是指浪费在表面着色上的计算工作量，因为这个计算好的像素着色结果之后会其他像素所遮挡，因此这个片元对于最终的图像贡献为 0。这也是延迟渲染（章节 20.1）和光线追踪的其中一个优点，即在执行所有可见性计算之后，然后再执行着色计算，这样就不会导致过度绘制所带来的计算浪费。

假设现在有两个三角形覆盖了同一个像素，那么它的深度复杂度就是 2。如果较远的那个三角形先被绘制，那么较近的这个三角形就会发生过度绘制，过度绘制量为 1。但是如果较近的这个三角形先被绘制，那么较远的那个三角形就不会通过深度测试，因此并不会被绘制，也就不会出现过度绘制现象了。对于覆盖一个像素的一组随机不透明三角形，平均绘制次数是一个调和级数（harmonic series）[296]：

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \tag{18.1}$$

这个公式背后的逻辑是：第一个被渲染的三角形肯定会发生一次绘制。而第二个三角形要么在第一个三角形的前面，要么在第一个三角形的后面，这个概率是 $50/50$ 。而对于第三个三角形而言，它可以出现在三个位置中的任何一个，其中有三分之一的概率会位于最前面。当三角形数量 n 趋于无穷时，有：

$$\lim_{n \to \infty} H(n) = \ln(n) + \gamma \tag{18.2}$$

其中 $\gamma = 0.57721\ldots$，它是 Euler–Mascheroni 常数。当深度复杂度较低时，过度绘制的数量会迅速上升，但是这个上升速度也会快速下降。例如：深度复杂度为 4 意

味着平均有 2.08 次过度绘制，深度复杂度为 11 意味着平均有 3.02 次过度绘制，但深度复杂度为 12367 却只意味着平均有 10.00 次过度绘制。

因此，过度绘制并不一定像看起来得那么糟糕，但是我们仍然希望在不花费太多 CPU 时间的情况下，将过度绘制所带来的额外成本最小化。我们可以对场景中不透明物体进行粗略排序，然后按照大致的前后顺序（从近到远）来绘制物体，这是减少过度绘制的一种常用方法[240, 443, 488, 511]。后来进行绘制的被遮挡物体，将不会写入颜色缓冲或者 z-buffer，即减少了过度绘制现象。此外，甚至在到达像素着色程序之前，这些像素片元还可以被遮挡剔除硬件所丢弃。有许多方法可以实现这种粗略排序，其中一种简单的方法是，根据所有不透明物体的质心沿着观察方向的距离来进行显式排序。如果使用了层次包围结构或者其他的一些空间结构来用于视锥体剔除，那么我们可以选择最近的子节点先进行遍历，然后再沿着层次结构向下进行遍历。

另一种优化技术可以用于具有复杂像素着色程序的表面。首先执行一个 z-prepass，来将场景中的几何图形渲染到 z-buffer 中，然后再正常渲染整个场景[643]。这样可以消除所有的过度绘制以及浪费的着色器计算工作，但代价是需要先对整个场景先单独渲染一遍。Pettineo [1405]写道，他的团队在电子游戏中使用了一个 z-prepass，其主要原因就是为了避免过度绘制。然而，先粗略地对物体进行排序，然后再按照从前到后的顺序进行绘制，可能会提供同样的好处，同时不需要进行额外 prepass 的工作。还有一种混合的方法是，首先识别并提取出几个大型、简单的遮挡物，在提取的时候，要求这些遮挡物能够带来较大的收益[1768]。正如 McGuire [1177]指出的那样，一个完整的 prepass 对于他的特定系统并不会带来性能提升。当然还是要根据实际情况进行测试，实际测试才是知道到底哪种技术是最有效的唯一方法。

前面我们建议可以按照着色器和纹理进行分组，从而最小化状态改变，而在这里，我们讨论的是按照距离远近的排序来渲染物体。这两个目标通常会给出不同的物体绘制顺序，并且会彼此相互冲突。实际上对于给定的场景和观察点，总是存在一些十分理想的绘制顺序，但是这个顺序是很难提前找到的。可以使用一些混合方案，例如：根据深度对附近的物体进行排序，并根据材质来对其他物体进行排序[1433]。一种通用、灵活的解决方案[438, 488, 511, 1434, 1882]，是为每个对象都创建一个排序 key，通过为每个对象分配一组 bit 来封装所有相关的条件，如图 18.6 所示。

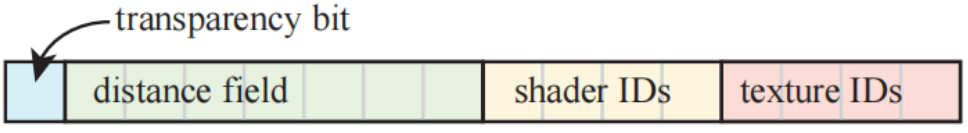


图 18.6：绘制顺序的排序 key 示例。这些 key 从低到高进行排序。其中如果设置了透明度 bit，意味着这个物体是透明的，因为透明物体会在所有不透明物体之后再进行渲染。物体到相机的距离会被存储为一个精度较低的整数。对于透明物体而言，这个距离是反向的或者负的，因为我们希望这些透明物体是从后向前进行绘制的。每个着色器都会有一个唯一的识别号码，纹理也是如此。

我们可以选择按照距离进行排序，通过限制存储深度的 bit 数，我们还可以允许在一个给定的范围内，将物体按照着色器来进行分组。将绘制仅仅分成两三个深度分区的情况并不少见。如果某些物体具有相同的深度并且使用了相同的着色器，则可以使用纹理标识符来对物体进行排序，然后将这些具有相同纹理的物体分成一组。

这是一个简单的例子，并且是根据情况而定的，例如：渲染引擎本身可能会将不透明物体和透明物体分开处理，这样透明物体就不需要进行排序分组了。其他字段的 bit 数当然也会随着着色器和纹理的最大数量而发生变化。还可以添加或者替换其他字段，例如其中一个字段用于混合状态，另一个字段用于 z-buffer 的读写。当然最重要的还是具体架构，例如：移动设备上的一些基于 tile 的 GPU 渲染器，并不会从前后排序中获得任何性能收益，因此对于这样的渲染器而言，状态排序是唯一需要进行优化的重要元素[1609]。这里的核心思想是，将所有属性都放入一个整数 key 中可以执行高效的排序，从而尽可能地减少过度绘制和状态改变所带来的额外开销。

18.4.6 帧缓冲技术

渲染一个场景通常会导致对帧缓冲（framebuffer）的大量访问，以及大量像素着色器的执行。为了减少缓存结构的压力，一个常见的建议是减少帧缓冲中每个像素的存储大小。虽然每个颜色通道使用 16 bit 的浮点值会带来更高的精度，但是 8 bit 浮点值的大小是 16 bit 的一半，如果在满足精度的前提下，使用 8 bit 浮点值会带来更快的访问速度。在许多图像和视频压缩方案中（例如 JPEG 和 MPEG），经常会对色度进行二次采样（subsample），即下采样。这种做法对于视觉效果的影响通常可以忽略不计，因为人类的视觉系统对于亮度更加敏感，而对于色度则没有那么敏感。例如：寒霜引擎[1877]中就使用了这种色度下采样（chroma subsampling）的想法，来降低后处理中每通道 16 bit 图像的带宽成本。

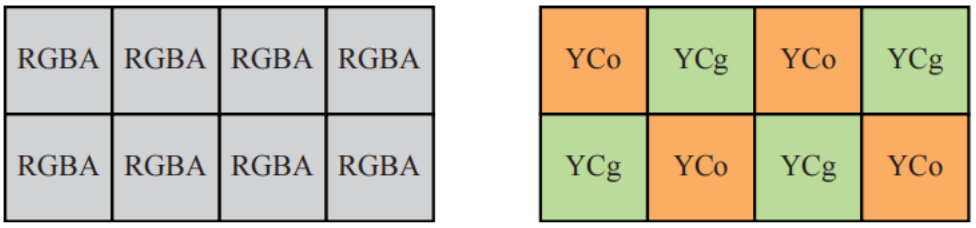


图 18.7：左： 4×2 个像素，每个像素存储 4 个颜色分量（RGBA）。右：另一种表示形式，其中每个像素以棋盘格模式来存储亮度 Y，以及第一（ C_o ）或者第二（ C_g ）色度分量。

Mavridis 和 Papaioannou [1144]指出，我们在章节 6.2 中所描述的有损 YCoCg 变换，应用在光栅化期间的颜色缓冲区上可以实现类似的效果，他们所使用的像素布局如图 18.7 所示。与 RGBA 格式相比，这可以降低一半的颜色存储需求（假设我们不需要 A），并且通常还可以提高性能表现，当然这要取决于具体的架构。由于每个像素都只有一个色度分量，因此需要一个重建滤波器，来推断每个像素的完整 YCoCg 表示，然后在显示之前将其转换回 RGB 格式。例如：对于一个缺失 C_o 值的像素，可以使用 4 个最接近的 C_o 值的平均值来进行代替。然而，这种方式并不能很好地重建出边缘信息，因此，可以使用一个简单的边缘感知（edge-aware）滤波器，其数学实现为：

$$C_o = \sum_{i=0}^{3} w_i C_{o,i}, \quad \text{where} \quad w_i = 1.0 - \text{step}\left(t - |L_i - L|\right) \tag{18.3}$$

对于缺少 C_o 值的像素，可以使用方程 18.3 来进行重建。其中 $C_{o,i}$ 和 L_i 分别是当前像素的左、右、上、下像素中的值， L 是当前像素的亮度， t 是边缘检测的阈值。Mavridis 和 Papaioannou 使用 $t = 30/255$ 。如果 $x < 0$ ，那么 $\text{step}(x)$ 函数值为 0，否则为 1。因此，这个滤波器权重 w_i 要么为 0，要么为 1，如果亮度梯度 $|L_i - L|$ 大于边缘阈值 t ，则权重 w_i 为零。他们还提供了一个在线 WebGL 的 Demo 和相应的源代码[1144]。

由于显示分辨率的不断提高，以及着色器执行成本的持续增加，在好几个系统中都使用了棋盘模式来进行渲染[231, 415, 836, 1885]。对于虚拟现实应用，Vlachos [1824]在视图周边的像素上使用了棋盘模式，Answer [59]则将每个 2×2 的四边形减少 1–3 个样本。

18.4.7 合并阶段

确保只在有用的时候才启用混合模式。理论上，可以将每个三角形都设置为“over”合成模式，无论是不透明物体还是透明物体，因为使用“over”的不透明表面将完全覆盖像素中原有的值。然而，这要比简单的“replace”光栅化操作更加昂贵，因此追踪记录那些具有镂空纹理和透明度材质的物体是值得的。此外，还有一些光栅化操作不需要任何额外的开销。例如：当使用 z-buffer 的时候，在某些系统上访问模板缓冲区并不会花费额外的时间。这是因为 8-bit 的模板缓冲值，与 24-bit 的 z 深度值存储在同一个字段中[890]。

仔细考虑各种缓冲区何时需要进行使用或者进行清除是非常有必要的。由于 GPU 具有快速清除机制（章节 23.5），因此我们建议始终清除颜色缓冲和深度缓冲，因为这样可以增加缓冲区的内存传输效率。

如果可以做到的话，我们通常应当避免从 GPU 端回读渲染目标到 CPU 端。CPU 对帧缓冲的任何访问操作，都会导致整个 GPU 管线在渲染返回之前被刷新，从而失去所有的并行性[1167, 1609]。

如果我们发现合并阶段是整个应用的性能瓶颈，那么我们可能就需要重新审视一下我们的方法了。我们能不能使用较低精度的输出目标，例如通过一些压缩算法？有没有办法对我们的算法进行重新排序，来减轻这个阶段的压力？对于阴影而言，有没有办法将那些不会发生移动的部分进行缓存和重用？

在本小节中，我们讨论了通过查找瓶颈位置和性能调优，来更好地使用每个阶段的方法。同时，如果我们直接换一种完全不同的技术，可能会得到更好的效果，那么对现有算法进行重复优化是很危险的。

18.5 多处理

传统 API 的发展方向是使用更少数量的调用，每次调用完成更多的事情[443, 451]。而新一代的图形 API——DirectX 12、Vulkan 和 Metal——则采取了不同的策略。对于这些 API 而言，它们有着精简且最小化开销的驱动程序，并将验证状态的大部分复杂性和责任、以及内存分配和其他一些功能，都转移到了调用它们的应用程序中[249, 1438, 1826]。这种重新设计在很大程度上是为了尽量降低 draw call 和状态改变的开销，这些开销很大一部分来自于必须将旧 API 映射到现代 GPU 上。这些新图形 API 鼓励的另一个操作是，使用多个 CPU 处理器来调用图形 API。

在 2003 年前后，由于一些物理问题（例如散热和功耗），CPU 频率不断上升的趋势在 3.4 GHz 左右趋于平缓[1725]。这些限制因素导致了多处理 CPU 的出现，在这种情况下，单个芯片中放入了更多的 CPU 核心，而不是进一步提高 CPU 的单核频率。

事实上，许多小内核可以提供单位面积上的最佳性能（译者注：Intel 大小核中的小核心效率很高）[75]，这也是 GPU 本身如此高效的主要原因。从那时起，程序如何高效可靠的利用并发能力（concurrency）一直是一个挑战。在本小节中，我们将介绍 CPU 核心上高效多处理的基本概念，最后讨论图形 API 是如何发生演变的，从而可以在驱动程序本身中实现更多并发能力。

多处理器计算机（multiprocessor computer）可以大致分为消息传递体系结构（message-passing architecture）和共享内存的多处理器（shared memory multiprocessor）。在消息传递设计中，每个处理器都有自己的内存区域，并且可以在处理器之间发送消息以传递结果，这种设计在实时渲染中并不常见。共享内存的多处理器顾名思义，所有的处理器彼此共享同一个内存的逻辑地址空间。大多数流行的多处理器系统都使用了共享内存的设计，其中大多数都具有对称多处理（symmetric multiprocessing，SMP）的设计。SMP 意味着所有处理器核心都是相同的，一个多核 PC 系统就是对称多处理架构的一个例子。（译者注：Intel 大小核设计就是非对称的）

在这里，我们将介绍两种使用多处理器进行实时图形处理的通用方法。第一种方法是多处理器流水线（multiprocessor pipelining），也被称为时间并行（temporal parallelism）；第二种方法叫做并行处理（parallel processing,），也被称为空间并行（spatial parallelism），其中我们将详细介绍第一种方法。图 18.8 具体展示了这两种方法。然后我们将这两种类型的并行方法与基于任务的多处理结合在一起，在基于任务的多处理中，应用程序可以创建作业，并由单个核心拾取和处理。

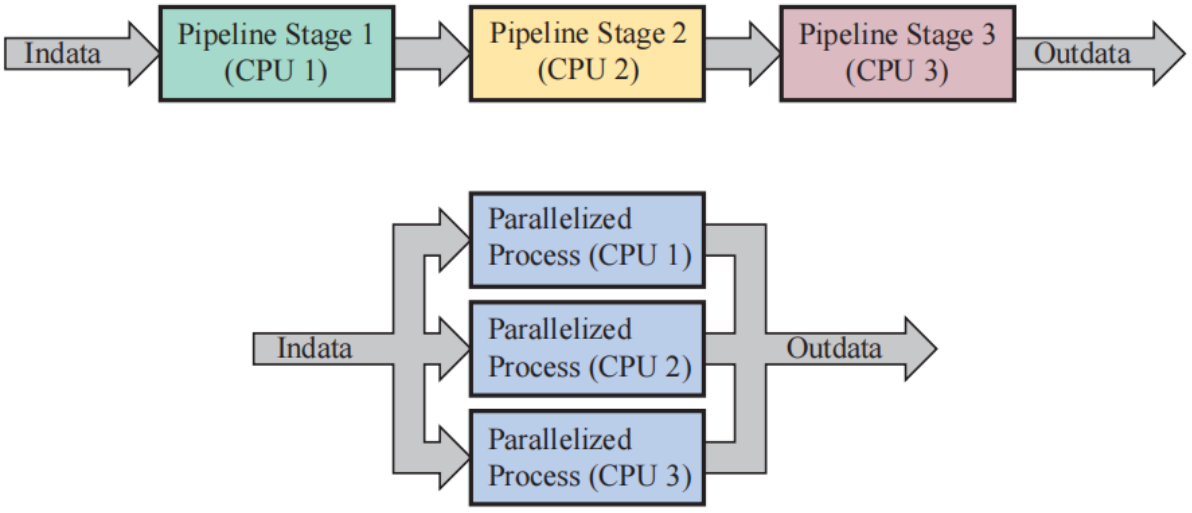


图 18.8：使用多处理器的两种不同方式。在顶部，我们展示了如何在多处理器管线中使用三个处理器（CPU）；在底部，我们展示了如何在三个 CPU 上并行执行任务。这两种实现方式的区别之一在于，使用第二行中的配置方法可以实现更低的延迟。另一方面，使用第一行中的多

处理器管线可能会更加容易。这两种配置方法的理想加速效果都是线性的，也就是说，使用 n 个 CPU 将获得 n 倍的理想加速效果。

18.5.1 多处理器流水线

正如我们所看到的，流水线通过将作业划分为不同的流水线阶段，这些流水线阶段会并行执行，从而来加快执行速度。流水线中一个阶段的结果会被传递到下一个阶段中。对于 n 个流水线阶段，理想的加速效果是 n 倍，其中流水线中最慢的那个阶段（瓶颈）决定了实际的加速效果。到目前为止，我们已经看到单个 CPU 核心和 GPU，使用流水线来并行执行应用程序阶段、几何处理阶段、光栅化阶段和像素处理阶段。当主机上存在多个处理器时，也可以使用流水线，在这种情况下，这被称为多进程流水线（multiprocess pipelining）或者软件流水线（software pipelining）。

这里我们将介绍一种软件流水线技术。这种技术可能会有无数的变体，它只适用于特定的应用中。在这个例子中，应用阶段被划分为了三个子阶段[1508]：APP、CULL 和 DRAW。这是一个粗粒度的流水线，即意味着每个阶段都相对较长。APP 阶段是流水线中的第一个阶段，它控制着其他的后续阶段。在这个阶段中，开发人员可以添加额外的代码，例如进行碰撞检测等。同时 APP 阶段还会对视点进行更新。CULL 阶段可以执行以下操作：

- 在一个场景图（scene graph）上进行遍历，并进行分层视锥体剔除（章节 19.4）。
- LOD 选择（章节 19.9）。
- 状态排序（章节 18.4.5）。
- 最后（总是会执行），生成一个简单的列表，其中包含了所有需要进行渲染的物体。

DRAW 阶段从 CULL 阶段获取物体列表，并在这个列表中执行所有的图形调用。这意味着这个阶段只是对列表进行遍历，并向 GPU 发送数据。图 18.9 展示了如何使用这个流水线的一些示例。

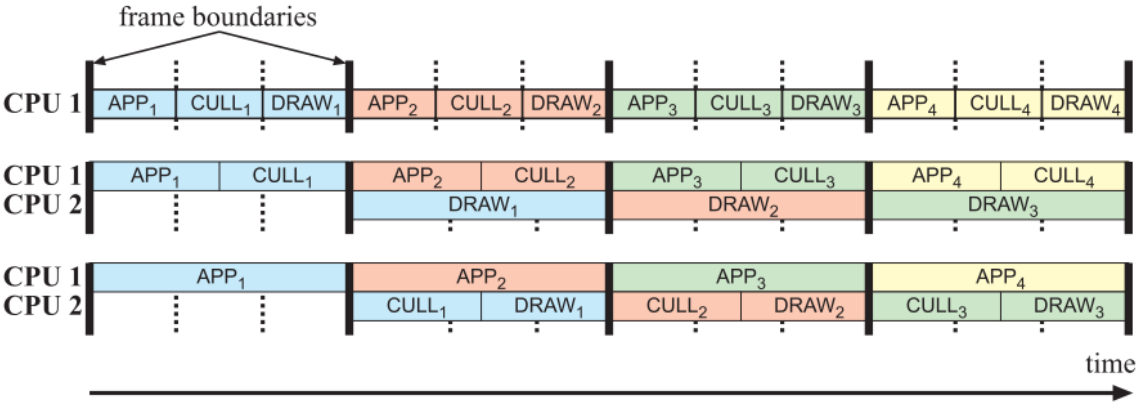


图 18.9：多处理器管线的不同配置方法。图中的粗线代表阶段之间的同步过程，数字下标代表帧号。第一行是单 CPU 管线。第二行和第三行则是使用两个 CPU 核心的情况，有着两种不同的管线划分方式。在第二行中，有一个核心用于 APP 和 CULL，另一个核心用于 DRAW。如果 DRAW 阶段的工作负载要大得多的话，那么这是一个合适的划分方法。在第三行中，有一个核心用于 APP，另一个核心用于 CULL 和 DRAW。如果此时 APP 的工作负载要比其他阶段大得多的话，那么这也是一个合适的划分方法。需要注意的是，在下面两种配置方法中，APP、CULL 和 DRAW 阶段都获得了更多的 CPU 核心时间。

如果只有一个处理器核心可以使用，那么所有的三个阶段都将在这个核心上运行。如果有两个可用的 CPU 核心，那么可以在一个核心上执行 APP 和 CULL，在另一个核心上执行 DRAW。另一种可选的配置方法是，在一个核心上执行 APP，在另一个核心上进行 CULL 和 DRAW。具体哪一种配置方法是最好的，取决于不同阶段的工作负载。最后，如果主机上有三个可用的 CPU 核心，那么每个阶段都可以在单独的核心上执行。图 18.10 展示了这种可能性。

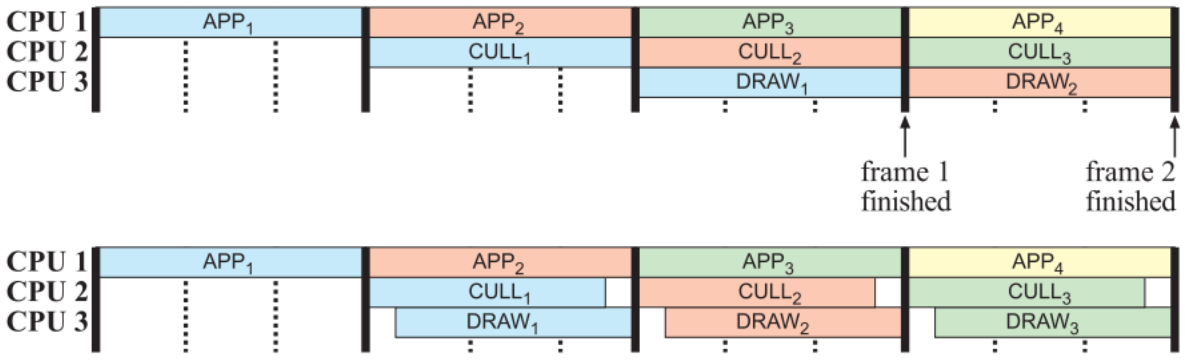


图 18.10：第一行展示了一个三阶段管线。与图 18.9 中的配置方法相比，这种配置方法在每个管线阶段都会有更多的执行时间。第二行展示了一种减少延迟的方法：CULL 阶段和 DRAW 阶段会相互重叠，中间使用了 FIFO 缓冲。

这种技术的优点在于较大的吞吐量，即提高了渲染速度。其缺点在于延迟更大了（与并行处理相比）。延迟（latency）或者时间延迟（temporal delay），是指从轮询用户操作，到显示最终图像所花费的时间[1849]。延迟不应当与帧率相混淆，帧率指的是每秒显示的帧数。例如：假设用户正在使用一个无线连接（untethered）的头戴式显示器。其中头部位置的确定可能需要 10 ms 才能到达 CPU，然后还需要 15 ms 才能渲染一帧。此时从初始输入到最终显示的延迟为 25 ms。即使此时的画面帧率是 66.7 Hz（1/0.015 秒），如果不进行位置预测或者其他补偿的话，用户也会因为需要将位置变化发送到 CPU 的延迟而感到缓慢迟钝（sluggish）。忽略由用户交互所引起的任何延迟（这在两个系统中都是恒定的），多处理仍然会比并行处理具有更大的延迟，因为它使用了一个流水线设计。有关并行处理的内容，我们将在下一小节中进行详细讨论，它将渲染一帧的工作分解为多个并发运行的部分。

与在主机上使用单个 CPU 相比，多处理器流水线提供了更高的帧率，并且由于同步所带来的成本，延迟会大致相同或者稍大一些。这个延迟会随着流水线中的阶段数量的增加而增加。而对于一个良好平衡的应用程序而言，对于 n 个 CPU 核心，会加速 n 倍。

一种减少延迟的技术是，在 APP 结束时再来更新视点和其他会导致延迟的关键参数[1508]，这将会减少了大约一帧左右的延迟。另一种减少延迟的方法，是将 CULL 和 DRAW 重叠执行。这意味着，一旦有任何需要渲染的物体准备就绪，CULL 的结果就会被立即发送到 DRAW 阶段中进行后续处理。为了实现这一点，在这些阶段之间必须要有一些缓冲，通常是一个 FIFO 缓冲。当这个 FIFO 缓冲区为空或者为满的情况下，这两个阶段都会停止运行；也就是说，当缓冲区已满时，CULL 阶段必须停止；而当缓冲区为空时，DRAW 阶段必须进行等待。这样做的缺点在于，诸如状态排序之类的技术无法以相同的程度进行使用，因为这些图元在经过 CULL 阶段处理之后，必须立即进行渲染。这种减少延迟的技术如图 18.10 所示。

图中的流水线最多使用了三个 CPU，每个阶段都有特定的任务。然而，这种技术并不仅仅局限于这种配置；相反，我们可以使用任意数量的 CPU 核心，并以任何想要的方式来分配工作。这个配置方式的关键在于，要对整个工作进行明智的划分，这样流水线才能趋于平衡。多处理器流水线技术只在帧切换时需要进行同步，因此对同步的要求最低。还可以使用一些额外的处理器进行并行处理，但是并行处理需要更加频繁的同步操作。

18.5.2 并行处理

使用多处理器流水线技术的一个主要缺点在于会增加一些延迟加。对于一些应用程序，例如：飞行模拟器、第一人称射击游戏和虚拟现实等，这是不可接受的。当我们在移动视角的时候，我们通常希望能够立即（下一帧）响应；但是当延迟很长时，是很难做到立即响应的。但是这也取决于具体情况，如果多处理能够将帧率从 1 帧延迟的 30 FPS，提高到 2 帧延迟的 60 FPS，那么额外的帧延迟将不会带来明显的感知差异。

如果有多个处理器核心可以调用，还可以尝试并发地运行代码，这可能会缩短延迟。想要做到这一点，程序的任务首先必须具有并行性（parallelism）。有几种不同的方法可以将一个算法进行并行化处理。假设现在我们有 n 个处理器核心可以调用，使用静态分配方法[313]，将整个工作包（例如加速结构的遍历）划分为 n 个子工作包（work package）。然后每个处理器都负责处理其中一个工作包，这样所有的处理器就可以并行执行各自的工作包。当所有的处理器都完成各自的工作包时，还需要将各个处理器的运算结果进行合并。想要做到这一点，那么工作负载必须是高度可预测的。

如果不是这种情况的话，那么可以使用一些适应不同工作负载的动态分配算法[313]。它们会使用一个或者多个工作池（work pool），当生成作业（job）时，它们就会被放入工作池中。当 CPU 在完成当前作业之后，可以从队列中再次获取一个或者多个作业。需要注意的是，同一时刻只有一个 CPU 能够获取特定的作业，并且还要确保维护作业队列所带来的开销并不会对性能产生损害。较大工作量的作业意味着维护队列的开销会变得不那么大，但是从另一方面来说，如果作业的工作量过大，那么可能会由于系统中的不平衡性而导致性能降低，即会有一个或者多个 CPU 处于饥饿状态，没有被分配作业。

对于多处理器流水线而言，运行在 n 个处理器核心上的并行程序，其理想的加速效果是 n 倍，这被称为线性加速（linear speedup）。尽管这种线性加速效果很少能够达到，但是实际结果有时会很接近。

在图 18.8 中，展示了一个多处理器流水线和一个具有三个 CPU 的并行处理系统。我们暂时假设这两种配置方法每帧所要完成的工作量是相同的，并且两种配置方法都可以实现线性加速。这意味着与串行执行（serial execution，即在单个 CPU 上执行）相比，执行速度将会快三倍。此外，我们假设每帧的总工作量为 30 ms，这意味着在单个 CPU 上的最大帧率将为 $1/0.03 \approx 33$ 帧。

多处理器流水线会将（在理想情况下）将工作负载划分为三个大小相等的工作包，并让每个 CPU 都负责完成其中的一个工作包，其中每个工作包大约需要 10 ms 才能完成。如果我们沿着流水线来追踪工作流，我们可以看到流水线中的第一个 CPU 确实

工作了 10 ms（即总工作量的三分之一），然后将结果发送给下一个 CPU。此时第一个 CPU 会重新开始处理下一帧中的第一部分工作。当最终计算完一帧时，它实际上总共花费了 30 ms 来完成，但是由于这个工作是在流水线中并行完成的，因此每 10 ms 就可以生成一帧。因此，最终的延迟是 30 ms，加速效果是 3 倍（$30/10$），即每秒 100 帧。

同一程序的并行化版本，同样也会将作业划分为三个工作包，但是这三个工作包将会在三个 CPU 上同时执行。这意味着最终的延迟将是 10 ms，其中每一帧的工作也会花费 10 ms。这里的结论是，使用并行处理时的延迟，要比使用多处理器流水线短得多。

18.5.3 基于任务的多处理

在了解了流水线和并行处理技术之后，可以很自然地就可以将二者结合在一个系统中。如果只有少数几个处理器可用，那么可以使用一个简单的系统，显式地将某个系统分配给某个特定的核心，这样做是可行的（译者注：可以参考 GT 小新的 CPU 优化视频）。然而，考虑到许多 CPU 上都有很多核心，现在的技术趋势是使用基于任务的多处理方法。就像是可以为一个并行化进程创建多个任务（也称为作业）一样，这种思想也可以扩展到流水线上。由任何核心生成的任何任务，首先都会被放入工作池中，任何处于空闲状态的处理器都会获取一个任务来进行处理。

转换为多处理的一种方法是，获取应用程序的工作流程，并确定其中哪些系统需要依赖于其他系统，如图 18.11 所示。

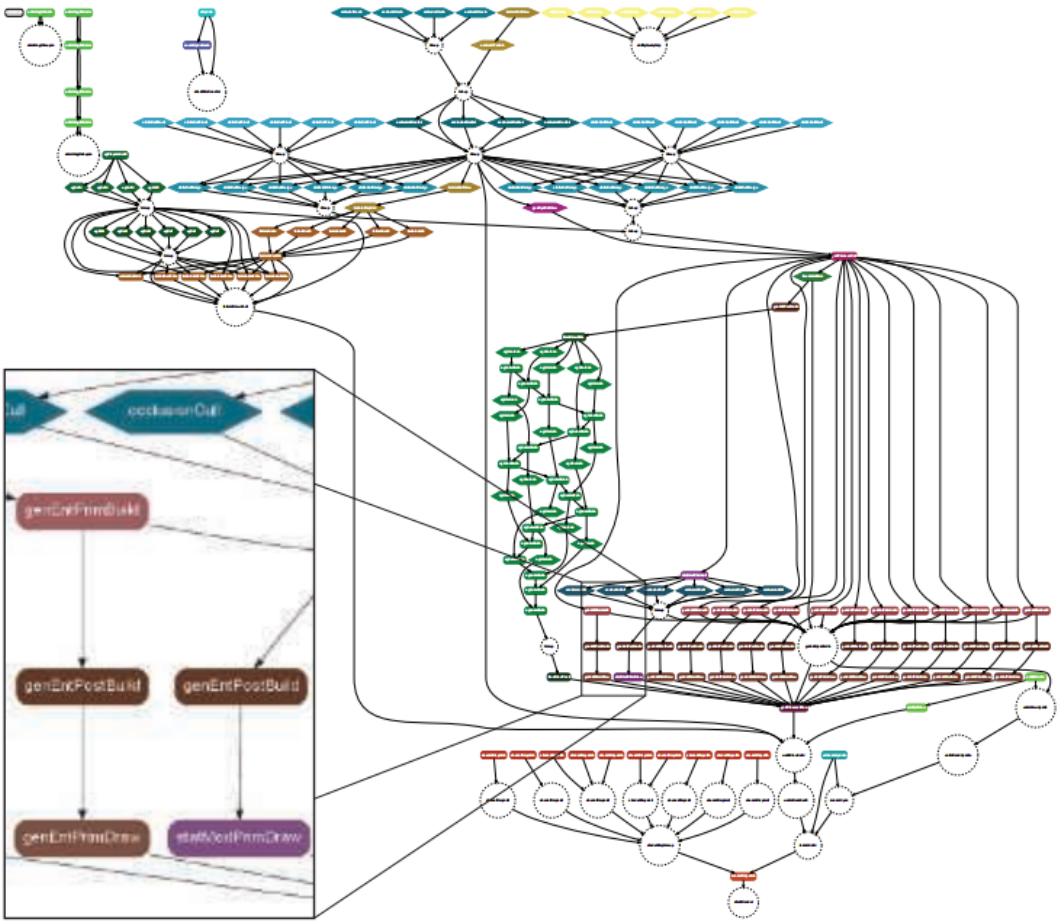


图 18.11：寒霜引擎的 CPU 作业图。（书中的图是矢量图，可以放大查看）[45]

如果处理器在等待同步的时候出现停滞现象，那么意味着这种基于任务的应用程序，甚至可能会因为这种停滞和任务管理所带来的额外开销，从而变得效率更低[1854]。然而，许多程序和算法确实可以同时执行大量任务，因此可以从中获得效率提升。

下一步是确定每个系统中的哪些部分可以分解为任务。适合成为任务的一段代码的包括以下特征[45, 1060, 1854]：

- 任务有着明确的输入和输出。
- 任务在运行时是独立的、无状态的，并且总能够完成。
- 它并不是一个工作量过大的任务，不会使其成为唯一运行的进程。

像 C++ 11 这样的语言中内置了多线程机制（multithreading）[1445]。在与 Intel 兼容的系统上，Intel 开源的线程构建模块（Threading Building Blocks，TBB）是一个高效的库，可以简化任务生成、任务池化和任务同步等[92]。

当性能问题十分关键时，可以让应用程序自己创建多处理任务集合，例如：模拟、碰撞检测、遮挡测试和路径规划等[45, 92, 1445, 1477, 1854]。我们在这里会再次注意

到，有时 GPU 内核也会处于空闲状态，例如在生成阴影贴图或者进行深度 prepass 的时候，很多 GPU 核心并未被充分利用。在这样的空闲时间中，可以使用计算着色器来计算其他任务[1313, 1884]。根据架构、API 和内容的不同，有时渲染管线无法使得所有的着色器都保持忙碌，这意味着总有一些线程池可用来运行计算着色器。我们不会讨论优化这些着色器的话题，因为 Lauritzen 提出了一个令人信服的论点，即由于硬件差异和语言限制，想要编写快速且可移植的计算着色器是不可能的[993]。如何对核心渲染管线本身进行优化，是下一小节的主题。

18.5.4 图形 API 的多处理支持

并行处理通常无法很好地映射到硬件约束上。例如：DirectX 10 及更早版本的 API 中，一次只允许一个线程来访问图形驱动程序，因此实际绘制阶段的并行化处理要更加困难[1477]。

在图形驱动程序中，有两个操作可以使用多个处理器：资源创建以及与渲染相关的调用（render-related call）。创建诸如纹理和缓冲区之类的资源是纯粹的 CPU 端的操作，因此可以很自然地进行并行化处理。但是有时候资源的创建和删除也可能会阻塞任务，因为它们可能会触发 GPU 上的操作，或者是需要特定的设备上下文。在任何情况下，那些老旧的图形 API 都是在消费者级多处理 CPU 出现之前进行开发的，因此需要对其进行重写才能支持这种并发性。

这里所使用的一个关键结构是命令缓冲区（command buffer）或者命令列表（command list），这可以追溯到一个更早的 OpenGL 概念，被称为显示列表（display list）。命令缓冲区（CB）是一个包含 API 状态改变和 draw call 的列表，它可以根据需要进行创建、存储和重放命令。它们也可以被组合起来，形成一个更长的命令缓冲区。由于最终只有一个 CPU 处理器可以通过驱动程序与 GPU 进行通信，因此可以向它发送一个命令缓冲区来进行以执行。但是，每个处理器（包括这个提交处理器）都可以并行地创建命令缓冲区，或者将已经存储下来的命令缓冲区连接起来。

例如：在 DirectX 11 中，负责与驱动程序通信的处理器，会将其渲染调用发送到所谓的即时上下文（immediate context）中。而其他处理器则使用一个延迟上下文（deferred context）来生成命令缓冲区。顾名思义，这些数据并不会直接发送给驱动程序，相反，它们要被发送到直接上下文中才能进行渲染，如图 18.12 所示。此外，可以将一个命令缓冲区发送到另一个延迟上下文中，然后使用这个延迟上下文来将命令缓冲区插入到自己的命令缓冲区中。除了将命令缓冲区发送到驱动程序进行执

行之外，直接上下文还可以执行 GPU 的查询和回读，而延迟上下文则无法执行这个操作。除此之外，这两种管理命令缓冲区的上下文看起来是一样的。

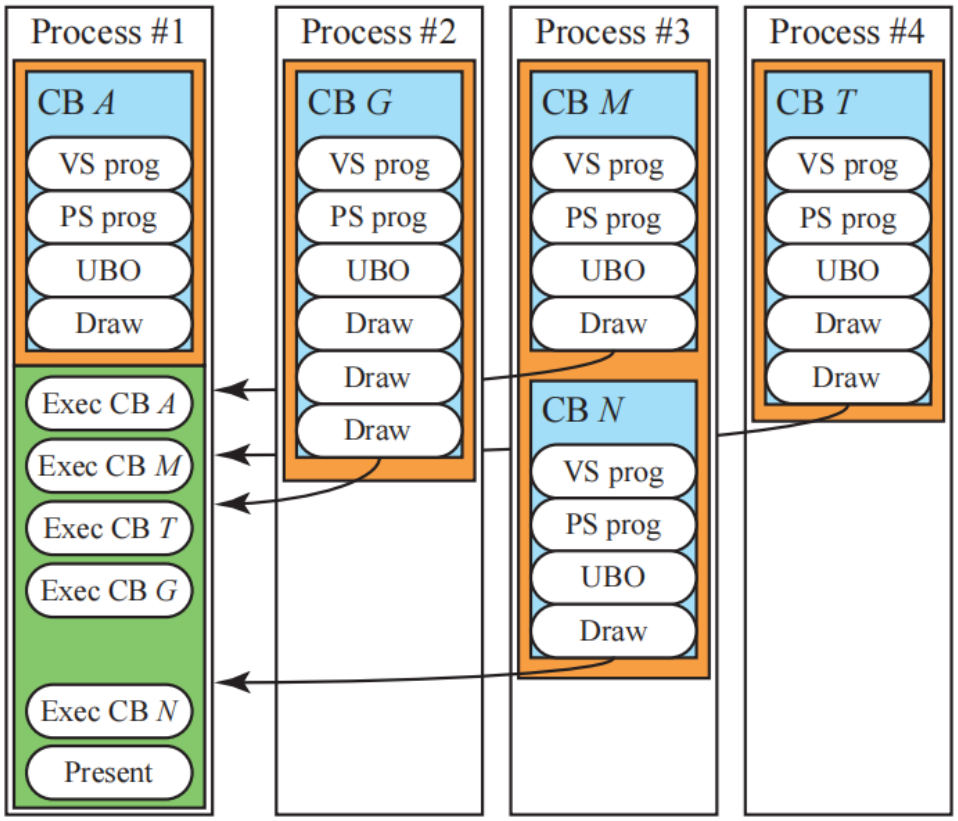


图 18.12：命令缓冲区。每个处理器都会使用一个延迟上下文（图中橙色区域）来创建、填充一个或者多个命令缓冲区（图中蓝色区域）。每个命令缓冲区都会被发送到进程#1 中，这个进程会使用一个即时上下文（图中绿色区域）来按需执行这些命令。进程#1 可以在等待进程#3 的命令缓冲区 N 的时候，执行一些其他操作。[1971]

命令缓冲区及其前身（显示列表）的一个优点在于，它们可以进行存储和重放。命令缓冲区在创建的时候没有进行完全绑定，这有助于它们的重复使用。例如：假设一个命令缓冲区中包含一个观察矩阵，此时摄像机发送了移动，因此这个观察矩阵发生了改变。然而，这个视图矩阵被存储在常量缓冲区中。而常量缓冲区中的内容实际上并没有存储在命令缓冲区中，只是存储了对它们的引用。因此可以在不重新创建命令缓冲区的情况下，对常量缓冲区的内容进行修改。关于如何最大化并行处理，涉及到选择一个合适的粒度（每个视图、每个物体、每个材质等）来对命令缓冲区进行创建、存储和组合[1971]。

在命令缓冲区成为现代图形 API 的一部分之前，这种多线程绘制系统就已经存在多年了[1152, 1349, 1552, 1554]。但是图形 API 的支持使得这个过程变得更加简单，并允许更多工具与创建的系统一起进行工作。但是，这些命令列表存在与之相关的创建开

销和内存开销。此外，在 DirectX 11 和 OpenGL 中，想要将 API 的状态设置映射到底层 GPU 上，这仍然是一个开销很大的操作，我们在章节 18.4.2 中讨论过。在这些系统中，如果应用程序成为性能瓶颈的时候，这些命令缓冲区可以提供一些帮助；但是当驱动程序成为性能瓶颈的时候，使用命令缓冲区反而可能是有害的。

这些早期图形 API 中的某些语义，不允许驱动程序对各种操作进行并行化，这有助于推动 Vulkan、DirectX 12 和 Metal 的开发工作。一个能够很好地映射到现代 GPU 的精简绘制提交接口，可以最大限度地降低这些新 API 的驱动开销。命令缓冲区的管理、内存的分配和同步决策，变成了由应用程序端进行负责的责任，而不是驱动程序的责任。此外，这些新图形 API 的命令缓冲区只会在形成时验证一次，因此重复回放操作的开销要比 DirectX 11 等早期 API 更小。所有这些因素结合起来可以有效提高 API 的效率、允许多处理、并降低驱动程序成为性能瓶颈的可能性。

补充阅读和资源

移动设备在时间分配上有着不同的平衡策略，尤其是当它们使用一个基于 tile 的渲染架构时。Merry [1200]对这些成本以及如何高效使用这类 GPU 进行了讨论。Pranckevicius 和 Zioma [1433]对有关移动设备优化的许多方面进行了深入介绍。McCaffrey [1156]比较了移动端和桌面端的架构以及性能特征。像素着色通常是移动 GPU 上最大的开销来源，Sathe [1545]和 Etuaho [443]讨论了移动设备上着色器的精度问题和优化方法。

对于桌面平台，Wiesendanger [1882]给出了一个现代游戏引擎架构的详细介绍。O'Donnell [1313]介绍了基于图形的渲染系统的优点。Zink 等人[1971]对 DirectX 11 进行了深入讨论。De Smedt [331]提供了一些关于视频游戏中常见热点的指导方法，其中包括对 DirectX 11 和 DirectX 12、多 GPU 配置方法和虚拟现实的优化。Coombes [291]给出了对 DirectX 12 最佳实践的概述，Kubisch [946]提供了一个何时使用 Vulkan 的指南。关于如何从旧 API 移植到 DirectX 12 和 Vulkan [249, 536, 699, 1438]中，由很多相应的介绍和演讲。当你读到这些文章的时候，毫无疑问会获得更多资料。查看 IHV 开发者网站，例如 NVIDIA、AMD 和 Intel；Khronos 组织的在线网站；以及本书的在线网站。

Cebenoyan 的文章[240]虽然有些过时了，但是仍然具有一些意义，这篇文章对于如何发现性能瓶颈以及如何提高效率进行了概述。Fog [476]和 Isensee [801]的指南中包含了一些流行的 c++ 优化方法，这些指南可以在网上免费找到。Hughes 等人 [783]对如何使用追踪工具，以及使用 GPUView 来分析性能瓶颈所在位置，进行了现

代而深入的讨论，虽然这个讨论侧重于虚拟现实系统，但是其中所涉及到的技术均适用于任何基于 Windows 的计算机。

Sutter [1725]讨论了 CPU 频率如何进行平衡，以及多处理器芯片组的出现。有关这种变化发生的原因，以及芯片如何进行设计的更多信息，请参阅 Asanovic 等人[75]的深度报告。Foley [478]对图形应用程序开发中各种形式的并行化处理进行了讨论。Game Engine Gems 2 [1024]有几篇关于为游戏引擎编程使用多线程元素的文章。Preshing [1445]展示了育碧是如何使用多线程的，并给出了使用 C++ 11 线程支持的细节。Tatarchuk [1749, 1750]对游戏《命运（Destiny）》中所使用的多线程架构和着色管线给出了两个详细的介绍。

Chapter 19 Acceleration Algorithms 加速算法

> Lewis Carroll——“Now here, you see, it takes all the running you can do to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!”

> 刘易斯·卡罗尔——“现在，你看，你需要用尽全力才能保持原地。如果你想去别的地方，你必须跑得至少比现在快一倍!”（英国著名作家，《爱丽丝梦游仙境》的作者；1832—1898）（这个典故出自于《爱丽丝镜中奇遇》（仙境的续作）中的红色皇后竞赛，也称为“红色皇后”效应，即拼命奔跑以保持在原地。）

关于计算机的一个伟大神话（great myths）是，终有一天我们将会拥有足够的处理能力。即使是在相对简单的应用程序中（例如文字处理），这些额外的处理能力也可以用于实现各种额外的功能，例如即时拼写、语法检查、抗锯齿文本显示和语音输入等。

而在实时渲染中，我们至少有四个性能目标：每秒更多的帧数、更高的分辨率和采样率、更加真实的材质和光照效果、以及更多的几何复杂性。通常认为每秒 60–90 帧的速度就已经足够快了。但即使使用了运动模糊，降低了图像质量所需的帧率，但是在与场景交互的时候，仍然需要较高的帧率来最小化延迟[1849]。

如今，我们已经有了分辨率为 3840×2160 的 4k 显示器；以及分辨率为 7680×4320 的 8k 显示器，虽然 8k 显示器还不是很常见。一个 4k 显示器每英寸大约有 140–150 个点（dots per inch，DPI），这个指标有时也被称为每英寸像素（pixels per inch，PPI）。而手机显示屏的数值则高达 400 DPI 左右。如今许多打印机公司都提供了 1200 DPI 的打印分辨率，这是 4k 显示器像素数量的 64 倍。即使在屏幕分辨率有限的情况下，抗锯齿效果也会增加生成高质量图像所需的样本数量。我们在章节 23.6 中讨论过，每个颜色通道的 bit 数也可以进行增加，从而需要更高精度的计算（开销也更大）。

正如前面的章节所提到的，描述和评估一个物体的材质在计算上是十分复杂的。对光线和表面的相互作用进行建模，这个过程可以消耗任意高的计算能力。这当然是真

的，因为一副图像最终是由光源所发出的光线，传播到眼睛中的无限条路径所形成的。

帧率、分辨率、着色效果总是可以做得更加复杂，但是增加其中的任何一项，都会带来边际收益递减的感觉。然而，对于场景复杂度而言，实际上并没有一个真正的上限。在这个波音 777 的渲染图中，包括了 132500 个独特的部件和超过 300 万个紧固件，这将产生一个超过 5 亿个多边形的模型[310]，如图 19.1 所示。即使其中的大多数物体都会由于尺寸过小或者位置原因而无法被看到，但是也必须做一些工作来确定它们的情况确实如此。如果不使用一些技术来降低所需的计算量，那么 z-buffer 和光线追踪都无法处理这样的模型。因此我们的结论：总是需要一些加速算法。

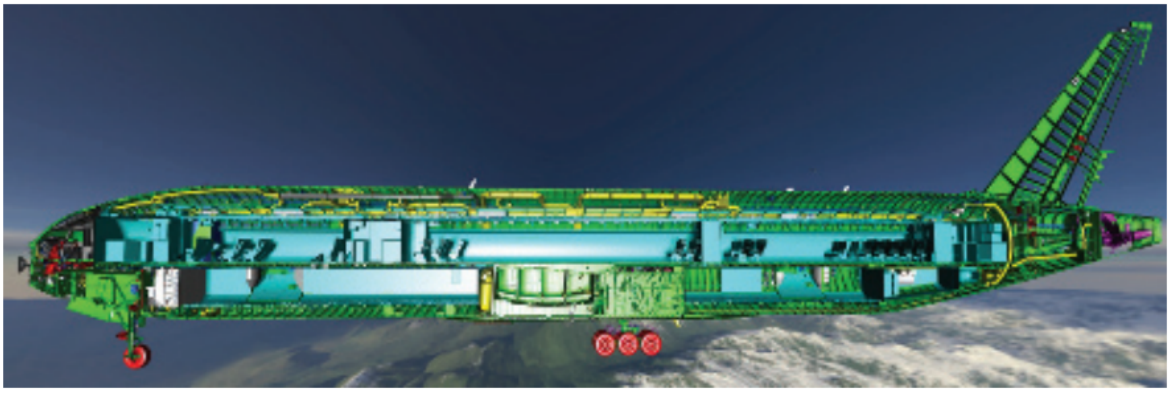


图 19.1：一个“简化”的波音飞机模型，只有 3.5 亿个三角形，使用光线追踪进行渲染。通过使用用户定义的近裁剪平面来实现切片效果。

在本章节中，我们提供了一系列加速计算机图形渲染的算法，尤其是针对大量几何图形的渲染。其中许多算法的核心都基于某种空间数据结构（spatial data structure），我们将在下一小节中介绍这些结构。基于这些知识，我们还会进一步使用剔除技术（culling technique）。这些剔除算法试图快速确定哪些物体是可见的，然后再对这些可见物体进行后续处理。细节层次技术（level of detail，LOD）降低了渲染剩余物体的复杂性。在本章节的最后，我们将会讨论渲染巨大模型的系统，包括虚拟纹理、流、编码转换和地形渲染等话题。

19.1 空间数据结构

空间数据结构指的是一个在 n 维空间中组织几何物体的数据结构，本书只讨论了二维结构和三维结构，但是这些概念通常可以很容易地扩展到更高的维度。这些数据结构可以用于对一些查询进行加速，例如几何物体是否发生了重叠等。此类查询可以用于

各种各样的操作，例如：剔除算法、测试光线与场景物体的求交算法、以及碰撞检测等。

空间数据结构的组织方式通常都是层次化的。粗略地说，这意味着顶层结构中包含一些子层级结构，每个子层级可以定义自己的空间体积，而这些空间中又包含了各自的子层级。因此，这类结构一般都是嵌套的，具有递归性质，场景中的几何物体被这个层次结构中的一些元素所引用。使用层次结构的主要原因在于，不同类型的查询速度能够显著提高，通常会从 $O(n)$ 提高到 $O(\log n)$ 。也就是说，在执行查找给定方向上最近物体等操作时，我们不需要搜索所有的 n 个物体，而是只会访问一个较小的子集。这些空间数据结构的构建时间可能会非常长，具体取决于结构中的几何物体数量，以及所需要的数据结构质量。然而，在这个领域中已经取得了很大的进展，大大减少了构建结构所需要的时间，并且在某些情况下可以实时完成。通过延迟计算（lazy evaluation）和增量更新（incremental update），这些结构的构造时间可以进一步降低。

一些常见的空间数据结构类型包括：层次包围体（bounding volume hierarchy）、二叉空间划分（binary space partitioning，BSP）树的各种变体、四叉树和八叉树等。其中 BSP 树和八叉树是基于空间细分的数据结构。这意味着场景的整个空间都会在数据结构中被细分和编码。例如：所有叶子节点空间的并集就等于整个场景空间。通常叶子节点的体积之间并不会发生重叠，除了不太常见的结构之外，例如松散八叉树（loose octree）。BSP 树的大多数变体都是不规则的（irregular），这意味着空间可以被任意细分。八叉树是规则的（regular），这意味着空间会以均匀的方式进行划分。尽管有着更多的限制，但是这种一致性通常可以成为效率的来源。另一方面，层次包围体并不是空间细分结构，相反，它包围了几何物体周围的空间区域，因此 BVH 不需要严格包围每一层的所有空间。

下面我们将会介绍 BVH、BSP 树和八叉树，以及场景图（scene graph），场景图是一种更加关心模型关系而不是高效渲染的数据结构。

19.1.1 层次包围体

包围体（bounding volume，BV）是指包含一组物体的空间体积。BV 的思想是，它应当是一个比所包含的物体更加简单的几何形状，因此我们使用 BV 来进行相交测试，要比使用内部物体本身快得多。BV 的例子有很多，包括：球体、轴对齐包围盒（axis-aligned bounding box，AABB）、定向包围盒（oriented bounding box，OBB）和 k-DOP 等。相关定义详见章节 22.2。BV 在视觉上对所要渲染的图像没有

任何贡献，相反，它会作为一个有界物体的代理形状，用来加速渲染、选择、查询以及其他的一些计算。

对于三维场景的实时渲染而言，层次包围体结构常用于分层视图的视锥体裁剪（章节 19.4）。整个场景会以层次化的树状结构进行组织，并由一组相连接的节点构成。最顶部的节点是根节点，它没有父节点。一个内部节点（internal node）包含了指向其子节点（即其他节点）的指针。因此，这棵树的根节点实际上就是一个内部节点，除非它是这棵树中的唯一节点。在最底部的叶子节点中，包含了要进行渲染的实际几何图形，叶子节点没有任何子节点。树中的每个节点（包括叶子节点）都有一个包围体，这个包围体会将整个子树中的所有几何物体都包围起来。也可以决定从叶子节点中移除 BV，将这个对应的 BV 包含在叶子节点上方的内部节点中，这种设置方式就是名称层次包围体的来源。每个节点的 BV 都包含了其子树中所有叶子节点的几何形状，这也意味着根节点中有一个包含整个场景的 BV。图 19.2 展示了一个 BVH 的例子，请注意，一些较大的包围圆可以变得更加紧密一些，因为每个节点只需要包含其子树中的几何物体即可，并不需要包含后代节点的 BV。对于包围圆（或者包围球）而言，形成这样的紧密节点可能是昂贵的，因为每个节点都必须对其子树中的所有几何物体进行检查。在实践中，一个节点的 BV 通常是通过树结构“自下而上”进行构建的，即创建一个包含其子节点 BV 的 BV。

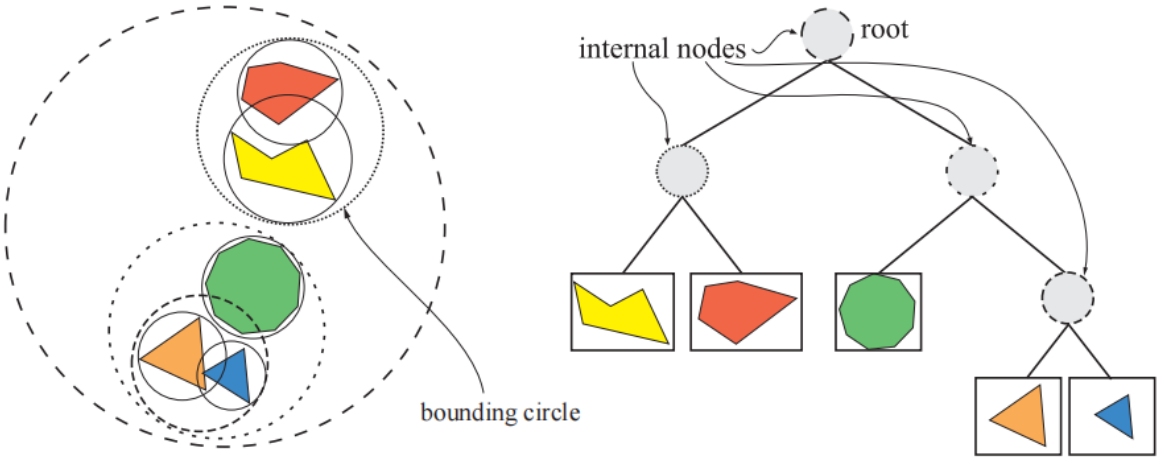


图 19.2：左侧展示了一个简单的场景，总共包含五个物体，并展示了右侧层次包围体中所使用的包围圆。一个包围圆中包含了内部的所有物体，大圆中包含了更小的圆，这是一种递归的包含关系。右侧展示了用于表示左侧物体的层次包围体结构（树）。

BVH 的底层结构是一棵树，在计算机科学领域中，有关树数据结构的文献非常多。下面我们只会介绍几个重要的结论。想要了解更多信息，可以参考 Cormen 等人[292]的《Introduction to Algorithms》一书。

考虑一个 k 叉树（k-ary tree），即每个内部节点有 k 个子节点的树。对于只有一个节点（根节点）的树，我们称这棵树的高度为 0；根节点的叶子节点的高度为 1，以此类推。平衡树（balanced tree）是指所有叶子节点的高度为 h 或者 $h-1$ 的树。一般来说，一棵平衡树的高度 h 为 $\lfloor \log_k n \rfloor$，其中 n 是树的节点总数（包括所有内部节点和叶子节点）。请注意，k 越大，树的高度就越低，这也意味着遍历这棵树所需的步骤越少，但是每个节点上的工作量会变大。二叉树（binary tree）通常是一种最简单的选择，也能够提供合理的性能表现。然而，有一些证据表明，较高的 k（例如：$k=4$ 或者 $k=8$）对于某些应用程序有着更好的性能表现[980, 1829]。使用 $k=2$、$k=4$ 或者 $k=8$ 可以简化树的构造过程；只要沿着 $k=2$ 的最长轴进行细分，沿着 $k=4$ 的两个最长轴进行细分，沿着 $k=8$ 的所有轴进行细分即可。对于其他的 k 值而言，很难形成更好的树结构。从性能的角度来看，每个节点中包含子节点数目较高的树（例如 $k=8$）通常是首选的，因为它们可以降低树的平均深度，以及间接引用（从父节点指向子节点的指针）的数量。

BVH 非常适合执行各种查询。例如：假设一根光线会与场景相交，我们需要找到并返回第一个相交点，就像阴影光线（shadow ray）一样。想要使用一个 BVH 来执行这个过程，首先会从根节点进行测试。如果这条光线没有命中它的 BV，那么它就会错过这个 BVH 中包含的所有几何形状。否则，会递归进行测试，即对这个根节点中的所有子节点的 BV 进行测试。一旦这条光线没有命中其中的某个 BV，那么会终止在该子树上的所有后续测试。如果这条光线击中了某个叶子节点的 BV，则会根据该叶子节点的几何物体，对光线进行真正的相交测试。之所以使用 BVH 可以加速求交，是因为使用这些 BV 来对光线进行相交测试的速度非常快。这也是为什么会使用一些简单的物体，例如球体和 box 来作为 BV。另一个性能提升的原因在于，BV 结构是相互嵌套的，如果我们没有与一个大 BV 相交，那么说明其内部的小 BV 也不可能相交，这样可以允许我们提前终止测试，来跳过大面积的无效空间。

通常我们想要的是最近的交点，而不是第一个被发现的交点。因此我们需要一个额外的数据，来记录遍历树结构的时候，所找到最近物体的距离和标识。这个当前的最近距离也可以用于在遍历期间对树进行剔除。如果我们与一个 BV 相交，但是其相交距离超过了目前我们所找到的最近距离，那么就可以丢弃这个 BV。在检查一个父包围盒的时候，我们会与所有的子包围盒进行相交测试，并找到其中最近的那个子包围盒。如果在这个 BV 的后代节点中发现了交点，则可以使用这个最新的最近距离，来判断是否需要对其他子节点进行遍历。正如我们将要看到的，BSP 树比普通的 BVH 更有优势，因为 BSP 树可以保证前后有序，而 BVH 只能提供这种粗略的排序效果。

BVH 也可以用于动态场景[1465]。当 BV 中的物体发生移动的时候，只需检查这个物体是否仍然包含在其父物体的 BV 中。如果是的话，那么这个 BVH 就仍然有效。如果不是的话，那么就删除这个节点，并重新计算其父节点的 BV。然后再将这个节点从根节点递归插入到 BVH 树中。另一种方法是按照需要对其父节点的 BV 进行递归扩展，从而将该子节点保留在树中。无论使用哪种方法，随着对 BVH 的编辑越来越多，这棵 BVH 都可能会变得不平衡和更加低效。另一种方法是在一段时间内，对物体的运动极限设置一个 BV，这被称为时域包围体（temporal bounding volume）[13]。例如：可以对钟摆设置一个包围盒，这个包围盒会包围钟摆运动所扫出的整个体积。另一种方法是执行一个自下而上的调整（refit）[136]，或者选择部分树结构来进行调整或者重建（rebuild）[928, 981, 1950]。

想要创建一个 BVH，首先必须能够计算一组物体的紧密 BV，这个话题将在章节 22.3 中进行讨论。然后，我们要创建 BV 的实际层次结构。有关更多 BV 构建策略的信息，请参阅在线网站 realtimerendering.com 上的碰撞检测章节（第 25 章）。

19.1.2 BSP 树

在计算机图形学中，二叉空间划分树（binary space partitioning tree），简称 BSP 树，有着两种明显不同的形式：轴对齐（axis-aligned）和多边形对齐（polygon-aligned）。通过使用一个平面来将空间划分成两部分，然后将场景中的几何物体分类到这两个空间中，从而递归完成 BSP 树的创建。一个值得注意的特性是，如果以某种方式来遍历一个 BSP 树，那么从任何角度来看，树的几何内容都可以从前到后进行排序。对于轴对齐的 BSP 树，这种排序是近似的；而对于多边形对齐的 BSP 树，这种排序则是精确的。请注意，轴对齐的 BSP 树也被称为 k-D 树。

轴对齐的 BSP 树（k-D 树）

一棵轴对齐的 BSP 树可以按照如下方式进行创建：首先，整个场景被包围在一个轴对齐包围盒（axis-aligned bounding box, AABB）中。然后将这个包围盒递归细分为更小的包围盒。现在，假设我们有一个任意递归级别的包围盒，选择这个包围盒的一个轴并生成一个垂直平面，使用这个平面来将包围盒空间划分为两个子包围盒。有些方案会使用一个固定的划分平面（partitioning plane），从而将这个包围盒精确地分割成两半，而其他的一些方案则允许划分平面改变自身的位置。通过这种平面位置的变化（被称为非均匀细分），最终生成的树可以变得更加平衡。而对于一个固定的划分平面位置（被称为均匀细分），节点在树中的位置会隐式给出它在内存中的位置。

在一个包围盒中，可能会有一些物体与划分平面相交，对于这些物体有多种处理方法，例如：这些物体可以被存储在树的这一层中，或者是作为两个子包围盒的成员，再或者是被这个平面分割成两个单独的物体。以树结构进行存储的好处在于，树中只有物体的一个副本，想要删除一个物体是很简单的。然而，与划分平面相交的小物体则会滞留在树的上层结构中，这样往往是低效的。将相交物体放置到两个子节点中，可以为较大物体提供更加紧密的包围盒，因为对于那些相交物体而言，它们会渗透到一个或者多个叶子节点中。每个子包围盒中都会包含一定数量的物体，并且会重复这个平面划分的过程，对每个 AABB 进行递归细分，直到满足某些标准才会停止。图 19.3 展示了一个轴对齐的 BSP 树。

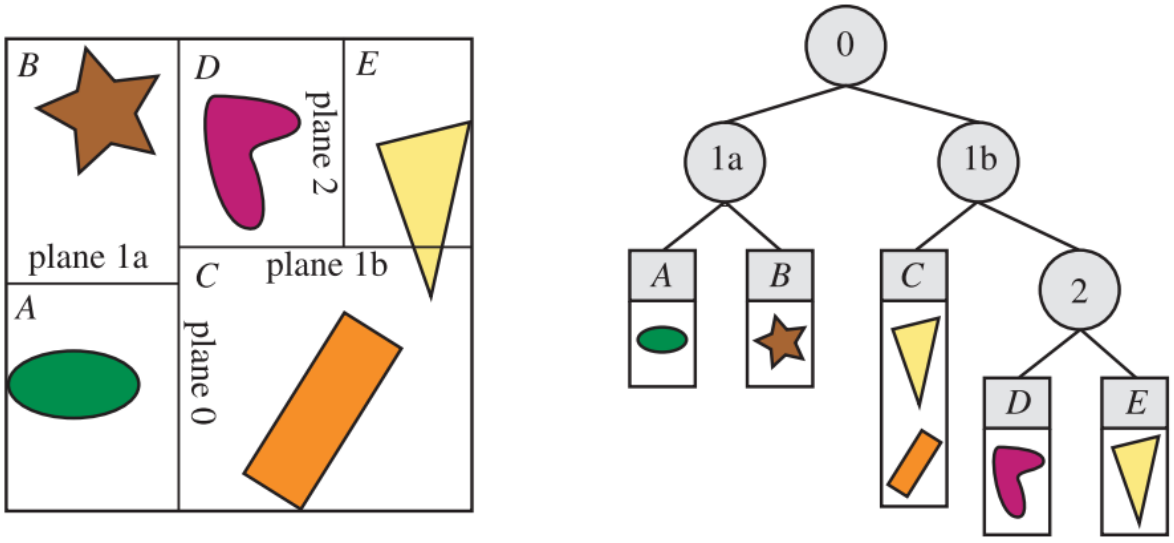


图 19.3：轴对齐的 BSP 树。在这个例子中，空间划分平面可以出现在轴上的任何位置上，而不仅仅是在它的中点位置。本例中所形成的空间体积被标记为 A 到 E。右侧的树结构展示了底层的 BSP 数据结构。每个叶子节点都代表一个空间区域，该空间区域中所包含的内容展示在节点的下方。请注意，图中黄色三角形位于两个区域 C 和 E 的物体列表中，因为这个三角形同时与这两个区域相重叠。

粗略的前后排序是如何使用轴对齐 BSP 树的一个例子，这对于遮挡剔除算法（章节 19.7 和章节 23.7），以及通过最小化像素过度绘制来降低像素着色器的成本而言，都是十分有用的。假设我们现在正在遍历一个名为 N 的节点，此时 N 是遍历开始时的根结点。我们会检查节点 N 的划分平面，并在观察者所在平面的一侧来继续对树进行递归遍历。因此，只有当这一半的树结构被遍历完时，我们才会开始遍历树的另一半。但是由于叶子节点中的内容并没有进行排序，而且一个物体可能会位于树的许多节点中，因此这种遍历方式并不会给出精确的前后顺序。然而它能够给出一个粗略的从前到后（front-to-back）排序，这通常来说会很有用。与观察者的位置相比较，通过在节点平面的另一侧开始遍历，可以获得大致的从后向前（back-to-

front）排序，这对于透明排序而言十分有用。BSP 遍历也可以用来测试光线与场景几何的相交情况，将观察者的位置直接转换为光线的原点即可。

多边形对齐的 BSP 树

另一种类型的 BSP 树是多边形对齐的（polygon-aligned）[4, 500, 501]。这种数据结构对于以精确排序来渲染静态几何物体或者刚性几何物体而言特别有用，这种算法在《毁灭战士》等游戏中十分流行，那时候还没有出现硬件 z-buffer。它在碰撞检测和相交测试中也有一些使用。

在多边形对齐的 BSP 树方案中，会选择一个多边形作来为分割器（divider），从而将空间划分成两部分。也就是说，会在根节点处选择一个多边形，使用该多边形所在的平面来作为划分平面，用于将场景中的其他多边形划分为两个集合。任何与划分平面相交的多边形，都会沿着交线被分成两部分。在一次划分之后，现在我们有了两个子空间，在每个子空间中，都会选择另一个多边形来作为分割器，它会对该子空间中的多边形进行进一步地划分。这个过程是递归进行的，直到所有的多边形都位于 BSP 树中。创建一个高效的多边形对齐的 BSP 树是一个耗时的过程，这种树通常只会构建一次，然后存储起来进行重复使用。图 19.4 展示了一个这种类型的 BSP 树。通常来说，最好是构建一棵平衡树，即每个叶子节点的深度相同，或者最多相差一个深度层级。

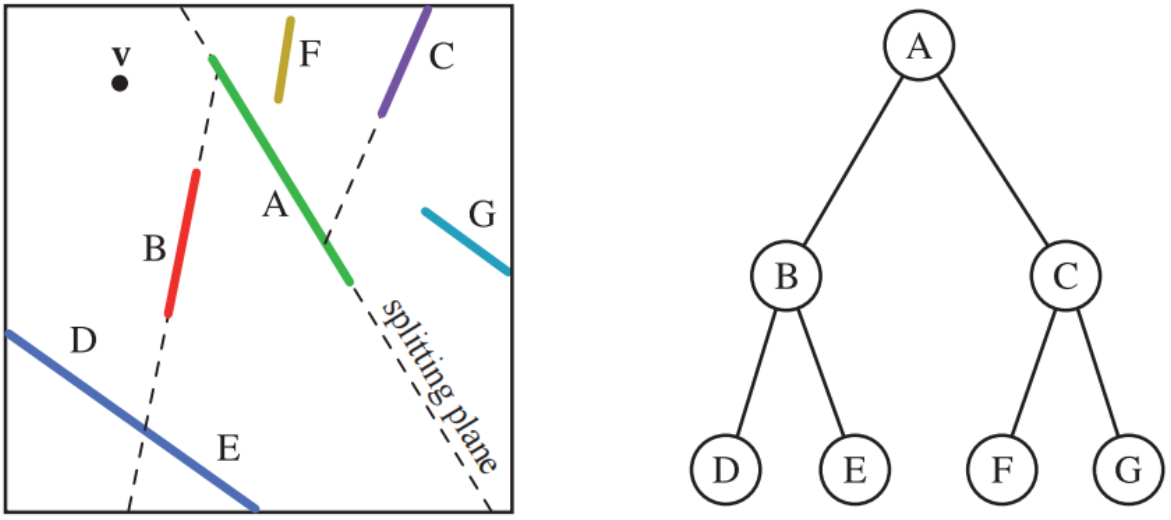


图 19.4：多边形对齐的 BSP 树。图中展示了多边形 A 到多边形 G。首先会由多边形 A 来划分空间，然后会由多边形 B 和多边形 C 来分别划分子空间。多边形 B 所形成的划分平面，会与场景左下角的深蓝色多边形相交，会将这个多边形分割为单独的多边形 D 和多边形 E，最终形成的 BSP 树如右图所示。

多边形对齐的 BSP 树具有一些有用的性质。首先，对于一个给定的视图，场景结构可以严格地按照从后到前（或者从前到后）的顺序进行遍历。与之相比，轴对齐的 BSP 树通常只能给出一个粗略的排序结果。首先确定相机此时位于根平面的哪一侧，分割平面远侧的多边形集合，肯定会位于近侧多边形集合的背后。对于远侧子空间而言，获取下一层级的划分平面并再次确定相机在位于平面的哪一侧。该空间的远侧子集就是距离相机最远的子集。通过继续这个递归过程，可以建立一个严格的从后到前的顺序，有了这个顺序，就可以使用一个画家算法（painter's algorithm）来渲染场景。画家算法在渲染场景时并不需要构建 z-buffer。如果所有物体都是按照从后到前的顺序进行绘制的，那么每个较近物体都会被绘制在较远物体的前面，因此也就不需要再比较 z-depth 了。

例如：考虑图 19.4 中观察者 $\mathbf{v}$ 所看到的场景内容。无论观察方向和视锥体的情况如何，$\mathbf{v}$ 都会位于多边形 A 形成的分割平面左侧，所以多边形 C、F、G 会位于多边形 B、D、E 的后面。对比 $\mathbf{v}$ 和多边形 C 的分割平面，我们会发现多边形 G 位于这个平面的另一侧，因此多边形 G 会被首先渲染。然后对多边形 B 的划分平面进行测试，我们发现多边形 D 应当在多边形 E 之前进行绘制。因此，从后到前的顺序依次是多边形 G，C，F，A，E，B，D。注意，这个顺序并不能保证一个物体会比另一个物体更加靠近观察者。相反，它提供的是一个严格的遮挡顺序，这是一个微妙的区别。例如：多边形 F 要比多边形 E 更加接近观察者 $\mathbf{v}$，尽管多边形 F 在遮挡顺序中要更加靠后。

19.1.3 八叉树

八叉树（octree）类似于轴对齐的 BSP 树。一个 box 会同时沿着三个轴进行分割，并且分割点必须位于 box 的中心。这将会创建 8 个新的 box，因此被称为八叉树。这会使得结构更加规则化，从而让一些查询变得更加高效。

八叉树是通过将整个场景包围在一个最小的轴对齐包围盒中来进行构建的。剩下的过程本质上是递归的，会在满足终止条件时结束。与轴对齐的 BSP 树一样，这些终止条件可以包括达到最大递归深度、或者在一个 box 中获得一定数量的图元[1535, 1536]。如果满足了某个条件，算法会将这些图元绑定到这个 box 上，并终止递归过程。否则，将会沿着这个 box 的主轴，使用三个平面来对这个 box 进行进一步地细分，从而形成 8 个相同大小的 box。对于形成的每个新 box，会再次进行测试，并可能会被再次细分为 $2 \times 2 \times 2$ 个更小的 box。图 19.5 中以二维的形式进行了说明，这个数据结构被称为四叉树（quadtree）。四叉树是八叉树的二维等效形式，即忽略

了第三个轴。相比于沿着三个轴来对数据进行分割，如果几乎没有什么好处的情况下，四叉树可能会更加有用。

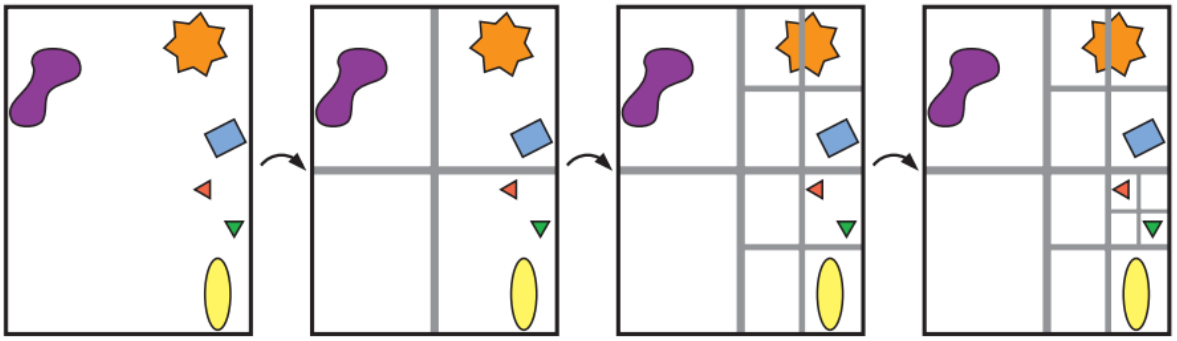


图 19.5：四叉树的构造过程。构造从最左侧开始，首先会将所有物体都包含在一个包围盒中。然后将这些 box 递归地划分为四个大小相等的 box，直到每个 box 为空、或者包含一个物体位置（这是本例中的终止条件）。

八叉树可以像轴对齐的 BSP 树一样使用，因此它也可以用于处理相同类型的查询。事实上，BSP 树可以提供与八叉树相同的空间划分结果。如果一个单元格首先沿着 x 轴的中点进行划分，然后两个子单元格沿着 y 轴的中点进行划分，最后这些子单元格沿着 z 轴的中点进行划分，那么就会形成 8 个相同大小的单元格，这些单元格与使用八叉树进行一次划分所得到的单元格完全相同。八叉树效率的一个来源在于，它不需要存储 BSP 树结构所需的额外信息，例如：八叉树中划分平面的位置是已知的，因此不需要进行明确描述。这种更加紧凑的存储方案在遍历过程中，可以访问更少的内存位置，从而节省了时间。但是，轴对齐的 BSP 树仍然可能会是更加高效的，因为更好的划分平面位置可以节省内存开销和遍历时间，它可以抵消在检索划分平面位置时所带来的额外内存开销和遍历时间。没有一个整体最佳效率的方案，实际的效率会取决于底层几何物体的性质、访问结构的使用模式、以及运行代码的硬件体系结构等因素。通常而言，内存布局的局部性和缓存友好程度是最重要的因素，这是下一小节的重点内容。

在上面的描述中，物体总是会存储在叶子节点中。因此，某些物体必须被存储在多个叶子节点中。另一种选择是将每个物体都放置在包含整个物体的最小 box 中，例如：图 19.5 中的橙色星形物体，应当放置在左起第二张插图的右上角 box 中。这种方式会有一个明显的缺点，例如：位于八叉树中心的一个小物体，将会被放置在整棵树的最顶层节点（最大 box）中。这是十分低效的，因为一个微小的物体会被包含在整个场景的包围盒中。一种解决方案是将物体进行拆分，但是这样会引入更多的图元。另一种解决方案是在每个叶子节点的 box 中放置一个指向物体的指针，但是这样做会降低效率，并使八叉树的编辑变得十分困难。

Ulrich 提出了第三种解决方案，即松散八叉树（loose octree）[1796]。松散八叉树的基本思想与普通的八叉树相同，但是对每个 box 的大小选择有所放宽。如果普通 box 的边长为 l，那么松散八叉树中则使用了 kl 进行代替，其中 $k > 1$，图 19.6 展示了 $k = 1.5$ 时的情况，并与普通的八叉树进行了比较。请注意，box 的中心点位置是相同的。通过使用一个更大的 box，会减少跨越分割平面的物体数量，从而使得物体可以放置在八叉树的更深处。由于一个物体总是只会被插入到一个八叉树节点中，因此从八叉树中删除某个物体的操作是很简单的。使用 $k = 2$ 可以获得一些额外的好处。首先，插入物体和删除物体的时间复杂度是 $O(1)$。知道一个物体的大小，意味着我们可以立即知道它能够成功插入的八叉树级别，即可以完全放入一个松散的 box 中。在实践中，有时可以将物体放置到八叉树中更深的 box 中。同样，如果 $k > 2$，当一个物体不适合当前级别的时候，那么可能要将其放入到树的上层级中。

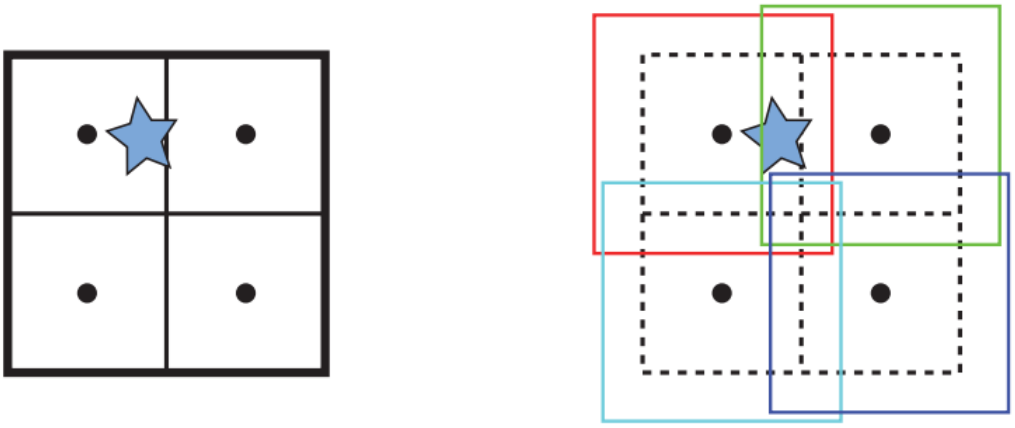


图 19.6：普通八叉树与松散八叉树的对比。图中的圆点表示 box 的中心点（第一次细分的 box）。在左边，蓝色星星跨越了普通八叉树的一个划分平面。因此，一种选择是将星星放在最大的 box 中（即根节点 box）。在右边，是一个 $k = 1.5$ 的松散八叉树（即 box 增大 50%）。这些 box 稍稍进行了一些移动，以便于辨认。此时星星可以完全放置在左上角的红色 box 中。

物体的质心决定了它会被放置在哪个松散八叉树的 box 中。由于这些特性，这个结构非常适合用于动态物体的绑定，但是代价是会牺牲一些 BV 的效率，并且在遍历该结构的时候，会失去排序顺序。此外，物体在两帧之间通常只会发生轻微移动，因此前一个帧中的 box 在下一帧中大概率会仍然有效。因此，在松散八叉树中，只有一小部分动画物体需要每帧进行更新。Cozzi 指出[302]，在将每个物体/图元分配给松散八叉树之后，可以计算每个节点中物体的最小 AABB，此时这个点实际上就变成了一个 BVH。这种方法避免了跨节点的物体拆分。

19.1.4 缓存无关和缓存感知的表示

由于内存系统的带宽与 CPU 的计算能力之间的差距每年都在增加，因此在设计算法和空间数据结构的时候，考虑缓存是至关重要的。在本小节中，我们将介绍缓存感知（cache-aware，或者叫做缓存敏感 cache-conscious）和缓存无关（cache-oblivious）的空间数据结构。缓存感知代表会假设缓存块的大小是已知的，因此我们可以针对特定的体系结构进行优化。相比之下，缓存无关的算法可以很好在所有大小的缓存上进行工作，因此是平台无关的。

想要创建一个缓存感知的数据结构，我们必须首先确定体系结构中的缓存块大小，例如：缓存块的大小可能是 64 字节。然后尝试最小化数据结构的大小。例如：Ericson [435]展示了如何使用 32 bit 来表示一个 k-d 树的节点，这在一定程度上是通过占用节点 32 bit 值中的最低两位来实现的。这两个 bit 组合起来可以表示四种类型：一个叶子节点、或者是在三个轴其中之一上进行划分的内部节点。对于叶子节点，较高的 30 个 bit 存储了一个指向物体列表的指针；对于内部节点，这 30 个 bit 代表了一个（精度稍低的）浮点分割值。因此，在一个 64 字节的缓存块中，可以存储包含 15 个节点的四层深度二叉树，最后的第 16 个节点会用于表示存在哪些子节点，以及这些子节点的位置，详情请参阅 Ericson 的书籍。其中的关键概念是，通过确保数据结构清晰地打包到缓存边界，从而使得数据访问的性能得到了显著改进。

一种流行且简单的缓存无关的树结构布局是 van Emde Boas 布局方法[68, 422, 435]。假设我们有一棵高度为 h 的树 $\mathcal{T}$，我们的目标对树中的节点计算一个缓存无关的布局或者排序。其中的关键思想是：将层次结构递归分解为越来越小的块，直到在某个级别上，可以将一组块将放入缓存中。这些块的位置在树中彼此靠近，因此相较于简单地自上而下列出所有节点，其缓存数据的有效时间要更长。使用自上而下的简单列表会导致内存位置之间的大范围跳转。

我们将 $\mathcal{T}$ 的 van Emde Boas 布局表示为 $v(\mathcal{T})$，这个结构是递归定义的，树中单个节点的布局就是节点本身。如果 $\mathcal{T}$ 中有多个节点，那么这棵树会在高度一半处 $\lfloor h/2 \rfloor$ 进行分割。最顶层的 $\lfloor h/2 \rfloor$ 会被放置在一棵树中，记为 $\mathcal{T}_0$，从 $\mathcal{T}_0$ 叶子节点开始的子树记为 $\mathcal{T}_1, \ldots, \mathcal{T}_n$，该树的递归性质描述如下：

$$v(\mathcal{T}) = \begin{cases} \{\mathcal{T}\}, & \text{if there is single node in } \mathcal{T} \\ \{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n\}, & \text{else.} \end{cases} \tag{19.1}$$

请注意，所有的子树 $\mathcal{T}_i$ （$0 \leq i \leq n$），也都是由上面的递归过程进行定义的。这意味着，例如：$\mathcal{T}_1$ 必须在其高度一半的地方被分割，以此类推。图 19.7 展示了这样

的一个例子。

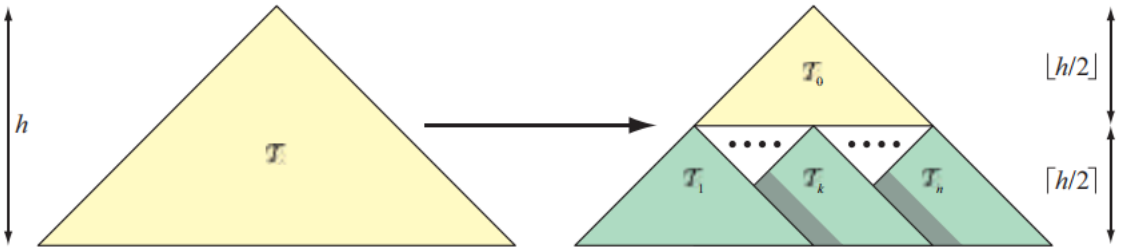


图 19.7：一棵树的 van Emde Boas 布局 $\mathcal{T}$，这是通过将树的高度 h 分成两部分来创建的。这个过程将会创建子树 $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n$，每个子树都会以同样的方式进行递归拆分，直到每棵子树只剩下一个节点。

一般来说，创建缓存无关的布局包括两个步骤：聚类（clustering）和聚类的排序。对于 van Emde Boas 布局而言，聚类由子树给出，而排序则隐含在创建顺序中。Yoon 等人[1948, 1949]专门设计并开发了为高效层次包围体和 BSP 树所使用的技术。他们开发了一个概率模型，这个模型考虑了父节点和其子节点之间的局部性以及空间局部性。这个想法的思路是：通过确保子节点低廉的访问成本，从而在访问父节点时最大限度地减少缓存未命中。此外，那些彼此接近的节点会在排序中被分组得更加接近。研究人员提出了一种贪婪算法，对概率最高的节点进行聚类。在不改变底层算法的情况下，可以大幅度地提高性能表现，不同的只是 BVH 中节点的顺序会发生一些变化。

19.1.5 场景图

BVH、BSP 树和八叉树，都使用某种树结构来作为它们的基本数据结构，它们之间的区别在于如何划分空间和存储几何物体。它们还都以分层的方式来存储几何物体，除此之外没有什么其他不同。然而，想要渲染一个三维场景，所涵盖的内容不仅仅是几何图形。还涉及对动画、可见性和其他元素的控制，通常会使用一个场景图（scene graph）来执行，在 glTF 中则被称为节点层次结构（node hierarchy）。这是一个面向用户的树形结构，它可以通过使用纹理、变换、LOD、渲染状态（例如材质属性）、光源和其他任何合适的东西来进行增强。它由一棵树来进行表示，并以某种顺序来遍历这个树，从而渲染整个场景。例如：一个光源可以放置在一个内部节点中，它只会对其子树中的内容产生影响。另一个例子是：当在树中遇到一个材质时，这个材质可以应用于该节点子树中的所有几何物体上，或者也可能会被子节点中的设置所覆盖，详见图 19.34 所示，从而了解如何在场景图中支持不同的 LOD。从某种意义

上来说，每个图形应用程序都会使用某种形式的场景图，即使这个场景图只是一个根节点，其中包含了要进行显示的子节点列表。

使物体动画化的一种方法是，改变树中内部节点的变换方式，然后场景图会对该节点子树的全部内容进行变换。由于可以在任何内部节点中进行变换，因此可以实现分层动画。例如：汽车的轮子可以进行旋转，同时汽车作为一个整体可以向前移动。

当多个节点可能指向同一个子节点的时候，这种树结构被称为一个有向无环图（directed acyclic graph，DAG）[292]。这里无环（acyclic）的意思是它不能包含任何循环或者环结构。有向（directed）的意思是两个节点会通过一条边进行连接，它们也是按照一定的顺序进行连接的，例如从父节点指向子节点。场景图通常是DAG，因为它们允许进行实例化，例如：当我们想要在不复制其几何物体的情况下，就能够复制出该物体的多个副本（实例）。图 19.8 展示了这样的一个例子，其中两个内部节点分别对各自的子树应用了不同的变换操作。使用实例可以节省很多内存开销，并且 GPU 可以通过调用图形 API 来快速渲染实例的多个副本（章节 18.4.2）。

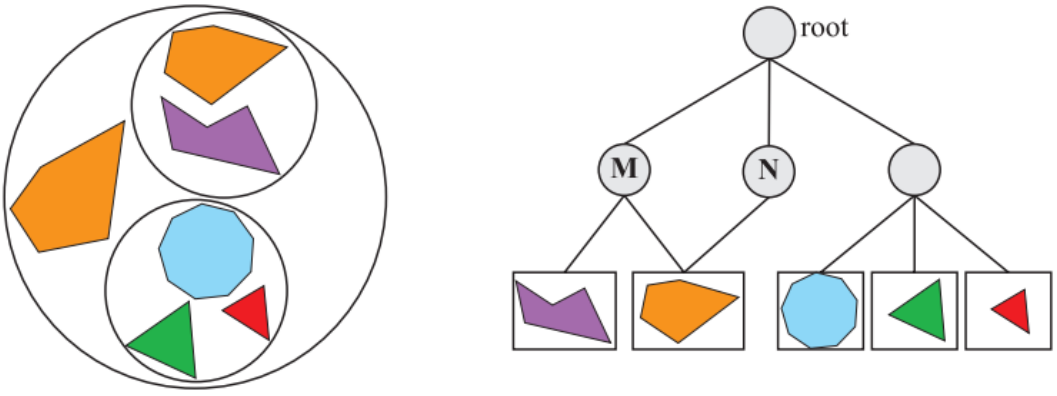


图 19.8：场景图对内部节点及其子树应用了不同的变换 M 和 N 。请注意，这两个内部节点指向的是相同物体，但是由于它们具有不同的变换，因此会出现两个不同的物体（其中一个应用了旋转和缩放）。

当物体在场景中发生移动的时候，需要对场景图进行更新，这可以通过对树结构的递归调用来实现。变形会从根节点到叶子节点的过程中进行更新，在这次遍历中，相关的变换矩阵会被相乘，并存储在相关的节点中。然而，当变换被更新的时候，任何附加的 BV 都是过时的，因此，在从叶子节点返回根节点的过程中，还需要对 BV 进行更新。由于过于松散的树结构会使得这些任务变得极为复杂，因此通常会避免使用DAG，或者是使用有限形式的 DAG，即只共享叶子节点。有关这个主题的更多信息，请参阅 Eberly 所撰写的书籍[404]。还要注意的是，当使用基于 javascript 的图形 API 时（例如 WebGL），需要将尽可能多的工作负载转移到 GPU 上，同时尽可能少地反馈给 CPU，这是非常重要的[876]。

场景图本身可以提供一些计算效率的提升。场景图中的节点通常都会具有包围体，因此与 BVH 非常相似。场景图中的叶子节点会存储几何信息。需要认识到的一点是，可以将完全不相关的高效方案与场景图一起使用。这就是一种空间化（spatialization）的思想，其中用户的场景图会为不同的任务（例如更快的剔除或者选择）创建单独的数据结构（例如 BSP 树或者 BVH）。大多数模型所在的叶子节点都是共享的，因此使用额外高效空间结构所带来的开销相对较低。

19.2 剔除技术

剔除（cull）的意思是“从一大群物体（flock）中移除某些物体”，而在计算机图形学的语境中，这正是剔除技术所做的事情。这个 flock 正是我们想要渲染的整个场景，而我们只需要移除那些对最终图像没有贡献的场景部分即可，而场景中的剩余部分则会被发送到渲染管线中。因此，可见性剔除（visibility culling）这个术语也经常被用于渲染的上下文中。不过，也可以对程序中的其他部分进行剔除，例如：碰撞检测（对屏幕外的物体，或者是隐藏的物体，计算精度不需要那么高）、物理计算和 AI 等。这里我们只会介绍与渲染相关的裁剪技术，它们包括背面剔除（backface culling），视锥体剔除（frustum culling）和遮挡剔除（occlusion culling），如图 19.9 所示。其中背面剔除移除了那些背对观察者的三角形。视锥体剔除移除了位于观察视锥体之外的三角形。遮挡剔除则会移除那些被其他物体所遮挡的物体，遮挡剔除是最为复杂的剔除技术，因为它需要计算物体之间是如何彼此影响的。

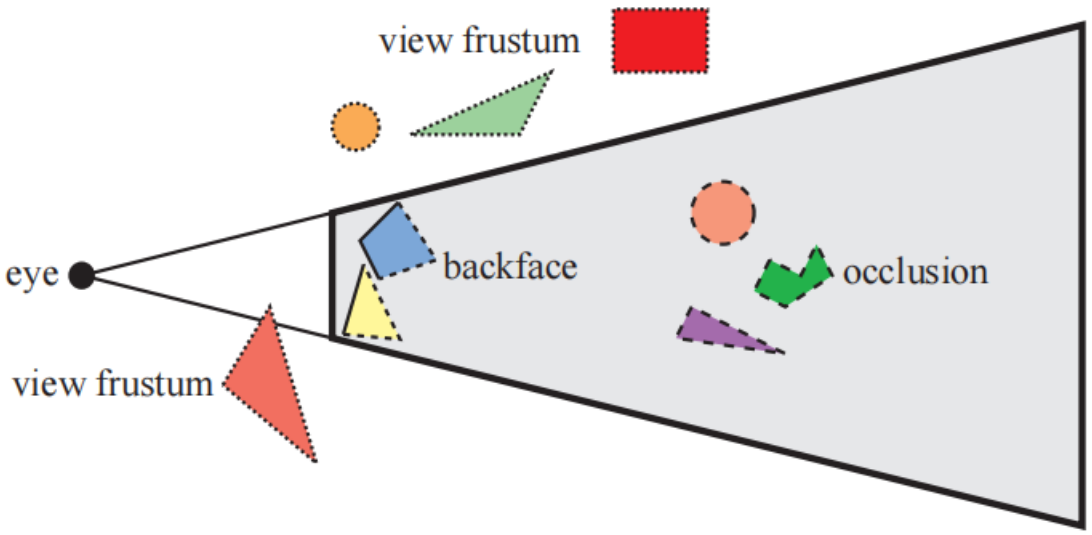


图 19.9：不同类型的剔除技术。图中使用虚线表示的物体会被剔除。[277]

理论上，实际的剔除可以发生在渲染管线的任何阶段中，对于一些遮挡剔除算法而言，它甚至可以预先进行计算。对于那些在 GPU 上实现的剔除算法，我们有时只能选择启用剔除或者禁用剔除，或者是设置一些参数。渲染速度最快的三角形，是那些从未被发送到 GPU 中的三角形。其次，在管线中越早进行进行剔除，效果就越好。剔除通常是通过使用几何计算来实现的，但是并不局限于几何计算，例如：有一些算法也可以使用帧缓冲区的内容。

对于一个理想情况下的剔除算法，它只会向管线中发送那些精确可见集（exact visible set，EVS）。在本书中，EVS 被定义为所有部分可见或者完全可见的图元。一种允许理想剔除的数据结构是方位图（aspect graph），即从任何视角中都可以提取出 EVS [532]。想要创建这样的数据结构在理论上是可行的，但是在实践中却是行不通的，因为其最坏的时间复杂度可能高达 $O(n^9)$ [277]。相反，更加实用的算法是试图找到一个集合，它被称为潜在可见集（potentially visible set，PVS），PVS 是对 EVS 的预测。如果 PVS 完全包含了 EVS，那么只有不可见的几何图形会被丢弃，我们将这样的 PVS 称为是保守的（conservative）。PVS 也可能是近似的（approximate），即 EVS 并不完全被包括在内，这种类型的 PVS 可能会生成不正确的图像，我们的目标是使这些误差尽可能地小。由于保守的 PVS 总是能够生成正确的图像，因此通常会认为它更加实用。通过高估或者近似 EVS，可以更快地计算 PVS，其中的难点在于如何进行这些评估从而获得整体上的性能表现。例如：一个算法能够在不同的粒度上来处理场景中的几何图形，可以是三角形、整个物体或者物体组。当找到一个 PVS 时，使用 z-buffer 进行渲染，它会对最终的逐像素可见性进行解析。

请注意，有一些算法可以对网格中的三角形进行重新排序，从而提供更好的遮挡剔除效果，即减少过度绘制，同时改进顶点缓存的局部性。这些技术与剔除技术有些关联，我们建议感兴趣的读者可以阅读相关的参考文献[256, 659]。

从章节 19.3 到章节 19.8，我们将讨论背面剔除（backface culling）、视锥体剔除（view frustum culling）、入口裁剪（portal culling）、细节剔除（detail culling）、遮挡剔除（occlusion culling）和剔除系统。

19.3 背面剔除

想象一下，我们正在观察场景中的一个不透明球体，大约有一半的球体我们是看不到的。从这个观察中得出的结论是：看不见的物体不需要进行渲染，因为它对最终的图像没有什么贡献。也就是说，球体的背面是不需要进行处理的，这就是背面剔除的核

心思想。这种类型的剔除不仅可以针对一个物体进行，也可以一次针对整组进行，因此也称为集群背面剔除（clustered backface culling）。

假设相机位于物体外部，并且没有发生穿透（即物体与近裁剪平面相交），那么所有不透明物体的背面三角形都可以通过进一步地处理进行剔除。如果已知投影三角形的顶点在屏幕空间中是顺时针方向的，那么说明逆时针顶点顺序的三角形（章节 16.3）就是朝后的。这个测试可以通过在二维屏幕空间中计算三角形的带符号面积来实现，面积为负意味着这个三角形应当被剔除。这个过程可以在屏幕映射完成之后立即执行。

另一种确定三角形是否背对相机的方法是：从三角形所在平面上的任意一点（最简单的就是直接选择其中一个顶点）到相机位置创建一个向量。对于正交投影而言没有一个实际的相机位置，此时到相机位置的向量会被替换为负观察方向，这对于场景来说是恒定的，我们会计算这个向量与三角形法线之间的点积。点积为负，意味着两个向量之间的夹角大于 90°，因此这个三角形肯定不是面向观察者的。这个测试过程相当于计算从相机位置到三角形平面的带符号距离，如果为正，则说明三角形是正面的。请注意，正确的距离只有在法线归一化的情况下才能得到，但是在这里并不重要，因为我们关心的只有符号。或者是在应用投影矩阵之后，在裁剪空间中构建顶点 $\overline{\mathbf{v}} = (v_x, v_y, v_w)$，并计算行列式 $d = |\overline{\mathbf{v}}_0, \overline{\mathbf{v}}_1, \overline{\mathbf{v}}_2|$ 的值[1317]。当 $d \leq 0$ 时，可以对这个三角形进行裁剪。图 19.10 展示了这些剔除技术。

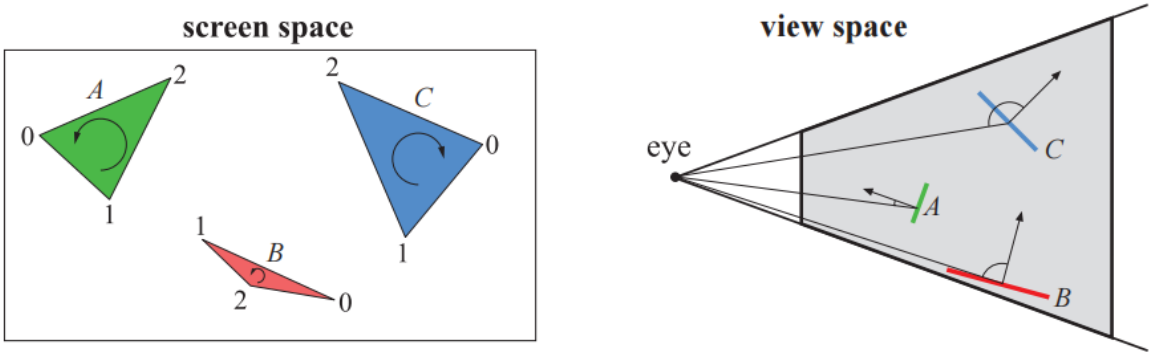


图 19.10：判断三角形是否朝后的两种不同测试。左图展示了如何在屏幕空间中进行测试。左边的两个三角形是正面的，而右边的三角形则是背面的，剔除之后可以省去进一步的处理开销。右图展示了如何在观察空间中进行背面测试。其中三角形 A 和三角形 B 是正面的，而三角形 C 是背面的。

Blinn 指出，这两种测试在几何上实际上是相同的[165]。从理论上来说，这些测试的区别主要在于计算测试的空间（坐标系），而不是其他因素。在实践中，屏幕空间中的测试通常会更加安全，因为在观察空间中看起来稍微向后的倾斜（edge-on）三角

形，在屏幕空间中可能会变得稍微向前，这是因为观察空间中的坐标会被四舍五入为屏幕空间中的亚像素坐标。

使用 OpenGL 或者 DirectX 之类的图形 API，通常可以使用一些函数来控制背面剔除，这些函数要么会启用背面剔除，要么会启用正面剔除，要么会禁用所有剔除。请注意，一次镜像变换（即负缩放操作）会将背面三角形转换为正面三角形，反之亦然[165]（章节 4.1.3）。最后，还可以在像素着色器中来确定三角形是否为正面。在 OpenGL 中，这是通过测试 $\mathrm{gl\_FrontFacing}$ 来完成的，在 DirectX 中则被称为 $\mathrm{SV\_IsFrontFace}$。在此之前，正确显示双面物体的主要方法是将它们渲染两次，第一次渲染会剔除背面；第二次渲染会剔除正面，同时反转法线。

对于标准的背面剔除有一个常见的误解，即它可以将渲染的三角形数量减少大约一半。虽然对许多物体而言，背面剔除将会移除大约一半的三角形，但是对于某些特殊类型的模型而言，它并不会带来什么好处。例如：室内场景中的墙壁、地板、天花板通常都是面向观察者的，因此在这种场景中背面三角形相对较少。类似地，在地形渲染中，通常而言大多数三角形都是可见的，只有丘陵或者峡谷等一些地面起伏较大的地形，才会出现较多的背面三角形，因此才可以从这种技术中获益。

然而，这种简单的背面剔除只能避免单个三角形被光栅化，如果我们可以通过一次测试，来决定是否可以剔除整个三角形集合，那么速度会更快。这类技术被称为集群背面剔除算法，这里我们将介绍其中的部分技术，这类算法所使用的基本概念是法线锥（normal cone）[1630]。对于表面上的某些部分，会创建一个包含所有法线方向和所有表面点的截锥体（truncated cone）。请注意，沿法线方向上，需要使用两个距离来截断这个圆锥体，图 19.11 给出了这样的一个例子。我们可以看到，这个圆锥由法线 $\mathbf{n}$、半角 α、锚点 $\mathbf{c}$ 以及沿法线截短圆锥体的偏移距离所定义。在图 19.11 的右侧展示了这个法线锥的横截面。Shirman 和 Abi-Ezzi [1630]证明，如果观察者位于正面锥体中，那么锥体中的所有面都是正面的；对于背面锥体也是如此。Engel [433]使用了一个类似的概念，被称为 GPU 剔除的排除体积（exclusion volume）。

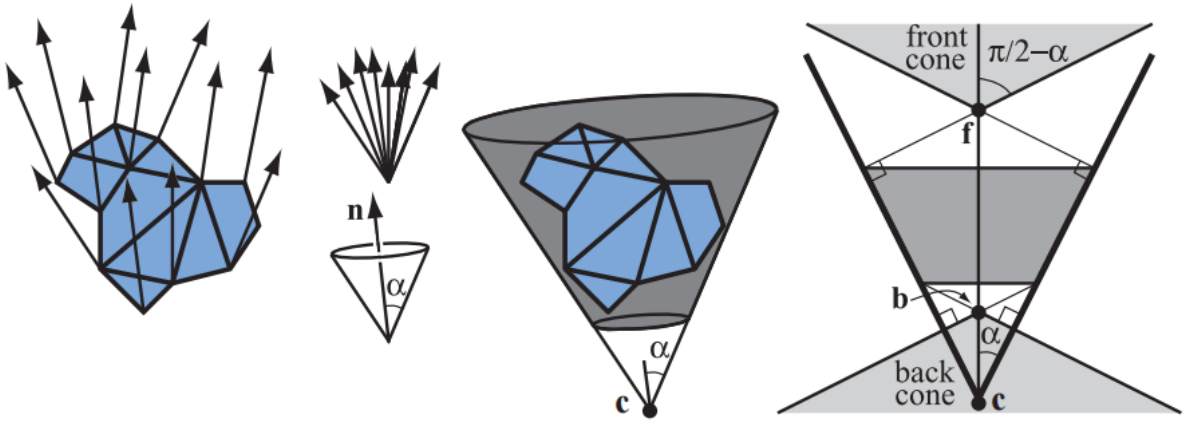


图 19.11：左：一组三角形及其法线。左中：收集法线（上），并构造一个最小锥（下），它由一个法线 $\mathbf{n}$ 和一个半角 α 所定义。右中：这个圆锥体被锚定在点 $\mathbf{c}$，并被截断，因此它也包含三角形中的所有点。右：截锥体的横截面。顶部的浅灰色区域是正面锥体（frontfacing cone），底部的浅灰色区域是背面锥体（backfacing cone）。点 $\mathbf{f}$ 和点 $\mathbf{b}$ 分别是前后锥体的顶点。

对于静态网格，Haar 和 Aaltonen [433]建议围绕 n 个三角形来计算一个最小立方体，每个立方体面被分割为 $r \times r$ 个“像素”，每个“像素”上都有一个 n 位掩码，来代表对应三角形在该“像素”上是否可见，如图 19.12 所示。如果相机位于立方体之外，那么可以找到相机所在的对应截锥体，立即查找其位掩码，并知道哪些三角形是朝后的（保守地）。如果相机位于立方体内部，那么所有三角形都会被认为是可见的（除非想执行进一步的计算）。Haar 和 Aaltonen 在每个立方体面只使用一个位掩码，并一次编码 $n = 64$ 个三角形。通过计算位掩码中被设置的 bit 数，我们可以高效地为未被剔除的三角形分配内存。《刺客信条：大革命》中也使用了这种方法。

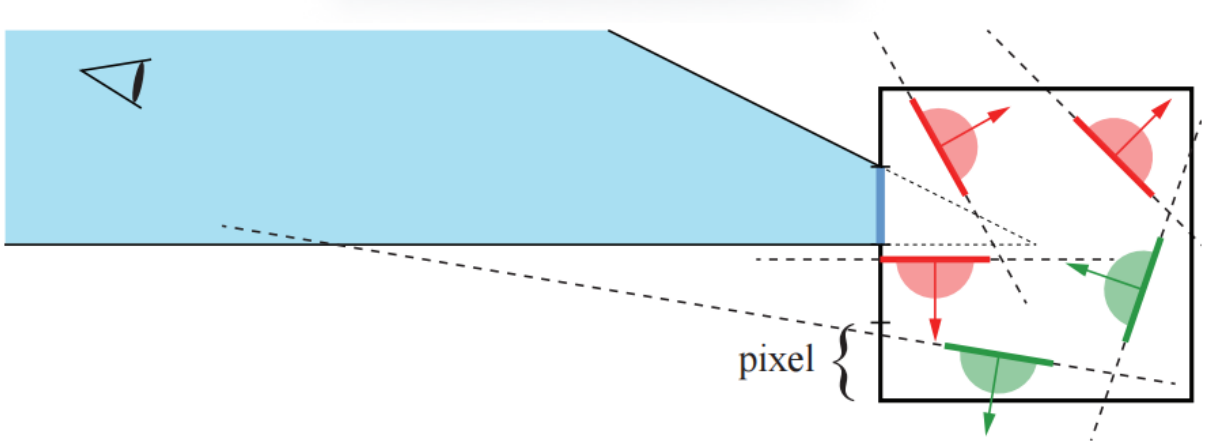


图 19.12：由五个静态三角形组成的一组三角形，从侧面进行观察，它们被一个二维正方形所包围。左边的方形面被分割为 4 个“像素”，我们聚焦于距离顶部 1 个“像素”的那个位置，它在 box 外的截锥体使用蓝色进行表示。三角形平面所形成的正半空间使用半圆进行表示（红色和

绿色）。对于截锥体中的所有点，所有在正半空间中没有蓝色截锥体的三角形（红色），都会被保守地认为面向后。绿色则代表正面。

接下来，我们将使用一个非截断的法线锥，与图 19.11 中的圆锥相反，它仅由中心点 $\mathbf{c}$ 、法线 $\mathbf{n}$ 和半角 α 进行定义。为了计算这样一个由许多三角形所组成的法线锥，我们会取三角形平面上的所有法线，并将它们放在相同的位置上，在单位球面上计算一个包含所有法线的最小圆[101]。第一步，假设从点 $\mathbf{e}$ 开始，我们要对锥体内所有共享原点 $\mathbf{c}$ 的法线进行背面测试（backface-test）。如果下列条件成立的话，则说明法线锥背对点 $\mathbf{e}$ [1883, 1884]：

$$\mathbf{n} \cdot (\mathbf{e} - \mathbf{c}) < \underbrace{\cos\left(\alpha + \frac{\pi}{2}\right)}_{-\sin\alpha} \Longleftrightarrow \mathbf{n} \cdot (\mathbf{c} - \mathbf{e}) < \sin\alpha \tag{19.2}$$

然而，这个测试只适用于所有几何图形都位于点 $\mathbf{c}$ 的情况。接下来，我们假设所有几何图形都位于一个圆心为 $\mathbf{c}$ 、半径为 r 的球体内，那么这个背面测试为：

$$\mathbf{n} \cdot (\mathbf{e} - \mathbf{c}) < \underbrace{\cos\left(\alpha + \beta + \frac{\pi}{2}\right)}_{-\sin(\alpha+\beta)} \Longleftrightarrow \mathbf{n} \cdot (\mathbf{c} - \mathbf{e}) < \sin(\alpha + \beta) \tag{19.3}$$

其中 $\sin\beta = r/\|\mathbf{c} - \mathbf{e}\|$ 。图 19.13 展示了推导这个测试所涉及的几何图形。这些量化法线可以存储为 8×4 个 bit，这对于某些应用而言可能已经足够了。

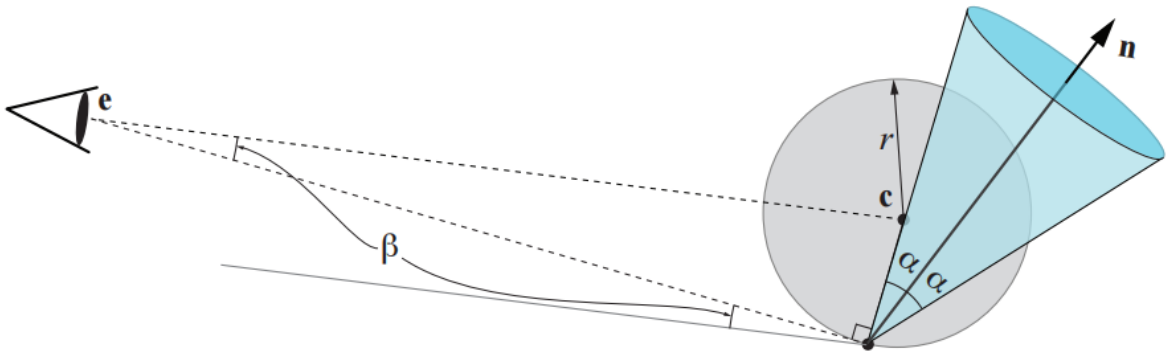


图 19.13：这种情况表明，当由中心点 $\mathbf{c}$ 、法线 $\mathbf{n}$ 和半角 α 所定义的法线锥，即将从半径为 r 和圆心 $\mathbf{c}$ 的圆内临界点对点 $\mathbf{e}$ 可见时，就会出现极限情况。这种情况发生在从点 $\mathbf{e}$ 到圆上一点的向量与圆相切，并且与垂直于法线锥的边（母线）时。请注意，法线锥从点 $\mathbf{c}$ 向下平移，因此它的原点会与球体边界相重合。

作为本小节的总结，这里我们提及一下运动模糊三角形的背面剔除，其中三角形的每个顶点在一帧内都有一个线性运动，这并不像人们想象的那么简单。一个三角形的顶点随着时间发生线性移动，这个三角形可能会在这一帧刚开始的时候向后，然后向前，然后再向后，所有这些都可能发生在同一帧内。因此，如果因为一个三角形在这帧开始和这帧结束的时候都向后，就将这个三角形剔除，那么可能会产生错误的结果。Munkberg 和 Akenine-Moller [1246]提出了一种方法，将标准背面测试中的顶点替换为线性移动的三角形顶点。他们将这个测试改写为 Bernstein 形式，并利用 Bezier 曲线的凸性质来作为一个保守检验。对于景深效果而言，如果整个镜头都位于三角形的负半空间的话（换句话说，位于三角形的背后），那么这个三角形可以安全地被剔除。

19.4 视锥体剔除

如章节 2.3.3 所述，只有完全或者部分位于视锥体内部的图元才需要进行渲染。加速渲染的一种方法是，将每个物体的包围盒与视锥体进行相交测试。如果该 BV 位于视锥体外部，那么它所包含的几何形状就不需要进行渲染；如果该 BV 位于视锥体内部或者与视锥体相交，那么它所包含的几何形状就可能是可见的，因此必须要发送到渲染管线中。请参阅章节 22.14，来了解各种包围体和视锥体之间的相交测试方法。

通过使用一种空间数据结构，这种剔除可以分层进行[272]。对于层次包围体，从根节点开始的前序遍历（preorder traversal）就可以完成这项工作[292]。每个具有包围体的节点都会针对视锥体进行测试，BV 与视锥体的相交测试一共会出现三种情况，分别是：BV 完全位于视锥体内部；BV 部分位于视锥体内部；BV 完全位于视锥体外部。如果该节点的 BV 位于视锥体之外，那么就不需要对该节点进行进一步处理，此时会对这棵树进行修剪，因为该 BV 中的内容以及子节点中的内容都位于视野之外。如果该 BV 完全位于视锥体内部，则说明该 BV 中的内容肯定全都在视锥体内部。此时会继续进行遍历，但是不再需要对该子树的其余部分进行进一步的视锥体测试了。如果该 BV 与视锥体相交，那么还需要继续进行遍历，并对其子节点的 BV 进行相交测试。当我们发现一个叶子节点与视锥体相交时，其内容（即它的几何物体）就会被送到管线中；但是这个叶子节点中的图元无法保证位于视锥体内部。图 19.14 展示了一个视锥体剔除的例子。还可以针对一个物体或者单元格使用多个 BV 进行测试。例如：假设我们发现某个单元格周围的球体 BV 与视锥体发生重叠，如果已知这个单元格 box 比球体小得多的话，那么可以进行一个更加精确（尽管更昂贵）的 OBB-视锥体相交测试[1600]。

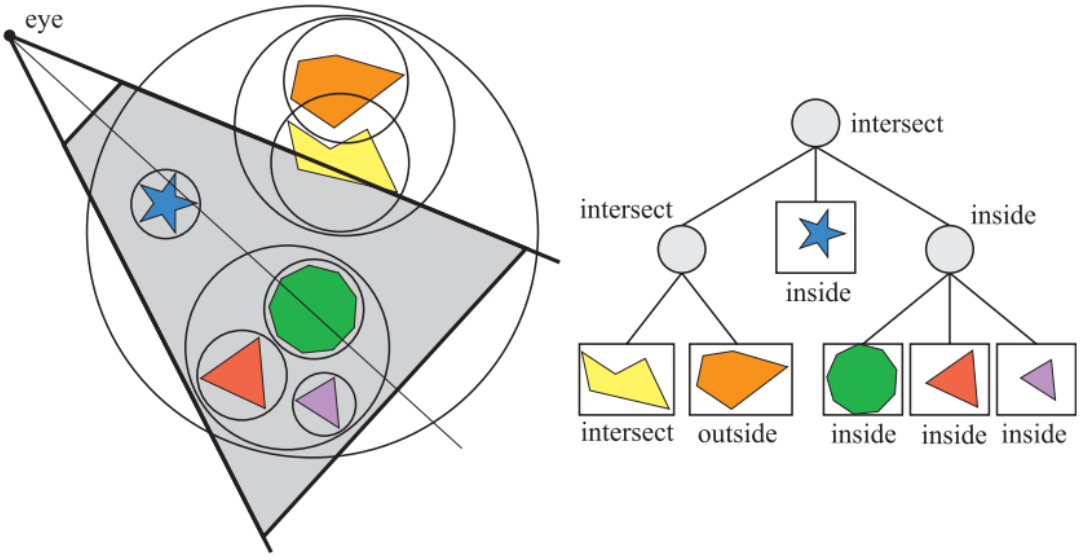


图 19.14：左侧展示了一组几何图形及其包围体（球体）。这个场景是从眼睛的位置出发，使用视锥体剔除来进行渲染的。该场景对应的 BVH 显示在右侧。根节点的 BV 与视锥体相交，继续遍历其子节点，并对相应的 BV 进行相交测试。根节点左子树的 BV 与视锥体相交，并且该子树的其中一棵子树也与视锥体相交（因此该子树会被渲染），而另一棵子树的 BV 则位于视锥体外部，因此并不会发送到管线中。根节点中间子树的 BV 完全位于视锥体内部，因此会立即被渲染。根节点右子树的 BV 也完全位于视锥体内部，因此整棵子树无需进一步进行相交测试，即可进行渲染。

对于“与视锥体相交”的情况，一个有用的优化方法是，追踪该 BV 完全位于哪个视锥体平面内[148]。这个信息通常会被存储为位掩码，然后可以与相交器（intersector）一起进行传递，用于对该 BV 的子节点进行相交测试。这种技术有时候会被称为平面遮挡（plane masking），因为只有那些与 BV 相交的平面，才需要对子节点继续进行相交测试。根节点的 BV 最初需要针对视锥体的 6 个平面进行相交测试，但是随着测试的进行，在每个子节点上进行的平面测试和 BV 测试的次数将会逐渐减少。Assarsson 和 Moller [83]注意到也可以使用时间一致性（temporal coherence）来加速这个过程。该视锥体平面可与 BV 一起进行存储，并在下一帧中作为第一个进行相交测试的平面。Wilidal 指出[1883, 1884]，如果视锥体剔除是在 CPU 上逐物体完成的，那么在 GPU 上进行细粒度剔除（finer-grained culling）时，它只要对左、右、底、顶四个平面执行视锥体剔除就足够了。此外，为了进一步提高性能，可以使用被称为顶点贴图（apex point map）的结构，来提供更加紧密的包围体表示，这将在章节 22.13.4 中进行更加详细地描述。有时候会在远处使用一些雾效，来避免物体在远裁剪平面处突然消失所带来的感官影响。

对于大型场景或者特定的相机视图，只有一小部分场景是可见的，因此也只有这一小部分场景内容需要被发送到渲染管线中。在这种情况下，剔除速度和渲染速度会被大

幅提高。视锥体剔除技术利用了场景中的空间一致性，因为彼此靠近的物体可以被包围在同一个 BV 中，并且附近的 BV 可以进行分层聚类。

需要注意的是，有些游戏引擎中并不会使用分层 BV，而是仅仅使用一个线性 BV 列表，即一个 BV 对应场景中的一个物体[283]。这种做法的主要动机是，在使用 SIMD 和多线程来实现算法时会更加简单，从而提供更好的性能表现。然而，对于某些应用程序而言（例如 CAD），场景中大部分或者全部几何形状都位于视锥体内部，在这种情况下应当避免使用这种类型的算法。分层视锥体剔除技术仍然可以使用，因为如果某个节点完全位于视锥体内部，那么其几何形状可以被立即绘制。

19.5 入口剔除

对于建筑模型，有一组被称为入口剔除（portal culling）的算法。其中第一个入口剔除是由 Airey 等人[17, 18]提出的。后来，Teller 和 Sequin [1755, 1756]以及 Teller 和 Hanrahan [1757]构建了更加高效、更加复杂的入口剔除算法。所有入口剔除算法的基本原理是：在室内场景中，墙壁通常都会充当大型的遮挡物。因此，入口剔除也是一种特殊的遮挡剔除，我们将在下一节中讨论遮挡剔除的话题。这个遮挡算法会在每个入口处（例如门或者窗）使用视锥体剔除机制。在穿过入口的时候，会将视锥体缩小，从而紧密地包围入口。因此，入口剔除算法也可以看作是视锥体剔除的一种扩展，位于视锥体之外的入口将会被丢弃。

入口剔除算法会以某种方式对场景进行预处理。场景会被划分为一个个单元格，这些单元通常对应于建筑物中的房间和走廊，连接相邻房间的门窗会被称为入口（portal）。单元格中的每个物体和对应单元格的墙壁，都会存储在与一个与单元格相关联的数据结构中。我们还会将信息存储相邻的单元格中，并在一个邻接图中存储这些连接单元格的入口信息。Teller 给出了计算这个邻接图的算法[1756]。虽然这种技术早在 1992 年被提出时就能够运行了，但是对于现代的复杂场景而言，想要让这个过程自动化进行是极其困难的。因此，单元格的定义、以及邻接图的创建工作，目前都是手动完成的。

Luebke 和 Georges [1090]使用了一种简单的方法，只需要进行少量的预处理。这个方法唯一需要的信息，就是与每个单元格相关联的数据结构，如上所述。其关键思想是：每个入口都定义了进入这个房间和离开这个房间的视图。想象一下，你正透过一扇门，看到一间有三扇窗户的房间。这个门定义了一个视锥体，我们可以使用这个视锥体来剔除房间内的不可见物体，并只对那些可以看到的物体进行渲染。透过这个门，我们看不到其中的两个窗户，因此可以忽略通过这些窗户所能看到的单元格。其中的第三扇窗户我们是可以看到的，但是它被门框部分遮挡了。只有通过门和这个窗

户都能看见的单元格，其中的物体才需要发送到管线中。单元格的渲染过程依赖于以这种递归方式来追踪可见性。

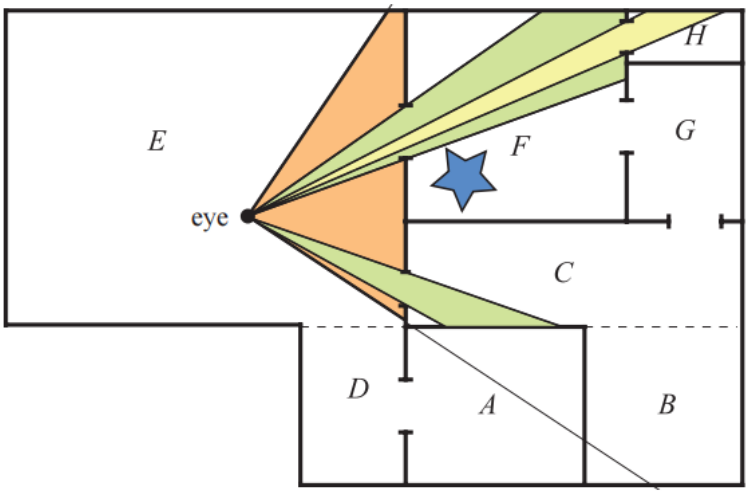


图 19.15：入口剔除：图中展示了从 A 到 H 的单元格，这些连接单元格的开口就是入口。我们只会渲染那些通过入口能够看到的几何图形。例如：细胞 F 中的蓝色五角星我们是看不见的，因此会被剔除。

图 19.15 中展示了入口剔除算法的一个例子。观察者（或者眼睛）位于单元格 E 中，因此 E 中的内容会被渲染。相邻的单元格是 C 、 D 和 F 。原始视锥体看不到通往单元格 D 的入口，因此单元格 D 会在进一步的处理中被忽略。单元格 F 是可见的，因此视锥体会被相应缩小，使其与到单元格 F 的入口相连接，单元格 F 中的内容会根据这个缩小的视锥体来进行渲染。然后，会对单元格 F 的邻近单元格进行检查，从这个缩小的视锥体中看不到单元格 G ，因此它会被忽略，而单元格 H 是可见的。同样地，视锥体会随着单元格 H 的入口而发生相应的缩小，并根据对应的视锥体来渲染单元格 H 中的内容。而单元格 H 的邻居此时都以及被访问过了，因此遍历到这里就结束了。此时，这个递归过程会回到单元格 C 的入口，视锥体相应地会被缩小，从而适应单元格 C 的入口，然后对单元格 C 中的物体进行视锥体剔除，然后再进行渲染。此时我们已经遍历了所有的可见入口，因此渲染过程也就完成了。

每个物体在被渲染的时候都可以进行标记，从而避免多次渲染同一个物体。例如：如果有两个窗户都可以通向同一个房间，那么这个房间中的内容将会分别针对每个视锥体进行剔除。如果没有使用标记的话，会导致某些物体被渲染两次。这不仅效率低下，而且可能会导致渲染错误，例如当物体是透明的时候。为了避免在每一帧中都必须清除这个标签列表，可以让每个物体在被访问的时候都标记为当前的帧号，只有存储了当前帧号的物体才是被访问过的。

一个很值得实现的优化是，可以使用模板缓冲区（stencil buffer）来进行更加精确的剔除。在实践中，这些入口会被 AABB 高估；而真正的入口很可能会更小，可以使用模板缓冲区来屏蔽真实入口之外的渲染。类似地，可以为 GPU 设置一个门户周围的裁剪矩形（scissor rectangle）来提高性能[13]。使用 stencil 和 scissor 功能还可以避免执行上述的标记过程，因为透明对象可能会被渲染两次，但是它们只会对每个入口中的可见像素产生一次影响。

图 19.16：入口剔除。左边是 Brooks House 的俯视图。右边是主卧室的视野。入口的裁剪框使用白色进行标记，镜子使用红色进行标记。

图 19.16 展示了使用入口剔除的另一个视图。这种形式的门户剔除也可以用于为平面反射裁剪内容（章节 11.6.2）。左侧图像展示了建筑物的俯视图，其中白色线条代表了视锥体的范围。而图中的红色线条则是通过镜子反射所形成的视锥体范围。右侧图像展示了实际的视野内容，其中白色矩形代表了入口，红色矩形代表了镜子。只有位于视锥体内部中的物体才需要进行渲染。还可以使用一些其他的变换来创建其他的效果，例如简单的折射。

19.6 细节剔除和小三角形剔除

细节剔除（detail culling）是一种为了渲染速度而牺牲质量的技术。细节剔除的基本原理是：当观察者处于运动状态时，场景中的微小细节对于图像渲染的贡献很小或者根本没有贡献。当观察者停止移动的时候，通常会禁用细节剔除。考虑一个具有包围体的物体，并将该物体的 BV 投影到投影平面上。然后以像素为单位来评估投影的面积，如果其像素数低于用户定义的阈值，则在进一步的处理中会忽略这个物体。因此，细节剔除有时也会被称为屏幕尺寸剔除（screen-size culling）。细节剔除也可以在场景图上分层进行。这种类型的技术经常用于游戏引擎中[283]。

由于每个像素的中心都有一个样本，而那些微小的三角形很有可能会落在两个相邻样本之间。除此之外，小三角形的光栅化效率很低。一些图形硬件实际上会剔除那些落在样本之间的三角形，但是当使用 GPU 上的代码来进行剔除的时候（章节 19.8），添加一些代码来剔除小三角形可能会有所帮助。Wihlidal 提出了一种简单的方法 [1883, 1884]，该方法首先会计算三角形的 AABB。如果下面的条件成立，则可以在着色器中对三角形进行裁剔除：

$$\text{any}(\text{round}(\text{min}) == \text{round}(\text{max})) \tag{19.4}$$

其中的 min 和 max 代表了三角形周围的二维 AABB。如果任意一个向量分量满足上述这个条件，那么函数 any 就会返回真。回顾一下，像素中心位于 $(x + 0.5, y + 0.5)$，这意味着如果该 AABB 的 x 坐标或者 y 坐标取整到相同的坐标时（或者同时满足这两个条件），则方程 19.4 为真。图 19.17 展示了一些示例。

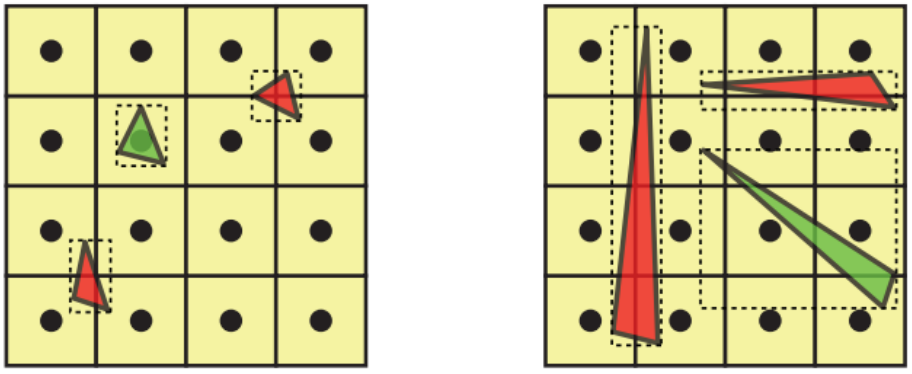


图 19.17：使用 $\text{any}(\text{round}(\text{min}) == \text{round}(\text{max}))$ 来剔除小三角形。图中的红色三角形会被剔除，而绿色三角形则会被保留。左：绿色三角形与样本重叠，因此不能被剔除。红色三角形的 AABB 坐标在取整之后，会得到相同的亚像素角点（pixel corner）。右：红色三角形可以被剔除，因为 AABB 坐标的其中一个分量会被舍入为相同的整数。虽然绿色三角形并没有与任何样本相重叠，但是无法被这个检验剔除。

19.7 遮挡剔除

正如我们所看到的，可以通过 z-buffer 来解决可见性问题。但是即使它能够正确地解决可见性问题，但是 z-buffer 也是相对简单和粗暴的，因此它并不总是最高效的解决方案。例如：假设观察者正沿着一条直线进行观察，这条直线上放置了 10 个球体，如图 19.18 所示。

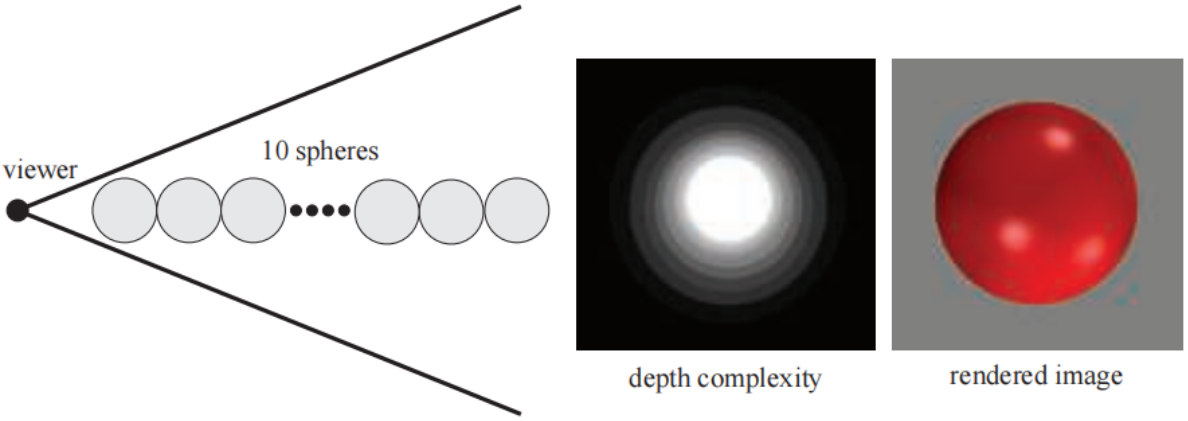


图 19.18：图中说明了遮挡剔除为什么是有用的。十个球体排列成一条直线，观察者沿着这条直线进行透视观察（左）。中间的深度复杂度图像，展示了这些像素分别被多次写入，即使最终图像只会显示出一个球体（右）。

从这个视角进行渲染的图像，只会显示出一个球体，但是所有的 10 个球体都将被光栅化，并与 z-buffer 进行比较，然后有一定概率会写入颜色缓冲区和 z-buffer 中。图 19.18 的中间部分展示了这个场景在给定视角下的深度复杂度，深度复杂度指的是一个像素所覆盖的表面数量。假设我们启用了背面剔除，在图中 10 个球体的情况下，由于所有的 10 个球体都位于最中间的像素上，因此其深度复杂度为 10。如果这个场景按照从后往前的顺序进行渲染，那么最中间的像素将会被着色 10 次，也就是说，其中有 9 次像素着色器执行是不必要的。即使场景是从前往后进行渲染的，那么这 10 个球体的三角形仍然会被光栅化，计算各自的深度，并与 z-buffer 中的深度进行比较，但是最终只生成了一个球体的画面。这个无聊的场景在现实中不太可能出现，但是它（从给定的视角）描述了一个物体密集的案例模型。这种类型的模型可以在真实场景中找到，例如：雨林、汽车引擎、城市和摩天大楼内部等。图 19.19 展示了这样的一个例子。

图 19.19：《我的世界》中的一个场景，叫做 Neu Rungholt，观察者位于图中的右下角位置，在那里我们可以看到遮挡剔除的可视化。浅色的几何体代表会被剔除，而深色的几何体则代表会被渲染。左下角展示了最终渲染出的图像。

根据这个例子，避免这种低效率的算法可能会提高性能表现。这种类型的方法被称为遮挡剔除算法（occlusion culling algorithm），因为它们试图剔除那些被遮挡的物体，即被场景中被其他物体所隐藏的物体。最优的遮挡剔除算法将只会选择那些可见的物体。从某种意义上说，z-buffer 只会选择并渲染那些可见的物体，而其他大部分位于视锥体内部的物体同样也会被发送到管线中进行计算，但是由于物体之间的相互遮挡，它们并不会出现在最终的图像中。高效遮挡剔除算法背后的思想是：在早期执行一些简单的测试，来剔除那些被隐藏的物体集合。从某种意义上说，背面剔除是遮挡剔除的一种简单形式。如果我们事先知道一个物体是封闭的，并且是不透明的，那么该物体的背面就会被正面所遮挡，因此并不需要进行渲染。

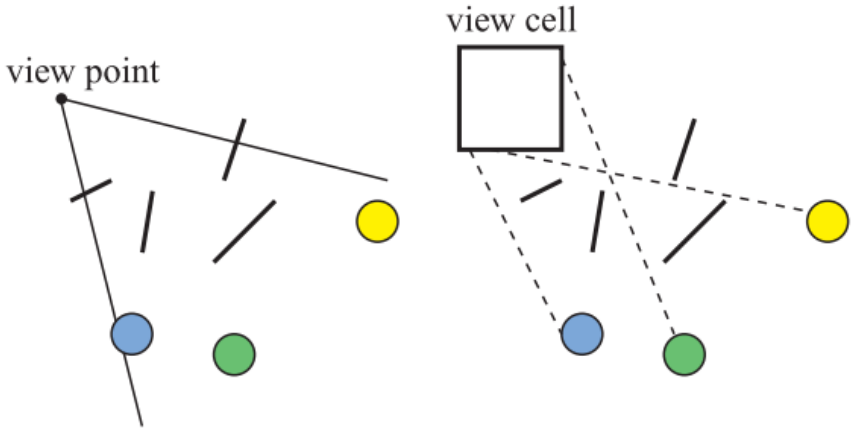


图 19.20：左侧展示了基于点的可见性，右侧展示了基于单元格的可见性，其中的单元格是一个 box。在左图中我们可以看到，从当前视点进行观察，这些圆圈都会被遮挡。然而在右侧，这些圆圈都是可见的，因为光线可以从单元格内的某个地方打到这些圆圈，并且不会与任何遮挡物相交。

遮挡剔除算法主要有两种形式，即基于点的（point-based）和基于单元格的（cell-based），如图 19.20 所示。基于点的可见性就是通常在渲染中所使用的，也就是说，从单一的观察位置所能够看到的东西。另一种是基于单元格的方法，其可见性是针对单元格进行定义的，这个单元格是包含一组观察位置的空间区域，通常是一个 box 或者球体。在基于单元格的可见性中，一个物体必须要对单元格内的所有点都不可见，我们才能说这个物体是不可见的。基于单元格可见性的优点在于，一旦计算了某个单元格的可见性，那么只要观察者位于这个单元格内部，那么通常可以在几帧中重复使用这个可见性信息。然而，这种基于单元格的可见性计算起来通常要比基于点的可见性更加耗时，因此，它通常都是在预处理阶段完成的。基于点的可见性和基于单元格的可见性，在本质上类似于点光源和面光源，可以认为是在光源位置上来观察这个场景。而对于那些不可见的物体来说，这相当于这个物体处于光源的本影区域，即完全处于阴影中。

我们还可以将遮挡剔除算法按照不同的空间进行分类，分别是：图像空间（image space）、物体空间（object space）或者光线空间（ray space）中进行的算法。图像空间算法在经过一些投影操作后，会进行二维的可见性测试。物体空间算法则会使用原始场景中的三维物体。光线空间算法[150, 151, 923]则会在对偶空间（dual space）中进行测试。每个感兴趣的点（通常是二维的），都会被转换为这个对偶空间中的一条射线。对于实时图形而言，这三种算法中应用最广泛的是图像空间中的遮挡剔除算法。

```
      OcclusionCullingAlgorithm(G)
1:    O_R =empty
2:    P =empty
3:    for each object g ∈ G
4:       if(isOccluded(g,O_R))
5:          Skip(g)
6:       else
7:          Render(g)
8:          Add(g, P)
9:          if(LargeEnough(P))
10:            Update(O_R, P)
11:            P =empty
12:         end
13:      end
14:   end
```

图 19.21：通用遮挡剔除算法的伪代码。其中 G 包含了场景中的所有物体，O_R 是遮挡表示。P 是一组潜在的遮挡物，当 P 中包含足够多的物体时，它会被合并 O_R 中。[1965]

图 19.21 展示了一种遮挡剔除算法的伪代码，其中的函数 isOccluded 通常会被称为可见性测试（visibility test），它用于检查物体是否被遮挡。其中 G 是要进行渲染的物体集合，O_R 是遮挡表示，P 是一个潜在的遮挡物集合，它可以与 O_R 进行合并。取决于所使用的特定算法，O_R 代表的是某种遮挡信息。O_R 在算法开始的时候会被设置为空。之后，会对所有的物体（这些物体已经通过了视锥体剔除测试）进行处理。

考虑一个特定的物体。首先，我们根据遮挡表示 O_R，来测试这个物体是否会被遮挡。如果它被遮挡，那么就不需要进行进一步的处理了，因为我们此时已经知道了这个物体不会对最终的图像有任何贡献。如果我们无法确定这个物体是否会被遮挡，那么就必须要渲染这个物体，因为它可能会对最终的图像有贡献（在对其进行渲染的那一刻）。然后将该物体添加到集合 P 中，如果 P 中的物体数量足够多，那么我们可以将这些物体的遮挡能力（occluding power）合并为 O_R，即 P 中的每个物体都可以作为遮挡物（occluder）。

请注意，对于大多数遮挡剔除算法而言，其性能取决于渲染物体的前后顺序，举个例子：假设现在有一辆汽车，汽车内部有一台发动机。如果汽车的引擎盖先进行绘制，那么内部的发动机（很可能）会被剔除。另一方面，如果我们先绘制发动机，那么没有任何物体会被剔除。使用一个粗略地、从前往后排序的渲染，可以获得相当大的性

能提升。此外，值得注意的是，一些较小的物体也可能会是遮挡效果很强的遮挡物，因为与遮挡物之间的距离决定了它能够遮挡多少内容（译者注：一叶障目）。例如：如果一个火柴盒到观察者的距离足够近，那么这个火柴盒也可以遮挡金门大桥。

19.7.1 遮挡查询

GPU 可以通过使用特殊的渲染模式来支持遮挡剔除，用户可以对 GPU 进行查询，从而确定一组三角形与 z-buffer 中的当前内容相比，是否是可见的。三角形通常会形成一个更加复杂物体的包围体（例如：一个 box 或者 k-DOP），如果这些三角形都不可见，那么就可以剔除这个物体。GPU 会将查询的三角形光栅化，并将其深度与 z-buffer 进行比较，也就是说，它是在图像空间中进行操作的。这些生成的三角形并不会修改像素颜色信息和深度信息，将其可见的像素数量记为 n，如果 n 为 0，则代表所有的三角形都会被遮挡或者裁剪。

但是，$n = 0$ 也并不足以确定这个包围体是否为不可见的。更准确地说，还需要考虑相机视锥体的近裁剪平面，这个包围体不能包含近裁剪平面以内的物体。如果这个条件也满足，那么说明整个包围体被完全遮挡了，它所包含的物体可以被安全地丢弃。如果 $n > 0$，则有一小部分像素没有通过测试。如果这个 n 小于用户设定的阈值像素数量，那么这个物体可能会因为对最终图像的贡献不太大，从而被丢弃 [1894]。通过这种方式，可以使用一些可能的质量损失，来换取速度的提升。另一个用法是使用这个可见像素计数 n，来帮助确定物体所使用的 LOD 层级（章节 19.9）。如果这个 n 很小，则说明只有物体的一小部分是（可能）可见的，因此可以使用一个不太详细的 LOD。

当发现某个包围体被遮挡时，我们通过避免向渲染管线发送这个潜在的复杂的物体，从而提高性能表现。但是如果测试失败的话（即这个包围体没有被遮挡），我们实际上会损失一些性能，因为我们花费了额外的时间来对这个包围体进行测试，但是却没有获得任何好处。

遮挡测试还有一些变体方法。出于剔除的目的，我们实际上并不需要可见片元的确切数量，使用一个布尔值来表示是否至少存在一个片元通过深度测试就足够了。OpenGL 3.3 和 DirectX 11 以及后续版本，都支持这种类型的遮挡查询，在 OpenGL 中为枚举变量 $\mathrm{ANY\_SAMPLES\_PASSED}$ [1598]。这些测试的速度还可以更快，因为它们可以在发现一个片元可见时就立即终止遮挡查询。OpenGL 4.3 及更高版本，还可以使用这种查询的更快变体，它被称为 $\mathrm{\_ANY\_SAMPLES\_PASSED\_CONSERVATIVE}$。这个实现可以选择提供不那么精确的测试，并且它是保守的（不会剔除那些未被遮挡的物体），例如：硬件供

应商可以通过仅对粗深度缓冲区（章节 23.7）来执行深度测试，而不是进行逐像素的深度测试来实现这一点。

这个查询通常会有一段相对较长时间的延迟。一般来说，这段延迟时间内可以渲染成百上千个三角形，有关延迟的更多信息详见章节 23.3。因此，当包围盒中包含大量的物体，并且发生了相对大量遮挡现象的时候，那么这种基于 GPU 的遮挡剔除方法是值得的。GPU 使用了这样的一个遮挡查询模型，其中 CPU 可以向 GPU 发送任意数量的查询请求，然后 CPU 会定期检查是否有任何结果可以使用，也就是说，这个查询模型是异步的。GPU 会执行每个遮挡查询，并将查询结果放入一个队列中。CPU 端的队列检查非常快，同时 CPU 还可以继续发送查询请求或者渲染物体，而不会发生停滞。DirectX 和 OpenGL 都支持断言/条件（predicated/conditional）的遮挡查询，其中遮挡查询与对应 draw call 的 ID 会被同时提交。只有当遮挡查询的几何物体可见时，GPU 才会自动处理相应的 draw call，这使得查询模型更加实用。

一般来说，应当对最有可能被遮挡的物体来执行遮挡查询。Kovaleik 和 Sochor [932]在应用程序的运行过程中，会对每个物体在若干帧内的查询结果的运行信息进行收集。一个物体被遮挡的帧数量，会对它在未来被检测遮挡的频率产生影响。也就是说，可见的物体很可能会一直保持可见，因此可以较少地进行遮挡测试。如果可能的话，会在每一帧中都对隐藏物体进行遮挡测试，因为这些物体是最有可能从遮挡查询中受益的。Mattausch 等人[1136]针对没有断言/条件渲染的遮挡查询（OC）提出了几种优化方法。他们使用了批量处理的 OC，将几个 OC 组合成一整个 OC；使用若干个包围盒而不是单个较大的包围盒；并使用时域抖动采样，来调度之前可见的物体。

这里所讨论的方案展示了遮挡剔除方法的潜力以及一些存在的问题。究竟什么时候使用遮挡查询，或者使用什么样的遮挡方案，这些答案通常都是并不清楚的。如果场景内的物体都是可见的，那么使用遮挡算法只会花费额外的时间，并且永远都不会提升性能表现。这里存在的一个挑战是，如何快速确定遮挡算法并没有起到作用，减少那些徒劳的遮挡测试，从而节省时间。另一个问题是，究竟决定使用哪组物体集合来作为遮挡物。视锥体内的第一个物体肯定是可见的，因此在这些物体上进行遮挡查询完全是浪费。在实现大多数遮挡剔除算法的时候，决定以什么顺序进行渲染，以及什么时候进行遮挡测试是一个难题。

19.7.2 层次 Z 缓冲

层次 z 缓冲（hierarchical z-buffering，HZB）[591, 593]对于遮挡剔除的研究有重要影响。虽然原始形式的、CPU 端的 HZB 很少会被使用，但是该算法是 GPU 硬件

z–culling 方法（章节 23.7）的基础，也是使用 GPU 或者 CPU 运行的、软件自定义的遮挡剔除的基础。我们首先会介绍该技术的基本算法，然后再介绍该技术是如何在各种渲染引擎中应用的。

该算法会将场景模型维护在一棵八叉树中，并将一帧画面的 z–buffer 来作为图像金字塔，我们称之为 z 金字塔（z–pyramid）。因此，该算法也是在图像空间中运行的。八叉树实现了对场景遮挡区域的分层剔除，而 z–金字塔则实现了图元的层次 z 缓冲。因此，z 金字塔是该算法的遮挡表示方法，图 19.22 展示了这种数据结构的例子。

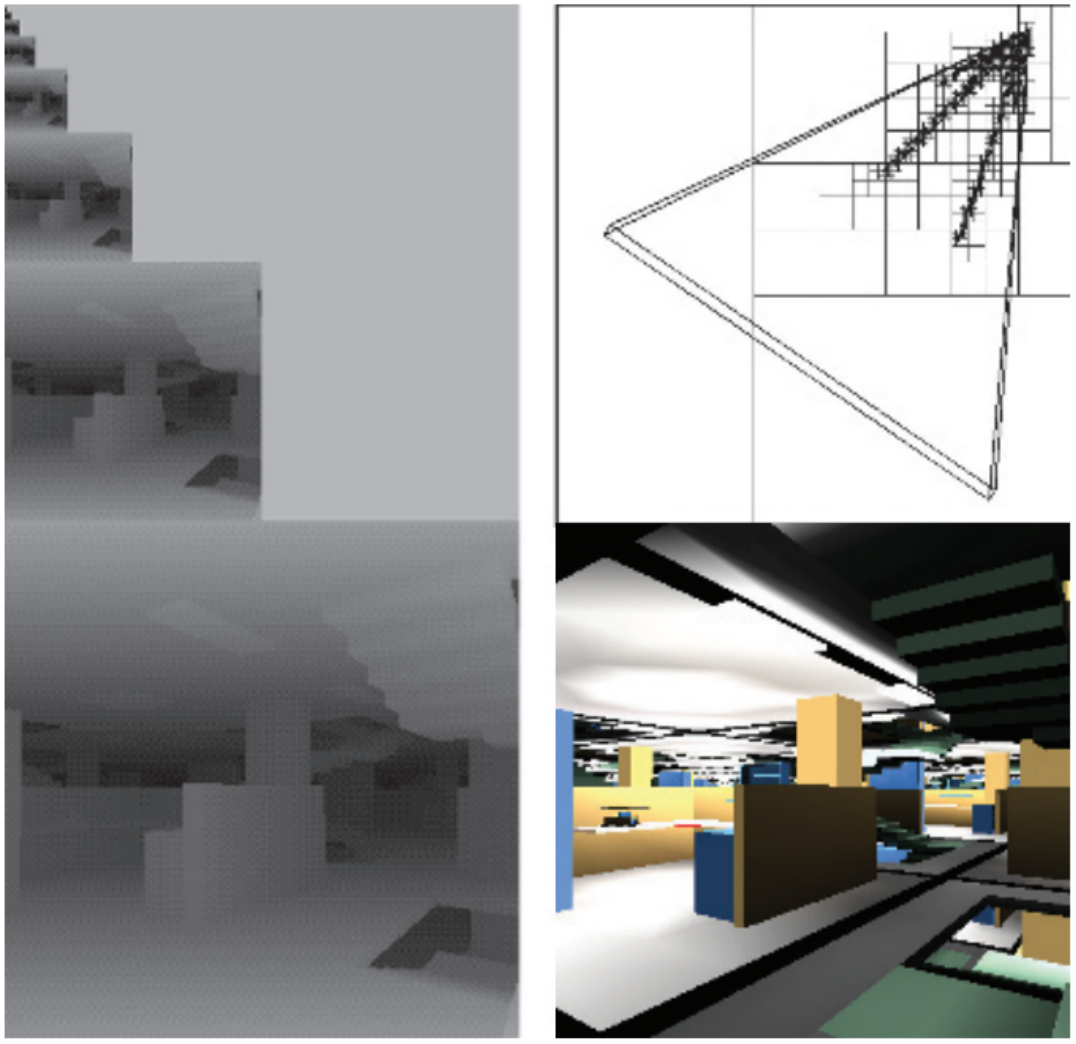


图 19.22：使用 HZB 算法进行遮挡剔除的例子[591, 593]，右下角展示了一个具有较高深度复杂度的场景，左侧展示了相应的 z–金字塔，右上角展示了八叉树的细分结构。通过从前向后遍历八叉树，并在遇到被遮挡的八叉树节点时将它们剔除，这个算法能够实现只访问那些可见的八叉树节点及其子节点（右上角所示的节点），并且只渲染那些位于可见框中的三角形。在这个例子中，通过剔除那些被遮挡的八叉树节点，可以将深度复杂度从 84 降低到 2.5。

z 金字塔的最精细级别（最高分辨率），实际上就是一个标准的 z-buffer。而在其他的所有级别上，每个 z 值都是相邻更精细级别中，对应 2×2 窗口中的最远 z 值。也就是说，每个 z 值都代表了屏幕正方形区域内的最远 z 值。每当 z-buffer 中的一个 z 值被覆盖时，它就会通过 z 金字塔的较粗糙层级向上进行传播。这个过程是递归完成的，直到到达图像金字塔的最顶端，最顶端只剩下一个单独的一个 z 值。金字塔的结构如图 19.23 所示。

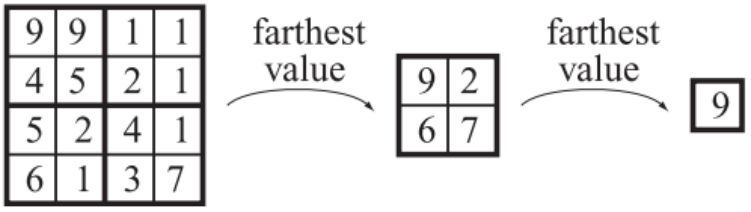


图 19.23：左侧展示了 z-buffer 的一个 4×4 区域，其中每个单元格内的数值就是实际的 z 值。这个区域会被下采样到一个 2×2 区域，其中每个值都是左侧对应 2×2 区域中最远的那个（最大的）。最后，再计算剩余 4 个 z 值中的最大值。这三个贴图便构成了一个被称为层次 z 缓冲的图像金字塔。

八叉树节点的分层剔除可以按照下列步骤进行。首先以粗略的从前到后的顺序来遍历八叉树中的节点，使用一个扩展的遮挡查询（章节 19.7.1）来对八叉树的包围盒和 z 金字塔进行深度测试。我们将包围盒投影到屏幕上，并从能够包围这个屏幕投影的、最粗糙的 z 金字塔单元格开始进行测试。然后将包围盒在单元格内的最近深度（ z_{near} ）与 z 金字塔中的值进行比较，如果发现 z_{near} 要更远，那么我们就知道这个包围盒被遮挡了。这个测试会一直沿着 z 金字塔的层级不断递归地进行下去，直到发现这个包围盒被遮挡了，或者是到达 z 金字塔的最底层（最精细级别，即 z-buffer），那么此时包围盒便是可见的。对于可见的八叉树包围盒，会在八叉树中继续向下递归测试，最后可能的可见几何图形，会被渲染到这个层次 z 缓冲区中。这样做是为了在后续的测试中，能够利用先前已渲染物体的遮挡能力。

完整的 HZB 算法如今没有被使用，但是对其进行了简化，并且能够在 GPU 上使用自定义的剔除、或者使用 CPU 上的软件光栅化，来很好地与计算着色器 pass 一起运行。一般来说，大多数基于 HZB 的遮挡剔除算法的流程如下：

1. 使用一些遮挡物表示，来生成一个完整的层次 z-金字塔。
2. 想要测试某个物体是否被遮挡，需要将其包围体投影到屏幕空间中，并估计 z 金字塔中对应的 mip 层级。

3. 针对选定的 mip 层级进行遮挡测试。如果遮挡结果并不明确，可以选择使用更加精细的 mip 层级来继续进行测试。

大多数实现都不会使用八叉树或者任何 BVH，也不会在渲染物体后取更新 z 金字塔，因为这些操作可能会过于昂贵而无法执行。

步骤 1 可以使用一些“最佳”遮挡物来完成[1637]，这些“最佳”遮挡物可以是最近的 n 个物体集合[625]，可以使用艺术家生成的简化遮挡物图元，或者使用上一帧中可见物体集合的统计数据。再或者，还可以使用上一帧的 z-buffer [856]，但是这样做可能会过于激进，因为部分物体有时可能会因为不正确的剔除而突然出现，尤其是在相机或者物体快速移动的情况下。Haar 和 Aaltonen [625]都将最佳遮挡物进行了渲染，并将结果与前一帧深度 $1/16$ 分辨率的重投影结果结合起来，然后再使用 GPU 来构建 z 金字塔，如图 19.23 所示。有些人则使用 AMD GCN 架构中的 HTILE（章节 23.10.3）来加速 z 金字塔的构建[625]。

在步骤 2 中，我们需要将物体的包围体投影到屏幕空间中。常见的 BV 选择包括球体、轴对齐包围盒（AABB）和定向包围盒（OBB）。BV 投影后的最长边 l （以像素为单位）会用于计算 z 金字塔中的 mip 层级 λ，计算公式如下[738, 1637, 1883, 1884]:

$$\lambda = \min\left(\left\lceil \log_2(\max(l, 1)) \right\rceil, n - 1\right) \tag{19.5}$$

其中方程 19.5 中的 n 是 z 金字塔中 mip 层级的最大数量。max 操作符是为了避免得到负的 mip 层级，min 操作符是为了避免访问不存在的 mip 层级。方程 19.5 会选择最小的整数 mip 层级，并能够使得投影后的 BV 最多覆盖 2×2 个深度值。这样做的原因是使得成本可以预测，因为它最多只需要对 4 个深度值进行读取和测试。此外，Hill 和 Collin 认为[738]，这个测试可以被视为“概率性”的，因为较大的物体要比较小的物体更加容易被看到，所以在这些情况下，不需要读取更多的深度值。

当到达步骤 3 时，我们此时已经知道投影后的 BV 位于哪一个 mip 层级中，并被一组（最多） 2×2 的深度值所包围。对于一个给定大小的 BV，它可能会完全落在某个 mip 层级的单个深度纹素中。然而，取决于 BV 投影在单元格上的方式，它也可能会覆盖所有的四个纹素。然后我们需要计算 BV 的最小深度，这个深度可以是精确的，也可以是保守的。对于观察空间中的 AABB 而言，这个深度就是包围盒的最小深度；而对于 OBB 而言，可以将所有顶点都投影到观察向量上，并选择其中的最小距离。对于球体 BV，Shopf 等人[1637]使用公式 $\mathbf{c} - r\mathbf{c}/\|\mathbf{c}\|$ 来计算球面上的最近点，其中 $\mathbf{c}$ 是观察空间中的球体中心， r 是球体半径。请注意，如果相机位于 BV 内部，那

么这个 BV 将会覆盖整个屏幕，因此这个物体需要进行渲染。将 BV 的最小深度 z_{min} 与层次 z 缓冲区中的（最多） 2×2 深度进行比较，如果 z_{min} 是其中最大的深度，那么说明这个 BV 会被遮挡。如果该 BV 没有被遮挡，那么可以在这里停止测试，并渲染这个 BV 所对应的物体。

我们还可以在金字塔的下一个更深层级（更高分辨率）上来继续进行测试。我们可以通过使用另一个存储最小深度的 z 金字塔，来看看这种测试是否有必要进行。我们可以根据这个新缓冲区中的相应深度，来对 BV 的最大距离 z_{max} 进行测试。如果 z_{max} 是其中最小的，那么说明这个 BV 肯定是可见的，并且可以立即进行渲染。否则，这个 BV 的 z_{min} 和 z_{max} 会与这两个层级 z 缓冲的对应深度相重叠，在这种情况下，Kaplanyan [856]建议可以在更高分辨率的 mip 层级上继续进行测试。请注意，针对单个深度值在层次 z 缓冲中测试 2×2 大小的纹素，与百分比接近过滤 PCF 的做法非常相似（章节 7.5）。事实上，这个测试可以使用双线性过滤和百分比接近过滤来完成，如果这个测试返回一个正值，那么说明至少有一个纹素是可见的。

Haar 和 Altonen [625]也提出了一种双 pass 方法，它总是可以至少渲染所有的可见物体。首先，在第一个 pass 中，使用前一帧中的 z 金字塔，来对所有物体进行进行遮挡剔除，并渲染所有的“可见”物体。或者，可以直接使用最后一帧中的可见性列表来渲染当前帧的 z 金字塔。虽然这是一个近似方法，但是上一帧中所有被渲染的物体，确实可以作为当前帧“最佳”遮挡物的良好估计，尤其是在具有较高帧间一致性的情况下。在第二个 pass 中，会获取这些渲染物体的深度缓冲，并创建一个新的 z 金字塔。然后，针对第一个 pass 中被剔除的物体，再次进行遮挡测试，如果某些物体这一次没有被剔除，那么就渲染这些物体。即使相机或者物体快速移动时，这种方法也能够生成完全正确的图像。Kubisch 和 Tavenrath [944]也使用了类似的方法。

Doghramachi 和 Bucci [363]将前一帧中的深度缓冲区进行下采样和重投影，并使用它们来对被遮挡物体的定向包围盒进行光栅化。他们强制着色器使用 early-z（章节 23.7），并且对于每个包围盒，可见片元都会将物体标记为在某个缓冲区位置上可见，这是由物体 ID 唯一确定的[944]。由于使用了定向包围盒，并且进行了逐像素的测试（而不是使用方程 19.5 针对 mip 层级所使用自定义测试）因此可以提供更高的剔除率。

Collin [283]使用了 256×144 大小的浮点 z-buffer（并不是分层的），并以较低的复杂度来光栅化艺术家生成的遮挡物。这是在使用 CPU 或者 SPU（在 PLAYSTATION 3 上）、以及高度优化的 SIMD 代码的软件中完成的。为了进行遮挡测试，该方法会计算物体的屏幕空间 AABB，并将其 z_{min} 与这个小尺寸 z-buffer 中的所有相关的深度进行比较，只有在剔除中幸存下来的物体才会被发送到 GPU 中。

这种方法是可行的，但是从保守角度来说它并不正确，因为它所使用的分辨率要低于最终帧缓冲的分辨率。Wihlidal 建议[1883]，低分辨率的 z-buffer 也可用于将 z_{max} 加载到 GPU 的 HiZ 中（章节 23.7），例如：在 AMD GCN 上触发 HTILE 结构。或者，如果 HZB 用于计算着色器 pass 的剔除，那么可以使用软件 z-buffer 来生成 z 金字塔，这样一来，算法可以利用软件中产生的所有信息。

Hasselgren 等人[683]提出了一种不同的方法，其中每个 8×4 的 tile 中，每个像素有一个 bit 以及两个 z_{max} 值[50]，即每个像素的总成本为 3 bit。通过使用这个 z_{max} 值，可以更好地处理深度不连续的情况，因为背景物体可以使用其中的一个 z_{max} 值，而前景物体则可以使用另一个 z_{max} 值。这种表示方法被称为掩码层次深度缓冲区（masked hierarchical depth buffer，MHDB），这是一种保守的表示方法，也可以用于 z_{max} 剔除。在软件三角形光栅化的过程中，每个 tile 只会生成覆盖蒙版和单个的最大深度值，这使得光栅化到 MHDB 中的效率很高。在将三角形光栅化到 MDHB 的过程中，也可以使用 MDHB 来对三角形进行遮挡测试，从而对光栅化器进行优化。每个三角形都会更新 MDHB，这是其他方法很少具备的优点。可以使用两种模式来进行评估。第一种是使用特殊的遮挡网格，并使用软件光栅化器来将这些网格渲染到 MDHB 中。然后遍历被遮挡物上的 AABB 树，并针对 MDHB 进行分层遮挡测试。这样做是十分有效的，尤其是当场景中存在许多小物体的时候。对于第二种方法，整个场景会被存储在一个 AABB 树中，通过使用一个堆结构，来使得场景的遍历大致是按从前向后的顺序完成的。在每一步中，对 MDHB 使用视锥体剔除和遮挡查询，每当渲染物体的时候，MDHB 也会相应地进行更新。图 19.19 中的场景就是使用这个方法进行渲染的，其开源代码针对 AVX2 指令集进行了大量优化[683]。

还有一些专门用于剔除和遮挡剔除的中间件（middleware）。Umbra 就是这样的一个框架，它被广泛地整合到各种游戏引擎中[13, 1789]。

19.8 剔除系统

剔除系统多年来已经发生了相当大的演变，并且将持续发生演变。在本小节中，我们将描述一些总体思想，并提及有关细节的一些文献。一些系统在 GPU 的计算着色器中来高效执行所有剔除操作，而其他的一些系统则将 CPU 上的粗粒度剔除与 GPU 上的细粒度剔除结合起来。

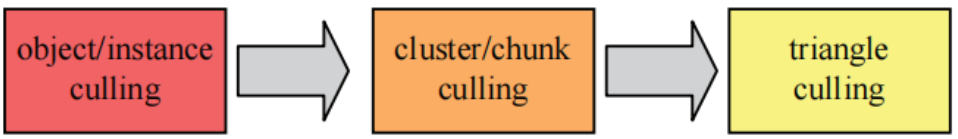


图 19.24：一个在三种不同粒度上工作的剔除系统示例。首先会在物体级别上进行剔除。幸存下来的物体，会在 cluster 级别上进行剔除。最后，进行三角形剔除，这个过程会在图 19.25 中有进一步的描述。

如图 19.24 所示，一个典型的剔除系统可以在许多粒度上进行运行。其中一个物体的簇或者块（cluster/chunk），指的是该物体三角形的一个子集。例如：可以使用包含 64 个顶点的三角形带[625]，或者是包含 256 个三角形的组[1884]。在每个步骤中，可以使用多种剔除技术的组合。El Mansouri [415]在物体上使用了小三角形剔除、细节剔除、视锥体剔除和遮挡剔除。由于 cluster 在几何上要比物体更小，因此对 cluster 使用相同的剔除技术是有意义的，因为它们更有可能会被剔除。例如：可以在 cluster 上使用细节剔除、视锥体剔除、集群背面剔除和遮挡剔除等。

在 cluster 级别上进行剔除之后，还可以执行一个额外的步骤，即在三角形级别上进行剔除。为了让这一步完全在 GPU 上完成，可以使用图 19.25 所展示的方法。三角形剔除技术包括透视除法（除 w ）后的视锥体剔除，即将三角形的范围与 ± 1 进行比较（译者注：裁剪）；背面剔除、退化三角形剔除、小三角形剔除、以及可能的遮挡剔除等。然后将历经所有剔除测试后剩下的三角形压缩到一个最小列表中，这样做的目的是为了在下一步中只对那些幸存下来的三角形进行处理[1884]。这里的一个想法是，让用于剔除的计算着色器在这个步骤中向 GPU 发送一个绘制命令，这是使用间接绘制命令（indirect draw command）完成的。这种调用在 OpenGL 中被称为“多重间接绘制（multi-draw indirect）”，在 DirectX 中则被称为“间接执行（execute indirect）”[433]。三角形的数量会被写入到 GPU 缓冲区中的一个位置，它与压缩列表一起，可以被 GPU 用来渲染三角形列表。

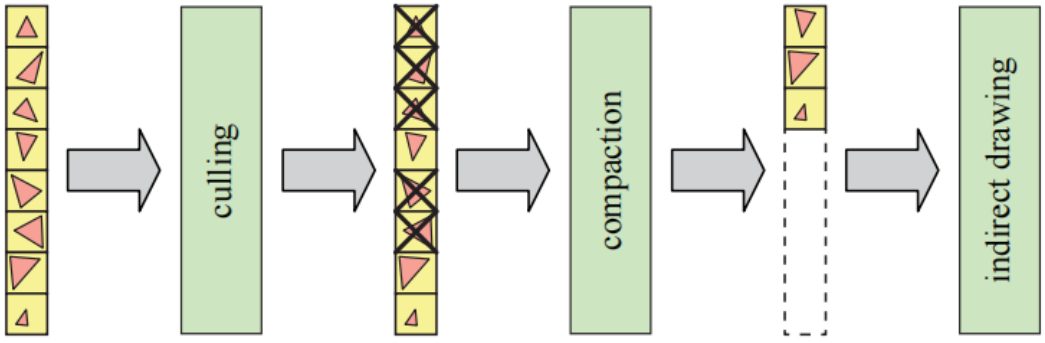


图 19.25：三角形剔除系统，其中首先会对所有单独的三角形应用一组剔除算法。为了能够使用间接绘制（即没有 GPU/CPU 往返），幸存下来的三角形会被压缩成一个更短的列表，这个列表是由 GPU 使用间接绘制命令来进行渲染的。

有许多方法可以将剔除算法与它们的执行地点（在 CPU 或者 GPU 上）结合在一起，每种剔除算法也有着许多不同的风格。现在还没有一种最好的组合方式，但可以肯定的是，这种最好的方法具体取决于目标架构以及要进行渲染的内容。接下来，我们会介绍一些 CPU/GPU 剔除系统领域中的重要工作，这些工作对于该领域产生了重大的影响。Shopf 等人[1637]在 GPU 上对角色进行了 AI 模拟，因此，每个角色的位置只能在 GPU 的显存中使用。这让他们探索了使用计算着色器来管理剔除和 LOD，随后的大多数剔除系统都受到了他们工作的巨大影响。Haar 和 Aaltonen [625]描述了他们为《刺客信条：大革命》所开发的系统。Wihlidal [1883, 1884]解释了寒霜引擎中所使用的剔除系统。Engel [433]提出了一个剔除系统，该系统使用一个可见性缓冲来帮助改进管线（章节 20.5）。Kubisch 和 Tavenrath [944]描述了如何渲染大量模型的方法，这些模型都具有大量的部件，并使用了不同的剔除方法和 API 调用来进行优化。一个值得注意的方法是，他们使用了遮挡剔除框（occlusion-cull box），并使用几何着色器来创建包围盒的可见边，然后使用 early-z 来快速剔除被遮挡的几何体。

19.9 LOD

细节层次（level of detail, LOD）的基本思想是：如果一个物体对最终渲染图像的贡献越来越小，那就使用一个更加简化版本的物体。例如：假设现在有一辆可能包含 100 万个三角形的精细汽车模型，当观察者靠近汽车进行观察的时候，可以使用这种精细表示；而当物体距离较远的时候，比如只覆盖了 200 个像素，我们并不需要这全部的 100 万个三角形。相反，我们可以使用一个简化模型，比如只包含 1000 个三角形的汽车模型。由于距离的关系，这个简化版本看起来与详细版本大致相同，如图 19.26 所示。通过这种方式，可以获得显著的性能提升。为了减少应用 LOD 技术所涉及的总工作量，最好是在应用剔除技术之后再应用 LOD。例如：可以只对视锥体内的物体计算对应的 LOD。

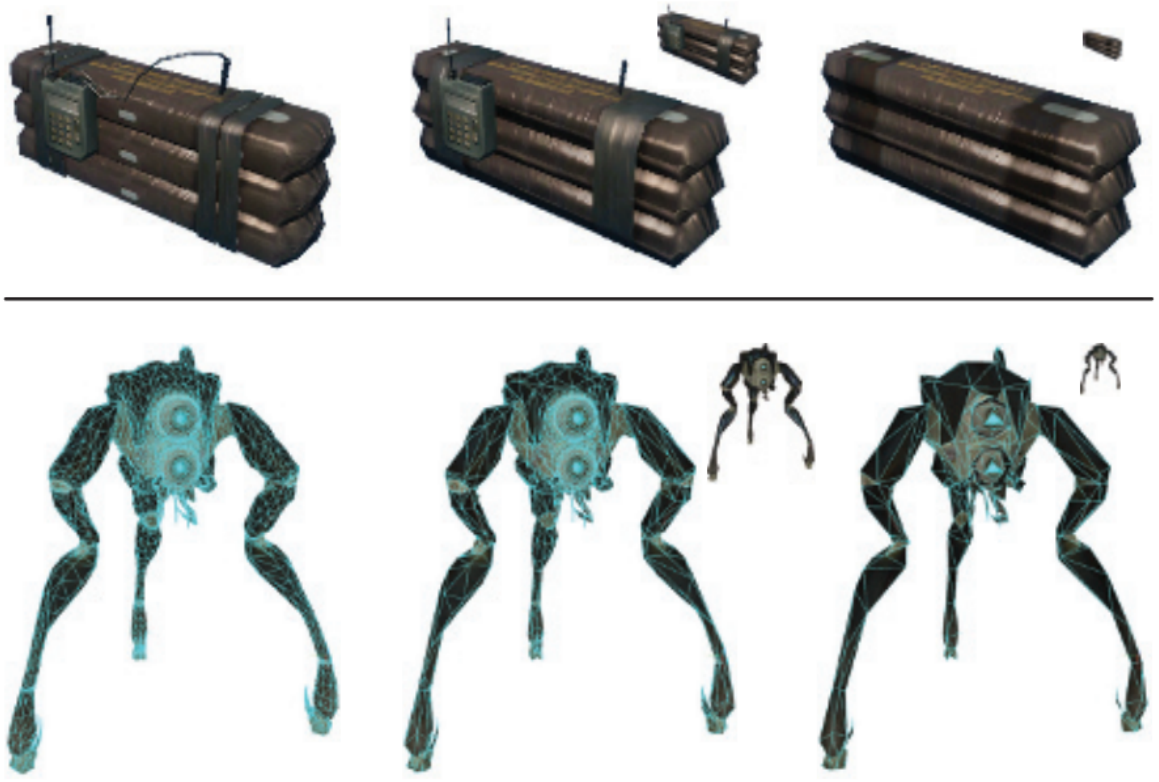


图 19.26：图中展示了 C4 炸药模型（上）和猎人模型（下），以及三个不同的 LOD。在较低的 LOD 上，一些元素会被简化或者完全删除。图中右上角的嵌入图像，展示了可以使用这个简化模型的相对尺寸。

LOD 技术还可以使得应用程序具有更好的可伸缩性，能够在一系列具有不同性能的设备上，以所需的帧率进行运行。在性能较低的系统上，可以使用较少细节的 LOD 来提高性能表现。需要注意的是，虽然 LOD 技术首先可以帮助减少顶点处理的开销，但是它们也同样减少了像素着色的成本。出现这种情况的原因在于，这个模型所有三角形的边长之和会变小，这意味着四边形过度渲染（quad overshading）的现象减少了（章节 18.2 和章节 23.1）。

雾和第 14 章中所描述的其他参与介质，可以与 LOD 一起进行使用。这允许我们可以完全跳过渲染一个物体，因为它可能会进入到完全不透明的雾中。此外，这种雾化机制（fogging mechanism）还可以用于实现限时渲染（time-critical，章节 19.9.3）。通过移动远裁剪平面，使其更加接近观察者，可以在早期剔除更多物体，从而提高帧率。此外，通常可以在雾中使用较低的 LOD。

还有一些物体（例如球体、Bezier 表面和细分表面），这些物体的几何描述本身就具备了一部分的 LOD 效果。其底层的几何图形表示是弯曲的，可以使用一个单独的 LOD 控制选项，来决定如何将其细分成可以显示的三角形。详见章节 17.6.2，其中包含了一些算法，它们可以控制参数化曲面和细分曲面的曲面细分质量。

一般来说，LOD 算法由三个主要部分组成，分别是生成（generation）、选择（selection）和切换（switching）。其中 LOD 的生成是指，使用不同数量的细节来生成模型的不同表示版本。章节 16.5 中所讨论的网格简化方法可以用于生成所需数量的 LOD。另一种方法则是人工制作具有不同数量三角形的模型。LOD 的选择是指，根据一些标准（例如屏幕上所占据的估计面积）来选择一个 LOD 模型。最后，我们需要将一个 LOD 切换到另一个 LOD，这个过程被称为 LOD 的切换。本小节中将会介绍不同的 LOD 切换和选择机制。

虽然本小节的重点在于在不同的几何表示中进行选择，但是 LOD 背后的思想也可以应用到模型的其他方面，甚至是所使用的渲染方法中。例如：较低 LOD 的模型也可以使用较低分辨率的纹理贴图，从而进一步节省内存开销，并且还可能提高缓存访问性能[240]。着色器本身也可以根据距离、重要性或者其他因素进行来进行适当简化[688, 1318, 1365, 1842]。Kajiya [845]提出了一个分层的尺度，展示了表面光照模型是如何与纹理映射方法相重叠的，以及纹理映射方法反过来是如何与几何细节相重叠的。另一种技术是，对于远处的物体，可以使用更少的骨骼来进行蒙皮操作。

图 19.27：左侧的原始模型由 150 万个三角形组成。右侧的模型只有 1100 个三角形，其原始的表面细节被存储为高度场纹理，并使用浮雕映射进行渲染。

当静态物体相对较远时，广告牌和 impostor（章节 13.6.4）是一种很自然的低成本表现方式[1097]。其他的一些表面渲染方法（例如凹凸映射或者浮雕映射），也可以

用来简化模型的表示，如图 19.27 所示。Teixeira [1754]讨论了如何使用 GPU 来将法线映射到表面上。这种简化技术最明显的缺陷是，silhouette 边缘会失去各自的曲率。Loviscach [1085]提出了一种方法，该方法会沿着 silhouette 边缘挤压出鳍片，从而创建出弯曲的轮廓。

图 19.28：从远处看，兔子的皮毛是使用体积纹理进行渲染的。当靠近兔子的时候，毛发会使用 alpha 混合的折线进行渲染。当非常靠近兔子的时候，沿着 silhouette 边缘的皮毛，会生成几何鳍片并进行渲染。

Lengyel 等人[1030, 1031]给出了一个例子，它展示了这些技术所能用来表现物体的尺度范围。在这项研究中，对于皮毛渲染，在非常近的地方会使用真实的几何形状来进行表示；在稍远的地方会用 alpha 混合的折线来进行表示，然后会使用体积纹理的“壳”来进行混合；最后在很远的地方使用纹理贴图来进行表示，如图 19.28 所示。知道何时以及如何最好地来从一组建模和渲染技术切换到另一组建模和渲染技术，从而最大化帧率和画面质量，仍然是一门艺术，也是一个有待探索的开放领域。

19.1.1 LOD 切换

当从一个 LOD 切换到另一个 LOD 时，突然出现的模型替换通常会很明显，并且会严重分散观众的注意力，这个突然出现的差异被称为 popping。这里将介绍几种不同的

切换方式，每一种都具有不同的 popping 特性。

离散几何 LOD

在最简单类型的 LOD 算法中，不同版本的 LOD 表示实际上是同一个物体模型，但是包含了不同数量的图元（三角形）。该算法非常适合现代图形硬件[1092]，因为这些独立的静态网格可以存储在 GPU 的显存中，并在后续可以进行重复使用（章节 16.4.5）。一个更加详细的 LOD 会有更多的图元。图 19.26 和图 19.29 中分别展示了三个物体的 LOD。图 19.26 还展示了距离观察者不同距离处的 LOD。

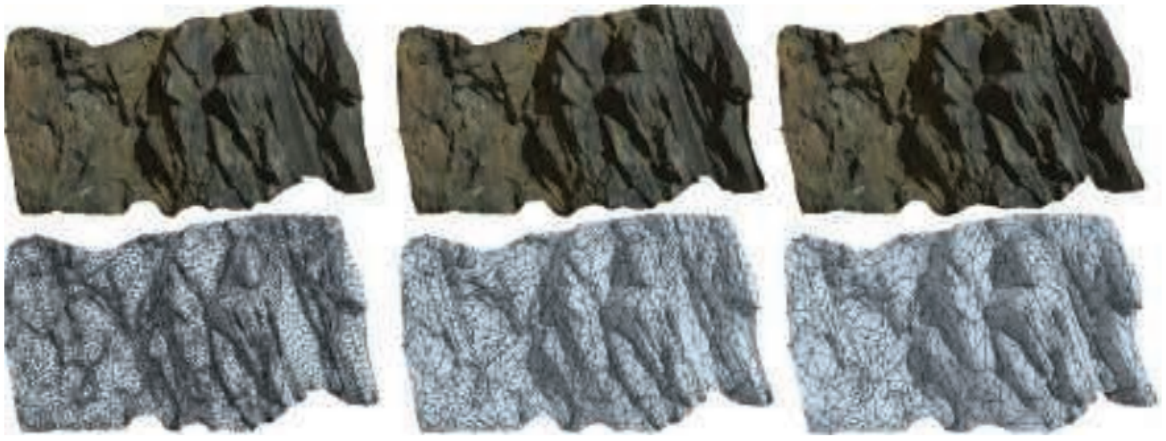


图 19.29：这个悬崖的一部分具有三个不同的 LOD，从左到右分别有 72200 个、13719 个和 7713 个三角形。

在这种方法中，从一个 LOD 切换到另一个 LOD 是突然发生的。也就是说，在当前帧中会使用某个 LOD，然后在下一帧中，LOD 选择机制则会选择另一个 LOD，并且会立即使用该 LOD 进行渲染。对于这种类型的 LOD 方法而言，其 popping 特性通常是最为糟糕的，但是如果这种切换发生在较远的距离处，并且两个相邻 LOD 的渲染差异几乎不可见时，这种方法也可以很好地运行。下面将介绍一些更好的替代方案。

混合 LOD

从概念上讲，一种简单的切换方法是，在短时间内对两个 LOD 之间进行一个线性混合，这样做肯定会使切换过程变得更加平滑。但是为一个物体渲染两个 LOD 模型，自然要比只渲染一个 LOD 模型开销更大，因此这在某种程度上违背了 LOD 的初衷。然而好在 LOD 切换通常只会在很短的时间内发生，并且场景内同时进行切换的物体也不可能会太多，因此相对于额外增加的成本，这种混合方法所提高的质量可能是值得的。

假设我们现在需要在两个 LOD（例如 LOD1 和 LOD2）之间进行过渡，而 LOD1 是当前正在渲染的那个 LOD。现在的问题在于，如何以一种合理的方式来混合和渲染这两个 LOD。如果将这两个 LOD 设置为半透明，那么此时屏幕上正在渲染的物体也将会变成半透明的（尽管会因为两个 LOD 重叠而不会那么透明），这看起来会很奇怪。

Giegl 和 Wimmer [528]提出了一种混合方法，这种方法在实践中的效果很好，并且实现起来十分简单。首先将 LOD1 以不透明的方式，渲染到帧缓冲中（即颜色缓冲和 z-buffer）。然后将 LOD2 的 alpha 值从 0 逐渐增加到 1，并使用“over”混合模式来淡入 LOD2。当 LOD2 的 alpha 值为 1 时，此时 LOD2 是完全不透明的，使其变成当前使用的 LOD，然后再让 LOD1 淡出。正在淡出那个的 LOD 应当在启用深度测试（z-test）和禁用深度写入（z-write）的情况下进行渲染。为了避免在渐变 LOD 的渲染结果上叠加绘制远处的物体，只需要在所有不透明内容绘制完成之后，再按照前后顺序来绘制所有的渐变 LOD 即可，就像在渲染透明物体时所做的那样。请注意，在过渡过程的最中间，此时两个 LOD 都是不透明的，其中一个 LOD 在另一个 LOD 的上面。如果过渡间隔很短，那么这种技术的效果最好，这也有助于保持较小的渲染开销。Mittring [1227]讨论了一个类似的方法，不同之处在于他使用了点阵剔除半透明（可能是在亚像素级别上进行）来消除不同版本之间的差异。

Scherzer 和 Wimmer [1557]通过每帧只更新一个 LOD，并重复使用前一帧的另一个 LOD，来避免同时渲染两个 LOD。前一帧的反向投影可以与一个使用可见性纹理的组合 pass 一起执行。可以得到更快的渲染速度，以及更好的过渡表现。

图 19.30：当观察者离开树木模型时，模型的树枝（以及树木的叶子，图中并没有显示出来）会被缩小，然后最终被移除。

有些物体可以使用其他的切换技术。例如，SpeedTree [887]会对树木 LOD 模型的一部分进行平滑地移动或者缩放，从而避免 popping 现象，图 19.30 展示了这样的一个例子。图 19.31 中展示了一组 LOD，以及用于表现远处树木的广告牌 LOD 技术。

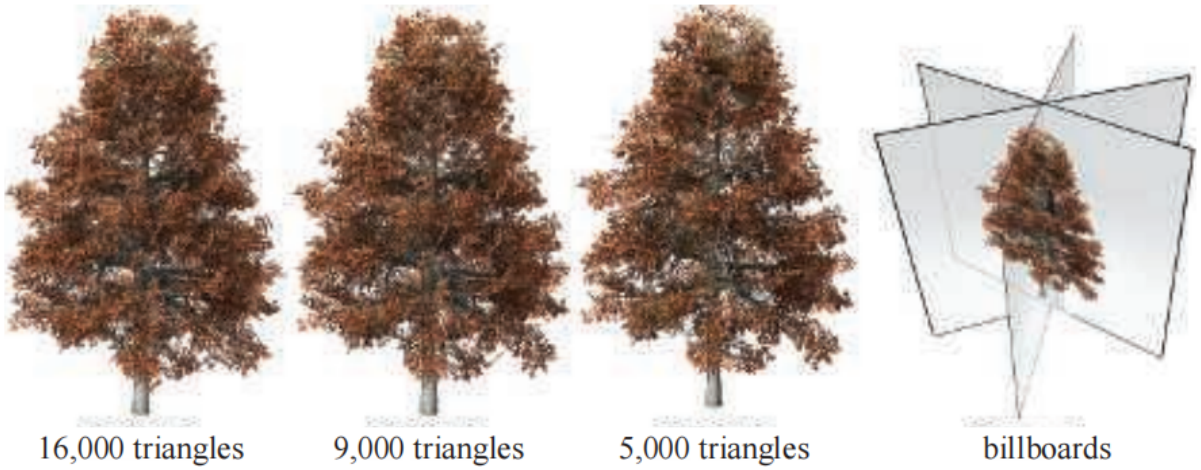


图 19.31：树的 LOD 模型，从左到右距离越来越远。当树木位于远处时，会使用一组广告牌中的其中一个来代表这棵树，如最右侧所示。其中每个广告牌都是从不同的角度对树进行渲染得来的，并由颜色贴图和法线贴图所组成，并会选择一个最面向观察者的广告牌。在实践中，我们通常会形成 8–12 个广告牌（这里显示了 6 个），并修剪掉广告牌的透明部分，从而避免丢弃那些完全透明的像素所带来的时间浪费（章节 13.6.2）。

Alpha LOD

一个避免 popping 的简单方法是使用 alpha LOD。这个技术可以单独使用，也可以与其他 LOD 切换技术结合使用。它应用在最简单的可见 LOD 上，如果只有一个 LOD 可用的话，也可以直接应用在原始模型上。当用于 LOD 选择的度量（例如：到该物体的距离）逐渐增加时，该物体的整体透明度也会逐渐增加（即 α 减小），当物体达到完全透明（$\alpha = 0.0$）时，它最终会消失。当这个 LOD 度量值大于用户定义的不可见阈值时，就会发生这种情况；在达到不可见阈值的时候，只要这个 LOD 度量值保持在阈值之上，就不需要将物体发送到渲染管线中。如果一个物体此时是不可见的，并且其 LOD 度量值低于不可见阈值时，它就会降低其透明度（即 α 增大）并重新开始变得可见。另一种选择是使用章节 19.9.2 中描述的延迟方法。

单独使用这种技术的优点在于，它要比离散的几何 LOD 方法效果更加连续，因此可以避免 popping 现象。此外，由于物体最终会完全消失，不需要进行渲染，因此可以获得显著的加速效果。这个技术的缺点在于，只有当物体完全消失之后，才能获得性能提升。图 19.32 展示了 alpha LOD 的一个例子。

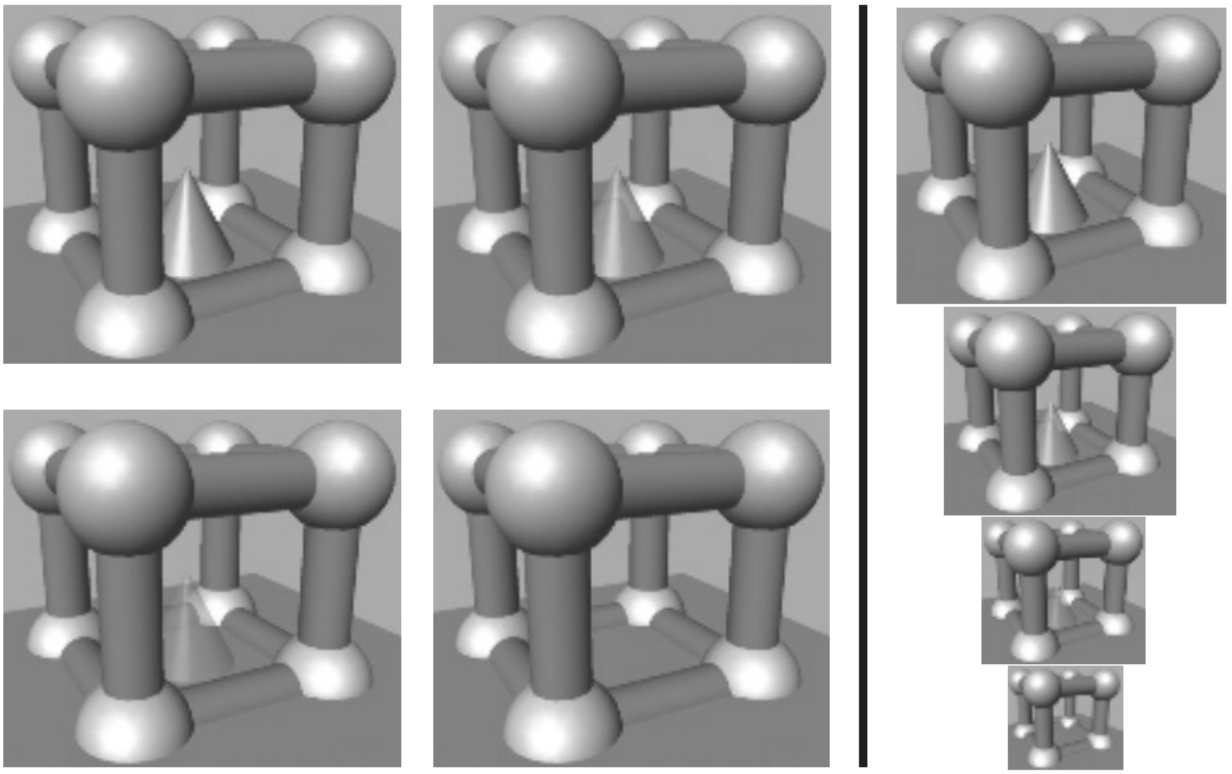


图 19.32：方框中间的圆锥体使用了 alpha LOD 进行渲染。这个圆锥的透明度会随着距离的增加而增加，最终会完全消失。左侧图像以相同的距离进行展示，以供观察圆锥的透明度变化；而右侧图像则以不同的尺寸进行展示，以供观察实际的圆锥效果。

使用 alpha 透明度的一个问题在于，需要对透明物体按照深度进行排序，从而确保这些透明物体能够正确混合。为了使得远处的植被逐渐淡出，Whatley [1876]讨论了如何将噪声纹理用于点阵剔除半透明方法（screen-door transparency）。这样做可以获得一种逐渐溶解的效果，随着距离的不断增加，物体上会有更多的纹理消失。虽然这种方法的质量并不像真正的 alpha 渐变那样好，但是使用点阵剔除半透明，意味着我们不需要进行排序和混合。

CLOD 和地貌 LOD

使用网格简化技术，可以从单个复杂物体中创建出各种不同版本的 LOD 模型，执行这种简化的算法我们在章节 16.5.1 中进行了讨论。其中一种方法是创建一组离散的 LOD，并按照上文中所描述的方法来使用它们。然而，边缘坍缩（edge collapse）方法有一个特性，它允许在不同 LOD 之间，使用其他方法来进行转换。这里我们会介绍两种利用此类信息的方法，这些方法是一些有用的背景，但是目前很少会在实践中进行使用。

每次执行边坍缩操作之后，模型的三角形都会减少两个。在边坍缩操作中，有一条边会被缩短，直到这条边的两个端点重合，然后这条边就消失了。如果将这个过程动画

化，那么就可以在原始模型和略微简化的模型之间进行平滑过渡。对于一次边坍缩过程而言，一个顶点会与另一个顶点相重合；而在一系列的边坍缩过程中，一组顶点会发生移动，并与另一组顶点相重合。通过存储这一系列的边坍缩操作，可以逆转这个过程，这样做的话，一个简化的模型就可以随着时间的推移，逐渐变得更加复杂。边坍缩的翻转过程被称为顶点分裂（vertex split）。因此，这种改变物体 LOD 的精确方法是，可以根据 LOD 选择值来确定可见三角形的数量。在 100 米外，模型可能会由 1000 个三角形组成；而当移动到 101 米时，它可能会下降到 998 个三角形。这种方案被称为连续 LOD 技术（continuous level of detail，CLOD）。因此，这并不是一组离散的模型，而是一组可供显示的庞大模型集合，每个模型的三角形都要比相邻模型（更复杂的那个邻居）少两个。

虽然这样做很有吸引力，但是在实践中使用这种方案仍然会存在一些缺点。并不是 CLOD 流中的所有模型看起来都很好。一组三角形网格的渲染速度要比单个三角形网格快得多，而 CLOD 技术所使用的是动态模型，这要比使用静态模型更加困难。如果场景中有几个相同物体的实例，那么每个 CLOD 物体需要指定自己特定的三角形集合，因为它不会与任何其他的三角形集合相匹配，也就是说，每个 CLOD 所使用的模型都是独立的。Forsyth [481]对这些问题以及其他一些问题的解决方案进行了讨论。大多数 CLOD 技术本质上是串行的，它们并不一定适合在 GPU 上进行实现。因此，Hu 等人[481]提出了一种更加适合 GPU 并行特性的 CLOD 修改版本。他们的技术也是视图依赖的，如果一个物体与视锥体的左侧边界相交，那么位于视锥体外部的模型可以使用更少的三角形，而位于视锥体内部的模型则可以使用更高密度的三角形。

在一次顶点分裂中，一个顶点会变成两个顶点。这意味着复杂模型上的每个顶点都来自于简单模型上的某个顶点。地貌 LOD（Geomorph LOD）[768]是一组通过网格简化所创建的离散模型，这些模型保持了顶点之间的连通性。当从一个复杂模型切换到一个简单模型的时候，复杂模型的顶点会在其原始位置和简单模型的顶点位置之间进行插值。在转换完成之后，就会使用更简单的 LOD 模型来表示这个物体，图 19.33 展示了一个过渡的例子。地貌 LOD 有几个优点，首先是可以提前选择所使用的静态模型，从而获得较高的质量，并且这些模型也可以很容易地转换为三角形网格。像 CLOD 一样，这种平滑过渡也可以避免出现 popping。这种方法的主要缺点在于，每个顶点都需要进行插值；而 CLOD 技术则通常不会使用插值，因此顶点位置的集合本身也不会发生改变。这种方法的另一个缺点在于，物体看起来总是在发生变化，这可能会分散观察者的注意力，尤其是对于具有纹理的物体。Sander 和 Mitchell [1543]描述了一个系统，其中地貌模型与静态模型、GPU 驻留的顶点缓冲和索引缓冲

会一起进行使用。也可以与 Mittring [1227]的点阵剔除半透明方法相结合，从而实现更加平滑的过渡效果。

图 19.33：最左边和最右边分别展示了一个低细节模型和一个高细节模型。中间则展示了一个在左右模型之间进行插值的地貌模型。请注意，中间的奶牛模型与右边的模型具有相同数量的顶点和三角形。[1196]

GPU 支持一种被称为分数曲面细分的相关思想。在这种方案中，表面的曲面细分因子可以被设置为任意的浮点数，因此可以避免 popping。例如：分数曲面细分可以用于 Bezier 面片和位移映射图元。有关这些技术的更多信息，详见章节 17.6.1。

19.9.2 LOD 选择

考虑到一个物体可能会存在不同的 LOD 模型，我们必须选择其中一个进行渲染，或者是对其中的一些进行混合。这是 LOD 选择（LOD selection）所负责的任务，这里我们将介绍几种不同的技术，这些技术也可以用于为遮挡剔除算法选择一组良好的遮挡物。

通常来说，LOD 选择的度量，也称为效益函数（benefit function），是针对当前观察视角和物体位置进行评估计算的，并且会根据该度量值来选择一个适当的 LOD。这个度量可以基于，例如：物体包围体的投影面积，或者是从观察点到物体的距离。在这里我们将效益函数的值记为 r 。有关如何快速估计一条线段在屏幕上的投影长度，详见章节 17.6.2。

基于范围

选择 LOD 的一种常用方法是，将物体的不同 LOD 与不同的距离范围关联起来。其中最详细的 LOD，其范围是从 0 到某个用户定义的值 r_1 ，这意味着当到物体到相机的距离小于 r_1 时，才能看到这个最详细的 LOD。下一个 LOD 的范围从 r_1 到 r_2 ，其中 $r_2 > r_1$ 。也就是说，如果物体到相机的距离大于等于 r_1 且小于 r_2 ，则使用这个

LOD，并依此类推。图 19.34 展示了这样的一个例子，其中包括场景图中的四种不同 LOD 范围，以及它们对应的 LOD 节点。

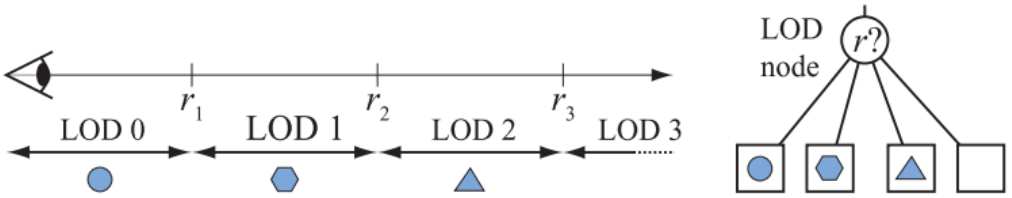


图 19.34：左侧展示了基于范围的 LOD 的工作原理。请注意，其中的第四个 LOD 实际上是一个空物体，因此当物体比距离 r_3 还要远时，我们不会渲染任何内容，因为此时这个物体对于图像的贡献很小，不值得我们去绘制。右侧展示了场景图中的 LOD 节点。这个 LOD 节点会根据距离 r ，只会选择一个子节点进行绘制。

如果这个用于确定使用哪个 LOD 的度量，会在不同帧之间围绕某些 r_i 发生变化，那么可能会出现我们不想看到的 popping 现象。此时会发生不同层级之间的快速循环切换，这个问题可以通过在 r_i 附近引入一些延迟切换来进行解决[898, 1508]。图 19.35 展示这种基于范围的 LOD 切换方法。在图中我们可以看到，当距离 r 增加时，会使用 LOD 范围的上面一行；当 r 减小时，则会使用 LOD 范围的下面一行。

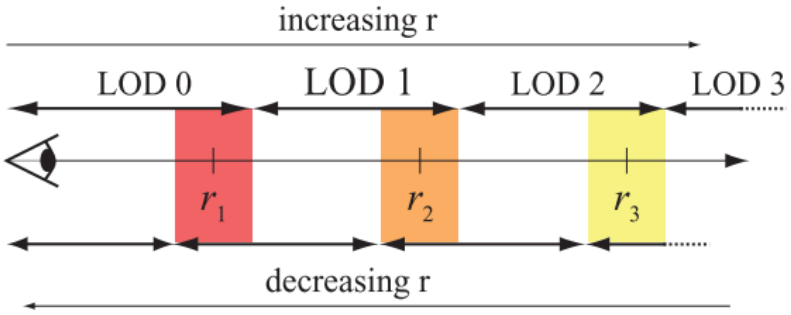


图 19.35：彩色区域代表了 LOD 技术的延迟切换区域。

图 19.36 展示了在过渡范围内对两个 LOD 进行混合。然而，这种方法并不理想，因为物体到相机的距离 r 可能会长时间停留在这个过渡范围内，由于需要混合两个 LOD，因此这样会增加渲染负担。相反，Mittring [1227]选择在有限的时间内，当物体达到一定的过渡范围时就执行 LOD 切换。为了获得最佳效果，这种方法应当与上述的延迟方法结合使用。

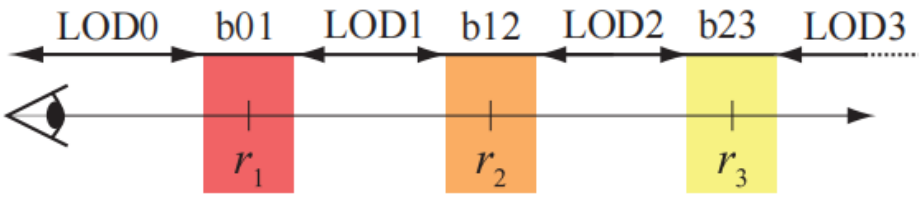


图 19.36：着色区域代表了在两个最近的 LOD 之间进行混合的范围。例如：b01 表示在 LOD0 和 LOD1 之间进行混合；而 LODk 表示在相应的范围内只会渲染 LODk 这个模型。

基于投影面积

LOD 选择的另一个常用度量是包围体的投影面积，或者是对包围体投影面积的估计。下面，我们将展示如何以透视方式，来估计球体和 box 区域的像素数量，这个像素数量也被称为屏幕空间覆盖率（screen-space coverage）。

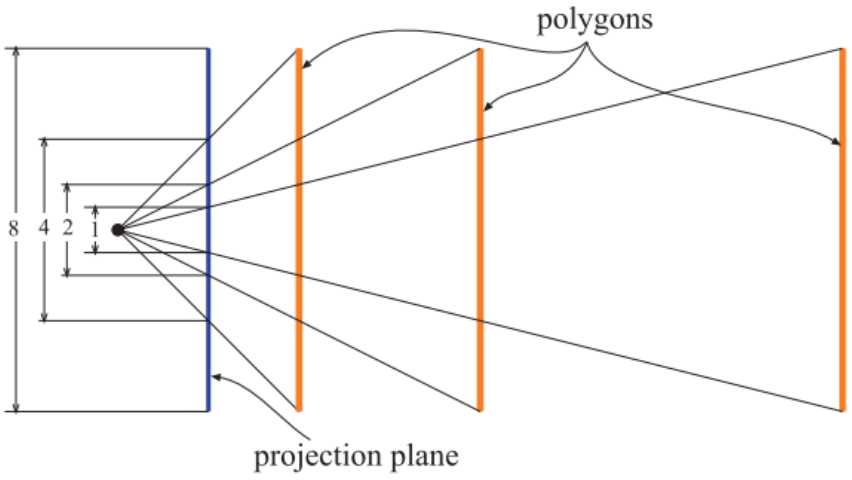


图 19.37：这幅图说明了，当到相机的距离翻倍时，物体（没有任何厚度）的投影尺寸是如何减半的。

我们首先从球体开始，这里的估计基于了这样的一个事实：即物体的投影大小会随着与相机沿观察方向的距离增大而减小，即物体距离相机越远，投影面积就越小。如图 19.37 所示，它说明了如果与相机的距离增加一倍，那么投影的大小将会减半，这个规律适用于面向观察者的平面物体。我们使用中心点 $\mathbf{c}$ 和半径 r 来定义一个球体。观察者位于点 $\mathbf{v}$ 处，沿着归一化的观察向量 $\mathbf{d}$ 进行观察。沿着观察方向 $\mathbf{d}$，从点 $\mathbf{c}$ 到点 $\mathbf{v}$ 的距离，其实就是球体中心在观察向量上的投影：$\mathbf{d} \cdot (\mathbf{v} - \mathbf{c})$。同时我们假设，从相机到视锥体近裁剪平面的距离为 n。我们会在对投影面积的估计中使用近裁剪平面，从而使得那些位于近裁剪平面上的物体能够获得其原始尺寸。有了上述的推理过程，对投影球面的估计半径为：

$$p = \frac{nr}{\mathbf{d} \cdot (\mathbf{v} - \mathbf{c})} \tag{19.6}$$

以像素为单位的投影面积为 $\pi p^2 wh$，其中的 $w \times h$ 实际上是屏幕分辨率。较高的投影面积值可以选择更加详细的 LOD。这是一个近似的估计值，实际上三维球体的二维投影是一个椭圆，正如 Mara 和 McGuire [1122]所指出的那样。他们还推导了一种计算保守包围多边形（conservative bounding polygon）的方法，即使是在球体与近裁剪平面相交的情况下。

通常的做法是直接在物体的包围盒周围使用一个近似的包围球。另一种估计方法是使用物体包围盒的屏幕包围框。然而，对于一些薄的、或者扁平的物体，它们实际覆盖的投影面积可能会有很大的变化。例如：想象一根意大利面，这根面条的一端在屏幕的左上角，另一端在屏幕的右下角，那么它的包围球将覆盖整个屏幕，并且它的包围盒的最小二维屏幕边界和最大二维屏幕边界也是一样的。

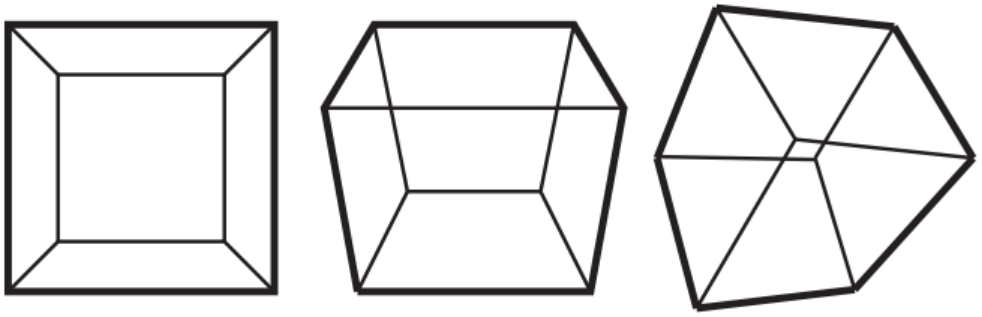


图 19.38：立方体的三种投影情况，从左到右分别会显示一个正面、两个正面和三个正面。这些立方体投影的轮廓分别由 4 个、6 个和 6 个顶点组成，每个投影轮廓的面积是根据每个形成的多边形来进行计算的。[1569]

Schmalstieg 和 Tobler [1569]提出了一种快速计算 box 投影面积的方法。这个想法是：根据 box 来将相机的视点（即观察位置）进行分类，并使用这个分类信息，来确定哪些投影顶点被包含在投影 box 的轮廓内部。这个过程是通过一个查找表（LUT）来完成的。使用这些投影顶点，我们可以计算出这个 box 在视野中的面积。相机视点与投影 box 的分类主要有 3 种情况，如图 19.38 所示。实际上，这种分类是通过确定相机视点位于包围盒平面的哪一侧来完成的。为了提高效率，我们会将视点转换到 box 的坐标系中，这样在分类的时候就只需要进行一些比较操作就可以了。这些比较的结果会放入一个位掩码中，这个位掩码会作为查找表的索引。这个 LUT 决定了从视点所看到的轮廓内部具体有多少个 box 顶点。然后会使用另一个查找表来实际查找轮廓顶点，在将轮廓顶点投影到屏幕上之后，我们再计算投影轮廓的面积。为了避免

估计误差（有时会很严重），需要使用视锥体的两侧平面，来对形成的投影多边形进行裁剪。网上可以找到相应的源代码。Lengyel [1026]对该方案进行了一些优化，可以使用更加紧凑的 LUT。

仅仅根据距离或者投影面积来选择 LOD 并不总是一个好主意。例如：如果一个物体具有特定的 AABB，其中包含一些很大的三角形和一些较小的三角形，那么这些较小的三角形可能会由于四边形过度渲染（quad overshading），从而出现严重的锯齿并降低性能表现。如果另一个物体具有完全相同的 AABB，但是其中包含了中等大小的三角形和一些较大的三角形，那么基于距离和基于投影面积的选择方法，将会选择和之前完全相同的 LOD。为了避免这种情况，Schulz 和 Mader [1590]使用了几何平均值 g 来帮助选择 LOD：

$$g = \sqrt[n]{t_0 t_1 \cdots t_{n-1}} \tag{19.7}$$

其中 t_i 是物体三角形的大小。使用几何平均值而不是算术平均值的原因在于，即使包含一些较大的三角形，许多小三角形也会使得 g 变小。对于最高分辨率的模型，这个值是离线计算的，同时会用于预先计算第一次切换应当发生的距离。后续的切换距离是一个第一次切换距离的简单函数。这使得他们的系统可以更加频繁地使用较低的 LOD，从而提高性能表现。

另一种方法是计算每个离散 LOD 的几何误差，即计算简化模型与原始模型的最大偏差有多少米。可以将这个偏差距离投影到屏幕空间中，从而确定使用该 LOD 在屏幕空间中的效果。然后选择那个能够满足屏幕空间误差（由用户定义）的最低 LOD。

其他选择方法

基于距离和基于投影面积的 LOD 选择通常是最常用的方法。然而，还可以使用一些其他的指标，我们将在这里介绍一些。除了投影面积之外，Funkhouser 和 Sequin [508]还建议使用物体的重要性（例如：墙壁模型比墙上的时钟更加重要）、运动、滞后（在切换 LOD 时，会降低效益函数）以及焦点。最后一个因素，即观察者的注意力焦点，可能是一个十分重要的选择因素。例如：在一些球类体育游戏中，控制球的那个角色会是用户最为关注的地方，因此其他角色可以具有相对较低的 LOD [898]。同理，当在虚拟现实应用程序中使用眼动追踪（eye tracking）时，应当在用户正在观察的地方使用更高的 LOD。

根据应用程序类型的不同，使用一些其他策略可能也会有奇效。可以使用整体可见性（overall visibility）这个指标，例如：穿过茂密的树叶来观察附近的物体，这些物体可以使用较低的 LOD 来进行渲染。还有更多的全局性指标可以利用，例如：对能够

使用的高精度 LOD 总数进行限制，从而使得整个场景始终都能保持在给定的三角形预算内[898]。有关这个主题的更多信息，详见下一小节。其他的一些因素还有可见度、颜色和纹理等。一些感知指标也可以用于 LOD 选择[1468]。

McAuley [1154]提出了一个植被系统，其中树干和树叶 cluster 在使用 impostor 表示之前，还有三个 LOD。他从不同角度和不同距离，对每个物体 cluster 之间的可见性进行了预处理。由于树木背后的 cluster 可能会被更近的 cluster 所遮挡隐藏，因此即使这些树木距离很近，也可以为这些位于背后的 cluster 选择较低的 LOD。对于草地渲染而言，通常会在靠近观察者的地方使用真正的几何体，在稍微远一点的地方使用广告牌，并在很远的地方上直接使用地面纹理[1352]。

19.9.3 限时的 LOD 渲染

一个恒定的、稳定的帧率，通常是渲染系统的理想特性。事实上，通常这被称为“硬实时（hard real-time）”或者限时渲染（time-critical rendering）。这样的系统会被赋予特定的执行时间（例如 16 毫秒），并且必须在这段时间内完成它的任务（例如：渲染图像）。当限制时间到了，系统必须停止处理。如果场景中的物体都由 LOD 进行表示，那么一个硬实时的渲染系统，将必须能够在每一帧中向用户显示更多或者全部场景，而不是在所分配的时间内只渲染少数几个非常详细的模型。

Funkhouser 和 Sequin [508]提出了一种启发式算法，该算法可以对场景中的所有可见物体调整它们的 LOD 选择，以满足恒定帧率的要求。这个算法是预测性的（predictive），因为它会根据所需的目标帧率、以及哪些物体是可见的，来调整选择可见物体的 LOD。与响应式的（reactive）算法相比，一个响应式的算法会根据渲染前一帧所花费的时间，来进行调整物体的 LOD 选择。

假设现在有一个叫做 O 的物体，并在细节水平 L 上进行渲染，我们将在给定细节水平来渲染指定物体记作 (O, L)。然后我们定义两种启发式算法。其中一个启发式算法，会对在一定细节水平上渲染物体的成本进行估计，即 $\mathrm{Cost}(O, L)$；另外一个启发式算法，会对在一定细节水平上渲染物体的效益进行估计，即 $\mathrm{Benefit}(O, L)$。这个效益函数会估计在一定 LOD 下，物体对图像画面的贡献。

假设位于视锥体内部或者与之相交的物体集合被称为 S，那么这个算法背后的主要思想是：使用启发式的选择函数，来优化物体 S 的 LOD 选择。具体来说，我们要最大化总效益：

$$\sum_{S} \mathrm{Benefit}(O, L) \tag{19.8}$$

同时最小化总成本，使其满足所要求的目标帧率：

$$\sum_{S} \text{Cost}(O, L) \leq T \tag{19.9}$$

其中 T 为目标帧时间。

换句话说，我们希望能够在理想的帧率内，选择物体适当的 LOD，从而获得“最佳图像”。接下来，我们会介绍是如何对成本函数和效益函数进行估计的，并给出了上述方程的优化算法。

成本函数和效益函数很难进行定义，因此它们要在所有情况下都能适用。其中的成本函数可以通过使用不同的观察参数，对 LOD 进行多次渲染计时来进行估计。而有关不同的效益函数，详见章节 19.9.2。在实践中，物体的 BV 投影面积可以作为一个效益函数。

最后，我们将会讨论如何选择场景中物体的 LOD。首先，我们注意到以下几点：对于某些视点而言，场景可能会过于复杂，以致于无法达到所需的帧率。为了解决这个问题，我们可以为每个物体都定义一个最低的 LOD，即一个没有图元的物体，也就是说，使用这个最低 LOD 的时候，我们不会渲染这个物体[508]。使用这个技巧，我们可以只渲染最重要的那些物体，跳过不重要的那些物体。

为了对一个场景选择“最佳”的 LOD，需要在方程 19.9 的约束下，对方程 19.8 进行优化。这是一个 NP 完全（NP-complete）问题，这意味着想要正确地解决这个问题，唯一要做的就是对所有不同的组合进行测试，并选择其中表现最好的那一组。对于任何算法而言，这显然都是不可行的。一种更简单、更可行的方法是使用贪婪算法，这个算法会试图最大化每个物体的 $\text{Value} = \text{Benefit}(O, L)/\text{Cost}(O, L)$ 。这个算法会对视锥体内的所有物体进行处理，并按照 Value 降序的顺序，依次选择物体进行渲染，即首先会渲染 Value 最大的那个物体。如果一个物体在多个 LOD 的情况下都具有相同的 Value 值，那么就选择效益最高的那个 LOD 来进行呈现。这种方法具有最大的“性价比（bang for the buck）”。对于视锥体内的 n 个物体，这个算法的时间复杂度为 $O(n \log n)$ ，并且这个算法所能产生的解至少也会有最佳解的一半好[507, 508]。还可以利用帧与帧之间的一致性，来加快 Value 值的排序。

我们可以在 Funkhouser 的博士论文中[507]，找到更多有关 LOD 管理、以及将 LOD 管理与入口剔除相结合的信息。Maciel 和 Shirley [1097]将 LOD 与 impostor 结合起来，提出了一种近似恒定时间的室外场景渲染算法。其大致想法是：使用一个物体不同表示方法的层次结构（例如：一组 LOD 和分层 impostor），然后以某种方式

来遍历这棵树，从而在给定的时间内给出最佳图像。Mason 和 Blake [1134]提出了一种增量的层次 LOD 选择算法。同样地，可以使用任意的物体的表示方法。Eriksson 等人[441]提出了分层 LOD（hierarchical levels of detail，HLOD）。使用这种方法，可以以一个恒定的帧率来渲染一个场景，或者只会出现有限的渲染错误。与此相关的是渲染的功耗预算，Wang 等人[1843]提出了一个优化框架，它可以选择良好的参数来降低功耗，这对于手机和平板电脑等设备而言十分重要。

与限时渲染相关的另一组技术是使用静态模型。当相机不发生移动的时候，整个模型会被渲染，累积缓冲可以用于实现抗锯齿、景深、软阴影等效果，并进行渐进式的更新。当相机进行移动的时候，为了满足一定的帧率，所有物体使用的 LOD 都可以适当降低，还可以使用细节剔除来完全移除场景中的微小物体。

19.10 渲染大型场景

到目前为止我们所提及的内容其实已经有所暗示了，即我们想要渲染的场景一定要能装载进计算机的内存中。但是情况可能并非总是如此。例如：有些主机只有 8 GB 的内存，而有些游戏世界可能会包含数百 GB 的数据。因此，本小节将会介绍纹理的流式传输和转换编码方法，以及一些通用的流式技术，最后会介绍一些地形渲染算法。请注意，这些方法几乎总是会与本章节前面所描述的剔除技术和 LOD 方法结合使用。

19.10.1 虚拟纹理和流式传输

想象一下，为了能够渲染一个巨大的地形数据集，我们可能需要使用一个超大分辨率的纹理，这个分辨率会大到令人难以置信，并且这个纹理大到装不进 GPU 的显存。例如：在游戏《狂怒（RAGE）》中，一些虚拟纹理的分辨率为 $128k \times 128k$，这将消耗大约 64 GB 的 GPU 显存[1309]。当 CPU 上的内存空间有限时，操作系统会使用虚拟内存来进行内存管理，根据需要来将数据从驱动程序交换到 CPU 内存中[715]。这就是稀疏纹理（sparse texture）所提供的功能，它使得分配一个巨大的虚拟纹理成为可能[109, 248]，这里的虚拟纹理也被称为 megatexture，这些技术有时会被称为虚拟纹理（virtual texture）或者部分驻留纹理（partially resident texturing）。应用程序会决定每个 mipmap 层级中的哪些区域（瓦片 tile）应当驻留在 GPU 显存中。一个 tile 通常为 64 kB，其纹理分辨率取决于具体的纹理格式。这里，我们将会介绍虚拟纹理和流式传输技术。

一个使用 mipmap 的高效纹理系统，有着这样一个关键准则：在理想情况下，所需要的纹素数量应当与正在渲染的最终图像的分辨率成比例，并且与纹理本身的分辨率无

关。因此，我们只要求那些可见的纹素位于物理 GPU 的显存中即可，与整个游戏世界中的纹素相比，这部分纹素是相当有限的。其主要概念如图 19.39 所示，其中整个 mipmap 链会在虚拟内存和物理内存中被划分为若干个块。这些结构有时会被称为虚拟 mipmap 或者 clipmap [1739]，后者指的是将一个较大 mipmap 中的一小部分裁剪出来以供使用。由于物理内存的大小比虚拟内存小得多，因此只有一小部分虚拟纹理块可以被装入物理内存中。几何图形在虚拟纹理中使用全局的 uv 参数化，并且在像素着色器中使用这样的 uv 坐标之前，它们需要被转换为指向物理纹理内存的纹理坐标。这个过程可以使用 GPU 支持的页表（如图 19.39 所示）或者间接纹理（在 GPU 上的软件中）来完成。任天堂 GameCube 的 GPU 支持虚拟纹理，近年来，PLAYSTATION 4、Xbox One 和许多其他 GPU 也都支持了硬件虚拟纹理。当贴图映射和解除映射（unmap）到物理内存中时，间接纹理需要使用正确的偏移量来进行更新。使用一个巨大的虚拟纹理和一个较小的物理纹理具有很好的效果，因为对于远处的几何体，只需要加载一些较高层级的 mipmap 贴图到物理内存中即可；而对于那些靠近相机的几何体，只需要加载一些较低层级的 mipmap 贴图即可。需要注意的是，虚拟纹理也可以用于从磁盘流式传输巨大尺寸的纹理，也可以用于稀疏阴影映射（sparse shadow mapping）等[241]。

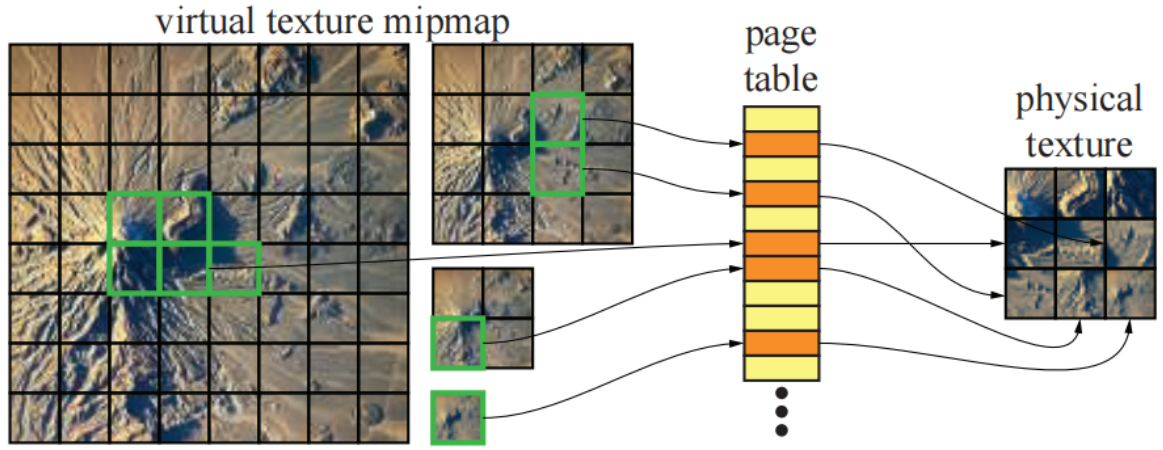


图 19.39：在虚拟纹理中，一个具有 mipmap 层次结构的大型虚拟纹理，会被划分为大小 128×128 个像素的 tile（左）。其中只有一小部分（在本例中为 3×3 大小的块）可以被装入物理内存（右）。想要找到虚拟纹理贴图的实际位置，需要将虚拟地址转换为物理地址，这里是通过页表（page table）完成的。请注意，为了减少内存碎片，并非所有物理内存中的 tile 都有来自虚拟纹理的箭头。

由于物理内存是有限的，所有使用虚拟纹理的引擎都需要一种方法来确定哪些贴图应当驻留在物理内存中，而哪些贴图不应当驻留在物理内存中，有几种这样的方法。Sugden 和 Iwanicki [1721]使用一个反馈渲染方法（feedback rendering

approach），其中第一个渲染 pass 会写出所有需要知道的信息，即哪个片元将会访问哪个纹理 tile。当这个 pass 执行完成之后，这个信息纹理将会被读取回到 CPU 中并进行分析，从而找到需要使用的贴图。没有驻留在物理内存中的 tile 将会被读取，并映射到物理内存中，而物理内存中那些不需要进行使用的 tile 将会被解除映射。但是他们的方法不适用于阴影、反射和透明效果。不过，可以使用点阵剔除半透明技术（章节 5.5）来生成透明效果，效果相当不错。van Waveren 和 Hart [1855]也使用了反馈渲染。请注意，这个 pass 既可以是单独的渲染 pass，也可以与一个 z-prepass 结合使用。当使用单独的 pass 时，只能使用 80×60 像素的分辨率来作为近似值，从而减少处理时间。Hollemeersch 等人[761]使用一个计算 pass 来执行这个过程，而不是将反馈缓冲区读回 CPU 中。这样做的结果是在 GPU 上创建了一个紧凑的 tile 标识符列表，并将其发送回 CPU 进行纹理映射操作。

使用 GPU 支持的虚拟纹理，驱动程序将会负责资源的创建和销毁，以及对 tile 建立映射和解除映射，并确保物理分配会得到虚拟分配的支持[1605]。使用 GPU 的硬件虚拟纹理，一次 $\mathrm{sparseTexture}$ 查找除了会返回过滤值之外（对于常驻贴图），还会返回一个代码，这个代码会指示相应的贴图是否为常驻贴图[1605]。但是如果使用软件支持的虚拟纹理的话，所有这些任务就都落在了开发人员身上。我们可以参考 van Waveren 的报告，来了解更多关于这个主题的信息[1856]。

为了确保所有内容都能够装入物理内存，van Waveren 对全局纹理的 LOD 偏移进行了调整，直到所使用的工作集合符合要求[1854]。此外，如果只有一个更高层级的 mipmap tile 可以使用时，如果此时需要使用较低层级的 mipmap tile，则需要一直先使用更高级别的 mipmap tile，直到较低级别的 mipmap tile 可以使用为止。在这种情况下，可以先对更高级别的 mipmap 贴图进行放大操作，并进行使用，然后随着时间的推移，新的低级别 mipmap 贴图可以逐渐混合在一起，从而在它可以使用时实现平滑过渡。

Barb [99]总是会加载小于或者等于 64 kB 的所有纹理，这样一来可以保证在还没有加载更高分辨率 mipmap 层级的时候，这些纹理化操作总是可以完成的，尽管质量会低一些。他使用离线的反馈渲染来在各种位置上，预先计算每个 mipmap 层级在标称（nominal）纹理和屏幕分辨率下，每种材质会覆盖玩家周围多大范围的立体角。在运行过程中，这些信息会被流式输入，并根据每个材质纹理的分辨率和最终的屏幕分辨率进行动态调整。这将为每个纹理的每个 mipmap 层级都生成一个重要性值，将每个重要性值除以对应的 mipmap 层级中的纹素数量，从而生成一个合理的最终度量，这样做的话，即使将纹理细分为更小的、具有相同映射的纹理，这个度量值也是不变

的。有关这个话题的更多信息，详见 Barb 的演讲[99]。图 19.40 展示了一个渲染示例。

图 19.40：在《毁灭战士（2016）》中，使用流式纹理来访问一个巨大图像数据库中的高分辨率纹理映射。

Widmark [1881]描述了如何将流式传输与程序化纹理的生成相结合，从而获得更加多样化和更加详细的纹理。Chen 对 Widmark 的方案进行了扩展，使其能够处理大一个数量级的纹理 [259]。

19.10.2 纹理转码

为了使得虚拟纹理系统能够更好地工作，可以将其与转码（transcoding）技术相结合。这是一个从磁盘上读取图像的过程，通常可以使用一些可变压缩比的压缩方案（例如 JPEG）；然后对图像进行解码，再使用 GPU 支持的一种纹理压缩方案（章节 6.2.6）对其进行编码。图 19.41 展示了这样的一个系统。其中反馈渲染 pass 的目的是确定当前帧需要哪些贴图，章节 19.10.1 中介绍的两种方法都可以在这里使用。其中的 fetch 步骤是指在存储层次结构中获取所需要的数据，即从光学存储或者硬盘驱动器（HDD）中进行获取数据，或者是从可选的磁盘缓存中获取数据，然后到达由软件管理的内存缓存中。unmap 指的是释放一个常驻的 tile。当读取新数据之后，会对其进行转码处理，并最终映射到新的常驻 tile 中。

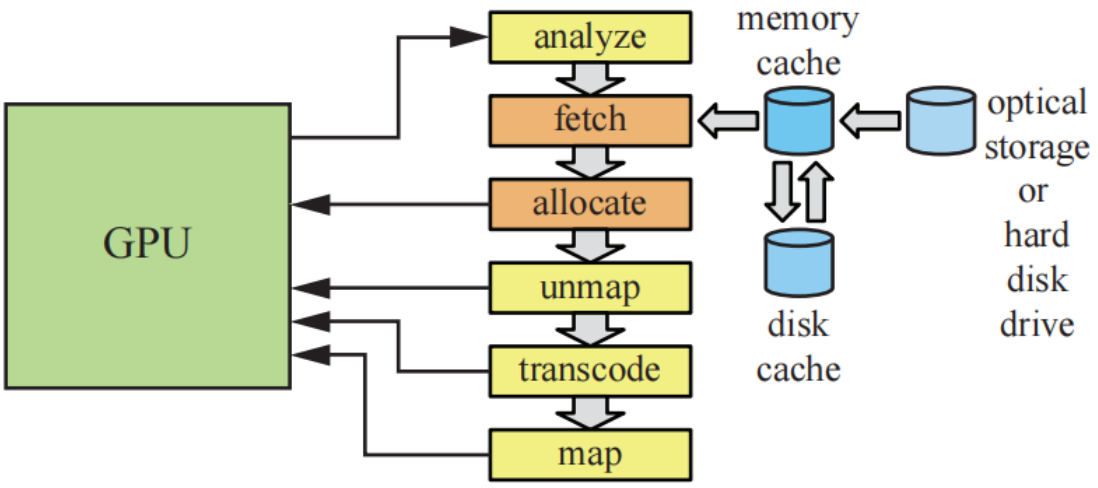


图 19.41：一个使用虚拟纹理和转码的流式纹理系统。[1855]

使用转码处理的优点在于，当纹理数据存储在磁盘上时可以使用更高压缩比的压缩算法，当通过纹理采样器访问纹理数据时可以使用 GPU 支持的纹理压缩格式。这需要对可变压缩比的压缩格式进行快速解压缩，以及快速压缩到 GPU 支持的格式[1851]。还可以对已经压缩过的纹理再次进行压缩，从而进一步降低文件大小[1717]。这种方法的优点在于，当从磁盘中读取纹理并进行解压缩时，它就已经是能够被 GPU 使用的纹理压缩格式了。crunch 库具有免费的开源代码[523]，它使用了类似的方法，能够达到每个纹素 1–2 bit 的结果，图 19.42 展示了一个例子。它的后继者称为 basis，它是一种对块进行可变 bit 压缩的专有格式，可以快速转换为各种纹理压缩格式[792]。对于 BC1/BC4 [1376]， BC6H/BC7 [933, 935, 1259]以及 PVRTC [934]，可以使用 GPU 上的快速压缩方法。Sugden 和 Iwanicki [1721]使用 Malvar 压缩方案[1113]的一种变体，来实现在磁盘上的可变压缩比的压缩方案。法线贴图的压缩比为 $40:1$，使用 YCoCg 变换（方程 6.6）的反照率纹理的压缩比为 $60:1$。Khronos 组织正在开发一种标准的通用纹理压缩文件格式。

图 19.42：展示了转换的质量。从左到右分别是：原始的部分鹦鹉图像；对原始图像眼睛放大结果（24bit/像素）；ETC 压缩图像（4bit/像素）；压缩后的 ETC 图像（1.21bit/像素）。

当需要高质量的纹理，并且同时要保证较短的纹理加载时间时，Olano 等人[1321]使用可变压缩比的压缩算法，来将压缩后的纹理存储在磁盘上。纹理也可以在 GPU 显存中以压缩形式进行存储，直到它们被需要的时候，此时 GPU 会使用自身的算法对它们进行解压缩操作，之后它们会以未压缩的形式进行使用。

19.10.3 通用流式传输

在游戏或者其他实时渲染应用程序中，如果模型所占据的存储空间要比物理内存大，那么还需要使用一个流式系统来处理实际的几何图形、脚本、粒子和 AI 等内容。一个平面可以使用三角形、正方形或者六边形等正凸多边形来进行平铺。因此，这些形状也是流式系统中常见的构建块，其中每个多边形都与该多边形中的资产相关联，如图 19.43 所示。需要注意的是，在这些形状中，最常用的是正方形和正六边形[134, 1522]，可能是因为它们的邻居要比三角形少。在图 19.43 中，观察者位于深蓝色多边形处，流式系统可以确保深蓝色多边形的直接邻居（浅蓝色和绿色）会被加载到内存中。这是为了确保周围的几何图形可以用于渲染，并保证当观察者移动到相邻多边形上的时候，该多边形上已经存有数据了。请注意，三角形和正方形有两种类型的邻居：其中一种邻居共享一条边，另一种邻居只共享一个顶点。

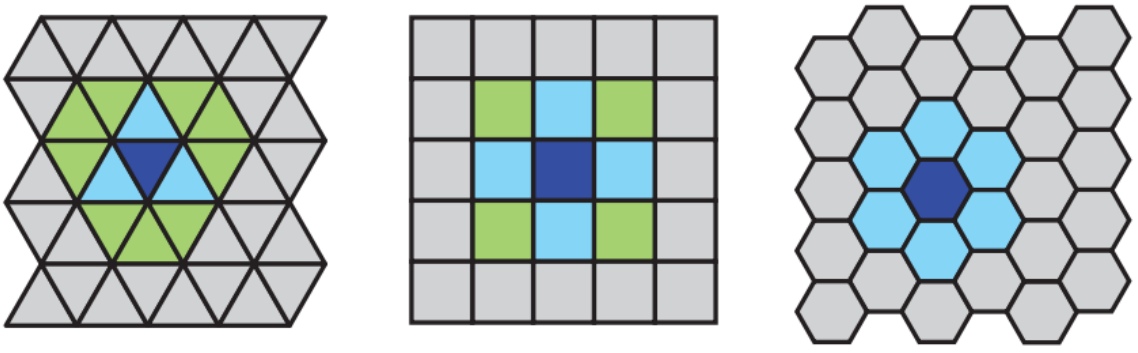


图 19.43：二维平面可以使用正多边形进行平铺，例如使用三角形（左）、正方形（中）和六边形（右）。从正上方看，这些 tile 通常会平铺覆盖在一个游戏世界上，多边形内的所有资产（模型、纹理、AI、粒子等）都与该多边形相关联。假设观察者此时位于深蓝色多边形中，其相邻多边形的资源也会被加载。

Ruskin [1522]使用了正六边形进行平铺，每个六边形都具有一个低分辨率的几何 LOD 和一个高分辨率的几何 LOD。由于低分辨率 LOD 的内存占用较小，因此会始终加载整个世界的低分辨率 LOD。因此，只有高分辨率的 LOD 和纹理会在内存中和内存外进行流式处理。Bentley [134]则使用了正方形，每个正方形的面积为 $100 \times 100m^2$。高分辨率的 mipmap 与其他资源会分开进行流式传输。该系统在进行中距离观察的时候，会使用 1–3 个 LOD；在远距离观察的时候，会使用烘焙的

impostor。在赛车游戏中，Tector [1753]的策略是在赛车前进时，沿着赛道来加载数据。他将使用 zip 格式将压缩的数据存储在磁盘上，并将数据块加载到压缩软件的缓存中。然后根据需要再对这些数据块进行解压缩，并由 CPU 和 GPU 的内存层次结构进行使用。

在某些应用中，可能还需要对三维空间进行平铺覆盖，而不是像上面所描述的那样只使用二维平铺。请注意，立方体（cube）是唯一可以平铺三维空间的正多面体，因此它自然是此类应用的唯一选择。

19.10.4 地形渲染

地形渲染（terrain rendering）是许多游戏和应用程序的重要组成部分，例如谷歌地球和 Cesium 用于大世界渲染的开源引擎[299, 300]，如图 19.44 所示。我们会介绍几种在当前 GPU 上表现良好的流行方法。值得注意的是，为了在地形放大的时候提供较高的 LOD，这些方法都可以添加分形噪声（fractal noise）。此外，许多系统会在加载游戏或者关卡场景的时候，程序化生成地形。

图 19.44。由航空摄影测量仪拍摄的 50 厘米地形和 25 厘米地形的 Chamberlin 山图像。

其中一种方法是几何 clipmap [1078]。它类似于纹理 clipmap [1739]，因为它使用了与 mipmap 相似的分层结构，即原始几何图形会被过滤成一个金字塔，位于靠近顶部的层级会更加粗糙，如图 19.45 所示。

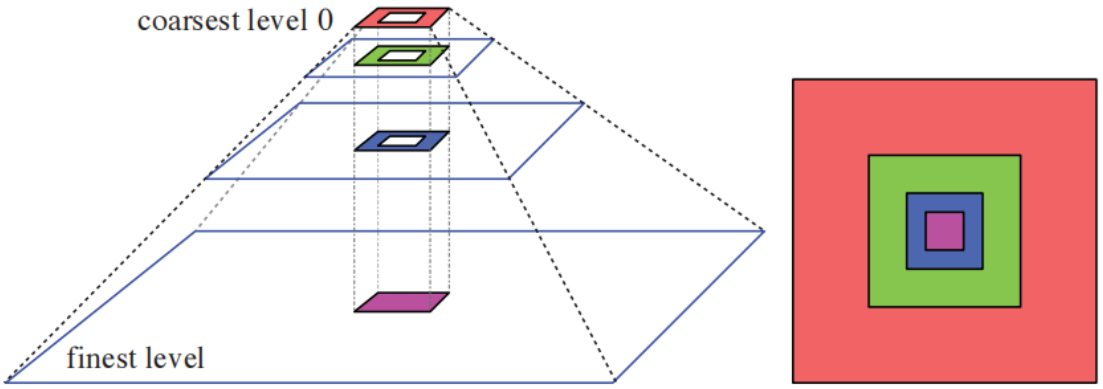


图 19.45：左：几何 clipmap 的结构，在每个分辨率级层级中，都会缓存一个相同大小的正方形窗口。右：几何图形的俯视图，其中观察者位于最中间的紫色区域。请注意，只有最精细的那个层级才会渲染它的完整正方形区域，而其他层级所渲染的正方形都是中空的。[82]

当渲染巨大的地形数据集时，只有观察者周围的 $n \times n$ 个样本（即高度）才会被缓存在内存中。当观察者进行移动的时候，图 19.45 中的层次窗口也会随之发生移动，并加载新的地形数据，而旧数据可能会被释放。为了避免不同层级之间出现裂缝，应当在两个连续层级之间使用一个过渡区域。在这个的过渡层级中，几何数据和纹理数据都会平滑地插值到下一个较为粗糙的层级中，这是在顶点着色器和像素着色器中实现的。Asirvatham 和 Hoppe [82]提出了一种高效的 GPU 实现，其中地形数据会被存储为顶点纹理，使用顶点着色器来访问这些数据就可以获得地形的高度信息。可以使用法线贴图来增强地形上的视觉细节，当视角拉近时，Losasso 和 Hoppe [1078]还添加了分形噪声的位移，从而获得更多视觉细节，如图 19.46 所示。Gollent 在《巫师 3》[555]中使用了几何 clipmap 的一种变体。Pangerl [1348]和 Torchelsen 等人[1777]给出了几何 clipmap 的相似方法，这些方法与 GPU 的功能也很契合。

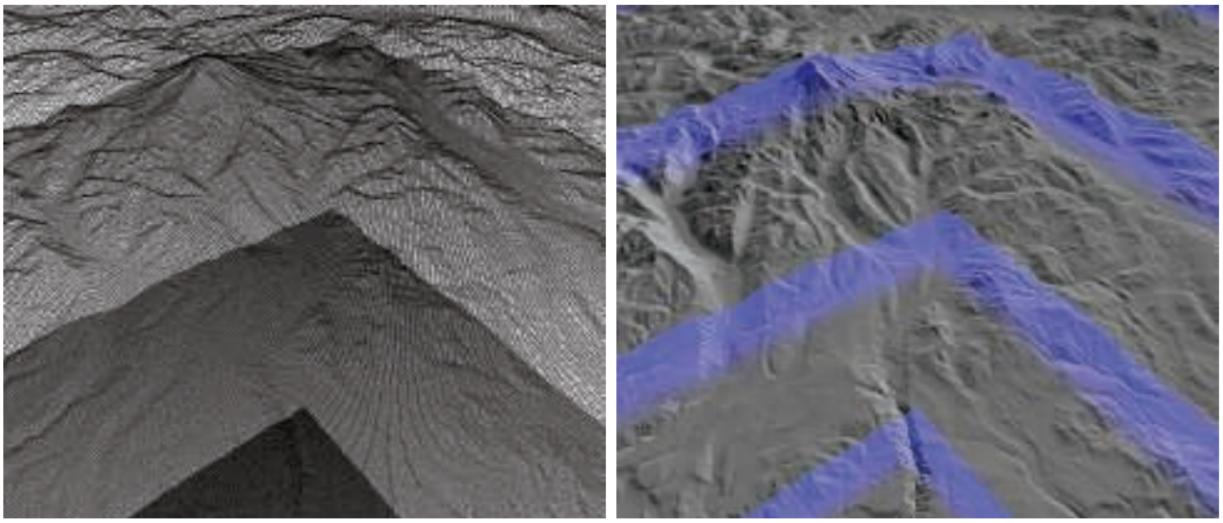


图 19.46：几何 clipmap。左：线框渲染图，不同层级的 mipmap 清晰可见。右：蓝色过渡区域代表了层级之间会发生插值的地方。

有一些方案专注于创建 tile 并对它们进行渲染。一种方法是将高度场数组分解成一定大小的 tile，例如每个 tile 17×17 个顶点。对于一个高度详细的视图，也可以渲染一个单独的 tile，而不是将单个三角形或者小三角形扇发送到 GPU 中。一个 tile 可以有多个 LOD，例如：通过只使用每个方向上的其他顶点，可以形成一个 9×9 大小的 tile。使用每四个顶点可以得到一个 5×5 大小的 tile，使用每八个顶点可以得到一个 2×2 大小的 tile，使用四个角点可以得到一个 1×1 大小的 tile，这个 tile 只包含两个三角形。请注意，原始 17×17 的顶点缓冲区可以存储在 GPU 上并重复使用，只需要提供不同的索引缓冲区来改变需要渲染的三角形数量即可。下面我们将介绍一种使用这种数据布局的方法。

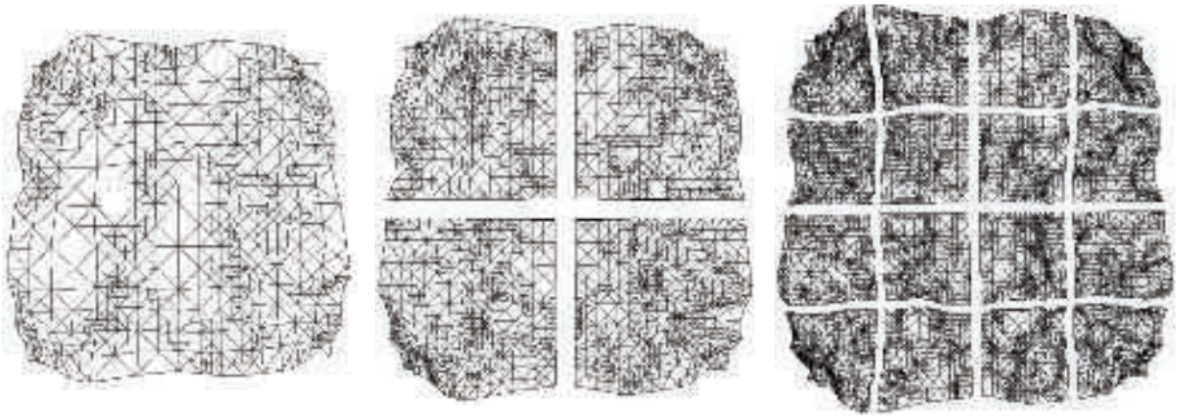


图 19.47：地形的分块 LOD 表示。

另一种在 GPU 上快速渲染大型地形的方法被称为分块 LOD（chunked LOD）[1797]。其核心想法是使用 n 个离散的 LOD 来表示地形，其中每个更加精细的 LOD 会被分割为父 LOD 的 4 倍，如图 19.47 所示。然后将这个结构编码在一棵四叉树中，并从根节点开始遍历来进行渲染。当访问到一个节点的时候，如果它的屏幕空间误差（马上将会介绍）低于某个像素误差阈值 τ，那么这个节点中的 LOD 将会进行渲染。否则，将会递归访问四个子节点。这样可以在一些需要的地方获得更好的分辨率，例如在靠近观察者的地方。在一个更高级的变体方法中，地形四边形会按需从磁盘中进行加载[1605, 1797]。遍历过程与刚才所描述的方法类似，不同之处在于，只有当子节点已经（从磁盘）被加载到内存中时，才会对其进行递归访问。如果这个子节点还没有被加载，那么它将会排队等待加载，并直接渲染当前节点。

Ulrich [1797]将屏幕空间误差计算为：

$$s = \frac{\epsilon w}{2d \tan \frac{\theta}{2}} \tag{19.10}$$

其中 w 是屏幕的宽度， d 是相机到这个地形 tile 的距离， θ 是以弧度为单位的水平视场角， ϵ 是一个几何误差，单位为距离 d 相同。对于其中的几何误差项 ϵ ，通常会使用两个网格之间的 Hausdorff 距离[906, 1605]。对于原始网格上的每个顶点，会找到其在简化网格上距离最近的顶点，并将这些距离中的最小值称为 d_1 。现在反过来对简化网格上的每个顶点执行相同的过程，在原始网格上找到距离最近的顶点，并将其中的最小的距离称为 d_2 。这个 Hausdorff 距离为 $\epsilon = \max(d_1, d_2)$ ，如图 19.48 所示。注意，原始网格上的顶点 $\mathbf{o}$ ，到简化网格的最近顶点是 $\mathbf{s}$ ，而从这个顶点 $\mathbf{s}$ 到原始网格的最近顶点是 $\mathbf{a}$ ，这就是为什么测量必须在这两种组合中进行的原因，即首先会从原始网格到简化网格，然后再从简化网格到原始网格。从直观上来说，Hausdorff 距离是使用简化网格来代替原始网格时的误差。如果应用程序无法计算这个 Hausdorff 距离，则可以为每次简化都手动调整一个误差常数，或者是在网格简化过程中查找误差[1605]。

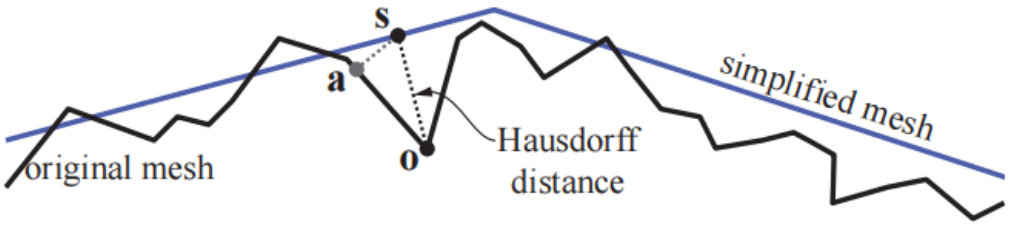


图 19.48：原始网格和简化网格之间的 Hausdorff 距离。[1605]

为了避免从一个 LOD 切换到另一个 LOD 时出现的 popping 现象，Ulrich [1797]提出了一种简单的变形技术，其中一个高分辨率 tile 中的顶点 (x, y, z) 会与一个顶点 (x, y', z) 进行线性插值，这个顶点 (x, y', z) 是父 tile 的近似顶点（例如：使用双线性插值获得）。线性插值因子被计算为 $2s\tau - 1$ ，它会被限制在 $[0, 1]$ 范围内。注意，在变形过程中只需要高分辨率的 tile 即可，因为下一个低分辨率 tile 的顶点已经包含在高分辨率 tile 中了。

可以使用一些启发式的方法，例如方程 19.10 中的启发式方法，来确定每个 tile 所使用的 LOD。这种 tile 平铺的方案，其主要挑战在于对裂缝的修复。例如：假设一个 tile 的分辨率是 33×33 ，而它相邻 tile 的分辨率只有 9×9 ，那么在它们交界的边缘处就会出现裂缝。一种纠正措施是，移除那些沿边缘高度详细的三角形，然后构建一组新的三角形，来适当连接两个 tile 之间的缝隙[324, 1670]。当两个相邻区域具有不同的 LOD 时就会出现裂缝，Ulrich 描述了一种使用额外带状几何的方法，如果设

定的像素误差阈值 τ 低于 5 个像素的话，那么这是一个合理的解决方案。Cozzi 和 Bagnell [300]使用了一个屏幕空间后处理 pass 来填充裂缝，对裂缝周围的片元（而不是裂缝中的片元）使用高斯滤波核进行加权。Strugar [1720]提出了一种优雅的方法来避免出现裂缝，它而无需使用屏幕空间中的方法或是额外的几何图形。其效果如图 19.49 所示，该方法可以使用一个简单的顶点着色器来进行实现。

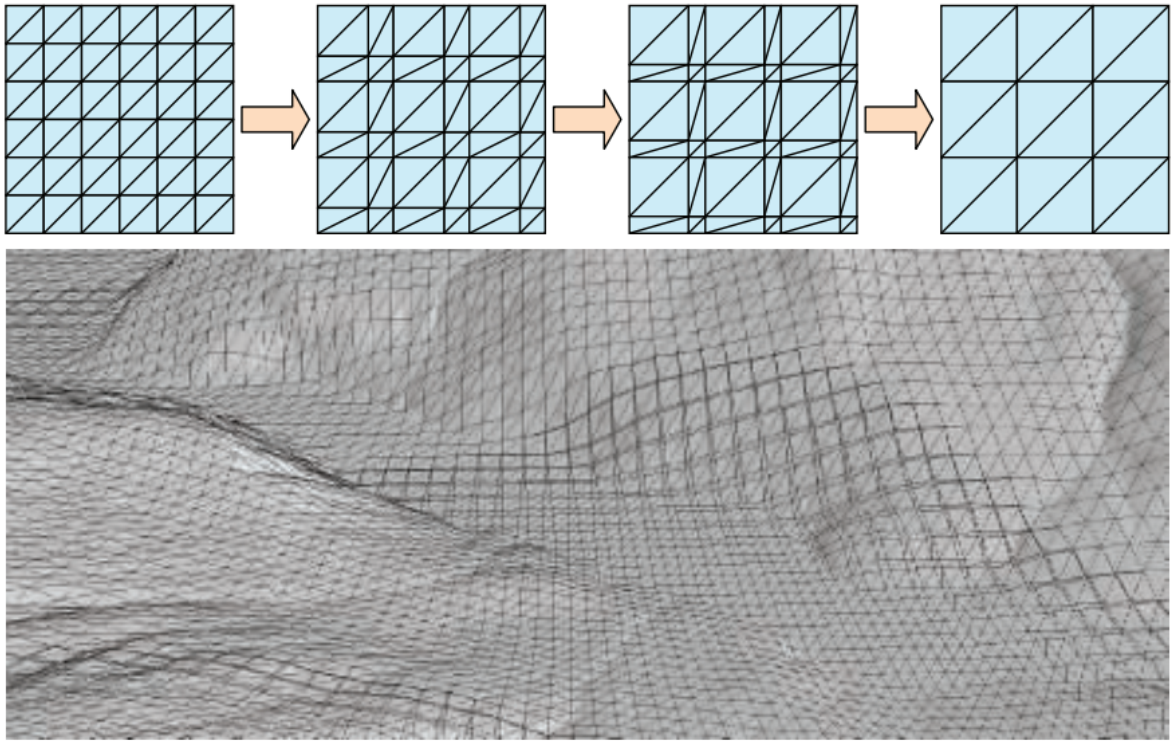


图 19.49：Strugar [1702]使用分块 LOD 系统来避免裂缝。左上角展示了一个高分辨率的地形 tile，它在右上角变成了一个低分辨率的地形 tile。在它们之间，我们展示了两个插值和变形的变体。实际上，随着 LOD 层级的改变，这是以一种平滑的方式发生的，如下面的屏幕截图所示。[1702]

为了提高性能表现，Sellers 等人[1605]将分块 LOD 与视锥体剔除、地平线剔除（horizon culling）结合起来。Kang 等人[852]提出了一种类似于分块 LOD 的方案，其最大的区别在于，他们使用了基于 GPU 的曲面细分来对节点进行细分，并确保边缘处的曲面细分因子可以很好的相互匹配，从而避免出现裂缝。他们还展示了几何图像如何与特征保持的贴图（feature-preserving map）一起，用来渲染地形悬垂结构（overhang），这是基于高度场的地形无法处理的。Strugar [1720]提出了一种分块 LOD 方案的扩展方法，它具有更好和更灵活的三角形分布。与 Ulrich 使用逐节点的 LOD 方法相反，Strugar 使用了逐顶点的变形以及单独的 LOD。虽然他只使用了距离因素作为确定 LOD 选择的度量，但是实际上也可以使用其他一些因素，例如：附近有多少深度变化等，这样可以生成更好的轮廓。

原始的地形数据通常会使用均匀的高度场网格进行表示，我们可以对这些数据使用与一些视图无关的简化方法，如图 16.16 所示。即将网格模型进行简化，直到满足某个极线准则为止[514]。一些较小的表面细节也可以通过颜色贴图或者凹凸贴图进行捕获。以这种方式所生成的静态网格，通常会被称为不规则三角网（triangulated irregular network，TIN），当地形面积较小，且在各个区域相对平坦时，它是一种十分有用的表示方法[1818]。

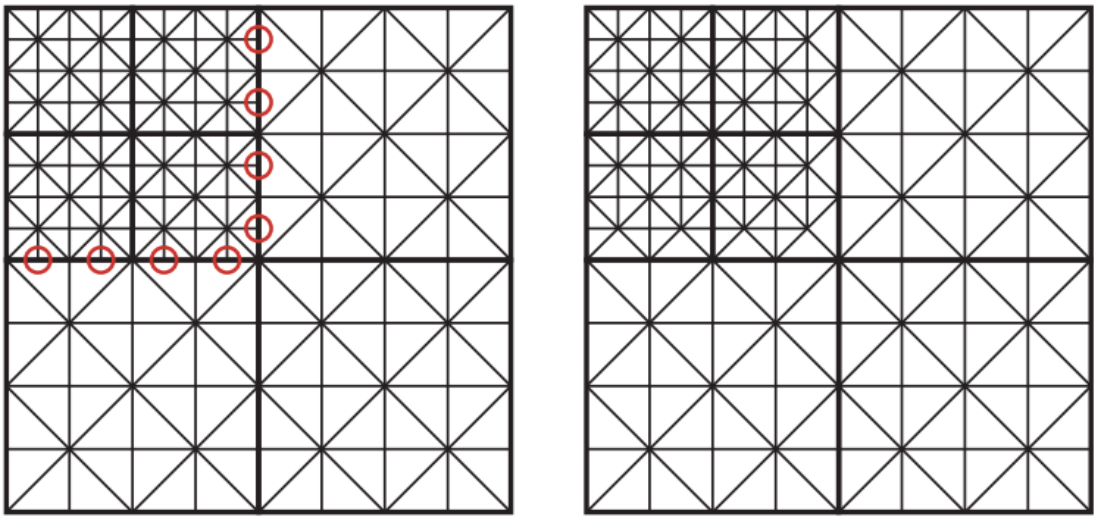


图 19.50：一种地形 tile 的受限四叉树，其中每个相邻 tile 在 LOD 上，最多只能高一级，或者低一级。每个 tile 都有 5×5 个顶点，除了左上角具有 2×2 个更高分辨率的 tile 之外。其余的地形均由三个低分辨率的 tile 进行填充。左侧：左上角 tile 的边缘上有一些顶点，这些顶点与相邻低分辨率 tile 的顶点不匹配，这会导致裂缝的出现。右侧：会对高分辨率 tile 的边缘进行修改，从而避免这个问题。每个 tile 都会在单个 draw call 中进行渲染。[40]

Andersson [40]使用了一种受限四叉树来弥补裂缝，并降低了渲染大型地形所需的 draw call 总数。他并没有使用不同分辨率的均匀地形网格，而是使用了一棵 tile 的四叉树。每个 tile 都具有相同的 33×33 分辨率，但是每个 tile 可以覆盖不同的面积。受限四叉树的理念是，每个 tile 相邻元素之间的 LOD 差异不能超过一个级别，如图 19.50 所示。这一限制意味着相邻 tile 出现分辨率不同的情况是有限的。与其在出现裂缝之后使用额外的索引缓冲区来填充这些裂缝，这里的想法是存储所有可能的索引缓冲区排列方式，这些排列方式可以创建出一个包含裂缝过渡三角形的 tile。每个索引缓冲区都由全分辨率的边缘（每条边缘上有 33 个顶点）和低 LOD 的边缘（由于四叉树的限制，每条边缘上有 17 个顶点）组成。图 19.51 展示了这种现代地形渲染的一个例子。Widmark [1881]描述了一个完整的地形渲染系统，该系统用在了寒霜 2 引擎中，它具有一些十分有用的功能，例如：贴花，水，地形装饰等，同时也可

以使用艺术家生成的或者程序生成的蒙版[40]，来将不同的材质着色器组合在一起，还可以使用程序化地形位移（procedural terrain displacement）。

图 19.51：这个地形渲染中包含了许多不同的 LOD。

有一种简单的技术可以用于海洋渲染，这个技术采用了均匀的网格，每帧都将其转换到相机空间中[749]，转换结果如图 19.52 所示。Bowles [186]针对克服某些质量问题，提供了许多技巧。

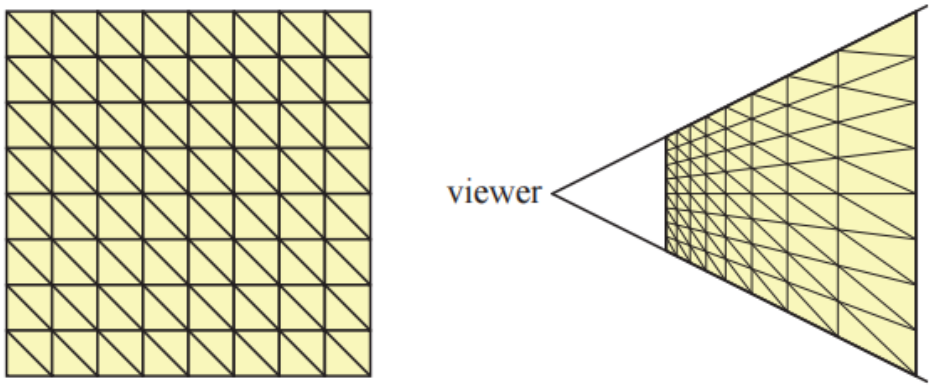


图 19.52：左：一个均匀网格。右：转换到相机空间中的网格。请注意，转换后的网格，在靠近观察者的地方拥有更高分辨率的细节。

除了上述的地形技术之外，还可以使用一些压缩技术，来减少存储在内存中的数据集大小。Yusov [1956]使用四叉树数据结构来对顶点进行压缩，他使用了一种简单的预测方案，只需要对差异进行编码即可（使用很少的 bit）。Schneider 和 Westermann [1956]使用了一种由顶点着色器进行解码的压缩格式，并在细节层次之

间探索集合过渡（geomorphing），同时最大化缓存一致性。Lindstrom 和 Cohen [1051]使用了一种具有线性预测和残差编码（residual encoding）的流式编解码器来进行无损压缩。此外，他们使用量化方案来进一步提高压缩比，尽管这个压缩结果是有损的。这个解压操作可以使用 GPU 完成，可以将压缩比从 $3:1$ 提到 $12:1$ 。

还有许多其他的地形渲染方法。Kloetzli [909]在《文明 5》中使用了一个自定义计算着色器，来为地形创建自适应的曲面细分，然后将其送入 GPU 进行渲染。另一种技术是使用 GPU 的曲面细分器[466]，来逐面片的处理曲面细分。请注意，许多用于地形渲染的技术同样也可以用于水面渲染。例如：GonzalezOchoa 和 Holder [560]在《神秘海域 3》中，使用了一种几何 clipmap 的变体方法，它可以很好地渲染水面效果。他们通过在不同层级之间动态添加三角形来避免 T 型连接。随着 GPU 的不断发展，有关这个主题的研究仍将继续。

补充阅读和资源

虽然 Ericson 这本书[435]的重点是碰撞检测，但是其中包含了构建和使用各种空间细分方案的相关材料。

有着大量有关遮挡剔除的文献。其中 Cohen–Or 等人[277]和 Durand [398]的可见性调查，可以帮助你很好地了解这些算法的早期工作。Aila 和 Miettinen [13]描述了一个用于动态场景的商业剔除系统架构。Nießner 等人[1283]对现有的背面剔除、视锥体剔除、以及位移细分表面的遮挡剔除方法进行了综述。Luebke 等人[1092]的《Level of Detail for 3D Graphics》一书，是一个有关 LOD 使用的信息资源，具有一定的价值。

Dietrich 等人[352]对渲染大规模模型领域的研究进行了综述。Gobbetti 等人[547]提供了另一个有关大规模模型渲染的良好综述。Sellers 等人[1605]的 SIGGRAPH 课程是一个比较新的资源，其中包含了一些优秀的材料。Cozzi 和 Ring 所撰写的书[299]介绍了地形渲染和大规模数据集管理的技术，以及如何处理精度问题的方法。Cesium 的博客[244]提供了许多实现细节和进一步的加速技术，可以用于大世界渲染和地形渲染。

Chapter 20 Efficient Shading 高效着色

> Mark Twain——“Never put off till tomorrow what you can do the day after tomorrow just as well.”

> 马克·吐温——“后天能做的事也不要拖到明天。”（今日事今日毕）（美国作家；1835——1910）

对于比较简单的场景（即相对较少的几何图形、比较基础的材质、少量的光源），我们可以使用标准的 GPU 管线来渲染图像，而且并不用担心帧率的问题。只有当其中的一个或者多个元素变得成本很高时，我们才需要使用一些更加复杂的技术来控制成本。在上一章节中，我们所关注的是在后续处理中如何剔除那些不需要的三角形和网格。而在本章节中，我们将专注于介绍在计算材质和光照时能够降低成本的技术。对于其中的许多方法，都有着额外的处理成本，我们希望这些额外成本能够由节省下来的时间所弥补。另一些方法会在带宽和计算之间进行权衡，通常会转移性能瓶颈。与所有的方案一样，具体使用哪种方案最好，这取决于硬件、场景结构和许多其他因素。

计算材质效果的像素着色器可能会有很高的成本。我们在章节 19.9 中提到过，可以通过各种各样的着色器 LOD 简化技术来减少这种开销。当场景中存在多个光源同时对一个表面产生影响的时候，可以使用两种不同的策略。第一种方法是构建一个支持多光源的着色器，这样只需要一个 pass 即可。另一种方法是使用多个 pass 来进行着色，我们为光源创建一个单一光源的像素着色器并对其进行评估计算，将每个计算结果都添加到帧缓冲中。这样一来，对于场景中的 3 个光源，我们将会绘制 3 次图元，在每次计算的时候都会变更当前使用的光源。第二种方法总体上要比单 pass 系统更加高效，因为每个 pass 所使用的着色器都更加简单、执行速度更快。但是如果一个渲染器中存在着许多不同类型的光源，那么这个单 pass 的像素着色器就必须要能够包括并处理这些光源，并依次测试该类型的光源是否被使用，这会产生一个十分复杂的着色器。

在章节 18.4.5 中，我们讨论了通过最小化或者完全消除过度绘制，从而避免不必要的像素着色器计算。如果我们能有效确定一个表面对最终图像没有任何贡献，那么我们

就可以节省花费在其上进行着色计算的时间。一种方法是执行一个 z-prepass，即预先渲染不透明的几何物体，这个 pass 中只会写入 z-depth。然后会再次渲染场景中的几何物体，这次会进行完整的着色计算，会使用第一个 pass 生成的 z-buffer 来剔除所有不可见的片元。这种类型的 pass 试图将寻找可见几何物体的过程，与随后对几何物体的着色操作分离开来。将这两个过程分开进行的想法也是贯穿本章的一个重要概念，它被几种不同的渲染方案所采用。

使用一个 z-prepass 的一个问题是，我们必须要对场景中的几何图形渲染两次。与正常的渲染相比，这是一项额外的开销，并且这个开销可能要比节省下来的时间更长。如果网格是通过曲面细分、蒙皮或者其他相关过程形成的，那么添加这个 pass 所带来的成本可能会相当大[992, 1177]。对于具有镂空 alpha 值的物体而言，需要在每个 pass 中检索纹理中对应的 alpha 值，这会带来额外的开销，因为在第一个 pass 中进行渲染的时候，我们其实并不知道这个片元的 alpha 值，或者我们也可以在第一个 pass 中完全忽略这类物体，仅在第二个 pass 来渲染它们，这可能会导致浪费一些像素着色器的计算。由于这些原因，有时我们会在初始 pass 中，只渲染那些体积较大（屏幕空间或者世界空间中）的遮挡物。其他一些屏幕空间中的效果（例如环境光遮蔽或者屏幕空间反射）[1393]，也可能会需要执行一个完整的 prepass。本章节中介绍的一些加速技术，也需要精确的 z-prepass 来帮助剔除光源列表。

即使没有过度绘制问题，想要为一个可见表面计算大量的动态光源也会产生十分可观的成本。假设一个场景中包含 50 个光源，一个多 pass 的渲染系统虽然可以成功地渲染这个场景，但是每个物体都需要 50 个 pass 来进行着色。一种降低成本的技术是，将每个局部光源的效果都限制在一定半径的球体内部、某个高度的圆锥体内、或者是其他的一些有限形状内[668, 669, 1762, 1809]，我们假设每个光源的贡献在一定距离之后都会衰减得微不足道。在本章节的剩余部分中，我们会将光源的体积称为球体，这样做是为了便于理解，在理解背后的原理之后，当然也可以使用其他的一些几何形状。通常光源的强度会被用作决定其影响半径的唯一因素。Karis [860]讨论了光泽镜面材质的存在将会如何增加这个半径，因为这样的表面会更容易受到光线的影响。对于非常光滑的表面而言，这个距离可能会无穷大，因此可能需要使用环境贴图或者其他的一些技术。

可以进行一个简单的预处理操作，即为每个网格模型都创建一个能够影响它的光源列表。我们可以把这个过程看作是在网格和光源之间执行碰撞检测，从而找到那些可能发生重叠的部分[992]。在对网格进行着色计算的时候，我们将会使用这个光源列表，从而减少应用在其表面上的光源数量。这种方法存在一些问题，如果物体或者光源发生了移动，那么这些变化可能会影响列表中的具体内容。我们在前几章中提到，

为了提高性能，通常我们会将共享相同材质的几何物体，合并到一个更大的网格中（章节 18.4.2），这可能会导致单个网格在其列表中，包含了场景中的部分甚至全部光源[1327, 1330]。也就是说，网格可以被合并，然后在空间上进行划分，从而提供更短的光源列表[1393]。

另一种方法则是将静态光源烘焙到世界空间的数据结构中。例如：在《正当防卫 2》的光照系统中，存在一个世界空间中自上而下的网格，这个网格存储了场景中的光照信息。一个网格单元代表了一个 $4m \times 4m$ 的区域。每个单元格都会存储在 $\mathrm{RGB}\alpha$ 纹理的一个纹素中，因此这个列表最多可以包含 4 个光源。在渲染一个像素的时候，会获取其所在区域的光源列表，并在着色计算中应用这些光源[1379]。这种方法的缺点在于，影响一个给定区域的光源数量会有一个固定的存储上限。这种方法对于精心设计的户外场景可能会有用，但是多层建筑就不适合使用这种存储方案了。

我们的目标是以一种高效的方式来处理动态网格和动态光源，同样重要的还有可预测的性能，即观察视角或者场景中的一些小变化，不会导致渲染成本发生剧烈波动。《毁灭战士（2016）》中的某些关卡拥有 300 多个可见光源[1682]；而《奇点灰烬》中的一些场景中则会包含多达 1 万个可见光源，如图 20.1 和图 20.15 所示。在一些渲染器中，大量的粒子也可以被视为微小光源。其他的一些技术使用光照探针（章节 11.5.4）来照亮附近的表面，这些光照探针可以被认为是一种近程（short-range）光源。

图 20.1：复杂的光照情况。注意肩膀上的微小光源、以及建筑结构上的每个亮点，这些亮点实际上都是光源。右上角远处的点光同样也是光源，在那个距离它们会被渲染为点状的 sprite。

[1387]

20.1 延迟着色

到目前为止，在本书中我们已经介绍了前向着色（forward shading），在这种着色方法中，每个三角形都会被发送到管线中，最后屏幕上的图像被更新为这个三角形的着色值。延迟着色（deferred shading）背后的想法是：在执行任何的材质光照计算之前，需要首先执行所有的可见性测试以及表面属性的评估计算。这个概念于 1988 年首次在硬件架构中引入[339]，后来作为实验性 PixelFlow 系统[1235]中的一部分，并作为一种离线的软件解决方案，并通过图像处理来帮助生成非真实感风格的图像[1528]。Calver 在 2003 年中的大量文章中[222]，详细阐述了在 GPU 上使用延迟着色的基本思想。Hargreaves、Harris [668, 669]和 Thibieroz [1762]则在第二年推广了延迟着色的使用，在当时，同时写入多个渲染目标（multiple render target，MRT）的能力变得越来越普及。

在前向着色中，我们使用一个着色器和一个代表物体的网格，执行单个 pass 来计算最终图像。这个 pass 首先会获取材质属性：常量、插值参数或者纹理值，然后对这些值来应用一组光源。前向渲染的 z-prepass 方法，可以看作是几何渲染和着色计算的轻度解耦（mild decoupling），因为第一个几何 pass 的目的只是为了确定可见性，而所有的着色工作（包括材质参数的检索），都会被推迟到第二个几何 pass 中进行，第二个 pass 才会对所有的可见像素进行着色。对于交互式渲染而言，延迟着色意味着与可见物体相关的所有材质参数，都由这个初始的几何 pass 进行生成和存储；然后再使用一个后处理，来将光源应用于这些存储下来的表面值。存储在第一个 pass 中的参数值包括：位置（存储为 z-depth）、法线、纹理坐标和各种材质参数等。这个 pass 为像素建立了所有的几何信息和材质信息，因此在之后的 pass 中，我们就不再需要几何物体了，也就是说，模型几何的贡献已经完全与光照计算解耦了。请注意，在这个初始 pass 中也可能会发生过度绘制，但是不同之处在于，这个 pass 中着色器所执行的计算（将值转移到缓冲区中）要比评估一组光源对材质的影响少得多。在前向着色中，有时 2×2 四边形中的所有像素可能都不在三角形边界的内部，但是所有像素都必须完全进行着色[1393]（章节 23.8）。这听起来像是一个很小的影响，但是想象现在有这样的一个网格，其中的每个三角形都只覆盖了一个像素，这将会生成四个完整着色的样本，如果使用前向着色的话，其中有三个会被丢弃。而使用延迟着色，每次着色器调用的成本更低，因此那些被丢弃的样本只会产生很小的影响。

用于存储表面属性的缓冲区通常被称为 G-buffer [1528]，这是“几何缓冲区（geometric buffer）”的缩写。这种缓冲区有时也称为 deep 缓冲区，尽管这个术语也可以指每个像素存储了多个表面（片元）的缓冲区，但是我们在这里还是避免使用它。图 20.2 展示了一些 G-buffer 的典型内容。G-buffer 可以存储程序员想要存储的任何东西，即完成后续光照计算所需要的任何东西。每个 G-buffer 都是一个单独的渲染目标（render target），通常一个 G-buffer 会包含 3 到 5 个渲染目标，但是在一些系统中，G-buffer 的渲染目标已经达到 8 个之多[134]。使用更多的渲染目标会带来更大的带宽压力，这就增加了 G-buffer 成为性能瓶颈的可能性。

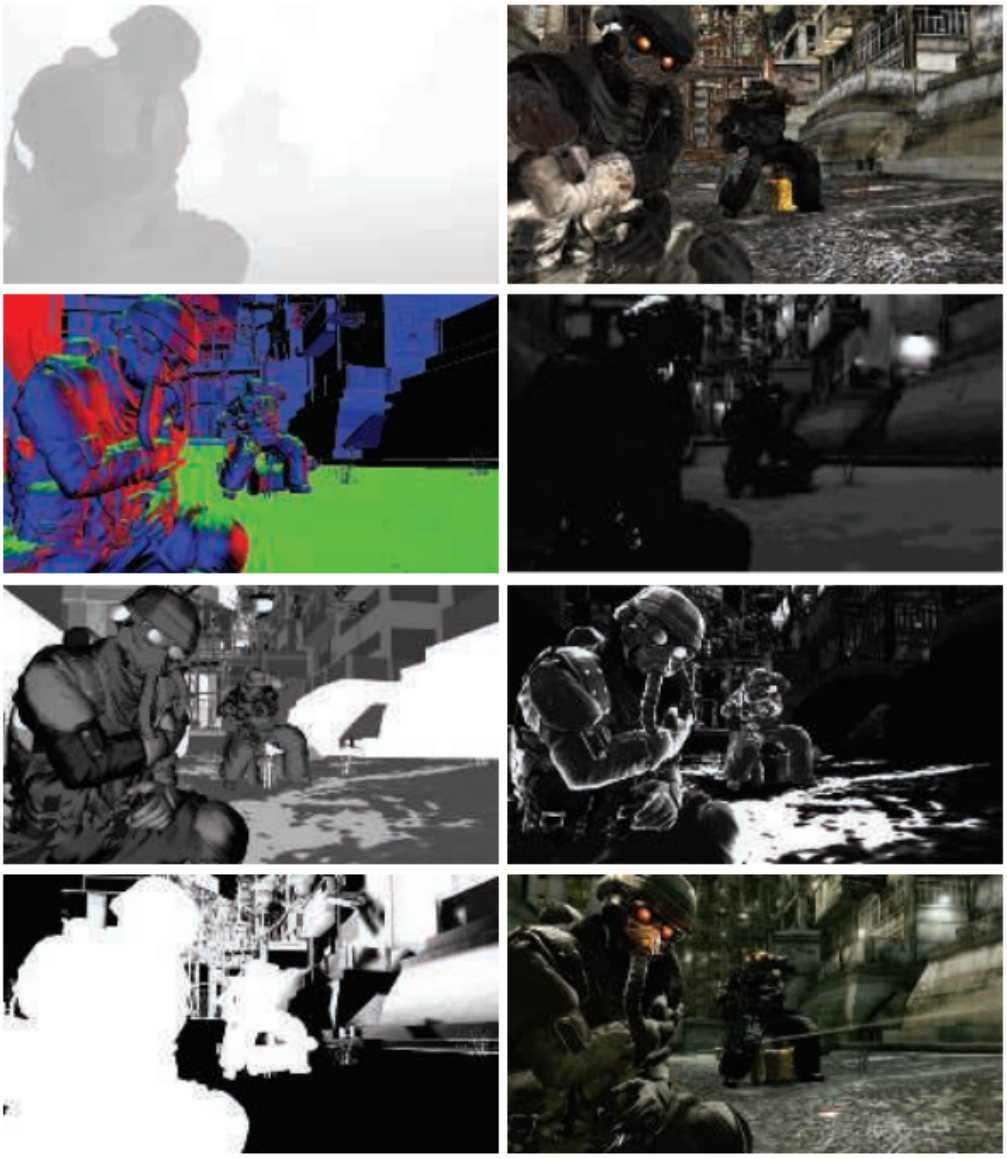


图 20.2：用于延迟着色的 G-buffer，在一些情况下可以将其转换为颜色输出，用于可视化、调试等目的。左边一列从上到下分别是：深度缓冲，法线缓冲，粗糙度缓冲和阳光遮挡（sunlight occlusion）。右边一列从上到下分别是：纹理颜色（又称反照率 albedo 纹理），光照强度，高光强度，和接近最终的图像（没有运动模糊）。[1809]

在第一个 pass 创建 G-buffer 之后，会使用一个单独的过程来计算光照效果。其中一种方法是依次应用每个光源，并使用 G-buffer 来计算其效果。对于每个光源，我们会绘制一个屏幕填充的四边形（章节 12.1），并将 G-buffer 作为纹理来进行访问[222, 1762]。在每个屏幕像素上，我们首先可以确定最近的表面位置，以及这个位置是否位于光源的范围内。如果该位置位于光源范围内，那么我们就计算其光照效果，并将结果放置在输出缓冲区中。我们会依次为每个光源都执行这个相同的过程，并通过混合来添加这个光源的贡献。最后，我们就获得了所有光源的贡献。

上述这个过程是使用 G-buffer 最为低效的方式，因为每个存储下来的像素参数，都是逐光源进行访问和计算的，这个过程其实类似于基本的前向渲染，即将所有的光源应用于所有的表面片元上。由于写入和读取 G-buffer 所带来的额外成本，这种方法最终可能要比前向着色更慢[471]。作为提高性能的第一步，我们可以确定一个光源体积（例如一个球体）的屏幕包围框，并使用这个包围框来对一个覆盖图像较小部分的屏幕空间四边形区域进行光照计算[222, 1420, 1766]，这样一来，像素处理的次数就大大减少了。绘制一个代表球体的平面椭圆形状，可以进一步对光源体积外的像素处理进行裁剪[1122]。同时，我们也可以使用第三个屏幕维度，即 z-depth。通过绘制一个包含体积的粗糙球体网格，我们可以对这个光源球体的效果区域进行进一步的裁剪[222]。例如：如果这个球体被深度缓冲所遮挡，那就说明这个光源体积位于最近表面的后面，因此不会对画面产生影响。一般来说，如果一个球体在某个像素上的最小深度和最大深度，不会与 G-buffer 中的最近表面相互重叠，那么这个光源就不会对该像素产生影响。Hargreaves [668]和 Valient [1809]针对如何高效且正确地确定这种重叠，讨论了各种选项和注意事项，以及其他的一些优化方法。我们将在前面的几个算法中，也看到了对表面和光源之间的深度重叠进行测试的想法。具体哪种方法的效率最高，则要取决于具体情况。

对于传统的前向渲染，顶点着色器和像素着色器会检索每个光源的参数和每个材质的参数，并计算光源对于材质的影响。前向着色需要一个复杂的顶点着色器和一个复杂的像素着色器，它们需要能够覆盖所有可能的材质和光源组合，或者使用一些专门的着色器来处理特定的组合。具有动态分支的较长着色器，通常会运行得十分非常缓慢[414]，因此大量较短的着色器可能会更加高效，但是这也意味着需要更多的工作来生成和管理这些着色器。在前向渲染中，由于所有的着色功能都是在一个 pass 中完成的，因此在渲染下一个物体的时候，很有可能需要改变所使用的着色器，这种着色器切换也会导致一定的效率下降（章节 18.4.2）。

延迟渲染方法允许光照和材质定义之间的高度分离，每个着色器都专注于参数提取或者光照计算，而不是同时专注于这两者。长度较短的着色器可以运行得更快，这既是

因为着色器的长度，也是由于可以对这些较短的着色器进行更好地优化。着色器中所使用的寄存器数量决定了占用率（章节 23.3），这个占用率是一个关键因素，它代表了有多少的着色器实例可以并行运行。这种光照和材质的分离也简化了着色器系统的管理，例如：这种分离使得材质测试和光照测试变得更加容易，因为我们只需要为一个新的光源或者新的材质，添加一个新的着色器到系统中即可，而不是每个光源与材质的组合都添加一个新的着色器[222, 927]。这当然是可能的，因为材质评估是在第一个 pass 中完成的，然后在第二个 pass 中，会将光照应用于存储下来的表面参数集合。

对于单 pass 的前向渲染，所有阴影贴图通常必须同时可用，因为所有光源的光照评估都是在一次计算中完成的。由于可以在单个 pass 中完整处理每个光源，因此延迟着色允许在同一时间内，在内存中只有一个阴影贴图[1809]。然而，我们稍后会介绍一些更加复杂的光源分配方案，在这种方案中，就没有这个单一阴影贴图的优势了，因为光源是按组进行评估的[1332, 1387]。

最基本的延迟着色只能支持一个具有固定参数集合的单一材质，这实际上限制了所能表现的材质模型。想要支持不同的材质描述，一种方法是在某个给定字段中，存储每个像素的材质 ID 或者掩码[414, 667, 992, 1064]，然后着色器可以根据 G-buffer 中的内容来执行不同的计算。这种方法还可以根据 ID 或者掩码值，来修改存储在 G-buffer 中的内容。例如：一种材质可能会使用 32 bit 来存储 G-buffer 中的第二层颜色和混合因子，而另一种材质可能会使用这些相同的 32 bit，来存储它所需要的两个切向量。但是这些方案需要使用更加复杂的着色器，这可能会对性能产生影响。

最基本的延迟着色还有一些其他的缺点。G-buffer 对于显存的要求可能会非常大，重复访问这些缓冲区也会带来较大的带宽成本[856, 927, 1766]。我们可以通过降低存储精度或者压缩数据来降低这些成本（显存成本和带宽成本）[1680, 1809]，图 20.3 展示了一个压缩例子。在章节 16.6 中，我们讨论了对网格世界空间数据的压缩。G-buffer 可以包含世界空间中或者屏幕空间中的坐标值，具体取决于渲染引擎的需要。Pesce [1394]讨论了对于 G-buffer，压缩屏幕空间坐标与压缩世界空间法线之间的权衡，并提供了指向相关资源的链接。法线的世界空间八面体映射是一种常见的压缩方案，该方案具有精度高、编解码时间短等优点。

	R8	G8	B8	A8
RT0	world normal (RGB10)			GI
RT1	base color (sRGB8)			config (A8)
RT2	metalness (R8)	glossiness (G8)	cavity (B8)	aliased value (A8)
RT3	velocity.xy (RGB8)			velocity.z (A8)

图 20.3：一个可能的 G-buffer 布局，在《彩虹六号：围攻》中使用了这种布局方法。除了深度缓冲和模板缓冲之外，还额外使用了四个渲染目标（render target，RT）。从图中可以看到，任何数据都可以放入这些缓冲区中。其中 RT0 中的“GI”字段为“GI 法线偏置（A2）”。[415]

延迟着色对两个重要的技术有一些限制，即透明度渲染和抗锯齿技术。在基本的延迟着色系统中并不支持透明度效果，因为在每个像素位置上只能存储一个表面坐标。一种解决方案是在使用延迟着色渲染不透明表面之后，再使用前向着色来渲染透明物体。对于早期的延迟渲染系统而言，这意味着对于场景中的每个透明物体，都需要应用全部的光源，这是一个开销很大的过程，因此必须进行一些简化。正如我们将要在接下来章节中探索的那样，不断改进 GPU 功能，使得延迟着色和前向着色的光源剔除方法也在不断的发展。虽然现在可以在像素位置处存储对应的透明表面列表[1575]，并使用一个纯延迟的方法，但是正常的做法是根据透明度和其他所需的效果，来混合使用延迟着色和前向着色[1680]。

前向着色的一个优点是，它容易支持 MSAA 等抗锯齿方案。对于 $N\times$ MSAA 算法，前向着色只需要在每个像素位置上存储 N 个深度样本和颜色样本即可。虽然延迟着色也可以将每个像素位置上的 N 个样本都存储在 G-buffer 中，从而实现 MSAA 抗锯齿，但是由于内存成本、填充率和以及额外的计算开销，使得这种方法过于昂贵，难以实现[1420]。为了克服这个限制，Shishkovtsov [1631]使用了一种边缘检测的方法，来对边缘覆盖率计算进行近似。其他的形态学后处理方法，也可以用于抗锯齿（章节 5.4.2），时域抗锯齿同样可以使用[1387]。一些用于延迟着色的 MSAA 方法，会通过检测哪些像素或者 tile 处存在边缘，从而避免对每个样本都进行着色计算[43, 990, 1064, 1299, 1764]。只有那些存在边缘信息的像素位置，才需要对多个样本进行评估。Sousa [1681]在这种类型方法的基础上，使用一个模板来识别那些需要更复杂处理的多样本像素。Pettineo [1407]描述了一种追踪这些像素的新方法，他使用一个计算着色器，来将边缘像素移动到线程组内存的一个列表中，从而实现高效的流式处理。

Crassin 等人[309]的抗锯齿研究侧重于高质量结果，并对该领域的其他研究进行了总结。在他们的方法中，首先会执行一个深度 prepass 和法线几何 prepass，并将相似的子样本分组在一起。然后再生成 G-buffer，并对每组子样本的最佳值进行统计分析。然后会使用这些深度界限值，来对每组样本进行着色，最终将着色结果混合在一起。虽然在本文写作的时候，这种处理方式想要达到交互式的速率对于大多数应用而言都是不切实际的，但这种方法能够让我们了解到，用于改善图像质量的到底需要多大的计算能力才能够实现。

即使存在这些限制，延迟着色在商业程序中也是一种十分实用的渲染方法。它能够将几何形状与着色分开，将光照与材质分开，这意味着每个元素都可以进行专项优化。其中有一个特别有趣的领域是贴花渲染（decal rendering），它可以对任何渲染管线产生影响。

20.2 贴花渲染

贴花（decal）是指将一些设计元素（例如图片或者其他纹理）应用在表面上。在电子游戏中，贴花经常会以轮胎印、弹孔或者喷在表面上的玩家标签等形式出现。贴花在其他应用程序中则用于 logo、注释或者其他一些内容，例如：对于地形系统或者城市场景而言，贴花可以让艺术家通过对细节纹理进行分层处理、或者以不同的方式来重新组合各种图案，从而避免明显的重复现象。

贴花可以以多种方式来与底层材质进行混合，它可能只会对底层材质的颜色进行修改，同时并不会影响材质本身的凹凸贴图，就像纹身一样。或者，它也可以部分代替材质原本的凹凸贴图，例如浮雕 logo。它也可以定义一种完全不同的材质，例如：在车窗上贴一张贴纸。多个贴花可能会应用于同一个几何图形，例如路径上的脚印等。同时一个贴纸可能会跨越多个模型，例如地铁车厢表面上的涂鸦。这些不同的贴花方法，对于前向着色系统和延迟着色系统如何存储和处理贴花存在一些影响。

首先，贴花必须像其他纹理一样映射到表面上。由于每个顶点上可以存储多个纹理坐标，因此可以将多个贴花绑定到单个表面上。但是这种方法是有限制的，因为每个顶点可以存储的纹理坐标数量相对较少，每个贴花都需要使用一组独立的纹理坐标。在同一个表面上应用大量的小贴花，这意味着即使每个贴花只会对网格中少数几个三角形产生影响，我们也需要在每个顶点上都存储各个贴花所对应的纹理坐标。

为了将贴花渲染在网格上，一种方法是让像素着色器对每个贴花进行采样，并将一个贴花混合在另一个贴花之上。这会使得着色器变得十分复杂，如果贴花的数量会随着时间变化，那么还可能需要对着色器进行频繁地重新编译，或者采取一些其他措施。另一种可以使得着色器独立于贴花系统的方法是，为每个贴花再次渲染一遍网格，并对每个 pass 进行分层，将后续 pass 的结果混合在前一个 pass 上。如果某个贴花很小，只会跨越少数几个三角形，那么可以创建一个单独的、更短的索引缓冲区，来渲染这个贴花的子网格。另一种贴花方法是对材质的纹理进行修改。如果只在一个大型网格上应用贴花（例如一个地形系统），那么修改这个纹理其实就是提供了一个简单的“set it and forget it”的解决方案[447]，即标记哪些地方存在贴花。如果这个材质纹理会在多个物体上进行使用，那么我们需要创建一个新的纹理，这个新纹理会将材质和贴花组合在一起。这种烘焙解决方案避免了着色器的过度复杂和以及过度绘制，

但是其代价是需要对纹理进行管理，以及一些额外的内存开销[893, 1393]。这种单独渲染贴花是一种规范做法，因为可以将不同分辨率的贴花应用在相同的表面上，并且这个基础纹理可以被重复使用，无需在内存中添加额外的修改副本。

这些解决方案对于计算机辅助设计来说是比较合理的，用户可以在其中添加单个 logo 和少量其他内容。这些方法也可以应用于动画模型的贴花效果，其中这些贴花需要在变形之前进行投影，从而使得它们能够像底层物体一样被拉伸。然而，对于存在多个贴花的应用场景而言，这些技术就会变得十分低效和繁琐。

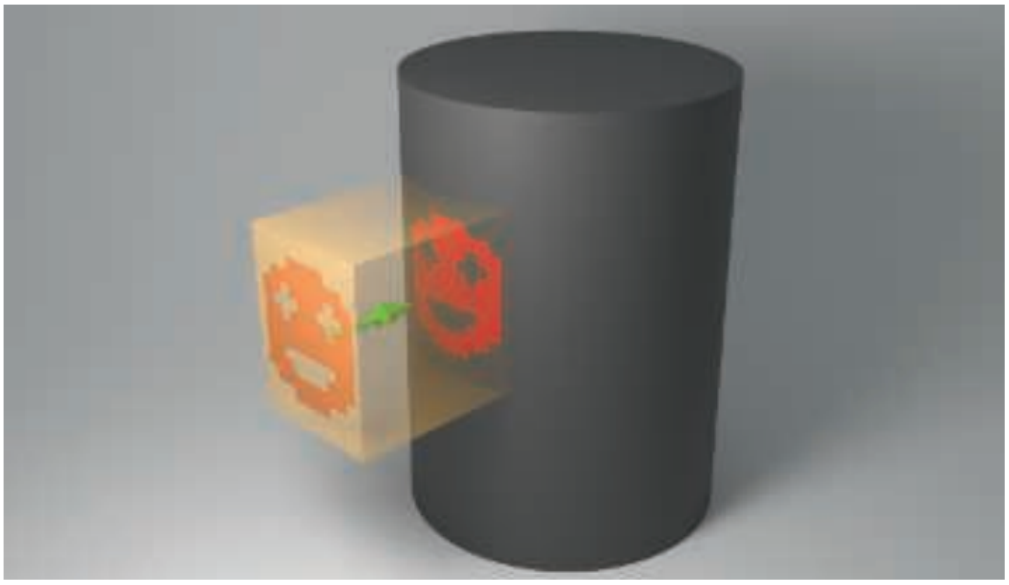


图 20.4：一个 box 定义了一个贴花投影，并且这个贴花会应用在该 box 内的表面上。图中的 box 具有一个夸张的厚度，从而展示这个投影过程及其效果。在实践中，这个 box 会被做得尽可能薄，并且紧贴表面，从而尽量减少在贴花应用过程中需要测试的像素数量。

对于静态物体或者刚性物体，一种流行的解决方案是，将贴花视为一个通过有限体积进行正投影的纹理[447, 893, 936, 1391, 1920]。会将一个定向 box 放置在场景中，贴花从 box 的其中一个面投影到另一个对面，就像一个电影放映机一样，如图 20.4 所示。这个 box 的表面会被光栅化，从而驱动像素着色器的执行。在这个体积内发现的任何几何形状，都会将贴花应用在其材质上。这是通过将几何表面的深度和屏幕位置，转换为体积中的位置来实现的，然后会为贴花提供一个 (u, v) 纹理坐标。或者，贴花也可以是一个真正的体积纹理[888, 1380]。通过指定 ID [900]、指定模板 bit [1778]或者依赖于渲染顺序，可以使得贴花只会对体积中的部分物体产生影响。它们也经常会根据表面法线和投影方向之间的角度，来进行渐隐和 clamp，从而避免在表面接近边缘的地方出现贴花拉伸或者扭曲现象[893]。

延迟着色十分擅长渲染这样的贴花效果。与标准的前向着色相比，延迟着色不需要对每个贴花都单独进行光照和着色，这些贴花的效果可以应用到 G-buffer 中。例如：如果一个代表轮胎痕迹的贴花取代了地面上的着色法线，那么这些修改将会直接在 G-buffer 中的对应位置处进行更改。之后我们仅仅使用 G-buffer 中的数据进行光照着色即可，从而避免了前向着色中会发生的过度绘制现象[1680]。由于贴花效果可以完全由 G-buffer 进行捕获存储，因此在着色期间也就不需要贴花了。这种整合还避免了多 pass 前向着色中的一个问题，即一个 pass 的表面参数，可能会对另一个 pass 的光照或者着色产生影响[1380]。这种简洁特性，也是决定寒霜 2 引擎从前向着色切换到延迟着色的主要因素之一[43]。贴花可以被认为与光源是一样的，因为二者都是通过渲染一个空间体积，来确定其在内部表面上的效果。正如我们将在章节 20.4 中所看到的，通过使用这一事实，一种改进形式的前向着色方法也可以获得类似的效率提升，以及其他的一些优势。

Lagarde 和 de Rousiers [960]对延迟着色中贴花的几个问题进行了描述。贴花的混合方式，仅限于管线中合并阶段可以使用的混合操作[1680]。如果材质和贴花都具有法线贴图，那么想要实现适当的混合结果可能会很困难，如果使用一些凹凸纹理过滤技术的话，那就更加困难了[106, 888]。正如章节 6.5 所述，这样做可能会出现黑白条纹状的瑕疵。诸如符号距离场之类的技术，可以用来精准分割这些材质[263, 580]，尽管这样做可能会导致一些锯齿问题。另一个值得关注的问题是贴花的轮廓边缘，这是由于屏幕空间中的信息投影回世界空间中所产生的梯度误差造成的。一种解决方案是限制或者忽略此类贴花的 mipmap，Wronski [1920]针对这一问题，讨论了更加详细的解决方案。

贴花可以用于动态元素，例如刹车痕或者弹孔，但也可以用来给不同的位置赋予一些变化。图 20.5 展示了一个将贴花应用于建筑墙壁和其他地方的场景。其中的墙壁纹理可以重复使用，而贴花则负责提供定制化的细节，从而使得每个建筑都具有独特的特征。

图 20.5：在第一行的图像中，将颜色贴花和凹凸贴花所覆盖的区域展示为棋盘图案。第二行展示的是没有贴花的建筑。第三行图像中的建筑大约应用了 200 个贴花。

20.3 分块着色（Tiled Shading）

在最基础的延迟渲染中，每个光源的效果都会被单独计算，并将结果被添加到输出缓冲区中。这是早期 GPU 的一个功能特性，即由于着色器复杂性的限制，想要同时计算多个光源是不可能的。延迟着色可以处理任意数量的光源，但是每次都需要访问 G-buffer。当场景中存在成百上千个光源的时候，这个基础的延迟渲染也会变得十分

昂贵，因为场景中的所有光源都需要对每个重叠的像素进行处理，并且每个像素上的每个光源，都会涉及到一次单独的着色器调用。在单个着色器调用中计算多个光源的效率更高。在接下来的几个小节中，我们将讨论在延迟着色和前向着色中，如何以交互式速率来快速处理大量光源的几种算法。

多年来，各种混合 G-buffer 的系统被不断开发出来，它们尝试在材质和光源存储之间取得平衡。例如：想象一个具有漫反射项和高光项的简单着色模型，其中材质的纹理仅会对漫反射项产生影响。我们可以先分别计算每个光源的漫反射项和高光项，然后将这些结果存储下来，而不是针对每个光源都从 G-buffer 中检索对应的纹理颜色。这些累计下来的项会在基于光源的 G-buffer 中被叠加在一起，它有时候也会被称为 L-buffer。最后，我们只需要检索一次纹理颜色即可，将其乘以漫反射项，然后再添加镜面项。

在这个过程中，纹理的效果会被排除在方程之外，因为它只会对所有光源使用一次。通过这种方式，每个光源所访问的 G-buffer 数据点会更少，从而节省了带宽。一个典型的存储方案是累积漫反射颜色和镜面强度值，这意味着这四个值可以通过叠加混合（additive blending）来输出到一个缓冲区中。Engel [431, 432]讨论了几种早期的延迟光照技术，也被称为预照明（pre-lighting）或者光照 prepass 方法。Kaplanyan [856]对不同方法进行了比较，旨在最小化 G-buffer 的存储和访问。Thibieroz [1766]还强调了位深较浅的 G-buffer，并对几种算法的优缺点进行了对比分析。Kircher [900]描述了如何使用较低分辨率的 G-buffer 和 L-buffer 来进行光照计算，并在最终的前向着色 pass 中，进行上采样和双边滤波。这种方法对于某些材质比较适用，但是如果光照效果会在像素之间发生迅速变化，则可能会导致瑕疵，例如：将一个粗糙度贴图或者法线贴图应用在一个反射表面上。Sousa 等人[1681]将二次采样和 YCbCr 颜色编码的反照率纹理一起使用，从而帮助降低存储成本。反照率会对漫反射项产生影响，而漫反射项则不太容易受到高频变化的影响。

还有更多类似这样的解决方案[892, 1011, 1351, 1747]，但是每种方案都有着不同的组成元素，例如：哪些分量会被存储和分解；执行哪些 pass；以及如何渲染阴影、透明度、抗锯齿和其他效果。所有这些技术的主要目标都是相同的，即光照的高效渲染，这些技术至今仍在进行使用[539]。其中一些方案存在这样的一个限制，它们可能需要更加严格的材质模型和光照模型[1332]。例如：Shulz 指出[1589]，使用一个基于物理的材质模型，意味着需要存储镜面反射率（ F_0 值），以便在光照中计算菲涅尔项。光照 prepass 的需求不断增加，推动了他的团队从光照 prepass 转向完全的延迟着色系统。

即使每个光源只会访问的少量 G-buffer，这样也会占用大量的带宽成本。更快的方法是在一个 pass 中，只对影响该像素的光源进行评估计算。Zioma [1973]是第一个探索为前向着色创建光源列表的人。在他的方案中，光源体积会被渲染，会在每个与其重叠的像素位置上存储光源的相对位置、颜色和衰减因子。会使用深度剥离来存储那些重叠相同像素的光源信息。然后使用存储下来的光源列表，来对场景中的几何物体进行渲染。虽然这种方案是可行的，但是它会受到一个限制，即一个像素位置上具体可以重叠多少数量的光源。Trebilco [1785]进一步提出了一个想法，即为每个像素都创建一个光源列表。他执行了一个 z-prepass，来避免过度绘制并剔除被隐藏的光源。然后对光源体积进行渲染，并将其存储为逐像素的 ID 值，然后在前向渲染的 pass 期间进行访问。他给出了几种在单个缓冲区中存储多个光源的方法，包括 bit 位移和混合技术等，该技术允许在不进行多个深度剥离 pass 的情况下存储四个光源。

分块着色（tiled shading）最早是由 Balestra 和 Engstad [97]在 2008 年，为游戏《神秘海域：德雷克的宝藏》提出的，随后在寒霜引擎[42]和 PhyreEngine [1727]等引擎中都进行了应用。分块着色的核心思想是：将光源分配到每个像素的 tile 上，从而对每个表面上需要进行计算的光源数量进行了限制，同时还限制了所需的工作量和存储空间。然后在单次着色器调用中访问这些逐 tile 的光源列表，而不是使用延迟着色中的方法，即为每个光源都调用一次着色器[990]。

这些用于光源分类的 tile 是屏幕上的一组方形像素，例如：tile 大小可以为 32×32 像素。请注意，还有一些其他的方法可以将屏幕进行分块处理，从而进行交互式渲染，例如：移动端处理器通过处理这些 tile 来渲染图像[145]，一些 GPU 架构使用屏幕 tile 来完成各种任务（详见第 23 章）。而这里的 tile 则是由开发人员所选择的结构，通常与底层硬件的关系不大。光源体积的分块渲染类似于场景的低分辨率渲染，它可以在 CPU 上执行，也可以在 GPU 中的计算着色器中执行[42, 43, 139, 140, 1589]。

可能会影响这个 tile 的光源会被记录在一个列表中。在执行渲染的时候，给定 tile 中的像素着色器会使用该 tile 对应的光源列表，来对表面进行着色，图 20.6 左侧展示了这一点。从图中我们可以看到，并不是所有的光源都会与每个 tile 重叠。这些 tile 的屏幕空间边界构成了一个不对称的视锥体，用于确定哪些光源与当前 tile 发生重叠。每个光源效果的球形体积，可以在 CPU 或者计算着色器上进行快速测试，从而判断该光源与哪些 tile 的视锥体相重叠。只有当发生重叠的时候，我们才需要进一步处理该 tile 中的像素。通过按 tile 而不是按像素来存储光源列表，我们会倾向于保守处理，因为光源的体积可能并不会与整个 tile 重叠，这样可以大大降低处理、存储成本和带宽成本[1332]。

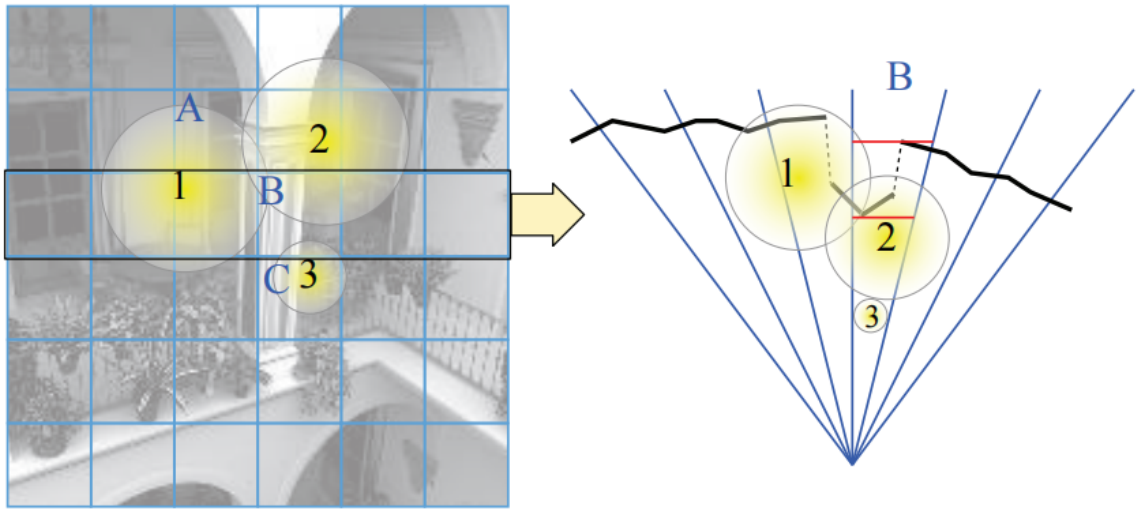


图 20.6：分块渲染的示意图。左侧：屏幕被划分成 6×6 的 tile，其中包含 3 个光源（分别是光源 $1-3$），这些光源照亮了这个场景。仔细观察 tile A – C，我们可以发现 tile A 可能会受到光源 1 和光源 2 的影响，tile B 可能会受到光源 1 – 3 的影响，tile C 可能会受到光源 3 的影响。右侧：展示了左侧黑色轮廓 tile 的俯视图。对于 tile B，其深度边界使用红线进行表示。在屏幕上，tile B 看起来与所有的光源都会发生了重叠，但是实际上只有光源 1 和光源 2 会与 tile B 的深度边界重叠，光源 3 会被遮挡。

为了确定光源是否与 tile 发生重叠，我们可以使用针对球体的视锥体测试，这在章节 22.14 中进行了描述。但是章节 22.14 中的视锥体测试，假设了一个大而宽的视锥体与一个相对较小的球体。然而，由于这里的视锥体源自于屏幕空间中的 tile，因此它通常会又长又细，而且是不对称的。这降低了剔除的效率，因为所得到的重叠数量可能会有所增加（假阳性，误报），如图 20.7 左侧所示。相反，我们可以在对视锥体的平面进行测试之后，再添加一个球体/box 测试（章节 22.13.2）[1701, 1768]，如图 20.7 右侧所示。Mara 和 McGuire [1122]对投影球体进行了替代测试，包括他们自己的 GPU 高效版本。Zhdan [1968]指出，这种方法并不适用于聚光灯，并讨论了使用分层剔除、光栅化和代理几何体的优化技术。

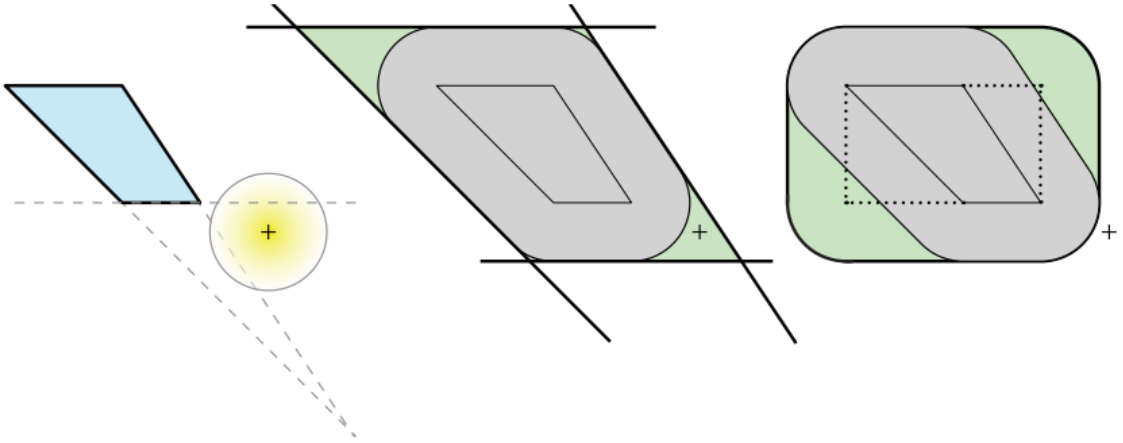


图 20.7：左侧：在简单的球体/视锥体测试中，这个圆形将会被认为是相交的，因为它与视锥体的底部平面和右侧平面相重叠。中间：左边的测试插图，其中这个视锥体被扩展了，圆圈的原点位于加号处，它仅会针对扩展平面（加粗黑色线条）进行测试，其中绿色区域会出现假相交现象。右侧：在视锥体周围放置了一个 box（虚线所示），在中间的平面测试之后，会再添加一个球/box 测试，形成如图所示的形状（加粗黑色线条）。注意，这个测试会在绿色区域中产生额外的假相交现象，但是应用这两个测试之后，这些假阳性区域会大大减少。由于球体的原点位于这个右侧形状之外，因此这个球体会被正确地报告为不与视锥体发生重叠。

这种光源分类的过程，可以与延迟着色或者前向着色一起使用，Olsson 和 Assarsson [1327]对此进行了详细的描述。在分块延迟着色（tiled deferred shading）中，会像往常一样建立 G-buffer，并将每个光源体积记录在与其重叠的 tile 中，然后将这些逐 tile 的光源列表应用于 G-buffer，从而计算最终的着色结果。在基本的延迟着色中，每个光源都是通过渲染代理几何体（例如一个四边形）来应用的，从而强制像素着色器对这个光源进行评估。使用分块着色，可以使用一个计算着色器来执行这个评估；或者使用一个逐 tile 渲染的四边形，来驱动像素着色器进行评估。当一个片元被计算的时候，该 tile 列表中的所有光源都会被应用在该片元上。应用光源列表有若干个优点，包括：

- 对于一个像素，G-buffer 最多会被读取一次，而不是每个重叠光源都读取一次。
- 对于一个像素，只会对输出图像缓冲区写入一次，而不是累积每个光源的计算结果。
- 可以将渲染方程中的常量项分解出来，并只需要计算一次即可，而不是针对每个光源都计算一次[990]。
- 一个 tile 中的每个片元都会评估相同的光源列表，从而确保 GPU warp 的执行一致性。

- 在所有的不透明物体被渲染之后，透明物体可以使用前向着色进行处理，并使用相同的光源列表。
- 由于所有光源的效果都是在一个 pass 中计算完成的，如果需要的话，可以适当降低帧缓冲的精度。

其中的最后一条，即帧缓冲的精度，在传统的延迟着色引擎中是十分重要的[1680]。在基础的延迟渲染中，由于每个光源的计算都发生一个单独的 pass 中，因此如果将着色结果都累积在一个每个颜色通道只有 8 bit 的帧缓冲区中，很可能会受到带状瑕疵和其他瑕疵的影响。但是尽管能够使用较低的精度，这也与许多现代的渲染系统无关，因为这些系统需要较高的精度输出，来执行色调映射和其他操作。

分块的光照分类也可以与前向渲染一起使用，这种类型的系统被称为分块前向着色（tiled forward shading）[144, 1327]或者 forward+ [665, 667]。首先会执行一个几何物体的 z-prepass，这样一方面可以避免在最终 pass 中的过度绘制，另一方面又允许进一步的光源剔除。使用一个计算着色器来逐 tile 对光源进行分类；第二个几何 pass 会执行前向着色，每个着色器会根据片元所在的屏幕空间位置来访问该 tile 的光源列表。

forward+ 已经在《教团：1886》[1267, 1405]等游戏中进行了使用。Pettineo [1401]提供了一个开源的测试套件，来对比分块延迟着色[990]和分块前向着色的实现。为了实现抗锯齿，在使用延迟着色的时候会存储每个样本。对比结果是好坏参半的（mixed），每种方案都会在特定的测试条件下优于其他方案。在没有抗锯齿的情况下，当场景中的光源增加到 1024 个，延迟着色往往会在许多 GPU 上胜出；而随着抗锯齿水平的不断增加，前向渲染会表现得更好。Stewart 和 Thomas [1700]使用一个更加广泛的测试，该测试针对一个 GPU 模型进行了分析，同样发现了类似的结果。

这个 z-prepass 还可以用于另一个目的，即根据深度来对光源进行剔除，其思想如图 20.6 所示。第一步是在这个 tile 中找到几何物体的最小深度 z_{min} 和最大深度 z_{max} 。这些都是通过执行一个 reduce 操作来确定的，它会对该 tile 的数据应用一个着色器，通过在一个或者多个 pass 中的采样[43, 1701, 1768]，来计算的 z_{min} 和 z_{max} 。例如：Harada 等人[667]使用一个计算着色器和无序访问视图（UAV），来高效地执行视锥体剔除和确定 tile 的深度。这些值可以用来快速剔除该 tile 中与这个范围不重叠的光源。一些空白的 tile（例如只有天空可见的 tile）也可以被忽略[1877]。场景和应用程序的类型，会对是否值得计算与使用最小最大值，或者两者兼

用产生影响[144]。这种方式的剔除也可以应用于分块延迟着色，因为深度信息已经包含在了 G-buffer 中。

图 20.8：图中红色的 tile 中存在较大的深度不连续性，主要集中在近景物体的边缘。[1387]

由于这个深度边界是从不透明表面中找到的，因此需要单独处理透明物体。为了处理透明表面，Neubelt 和 Pettineo [1267]渲染了一组额外 pass，来创建每个 tile 中的光源，这些光源仅会用于对透明表面的光照和着色。首先，将透明表面渲染在不透明几何物体的 z-prepass 缓冲区上。其中透明表面的 z_{min} 会被保留，而不透明表面的 z_{max} 则被用来代替视锥体的远裁剪平面，因为位于不透明物体背后的物体是看不见的。第二个 pass 会执行一个单独的光源分类 pass，它会生成新的、逐 tile 的光源列表。第三个 pass 只会将透明表面发送到渲染器中，以类似的方式来进行分块前向着色。所有这样的表面都会被新的光源列表照亮并着色。

对于具有大量光源的场景，找到有效且正确的 z 值范围，对于在进一步处理中剔除大多数光源而言至关重要。然而，常见情况中的深度都是不连续的，这种优化几乎不会获得什么好处。例如：假设现在有一个 tile 包含了一个附近的角色，而这个 tile 的背景是一座遥远的山。二者之间的 z 值范围是巨大的，因此它对于光源剔除几乎没什么用。这个深度范围的问题会对场景中的很大一部分产生影响，如图 20.8 所示。而且刚才的这个例子并不是极端情况。诸如森林、或者包含高草和其他植被的场景，会有很高比例的 tile 都会出现深度不连续性[1387]。

一种解决方案是在 z_{min} 和 z_{max} 之间再次进行分割。这种测试称为双峰聚类（bimodal cluster）[992]或者 HalfZ [1701, 1768]，这个测试会将相交光源分成 3

类，分别是与 z_{min} 和 z_{max} 的中点相比，更近、更远或者全范围的重叠。这样做会对一个 tile 中的物体进行分类，一种物体比中点近，一种物体比中点远。这种方法并不能解决所有的问题，例如：一个光源可能并不会与 tile 中的物体重叠，或者会与两个以上物体在不同的深度范围内重叠。尽管如此，这种方法可以在总体上能够减少光源的计算量。

Harada 等人[666, 667]提出了一种更加复杂的算法，它称为 2.5D 剔除，其中每个 tile 的深度范围 z_{min} 和 z_{max} ，会沿着深度方向分割为 n 个单元格，这个过程如图 20.9 所示。创建一个包含 n 个 bit 的几何位掩码，在存在几何体的地方，对应的 bit 会被设置为 1。为了提高效率，他们使用 $n = 32$ 。对所有的光源进行迭代，并为每个与 tile 视锥体重叠的光源都创建一个光源位掩码，这个光位掩码代表了光源具体位于哪个单元格中。这个几何位掩码和光源位掩码使用按位与（AND）操作，如果结果为零，那么说明这个光源不会影响 tile 中的任何几何物体，如图 20.9 右侧所示。否则，这个光源就会被添加到该 tile 的光源列表中。对于一种特定 GPU 架构，Stewart 和 Thomas [1700]发现，当光源数量增加到 512 个以上时，HalfZ 方法就会开始优于基本的分块延迟着色；当光源数量进一步增加到 2300 个以上时，2.5D 剔除将开始占据主导地位，尽管不是特别明显。

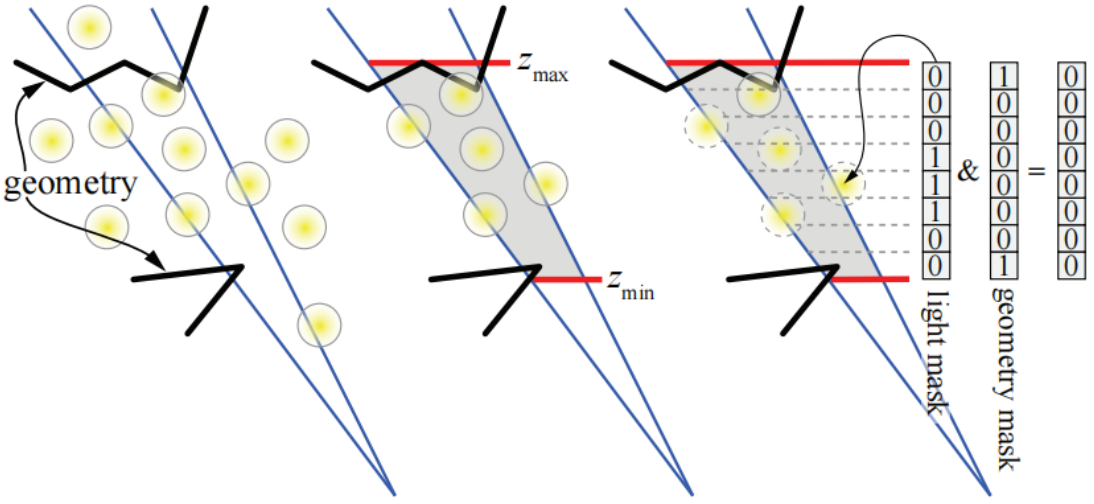


图 20.9：左侧：蓝色的是一个 tile 的视锥体，黑色折线的是场景中的几何图形，黄色圆形是一组光源。中间：分块剔除，并使用红色的 z_{min} 和 z_{max} 值来剔除那些与灰色区域不重叠的光源。右侧：使用聚类剔除， z_{min} 和 z_{max} 之间的区域会被分割为 n 个单元格，在本例中 $n = 8$ 。根据像素的深度来计算几何位掩码（10000001），并为每个光源都计算一个光源位掩码。如果几何位掩码和光源位掩码之间按位与的值为 0，那么该 tile 就不需要再考虑这个光源了。右侧最顶部的光源，其光源位掩码为 11000000，因此它是唯一要进行光照计算的光源，因为

光源位掩码 11000000 和该 tile 的几何位掩码 10000001，二者按位与的结果是 10000000，结果是非零的。

Mikkelsen [1210]通过使用不透明物体的像素位置，来对光源列表进行进一步地修剪。为每个 16×16 像素的 tile 生成一个列表，其中包含了每个光源的屏幕空间包围框，以及用于剔除的 z_{min} 和 z_{max} 几何边界。然后使用 64 个计算着色器线程，其中每个线程都会负责将 tile 中的 4 个像素与每个光源进行比较，从而对光源进行进一步地裁剪。如果在一个 tile 中，发现没有像素的世界空间位置位于光源体积内部，则从这个列表中删除该光源。最终得到的光源集合相当准确，因为那些保证影响至少一个像素的光源都会被保留下来。Mikkelsen 发现，在他的场景中，使用 z 轴的进一步剔除，会降低整体的性能表现。

由于光源被放置到一个列表中，并将其作为一个集合来进行评估，可能会使得延迟系统的着色器变得相当复杂，因为一个着色器必须能够处理所有的材质和所有的光源类型。使用 tile 可以帮助降低这种复杂性，这个想法是在每个像素上存储一个位掩码，其中每个 bit 与材质在该像素中所使用的着色器特征相关联。对于每个 tile，这些位掩码会使用按位或（OR）组合在一起，从而确定该 tile 所使用的最小特征数量。这些位掩码也可以同时使用按位与，来查找所有像素都使用的共同特征，这意味着色器不需要使用一个 if 测试来检查是否需要执行这段代码（所有像素都使用的特征）。然后对贴图中的所有像素，都使用满足这些要求的着色器[273, 414, 1877]。这种着色器的特化十分重要，不仅因为每个着色器需要执行的指令变得更少了，还因为生成的着色器可以实现更高的占用率（章节 23.3），否则这个着色器就必须为最坏情况下的代码路径来分配寄存器。除了材质和光源之外的属性，也可以被追踪并用于对着色器产生影响。例如：在游戏《争分夺秒（Split/Second）》中，Knight 等人[911]会根据 4×4 tile 是否完全或者部分位于阴影中、是否包含需要进行抗锯齿的多边形边缘，以及其他一些测试，来对其进行分类。

20.4 聚类着色（Clustered Shading）

分块的光源分类使用一个基于 tile 的二维空间范围，以及几何物体的深度边界（可选的）。而聚类着色（clustered shading）会将视锥体划分为一组三维单元格，称为聚类（cluster）。与分块着色的 z-depth 方法不同，这种聚类着色的细分是在整个视锥体上进行的，独立于场景中的几何形状。由此产生的算法对于不同相机位置的性能波动较小[1328]，并且可以在 tile 中包含深度不连续性时表现更好[1387]。聚类着色也可以应用于前向着色和延迟着色中。

由于透视效应，一个 tile 的横截面积会随着与相机距离的增加而增加。这种均匀的细分方案会将 tile 的视锥体变成长而薄的体素，这并不是最佳方案。为了弥补这一点，Olsson 等人[1328, 1329]在观察空间中，以指数形式来对场景中的几何形状进行聚类，而不依赖几何形状的 z_{min} 和 z_{max} ，这会使得聚类空间变得更加立方化，即更接近一个立方体。例如：《正当防卫 3》的开发者，使用了具有 16 个深度切片的 64×64 像素 tile，并且他们在每个轴上尝试了更大的分辨率，以及使用固定数量的屏幕图像 tile（无论屏幕分辨率如何）[1387]。虚幻引擎中使用了相同大小的 tile 以及典型的 32 深度切片[38]，如图 20.10 所示。

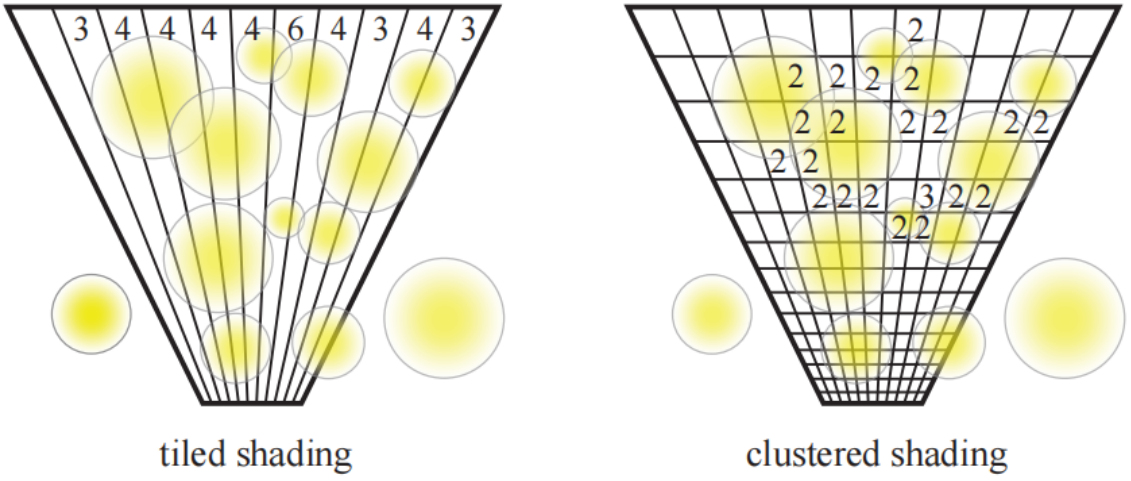


图 20.10：分块着色和聚类着色，以二维俯视角进行展示。视锥体会被细分，场景中的光源体积会根据它们所重叠的区域来进行分类。其中分块着色会在屏幕空间中进行细分，而聚类着色还会按 z-depth 进行切片划分。每个细分空间中都包含一个光源列表，图中包含两个或者更多光源的空间，会标注对应的光源数量。对于分块着色而言，如果不计算该 tile 中 z_{min} 和 z_{max} 的话，这个光源列表中可能会包含大量不需要的光源。而聚类着色则不需要对场景几何进行预渲染来帮助剔除光源列表，尽管这样的 pass 也可以起到一定帮助。[1387]

光源会按照与它们重叠的 cluster 来进行分类，并形成对应的光源列表。由于 cluster 并不依赖场景几何的 z-depth，因此可以仅针对视图和光源集合来计算这些 cluster [1387]。对于每个表面，无论是不透明的还是不透明的，都会使用该物体的位置来检索相关的光源列表。这种聚类方法提供了一种高效且统一的照明解决方案，适用于场景中的所有物体，包括透明物体和体素物体等。

与分块着色方法一样，这种使用聚类的算法也可以与前向着色或者延迟着色相结合。例如：《极限竞速：地平线 2》在 GPU 上计算对应的 cluster，并使用了前向着色方案，因为前向着色可以提供 MSAA 的支持，而无需任何额外的工作[344, 1002, 1387]。虽然在单个 pass 中，前向着色可能会发生过度绘制，但是使用一些其他方法

（例如粗略的前后排序[892, 1766]，或者是仅对一个物体子集来执行 prepass [145, 1768]），可以在不执行第二个完整几何 pass 的情况下，避免大量的过度绘制。但是 Pettineo [1407]发现，即使使用了这些优化方法，使用一个单独的 z-prepass 速度也会更快。又或者，可以对不透明表面使用延迟着色，并使用相同的光源列表结构，然后将其用于对透明表面的前向着色。这种方法在《正当防卫 3》中进行了使用，它在 CPU 上创建光源列表[1387]。Dufresne [390]在 CPU 上并行生成聚类光源列表，因为这个过程并不依赖于场景中的几何物体。

聚类光源给每个列表分配的光源数量更少，并且要比分块方法具有更低的视图依赖性[1328, 1332]。由 tile 所定义的长而薄的视锥体，相机的一次微小移动可能会产生相当大的内容变化。例如：位于同一条直线上的路灯可以对齐填充一个 tile [1387]，但是当相机发生移动的时候，这些路灯可能就不再对齐了。即便使用了 z-depth 的细分方法，每个 tile 中表面的远近距离，也可能会因为单个像素的深度变化而发生剧烈变化。而聚类则不太容易受到此类问题的影响。

如前所述，Olsson 等人[1328, 1329]以及其他人，对聚类着色进行了一些优化。其中一种技术是为光源构建一个 BVH，然后使用这个 BVH 来快速确定哪些光源体积与给定的 cluster 相重叠。但是只要有一个光源发生了移动，就需要对这个 BVH 进行重建。一个可以用于延迟着色的优化选项是，使用量化的法线方向来对一个 cluster 中的表面进行剔除。Olsson 等人根据方向，来将表面法线分类为一个结构，这个结构会在立方体上的每个面上，都会维护一个 3×3 的方向集合，从而构建一个法线锥（章节 19.3），六个面总计共有 54 方向。然后在创建 cluster 列表的时候，这个结构可以用于进一步对光源进行剔除，即剔除 cluster 中那些位于所有表面背后的光源。对于大量的光源而言，排序操作可能会非常昂贵，van Oosten [1334]对各种策略和优化进行了探索。

当我们拥有场景中的可见几何位置信息时（例如延迟着色或者 z-prepass），还可以进行一些其他优化。那些不包含几何形状的 cluster 可以从处理中移除，从而得到一个稀疏的 cluster 网格，只需要更少的处理和存储即可。这样做意味着，我们必须首先对场景进行预处理，从而找到哪些 cluster 被几何物体所占用。由于这个过程需要访问深度缓冲区中的数据，因此它必须在 GPU 上来执行。与一个 cluster 的体积相比，其中的几何物体可能只会与较小的区域相互重叠。通过将这些样本形成一个更加紧密的 AABB 来对光源进行测试[1332]，可以剔除更多的光源。一个经过优化的系统可以处理超过 100 万个光源，并且随着这个数字的增加，这样的系统也具有很好的可伸缩性，同时它对于少量光源也十分高效。

实际上我们并不需要使用一个指数函数来对屏幕的 z 轴进行细分，对于那些具有许多远处光源的场景而言，这种细分方法反而可能产生负面影响。当使用一个指数分布的时候，cluster 的体积会随着屏幕深度的增加而增加，这可能会导致远处 cluster 的光源列表过长。一种解决方案是对 cluster 集合的最大距离进行限制，即构建一个光源聚类的“远裁剪平面”，这样远处的光源会逐渐消失，并使用粒子或者光晕来表示它们，或者也可以使用一些烘焙方法[293, 432, 1768]，也可以使用一些更加简单的着色器、lightcut 方法[1832]或者其他 LOD 技术。相反，最接近观察者的空间可能相对来说是光源较少的，但是这些空间反而会被大量细分。一种方法是将这个分类视锥体的“近裁剪平面”强制设定到一定的合理距离，并将比“近裁剪平面”更近的光源分类到第一个深度切片中[1387]。

在《DOOM（2016）》中，开发人员[294, 1682]使用了 Olsson 等人[1328]和 Persson [1387]的聚类方法，组合实现了他们的前向着色系统。他们首先会执行一个 z-prepass，耗时约 0.5 ms。他们的光源列表构建方案，可以被认为是裁剪空间（clip-space）的体素化。光源、环境光照探针和贴花，会通过与每个单元格的 AABB 进行测试，从而将它们插入到这些单元格中。其中贴花的添加是一个显著的改进，因为这使得聚类前向渲染系统可以像延迟渲染一样，获得对于这些实体的渲染优势。在前向着色过程中，引擎会对一个单元格中发现的所有贴花进行循环遍历。如果发现一个贴花与表面位置相重叠，则对该贴花的纹理值进行检索和混合。这些贴花可以以任何想要的方式来与底层表面混合，而不是仅限于混合阶段中可用的操作（而在延迟着色中则是这样的）。使用聚类前向着色，贴花还可以渲染到透明表面上，然后会应用该单元格中所有相关的光源来进行着色。

可以使用 CPU 来构建光源列表，因为这个过程并不需要场景中的几何物体，同时对光源球体和 cluster 包围盒的重叠测试的开销也很低。然而，如果涉及到聚光灯或者其他体积形状的光源，在其周围使用一个球形包围体，可能会导致将这样的光源添加到许多并不会产生效果的 cluster 中，并且想要精确的求解相交测试的成本也可能会很高。沿着这些思路，Persson [1387]提供了一种将球体体素化为一组 cluster 的快速方法。

GPU 的光栅化管线可以用来对光源体积进行分类，从而避免这些问题。Ortegren 和 Persson [1340]描述了一个构建光源列表的两 pass 过程。在第一个壳 pass 中，每个光源都会由包含它的低分辨率网格进行表示，并使用保守光栅化（章节 23.1.2）来将每个光源的壳渲染到 cluster 网格中，记录与每个光源重叠的最小 cluster 和最大 cluster。在第二个填充 pass 中，使用一个计算着色器来将光源添加到位于这些包围体之间的每个集群链表中。使用网格来代替包围球体，可以为聚光灯提供更加紧密的

边界表示，并且场景中的几何形状可以对光线可见性进行直接遮挡，从而进一步对光源列表进行剔除。当保守光栅化不可用时，Pettineo [1407]描述了一种方法，该方法利用表面梯度来保守估计三角形在每个像素处的 z 界限。例如：如果需要某个像素位置上的最远距离，则使用 x-depth 和 y-depth 的梯度，来选择像素中的最远角落，并计算该点位置上的深度。由于这样找到的点可能会位于三角形之外，因此他也将光源的 z-depth 范围作为一个整体来进行限制，从而避免那些接近边缘的三角形使得估计的 z-depth 偏离太远。Wronski [1922]对各种解决方案进行了探索，并提出了在网格单元格周围放置一个包围球，对圆锥进行相交测试的想法。这种测试的评估速度很快，当这个单元格接近立方体的时候工作良好，但是当这个单元格被拉长时效果会较差。

Drobot [385]描述了在《使命召唤：无限战争》中，是如何使用网格来插入光源的。想象一个静态的聚光灯，它在空间中形成一个圆锥形的体积。如果没有进一步处理的话，这个圆锥可以延伸到相当长的距离，这个距离要么到达场景的最大范围，要么到达为光源定义的最大距离。现在想象一下这个聚光灯的阴影贴图，这个阴影贴图是使用场景中的静态几何物体生成的。这个贴图定义了光源在每个方向上的最大照射距离。在烘焙过程中，这个阴影贴图被转换成了一个低分辨率的网格，然后将其作为光源的有效体积。这个网格是保守的，它使用每个阴影贴图区域中的最大深度来进行构建，因此它完全包含了被光源照射到的空间体积。聚光灯的这种表示方法相较于原始的圆锥体积而言，可能会重叠数量更少的 cluster。

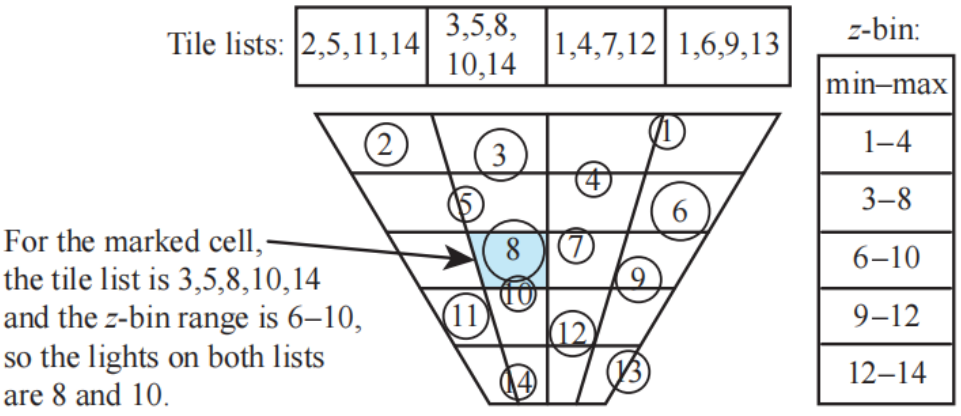


图 20.11：使用 z-bin 方法，每个光源会根据其 z-depth 被赋予一个 ID。为每个 tile 都生成一个列表。每个 z-bin 都会存储一个最小 ID 和最大 ID，即一个可能与该切片重叠的、保守的光源范围。对于标记单元格中的任何像素，我们可以通过检索两个列表来找到重叠的部分。

与上述过程无关，光源列表的存储和访问方法被称为 z-bin，这要比聚类着色所占用的内存少得多。在这种方法中，光源会按照屏幕的 z-depth 进行排序，并根据这些深度给出相应的 ID。然后使用一组 z 切片来对这些光源进行分类，每个切片都具有相

同的深度厚度，而不是指数分布的厚度。每个 z 切片只会存储与之重叠光源的最小 ID 和最大 ID，详见图 20.11 所示。同时还会生成分块着色的光源列表，还可以选择使用几何深度来进行裁剪。然后，每个表面位置都会访问这个二维 tile 结构，以及每个切片中的一维 z-bin ID 范围。tile 列表给出了该 tile 中可能会影响这个像素的所有光源，而像素的深度信息则用来检索可能与 z 切片重叠的 ID 范围。这两者的重叠是实时计算的，最终会给出这个 cluster 中的有效光源列表。

该算法不需要为三维网格中的每个 cluster 都创建和存储一个光源列表，而是只需要每个二维 tile 的光源列表，以及一个用于存储 z 切片集合的小数组即可，这个小数组具有固定的大小。使用这种方法，可以实现更少的存储和带宽使用，以及更少的预计算过程，其代价是在确定每个像素的相关光源时，需要进行更多的工作。使用 z-bin 可能会导致一些光源被错误分类，但是 Drobot 发现，对于人造环境而言，光源通常很少会在 xy 屏幕坐标和 z-depth 之间同时发生重叠。使用像素着色器和计算着色器，该方案能够在深度不连续的 tile 中给出近乎完美的剔除效果。

用于访问物体的三维数据结构通常可以分类为以下几类：与体积相关的，即将一个网格或者八叉树强加在空间上；物体相关的，即构建一个 BVH；或者是二者混合的，例如在网格单元格的内容周围使用一个包围体。Bezrati [139, 140]在一个计算着色器中来执行分块着色，从而构建一个增强的光源列表，其中每个光源都会包括其影响范围，即最小 z 深度和最大 z 深度。通过这种方式，一个片元可以快速拒绝任何不与之发生重叠的光源。ODonnell 和 Chajdas [1312]提出了分块光源树（tiled light tree），这个结构会在 CPU 端进行构建。他们使用带有深度界限的分块光源列表，并构建了一个界限区间的层次结构。也就是说，与 Olsson 等人[1328]的做法不同，他们并没有构建一个单独的三维层次结构，而是通过一个 tile 中每个光源的 z 范围，创建了一个更加简单的一维层次结构。这种结构能够很好地映射到 GPU 的架构上，并且能够更好地处理大量光源落在单个 tile 中的情况。他们还提供了一种混合算法，可以选择将 tile 划分为单元格（上面提到的法线聚类着色方法）、或者是使用光源树。当单元格与光源的平均重叠度较低的情况下，光源树工作得最好。

局部光源列表的想法也可以在移动设备上进行使用，但是有着一些不同的限制和选择。例如：在移动设备上，以传统的延迟着色方式，一次渲染一个光源可能是最高效的方法，因为移动设备具有将 G-buffer 存储在本地内存中的独特属性。分块前向着色可以在支持 OpenGL ES 2.0 的设备上进行实现，移动 GPU 几乎是都支持了 OpenGL ES 2.0。使用 OpenGL ES 3.0 和被称为像素本地存储（pixel local storage）的扩展，ARM GPU 中的基于 tile 的渲染系统，可用于高效地生成和应用光源列表。更多信息请详见 Billeter 的演讲[145]。Nummelin [1292]对寒霜引擎从桌

面端移植到移动端的相关问题进行了讨论，其中包括了一些光源分类方案的权衡，因为计算着色器在移动硬件上的支持较少。由于移动设备使用了基于 tile 的渲染，因此延迟着色生成的 G-buffer 数据可以存储在本地内存中。Smith 和 Einig [1664]描述了使用帧缓存 fetch 和像素本地存储来实现这一目的，并发现这些机制能够将总带宽成本降低一半以上。

总之，分块、聚类或者其他光源列表的剔除技术，可以与延迟着色和前向着色一起进行使用，并且这些技术也可以应用于贴花效果。光源体积剔除算法的重点在于最小化每个片元需要计算的光源数量；而将几何过程和着色过程进行解耦的思想，可以用来平衡计算成本和带宽成本，从而最大限度地提高效率。正如视锥体剔除一样，它需要花费额外的处理时间，但是如果所有的物体总是会出现在视野中，那么视锥体剔除方法并不会带来任何好处，反而会带来额外的开销，在一些特定条件下，本章节中的一些技术也几乎不会获得什么好处，例如：如果太阳是唯一的光源，那么不需要进行任何光源剔除的预处理。如果只有很少的表面过度绘制和很少的光源，那么延迟渲染可能会花费更多的时间。如果场景中包含了许多有限效果（只会影响一定范围）的光源，那么无论是使用前向着色还是延迟着色，花费一些时间来创建局部光源列表都是十分值得的。当场景中的几何图形处理起来十分复杂、或者表面渲染十分昂贵时，延迟着色提供了一种避免过度绘制的方法，它还可以最小化程序和状态切换所带来的驱动程序成本，并且能够使用更少的调用来渲染更大的合并网格。请记住，在渲染单帧的时候，可以使用上述方法的其中一种。具体什么样的技术组合才是最佳的，这不仅仅取决于场景，也取决于每个物体或者每个光源上的变化因素[1589]。

	Trad. forward	Trad. deferred	Tiled/clust. defer.	Tiled/clust. fwd.
Geometry passes	1	1	1	1–2
Light passes	0	1 per light	1	1
Light culling	per mesh	per pixel	per volume	per volume
Transparency	easy	no	w/fwd→	easy
MSAA	built-in	hard	hard	built-in
Bandwidth	low	high	medium	low
Vary shading models	simple	hard	involved	simple
Small triangles	slow	fast	fast	slow
Register pressure	poss. high	low	poss. low	poss. high
Shadow map reuse	no	yes	no	no
Decals	expensive	cheap	cheap	expensive

表 20.1：对于一个典型的桌面端 GPU，对比了不同的渲染方法，分别是：传统的单 pass 前向渲染；单 pass 延迟渲染；使用分块/聚类光源分类方法的延迟渲染；使用分块/聚类光源分类方法的前向渲染。[1332]

作为本节的总结，我们在表 20.1 中总结了各种方法的主要差异。其中“透明度（Transparency）”一行中的右箭头，代表了不透明表面使用延迟着色，透明表面使

用前向着色。其中的“小三角形（Small triangle）”是延迟着色的其中一个优势，四边形着色（章节 23.1）在前向着色时可能会效率低下，因为所有的四个样本都会被充分计算。“寄存器压力（Register pressure）”指的是相应着色器的整体复杂程度，在着色器中使用更多的寄存器，意味只能形成较少数量的线程，从而导致 GPU 的 warp 无法被充分利用[134]。如果采用一些着色器精简（shader streamlining）方法的话[273, 414, 1877]，那么分块延迟着色和聚类延迟着色技术的成本会降低。当 GPU 显存变得更加有限时，阴影贴图的重用并通常不像以前那样重要[1589]。

当存在大量光源的时候，阴影的渲染是一个挑战。一种应对方法是，除了最近光源、最亮光源以及太阳之外，忽略所有其他光源的阴影计算，这样可能会导致一些较弱的光源发生漏光问题。Harada 等人[667]讨论了他们如何在一个分块前向着色系统中使用光线投射，从而为每个可见表面的像素都生成一条指向每个附近光源的光线。Olsson 等人[1330, 1331]讨论了使用占用网格单元（occupied grid cell）来作为场景几何图形的代理，从而生成阴影贴图，并根据需要来创建样本。他们还提出了一个混合系统，将这些有限的阴影贴图与光线投射相结合。

使用世界空间而不是屏幕空间来生成光源列表，是另一种构建聚类着色的方法。这种方法在某些情况下是合理的，但是由于内存的限制[385]，因此在大型场景中可能需要避免使用这种方法，并且远处的 cluster 将会只有若干个像素级别的大小，这会对性能表现产生影响。Persson [1386]提供了一个基本的聚类前向着色系统的代码，其中静态光源被存储在一个三维世界的空间网格中。

20.5 延迟纹理

延迟着色避免了过度绘制问题，即避免了先计算片元的着色，然后再将这些片元丢弃。然而在形成 G-buffer 的时候，仍然会发生过度绘制现象。当一个物体被光栅化的时候，它的所有参数都会被检索，在这个过程中会执行若干次纹理访问。如果稍后绘制的另一个物体会遮挡这些已经存储下来的样本，那么在渲染第一个物体时花费的所有带宽就都被浪费了。有一些延迟着色系统会执行部分或者完整的 z-prepass，从而避免对被遮挡的表面进行纹理访问[38, 892, 1401]。然而，额外的几何 pass 是许多系统尽可能需要避免的，其成本太高。带宽不仅仅需要用来获取纹理数据，同时也会用于访问顶点数据和其他的一些数据。对于非常精细的几何图形而言，相较于能够节省下来的纹理访问成本，这个额外的 pass 可能会带来更高的带宽开销。

形成和访问的 G-buffer 数量越多，内存和带宽的成本就越高。而在某些系统中，带宽可能并不是主要问题，因为性能瓶颈可能主要集中在 GPU 的处理器中。我们在第 18 章中曾经详细地讨论过，性能瓶颈总是存在的，而且随时都有可能发生变化。之

所以会有如此多的效率方案被不断提出，其中一个主要原因就是每个方案都是针对特定的平台和场景类型进行开发的。其他的一些因素，例如系统实现和系统优化的难易程度、内容创作的难易程度、以及其他各种人为因素，也可以用于决定要构建一个什么样的方案。

虽然 GPU 的计算能力和带宽能力都会随着时间的推移而不断提高，但是这两种能力的增长速度有所不同。GPU 计算能力的增长速度更快，这种趋势与 GPU 上的新功能相结合，意味着一种面向未来的、不会过时（future-proof）系统，应当将性能瓶颈瞄准 GPU 计算而不是缓冲区访问上。也就是说，使用额外的 GPU 计算来降低带宽压力[217, 1332]。

已经有一些不同的方案被开发出来，它们使用了单个的几何 pass，并且可以在需要之前避免对纹理进行检索。Haar 和 Aaltonen [625]描述了如何在《刺客信条：大革命》中使用虚拟延迟纹理化（virtual deferred texturing）的。他们的系统管理了一个本地 8192×8192 可见纹理图集，每个纹理的分辨率为 128×128，这些纹理是从一个更大的集合中选择的。这个纹理图集允许存储 (u, v) 纹理坐标，可以用来访问图集中的任何纹素。有 16 bit 会用于存储纹理坐标，其中区分 8192 个位置需要 13 个 bit（$2^{13} = 8192$），剩下的 3 个 bit 用于亚纹素精度，即 8 个级别。还会存储一个 32 bit 的切线基底，它被编码为一个四元数[498]（章节 16.6）。这样做意味着只需要一个 64 bit 的 G-buffer。由于在几何 pass 中并不会执行任何的纹理访问，因此过度绘制所带来的额外成本也很低。在这个 G-buffer 建立之后，在着色期间可以访问这个虚拟纹理。前面我们提到，想要使用纹理的 mipmap，需要梯度信息，在上述这个过程中并不会存储梯度信息。相反，会检查每个像素的邻居，并使用那些最接近 (u, v) 值的像素来动态计算梯度。材质 ID 也是通过确定访问的是哪个纹理图集 tile 而得到的，通过将纹理坐标值除以 128（即纹理分辨率）来得到材质 ID。

在《刺客信条：大革命》中使用了另一种降低着色成本的技术，该技术以四分之一的分辨率来进行渲染，并使用一种特殊形式的 MSAA。在使用 AMD GCN 的游戏主机上，或者使用 OpenGL 4.5、OpenGL ES 3.2 以及其他扩展的系统上[2, 1406]，可以按需来设置 MSAA 的采样模式。Haar 和 Aaltonen 为 4× MSAA 设置了一个网格图案（采样模式），使得每个网格样本直接对应于全屏像素的中心。通过以四分之一的分辨率来进行渲染，他们可以利用 MSAA 的多重采样特性。纹理坐标 (u, v) 和切线基底也可以在没有精度损失的情况下进行跨表面插值，并且使用 8× MSAA 也是可能的（相当于每像素 2× MSAA）。在渲染具有明显过度绘制的场景时（例如树叶和树木），他们的技术显着减少了着色器调用的次数，以及 G-buffer 的带宽成本。

只存储纹理坐标和切线基底的情况是非常少的，还可以使用一些其他的方案。Burns 和 Hunt [217]对可见性缓冲区（visibility buffer）进行了描述，他们在其中存储了两部分数据，分别是一个三角形 ID 和一个实例 ID，如图 20.12 所示。这个执行几何 pass 的着色器速度非常快，因为不会进行纹理访问，而且需要存储这两个 ID 值即可。所有的三角形和顶点数据（位置、法线、颜色、材质等）都存储在全局缓冲区中。在延迟着色 pass 期间，每个像素位置上存储的三角形 ID 和实例 ID 会用于检索这些数据。从屏幕像素处发射的观察射线与三角形相交，从而找到重心坐标，用于在三角形的顶点数据之间进行插值。其他通常不太频繁的计算也必须逐像素地进行，例如顶点着色器中的计算。纹理梯度信息也是根据屏幕上的每个像素计算得到的，而不是插值得到的。然后使用这些数据来对像素进行着色，可以使用任何所需的分类方案来应用光源。

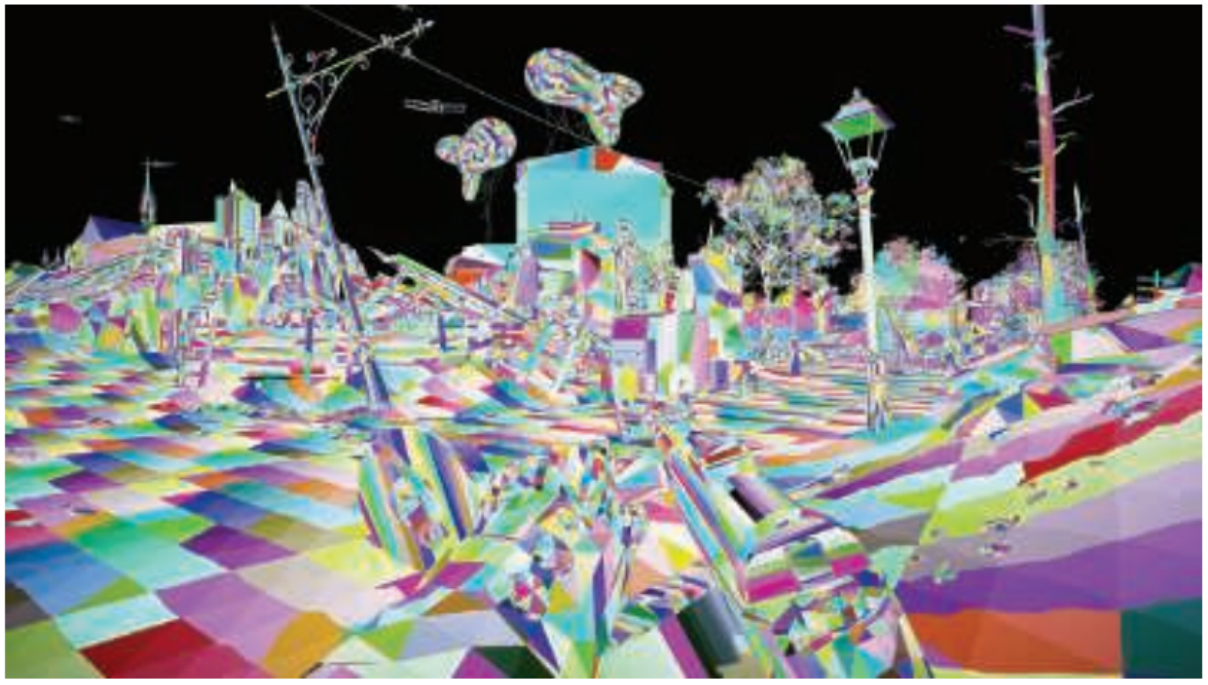


图 20.12：在生成可见性缓冲[217]的第一个 pass 中，只有三角形 ID 和实例 ID 会被渲染并存储在单个 G-buffer 中，图中的每个三角形会以不同的颜色进行显示。[1885]

虽然上述这些过程听起来成本很高，但是请记住，计算能力的增长速度远比带宽能力更快。这个结论倾向于一个计算量更大的管线，它可以最大限度地减少由于过度绘制所造成的带宽损失。如果场景中的网格数量低于 64k，并且每个网格中的三角形也低于 64k，那么每个 ID 长度可以为 16 bit，G-buffer 可以降低到每个像素 32 bit。而更大一些的场景则需要 48 bit 或者 64 bit。

Stachowiak [1685]描述了可见性缓冲的一个变体，它使用了 GCN 架构上的一些可用功能。在初始 pass 期间，三角形上的重心坐标也可以进行计算并按像素进行存储，

这样 GCN 的片元（即像素）着色器可以以非常低的成本来地计算重心坐标，而不是稍后对每个像素都执行观察光线与三角形相交。虽然这样会增加额外的存储开销，但是这种方法有一个重要的优点。对于动画网格而言，原始的可见性缓冲方案需要将任何有过修改的网格数据流式传输到一个缓冲区中，以便在延迟着色期间可以检索这些修改过的顶点位置。保存这些动画后的网格坐标会带来额外的带宽消耗。通过在第一个 pass 中存储重心坐标，我们就已经获得了顶点位置，不需要再次进行获取，这是原始可见性缓冲区的一个缺点。然而，如果还需要像素到相机的距离的话，那么这个值也必须在第一次 pass 中进行存储，因为这个信息之后无法重建出来，而在原始版本中，观察光线与三角形的相交测试，可以获得这个距离信息。

这种管线可以使得几何形状和着色频率相解耦，类似于之前提到的方案。Aaltonen [2]指出，MSAA 网格采样方案可以应用于每一种方法，从而进一步减少所需的平均存储量。他还讨论了这三种方案在存储布局、以及计算成本和性能上的差异。Schied 和 Dachsbacher [1561, 1562]则选择了另一个改进方向，在可见性缓冲区的基础上，使用 MSAA 功能来减少内存消耗和着色计算量，从而实现高质量的抗锯齿效果。

Pettineo [1407]指出，无绑定纹理（bindless texture）功能（章节 6.2.5）可以使得延迟纹理的实现更加简单。他的延迟纹理系统创建了一个更大的 G-buffer，用于存储深度、单独的材质 ID 和深度梯度。他针对 Sponza 模型的渲染进行了测试，并将该系统的性能表现与使用 z-prepass 的聚类前向着色方法、以及不使用 z-prepass 的聚类前向着色方法进行了比较。当关闭 MSAA 的时候，延迟纹理总是要比前向着色快；当开启 MSAA 的时候，延迟纹理则要比前向着色慢一些。正如章节 5.4.2 中所述，随着屏幕分辨率的不断提高，大多数电子游戏已经不再使用 MSAA，而是依赖于时域抗锯齿，因此实际上这种 MSAA 支持并没有那么重要。

Engel [433]指出，由于 DirectX 12 和 Vulkan 中暴露出来的 API 特性，这个可见性缓冲区的概念变得更具吸引力。使用计算着色器来对三角形集合进行剔除（章节 19.8），以及使用计算着色器的其他一些移除技术，可以帮助减少需要光栅化的三角形数量。DirectX 12 的 ExecuteIndirect 命令可以用来创建一个优化的索引缓冲区，只显示那些没有被剔除的三角形。当与一些先进的剔除系统一起使用时[1883, 1884]，Engel 的分析确定了，在 San Miguel 场景的所有分辨率和抗锯齿设置下，可见性缓冲都要优于延迟着色方案。并且随着屏幕分辨率的不断提高，性能差距也会逐渐增大。GPU 上 API 与功能的未来变化，可能会进一步提高其性能表现。Lauritzen [993]对可见性缓冲区进行了讨论，以及需要如何对 GPU 进行改进，从而改善在延迟设置中访问和处理材质着色器的方式。

Doghramachi 和 Bucci [363]详细讨论了他们的延迟纹理系统，他们称之为 deferred+。他们的系统在早期就集成了激进的剔除技术。例如：将前一帧的深度缓冲以一种保守的方式进行下采样和重投影，来为当前场景中的每个像素提供一个深度剔除。通过对视锥体中可见的所有网格包围体进行渲染，这些深度信息可以帮助测试遮挡，正如章节 19.7.2 中简要讨论的那样。

他们还考虑到了 alpha 镂空纹理，如果存在这种纹理的话，则必须在任何初始 pass（或者任何 z-prepass）中进行访问，从而保证镂空纹理背后的物体不会被隐藏。在他们的系统中，剔除和光栅化过程的结果是一组 G-buffer，其中包括深度、纹理坐标、切线空间、梯度和材质 ID 等，这些信息会用于对像素进行着色。虽然该系统中的 G-buffer 数量要比其他延迟纹理方案更多，但是它也确实避免了不必要的纹理访问。对于《杀出重围：人类分裂》中的两个简化场景模型而言，他们发现 deferred+ 要比聚类前向着色的速度更快，并且他们认为使用一些更加复杂的材质和光照模型，会进一步拉大差距。他们还注意到，GPU warp 的使用要明显更好，这意味着微小三角形所造成的问题要更少，即 GPU 曲面细分的表现更好。与延迟着色相比，他们的延迟纹理实现还具有其他的几个优点，例如能够高效地处理更加广泛的材质。该系统的主要的缺点也是大多数延迟方案中常见的，即透明度和抗锯齿。

20.6 对象空间和纹理空间着色

将计算着色值的速率与几何采样的速率相解耦，这个想法是本章节中反复出现的主题。在本小节中，我们将介绍几种不同思路的方法，它们不太容易归类到迄今为止所介绍的各种类别中。特别地，我们将对 Reyes 渲染器[289]中首次出现的概念混合模型进行讨论，皮克斯和其他公司多年来一直使用 Reyes 来制作电影。虽然现在他们主要使用某种形式的光线追踪或者路径追踪来进行渲染，但是在当时，Reyes 以一种创新和高效的方式解决了几个渲染问题。

> “Reyes”这个名字的灵感来自于 Reyes 半岛，有时会缩写为“Reyes”，具体意思是“渲染你所看到的一切（Renders Everything You Ever Saw）”。

Reyes 的核心概念是微多边形（micropolygon）。每个表面都会被切割成一个非常精细的四边形网格。在原始系统中，这个切割（dice）是针对眼睛进行的，其目的是让每个微多边形的尺寸大约是单个像素的一半宽高，以保持 Nyquist 极限（章节 5.4.1）。位于视锥体外部或者背对眼睛的四边形将会被剔除。在这个系统中，每个微多边形都会被着色并赋予单一的颜色。这种技术后续发展到对微多边形网格中的顶点

进行着色[63]。我们在这里将集中对原始系统进行讨论，因为该系统探索了一些十分有价值的想法。

每个微多边形都会被插入到一个像素中的抖动 4×4 采样网格中，即一个超采样的 z-buffer。这个抖动的目的是通过产生噪声来避免瑕疵。因为在光栅化之前，着色是相对于微多边形的覆盖率而发生的，所以这种类型的技术被称为基于对象的着色（object-based shading）。可以将其与前向着色和延迟着色进行比较，其中前向着色是在光栅化期间，在屏幕空间中进行的；而延迟着色则是在光栅化阶段之后进行的，如图 20.13 所示。

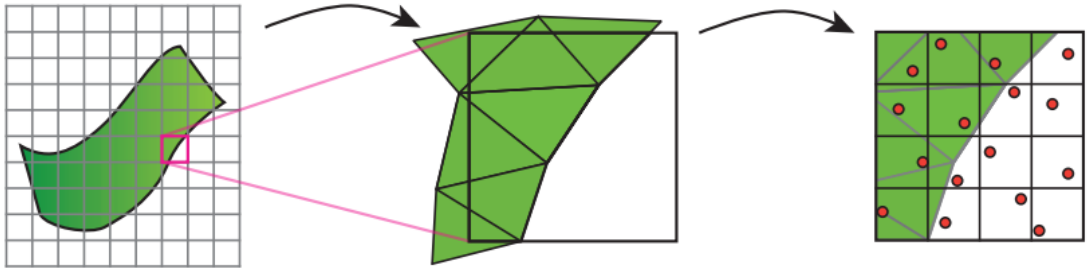


图 20.13：Reyes 渲染管线。每个物体都会被细分成微多边形，然后单独进行着色。每个像素的一组抖动样本（红色点）会与微多边形进行比较测试，测试结果将会用于渲染图像。

在对象空间中进行着色的其中一个优点在于，材质纹理通常会与其微多边形直接相关。也就是说，可以对几何物体进行细分，使得每个微多边形中包含 2 次幂个的纹素。在着色过程中，可以为这个微多边形检索精确过滤的 mipmap 样本，因为它直接与着色的表面面积相关。原始的 Reyes 系统还意味着在访问纹理的时候具有缓存一致性，因为这些微多边形是按照顺序进行访问的。但是这种优势并不适用于所有的纹理，例如：用作反射贴图的环境纹理，必须要使用传统的方法进行采样和过滤。

运动模糊和景深效果也可以很好地与这种布局安排一起工作。对于运动模糊而言，在一帧间隔的抖动时间内，每个微多边形都会沿着其路径来分配一个位置。因此，每个微多边形在移动方向上将会具有不同的位置，从而产生模糊效果。景深效果的实现方式也是类似的，微多边形的分布会基于弥散圆。

Reyes 算法当然存在一些缺点。首先，所有的物体都必须能够被细分，并且必须能够被切割到一个十分精细的程度。着色过程发生在 z-buffer 的遮挡测试之前，这可能会导致一些过度绘制。Nyquist 采样极限并不意味着能够捕获尖锐的镜面高光等高频现象，而是说这种采样足以重建那些较低频率的现象。

一般来说，每个物体都必须是“可绘制的（chartable）”，换句话说，就是顶点必须具有 (u, v) 纹理坐标，从而为模型上的不同区域提供唯一对应的纹素。如图 2.9 和

图 6.6 所示。基于对象的着色可以被认为是着色中的第一个烘焙方法，它使用相机来确定视图依赖的效果，并对每个表面区域所花费的工作量进行限制。在 GPU 上执行基于对象的着色，一种简单方法是将物体细分到精细的亚像素级别，然后对网格上的每个顶点进行着色。这样做的代价可能会很高，因为每个三角形的设置成本并不会分摊到多个像素上。并且由于四边形渲染（章节 23.1），单像素的三角形将会产生四个像素着色器调用，因此开销会变得更大。GPU 针对渲染覆盖一定数量像素的三角形进行了专门优化，例如：16 个像素或者更多（章节 23.10.3）。

Burns 等人[216]对于对象空间中的着色进行了探索，他们在确定哪些物体位置是可见的之后，再来执行对象空间中的着色。他们使用一个“多边形网格”来确定一个被切割的物体是否可见，并尽可能地进行剔除，然后再进行光栅化。然后使用一个独立的对象空间“着色网格”来对可见区域进行着色，每个纹素都对应于表面上的一个区域。着色网格的分辨率可以与多边形网格不同。他们发现对几何表面进行精细细分几乎没有什么好处，因此将二者分离可以更加高效地利用资源。虽然他们只在模拟器中实现了相关工作，但是他们的技术对新的研究和开发工作产生了影响。

有相当多的研究从 Reyes 中获得了灵感，它们对 GPU 上针对各种现象的、更快的着色方法进行了研究。Ragan-Kelley 等人[1455]提出了一种基于解耦采样的硬件扩展，并将其思想应用于运动模糊和景深效果。样本有 5 个维度：2 个表示亚像素位置，2 个表示镜头位置，1 个表示时间。可见性和着色会分别进行采样。对于一个给定可见性的样本，“解耦映射（decoupling mapping）”决定了其所需的着色样本。Liktor 和 Dachsbacher [1042, 1043]以类似的方式提出了一个延迟着色系统，其中着色样本会在计算和随机光栅化期间被缓存下来。运动模糊和景深等效果并不需要很高的采样率，因此其着色计算的结果可以被重复使用。Clarberg 等人[271]提出了在纹理空间中计算着色的硬件扩展。这些方法消除了四边形过度绘制的问题，因此允许出现尺寸很小的三角形。由于着色是在纹理空间中进行计算的，因此在从纹理中查找着色值的时候，像素着色器可以使用双线性滤波或者其他更加复杂的滤波器。这允许通过降低纹理分辨率来减少着色成本。对于一些低频信息，这种技术的效果通常会很好，因为可以使用一些过滤技术。Andersson 等人[48]采用了一种不同的方法，它被称为纹理空间着色（texture-space shading）。对每个三角形进行视锥体剔除测试和背面剔除测试，然后将可绘制的表面应用到输出目标的相应区域中，再根据其 (u, v) 参数化表示来对该三角形进行着色。同时使用一个几何着色器，来计算相机视野中每个可见三角形的尺寸。这个尺寸值将会用于确定该三角形需要插入到哪个层级中，这个层级结构类似于 mipmap。通过这种方式，对于物体执行的着色计算量与其屏幕覆盖率有关，如图 20.14。他们使用随机光栅化来渲染最终的图像。生成的每个

片元将会从纹理中查找其对应的着色颜色。同样，这些计算出来的着色值也可以重复用于运动模糊和景深效果。

图 20.14：对象空间纹理着色。左边是最终的渲染结果，包括了动态模糊。中间则展示了图表中的可见三角形。在右边，每个三角形会根据其屏幕覆盖率，插入到适当的 mipmap 层级中，以便在最终的基于相机的光栅化 pass 中进行使用。[48]

Hillesland 和 Yang [747, 748]基于纹理空间着色的概念，以及类似于 Liktor 和 Dachsbacher 的缓存概念。他们将几何形状绘制到最终视图，并使用一个计算着色器来填充基于一个类似 mipmap 的结构，这个结构中存储了对象的着色结果，再次渲染几何形状并访问这个纹理，从而显示出最终的着色结果。在第一个 pass 中还会保存一个包含三角形 ID 的可见性缓冲区，这样他们的计算着色器可以在稍后访问顶点属性，并进行插值。他们的系统还考虑了随时间的一致性，由于着色是在对象空间中进行的，所以相同的区域会与每帧相同的输出纹理位置相关联。对于给定 mipmap 层级的表面区域，如果其着色结果之前就已经计算过了，并且不是太旧，那么会将其重复使用而不是重新进行计算。最终的结果会因为材质、光照和其他因素而有所不同，但是他们发现，以 60 FPS 的帧率，每帧重复使用一个着色样本所带来的误差可以忽略不计。他们还确定，这个 mipmap 层级不仅可以通过屏幕覆盖区域的大小来进行选择，还可以通过其他一些因素的变化来进行选择，例如在某个区域上的法线变化。更高层级的 mipmap 意味着每个屏幕片元的着色计算量会更少，他们发现这可能会带来相当大的开销节省。

Baker [94]描述了 Oxide Games 为游戏《奇点灰烬》所开发的渲染器。它受到了 Reyes 的启发，尽管在具体实现细节上存在很大的不同，并且每个模型都会作为一个整体，来使用纹理空间中的着色。物体表面可以覆盖任意数量的材质，这些材质可以通过使用蒙版来进行区分。他们的过程如下：

- 若干个巨大的“主”纹理，尺寸为 $4k \times 4k$，每通道 16 bit。该纹理会用于着色。

- 所有物体都会被评估。如果在可视范围内，则计算物体在屏幕上的估计面积。
- 这个面积值用于为每个物体分配主纹理的比例。如果总的请求面积要大于纹理空间的面积，则会按比例进行缩小，从而使得请求能够匹配。
- 基于纹理的着色在一个计算着色器中执行，每个材质会依次附加到模型表面上。每个材质的结果都会累积在指定的主纹理中。
- 根据需要计算主纹理的 mipmap 层级。
- 对场景物体进行光栅化，并使用主纹理来对它们进行着色。

允许每个物体使用多种材质，可以实现这样的效果：有一个单独的地形模型，上面覆盖了泥土、道路、地面、水、雪、冰等，每一种效果都有自己的材质 BRDF。如果需要的话，可以在像素级别和着色器级别上运行抗锯齿技术，因为在着色期间，可以访问有关物体表面面积及其与主纹理关系的完整信息。这种能力允许该系统能够稳定地处理具有极高镜面反射率的模型。由于着色结果是作为一个整体被附加到物体上的，并不会考虑可见性，因此这个着色过程和光栅化过程也可以以一个不同的帧率来进行计算。在以 60 FPS 进行光栅化的游戏中，或者 90 FPS 的虚拟现实系统中，他们发现 30 FPS 的着色帧率就已经足够了。拥有这种异步着色的能力意味着，即使着色器的负载变得太高，几何图形的帧率也可以保持不变。

在实现这样一个系统的时候，有以下几个挑战。与常见的游戏引擎相比，大约会有两倍的批次需要被发送到管线中，因为每个物体的“材质四边形”在对象着色步骤中，是使用一个计算着色器进行处理的，然后在光栅化期间来绘制物体。然而，大多数批次都很简单，使用 DirectX 12 和 Vulkan 等现代图形 API 有助于消除这种开销。如何根据物体的大小来分配主纹理，将会对图像质量产生显著影响，对于屏幕上一些较大的物体，或者纹素密度不同的物体（例如地形）可能会出现一些问题。需要执行一个额外的拼接过程，用于在主纹理中保持不同分辨率地形块之间的平滑过渡。一些屏幕空间中的技术实现起来比较困难，例如屏幕空间环境光遮蔽等。与原始的可见性缓冲一样，影响物体形状的动画效果必须进行两次处理，分别用于着色计算和光栅化。物体进行着色计算，然后发现被遮挡，这是性能浪费的来源之一。对于深度复杂度较低的应用程序而言（例如实时策略游戏 RTS），这种成本可能相对较低。与复杂的延迟着色器不同，每个材质计算起来很简单，并且着色是在整个物体的图表上完成的。而那些具有简单着色器的物体（例如粒子和树木），从这种技术中能够获得的好处将会很少。为了进一步提高性能，这些效果可以使用前向着色来进行渲染。如图 20.15 所示，能够处理大量光源，可以使得渲染场景变得更加丰富。

我们将在这里结束对于高效着色话题的讨论。本章节中，我们只涉及了在不同应用中，提高渲染速度和结果质量的一整套专门技术。我们的目标是介绍用于加速着色的流行算法，并解释它们是如何工作的，以及产生的原因。随着图形硬件功能和图形 API 的发展、屏幕分辨率的不断提高、以及艺术工作流程和其他随着时间的推移而变化的因素，这些高效的着色技术将会继续以新的、可能的、意想不到的方式进行研究和开发。

图 20.15：游戏《奇点灰烬》的场景被大约一千个光源所照亮。每辆车和每颗子弹都至少包括一个光源。

如果你通读本书到这里，那么你就已经掌握了现代交互式渲染引擎中主要算法的工作原理。我们的目标之一是让您快速了解，以便您能够理解该领域中的论文和演讲。如果您希望了解这些技术是如何在一起工作的，我们强烈建议您阅读 Courreges [293, 294]和 Anagnostiou [38]各有关不同商业渲染器的优秀论文。在此之后，接下来的几个章节将会对几个领域进行深入研究，例如虚拟现实和增强现实的渲染、相交测试和碰撞检测的一些算法、以及图形硬件的架构特性等。

补充阅读和资源

那么，在延迟着色、前向着色、分块着色、聚类着色、可见性缓冲区的各种混合方法中，究竟哪一种是最好的呢？每一种方案都有自己的独特优势，因此这个问题的答案是：视情况而定（it depends）。硬件平台、场景特征、光照模型和设计目标等因

素，都可以对方案的选择产生影响。作为一个起点，我们推荐 Pesce 的论文[1393, 1397]，他对各种方案的有效性和权衡进行了广泛讨论。

SIGGRAPH 课程“实时多光源管理与聚类着色的阴影（Real-Time Many-Light Management and Shadows with Clustered Shading）”[145, 1331, 1332, 1387]，介绍了分块着色和聚类着色技术，以及它们在延迟着色和前向着色中的使用，还涉及了一些相关主题，例如阴影映射以及在移动设备上实现光源分类。Stewart 和 Thomas [1700]的早期报告阐释了分块着色，并展示了大量的计时结果，揭示了各种因素是如何影响性能表现的。Pettineo 的开源框架[1401]对分块前向系统和分块延迟系统进行了对比，并包括了各种 GPU 上的结果。

关于具体的实现细节，在 Zink 等人[1971]有关 DirectX 11 的书中，有大约 50 页内容关于延迟渲染，并包括了许多代码示例。NVIDIA GameWorks 代码示例[1299]包括了延迟渲染中的 MSAA 实现。Mikkelsen 的论文[1210]，以及 Ortegren 和 Persson [1340]在《GPU Pro 7》一书中，描述了基于 GPU 的、用于分块着色和聚类着色的现代系统。Billeter 等人[144]给出了实现分块前向着色的编码细节，Stewart [1701] 介绍了在计算着色器中执行分块裁剪的代码。Lauritzen [990]提供了一个分块延迟渲染的完整实现，Pettineo [1401]构建了一个框架，将其与分块前向着色方案进行了比较。Dufresne [390]提供了一个包含聚类前向着色的演示代码。Persson [1386]提供了一个基本的世界空间、聚类前向渲染解决方案的代码。最后，van Oosten [1334] 讨论了各种优化方法，并给出了一个演示系统，其中实现了不同形式的聚类、分块以及 vanilla 前向渲染，并展示了它们之间的性能差异。

Chapter 21 Virtual and Augmented Reality 虚拟现实和增强现实

> Philip K. Dick——“Reality is that which, when you stop believing in it, doesn’t go away.”

> 菲利普·K·迪克——“现实就是，即使你不再相信它，它也不会消失。”（美国的科幻小说作家，有很多小说被改编成电影，例如《银翼杀手》；1928——1982）

虚拟现实（virtual reality，VR）和增强现实（augmented reality，AR）是一种试图以现实世界的方式来刺激人类感官的技术。在计算机图形学领域，增强现实会将数字合成的物体与我们周围的真实世界结合在一起；而虚拟现实则是试图完全取代现实世界，如图 21.1 所示。本章节将会重点介绍这两种技术中的渲染技术，这两种技术有时候会使用总称“XR”来将它们组合在一起，其中的 X 可以代表任何字母。本章节的重点将放在虚拟现实技术上，因为直到本文撰写的时候，这种技术已经得到了更加广泛的应用。

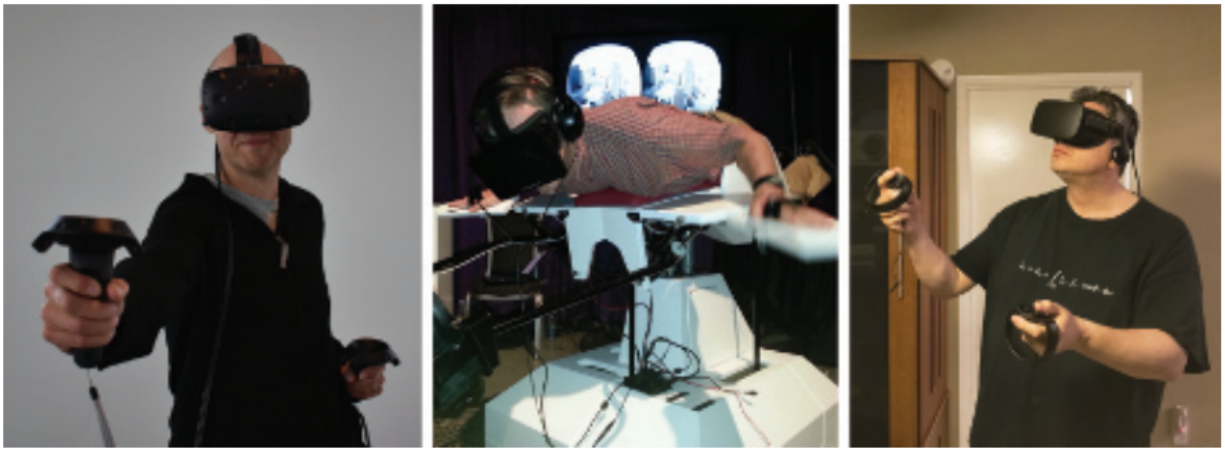


图 21.1：三位作者使用了各种不同的 VR 系统。从左到右：Tomas 在使用 HTC Vive；Eric 使用 Birdly 模拟器，可以让使用者像鸟一样飞翔；Naty 在使用 Oculus Rift。

渲染只是这些领域中的一小部分。从硬件的角度来看，它们也会使用某种类型的 GPU，这是系统中一个较为容易理解的部分。这个系统还面临许多其他的挑战，例如：准确且舒适的头部追逐传感器[994, 995]，高效的输入设备（可能还会带有触觉

反馈系统或者眼球追踪控制），舒适的头戴设备和光学设备，以及令人信服的音频等等。由于需要对性能、舒适度、移动自由度、价格和其他的一些因素进行平衡，使得它成为了一个要求很高的设计空间。

本章节将会专注于交互式渲染，以及这些技术影响图像生成的方式，首先我们会简要调查当前可用的各种虚拟现实系统和增强现实系统。然后对一些系统的 SDK 和 API 的功能与目标进行讨论。我们会以介绍特定的计算机图形技术来结束本章节，为了提供最佳的用户体验，应该避免使用这些技术，或者避免对这些技术进行修改。

21.1 设备和系统概述

除了 CPU 和 GPU 之外，用于图形功能的虚拟现实和增强现实设备，还可以归类为传感器或者显示器。其中传感器包括检测用户旋转信息和位置信息的追踪器，以及各种输入设备。在显示器方面，一些系统直接使用手机屏幕，这种方法在逻辑上被划分成了两部分。一些专用系统通常会有两个独立的显示器，这里的显示器是指用户在虚拟现实系统中用于看到画面的屏幕。而对于增强现实来说，会通过使用一些特殊设计的光学装置，来将虚拟世界与现实世界的视野相结合。

虚拟现实和增强现实是一个古老的领域，在几十年前就已经出现了，但是近年来由于系统成本的大幅降低，它们得到了爆炸式地发展，这在很大程度上直接或者间接地归功于各种手机和主机技术的出现[995]。手机当然也可以用于沉浸式体验，有时候效果还会出奇地好。手机可以直接放置在头戴式显示器（head-mounted display，HMD）内部，这类设备有很多，从简单的查看器（例如 Google Cardboard），再到那些手部自由并且提供额外输入的设备（例如 GearVR）。手机的重力、磁北（magnetic north）和其他机制的方向传感器，可以用于确定显示屏的方向。方向（orientation），有时候也被称为姿态（attitude），它有三个自由度，即偏航（yaw）、俯仰（pitch）和滚转（roll），如章节 4.2.1 中所述。相关 API 可以将方向返回为一组欧拉角、旋转矩阵或者四元数。从现实世界中生成的内容，例如固定视角的全景图和全景视频，可以很好地与这些设备一起工作，因为根据用户方向来展示正确的二维视图，这个操作的实现成本相当低。

> 许多手机的惯性测量单元包含六个自由度（3 个用于方向 +3 个用于位置），但是位置信息的追踪误差会迅速累积。

移动设备的计算能力相对有限，同时还会有对 GPU 和 CPU 硬件的功耗需求，这些因素限制了它们的使用范围和表现能力。一些捆绑式的虚拟现实设备，用户的头戴式设

备会通过一组电线连接到一台位置固定的计算机上，虽然这样做限制了移动自由，但是可以使用更加强大的处理器。

这里我们将简要介绍两种系统的传感器，即 Oculus Rift 和 HTC Vive，二者都能够提供六个自由度（6-DOF）的追踪：方向和位置。其中 Rift 通过三个独立的红外摄像机，来追踪 HMD 和控制器的位置。当头戴设备的位置信息由固定的外部传感器进行确定时，这种方式被称为由外向内的追踪（outside-in tracking），头戴设备外部的有一组红外 LED 阵列，可以对其进行跟踪。Vive 则使用一对“灯塔（lighthouse）”，它会以快速的间隔来向房间内发射一些不可见的光线，头戴设备中的传感器和控制器会检测到这些不可见光，从而对自身的位置进行三角测量。这种方式被称为由内向外的追踪（inside-out tracking），其中传感器是 HMD 中的一部分。

与 PC 上的鼠标和键盘不同，VR 中的标准输入设备是一个手持控制器，它是可跟踪的，并且能够跟随用户进行移动。基于一系列广泛的技术，已经为 VR 开发了许多其他类型的输入设备。这些设备包括：手套或者其他肢体和身体的追踪设备；眼动追踪设备；以及模拟原地（in-place）运动的设备，例如压力垫，单向跑步机或者全向跑步机，固定的自行车和人体大小的仓鼠球（hamster ball，译者注：仓鼠球是一种透明塑料球，让仓鼠在里面滚动娱乐）等。除了光学追踪系统之外，还探索了基于磁场、惯性、机械、深度探测和声学现象的追踪方法。

增强现实被定义为：将计算机生成的数字内容与用户现实世界的视图相结合。任何可以将文本数据叠加在图像上的平视显示（heads-up display，HUD）应用程序，都是增强现实的基本形式。2009 年推出的 Yelp 单片眼镜（Monocle），可以将商业用户的评分和距离叠加在摄像头的画面之上。移动端的谷歌翻译还可以使用翻译后的结果来替换原有的语言符号。《Pokemon GO》等一些游戏将虚构的生物放置于真实环境中。Snapchat 可以检测面部特征，并添加一些服装元素或者动画效果。

对于合成渲染而言，混合现实（mixed reality，MR）是增强现实的一个子集，其中真实世界可以和三维虚拟内容进行实时融合和交互[995]。混合现实的一个经典用例是手术，在手术过程中，患者器官的扫描数据与外部身体的摄像头画面合并在一起。这个场景假设了一个具有相当计算能力和精度的系留系统（tethered system，译者注：即使用实体线材进行连接，以供数据传输）。另一个例子是与虚拟袋鼠玩“捉人”游戏，现实世界的墙壁可以隐藏你的对手。在这种情况下，移动性将会变得更加重要，而登记（registration）或者其他影响质量的因素则不那么重要。

这个领域中常用的一项技术，是在 HMD 前端安装一个摄像机。例如：每个 HTC Vive 都有一个前置摄像头，开发者可以对其进行访问。这个这个摄像头所拍摄的画面

可以显示在 VR 的显示屏上，并且数字合成的图像可以与之融合。这种形式有时会被称为穿透式 AR 或者穿透式 VR，或者叫做介导现实（Mediated Reality）[489]，即用户不会直接观察到现实环境。这种使用视频流方法的一个优点在于，它可以更好地控制虚拟物体与真实物体的融合；其缺点在于现实世界的画面可能会有一定的滞后，因为摄像机拍摄的画面在经过处理之后，再显示到屏幕上存在一定的延迟。Vrvana 的 Totem 和 Occipital 的 Bridge，都是使用这种类型头戴式显示器的例子，它们都属于 AR 系统。（译者注：Quest2 和 Pico Neo3 也都支持）

在撰写本书的时候，微软的 HoloLens 是最著名的混合现实系统。这是一个不受束缚的系统，其中 CPU、GPU 和微软所说的全息处理单元（holographic processing unit，HPU）都被内置在头戴式设备中。其中的 HPU 是一款定制芯片，它由 24 个数字信号处理核心组成，其功耗低于 10 瓦。这些核心用于处理来自一个类似于 kinect 摄像头的世界数据，这个摄像头可以对真实世界中的环境进行观察。这个视图画面与其他传感器（例如加速度计）一起，执行由内而外的追踪，它还有一个额外的优势，即不需要使用灯塔、QR 码（又名二维码基准）或者其他的外部元素。HPU 可以对一组有限的手势进行识别，这意味着不需要使用额外的输入设备就可以实现基本的交互。在扫描环境的同时，HPU 还可以提取深度信息并生成几何数据，例如代表世界表面的平面和多边形。然后，这些几何图形还可以用于碰撞检测，例如：让虚拟物体立在现实世界的桌面上。

通过在真实世界中创建的路径点（被称为空间锚点 spatial anchor），并使用 HPU 进行追踪，可以实现更大范围内的活动。虚拟物体的位置会相对于特定的空间锚点来进行设置[1207]。随着时间的推移，设备对于这些锚点位置的估计也会得到一定的改善。这些数据还可以进行共享，这意味着少数用户可以在相同的位置上看到相同的内容。还可以对锚点进行自定义，从而使得不同位置上的用户可以在同一个模型上进行协作。

Hololens 具有一对透明的屏幕，它允许用户看到现实世界中的物体，以及投射到这些屏幕上的任何数字画面。请注意，这与手机使用的增强现实有所不同，后者的世界画面是使用摄像头进行捕获的。使用透明屏幕的一个好处是，现实世界本身不会带来任何的延迟或者显示问题，并且不会消耗处理能力。然而，这种显示系统的一个缺点在于，显示在屏幕上的虚拟内容只能在用户所看到的世界画面上增加亮度。例如：一个黑暗的虚拟物体并不会遮挡住背后更亮的现实世界物体，因为显示在透明屏幕上的亮度只能增加不会减少，这会使得虚拟物体具有一种半透明的感觉。HoloLens 还有一个 LCD 调光器，可以帮助避免这种影响。通过适当的调整，这个系统可以有效地显示出与现实世界融合的三维虚拟物体。

苹果的 ARKit 和谷歌的 ARCore，可以帮助开发者开发手机和平板电脑上的增强现实应用。通常的做法是显示单一的视图画面（而非立体的），显示设备与眼睛保持一定距离。这些虚拟物体可以是完全不透明的，因为它们会覆盖在摄像机的世界画面之上，如图 21.2 所示。对于 ARKit 而言，通过使用设备的动作感应硬件，以及一组在相机中可见的显著特征，来实现由内而外的追踪。对这些特征点进行逐帧追踪有助于精确确定设备当前的位置和方向。与 HoloLens 一样，它可以检测到现实世界中的水平表面和垂直表面，并确定其范围，然后再将这些信息提供给开发人员[65]。

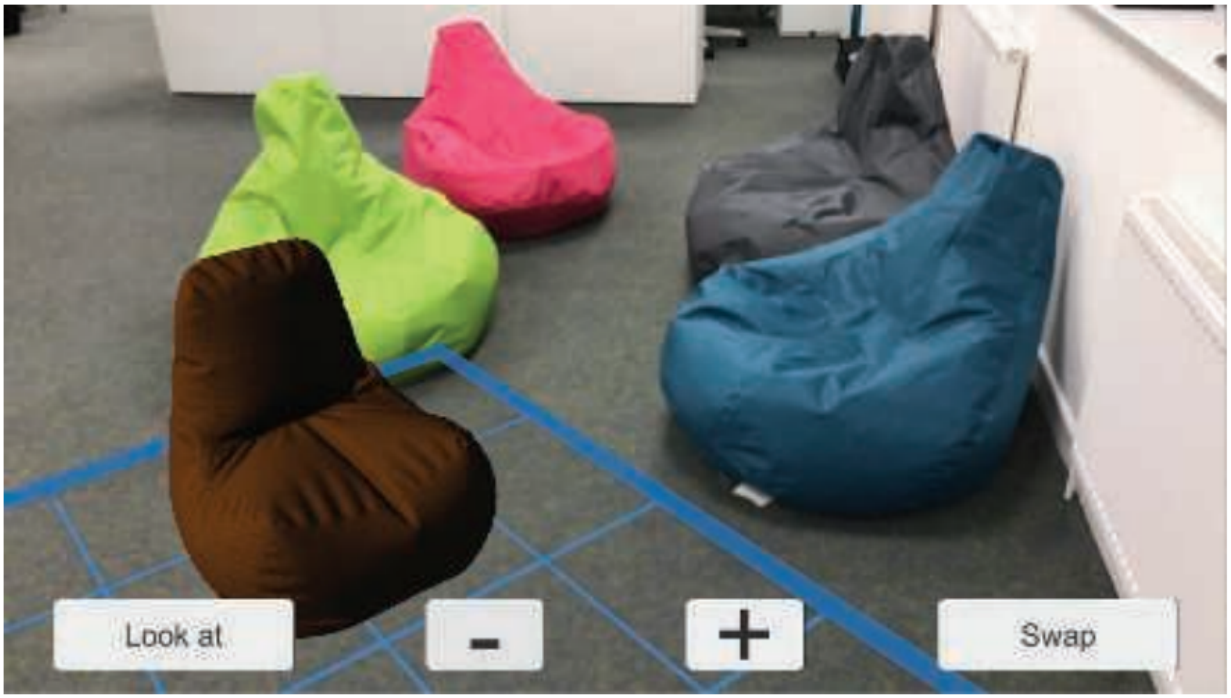


图 21.2：来自 ARKit 的图片。它可以检测到地面，并以蓝色网格进行显示。其中最接近的棕色豆袋椅是一个叠加在地面上的虚拟物体。虽然它缺少阴影效果，但是这个阴影效果也是可以实现并添加到物体上的，并且可以与场景相混合。

英特尔的 Alloy 项目是一款无绳式（untethered）的头戴式显示器，与 HoloLens 一样，它具有一个传感器阵列，可以探测到房间中的大型物体和墙壁。然而与 HoloLens 不同的是，这款 HMD 无法让用户直接看到现实世界。然而，其感知周围环境的能力可以提供英特尔所谓的“融合现实”功能，即现实世界中的物体可以在虚拟世界中以一种令人信服形式的存在。例如：用户可以伸手到虚拟世界中的控制台上，而真实世界中的手则会触摸到现实世界中的桌子。

虚拟现实和增强现实的传感器和控制器正在快速发展，许多迷人的技术正在以极快的速度出现。这些产品有望带来干扰更少的头戴式设备、更强的移动性和更好的体验。例如：谷歌的 Daydream VR 和高通的 Snapdragon VR 头戴设备都是无绳式的，并且使用了由内而外的位置追踪，不需要外部传感器或者其他定位设备。惠普、Zotac

和 MSI 的一些系统，可以将计算机背在背上，从而提供更加强大的计算能力。英特尔的 WiGig 无线网络技术使用了一个短程的 90 GHz 无线电，来将图像从 PC 发送到头戴设备中。另一种方法是在云端进行昂贵的光照计算，然后将这些压缩信息发送给头戴设备，使用其内置的、更轻量的、功能更弱的 GPU 来进行渲染[1187]。一些软件方法为虚拟画面和真实世界的融合开辟了新的途径，例如：获取点云数据，然后再对这些点云数据进行体素化，并以交互速率来渲染这些体素化表示[930]。

本章节的主要内容集中在显示器及其在 VR 和 AR 中的应用。我们首先会介绍一些图像在屏幕上显示的物理机制，以及这些物理机制将会涉及到的一些问题。之后，本章节将会继续介绍一些 SDK 和硬件系统，它们提供了一些功能，可以简化编程并增强用户对场景的感知。然后我们会介绍这些不同的因素是如何影响图像生成的，并讨论了一些图形技术需要进行怎样的修改，或者可能要完全避免使用其中的一些图形技术。最后，我们讨论了一些渲染方法和硬件增强，以提高效率和改善参与者的体验。

21.2 物理元素

本小节将会介绍现代 VR 系统和 AR 系统的各个组成部分，以及它们各自的特点，尤其是那些与图像显示有关的系统。本小节中的信息提供了一个框架，用于理解设备供应商所提供的工具及其背后的逻辑。

21.2.1 延迟

减轻延迟的所带来的影响在 VR 系统和 AR 系统中尤其重要，它通常是最为关键的问题[5, 228]。我们在第 3 章中讨论了 GPU 是如何隐藏内存延迟的，但是这种类型的延迟是由纹理获取等操作所引起的，并且只针对了整个系统中的一小部分。而这里我们指的是整个系统的“动显（motion-to-photon，MTP）”延迟。也就是说，假设你开始将头转向左侧，从你的头朝向一个特定的方向开始，直到显示出从这个方向所产生的视图，这之间具体要经过多长时间？这个过程中的每个硬件都有处理成本和通信成本，从检测用户输入开始（例如：头部朝向的改变）再到响应（将新图像显示在屏幕上），这些加起来有几十毫秒的延迟。

在一个普通显示器系统中（即没有连接到头部的显示器），延迟最坏的情况是十分令人讨厌的，它破坏了交互性和连接感（沉浸感）。对于增强现实和混合现实应用而言，较低的延迟将会有助于提高“像素粘性”，即场景中的虚拟物体与现实世界的贴合程度。系统中的延迟时间越长，虚拟物体相对于现实世界中的物体看起来就会越像是在游动或者漂浮。在沉浸式虚拟现实中，显示器是唯一的视觉输入，延迟可能会带来更加严重的后果。延迟会引起人体的不适，虽然它并不是真正的疾病，但是这种不适

也会被称为模拟病（simulation sickness），它会引起出汗、头晕、恶心、甚至更糟的不适感。如果你开始感到这种不适感，你需要立即摘下 HMD，因为我们是无法“强制克服”这种不适感的，一直强忍只会让我们变得更加不适[1183]。引用 Carmack（译者注：约翰·卡马克）的话[1183]：“别太勉强。我们不需要在演示室里清理病人。”实际上，使用 VR 设备很少会引发真正的呕吐，但是尽管如此，它的影响仍然是十分严重的，它会使人感觉很虚弱，并且可能会持续上一整天。

当显示出的图像与用户的期望、或者其他感官的感知不匹配时（例如负责平衡和运动的内耳前庭系统），就会出现 VR 中的模拟病。头部运动与正确匹配的显示图像之间的延迟越低越好。一些研究指出，低于 15 ms 的延迟通常是人们无法察觉的。超过 20 ms 的延迟可以被明确感知，并且会产生一些有害的影响[5, 994, 1311]。相比之下，从鼠标开始移动到显示出图像，电子游戏通常会有 50 ms 或者更大的延迟时间，关闭垂直同步通常则会有 30 ms（章节 23.6.2）。虚拟现实系统的帧率一般为 90 FPS，对应的帧时间为 11.1 ms。在常见的桌面系统中，通过电缆将画面传输到显示器中大约需要 11 ms。因此，即使我们可以在 1 ms 内完成渲染，但是仍然会有 12 ms 的延迟。

有许多基于应用程序的技术，可以防止或者减轻这种不适感[1089, 1183, 1311, 1802]。这些方法有些会通过最小化视觉流动（例如在向前移动的时候，不诱使用户向四周进行观察，避免上楼梯等操作），还有一些会使用心理方法（例如播放一些环境音乐，或者渲染一个代表用户鼻子的虚拟物体）[1880]。更加柔和的颜色和更加混合的灯光也有助于避免模拟病。使得系统的响应与用户的行动和期望相匹配，是提供舒适虚拟现实体验的关键所在。可以让所有物体都对头部运动做出响应；不要对相机进行拉近拉远操作，或者以其他方式来改变视野；适当缩放虚拟世界；不要将相机的控制权从用户手中夺走等。在用户周围设置一个固定的视觉参考物，例如汽车驾驶舱或者飞机驾驶舱，也可以帮助减少模拟病。对用户应用视觉加速可能会引起一些不适，因此最好是使用恒定的速度。一些硬件解决方案也被证明可能是有用的，例如：三星的 Entrim 4D 耳机会发出影响前庭系统的微小电脉冲，这使得能够将用户的视觉系统与平衡感相匹配。时间将会证明这项技术是否有效，但是这也正表明了，人们正在进行很多的研究和开发工作，来减轻模拟病所带来的影响。

位姿追踪（tracking pose），是指对观察者在现实世界中的头部方向和头部位置（如果可以的话）进行追踪和定位。这个位姿信息会用于构建渲染所需的相机变换矩阵。可以在一帧开始的时候使用粗略的位姿估计，用于执行一些模拟（译者注：一般引擎的生命周期都是按照物理、输入、逻辑更新、渲染的顺序，例如 Unity 引擎），例如对角色和环境中的元素进行碰撞检测。在渲染即将开始的时候，此时可以检索一

个新的位姿预测，并用于更新相机的视图。这种预测将会更加准确，因为它进行检索的时间较晚，持续的时间也更短。当图像即将进行显示的时候，仍然可以检索另一个更加准确的位姿预测，并使用这个信息对生成的图像进行扭曲，从而更好地匹配用户所在的位置。虽然之后的每次预测，都无法对基于早期不准确预测的计算进行完全补偿，但是尽可能地使用这些位姿预测，可以显著改善整体的体验。各种平台的硬件增强（hardware enhancement）提供了在需要时快速查询和获取头部位姿（更新后的）的能力。

除了视觉效果之外，还有一些其他元素可以让玩家与虚拟环境之间的交互变得更加令人信服，但是如果生成的图像出现错误，那么最多就只能给用户带来不愉快的体验了。最小化延迟和提高应用程序的真实感，可以帮助实现沉浸感（immersion）或者存在感（presence），这种情况下可以隐藏 UI 界面，使得参与者感觉自己是虚拟世界中的一部分一样。

21.2.2 光学

通过精确设计的物理光学元件来将头戴式显示器的内容映射到人眼视网膜上的相应位置，这是一项成本很高的主张和方法。现如今的虚拟现实显示系统之所以经济实惠，是因为 GPU 产生的图像可以在单独的后期处理中被一定程度的扭曲，这样它们就能够正确地到达人类的眼睛，而不是使用昂贵的光学镜片组合。

虚拟现实系统的镜头会向用户呈现具有一个枕形畸变（pincushion distortion）的宽 FOV 图像，图像的边缘会向内弯曲；这种效果可以通过使用桶形畸变（barrel distortion）对生成的图像进行反向变形来进行抵消，如图 21.3 右侧所示。光学系统通常也会受到色差（chromatic aberration）的影响，在这种情况下，透镜会导致颜色分离，就像棱镜一样。这个问题可以通过供应商的软件来进行补偿，即通过生成具有反向色差的图像，这是在“另一个方向”上的人工色差。当通过 VR 的光学系统进行显示的时候，这些分离的颜色可以正确地被结合在一起。这种校正手段可以在扭曲图像的橙色边缘处观察到。

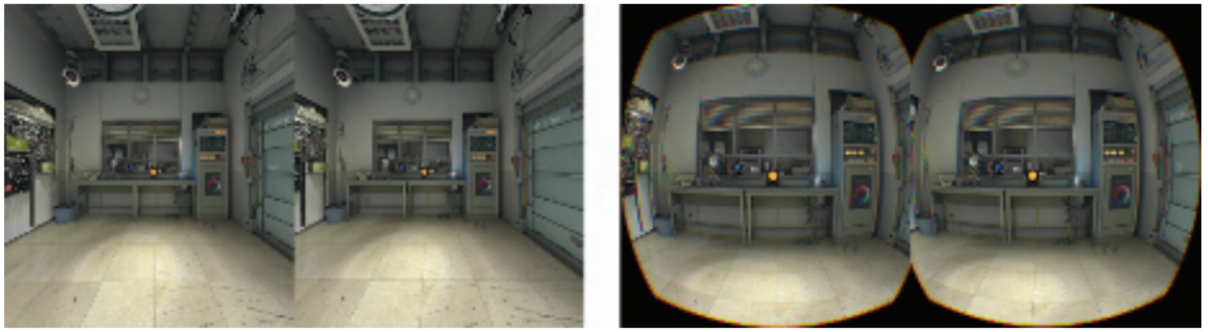


图 21.3：在 HTC Vive 上显示的原始渲染目标（左）和它们的扭曲版本（右）。[1823]

有两种显示类型，分别是滚动显示和全局显示[6]。对于这两种显示方式，图像都是在串行流（serial stream）中发送的。在滚动显示（rolling display）中，接收到的比特流会立即进行逐行显示。而在全局显示（global display）中，一旦接收到了一个完整的图像，它就会在一个很短的时间内（burst）中显示出来。在虚拟现实系统中，每一种显示方式都有自己的优缺点。其中全局显示必须等待整个图像被渲染结束之后才能进行显示；相比之下，滚动显示则可以最小化延迟，因为渲染出来的结果会在可以使用时立即进行显示，例如：如果图像是按条生成的，那么每条图像都可以在渲染之后立即进行发送，即“与光束赛跑（racing the beam）”[1104]。其缺点在于，不同的像素会在不同的时间被照亮，因此根据视网膜和显示器之间的相对运动，图像看起来可能会不稳定。这种不匹配对于增强现实系统而言尤其令人不安。好消息是，合成器（compositor）通常可以通过在扫描线块上，对预测的头部位姿进行插值，从而进行一些补偿。这主要解决了在快速头部旋转时可能发生的抖动或者剪切，尽管它无法对场景中移动的物体进行修正。

全局显示则不存在这种时间问题，因为图像在显示之前必须被完整渲染。相反，全局显示的挑战是技术性的，因为在一个短时间内的突发显示，会将多种显示选项排除在外（译者注：对显示面板的刷新速度有要求）。有机发光二极管（organic light-emitting diode，OLED）显示器是目前全局显示的最佳选择，因为它们的显示速度够快，可以跟上 VR 中流行的 90 FPS。

21.2.3 立体视觉

如图 21.3 所示，两幅图像会被偏移，意味每只眼睛的视角都不同。这样做会引起立体视觉（stereopsis），即使用两只眼睛来感知深度。虽然立体视觉十分重要，但是它会随着距离的增加而减弱，而且它并不是我们感知深度的唯一方式，例如：当我们在一个标准显示器上观察图像的时候，我们根本不使用到立体视觉。物体的大小、纹理模式的变化、阴影、相对运动（视差）以及其他视觉上的深度暗示，这些因素导致我们只需要一只眼睛就可以正常工作。

眼睛必须要进行一定的形状调整才能使得物体聚焦，这被称为调节需求（accommodative demand）。例如：Oculus Rift 的光学系统相当于在距离用户约 1.3 米的地方设置了一个屏幕。眼睛还需要向内进行一定的转动，才能聚焦在一个物体上，这被称为会聚需求（vergence demand），如图 21.4 所示。在现实世界中，眼睛会改变晶状体的形状，同时会向内进行转动，这种现象被称为调节–辐辏反射（accommodation–convergence reflex）。对于一个显示器而言，调节需求是恒定

的，但是随着眼睛聚焦于不同感知深度的物体上，会聚需求会发生变化。这种不匹配会导致眼睛变得疲劳，因此 Oculus 建议将用户长时间看到的物体，放置在 0.75 到 3.5 米的距离上[1311, 1802]。这种不匹配也会在某些 AR 系统中产生感知效应，例如：用户此时可能会想关注现实世界中的一个远处物体，但是随后又必须重新关注位于眼睛附近固定深度的相关虚拟广告牌。一些团队正在研究和开发能够根据用户眼球运动，来调整感知焦距的硬件，这种硬件有时被称为自适应聚焦（adaptive focus）或者变焦（varifocal）显示器[976, 1186, 1875]。

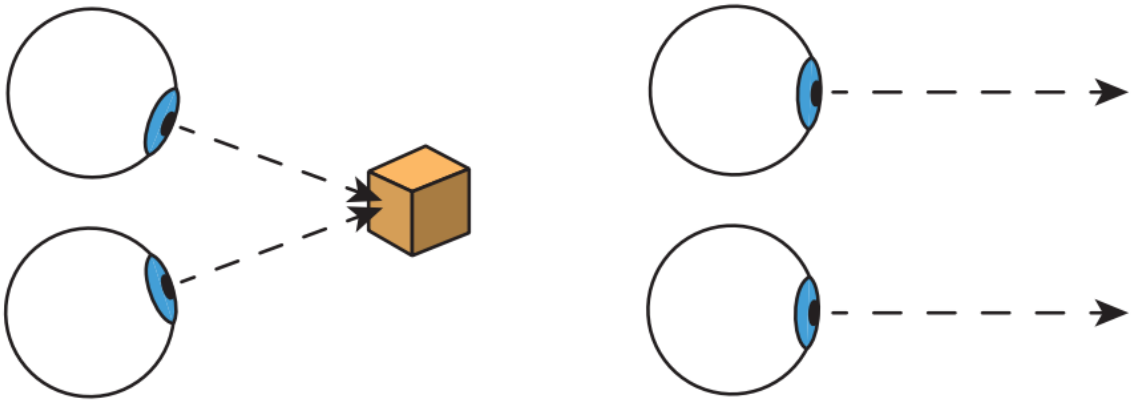


图 21.4：两个眼睛需要旋转多少角度才能看到一个物体，这就是会聚（vergence）。其中会聚现象（convergence）指的是眼睛向内聚焦在同一个物体上的运动，如左侧所示。而发散现象（divergence）指的是当眼睛改变视线，去观察远处物体时向外的运动，如右侧所示。在观察远处物体的时候，眼睛的视线实际上是平行的。

为 VR 和 AR 生成立体视觉的规则，与单显示器的系统完全不同，在单显示器的系统中，会使用一些技术（偏振光透镜、快门眼镜、多视图显示光学）来从同一个屏幕向每个眼睛展现不同的图像。而在 VR 中，每个眼睛都有一个单独的显示器，这意味着每只眼睛的位置都必须保证投射到视网膜上的图像与现实紧密吻合。两个眼睛之间的距离被称为瞳孔间距（interpupillary distance，IPD），在一项针对 4000 名美国陆军士兵的研究中，不同人的 IPD 范围集中在 52 毫米到 78 毫米之间，平均为 63.5 毫米[1311]。VR 系统和 AR 系统有一些校准方法，可以来帮助确定和调整适应用户的 IPD，从而提高图像质量和舒适度。系统的 API 一般都可以控制包含这个 IPD 的相机模型，但是最好避免对用户感知到的 IPD 进行修改来实现某些效果，例如：增大瞳孔间距可以增强对深度的感知，但是也可能会导致眼睛疲劳。

让开发人员从头开始实现头戴式显示器的立体渲染，这是一个不小的挑战。好消息是，为每个眼睛设置并使用合适的相机变换，其中的大部分过程都是由系统 API 进行处理的，这是下一小节的主题。

21.3 API 和硬件

让我们从这句话开始：始终使用系统供应商提供的 VR 软件开发工具包（software development kit，SDK）和应用程序接口（application programming interface，API），除非你有充足的理由不这样做。例如：你可能会认为自己编写的扭曲着色器的速度更快，看起来也差不多。但是在实践中，它很可能会导致用户感到严重的不适——我们无法知道这是否是正确的，除非进行广泛的测试。出于这样或者那样的原因，所有主流的 API 都删除了让应用程序来控制扭曲的接口，正确使用 VR 显示器是一个系统级的任务。为了对性能进行优化，同时保持质量，SDK 和 API 中做了许多细致的工程设计。本小节将会对不同厂商的 SDK 和 API 提供的功能支持进行讨论。

将三维场景的渲染图像发送到头戴式显示器的过程十分简单。在本小节中，我们将使用大多数虚拟现实和增强现实 API 中的常见元素来对其进行讨论，并在这个过程中注意不同供应商提供的特定功能。首先，我们将会确定下一个渲染帧的显示时间，通常都会有一些支持来帮助我们对这个时间延迟进行估计。这个值是必需的，这样 SDK 就可以计算出眼睛在看到这帧时的位置和方向。给定这个估计出来的延迟时间，API 会对位姿进行查询，其中包含了有关每个眼睛的相机设置信息。如果传感器也会追踪这些信息的话，那么这些信息至少会包括头部的方向和位置。OpenVR 的 API 还需要知道用户是站着的还是坐着的，这会影响使用哪个位置来作为原点，例如：原点位于追踪区域的中心还是用户的头部位置。如果这个预测是完美的，那么在头部达到预测位置和预测方向的那一刻，将会显示出这个渲染图像。通过这种预测方式，可以将延迟的影响最小化。

给定两个眼睛的预测位姿，我们通常需要将场景渲染到两个单独的目标中，这些渲染目标会被作为纹理发送到 SDK 的合成器（compositor）。

> 有些 API 会将一个渲染目标分为两个视图。

合成器负责将这些图像转换为头戴设备上观看效果最好的形式，同时合成器还可以将各种图层组合在一起，例如：如果需要单目平视显示器（monoscopic heads-up display）的话，即双眼的视图是完全相同的，那么可以提供一个包含该元素的单一纹理，来将其作为一个单独的图层，并在每个眼睛的视图上进行合成。这些纹理可以是不同的分辨率和不同的格式，合成器负责将它们转换到最终的眼睛缓冲区中。这样做可以进行一些优化，例如：可以动态降低三维场景图层的分辨率，从而节省渲染时间[619, 1357, 1805]，同时保持其他图层的高分辨率和高质量[1311]。一旦每个眼睛的图像都组合完成，则会由 SDK 来执行扭曲、色差和任何其他需要进行的处理，然后再显示结果。

如果我们选择依赖于 API 来实现这些功能，那么其实并不需要完全理解其中这些步骤背后的算法原理，因为供应商实际上已经为我们做了很多工作。然而，了解一些这方面的知识仍然是十分值得的，我们需要认识到最明显的解决方案并不总是最好的。首先我们考虑一下合成的方法，最高效的方法是直接将所有图层组合在一起，然后在单幅图像上应用各种修正措施。相反，Oculus 首先会对每个图层单独进行这些修正措施，然后再将这些扭曲的图层合成在一起，从而形成最终的显示图像。这种方法的一个优点在于，每个图层都会在自己的分辨率上被扭曲，这可以提高文本渲染的质量，因为单独对文本进行处理，意味着在扭曲过程中的重采样和过滤操作只需要关注文本的内容即可[1311]。

用户感知到的视野近似为圆形，这意味着我们并不需要渲染每个图像的边缘，即那些靠近角落处的一些像素。虽然这些像素仍然会出现在显示器上，但是它们几乎无法被观察者察觉。为了避免浪费时间来生成这些像素，我们可以首先渲染一个网格，来隐藏原始图像中的这些像素。这个网格会被渲染到模板缓冲区中作为蒙版，或者是直接渲染到最前面的 z-buffer 中，后续位于这些区域中的渲染片元将会在计算之前被丢弃。Vlachos [1823]报告称，这会使得 HTC Vive 的填充率降低约 17%，如参见图 21.5。Valve 的 OpenVR API 将这种预渲染蒙版（pre-render mask）称为“隐藏区域网格（hidden area mesh）”。

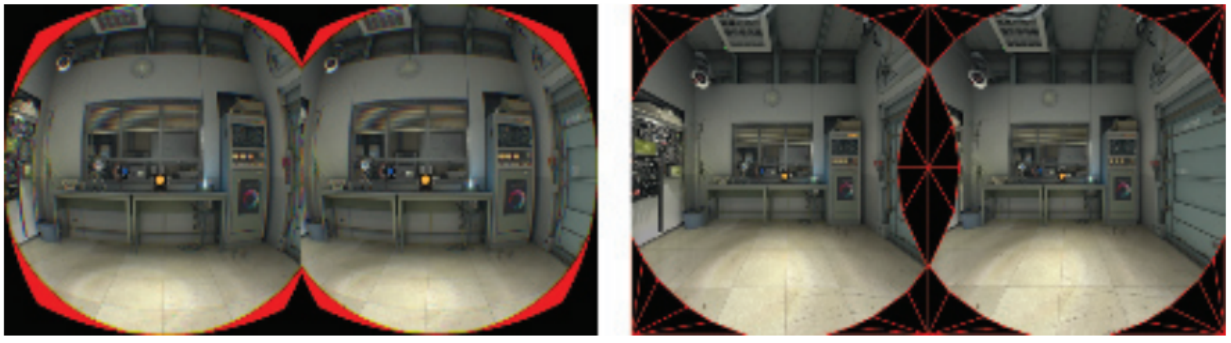


图 21.5：左侧：显示图像中的红色区域展示了渲染后被扭曲的像素，实际上 HMD 用户是看不到这些像素的。请注意，黑色区域位于转换后渲染图像的边界之外，因此不需要对黑色区域进行渲染。右侧：这些红色区域在渲染开始的时候，会提前使用一个红边网格来进行遮挡，这可以使得渲染（预扭曲）图像需要更少的像素着色计算[1311]。请仔细比较右侧图像和原始图像（图 21.3 的左侧图像）。

一旦我们生成了渲染图像，就需要对其进行扭曲，从而补偿光学系统所带来的畸变。这个概念实际上是定义了一个从原始图像到所需显示形状的重映射，如图 21.3 所示。换句话说就是，给定一个渲染图像上的像素样本，这个样本在显示图像中需要被移动到哪个位置？光线投射方法可以给出精确的答案，并可以根据波长进行调整

[1423]，但是对于大多数硬件而言，这种方法是不切实际的，其成本开销太高。一种方法是将渲染图像视为纹理，并通过绘制一个填充屏幕的四边形来运行一个后处理。像素着色器会针对这个输入像素，计算它在输出纹理上的对应精确位置[1430]。然而，这种方法的成本可能也会很高，因为这个像素着色器必须在每个像素上来计算扭曲方程。

将这个纹理应用到一个三角形网格上的效率更高，这个三角形网格的形状可以通过扭曲方程来进行修改，然后再进行绘制。只对网格扭曲一次无法纠正色差，因此还会使用三组独立的 (u, v) 坐标来用于扭曲图像，每个颜色通道都对应一个坐标[1423, 1823]。也就是说，网格中的每个三角形都会被渲染一次，但是对于屏幕上的每个像素而言，渲染图像都会在略有不同的位置上进行三次采样。这些红、绿、蓝通道值，就构成了输出像素的实际颜色。

我们可以先将渲染图像应用到一个均匀间隔的三角形网格上，然后再对三角形网格进扭曲，使其变成显示图像的形状，反过来也可以。将网格应用于显示图像，并将其扭曲回渲染图像的优点在于，这样可能会生成更少的 2×2 四边形，因为会包含较少数量的纤细三角形。在这种情况下，这个网格的顶点位置并不会被扭曲，而是会将其作为一个单元格来进行渲染，并且只有顶点的纹理坐标会被调整，从而对应用到网格上的图像进行扭曲。一个常用的网格尺寸是每个眼睛 48×48 个四边形，如图 21.6 所示。这个网格的纹理坐标会通过使用逐通道的（per-channel）、显示到渲染（display-to-render）的图像变换来计算一次。通过将这些值存储在网格中，因此在着色器执行过程中就不需要进行复杂的变换了。GPU 支持纹理的各向异性采样和各向异性滤波，可以用于生成清晰的显示图像。

图 21.5 右侧的立体渲染，被显示网格进行了扭曲。图片中央被移除的部分切片与扭曲变换生成显示图像的方式相对应，请注意，在图 21.5 左侧的显示图像中，两侧图像交汇的地方并没有出现切片空白。通过将显示的扭曲网格裁剪到用户可见的区域（如图 21.6 右侧所示），我们可以将最终扭曲 pass 的成本降低约 15%。

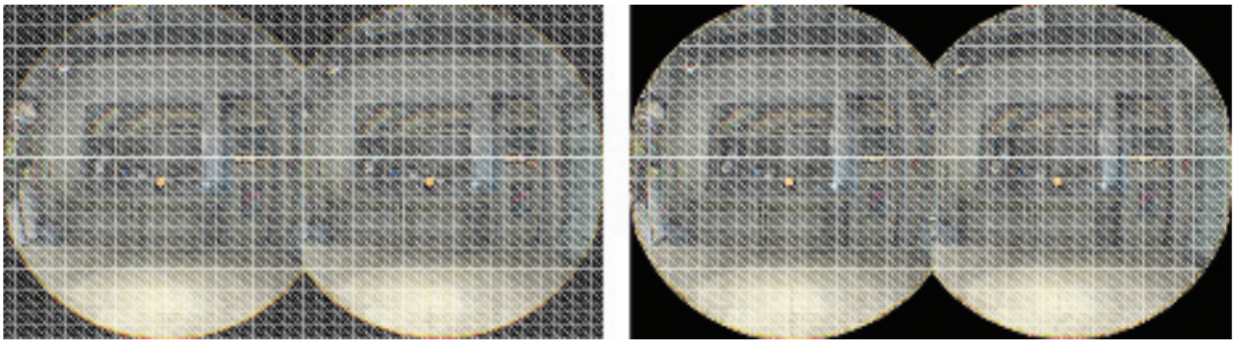


图 21.6：左边展示了最终显示图像的网格。在实践中，可以对这个网格进行裁剪，使其变为右侧的裁剪版本，因为绘制那些黑色区域上的三角形，并不会给最终图像增加任何东西。[1823]

这里对上述所描述的优化进行一下总结。我们首先会绘制一个隐藏区域网格，从而避免对那些不可见或者未使用的区域（例如中间的切片）中的片元进行计算。我们为两个眼睛都渲染一遍场景。然后，我们将这个渲染图像应用到一个均匀网格上，这个网格会被修剪，从而只包含那些相关的渲染区域。将这个网格渲染为一个新的目标，我们就可以获得最终的显示图像。在常见的虚拟现实系统和增强现实系统的 API 中，都内置了上述的全部优化或者部分优化。

21.3.1 立体渲染

想要渲染两个独立的视图，这个过程的工作量似乎是渲染一个视图的两倍。然而，正如 Wilson [1891]所指出的，即使对于一个十分简单的实现，这也是不正确的。阴影贴图的生成、模拟、动画以及其他元素都是与视图无关的。调用的像素着色器数量也并不会翻倍，因为显示器示本身会在两个视图之间被分为两半。类似地，后处理效果的实现也依赖于分辨率，因此这些成本也并不会改变。然而，依赖于视图的顶点处理会增加一倍，因此已经有许多人对减少这种开销的方法进行了探索。

视锥体剔除通常会在任何网格被发送到 GPU 管线之前进行执行。可以使用一个较大的视锥体，来包含两个较小的眼睛视锥体[453, 684, 1453]。由于这个剔除过程发生在渲染之前，因此在剔除之后，可以再次检索要进行使用的精确渲染视图。然而，这意味着在剔除过程中需要有一个安全余量，因为这两个检索视图可能会查看一些已经被视锥体所剔除的模型。Vlachos [1823]建议将 FOV 增加 5 度来进行预测性剔除。Johansson [838]讨论了如何将视锥体剔除和其他策略（例如实例化和遮挡剔除查询）结合起来，用于大型建筑模型的 VR 显示。

渲染两个立体视图的一种方法是连续进行渲染，先完整渲染一个视图，然后再去渲染另一个视图。虽然这个实现看起来很简单，但是它有一个明显的缺点，即状态改变的数量也会加倍，这是需要避免的（章节 18.4.2）。对于基于 tile 的渲染器而言，频繁更改视图和渲染目标（或者裁剪矩形）将会导致很差的性能表现。

一个更好的选择是将每个物体渲染两次，并在相机变换之间进行切换使用。然而，这种方法的 draw call 数量是原来的两倍，这会导致额外的开销。一种方法是使用几何着色器来复制几何图形，为每个视图都创建对应的三角形，例如：DirectX 11 支持几何着色器将其生成的三角形发送到单独的目标中。不幸的是，人们发现这种技术会将几何吞吐量降低三倍或者更多，因此在实践中没有进行使用。一个更好的解决方案是

使用实例化，其中每个物体的几何图形，会通过单个 draw call 来绘制两次[838, 1453]。使用由用户进行定义的裁剪平面设置，来保持每个眼睛的视图分离开来。使用实例化要比使用几何着色器快得多，并且它是一个很好的解决方案，因为它不需要任何额外的 GPU 特性支持[1823, 1891]。另一种方法是在渲染一个眼睛的图像时，生成一个命令列表（章节 18.5.4），并将引用的常量缓冲区转移到另一个眼睛的变换中，然后重播（replay）这个命令列表，来渲染第二个眼睛的图像[453, 1473]。

有几个扩展可以避免向管线发送多次几何图形。在一些手机上，一个叫做 multi-view 的 OpenGL ES 3.0 扩展，增加了对只发送一次几何图形并将其渲染到多个视图中的支持，还可以对屏幕顶点的位置以及任何与视图相关的变量进行调整[453, 1311]。这个扩展在实现立体渲染器的时候提供了更多的自由度。例如：最简单的扩展可能是在驱动程序中使用实例化，向管线发送两次几何图形，而一些需要 GPU 支持的实现，可能会将每个三角形都发送到各个视图中。不同的实现有着不同的优点，但由于 API 的成本总是在降低的，因此这些方法中的任何一个都可以帮助到那些 CPU 瓶颈的应用程序。例如：一些更加复杂的实现可以提高纹理缓存的效率[678]，并且只需要执行一次对视图无关属性的顶点着色。在理想情况下，可以为每个视图都设置完整的矩阵，并且任何逐顶点的属性都可以针对每个视图来进行着色。为了使得硬件实现使用更少数量的晶体管，GPU 可以选择实现这些特性的一个子集。

AMD 和 NVIDIA 提供了针对 VR 立体渲染的多 GPU 解决方案。对于两个 GPU 而言，每个都会负责渲染一个单独的眼睛视图。通过使用一个关联掩码（affinity mask），CPU 需要为所有接收特定 API 调用的 GPU 设置一个 bit。通过这种方式，可以将调用发送到一个或者多个 GPU 中[1104, 1453, 1473, 1495]。如果左右两个眼睛视图之间的调用不同，那么即便使用关联掩码，也仍然需要调用两次 API。

厂商所提供的另一种渲染方式是被 NVIDIA 称作的广播（broadcasting），其中两个眼睛的渲染也仅仅使用单个 draw call，也就是说，它会被广播到所有 GPU 上。常量缓冲区会用于发送不同的数据（例如：眼睛位置）到不同的 GPU 中。广播可以创建两个眼睛的图像，它比单一视图方法不会有更多的 CPU 开销，其唯一的成本是需要设置第二个常量缓冲区。

多个 GPU 意味着多个渲染目标，但是合成器通常只需要单个渲染图像。有一个特殊的子矩形传输命令，可以在一毫秒或者更短的时间内[1471]，将渲染目标的数据从一个 GPU 转移到另一个 GPU 中。这个过程是异步的，这意味着可以在 GPU 执行其他工作的时候进行传输。在两个 GPU 并行运行的情况下，它们也可以分别创建渲染所需的阴影缓冲区。这虽然是重复的工作，但是要比对并行化进程进行优化、以及在 GPU 之间进行传输数据更加简单，通常也会更快。以这种方式来设置两个 GPU，可

以使渲染速度提高 30% 到 35% [1824]。对于已经针对单个 GPU 进行了优化的应用程序，多个 GPU 可以将额外的计算能力应用于额外的样本，从而获得更好的抗锯齿效果。

立体视差（parallax from stereo viewing）对于近处的模型而言十分重要，但是对于远处的物体则可以忽略不计。Palandri 和 Green [1346]利用了这样的一个事实，他们在移动 GearVR 平台上，使用了一个垂直于观察方向的分离平面，他们发现 10 米左右的距离是一个不错的默认值。比这个距离更近的不透明物体会被渲染为立体的，而那些比这个距离更远的物体，则使用放置在两个立体相机之间的单目相机进行渲染。为了最小化过度绘制，首先会绘制立体视图，然后使用它们的深度缓冲区交集，来对这个单目渲染的 z-buffer 进行初始化。然后再将这些远处物体的图像与每个立体视图合成在一起。透明物体会在每个视图的最后进行渲染。虽然这种方法更加复杂，并且那些跨越分离平面的物体还需要额外的 pass 进行处理，但是这种方法大约可以节省约 25% 的总体开销，并且并不会损失质量或者深度感知。

在图 21.7 中可以看到，由于光学元件会发生畸变，因此会在每个眼睛图像的外围产生更高密度的像素。并且，图像四周的内容通常没有那么重要，因为用户在相当多的时间里都会看着屏幕的中央。出于这些原因，人们开发了各种技术，从而减少对每个眼睛视野外围像素的工作量。

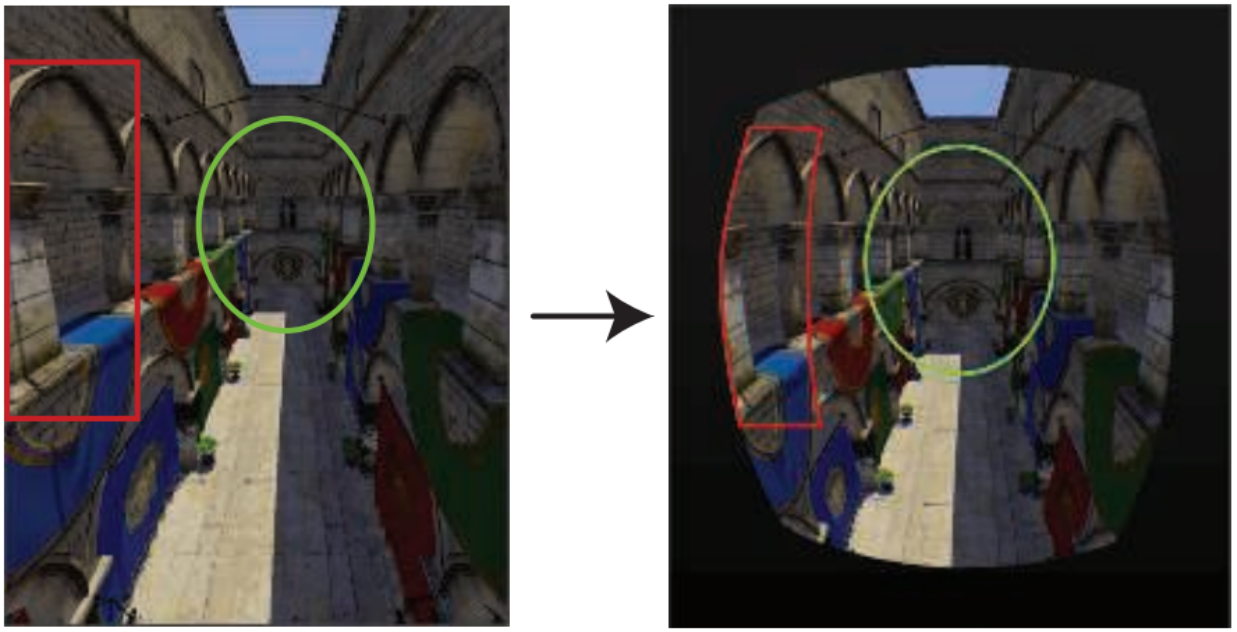


图 21.7：左侧是一个眼睛的渲染图像，右边是用于显示的扭曲图像。我们可以注意到，中间的绿色椭圆保持了大致相同的面积。而在图像外围，渲染图像中的较大区域（红色轮廓）被扭曲称了显示图像中的较小区域[1473]。

降低外围分辨率的一种方法是使用 NVIDIA 的多分辨率着色（multi-resolution shading）和 AMD 的可变速率着色（variable rate shading）。其核心思想是将屏幕划分为 3×3 的部分，并以较低的分辨率来渲染图像四周的区域[1473]，如图 21.8 所示。NVIDIA 从其 Maxwell 架构开始就支持这种分区渲染方案，但是 Pascal 及其之后的架构支持更加通用的投影类型，这被称为同步多重投影（simultaneous multi-projection，SMP）技术。可以通过最多 16 个单独的投影乘以 2 个单独的眼睛位置来处理几何形状，这样允许一个网格被复制多达 32 次，而在应用程序方面却没有带来额外的成本。其中第二个眼睛的位置，其沿 x 轴的偏移量必须与第一个眼睛的位置相等。每个投影都可以独立地围绕一个轴进行倾斜或者旋转[1297]。

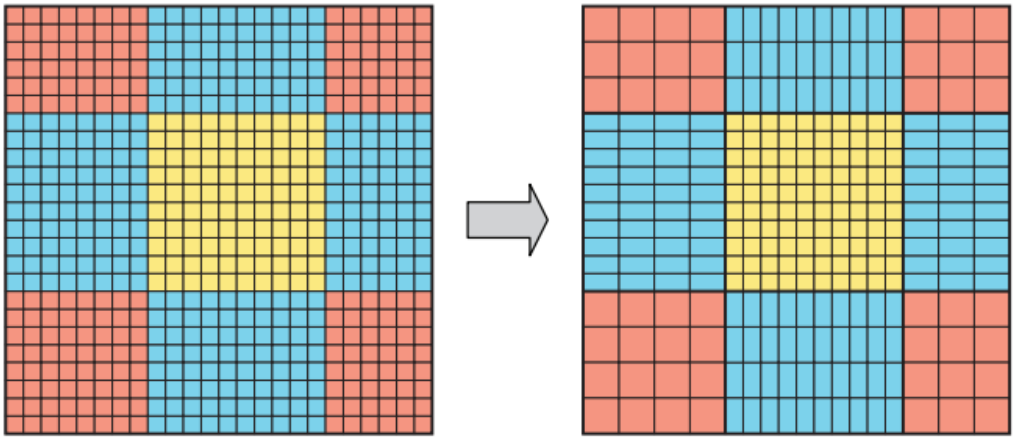


图 21.8：假设我们想要渲染左边的视图，同时在外围使用较低的分辨率。我们可以根据需要降低任何区域中的分辨率，但是通常最好在共享的边缘处保持相同的分辨率。在右侧，我们展示了蓝色区域的像素数量是如何减少 50% 的，同时红色区域的像素数量是如何减少 75% 的。画面的 FOV 保持不变，但是用于周边区域的分辨率却降低了。

通过使用 SMP，可以实现镜头匹配的着色（lens matched shading），其目标是让渲染分辨率与显示内容能够更好地匹配，如图 21.7 所示。会渲染出四个带有倾斜平面的视锥体，如图 21.9 左侧所示。这些修改之后的投影可以在图像中心的提供更大的像素密度，而在外围提供更少的像素密度。这使得截面之间的过渡要比多分辨率着色更加平滑。这种方法存在一些缺点，例如：泛光（bloom）等效果需要重新进行设计才能够正确显示。Unity 和 UE 4 已经将这个技术集成到了他们的系统中[1055]。Toth 等人[1782]对这些算法以及其他的多视图投影算法进行了形式上的对比和比较，并为每个眼睛使用最多 3×3 个视图来进一步减少像素着色。请注意，SMP 可以同时应用于双眼，如图 21.9 右侧所示。

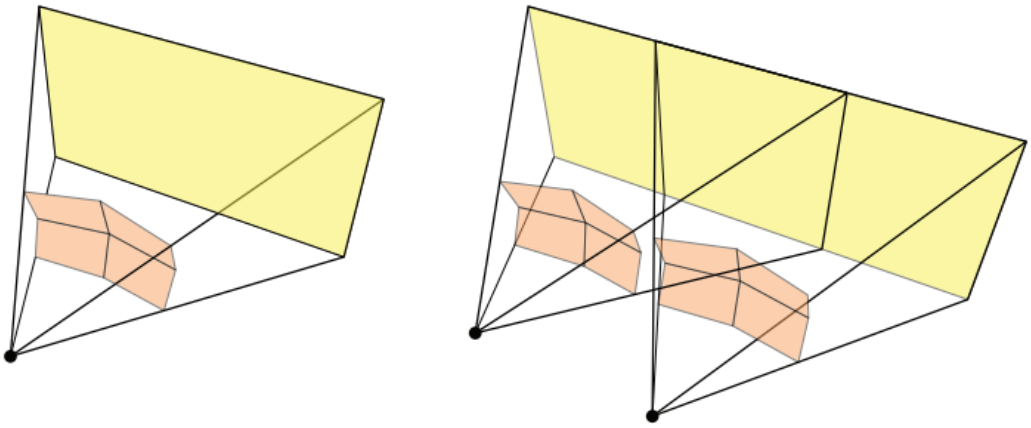


图 21.9：左图：同步多重投影（SMP），一只眼睛使用四个投影平面。右图：SMP 为每个眼睛都分别使用四个投影平面。

为了节省片元处理的时间，有一种应用程序级别的方法，该方法被称为径向密度遮罩（radial density masking），它会以一个四边形的棋盘格图案来对边缘像素进行渲染。换句话说，每个另外的 2×2 四边形片元都不会被渲染。然后使用一个后处理 pass，来从它们的邻居重建出缺失的像素[1824]。这种技术对于只有单个低端 GPU 的系统而言尤其有价值。使用这种方法进行渲染可以减少像素着色器的调用，但是如果后续执行重建过滤器的成本太高，则可能会起到反效果。索尼的伦敦工作室在这一过程中更进一步，直接从 2×2 的集合中去掉 1 个、2 个或者 3 个四边形，越靠近图像边缘，丢弃的四边形数量就越多。缺失的四边形会以类似的方式进行填充，并且每帧会改变丢弃的方式，即使用抖动模式来丢弃四边形。使用一个时域抗锯齿也有助于隐藏阶梯状的瑕疵。索尼的系统节省了大约 25% 的 GPU 时间[59]。

另一种方法是每个眼睛都渲染两个独立的图像，其中一个图像位于中心的圆形区域，另一个图像是构成外围的环形区域。然后，对这两张图像进行合成和扭曲，从而构成该眼睛的显示图像。外围图像可以以一个较低的分辨率进行生成，从而节省对于像素着色器的调用，其代价是发送的几何图形会形成四个不同的图像。这种技术刚好与 GPU 支持将几何图形发送到多个视图中相吻合，并且为具有两个或者四个 GPU 的系统提供了一种十分自然的工作划分。虽然是为了减少由于 HMD 所涉及的光学现象而导致外围像素的过多着色，但是 Vlachos 则将这种技术称为固定注视点渲染（fixed foveated rendering）[1824]。这个术语来自一个更加高级的概念，即注视点渲染（foveated rendering）。

21.3.2 注视点渲染

为了理解这种渲染技术，我们必须对我们的眼睛有更多的了解。中央凹（fovea）是我们眼睛视网膜上的一个小凹陷，里面充满了高密度的视锥细胞，而视锥细胞则是与颜色视觉相关的感光细胞。人眼在这个区域内的视觉灵敏度最高，我们会通过旋转眼睛来利用这种能力，例如：追踪飞行中的鸟，或者阅读书本上的文本。在离开中央凹的地方，人眼的视力会快速下降，在开始的 30 度角内，每距离中央凹的中心 2.5 度，视力会下降约 50%，在更远的地方下降的幅度会更大。我们的双眼 FOV（两个眼睛都能看到同一个物体）约为 114 水平角度。第一代消费级头戴式设备的 FOV 较小，双眼的水平角度约为 80 度–100 度，这个角度之后可能会有所上升。从 2016 年起，中心 20 度视角的区域约占整个 HMD 显示屏的约 3.6%，预计到 2020 年左右将降至 2% [1357]。在此期间，HMD 显示器的分辨率可能会提高一个数量级[8]。

由于眼睛会在视觉灵敏度较低的区域看到显示器的绝大多数像素，这为使用注视点渲染来进行更少的着色工作提供了机会[619, 1358]。这个想法是以高分辨率和高质量来渲染眼睛所指向的区域，而在其他地方则花费的较少的精力。这个想法的问题在于，人的眼睛会可以旋转，因此注视的区域也会发生改变，例如：在研究一个物体的时候，眼睛会进行一系列被称为扫视（saccades）的快速移动，这个移动速度可以达到每秒 900 度，即在一个 90 FPS 系统中，可以达到每帧旋转 10 度。精确的眼动追踪硬件，可以通过在注视点区域以外执行更少的渲染工作，来大幅提高性能表现，但是这种类型传感器也是一个不小的技术挑战[8]。此外，在边缘处渲染面积“较大”的像素，往往会增加锯齿问题。低分辨率外围区域的渲染效果，可以通过尝试保持对比度，同时避免随着时间发生的较大变化来进行改善，使得这些区域在感知上更加容易被接受[1357]。Stengel 等人[1697]对之前减少着色器调用数量的注视点渲染方法进行了讨论，并提出了他们自己的方法。

21.4 渲染技术

适用于单视图渲染的技术，并不一定适用于两个视图。即使是在立体渲染中，在单个固定屏幕上所使用的技术，与在随着观察者移动的屏幕上所使用的技术之间也存在相当大的差异。在这里，我们将介绍一些特定的算法，这些算法可能会在单个屏幕上运行良好，但是在 VR 和 AR 系统中则会出现问题。我们借鉴了 Oculus、Valve、Epic Games、Microsoft 等公司的专业知识。这些公司的后续研究会被纳入用户手册，并在博客中进行讨论，因此我们建议访问他们的网站，从而获取当前的最佳实践[1207, 1311, 1802]。

正如上一节中所强调的，供应商希望我们能够理解他们的 SDK 和 API，并正确地使用它们。视图是至关重要的，因此我们需要遵循供应商提供的头部模型，并且需要完

全正确地获得相机的投影矩阵。同时应当避免使用闪光灯（strobe light）等效果，因为闪烁会导致头痛和眼睛疲劳；在视野边缘附近的闪烁也可能会导致模拟病；闪烁效果和高频纹理（例如细条纹），也会引发某些人的癫痫发作。

基于显示器的视频游戏通常会使用一个 HUD，其中包含了有关生命值、弹药或者剩余燃料的数据信息。然而，对于 VR 和 AR 来说，双目视觉意味着靠近观察者的物体会在两个眼睛之间具有更大的偏移，即会聚效应（章节 21.2.3）。如果这个 HUD 在双眼屏幕上处于相同的位置，那么在感知上，会暗示这个 HUD 一定离得很远，如图 21.4 所示。然而，HUD 被绘制在所有场景内容的前面，这种感知上的不匹配使得用户很难将两幅图像融合在一起，并理解他们所看到的内容，这可能会导致一些不适感[684, 1089, 1311]。将 HUD 内容以接近眼睛附近的深度来进行渲染，可以解决这个问题，但是仍然需要以牺牲屏幕空间为代价，如图 21.10。如果附近的墙壁比准星更近，那么还存在深度冲突的风险，因为这个准星图标仍然会在给定深度来进行渲染。发射一条光线，并在给定方向上找到最近的表面深度，从而可以使用各种方法来调整 HUD 的深度，可以直接使用射线检测到的深度，也可以在需要时将它平滑地移动到更靠近的位置[1089, 1679]。

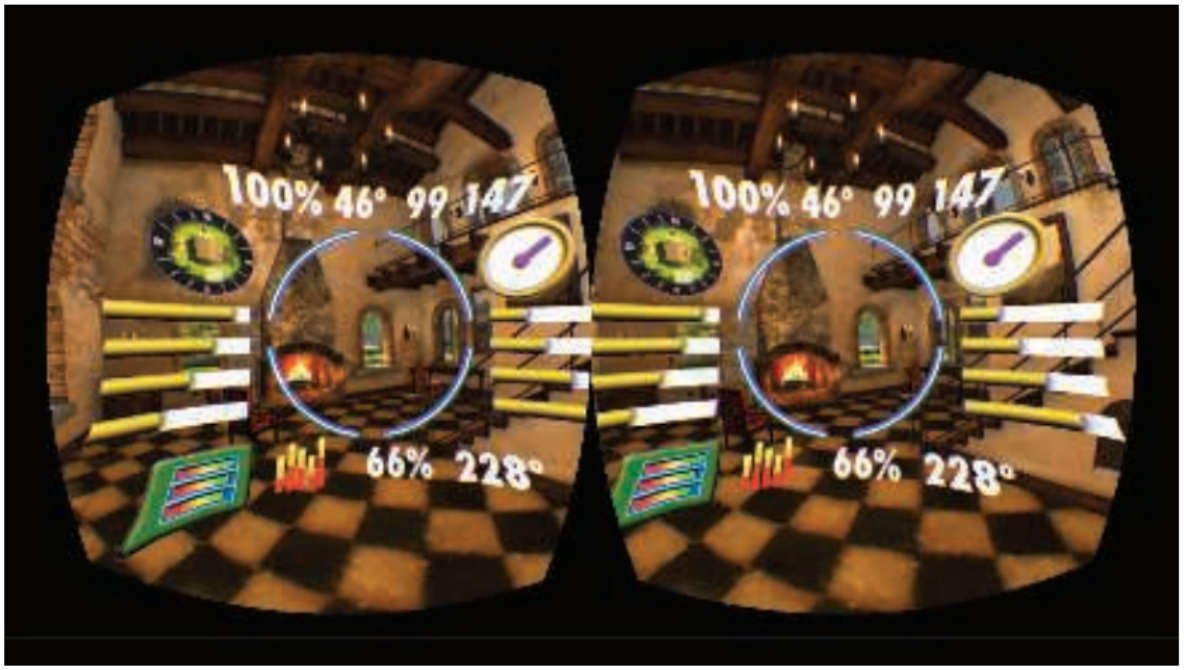


图 21.10：一个内容繁多的 HUD 占据了大部分视野。请注意，这些 HUD 元素必须针对每个眼睛进行移动，从而避免混乱的深度暗示。一个更好的解决方案是考虑将这些信息直接放入虚拟世界中（作为一个实体）；或者放入是玩家角色的设备和头显中，因为用户可以倾斜或者转动头部，游戏内的头显和设备可以跟着玩家一起运动[1311]。想要从这幅图中看到立体效果，你可以靠近图片，在图片之间垂直放置一张小而硬的纸，并让每只眼睛都看着各自的图片。

在某些情况下，凹凸贴图在任何立体视觉系统中表现得都很差，因为它是在一个平坦的表面上来进行着色的。它可以用于精细的表面细节或者远处的物体，但是对于表示较大几何形状的法线贴图，这种错觉会快速消失，并且用户可以感知到这种变化，如图 21.11 所示。基础视差映射存在的游动问题，会在立体视觉中变得更加明显，但是这个问题可以通过一个简单的校正因子来进行改善[1171]。在某些情况下，可能需要一些成本更高的技术，例如陡峭视差映射（steep parallax mapping）、视差遮挡映射（章节 6.8.1）或者位移映射[1731]等，从而产生令人信服的效果。

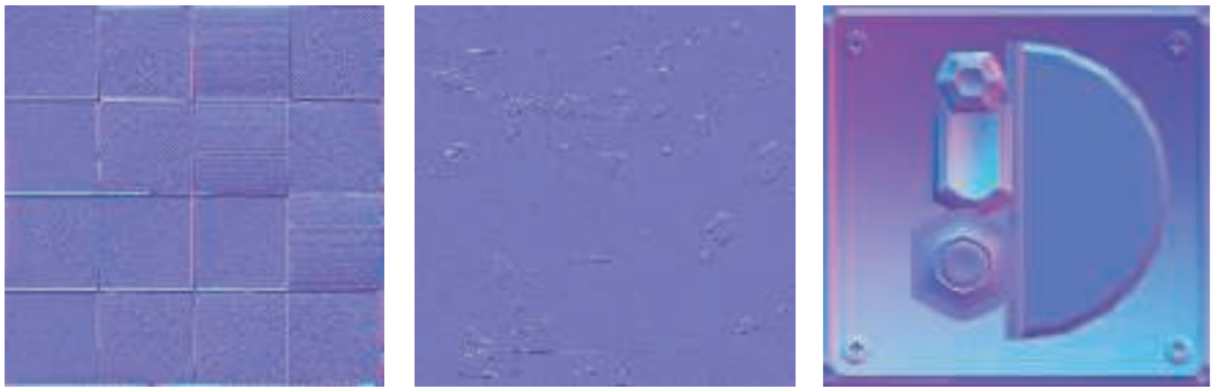


图 21.11：代表较小表面特征的法线贴图（例如左边和中间的两个纹理），可以在 VR 中很好地工作。当凹凸纹理代表了比较大的几何特征时（例如右侧的纹理），在立体渲染中近距离观察，将无法令人信服。[1823]

广告牌技术和 impostor 在立体视觉中有时无法令人信服，因为它们缺乏表面 z 深度。使用一些体积技术或者网格可能会更加合适[1191, 1802]。天空盒的大小也需要进行调整，使它们在“无限远”处或者附近处被渲染，即眼睛位置的差异不应该影响到天空盒的渲染。如果使用了色调映射，那么这两个渲染图像都需要使用相同的色调映射，从而避免眼睛疲劳[684]。屏幕空间环境光遮蔽和屏幕空间反射技术，也会产生不正确的立体差异[344]。类似地，一些后期处理效果（例如泛光或者镜头光晕），也需要根据每个眼睛视角的 z 深度来生成，以便保证图像能够正确融合。一些水下效果或者热浪扭曲效果也需要重新进行设计。屏幕空间反射技术所产生的反射可能会存在匹配问题，因此使用反射探针可能会更加有效[1802]。即使是高光效果也可能需要进行修改，因为立体视觉会影响人们对于光泽材质的感知，两幅眼睛图像之间的高光位置可能会存在很大的差异。研究人员发现，修正这种差异可以使得图像融合更加容易，同时更有说服力。换句话说，在计算光泽分量的时候，眼睛的位置可能会彼此靠近一些。相反，远处物体的高光差异在两幅图像之间可能难以察觉，因此可以共享着色结果[1781]。如果将计算结果存储在纹理空间中，那么可以在眼睛图像之间共享这些着色结果[1248]。

VR 对于显示器的技术要求非常高，例如：使用一个具有 50 度水平 FOV 的显示器，每度可能会跨越 50 个像素；而 VR 显示器通常具有更大的 FOV，例如 110 度，在 HTC Vive 中，每个眼睛的像素分辨率为 1080 × 1200 像素，其结果是每度只会跨越 15 个像素[1823]。从渲染图像到显示图像的转换，也会使得正确的重采样和滤波过程变得更加复杂。用户的头部会一直移动，即使只是移动一点点，也会导致时域锯齿的增加。出于这些原因，高质量的抗锯齿技术实际上是提高图像质量和融合效果的必要条件。由于潜在的模糊问题，因此通常不建议使用时域抗锯齿技术[344]，尽管至少有一个索尼团队成功地使用了时域抗锯齿技术[59]。他们发现这样做是有利有弊的，但是去除闪烁的像素要比提供清晰的图像更加重要。然而，对于大多数 VR 应用程序而言，MSAA 可以提供更加清晰的视觉效果，因此 MSAA 是首选的抗锯齿方案[344]。请注意，4× MSAA 就已经很好了，8× MSAA 效果会更好，使用抖动超采样也会更好，如果设备可以负担得起的话。这种对于 MSAA 的偏好与使用各种延迟渲染方法相抵触，因为在延迟渲染中想要对每个像素采集多个样本是成本很高的。

在着色表面上颜色缓慢变化的条带（章节 23.6），在 VR 显示器上会特别明显。这个瑕疵可以通过添加一点抖动噪声来进行掩盖[1823]。

在 VR 中不应当使用运动模糊效果，因为它们会使图像变得更加模糊，其影响超出了由于眼球运动所引起的任何瑕疵。由于 VR 显示器通常会以 90 FPS 的帧率运行，它具有低余晖（low-persistence）的特点，运动模糊的效果与这种特点是不一致的。由于我们的眼睛会进行快速移动（扫视）从而获得广阔的视野，所以应当避免使用景深技术。这种方法会让场景周边的内容毫无理由地变模糊，并可能会导致模拟病[1802, 1878]。

混合现实系统带来了额外的挑战，例如：在虚拟物体上应用与现实环境中相似的光照。在某些情况下，现实世界中的光照可以进行预先控制，并转换为虚拟照明。如果我们无法做到这一点，那么我们可以使用各种光线估计技术来对环境中的照明条件进行捕获和近似。Kronander 等人[942]对各种光照捕获和表示方法进行了深入调查。

21.4.1 抖动

即使虚拟世界和现实世界之间有着完美的追踪和适当的通信，延迟仍然会成为一个问题。生成图像需要一定的时间，大多数 VR 设备的更新率为 45 FPS 到 120 FPS[125]。

当图像没有及时生成并发送到合成器中进行处理，以及没有及时显示在屏幕上时，就会发生掉帧（dropped frame）。对早期发布的 Oculus Rift 游戏的调查显示，它们的帧数下降了约 5% [125]。掉帧会增加抖动（judder）的感觉，这是 VR 头显中的一

种模糊和频闪瑕疵，当眼睛相对于显示器发生移动的时候最为明显，如图 21.12 所示。如果屏幕上的一些像素在掉帧持续的时间内被照亮，那么眼睛的视网膜上就会接收到一些光影余晖（smears）。降低每一帧的持久性（即像素在一帧中被显示器照亮的时间）可以减轻这种余晖效应。然而，这样做可能会导致频闪现象，即如果两帧之间有很大的变化，则会使人感知到多个分离的图像。Abrash [7]对抖动以及抖动与显示技术之间的关系进行了深入讨论。

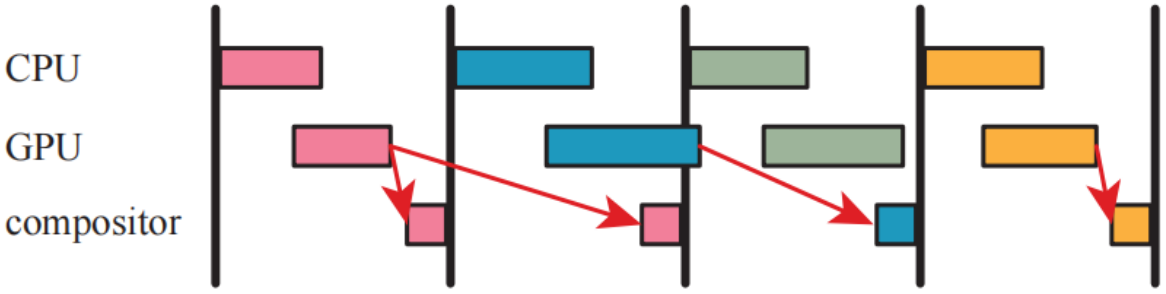


图 21.12：抖动。图中展示了四帧画面的生成过程，CPU 和 GPU 会试图为每一帧都计算一个图像。其中第一帧的图像（粉色）被及时计算出来，可以将其发送给这一帧的合成器。下一帧的蓝色图像没有及时渲染完成，因此需要再次显示第一帧中的图像。第三帧中的绿色图像也没有及时准备好，因此第二帧的图像（此时已经渲染完成）会被发送给第三帧的合成器。第四帧的橙色图像及时渲染完成，因此可以正常显示。请注意，这里第三帧的渲染结果永远都不会显示。[1311]

供应商提供了可以帮助最小化延迟和抖动效果的方法。Oculus 将这套技术称之为时间扭曲（timewarp）和空间扭曲（spacewarp），它们可以对生成的图像进行扭曲或者修改，从而更好地匹配用户的方向和位置。首先，假设我们没有发生掉帧，并且我们检测到用户正在匀速旋转头部。我们可以利用检测到的旋转信息，来预测每个眼睛的位置和视角方向。通过完美的预测，我们所生成的图像将会完全符合需求 。

假设用户正在旋转头部，并且旋转速度变慢了。对于这种情况，我们的预测将稍微超前一点，因此生成的图像要比显示时的位置稍早。除了速度之外，对旋转加速度也进行估计，可以帮助改善预测效果[994, 995]。

当某一帧被丢弃的时候，会发生更加严重的情况。此时，我们必须使用前一帧生成的图像，因为我们总是需要在屏幕上显示一些东西。如果我们有了对用户视图的最佳预测，我们可以对现有图像进行修改，从而对缺失帧的图像进行近似。这里我们可以执行的一个操作是二维的图像扭曲，Oculus 称之为时间扭曲，它只会对头部姿态的旋转进行补偿。这种扭曲操作是一种快速的修正措施，它总比什么都不做要好得多。Van Waveren [1857]讨论了各种时间扭曲实现中的权衡，包括那些在 CPU 和数字信

号处理器（DSP）上运行的实现，他得出的结论是：GPU 是迄今为止完成这个任务速度最快的。大多数 GPU 都可以在不到半毫秒的时间内，完成这个图像的扭曲过程[1471]。对之前显示的图像进行旋转操作，会使显示图像的黑色边框在用户的余光中变得可见。渲染一个比当前帧更大的图像，是一种避免这个问题的方法。然而，在实践中，这个边缘区域实际上并不会引起用户的注意[228, 1824, 1857]。

除了速度之外，这种纯旋转扭曲的一个优点在于，场景中的其他元素都是一致的。用户实际上处于环境天空盒的中心位置（章节 13.3），只有观察方向和朝向发生了改变。这个技术的执行速度很快，效果也很好。掉帧已经足够糟糕了，更严重的是，由于间歇性掉帧造成的可变且不可预测的延迟，可能会加速模拟病的发生[59, 1311]。为了能够提供更加平滑的帧率，Valve 在检测到帧率下降的时候，会启用交错重投影系统（interleaved reprojection），它会将渲染速率降至 45 FPS，并每隔一帧进行一次扭曲操作。类似地，PLAYSTATION 上的一个 VR 版本具有 120 Hz 的刷新率，其中渲染会以 60 Hz 的速率执行，并使用重投影来填充交替帧[59]。

仅仅是对旋转进行校正是不够的。即使用户没有移动或者改变他们的位置，当玩家头部旋转或者倾斜的时候，眼睛的位置也会发生改变。例如：在只使用图像扭曲的时候，双眼之间的距离将会变窄，因为这个新图像是使用分离眼睛生成的，每个眼睛都指向看不同的方向[1824]。这是一个很小的影响，但是如果观察者附近存在物体、或者观察者正在观察一个具有纹理的地面时，如果不进行适当的位置变化补偿，可能会导致用户迷失方向并产生眩晕感。为了对位置的变化进行修正，可以执行一个完整的三维重投影（章节 12.2）。图像中的所有像素都有一个与之相关的深度，所以这个过程可以被认为是将这些像素投影回它们在世界中的位置，然后移动眼睛所在的位置，再将这些点重新投影回屏幕。Oculus 将这个过程称为位置时间扭曲（positional timewarp）[62]。除了成本开销之外，这种方法还有一些缺点，其中一个缺点是：当眼睛发生移动的时候，一些表面会进入视野或者离开视野。这种现象可以以很多种形式发生，例如：立方体的表面可以变得可见；或者由于视差效应，导致前景中的物体相对于背景发生移动，从而遮挡一些细节，露出另外一些细节。重投影算法试图识别不同深度的物体，并使用局部图像的扭曲来填充任何被发现的空白[1679]。这种技术可能会造成一些错位拖尾（disocclusion trail），当物体从远处经过时，这种扭曲会使得远处的细节看起来好像发生了移动和动画一样。由于我们只有一个表面深度是已知的，所以这种基础的重投影方法无法处理透明度效果，例如：这个限制可能会影响粒子系统的外观表现[652, 1824]。

图像扭曲和重投影技术的一个问题在于，新片元的颜色是根据旧位置进行计算的。虽然我们可以改变这些片元的位置和可见性，但是我们无法改变任何高光或者反射效

果。即使这个表面本身被完美地移动了，掉帧也可能会导致在这些表面高光处产生抖动现象。即使没有发生任何头部运动，这些方法的基础版本也无法对场景中的物体运动或者物体动画进行补偿[62]。由于我们只知道表面的位置信息，并不清楚它们的速度信息。因此，对于推测出的图像而言，场景中的物体并不会自行地在帧与帧之间进行移动。我们在章节 12.5 中提到，我们可以通过一个速度缓冲区（velocity buffer）来捕获物体的运动。有了这个速度缓冲区，使得重投影技术也可以根据这种变化来进行动态调整。

上述的旋转补偿和位置补偿技术，通常都是在独立的异步过程中运行的，从而作为防止掉帧的一种手段。Valve 将其称为异步重投影（asynchronous reprojection），而 Oculus 将其称为异步时间扭曲（asynchronous timewarp）和异步空间扭曲（asynchronous spacewarp）。Spacewarp 通过对之前的帧进行分析，从而推断出缺失的帧，并考虑到了相机和头部的平移，以及动画和控制器的运动所带来的影响。在执行空间扭曲的时候并不会使用深度缓冲区。在正常渲染的同时，推测图像的计算是完全独立的。由于这个推测过程是基于图像的，因此它具有一个可以精确预测的时间，这意味着如果正常的渲染不能及时完成，通常都可以使用这个重投影图像。所以，我们不用去决定是继续尝试完成这帧的渲染，还是使用时间扭曲或者空间扭曲重投影，这两个过程的独立的，会同时进行。如果某一帧没有及时渲染完成，那么就可以使用空间扭曲的结果。这些扭曲技术对硬件的要求不高，它们主要是为了帮助那些性能较差的系统。Reed 和 Beeler [1471]讨论了 GPU 共享可以实现的不同方式，以及如何高效地使用异步扭曲技术，Hughes 等人也进行了类似的工作[783]。

旋转技术和位置技术是互补的，每个技术都可以贡献自己的改进。其中旋转扭曲可以完美地适应头部在旋转时，观察远处的静态场景或者图像。而位置重投影则适用于附近的动画物体[126]。相比于位置的改变，方向的改变通常会引起更加严重的匹配问题，所以即便是仅仅使用旋转校正，也能提供相当大的改进[1857]。

我们在这里的讨论，涉及到了这些补偿过程背后的基本思想。当然，有很多文章都探讨了这些方法的技术挑战和局限性，有兴趣的读者可以阅读相关的参考资料[62, 125, 126, 228, 1311, 1824]。

21.4.2 计时

虽然异步时间扭曲和异步空间扭曲技术可以帮助避免抖动，但是保持画面质量的最佳建议，还是让应用程序本身尽可能避免掉帧[59, 1824]。即使没有抖动现象，我们也可以注意到，用户在显示时的实际姿态可能会与预测姿态不同。因此，一种被称为后期定向扭曲（late orientation warping）的技术可能有助于更好地匹配用户应当看到

的内容。其核心想法是像往常一样获取姿态并生成帧，然后在帧中检索更新后的姿态预测。如果这个新姿态与用于渲染场景的原始姿态不同，则在此帧上执行旋转扭曲（即时间扭曲）。由于这个扭曲操作通常只需要不到 0.5 毫秒就可以完成，因此这种投入通常是值得的。在实践中，这种技术通常是合成器本身负责的。

通过使用一种被称为延迟锁存（late latching）的技术，可以使得该进程运行在一个单独的 CPU 线程上，能够最大限度地减少获取这些后期方向数据所花费的时间[147, 1471]。这个 CPU 线程会定期地将预测姿态发送到 GPU 的一个私有缓冲区中，GPU 可能会在扭曲图像之前时刻获取到这个最新的设置。延迟锁存可以用于直接向 GPU 提供所有头部姿态数据，这样做的限制在于，此时应用程序无法获得每个眼睛的观察矩阵，因为现在只有 GPU 才能提供这个信息。AMD 有一个改进版本的技术，它被称为最新数据锁存器（latest data latch），它允许 GPU 在需要这些数据的时刻，来获取最新姿态信息[1104]。

在图 21.12 中，你可能已经注意到了，CPU 和 GPU 有着相当长的停滞时间，因为 CPU 直到合成器完成工作之后才会开始处理下一帧的内容。这是针对单个 CPU 系统的简化图示，其中所有的工作都发生在一帧中。正如章节 18.5 中所讨论的，大多数系统都有多个 CPU，它们可以以各种方式来保持工作。在实践中，CPU 经常还会处理碰撞检测、路径规划以及其他的一些任务，并且还要为 GPU 在下一帧中的渲染准备相关数据。当构建好一个流水线时，CPU 在前一帧中设置的任何内容上，GPU 都可以进行工作[783]。为了提高效率，每帧中 CPU 和 GPU 的工作，都应当少于一帧时间，如图 21.13 所示。合成器通常会使用一种方法来知道 GPU 何时会完成工作。这个方法被称为 fence，它由应用程序作为命令发出，并在所有 GPU 调用完全执行之前发出信号。fence 对于知道 GPU 何时使用完各种资源十分有用（译者注：类似于 DX12 和 Vulkan 中的 Fence，一种信号同步机制）。

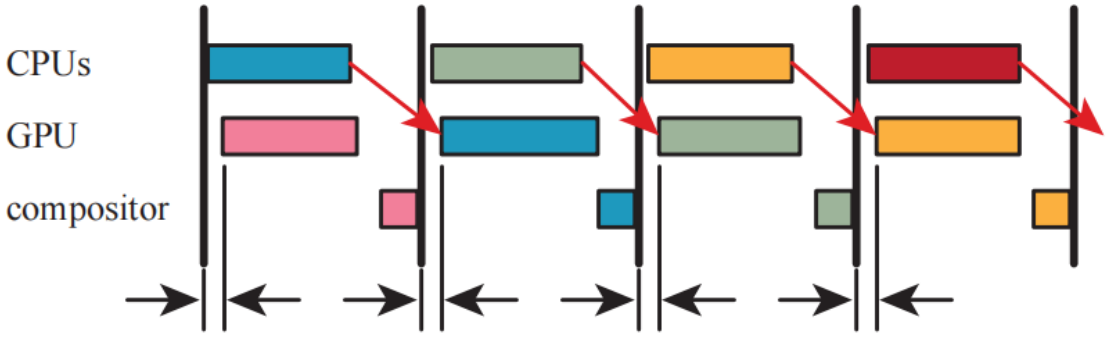


图 21.13：流水线。为了最大限度地利用资源，CPU 会在一帧内执行相关的设置任务，而 GPU 则会在下一帧中执行这些渲染任务。通过运行启动和自适应提前队列，底部显示的时间间隔可以添加到 GPU 的每帧执行时间中。

图中显示的 GPU 持续时间代表了渲染图像所要花费的时间。一旦合成器完成了最终图像的创建和显示，GPU 就可以开始渲染下一帧画面了。而 CPU 则需要等到合成完成之后，才能向 GPU 发出下一帧的渲染命令。但是，如果我们等到图像显示出来，那么应用程序在 CPU 上生成新命令时就会额外花费一些时间，这些命令由驱动程序进行解释，并最终向 GPU 发出渲染命令。在这段时间内（可能高达 2 ms），GPU 会处于空闲状态。Valve 和 Oculus 通过分别提供了名为运行启动（running start）和自适应提前队列（adaptive queue ahead）的支持，来避免这种空闲状态，这种类型的技术可以在任何系统上进行实现。通过计时前一帧的预计完成时间，并在此之前向 GPU 发出命令，其目的是让 GPU 在完成前一帧的任务之后，立即开始下一帧的工作。大多数 VR API 都提供了一些隐式或者显式的机制，以便能够以固定的节奏（regular cadence）来以处理下一帧，并拥有足够的时间来最大化吞吐量。在这一小节中，我们提供了一个简化的管线例子，以便让读者了解这种优化所带来的好处。详见 Vlachos [1823]和 Mah 的[1104]演讲，他们深入讨论了管线和计时策略。

有关虚拟现实系统和增强现实系统的讨论到此结束。鉴于写作和出版之间的时间差，我们预计会有许多新的技术出现，并取代这里所介绍的技术。我们的主要目标是为这个快速发展的领域展现其中的渲染问题，并提供一种解决方案的大致思路。最近的研究探索了一个有趣的方向，即使用光线投射来进行渲染。例如：Hunt [790]讨论了这个方案的可能性，并提供了一个开源的 CPU/GPU 混合光线投射渲染器，每秒计算超过 100 亿条光线。光线投射直接解决了光栅化系统所面临的许多问题，例如宽 FOV 和镜头畸变，同时也能够很好地处理注视点渲染。McGuire [1186]注意到，可以在一个滚动显示方案显示画面之前，将这些光线投射到像素上，从而将该系统的延迟降低到几乎为零。这与其他许多研究一起，使他得出结论：我们将在未来广泛使用虚拟现实，但是并不会称之为虚拟现实，因为它只是每个人的计算界面。

补充阅读和资源

Abrash 的博客[5]中包含了有关虚拟现实显示器、延迟、抖动和其他相关主题的基础性文章。对于高效的应用程序设计和渲染技术，Oculus 最佳实践网站[1311]和博客[994]中有很多有用的信息，Epic Games 的虚幻 VR 页面[1802]也包含了很多有用的信息。你可以研究一下 OpenXR，它是一个跨平台虚拟现实开发的代表性 API 和架构。Ludwig 将《军团要塞 2》转换为了 VR 版本[1089]，这个案例涵盖了一系列有关用户体验的问题，以及相关的解决方案。

McGuire [1186, 1187]概述了 NVIDIA 在 VR 和 AR 等多个领域中的研究工作。Weier 等人[1864]提供了一份全面的最新报告，这个报告对人类的视觉感知进行了讨论，并

讨论了如何在计算机图形学中利用人眼视觉的局限性。在 Patney [1358]组织的 SIGGRAPH 2017 课程中，包含了与视觉感知相关的虚拟现实和增强现实研究。Vlachos 的 GDC 演讲[1823, 1824]讨论了高效渲染的具体策略，并且给出了一些技术的更多细节，这些技术我们只进行了简要介绍。在 NVIDIA 的 GameWorks 博客[1055]中，包含了有关 VR GPU 的改进，以及如何最好地利用它们的文章，这些文章很有价值。Hughes 等人[783]提供了一个深入教程，其中包含了使用 XPerf、ETW 和 GPUView 工具来调整你的 VR 渲染系统，从而使其表现良好。Schmalstieg 和 Hollerer 的新书《Augmented Reality》[1570]，其中涵盖了与该领域相关的大量概念、方法和技术。

Chapter 22 Intersection Test Methods 相交测试方法

> Robert Frost——“I’ll sit and see if that small sailing cloud Will hit or miss the moon.”

> 罗伯特·弗罗斯特——“我要坐下来看看那朵小小的帆船云，会不会撞到月亮。”（20 世纪最受欢迎的美国诗人之一；1874—1963）

计算机图形学中经常会使用到相交测试（intersection test）。我们可能希望确定两个物体是否发生了碰撞；或者找到某个点到地面的距离，从而使得相机保持在恒定的高度。相交测试的另一个重要的用途是，判断某个物体是否应该被发送到管线中。所有这些操作都可以通过相交测试来执行。在本章节中，我们将介绍最常见的射线与物体的相交测试，以及物体与物体之间的相交测试。

在同样基于层次结构的碰撞检测算法中，该系统必须能够判断两个基本物体是否发生了碰撞。这些物体包括三角形、球体、轴对齐包围盒（axis-aligned bounding box，AABB）、定向包围盒（oriented bounding box，OBB）和离散定向多面体（discrete oriented polytopes，k-DOP）。

正如我们在章节 19.4 中所看到的，视锥体剔除是一种能够有效地丢弃视锥体外几何物体的方法。在使用这个方法的时候也需要进行测试，从而判断包围体（BV）是完全位于视锥体外部、完全位于视锥体内部还是部分在内部。

在所有的这些情况下，我们都遇到了一类需要进行相交测试的问题。相交测试会确定两个物体 A 和 B 是否相交，这可能意味着 A 可能会完全位于 B 的内部（反之亦然），也可能意味着 A 和 B 的包围盒相交，或者它们完全不相交。然而，有时候我们可能会需要更多的信息，例如距离某个位置最近的相交点，或者穿透的距离和方向等。

在本章节中，我们主要讨论快速的相交测试方法。我们不仅会给出基本的算法原理，还会对如何构建新的、高效的相交测试方法提出了建议。当然，本章节中所介绍的方法也可以用于离线的计算机图形应用程序，例如：章节 22.6 到章节 22.9 中所介绍的光线相交算法，就可以用于光线追踪程序。

我们首先会简要介绍基于硬件加速的拾取方法，之后本章节将继续介绍一些有用的定义，然后介绍围绕图元来构建包围体的算法。接下来我们会介绍构建高效相交测试方法的经验法则。最后，本章中的大部分内容都是关于相交测试方法的指南。

22.1 GPU 加速的拾取

通常我们需要让用户通过用鼠标或者其他输入设备，来选择（点击）场景中的某个物体。当然，这种操作对于性能的需求很高。

如果我们需要将所有物体都放在屏幕上的一个点上或者更大的区域内，而不需要考虑可见性，那么我们可以考虑使用 CPU 端的拾取解决方案。这种类型的拾取有时可以在建模软件或者 CAD 软件中看到，它可以通过使用层次包围体（章节 19.1.1），来在 CPU 上高效地解决。在像素的位置处生成一条射线，并从视锥体的近裁剪平面发射到远裁剪平面。然后根据需要，来测试该射线与层次包围体的相交情况，这个过程类似于在全局光照算法中对追踪光线进行加速。对于由用户在屏幕上框选矩形而形成的一个矩形区域，我们将会创建一个视锥体而不是射线，并根据场景的层次结构来对其进行相交测试。

在 CPU 上进行相交测试有几个缺点，这取决于具体的需求。对于一个包含数千个三角形的网格而言，逐三角形的相交测试是非常昂贵的，除非在网格本身上再施加一些加速结构，例如层次结构或者均匀单元格。如果对精度的要求很高，那么由位移映射或者 GPU 曲面细分所生成的几何图形，还需要与 CPU 进行匹配。对于 alpha 映射的物体（例如树叶），用户不应当能够选择到那些完全透明的纹素。因此在 CPU 上还需要大量的工作来模拟纹理访问，以及任何其他出于任何原因被着色器丢弃的纹素。

通常，我们只需要在屏幕上某个像素或者某个区域内可见的内容即可。对于这种类型的选择，可以使用 GPU 管线本身来执行。Hanrahan 和 Haeberli [661]首先提出了一种方法。为了支持拾取功能，在渲染场景的时候，每个三角形、多边形或者网格物体都会有一个唯一的标识符，它可以被认为是一种颜色。这个想法在意图上与可见性缓冲相类似，可见性缓冲的效果如图 20.12 所示。形成的图像进行离屏存储，然后将其用于极其快速的拾取操作。当用户单击一个像素的时候，将在这个图像中查找颜色标识符，并能够立即识别该物体。当使用一些简单的着色器来执行标准渲染的时候，这些标识符可以渲染到单独的渲染目标中，因此这个操作的成本相对较低。其最主要的开销可能是将像素信息从 GPU 读取回到 CPU 中。

像素着色器接收到的、或者计算出的任何其他类型的信息，也可以存储在屏幕外的渲染目标中。例如：法线或者纹理坐标明显可以这样做。也可以利用插值，来找到三角

形内某一点的相对位置[971]。在单独的渲染目标中，每个三角形的顶点颜色分别为红色 $(255, 0, 0)$ 、绿色 $(0, 255, 0)$ 和蓝色 $(0, 0, 255)$ 。假设此时用户选定像素的插值颜色为 $(23, 192, 40)$ ，这意味着红色顶点的系数为 $23/255$ ，绿色顶点的系数为 $192/255$ ，蓝色顶点的系数为 $40/255$ 。这些值实际上就是重心坐标，我们将在章节 22.8.1 中进一步讨论。

使用 GPU 进行拾取操作，最初是作为三维绘画系统中的一部分。这种拾取操作特别适合于这样的系统，即场景中的相机和物体不发生移动，因此整个拾取缓冲区可以只生成一次，并进行重复使用。为了在相机移动的时候进行选择，另一种方法是将场景再次渲染到一个微小的渲染目标，例如 3×3 尺寸，并使用一个离轴相机（off-axis camera）聚焦到屏幕上的一小部分。CPU 端的视锥体剔除应当消除几乎所有的几何物体，只有少数像素会进行着色计算，从而使得这个 pass 计算起来相对较快。为了能够拾取所有物体（而不仅仅是可见物体），这个小窗口方法可以执行多次，并使用深度剥离或者直接不渲染之前选择过的物体[298]。

22.2 定义和工具

本小节将会介绍一些符号和定义，这些内容对本章节的后续内容十分有用。

一条射线 $\mathbf{r}(t)$ 由原点 $\mathbf{o}$ 和方向向量 $\mathbf{d}$ 进行定义，为了方便起见，通常会将这个方向向量进行归一化，即 $||\mathbf{d}|| = 1$ 。其数学表达如方程 22.1 所示，其示意图如图 22.1 所示：

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{22.1}$$

方程中的标量 t 是一个变量，它用于在射线上生成不同的点，其中 $t < 0$ 所对应的点位于射线原点的后面（因此这些点并不是射线上的一部分），而 $t > 0$ 所对应的点则位于射线原点的前面。此外，由于方程中的射线方向已经被归一化了，因此参数 t 在射线上生成一个点，该点会距离射线原点 t 个单位距离。

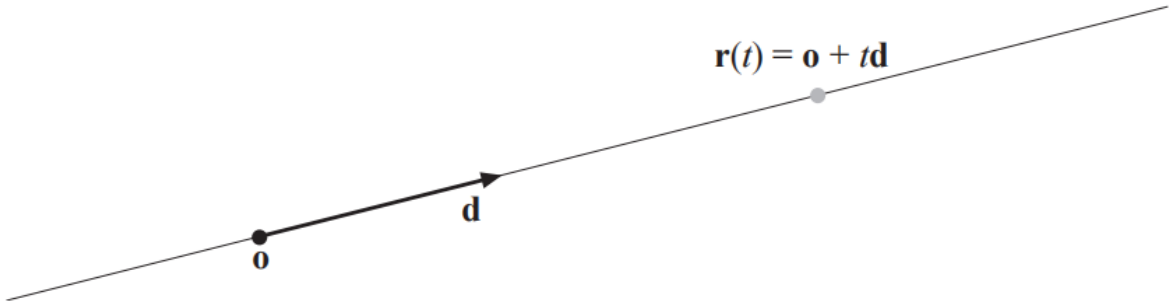


图 22.1：一条简单的射线及其参数： $\mathbf{o}$ （射线原点）、 $\mathbf{d}$ （射线方向）和 t ，其中参数 t 用于在射线上生成不同的点， $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ 。

在实践中，我们通常还会存储一个当前距离 l ，它是我们希望沿着射线进行搜索的最大距离。例如：在拾取的时候，我们通常想要获得射线方向上的最近交点，而超出这个交点的物体则可以直接忽略。这个距离 l 会被初始化为 ∞ ，当物体发生相交的时候， l 会被更新为此时的相交距离。一旦设置了 l ，射线就变成了一个线段来进行测试。在我们将要讨论的射线与物体的相交测试中，我们通常并不会将这个 l 包含在讨论中。如果我们希望使用这个距离 l ，我们所要做的就是执行普通射线–物体相交测试，然后将计算得到的相交距离与 l 进行比较，并采取适当的操作。

在讨论表面话题的时候，我们会将隐式表面与显式表面区分开来。隐式表面由方程 22.2 进行定义：

$$f(\mathbf{p}) = f(p_x, p_y, p_z) = 0 \tag{22.2}$$

这里的 $\mathbf{p}$ 是表面上的任意一点。这意味着如果我们将表面上的一点代入到函数 f 中，那么其结果将会为 0 。如果说这个点并不位于表面上，那么 f 的结果将是非零的。隐式表面的其中一个例子是 $p_x^2 + p_y^2 + p_z^2 = r^2$ ，它描述了一个半径为 r ，球心位于原点的球体。我们可以很容易看出，它可以被重写为 $f(\mathbf{p}) = p_x^2 + p_y^2 + p_z^2 - r^2 = 0$ ，这意味着它确实是隐式的。我们在章节 17.3 中简要介绍了有关隐式表面的话题，而 Gomes 等人[558]和 de Araujo 等人[67]，则很好地介绍了各种隐式表面类型的建模技术和渲染技术。

另一方面，显式表面是由向量函数 f 和一些参数 (ρ, ϕ) 所定义的，而不是表面上的一个点。这些参数将会在表面上产生点 $\mathbf{p}$ 。方程 22.3 给出了显式表面的思想：

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \mathbf{f}(\rho, \phi) = \begin{pmatrix} f_x(\rho, \phi) \\ f_y(\rho, \phi) \\ f_z(\rho, \phi) \end{pmatrix} \tag{22.3}$$

一个显式表面的例子还是球面，这次我们使用球坐标来进行表示，其中 ρ 是纬度和 ϕ 是经度，如方程 22.4 所示：

$$\mathbf{f}(\rho, \phi) = \begin{pmatrix} r \sin\rho \cos\phi \\ r \sin\rho \sin\phi \\ r \cos\rho \end{pmatrix} \tag{22.4}$$

另一个显式表面的例子是三角形，例如三角形 $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ 可以被显式描述为：

$$\begin{gathered}\mathbf{t}(u,v)=(1-u-v)\mathbf{v}_0+u\mathbf{v}_1+v\mathbf{v}_2\\ u\geq 0\\ v\geq 0\\ u+v\leq 1\end{gathered}$$

下面，我们将给出除了球面之外，一些常见包围体的定义。

定义：轴对齐包围盒（axis-aligned bounding box，也被称为矩形包围盒 rectangular box），简称 AABB，它是一个面法线与标准坐标轴重合的 box。例如：一个 AABB A，由一条对角线上两个相对的点 $\mathbf{a}^{\min}$ 和 $\mathbf{a}^{\max}$ 进行描述，其中 $\mathbf{a}_i^{\min}\leq\mathbf{a}_i^{\max},\forall i\in\{x,y,z\}$。图 22. 2 中包含了一个三维 AABB 的示意图和符号表示。

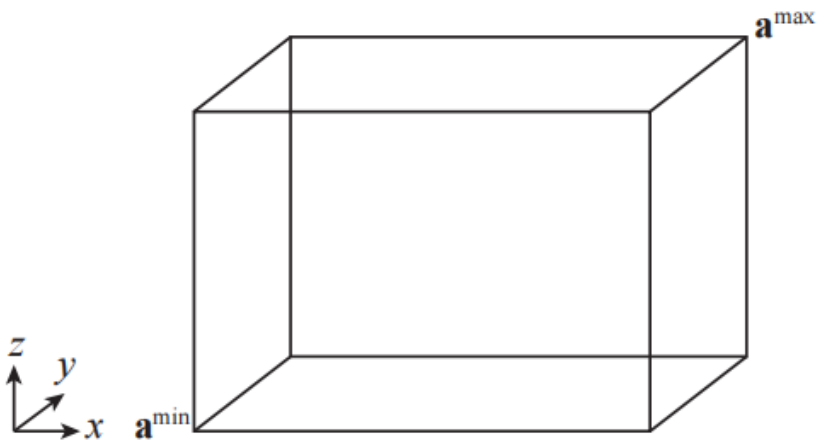


图 22.2：一个三维的 AABB，它由极值点 $\mathbf{a}^{\min}$ 和 $\mathbf{a}^{\max}$ 进行定义，图中还展示了标准基底的坐标轴。

定义：定向包围盒（oriented bounding box，简称 OBB）是一种面法线成对正交的 box，它是一个任意旋转的 AABB。对于一个 OBB B，可以使用 box 的中心点 $\mathbf{b}^c$ 和三个归一化向量 $\mathbf{b}^u$ 、$\mathbf{b}^v$ 、$\mathbf{b}^w$ 来进行描述，这三个向量分别描述了 box 的侧面方向。它们各自的正半长度分别表示为 h_u^B ， h_v^B ， h_w^B ，它们分别代表了中心点 $\mathbf{b}^c$ 到各自面中心的距离。图 22.3 展示了一个三维 OBB 及其符号表示。

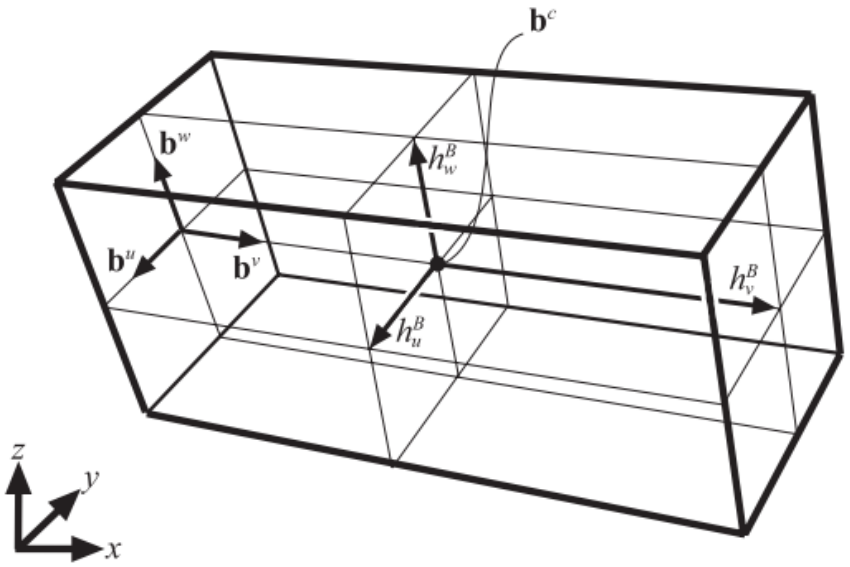


图 22.3：一个三维的 OBB B，其中心点位于点 $\mathbf{b}^c$，其归一化的面向量为 $\mathbf{b}^u$，$\mathbf{b}^v$，$\mathbf{b}^w$。边的半长 h_u^B，h_v^B，h_w^B，它们代表了从中心点 $\mathbf{b}^c$ 到各个面中心的距离。

定义：一个 k–DOP（discrete oriented polytope，DOP）由 $k/2$（其中 k 为偶数）个归一化法线（方向）$\mathbf{n}_i$ 进行定义，其中 $1 \leq i \leq k/2$。每个法线 $\mathbf{n}_i$ 都有两个相关的标量值 d_i^{min} 和 d_i^{max}，其中 $d_i^{min} < d_i^{max}$。每个三元组 $(\mathbf{n}_i, d_i^{min}, d_i^{max})$ 都描述了一个平板（slab）S_i，它是两个平面之间的空间，这个两个平面分别是 $\pi_i^{\min}: \mathbf{n}_i \cdot \mathbf{x} + d_i^{\min} = 0$ 和 $\pi_i^{\max}: \mathbf{n}_i \cdot \mathbf{x} + d_i^{\max} = 0$；各个 slab 的交集为 $\bigcap_{1 \leq l \leq k/2} S_l$，这是真正的 k–DOP 体积。k–DOP 被定义为贴合物体的最紧密的一组 slab [435]。AABB 和 OBB 可以表示为 6–DOP，因为它们都有 6 个平面，每个平面由 3 个平板定义。图 22.4 描述了一个二维的 8–DOP。

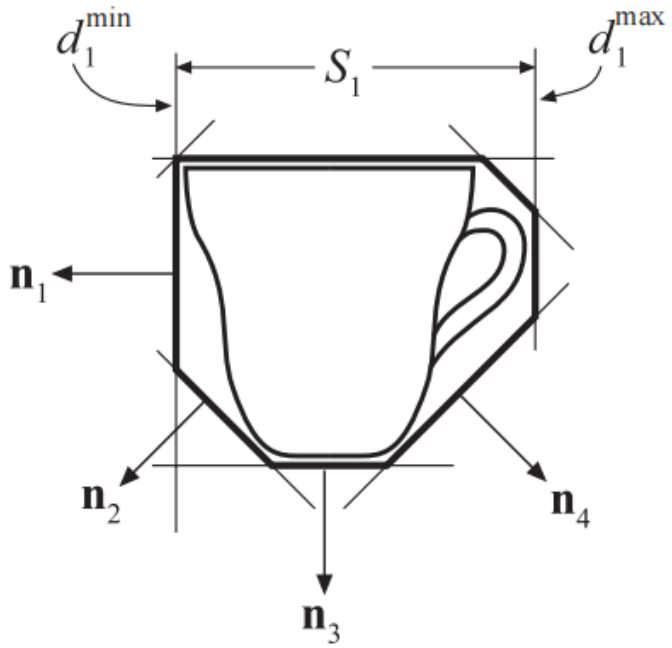


图 22.4：一个茶杯的二维 8-DOP，以及所有的法线 $\mathbf{n}_i$；还展示了第一个平板 S_1 和该平板的“大小”： d_1^{min} 和 d_2^{max} 。

对于凸多面体的定义，使用平面半空间（half-space）的概念是十分有用的。正半空间包括所有 $\mathbf{n} \cdot \mathbf{x} \mid +d \geq 0$ 的点 x，而负半空间包括 $\mathbf{n} \cdot \mathbf{x} \mid +d \leq 0$ 的点 x 。

定义：一个凸多面体（convex polyhedron），是由 p 个平面的负半空间交点所定义的有限体积，其中每个平面的法线都指向远离多面体的方向。

AABB、OBB、k-DOP、以及任何的视锥体，都是凸多面体的特殊形式。其中更加复杂的 k-DOP 和凸多面体，主要会用于碰撞检测算法，在这些表示方法中，计算底层网格的精确相交情况可能会十分昂贵。用于形成这些包围体的额外平面，可以从物体中裁剪掉额外的体积（不属于该物体），因此所带来额外成本是合理的。

另外两个有趣的包围体是线扫球体（swept sphere）和矩形扫球体（rectangle swept sphere）。它们也分别被称为胶囊体（capsule）和含片体（lozenge），如图 22.5 所示。

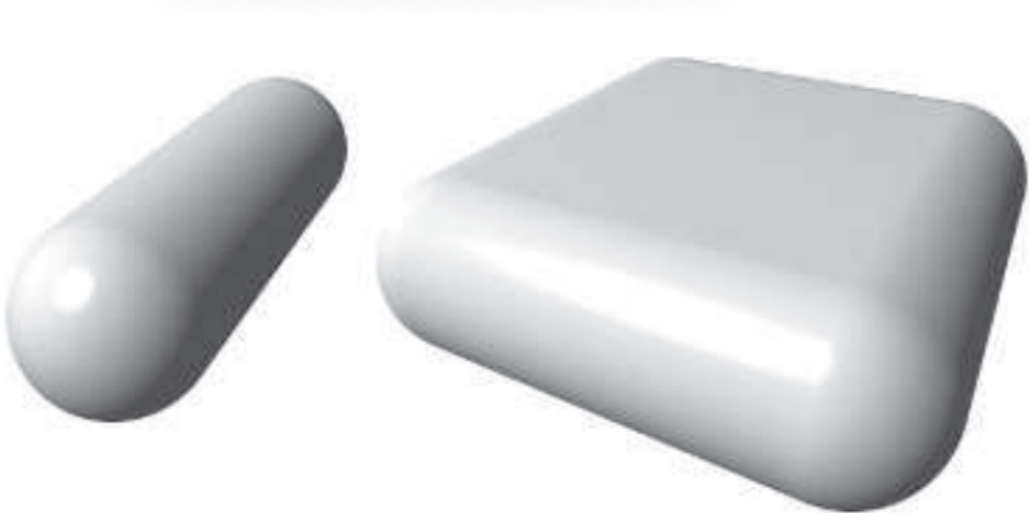


图 22.5：线扫球体和矩形扫球体，又被称胶囊体（capsule）和含片体（lozenge）。

分隔轴（separating axis）指定了一条直线，其中两个不重叠（不相交）的物体，在这条直接上的投影也不重叠。同样地，当一个平面可以插入到两个三维物体之间时，这个平面的法线便定义了一个分离轴。一个重要的相交测试工具也随之而来[576, 592]，它适用于凸多面体，例如 AABB、OBB、k-DOP。它是分离超平面定理（separating hyperplane theorem）的其中一个方面[189]。

> 这个测试在计算机图形学中有时也被称为“分离轴定理（separating axis theorem）”，它实际上是之前版本中的一个用词不当。它本身并不是一个定理，

> 而是分离超平面定理中的一个特例。

分离轴测试（Separating Axis Test，SAT）。对于任意两个不相交的凸多面体 A 和 B，至少存在一个分离轴，它们在这个分离轴上形成区间的投影也是不相交的。如果其中一个物体是凹的，那么这个结论就不成立了。例如：一口井的内壁和井里的水桶也可能并不发生接触，但是没有一个平面可以将它们分离开来。此外，如果 A 和 B 是不相交的，那么它们可以被这样的一个平面分开，这个平面会与下列轴（法线）中的一个正交（即平行于其中一个平面）[577]：

1. 物体 A 的一个面。
2. 物体 B 的一个面。
3. 每个多面体的一条边（即一个叉积）。

前两个测试表明，如果一个物体完全位于另一个物体中任何一个面的背面（far side），它们就不会发生重叠。前两个测试对针对面进行的，最后一个测试针对物体边缘进行的。为了能够在第三个测试中分离这两个物体，我们想要在一个尽可能靠近这两个物体的平面（该平面的法线是分离轴）上塞进一个平面，并且这个平面与物体之间的距离，不能比其任何一个边缘更近。因此，用于测试的分离轴都是由两个物体边缘的叉积形成的。图 22.6 展示了对两个 box 的测试。

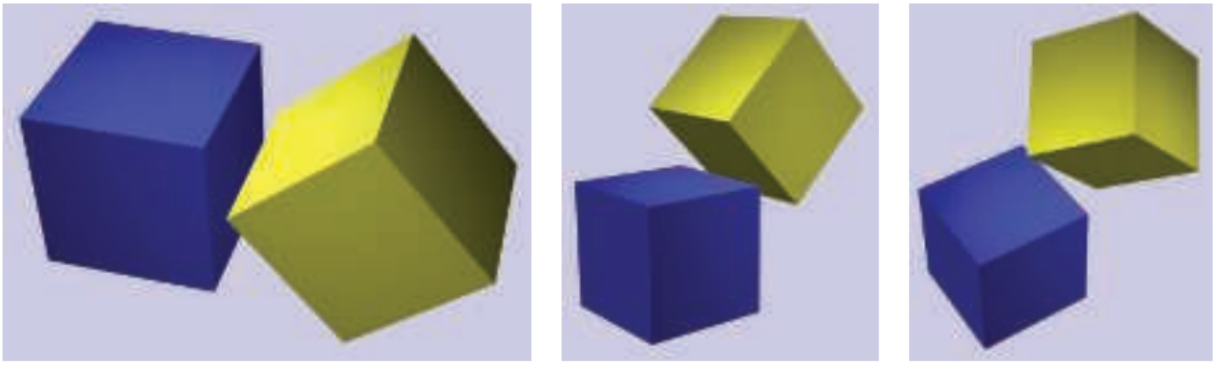


图 22.6：分离轴。这里将蓝色 box 记作 A，将黄色 box 记作 B。第一张图片显示出，B 完全位于 A 右侧面的右边；第二张图片显示出，A 完全在 B 左下角面的下方。在第三张图像中，没有任何一个面可以构成这样一个平面，能够将两个物体完全分隔在两侧；因此由 A 的右上边缘和 B 的左下边缘的叉乘，定义了分隔两个物体的平面法线。

注意，这里凸多面体的定义是自由的。线段和凸多边形（例如三角形）也是凸多面体，虽然它们都是退化的凸多面体，因为它们没有体积。线段没有面，因此不存在第一个测试。这个测试可以用于推导章节 22.12 中的三角形–box 相交测试，以及章节 22.13.5 中的 OBB–OBB 相交测试。Gregorius [597]注意到对任何使用分离轴的相交

测试，存在一个重要的优化：时间一致性。如果在某一帧中找到了这个分离轴，那么可以将这个轴存储为下一帧中第一组对测试物体的分离轴。

我们回到所使用方法的讨论中，对相交测试进行优化的一种常用技术是：在早期进行一些简单的计算，从而确定光线或者物体是否与另一个物体不相交。这样的测试被称为拒绝测试（rejection test），如果测试成功了，则我们称这次相交被拒绝了。

本章节中经常使用的另一种方法是，将三维物体投影到“最佳”的正交平面上（xy、xz 或者 yz），然后在二维平面上来求解问题。

最后，由于数值精度问题，我们通常会在相交测试中使用一个很小的数字，这个数字被表示为 ϵ (epsilon)，它的值将因具体的测试情况而异。然而，通常会选择一个适用于问题案例的 ϵ （Press 等人[1446]称之为“便利的虚构（convenient fiction）”），而不是进行仔细的舍入误差分析和 ϵ 调整。在另一种环境中使用的这种代码很可能会由于条件不同而中断。Ericson 的书[435]在几何计算的背景下，对数值健壮性领域进行了深入讨论。这个警告实际上非常到位，因为我们有时确实会为“普通的”、小规模且靠近原点的数据（例如小于 100，大于 0.1），提供至少是合理的初始 ϵ 。

22.3 创建包围体

给定一个物体集合，为其找到一个紧密贴合的包围体，对于最小化相交测试的开销而言十分重要。对于一条任意的射线，它与凸物体相交的几率与该物体的表面积成正比（章节 22.4）。将这个面积最小化可以提高任何相交算法的效率，因为一次拒绝运算永远都要比相交运算快得多。相比之下，对于碰撞检测算法而言，通常最好的方法是对每个 BV 的体积进行最小化。本小节将会简要介绍对于一个给定的多边形集合，如何寻找最佳或者近似最佳的包围体。

22.3.1 创建 AABB 和 k-DOP

AABB 是创建起来最简单的包围体。沿着每个标准坐标轴，获取多边形顶点集合的最小范围和最大范围，就构成了一个 AABB。k-DOP 是对 AABB 的扩展：将顶点投影到 k-DOP 的每个法线 $\mathbf{n}_i$ 上，并将这些投影极值 $(\min, \max)$ 存储在 d^i_{min} 和 d^i_{max} 中。这两个值定义了该方向上贴合最紧密的 slab。所有这些值一起，共同定义了一个最小的 k-DOP。

22.3.2 创建球体

包围球的构建并不像确定 slab 的范围那么简单。有许多算法都可以完成这个任务，并且它们在速度和质量之间进行了权衡。一种快速的、恒定时间的单 pass 算法，是对多边形集合构建一个 AABB，然后使用这个包围盒的中心和对角线来构建包围球。这种方法有时会产生不太合适的结果，这可以通过另一个 pass 来进行改善：从将 AABB 的中心作为球体 BV 的中心开始，再次遍历多边形集合中的所有顶点，并找到距离该中心最远的顶点（使用距离的平方进行比较，从而避免开方运算）。然后将这个距离作为新的半径，如图 22.7 所示。

图 22.7：包围球。左边是最简单的方法，先构建这个物体的包围盒，然后再使用这个包围盒的外接球来作为这个物体的包围球。左边这种简单方法会出现很多冗余的空白空间，可以将这个包围盒的中心作为包围球的中心，然后再遍历所有顶点，来找到距离球心最远的顶点，并将这个距离作为包围球的半径，这样可以对包围球进行一些改进，如中间图像所示。如右图所示，可以通过移动球体的中心来进一步减小包围球的半径。

如果我们要在父球体中嵌套若干个子球体，那么上面这两种技术只需要进行稍微修改即可。如果所有子球体都具有相同的半径，那么可以将子球体的中心视为顶点，并在任意一个子过程结束的时候，将这个子半径添加到父球体的半径中。如果子球体的半径会发生变化，那么可以通过将这些半径纳入边界计算中来找到合理的中心点，从而找到这个 AABB 的边界。如果执行了第二个 pass，则将每个半径都添加到顶点到父节点中心的距离上。

Ritter [1500]提出了一个简单的算法，该方法可以创建一个接近最优的包围球。其思想是在 x 轴、 y 轴、 z 轴上，分别找到最小值顶点和最大值顶点。对于这三对顶点，找出其中间距最大的那一对，并使用这一对顶点来构建这个包围球，即球体中心位于这对顶点连线的中点处，球体半径等于这对顶点之间距离的一半。对所有其他的顶点进行遍历，检查它们到球心的距离 d 。如果顶点位于球体半径 r 之外（即 $d > r$ ），那么将球体中心向该顶点移动 $(d - r)/2$ ，并将半径设置为 $(d + r)/2$ ，然后

继续遍历。这步操作的效果是：将这个新顶点和现有的球体都包围在一个新的球体中。在第二次遍历这个顶点列表之后，包围球内就保证包含了所有的顶点。

Welzl [1867]提出了一种更加复杂的算法，Eberly [404, 1574]、Ericson [435]以及其他人在 web 上实现了这个算法，并提供了相应的代码。其思想是找到定义一个球面点的支撑集（supporting set）。一个球体可以由表面上的两个、三个或者四个点来进行定义。当发现一个顶点位于当前球体之外时，则将该顶点的位置添加到支撑集中（可能会有一些旧的支撑点从集合中被移除），然后计算新的球体，并再次遍历整个列表。重复执行这个过程，直到生成的球体包含所有顶点为止。虽然这个算法要比之前的算法更加复杂，但是它能够保证找到一个最优的包围球。

Ohlarik [1315]对 Ritter 算法和 Welzl 算法的速度进行了比较。其中简化形式的 Ritter 算法只比基础版本多带来了 20% 的开销，但是有时可能会得到更差的结果，因此这两种方法的运行都是值得的。而 Welzl 算法的 Eberly 实现，虽然复杂度与随机点列表的长度是线性相关的，但是其运行速度要慢一个数量级左右。

22.3.3 创建凸多面体

凸多面体是一种一般形式的包围体，凸物体可以用于分离轴测试。AABB、k-DOP、OBB 都是凸多面体，但是它们与物体之间的贴合不够紧密，存在一些贴合更加紧密的凸多面体。k-DOP 可以被认为是通过添加额外的平面来减少物体的体积；同理，凸多面体可以由任意一组平面来进行定义。通过减少额外的空白体积，可以避免对整个网格执行更加昂贵的相交测试。我们想要对多边形物体进行“收缩包裹（shrink-wrap）”，并找到一组平面，这组平面会形成一个凸包（convex hull），如图 22.8 所示。凸包可以使用 Quickhull 算法来找到 [100, 596]。尽管这个算法名称中包含了 Quick，但是实际上这个过程要比线性时间慢，因此通常会作为复杂模型的离线预处理来进行执行。

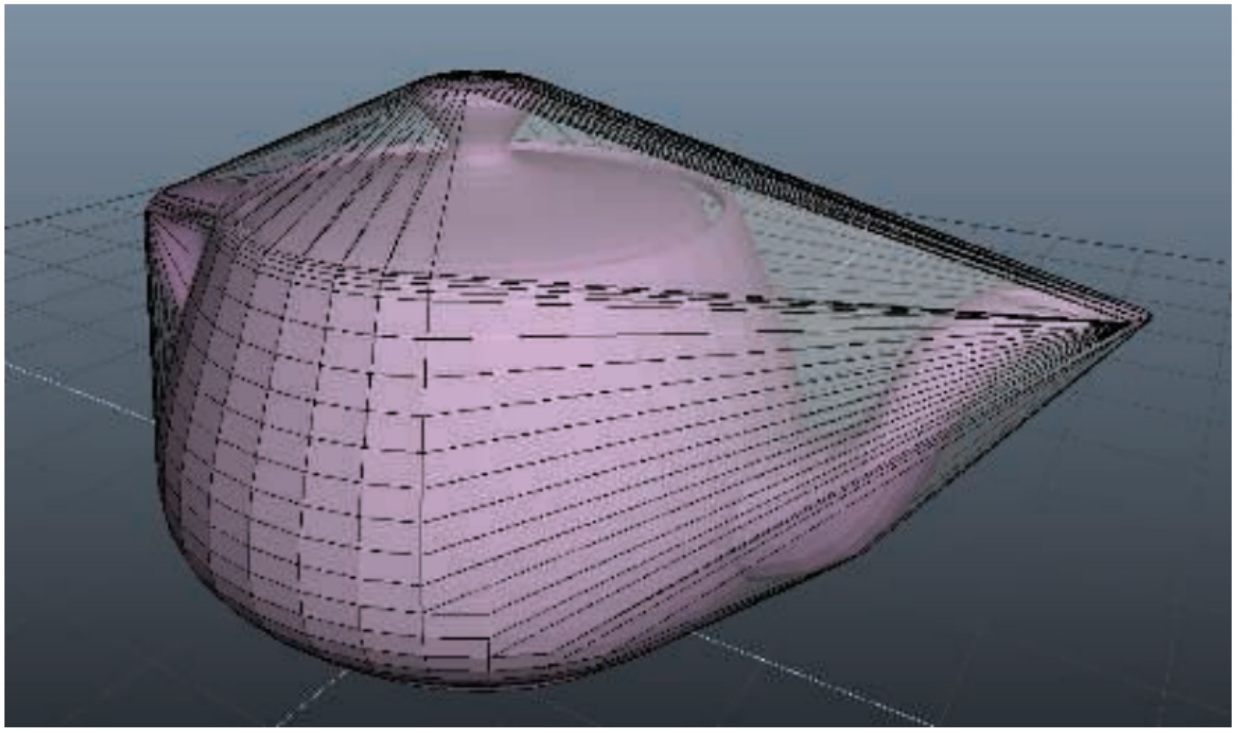


图 22.8：用 Quickhull 计算出的茶壶凸包。[596]

我们可以看到，这个过程可能会产生大量的平面，每个平面都由凸包上的一个多边形所定义。而在实践中，我们可能并不需要这种级别的精度。我们首先会创建一个原始网格的简化版本，这个版本可能会稍微向外扩展，从而将原始网格完全包含在内，这将会产生一个精度不高但是更加简单的凸包。还要注意的是，对于 k-DOP 而言，随着 k 的不断增大，所形成的 BV 会越来越像凸包。

22.3.4 创建 OBB

一个物体可能会具有一个天然的 OBB，因为它可以从一个 AABB 开始，然后经过一个旋转操作，从而使得 AABB 变成一个 OBB。然而，我们所使用的 OBB 可能并不是最优的。想象一下，现在有一个旗杆模型，它从建筑物中以一定的角度延伸出来，在其周围的 AABB 不如沿旗杆延伸的 OBB 那么紧密。对于没有明显最佳轴的模型而言，想要创建一个具有任意基底方向的 OBB，甚至要比寻找合理大小和位置的包围球更加复杂。

为这个问题建立解决算法已经做了大量的相关工作。O'Rourke [1338]在 1985 年给出了一个精确解决方案，其时间复杂度为 $O(n^3)$ 。Gottschalk [577]提出了一种更快、更简单的方法，该方法能够给出最佳 OBB 的近似值，该方法首先会计算多边形网格的凸包，从而避免凸包内的模型顶点对结果产生影响和偏移。然后再使用线性时间运行的主成分分析（principle component analysis，PCA）来寻找合理的 OBB 轴。这

种方法的缺点在于，最终形成的包围盒有时候是比较松散的，贴合没有那么紧密[984]。Eberly 描述了一种方法，该方法使用最小化技术来计算一个最小体积的 OBB。他对包围盒的一组可能方向进行采样，并使用 OBB 的最小坐标轴来作为数值最小化器（numeric minimizer）的起点。然后使用 Powell 的方向集方法[1446]，来寻找最小体积的包围盒。在网上有 Eberly 执行这个操作的相关代码[404]。还有一些其他的算法，Chang 等人[254]对之前的工作进行了合理的综述，并介绍了他们自己的最小化技术，该技术使用遗传算法来帮助对解空间进行搜索。

这里我们介绍一下 Larsson 和 Kallberg [984]提出的算法，这是一种不需要计算凸包，并且能够在线性时间内执行完成的近似最优方法。相较于 Gottschalk 基于 PCA 的方法，该方法可以提供更好的质量，并且执行速度要快得多，适合 SIMD 并行执行，并且作者还提供了实现的代码。该方法首先会为物体构建一个 k-DOP，并保存一对（任意一对）接触每个 k-DOP slab 的对边顶点。所有这些顶点对在一起，被称为物体的极值点（extremal point）。例如：一个 26-DOP 会生成 13 对点，其中一些点可能会是相同的顶点，因此可能会给出一个较小的总体集合。这个“最佳 OBB”首先会被初始化为物体周围的 AABB。然后，该算法会向着能够提供更好拟合 OBB 的方向来进行处理。会构造一个较大的基础三角形，并从其表面延伸出两个四面体。这会创建一组包含 7 个三角形的集合，这些三角形可能会产生接近最优的 OBB。

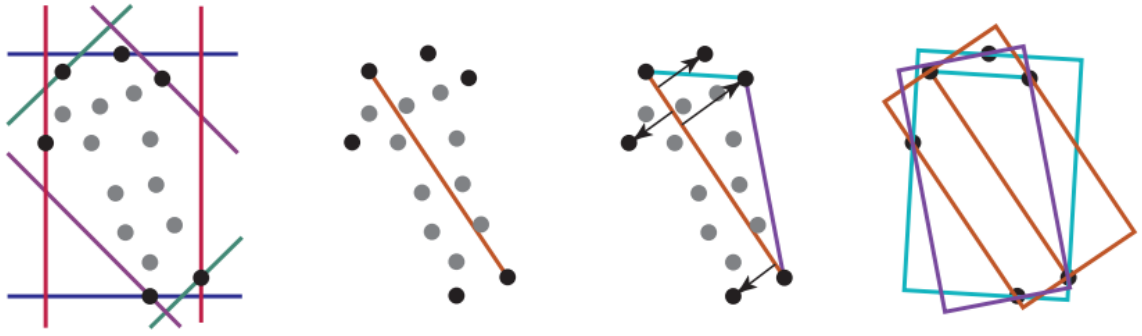


图 22.9：接近最优的 OBB 构建，需要记住的是，图中的所有点都位于三维空间中。对于每个 k-DOP slab（使用一对彩色线段进行表示），在其极限处有一对点（使用黑色进行表示），对于两个 slab 平面，底部的两个顶点分别位于两个 slab 平面的极值处。其他标记为灰色的顶点，在之后的步骤中不会进行使用。在这四对顶点中，相距最远的两个顶点会构成一条边；距离这条直线最远的那个极值点，与这条边一起形成了一个三角形。这样就形成了三个 box，其中每个 box 都会使用一条三角形边来定义自己的轴，并使用剩余的极值点来定义自己的边界。而在这三个 box 中，最好的那个 box 会被保存下来。

相距最远的一对点会构成基础三角形的其中一条边。而在剩余的极值点中，距离这条直线最远的那个顶点会构成三角形的第三个顶点。每个三角形的边和三角形平面上的边法线，都被用来构成潜在新 OBB 的两个轴。将剩余的极值点投影到这些轴上，从而找到每个 OBB（一共 3 个）在平面上的二维边界，如图 22.9 所示。使用最小的包围二维矩形，来从 3 个 OBB 中选择最优的那个 OBB。由于这三个 OBB 的高度是相同的，即沿三角形法线的距离是相同的，因此仅仅使用每个 OBB 周围的二维包围盒，就足以决定哪个 OBB 是最好的了。

然后使用剩余的极值点，在三角形的法线上通过投影操作，来在三维空间中找到这个 OBB 的具体范围。将这个最终形成的 OBB 与初始 AABB 进行对比检查，以确定具体哪个更好。在这个过程中会找到的两个极值点，一个在高度最大处，一个在高度最小处，然后使用原来的大三角形作为每个四面体的底，构成两个四面体。每个四面体会依次形成三个额外的三角形，并对每个三角形的三个候选 OBB 再次执行一遍评估过程，就像对原始三角形所做的那样。与之前一样，每个三角形的最佳二维 OBB 也会沿着其高度进行扩展，但是这次只是为了获得候选 OBB 的最终大小，而不是形成更多的三角形。总共会形成 7 个三角形，从每个三角形中生成二维 OBB 并进行比较，最终获得一个完整的 OBB。

一旦我们找到了最佳的 OBB，可以将原始物体中的所有点都投影到这个 OBB 的轴上，并根据需要来增加 OBB 的大小。最后再对原始的 AABB 进行检查，看看这个 OBB 是否更加合适，即是否能够更好地与模型相匹配。得益于在大多数步骤中都会使用一小组极值点，因此整个过程要比之前的技术更快。值得注意的是，作者更喜欢基于表面积而不是体积来对包围盒进行优化，原因我们将在下一小节中进行讨论。

22.4 几何概率

常见的几何运算包括一个平面或者一条射线是否与物体相交，以及某个点是否位于物体内部。一个与之相关的问题是：点、射线或者平面与物体相交的相对概率是多少呢？空间中随机一点在物体内部的相对概率是十分明显的：这个概率与物体的体积成正比。因此，对于空间中的一个随机点而言，一个 $1 \times 2 \times 3$ 的 box，它包含这个随机点的概率是 $1 \times 1 \times 1$ box 的 6 倍。

对于空间中的任意一条射线而言，这条射线与一个物体相交的相对概率是多少呢？这个问题与另一个问题是类似的：当使用正交投影的时候，一个具有任意朝向的物体，其所覆盖的平均像素数是多少？正交投影可以被认为是观察空间中的一组平行射线，

每个像素中都会穿过一条射线。给定一个随机朝向的物体，它所覆盖的像素数量等于与物体相交的数量。

背后的答案出奇地简单：任何凸实体（convex solid）物体的平均投影面积，是其表面积的四分之一。对于屏幕上的一个球体而言，这显然是正确的。这个球体的正投影区域总是一个面积为 πr^2 的圆，而这个球体的表面积为 $4\pi r^2$ 。这个比例也适用于其他任意朝向的凸物体（例如包围盒或者 k-DOP）的平均投影。请参阅 Nienhuys 的论文[1278]来了解一种非正式的证明。

球体、box 或者其他凸物体，在每个像素上总会有一个正面和一个反面，因此它们的深度复杂度为 2。这个概率指标也可以推广到任何多边形上，因为一个（双面）多边形的深度复杂度通常为 1。因此，任何多边形的平均投影面积，都是其表面面积的一半。

这个指标在光线追踪文献中被称为表面积启发式方法（surface area heuristic，SAH）[71, 1096, 1828]，它对于构建一个数据集的高效可见性结构而言非常重要。SAH 的其中一个用途就是对比包围体的效率，例如：与一个内接立方体（即一个角点接触到球面的立方体）相比，这个球体被射线击中的相对概率为 $1.57(\pi/2)$ 。类似地，与一个球体相比，这个球体的外接立方体被击中的相对概率为 $1.91(6/\pi)$ 。

这种类型的概率指标在计算 LOD 等领域非常有用，例如：想象现在有一个又长又薄的物体，它所覆盖的像素数量要比一个圆形物体少得多，虽然二者具有相同大小的包围球。通过包围盒的面积，我们可以提前知道命中率，因此这个长而薄的物体，在视觉影响中的重要性相对较低。

我们现在知道了：一个点被包围的概率与这个空间的体积有关；一条射线与物体相交的概率与这个物体的表面积有关。一个平面与一个 box 相交的概率，与这个 box 在三维空间中的范围（extent）总和成正比[1580]，这个总和被称为物体的平均宽度（mean width），例如：一个边长为 1 的立方体，其平均宽度为 $1+1+1=3$ 。一个 box 的平均宽度，与它和平面的相交概率成正比。因此，一个 $1\times1\times1$ 的 box，其平均长度为 3；一个 $1\times2\times3$ 的 box，其平均长度为 6，这意味着第二个 box 与任意平面相交的概率，是第一个 box 的两倍。

然而，这个总和要大于真正的几何平均宽度（geometric mean width），这里的几何平均宽度是指一个物体在所有可能的方向上，沿固定轴的平均投影长度。对于这个平均宽度的计算而言，不同凸物体类型之间并没有简单的关系（例如表面积）可以直接进行利用。一个直径为 d 的球体，其几何平均宽度为 d ，因为球体在任何方向上的跨度都是相同的。这里我们简单地介绍一下，将一个 box 的尺寸之和（即它的平均宽

度）乘以 0.5，就可以得到它的几何平均宽度，这个几何平均宽度可以直接与球体的直径进行比较。因此，一个尺寸为 3 的 $1 \times 1 \times 1$ box，它的几何平均宽度为 $3 \times 0.5 = 1.5$ 。而这个 box 的外接球，是一个直径为 $\sqrt{3} = 1.732$ 的球体。因此，对于一个立方体而言，其外界球与任意平面相交的可能性是这个立方体的 $1.732/1.5 = 1.155$ 倍。

这些概率关系有助于确定各种算法的优势。其中视锥体裁剪就是这样的一个算法，因为它涉及到与包围体相交的平面。这些关系的另一个用途是，确定是否以及在何处分割包含物体的 BSP 节点才是最好的，从而使得视锥体剔除的性能变得更好（章节 19.1.2）。

22.5 经验法则

在我们开始研究具体的求交方法之前，这里有一些经验法则，可以让我们获得速度更快、更加健壮、更加精确的相交测试。在设计、发明和实现相交程序的时候，应当牢记以下几点：

- 在早期执行一些计算和比较，使用计算和比较来拒绝或者接受各种类型的相交事件，从而尽早避免后续的计算。
- 如果可能的话，可以利用之前测试的结果。
- 如果使用了多个拒绝测试或者接受测试，那么可以尝试改变它们的内部执行顺序（如果可能的话），因为这样做可能会产生更加高效的测试。不要认为这些看起来很小的改变不会带来任何影响。
- 推迟那些开销很大的计算（尤其是三角函数、平方根和除法），直到真正需要进行计算的时再执行（章节 22.8 中给出了一种延迟执行昂贵除法的例子）。
- 通过降低问题的维度（例如：从三维降低为二维甚至是一维），通常可以大大简化相交问题。章节 22.9 中介绍了此类方法的例子。
- 如果一条射线或者一个物体同时与许多其他物体进行测试，可以在测试开始之前，查找那些只需要进行一次的预计算。
- 当相交测试的成本很高时，通常最好是从球体或者是其他简单的 BV 开始，从而提供第一级的快速拒绝。
- 养成在计算机上进行时间比较的习惯，并使用真实的数据和测试环境来进行时间比较。

- 利用前一帧的结果，例如：如果在前一帧中发现某个轴可以将两个物体分开，那么在下一帧中，优先尝试这个轴可能会是一个好主意。
- 最后，尽量让我们的代码变得健壮。这意味着它应当适用于所有的特殊情况，并且对尽可能多的浮点精度错误不敏感。我们要意识到代码可能具有的任何限制。有关数值健壮性和几何健壮性的更多信息，请参阅 Ericson 的书[435]。

最后，我们要强调这样的一个事实：对于一个特定的测试，很难确定是否存在一个“最佳”算法。我们在进行算法评估的时候，通常会使用一组具有不同预定命中率的随机数据，但是实际上这些随机数据只覆盖了部分的真实情况。我们需要将算法在真实场景中进行应用（例如在游戏中），并在该环境中进行最佳评估。使用的测试场景越多，我们对性能问题的理解就越好。对于某些架构而言（例如 GPU 和 wide-SIMD 实现），可能会因为需要执行多个拒绝分支而降低部分性能。我们最好要避免做假设，而是创建一个可靠的测试计划。

22.6 射线/球体相交

让我们从一个数学上比较简单的相交测试开始，即射线与球体之间的相交测试。正如我们稍后将看到的，如果我们开始考虑所涉及的几何问题，那么这种直接的数学解决方案可以变得更快[640]。

22.6.1 数学解法

球体可以由一个中心点 $\mathbf{c}$ 和一个半径 r 来进行定义。一个更紧凑的（与之前介绍的方程相比）球体隐式方程是：

$$f(\mathbf{p}) = \|\mathbf{p} - \mathbf{c}\| - r = 0 \tag{22.5}$$

其中点 $\mathbf{p}$ 是球面表面上的任意一点。想要求解射线与球体之间的交点，只需要使用射线 $\mathbf{r}(t)$ 替换方程 22.5 中的 $\mathbf{p}$ 即可，即：

$$f(\mathbf{r}(t)) = \|\mathbf{r}(t) - \mathbf{c}\| - r = 0 \tag{22.6}$$

由方程 22.1 可知，$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$，因此方程 22.6 可以化简为：

$$\|\mathbf{r}(t) - \mathbf{c}\| - r = 0$$
$$\Longleftrightarrow$$

$$
\begin{aligned}
&\|\mathbf{o}+t\mathbf{d}-\mathbf{c}\| = r \\
&\Longleftrightarrow \\
&(\mathbf{o}+t\mathbf{d}-\mathbf{c})\cdot(\mathbf{o}+t\mathbf{d}-\mathbf{c}) = r^2 \\
&\Longleftrightarrow \\
&t^2(\mathbf{d}\cdot\mathbf{d}) + 2t(\mathbf{d}\cdot(\mathbf{o}-\mathbf{c})) + (\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c}) - r^2 = 0 \\
&\Longleftrightarrow \\
&t^2 + 2t(\mathbf{d}\cdot(\mathbf{o}-\mathbf{c})) + (\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c}) - r^2 = 0
\end{aligned} \tag{22.7}
$$

在最后一步推导中，我们假设射线的方向 $\mathbf{d}$ 是归一化的，即 $\mathbf{d}\cdot\mathbf{d} = \|\mathbf{d}\|^2 = 1$ 。毫不意外，我们最终得到的方程是一个二次多项式，这意味着如果射线与球体相交，最多会有两点相交，如图 22.10 所示。如果方程的解是虚数的（即没有实数解），那么射线就会偏离球体。如果不是，那么我们可以将两个解 t_1 和 t_2 带入到射线方程中，从而计算球面上的交点。

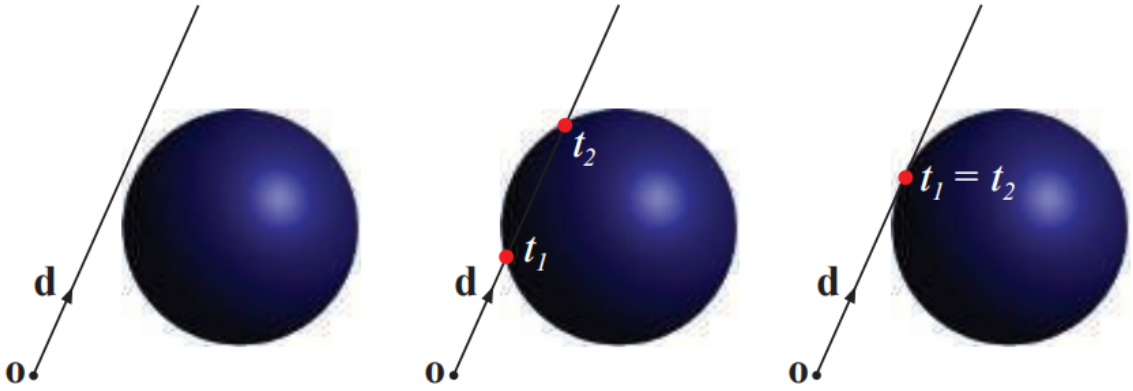


图 22.10：左侧图像中，一条射线没有击中球体，因此 $b^2 - c < 0$ 。中间的图像中。一条射线与球体有两个交点，即 $b^2 - c > 0$ ，可以通过两个参数标量 t_1 和 t_2 来找到这个交点。右侧图像中， $b^2 - c = 0$ ，这意味着两个交点重合。

得到的方程 22.7 也可以写成一个标准的二次方程形式：

$$
\begin{aligned}
t^2 + 2bt + c &= 0 \\
b &= \mathbf{d}\cdot(\mathbf{o}-\mathbf{c}) \\
c &= (\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c}) - r^2
\end{aligned} \tag{22.8}
$$

这个二次方程的解如下：

$$
t = -b \pm \sqrt{b^2 - c} \tag{22.9}
$$

请注意，如果 $b^2 - c < 0$，那么射线就不会与球体相交，因此可以避免一些计算（例如：平方根和一些加法）。如果这个测试通过，那么可以计算两个解 $t_0 = -b - \sqrt{b^2 - c}$ 和 $t_0 = -b + \sqrt{b^2 - c}$。还需要进行一次额外的比较，来找到 t_0 和 t_1 中的最小正值。请参阅在线网站 realtimerendering.com 上的碰撞检测章节（第 25 章），来了解另一种数值更加稳定的求解方法[1446]。

如果从几何角度来看待这些计算，那么我们可以发现一些更好的拒绝测试方法。我们将在下一小节中描述这样程序。

22.6.2 优化解法

对于射线/球体相交问题，我们首先观察到的是：位于射线原点后面的点不需要进行相交测试。例如：这通常会发生在拾取的时候。为了尽早检查出这种情况，我们会首先计算一个向量 $\mathbf{l} = \mathbf{c} - \mathbf{o}$，它是从射线原点指向球体中心的向量。图 22.11 中展示了所使用的符号表示。同样，会计算这个向量长度的平方，即 $l^2 = \mathbf{l} \cdot \mathbf{l}$。如果 $l^2 < r^2$，那么意味着射线原点位于球体内部，这意味着这条射线肯定会击中球体。如果我们只想检测射线是否击中球体的话，此时我们就可以结束测试了；如果我们想获得精确的交点，那么可以继续进行计算。接下来，计算向量 $\mathbf{l}$ 在光线方向 $\mathbf{d}$ 上的投影：$s = \mathbf{l} \cdot \mathbf{d}$。

现在我们来进行第一个拒绝测试：如果 $s < 0$，并且射线原点位于球体外部，那么则说明球体位于射线原点背后，因此我们可以拒绝相交。否则，我们需要使用勾股定理来计算从球心到投影点距离的平方：$m^2 = l^2 - s^2$。这里我们进行第二个拒绝测试，这个测试要比第一个更加简单：如果 $m^2 > r^2$，那么射线一定会偏离球体，剩下的计算可以直接省略。如果球体与射线通过了最后的测试，那么这个射线就一定会击中球体，如果这就是我们想要知道的，那么此时也可以退出计算了。

想要找到真正的交点坐标，还需要进行一些工作。首先，计算距离的平方 $q^2 = r^2 - m^2$。

> 这个标量 r^2 可以只计算一次，然后存储在球体的数据结构中，从而进一步提高效率。在实践中，这种“优化”反而会变慢，因为需要访问更多的内存，这是影响算法性能的一个主要因素。

如图 22.11 所示，由于 $m^2 \leq r^2$，因此 $q^2 \geq 0$，这意味着我们可以计算出 $q = \sqrt{q^2}$。最后，到交点的距离为 $t = s \pm q$，这个解与章节 22.6.1 中通过数学求解得到的二次方程的解十分相似。如果我们只对第一个正交点感兴趣，那么我们可以使用 $t_1 = s - q$ 来表示射线原点在球外时的情况，使用 $t_2 = s + q$ 来表示射线原点在球

内时的情况。通过将这个 t 值代入到射线方程（方程 22.1）中，便可以找到真正的交点（ s ）。

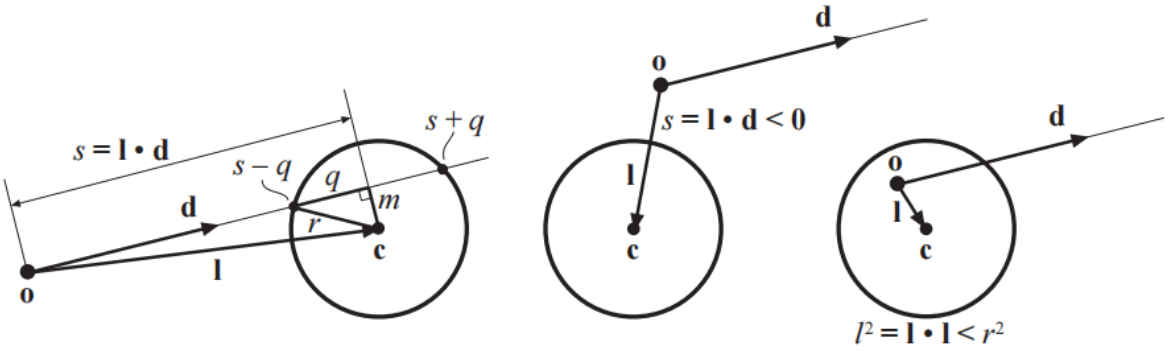


图 22.11：优化的射线/球体相交的几何符号表示。左侧：射线与球面相交于两个点，沿着射线方向上的距离分别是 $t = s \pm q$ 。中间：展示了当球体位于射线原点背后时所产生的拒绝情况。右侧：射线原点位于球体内部，在这种情况下，射线总是会命中球体。

优化版本的伪代码如下所示。这个函数会返回三个参数，分别是：一个布尔值，一个代表相交距离的标量，以及交点坐标。如果射线与球面相交，则会返回一个代表 REJECT 的布尔值，并将后面两个参数置为默认值 0 和 $(0, 0, 0)$ 。如果射线与球面相交，则会返回一个代表 INTERSECT 的布尔值，并返回射线原点到交点的距离 t ，以及交点坐标 $\mathbf{p}$ 。

```
     RaySphereIntersect(o, d, c, r)
     returns ({REJECT, INTERSECT}, t, p)
1 :  l = c − o
2 :  s = l · d
3 :  l² = l · l
4 :  if(s < 0 and l² > r²) return (REJECT, 0, 0);
5 :  m² = l² − s²
6 :  if(m² > r²) return (REJECT, 0, 0);
7 :  q = √(r² − m²)
8 :  if(l² > r²) t = s − q
9 :  else t = s + q
10 : return (INTERSECT, t, o + td);
```

请注意，在伪代码的第 3 行之后，我们可以判断点 $\mathbf{p}$ 是否位于球体内部，如果我们只想知道射线是否会与球面相交，那么这个函数可以在判断是否相交之后直接结束。并且在第 6 行之后，这个射线肯定会击中球体。如果我们执行这些操作之后（加、

乘、比较等操作），我们会发现，所得到的几何解与之前的代数解十分类似。二者之间的重要区别在于，几何解法中的拒绝测试会在过程中更早地进行，这使得该算法的平均总成本更低。

对于计算射线与其他二次曲面的交点、以及混合物体的交点，存在一些优化的几何算法。例如：圆柱体[318, 713, 1621]、圆锥体[713, 1622]、椭球体、胶囊体和含片体[404]等，都有一些优化方法。

22.7 射线/Box 相交

下面给出了三种判断射线是否与实心 box 相交的方法。第一种方法是处理 AABB 和 OBB。第二种是一个变体方法，通常这种方法的速度要更快，但是它只能处理一些更加简单的 AABB。第三种方法基于分离轴测试，它只能处理线段与 AABB。这里，我们使用章节 22.2 中 BV 的定义和符号表示。

22.7.1 平板法

射线–AABB 相交的一种方案是基于 Kay 和 Kajiya 的平板（slab）法[640, 877]，该方法的灵感来自 Cyrus–Beck 的直线裁剪算法[319]。

我们对该方案进行了扩展，使其能够处理更加一般的 OBB，它可以返回距离最近的正 t 值（即从射线原点 $\mathbf{o}$ 到交点的距离，如果存在的话）。有关针对 AABB 的优化，我们将在介绍一般情况之后再进行处理。这个问题是通过计算射线与属于 OBB 的所有平面的相交 t 值来解决的。这个 box 会被认为是一个包含三个 slab 的集合，如图 22.12 左侧二维示意图。对于每个 slab，都有一个最小值 t_i^{min} 和一个最大值 t_i^{max}，其中 $i \in \{u, v, w\}$。下一步是计算方程 22.10 中的变量：

$$\begin{aligned} t^{\min} &= \max\left(t_u^{\min}, t_v^{\min}, t_w^{\min}\right), \\ t^{\max} &= \min\left(t_u^{\max}, t_v^{\max}, t_w^{\max}\right). \end{aligned} \tag{22.10}$$

现在有一个巧妙的测试：如果 $t^{min} \leq t^{max}$，那么由这个由射线所定义的直线与 box 相交；反之就不相交。换句话说，我们找到每个 slab 的远相交距离和近相交距离。如果找到的最远的“近”距离小于或者等于最近的“远”距离，那么这条射线所定义的直线就击中了 box。仔细观察图 22.12 右侧的插图，我们就可以明白这一点。这两个距离实际上定义了射线与直线的交点，因此，如果这个最近的“远”距离是非负的，则说明射线本身与 box 相交，即 box 位于射线的前方。

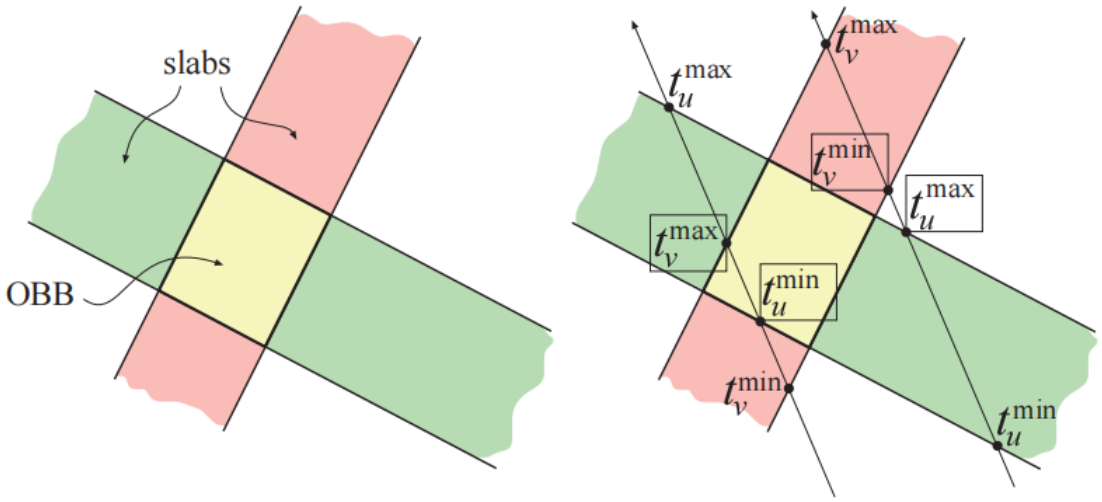


图 22.12：左侧展示了由两个平板构成的二维 OBB，右侧展示了两条射线，这两条射线与 OBB 相交。所有相交的 t 值都被显示了出来，其中与绿色 slab 的交点下标为 u，与橙色 slab 的交点下标为 v。t 值的极值使用方框进行标记。左边的射线击中了这个 OBB，因为自 $t^{min} < t^{max}$；而右边的射线则没有击中这个 OBB，因为 $t^{max} < t^{min}$。

下面给出了射线与 OBB 相交测试的伪代码，即 OBB （A）和射线（由方程 22.1 描述）之间的相交测试。这段代码会返回两个值：第一个值是一个布尔值，代表光线是否与 OBB 相交（相交或者拒绝），第二个值是射线原点到交点的距离（如果存在的话）。回顾一下，对于 OBB A 而言，其中心点表示为 $\mathbf{c}$；u、v 和 w 是这个 box 归一化的侧边方向；h_u、h_v 和 h_w 是正的半长距离（从中心到各个 box 面的距离）。

```
     RayOBBIntersect(o, d, A)
     returns ({REJECT, INTERSECT}, t);
1 :  t^min = −∞
2 :  t^max = ∞
3 :  p = a^c − o
4 :  for each i ∈ {u, v, w}
5 :     e = a^i · p
6 :     f = a^i · d
7 :     if(|f| > ε)
8 :        t1 = (e + h_i)/f
9 :        t2 = (e − h_i)/f
10 :       if(t1 > t2) swap(t1, t2);
11 :       if(t1 > t^min) t^min = t1
12 :       if(t2 < t^max) t^max = t2
13 :       if(t^min > t^max) return (REJECT, 0);
14 :       if(t^max < 0) return (REJECT, 0);
15 :    else if( − e − h_i > 0 or  − e + h_i < 0) return (REJECT, 0);
16 : if(t^min > 0) return (INTERSECT, t^min);
17 : else return (INTERSECT, t^max);
```

第 7 行检查了射线方向是否垂直于当前被测试 slab 的法线。换句话说，它对射线是否平行于 slab 平面进行了测试，如果二者平行，则不可能相交；如果射线与 slab 平面不平行，那么二者才可能会相交。请注意， ϵ 在这里代表一个很小的数字，约为 10^{-20} ，这是为了避免在除法时发生数值溢出。第 8 行和第 9 行中包含了对 f 的除法，实际上，可以先计算一次 $1/f$ ，然后再使用这个值来进行相乘，这样做通常会更快，因为除法运算的开销通常会很大。第 10 行确保了 t_1 和 t_2 中的最小值会存储在 t_1 中；同样，它们中的最大值会存储在 t_2 中。实际上，并不一定要真的进行这个交换操作；相反，可以对这个分支重复执行第 11 行和第 12 行， t_1 和 t_2 可以在那里改变它们的位置。如果函数在第 13 行返回，则说明射线没有击中 box；类似地，如果在第 14 行返回，则说明 box 位于射线原点的后面。如果这条射线平行于待测 slab（因此无法与这个 slab 相交），则执行第 15 行，它会测试这条射线是否位于 slab 外面。如果射线位于 slab 外面，则代表射线不会击中 box，此时终止测试。对于执行速度更快的代码[640]，Haines 讨论了一种展开循环的方法，从而可以避免执行一些代码。

这段伪代码中没有展示另外的一个测试，这个测试在实际代码中值得被添加。正如我们在定义射线时所提到的，我们通常想要找到距离射线原点最近的物体。因此，在第 15 行之后，我们还可以测试 $t^{min} \geq l$ ，其中 l 是当前射线的长度。这样可以将射线视为一段线段，如果新的交点要大于目前已知的最近交点，则拒绝这个交点。这个测

试可以推迟到整个射线与 OBB 测试完成之后再进行，但是在循环内部尝试早期拒绝，通常会更加高效。

对于 OBB 的特殊情况（即 AABB），还有一些其他优化。例如：第 5 行和第 6 行可以更改为 $e = p_i$ 和 $f = d_i$，这样可以使得测试速度更快。通常在第 8 行和第 9 行中可以使用 AABB 的最小角 $\mathbf{a}^{min}$ 和最大角 $\mathbf{a}^{max}$，这样可以避免一些加法和减法。Kay 和 Kajiya [877]以及 Smits [1668]注意到，可以通过允许除 0 并正确解释处理的结果，从而避免第 7 行中的判断。Kensler [1629]给出了这个测试的最短版本代码。Williams 等人[1887]提供了正确处理除 0 的实现细节，以及其他的一些优化。Aila 等人[16]展示了如何在某些 NVIDIA 架构上的单个 GPU 操作中，执行这个最大的最小值测试，反之亦然。也可以使用 SAT 来对射线与 box 进行测试，但是无法获得相交距离，有时候这是有用的。

可以将平板法推广开来，用于计算射线与 k-DOP、视锥体或者任何凸多面体的交点，相关代码可以在网上找到[641]。

22.7.2 射线斜率法

2007 年，Eisemann 等人[410]提出了一种 box 求交的方法，这种方法似乎要比之前的方法更快。与之前三维测试不同的是，在这个方法中，射线会在二维空间中对 box 的三个投影来进行测试。该方法的核心想法是：对于每个二维测试，会有两个 box 角点来定义射线能够“看到”的极限范围，类似于模型的轮廓边缘（silhouette）。为了使得射线与这个 box 的投影相交，对射线的斜率有一定的要求，它必须要在射线原点和这两个角点所定义的斜率之间。如果三次投影测试都通过了，那么这个射线就一定会击中 box。这个方法的速度非常快，因为其中一些的比较项完全依赖于射线的值。通过对这些项进行一次计算，可以高效地将射线与大量 box 进行比较测试。这个方法可以只返回 box 是否被击中的布尔值，也可以返回交点的距离，但是需要一些额外的开销。

22.8 射线/三角形相交

在实时图形库和 API 中，三角形通常会被存储为一组具有相应着色法线的顶点，每个三角形都是由三个这样的顶点定义的。三角形所在平面的法线通常不会进行存储，在这种情况下，如果我们需要使用这个法线数据的话，就必须手动计算它。目前有许多针对射线与三角形相交测试的不同方法，其中许多方法首先会计算射线与三角形平面的交点。然后，再将交点和三角形顶点投影到某个坐标轴对齐的平面上（xy、yz、xz），要求这个平面上能够使得三角形的面积最大化。通过这样做，我们可以将

问题简化为二维问题，只需要判断（二维）点是否位于（二维）三角形内部即可。有几种这样思路的方法，Haines [642]对这些方法进行了审查和比较，并在网上提供了相应的代码。在章节 22.9 中介绍了使用这种技术的一个流行算法。在不同的 CPU 架构、编译器和命中率[1065]下，对大量算法进行了评估，我们无法得到一个在所有情况下都是最佳的方法。

这里我们将重点关注这样的一种算法，该算法不会假设我们已知三角形的法线，对于三角形网格而言，这样可以显著节省内存占用。对于动态几何而言，我们并不需要每帧都重新计算三角形的平面方程。我们并不是先针对三角形的平面来与射线进行测试，然后再去检查射线与平面的交点是否位于这个二维三角形的内部，而是会直接针对三角形的顶点执行这个测试。Moller 和 Trumbore [1231]讨论了该算法的实现以及优化，这里将使用他们的相关展示内容。Kensler 和 Shirley [882]指出，大多数直接在三维空间中进行的射线/三角形测试，在计算上实际是等效的。他们开发了一种新的测试方法，使用 SSE 来测试三角形上的四条射线，并在这个等价的测试中，使用遗传算法来找到最佳的操作顺序。他们的论文中展示了表现最好的测试代码。请注意，有许多不同的方法都可以做到这一点。例如：Baldwin 和 Weber [96]提供了一种具有不同空间速度权衡的方法。这类测试的一个潜在问题是：恰好与三角形边或者顶点相交的射线，可能会被判定为没有击中这个三角形。这意味着一条射线可能会击中两个三角形之间的共享边界，从而穿过网格。Woop 等人[1906]提出了一种射线/三角形相交测试，这个测试在边和顶点上都是无懈可击的。根据所使用的遍历类型，这个方法的性能表现会有所降低。

使用方程 22.1 中的射线表示方法，来与由三个顶点 $\mathbf{p}_1$ 、 $\mathbf{p}_2$ 、 $\mathbf{p}_3$ 所定义的三角形（即 $\triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ ）进行相交测试。

22.8.1 相交算法

三角形上的一个点 $\mathbf{f}(u,v)$ 由下面这个显式方程给出：

$$\mathbf{f}(u,v) = (1-u-v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2 \tag{22.11}$$

其中 (u,v) 是重心坐标，它必须满足 $u \geq 0$ ， $v \geq 0$ ， $u+v \leq 1$ 。请注意，这里的重心坐标 (u,v) 可以用于纹理映射，以及法线插值或者颜色插值等操作。也就是说， u 和 v 代表了每个顶点对特定位置的贡献权重， $w = (1-u-v)$ 是第三个权重。这些坐标在其他的一些资料中通常被表示为 α 、 β 和 γ 。这里为了可读性和符号的一致性，我们仍然使用 u 、 v 、 w 来进行表示，如图 22.13 所示。

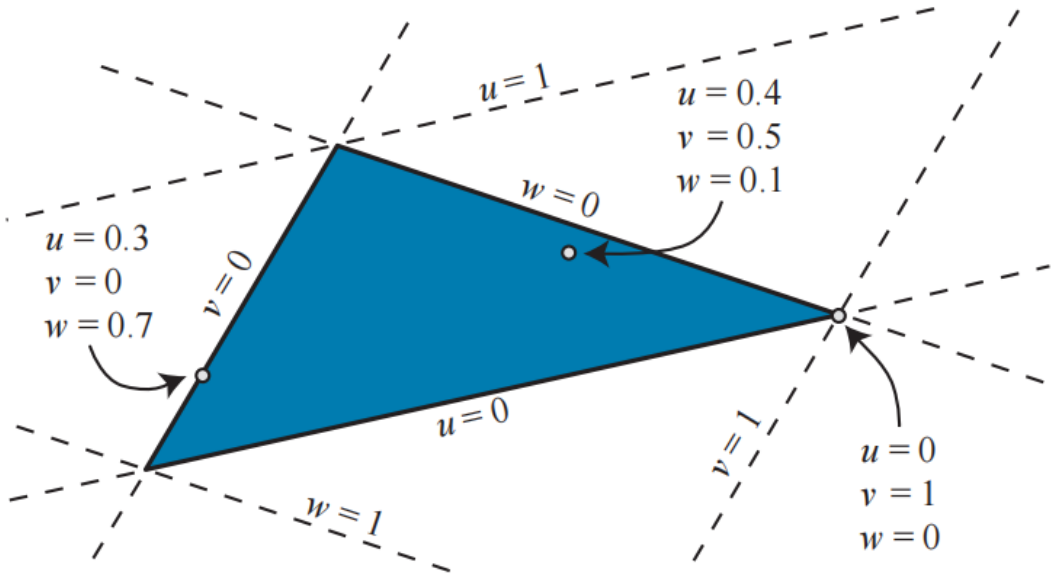


图 22.13：三角形的重心坐标，以及示例点对应的重心坐标。u 、v 、w 的值在三角形内都是从 0 到 1 变化的，这三个值的和在整个平面上始终为 1。这些值可以用作权重，来表示三角形顶点上的数据，是如何影响三角形上任意一个点的。请注意，在三角形的每个顶点上，有一个值为 1，而其他值为 0；而在三角形的边上，会有一个值始终为 0。

计算射线 $\mathbf{r}(t)$ 和三角形 $\mathbf{f}(u,v)$ 的交集，即 $\mathbf{r}(t)=\mathbf{f}(u,v)$，结果为：

$$\mathbf{o}+t\mathbf{d}=(1-u-v)\mathbf{p}_0+u\mathbf{p}_1+v\mathbf{p}_2 \tag{22.12}$$

整理可得：

$$\begin{pmatrix} -\mathbf{d} & \mathbf{p}_1-\mathbf{p}_0 & \mathbf{p}_2-\mathbf{p}_0 \end{pmatrix}\begin{pmatrix} t \\ u \\ v \end{pmatrix}=\mathbf{o}-\mathbf{p}_0 \tag{22.13}$$

这意味着重心坐标 (u,v)，以及从射线原点到交点的距离 t，可以通过求解这个线性方程组得到。

方程 22.13 中的操作，在几何上可以理解为：将这个三角形平移到原点上，并将其转换为 y 和 z 方向上的单位三角形，并使得射线方向与 x 轴对齐，如图 22.14 所示。如果记方程 22.13 中的矩阵为 $\mathbf{M}=\begin{pmatrix} -\mathbf{d} & \mathbf{p}_1-\mathbf{p}_0 & \mathbf{p}_2-\mathbf{p}_0 \end{pmatrix}$，那么方程 22.13 与 $\mathbf{M}^{-1}$ 相乘即可获得解，即 $(t,u,v)^T$ 。

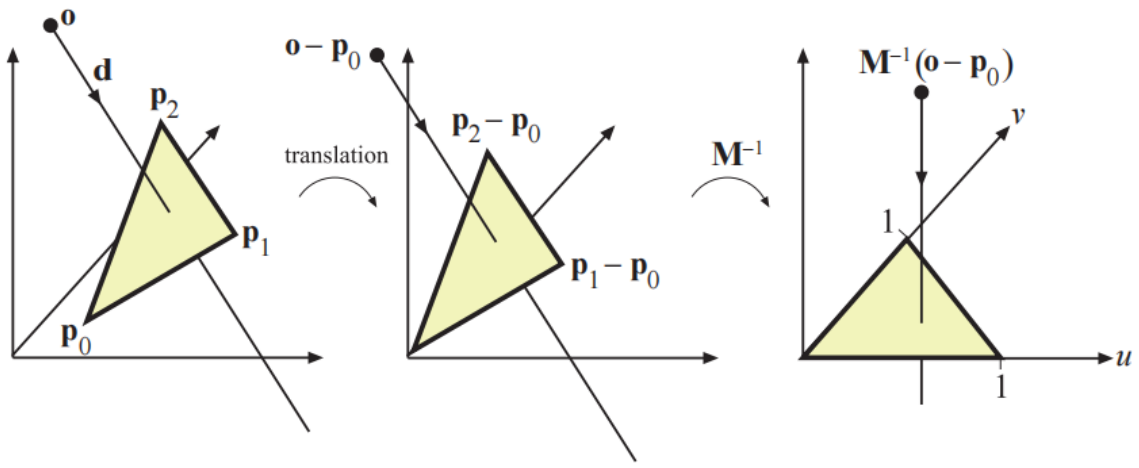


图 22.14：射线原点的平移和变化。

设 $\mathbf{e}_1 = \mathbf{p}_1 - \mathbf{p}_0$， $\mathbf{e}_2 = \mathbf{p}_2 - \mathbf{p}_0$， $\mathbf{s} = \mathbf{o} - \mathbf{p}_0$，利用 Cramer 法则，可以获得方程 22.13 的解：

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix} \tag{22.14}$$

根据线性代数的一些知识，我们可以知道：

$$\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) = |\mathbf{abc}| = -(\mathbf{a} \times \mathbf{c}) \cdot \mathbf{b} = -(\mathbf{c} \times \mathbf{b}) \cdot \mathbf{a}$$

因此，方程 22.14 可以重写为：

$$\begin{aligned} \begin{pmatrix} t \\ u \\ v \end{pmatrix} &= \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{pmatrix} \\ &= \frac{1}{\mathbf{q} \cdot \mathbf{e}_1} \begin{pmatrix} \mathbf{r} \cdot \mathbf{e}_2 \\ \mathbf{q} \cdot \mathbf{s} \\ \mathbf{r} \cdot \mathbf{d} \end{pmatrix} \end{aligned} \tag{22.15}$$

其中 $\mathbf{q} = \mathbf{d} \times \mathbf{e}_2$， $\mathbf{r} = \mathbf{s} \times \mathbf{e}_1$。这些项只需要提前计算一次，用来加快计算速度。

如果我们能够负担得起额外的存储空间，可以对这个测试进行重新排序，从而减少计算次数。方程 22.15 可以改写为：

$$
\begin{aligned}
\begin{pmatrix} t \\ u \\ v \end{pmatrix} &= \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{pmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{pmatrix} \\
&= \frac{1}{-(\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{d}} \begin{pmatrix} (\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{d}) \cdot \mathbf{e}_2 \\ -(\mathbf{s} \times \mathbf{d}) \cdot \mathbf{e}_1 \end{pmatrix} \\
&= \frac{1}{-\mathbf{n} \cdot \mathbf{d}} \begin{pmatrix} \mathbf{n} \cdot \mathbf{s} \\ \mathbf{m} \cdot \mathbf{e}_2 \\ -\mathbf{m} \cdot \mathbf{e}_1 \end{pmatrix},
\end{aligned}
\tag{22.16}
$$

其中 $\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_2$ 是三角形的非归一化法线，对于静态几何物体而言，它是一个常量；以及 $\mathbf{m} = \mathbf{s} \times \mathbf{d}$ 。如果我们为每个三角形存储 $\mathbf{p}_0$ ， $\mathbf{e}_1$ ， $\mathbf{e}_2$ 和 $\mathbf{n}$ ，我们可以避免许多射线与三角形相交测试中的计算量，大部分降低计算量的收益都来自于避免了叉乘计算。但是需要注意的是，这样的存储方法违背了该算法的原始思想，即在每个三角形上存储最少的信息。然而，如果对于这个测试而言，速度指标是最重要的，那么这可能也是一个合理的选择。这里还有另一个权衡，即我们在降低计算开销的同时，还增加了更多的内存访问，内存访问也会降低一些性能表现。只有经过仔细全面的测试，才能最终证明具体哪种方法才是最快的。

22.8.2 实现

下面的伪代码对该算法进行了总结。除了返回射线是否与三角形相交之外，该算法还会返回前面所描述的三元组 (u, v, t) 。下面的代码并不会剔除背面三角形，它将会返回 t 值为负的交点；如果需要的话，这些交点也可以被剔除。

```
     RayTriIntersect(o, d, p0, p1, p2)
     returns ({REJECT, INTERSECT}, u, v, t);
1 :  e1 = p1 − p0
2 :  e2 = p2 − p0
3 :  q = d × e2
4 :  a = e1 · q
5 :  if(a > −ε and a < ε) return (REJECT, 0, 0, 0);
6 :  f = 1/a
7 :  s = o − p0
8 :  u = f(s · q)
9 :  if(u < 0.0) return (REJECT, 0, 0, 0);
10 : r = s × e1
11 : v = f(d · r)
12 : if(v < 0.0 or u + v > 1.0) return (REJECT, 0, 0, 0);
13 : t = f(e2 · r)
14 : return (INTERSECT, u, v, t);
```

其中有几行可能需要解释一下。第 4 行计算了 a，即矩阵 $\mathbf{M}$ 的行列式。接下来的第五行，是一个避免行列式接近于零的检验。使用一个适当的 ϵ，能够使得这个算法变得更加健壮；对于浮点数精度和一些“正常”的条件而言，$\epsilon = 10^{-5}$ 是一个比较好的选择。在第 9 行中，将 u 的值与三角形的一条边进行比较，如果刚好位于三角形的一条边上，那么 $u = 0$。

可以在网上找到该算法的 C 语言代码（包括剔除和非剔除的版本）[1231]，这个 C 语言代码有两个分支：其中一个用于高效地剔除所有背面三角形，另一个用于对双面三角形执行相交测试。所有的计算都会被延迟到需要的时候再进行，例如：直到我们确认 u 的值在允许的范围内，才会去计算 v 的值（这一点在伪代码中也可以看到）。

单面三角形的相交测试，会消除了所有行列式值为负的三角形。这个过程允许将程序中唯一的除法操作延迟进行，直到一个交叉点被确认之后，我们才真正执行这个除法操作。

22.9 射线/多边形相交

尽管三角形是最常见的渲染图元，但是拥有一个计算射线和多边形相交的算法还是非常有用的。一个包含 n 个顶点的多边形，由一个有序的顶点列表 $\{\mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_{n-1}\}$ 进行定义，其中当 $0 \leq i < n-1$ 的时候，顶点 $\mathbf{v}_i$ 与

$\mathbf{v}_{i+1}$ 形成一条边；这个多边形最终被从 $\mathbf{v}_{n-1}$ 到 $\mathbf{v}_0$ 的边所封闭。这个多边形所在的平面可以表示为 $\pi_p : \mathbf{n}_p \cdot \mathbf{x} + d_p = 0$ 。

我们首先计算射线（方程 22.1）和 π_p 之间的交点，这可以很容易地通过用使用射线方程替换 $\mathbf{x}$ 来完成。这个方程的解如下：

$$\mathbf{n}_p \cdot (\mathbf{o} + t\mathbf{d}) + d_p = 0 \quad \Longleftrightarrow \quad t = \frac{-d_p - \mathbf{n}_p \cdot \mathbf{o}}{\mathbf{n}_p \cdot \mathbf{d}} \qquad (22.17)$$

如果判别式 $|\mathbf{n}_p \cdot \mathbf{d}| < \epsilon$ ，其中 ϵ 是一个极小的数，那么我们可以认为射线与多边形平行，不存在交点。在这个计算中， ϵ 可以取 10^{-20} 或者更小的值，其目的是避免在进行除法运算时发生数值溢出。这里我们忽略射线位于多边形平面内的情况。

如果通过了判别式，那么我们就可以计算射线与多边形平面的交点 $\mathbf{p}$ ： $\mathbf{p} = \mathbf{o} + t\mathbf{d}$ ，其中 t 值就是方程 22.17 中的值。然后，判断交点 $\mathbf{p}$ 是否位于多边形内部的问题，会由三维问题简化为一个二维问题。这是通过将多边形的所有顶点以及交点 $\mathbf{p}$ ，投影到 $xy-$ 、 $xz-$ 或者 $yz-$ 的其中一个平面上来实现的，具体要投影在哪个平面上，需要使得投影后的多边形面积最大。也就是说，可以跳过绝对值最大的坐标分量 $\max\left(|n_{p,x}|, |n_{p,y}|, |n_{p,z}|\right)$ ，并将其他分量保留为二维坐标。例如：给定一个法线 $(0.6, -0.692, 0.4)$ ，其中 y 分量的绝对值最大，因此所有的 y 坐标都会被忽略。这里选择最大的那个绝对值，是为了避免投影到一个可能产生退化三角形（零面积）的平面上。请注意，这个分量信息可以进行预先计算，并存储在多边形中以提高效率。在这个投影过程中，多边形和交点的拓扑结构是不变的（即假设这个多边形是平坦的；有关这个主题的更多信息，详见章节 16.2）。投影过程如图 22.15 所示。

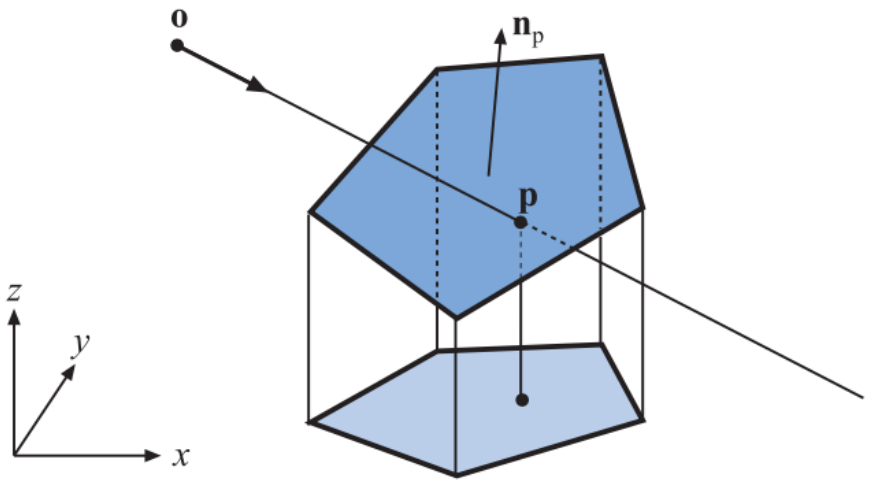


图 22.15：多边形顶点与交点 $\mathbf{p}$ 在 xy 平面上的正交投影，这个投影可以使得多边形的投影面积最大。这是一个使用降维来获得更简便计算的例子。

剩下的问题是，这个射线与平面的二维交点 $\mathbf{p}$，是否位于这个二维多边形中。在这里，我们将回顾其中一个十分有用的算法——“交叉点（crossings）”测试。Haines [642]、Schneider 和 Eberly [1574]对判断二维点是否位于多边形内部的策略进行了广泛的调研。在计算几何的相关文献中可以找到更正式的处理方法[135, 1339, 1444]。Lagae 和 Dutre [955]基于 Moller 和 Trumbre 的射线/三角形检验方法，提供了一种快速判断射线/四边形相交的方法。Walker [1830]提供了一种能够快速测试 10 个以上顶点多边形的方法。Nishita 等人[1284]讨论了具有弯曲边缘形状的点包含测试。

22.9.1 交叉点测试

交叉点测试基于 Jordan 曲线定理，这是拓扑学中的结果。如果从某个点出发，沿平面任意方向的射线会穿过奇数条的多边形边界，那么这个点位于多边形内部。Jordan 曲线定理实际上只适用于非自相交的环。而对于自相交的环来说，这种射线测试会导致多边形内部的一些区域被认为是外部的，如图 22.16 所示。这种检验也称为奇偶校验（parity， even–odd test）。

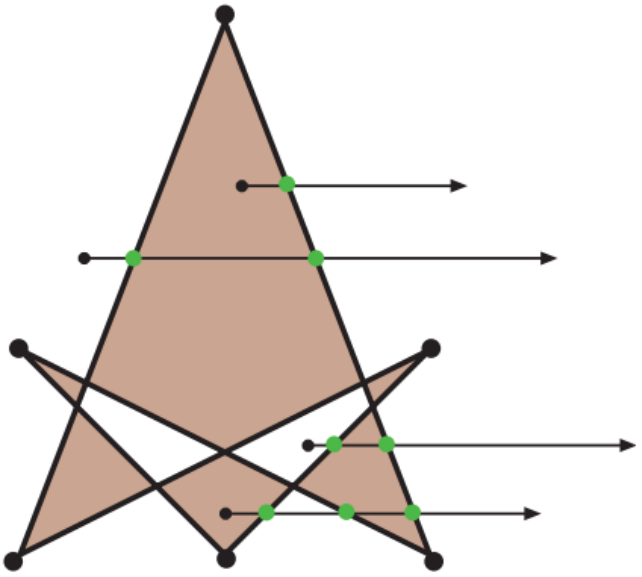


图 22.16：一种自相交的凹多边形，但是其中的封闭区域并不都是被认为位于多边形内部的（只有棕色区域才会被认为是内部的）。图中多边形的顶点使用较大的黑点进行标记。图中显示了四个待测点以及它们的测试射线。根据 Jordan 曲线定理，如果一个点与多边形边界相交的次数为奇数，那么这个点就位于多边形内部。因此，最上面的点和最下面的点都位于内部（分别有一个交叉点和三个交叉点）。中间两个点的射线与两条边相交，因此被认为位于多边形外部。

交叉点测试算法的工作原理是：从点 $\mathbf{p}$ 的投影向 x 轴正方向（或者任意方向，使用 x 正方向的编码很高效）发射一条射线，然后计算多边形边界与这条射线之间的交叉次数。正如 Jordan 曲线定理所证明的那样，相交次数为奇数，代表该点位于多边形内部。

也可以将测试点 $\mathbf{p}$ 放置在原点处，针对 x 轴正方向上的（平移后的）边进行测试，如图 22.17 所示。如果多边形边界的 y 坐标具有相同的符号，则说明这条边界不会与 x 轴相交；否则说明可以与 x 轴相交。然后再检查 x 坐标，如果二者都为正，则增加交叉点的数量，因为测试射线必须到达这条边。如果二者的正负号不同，则必须计算边界线与 x 轴交点的坐标，如果交点的 x 正分量为正，则增加交叉的数量。

在图 22.17 中，所有的封闭区域都可以被归类为内部区域。这个变体测试方法找到了缠绕数（winding number），即多边形环路围绕某个测试点的次数，详见 Haines 的论文[642]。

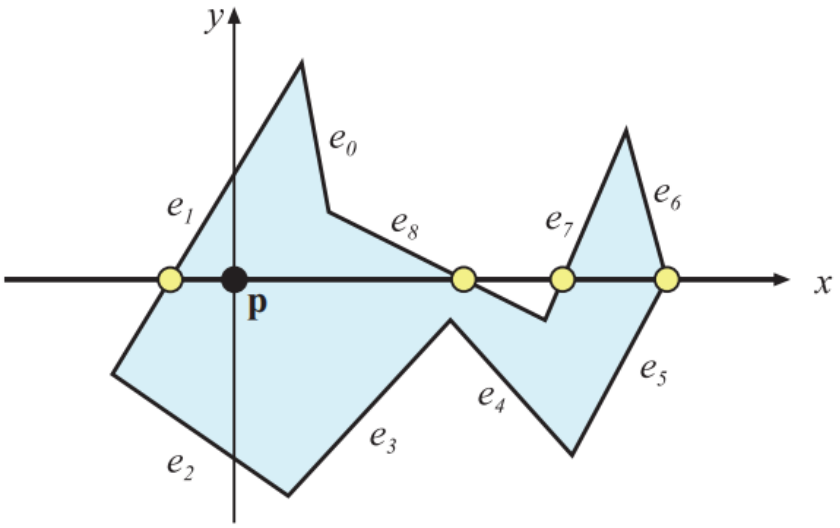


图 22.17：这个多边形被平移了 $-\mathbf{p}$ （点 $\mathbf{p}$ 是待测点），因此与正 x 轴的交叉次数决定了点 $\mathbf{p}$ 是否位于多边形内部。边 e_0 ， e_2 ， e_3 ， e_4 不与 x 轴相交。边 e_1 与 x 轴的交点必须要进行计算，但是它并不会产生交叉，因为交点坐标的 x 分量是负的。边 e_7 和边 e_8 都会增加交叉点的数量，因为这两条边的两个顶点都具有正的 x 分量，以及以一正一负的 y 分量。最后，边 e_5 和边 e_6 共享一个顶点，其中 $y = 0$ 而 $x > 0$ ，它们在一起将使得交叉点的数量增加 1。通过将 x 轴上的顶点视为在射线上方，边 e_5 会被归类为穿过射线，而边界 e_6 则会被归类为位于射线上方。

当测试射线与多边形顶点相交时可能会出现一些问题，因为此时可能会检测到两个交叉点。这些问题是这样进行解决的：可以认为多边形顶点位于射线上方的无穷小处；

实际上，这是通过将 $y \geq 0$ 的顶点解释为位于 x 轴（射线）上方来实现的。这样代码将会变得更加简洁和高效，因为不会有顶点与射线相交[640]。

下面是一种高效交叉点测试的伪代码。它的灵感来自 Joseph Samosky [1537]和 Mark Haigh-Hutchinson 的工作，其代码可以在网上找到[642]。它将一个二维测试点 $\mathbf{t}$ ，与顶点从 $\mathbf{v}_0$ 到 $\mathbf{v}_{n-1}$ 的多边形 P 进行比较。

```
      bool PointInPolygon(t, P)
      returns ({TRUE, FALSE});
1 :   bool inside = FALSE
2 :   e0 = vn−1
3 :   bool y0 = (e0y ≥ ty)
4 :   for i = 0 to n − 1
5 :      e1 = vi
6 :      bool y1 = (e1y ≥ ty)
7 :      if(y0 ≠ y1)
8 :         if(((e1y − ty)(e0x − e1x) ≥ (e1x − tx)(e0y − e1y)) == y1)
9 :            inside = ¬inside
10 :     y0 = y1
11 :     e0 = e1
12 :  return inside;
```

其中第 3 行检查了多边形最后一个顶点的 y 值，是否大于等于待测点 $\mathbf{t}$ 的 y 值，并将结果存储在布尔值 y_0 中。换句话说，它实际上是测试了第一条边的第一个端点，是位于 x 轴之上还是 x 轴之下。第 7 行测试了端点 e_0 和端点 e_1 是否位于 x 轴的不同侧。如果是，那么第 8 行则测试 x 轴上的截距是否为正。实际上，上面的伪代码要更快一些：为了避免在计算截距时通常需要进行的除法，我们在这里将执行一个消号（sign-canceling）操作。通过将 inside 取反，第 9 行记录了一个交叉点。第 10 行到第 12 行则转移到了对下一个顶点进行迭代。

在上述伪代码中，我们没有在第 7 行之后执行测试，以查看与测试点相比，两个端点的 x 坐标是更大还是更小。虽然我们在算法中使用了快速接受或者快速拒绝这些类型边界的测试，但是基于上述伪代码的实际代码，在没有这个测试的情况下通常会运行得更快。一个主要因素是待测多边形的顶点数量——顶点数量越多，先检查 x 坐标的差异就会越高效。

交叉点测试的优点在于相对快速和健壮，并且不需要额外的信息或者是对多边形进行预处理。这个方法的一个缺点在于，除了能够判断一个点是位于多边形内部还是外部之外，它不会产生任何的其他结果。其他的一些方法（例如章节 22.8.1 中的射线/三

角形测试），还可以计算重心坐标，用于对测试点的额外信息进行插值[642]。请注意，重心坐标可以进行一些扩展，从而处理包含 3 个以上顶点的凸多边形和凹多边形[474, 773]。Jimenez 等人[826]提供了一种基于重心坐标的优化算法，该算法的目标是包含多边形边缘上的所有点，并与交叉点测试相竞争。

判断一个点是否位于由线段和 Bezier 曲线所组成的封闭轮廓内部，是一个更加一般的问题。Bezier 曲线可以使用类似的方式进行算法执行，通过对射线的交叉点进行计数，来判断点是否位于曲线内部。Lengyel [1028]为这个过程提供了一种健壮的算法，在像素着色器中可以使用这种算法来渲染文本。

22.10 平面/Box 相交

我们能够知道一点到平面的距离，只要把这个点代入到平面方程 $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ 中即可。计算结果的绝对值就是这个点到平面的距离。平面/球体的相交测试也很简单：将球心带入到平面方程中即可，来看看计算结果的绝对值是否小于或者等于球体的半径。

判断一个 box 是否与平面相交的一种方法是：将这个 box 的所有顶点都代入到平面方程中。如果结果的符号不一致（或者为 0），那么说明这些顶点位于平面的两侧（或者位于平面上），因此可以检测到相交。还有一些更加巧妙、更加快速的方法来完成这个测试，我们将在接下来的两小节中进行介绍，其中一节针对 AABB，另一节针对 OBB。

这两种方法背后的思想都是：只需要将八个顶点中的两个代入到平面方程中即可。对于任意朝向的 box 而言，无论它是否与平面相交，box 上存在两个沿着对角线相对的顶点，当沿着任意平面的法线进行测量时，这两个顶点之间的距离都是最大距离。每个 box 有 8 个顶点，一共有四条对角线。让每条对角线方向与平面法线进行点乘，其中值最大的那个对角线代表了这两个相距最远的顶点。通过对这两个顶点进行测试，可以将这个 box 作为一个整体，来与平面进行相交测试。

22.10.1 AABB

假设我们现在有一个 AABB B 。它由一个中心点 $\mathbf{c}$ 和一个正半对角向量 $\mathbf{h}$ 定义。注意，这个中心点 $\mathbf{c}$ 和正半对角向量 $\mathbf{h}$ 可以很容易地从 B 的最小顶点 $\mathbf{b}^{min}$ 和最大顶点 $\mathbf{b}^{max}$ 中推导出来，即：

$$\mathbf{c} = \left(\mathbf{b}^{\max} + \mathbf{b}^{\min}\right)/2$$

$$\mathbf{h} = \left(\mathbf{b}^{\max} - \mathbf{b}^{\min}\right)/2$$

现在，我们要用平面 $\mathbf{n} \cdot \mathbf{x} + d = 0$ 来对这个 AABB 进行相交测试。有一种非常快的方法可以执行这个测试，其想法是计算这个 box 投影到平面法线平面 $\mathbf{n}$ 上的“范围（extent）”，这里记为 e 。理论上，这可以通过将 box 所有 8 条不同的半对角线都投影到法线上，并选择其中最长的一条来完成。但是在实践中，这个过程可以快速实现为：

$$e = h_x \left|n_x\right| + h_y \left|n_y\right| + h_z \left|n_z\right| \tag{22.18}$$

为什么方程 22.18 等价于找到八个不同对角线投影中的最大值呢？这 8 条对角线是这样的若干组合： $\mathbf{g}^i = (\pm h_x, \pm h_y, \pm h_z)$ ，而我们想要计算所有 8 个 i 对应的 $\mathbf{g}^i \cdot \mathbf{n}$ 。当点积中的每一项都为正时，点积 $\mathbf{g}^i \cdot \mathbf{n}$ 将达到最大值。对于其中的 x 项，当 n_x 和 h_x^i 具有相同符号的时候，才会出现这种最大情况，由于我们已经知道 h_x 是正的了，因此我们可以通过 $h_x \left|n_x\right|$ 来计算这个最大项。对于 y 项和 z 项重复这样的操作，便可以得到方程 22.18。

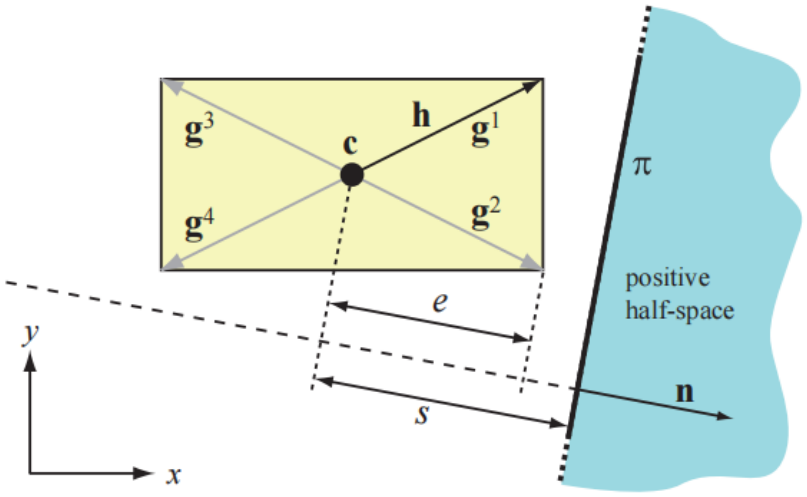


图 22.18：一个中心点为 $\mathbf{c}$ ，正半对角线为 $\mathbf{h}$ 的 AABB，在平面 π 上进行测试。其思想是计算从 box 中心到平面的带符号距离 s ，并将其与 box 的“范围” e 进行比较。图中向量 $\mathbf{g}^i$ 是这个二维 box 的不同对角线，而在这个例子中 $\mathbf{h}$ 等于 $\mathbf{g}^1$ 。还需要注意的是，当这个带符号距离 s 为负，且其绝对值大于 e 时，则代表这个 box 位于位于平面内部（ $s + e < 0$ ）。

接下来，我们计算从中心点 $\mathbf{c}$ 到这个平面的带符号距离 s 。这是通过 $s = \mathbf{c} \cdot \mathbf{n} + d$ 来完成的。图 22.18 给出了 s 和 e 的示意图。假设平面的“外部”是正半空间，我们可以进行简单地测试，如果 $s - e > 0$ ，则代表 box 完全位于平面外部；类似地，

如果 $s+e<0$，则代表 box 完全位于平面内部。否则，这个 box 会与平面相交。这项技术基于了 Ville Miettinen 的想法，以及他的巧妙实现。其伪代码如下：

```
    PlaneAABBIntersect(B, π)
    returns({OUTSIDE, INSIDE, INTERSECTING});
1 : c = (b^max + b^min)/2
2 : h = (b^max − b^min)/2
3 : e = h_x|n_x| + h_y|n_y| + h_z|n_z|
4 : s = c · n + d
5 : if(s − e > 0) return (OUTSIDE);
9 : if(s + e < 0) return (INSIDE);
10 : return (INTERSECTING);
```

22.10.2 OBB

在运行过程中对 OBB 和平面进行相交测试，与上一小节中的 AABB 与平面的相交测试略有不同。只有 box 的“范围” e 需要进行一些修改，具体的修改如下：

$$e = h_u^B \left|\mathbf{n}\cdot\mathbf{b}^u\right| + h_v^B \left|\mathbf{n}\cdot\mathbf{b}^v\right| + h_w^B \left|\mathbf{n}\cdot\mathbf{b}^w\right| \tag{22.19}$$

回顾一下，$\left(\mathbf{b}^u, \mathbf{b}^v, \mathbf{b}^w\right)$ 是 OBB 的坐标系轴，$\left(h_u^B, h_v^B, h_w^B\right)$ 是 box 沿着这些轴的长度，详见章节 22.2 中的 OBB 定义。

22.11 三角形/三角形相交

由于三角形是图形硬件所使用的最重要（和针对性优化）的图元，因此对这类数据进行碰撞检测也是很自然的。因此，在碰撞检测算法的最底层，通常会有一个判断两个三角形是否相交的例程。给定两个三角形 $T_1 = \triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ 和 $T_2 = \Delta\mathbf{q}_1\mathbf{q}_2\mathbf{q}_3$（二者分别位于平面 π_1 和 π_2 上），我们想确定它们是否相交。

从较高的层次来看，我们通常会首先检查三角形 T_1 是否与平面 π_2 相交，以及三角形 T_2 是否与平面 π_1 相交[1232]。如果其中的任何一个测试失败的话，就说明这两个三角形不可能有交集。假设这两个三角形不是共面的，我们能够知道这两个平面（π_1 和 π_2）的交集是一条直线 L，如图 22.19 所示。由图可知，如果这两个三角形相交，那么它们在 L 上的交点也会重叠。否则，这两个三角形就没有交集。有许多不同的方法可以实现这一点，接下来我们将介绍 Guigue 和 Devillers 的方法[622]。

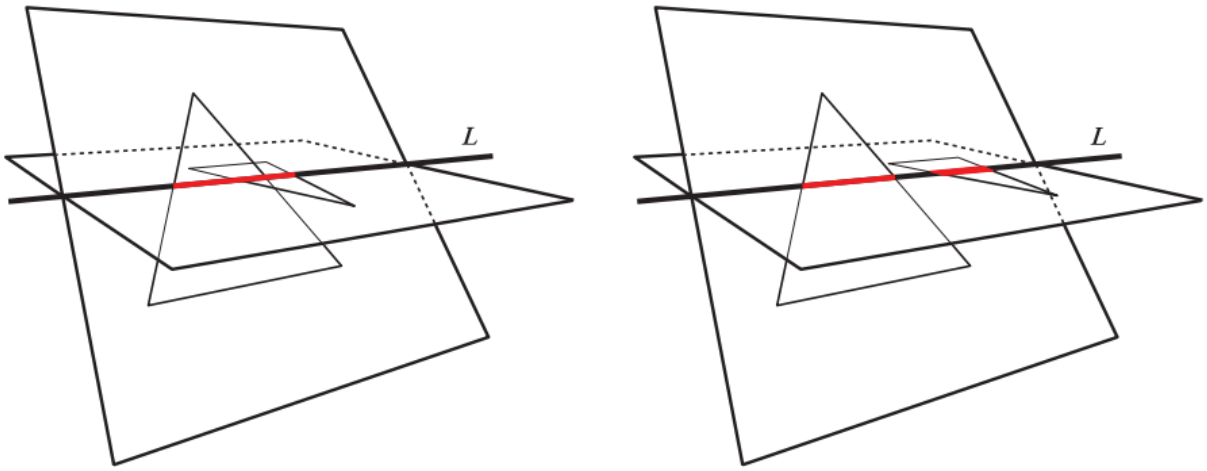


图 22.19：三角形和它们所在的平面。在两幅图中，交集部分使用红色进行标记。左：沿直线 L 的交叉部分发生重叠，这两个三角形也相交。右：两个三角形没有交集，三角形在直线 L 上的对应部分也不重叠。

在这个实现中，对 $\mathbf{a}$ 、 $\mathbf{b}$ 、 $\mathbf{c}$ 、 $\mathbf{d}$ 四个三维向量大量使用了下面这个 4×4 行列式：

$$[\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}] = -\begin{vmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 1 & 1 & 1 & 1 \end{vmatrix} = (\mathbf{d}-\mathbf{a}) \cdot ((\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})) \tag{22.20}$$

在几何上，方程 22.20 有一个直观的解释。其中的叉乘 $(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})$ 可以看作是计算三角形 $\triangle\mathbf{abc}$ 的法线。通过取这个法线与向量 $\mathbf{ad}$ （ $\mathbf{d}-\mathbf{a}$ ）之间的点积，如果点 $\mathbf{d}$ 位于三角形 $\triangle\mathbf{abc}$ 平面的正半空间中，那么我们将会得到一个正值。另一种解释是，行列式的符号将会告诉我们，沿着 $\mathbf{b}-\mathbf{a}$ 方向的螺旋方向（screw）是否与 $\mathbf{d}-\mathbf{c}$ 的相同，如图 22.20 所示。

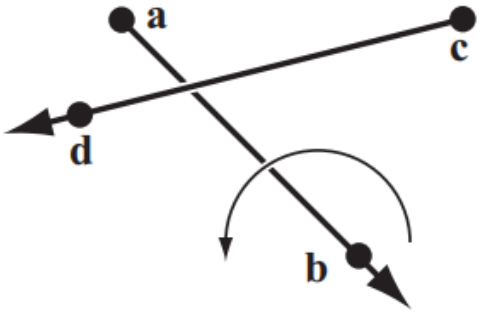


图 22.20：螺旋矢量 $\mathbf{b}-\mathbf{a}$ 在 $\mathbf{d}-\mathbf{c}$ 方向上的图示。

我们首先测试三角形 T_1 是否与平面 π_2 相交，然后判断三角形 T_2 是否与平面 π_1 相交。这可以使用方程 22.20 中的特殊行列式来完成，通过计算 $[\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{p}_1]$，$[\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{p}_2]$，$[\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{p}_3]$。其中第一个测试相当于计算三角形 T_2 的法线，然后测试点 $\mathbf{p}_1$ 位于哪一半空间中。如果这些行列式的正负号都相同且不为零，那么则说明 T_1 与平面 π_2 不可能有交集，测试到此结束。如果行列式的值全部为零，那么则说明这两个三角形共面，后续需要进行单独的相交测试。否则，我们需要继续测试三角形 T_2 是否与平面 π_1 相交，这里使用的是相同的测试方法。

此时，我们将计算直线 L 上的两个区间 $I_1 = [i, j]$ 和 $I_2 = [k, l]$，其中区间 I_1 是根据三角形 T_1 计算的，区间 I_2 是根据三角形 T_2 计算的。为了实现这一点，我们需要将每个三角形的顶点重新进行排序，使得第一个顶点单独位于另一个三角形平面的一侧。如果区间 I_1 与区间 I_2 重叠，当且仅当 $k \leq j$ 且 $i \leq l$ 时，这两个三角形才会相交。为了实现 $k \leq j$，我们可以使用行列式的符号检验（方程 22.20），请注意 j 是由 $\mathbf{p}_1\mathbf{p}_2$ 推导而来，k 是由 $\mathbf{q}_1\mathbf{q}_2$ 推导而来的。利用行列式计算的“螺旋检验”解释，我们可以得出，当 $[\mathbf{p}_1, \mathbf{p}_2, \mathbf{q}_1, \mathbf{q}_2] \leq 0$ 时，$k \leq j$。因此，最后的检验（$k \leq j$ 且 $i \leq l$）为：

$$[\mathbf{p}_1, \mathbf{p}_2, \mathbf{q}_1, \mathbf{q}_2] \leq 0 \quad and \quad [\mathbf{p}_1, \mathbf{p}_3, \mathbf{q}_3, \mathbf{q}_1] \leq 0 \qquad (22.21)$$

整个测试从 6 个行列式测试（两次三角形–平面测试）开始，其中前 3 个测试共用第一个参数，因此可以重用很多中间计算。原则上来说，行列式可以使用许多更小的 2×2 子行列式来进行计算，当这些子行列式重复出现在 4×4 行列式中时，计算结果可以进行共享。网上可以找到这个测试的相应代码[622]，也可以对代码进行扩充，来计算实际的交点线段。

如果两个三角形共面，则可以将它们投影到与一个轴对齐的平面上，在该平面上使得投影三角形的面积最大化（章节 22.9）。然后，进行一个简单的二维三角形重叠测试。首先，测试三角形 T_1 的所有闭合边（包括端点）是否与三角形 T_2 的闭合边相交。如果我们能够找到任何交点，则说明这两个三角形相交。否则，我们就必须测试三角形 T_1 是否被完全包含在三角形 T_2 中，反之亦然。这可以通过对三角形 T_1 的每个顶点，针对三角形 T_2 进行三角形内点测试来实现（章节 22.8），反之亦然。

请注意，分离轴测试可以用于推导出一个三角形与三角形的相交测试。但是这里我们只介绍了 Guigue 和 Devillers [622]所提出的一个测试，它要比使用 SAT 算法更快。

还有一些其他的算法可以实现三角形与三角形之间的相交测试[713, 1619, 1787]。由于架构和编译器之间的差异、以及预期命中率的差异，这意味着我们无法推荐一种性能总是最好的算法。请注意，与任何几何测试一样，这个测试也可能会出现精度问题。Robbins 和 Whitesides [1501]使用了 Shewchuk [1624]所提出的精确算法，来避免出现这种情况。

22.12 三角形/Box 相交

本小节将会介绍一种判断三角形是否与 AABB 相交的算法。这样的测试对于体素化和碰撞检测十分有用。

Green 和 Hatch [581]提出了一种算法，可以判断任意多边形是否与 box 相重叠。Akenine–Moller [581]开发了一种基于分离轴测试的方法，这种方法的速度更快，我们将在这里介绍这种方法。三角形与球体的相交测试也可以使用这个方法来实现，详见 Ericson 的文章[440]。

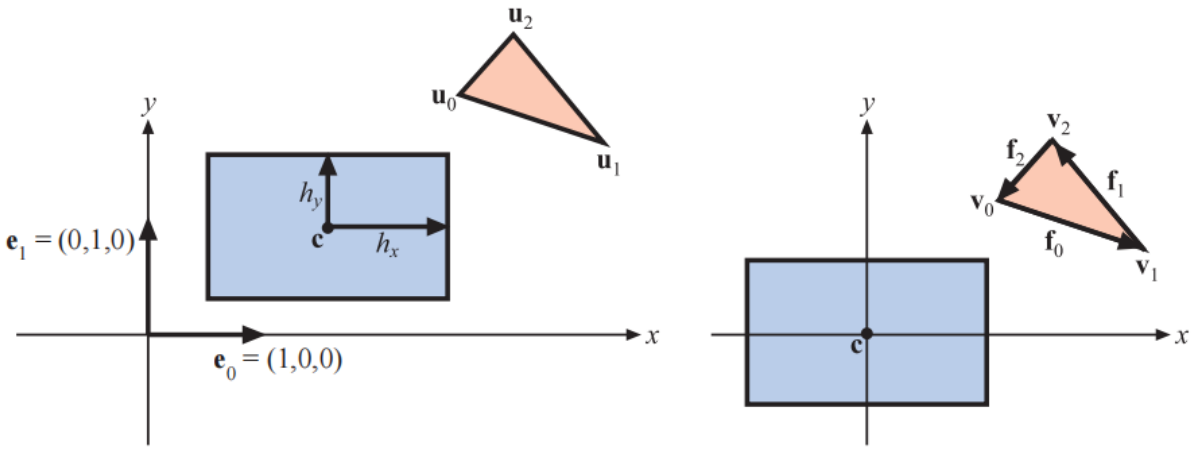


图 22.21：用于三角形与 box 重叠测试的符号表示。在左边，展示了 box 和三角形的初始位置。在右边，box 和三角形进行了平移，使得 box 的中心与原点重合。

这里我们将针对一个轴对齐包围盒（AABB）来进行测试，其中这个 AABB 由中心点 $\mathbf{c}$，以及一个半长向量 $\mathbf{h}$ 进行定义，待测三角形为 $\Delta\mathbf{u}_0\mathbf{u}_1\mathbf{u}_2$。为了简化测试过程，我们首先对 box 和三角形进行移动，使得这个 box 以原点为中心，即 $\mathbf{v}_i = \mathbf{u}_i - \mathbf{c}, i \in \{0,1,2\}$。这个平移过程以及所使用的符号表示方法如图 22.21 所示。如果想要对一个 OBB 进行测试，我们首先要需要对 OBB 和三角形顶点进行反向的旋转变换，然后再应用这里的测试方法。根据分离轴测试（SAT），我们会对以下 13 个轴进行测试：

1. [3 次测试] $\mathbf{e}_0 = (1,0,0)$， $\mathbf{e}_1 = (0,1,0)$， $\mathbf{e}_2 = (0,0,1)$ （AABB 的法线）。换句话说，将这个 AABB 与三角形周围的最小 AABB 进行测试。
2. [1 次测试] $\mathbf{n}$， $\Delta\mathbf{u}_0\mathbf{u}_1\mathbf{u}_2$ 的法线。这里我们使用快速的平面与 AABB 的重叠测试（章节 22.10.1），它只会对 box 对角线上的两个顶点进行测试，其中这个对角线的方向与三角形的法线最为接近。
3. [9 次测试] $\mathbf{a}_{ij} = \mathbf{e}_i \times \mathbf{f}_j, i,j \in \{0,1,2\}$，其中 $\mathbf{f}_0 = \mathbf{v}_1 - \mathbf{v}_0$， $\mathbf{f}_1 = \mathbf{v}_2 - \mathbf{v}_1$， $\mathbf{f}_2 = \mathbf{v}_0 - \mathbf{v}_2$，即边向量。这些测试在形式上是相似的，下面我们将只展示 $i = 0$ 和 $j = 0$ 时的推导（详见下文）。

一旦我们找到了一个分离轴，那么算法就会终止并返回“没有重叠”。如果上述所有的测试都通过了，也就是说没有找到分离轴，那么说明这个三角形与 box 会发生重叠。

在这里，我们将推导步骤 3 中 9 个测试中的一个，其中 $i = 0$， $j = 0$。这意味着 $\mathbf{a}_{00} = \mathbf{e}_0 \times \mathbf{f}_0 = (0, -f_{0z}, f_{0y})$。因此，现在我们需要将三角形顶点投影到 $\mathbf{a}_{00}$ 上（以下称为 $\mathbf{a}$）：

$$\begin{aligned} p_0 &= \mathbf{a}\cdot\mathbf{v}_0 = (0,-f_{0z},f_{0y})\cdot\mathbf{v}_0 = v_{0z}v_{1y} - v_{0y}v_{1z} \\ p_1 &= \mathbf{a}\cdot\mathbf{v}_1 = (0,-f_{0z},f_{0y})\cdot\mathbf{v}_1 = v_{0z}v_{1y} - v_{0y}v_{1z} = p_0 \\ p_2 &= \mathbf{a}\cdot\mathbf{v}_2 = (0,-f_{0z},f_{0y})\cdot\mathbf{v}_2 = (v_{1y}-v_{0y})\,v_{2z} - (v_{1z}-v_{0z})\,v_{2y}. \end{aligned} \tag{22.22}$$

通常，我们需要找到 $\min(p_0,p_1,p_2)$ 和 $\max(p_0,p_1,p_2)$，但幸运的是这里 $p_0 = p_1$，这简化了计算。现在我们只需要找到 $\min(p_0,p_2)$ 和 $\max(p_0,p_2)$ 即可，这要快得多，因为条件语句在现代 CPU 上的开销很大。

在将三角形投影到 $\mathbf{a}$ 上之后，我们还需要将 box 投影到 $\mathbf{a}$ 上。我们需要对投影到 $\mathbf{a}$ 上的 box 计算一个“半径” r：

$$r = h_x\,|a_x| + h_y\,|a_y| + h_z\,|a_z| = h_y\,|a_y| + h_z\,|a_z| \tag{22.23}$$

其中最后一步来自于这个特定轴上的 $a_x = 0$。那么，这个轴测试就变成了：

$$\text{if}\,(\min(p_0,p_2) > r \text{ or } \max(p_0,p_2) < -r)\ \text{return false;} \tag{22.24}$$

相关代码可以在网上找到[21]。

22.13 BV/BV 相交

包围体（bounding volume，BV）的目的是提供更加简单的相交测试，以及进行更加有效的拒绝测试，例如：假设我们想要测试两辆车是否发生碰撞，那么首先找到它们的 BV，并测试 BV 之间是否发生重叠。如果它们没有发生重叠，那么这两辆汽车就保证不会发生碰撞（这是最常见的情况）。这样我们就避免了对一辆车的每个图元都进行相交测试，从而节省了很多的计算量。

一个基本的操作是判断两个包围体是否重叠。下面我们将介绍 AABB、k-DOP、OBB 之间的重叠测试方法。有关在图元周围构建 BV 的算法，详见章节 22.3。

除了球体和 AABB 之外，我们还会使用一些更加复杂的 BV，其原因在于更加复杂的 BV 通常有着更加紧密的贴合程度，如图 22.22 所示。当然，使用一些其他的包围体也是可能的，例如：圆柱体和椭球体有时也会被用作物体的包围体。此外，还可以放置若干个球体来包围一个物体[782, 1582]。

对于胶囊体和含片体这样的 BV，计算最小距离是一个相对较快的操作。因此，它们经常用于公差验证（tolerance verification）的应用中，在这些应用中，人们希望验证两个（或者多个）物体之间至少存在一定的距离。Eberly [404]和 Larsen 等人[979]推导了这些类型边界体的方程和高效算法。

22.13.1 球体/球体相交

对于两个球体而言，其相交测试是简单而快速的：计算两个球心之间的距离，如果这个距离大于两个球体的半径之和，则拒绝相交；否则，它们会发生相交。在实现这个算法的时候，最好使用这两个距离的平方，因为我们只需要比较之后的布尔结果即可。这样我们可以避免了计算平方根（一个昂贵的操作）。Ericson [435]给出了同时测试四对球体的 SSE 代码。

22.13.2 球体/Box 相交

Arvo [70]首先提出了一种检测球体与 AABB 是否相交的算法，该算法非常简单。其想法是找到 AABB 上最接近球体中心 $\mathbf{c}$ 的点。使用三组一维测试（AABB 的三个轴各一个），根据 AABB 的边界来对一个轴上的球心坐标进行测试。如果球体位于这个边界之外，则计算球心与沿着这个轴到 box 的距离（一个减法），并将其平方。在我们沿着三个轴都进行了这个计算之后，将这些距离的平方和与球体半径的平方 r^2 进行比较。如果这个和小于半径的平方，则说明距离最近的点位于球体内部，即球体与 box 是相互重叠的。Arvo 指出，这个算法可以进行一些修改，来处理空心 box 与球体以及轴对齐椭球体的相交测试。

Larsson 等人[982]提出了该算法的一些变体，包括一个速度更快的 SSE 向量化版本。他们的见解是在早期使用简单的拒绝测试，要么是逐轴进行、要么是在开始的时候全部进行。这个拒绝测试是：检查球心到 box 的距离是否大于半径。如果是的话，那么这个测试可以提前结束，因为此时球体不可能与 box 发生重叠。当二者重叠的概率较低时，这种早期拒绝方法的速度明显要更快。下面是他们测试的快速拒绝（quick rejections intertwined，QRI）版本。提前退出的测试出现在第 4 行和第 7 行，如果需要的话，可以将它们删除。

```
     bool SphereAABB_intersect(c, r, A)
     returns({OVERLAP, DISJOINT});
1 :  d = 0
2 :  for each i ∈ {x, y, z}
3 :     if ((e = c_i − a_i^min) < 0)
4 :        if (e < −r)return (DISJOINT);
5 :        d = d + e^2;
6 :     else if ((e = c_i − a_i^max) > 0)
7 :        if (e > r)return (DISJOINT);
8 :        d = d + e^2;
9 :  if (d > r^2) return (DISJOINT);
10 : return (OVERLAP);
```

对于快速向量化（使用 SSE）的实现，Larsson 等人建议消除大部分的分支语句。其想法是使用下面的表达式，来同时计算第 3 行和第 6 行：

$$e = \max\left(a_i^{\min} - c_i, 0\right) + \max\left(c_i - a_i^{\max}, 0\right) \tag{22.25}$$

待摘要内容过短待摘要内容过短本文介绍了碰撞检测（collision detection，CD）在计算机图形学中的应用和实现方法。碰撞检测分为碰撞检测、碰撞确定和碰撞响应三个部分，通过计算物体之间的相交来判断是否发生碰撞。文章提到了在大型场景中进行碰撞检测的三个阶段：宽阶段 CD、中阶段 CD 和窄阶

```
     bool SphereAABB_intersect(c, r, A)
     returns({OVERLAP, DISJOINT});
1 :  e = (max(a_x^min − c_x, 0), max(a_y^min − c_y, 0), max(a_z^min − c_z, 0))
2 :  e = e + (max(c_x − a_x^max, 0), max(c_y − a_y^max, 0), max(c_z − a_z^max, 0))
3 :  d = e · e
4 :  if (d > r^2) return (DISJOINT);
5 :  return (OVERLAP);
```

请注意，第 1 行和第 2 行可以使用一个并行 SSE max 函数来实现。尽管在这个测试中没有早期退出，但是它仍然要比其他技术更快。这是因为在这个测试中消除了分支，并且使用了并行计算。SSE 的另一种方法是对物体对进行向量化。Ericson [435]提出的 SIMD 代码，能够同时比较四个球体和四个 AABB。

对于球体与 OBB 的相交测试，首先将球体中心变换到 OBB 的空间中。也就是说，使用 OBB 的归一化轴来作为变换球体中心的基底。现在这个中心点是相对于 OBB 的轴来进行表示的，因此这个 OBB 此时可以被视为一个 AABB。然后再使用球体与 AABB 的算法来进行相交测试即可。

Larsson [983]给出了一种高效的椭球体与 OBB 的相交测试方法。首先，将两个物体进行缩放，使得这个椭球体变成一个球体，OBB 变成一个平行六面体。使用球体与 slab 的相交测试可以进行快速接受和拒绝。最后，将球体与那些面向它的平行四边形进行相交测试。

22.13.3 AABB–AABB 相交

顾名思义，AABB 是一个与主轴方向对齐的 box。因此，使用两个顶点就足以描述这样的体积。这里我们使用章节 22.2 中所介绍的 AABB 定义。

由于 AABB 十分简单，因此它通常被用于碰撞检测算法和场景图节点的包围体。两个 AABB（ A 和 B ）之间的相交测试很简单，其伪代码如下所示：

```
     bool AABB_intersect(A, B)
     returns({OVERLAP, DISJOINT});
1 :  for each i ∈ {x, y, z}
2 :     if(a_i^min > b_i^max or b_i^min > a_i^max)
3 :        return (DISJOINT);
4 :  return (OVERLAP);
```

22.13.4 k-DOP/k-DOP 相交

一个 k-DOP 与另一个 k-DOP 的相交测试，只包含 $k/2$ 个区间的重叠测试。Klosowski 等人[910]表明，对于中等大小的 k 值，两个 k -DOP 的重叠测试要比两个 OBB 的重叠测试快一个数量级。在图 22.4 中，我们给出了一个简单的二维 k-DOP。请注意，AABB 是一个 6-DOP 的特殊情况，其法线刚好是正负主轴的方向。OBB 也是 6-DOP 的一种形式，但是这种快速测试只能在两个待测 OBB 共享相同轴的时候使用。

下面的相交测试简单且快速、不精确但很保守。如果要测试两个 k-DOP A 和 B （上标分别为 A 和 B ）是否相交，那么就需要测试所有平行的 slab 对 (S_i^A, S_i^B) 是否发生重叠； $s_i = S_i^A \cap S_i^B$ 是一个一维的区间重叠测试，它很容易求解。这是一个降维的例子，正如章节 22.5 中的经验法则所推荐的那样。这里我们将一个三维的 slab 测试简化为一个一维的区间重叠测试。

如果在任何时候都有 $s_i = \emptyset$ （即空集），那么这两个 BV 不相交，终止测试；否则将会继续进行 slab 的重叠测试。当且仅当所有的 $s_i \neq \emptyset$ ，$1 \leq i \leq k/2$ ，我们才认为两个 BV 重叠。根据分离轴测试（章节 22.2），我们还需要对一组轴进行测试，这组轴与每个 k-DOP 和一条边的叉乘平行。然而，这些测试通常会被忽略，因为它们的开销要大于在性能上的回报。因此，如果下面的测试伪代码返回的是 k-DOP 相互重叠，那么它们实际上可能是不相交的。下面是 k-DOP 与 k-DOP 重叠测试的伪代码：

```
    kDOP_intersect(d_1^{A,min}, ..., d_{k/2}^{A,min}, d_1^{A,max}, ..., d_{k/2}^{A,max},
                   d_1^{B,min}, ..., d_{k/2}^{B,min}, d_1^{B,max}, ..., d_{k/2}^{B,max})
    returns({OVERLAP, DISJOINT});
1:  for each i ∈ {1, ..., k/2}
2:      if(d_i^{B,min} > d_i^{A,max} or d_i^{A,min} > d_i^{B,max})
3:          return (DISJOINT);
4:  return (OVERLAP);
```

请注意，每个 k-DOP 实例只需要存储 k 个标量值（法线 $\mathbf{n}_i$ 对于所有 k-DOP 实例都只会存储一次，因为它们是静态的）。如果 k-DOP 分别被 $\mathbf{t}^A$ 和 $\mathbf{t}^B$ 进行了平移变换，那么这个测试就会变得稍微复杂一些。我们需要将 $\mathbf{t}^A$ 投影到法线 $\mathbf{n}_i$ 上，例如：$p_i^A = \mathbf{t}^A \cdot \mathbf{n}_i$ （请注意，这与任何的 k-DOP 都无关，因此对于每个 $\mathbf{t}^A$ 和 $\mathbf{t}^B$ 只需

要进行计算一次即可），并在 if 语句中将 p_i^A 加到 $d_i^{A,\min}$ 和 $d_i^{A,\max}$ 中。对于 $\mathbf{t}^B$ 也是如此，换句话说，平移改变了 k-DOP 沿每个法线方向的距离。

Laine 和 Karras [965]提出了一种 k-DOP 的扩展，它被称为顶点贴图（apex point map）。其想法是将一组平面法线映射到 k-DOP 的各个点上，这样每个存储点就代表了该方向上的最远位置。这个点和方向构成了一个平面，该平面在一个半空间中完全包含了该模型，即该点位于模型 k-DOP 的顶点（apex）。在测试过程中，在给定方向上检索到的顶点可以用于 k-DOP 之间更加精确的相交测试，用于对视锥体剔除进行改进、以及在旋转后寻找更加紧密的 AABB 等。

22.13.5 OBB/OBB 交集

在本小节中，我们简要介绍了一种快速方法来测试两个 OBB（ A 和 B ）之间的交集 [436, 576, 577]。该算法使用分离轴测试，这要比之前使用最近特征或者线性规划的方法快一个数量级。有关 OBB 的定义详见章节 22.2。

这个测试是在由 OBB A 的中心和轴所组成的坐标系中进行的。这意味着原点位于 $\mathbf{a}^c = (0,0,0)$，这个坐标系中的主轴分别是 $\mathbf{a}^u = (1,0,0)$， $\mathbf{a}^v = (0,1,0)$， $\mathbf{a}^w = (0,0,1)$。此外，我们假设 B 的位置相对于 A，具有一个平移变换 $\mathbf{t}$ 和一个旋转（矩阵） R。

根据分离轴测试，找到一个分离轴就足以确保 A 和 B 不相交（不重叠）。这里有 15 个轴需要进行测试：3 个来自 A 的面，3 个来自 B 的面，以及 $3 \times 3 = 9$ 个来自 A 和 B 边界的组合。图 22.23 以二维展示了这种情况。

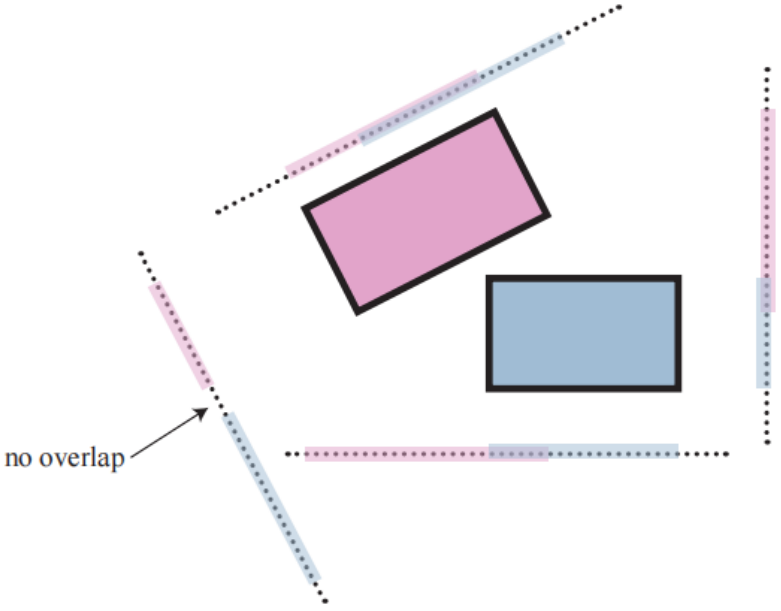


图 22.23：为了确定两个 OBB 是否重叠，可以使用分离轴试验。这里我们以二维形式进行展示。图中的四个分离轴与两个 OBB 的面正交，每个 box 都有两个这样的轴。然后，OBB 被投影到这些轴上。如果两个投影在所有轴上都发生重叠，那么则代表这两个 OBB 重叠；否则，则代表这两个 OBB 不重叠。因此，找到一个能够分离投影的轴，就足以知道这两个 OBB 不重叠了。而在这个例子中，左下方的轴是唯一能够分离投影的轴。[436]

由于矩阵 $\mathbf{A} = \begin{pmatrix} \mathbf{a}^u & \mathbf{a}^v & \mathbf{a}^w \end{pmatrix}$ 的正交性，可能与 A 面正交的潜在分离实际上就是轴 $\mathbf{a}^u$ ， $\mathbf{a}^v$ ， $\mathbf{a}^w$ ；B 面也是同样的。剩下 9 个潜在的轴，分别由 A 的一个边和 B 的一个边构成，即 $\mathbf{c}^{ij} = \mathbf{a}^i \times \mathbf{b}^j$ ，其中 $\forall i \in \{u, v, w\}$ ， $\forall j \in \{u, v, w\}$ 。幸运的是，网上有针对这一点的优化代码[1574]。

22.14 视锥体相交测试

正如我们在章节 19.4 中所看到的，分层视锥体剔除对于快速渲染复杂场景而言至关重要。在层次包围体剔除遍历期间，调用的少数几个操作之一就是视锥体和包围体之间的相交测试。因此，这些操作的效率对于快速执行剔除来说至关重要。在理想情况下，他们应当能够确定 BV 是完全位于视锥体内部（包含），还是完全位于视锥体外部（排除），或者 BV 与视锥体相交。

回顾一下，视锥体是一个金字塔，它被一个近裁剪平面和一个远裁剪平面（二者相互平行）所截断，从而使得体积变得有限。事实上，视锥体是一个多面体，在图 22.24 中给出了这 6 个平面的名称，分别是近（near）、远（far）、左（left）、右（right）、上（top）、下（bottom）。视锥体空间定义了场景中可见的部分，位于视锥体内部的物体才会被渲染（在一个透视视锥体中）。

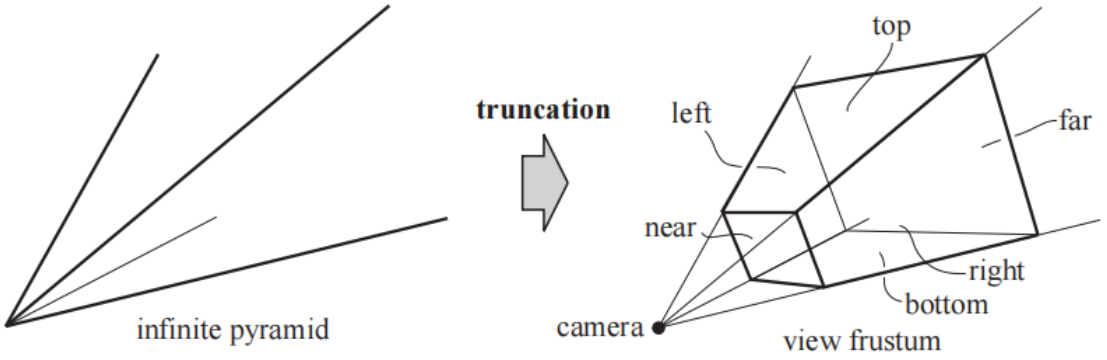


图 22.24：左边是一个无限延伸的金字塔，这个金字塔被相互平行的近裁剪平面和远裁剪平面裁剪成一个视锥体。右侧还展示了各个平面的名称，相机的位置在金字塔的顶端。

用于层次结构（例如场景图）中的内部节点和封闭几何物体上最常见的包围体就是球体、AABB、OBB。因此，我们将在这里讨论和推导视锥体与球体，以及视锥体与 AABB 和 OBB 的相交测试。

为了了解为什么我们需要外部、内部、相交这三种返回结果，我们将回顾遍历层次包围体时会发生什么。如果我们发现一个 BV 完全位于视锥体之外，那么这个 BV 的子树将不会被进一步遍历，并且其中的任何几何形状都不会进行渲染。另一方面，如果 BV 完全位于视锥体内部，则不需要为该子树计算更多的视锥体/BV 测试，并且后续将将对每个可渲染的叶子节点进行绘制。而对于部分可见的 BV（即那些与视锥体相交的 BV），则会根据视锥体，对该 BV 的子树进行递归测试。如果该 BV 是一个叶子节点，那么就必须渲染这个叶子节点，从而保证画面的准确性。

完整的测试被称为排除/包含/相交测试（exclusion/inclusion/intersection test）。有时，第三种状态（相交）的计算成本会被认为过高，在这种情况下，BV 会被分类为“可能位于内部”。我们将这种简化的算法称为排除/包含测试（exclusion/inclusion test）。如果一个 BV 无法被成功排除，那么后续有两种选择：一种是将“可能在内部”的状态视为包含，这意味着这个 BV 内部的所有内容都会被渲染。这样做通常是比较低效的，因为我们没有再次进行剔除。另一种选择是依次对该子树中的每个节点进行测试，从而进行精确剔除。但是这样的测试通常并没有什么好处，因为该子树的大部分内容可能确实位于视锥体内部。由于这两种选择都不是特别好，因此尝试快速区分相交情况和包含情况通常是值得的，即使这个测试并不那么完美。

这里有一个重要的认识，即快速分类测试不必是精确的场景图剔除，只需保守即可。为了从包含情况中区分出排除情况，所需要做的就是在包含一侧来对错误进行检测。也就是说，应当被排除在外的物体可能会被错误地包含在内，而这样的错误会浪费额外的时间。另一方面，应当包含在内的物体永远不应该被快速分类为排除在外，否则将会发生渲染错误。而对于包含和相交这两种情况，任何一种错误分类通常都是合法的。如果一个完全包含的 BV 被分类为相交，那么对其子树进行测试就会浪费一些时间。如果将一个相交的 BV 完全认为位于内部，那么渲染所有的物体也会造成一些时间浪费，因为其中一些物体可能已经被剔除了。

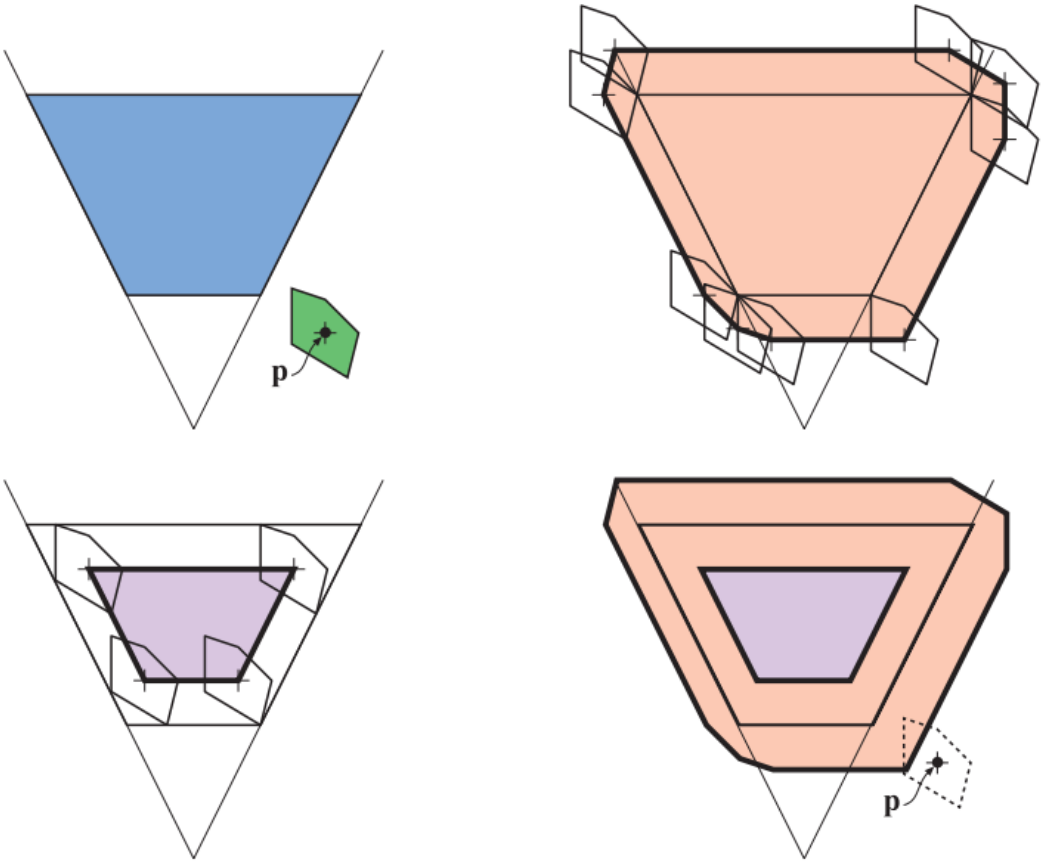


图 22.25：左上角展示了一个视锥体（蓝色）和一个一般的包围体（绿色），其中选择了一个相对于物体的点 $\mathbf{p}$ 。通过追踪物体在视锥体的外部（右上）和视锥体内部（左下）移动时的点 $\mathbf{p}$，尽可能地靠近视锥体，可以将视锥体与 BV 的相交测试，重新定义为测试点 $\mathbf{p}$ 与外部体积和内部体积之间的测试，如右下角所示。如果点 $\mathbf{p}$ 位于橙色体积外部，则说明该 BV 也位于视锥体外部；如果点 $\mathbf{p}$ 位于橙色区域内，则说明该 BV 与视锥体相交；如果点 $\mathbf{p}$ 位于紫色区域内，则说明该 BV 完全位于视锥体内部。

在我们介绍视锥体与球体、AABB 或者 OBB 之间的相交测试之前，我们将首先描述视锥体与一般物体之间的相交测试方法，这个测试如图 22.25 所示。这个想法是将一个 BV 与视锥体的测试，转变为一个点与体积的测试。首先，选择一个相对于 BV 的点。然后，将 BV 沿着视锥体的外部进行移动，并且尽可能靠近视锥体但不与其重叠。在这个移动过程中，相对于 BV 的点会被追踪，它的轨迹会形成一个新的体积（图 22.25 中带有粗边的多边形）。BV 尽可能靠近视锥体的事实意味着，如果这个相对于 BV 的点（在其原始位置）位于所描出的体积内部，则说明 BV 与视锥体相交，或者位于视锥体内部。因此，我们并不是对这个 BV 与视锥体进行相交测试，而是针对一个相对于 BV 的点与另一个新体积进行相交测试，这个新体积是根据该点追踪得来的。以同样的方式，BV 可以沿着视锥体的内部进行移动，并且尽可能地靠近视锥体。这将绘制出一个新的、更小的视锥体，并且其平面与原始视锥体的平面相平

行[83]。如果相对于物体的点位于这个新体积内部，那么就说明这个 BV 完全位于视锥体内部。在接下来的若干小节中，我们将使用这种技术来进行测试。请注意，新体积的创建与实际 BV 的位置无关，它只依赖于点相对于 BV 的位置和 BV 的形状。这意味着任意位置的 BV 可以针对相同的体积进行测试。

只保存父 BV 与每个子 BV 的相交状态是一个有用的优化。如果已知父节点完全位于视锥体内部，那么子孙节点就不需要进一步的视锥体测试了。章节 19.4 中所讨论的平面遮挡（plane masking）和时间一致性（temporal coherence）技术也可以显著改善针对一个层次包围体的相交测试，尽管它们在 SIMD 实现中不太有用[529]。

首先，我们会推导出视锥体的平面方程，因为这类测试中需要使用到这些方程。然后接着介绍了视锥体与球体的相交测试，之后再介绍视锥体与 box 的相交测试。

22.14.1 提取视锥体平面

为了进行视锥体剔除，我们需要得到视锥体六个不同侧面的平面方程。我们在这里给出一种巧妙且快速的方法来推导这些。假设观察矩阵为 $\mathbf{V}$ 、投影矩阵为 $\mathbf{P}$ ，组合后的复合变换为 $\mathbf{M} = \mathbf{PV}$ 。当 $\mathbf{t} = \mathbf{Ms}$ 时，点 $\mathbf{s}$ （其中 $s_w = 1$ ）被变换为 $\mathbf{t}$ 。此时，由于透视投影的原因， $\mathbf{t}$ 可能会出现 $t_w \neq 1$ 的情况。因此，将 $\mathbf{t}$ 中的所有分量除以 t_w ，得到点 $\mathbf{u}$ ，其中 $u_w = 1$ 。对于视锥体内部的点，当 $i \in x, y, z$ 时，满足 $-1 \leq u_i \leq 1$ ，即点 $\mathbf{u}$ 位于一个标准立方体内部。这里针对的是 OpenGL 类型的投影矩阵（章节 4.7）。对于 DirectX 而言，其中的 $0 \leq u_z \leq 1$ ，其他都相同。视锥体的各个平面可以直接由这个复合变换矩阵的行向量推导得出。

我们将注意力集中在标准立方体左平面右侧的体积上，其中 $-1 \leq u_x$ 。我们可以对其进行如下扩展：

$$\begin{aligned} -1 \leq u_x \iff -1 \leq \tfrac{t_x}{t_w} \iff t_x + t_w \geq 0 \iff \\ \iff (\mathbf{m}_0, \mathbf{s}) + (\mathbf{m}_3, \mathbf{s}) \geq 0 \iff (\mathbf{m}_0, +\mathbf{m}_3) \cdot \mathbf{s} \geq 0 \end{aligned} \tag{22.26}$$

在上面的这个推导过程中， $\mathbf{m}_i$ 表示复合变换矩阵 $\mathbf{M}$ 中的第 i 行。最后一步 $(\mathbf{m}_0, +\mathbf{m}_3) \cdot \mathbf{s} \geq 0$ 实际上代表了视锥体左平面的（一半）平面方程。这是因为标准立方体中的左平面已经被转换回了世界坐标。还要注意确保 $s_w = 1$ ，从而使得方程变成一个平面。为了使得这个平面的法线从视锥体内部指向视锥体外部，这个方程还必须被取反（原始方程描述的是标准立方体内部）。因此对于视锥体的左平面而言，其平面方程为 $-(\mathbf{m}_{3,} + \mathbf{m}_{0,}) \cdot (x, y, z, 1) = 0$ ，这里我们使用 $(x, y, z, 1)$ 来

代替使用平面方程的形式： $ax + by + cz + d = 0$ 。总的来说，视锥体的各个平面方程如下：

$$\begin{array}{ll} -(\mathbf{m}_{3,} + \mathbf{m}_{0,}) \cdot (x, y, z, 1) = 0 & [\,\text{left}\,], \\ -(\mathbf{m}_{3,} - \mathbf{m}_{0,}) \cdot (x, y, z, 1) = 0 & [\,\text{right}\,], \\ -(\mathbf{m}_{3,} + \mathbf{m}_{1,}) \cdot (x, y, z, 1) = 0 & [\,\text{bottom}\,], \\ -(\mathbf{m}_{3,} - \mathbf{m}_{1,}) \cdot (x, y, z, 1) = 0 & [\,\text{top}\,], \\ -(\mathbf{m}_{3,} + \mathbf{m}_{2,}) \cdot (x, y, z, 1) = 0 & [\,\text{near}\,], \\ -(\mathbf{m}_{3,} - \mathbf{m}_{2,}) \cdot (x, y, z, 1) = 0 & [\,\text{far}\,]. \end{array} \tag{22.27}$$

在网上可以找到使用 OpenGL 和 DirectX 实现这个操作的代码[600]。

22.14.2 视锥体/球体相交

正交投影的视锥体是一个 box，因此在这种情况下，视锥体与球体的相交测试变成了一个 OBB 与球体的相交测试，可以使用章节 22.13.2 中给出的算法来进行解决。为了进一步测试球体是否完全位于 box 内部，我们会首先检查球体中心沿着每个轴与 box 边界之间的距离是否大于它的半径。如果它在所有三个维度上都满足这个条件，那么这个球体就是被完全包含在 box 内部的。有关这个修改算法的高效实现以及相关代码，请参阅 Arvo 的文章[70]。

按照视锥体与 BV 相交测试的推导方法，对于任意的一个视锥体，我们选择使用球心来作为追踪点 $\mathbf{p}$ ，结果如图 22.26 所示。如果半径为 r 的球体沿着视锥体的内部和外部进行移动，并尽可能地靠近视锥体，那么点 $\mathbf{p}$ 的轨迹便给出了我们重新制定视锥体与球体相交测试所需要的体积。实际的体积如图 22.26 的中间部分所示。与之前一样，如果点 $\mathbf{p}$ 在橙色区域之外，那么说明球体位于视锥体外部。如果点 $\mathbf{p}$ 在紫色区域内，那么说明球体完全位于视锥体内部。如果点 $\mathbf{p}$ 在橙色区域内，则说明球体与视锥体平面相交。通过这种方法，我们可以进行精确的相交测试。不过为了提高效率，我们会使用图 22.26 右侧的近似值。其中橙色区域的体积被扩展了，从而避免圆角所带来的复杂计算。请注意，这个外部体积由沿着视锥体平面的法线方向，向外移动 r 个单位距离的视锥体平面所组成；而内部体积可以通过沿着视锥体的平面法线方向，向内移动 r 个单位距离的视锥体平面所组成。

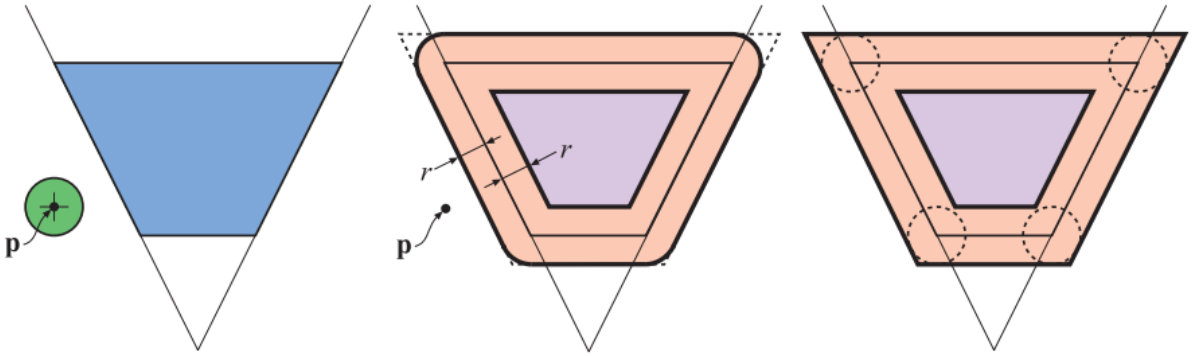


图 22.26：左边是一个视锥体和一个球体。精确的视锥体/球体测试，可以表示为将点 $\mathbf{p}$ 与中间图中的橙色区域和紫色区域进行测试。右边是中间体积的合理近似，如果一个球体的中心位于圆角之外，但是又位于所有外部平面之内，那么这个球体将会被错误地分类为与视锥体相交，即使实际上它位于视锥体外部。

我们假设视锥体的平面方程是这样的，其中平面的正半空间位于视锥体外部。然后，实际的代码实现将会在视锥体的六个平面上进行循环，对于每个视锥体平面，都会计算从球心到该平面的带符号距离，这是通过将球心坐标带入到平面方程中来实现的。如果这个距离大于半径 r，则说明球体位于视锥体外部；如果到所有六个平面的距离都小于 $-r$，则说明球体位于视锥体内部；否则球体与视锥体相交。更准确地说，我们称这个球体与视锥体相交，但是球体的中心也可能会位于圆角之外的尖角区域中（如图 22.26 所示）。这实际上意味着球体位于视锥体之外，但是我们保守地认为这个球体是相交的。

为了使得测试更加准确，可以添加额外的平面来测试这个球体是否位于视锥体外部。然而，为了快速剔除场景图中的节点，偶尔发生的错误命中只会导致我们进行一些不必要的测试，而不是算法的失败，但是这些额外测试将会花费更多的时间。另一种更加精确的方法将在章节 20.3 中进行介绍，该方法适用于这些尖角区域非常显著的情况。

对于一些高效的着色方法，视锥体通常是高度不对称的，图 20.7 中就描述了一种特殊的方法。Assarsson 和 Moller [83]提供了一种方法，该方法通过将视锥体划分为八个象限（octant），并找到物体的中心位于哪个分区中，从而在每次测试中消除三个平面。

22.14.3 视锥体/box 相交

对于正交投影（即视锥体是一个 box），可以使用 OBB 与 OBB 的相交测试来进行精确测试（章节 22.13.5）。而对于一般的视锥体与 box 的相交测试，常用的方法有两

种。一种简单的方法是使用视锥体的观察矩阵和投影矩阵，将所有八个 box 的顶点都转换到视锥体的坐标系中。对沿着每个轴扩展 $[-1,1]$ 范围的标准观察体进行裁剪测试（章节 4.7.1）。如果所有顶点都位于这个边界之外，则拒绝这个 box；如果所有顶点都位于边界内部，则这个 box 被完全包含在视锥体内部[529]。由于这个方法对裁剪过程进行了模拟，因此可以将其用于由任意一组点（例如线段、三角形或者 k-DOP）所分隔的任何物体。该方法的优点在于不需要提取视锥体的平面方程。其独立的（self-contained）简单性，使其可以在计算着色器中进行高效使用[1883, 1884]。

一种在 CPU 上相当高效的方法，是使用章节 22.10 中所描述的平面与 box 的相交测试。与视锥体和球体的相交测试一样，OBB 或者 AABB 是针对六个视锥体平面进行逐个检查的。而在平面与 box 的相交测试中，我们最多只会检查两个对角顶点（具体使用哪两个对角顶点由平面的法线决定），而不是计算所有八个顶点到平面的带符号距离。如果最近的顶点位于平面外侧，那么说明这个 box 完全位于视锥体外部，相交测试可以提前结束。如果每个平面的最远顶点位于平面内侧，那么说明这个 box 被包含在视锥体内部。请注意，远近平面点积距离的计算结果是可以共用的，因为这两个平面是平行的。第二种方法的唯一额外开销，就是必须首先推导出视锥体的平面方程，如果要对几个 box 进行测试，那这实际上是微不足道的开销。

与视锥体/球体的相交测试算法一样，这个测试也无法将实际上完全位于视锥体外部的 box 分类为相交，这些类型的错误如图 22.27 所示。Quilez 指出[1452]，对于固定大小的地形网格或者其他大型物体，这种情况会更加频繁地发生。当报告了一个相交时，他的解决方案是在形成包围框的各个平面上，再次对视锥体的各个角进行测试。如果所有角都在 box 的平面之外，那么视锥体与这个 box 不相交。这个附加测试相当于分离轴测试的第二部分，其中被测试的轴与第二个物体的表面正交。尽管如此，这种额外检测的成本可能会高于所带来的好处。对于他的 GIS 渲染器而言，Eng [425]发现这种优化每帧将会花费 2 毫秒的 CPU 时间，但是只能节省几个 draw call。

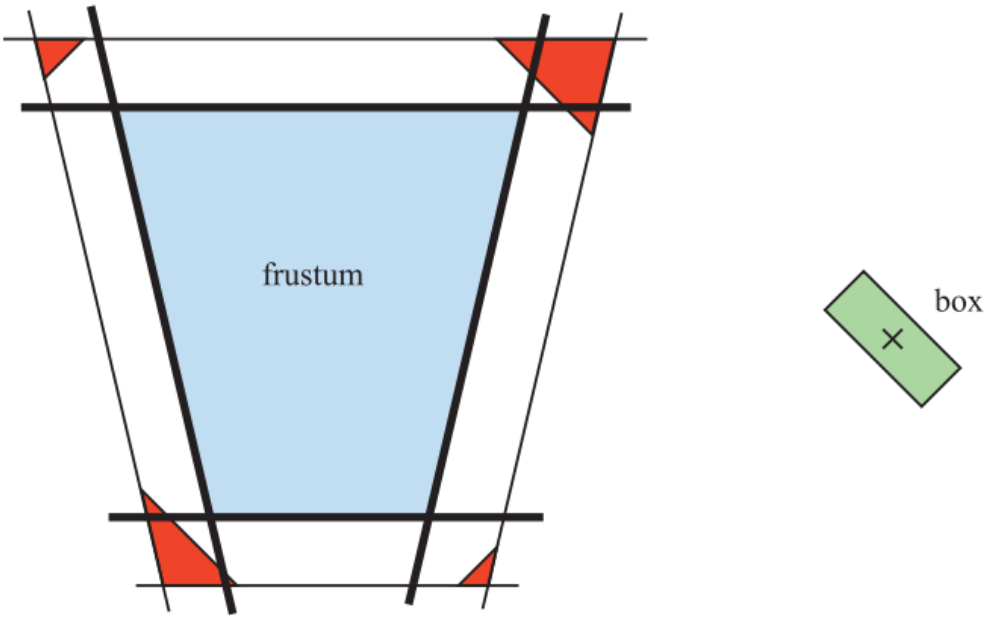


图 22.27：图中的粗黑线条代表了视锥体的平面。当使用所介绍的算法对 box（左）和视锥体进行相交测试时，如果这个 box 位于视锥体外部，它可能会被错误地分类为相交。而对于图中所示的情况，当这个 box 的中心点位于红色区域时，就会发生这种错误分类情况。

Wihlidal [1884]采用了另一个方向上的视锥体剔除，该方法只使用四个视锥体侧面进行剔除测试，而不进行近裁剪平面和远裁剪平面的剔除测试。他指出，这两个平面在电子游戏中的帮助不大。其中近裁剪平面大多数情况都是冗余的，因为侧边平面裁剪了几乎所有的空间；而远裁剪平面则通常用于查看场景中的所有物体。

另一种方法是使用分离轴测试（章节 22.13）来派生出一个相交例程。一些作者使用分离轴测试来求解两个凸多面体之间的通解[595, 1574]。然后，一个单独的优化测试可以用于线段、三角形、AABB、OBB、k-DOP、视锥体和凸多面体的任何组合。

22.15 线/线相交

在这一小节中，我们推导并了二维和三维的线/线相交测试。直线、射线和线段彼此之间都可以发生相交，这里我们介绍了一些既快速又优雅的方法。

22.15.1 二维

方法 1

从理论的角度来看，这种计算一对二维直线交点的方法确实很漂亮。假设现在有两条直线，分别是 $\mathbf{r}_1(s) = \mathbf{o}_1 + s\mathbf{d}_1$ 和 $\mathbf{r}_2(t) = \mathbf{o}_2 + t\mathbf{d}_2$ 。由于 $\mathbf{a} \cdot \mathbf{a}^{\perp} = 0$ （章节

1.2.1 中的垂直点积[735]），使得 $\mathbf{r}_1(s)$ 和 $\mathbf{r}_1(t)$ 之间的交点计算变得简洁而优雅。请注意，在这一节中所有向量都是二维的：

$$
\begin{aligned}
1: &\quad \mathbf{r}_1(s) = \mathbf{r}_2(t) \\
&\iff \\
2: &\quad \mathbf{o}_1 + s\mathbf{d}_1 = \mathbf{o}_2 + t\mathbf{d}_2 \\
&\iff \\
3: &\begin{cases} s\mathbf{d}_1 \cdot \mathbf{d}_2^{\perp} = (\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^{\perp} \\ t\mathbf{d}_2 \cdot \mathbf{d}_1^{\perp} = (\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^{\perp} \end{cases} \\
&\iff \\
4: &\begin{cases} s = \dfrac{(\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^{\perp}}{\mathbf{d}_1 \cdot \mathbf{d}_2^{\perp}} \\ t = \dfrac{(\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^{\perp}}{\mathbf{d}_2 \cdot \mathbf{d}_1^{\perp}} \end{cases}
\end{aligned}
\tag{22.28}
$$

如果 $\mathbf{d}_1 \cdot \mathbf{d}_2^{\perp} = 0$，那么这两条直线是平行的，即没有交点。这对于无限长的直线而言，所有 s 值和 t 值都是有效的；但是对于长度为 l_1 和 l_2 的线段而言（从 $s = 0$ 和 $t = 0$ 开始，到 $s = l_1$ 和 $t = l_2$ 结束），当且仅当 $0 \leq s \leq l_1$ 和 $0 \leq t \leq l_2$ 时，才会有一个有效的交点。或者，如果我们设 $\mathbf{o}_1 = \mathbf{p}_1$ 与 $\mathbf{d}_1 = \mathbf{p}_2 - \mathbf{p}_1$（这意味着线段从点 $\mathbf{p}_1$ 开始，到点 $\mathbf{p}_2$ 结束），对 $\mathbf{r}_2$ 也这样做，起点和终点分别是 $\mathbf{q}_1$ 和 $\mathbf{q}_2$，那么当且仅当 $0 \leq s \leq 1$，$0 \leq t \leq 1$ 时才会存在有效的交点。对于从原点出发的射线而言，这个有效范围是 $s \geq 0$ 和 $t \geq 0$。这些有效交点可以通过将 s 代入 $\mathbf{r}_1$ 中，或者将 t 代入 $\mathbf{r}_2$ 中得到。

方法 2

Antonio [61]描述了另一种判断两条线段（即长度有限的直线）是否相交的方法，该方法通过进行更多的比较和提前拒绝，并且避免了方法一中的昂贵计算（除法）。因此，这种方法的效率要更高。我们再次使用前面的符号表示，即第一个线段从点 $\mathbf{p}_1$ 到点 $\mathbf{p}_2$，第二个线段从点 $\mathbf{q}_1$ 到点 $\mathbf{q}_2$。这意味着 $\mathbf{r}_1(s) = \mathbf{p}_1 + s(\mathbf{p}_2 - \mathbf{p}_1)$，$\mathbf{r}_2(t) = \mathbf{q}_1 + t(\mathbf{q}_2 - \mathbf{q}_1)$。利用方程 22.28 的结果，我们可以得到 $\mathbf{r}_1(s) = \mathbf{r}_2(t)$ 的解：

$$
\begin{cases} s = \dfrac{-\mathbf{c} \cdot \mathbf{a}^{\perp}}{\mathbf{b} \cdot \mathbf{a}^{\perp}} = \dfrac{\mathbf{c} \cdot \mathbf{a}^{\perp}}{\mathbf{a} \cdot \mathbf{b}^{\perp}} = \dfrac{d}{f}, \\ t = \dfrac{\mathbf{c} \cdot \mathbf{b}^{\perp}}{\mathbf{a} \cdot \mathbf{b}^{\perp}} = \dfrac{e}{f}. \end{cases}
\tag{22.29}
$$

其中：

$$\begin{aligned}\mathbf{a} &= \mathbf{q}_2 - \mathbf{q}_1 \\ \mathbf{b} &= \mathbf{p}_2 - \mathbf{p}_1 \\ \mathbf{c} &= \mathbf{p}_1 - \mathbf{q}_1 \\ d &= \mathbf{c} \cdot \mathbf{a}^{\perp} \\ e &= \mathbf{c} \cdot \mathbf{b}^{\perp} \\ f &= \mathbf{a} \cdot \mathbf{b}^{\perp}\end{aligned}$$

方程 22.29 中的简化步骤，实际上来自于：

$$\begin{aligned}\mathbf{a}^{\perp} \cdot \mathbf{b} &= -\mathbf{b}^{\perp} \cdot \mathbf{a} \\ \mathbf{a} \cdot \mathbf{b}^{\perp} &= \mathbf{b}^{\perp} \cdot \mathbf{a}\end{aligned}$$

如果 $\mathbf{a} \cdot \mathbf{b}^{\perp} = 0$，那么这两条线是共线了。Antonio [61]注意到， s 和 t 的分母实际上是相同的，并且由于 s 和 t 不是明确需要的，因此我们可以省略这个除法运算。定义 $s = d/f$ 和 $t = e/f$ 。为了测试是否满足 $0 \leq s \leq 1$，可以使用以下代码：

```
1 : if(f > 0)
2 :     if(d < 0 or d > f) return NO_INTERSECTION;
3 : else
4 :     if(d > 0 or d < f) return NO_INTERSECTION;
```

经过测试，可以确保 $0 \leq s \leq 1$ 。然后再对 $t = e/f$ 执行相同的操作（在测试代码中使用 e 来替换 d ）。如果这个例程在测试之后没有返回值，则说明这两个线段确实相交，因为此时 s 值和 t 值都是是有效的。

这个例程的整数版本的源代码可以在网上找到[61]，并且可以很容易地转换为浮点数版本。

22.15.2 三维

假设我们想在三维空间中计算两条直线的交点（仍然由射线方程进行定义，即方程 22.1）。我们将这两条直线继续称为 $\mathbf{r}_1(s) = \mathbf{o}_1 + s\mathbf{d}_1$ 和 $\mathbf{r}_2(t) = \mathbf{o}_2 + t\mathbf{d}_2$，其中 t 的值没有限制。垂直点积的三维对应形式是叉乘，即 $\mathbf{a} \times \mathbf{a} = 0$，因此三维版本的推导过程与二维版本十分相似。两条直线的交点可以按照如下形式进行计算：

$$
\begin{aligned}
&1: \mathbf{r}_1(s) = \mathbf{r}_2(t) \\
&\Longleftrightarrow \\
&2: \mathbf{o}_1 + s\mathbf{d}_1 = \mathbf{o}_2 + t\mathbf{d}_2 \\
&\Longleftrightarrow \\
&3: \begin{cases} s\mathbf{d}_1 \times \mathbf{d}_2 = (\mathbf{o}_2 - \mathbf{o}_1) \times \mathbf{d}_2 \\ t\mathbf{d}_2 \times \mathbf{d}_1 = (\mathbf{o}_1 - \mathbf{o}_2) \times \mathbf{d}_1 \end{cases} \\
&\Longleftrightarrow \\
&4: \begin{cases} s\left(\mathbf{d}_1 \times \mathbf{d}_2\right) \cdot \left(\mathbf{d}_1 \times \mathbf{d}_2\right) = \left(\left(\mathbf{o}_2 - \mathbf{o}_1\right) \times \mathbf{d}_2\right) \cdot \left(\mathbf{d}_1 \times \mathbf{d}_2\right) \\ t\left(\mathbf{d}_2 \times \mathbf{d}_1\right) \cdot \left(\mathbf{d}_2 \times \mathbf{d}_1\right) = \left(\left(\mathbf{o}_1 - \mathbf{o}_2\right) \times \mathbf{d}_1\right) \cdot \left(\mathbf{d}_2 \times \mathbf{d}_1\right) \end{cases} \\
&\Longleftrightarrow \\
&5: \begin{cases} s = \dfrac{\det\left(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{d}_2, \mathbf{d}_1 \times \mathbf{d}_2\right)}{\left\|\mathbf{d}_1 \times \mathbf{d}_2\right\|^2} \\ t = \dfrac{\det\left(\mathbf{o}_2 - \mathbf{o}_1, \mathbf{d}_1, \mathbf{d}_1 \times \mathbf{d}_2\right)}{\left\|\mathbf{d}_1 \times \mathbf{d}_2\right\|^2} \end{cases}
\end{aligned}
\tag{22.30}
$$

其中步骤 3 是在两侧减去 $\mathbf{o}_1$ （或者 $\mathbf{o}_2$ ）并与 $\mathbf{d}_2$ （或者 $\mathbf{d}_1$ ）进行叉乘。步骤 4 是与 $\mathbf{d}_1 \times \mathbf{d}_2$ $(\mathbf{d}_2 \times \mathbf{d}_1)$ 进行点乘。最后，步骤 5 是最后的解，它是通过将右边重写为行列式（并改变底部方程中的一些符号），然后除以位于 s （或者 t）右侧的项来得到的。

Goldman [548]指出，如果分母中的 $\left\|\mathbf{d}_1 \times \mathbf{d}_2\right\|^2$ 等于 0，那么这两条直线实际上是平行的。他还观察到，如果这些直线是倾斜的（即它们位于两个平面上），那么参数 s 和 t 代表的是距离该直线最近的点。

如果将直线视为线段，长度分别为 l_1 和 l_2 （假设方向向量 $\mathbf{d}_1$ 和 $\mathbf{d}_2$ 是归一化的），则需要检查 $0 \leq s \leq l_1$ 和 $0 \leq t \leq l_2$ 是否成立。如果不成立的话，则拒绝相交。

Rhodes [1490]对两条直线或者两个线段相交的问题，给出了一个深入的解决方案。他给出了能够处理特殊情况的健壮方法，对相关优化进行了讨论，并提供了相应的源代码。

22.16 三平面相交

给定三个平面，每个平面都由一个归一化的法线 $\mathbf{n}_i$ 和平面上的任意一点 $\mathbf{p}_i, i = 1, 2, 3$ 进行描述，这些平面相交的唯一点 $\mathbf{p}$ 由方程 22.31 给出。请注意，如果其中两个或者多个平面平行，那么分母（即三个平面法线的行列式）为零：

$$\mathbf{p} = \frac{(\mathbf{p}_1 \cdot \mathbf{n}_1)(\mathbf{n}_2 \times \mathbf{n}_3) + (\mathbf{p}_2 \cdot \mathbf{n}_2)(\mathbf{n}_3 \times \mathbf{n}_1) + (\mathbf{p}_3 \cdot \mathbf{n}_3)(\mathbf{n}_1 \times \mathbf{n}_2)}{\begin{vmatrix} \mathbf{n}_1 & \mathbf{n}_2 & \mathbf{n}_3 \end{vmatrix}}. \tag{22.31}$$

这个方程可以用于计算由一组平面所组成的 BV 角点。k-DOP 就是其中一个例子，它由 k 个平面方程所组成。如果提供适当的平面方程，方程 22.31 还可以计算凸多面体的角点。

通常，如果一个平面以隐式形式给出，即 $\pi_i : \mathbf{n}_i \cdot \mathbf{x} + d_i = 0$，那么我们需要找到点 $\mathbf{p}_i$，以便能够使用方程 22.31。我们可以选择平面上的任意一点。这里我们计算距离原点最近的点，因为这些计算的开销通常会很低。给定一条从原点指向平面法线的射线，将其与平面相交，从而得到最靠近原点的平面上一点：

$$\left.\begin{aligned} \mathbf{r}_i(t) &= t\mathbf{n}_i \\ \mathbf{n}_i \cdot \mathbf{x} + d_i &= 0 \end{aligned}\right\} \Rightarrow$$
$$\mathbf{n}_i \cdot \mathbf{r}_i(t) + d_i = 0 \Longleftrightarrow t\mathbf{n}_i \cdot \mathbf{n}_i + d_i = 0 \Longleftrightarrow t = -d_i \tag{22.32}$$
$$\Rightarrow$$
$$\mathbf{p}_i = \mathbf{r}_i\left(-\overline{d_i}\right) = -d_i\mathbf{n}_i.$$

这个结果并不令人感到惊讶，因为平面方程中的 d_i 代表了垂直方向上，从原点到平面的负距离（如果这是正确的话，那么要求法线必须是归一化的）。

补充阅读和资源

Ericson 的《Real-Time Collision Detection》[435]和 Eberly 的《3D Game Engine Design》[404]涵盖了各种各样的物体与物体的相交测试，以及层次遍历方法和其他许多的方法，并提供了源代码。Schneider 和 Eberly 的《Geometric Tools for Computer Graphics》[1574]为二维和三维几何相交测试提供了许多实用的算法。开放获取《Journal of Computer Graphics Techniques》发表了有关相交测试的改进算法和代码。早期的《Practical Linear Algebra》[461]是一个良好的来源，其中包含了二维求交例程和计算机图形学中许多其他的几何操作。《Graphics Gems》系列书籍[72, 540, 695, 902, 1344]中包含了许多不同类型的相交测试例程，相关代码可以在网上获得。免费的 Maxima 软件[1148]可以很好地处理方程和推导方程。本书的网站包括一个页面（realtimerendering.com/intersections.html），其中对许多物体与物体之间相交测试可用的资源进行了总结。

Chapter 23 Graphics Hardware 图形硬件

> J. Allard——“When we get the final hardware, the performance is just going to skyrocket.”

> J· 阿拉德——“当我们得到最终的硬件时，性能将会直线上升。”（Xbox 之父，前 Xbox 首席执行官；1969—）

尽管图形硬件正在快速发展，但是在其设计中仍然有一些常用的通用概念和体系结构（architecture，架构）。本章节的目标是让读者理解图形系统中的各种硬件元素，以及它们之间的相互关系。而本书中的其他部分将讨论它们在特定算法中的应用。在这里，我们将介绍硬件本身。我们首先会介绍如何光栅化线条和三角形，随后介绍了 GPU 的大量计算能力是如何工作的，以及是如何安排任务的，包括处理延迟和占用等问题。然后我们将讨论内存系统、缓存、压缩、颜色缓冲以及 GPU 中与深度系统相关的所有内容。然后会介绍纹理系统中的细节，再然后是介绍 GPU 的架构类型。章节 23.10 中介绍了三种不同架构的案例研究，最后我们对光线追踪架构进行了简要讨论。

23.1 光栅化

绘制三角形和线条的速度是任何 GPU 的一个重要特征。如章节 2.4 中所述，光栅化阶段包括三角形设置和三角形遍历。此外，我们还将介绍如何在三角形上进行属性插值，这与三角形遍历密切相关。最后我们会介绍保守光栅化，这是标准光栅化的一种扩展。

回想一下，像素的中心由 $(x+0.5, y+0.5)$ 给出，其中 $x \in [0, W-1]$， $y \in [0, H-1]$。x 、 y 是整数， $W \times H$ 是屏幕分辨率（例如 3840×2160 ）。我们设未变换的顶点为 $\mathbf{v}_i, i \in 0, 1, 2$，变换后的顶点为 $\mathbf{q}_i = \mathbf{M}\mathbf{v}_i$ （包含投影操作，但是并不包含除以 w ）。二维屏幕空间中对应的坐标是 $\mathbf{p}_i = ((q_{ix}/q_{iw}+1)\,W/2, (q_{iy}/q_{iw}+1)\,H/2)$，即首先执行透视除法（除以 w 分

量），然后进行缩放和平移操作，从而与屏幕分辨率相匹配，如图 23.1 所示。我们可以看到，像素网格被划分为了 2×2 像素组，它被称为一个 quad（四边形）。

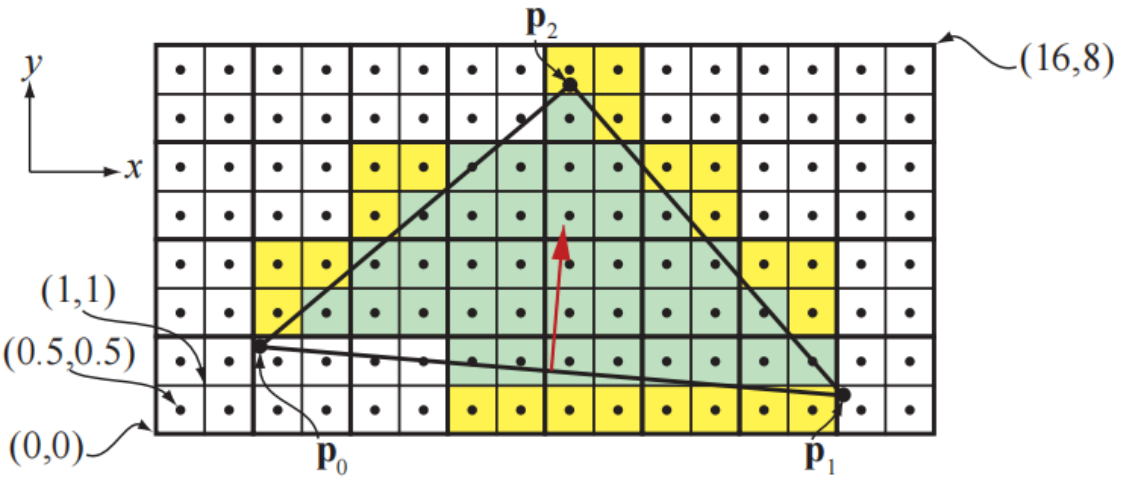


图 23.1：一个三角形在屏幕空间中有三个二维顶点 $\mathbf{p}_0$ 、 $\mathbf{p}_1$ 和 $\mathbf{p}_2$ 。屏幕大小为 16×8 像素。请注意，像素 (x, y) 的中心为 $(x + 0.5, y + 0.5)$ 。底部边缘的法线（按长度缩放 0.25）使用红色进行表示。只有绿色像素位于三角形内部。辅助像素使用黄色进行表示，它属于各个 quad（ 2×2 像素），每个 quad 中至少有一个像素被认为位于三角形内部，而辅助像素的样本点（中心）则位于三角形外部。使用有限差分（finite difference）计算导数时需要使用到这些辅助像素。

我们在确定纹理细节层级（章节 23.8）的时候，需要使用像素之间的梯度信息，为了能够计算这个梯度信息，只要这些 quad 中有一个像素位于三角形内部，就会对整个 quad 执行像素着色（在章节 3.8 中也讨论过）。这是大多数 GPU 的核心设计，这个设计影响了接下来的许多阶段。三角形的面积越小，辅助像素与三角形内部像素的比例就越大。这种关系意味着我们在执行像素着色的时候，小三角形所带来的开销会更大（与三角形面积成比例）。最糟糕的情况是这个三角形只覆盖了单个像素，这意味着它需要三个辅助像素。辅助像素的数量有时会被称为四边形过度渲染（quad overshading）。

为了确定像素中心或者任何其他的样本位置是否位于三角形内部，硬件会对每个三角形边缘使用边缘函数[1417]。这些边缘函数基于直线方程，即：

$$\mathbf{n} \cdot ((x, y) - \mathbf{p}) = 0 \tag{23.1}$$

其中 $\mathbf{n}$ 是一个向量，有时会称为边缘法线，它与一个三角形边缘正交（垂直）， $\mathbf{p}$ 是直线上的一个点。这样的方程可以重写为 $ax + by + c = 0$ 。接下来，我们将通过点 $\mathbf{p}_0$ 和点 $\mathbf{p}_1$ 推导出边缘函数 $e_2(x, y)$ 。边向量为 $\mathbf{p}_1 - \mathbf{p}_0$ ，因此其对应的法线

就是沿逆时针旋转 90 度的边缘，即 $\mathbf{n}_2 = \left(-\left(p_{1y} - p_{0y}\right), p_{1x} - p_{0x}\right)$，这个边缘法线指向三角形内部，如图 23.1 所示。将法线 $\mathbf{n}_2$ 和点 $\mathbf{p}_0$ 代入方程 23.1 中，可以得到边缘函数 $e_2(x, y)$ 为：

$$\begin{aligned} e_2(x, y) &= -\left(p_{1y} - p_{0y}\right)\left(x - p_{0x}\right) + \left(p_{1x} - p_{0x}\right)\left(y - p_{0y}\right) \\ &= -\left(p_{1y} - p_{0y}\right)x + \left(p_{1x} - p_{0x}\right)y + \left(p_{1y} - p_{0y}\right)p_{0x} - \left(p_{1x} - p_{0x}\right)p_{0y} \\ &= a_2 x + b_2 y + c_2 \end{aligned} \tag{23.2}$$

对于正好位于一条边上的点 (x, y)，我们有 $e(x, y) = 0$。由于法线指向三角形内部，这意味着与法线方向在同一边的点，都有 $e(x, y) > 0$。这个边缘将空间划分为了两个部分，其中 $e(x, y) > 0$ 有时会被称为正半空间，$e(x, y) < 0$ 被称为负半空间。利用这些属性，我们可以确定一个点是否位于三角形内部。我们将三角形的边缘称为 $e_i, i \in 0, 1, 2$。如果样本点 (x, y) 位于三角形内部或者恰好在边缘上，那么对于所有 i，它都必须保持 $e_i(x, y) \geq 0$。

图形 API 规范经常会要求将屏幕空间中的浮点顶点坐标转换为定点坐标（整数顶点）。这样做是为了以一致的方式来定义决胜局规则（tie-breaker rule，稍后会进行描述）。它还可以提高样本内部测试的效率，例如：p_{ix} 和 p_{iy} 都可以存储在 1.14.8 bit 中，即 1 个符号 bit、14 个整数坐标 bit 和 8 个像素内的小数位置 bit。在这个例子中，这意味着一个像素内的坐标 x 和 y 可以有 2^8 个可能的位置，即整数坐标必须在 $\left[-(2^{14} - 1)，2^{14} - 1\right]$ 的范围内。实际上，这种捕获（snapping）是在计算边缘方程之前完成的。

边缘函数的另一个重要特征是它们的增量特性。假设我们在某个像素中心 $(x, y) = (x_i + 0.5, y_i + 0.5)$ 处对边缘函数进行求值，其中 (x_i, y_i) 为整数的像素坐标，即我们要对 $e(x, y) = ax + by + c$ 进行求值，例如：要计算右边的像素，我们可以计算 $e(x + 1, y)$，可以将其重写为：

$$\begin{aligned} e(x + 1, y) &= a(x + 1) + by + c \\ &= a + ax + by + c \\ &= a + e(x, y) \end{aligned} \tag{23.3}$$

也就是说，这实际上就是在当前像素值 $e(x, y)$ 上，再加上 a。类似的推理还可以应用于 y 方向上，并且这些特性会经常被用来快速计算一小块像素（例如 8×8 像素）中的三个边缘方程，从而“stamp（译者注：字面意思是盖章）”出一个覆盖掩码，每

个像素使用一个 bit，来表示像素是否在内部。本小节稍后部分将会介绍这种层次遍历。

在这个过程中，考虑当边缘或者顶点恰好经过像素中心时会发生什么是十分重要的，例如：假设现在两个三角形共享一条边，并且这条边会经过一个像素中心。那么这个像素应当属于第一个三角形，还是第二个三角形，还是两者都属于呢？从效率的角度来看，这些都是错误答案，因为这个像素会首先被其中一个三角形写入，然后再被另一个三角形所覆盖。处于这个目的，我们通常会使用决胜局规则（tie-breaker rule），这里我们将展示 DirectX 中使用的左上规则（top-left rule）。其中 $e_i(x,y) > 0$，对于所有 $i \in 0,1,2$，$e_i(x,y) > 0$ 总是被认为位于三角形内部。当一条边穿过一个像素时，左上规则开始发挥作用。如果这个像素中心位于上边缘或者左边缘，那么则认为该像素位于这个三角形内部。如果一条边是水平的，而其他边都在这条边的下方，那么这条边就是上边缘。如果一条边不是水平的，并且位于三角形的左侧，那么它就是左边缘，这意味着三角形最多可以有两条左边缘。检测一条边是左边缘还是上边缘是很简单的。对于上边缘而言，边缘方程中的 $a = 0$（水平方向），$b < 0$；而对于左边缘而言，$a > 0$。判断样本点 (x,y) 是否位于三角形内部的整个测试，有时候会称为内部测试（inside test）。

到目前为止，我们还没有解释如何遍历一条直线。一般来说，一条线段可以被渲染为一个长方形，即宽度为一个像素的矩形，它可以由两个三角形组成，也可以对矩形使用额外的边缘方程。这种设计的优点在于，这些用于处理边缘方程的相同硬件，也可以用于处理直线。而一个单独的点则可以直接被绘制为一个 quad。

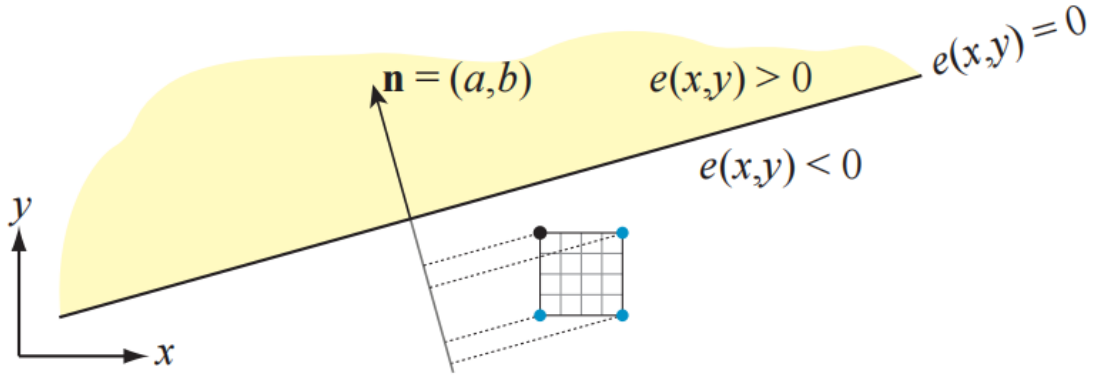


图 23.2：边缘函数的负半空间 $e(x,y) < 0$ 总是会被认为位于三角形之外。在这里，4×4 像素 tile 的角会被投影到边缘法线上。只有图中黑色圆圈标注的角需与这条边缘进行测试，因为这个角在法线 $\mathbf{n}$ 上的投影最大。然后我们可以得出结论，这个 tile 位于三角形外部。

为了提高效率，通常会采用分层方式来进行三角形遍历[1162]。通常，硬件会计算屏幕空间顶点的包围盒，然后确定哪些 tile 位于包围盒内部，并且与三角形重叠。这里

可以使用 AABB 与平面相交测试的二维版本，来确定 tile 是否位于边缘之外，这个过程的一般原理如图 23.2 所示。为了适应这个分块的三角形遍历，我们可以在遍历开始之前，首先确定边缘应当针对哪个 tile 角点来进行测试[24]。对于特定边缘的上所有 tile，所使用的 tile 角点都是相同的，因为最近的 tile 角点仅仅取决于边缘法线。针对这些预定的角点计算边缘方程，如果选定的角点位于这个边缘之外，则代表整个 tile 都位于三角形外部，这意味着硬件不需要在该 tile 内部执行任何逐像素的测试。想要移动到相邻的 tile 上，可以在每个 tile 上使用上面所描述的增量特性，例如：要向右水平移动 8 个像素，就需要加上 $8a$ 即可。

有了 tile 与边缘的相交测试，我们就可以对三角形进行分层遍历，结果如图 23.3 所示。这些 tile 也需要以某种顺序来进行遍历，这里可以使用之字形顺序（zigzag），或者使用一些空间填充曲线来完成[1159]，这两种方法都倾向于增加一致性。如果需要的话，还可以在层次遍历中添加其他级别，例如：可以先访问 16×16 的 tile，然后对重叠在三角形上的每个 tile 测试 4×4 的子 tile [1599]。

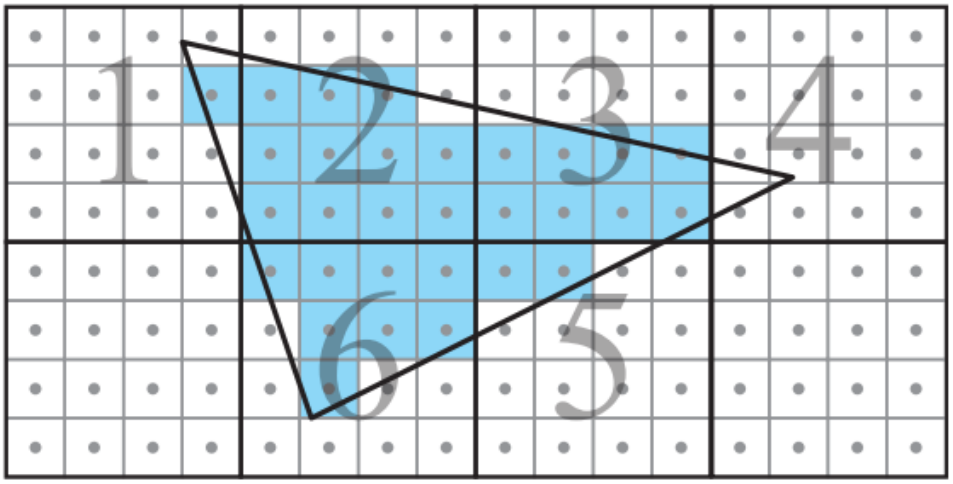


图 23.3：使用 4×4 像素 tile 时可能的遍历顺序。在这个例子中，遍历从左上角开始，并继续向右进行遍历。其中上面的每个 tile 都与三角形重叠，尽管右上角的 tile 中并没有生成任何像素。遍历会继续到 4 号 tile 下面的那个 tile，这个 tile 完全位于外部，因此不需要对内部的像素进行测试。然后继续向左遍历，依次发现两个 tile 与三角形重叠，而左下角的 tile 则没有发生重叠。

相对于以扫描线顺序来对三角形进行遍历，这种 tile 遍历的主要优势在于，它可以使得像素处理变得更加连贯（一致性），因此访问纹素的方式也更加连贯。在访问颜色缓冲和深度缓冲的时候，它还可以更好地利用这种局部性，例如：假设我们使用扫描线顺序遍历一个大三角形。我们会对纹素进行缓存处理，以便最近访问的纹素保留在缓存中以供重用。假设我们使用 mipmap 用于纹理化操作，这增加了缓存中纹素的重

用级别。如果我们按照扫描线顺序来访问像素，在到达当前扫描线末尾的时候，扫描线开头所使用的纹素很可能已经从缓存中移除了。由于在缓存中重复使用纹素，要比从内存中重复获取纹素更加高效，因此三角形通常会以 tile 的形式进行遍历[651, 1162]。这对纹理化[651]、深度缓冲[679]和颜色缓冲[1463]有很大的好处。事实上，纹理、深度缓冲和颜色缓冲也会被存储在 tile 中，原因与此相同。这将在章节 23.4 中进行进一步地讨论。

在三角形遍历开始之前，GPU 通常会有一个三角形设置阶段。这一阶段的目的是计算三角形上的常数，以便更加高效地进行遍历，例如：三角形的边缘方程（方程 23.2）中的常数 $a_i, b_i, c_i, i \in 0, 1, 2$ 会在这里进行一次计算，然后在当前三角形的整个遍历步骤中进行使用。三角形设置还负责计算与属性插值相关的常数（章节 23.1.1）。随着讨论的继续，我们还将会发现其他的一些常数，这些常数也可以在三角形设置中只计算一次。

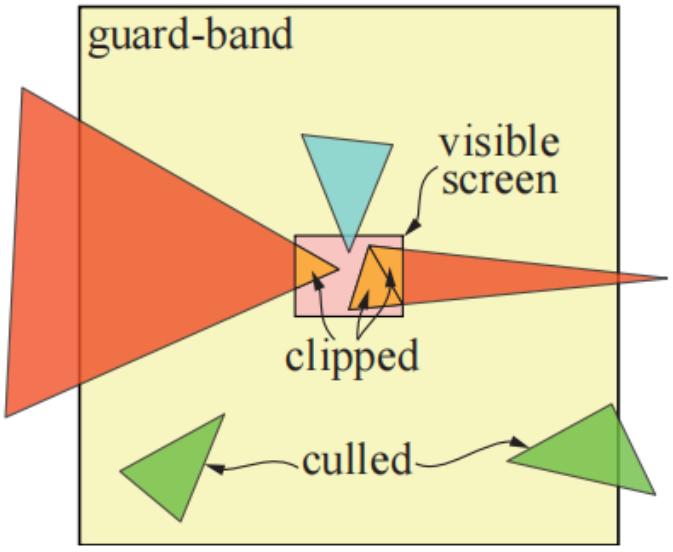


图 23.4：保护带（guard-band）会尽量避免完全裁剪。假设这个保护带区域在 x 和 y 方向上都是 $\pm 16K$ 像素，那么中间的屏幕大约是 6500×4900 像素，这表明这些三角形都是十分巨大的。底部的两个绿色三角形要么会在三角形设置中被剔除，要么会在前面的步骤中被剔除。最常见的情况是中间的蓝色三角形，它与屏幕区域相交，并完全位于保护带内。我们不需要对其进行完全的裁剪操作，因为只有可见的 tile 会被处理。红色三角形位于保护带之外，并且与屏幕区域相交，因此这些三角形需要进行裁剪。请注意，右边的红色三角形被裁剪成了两个三角形。

由于裁剪操作可能会产生更多的三角形，因此必须在三角形设置之前进行裁剪。在裁剪空间中根据观察体积对三角形进行裁剪是一个昂贵的过程，所以如果不是绝对必要的话，GPU 会避免这样做。但是对近裁剪平面的裁剪总是需要的，这会使得原来的一个三角形，生成一个或者两个新的三角形。对于屏幕边缘而言，大多数 GPU 都会

使用保护带裁剪（guard-band clipping），这是一种更加简单的方案，可以避免更加复杂的完整裁剪过程。该算法的可视化结果如图 23.4 所示。

23.1.1 插值

在章节 22.8.1 中，重心坐标是计算射线和三角形交点时的副产物。任何顶点属性 $a_i, i \in 0, 1, 2$，都可以用重心坐标 (u, v) 来进行插值：

$$a(u, v) = (1 - u - v)a_0 + ua_1 + va_2 \tag{23.4}$$

其中 $a(u, v)$ 是三角形 (u, v) 处的插值属性。重心坐标的数学定义是：

$$u = \frac{A_1}{A_0 + A_1 + A_2}, \quad v = \frac{A_2}{A_0 + A_1 + A_2} \tag{23.5}$$

其中 A_i 是图 23.5 左侧所示的子三角形面积。第三个坐标 $w = A_0/(A_0 + A_1 + A_2)$ 实际上也是定义中的一部分，它表明 $u + v + w = 1$，即 $w = 1 - u - v$。在这里我们使用 $1 - u - v$ 来代替 w。

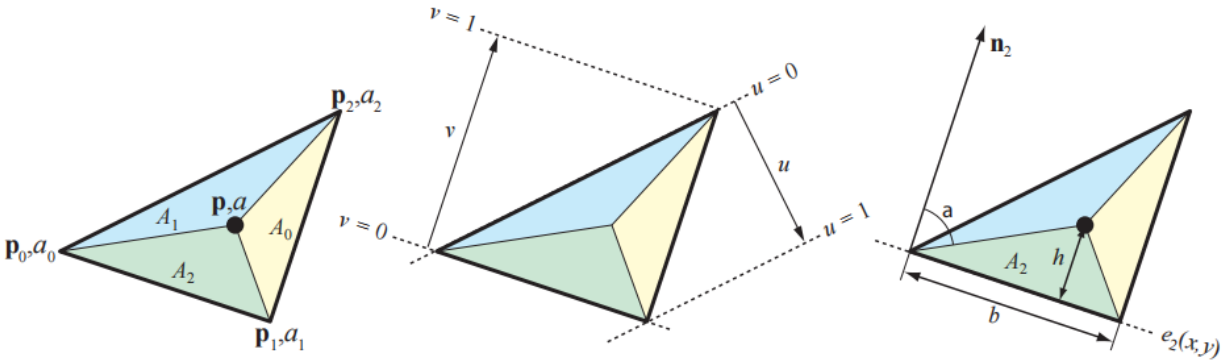


图 23.5：左侧：一个顶点上具有标量属性 (a_0, a_1, a_2) 的三角形。点 $\mathbf{p}$ 的重心坐标与带符号面积 (A_1, A_2, A_0) 成正比。中间：展示了重心坐标 (u, v) 如何在三角形上进行变化。右侧：法线 $\mathbf{n}_2$ 实际上就是 $\mathbf{p}^0\mathbf{p}^1$ 逆时针旋转 90 度后形成的。因此面积 A_2 为 $bh/2$。

方程 23.2 中的边缘方程，可以使用边缘法线 $\mathbf{n}_2 = (a_2, b_2)$ 来进行表示：

$$e_2(x, y) = e_2(\mathbf{p}) = \mathbf{n}_2 \cdot ((x, y) - \mathbf{p}_0) = \mathbf{n}_2 \cdot (\mathbf{p} - \mathbf{p}_0) \tag{23.6}$$

其中 $\mathbf{p} = (x, y)$。根据点积的定义，上述方程可以改写为：

$$e_2(\mathbf{p}) = \|\mathbf{n}_2\| \, \|\mathbf{p} - \mathbf{p}_0\| \cos\alpha \tag{23.7}$$

其中 α 是 $\mathbf{n}_2$ 和 $\mathbf{p}-\mathbf{p}_0$ 之间的夹角。请注意，$b=\|\mathbf{n}_2\|$ 实际上等于边缘 $\mathbf{p}^0\mathbf{p}^1$ 的长度，因为 $\mathbf{n}_2$ 就是这条边缘旋转 90 度所产生的。第二项 $\|\mathbf{p}-\mathbf{p}_0\|\cos\alpha$ 的几何解释是：将 $\mathbf{p}-\mathbf{p}_0$ 投影到 $\mathbf{n}_2$ 所得到向量的长度，这个长度正好是子三角形（面积为 A_2）的高度 h，如图 23.5 所示。值得注意的是，我们有 $e_2(\mathbf{p})=\|\mathbf{n}_2\|\,\|\mathbf{p}-\mathbf{p}_0\|\cos\alpha=bh=2A_2$，这是一个非常好的特性，因为我们需要子三角形的面积来计算重心坐标。这意味着：

$$\begin{aligned}(u(x,y),v(x,y)) &= \frac{(A_1,A_2)}{A_0+A_1+A_2}\\ &= \frac{(e_1(x,y),e_2(x,y))}{e_0(x,y)+e_1(x,y)+e_2(x,y)}\end{aligned} \tag{23.8}$$

在三角形设置阶段中，通常会提前计算 $1/A_0+A_1+A_2$，因为三角形的面积是不会发生变化的，这也避免了逐像素的除法操作。因此，当我们使用边缘方程来遍历一个三角形的时候，我们将会得到方程 23.8 中的所有项，这是内部测试的副产物。重心坐标在为正交投影插值深度的时候表现良好（之后我们将看到这一点）；但是对于透视投影而言，使用重心坐标对深度信息进行插值，并不会生成预期的结果，如图 23.6 所示。

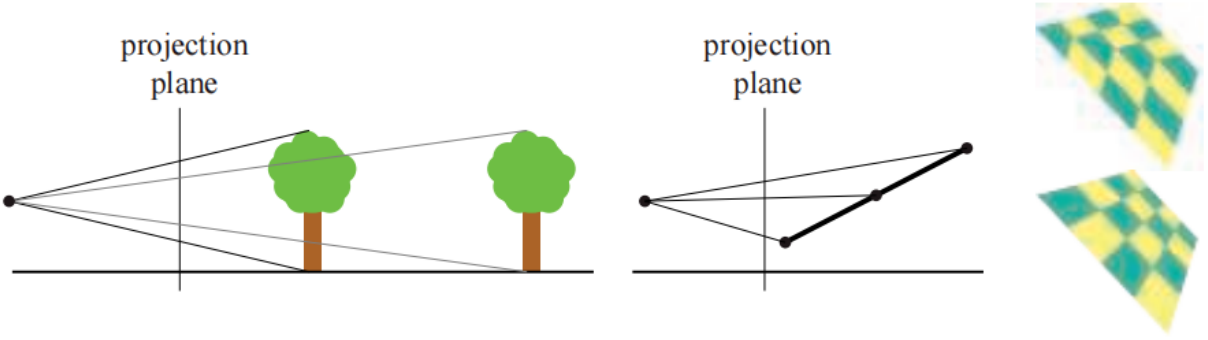


图 23.6：左侧：在透视投影中，几何物体的投影图像会随着距离的增大而缩小。中间：一个斜着的三角形投影。请注意，这个三角形的上半部分所覆盖的投影平面区域，要比下半部分所覆盖的更小。右侧：一个具有棋盘格纹理的四边形。上面的图像使用重心坐标进行纹理渲染，而下面的图像则使用透视正确的重心坐标进行纹理渲染，上面的图像会出现一些错误。

透视正确的重心坐标要求每个像素都进行除法操作[163, 694]。这里我们省略了推导过程[26, 1317]，直接总结最重要的结果。由于线性插值成本并不高，并且我们已经知道了如何来计算 (u,v)，所以我们希望尽可能地在屏幕空间中使用线性插值，即使是用于透视校正。令人惊讶的是，我们可以在三角形上对 a/w 和 $1/w$ 进行线性插

值，其中 w 是顶点在经过所有变换后的第四个分量。想要恢复被插值的属性 a，只需使用这两个插值后的值即可：

$$\frac{\overbrace{a/w}^{\text{linearly interpolated}}}{\underbrace{1/w}_{\text{linearly interpolated}}} = \frac{aw}{w} = a \tag{23.9}$$

这就是前面提到的逐像素除法。

下面通过一个具体的例子来说明它的效果。假设我们沿着一条水平的三角形边缘进行插值，最左边的属性是 $a_0 = 4$，最右边的属性是 $a_1 = 6$。那这两个端点中间的值是多少呢？对于正交投影而言（或者当端点的 w 值相匹配时），答案就是 $a = 5$，即 a_0 和 a_1 中间的值。

假设两侧端点的 w 值分别为 $w_0 = 1$ 和 $w_1 = 3$。在这种情况下，我们需要进行两次插值，从而获得 a/w 和 $1/w$。首先对于 a/w，左侧端点为 $4/1 = 4$，右侧端点为 $6/3 = 2$，因此中点的值为 3。而对于 $1/w$ 左侧端点和右侧端点的值分别是 $1/1$ 和 $1/3$，所以中点的值为 $2/3$。最后，将二者相除，即将 3 除以 2/3，得到透视投影下的中点值 $a = 4.5$。

在实践中，我们经常需要通过对三角形进行透视校正来插值出多个属性。因此，通常都要计算透视正确的重心坐标，我们将其记为 $(\tilde{u}, \tilde{v})$，然后将其用于所有的属性插值。为此，我们引入以下若干辅助函数[26]：

$$f_0(x,y) = \frac{e_0(x,y)}{w_0}, \quad f_1(x,y) = \frac{e_1(x,y)}{w_1}, \quad f_2(x,y) = \frac{e_2(x,y)}{w_2} \tag{23.10}$$

请注意，由于 $e_0(x,y) = a_0 x + b_0 y + c_0$，因此三角形设置可以计算和存储 a_0/w_0 和其他类似的项，从而使得逐像素的计算速度更快。或者，所有 f_i 函数都可以乘以 $w_0 w_1 w_2$，即我们存储 $w_1 w_2 f_0(x,y)$，$w_0 w_2 f_1(x,y)$ 和 $w_0 w_1 f_2(x,y)$ [1159]。因此，透视正确的重心坐标是：

$$(\tilde{u}(x,y), \tilde{v}(x,y)) = \frac{(f_1(x,y), f_2(x,y))}{f_0(x,y) + f_1(x,y) + f_2(x,y)} \tag{23.11}$$

每个像素都需要计算一次，然后使用透视正确的重心坐标来对任何顶点属性进行插值。请注意，这些坐标并不像 (u,v) 那样，与子三角形的面积成正比。此外，其中的

分母也不像重心坐标那样是恒定的，这就是必须逐像素进行除法的原因。

最后，请注意，由于像素的深度值是 z/w ，我们可以在方程 23.10 中看到，我们不应当使用这些方程，因为它们已经除以 w 了。因此，应当计算每个顶点上的 z_i/w_i ，然后再使用 (u, v) 进行线性插值。这有几个优点，例如：可以对深度缓冲进行压缩（章节 23.7）。

23.1.2 保守光栅化

从 DirectX 11 开始，或者通过使用 OpenGL 的扩展，可以使用一种被称为保守光栅化（conservative rasterization，CR）的新型三角形遍历方法。保守光栅化有两种类型，分别是高估 CR （overestimate CR，OCR）和低估 CR （underestimate CR，UCR）。有时也会被称为外保守光栅化（outer–conservative rasterization）和内保守光栅化（inner–conservative rasterization），如图 23.7 所示。

粗略地说，OCR 会访问所有与三角形重叠或者位于三角形内部的像素，而 UCR 只会访问完全位于三角形内部的像素。OCR 和 UCR 都可以通过 tile 遍历的方法进行实现，即将 tile 大小缩小为一个像素[24]。当硬件不支持的时候，也可以使用几何着色器或者三角形扩展[676]来实现 OCR。有关 CR 的更多信息，请参阅相应 API 的规范。CR 可以用于图像空间中的碰撞检测、遮挡剔除、阴影计算[1930]和抗锯齿等算法。

最后，我们注意到所有类型的光栅化，都是几何处理和像素处理之间的桥梁。为了计算三角形顶点的最终位置和计算像素的最终颜色，GPU 需要大量灵活的计算能力。我们将在下一小节中对此进行解释和介绍。

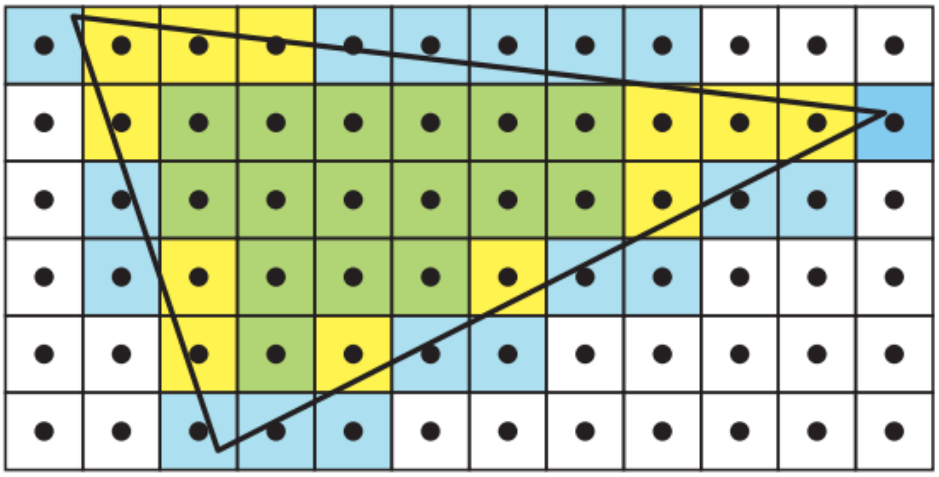


图 23.7：三角形的保守光栅化。当使用外保守光栅化 OCR 时，图中所有的彩色像素都属于这个三角形。其中在使用标准光栅化的时候，只有黄色像素和绿色像素会被认为位于三

角形内部；在使用内保守光栅化的时候，只有绿色像素会被认为位于三角形内部。

23.2 大规模计算和调度

为了提供可用于任意计算的大规模计算能力，大多数 GPU 架构都采用了统一的着色器架构，使用多线程的 SIMD 处理，有时被称为 SIMD 处理或者超线程（hyperthreading）。有关线程、SIMD 处理、warp 和线程组的术语，详见章节 3.10。请注意，这里我们使用了术语 warp，这是 NVIDIA 的术语；而在 AMD 硬件中，它们被称为 wave 或者 wavefront。在本小节中，我们将首先介绍 GPU 中使用的一种典型统一算术逻辑单元（arithmetic logic unit，ALU）。

ALU 是一个硬件单元，它针对单个实体（例如顶点或者片元）的执行程序进行优化。有时候我们会使用术语 SIMD 通道（lane）而不是 ALU。一个典型的 GPU ALU 如图 23.8 所示。其中主要的计算单元是一个浮点（FP）单元和一个整数单元。FP 单元通常遵循 IEEE 754 FP 标准，并支持 FMA（fused-multiply and add）指令，这是其最复杂的指令之一。除了超越操作（例如余弦、正弦和指数）之外，ALU 通常还会包含移动/比较、加载/存储等功能，以及一个分支单元。应当注意的是，在某些架构中，其中一些功能可能会位于独立的硬件单元中，例如：使用一小组专门执行超越操作的硬件单元，可以为大量的 ALU 服务。对于不是经常执行的操作而言，可能就会进行这样的处理。它们在特殊单元（special unit，SU）中被组合在一起，如图 23.8 中右侧所示。ALU 架构通常会使用几个硬件流水线阶段进行构建，也就是说，存在若干个在硅芯片中构建的、可以并行执行的实际硬件块，例如：在当前指令执行乘法的时候，下一条指令可以读取寄存器。在理想情况下，如果有 n 个流水线阶段，那么 ALU 的吞吐量可以增加 n 倍。这通常被称为流水线并行（pipeline parallelism）。使用流水线的另一个重要原因是，流水线处理器中最慢的那个硬件块，决定了这个块可以执行的最大时钟频率。增加流水线阶段的数量，可以降低每个流水线阶段的所使用的硬件块数量，通常这会使得增加时钟频率成为可能。然而，为了简化设计，ALU 通常只会有很少的流水线阶段，例如 4–10 个。

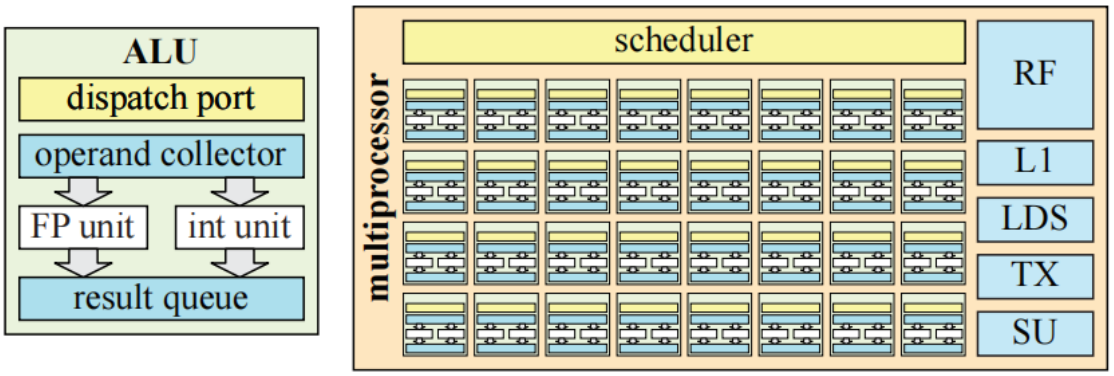


图 23.8：左侧：一个算术逻辑单元 ALU 的例子，每次只会执行一项。其中的调度端口（dispatch port）会接收当前需要执行的指令信息，运算对象收集器（operand collector）会读取指令所需的寄存器。右侧： 8×4 个 ALU 与其他几个硬件单元一起，被组装成一个称为多处理器（multiprocessor）的块。这 32 个 ALU 有时也会被称为 SIMD lane，它们会以锁步（lock-step）的方式来执行相同的程序，也就是说，它们构成了一个 SIMD 引擎。旁边还有一个寄存器堆（register file）、一个 L1 高速缓存、本地数据存储（local data storage）、纹理单元（texture unit），以及一个特殊单元（special unit），它用于实现 ALU 中未处理的各种指令。

统一 ALU 与 CPU 核心的不同之处在于，它没有很多花哨的东西，例如分支预测（branch prediction）、寄存器重命名（register renaming）和深度指令流水线（deep instruction pipelining）等。相反，芯片的大部分面积都花在了复制 ALU 上，从而提供巨量的计算能力；以及增加寄存器堆的大小，以便可以切换 warp。例如：NVIDIA GTX 1080 Ti 有 3584 个 ALU。为了能够高效调度分发给 GPU 的工作，大多数 GPU 会将 ALU 分为一组 32 个。同一组内的 ALU 是锁步执行的，这意味着整个 32 个 ALU 的集合是一个 SIMD 引擎。不同的厂商对这类硬件单元以及其他的一些硬件单元使用不同的名称，这里我们使用通用术语多处理器（multiprocessor，MP）。例如：NVIDIA 使用术语流多处理器（streaming multiprocessor，SM），英特尔使用执行单元（execution unit），而 AMD 则使用计算单元（compute unit，CU）。图 23.8 中展示了 MP 的一个例子。一个 MP 通常会有一个调度程序，来将工作分派给 SIMD 引擎，还有一个 L1 缓存、本地数据存储（local data storage，LDS）、纹理单元（TX），以及一个特殊单元，它负责处理 ALU 中无法执行的指令。MP 将指令分派到各个 ALU，在 ALU 中指令会以锁步的方式进行执行，即 SIMD 处理（章节 3.10）。请注意，一个 MP 中的实际内容，会因为供应商和不同代的架构而异。

SIMD 处理对于图形工作负载是有意义的，因为会有许多执行相同程序的顶点和片元。这里，该架构利用了线程级并行（thread-level parallelism），也就是说，顶点

和片元可以独立于其他的顶点和片元，独立执行它们的着色器程序。此外，对于任何类型的 SIMD/SIMT 处理，都可以利用数据级并行（data-level parallelism），因为一条指令会在 SIMD 机器中的所有 lane 上执行。还有指令级并行（instruction-level parallelism），这意味着如果处理器能够找到相互独立的指令，那么它们也可以同时执行，当然前提是有可以并行执行的资源。

与 MP 相近的是一个（warp）调度器，它会接收将要在 MP 上执行的大块工作。warp 调度器的任务是将 warp 中的工作分配给 MP，并将寄存器堆（RF）中的寄存器分配给 warp 中的线程，然后以最佳方式来确定工作的优先级。通常来说，下游工作要比上游工作具有更高的优先级，例如：像素着色位于可编程阶段的末尾，它具有比顶点着色（位于管线的早期阶段）更高的优先级。这样设计是避免了停滞，因为接近末尾的阶段不太可能阻塞较早的阶段。关于图形管线的回顾与复习，详见图 3.2。一个 MP 可以处理数百甚至数千个线程，从而隐藏内存访问等带来的延迟。调度器可以将 MP 上正在执行（或者等待）的 warp 切换到另一个准备执行的 warp。由于这个调度器是在专用硬件中进行实现的，因此这个切换操作通常是零开销的[1050]。例如：如果当前 warp 执行了一条纹理加载的指令，而这条指令预计会有很长的延迟，那么调度器可以立即切换出当前 warp，替换为另一个 warp，并继续执行这个已经准备好的 warp。通过这种方式，可以更好地利用计算单元，避免发生等待和停滞。

请注意，对于像素着色工作，warp 调度器会分配几个完整的四边形（quad），这是因为像素是在四边形粒度上进行着色的，从而允许计算导数。这一点我们在章节 23.1 节提到过，后续将在章节 23.8 中进一步讨论。因此，如果一个 warp 的大小为 32，那么 $32/4 = 8$，即有 8 个四边形可以被安排执行。这里有一个架构设计上的选择，我们可以选择将整个 warp 都锁定在一个三角形上；或者让 warp 中的每个四边形属于不同的三角形。前者实现起来比较简单，但是对于较小的三角形而言，效率会受到影响，因为这些较小的三角形可能凑不满 8 个四边形。后者要更为复杂，但是对于较小的三角形而言会更加高效。

一般来说，为了在芯片上获得更高的计算密度，MP 也会被复制，因此，GPU 通常也会具有更高级别的调度器。这个高级别调度器的任务是根据提交给 GPU 的工作，将这些工作分配给不同的 warp 调度器。在一个 warp 中包含多个线程，通常也意味着一个线程的工作需要与其他线程相独立。当然，这在图形处理中是经常出现的，例如：一个顶点的着色处理通常不会依赖于其他顶点，一个片元的颜色通常也不会依赖于其他片元。

请注意，不同架构之间存在许多差异。其中一些将在章节 23.10 中进行重点说明，我们将介绍几个不同的实际案例研究。到目前为止，我们知道了光栅化是如何完成的，

以及如何使用许多重复的统一 ALU 来计算着色。剩下一大块内容是内存系统、所有相关的缓冲区和纹理。从章节 23.4 开始，我们将讨论这些主题，但是首先我们要介绍一些有关延迟和占用率的更多信息。

23.3 延迟和占用率

一般来说，延迟是指从发起查询到接收结果之间的时间。举个例子：我们可能会请求内存中某个地址中的值，从发起查询到得到结果之间所花费的时间就是延迟。另一个例子是从纹理单元中请求过滤后的颜色，从发起请求到这个值可以使用，中间可能会需要数百甚至数千个时钟周期。为了高效利用 GPU 中的计算资源，我们需要隐藏这个延迟。如果不隐藏这些延迟的话，那么内存访问很容易会占据大部分执行时间。

其中一种隐藏延迟的机制，是 SIMD 处理的多线程部分，如图 3.1 所示。一般来说，一个 MP 可以处理的 warp 数量是有上限的。能够处于活跃状态的 warp 数量取决于寄存器的使用量，也可能会取决于纹理采样器、L1 缓存、插值和其他因素的使用量。这里，我们将占用率（occupancy） o 定义为：

$$o = \frac{w_{\text{active}}}{w_{\text{max}}} \tag{23.12}$$

其中 w_{max} 是一个 MP 上允许的最大 warp 数， w_{active} 是当前活跃的 warp 数。也就是说，占用率 o 是计算资源使用情况的指标，举个例子：假设 $w_{max} = 32$ ，一个着色处理器有 256 kB 大小的寄存器；而现在有一个单线程着色程序会使用 27 个 32 bit 浮点寄存器，另一个则会使用 150 个 32 bit 浮点寄存器。此外，我们假设此时寄存器的使用量决定了活跃 warp 的数量。假设这里的 SIMD 宽度为 32，我们可以分别计算这两种情况下的活跃 warp 数量：

$$w_{\text{active}} = \frac{256 \cdot 1024}{27 \cdot 4 \cdot 32} \approx 75.85, \quad w_{\text{active}} = \frac{256 \cdot 1024}{150 \cdot 4 \cdot 32} \approx 13.65 \tag{23.13}$$

在第一种情况下，即对于使用 27 个寄存器的较短程序而言， $w_{active} > 32$ ，因此其占用率为 $o = 1$ ，这是理想状态下的情况，它是隐藏延迟的好兆头。而在第二种情况下， $w_{\text{active}} \approx 13.65$ ，因此 $o \approx 13.65/32 \approx 0.43$ ，由于此时活跃的 warp 数量较少，因此相应的占用率也较低，这可能会阻碍延迟的隐藏。因此，设计一个较为均衡的最大 warp 数、最大寄存器和其他共享资源的架构十分重要。

有时候过高的占用率反而会适得其反，因为如果我们的着色器使用了过多的内存访问，它反而可能会破坏缓存[1914]。另一种延迟隐藏机制是在内存请求之后，继续执

行这个相同的 warp，如果存在独立于内存访问结果的指令的话，那么这就是可能的。虽然这样会使用更多的寄存器，但是有时候低占用率反而会更高效[1914]。一个例子就是循环展开（loop unrolling），它为实现指令级并行提供了更多的可能性，因为这样做通常会生成更长的独立指令链，这使得在切换 warp 之前，可以执行更长的时间。但是这也会使用更多的临时寄存器。通用的准则是争取更高的占有率，低占用率意味着当着色器在请求纹理访问时，不太可能切换到另一个 warp，因为根本没有更多的 warp 了。

另一种类型的延迟是从 GPU 回读数据到 CPU 中所带来的延迟。一个好的思维模型是，将 GPU 和 CPU 视为异步工作的独立计算机，二者之间的数据通信需要很大的开销。改变信息流向所带来的延迟会严重影响性能表现。当从 GPU 回读数据时，可能必须在读取之前刷新管线。在此期间，CPU 必须等待 GPU 完成其工作。对于诸如英特尔的 GEN 架构[844]，GPU 和 CPU 位于同一芯片上，并且使用共享内存模型，对于这类架构而言，这种类型的延迟会大大降低。低级缓存会在 CPU 和 GPU 之间进行共享，而高级缓存则不会进行共享。共享缓存所带来的延迟降低，允许使用不同类型的优化和其他类型的算法。例如：这个特性已经被用于对光线追踪进行加速，其中光线会在 GPU 和 CPU 核心之间来回进行传输，并且不会带来额外的开销[110]。

有一个不会导致 CPU 停滞的回读机制，就是遮挡查询（occlusion query），详见章节 19.7.1。对于遮挡测试而言，其机制是首先执行遮挡查询，然后偶尔对 GPU 进行检查，来看看查询的结果是否可用。在等待结果的同时，其他的一些工作可以继续在 CPU 和 GPU 上完成。

23.4 内存架构和总线

在本小节中，我们将介绍一些相关术语，讨论几种不同类型的内存架构，然后再介绍一下压缩和缓存。

端口（port）是指两个设备之间发送数据的通道，总线（bus）是指两个以上设备之间发送数据的共享通道。带宽（bandwidth）是一个用来描述端口或者总线上数据吞吐量的术语，单位是每秒字节数（B/s）。端口和总线在计算机图形体系结构中十分重要，因为简单来说，它们将不同的构建块粘合在了一起。同样重要的是，带宽是一种稀缺资源，因此在构建图形系统之前，必须进行仔细的设计和分析。由于端口和总线都提供数据传输功能，因此端口通常也被称为总线，这里我们将遵循这个约定，将二者统称为总线。

对于许多 GPU 而言，在图形加速器上拥有专属的 GPU 内存是很常见的，这种内存通常被称为显存（video memory）。相对于让 GPU 通过总线访问系统内存而言，访问这种内存的速度通常要快得多，例如：PC 中使用的 PCI Express （PCIe），16 通道 PCIe v3 的双向速率为 15.75 GB/s，PCIe v4 的双向速率为 31.51 GB/s。然而，Pascal 架构的显存（GTX 1080）可以提供 320 GB/s 的传输速度。

传统上，纹理和渲染目标都会存储在显存中，但是显存也可以用于存储其他数据。例如：场景中的许多物体在帧与帧之间并没有明显的形状变化。即使是人类角色，通常也会使用一组不变的网格来进行渲染，这些网格在关节处使用 GPU 侧的顶点混合。对于这种纯粹通过模型矩阵和顶点着色器程序来生成动画的数据，通常会使用静态的顶点缓冲区（vertex buffer）和索引缓冲区（index buffer），它们被存储在显存中，这样做可以让 GPU 进行快速访问。对于每帧由 CPU 进行更新的顶点数据，可以使用动态的顶点缓冲区和索引缓冲区，这些缓冲区放置在系统内存中，可以通过总线（例如 PCI Express）进行访问。PCIe 的一个优点是查询可以被流水线化，因此可以在结果返回之前请求多个查询。

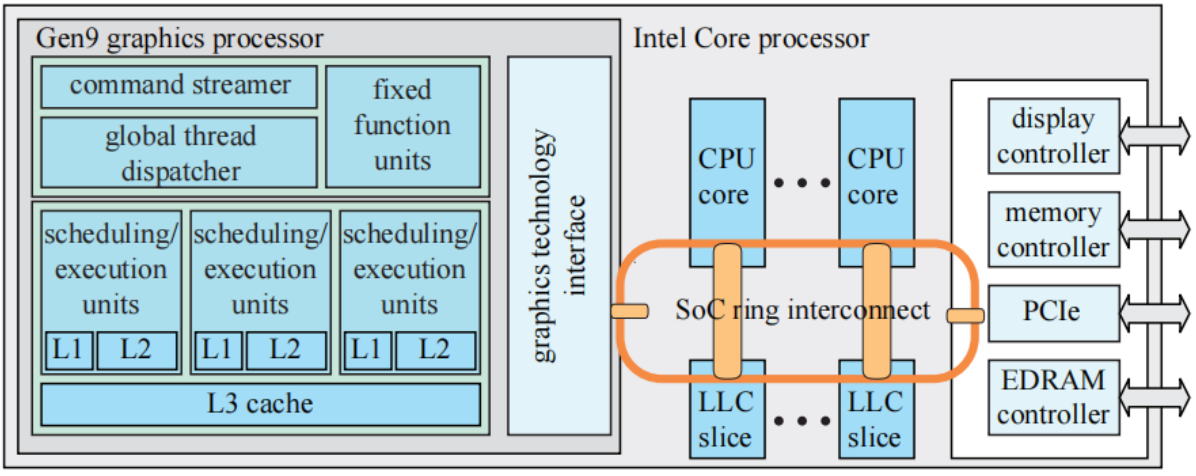


图 23.9：英特尔 SoC（system-on-a-chip，单芯片系统）上 Gen9 图形架构中的内存架构简化示意图，该架构与 CPU 核心和一个共享内存模型相连。请注意，最后一级缓存（LLC）在图形处理器和 CPU 内核之间进行共享。[844]

大多数游戏主机（例如所有的 Xbox 系列主机和 PLAYSTATION 4），都使用了统一内存架构（unified memory architecture，UMA），这意味着图形加速器可以使用主机系统内存中的任何部分，来处理纹理和不同类型的缓冲区[889]。CPU 和图形加速器都使用了相同的内存，因此也都使用相同的总线。这种内存架构与使用专用显存具有明显区别。Intel 也使用 UMA 架构，使得内存在 CPU 内核和 GEN9 图形架构之间进行共享[844]，如图 23.9 所示。但并非所有缓存都是共享的，图形处理器有着属于自己的一组 L1 高速缓存、L2 高速缓存和 L3 高速缓存。最后一级缓存（last-

level cache，LLC）是存储层次结构中的第一个共享资源。对于任何计算机或者图形架构而言，拥有一个缓存层次结构都是很重要的。如果访问中存在某种局部性，那么这样做可以减少对内存的平均访问时间。在下一小节中，我们将讨论 GPU 中的缓存和压缩。

23.5 缓存和压缩

缓存位于 GPU 的几个不同部分，而且会因架构而异，我们将在章节 23.10 中看到。通常来说，在架构中添加一个缓存层次结构的目的，是通过利用内存访问模式的局部性，来减少内存延迟和带宽使用。这里的内存访问局部性，是指如果 GPU 访问一个了内存中的一项数据，那它很有可能马上就会再次访问这个相同的项或者访问其附近的项[715]。大多数缓冲区和纹理格式都会以分块（tile）格式进行存储，这也有助于增加局部性[651]。假设一个缓存行（cache line）由 512 bit 组成，即 64 byte，而当前使用的颜色格式为每像素 4 B。一种设计选择是将所有像素都存储在一个 4×4 的区域中，即一个 tile 64 B，也就是说，整个颜色缓冲区将被分割为 4×4 的 tile。一个 tile 也可以跨越多个缓存行。

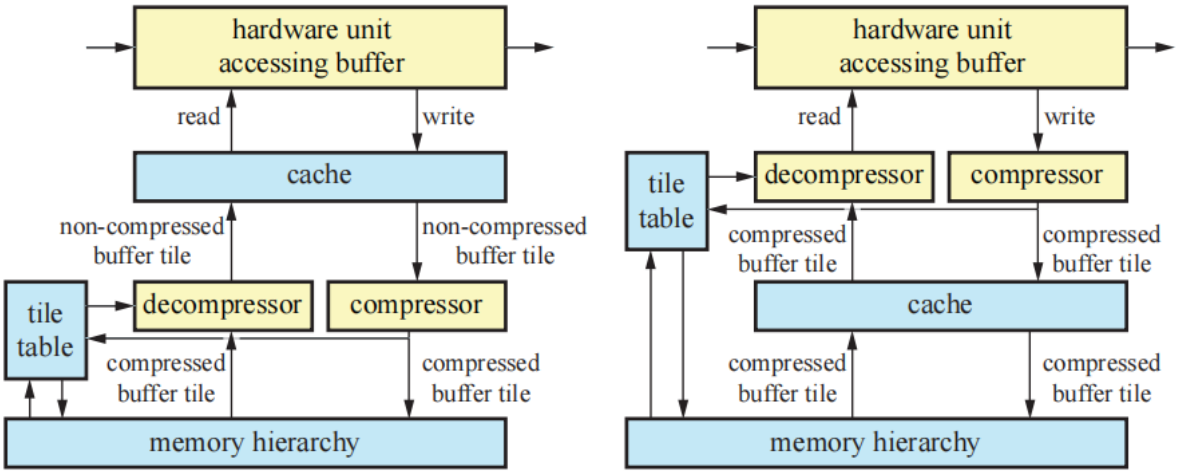


图 23.10：在 GPU 中压缩和缓存渲染目标的硬件技术示意图。左侧：缓存后再压缩，压缩/解压硬件单元位于缓存之后（图片下方）。右侧：预缓存压缩，压缩/解压硬件单元位于缓存之前（图片上方）。

为了获得高效率的 GPU 架构，需要在各个方面努力减少带宽使用。大多数 GPU 中都包含相应的硬件单元，可以进行对渲染目标进行动态压缩和解压缩，例如在渲染图像的时候。很重要的一点是，需要认识到这些类型的压缩算法都是无损的，也就是说，总是可以精确地恢复原始数据。这些算法的核心就是我们所说的分块表（tile table），它为每个 tile 存储了额外的信息。它们可以存储在芯片上，或者在内存层次

中通过缓存进行访问，这两类系统的结构如图 23.10 所示。一般来说，同样的设置可以用于深度缓冲、颜色缓冲和模板缓冲的压缩，有时候还需要进行一些修改。分块表中的每个元素会存储 tile 像素在 framebuffer 中的状态，每个 tile 的状态可以是已压缩（compressed）、未压缩（uncompressed）或者被清除（cleared）（接下来会进行讨论）。一般来说，也可以有不同类型的压缩块，例如：一种压缩模式可能会压缩到 25%，而另一种则可能会压缩到 50%。最重要的是要认识到，这个压缩级别取决于 GPU 可以处理的内存传输大小。假设在一种特定的架构中，最小的内存传输是 32 B。此时如果选择 64 B 的分块大小，那么就只能压缩到 50%。然而，对于 128 B 大小的贴图而言，则可以选择压缩到 75%（96 B），50%（64 B）和 25%（32 B）。

这个分块表也经常会用于实现渲染目标的快速清除。当系统发出清除渲染目标的指令时，表中每个 tile 的状态会被设置为已清除，而不会触及 framebuffer 本身。当访问渲染目标的硬件单元需要读取已清除的渲染目标时，解压器（decompressor）会首先检查分块表中的状态，来查看 tile 是否已经被清除。如果是，则将渲染目标对应的 tile 放置在缓存中，并将所有值都设置为 clear 值，而不需要对实际的渲染目标数据进行读取和解压缩。通过这种方式，在清除期间对渲染目标本身的访问可以被最小化，从而节省带宽。如果状态未被清除，则必须读取该 tile 所对应的渲染目标。存储在 tile 上的数据会被读取，如果其中的数据处于压缩状态的话，则会在发送出去之前通过解压缩器进行解压。

当访问渲染目标的硬件单元完成了新值的写入，并且最终从缓存中移除这个 tile 的时候，它会被发送到压缩器（compressor）中，并在那里尝试对其进行压缩。如果有两种压缩模式，那么两种模式都可以进行尝试，并且使用能够以最少 bit 对其该 tile 进行压缩的模式。由于 API 需要无损的渲染目标压缩结果，因此如果所有压缩技术都失败了，则需要退回到使用未压缩的数据。这也意味着渲染目标的无损压缩，永远无法减少实际渲染目标中的内存使用，这种技术只会减少内存带宽的开销。如果压缩成功了，那么该 tile 的状态会被设置为已压缩，并以压缩形式发送信息。否则会以未压缩形式进行发送，状态也会相应地设置为未压缩。

注意，压缩器单元和解压器单元可以位于缓存之后（称为 post-cache），也可以位于缓存之前（称为预缓存，pre-cache），如图 23.10 所示。预缓存压缩可以大幅增加有效缓存的大小，但是通常也会增加系统的复杂性[681]。有些特定的算法可以用于压缩深度[679, 1238, 1427]和颜色[1427, 1463, 1464, 1716]。后者还包括了对有损压缩的研究，然而，据我们所知，在任何硬件中都没有使用这种方法[1463]。大多数算法都编码了一个锚点值（anchor value），它代表了一个 tile 中的所有像素，然后

根据这个锚点值，以不同的方式对差异进行编码。对于深度数据而言，通常会存储一组平面方程[679]或者使用一种双重差分方法（difference-of-differences）[1238]，这两种方法都可以得到很好的结果，因为深度在屏幕空间中是线性的。

23.6 颜色缓冲

使用 GPU 进行渲染，涉及到对几个不同缓冲区的访问，例如颜色缓冲、深度缓冲、模板缓冲。请注意，虽然它被称为“颜色”缓冲区，但是实际上任何类型的数据都可以在其中进行渲染和存储。

根据表示颜色的字节数量，颜色缓冲区通常会有几种不同的颜色模式。这些模式包括：

- 高彩色（High color）：每像素 2 byte，其中有 15 bit 或者 16 bit 用于颜色，分别可以表示 32,768 和 65,536 种颜色。
- 真彩色（True color，RGB color）：每像素 3 byte 或者 4 byte，其中 24 bit 用于表示颜色，可以获得 $16777216 \approx 1680$ 万种不同的颜色。
- 深彩色（Deep color）：每像素 30 bit，36 bit 或者 48 bit，至少可以产生 10 亿种不同的颜色。

其中 high color 模式有 16 bit 的颜色分辨率可以使用。通常，这个数值会被划分为至少 5 个 bit，分别代表红色、绿色和蓝色，每个颜色通道种包含 32 个等级。这样还剩下 1 bit，通常会给绿色通道，这样就产生了一种 5-6-5 的划分方法。选择绿色通道是因为绿色对眼睛的亮度影响最大，即人眼对绿色最为敏感，因此需要更高的精度。high color 要比 true color 和 deep color 具有速度上的优势，这是因为访问每个像素 2 byte 的内存，通常要比访问每个像素 3 byte 或者更多的内存要快。但是，high color 模式的使用在这里实际上是非常罕见的，甚至根本不会使用 high color 模式。这是因为每个通道中只有 32 个或者 64 个级别的颜色，相邻颜色级别之间的差异可以很容易地分辨出来。这个问题有时被称为色带（banding）或者色调分离（posterization）。由于一种称为马赫带（Mach banding）[543, 653]的感知现象，人类的视觉系统会进一步放大了这种差异，如图 23.11 所示。抖动（dither）方法[102, 539, 1081]，即相邻的颜色水平被混合在一起，可以通过牺牲空间分辨率，来增加有效的颜色分辨率，从而减少这样效应的影响。但是，即使是在 24 bit 显示器上，这种渐变的带状效果也会很明显。向帧缓冲图像种添加噪声可以掩盖这个问题[1823]。

图 23.11：当矩形从白色渐变为黑色时，会出现带状颜色现象。尽管对于这 32 个灰度条，其中每个灰度的强度级别都是恒定的，但是由于马赫带错觉，每个灰度条看起来都是左边更暗，右边更亮。

true color 使用了 24 bit RGB 颜色（共 3 byte），每个颜色通道 1 byte。在 PC 系统上，存储顺序有时会被颠倒为 BGR。在内部，这些颜色通常会使用每像素 32 bit（共 4 byte）来进行存储，因为大多数内存系统都是针对访问 4 byte 元素进行优化的。在某些系统上，额外的 8 bit 也可以用来存储 alpha 通道，即赋予像素一个 RGBA 值。24 bit 颜色（无 alpha）表示也被称为打包像素格式（packed pixel format），与 32 bit 的未打包格式相比，它可以节省 framebuffer 的内存占用。对于实时渲染而言，使用 24 bit 颜色几乎总是可以接受的。虽然仍然有可能看到颜色条带，但是要比只有 16 bit 的情况少得多。

对于 RGB 颜色，deep color 可以使用 30 bit、36 bit 或者 48 bit 来进行表示，即每个通道使用 10 bit、12 bit 或者 16 bit。如果加上 alpha 通道，那么这些数字将增加到 40/48/64。HDMI 1.3 支持所有 30/36/48 模式，DisplayPort 标准也支持每通道最多 16 bit。

颜色缓冲通常都会被压缩和缓存，如章节 23.5 所述。此外，在章节 23.10 的每个案例研究中，都进一步描述了如何将传入的片元数据与颜色缓冲混合。这个混合由光栅操作（raster operation，ROP）单元进行处理，每个 ROP 通常会连接到一个内存分区，例如使用通用的棋盘格模式[1160]。接下来我们将对视频显示控制器进行讨论，它接收一个颜色缓冲区，并将其显示在显示器上。然后我们再介绍单缓冲、双缓冲和三缓冲。

23.6.1 视频显示控制器

在每个 GPU 中，都有一个视频显示控制器（video display controller，VDC），也被称为显示引擎（display engine）或者显示接口（display interface），它负责将一个彩色缓冲区输出到显示器上。它是 GPU 中的一个硬件单元，可以支持多种接口，例如 HDMI（高清多媒体接口，high-definition multimedia interface）、DisplayPort（DP）、DVI（数字视频接口，digital visual interface）和 VGA（视频

图形阵列，video graphics array）等。要显示的颜色缓冲区可能会位于不同的存储器中，例如：CPU 用于执行该任务的相同内存中、专用的帧缓冲存储器中、视频存储器中，后者可以包含任何 GPU 数据，但是 CPU 无法直接进行访问。每个接口都使用各自的标准协议，来传输颜色缓冲中的部分内容、时间信息，有时甚至可以是音频信息。VDC 还可以执行图像缩放、降噪、合成多个图像源等功能。

显示器（例如液晶显示器 LCD）更新图像的速度通常在每秒 60 到 144 次（赫兹 Hz）之间，这也被称为垂直刷新率（vertical refresh rate）。大多数观众会在低于 72 Hz 的刷新率下注意到闪烁现象。有关此主题的更多信息，详见章节 12.5。

显示器技术在很多方面都取得了进步，包括刷新率、每分量的比特数、色域和同步技术等。过去的刷新率通常是 60 Hz，但是现在 120 Hz 也变得越来越普遍，甚至可以达到 600 Hz。对于高刷新率的显示器而言，图像通常会被显示多次，有时还会插入黑色帧，从而尽量减少由于帧显示期间，由于眼睛移动而造成的模糊伪影[7, 646]。显示器中的每个通道也可以超过 8 bit，同时 HDR 显示器可能会成为下一个重大的显示技术革新，其中每个通道可以使用 10 bit 甚至更多。杜比的 HDR 显示技术使用了低分辨率的 LED 背光阵列来增强他们的 LCD 显示器，这样做可以使得显示器的亮度是普通显示器的 10 倍，对比度是普通显示器的 100 倍[1596]。更宽色域的显示器也正在变得越来越普遍，由于能够表示纯光谱色调（pure spectral hue），例如更加鲜艳的绿色，这些显示器可以显示更加广泛的颜色范围。有关色域的更多信息，详见章节 8.1.3。

为了减少画面撕裂效应，一些公司开发了自适应的同步技术，例如 AMD 的 FreeSync 和 NVIDIA 的 G-sync。这里的想法是对显示器的刷新率进行动态调整，而不是使用固定的显示器刷新率，从而适应 GPU 能够产生的显示帧率。例如：如果某一帧需要 10 毫秒来进行渲染，而下一帧需要 30 毫秒来进行渲染，那么可以在每个图像完成渲染之后，立即开始对需要显示的图像进行更新。使用这些技术，可以使得渲染看起来更加流畅。此外，如果图像不进行更新的话，那么就不需要将颜色缓冲发送到显示器中，这样还可以省电。

23.6.2 单、双、三重缓冲

在章节 2.4 中，我们提到过双缓冲机制是为了确保在渲染完成之前，图像不会显示在屏幕上。在这里，我们将介绍单缓冲、双缓冲甚至三缓冲机制。

假设我们现在只有一个缓冲区，那么这个缓冲区必须是当前显示在显示器上的那个。当绘制一个帧画面中的三角形时，随着显示器的刷新，屏幕上会出现越来越多的三角形，这是一个难以令人信服的效果。即使我们的帧率等于显示器的刷新率，这种单缓

冲也会存在问题。如果我们决定清除缓冲区或者绘制一个较大的三角形，那么当视频显示控制器传输正在绘制的部分颜色缓冲区时，我们将短暂地看到颜色缓冲区中的实际部分变化。这种现象有时会称为撕裂（tearing），因为显示的图像看起来就像是被短暂地撕成了两半一样，这不是实时图形所希望的功能和效果。在一些古老的系统上（例如 Amiga），我们可以测试光束的位置，从而避免在那里进行绘制，从而使得单缓冲也能够正常运行。如今，单缓冲已经很少使用了，虚拟现实系统可能是为数不多的例外，“与光束赛跑（racing the beam）”是一种可以减少延迟的方法[6]。

为了避免撕裂问题，我们通常会使用双缓冲机制。完成的图像显示在前置缓冲区（front buffer）中，而屏幕外的后置缓冲区（back buffer）中则包含了当前正在绘制的图像。后置缓冲和前置缓冲会由图形驱动进行交换，这个交换通常会发生在整个图像转移到显示器之后，从而避免撕裂现象。这种交换操作通常通过交换两个颜色缓冲区的指针来完成的。对于 CRT 显示器而言，这个事件被称为垂直回扫（vertical retrace），在这段时间内的视频信号被称为垂直同步脉冲（vertical synchronization pulse），简称垂直同步（vsync）。对于 LCD 显示器而言，并不会发生光束的物理回扫，但是我们会使用相同的术语，来表示整个图像刚刚被转移到显示器上。在图像渲染完成之后立即交换前后缓冲区，这对于测试渲染系统而言非常有用，并且也用于许多应用程序中，因为这样可以最大化帧率。不在垂直同步上进行更新也会导致撕裂现象，但是由于我们此时有了两个完整形成的图像，因此这里的瑕疵并不像单一缓冲时的那么糟糕。在交换完成之后，（新的）后置缓冲区会立即成为图形命令的接收方，而新的前置缓冲区则会显示给用户。这个过程如图 23.12 所示。

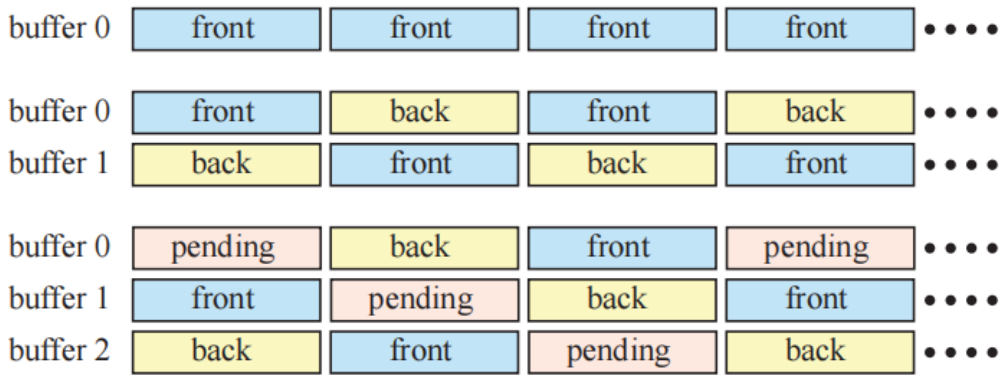


图 23.12：对于单一缓冲而言（顶部），总是会显示前置缓冲。对于双缓冲（中间）而言，第一个缓冲区 0 位于前台，缓冲区 1 位于后台。他们会在每一帧中从前台切换到后台，反之亦然。三缓冲（底部）还会通过一个等待缓冲区（pending buffer）来进行工作。在三缓冲机制中，首先会清除一个缓冲区，并开始对其进行渲染（pending）。其次，系统会继续使用缓冲区来进行渲染，直到图像完成（back）。最后，显示缓冲区（front）。

双缓冲机制还可以通过第二个后置缓冲来进行增强[1155]，我们称之为等待缓冲区（pending buffer），这种机制被称为三缓冲。这个等待缓冲区与后置缓冲区相类似，因为它也是一个离屏缓冲区，并且可以在显示前置缓冲区时对其进行修改。等待缓冲区是三缓冲循环中的一部分，在一帧期间，我们可以访问这个等待缓冲区。而在下一次交换时，它将成为新的后置缓冲区，并在此完成渲染。然后，它会成为新的前置缓冲区，并显示给观察者。而在下一次交换的时候，这个缓冲区会再次变为等待缓冲区。这个过程可以在图 23.12 底部直观地看到。

与双缓冲机制相比，三缓冲有一个主要优势，即系统可以在等待垂直回扫的时候访问这个等待缓冲区。使用双缓冲机制，当我们在等待垂直回扫完成以便可以进行交换的时候，相关的构造函数必须一直进行等待。这是因为前置缓冲区此时必须显示给观察者，而后置缓冲区则必须保持不变，因为其中存储了一个已经完成的图像，这个图像正在等待显示。三缓冲机制的缺点在于会增加一帧的延迟，它会延迟对用户输入（例如按键、鼠标或者操纵杆移动）的反应。控制可能会感觉变得缓慢，因为这些用户事件会在等待缓冲区开始渲染之后才会被更新，因此会被延迟。

理论上，我们可以使用 3 个以上的缓冲区。如果每帧计算的时间差异很大，那么更多的缓冲区会带来更多的平衡，以及更高的整体显示速率，但代价是更多的潜在延迟。概括地说，多缓冲机制可以被认为是一种循环结构，它有一个渲染指针和一个显示指针，分别指向不同的缓冲区。其中的渲染指针会对显示指针进行引导，并在当前渲染缓冲区计算完成的时候，移动到下一个缓冲区上。唯一的规则就是显示指针永远不会与渲染指针相同，即不会指向相同的缓冲区。

为 PC 图形加速器实现额外加速的一种相关方法是使用 SLI 模式。早在 1998 年，3dfx 公司就使用 SLI 来作为扫描线交错（scanline interleave）的缩写，其中两个图形芯片组会并行运行，其中一个处理奇数扫描线，另一个处理偶数扫描线。NVIDIA（收购 3dfx 资产的公司）使用这个缩写来表示可扩展的链接接口（scalable link interface），用于表示连接两个（或者更多）显卡的技术，它与前面的扫描线交错完全不同。AMD 将其称为 CrossFire X（译者注：一般叫做显卡交火）。这种形式的并行化，是通过将屏幕划分成两个（或者更多）水平部分来实现的，其中每张显卡负责渲染其中的一部分，或者是让每张显卡负责渲染自己的帧，然后进行交替输出。还有一种模式允许显卡对相同帧的抗锯齿进行加速。最常见的做法是让每个 GPU 渲染一个单独的帧，被称为交替帧渲染（alternate frame rendering，AFR）。虽然这种方案听起来似乎会增加延迟，但是它通常几乎没有影响。假设单个 GPU 的系统以 10 FPS 的速度进行渲染。如果此时 GPU 是性能瓶颈的话，那么两个使用 AFR 机制的 GPU，可以以 20 FPS 的速度进行渲染，甚至四个 GPU 可以以 40 FPS 的速度进行

渲染。每个 GPU 都会花费相同的时间来渲染自己负责的一帧，因此不一定会改变延迟。

由于屏幕分辨率的不断提高，这为基于像素采样的渲染器带来了严峻的挑战。保持帧率的一种方法是，自适应改变屏幕上[687, 1805]和表面上[271]的像素着色率。

23.7 深度剔除、测试和缓冲

在本小节中，我们将介绍与深度相关的所有内容，包括深度缓冲的分辨率、深度测试、基于深度的剔除、深度压缩、深度缓存、深度缓冲和 early-z。

深度的分辨率十分重要，因为它有助于避免渲染错误。例如：假设我们想要模拟一张放置在桌子上的纸，这张纸略高于桌子的表面。如果桌面和纸张的 z 深度具有精度限制，那么桌面可能会在不同的位置上穿透纸张，这个问题有时被称为 z-fighting。请注意，如果这个纸张与桌面具有完全相同的高度，即纸张和桌面共面，那么在没有额外信息的情况下，我们是无法正确判断二者的位置关系的。这个问题的原因是由于糟糕的建模，它无法通过更好的 z 精度来解决。

正如我们在章节 2.5.2 中所看到的，z-buffer（也叫深度缓冲）可以用来解决可见性问题。这种缓冲区通常会在每个像素上（或者样本）具有 24 bit 或者 32 bit，并且可以使用浮点数表示或者定点数表示[1472]。对于正交投影而言，这个距离值与 z 值成正比，即深度是均匀分布的（线性）。然而，对于透视投影而言，深度是不均匀分布的（非线性）。在应用透视变换之后（方程 4.74 或者方程 4.76），还需要除以 w 分量（方程 4.72）。此时深度分量会变为 $p_z = q_z/q_w$，其中点 $\mathbf{q}$ 是与投影矩阵相乘后的点。对于定点数表示法，值 $p_z = q_z/q_w$ 会从其有效范围（例如：DirectX 的 $[0, 1]$）映射到整数范围 $[0, 2^{b-1}]$ 内，并存储在 z-buffer 中，其中 b 是 bit 数。有关深度精度的更多信息，详见章节 4.7。

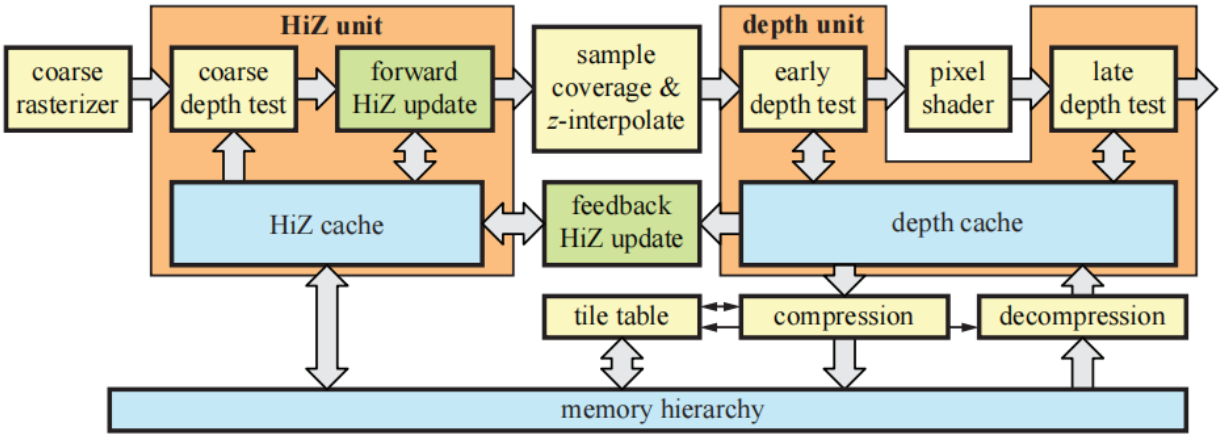


图 23.13：一种深度管线的可能实现方式，其中的 z 插值模块（z-interpolate）会使用插值来直接计算深度值。[46]

硬件的深度管线如图 23.13 所示，这个管线的主要目标是，在对一个图元进行光栅化的时候，对每个传入深度进行测试，当某个片元通过了深度测试，可能会将传入的深度写入到深度缓冲区中的对应位置。同时，这个管线必须十分高效。图的左侧部分从粗光栅化（coarse rasterization）开始，即在 tile 级别上进行光栅化（章节 23.1）。此时，只有与图元重叠的 tile 才会被传递到下一阶段中，即所谓的 HiZ 单元，即执行 z-culling 技术的阶段。

HiZ 单元从一个被称为粗深度测试（coarse depth test）的块开始，这里通常会执行两种类型的测试。我们首先介绍 z_{max} 剔除，它是 Greene 层次 z 缓冲算法的简化[591]，我们在章节 19.7.2 中进行了介绍。这个想法是在每个 tile 中存储所有深度的最大值，称为 z_{max} 。具体的 tile 大小与架构相关，通常会使用 8×8 像素[1238]。这些 z_{max} 值可以存储在固定的片上存储器中（on-chip memory），或者是通过缓存进行访问。在图 23.13 中，我们将其称为 HiZ 缓存。简单来说，我们想测试三角形是否完全被这个 tile 遮挡。为此，我们需要计算这个 tile 中三角形的最小 z 值 z_{min}^{tri} 。如果 $z_{\min}^{\text{tri}} > z_{\max}$ ，那么可以保证这个三角形被该 tile 中先前渲染的几何物体所遮挡。因此可以终止对该 tile 中三角形的后续处理，这样就节省了逐像素的深度测试。请注意，它不会产生任何像素着色器的调用和执行，因为逐样本的深度测试将会移除在管线中被遮挡的片元。在实际应用中，我们无法计算出 z_{min}^{tri} 的确切值，因此只能计算一个保守的估计值。计算 z_{min}^{tri} 有几种不同的方法，每种方法都有自己的优缺点：

1. 可以使用三角形三个顶点中的最小 z 值。这种方法并不总是准确的，但是开销很小。
2. 使用三角形的平面方程，计算处该 tile 四个角点的 z 值，并使用其中的最小值。

将这两种策略结合使用，可以获得最佳的剔除性能，这是通过取两个 z_{min} 值中较大的那个来实现的。

另一种类型的粗深度测试是 z_{min} 剔除，其思想是将所有像素的 z_{min} 存储在一个 tile 中[22]。它有两种用途，首先，它可以用来避免 z-buffer 的读取。对于正在渲染的三角形，如果我们能够绝对确保它位于所有之前渲染的几何图形前面，那么就没有必要进行逐像素的深度测试了。在某些情况下，这种 z-buffer 的读取是可以完全避免的，这进一步提高了性能。其次，它可以用于支持不同类型的深度测试。对于 z_{max} 剔除方法，我们假设使用标准的“小于（less than）”深度测试。但是，如果可以将

剔除与其他深度测试一起使用，并且如果 z_{min} 和 z_{max} 都是可用的时候，那么使用这个剔除过程可以支持所有类型的深度测试，这将是十分有益的。有关深度管线更加详细的硬件描述，详见 Andersson 的博士论文[49]。

图 23.13 中的绿色方框代表了更新 tile 中 z_{min} 和 z_{max} 的不同方式。如果三角形覆盖了整个 tile，那么可以直接在 HiZ 单元中进行更新。否则，就需要读取整个 tile 中的逐样本深度，并将其修改为对应的最小值和最大值，然后再发送回 HiZ 单元中，这将会引入一些延迟。Andersson 等人[50]提出了一种新的方法来执行这个操作，该方法不需要来自深度缓存的昂贵反馈，并且仍然能够保留大部分的剔除效率。

对于在粗深度测试中幸存下来的 tile，将会确定像素或者样本的覆盖率（使用章节 23.1 中所描述的边缘方程），并计算每个样本的深度（在图 23.13 中称为 z-interpolate）。这些值会被转发到深度单元中，如图中右侧所示。根据 API 的描述，像素着色器的计算应当遵循以下步骤。然而，在某些情况下（下文中将进行讨论），可以在不改变预期行为的情况下，进行一些额外的测试，这被称为 early-z [1220, 1542]或者早期深度测试。Early-z 实际上只是在像素着色器之前执行逐样本的深度测试，被遮挡的片元将会被丢弃。通过这个操作，可以避免不必要的像素着色器的执行。early-z 测试经常会与 z 剔除相混淆，实际上二者是由完全独立的硬件进行执行的。其中的任何一项技术都可以独立使用。

在许多情况下，所有的 z_{max} 剔除、 z_{min} 剔除、early-z 都会由 GPU 自动使用并执行。但是，如果像素着色器中存在写入自定义深度、使用丢弃（discard）操作、或者是将值写入无序访问视图（UAV）时，则必须禁用这些功能[50]。如果无法使用 early-z 的话，那么深度测试将会在像素着色器之后再进行，这被称为后期深度测试（late depth test）。

在一些较新的硬件中，可以对着色器中的图像执行原子性的读-改-写操作，以及加载和存储。在这些情况下，如果我们知道这样做是安全的话，那么我们可以显式启用 early-z 并覆盖这些约束。另一个可以在像素着色器输出自定义深度时使用的特性是保守深度（conservative depth）。在这种情况下，如果程序员能够保证自定义深度一定大于三角形深度的话，则可以启用 early-z。而在本例中，我们可以启用 z_{max} 剔除，但是不能启用 early-z 和 z_{min} 剔除。

与之前一样，遮挡剔除可以在从前向后的渲染顺序中获得好处。另一种具有类似名称和类似意图的技术是 z-prepass。这个想法是，我们首先渲染一遍场景，但是只会写入深度信息，同时禁用像素着色和写入颜色缓冲的操作。在渲染后续 pass 的时候，使用“相等（equal）”深度测试，这意味着只有位于最前面的表面才会被着色，因为此时 z-buffer 已经被初始化了，详见章节 18.4.5。

作为本小节的总结，我们将简要介绍深度管线中的缓存和压缩，如图 23.13 右下角所示。这里的一般压缩系统类似于章节 23.5 中所描述的系统，其中每个 tile 都可以被压缩到几个选定的大小，并且总会有一个未压缩数据的备用方案（fallback），当压缩无法达到任何选定大小的时候，就会直接使用未压缩数据。快速清除（fast clear）用于在清除深度缓冲区时节省带宽占用。由于深度在屏幕空间中是线性的，因此常见的压缩算法要么会以高精度来存储平面方程、要么会使用增量编码（delta encoding）的差分技术、要么会使用一些锚点方法[679, 1238, 1427]。分块表和 HiZ 缓存可以完全存储在片上缓冲区中，或者也可以通过内存层次的其他部分来进行通信，就像深度缓冲一样。在芯片上进行存储是十分昂贵的，因为这些缓冲区需要足够大，才能处理所支持的最大分辨率。

23.8 纹理化

虽然一些纹理操作（包括获取、过滤和解压等），可以在 GPU 多处理器上的纯软件中实现，但是有相关研究表明，使用固定功能的纹理硬件可以使得这些操作快 40 倍[1599]。纹理单元会进行寻址、过滤、clamping 和解压纹理格式等操作（第 6 章）。它与纹理缓存一起使用，可以减少带宽开销。我们首先讨论过滤相关的话题，以及它对纹理单元的影响。

为了能够使用纹理缩小滤波器（例如 mipmap 和各向异性过滤），我们需要获取纹理坐标在屏幕空间中的导数。也就是说，想要计算纹理的细节层级，我们需要 $\partial u/\partial x$ ， $\partial x/\partial x$ ， $\partial u/\partial y$ ， $\partial v/\partial y$ 。这些梯度信息能够告诉我们这个片元所代表纹理的区域或者函数的范围（extent）。如果直接使用顶点着色器传入的纹理坐标来访问纹理，那么这个导数可以通过解析计算得到。如果纹理坐标会使用一些函数进行变换，例如 $(u', v') = (\cos v, \sin u)$ ，那么使用解析方法来计算导数就会变得更加复杂。然而，使用链式法则（chain rule）或者符号微分法（symbolic differentiation）仍然是可能的[618]。尽管如此，图形硬件都不会选择使用这些方法，因为实际情况可能会非常复杂。想象一下我们现在要使用环境贴图来计算表面上的反射信息，其中表面的法线信息使用了凹凸贴图。我们是很难用解析的方法来计算导数信息的，例如：计算根据法线贴图进行反弹的反射向量导数，然后再使用这个导数用来访问环境贴图，这是十分困难的。因此，导数通常会在一个四边形基底上（即 2×2 像素），使用 x 和 y 的有限差分来进行数值计算。这也是为什么 GPU 架构专注于调度四边形的原因。

一般来说，导数的计算都是在幕后进行的，也就是说，它们对用户是隐藏的。实际的实现通常是在四边形上，使用 cross-lane 指令（shuffle/swizzle）来完成的，这样

的指令可以由编译器进行插入。一些 GPU 会使用固定功能的硬件单元来计算这些导数，具体如何计算导数并没有确切的规范和标准。一些常用的方法如图 23.14 所示。OpenGL 4.5 和 DirectX 11 同时支持粗略导数（coarse derivative）和精细导数（fine derivative）[1368]。

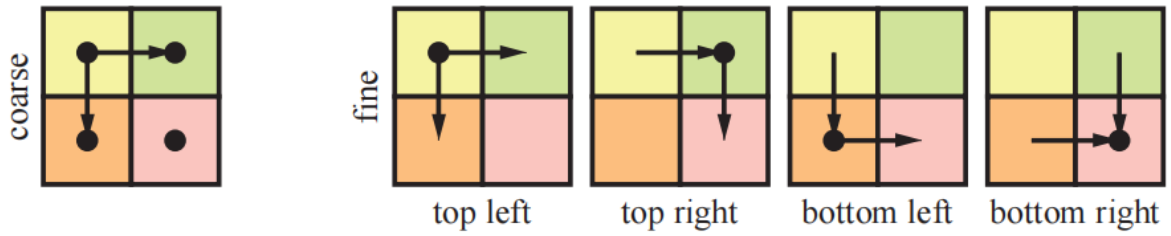


图 23.14：如何计算导数的示意图。图中的箭头代表了，从箭头结束处像素与箭头开始处像素之间的差值。例如：左上角（top left）的水平差值为右上角像素减去左上角像素。对于粗略导数（左），四边形内的所有四个像素都会使用同一个水平差值和同一个垂直差值。对于精细导数（右），则会使用最接近该像素的差值。[1368]

所有 GPU 都使用了纹理缓存[362, 651, 794, 795]，用来减少纹理的带宽使用。一些架构会使用一个专用的缓存来处理纹理，甚至是使用两个专用级别的纹理缓存，而另一些架构则会在所有类型的访问之间共享一个缓存，包括纹理信息。通常会使用一个小的片上存储器（通常是 SRAM）来实现纹理缓存。这个缓存存储了最近纹理读取的结果，其访问速度非常快。具体的替换策略和缓存大小取决于架构的实现。如果相邻像素需要访问相同纹素或者位置相近的纹素，它们很可能会在缓存中找到这些纹素信息。正如章节 23.4 所述，内存访问通常会以 tile 的方式进行，因此缓存中并不是按照扫描线顺序来存储纹素的，而是会将其存储在较小的 tile 中，例如 4×4 纹素。将同一个 tile 的纹素放在一起，因此这样可以提高效率[651]。tile 的大小（以 byte 为单位）通常与缓存行的大小相同，例如 64 byte。另一种存储纹理的方法是使用混合模式（swizzled pattern）。假设纹理坐标已经被转换为了定点数字 (u, v)，其中每个 u 和 v 都有 n bit。u 中第 i 个 bit 表示为 u_i。然后将 (u, v) 重新映射到一个混合的纹理地址 A 上，即：

$$A(u, v) = B + (v_{n-1}u_{n-1}v_{n-2}u_{n-2} \ldots v_1u_1v_0u_0) \cdot T \qquad (23.14)$$

其中 B 是纹理的基址（base address），T 是一个纹素所占用的 byte 数。这种重新映射的优点在于，它可以产生如图 23.15 所示的纹素顺序。我们从图中可以看到，这是一个空间填充曲线（space-filling curve），被称为 Morton 序列[1243]，它可以提高一致性[1825]。在这种情况下，曲线是二维的，因为纹理通常也是二维的。

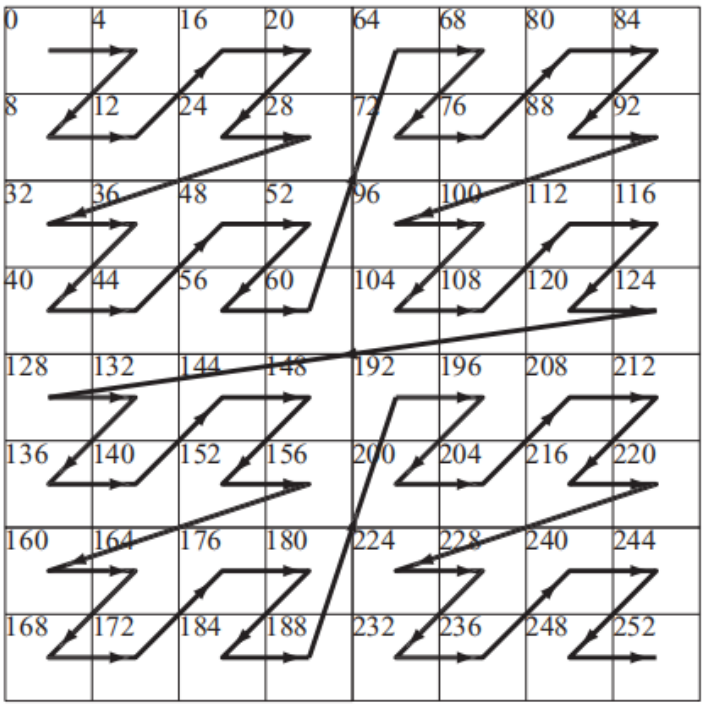


图 23.15：这种纹理混合（texture swizzle）方式增加了纹素内存访问的一致性。请注意这里的纹素大小为 4 byte，具体的纹素地址显示在了每个纹素的左上角。

纹理单元中还包含了一些定制结构，来对几种不同的纹理格式进行解压（章节 6.2.6）。与软件实现相比，在固定功能的硬件单元中实现这些功能，其效率通常会高很多倍。请注意，在使用纹理作为渲染目标和纹理映射的时候，还会出现一些其他的压缩机会。如果启用了对颜色缓冲区的压缩（章节 23.5），那么当访问这样一个渲染目标来作为纹理时，有两个设计选项。当渲染目标完成渲染时，第一种选择是从颜色缓冲压缩格式中，对整个渲染目标进行解压，并将其进行未压缩存储，以供后续纹理访问。第二种选择是在纹理单元中添加相应的硬件支持，来对颜色缓冲压缩格式进行解压[1716]。后者通常是更加高效的选择，因为其中的渲染目标即使在作为纹理访问期间，也可以保持压缩格式。有关缓存和压缩的更多信息，详见章节 23.4。

mipmap 对于纹理缓存的局部性也非常重要，因为它限制了纹素–像素的最大比例。在遍历一个三角形的时候，每个新像素大约代表了纹理空间中的一格纹素。mipmap 是渲染中少数几种，能够同时提高视觉效果和性能的情况之一。

23.9 架构

实现更快图形计算的最佳方法是利用并行性，这几乎可以在 GPU 的所有阶段中进行。其思想是同时计算多个结果，然后在稍后阶段再对这些结果进行合并。一般来

说，并行图形架构的组织方式如图 23.16 所示。应用程序将任务发送到 GPU 中，在经过一些调度之后，会在若干个几何单元（geometry unit）中并行进行几何处理。几何处理的结果会被转发到一组光栅化单元（rasterizer unit）中，由这些光栅化单元进行光栅化。然后通过一组像素处理单元（pixel processing unit）来并行执行像素着色和混合操作。最后，将得到的图像发送到显示器以供查看。

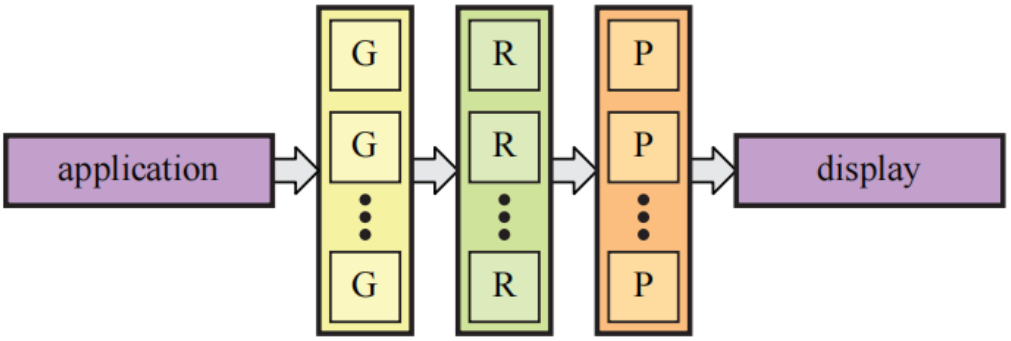


图 23.16：高性能、并行化计算机图形体系结构的一般架构，它由几个几何单元（G）、光栅化单元（R）和像素处理单元（P）组成。

对于软件和硬件而言，最重要的是要意识到，如果我们的代码或者硬件中存在串行部分，那么它可能会限制整体的性能提升。这可以使用 Amdahl 定律进行表示，即：

$$a(s,p) = \frac{1}{s + \dfrac{1-s}{p}} \tag{23.15}$$

其中 s 是程序/硬件中串行部分所占据的百分比，相应的 $1-s$ 就是可以并行化的百分比。此外， p 是程序或者硬件在进行并行化之后，所能达到的最大性能提升因子。例如：如果我们最初只有一个多处理器，后来增加了三个，那么这里 $p=4$ 。方程 23.15 中的 $a(s,p)$ 是指我们从改进中所得到的加速因子。如果我们有这样的一个架构，例如：其中的 10% 是串行的，即 $s=0.1$ ；现在我们对架构进行改进，使得剩余部分（非串行）的性能提高 20 倍，即 $p=20$ ，那么我们可以得到 $a = 1/(0.1+0.9/20) \approx 6.9$ 。此时可以看到，我们并没有得到 20 倍的加速，原因在于代码/硬件中的串行部分严重限制了性能的提升。实际上，当 $p \to \infty$ 时，我们也只能得到 $a=10$ 。究竟是把精力花在改进并行部分上更好，还是花在改进串行部分上更好，这一点并不总是十分明确，但是在并行部分得到实质性改进之后，串行部分将会对性能产生更多的限制。

对于图形架构而言，多个结果是并行计算的，但是 draw call 中的图元将会按照 CPU 上的提交顺序来进行处理。因此，必须进行某种形式的排序，从而使得并行单元能够一起渲染用户想要的图像。具体来说，我们所需要的排序是从模型空间到屏幕空间的（章节 2.3.1 和章节 2.4）。需要注意的是，几何单元和像素处理单元可以映射为相同的硬件单元，即统一的 ALU。在我们的案例研究部分中，所有的架构都使用了统一的着色器架构（章节 23.10）。即使是这样，理解排序所发生的位置也十分重要，我们在这里提出一种并行架构的分类方法[417, 1236]。排序可以发生在图形管线中的任何地方，这在并行架构中产生了四种不同类型的工作分布，如图 23.17 所示。它们分别被称为 sort–first、sort–middle、sort–last fragment 和 sort–last image。请注意，这些架构给出了不同的方式来分配 GPU 中并行单元之间的工作。

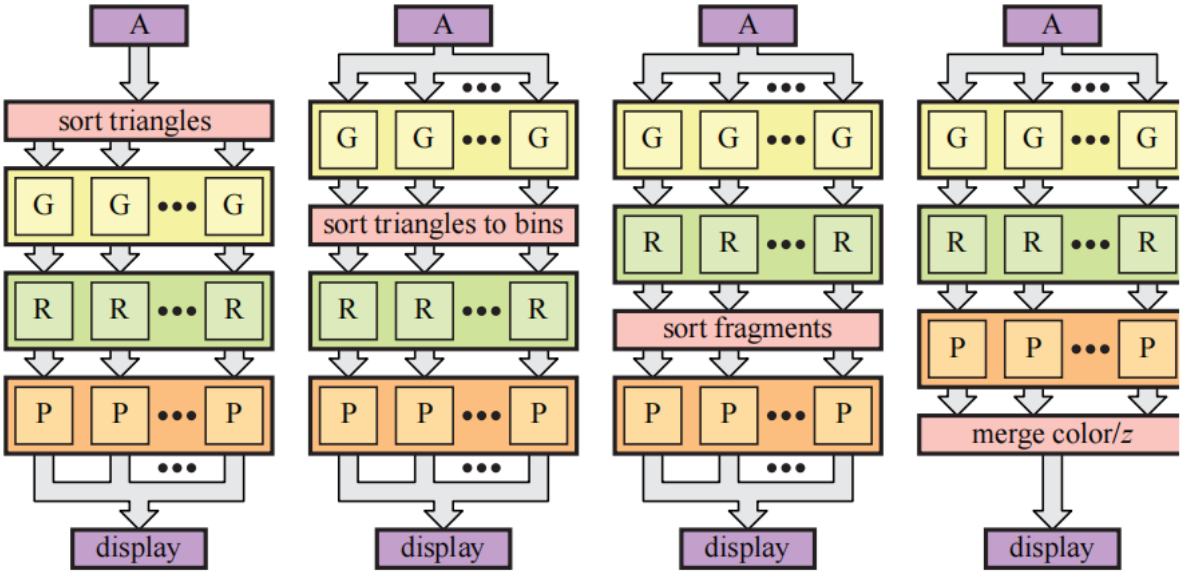


图 23.17：并行图形架构的分类。其中 A 是应用程序，G 是几何单元，R 是光栅化单元，P 是像素处理单元。从左到右，架构分别是：sort–first、sort–middle、sort–last fragment 和 sort–last image。[417]

基于 sort–first 的架构会在几何阶段之前对图元进行排序。其策略是将屏幕划分为一组区域，并将该区域内的图元发送到“拥有”该区域的完整管线中，如图 23.18 所示。图元在被初始化之后，就知道它需要被发送给哪个区域管线了，这就是排序步骤。sort–first 是在单台机器上探索最少的架构[418, 1236]，即它适用于多设备的渲染。在驱动具有多个屏幕或者投影仪的系统时，这种方案确实有用，因为每个屏幕可以专门使用一台计算机进行渲染[1513]。一种叫做 Chromium [787]的系统已经被开发出来了，它可以使用一个工作站集群，来实现任何类型的并行渲染算法。例如：sort–first 和 sort–last 可以实现较高的渲染性能。

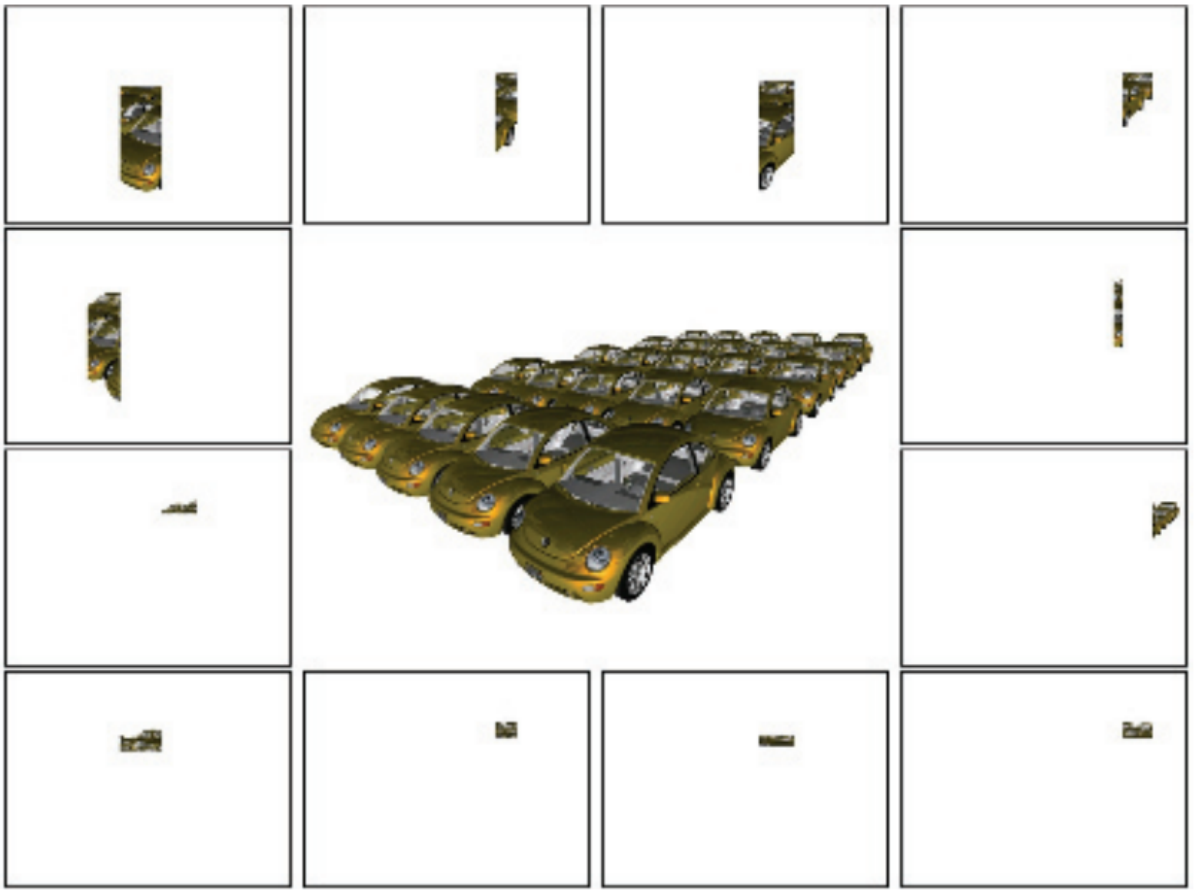


图 23.18。sort-first 会将屏幕划分成多个 tile，并为每个 tile 分配一个处理器。然后将图元发送给与它们相重叠的处理器。这与 sort-middle 架构不同，后者需要在几何处理完成之后，再对所有的三角形进行排序。只有在所有三角形都被排序之后，才能开始逐像素的光栅化。

Mali 架构（章节 23.10.1）属于 sort-middle 类型。几何处理单元会被赋予大约相同数量的几何处理工作量。然后，变换后的几何图形会被排序为相互不重叠的矩形，这些也称为 tile，所有的 tile 一起会覆盖整个屏幕。请注意，变换后的三角形可能会与多个 tile 重叠，因此可能会被多个光栅化单元和像素处理单元进行处理。这里效率的关键是，每一对光栅化单元和像素处理单元，在芯片上都有一个 tile 大小的帧缓冲，这意味着所有的帧缓冲访问都是十分快速的。当所有的几何图形都被分类到各个 tile 之后，每个 tile 的光栅化和像素处理就可以彼此独立地进行。有些 sort-middle 类型的架构会对不透明几何图形执行逐 tile 的 z-prepass，这意味着每个像素只会进行一次着色。然而，并不是所有的 sort-middle 架构都会这样做。

sort-last fragment 架构会在光栅化之后（有时称为片元生成）和像素处理之前，对生成的片元进行排序。一个例子是章节 23.10.3 中所描述的 GCN 架构。就像 sort-middle 架构一样，图元会在几何单元中尽可能均匀地分布。sort-last fragment 的一个优点在于不会发生任何重叠，这意味着生成的片元只会被发送到一个像素处理单元

中，这是最优情况。但是如果一个光栅化单元处理的是大三角形，而另一个光栅化单元处理的是小三角形，那么就可能会发生负载不平衡的问题。

最后，sort-last image 架构会在像素处理之后进行排序。这种架构的可视化结果如图 23.19 所示。这个架构可以被看作是一组独立的管线。图元分布在不同的管线上，每个管线都会渲染一个具有深度的图像。然后在最后的合成阶段，所有图像会根据它们的 z-buffer 来进行合并。需要注意的是，sort-last image 系统无法完全实现 OpenGL 和 DirectX 之类的 API，因为这些 API 要求图元按照发送的顺序进行渲染。PixelFlow [455, 1235]是 sort-last image 架构的一个例子。PixelFlow 架构也值得注意，因为它使用了延迟渲染，这意味着它只会对可见的片元进行着色计算。然而，应当注意的是，目前没有任何一个架构（译者注：应该指的是商用架构）使用 sort-last image，因为这种架构会在管线末尾会占用大量带宽。

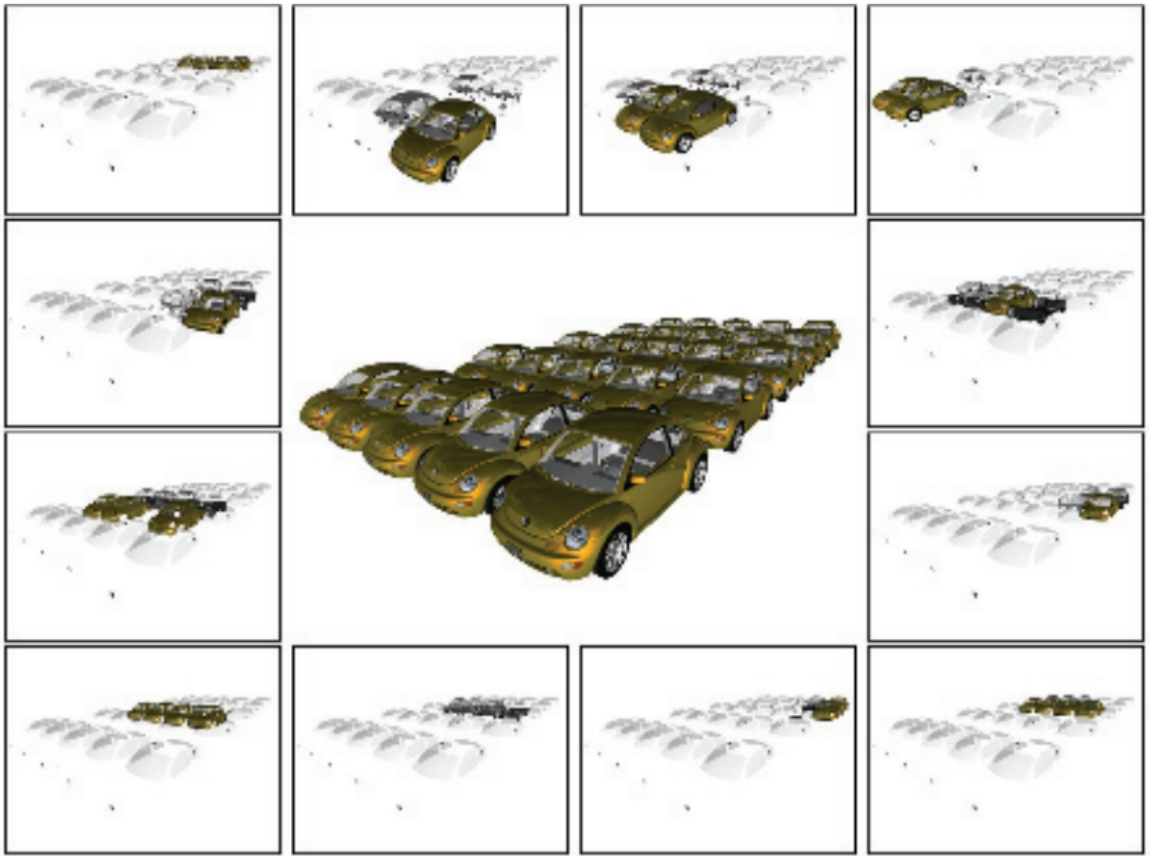


图 23.19：在 sort-last image 中，场景中的不同物体会被发送到不同的处理器中。在对最终渲染图像进行合成的时候，透明度是很难处理的，因此透明物体通常会被发送到所有节点中。

对于大型的 tile 显示系统，纯 sort-last image 方案的一个问题在于，需要在渲染节点之间传输大量的图像数据和深度数据。Roth 和 Reiners [1513]通过使用每个处理器结果的屏幕界限和深度界限，来对数据传输和合成的成本进行优化。

Eldridge 等人[417, 418]提出了 Pomegranate 架构，这是一种 sort-everywhere 架构。简而言之，它会在几何单元（G）和光栅单元（R）之间、光栅单元和（G）像素处理单元（P）之间、以及像素处理单元和显示器之间，都插入排序阶段。因此，随着系统规模的扩大（即增加了更多的管线），工作负载会变得更加平衡。其中的排序阶段被实现为一个具有点对点链接的高速网络。模拟实验表明，随着管线规模的增加，性能几乎会呈线性增长。

图形系统中的所有组件（主机、几何处理、光栅化和像素处理）连接在一起，构成了一个多处理系统。对于这样的系统，有两个众所周知的问题，并且它们几乎总是会与多处理相关：负载平衡和通信[297]。FIFO（先进先出 first-in，first-out）队列通常会存在于管线中的许多不同位置，以便作业可以按照顺序进行排队，从而避免管线的某些部分发生停滞。例如：可以在几何单元和光栅化单元之间放置一个 FIFO，这样一来，如果光栅化单元由于正在处理一个巨大尺寸的三角形，而无法跟上几何单元的速度，那么几何处理完成的三角形可以进行缓冲。

本文所描述的不同排序架构具有不同的负载平衡优点和缺点。详见 Eldridge 的博士论文[418]或者 Molnar 等人[1236]的论文，来获取更多相关信息。程序员也可以对负载平衡产生影响，相关技术在第 18 章中进行了讨论。如果总线的带宽过低或者带宽使用不当，那么通信可能会成为问题。因此，在设计一个应用程序的渲染系统时，应当使得任何总线（例如从主机到图形硬件的总线）都不会发生瓶颈，这是极其重要的。章节 18.2 中介绍了检测性能瓶颈的不同方法。

23.10 案例分析

在本小节中，我们将介绍三种不同的图形硬件架构。我们首先会介绍面向移动设备和电视的 ARM Mali G71 Bifrost 架构；然后介绍 NVIDIA 的 Pascal 架构；最后我们会介绍一种被称为 Vega 的 AMD GCN 架构。

请注意，对于尚未构建完成的 GPU，图形硬件公司通常会基于大量的软件模拟，来做出相应的设计决策。也就是说，多个应用程序（例如游戏）会在具有不同配置的参数化模拟器中进行运行，可能的参数包括：MP 的数量、时钟频率、缓存数量、光栅化引擎数量、曲面细分引擎数量和 ROP 数量等。这些模拟用于收集有关性能、功耗使用和内存带宽使用等因素的信息。最终，会选择在大多数用例中都最为高效的最佳配置，并根据该配置来构建 GPU 芯片。此外，软件模拟还可以帮助发现架构中的典型性能瓶颈，并解决这些瓶颈，例如增加缓存的大小等。对于一个特定的 GPU 而言，使用不同的速度和单元数量的原因其实非常简单，因为“这样涉及效果最好”。

23.10.1 案例研究：ARM Mali G71 Bifrost

Mali 产品线覆盖了 ARM 的所有 GPU 架构，Bifrost 是他们于 2016 年发布的架构（译者注：Bifrost 是北欧神话中，连接天宫和大地的彩虹桥）。这种架构的目标平台是移动设备和嵌入式系统，例如：手机、平板电脑和电视等。2015 年，基于 Mali 的 GPU 出货量为 7.5 亿块。由于其中许多设备都是由电池进行供电的，因此设计一个节能的、功耗较低的架构十分重要，而不仅仅是关注性能问题。因此，使用一个 sort-middle 架构是有意义的，在这种架构中，对帧缓冲的所有访问都发生在芯片上，这样做降低了功耗。所有的 Mali 架构都属于 sort-middle 类型，有时也被称为分块架构（tiling architecture）。图 23.20 展示了一个 GPU 的架构示意图。我们可以看到，G71 最多可以支持多达 32 个统一的着色器引擎。实际上 ARM 会使用“着色器核心（shader core）”这个术语，而不是“着色器引擎（shader engine）”，这里使用“着色器引擎”的目的是为了避免与本小节的其余部分相混淆。一个着色器引擎能够一次执行 12 个线程的指令，也就是说，一个着色器引擎中包含了 12 个 ALU。该架构选择使用 32 个着色器引擎，这是专门为 G71 配置的，该架构实际上可以扩展超过 32 个着色器引擎。

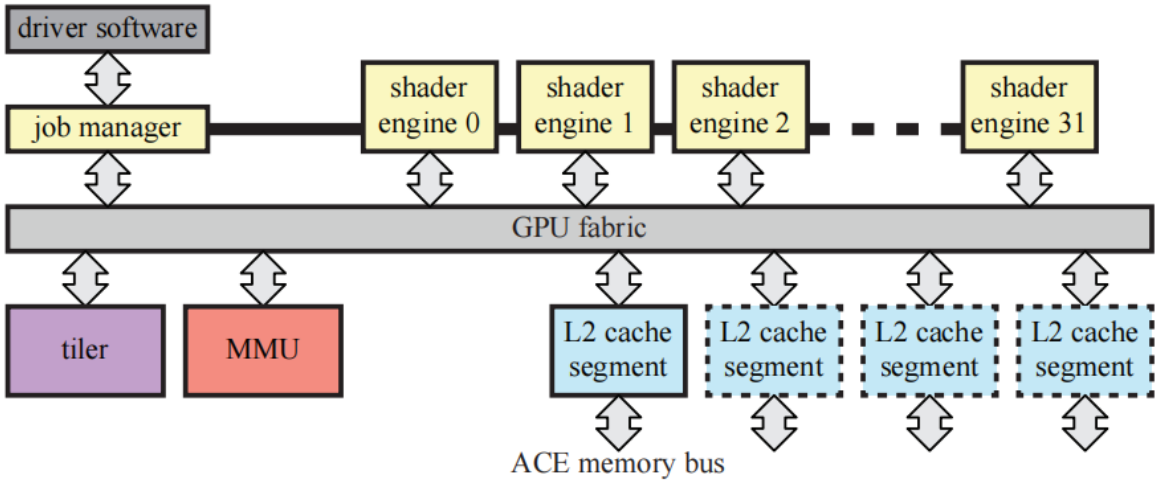


图 23.20：Bifrost G71 GPU 架构，该架构可扩展至 32 个着色器引擎，其中每个着色器引擎的架构如图 23.21 所示。[326]

会由驱动软件来为 GPU 分配工作。作业管理器（job manager），也就是调度器（scheduler），会将任务分配给着色器引擎。这些引擎通过一个 GPU 结构相连接，这个结构是一个总线，着色器引擎可以在该总线上与 GPU 中的其他单元进行通信。所有的内存访问都会提交给内存管理单元（memory management unit，MMU），它将一个虚拟内存地址转换到一个物理内存地址。

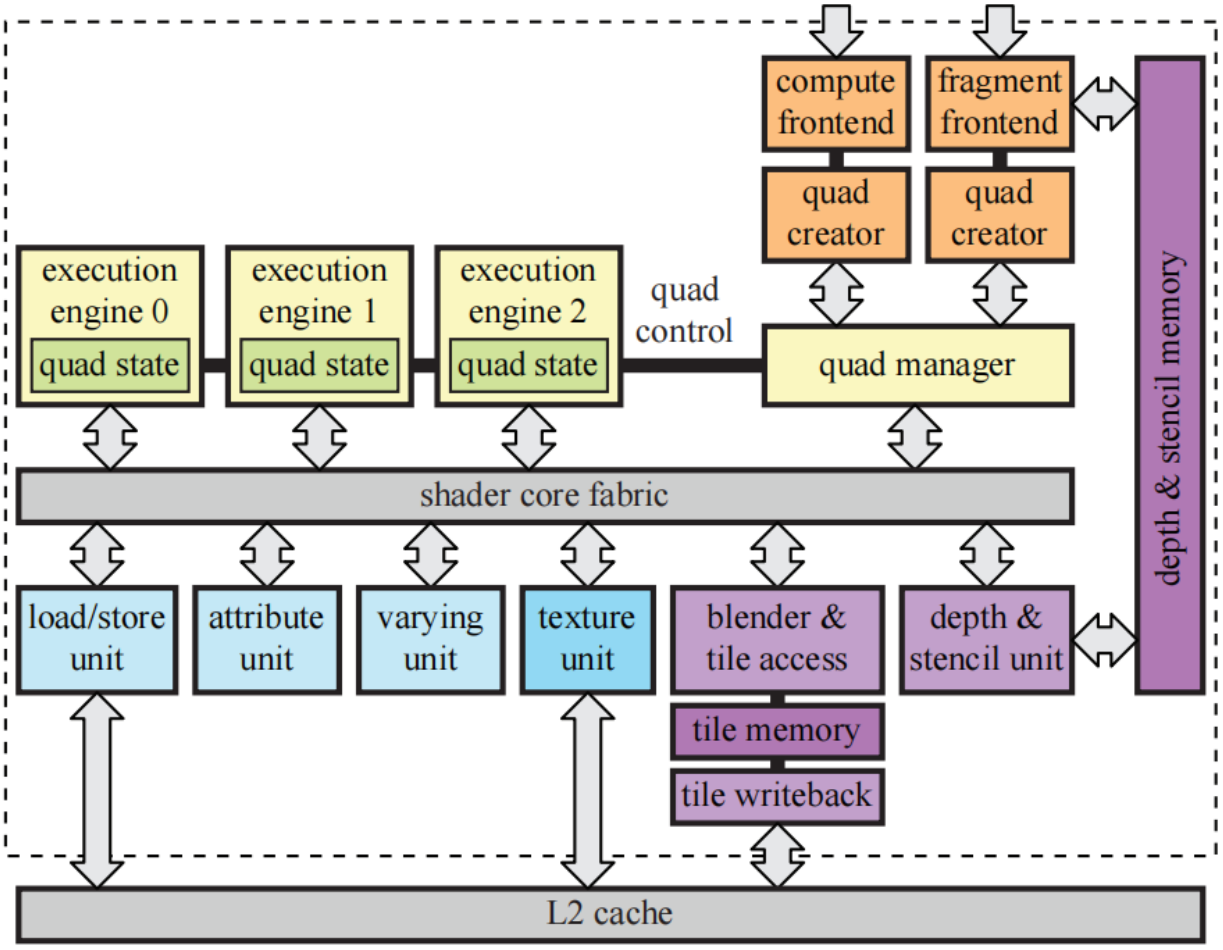


图 23.21：Bifrost 着色器引擎架构，其中 tile 内存位于芯片上，这有助于快速访问本地帧缓冲，并降低功耗。[326]

着色器引擎的架构图如图 23.21 所示。我们可以看到，它包含三个执行引擎，以着色 quad 为中心围绕分布。因此，它们被设计成 SIMD 宽度为 4 的小型通用处理器。每个执行引擎都包含了四个用于 32 bit 浮点运算的 FMA（fused–multiply–and–add）单元，以及四个用于 32 bit 加法运算的 FMA 单元。这意味着每个着色器引擎都有 3×4 个 ALU，即 12 个 SIMD lane。用本文之前的术语来说，这里的 quad 实际上就相当于一个 warp，例如：为了隐藏纹理访问所带来的延迟，架构可以在每个着色器引擎中保持至少 256 个线程。

请注意，着色器引擎是统一的，它们可以执行相关计算、顶点处理和像素着色等任务。其中的执行引擎（execution engine）还包含对许多超越函数的支持，例如正弦函数和余弦函数等。在使用 16 bit 浮点精度的时候，性能可以达到原来的 2 倍。当寄存器结果仅用作后续指令的输入时，这些单元还支持绕过寄存器中的内容，这样可以节省功耗，因为不需要访问寄存器堆。此外，在执行纹理访问或者其他内存访问的时候，quad 管理器还可以切换单个 quad，这与其他架构隐藏此类操作延迟的方式相类似。请注意，这是在小粒度级别上发生的，即只交换 4 个线程而不是全部的 12 个线

程。加载/存储（load/store）单元负责通用的内存访问、存储地址转换和一致性缓存[264]。属性单元（attribute unit）负责处理属性索引和寻址，它会将访问请求发送到加载/存储单元中。可变单元（varying unit）负责对不同的属性进行插值。

分块架构（也被称为 sort-middle）的核心思想是，首先执行所有的几何处理，以便找到每个待渲染图元的屏幕空间位置。同时，为帧缓冲中的每个 tile 都构建一个多边形列表（polygon list），其中包含了指向与该 tile 重叠的所有图元的指针。在这一步之后，我们就能够确定与一个 tile 重叠的所有图元集合。因此，tile 中的图元可以进行光栅化和着色计算，并将结果存储在片上的 tile 内存中。当该 tile 完成对其所有图元的渲染时，来自 tile 内存的数据会通过 L2 缓存写回到外部存储空间中。通过这种方式，能够减少了内存带宽的使用。然后下一个 tile 会被光栅化，以此类推，直到完整的一帧被渲染完成。第一个分块架构是 Pixel-Planes 5 [502]，该系统与 Mali 架构在高层次设计上存在一些相似性。

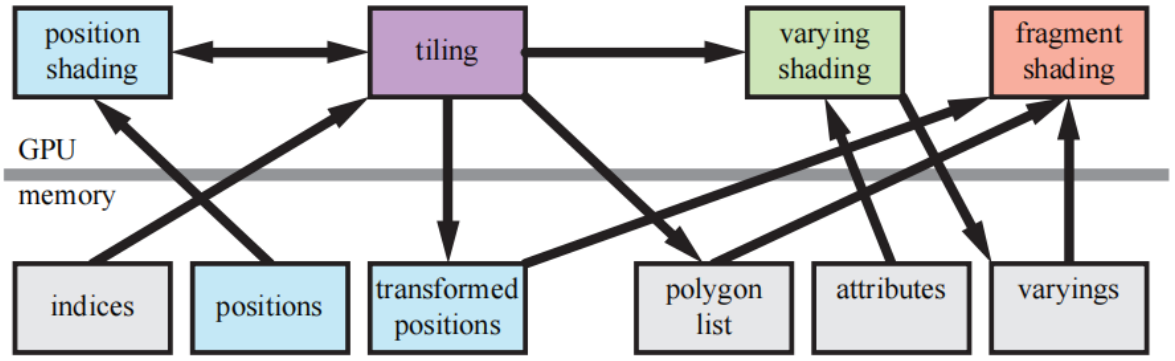


图 23.22：几何图形流经 Bifrost 架构的示意图。顶点着色器中包含了位置着色（position shading）和可变着色（varying shading），其中位置着色的结果会被分块（tiling）操作所使用，而可变着色则位于分块操作之后，只在需要的时候才会执行。[264]

图 23.22 展示了几何处理和像素处理。我们可以看到，顶点着色器被划分成了两个部分，其中一个部分只执行位置着色（position shading），另一个部分则被称为可变着色（varying shading），这是在分块之后完成的。与 ARM 之前的架构相比，这节省了内存带宽的开销。执行分组（binning）操作，即确定一个图元与哪些分块重叠，这个操作所需的唯一信息就是顶点的位置。执行分组操作的分块单元（tiler unit），会以分层方式进行工作，如图 23.23 所示。这有助于使得分块操作的内存占用更小，同时更加可预测，因为它不再与图元的大小成比例。

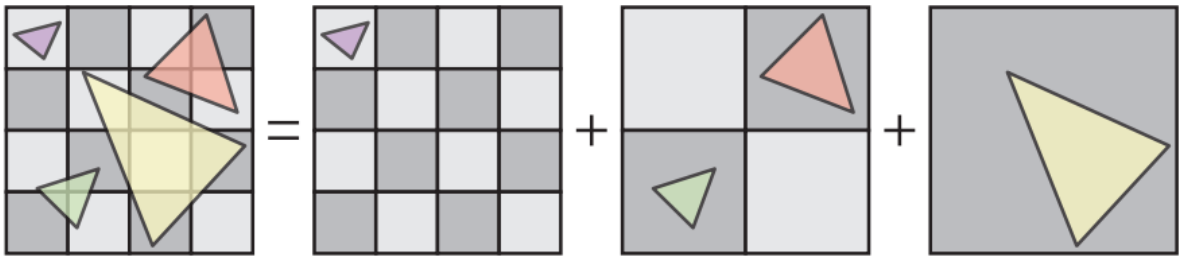


图 23.23：Bifrost 架构的分层分块操作。在这个例子中，图元分组是在三个不同层次上进行的，其中每个三角形都会被分配到与单个正方形重叠的层次上。[191]

当分块单元完成对场景中所有图元的分组之后，我们就可以确切地知道某个 tile 上重叠了哪些图元。因此，只要有空闲的着色器引擎可以并行工作，那么剩余的光栅化、像素处理和混合操作，可以对任何数量的 tile 进行并行执行。一般来说，一个 tile 会被提交给一个着色器引擎，由它来处理这个 tile 中的所有图元。当对所有的 tile 都完成上述这些操作的时候，也可以开始为下一帧进行几何处理和图元分组。这个处理模型意味着，在分块架构中可能会有更多的延迟。

在完成几何处理之后，光栅化过程、像素着色器执行、混合和其他逐像素的操作会紧随其后。分块架构的一个最重要的特性是，单个 tile 所对应的帧缓冲（例如：颜色、深度和模板等）可以存储在速度很快的片上内存中，这里将其称为 tile 内存（tile memory）。这个内存是负担得起的，因为每个 tile 都很小（16×16 像素）。当 tile 中的所有渲染操作都完成时，这个 tile 的预期输出（通常是颜色，可能还会有深度）将会被复制到与屏幕大小相同的片外帧缓冲中（位于外部内存中）。这意味着在逐像素处理期间，对帧缓冲的所有访问实际上都是近乎免费的。避免使用外部总线是非常可取的操作，因为这种对带宽的使用会带来较高的功耗成本[22]。将片上 tile 内存中的内容移出到片外帧缓冲的时候，仍然可以使用一些帧缓冲的压缩方法。

Bifrost 支持像素本地存储（pixel local storage，PLS），这是一组在 sort-middle 架构上通常会支持的扩展。使用 PLS，可以让像素着色器访问帧缓冲的颜色，从而实现自定义的混合技术。相比之下，混合阶段通常会使用 API 来进行配置，而不像像素着色器那样是可编程的。我们还可以使用 tile 内存，在每个像素上存储任意固定大小的数据结构。这允许程序员能够高效地实现一些技术，例如：延迟渲染技术。G-buffer（例如：法线、位置和漫反射纹理）会在第一个 pass 中存储在 PLS 中；而第二 pass 则会执行光照计算，并在 PLS 中累积计算结果，第三 pass 则使用了 PLS 中的信息，来计算最终的像素颜色。请注意，对于单个 tile 而言，所有这些计算都是在整个片上 tile 内存中进行的，这使得数据访问速度非常快。

所有的 Mali 架构在设计的时候，都会考虑到多重采样抗锯齿（MSAA），并实现了章节 5.4 中所描述的旋转网格超采样（rotated grid supersampling，RGSS）方案，其中每个像素使用 4 个子样本。sort-middle 架构非常适合抗锯齿技术，这是因为过滤操作是在 tile 数据离开 GPU，并发送到外部内存之前完成的。因此，外部内存中的帧缓冲只需要为每个像素存储一种颜色即可，而一个标准架构中的帧缓冲，需要是原来的四倍大。对于分块架构而言，我们只需要将片上的 tile 缓冲区增加 4 倍，或者使用较小的 tile 即可（宽度和高度为原来的一半）。

Mali Bifrost 架构还可以有选择地，只对一批渲染图元使用多重采样或者超采样。这意味着在需要的时候，可以使用一些更加昂贵的超采样方法，即为每个样本都执行一次像素着色器。其中一个例子就是使用 alpha 映射来渲染一棵纹理化的树，我们需要高质量的采样从而避免视觉瑕疵。对于这些图元而言，可以启用超采样技术。当这种复杂的情况结束，需要渲染一些更简单的物体时，可以切换回使用不太昂贵的多重采样方法。该架构还支持 8× MSAA 和 16× MSAA。

Bifrost（以及之前的架构 Midgard），还支持一种被称为事务消除（transaction elimination）的技术。这个想法是针对那些场景中不随帧变化的部分，避免将这些部分的 tile 内存转移到片外内存的操作。对于当前帧而言，当 tile 中的帧缓冲被移动到芯片外的帧缓冲时，会为每个 tile 都计算一个唯一的签名，这个签名是一种校验和（checksum）。对于下一帧而言，将会为即将被移除的 tile 计算签名，如果前一帧中的签名与当前帧中的签名相同，则架构会避免将对应的颜色缓冲区写入到片外内存中，因为此时对应部分的帧缓冲中已经存在了正确的内容。这对于休闲手机游戏而言（例如《愤怒的小鸟》）尤其有用，因为在这些游戏中，每一帧中所更新的场景比例比较小。还要注意的是，这种技术很难在 sort-last 架构上实现，因为这些架构并不是基于逐 tile 进行操作的。G71 还支持智能合成（smart composition），即将事务消除应用于用户界面的合成。如果所有源都与前一帧中的相同，并且操作也相同，那么则可以避免读取、合成和写入 tile 等操作。

低层级的低功耗技术（Low-level power-saving technique），例如时钟门控（clock gating）和功耗门控（power gating）技术，也被大量应用于该架构中。这意味着未使用的或者不活动的管线部分，将会被关闭或者保持闲置，从而降低能耗，减少电量使用。

为了减少纹理的带宽开销，有一个具有专门用于 ASTC 和 ETC 解压单元的纹理缓存。此外，压缩纹理会以压缩形式存储在缓存中，而不是解压之后再将纹理存储在缓存中。这意味着当我们对一个纹素发出访问请求的时候，硬件将会从缓存中读取对应

的数据块，然后对该数据块中的纹素进行动态解压。这种配置方式增加了缓存的有效大小，从而提高了效率。

一般来说，分块架构的一个优点在于，它天生可以很好地对各个 tile 进行并行处理，例如：我们可以添加数量更多的着色器引擎，使得每个着色器引擎都负责一个 tile 的独立渲染。分块架构的一个缺点在于，整个场景数据需要发送到 GPU 中进行分块，处理后的几何图形需要流式输出到内存中。一般来说，这种 sort-middle 架构并不适合处理几何图形的增强，例如使用几何着色器和曲面细分着色器，因为更多数量的几何图形，会增加变换几何图形时的内存传输量。对于 Mali 架构而言，几何着色（章节 18.4.2）和曲面细分着色器都是在 GPU 上的软件中实现的，Mali 的最佳实践指南建议[69]，永远不要使用几何着色器。对于大多数内容而言，sort-middle 架构非常适用于移动端和嵌入式系统。

23.10.2 案例研究：NVIDIA Pascal

Pascal 是 NVIDIA 开发的 GPU 架构（译者注：GTX 10 系显卡等）。它既可以作为一个图形部分存在[1297]，也可以作为一个计算部分存在[1298]，其中计算部分的目标是高性能计算和深度学习应用。在本小节的介绍中，我们将主要关注图形部分，特别是一种名为 GeForce GTX 1080 的显卡配置。我们将以自下而上的方式来介绍该架构，从最小的统一 ALU 开始，然后再构建整个 GPU。我们将在本小节的末尾，简要介绍一些其他的芯片配置。

Pascal 图形架构中所使用的统一 ALU（用 NVIDIA 术语来说就是 CUDA 核心）与图 23.8 左侧的 ALU 具有相同的高级示意结构。ALU 的重点是浮点运算和整数运算，但是它们也支持其他的一些操作。为了提高计算能力，几个这样的 ALU 被组合成一个流式多处理器（streaming multiprocessor，SM），简称流多处理器，或者叫做 SM 单元。在 Pascal 的图形部分中，SM 由四个处理块所组成，每个块包含 32 个 ALU。这意味着一个 SM 可以同时执行 4 个 warp，每个 warp 包含 32 个线程，如图 23.24 所示。

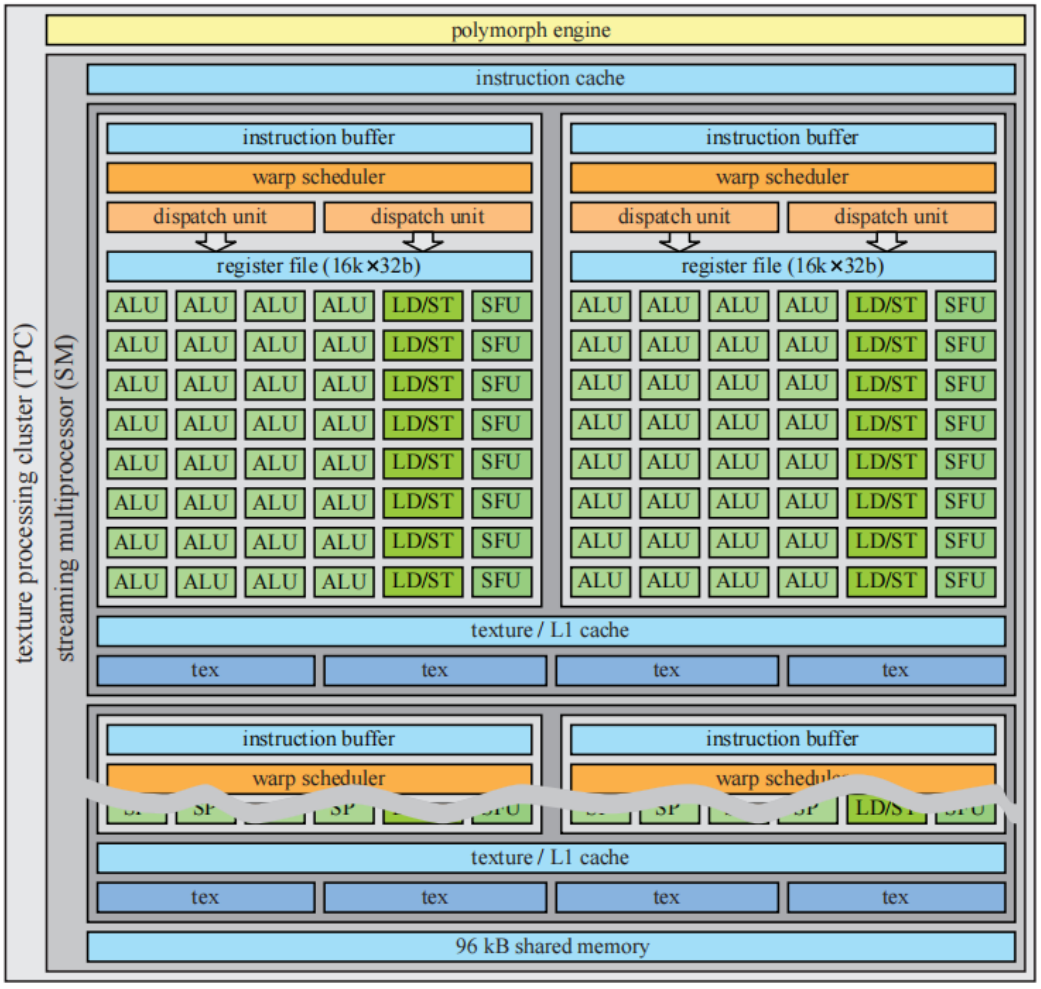


图 23.24：Pascal 的流多处理器（SM）具有 $32 \times 2 \times 2$ 的统一 ALU，并被一个多边形变形引擎（polymorph engine）所封装，它们共同构成一个纹理处理集群（texture processing cluster，TPC）。请注意，顶部深灰色框中的内容在下面会被复制，但是重复的部分被省略了。[1297]

每个处理块是一个宽度为 32 的 SIMT 引擎，还具有 8 个加载/存储（LD/ST）单元，以及 8 个特殊函数单元（special function unit，SFU）。加载/存储单元会对寄存器堆（register file）中的寄存器读写进行处理，这个寄存器堆的大小为 $16,384 \times 4$ byte，即每个处理块 64 kB，四个处理块加在一起，整个 SM 就是 256 kB。SFU 单元负责处理超越函数指令，例如：正弦、余弦、指数（以 2 为底数）、对数（以 2 为底数）、倒数和倒数平方根等。它们还支持属性插值[1050]。

SM 中的所有 ALU 都共享同一个指令缓存（instruction cache），而每个 SIMT 引擎都有属于自己的指令缓存，其中包含了最近加载的一组本地指令集合，从而进一步提高指令缓存的命中率。warp 调度器能够在每个时钟周期内调度两条 warp 指令[1298]，例如：可以在同一个时钟周期内，将工作同时调度到 ALU 和 LD/ST 单元中。请注意，每个 SM 还有两个 L1 缓存，每个 L1 缓存中包含 24 kB 的存储空间，即

每个 SM 48 kB。使用两个 L1 高速缓存的原因，可能是因为较大的 L1 高速缓存需要更多的读写端口，这增加了高速缓存的复杂性，并且使得其在芯片上的实现面积更大。此外，每个 SM 还有 8 个纹理单元。

由于着色计算必须在 2×2 大小的像素四边形中完成，因此 warp 调度器会找到 8 个不同的像素四边形，并将它们分为一组，以便在 32 个 SIMT lane 中进行执行[1050]。由于统一的 ALU 架构，因此 warp 调度器可以一个在 warp 中指定并分配顶点、像素、图元或者计算着色器的工作。请注意，一个 SM 可以同时处理不同类型的 warp（例如顶点、像素和图元）。在将当前执行的 warp 切换为待执行的 warp 时，这个架构也没有任何开销。至于下一次在 Pascal 上选择执行什么样 warp，相关细节并没有公开，但是之前的 NVIDIA 架构会给我们一些提示。在 2008 年 NVIDIA 的 Tesla 架构中[1050]，使用了一个计分板（scoreboard）来限定每个 warp 中的每个时钟周期。记分版是一种通用机制，它允许乱序执行而不会发生冲突。warp 调度器会在等待执行的 warp 中进行选择，例如那些不需要等待纹理加载返回的 warp，并选择其中优先级最高的那个。warp 的类型、着色指令的类型以及“公平性（fairness）”，这些参数会用来选择优先级最高的那个 warp。

SM 会与多边形变形引擎（polymorph engine，PM）一起工作，这个单元首次出现在 Fermi 芯片中[1296]。PM 会执行几个与几何相关的任务，包括顶点获取、曲面细分、同步多重投影（simultaneous multiprojection）、属性设置和流式输出等。第一阶段会从全局顶点缓冲区中获取顶点数据，并将顶点着色和壳着色的 warp 分配给 SM。第二阶段是可选的曲面细分阶段（章节 17.6），其中新生成的 (u, v) 面片坐标，会被分配到 SM 中，用于后续域着色和可选的几何着色。第三阶段会处理视口变换和透视校正。此外，这里执行了一个可选的同步多重投影步骤，它可以用于高效的 VR 渲染（章节 21.3.1）。接下来是可选的第四阶段，顶点会被流式输出到内存中。最后，会将结果转发给相关的光栅化引擎。

一个光栅化引擎包含三个任务，即三角形设置、三角形遍历、 z 剔除。其中三角形设置会获取顶点，计算边缘方程，并执行背面剔除。三角形遍历会使用分层的分块遍历技术，来访问与三角形相重叠的 tile，它会使用边缘方程来执行 tile 测试和内部测试。在 Fermi 芯片上，一个光栅化器每个时钟周期最多可以处理 8 个像素[1296]，而 Pascal 在这方面则没有相应的公开信息。 z 剔除单元会使用章节 23.7 中所描述的技术，在每个 tile 上来处理剔除。如果一个 tile 被剔除了，那么对该 tile 的处理将会立即终止。对于幸存下来的三角形而言，每个顶点的属性会被转换为平面方程，以便在像素着色器中进行高效计算。

一个流多处理器（SM）与多边形变形引擎 PM 结合在一起，被称为一个纹理处理集群（texture processing cluster，TPC）。在更高的层次上，5 个 TPC 会被分组到一个图形处理集群（graphics processing cluster，GPC）中，该集群有一个独立的光栅化引擎服务于这 5 个 TPC。GPC 可以被认为是一个小型的 GPU，其目标是为图形功能提供一组平衡的硬件单元，例如顶点、几何、光栅化、纹理、像素和 ROP 单元等。正如我们将在本小节结束时会看到的，创建独立的功能单元，可以使得设计人员能够更轻松地创建具有一系列功能的 GPU 芯片（即不同型号的 GPU 芯片）。

到目前为止，我们已经介绍了 GeForce GTX 1080 中的大部分构建块。它由 4 个 GPC 组成，一般的配置如图 23.25 所示。请注意，这里还有另一个级别的调度，它由 GigaThread 引擎提供支持，以及一个 PCIe v3 的接口。GigaThread 引擎是一个全局的工作分发引擎，它负责将线程块调度给所有 GPC。

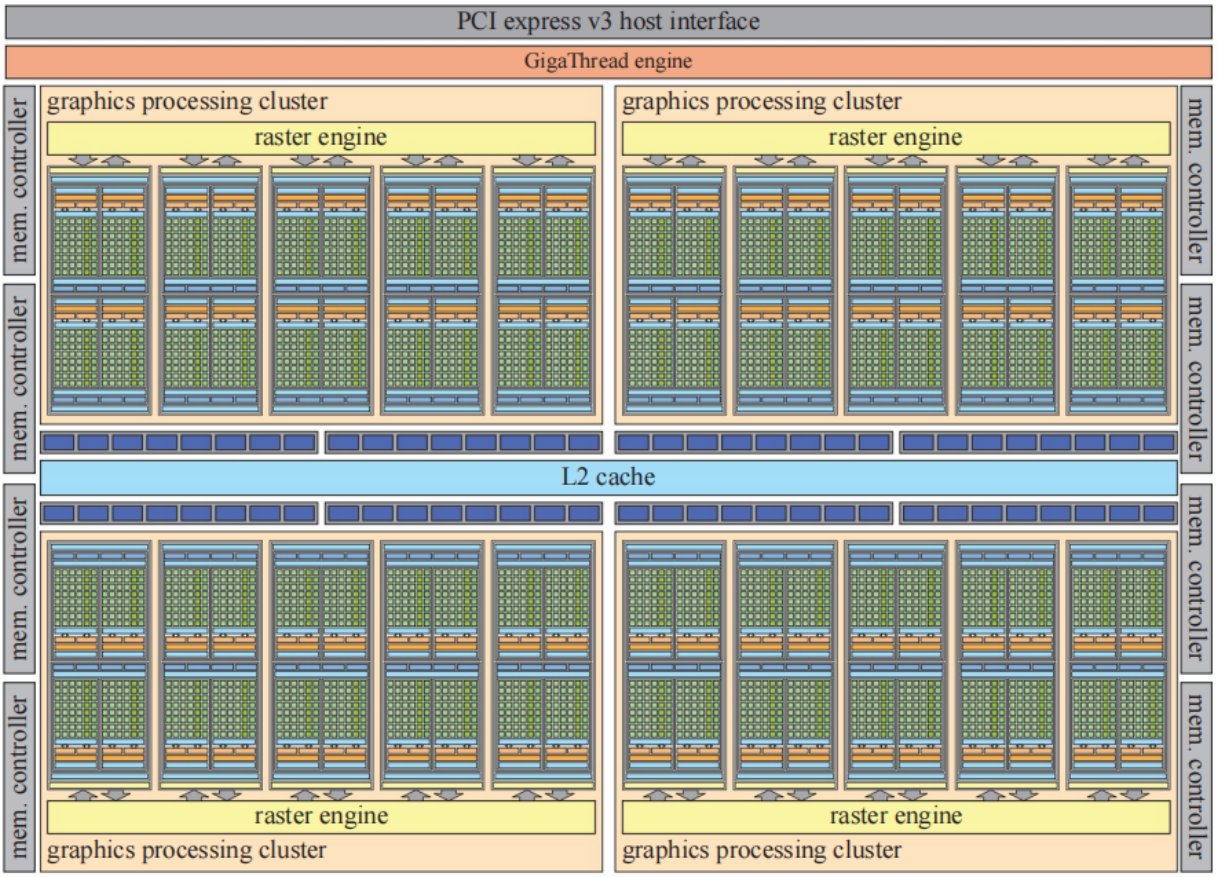


图 23.25：Pascal GPU 在 GTX 1080 显卡下的配置，它包含 20 个 SM 单元，20 个多边形变形引擎，4 个光栅化引擎，$8 \times 20 = 160$ 个纹理单元（峰值速率为 277.3 Gtexels/s），$256 \times 20 = 5120$ kB 的寄存器堆，总共 $20 \times 128 = 2560$ 个统一 ALU。[1297]

光栅操作单元（raster operation unit）也展示在了图 23.25 中，尽管有些隐藏。它们位于图中 L2 缓存的正上方和正下方。其中的每个蓝色区块都是一个 ROP 单元，

一共有 8 组，每组包含了 8 个 ROP 单元，共计 64 个 ROP 单元。ROP 单元的主要任务是将输出内容写入到像素和其他缓冲区中，并执行各种操作（例如混合）。从图 23.25 的左右两侧可以看出，一共有 8 个 32 bit 的内存控制器，加起来总共是 256 bit。8 个 ROP 单元各自被绑定在一个内存控制器和 256 kB 的 L2 缓存上。这为整个芯片提供了 2 MB 的 L2 缓存。每个 ROP 都被绑定到一个特定的内存分区中，这意味着一个 ROP 会对缓冲区中的某个像素子集进行处理。ROP 单元还可以处理无损压缩。除了支持未压缩和快速清除之外，还有三种不同的压缩模式[1296]。对于 $2:1$ 压缩模式而言（例如：从 256 B 压缩到 128 B），每个 tile 中都存储一个参考颜色值，并对像素之间的差值进行编码，其中每个差值信息，相比于未压缩状态，编码所使用的 bit 数会更少。$4:1$ 压缩模式是 $2:1$ 压缩模式的扩展，但是这种模式所能使用的差异编码 bit 要更少，并且它只适用于内容平滑变化的 tile。还有一种 $8:1$ 压缩模式，它是 2×2 像素块的 $4:1$ 恒定颜色压缩与上述 $2:1$ 压缩模式的组合。$8:1$ 模式的优先级要高于 $4:1$ 模式，$4:1$ 模式的优先级要高于 $2:1$，即总是会优先使用压缩比最高、并且能成功压缩 tile 的模式。如果所有这些压缩模式的尝试都失败了，那么则必须将 tile 以未压缩的形式存储在内存中。Pascal 压缩系统的效率如图 23.26 所示。

图 23.26：左侧展示了渲染后的图像，中间展示了 Maxwell 架构（Pascal 之前的架构）的压缩结果，右边展示了 Pascal 架构的压缩结果。图像越紫，说明缓冲区压缩的成功率越高。[1297]

Pascal 架构所使用的显存是 GDDRX5，其时钟频率为 10 GHz。在上面我们看到，8 个内存控制器总共可以提供 256 bit，即总共 32 B。这就提供了总计 320 GB/s 的峰值内存带宽，但是将多级缓存和压缩技术相结合，给人的印象总是能实现更高的有效速率。

该 GPU 芯片的基本时钟频率为 1607 MHz，当功耗预算十分充足时，还可以工作在 boost 模式（1733 MHz）。其峰值计算能力为：

$$\underbrace{2}_{\text{FMA}}\cdot\underbrace{2560}_{\text{num. SPs}}\cdot\underbrace{1733}_{\text{clock freq.}}=8,872,960\ \text{MFLOPS}\approx 8.9\ \text{TFLOPS} \tag{23.16}$$

方程 23.16 中的 2 来自于这样一个事实：一个 FMA（fused-multiply-and-add）运算，通常会被计数为两次浮点运算，我们将结果除以 10^6，可以将结果从 MFLOPS 转换为 TFLOPS。GTX 1080 Ti 拥有 3584 个 ALU，相当于 12.3 TFLOPS。

NVIDIA 开发 sort-last fragment 架构已经有很长时间了。然而，自 Maxwell 架构以来，它们还支持一种新的渲染类型，被称为分块缓存（tiled caching），它介于 sort-middle 和 sort-last fragment 之间，这种架构如图 23.27 所示。其核心思想是利用局部性和 L2 缓存。几何图形会以足够小的 tile 进行处理，以便每个 tile 的输出可以保存在这个缓存中。此外，只要与该 tile 重叠的几何图形还没有完成像素着色，则对应的帧缓冲也会保持在 L2 状态。

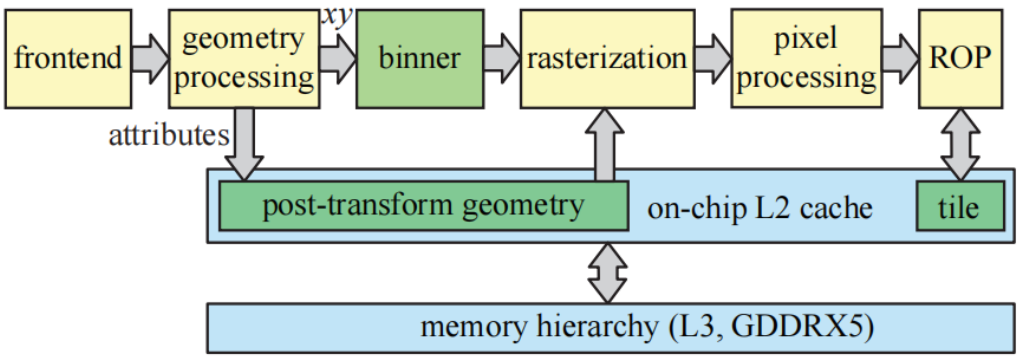


图 23.27：分块缓存引入了一个分组器（binner），它将几何图形分类到各个 tile 中，并让转换后的几何图形保持在 L2 缓存中。当前正在处理的 tile 也保持在 L2 缓存中，直到当前 tile 的几何图形完成渲染。

图 23.25 中包含了四个光栅化引擎，但是我们知道，图形 API 必须（在大多数情况下）遵循原始数据的提交顺序[1598]。帧缓冲通常会使用通用的棋盘格模式[1160]，来将其划分为不同的 tile，每个光栅化引擎都“拥有”一组 tile。如果当前三角形与光栅化引擎中的任意一个 tile 相重叠，那么这个这个三角形就会被发送到这个光栅化引擎中；如果这个三角形覆盖了多个 tile，并且这些 tile 分属于不同的光栅化引擎，那么会将这个三角形发送给每个光栅化引擎进行处理。通过这种方式，从而独立解决每个 tile 中的排序问题，这也有助于更好的负载均衡。在 GPU 架构中通常也有几个 FIFO 队列，它们的存在是为了减少硬件单元的停滞（饥饿）。这些队列并没有展示在我们的图表中。

显示控制器中的每个颜色分量有 12 bit，并且具有 BT.2020 的宽色域支持；同时它还支持 HDMI 2.0b 和 HDCP 2.2。在视频处理方面，这个视频控制器支持 SMPTE 2084，这是一个高动态范围（HDR）视频的传输函数。Venkataraman [1816]描述了 Fermi 及其之后的 NVIDIA 架构，是如何拥有一个或者多个复制引擎（copy

engine）的。这里的复制引擎是指可以执行直接内存访问（direct memory access，DMA）传输的内存控制器。DMA 传输发生在 CPU 和 GPU 之间，并且这样的传输过程通常是在这二者中的任何一个上开始的。在传输过程中，启动处理单元（starting processing unit）可以继续进行其他计算。复制引擎可以在 CPU 内存和 GPU 显存之间发起数据的 DMA 传输，并且它们可以独立于 GPU 的其余部分进行执行。因此，当存储数据从 CPU 传输到 GPU 时（反之亦然），GPU 可以继续渲染三角形并执行其他功能。

Pascal 架构还可以为非图形应用程序进行配置，例如训练神经网络或者大规模数据分析。Tesla P100 就是这样的一种配置[1298]，它与 GTX 1080 之间的一些区别包括，Tesla P100 使用了高带宽存储器 2 （high-bandwidth memory 2，HBM2），它具有 4096 bit 的内存总线，可以提供高达 720 GB/s 的总内存带宽。此外，它们还原生支持 16 bit 浮点数，其运算性能是 32 bit 浮点的 2 倍，并且具有更快的双精度处理速度。Tesla P100 的 SM 配置和寄存器堆设置也有所不同[1298]。

GTX 1080 Ti（Titanium）是一种高端配置。它拥有 3584 个 ALU，352 bit 的内存总线，484 GB/s 的总内存带宽，88 ROP 和 224 个纹理单元；相比之下，GTX 1080 的各项对应参数为 2560、256 bit、320 GB/s、64 和 160。GTX 1080 Ti 使用了 6 个 GPC 配置，即它拥有 6 个光栅化引擎，而 GTX 1080 只有 4 个。其中 4 个 GPC 与 GTX 1080 完全相同，而剩下的 2 个 GPC 则要稍微小一些，其中每个 GPC 只包含 4 个 TPC，而不是 5 个。GTX 1080 Ti 芯片由 120 亿个晶体管组成，而 1080 则使用了 72 亿个晶体管。Pascal 架构是十分灵活的，因为它还可以按比例进行缩小。例如：GTX 1070 就是 GTX 1080 减去 1 个 GPC，而 GTX 1050 则由 2 个 GPC 组成，其中每个 GPC 中包含 3 个 SM 单元。

23.10.3 案例研究：AMD GCN Vega

AMD 的 Graphics Core Next （GCN）架构用于 AMD 的一些显卡产品，以及 Xbox One 和 PLAYSTATION 4 中。在这里，我们将介绍 GCN Vega 架构[35]中的通用元素，它们是游戏主机所使用架构的演变版本。

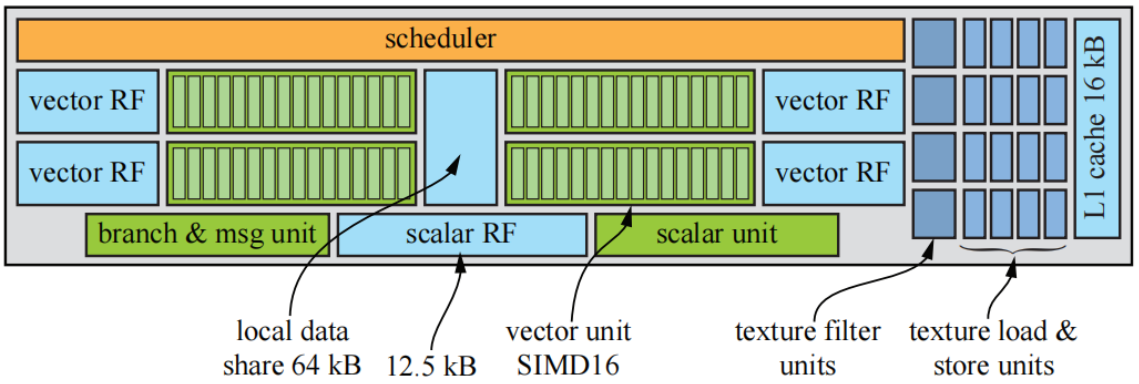


图 23.28：Vega 架构的 GCN 计算单元（CU）。每个向量寄存器堆（vector register file）具有 64 kB 的容量，而标量寄存器堆则具有 12.5 kB 的容量，本地数据共享（local data share）具有 64 kB。请注意，每个 CU 中包含有 4 组 SIMD 单元，每个 SIMD 单元包含 16 个 SIMD lane（淡绿色），支持 32 bit 浮点，它们用于 CU 中的实际计算。[1103, 35]

GCN 架构中的核心构建块是计算单元（compute unit，CU），如图 23.28 所示。一个 CU 中包含四个 SIMD 单元，每个单元包含 16 个 SIMD lane，即 16 个统一 ALU（使用章节 23.2 中的术语）。每个 SIMD 单元可以执行 64 个线程的指令，这被称为一个 wavefront。每个 SIMD 单元可以在一个时钟周期内发出一条单精度的浮点指令。因为架构需要处理每个 SIMD 单元中包含 64 个线程的 wavefront，因此它需要 4 个时钟周期才能完全发布一个 wavefront [1103]。需要注意的是，CU 可以同时运行来自不同内核的代码。由于每个 SIMD 单元拥有 16 个 lane，并且每个时钟周期可以发出一条指令，因此整个 CU 的最大吞吐量为：每个 CU 有 4 个 SIMD 单元 × 每个 SIMD 单元有 16 个 SIMD lane = 每个时钟周期 64 个单精度 FP 操作。与单精度 FP 相比，CU 还可以执行两倍数量的半精度（16 bit 浮点）指令，这对于只需要较少精度的情况时很有用，例如：机器学习和着色器计算。请注意，两个 16 位 FP 值会被打包到一个 32 bit 的 FP 寄存器中。每个 SIMD 单元都有一个大小为 64 kB 的寄存器堆，由于单精度 FP 只使用了 4 个 byte，并且每个 wavefront 包含 64 个线程，因此相当于每个线程拥有 $65536/(4 \times 64) = 256$ 个寄存器。ALU 包含四个硬件流水线阶段[35]。

每个 CU 都有一个指令缓存（图 23.28 中没有展示出来），它最多可以被 4 个 SIMD 单元共享。相关指令会被转发到 SIMD 单元的指令缓冲区（instruction buffer，IB）中。每个 IB 都拥有能够处理 10 个 wavefront 的存储空间，并且可以根据需要在 SIMD 单元内外进行切换，从而隐藏延迟。这意味着一个 CU 可以处理 40 个 wavefront，这相当于 $40 \times 64 = 2560$ 个线程。因此，图 23.28 中的 CU 调度器一次可以处理 2560 个线程，其任务就是将工作分配到 CU 的不同单元。在每个时钟周

期中，当前 CU 上的所有 wavefront 都需要被考虑到指令问题，因为最多只能向每个执行端口发出一条指令。CU 的执行端口包括分支（branch）、标量/向量 ALU、标量/向量内存、局部数据共享、全局数据共享或者导出、特殊指令[32]，即每个执行端口大致映射到 CU 中的一个单元。

其中的标量单元（scalar unit）是一个 64 bit 的 ALU，它也在 SIMD 单元之间进行共享。标量单元拥有自己的标量寄存器堆和标量数据缓存（未显示在图中）。每个 SIMD 单元中的标量 RF 拥有 800 个 32 bit 寄存器，即 $800 \times 4 \times 4 = 12.5$ kB。标量单元的执行与 wavefront 紧密耦合，由于向 SIMD 单元发出完整一条指令需要 4 个时钟周期，因此标量单元只能在每 4 个时钟周期内，为一个特定的 SIMD 单元服务。标量单元会负责处理控制流、指针运算和其他可以在线程之间进行共享的计算。条件分支和无条件分支的指令会从标量单元发出，并在分支单元（branch unit）和消息单元（message unit）中进行执行。每个 SIMD 单元都有一个 48 bit 的程序计数器（program counter，PC），并在 lane 之间进行共享。实际上这就已经足够了，因为它们执行的都是相同的指令。对于已经获取的分支，会对程序计数器进行更新。消息单元中可以发送的消息包括：调试消息、特殊的图形同步消息和 CPU 中断等[1121]。

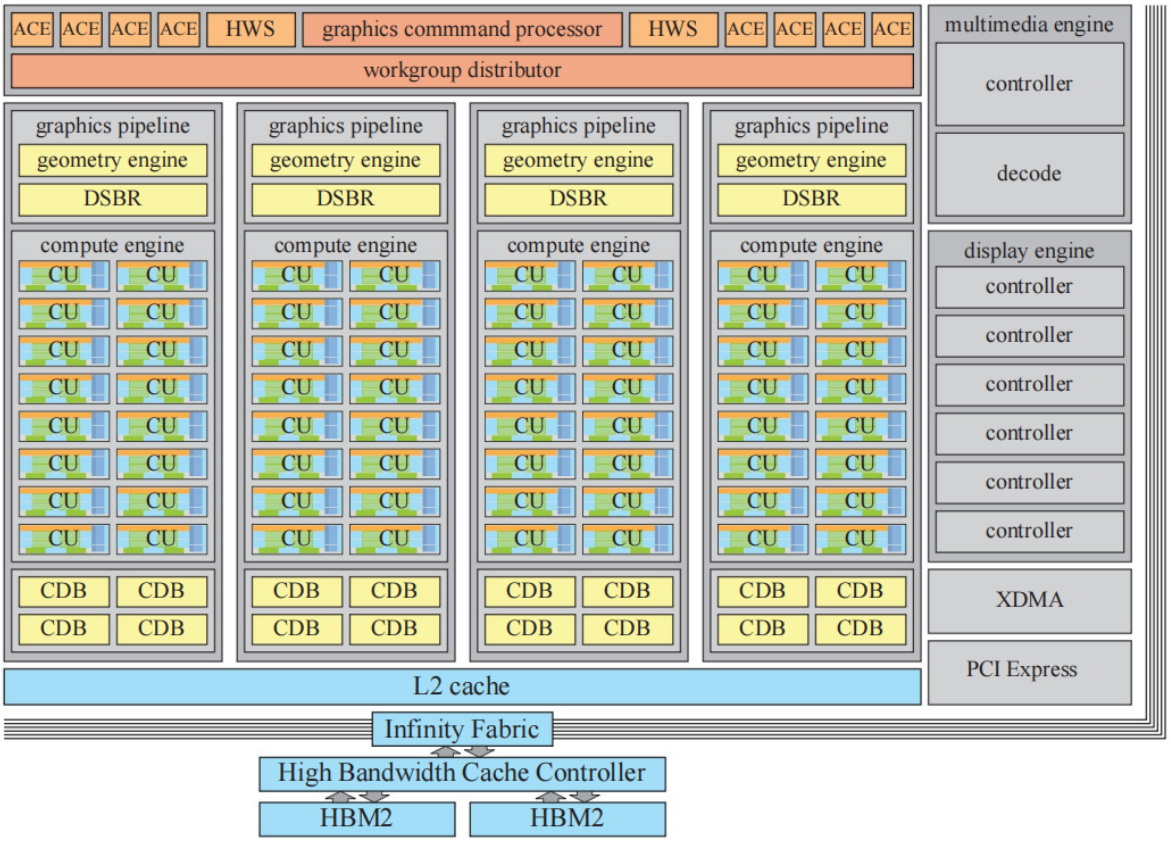


图 23.29：使用 64 个 CU 构建的 Vega 10 GPU。请注意，其中的每个 CU 都包含了如图 23.28 中所展示的硬件单元。[35]

Vega 10 的架构[35]如图 23.29 所示。架构顶部包括一个图形命令处理器（graphics command processor，GPC）、两个硬件调度程序（hardware scheduler，HWS）和 8 个异步计算引擎（asynchronous compute engine，ACE）[32]。GPC 的任务是将图形任务分配到 GPU 的图形管线和计算引擎上。HWS 的缓冲区会在队列中进行工作，使得它们能够尽快分配给 ACE。ACE 的任务是将计算任务调度到计算引擎上。还有两个 DMA 引擎可以处理复制任务（图中没有展示）。GPC、ACE 和 DMA 引擎可以并行地将工作提交给 GPU，这提高了利用率，因为任务可以从不同的队列中交叉进行。可以从任何队列中进行工作分发，而无需等待其他工作完成，这意味着在计算引擎上可以同时执行独立的任务。ACE 可以通过缓存或者内存进行同步。它们可以一起支持任务图（task graph），使得一个 ACE 的任务可以依赖于另一个 ACE 的任务，或者依赖于图形流水线中的任务。建议将较小的计算任务和复制任务，与较重的图形任务交错执行[33]。

如图 23.29 所示，Vega 10 架构中总共包含 4 个图形管线和 4 个计算引擎。其中每个计算引擎包含 16 个 CU，加起来总共是 64 个 CU。其中图形管线包含两个模块，即几何引擎（geometry engine）和流式传输光栅化器（draw-stream binning rasterizer，DSBR）。几何引擎包括几何装配器（geometry assembler）、镶嵌单元（tessellation unit）和顶点装配器（vertex assembler）。此外，还支持一个新的图元着色器（primitive shader），这个图元着色器的目的是实现更加灵活的几何处理和速度更快的图元剔除[35]。DSBR 结合了 sort-middle 和 sort-last 架构的优点，这也是分块缓存的目标（章节 23.10.2）。图像在屏幕空间中被划分为很多 tile，在经过几何处理之后，会将每个图元分配给与之重叠的 tile。在 tile 光栅化的过程中，所有需要的数据（例如 tile 缓冲区）都会保存在 L2 缓存中，这可以提高性能。像素着色可以被自动延迟，即直到处理完 tile 中的所有几何图形之后，再进行像素着色计算。因此，z-prepass 是在底层完成的，并且只会进行一次像素着色。延迟着色可以按需打开或者关闭，例如：对于透明的几何物体，需要关闭延迟着色。

为了处理深度缓冲、模板缓冲和颜色缓冲，GCN 架构还有一个被称为颜色和深度块（color and depth block，CDB）的构建块。除了颜色混合之外，它们还可以处理颜色、深度和模板读写等操作。这里的 CDB 可以使用章节 23.5 中所描述的通用方法，来对颜色缓冲区进行压缩。通过使用增量压缩（delta compression）技术，每个 tile 只会存储一个像素的颜色，其余的颜色值会通过相对于这个像素颜色的差异来进行编码[34, 1238]。为了提高效率，可以根据访问模式来动态选择 tile 的大小。对于一个最初使用 256 byte 进行存储的 tile 而言，最大压缩比为 8：1，即压缩到 32 byte。经过压缩的颜色缓冲区可以在后续 pass 中作为纹理进行使用，在这种情况下，纹理单元会对压缩后的 tile 进行解压，从而进一步节省带宽开销[1716]。

光栅化器在每个时钟周期内，可以对最多四个图元进行光栅化。连接到图形管线和计算引擎的 CDB，在每个时钟周期内可以写入 16 个像素。也就是说，那些小于 16 像素的三角形会降低效率。光栅化器还会处理粗深度测试（HiZ）和分层模板测试。用于 HiZ 的缓冲区被称为 HTILE，它可以由开发人员进行编程，例如：用于向 GPU 提供遮挡信息。

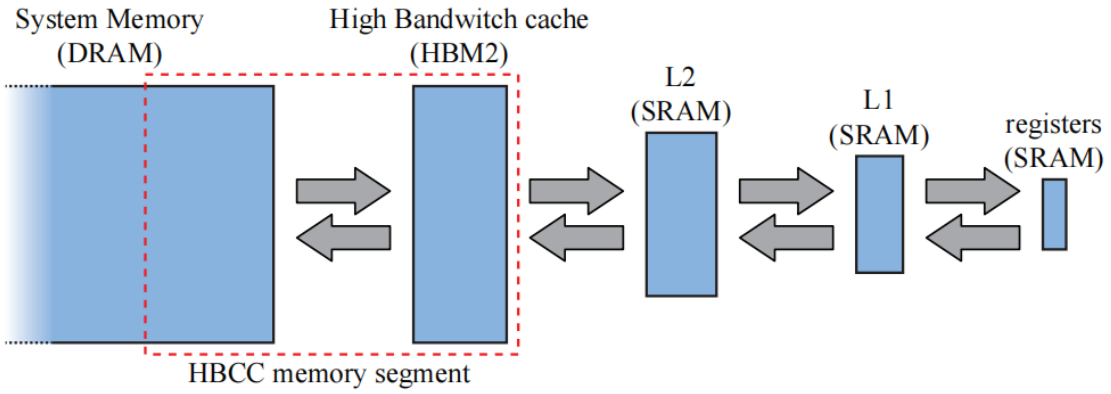


图 23.30：Vega 架构的缓存层次结构。

Vega 的缓存层次结构如图 23.30 所示。在层次结构的顶部（图中最右侧），我们有寄存器，其次是 L1 缓存和 L2 缓存；然后是高带宽存储器 2 （HBM2），它也位于显卡上，最后是位于 CPU 侧的系统内存。Vega 架构的一个新特性是高带宽的缓存控制器（High–Bandwidth Cache Controller，HBCC），如图 23.29 底部所示，它允许显存表现得像最后一级缓存一样。这意味着，如果进行内存访问，而相应的内容不在显存中（即 HBM2）的话，那么 HBCC 将会自动通过 PCIe 总线，获取相应的系统内存页面，并将其放入显存中。因此，显存中最近使用较少的页面可能会被换出。HBM2 和系统内存之间共享的内存池，被称为 HBCC 内存段（HBCC memory segment，HMS）。所有的图形模块都会通过 L2 缓存来访问内存，这与之前的架构有所不同。该架构还支持虚拟内存（章节 19.10.1）。

请注意，所有的片上模块，例如 HBCC、XDMA （CrossFire DMA）、PCI express、显示引擎和多媒体引擎，都会通过一个名为 Infinity Fabric （IF）的互连（interconnect）进行通信。AMD 的 CPU 也可以连接到这个 IF，IF 还可以连接不同芯片上的模块。这个 IF 也是一致的，这意味着所有块所能看到的内存内容都是相同的。

Vega 架构芯片的基础时钟频率为 1677 MHz，其峰值计算能力为：

$$\underbrace{2}_{\text{FMA}} \cdot \underbrace{4096}_{\text{num SPs}} \cdot \underbrace{1677}_{\text{clock freq}} = 13,737,984 \text{ MFLOPS} \approx 13.7 \text{ TFLOPS} \quad (23.17)$$

其中 FMA 和 TFLOPS 的计算，与方程 23.16 中的计算相匹配。Vega 架构也是灵活且可扩展的，因此可以有很多种的配置方式。

23.11 光线追踪架构

本小节我们将会简要介绍光线追踪硬件。我们不会并列出最近关于这个主题的所有参考资料，而是提供一组建议读者遵循的指南和链接。有关该领域的研究是由 Schmittler 等人[1571]在 2002 年开始的，他们的重点是遍历和求交，并使用固定功能的单元来计算着色。这项工作后来被 Woop 等人[1905]跟进，他们提出了一种具有可编程着色器的架构。

在过去几年中，对这一主题的商业兴趣大大增加。这一点可以从 Imagination Technologies [1158]、LG Electronics [1256]和 Samsung [1013]等公司提出的用于实时光线追踪的硬件架构中看出。然而，截止到本文撰写时，只有 Imagination Technologies 发布了一款商业产品。

在这些架构中有几个共同的特点。首先，他们通常会使用基于 AABB 的 BVH 结构。其次，它们倾向于通过降低射线与 box 相交测试的精度，来降低硬件复杂度（章节 22.7）。最后，他们使用可编程内核来支持可编程着色，这在今天或多或少算是一种需求。例如：Imagination 公司对他们传统的芯片设计进行了扩展，增加了一个光线追踪单元，可以利用着色器核心来进行着色处理。这个光线追踪单元由一个光线求交处理器（ray intersection processor）与一个一致性引擎（coherency engine）组成，其中的一致性引擎[1158]，会将具有相似属性的光线聚集在一起，并将它们一起进行处理，从而利用局部性来实现更快的光线追踪。在 Imagination Technologies 公司的架构中，还包含一个用于构建 BVH 的专用单元。

对该领域的研究还将继续探索几个子领域，其中包括通过降低精度来实现高效遍历[1807]，BVH 的压缩表示[1045]，以及能源效率（energy efficiency）[929]等。毫无疑问，还有更多的研究要进行。

补充阅读和资源

Akeley 和 Hanrahan [20]，以及 Hwu 和 Kirk [793]有关计算机图形架构的课程讲义中提供了大量的参考资源。Kirk 和 Hwu [903]的书，也是一个关于在 GPU 上使用 CUDA 编程的良好信息资源。每年的高性能图形（High-Performance Graphics）和 SIGGRAPH 会议论文集，是介绍新架构特征的良好来源。Giesen 图形管线之旅

（trip down the graphics pipeline），对于任何想要了解更多关于 GPU 细节的人而言，都是一个很棒的在线资源[530]。我们还建议感兴趣的读者参考 Hennessy 和 Patterson 的书[715]，从而了解有关内存系统的详细信息。有关移动渲染的信息分散在许多来源中，其中值得注意的是，在《GPU Pro 5》中，有 7 篇关于移动渲染技术的文章。

Chapter 24 The Future 未来

> Billy Zelsnack——“Pretty soon, computers will be fast.”

> 比利·泽尔纳克——“很快，计算机就会变得很快。”（引擎程序员，曾参与开发《雷神之锤》、《毁灭公爵》，其创建的游戏公司 Rebel Boat Rocker 与后来的 Gearbox Software 有关）

> Niels Bohr or Yogi Berra——“Prediction is difficult, especially of the future.”

> 尼尔斯·玻尔 or 尤吉·贝拉——“预测是困难的，尤其是预测未来。”（丹麦理论物理学家，哥本哈根学派创始人；1885—1962）（前 MLB 捕手、教练；Yogi 是少年友人起的外号，由于盘腿坐姿类似印度瑜伽士，1925—2015）

> Alan Kay——“The best way to predict the future is to create it.”

> 艾伦·凯——“预测未来的最好方法就是创造未来。”（美国计算机科技家，2003 年图灵奖得主；1940—）

在这个展望未来的章节中，由两部分组成：其他事项和你。这一章就是关于这两方面的内容。首先，我们将会做出一些预测，其中的一些预测甚至可能会成为现实。更重要的是第二部分，关于下一步你可以前往哪里。你可以将本章节视为一个扩展的“补充阅读和资源”，就像是之前每章末尾的那样；但是我们也对从这里开始的方法进行了讨论，包括一般的信息来源、会议、代码等。首先我们先来看一个图片，如图 24.1 所示。

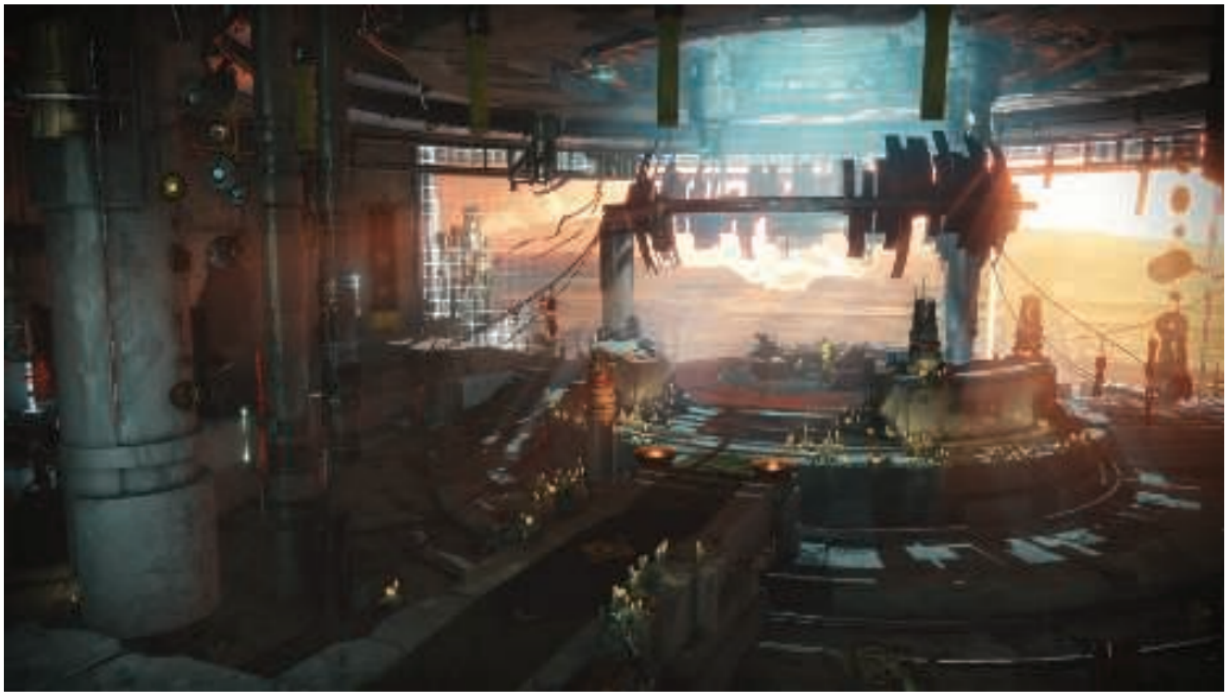


图 24.1：通过游戏《命运 2》来瞥见未来。

24.1 其他事项

图形学有助于游戏销售，而游戏则有助于芯片销售。从芯片制造商的营销角度来看，实时渲染的最佳特性之一，就是图形渲染将会消耗大量的处理能力和其他资源。与硬件相关的特性，例如帧率、分辨率和颜色深度等也会在一定程度上不断增长，从而进一步增加处理负载。最低 90 FPS 固定帧率是虚拟现实应用程序的标准，4k 显示器目前已经对图形系统保持速度的能力提出了考验[1885]。

在场景中模拟光线效果的复杂任务，本身就需要大量的计算能力。在场景中添加更多的物体或者更多的光源，显然会使得渲染计算变得更加昂贵。物体的类型（包括固体和体积，例如雾）、这些物体表面的描绘方式、以及所使用的光源类型，也是一些可以增加复杂性的因素。如果我们能够获取更多的样本、计算更加精确的方程、或者仅仅是能够使用更多的内存，许多算法的质量都会得到很大的提升。这种不断增加的复杂性，使得图形处理能力几乎成了一个无底洞。

为了解决长期的性能问题，乐观主义者喜欢求助于摩尔定律（Moore's Law）。摩尔定律的观测结果表明：每 1.5 年性能增加 2 倍；或者更加实用的是，每 5 年性能增加 10 倍[1663]。但是处理器的速度通常并不是性能瓶颈，而且随着时间的推移，性能瓶颈发生在处理器上的概率会越来越小。带宽才是真正可能成为性能瓶颈的地方，因为其性能每 10 年才会增加 10 倍，而不是像处理器那样，每 5 年增加 10 倍[1332]。

来自电影行业中的图形算法，经常可以应用到实时渲染中，因为这些领域的共同目标都是生成逼真的图像。让我们看看电影行业中的做法，我们能够看到一些统计数据，例如 2016 年的电影《奇幻森林（the Jungle Book）》，某一帧的场景中可能包含数百万根头发，每帧的渲染时间为 30 到 40 个小时[1960]。由于 GPU 是专为实时渲染而构建的，因此相比 CPU 具有明显的处理优势，但从 $1/(40 \times 60 \times 60) = 0.00000694$ FPS 到 60 FPS，也大约有 7 个数量级。

本章节一开始的时候，我们承诺会进行一些预测，其中“更快更灵活（faster and more flexible）”就是最简单的一个。就 GPU 架构而言，一种可能性是 z-buffer 三角形光栅化管线将继续统治实时渲染。除了最简单的游戏之外，所有的游戏都会使用 GPU 来进行渲染。即使明天一醒来，就出现了某种令人难以置信的技术可以取代当前的管线，即使这种技术的速度要快一百倍；并且这种技术只需要下载一个系统补丁就可以运行，业界仍可能需要数年时间，才能迁移到这种新技术上。其中一个问题是在于，这种新方法是否可以使用与现有方法完全相同的 API，如果不行的话，那么想要采纳这种方法就仍然需要一段时间。一款复杂的游戏需要花费数千万美元甚至更多的开发成本，并且经常需要数年时间才能完成。游戏的目标平台一般都是在开发过程的早期选定的，这就决定了所使用算法和着色器的复杂程度，以及美术资产的大小和复杂程度。除了这些因素之外，还需要考虑到编写或者生产这些元素所需要的开发工具，并且用户还需要对这些工具的使用十分精通。即使真的有奇迹发生，当前光栅化流水线背后的惯性，也仍然会给它带来若干年的生命。

变化仍在发生（change still happens）。实际上，简单的“一个光栅化器统治一切（one rasterizer to rule them all）”的想法已经开始消退。在本书中，我们讨论了计算着色器所能承担的各种任务，证明了光栅化并不是 GPU 所能提供的唯一服务。如果某个新技术真的很有吸引力，那么从游戏公司到商业引擎，再到相关的内容创造工具，都会发生工作流的重新调整。

那么，从长远来看情况如何呢？用于渲染三角形、访问纹理和混合结果样本的 GPU 专用固定功能硬件，仍然可以显著提升性能。但是移动设备的需求改变了这个情况，因为功耗成为了与原始性能一样重要的因素。然而，基础管线的“即发即弃（fire-and-forget）”概念，即我们将一个三角形只发送到管线中一次，并使用这些信息来完成该帧的渲染处理，这种概念并不是现代渲染引擎中所使用的模型。变换、扫描、着色和混合的基本流水线模型，已经发展得几乎面目全非了。GPU 已经成为了一个基于流处理器的大型集群，我们可以随心所欲地使用它。

图形 API 和 GPU 发生了共同进化以适应这一现实，这里的咒语（mantra）是“灵活性（flexibility）”。图形渲染方法由研究人员进行探索，然后由开发人员在现有硬件

上进行实现，从而确定他们所希望的功能是可以实现的。独立硬件供应商可以利用这些发现，以及他们自己的研究来开发通用功能，从而形成一个良性循环。对于任何单一算法进行优化都是徒劳的（ fool's errand），而创建新的、更加灵活的方式来访问和处理 GPU 上的数据则不是。

考虑到这一点，我们将射线与物体相交的功能，视为一种具有多种用途的通用工具。我们知道，使用路径追踪的完全无偏采样，最终将会产生正确的、真实的图像，并达到场景描述的极限。这里的“最终”才是问题所在，正如我们在章节 11.7 中所讨论的，路径追踪作为一种可行的算法，目前正面临着严峻的挑战。其主要问题在于，想要得到无噪声的结果，并且保证在动画过程中不会发生闪烁，所需的样本数量实在太大。但是，路径追踪的纯粹性和简单性使其极具吸引力。在当前的交互式渲染状态下，其中所使用的许多技术，都是专门为特定情况量身定制的，也就是说，我们想要实现某种效果，可能会为这个效果专门设计一种算法来进行处理，而现在只需要一种算法就可以完成所有的工作。电影工作室当然已经意识到了这一点，因为在过去的十年中，他们已经完全转向了光线追踪和路径追踪方法。这样做可以使得他们只对一组光线传输的几何操作进行优化即可。

实时渲染（包括所有的渲染方式），最终都是关于采样和过滤的问题。除了提高光线发射和求交的效率之外，路径追踪还可以从更加智能的采样和过滤策略中受益。事实上，无论营销文学（marketing literature）如何，几乎所有的离线路径追踪渲染器都是有偏的（biased）[1276]。对发射采样射线的位置做出合理的假设，能够大幅提高性能表现。路径追踪可以从中受益的另一个领域是智能过滤（字面意思的智能）。深度学习目前是一个白热化的研究领域和开发领域，由于在 2012 年取得了令人印象深刻的进展，当它的表现远远超过了手动调整的图像识别算法时[349]，人们重新燃起了对其的兴趣。使用神经网络进行降噪[95, 200, 247]和抗锯齿[1534]是令人着迷的发展，如图 24.2 所示。我们已经看到，使用神经网络进行渲染相关任务的研究论文的数量大幅增加，更不用说建模和动画了。

图 24.2：基于神经网络的图像重建。左边是通过路径追踪生成的噪声图像。在右边的图像中，使用 GPU 加速的降噪算法，可以以交互式的速度来对图像进行处理。[200]

最早可以追溯到 1987 年 AT&T 的 Pixel Machine，它可以使用交互式的光线追踪，来在小场景、低分辨率、较少光源的情况下，实现尖锐的反射、折射和阴影。微软在 DirectX API 中添加了光线追踪功能，被称为 DXR，它简化了光线的发射过程，并激励硬件供应商增加对光线求交的硬件支持。光线发射（ray shooting），通过降噪技术或者其他滤波技术进行增强，将首先成为提高各种元素（例如阴影或者反射）渲染质量的一种技术。它将与许多其他算法进行竞争，每个渲染引擎会根据速度、质量和易用性等因素做出选择，如图 24.3 所示。

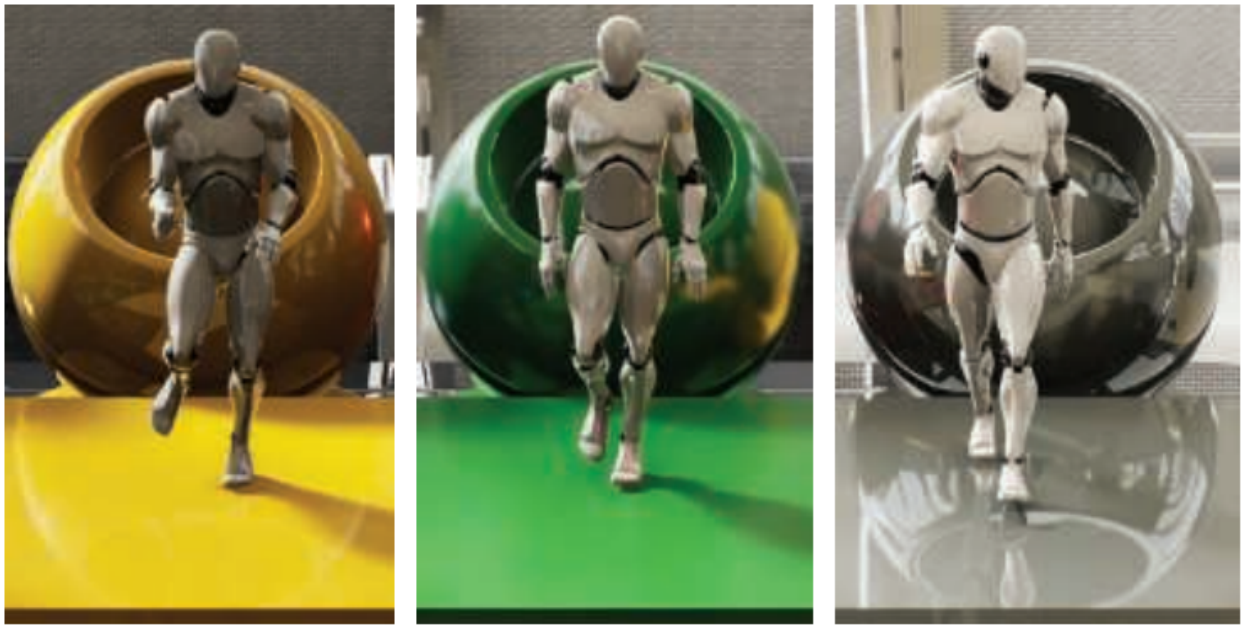


图 24.3：这些图像以交互速率进行渲染，每个像素上有 2 条反射光线；1 条用于屏幕位置的阴影射线；2 条用于反弹光线；以及 2 条环境光遮蔽光线，每个像素总计 7 条射线。对阴影和反射使用了降噪滤波器。

截至到本文撰写时，作为基本操作的分层光线发射（hierarchical ray shooting）还不是任何主流商业 GPU 的明确部分。我们把 PowerVR 的 Wizard GPU [1158]作为一个好兆头，因为已经有一家移动设备公司，正在考虑对分层场景描述提供光线求交的硬件支持。直接支持光线发射的新 GPU，将会改变效率方程（equations of efficiency），并且可能构建一个良性循环，使得各种渲染效果的定制化和专业化程度都大幅降低。一种方法是对眼睛光线（eye ray）进行光栅化，并使用光线追踪或者计算着色器来实现其他的一切效果，并且这种方法已经在各种 DXR demo [1, 47, 745]中进行了使用。通过改进的降噪算法，更快的光线追踪 GPU，以及先前研究的重新应用，或者使用一些新的研究，我们预计很快就会看到相当于 10 倍的性能提升。

我们希望 DXR 能够在其他方面为开发人员和研究人员带来好处。对于游戏而言，通过投射光线的烘焙系统现在可以在 GPU 上进行运行，并使用与交互式渲染器相似或者相同的着色器，从而提高性能。可以更加容易地生成 ground truth 图像，使其能够更加容易地用于测试，甚至是对算法进行自动调整。架构改变的相关想法，允许更加灵活地生成 GPU 任务（例如：创建着色器工作），这似乎是一个强大的想法，可能会有其他方面的应用。

GPU 的发展当然还有着其他令人着迷的可能性。另一种理想化的世界观，是世界中的所有物质都是体素化的。如章节 13.10 中所述，这种表示对于光线传输和光照模拟来说有许多优点。但是由于需要进行大量的数据存储，以及较难处理场景中的动态物体，这些因素使得完全切换到体素表示的可能性极小。尽管如此，我们相信体素可能会得到更多的关注，因为体素技术会在很广泛的领域中进行使用，包括高质量的体积效果、3D 打印和无约束的物体修改等（例如：Minecraft）。当然，考虑到自动驾驶汽车系统、激光雷达（LIDAR）和其他传感器会产生的大量的此类数据，因此相关的表示形式（点云）将会在未来几年中成为更多研究中的一部分。符号距离场（SDF）是另一种有趣的场景描述方法，与体素类似，SDF 可以对场景进行无限制的修改，也可以对光线跟踪进行加速。

有时，某些特定应用程序的独特约束，允许其开发人员“打破常规（break the mold）”，并且使用一些以前被认为是外来的或者是不可行的技术。诸如 Media Molecule（译者注：索尼的一个第一方工作室）的《Dreams》和 Second Order 的

《粘土之书（Claybook）》（如图 24.4 所示）这样的游戏，这些游戏都使用了非传统的渲染算法，能够让我们对这样一个可能的渲染未来有一个有趣的了解。

图 24.4：《粘土之书》是一款基于物理原理的益智游戏，其中的黏土世界可以由用户进行自由塑造。粘土世界使用符号距离场进行建模，并使用光线追踪来进行渲染，其中包括主光线（primary ray），光线追踪阴影和 AO 等效果。并在 GPU 上对固体和液体效果进行了物理模拟。

虚拟现实和混合现实值得一提。当 VR 运作良好时，它的效果是惊人的。而混合现实则展示了合成内容与现实世界相融合的迷人场景。每个人都想要一款能兼顾两者的轻型眼镜，但是短期内这种设备也可能会进入“个人喷气背包、水下城市”等无法实现的类别，但是谁又能知道呢？考虑到这些努力背后的大量研究和开发工作[1187]，可能会有一些改变世界的技术突破。

24.2 你

那么，当你的孩子和你孩子的孩子在等待奇点（The Singularity）到来的同时，你在做什么呢？当然是写程序：发现新的算法，创建应用程序，或者做其他你喜欢的事情。几十年前，一台机器图形硬件的价格要比一辆豪华汽车还贵；但是现在，它被内置到几乎所有具有 CPU 的设备中，而且这些设备还可以放在你的手中。图形入侵（graphics hacking）是廉价且主流的。在本小节中，我们将介绍在更多有用的资源，这些资源可以帮助你学习更多有关实时渲染领域中的知识和技术。

我们这本书并不是凭空存在的，它利用了大量已有的信息来源。如果你对其中的某个算法感兴趣，可以找到相关的原始出版物。我们的网站上有一个页面，其中包含了我们引用的所有文章，你可以在那里找到相应的资源链接（如果可以的话）。

大多数研究文章都可以使用谷歌学术（Google Scholar）或者作者的个人网站中找到，如果其他方法都失败了，那么还可以向作者索要一份文章副本，几乎每个人都喜欢让他们的工作被阅读和欣赏。

如果有些论文不是免费的话，那么诸如 ACM 数字图书馆（ACM Digital Library）等服务，包含了大量可用的文章。如果您是 SIGGRAPH 的成员，那么您可以自动免费访问他们的许多图形论文和演讲。还有一些期刊会发表相关的技术文章，例如 ACM Transactions on Graphics（现在包括了 SIGGRAPH 的论文集）， The Journal of Computer Graphics Techniques（这是开放获取的），IEEE Transactions on Visualization and Computer Graphics，Computer Graphics Forum，IEEE Computer Graphics and Applications 等。最后，还有一些专业博客中存在有很好的信息，Twitter 上的图形开发人员和研究人员，经常会指出很棒的新资源。

学习他人和结识他人的最快方法之一就是参加会议，很有可能另一个人正在做你感兴趣或者可能感兴趣的事情。如果你资金紧张的话，那么请联系相关的会议组织者，询问志愿者机会或者奖学金。SIGGRAPH 和 SIGGRAPH Asia 年会是新思想的主要场所，但并非是唯一场所。还有一些其他的技术集会，例如 Eurographics 会议和 Eurographics 图形学渲染研讨会（Eurographics Symposium on Rendering，EGSR），交互式 3D 图形和游戏研讨会（Symposium on Interactive 3D Graphics and Games，I3D），以及高性能图形系统（High Performance Graphics，HPG）论坛展示并发布了大量与实时渲染相关的资料。还有一些专门针对开发者的会议，例如 GDC （Game Developers Conference，游戏开发者大会）。当你在排队或者参加活动的时候，记得向陌生人打招呼。特别是在 SIGGRAPH 上，请留意您感兴趣领域的同类聚会（birds of a feather，BOF）。与人们面对面地交流思想是既有益又充满活力的。

还有一些与交互式渲染相关的电子资源。特别值得注意的是，《Graphics Codex》是一个高质量的纯电子参考资料，并且具有不断更新的优点。沉浸式线性代数（immersive linear algebra）网站中的部分内容，由本书的合著者所创建，其中包括了交互式的演示内容，对这个主题的学习有所帮助。Shirley 有一本关于光线追踪的优秀的 Kindle 短篇书籍（《Ray Tracing in One Weekend》）。我们期待着这类资源变得廉价和更容易获取。

印刷书籍仍有一席之地。除了通用论文和特定领域的书籍，还有一些经过编辑的文章集合，其中包括了大量的研究信息和发展信息，其中的许多内容在本书中进行了引用。最近的例子是《GPU Pro》和《GPU Zen》书籍。一些较旧的书籍，例如《Game Programming Gems》、《GPU Gems》（免费在线）和《ShaderX》系列中仍然包含一些相关的文章，因为其中一些算法的原理并不会过时。所有的这些书籍，都允许游戏开发者在不撰写正式会议论文的前提下，就可以展示他们的方法。这样的文章合集还允许学者对他们工作的技术细节进行讨论，这些细节可能并不适合出现在研究论文中。对于专业的开发人员而言，从一篇文章中阅读一些实现细节所节省的一个小时，可能要比阅读整本书的成本更有价值。如果你等不及实体书被送到你的手里，可以使用亚马逊上的“Look Inside”功能，或者是在谷歌图书上搜索文本，可能会得到一个能够让我们开始（get you started）的摘录。

说完了，做完了，就需要开始写代码了。随着 GitHub、Bitbucket 和类似仓库的兴起，我们拥有一个丰富的代码仓库可以进行利用。困难的部分是知道哪些部分不属于 Sturgeon 定律（Sturgeon’s Law：任何事物中，其中 90% 都是垃圾）。像虚幻引擎这样的产品已经开放了它们的源代码，它是一个不可思议的资源。ACM 现在鼓励任何已发表的技术文章公布各自的源代码。你所尊敬的作者有时候也会提供他们的源代码代码，尝试进行搜索。

Shadertoy 是一个特别值得注意的网站，它经常在像素着色器中使用光线步进来展示各种技术。虽然其中的许多程序首先是赏心悦目的，但是该网站中包含了许多具有教育意义的 demo，并且所有的代码都是可见的，并且可以在浏览器中进行运行。另一个基于浏览器的 demo 来源是 three.js 仓库和相关网站。“Three.js”是对 WebGL 的封装，它很容易进行实验，因为只需要几行代码就可以生成渲染结果。并且只需要点击一个超链接，就可以在网上发布 demo，任何人都可以进行运行和分析，这对于教育用途和分享想法而言是非常棒的。本书的作者之一基于 three.js，为 Udacity（一个在线教育平台）创建了一门图形入门课程。

我们再次向您推荐我们的在线网站 realtimerendering.com。在那里你会发现许多其他有用的资源，例如推荐书目和新书列表（其中还包含一些免费的高质量内容），以及一些链接，它们指向具有价值的博客、研究网站、课程演示和许多其他的信息来源。去寻找快乐吧！

我们最后的忠告是去学习，去实践。实时计算机图形领域在不断地发展，新的想法和新的特性不断被发明和集成。你可以参与其中。虽然所使用的技术种类繁多，看起来也令人生畏，但是你不需要实现一长串的 buzzwords-dujour（字面意思是今日流行语）就能得到好的结果。根据应用程序的约束条件以及所需要的视觉风格，巧妙地组

合少量技术就可以得到与众不同的视觉效果。在 GitHub 上分享你的结果，GitHub 还可用来托管博客。去参与吧！

这个领域最好的地方之一，就是它每隔几年就会自我改造一次（reinvents itself）。计算机架构也在不断变化和改进。几年前行不通的方案，现在可能就可以实现了。每一款新的 GPU 都会带来不同的功能、速度和内存组合。具体什么是高效的，哪部分会成为性能瓶颈，都在不断变化和发展。即使是那些看起来古老而成熟的部分，也值得重新审视。有有一种说法是，创造是一种扭曲、打破和融合其他想法的过程，而不是从无到有的过程。

44 年前（2018–1974 年），Sutherland、Sproull 和 Schumacker 发表了计算机图形学领域中的一个里程碑论文：“A Characterization of Ten Hidden–Surface Algorithms”。在他们长达 55 页的论文中，进行了令人难以置信的全面对比分析。其中有一个被描述为“极其昂贵”的算法，这种暴力技术甚至连研究人员的名字都没有，只在附录中被提到过，它现在被称为 z–buffer。公平地说，z–buffer 的发明者是 Ed Catmull，Sutherland 是他的导师，Ed Catmull 对 z–buffer 概念进行讨论的论文，在几个月之后才发表。

在 1974 年的这篇论文中，讨论并对比了 10 种隐藏表面的算法，但是实际上最终胜出的是第 11 种，因为它在硬件上易于实现，并且随着硬件技术的进步，存储密度在不断上升，成本也随之下降。但是 Sutherland 等人所做的“十大算法”调查，在当时是完全有效的。随着条件的不读变化，所使用的算法也会发生改变。未来几年到底会发生什么变化，是一件令人兴奋的事情。在将来的某一天，当我们对目前渲染技术时代进行回顾的时候，会有什么感觉呢？没有人知道，每个人都可能会对未来产生重大影响。没有唯一的未来，没有必然会发生的过程。是你创造了它。

接下来你想做什么呢?

Chapter 25 Collision Detection 碰撞检测

> George Santayana——“To knock a thing down, especially if it is cocked at an arrogant angle, is a deep delight to the blood.”

> 乔治·桑塔亚纳——“把一件东西撞倒，尤其是以傲慢的角度把它撞倒，是一种血液深处的喜悦。”（西班牙著名自然主义哲学家、美学家；1863—1952）

> 译者注：本章节中引用的参考文献编号是单独列出的，注意不要与第 1 章–第 24 章的参考文献相混淆。

碰撞检测（collision detection，CD）是计算机图形学中一个基本而重要的组成部分。CD 在虚拟制造、CAD/CAM、计算机动画、基于物理的建模、游戏、飞行模拟器、车辆模拟器、机器人、路径和运动规划（公差验证）、装配以及几乎所有的虚拟现实模拟领域中，都发挥着重要作用。由于其广泛的用途，因此 CD 一直是、并且仍然是一个被广泛研究的主题。

碰撞检测是通常被称为碰撞处理（collision handling）中的一部分，它可以被分为三个主要部分：碰撞检测（collision detection）、碰撞确定（collision determination）和碰撞响应（collision response）。碰撞检测的结果是一个布尔值，它代表了两个或者多个物体是否发生碰撞；而碰撞确定则会找到物体之间的实际交点；最后，碰撞响应决定了两个物体在发生碰撞的时候，应当采取什么样的行动。

考虑一个由弹簧和齿轮组成的老式时钟。假设这个时钟在计算机中被表示为一个详细的三维模型，我们现在要使用碰撞检测来模拟时钟的运动，当检测到碰撞并产生响应的时候，将弹簧上紧并模拟时钟的运动。这样的一个系统，可能需要在数千对物体之间进行碰撞检测。另一个挑战是对蚁丘中所有的蚂蚁和松针进行模拟，直接对所有可能的碰撞对进行暴力搜索将是非常低效的，在这种情况下根本不切实际。图 25.1 展示了一个复杂碰撞检测的例子。

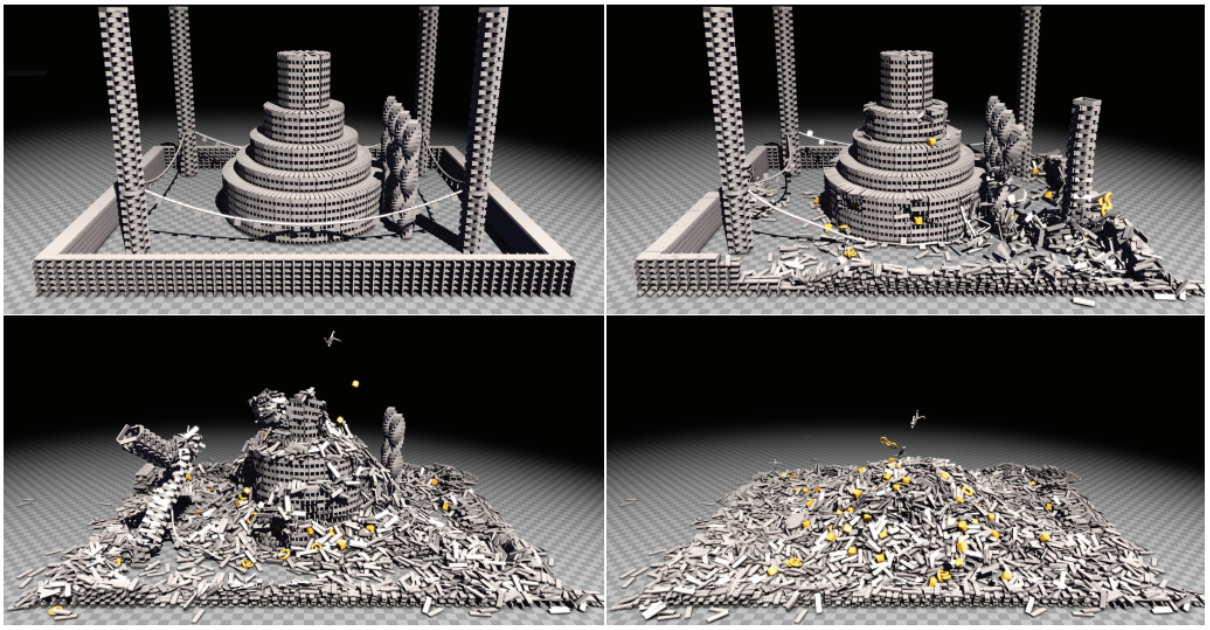


图 25.1：刚体碰撞检测会使用数以万计的微小几何物体，主要是 box，鸟居和少量的布娃娃。

为了应对大型场景中的碰撞检测，我们通常会将一个碰撞检测系统划分为 3 个阶段。宽阶段 CD（broad phase CD）将会在每个物体级别上进行工作，并找到与 BV 重叠的物体对（章节 25.1）。中阶段 CD（mid phase CD）将继续工作，并检测两个物体之间可能重叠的部分（章节 25.2）。最后，窄阶段 CD（narrow phase CD）会在图元叶子或者物体的凸面部分上进行工作（章节 25.3）。这三个阶段可以在大型 CD 系统中一起使用。

在章节 25.4 中，我们将讨论一些在某些场景下非常有用且简单快速的碰撞检测技术。其主要思想是使用一组线段来近似一个复杂的物体，然后测试这些线段是否会与环境中的图元相交，这种技术有时也会被用于游戏中。在章节 25.5 中，我们将介绍另外一种近似方法，它使用 BSP 树来表示环境，从而可以使用圆柱体来代表人物角色。然而，并非所有物体都可以被近似为线段或者圆柱体，有些应用可能还需要更加精确的测试。

限时碰撞检测（time-critical collision detection）是指一种能够在常数时间内进行近似碰撞检测的技术，我们将在章节 25.6 中进行讨论。然后依次讨论可变形模型、连续 CD、碰撞响应等内容，最后在章节 25.10 中讨论如何处理粒子之间的碰撞。

必须指出的是，对于 CD 性能的评估是很困难的，因为碰撞检测算法对于碰撞场景十分敏感，并且没有任何一种算法能够在所有情况下都表现最好[33]。另外请注意，CD 通常会在 CPU 端执行，但是由于 GPU 的通用性，所有这些算法也都可以在

GPU 上进行执行。一般来说，为了使得性能最大化，算法需要与特定架构的内存系统和特性集合相适应。

最后，我们介绍了一些动态相交测试，例如确定两个移动的物体，在给定的时间内是否会发生碰撞，这实际上是对问题又增加了另一个维度：时间。在最后一个小节中，我们提供了各种基本组合的常用测试，以及用于派生更加复杂交互的工具。

25.1 宽阶段碰撞检测

在本小节中，我们将介绍几种宽阶段 CD 算法。不过，首先我们在图 25.2 中展示了 CD 的宽阶段（broad phase）、中阶段（mid phase）和窄阶段（narrow phase）过程，这是一个两级 CD 系统[16]，其中最后一个阶段被划分为中阶段和窄阶段。这个 CD 系统的目标是具有多个运动物体的大型环境，该系统的宽阶段将会报告环境中所有物体之间的潜在碰撞。接下来，中阶段会处理成对的物体，并试图找到任何可能重叠的图元叶子或者凸面部分。窄阶段会计算图元与图元之间的交点，或者使用距离查询（distance query）来确保物体之间不会相互穿透。最后，计算碰撞响应（章节 25.9），碰撞响应的计算结果会被馈送到仿真子系统中，这个子系统可以计算每个物体的最终变换。

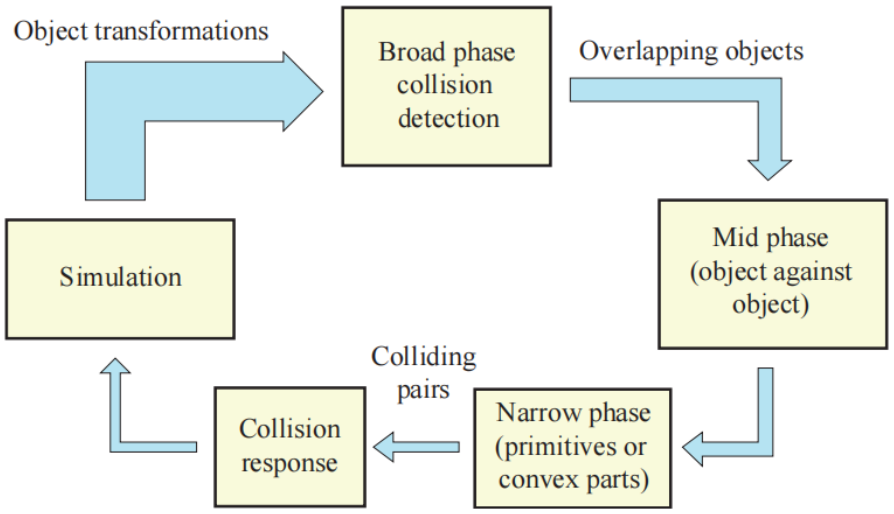


图 25.2：一个碰撞检测系统，它使用某种模拟来进行物体变换。会对场景中的所有物体都进行宽阶段 CD 处理，从而快速找到包围体（BV）相互重叠的物体对；接下来，中阶段 CD 将会继续在物体对上进行工作，从而找到重叠的图元叶子或者凸面部分。最后，CD 系统会在窄阶段中执行最低层次上的操作，在这个阶段中，可以计算图元与图元之间的交点，或者使用距离查询，然后将结果提供给碰撞响应。

由于一个场景中可能会包含数千个移动物体，因此一个好的 CD 系统必须能够很好地处理这种情况。如果场景包含 n 个移动物体和 m 个静态物体，那么可以一种比较简单的碰撞方法，每帧将会执行的次数如下：

$$nm + \begin{pmatrix} n \\ 2 \end{pmatrix} = nm + n(n-1)/2 \tag{25.1}$$

其中的第一项 mn，对应了动态物体与静态物体之间的测试数量；第二项 $\begin{pmatrix} n \\ 2 \end{pmatrix}$ 对应了动态物体与动态物体之间的测试数量。随着 m 和 n 的数量增加，这种简单的方法将会很快变得十分昂贵。这种情况需要一种更加智能的方法，这就是本小节的主题。

每个阶段的目标都是将传递给后续阶段的工作量进行最小化，也就是说，我们希望能够最小化提供给下一阶段的物体对、图元对、凸面对等。在第一个阶段中，测试是在物体级别上完成的，因此它通常也被称为宽阶段 CD。

这一阶段的大多数算法，首先会将每个物体包围在一个 BV 中，然后使用一些技术来找到所有重叠的 BV–BV 对。一种简单的方法是为每个物体都使用轴对齐包围盒（AABB）。对于进行刚体运动的物体，为了避免重新计算该物体的 AABB，因此会将 AABB 调整为一个足够大的固定立方体（fixed cube），从而容纳在任何方向上的物体，并使用这个固定立方体来快速确定哪些物体的 BV 对是不相交的。有时候使用动态调整大小的 AABB 可能会更好，这些 AABB 可以进行快速的重新计算，例如：OBB、球体和胶囊体等。

可以使用球体来代替固定立方体，这是合理的，因为球体是一个可以在任何方向上包围物体的完美 BV。也可以使用顶点贴图（apex map，章节 22.13.4）[53]。接下来，我们将介绍三种宽阶段 CD 算法，即扫描剪枝算法（sweep–and–prune）、网格算法和 BVH 算法。另一种完全不同的方法，是使用章节 19.1.3 中所介绍的松散八叉树结构（loose octree structure）。

25.1.1 扫描剪枝算法

我们假设每个物体都有一个封闭的 AABB。在扫描剪枝（sweep–and–prune，SAP）技术[3, 62, 93]中，我们利用了在典型应用中存在的时间一致性（temporal coherence）。这里的时间一致性意味着物体的位置和方向，在帧与帧之间只会发生相对较小的（如果有的话）变化，因此也被称为帧与帧的一致性（frame–to–frame coherence）。

Lin [62]指出，找到三维重叠包围盒的问题可以在 $O(n \log^2 n + k)$ 时间内求解（其中 k 是成对重叠的数量），但是还可以通过利用一致性来进行改进，从而可以减少到 $O(n + k)$ 。然而，这需要假设动画具有相当数量的时间一致性。

如果两个 AABB 重叠，那么每个主轴方向上的三个一维区间（由 AABB 的起点和终点组成）也必须重叠。在这里，我们将介绍当时间一致性很高的时候，如何高效检测多个一维区间中的所有重叠。在给定该解的情况下，对三个主轴分别采用一维算法，即可求解三维 AABB 的重叠问题。

假设 n 个区间（沿特定轴）使用 s_i 和 e_i 进行表示，其中 $s_i < e_i$ ， $0 \leq i < n$ 。这些值将会存储在一个列表中，并按照递增顺序进行排序，然后这个列表会被从头扫到尾。在遇到起始点 s_i 时，会将相应的区间放入活动区间列表中（active interval list）。在遇到端点 e_i 的时候，会从活动区间链表中删除相应的区间记录。此时，如果活动列表中存在区间记录，同时遇到了某个区间的起始点，那么这个起始点所对应的区间，将会与活动链表中的所有区间相重叠，结果如图 25.3 所示。

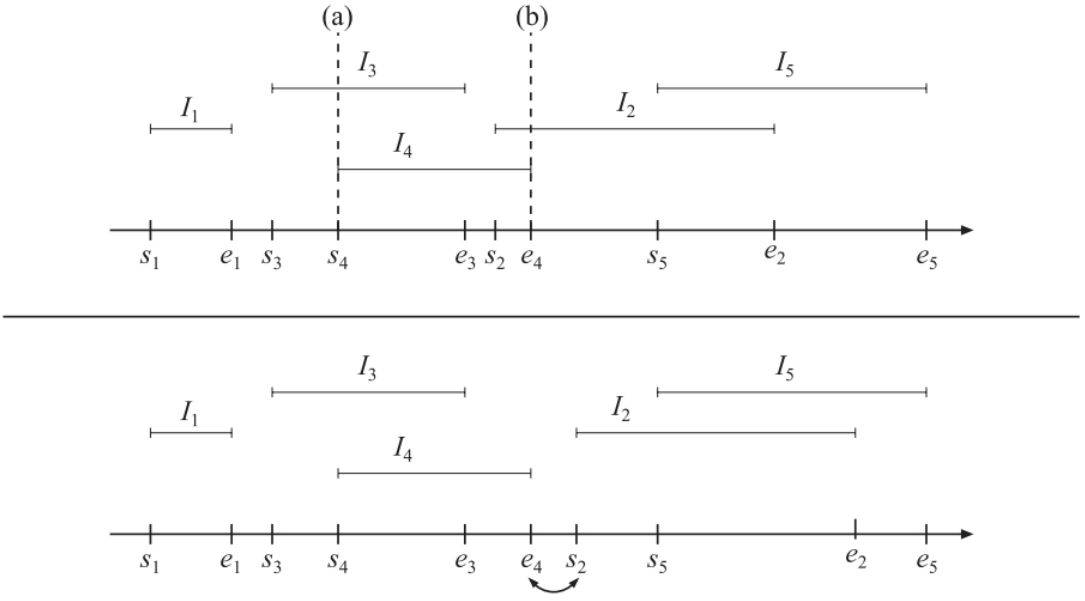


图 25.3：第一行中，当活动列表（ I_3 ）中只有一个区间时，它会在标记为（a）的地方遇到区间 I_4 ，因此我们可以得出结论，区间 I_4 和区间 I_3 是重叠的。而在后续遇到 I_2 的时候， I_4 仍然位于活动列表中（因为还没有遇到端点 e_4 ），因此区间 I_4 和区间 I_2 也是重叠的。在遇到端点 e_4 的时候（标记为（b）的地方），会将 I_4 从活动列表中移除。第二行中，区间 I_2 移动到了右边，当插入排序发现 s_2 和 e_4 需要改变位置的时候，也可以得出新的结论，即区间 I_2 和区间 I_4 不再重叠。[93]

这个过程需要对所有的区间进行排序，排序算法的时间复杂度为 $O(n \log n)$，遍历列表的时间复杂度为 $O(n)$，报告 k 个重叠区间的时间复杂度为 $O(k)$，因此算法的整体时间复杂度为 $O(n \log n + k)$。但是由于时间上的一致性，因此这个列表中的区间信息在帧与帧之间并不会有太大的变化，因此可以使用冒泡排序（bubble sort）或者插入排序（insertion sort）[62]，这两个排序算法可以在对列表进行一次遍历之后，就能够报告排序是否已经完成。对于近乎有序的列表而言，这些排序算法可以实现 $O(n)$ 的时间复杂度。

插入排序的工作原理是逐步构建有序序列。我们可以从列表中的第一个数字开始，如果我们只考虑这一项，那么这个列表就是有序的。接下来，我们添加第二项，如果第二项比第一项小，那么我们就改变第一项和第二项的位置；否则这个数组就是有序的，我们不需要进行处理。我们继续添加其他项，并根据大小关系来改变这些项的位置，直到整个列表被排序完成。我们要对所有需要进行排序的物体重复这个过程，这个过程的结果就是一个有序列表。

为了充分利用时间一致性，我们会为每个可能的区间对保留一个布尔值。对于大型模型而言，这样做可能并不实用，因为它意味着 $O(n^2)$ 的存储成本。相反，可以使用一个哈希映射（hash map）来解决这个问题。这个特定的布尔值在两个物体重叠时为 TRUE，否则为 FALSE。当列表排序完成的时候，这个布尔值会在算法的第一步中进行初始化。当区间对的状态发生改变时，布尔值会被反转，图 25.3 也说明了这一点。

我们可以为三个主轴都创建一个排序的区间列表，并使用前面所介绍的算法为每个主轴都找到相互重叠的区间。如果某对物体的三个区间都发生了重叠，那么说明它们的 AABB（区间所代表的）也相互重叠；否则，它们的 AABB 不会发生重叠。这个算法的期望时间是线性的，因此扫描剪枝算法的期望时间复杂度为 $O(n + k)$，其中 k 是成对重叠的数量。如果所有物体对都会在三个轴上发生重叠（即三个布尔值都为 true），那么这对物体可以添加到碰撞对列表中，以便后续进行快速访问。这样做有助于 AABB 的快速重叠测试。请注意，由于所使用的排序算法，因此扫描剪枝算法可能会恶化到最坏的时间复杂度，可能是 $O(n^2)$。当区间发生聚集的时候，就会发生这种情况。一个常见的例子就是大量物体平放在地板上，如果 z 轴指向地板的法线方向，那么就会在 z 轴方向上发生聚集现象。其中一种解决方案是完全跳过 z 轴，只在 x 轴和 y 轴上执行测试[25]，在很多情况下，这种方法都十分有效。Liu 等人[63]提出了 SAP 的并行版本，该版本对扫描方向也进行了优化。他们的算法针对 GPU 进行，并且提供了在线代码。

25.1.2 网格

虽然网格和分层网格是最著名的光线追踪加速结构，但是它们也可以用于宽阶段碰撞检测[88]。在其最简单的形式中，网格只是一个 n 维的非重叠网格单元数组，它覆盖了整个场景空间。因此，每个单元格都是一个 box，并且场景中的所有 box 尺寸都相同。从高层次来看，带有网格的宽阶段 CD，首先会将我们场景中所有物体的 BV 都插入到网格中。然后，如果两个物体与同一个网格单元相关联，那么我们就能够立即知道这两个物体的 BV 可能会发生重叠。因此，我们执行一个简单的 BV 与 BV 之间的重叠测试，如果它们确实发生了碰撞，那么我们可以继续进行 CD 系统的第二级。在图 25.4 的右侧，展示了一个包含四个物体的二维网格。

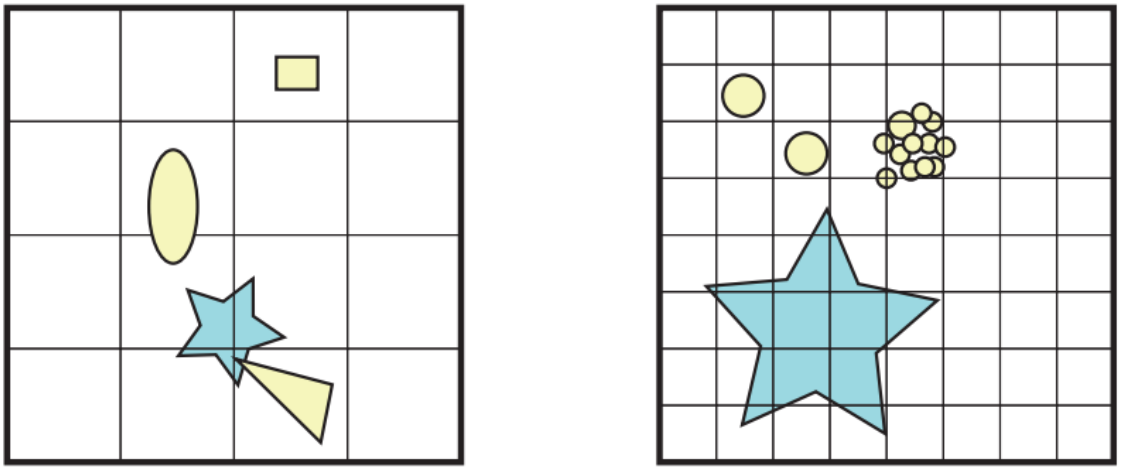


图 25.4：左侧：包含四个物体的低分辨率二维网格。请注意，由于其中的椭圆和星形重叠在一个共享的单元格上，因此这两个物体必须使用 BV 与 BV 之间的重叠测试，来检查它们之间是否真的发生了碰撞。在这种情况下，它们并不会发生碰撞。左侧的三角形和星形也重叠在一个共享的单元格中，并且它们确实发生了碰撞。右侧：更高分辨率的网格。我们可以看到，其中的星形与许多单元格相重叠，这会使得这个判定 BV 之间是否重叠的过程变得非常昂贵。还要注意的是，某个单元格中有许多较小的物体聚集在一起，这是导致效率低下的另一个原因。

为了获得良好的性能表现，选择一个合理的单元格大小是十分重要的，这个问题在图 25.4 中进行了展示说明。一个想法是找到场景中最大的那个物体，并使得单元格的尺寸足够大，从而适应所有可能朝向的物体[88]。这样一来，在三维网格的情况下，每个物体最多只会与 8 个单元格相重叠。

存储一个大型网格可能是相当浪费的，尤其是当网格中的很大一部分空间没有被使用时。因此，有人建议使用空间哈希（spatial hashing）来代替网格存储[25, 86, 88]。通常，每个单元格都会映射到哈希表中的一个索引，所有与单元格重叠的物体，都会被插入到哈希表中，然后测试可以像往常一样继续进行。在没有空间哈希的

情况下，我们需要确定包围整个场景的 AABB 尺寸，然后才能为网格分配相应的内存。同时我们还必须限制物体的移动范围，使得它们不会离开这些预先划定的界限。而空间哈希则可以完全避免这些问题，因为无论物体位于何处，都可以立即插入到哈希表中。

如图 25.4 所示，在整个网格中使用相同大小的单元格并不总是最佳的方式。另一个选择是使用分层网格，在这个方案中，会使用几个不同单元格大小的嵌套网格，物体只会被插入到比其 BV 刚好小一点的单元格中。如果在这个分层网格中，相邻层次之间的单元格大小刚好相差两倍，那么这种结构实际上与八叉树非常相似。在为 CD 实现网格和分层网格的时候，有许多细节值得了解，我们参考了 Ericson [25]和 Pouchol 等人[76]的文章，他们对该主题进行了出色的处理。

25.1.3 层次包围体

我们在章节 19.1.1 中介绍了层次包围体（bounding volume hierarchy，BVH），当时是作为使用视锥体剔除加速渲染的一种手段。至于如何构建 BVH、以及如何使用 BVH 来执行碰撞测试，将会在章节 25.2 中进行详细描述。宽阶段 CD 也可以使用 BVH 来进行实现。假设物体每帧都会发生移动，并且场景中的每个物体都存在 AABB，即每个叶子节点都包含一个 AABB 以及对应的底层物体，那么使用这些 AABB 就可以快速构建 BVH。BVH 中子节点的最大数量，通常会根据潜在的目标架构进行调整，例如：如果目标 SIMD 宽度为 8，则会使用 8 个子节点。

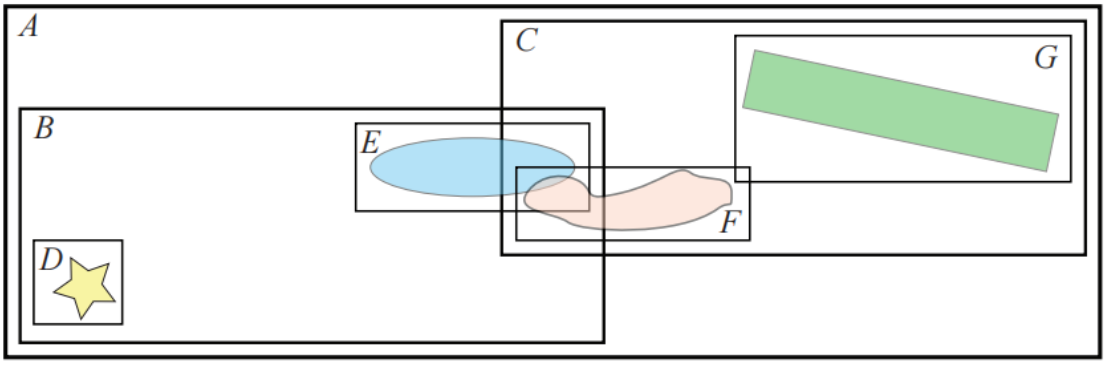


图 25.5：该 BVH 由 AABB 构成，共有 7 个节点，分别表示为 A 到 G，其中 A 是根节点，$D-G$ 是叶子节点，B 和 C 是内部节点。有两种不同的方法可以用来检测哪些叶子节点相互重叠，详见正文。

在 BVH 构建完成之后，可以实用这个 BVH 来检测哪些物体是发生重叠的，即哪些叶子节点的 AABB 会发生重叠。一种方法是根据 BVH，来对每个物体的 AABB 进行分

层测试。如果某个物体的 AABB 在遍历 BVH 期间没有与任何节点的 BV 相重叠，则可以终止对该节点子树的后续处理；如果发生了重叠，则继续向下递归测试。在图 25.5 中，为了找出与节点 D 的重叠部分，我们从根节点开始测试，因为物体总是会与根节点重叠。节点 D 并不与节点 C 不重叠，因此不需要对该子树进行进一步地测试。

测试会在节点 B 中继续，而节点 D 总是会与节点 B 重叠，因为它就是节点 B 子树中的一个子树。最后，会将节点 D 与节点 E 进行重叠测试，最终的结果表明，节点 D 不与任何其他物体相重叠。接下来，我们再确定节点 E 是否会与任何其他物体重叠，它首先会与节点 A 和节点 B 重叠，因为节点 E 就位于它们的子树中，但同时节点 E 也与节点 C 重叠。我们对节点 B 进行遍历，最终发现节点 B 中没有任何物体与节点 E 重叠。然而，在节点 C 的子树中，我们会发现节点 E 与节点 F 重叠，与节点 G 不重叠。通过使用这种方法，我们对一个叶子节点 X 进行测试，如果递归到达某一个叶子节点时，并且该叶子节点的 AABB（记作 Y ）与 X 重叠，则将这对 (X, Y) 添加到重叠的物体列表中。需要注意的是，我们不能同时将 (E, F) 和 (F, E) 都添加到列表中，这两个代表的含义实际上是一样的。

又或者，可以进行 BVH 和 BVH 之间的测试，其中 BVH 会针对自身进行测试，当两个 AABB 不重叠时，则终止递归。如果发现一对叶子节点的 AABB（对应两个不同的物体）发生重叠，则将这对叶子节点添加到一个列表中，并将其馈送到下一阶段。请注意，在对自身进行检测的时候必须要小心，不要报告 AABB 与其自身之间的重叠。我们继续使用图 25.5 中的例子，测试会从根节点开始，由于根节点总是会有与之重叠的部分，因此我们会深入到节点 A 中，对节点 B 和节点 C 进行检测。我们发现节点 B 有重叠部分，但是当我们继续深入到子树 B 之后，却发现节点 D 和节点 E 并不重叠。我们还对另一侧的子树 C 进行了研究，发现节点 F 和节点 G 重叠，因此 (F, G) 被添加到重叠列表中。接下来，由于节点 B 和节点 C 重叠，因此我们还必须对子树 B 中的所有子节点，与子树 C 中的所有子节点进行挨个测试。在这里，我们会发现节点 E 和节点 F 重叠，并将 (E, F) 也添加到列表中。

使用 BVH 方法的一个重要优点在于，我们只需要在物体的 AABB 上构建一个单一的数据结构即可，然后这个构建完成的 BVH，可以用于宽阶段 CD、视锥体剔除、遮挡剔除和光线追踪等不同的任务。

25.2 中阶段碰撞检测

在宽阶段处理之后，我们假设对物体对进行了处理，并且我们希望找到这些物体之间的重叠部分。为此，本小节将会介绍通用的层次包围体碰撞检测算法。模型可以进行

刚体运动，即旋转变换加上平移变换，或者是更一般的变形类型（章节 25.7）。由于这些方法试图为一组几何图形创建更加紧密贴合的包围体，因此这些方法可以提供高效的包围体（BV）表示，较小的 BV 可以提高算法的性能表现。还要注意的是，通常会对用于渲染的原始网格进行简化，并使用这些简化网格来用于碰撞检测，从而降低 CD 的复杂性和成本[73]。对于简单的碰撞体而言（例如 box、球体和胶囊体），这部分的处理可以跳过，直接进入到窄阶段碰撞检测。

本节将会介绍一些通用思想和方法，它们使用层次包围体（BVH）来检测两个给定模型之间的碰撞。这些算法具有两个共同的特征元素，即它们会使用 BV 来创建每个模型的层次表示；并且无论使用哪种 BV，碰撞查询的高级代码（high-level code）都是相似的。

25.2.1 BVH 构建

一开始，一个模型会由几个图元进行表示，在本小节中我们假设这些图元是三角形，但是一般来说，它们可以是任何类型的图元。由于每个模型都应当被表示为某种 BV 的层次结构，因此必须开发一些方法，使用所需的属性来构建这种层次结构。在碰撞检测算法中，我们通常使用的层次结构是一种称为 k 叉树（k-ary tree）的数据结构，其中每个节点最多可以有 k 个子节点（章节 19.1）。有许多算法会使用 k 叉树的最简单实例，即二叉树，其中 $k = 2$。但是在当前的架构中，这个 k 可能会更高一些（例如：为了匹配特定的 SIMD 宽度，或者最小化间接指针的数量等）。在每个内部节点中都会有一个 BV，并将所有子节点包含在其体积内部，在每个叶子节点会存储一个或者多个图元。对于任意节点（内部节点或者叶子节点） A，我们将其包围体记为 A_{BV}，属于节点 A 的子节点集合记为 A_c。

构建一个层次结构有四种主要方法，分别是：自下而上（bottom-up）方法、增量树插（incremental tree-insertion）方法、自上而下（top-down）方法或者线性 BVH 方法。为了创建一个高效的、紧凑的结构，通常需要尽可能地减少 BV 的表面积或者体积[31, 47]，例如：在使用地板物体的时候，就可以看到这个体积问题，由于地板并没有体积，因此在使用这个指标的时候，其本身并没有大小或者影响。所有单轴对齐的多边形都将具有相同的影响，即无论这些物体的大小如何，它们都没有任何影响。有证据表明，在大多数情况下，表面积是更好的启发式方法；而当具有较大表面的物体彼此面对时，体积则可以提供更好的性能（详见下方注释）[95]。因此，我们将在本章节的剩余部分中都使用表面积这个指标，但是请记住，有时候使用体积可能会更好。

> 在撰写本章的时候，我们思考了这个问题，我们认为两个物体相交的几率与 Minkowski 差的表面积有关，因此会取决于两个物体。

第一个方法是自下而上的（bottom-up），它首先会将几个图元组合一起，并为它们找到一个 BV。这些图元之间应当靠得很近，这可以通过使用图元之间的距离来进行确定。再然后，要么使用同样的方法来创建新的 BV；要么使用相似的方法将已有 BV 与一个或者多个 BV 组合在一起，从而产生一个新的、更大的父 BV。这个过程会一直重复下去，直到整个场景中只剩下一个 BV，此时这个 BV 就是层次结构的根节点。通过这种方式，空间中距离很近的图元在层次包围体结构中的位置也会靠得很近 [31]。

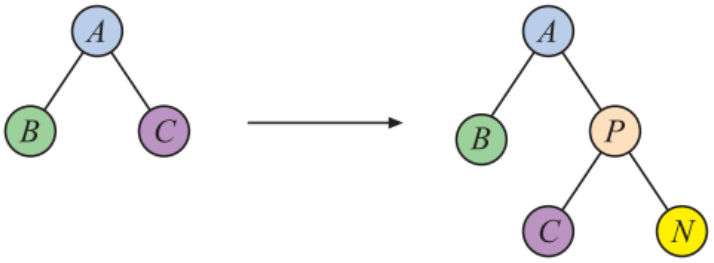


图 25.6：左侧展示了一个包含三个节点的二叉树。我们假设每个叶子节点中只存储一个图元。现在我们想要插入一个新节点 N，这个节点 N 包含一个 BV，以及一个被 BV 包围的图元。因此，该节点会被作为叶子节点插入到左侧的树中。在这个例子中，假设我们发现，如果我们将节点 N 插入到节点 C 中（而不是节点 B 中），那么这棵树的总体体积会变得更小。然后我们会创建一个新的父节点 P，将旧节点 C 和新节点 N 都包含进来。

增量树插（incremental tree-insertion）方法会从一棵空树开始，然后将所有其他的图元及其 BV，逐个添加到这棵树中，如图 25.6 所示。想要构建一棵高效的树，我们必须在树中找到一个合理的插入点。至于选择这个插入点的指标，应当能够使得树的整体体积的增加量最小。一个简单的方法是将新节点下降到使得树体积增加较小的子节点中。这种算法的运行时间通常为 $O(n \log n)$。将图元的插入顺序进行随机化，可以改善树的构建。

自上而下（top-down）是一种十分流行的方法，它首先会为模型的所有图元都找到一个 BV，并将这个最大的 BV 作为树的根节点。然后采用一些分治策略（divide-and conquer strategy），即将 BV 分成 k 个部分或者更少的部分。对于每个这样的部分，找到所有包含的图元，然后使用与根节点相同的方式再次创建一个 BV，即递归创建层次结构。最常见的方法就是找到一个图元应当被划分开来的轴，然后在这个轴上寻找一个合适的分割点。章节 22.4 中对几何概率（geometric probability）进

行了讨论，这个指标可以用于寻找一个较好的分割点，从而得到一个更好的 BVH。在光线追踪中，会使用表面积启发式（surface area heuristic，SAH）方法，其方程为：

$$C(n) = \begin{cases} C_i A(n) + C\left(n_l\right) + C\left(n_r\right), & n \in I \\ C_t A(n) N(n), & n \in L \end{cases} \tag{25.2}$$

其中 $C(n)$ 为节点 n 的 SAH 代价， n_l 和 n_r 分别为节点 n 的左右子节点。包围体 n 的表面积记为 $A(n)$， C_i 为遍历内部节点的代价， C_t 为射线与三角形求交的代价。同样，所有内部节点的集合记为 I，所有叶子节点的集合记为 L。 $N(n)$ 是叶子节点中三角形的个数。请注意，方程 25.2 中的第一行代表了使得节点 n 成为内部节点的代价，第二行代表了使得节点 n 成为叶子节点的代价。在理想情况下，方程中的第一行应当被最小化，这是通过沿着 x 轴、 y 轴和 z 轴搜索最佳分割点来实现的。Embree [90]为此实现了许多最先进（state-of-the-art）的方法。可以根据具体哪个成本最低，来选择构建一个内部节点还是叶子节点。由于其中的表面积指标是用于 SAH 度量的，它可能并不适合用于碰撞检测。但是原理是一样的，其中的表面积函数 $A(n)$ 可以替换为体积函数 $V(n)$，这个体积函数 $V(n)$ 计算的是 BV 的体积。

请注意，自上而下方法的一个潜在优点在于，可以惰性地创建层次结构，即按需创建层次结构。这意味着我们可以只为场景中真正需要的部分来构建层次结构。但是由于这个构建过程是在运行过程中执行的，因此每当需要创建部分层次结构的时候，性能可能会显著下降。这对于具有实时要求的应用程序（例如游戏）而言是不可接受的，但是对于 CAD 应用程序或者一些离线计算（例如路径规划、动画等）而言，可以大大降低时间开销和内存开销。

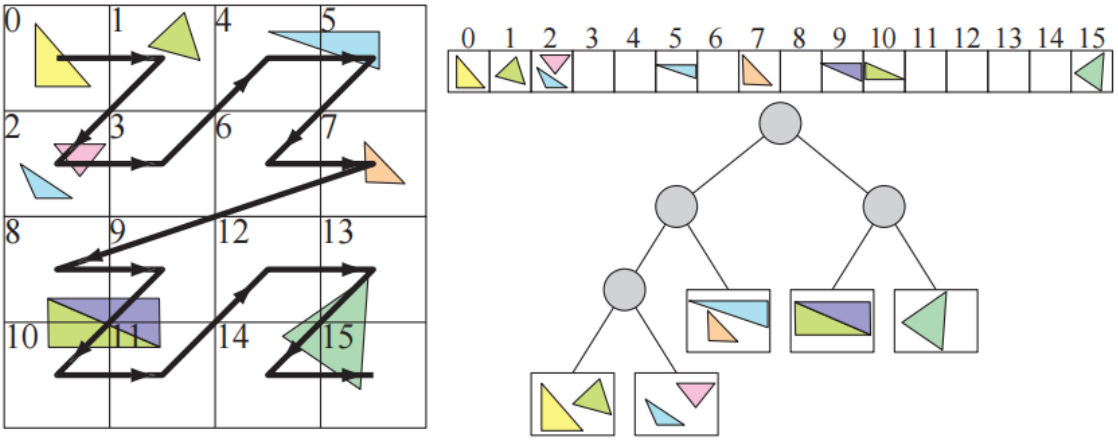


图 25.7：左侧：加粗的 z 型曲线就是 Morton 曲线。每个三角形都会被分配一个 Morton 码，其单元格的中心与三角形的质心相重叠。三角形会根据 Morton 码进行排序（右上角）。左下角展示了最终的树结构，它是通过分层划分排序列表来进行创建的。

线性 BVH 方法[59]最初的目标是在 GPU 上构建 BVH，但是它在 CPU 上的表现也十分出色，其核心思想如图 25.7 所示。首先，在所有三角形周围都找到一个 box；其次，将三角形按照某种顺序存储在一个数组中。然后，根据每个三角形的质心，为其分配一个整数 Morton 码（章节 23.8）。再然后，根据它们的 Morton 码对三角形进行排序。最后，对这个三角形列表进行层次分割，每一步都会创建一个包围其三角形的内部节点。当子列表中没有剩余三角形的时候，或者当子列表中的三角形数量很少的时候（可以打包放入一个叶子节点中），这个过程就会停止。Karras 和 Aila [45, 46]对这个主题的许多研究论文进行了综述，并提出了适用于 BVH、八叉树和 k-d 树的高度优化版本，其中的 k-d 树基于将排序后的 Morton 码解释为一个基数树（radix tree），从而快速构建 BVH。Apetrei [1]对 Karras 的技术进行了改进，在可以同样生成一棵树的前提下，将两个 pass 合并为了一个 pass。

CD 算法的一个挑战在于，我们需要找到与物体紧密贴合的包围体和层次结构构建方法，从而创建平衡且高效的树结构。请注意，平衡树在最差情况下的执行效果会更好，因为每个叶子节点的深度都是相同的（或者是几乎相同的）。这意味着沿着层次结构，向下遍历到任何叶子节点（即图元）所需要的时间都是相等的，并且碰撞查询的时间不会因为访问不同的叶子节点而发生改变。从这个意义上说，平衡树是最优选择。然而，这并不意味着它对于所有输入都是最好的，例如：如果模型中的某一部分很少或者永远都不会发生碰撞，那么这些部分可以位于一个非平衡树的深处，这样可以使得被查询最多的部分更加接近根节点，从而降低遍历时间[34]。章节 25.2.4 中对 OBB 树的这个过程细进行了描述。

还要注意，章节 19.1 中描述了几个与加速算法相关的空间数据结构，而章节 22.3 中则讨论 BV 的创建方法。

25.2.2 BVH 间的碰撞测试

在中阶段 CD，我们将会找出哪些叶子节点的 BV 与其他叶子节点的 BV 相互重叠，即这个阶段将会创建一个列表，其中包含了 BV 相互重叠的叶子节点对。这些信息将被进一步发送到窄阶段 CD（章节 25.3）。或者对于重叠的 BV 对，可以立即使用窄阶段测试，来检查其内容是否发生碰撞。

这里我们回顾一下相应的符号标记，其中 A 和 B 是模型层次结构中的两个节点，在第一次调用的时候，它们就是模型的根节点。 A_{BV} 和 B_{BV} 用于访问相应节点所对应的 BV； A_c 是节点 A 子节点的集合。其基本思想是在检测到重叠的时候，打开一个较大的 box，并对其中的内容（子节点）进行递归测试。

```
    MidPhaseCD(A, B)
1 : if(isLeaf(A) and isLeaf(B))
2 :     add(A, B, L);  // add pair (A, B) to list L
3 : else if(isNotLeaf(A) and isNotLeaf(B))
4 :     if(overlap(A_BV, B_BV))
5 :         if(SurfaceArea(A) > SurfaceArea(B))
6 :             for each child C ∈ A_c
7 :                 MidPhaseCD(C, B)
8 :         else
9 :             for each child C ∈ B_c
10 :                MidPhaseCD(A, C)
11 : else if(isLeaf(A) and isNotLeaf(B))
12 :    if(overlap(A_BV, B_BV))
13 :        for each child C ∈ B_c
14 :            MidPhaseCD(C, A)
15 : else
16 :    if(overlap(B_BV, A_BV))
17 :        for each child C ∈ A_c
18 :            MidPhaseCD(C, B)
```

从这段伪代码中我们可以看出，有些代码是可以共享的，这里将其展开是为了展示算法是如何进行工作的。其中有几行值得注意，第 $1-2$ 行将一对叶子节点 (A, B) 添加到列表 L 中，这个列表 L 存储了所有的重叠叶子节点对。第 $3-10$ 行对两个节点都是内部节点的情况进行了处理。通过比较二者的表面积，即 $\text{SurfaceArea}(A) > \text{SurfaceArea}(B)$，使得表面积最大的节点下降。这个测试表面积背后的思想是，使得调用 $\textbf{MidPhaseCD}(A, B)$ 和 $\textbf{MidPhaseCD}(B, A)$ 能够得到相同的树遍历，这样遍历过程就变得确定了。这个测试十分很重要，由于在每个步骤中，我们都会首先遍历最大的那个 box，因此它往往可以提供更好的性能表现。

另一个想法是交替下降 A 和 B，这样可以避免计算表面积，因此执行速度也会更快。或者，可以预先计算刚体的表面积并存储在节点中，但是这种计算需要在每个节点上存储额外的信息。此外，对于许多 BV 而言，我们并不需要计算实际的表面积，只需要一个保持“表面积顺序”的计算即可，举个例子：只需要比较球体的半径就够了。请注意，最终的列表 L 通常会被发送到窄阶段 CD 中。

25.2.3 BVH 成本函数

方程 25.3 中的函数 t 首先被光线追踪所引入（以一种稍微不同的形式，没有其中的最后一项），并将其作为在光线追踪加速算法中，对分层 BV 结构性能的评估准则[92]。自那以后，它也一直被用于评估 CD 算法的性能[33]，并对其中的最后一项进行了扩展，使其包含一些碰撞系统特有的新成本[47, 50]，这些新成本可能会对性能产生重大影响。这个成本来自于这样的一个事实，即如果一个模型正在进行刚体运动，那么它的 BV 和层次结构（部分或者所有）可能必须要进行重新计算，这取决于具体的运动情况和 BV 的选择：

$$t = n_v c_v + n_p c_p + n_u c_u \tag{25.3}$$

其中： n_v 是 BV–BV 重叠测试的数量， c_v 是一个 BV–BV 重叠测试的成本， n_p 是测试重叠的图元对的数量， c_p 是测试两个图元是否重叠的成本， n_u 是由于模型运动而需要进行更新的 BV 数量， c_u 是一个 BV 的更新成本。

创建一个更好的模型层次分解方式，将会导致 n_v 、 n_p 和 n_u 的值更低。创建一些更好的方法来确定两个 BV 或者两个三角形之间是否重叠，将会降低 c_v 和 c_p 。然而，这些目标通常是相互冲突的，这是因为，为了使用更快的重叠测试而去改变 BV 的类型，通常意味着我们会获得更加宽松的拟合体积，使得 BV 紧密贴合的程度降低。

在过去，使用不同 BV 的例子有球体[42]、轴对齐包围盒（AABB）[5, 40]、定向包围盒（OBB）[33]、k–DOP（离散有向多面体）[47, 49, 96]和胶囊体[55]。其中球体的变换速度是最快的，重叠测试也很快，但是它们的拟合效果非常差。AABB 通常可以提供更好的拟合效果以及相对快速的重叠测试，如果模型中包含大量轴对齐的几何形状（例如大多数的建筑模型），那么它们是一个很好的选择。OBB 的拟合效果更好，但是重叠测试要更慢。k–DOP 的拟合效果取决于参数 k ， k 值越大，拟合越好，但是相对应的重叠测试也就越慢，变换速度也越慢。

25.2.4 OBB 树

OBB 树[33]是一种特殊的 BVH 结构，它也可以用于中阶段碰撞检测，我们将在这里对其进行详细介绍，因为它是典型碰撞检测系统的代表。在碰撞检测的过程中，如果发现两个表面非常接近并且几乎平行的时候（parallel close proximity），那么这个方案的性能会特别好。这种情况经常发生在公差分析（tolerance analysis）和虚拟样机（virtual prototyping）中。其中一个例子是安装发动机的机器部件，人们希望能够在不需要制造零件的前提下，确保这些零件可以很容易地组装起来。

选择包围体

该方法使用定向包围盒（oriented bounding box，OBB）来作为包围体。然而，本小节中的许多概念也同样适用于其他 BV，例如 AABB。使用 OBB 的一个原因是，相比于使用 AABB 和球体，这样的树能够更好地聚集底层几何物体。然而，使用 AABB 来构建树结构，要比使用 OBB 更快，因此真正的性能表现还取决于实际情况。章节 22.13.5 中讨论了 OBB 与 OBB 之间的重叠测试。使用章节 25.2.3 中的性能评估框架和之前的推理方式，意味着 OBB 的 n_v 和 n_p 要低于 AABB 和球体。

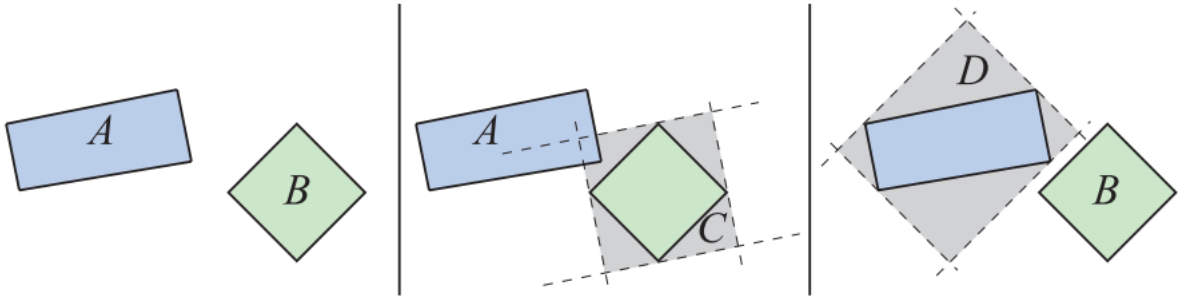


图 25.8：左边是两个 OBB （ A 和 B ），它们所进行的 OBB 与 OBB 重叠测试，将会省略最后 9 个轴的测试。如果从几何角度来解释，那么这种 OBB 与 OBB 的重叠测试，可以近似为两个 AABB 与 AABB 的重叠测试。中间的插图展示了在 A 的坐标系中， B 被 AABB C 所包围，而这个 C 会与 A 发生重叠，因此将会继续进行右边的测试。在右边，在 B 的坐标系中， A 被 AABB D 所包围，而 B 与 D 不会发生重叠，因此测试到此结束。上述情况都是在二维空间中进行的，在三维空间中， D 和 B 也可能会发生重叠；而实际上 A 和 B 是不会发生重叠的，通过剩余的测试就可以进行判断。在这种情况下， A 和 B 将会被错误地报告为发生重叠。

Van den Bergen 提出了一种简单的技术，用来加速两个 OBB 之间的重叠测试[5, 7]，该方法跳过了最后 9 个轴的测试，这些测试对应于垂直于第一个 OBB 边缘和第二个 OBB 边缘的方向。这种测试通常被称为简化 SAT（SAT lite）。从几何上来说，这可以被认为是进行两个 AABB 与 AABB 的相交测试，其中第一个测试会在第一个 OBB 的坐标系中进行，第二个测试会在第二个 OBB 的坐标系中进行，如图 25.8 所示。这种简化的 OBB 与 OBB 测试（省略了最后 9 个轴的测试），有时会将两个不相交的 OBB 报告为重叠。在这种情况下，OBB 树中的递归将会超出必要的深度。在实际测试中，跳过这些测试（最后 9 个轴的测试）的最终结果是，算法的平均性能得到了提高。Van den Bergen 的技术已经在名为 SOLID 的碰撞检测包[5, 6, 7, 9]中进行了实现，它还可以处理变形物体。

建立层次结构

OBB 树的基本数据结构是一棵 k 叉树，也就是说，一个内部节点可以有多达 k 个子节点；其中每个内部节点都会保存一个 OBB（章节 22.2），每个外部节点（叶子节点）会保存一个或者多个三角形。通常会根据目标架构来选择合适的 k 值，例如：如果使用 AVX 实现，则可以选择 $k = 8$，AVX 对于 32 位浮点数的 SIMD 宽度为 8。Gottschalk 等人开发了一种自上而下创建层次结构的方法，该方法可以分为以下步骤：首先为一个三角形 soup（triangle soup，及乱序集合）找到一个紧密贴合的 OBB，然后会沿着 OBB 的一个轴对其进行分割，这可以将三角形划分为两组。对于每一个组三角形，都会计算一个新的 OBB。章节 22.3 中讨论了 OBB 的创建。

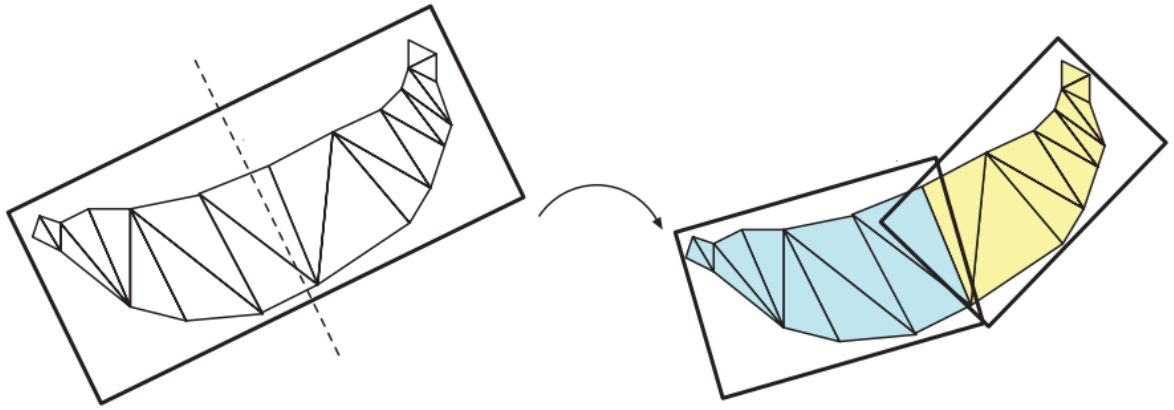


图 25.9：这个示意图展示了一组具有 OBB 的几何图形，是如何沿着 OBB 的最长轴进行分割的，分割点由虚线进行标记。然后将几何图形划分为两个子集合，并为每个子集合分别找到一个 OBB；这个过程会递归进行，从而建立一棵 OBB 树。

在我们计算了一组三角形的 OBB 之后，应当将体积和三角形进行分割，并形成两个新的 OBB。Gottschalk 等人使用了这样一种策略，他们将 box 的最长轴划分成两个长度相同的部分，图 25.9 给出了这个过程的示意图。一个能够将 box 等分成两个部分的平面，会被用来将三角形分割成两个子集合，而对于那些穿过该平面的三角形，将会根据质心所在的位置，被分配到对应的一侧中。在极少数情况下，可能所有三角形的质心都位于这个分割平面上，或者所有三角形的质心都位于分割平面的同一侧，对于这种情况，我们可以按照递减顺序，尝试对其他坐标轴进行分割。请注意，使用表面积启发式（方程 25.2）可能产生更好的 BVH [95]。

对于每个子集合，可以使用章节 22.3 中简要介绍的矩阵方法，来计算其对应的 OBB。如果父 OBB 在中位中心点（median center）进行分割，那么每个子节点将会具有相同数量的三角形，这样可以得到一棵平衡树。在光线追踪的相关领域中，对其他的分割策略进行了广泛的研究[89]。

处理刚体运动

在 OBB 树层次结构中，每个 OBB A 会与刚体变换矩阵 $\mathbf{M}_A$ 一起存储（包含一个旋转矩阵 $\mathbf{R}$ 和一个平移向量 $\mathbf{t}$ ）。这个矩阵保存了该 OBB 相对于其父物体的相对方向和相对位置。

现在，假设我们开始对两个 OBB（ A 和 B ）进行重叠测试。我们会在其中一个 OBB 的坐标系中，来进行 A 和 B 之间的重叠测试，假设我们决定在 A 的坐标系中来执行这个测试。这样一来，OBB A 就是一个以原点为中心的 AABB（在它自己的坐标系中）。接下来的思路是将 B 转换到 A 的坐标系中，这是通过一个矩阵变换（方程 25.4）来完成的，它首先将 B 变换到其自身的位置和方向上（使用矩阵 $\mathbf{M}_B$ ），然后再将其转换到 A 的坐标系中（使用 A 变换的逆矩阵 $\mathbf{M}_A^{-1}$ ）。让我们回顾一下，对于刚体变换而言，其转置矩阵就是逆矩阵，因此只需要很少的额外计算：

$$\mathbf{T}_{AB} = \mathbf{M}_A^{-1}\mathbf{M}_B \tag{25.4}$$

OBB 与 OBB 之间的重叠测试，以一个 3×3 旋转矩阵 $\mathbf{R}$ 和一个平移向量 $\mathbf{t}$ 所组成的矩阵作为输入，其中平移向量 $\mathbf{t}$ 保存了 B 相对于 A 的方向和位置（章节 22.13.5），因此这个 $\mathbf{T}_{AB}$ 可以进行如下分解：

$$\mathbf{T}_{AB} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 0 \end{pmatrix} \tag{25.5}$$

现在我们假设 A 和 B 发生了重叠，此时我们想要深入到 A 的子节点 C 中。我们可以这样做：我们选择在 C 的坐标系中进行测试。然后，这里的想法是将 B 转换为 A 的坐标系中（使用 $\mathbf{T}_{AB}$ ），然后再将其转换到 C 的坐标系中（使用 $\mathbf{M}_C^{-1}$ ）。这是通过下面的矩阵来完成的，然后再次将其作为 OBB 与 OBB 重叠测试的输入：

$$\mathbf{T}_{CB} = \mathbf{M}_C^{-1}\mathbf{T}_{AB} \tag{25.6}$$

然后使用这个过程，对所有 OBB 进行递归测试。

其他

在章节 25.2.2 中，我们介绍了用于两棵层次树之间碰撞检测的 $\mathbf{MidPhaseCD}$ 伪代码，它同样也可以用于本小节所创建的两棵树。需要进行修改的是 $\mathrm{overlap}()$ 函数，它应当指向一个测试两个 OBB 是否发生重叠的函数。

OBB 树中涉及的所有算法，都已经在一个名为 RAPID （健壮和精确的多边形干扰检测，Robust and Accurate Polygon Interference Detection）的免费软件包[33]中

进行了实现。

25.3 窄阶段碰撞检测

此时，宽阶段 CD 和中阶段 CD 已经将数据缩小到了一个列表 L，其中包含了 BV 相互重叠的成对叶子节点。这里的目标是对所有相交的图元对进行计算，从而避免物体发生相互穿透，或者使用距离查询来确定物体之间的距离是否足够远。下面我们将介绍这两个主题。

25.3.1 图元 vs 图元

一般来说，此时我们有了一个列表 L，其中包含了在进入这个阶段时，相互重叠的成对叶子节点。在最简单的实现中，可以对第一个叶子节点中的每个图元都执行一个循环，并与另一个分支中的所有图元进行测试。我们在第 22 章中介绍了图元与图元之间的相交测试。

应当注意的是，从代码的角度来看，可以将中阶段和窄阶段合并为一个阶段，这样做可能是有利的，也可能是不利的。将这两个阶段合并起来，可能会简化代码实现；但是对于一个包含所有图元对的列表，我们可以对其进行排序，从而使得所有相同类型的图元对位置相近。这种排序可以实现更快的并行 SIMD 实现。

当列表 L 中包含凸多面体的时候，可以使用距离查询来确定两个物体之间是否会发生相互穿透；如果发生了穿透，则需要将它们分开，直到它们不会发生穿透为止。这是下一小节中的主题。

25.3.2 距离查询

在某些应用程序中，我们可能会想要测试某个物体与环境之间是否至少有一定的距离，例如：在设计一辆新车的时候，必须为不同体型的乘客都提供足够的空间，使得他们能够舒适地坐下。因此，可以在汽车座椅上尝试放置不同尺寸的虚拟人，来看看他们是否能够在不碰到到汽车其他部分的情况下就座。最好的情况是，乘客自己能够坐下，并且与汽车内部的一些部件保持至少 10 厘米的距离，这种测试被称为公差验证（tolerance verification）。它也可以用于路径规划，即如何通过算法，来确定某个物体从一点到另一点的无碰撞路径。给定一个物体的加速度和速度，这个最小距离可以用来估计撞击时间的下限，通过这种方式，在这个下限时间之前就可以避免碰撞检测[62]。另一个相关的查询是穿透深度（penetration depth），即找出两个物体相

互移动了多远。这个距离可以用来将物体向后移动，使得它们不再穿透，并且可以在那个点上计算出一个适当的碰撞响应。

最早计算凸多面体之间最小距离的实用方法之一被称为 GJK，该方法以其发明者 Gilbert、Johnson 和 Keerthi [30]的名字进行命名。本小节将对这个算法进行概述。GJK 会计算两个凸物体 A 和 B 之间的最小距离。为此，会使用 A 和 B 之间的差物体（difference object，有时被称为和物体 sum object）[6]：

$$A - B = \{\mathbf{x} - \mathbf{y} : \mathbf{x} \in A, \mathbf{y} \in B\} \tag{25.7}$$

这也被称为物体 A 和（反射的）物体 B 的 Minkowski 和（章节 25.11.3）。所有的差值 $\mathbf{x} - \mathbf{y}$ 可以被视为一个点集，它构成了一个凸物体，图 25.10 展示了这种差值的一个例子。

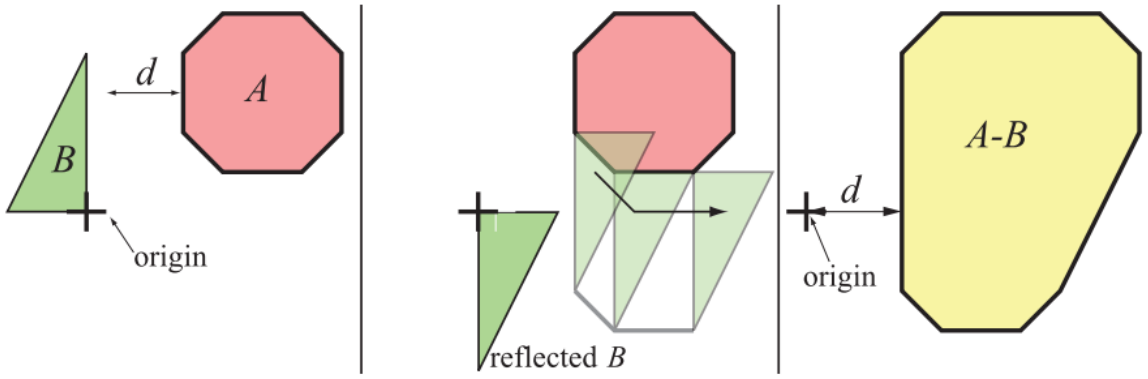


图 25.10：左边是两个凸物体 A 和 B 。想要构造 $A - B$ ，首先需要移动 A 和 B ，使得一个参考点位于原点（已经在左侧完成，其中物体 B 的右下角顶点位于原点处）。然后将 B 进行反射（取反），如右图所示，并将物体 B 上所选择的参考点放在物体 A 的表面上，然后将反射的物体 B 围绕物体 A 周围扫一圈，这样就构成了 $A - B$ ，如最右侧所示。左右两边同时展示了这个最小距离 d 。

GJK 的思想是，我们会去计算 $A - B$ 与原点之间的最小距离，而不是计算物体 A 和物体 B 之间的最小距离，这两个距离可以证明是等价的。该算法的可视化结果如图 25.11 所示。请注意，如果原点位于 $A - B$ 内部，则说明 A 和 B 重叠。

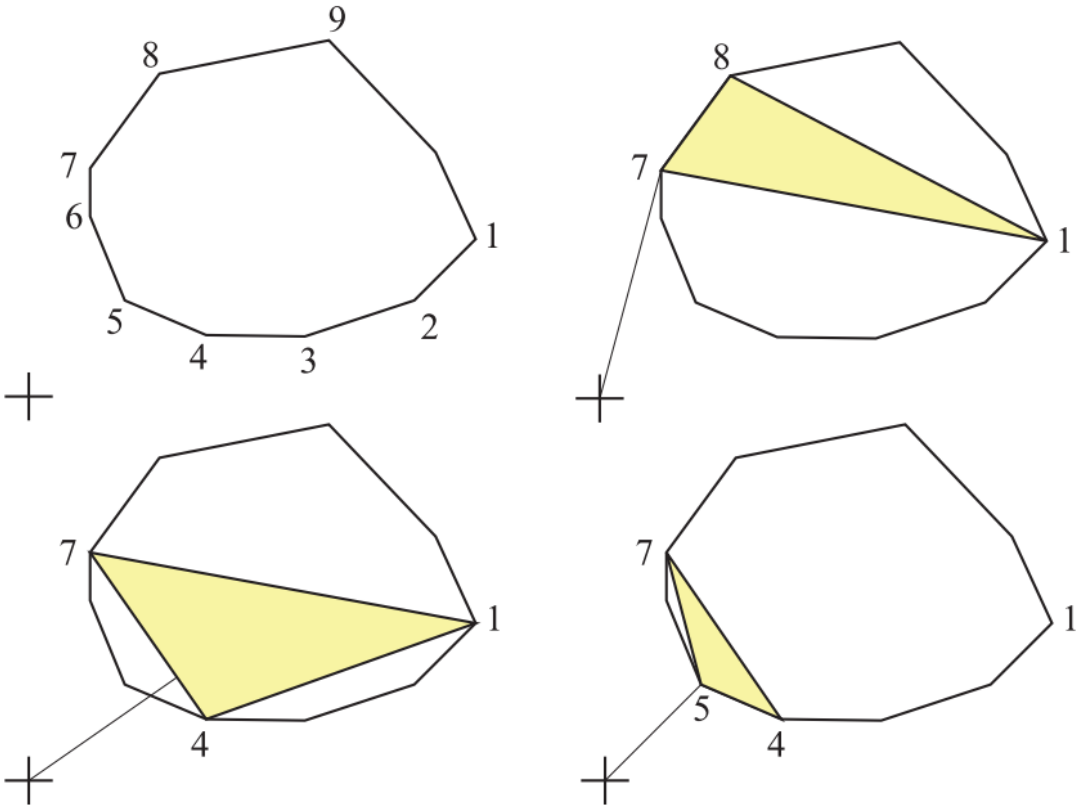


图 25.11：GJK 算法示意图。左上角：计算原点到多边形的最小距离。右上角：GJK 算法会选择一个任意的三角形作为起点，然后计算原点到该三角形的最小距离，其中顶点 7 是最近的。左下角：在下一步中（未显示），会将所有顶点都投影到从原点到顶点 7 的直线上，投影之后最近的顶点会取代投影中距离三角形最远的顶点。因此顶点 4 取代了顶点 8。然后找到这个新三角形上距离原点最近的点，它位于顶点 4 到顶点 7 的边上。右下角：顶点会继续被投影到从原点到上一步中最近点的直线上，因此顶点 5 取代了投影上最远的顶点 1。顶点 5 是这个三角形上距离原点最近的点，当顶点 5 投影到原点与顶点 5 之间的直线上时，我们发现顶点 5 是整体上最近的点，这样迭代过程就完成了。此时我们找到三角形上距离原点最近的点，正好也是顶点 5，最后返回这个点即可。[44]

这个算法会从多面体中的任意一个单纯形（simplex）开始。单纯形是指各个维度中最简单的图元，因此在二维中它是三角形，在三维中它是四面体（tetrahedron）。这个起始元素可以是任何有效的单纯形，例如：完全位于多面体内部的任意四面体。然后计算这个单纯形上距离原点最近的点。Van den Bergen 展示了如何通过求解一组线性方程来实现这一点[6, 7]。有了这个最近点之后，就可以形成一个向量，即从原点指向最近点。将多面体上的所有顶点都投影到这个向量上，并在这些投影顶点中选择距离原点最近的顶点，将其来作为单纯形更新后的一个新顶点。由于单纯形中添加了一个新顶点，因此我们还必须删除一个单纯形中的现有顶点（否则它将不再是单纯形了），这里我们会删除距离原点最远的那个投影点。一旦完成了这个过程，我们

就会计算到这个新单纯形的最小距离，然后使用同样的算法对所有顶点进行再次迭代，直到不再更新单纯形为止。对于两个多面体而言，算法会在有限步骤内结束[30]。通过使用增量计算（incremental computation）和缓存等技术，可以提高算法的性能表现[6]。

Van den Bergen 描述了一个快速且健壮的 GJK 实现[6, 7]。GJK 还可以扩展到计算穿透深度[8, 12, 37]。特别地，van den Bergen [8, 9]还描述了一种基于 GJK 的扩展多边形算法（expanding polytope algorithm，EPA）。当发生穿透的时候，如图 25.11 所示的原点（使用加号进行表示）会位于多边形内部。然后，EPA 会执行与 GJK 类似的步骤，从而找到多边形上最接近原点的顶点。还有一些其他计算最小距离的算法，例如 Lin–Canny 算法[61]、V–Clip [70]、PQP [55]、SWIFT [23]、SWIFT++ [24]等，它们还可以计算出凹刚体之间的距离。

在为当前帧计算 GJK 时，使用前一帧中的分离轴来作为起始向量，通常是很有益的[8]。当时间一致性较高时（画面变化较少），算法往往会在第一步就终止了，Gregorius [37]指出这样会有一个数量级的加速。请注意，还可以使用基于射线追踪的方法来计算穿透距离，参见下一小节的图 25.13。

当在凸多面体上使用分离轴定理来寻找穿透深度的时候，另外一种优化方法是将一个更简单的物体（例如球体）完全存储在凸多面体内，并使用这个简单物体来进行剔除轴测试[85]。两个球体在分离轴上的重叠量，总是会小于或者等于两个凸多面体之间的重叠量。我们的目标是找到最小的穿透深度，从而得知我们具体需要移动这两个物体多少距离，才能使得它们分离。假设在经过一些轴测试之后，当前的最小穿透深度为 d，那么如果此时球体的重叠量大于 d，那么我们就不需要进一步关心当前轴了，因为这样的凸多面体之后会有更大的重叠量。这样可以大大提高性能表现。

请注意，在使用距离查询的时候，我们不需要计算三角形与三角形之间的交点，因为我们不需要这个交点信息，就可以直接获得最小距离或者最小穿透深度。

25.4 射线碰撞检测

在本小节中，我们将会介绍一种在特定情况下运行良好的快速技术。想象现在有一辆汽车正在倾斜的道路上向上行驶，我们希望使用一些有关道路的信息（即构建道路的图元）来引导汽车向上行驶。当然，这个目的可以通过使用章节 25.1 到章节 25.3 中的技术来进行实现，即对汽车车轮的所有图元，与道路上的所有图元进行碰撞测试。然而，对于游戏或者其他的一些应用程序而言，这种详细的碰撞检测并不总是需要的。相反，我们可以使用一组射线来对正在移动的物体进行近似。以刚才的汽车为

例，我们可以在汽车的四个轮子上各自放置一条射线（如图 25.12 所示）。只要我们能够假设这辆汽车只有四个轮子与环境（道路）相接触，那么这种近似在实践中就是有效的。假设一开始汽车位于一个平面上，我们将每条射线放置在一个车轮上，射线的每个原点都位于车轮与环境接触的位置处。然后，车轮上的射线会根据环境进行相交测试，如果射线原点到环境的距离为零，则代表这个轮子刚好在地面上；如果这个距离大于零，则代表这个车轮与环境没有任何接触；如果这个距离小于零，则代表车轮已经穿透到了环境中。应用程序可以使用这些距离信息来计算碰撞响应：负的距离应当使得汽车（在方向盘处）向上移动，而正的距离应当使得汽车向下移动（除非汽车在空中飞行了一段时间）。请注意，这种类型的技术可能很难适应非常复杂的场景。例如：如果汽车与环境发生了碰撞并处于旋转状态，那么则需要更多不同方向的射线来帮助判断汽车此时所处的状态。

为了加快相交测试的速度，我们还可以使用在计算机图形学中最为常用的效率提高技术——分层表示方法。环境可以使用一个轴对齐的 BSP 树（章节 19.1.2）、BVH、网格、（松散）八叉树或者其他数据结构来进行表示。例如：对于动态几何物体，Frye [28]使用一个松散八叉树（章节 19.1.3）来进行表示；对于静态几何物体，则使用一个轴对齐的 BSP 树来进行表示。根据环境中所使用的图元，所需要的射线与物体相交测试的方法也有所不同，详见第 22 章。

与标准的光线追踪不同，标准的光线追踪所需要的是射线前方的最近物体；而在这里吗，我们实际上也需要射线最后面的交点，它可以有一个负的距离，即这个交点可以位于射线原点的后面。为了避免让射线在两个方向上（前方和后方）进行搜索，因此测试射线的原点会被向后移动，直到原点位置位于物体包围盒的外面，然后再针对环境进行相交测试。实际上这只是意味着，射线并不是从距离 0 开始的，而是从一个负距离开始的，这使得射线会从物体包围盒外面开始发射。为了处理更加一般的场景，例如驾驶汽车通过隧道并检测隧道与车顶的碰撞，我们就必须在两个方向上进行搜索。

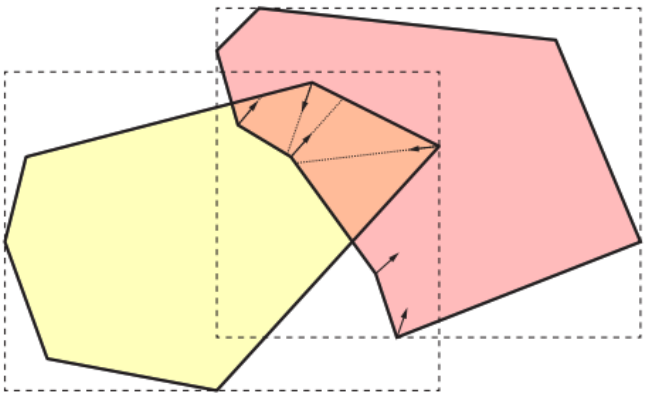


图 25.13：两个 box 重叠区域中的顶点，会向负法线方向上发射一条射线。其中右下角的两条射线击中的物体，与射线原点所属的物体相同，因此不会进行进一步的处理。会根据虚线的长度和方向来计算碰撞响应。

Hermann 等人[41]提出了一种在碰撞检测中使用射线的不同方法，如图 25.13 所示。当我们找到两个重叠区域的时候，只对重叠区域内的顶点进行处理。对于每个这样的顶点，我们会在顶点的负法线方向上发射一条射线，并且只针对另一个物体进行相交测试。如果与另一个物体存在正相交距离的交点，那么这个顶点会与交叉点维护为一个碰撞对。Hermann 等人进一步描述了如何对其中的无效碰撞对进行过滤。然后会使用法线和碰撞对来计算碰撞响应。例如：可以在碰撞对中两个点之间的向量方向上，施加一个惩罚力（penalty force），这个惩罚力的大小与该向量的长度成比例。Lehericey 等人[60]利用时间一致性，提出了一种更快的变体方法。

25.5 使用 BSP 树的动态 CD

本小节将会介绍 Melax [65, 66]的碰撞检测算法。该方法使用 BSP 树（章节 19.1.2）来描述的场景中的几何物体，并检测几何物体之间的碰撞，所使用的碰撞器可以是球体、圆柱体或者物体的凸包。它还允许动态的碰撞检测，例如：假设一个球体从第 n 帧的 $\mathbf{p}_0$ 位置，移动到了第 $n+1$ 帧的 $\mathbf{p}_1$ 位置，该算法能够检测出从 $\mathbf{p}_0$ 到 $\mathbf{p}_1$ 的直线路径上是否发生了碰撞。该算法已经被用在了商业游戏中，它会将角色的几何形状近似为一个圆柱体。

标准的 BSP 树可以非常高效地对直线段进行测试。线段可以表示一个从 $\mathbf{p}_0$ 移动到 $\mathbf{p}_1$ 的点（粒子），它可能会有几个交点，但是其中的第一个（如果有的话）交点代表了该点和 BSP 树中所表示几何物体之间的碰撞。请注意，在这种情况下，BSP 树是表面对齐（surface aligned）的，而不是轴对齐（axis-aligned）的。也就是说，BSP 树中的每个平面，都与场景中的墙壁、地板或者天花板对齐。这种方法可以很容易扩展到半径为 r 的球体（而不是一个点），它从 $\mathbf{p}_0$ 移动到 $\mathbf{p}_1$ 。它并不是针对 BSP 树节点中的平面来对线段进行测试，而是会沿着平面的法线方向移动距离 r 。在章节 25.11 中，有一种与这种相交测试的相类似的方法。这种平面调整如图 25.14 所示。

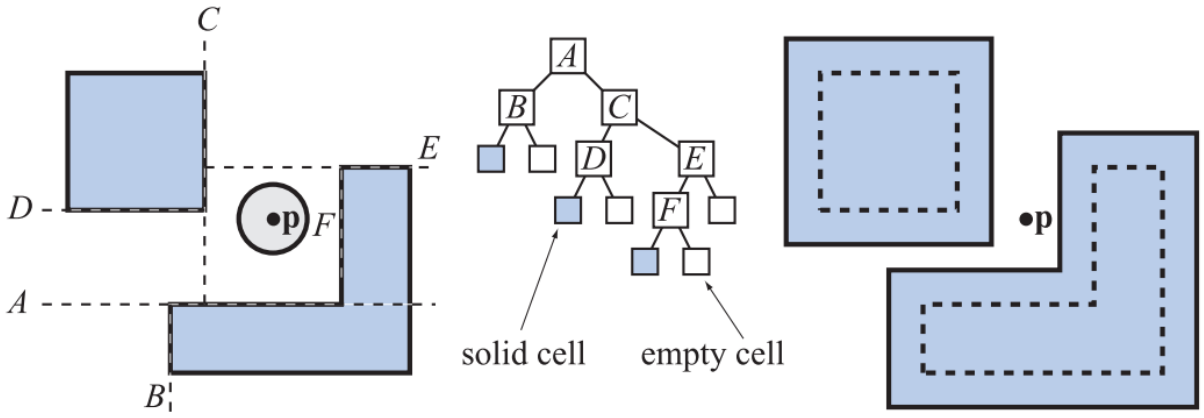


图 25.14：左侧是从上方进行观察（俯视图）的一些几何物体（蓝色）。中间展示了对应的 BSP 树。想要使用一个原点为 $\mathbf{p}$ 的圆来与这个 BSP 树进行测试，会根据圆的半径来对 BSP 树进行向外扩张，然后点 $\mathbf{p}$ 就可以针对这个扩展后的 BSP 树来进行测试了，如右侧所示。请注意，这些角应当是圆角的，右图中的尖角是算法所引入的近似。

每个碰撞查询都是动态执行的，因此可以将一棵 BSP 树用于任意大小的球体。假设一个平面是 $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$，而调整后的平面是 $\pi' : \mathbf{n} \cdot \mathbf{x} + d \pm r = 0$，其中 r 的符号取决于你将在继续平面的哪一侧来进行测试和遍历，从而寻找碰撞。假设此时角色应当位于平面的正半空间中，即 $\mathbf{n} \cdot \mathbf{x} + d > 0$，那么我们必须从 d 中减去半径 r。请注意，这里的负半空间被认为是“固体”的内部，即角色无法穿过的地方。

使用一个球体并不能对游戏中的角色进行很好地近似，但是将几个球体组合起来，可能会表现得不错[42]。使用角色顶点的凸包，或者是一个围绕角色的圆柱体效果会更好。为了能够使用这些的 BV，平面方程中的 d 必须进行不同的调整。为了在 BSP 树上针对一组顶点的移动凸包 S 进行测试，需要将方程 25.8 中的标量值，添加到平面方程的 d 值中[65]：

$$-\max_{\mathbf{v}_i \in S} \left(\mathbf{n} \cdot (\mathbf{v}_i - \mathbf{p}_0) \right) \tag{25.8}$$

同样，这里的负号代表了我们假设角色位于平面的正半空间中。其中点 $\mathbf{p}_0$ 可以是任何适合作为参考点的顶点。对于球体而言，球体的中心的当然是不二选择；对于角色而言，可以选择一个靠近脚的点，或者位于肚脐的点。有时这种选择可以对方程进行简化（就像球心那样）。正是这个点 $\mathbf{p}_0$ 针对调整后的 BSP 树进行了测试。对于动态查询而言，即在角色在一帧中，从一个点移动到了另一个点时，可以将这个点 $\mathbf{p}_0$ 作为线段的起点。假设角色在一帧中的移动向量为 $\mathbf{w}$，那么这条线段的端点是 $\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{w}$。

圆柱体可能是更加实用的，因为它在可以进行更快测试的同时，仍然能够对游戏中的角色进行良好近似。然而，对于圆柱体而言，推导调整平面方程的值较为复杂。对于这个算法，我们通常会将针对 BSP 树包围体（在本例中是球体、凸包和圆柱体）的测试，转换为使用调整后的 BSP 树，对点 $\mathbf{p}_0$ 进行测试。这与 Minkowski 和（章节 25.11.3 和章节 25.3.2）是一样的。然后，为了将其扩展到一个移动的物体，针对点 $\mathbf{p}_0$ 的测试会被替换为从点 $\mathbf{p}_0$ 到目的地点 $\mathbf{p}_1$ 的线段测试.

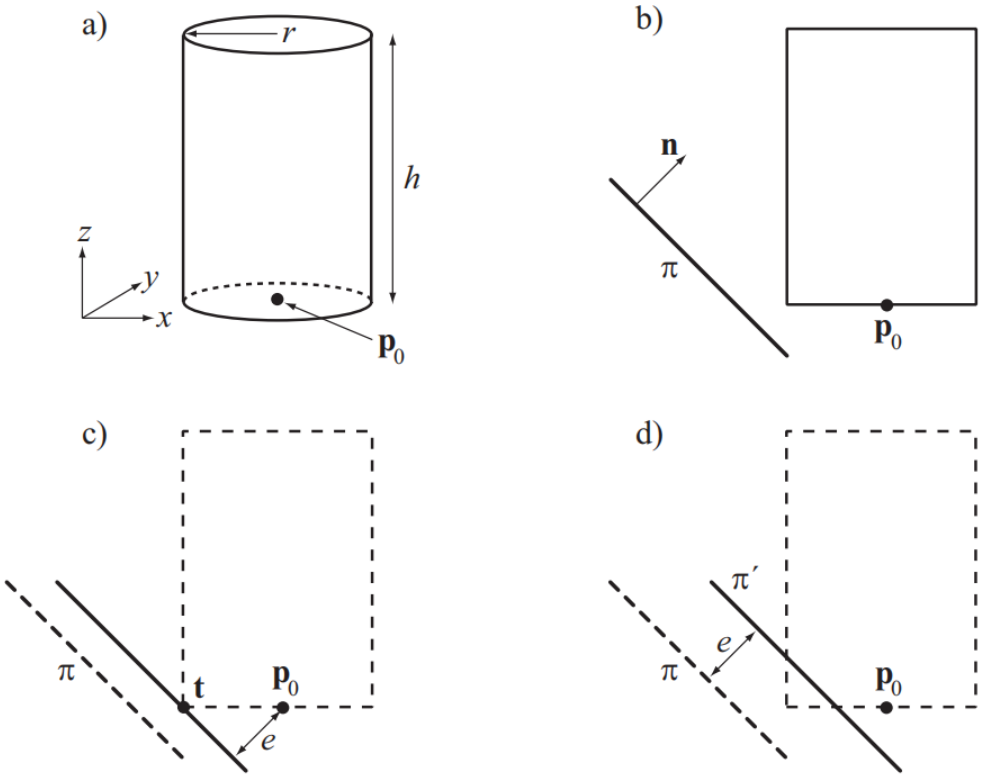


图 25.15：图（a）是一个高度为 h，半径为 r，参考点为 $\mathbf{p}_0$ 的圆柱体。图像序列（b）–（d）展示了如何针对一个平面 π，来对一个圆柱体（从侧面进行观察）进行测试。这个测试可以进行转换，将其变为使用一个新平面 π'，来对点 $\mathbf{p}_0$ 进行测试。因为点 $\mathbf{p}_0$ 位于在新平面 π' 的正半空间中，因此在这种情况下，二者并没有发生重叠。

我们为一个圆柱体推导了这样的测试，其性质如图 25.15 左上角所示，其中参考点 $\mathbf{p}_0$ 位于圆柱体的底部中心。图 25.15（b）展示了我们需要解决的问题：在平面 π 上对圆柱体进行测试。在图 25.15（c）中，我们对平面 π 进行移动，使其几乎不接触圆柱体，并计算此时点 $\mathbf{p}_0$ 到移动后平面的距离 e。图 25.15（d）使用这个距离 e 将平面 π 移动到新位置 π'。因此，这个测试被简化为平面 π' 与点 $\mathbf{p}_0$ 之间的测试。这个 e 值是每帧动态计算的，在实践中，我们首先计算一个从点 $\mathbf{p}_0$ 到点 $\mathbf{t}$（移动后平面与圆柱体相接触的位置）的向量。结果如图 25.15（c）所示。接下来，实用下列方程来计算 e：

$$e = |\mathbf{n} \cdot (\mathbf{t} - \mathbf{p}_0)| \tag{e}$$

现在剩下的任务就是计算这个点 $\mathbf{t}$ 了。首先， $\mathbf{t}$ 的 z 分量（即圆柱体的轴向方向）很简单，如果若 $n_z > 0$ ，则 $t_z = p_{0z}$ ，即点 $\mathbf{p}_0$ 的 z 分量；否则， $t_z = p_{0z} + h$ 。实际上这些 t_z 值对应了圆柱体的底部平面和顶部平面，如果 n_x 和 n_y 都为零（例如：地板或者天花板），那么我们可以使用圆柱体盖子上的任意点。一个自然的选择是 $(t_x, t_y) = (p_x, p_y)$ ，即圆柱体盖子的中心点。否则，对于非垂直的 $\mathbf{n}$ 而言，下面的方程给出了圆柱体边缘上的一个点：

$$t_x = \frac{-rn_x}{\sqrt{n_x^2 + n_y^2}} + p_x, \quad t_y = \frac{-rn_y}{\sqrt{n_x^2 + n_y^2}} + p_y \tag{25.10}$$

也就是说，我们将平面法线 $\mathbf{n}$ 投影到 xy 平面上，并对其进行归一化，然后对 r 进行缩放，使其落在圆柱体的边缘上。

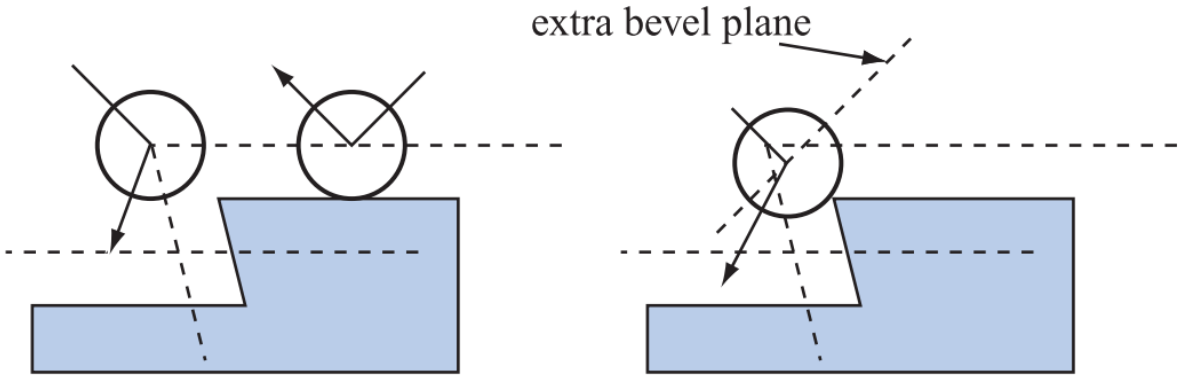


图 25.16：左侧：右边的球体进行了正确的碰撞，而左边的球体则过早地检测到了碰撞。右侧：这个问题可以通过引入一个额外的斜面来进行解决，而这个斜面实际上并不对应任何实际的几何图形。使用一个这样的平面，可以使得碰撞变得更加准确。[65]

使用这种方法可能会出现不准确的情况，图 25.16 展示了其中一个例子。可以看出，这个问题可以通过引入额外的斜面（bevel plane）来进行解决。在实践中，我们会计算两个相邻平面之间的“外”角，如果这个角度大于 90° ，则插入一个额外的平面。这样做的目的是为了改进圆角的近似值。在图 25.17 中，我们可以看到普通的 BSP 树与添加了斜面的 BSP 树之间的区别。这些斜面当然可以提高精度，但是并不能消除所有的误差。

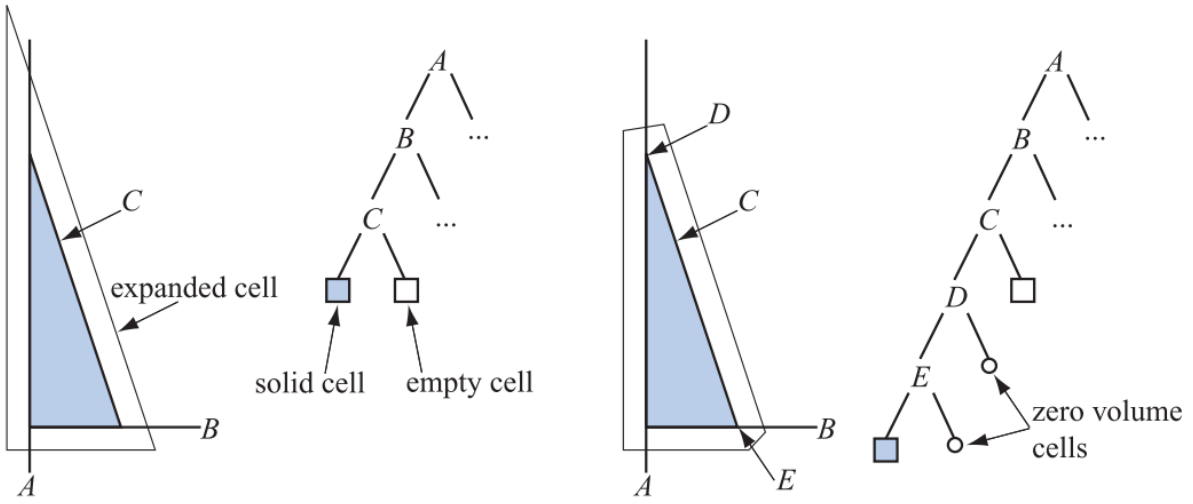


图 25.17：左侧是一个正常的单元格和它的 BSP 树。右侧在单元格中添加了一个斜面，并展示了 BSP 树中的相应更改。[65]

下面是这个碰撞检测算法的伪代码。它是在 BSP 树的根节点 N 上进行调用的，BSP 树的子节点分别是 N.negativechild 和 N.positiveechild，以及由点 $\mathbf{p}_0$ 和点 $\mathbf{p}_1$ 所定义的线段。请注意，最终的碰撞点（如果有的话）是在一个名为 $\mathbf{p}_{\text{impact}}$ 的全局变量中返回的：

```
     HitCheckBSP(N, p0, p1)
     returns ({TRUE, FALSE});
1 :  if(isEmptyLeaf(N)) return FALSE;
2 :  if(isSolidCell(N))
3 :     p_impact = p0
4 :     return TRUE;
5 :  end
6 :  hit = FALSE;
7 :  if(clipLineInside(N shift out, p0, p1, &w0, &w1))
8 :     hit = HitCheckBSP(N.negativechild, w0, w1);
9 :     if(hit) p1 = p_impact
10 : end
11 : if(clipLineOutside(N shift in, p0, p1, &w0, &w1))
12 :    hit = HitCheckBSP(N.positivechild, w0, w1);
13 : end
14 : return hit;
```

如果我们到达了一个叶子节点，并且该节点位于实体一侧的话（而不是空的一侧），函数 isSolidCell 会返回 TRUE，图 25.14 中展示了空单元格和实单元格。如果线段的一部分（由移动路径 $\mathbf{v}_0$ 和 $\mathbf{v}_1$ 进行定义）位于节点的移动平面中（即位于移动平面的负半空间中），那么函数 clipLineInside 返回 TRUE。它还会将直线裁剪到节点的移动平面上，并返回结果线段 $\mathbf{w}_0$ 和 $\mathbf{w}_1$。clipLineOutside 函数与此类

似。还要注意的是，函数 clipLineInside 与函数 clipLineOutside 所返回的线段会相互重叠，这样做的原因如图 25.18 所示，图 25.18 还展示了这条线段是如何被裁剪的。第 9 行中我们设置 $\mathbf{v}_1 = \mathbf{p}_{\text{impact}}$，这只是一个优化。如果我们找到了一个碰撞点（即一个潜在的撞击点 $\mathbf{p}_{\text{impact}}$），那么超过这个点之外的任何东西我们都不需要进行测试了，因为我们想要的是第一个碰撞点。在第 7 行和第 11 行中，节点 N 被“移出（shift out）”而不是“移入（shift in）”。这里的移位指的是先前为球体、凸壳和圆柱体所推导的、调整后的平面方程。

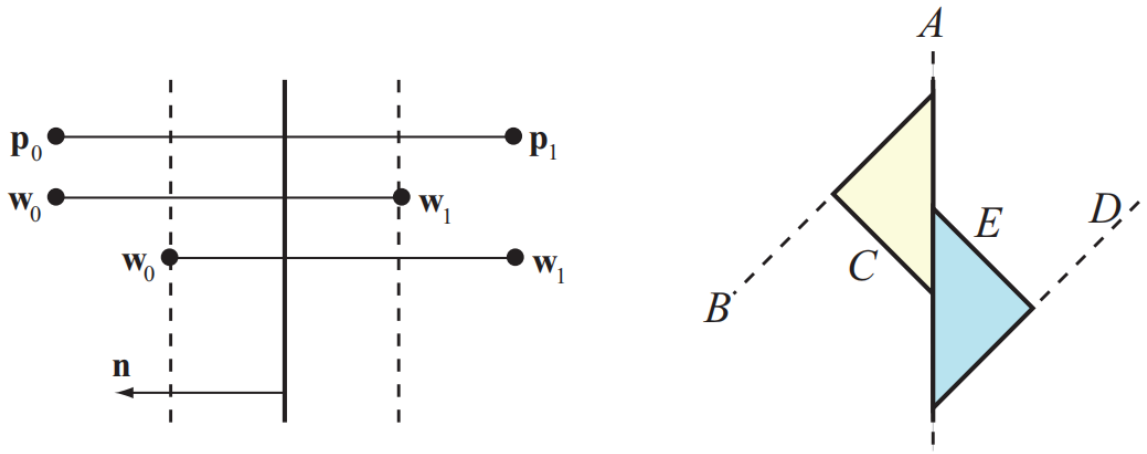


图 25.18：在左侧，由点 $\mathbf{p}_0$ 和点 $\mathbf{p}_1$ 所定义的线段，被一个由法线 $\mathbf{n}$ 所定义的平面裁剪，这个平面的动态调整使用虚线进行表示。函数 clipLineInside 和 clipLineOutside 会返回由 $\mathbf{w}_0$ 和 $\mathbf{w}_1$ 所定义的线段。请注意，这三条线段应当具有相同的 y 坐标，但是为了展示得更加清晰，使用图中的方式进行展示。右侧展示了一个示例，解释了为什么这些线条应当被裁剪成左边所示的样子。BSP 树中的节点 A 既属于左边的三角形，也属于右边的三角形。因此，有必要在两个方向上对它的平面进行移动。

该方案的优点在于，只需要使用一棵 BSP 树即可完成对所有角色和物体的碰撞测试。另一种方法是为每个不同的半径和不同物体类型，存储不同的 BSP 树。章节 25.11 中介绍了其他的动态相交测试方法。

25.6 限时碰撞检测

假设当观众抬头看向天空，此时由于渲染压力较小，某个游戏引擎会在 14 毫秒内渲染整个场景；但是当观众看向地平线的时候，由于渲染压力较大，此时的渲染需要 30 毫秒来完成。显然，不同的帧渲染时间将会得到相当不同的帧率，波动的帧率通常会对用户产生干扰。章节 19.9.3 中介绍了一种试图实现恒定帧率的渲染算法。这里我们将使用另外一种方法，被称为限时碰撞检测（time-critical collision

detection），如果某些应用程序中也使用了 CD，那么它们也可以使用这种方法。之所以称其为“限时（time-critical）”，是因为 CD 算法需要在一个特定的时间范围内（例如 9 毫秒）完成任务，并且它必须要在这个时间内完成。在 CD 中使用这种算法的另一个原因是，它对因果关系的感知至关重要[74, 75]，例如：快速检测一个物体是否会导致另一个物体发生移动。

以下算法由 Hubbard [42]所提出，其思想是按照广度优先（breadth-first）的顺序，对 BVH 进行遍历。广度优先意味着，在下降到 BVH 的下一层之前，我们需要访问树中某一层的所有节点。这与深度优先（depth-first）的遍历方式有所不同，深度优先遍历会以最短的方式到达叶子结点（如章节 25.2.2 中的伪代码所示）。图 25.19 展示了这两种遍历方式的区别。使用广度优先遍历的原因在于，我们可以同时访问节点的左子树和右子树，也就是说先去访问包含整个物体的大型 BV。而在使用深度优先遍历时，我们可能只能够访问左子树，因为算法可能已经耗尽时间了。当我们无法得知是否有足够的时间来遍历整棵树的时候，那么至少对两棵子树都进行一些遍历会更好一些。

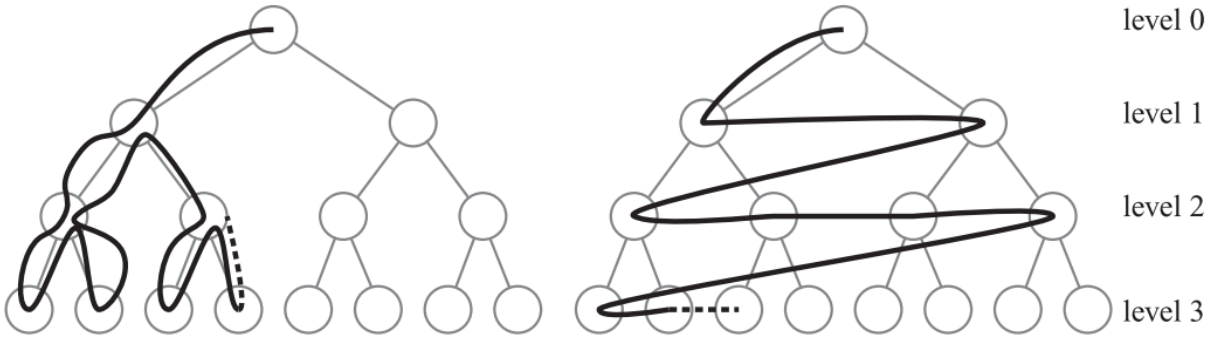


图 25.19：深度优先遍历（左）与广度优先遍历（右）。在碰撞检测中，当对 BVH 进行遍历的时候，通常会使用深度优先遍历，但是对于限时 CD 而言，则会使用广度优先遍历。

该算法首先会找到所有 BV 相互重叠的物体对，例如使用章节 25.1 中的算法。这些 BV 对被放入一个叫做 Q 的队列中。在下一阶段中，我们会从队列中取出第一个 BV 对，并对它们的子 BV 进行测试，如果它们发生重叠，则将子 BV 对放在队列的末尾。然后继续对队列中的下一个 BV 对进行测试，直到整个队列为空（在这种情况下我们就完成了整个树结构的遍历）或者时间耗尽[42]。

另一种相关的方法是给每个 BV 对都指定一个优先级，并根据这个优先级来对队列进行排序。这种优先级可以基于可见性、偏心率、距离等因素。Dingliana 和 Osullivan 描述了计算近似碰撞响应和近似碰撞接触判定的算法[19]，对于限时 CD 而言，这是非常需要的，因为时间可能会在树结构遍历结束之前就耗尽了。Mendoza 和

Osullivan 还提出了一种针对变形物体的限时 CD 算法[67]。Kulpa 等人[51]对 LOD 技术的使用进行了用户研究，并提供一种可以以最不易察觉的方式来避免碰撞的系统。

25.7 可变形模型

到目前为止，本文的主要重点都是静态模型或者刚体动画模型。但是实际上还有其他类型的运动，例如水面上的波浪或者一块在风中摇曳的布，这种类型的运动通常无法使用刚体运动来进行描述；相反，我们可以将每个顶点都视为一个随着时间变化的独立向量函数。这种模型的碰撞检测的成本通常要更高。

假设在变形过程中，物体的网格连通性保持不变，那么就有可能利用这一特性来设计出一种巧妙的算法，实际上一块布在风中摇曳时所发生的就是这种类型的变形（除非这块布会以某种方式被撕裂）。作为一个预处理阶段，首先需要建立一个初始的 BV 树，当物体发生变形的时候，我们并不会对这棵树进行重建，而是直接对该物体的包围体进行重新调整，使其能够与变形后的几何形状相适应[5, 56]。通过使用 AABB，物体的 BV 可以进行快速重新计算，这个操作十分高效（与 OBB 相比）。此外，将 k 个子 AABB 合并到父 AABB 的速度也很快，并且能够得到一个最优的父级 AABB。但是一般来说，任何类型的 BV 其实都是可以使用的。Van den Bergen 对他的树进行了一定的组织，这样所有的 BV 都会被分配并放置在一个数组中，其中一个节点的索引总是会低于其子节点的索引[5, 7]。通过这种方式，可以通过从数组的末尾向前进行遍历，并在每个节点上重新计算各自的 BV，来完成一个自下而上的更新。这意味着，叶子结点的 BV 首先会被重新计算，然后利用这些新计算的 BV，对其父节点的 BV 进行重新计算，并以此类推，直到树的根结点被更新完成。根据相关论文的报告，这种重整（refit）操作的速度大约是从零开始构建树的 10 倍[5]。

然而，需要注意的是，在通常情况下，树中只有很少的 BV 需要进行更新。因为它们中的大多数都不会在碰撞查询期间进行使用。因此，提出了一种自下而上（bottom up）和自上而下（top–down）混合式的树结构更新算法[56]。其核心思想是对较高层次的节点（包括根节点）使用自下而上的更新，这意味着在每一帧中，只有较高层次的 BV 会被更新。这样做的理由在于，上层节点通常会修剪掉大部分的几何形状。对于这些更新后的上层节点，将会与其他树结构（也可能变形和更新）来进行重叠测试。对于没有发生重叠的节点而言，我们可以跳过对其子树的更新，从而节省大量工作。另一方面，对于发生重叠的节点，我们会使用一个自上而下的更新策略，在对树进行遍历的时候，来对其子树中的节点进行更新。这里当然也可以使用自下而上的方法，但是自上而下方法被发现是更加高效的[56]。使用自下而上的方法对前 $n/2$ 个上

层节点进行更新，使用自上而下的方法对后 $n/2$ 个下层节点进行更新，我们可以获得较为良好的结果，最终结果如图 25.20 所示。为了对一个节点进行自下而上的更新，每个节点都会记录各自完整子树中的顶点信息，我们对这个列表进行遍历，从而计算其最小 BV。当使用自上而下更新的时候，在 BV 更新之后，会立即进行重叠测试，这样如果没有发现重叠，那么就可以结束遍历，这进一步提高了效率。初步测试表明，这种方法要比 van den Bergen 的方法快 4 到 5 倍[56]。

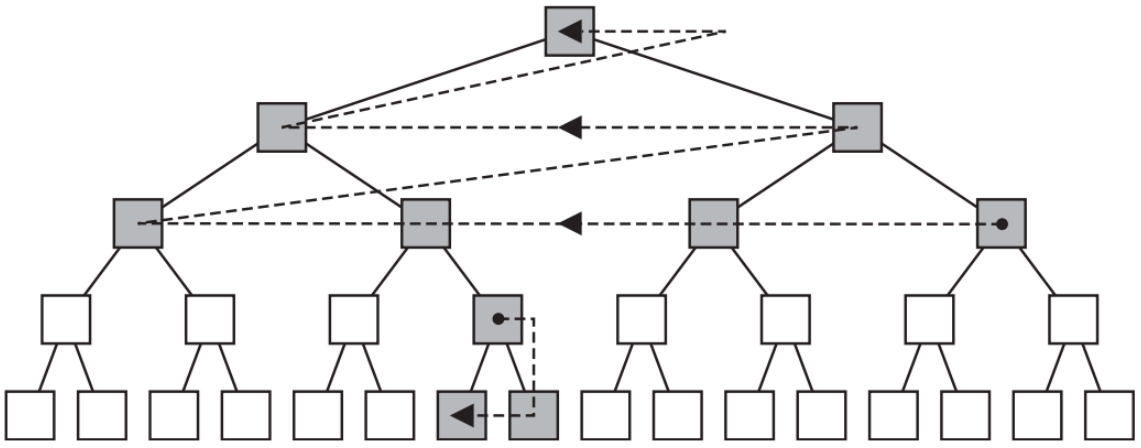


图 25.20：自下而上和自上而下的混合式树结构更新方法。在上层使用自下而上的策略进行更新，只有在遍历树的过程中到达的更深层次的节点时，才会使用自上而下的方法进行更新。

有些时候，我们可能已经拥有一些关于变形类型的已知信息，根据这些信息，我们可以设计出更加高效的算法。例如：如果一个模型使用变形（morphing）技术（章节 4.5）来进行变形，那么我们可以在 BVH 中对 BV 进行变形（即混合），就像是对实际几何物体进行变形一样[57]。这样做并不能在变形过程中创建最佳的 BV，但是可以保证这些 BV 始终包含变形后的几何图形。这种更新也可以自上而下地进行，即只在需要进行更新的地方进行更新即可。这种技术可以用于 AABB、k-DOP、球体。根据 k 个不同的 BV 计算一个变形后的 BV，其开销为 $O(k)$，但是这里的 k 通常会很小，可以看作是一个常数。James 和 Pai [43]提出了一种简化变形模型的框架，这个模型由位移场（displacement field）的组合进行描述，这可以大大提高性能表现。Ladislav 和 Zara 为蒙皮模型也提供了类似的 CD 技术[52]。

然而，如果物体的运动是完全非结构化的，并且包含破碎物体的时候，这些方法就不起作用了。最近的一些技术试图对 BVH 中的子树与底层几何物体的匹配程度进行追踪，并且仅在需要的时候，使用一些启发式的方法来对它们进行重建[58, 94]。

另一种用于可变形物体的通用方法是，首先为两个需要测试碰撞的物体，各自计算一个周围的最小 AABB [83]。如果它们发生了重叠，那么此时再去计算重叠的 AABB 区域，即 AABB 的相交体积。只有在重叠区域内才会发生碰撞，并为该区域内的所有

三角形都创建一个列表。然后使用一个包围整个场景的八叉树（章节 19.1.3）。接下来的想法是在八叉树的节点中插入这些三角形，如果某个叶子节点中同时包含了来自两个物体的三角形，则对这些三角形进行相交测试。在这个过程中可以应用一些优化方法。首先，这棵八叉树并不需要进行显式构建。当某个节点获得一个三角形列表的时候，会根据其 8 个子节点来对这些三角形进行测试，并创建 8 个新的三角形列表。这个递归会在叶子节点处结束，在叶子节点中可以进行三角形与三角形之间的相交测试。其次，如果这个三角形列表中的三角形都来自同一个物体时，这个递归就可以直接结束。为了避免对一对三角形进行多次测试，我们需要一个检查表，来对测试过的三角形对进行追踪。当重叠区域较大、或者重叠区域内存在较多数量的三角形时，这个方法的效率可能会受到一定影响。

另一种方法是使用 GPU 的曲面细分功能来检测碰撞[72]。首先找到两个物体之间的重叠区域，然后对该区域内的所有面片，都采用每个体素 1 bit 的细分法进行体素化，然后使用这个体素网格来检测碰撞。这个方法还可以被扩展到处理位移映射，例如：由于汽车在沙地中行驶所留下的痕迹[79]。

25.8 连续碰撞检测

通常，碰撞检测只会在每帧中进行一次，并且都是在离散的时间内进行的。章节 25.11 中介绍了动态的相交测试，它会对简单物体之间的碰撞时间进行计算，进行此类测试的原因是为了避免奇怪的瑕疵，例如：如果有一个球被高速地扔向一堵墙壁，在某一个时刻，球可能会出现在墙壁的前面，而在下一帧中，由于球的速度过快，它会直接移动到墙壁的后面，并躲过了与墙壁的碰撞检测。连续碰撞检测（continuous collision detection，CCD），与动态相交测试类似，旨在消除这类瑕疵。

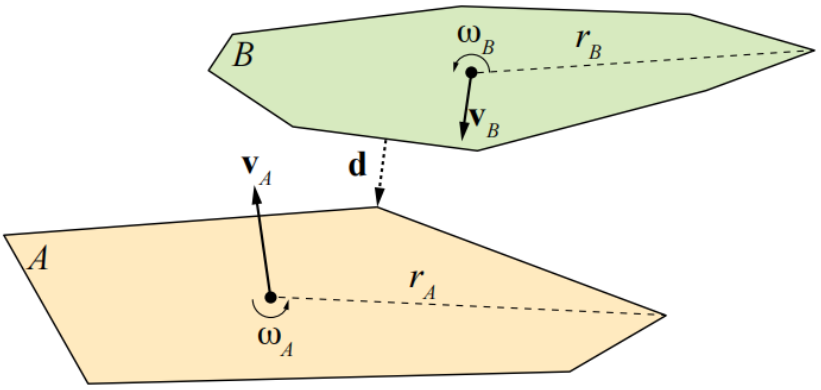


图 25.21：方程 25.11 中所使用的连续碰撞检测的符号表示。[14]

在这里，我们将介绍一种在游戏行业中所使用的方法。我们希望确定两个凸物体 A 和 B 何时发生碰撞，每个物体都有一个线性速度 $\mathbf{v}$ 和角速度 ω，半径 r 是物体质心到最远顶点的距离，图 25.21 中展示了这种符号表示方法。首先，使用 GJK 算法（章节 25.3.2）来计算 A 和 B 之间的最短距离向量 $\mathbf{d}$。然后，我们的想法是计算一个时间 Δt，可以保证两个物体在这个时间间隔内不会发生碰撞。这是可以证明的[14, 97]：

$$\underbrace{\left((\mathbf{v}_B - \mathbf{v}_A) \cdot \frac{\mathbf{d}}{\|\mathbf{d}\|} + \|\omega_A\| r_A + \|\omega_B\| r_b\right)}_{c} \Delta t \leq \|\mathbf{d}\| \qquad (25.11)$$

其中 c 是沿方向 d 的速度边界。如果物体 A 和物体 B 都不发生旋转，那么 $c = (\mathbf{v}_B - \mathbf{v}_A) \cdot \mathbf{d}/\|\mathbf{d}\|$，即投影在方向 d 上的相对速度，这意味着使用经典公式，即速度乘以时间就可以获得这个距离。其中的旋转项（使用 ω）对由于旋转所引起的最大变化进行了保守估计。因此，不会发生碰撞的最大时间量为 $\Delta t = \|\mathbf{d}\|/c$。如果这个时间 Δt 大于两帧之间的时间，那么 A 和 B 就不会发生碰撞。如果这个时间 Δt 小于等于两帧之间的时间，那么则可以在这个 Δt 期间将两个物体移动到它们所能到达的、尽可能远的地方，然后再次使用算法，直到收敛到某个容忍范围。由于这个时间估计 Δt 是保守的，因此该算法也被称为保守推进（conservative advancement）。Catto 使用一些寻根技术来提高性能表现，并在《暗黑破坏神 3（Diablo 3）》[14]中使用了这种技术。Zhang 等人[97]描述了一种处理非凸多面体的方法。

25.9 碰撞响应

碰撞响应（collision response）是为了避免物体（异常） 相互穿透而应当采取的行动，例如：假设一个球体正在向着一个立方体移动，当这个球体第一次碰到立方体的时候（这是由碰撞检测算法进行确定的），我们希望球体的运动轨迹（例如：速度方向）能够发生相应的改变，这样它们看起来就像是发生了碰撞一样。这是碰撞响应技术所负责实现的任务，它已经是并且仍然是一个需要深入研究的主题[2, 18, 38, 69, 71, 93]。碰撞响应是一个复杂的话题，本小节中只会介绍一些最简单的技术。

在章节 25.11.1 中，我们介绍了一种计算球体与平面碰撞时间的精确方法。在这里，我们将会解释在发生碰撞的时候，球体的运动发生了什么样的变化。

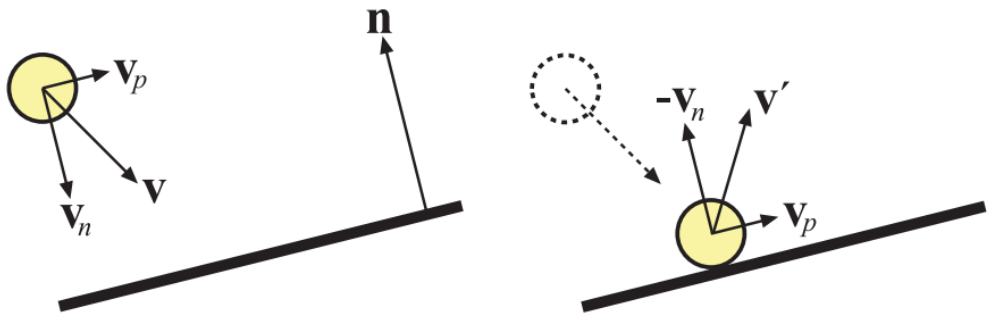


图 25.22：球体接近平面时的碰撞响应。在左边，速度向量 $\mathbf{v}$ 被分解成了两个分量，分别是 $\mathbf{v}_n$ 和 $\mathbf{v}_p$ 。右边展示的是完全弹性碰撞，新的速度为 $\mathbf{v}' = \mathbf{v}_p - \mathbf{v}_n$ 。对于弹性较小的碰撞而言， $-\mathbf{v}_n$ 的长度可以适当减小。

假设一个球体正朝着一个平面进行移动，球体的速度向量是 $\mathbf{v}$ ，平面是 $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ ，其中平面法线 $\mathbf{n}$ 是归一化的，如图 25.22 所示。为了能够计算出这个最简单的响应，我们将速度向量表示为：

$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_p, \text{ where } \mathbf{v}_n = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}, \text{ and } \mathbf{v}_p = \mathbf{v} - \mathbf{v}_n \quad (25.12)$$

根据这个表示，那么碰撞后的速度向量 $\mathbf{v}'$ 是[54]：

$$\mathbf{v}' = \mathbf{v}_p - \mathbf{v}_n \quad (25.13)$$

在这里，我们假设碰撞响应是完全弹性的。这意味着不会发生动能损失，即响应是“完美弹性的（perfectly bouncy）”。而在通常情况下，球体会在发生碰撞时轻微变形，使得一些能量转化为热量，因此会损失一些能量。这可以使用恢复（restitution）系数 k （有时候也记作 ϵ ）来进行描述。当 $k \in [0, 1]$ 时，平行于平面的速度 $\mathbf{v}_n$ 保持不变，而 $\mathbf{v}_n$ 则会受到抑制：

$$\mathbf{v}' = \mathbf{v}_p - k\mathbf{v}_n \quad (25.14)$$

方程 25.14 就是碰撞的经验“定律”。随着 k 值的不断减小，能量的损失也就会越来越大，即碰撞的弹性越来越小。当 $k = 0$ 时，碰撞后的运动会平行于平面，所以球看起来就是在平面上滚动。

更加复杂的碰撞响应是基于物理模拟的，这个过程涉及到创建一个方程组，并使用常微分方程（ordinary differential equation，ODE）求解器进行求解。在这种算法中，需要一个碰撞点坐标与该点的法线数据。有兴趣的读者可以参考 Witkin 等人

[93]的 SIGGRAPH 课程讲义，以及 Dingliana 等人[18]的论文。Catto [15]描述了如何将连续冲量（sequential impulses）应用于迭代求解器中。此外，Osullivan 和 Dingliana 所提出的实验表明，人类是很难判断碰撞响应是否正确的[74, 75]。当涉及更多维度的时候尤其如此（在一维上判断碰撞响应是否正确，要比在三维上更加容易）。为了构建一个实时算法，他们发现，当没有足够的时间来计算精确响应的时候，可以使用一个随机的碰撞响应。相关研究发现，这些随机响应和更加准确的响应一样可信。

25.10 粒子

在本小节中，我们将介绍两种用于粒子碰撞检测的方法。其中第一种方法用于粒子系统中（章节 13.8），通常用作特殊效果；另一种方法则更多地用于物理模拟中，例如模拟流体和烟雾，在这种方法中，物质会使用大量的微小粒子进行近似。这两种方法密切相关，这些技术有时候可以同时用于两种类型。

25.10.1 粒子系统

我们首先会介绍一种基于深度缓冲的、廉价的、近似的粒子碰撞系统[84]。它与屏幕空间环境光遮蔽方法（章节 11.3）的思路相同。其基本思想是提供一种方法，能够在极短的时间内处理数万个粒子，这个系统可以用于雨滴、火花、岩屑和飞溅等效果。每个粒子都要与场景进行碰撞测试，这里所使用的近似值是从相机处渲染的深度缓冲。这意味着粒子只能与屏幕上的可见表面发生碰撞。为了避免粒子直接穿过一个可见表面，我们会在发生碰撞的时候，假设每个表面都具有一定的厚度。当碰撞发生时，我们会从法线缓冲中获取法线信息；对于到达表面上的入射粒子，首先会根据粒子的速度向量，将其在深度缓冲中进行反向移动。然后计算粒子的反射方向，并向该方向进行移动。从背后移动过来的粒子会被直接销毁，这样做可以避免它们“穿过”场景的可见表面。

使用深度缓冲当然存在一些缺点，例如：只能在肉眼可见的表面上发生碰撞，但是使用这种方法可以很好地实现近似效果。如果需要更高的精度，还可以使用符号距离场（signed distance field，SDF）来进行碰撞检测，我们在章节 17.3 中进行了相应的介绍。使用 SDF 的主要优点在于，可以为场景的不可见部分也创建相应的 SDF，因此这种表示通常会存储在一个三维纹理中。Fisher 和 Lin [27]使用 SDF 来计算变形模型之间的穿透距离。Furhmann 等人[29]仅使用 SDF 来表示刚体物体，然后使用 SDF 来对变形物体的粒子进行测试。进入 SDF 内部的粒子会被移动回表面本身。还可以使用一些更加复杂的碰撞检测方法和碰撞响应方法。Di Donato [17]在移动设备

上，使用 ASTC 压缩的体积纹理，来代表用于粒子碰撞检测的场景体素化表示，并与变换反馈一起，存储下一步的模拟结果。

25.10.2 粒子的物理模拟

在这里，我们将简要介绍如何使用粒子来进行物理模拟，例如模拟一个空间内的水体，如图 25.23 所示。通常会将这种物质表示为一个大的粒子集合，其中的每个粒子可以具有任何位置，只要它尊重（respect）其他粒子即可，例如：两个粒子之间不能彼此过于接近。粒子需要考虑的其他物理量包括：重力、速度以及用户提供的力等。这被称为 Lagrangian 方法或者基于粒子的方法[35]。由于粒子受到的力会随着距离的增加而减小，因此只让粒子与一定半径内的粒子发生相互作用就足够了。因此，我们可以使用任何类型的空间数据结构，来加速发现粒子间碰撞的过程。例如：可以快速构建 AABB 树（章节 25.1.3），构建一个均匀或者分层的网格（章节 25.1.3）。

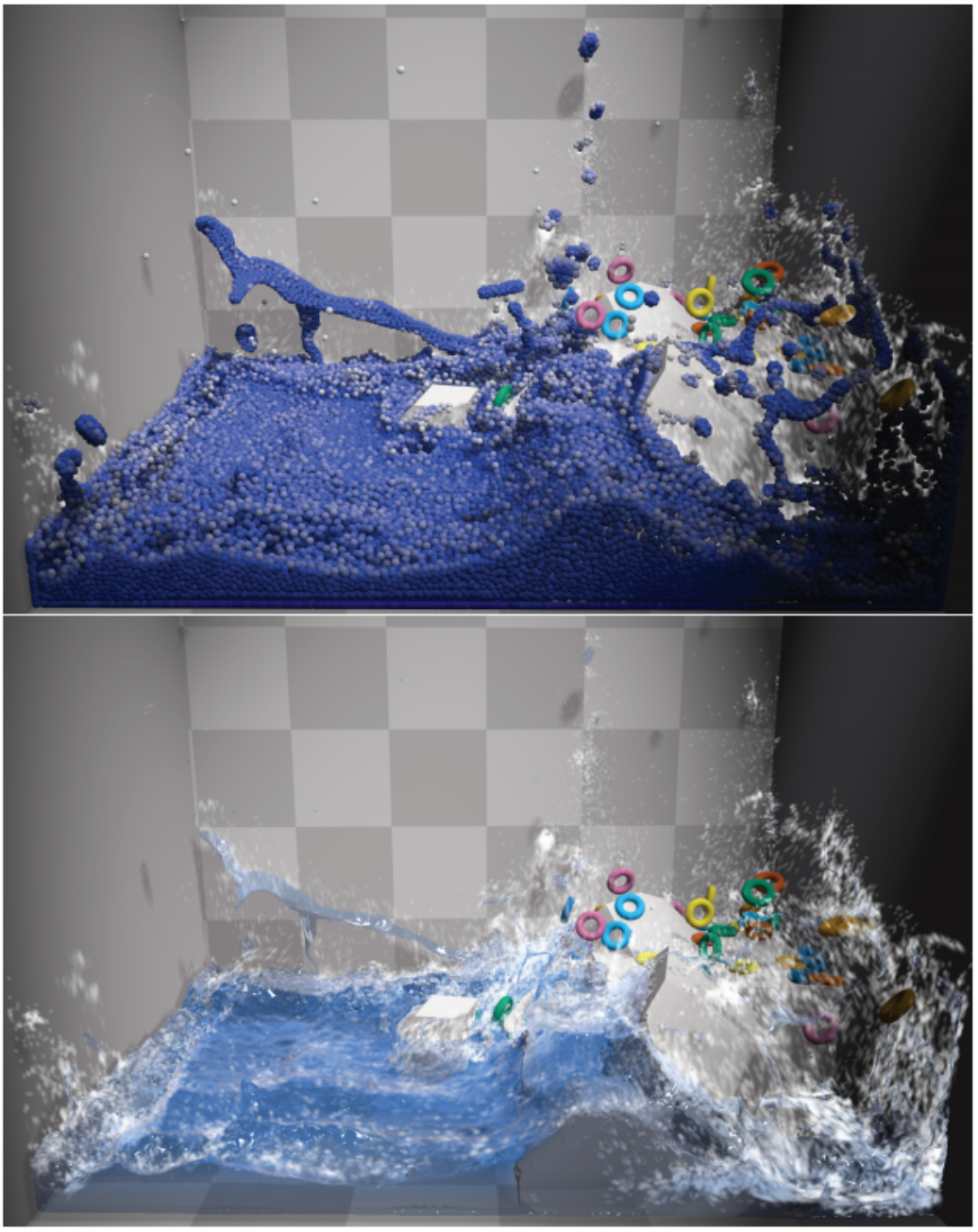


图 25.23：上图：使用粒子模拟的水体表面，其中每个粒子都被可视化为了一个蓝色球体。水体会与环境发生相互作用，包括一些岩石，环面（tori），墙壁和地板等。下图：水体的表面由粒子模拟产生，并使用运动模糊和折射来进行渲染。

Macklin 等人[64]提出了一种在一个统一框架内处理液体、气体、刚体、布料和变形固体的方法，以便系统中所有类型的物质都可以对所有其他类型的物质产生影响。他们使用符号距离场来快速解决碰撞问题。他们提出了一个具有实时性能的 GPU 实现，图 25.23 展示了一个例子。

25.11 动态相交测试

在第 22 章中，我们只考虑了静态物体之间的相交测试，这意味着所有涉及的物体，在测试的时候都没有发生移动。然而，这并不总是一个真实世界中的场景，尤其是我们会在离散的时间来对每帧进行渲染。例如：离散测试意味着，一个球在时间 t 时位于封闭门的这一侧，而在时间 $t + \Delta t$ （即下一帧）时，它会移动到门的另一侧，静态相交测试并不会注意这期间的任何碰撞。一种解决方案是在 t 和 $t + \Delta t$ 之间，以均匀间隔来进行多次测试。这种方案将会增加计算负荷，但是仍然可能会错过相交情况。为了达到测试目的，我们可以增加门的远端厚度，但是如果球体移动得足够快，那么这种技术也可能会失败[14]。针对这一问题，相关研究人员设计了一种动态相交测试（dynamic intersection test）方法。章节 25.5 中对 BSP 与圆柱体的动态相交测试进行了深入的检查。本小节中将会提供其他常见的动态相交测试。更多信息可以在 Ericson 的书籍[25]和 Eberly 的书籍[22]中找到，Catto [14]和 Gregorius [37]的展示中也包含了相关内容。

可以使用诸如轴剔除（shaft culling）之类的方法[39]，来辅助运动 AABB 的相交测试。在空间中运动的物体由两个不同时间的 AABB 进行表示，这两个 AABB 由一组小平面相连接。相交物体可以针对这个简单的凸包来进行测试。然而，包围球相交算法的计算速度要快得多，而且如果球体可以将各自的物体包裹得相当紧密，那么总体上可以更加高效。事实上，使用一小组球体来紧密包围并代表一个移动中的物体通常是十分值得的[81]。胶囊体（一个管子将两个球体连接起来）可以作为动画角色的包围体，锥形胶囊体（tapered capsule）可以用于四肢和布料的模拟。

对于一个只有平移（没有旋转）的动态相交测试而言，可以应用这样一条原则：运动是相对的。假设物体 A 以速度 $\mathbf{v}_A$ 进行移动，物体 B 以速度 $\mathbf{v}_B$ 进行移动，这两个速度都是物体在一帧内移动的量。为了对计算进行简化，我们可以假设物体 A 正在发生移动，而物体 B 则是静止的。为了对物体 B 的速度进行补偿，则物体 A 的速度被修正为： $\mathbf{v} = \mathbf{v}_A - \mathbf{v}_B$ 。因此，在接下来的算法中，只有一个物体是具有速度的。

25.11.1 球体/平面

在平面上对球体进行动态测试是很简单的。假设球体的圆心为 $\mathbf{c}$ ，半径为 r 。与静态测试相反，球体在整个帧时间 Δt 内的速度为 $\mathbf{v}$ 。因此，在下一帧中，球体将会位于 $\mathbf{e} = \mathbf{c} + \Delta t\mathbf{v}$ 处。为了简单起见，我们假设该帧从时间 0 开始， Δt 为 1。那么这个问题就变成了：在这段时间内，球体是否与平面 $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ 发生了碰撞?

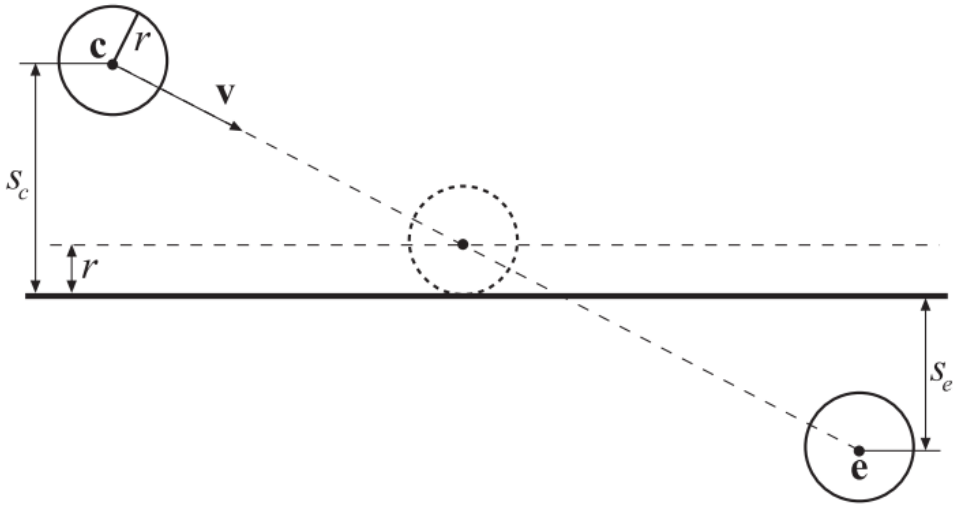


图 25.24：在球体/平面动态相交测试中所使用的符号。中间的球体展示了碰撞发生时球体所在的位置。请注意， s_c 和 s_e 都是带符号的距离。

我们将球心代入平面方程，可以得到从球心到平面的符号距离 s_c 。从这个距离中减去球体半径，就得到了球体在到达平面之前可以移动的最大距离（沿着平面法线）。如图 25.24 所示。对于终点 $\mathbf{e}$ ，计算一个类似的符号距离 s_e 。此时，如果两个球体中心都位于平面的同一侧（即 $s_c s_e > 0$ ，且 $|s_c| > r, |s_e| > r$ ，那么就不会发生相交，球体可以安全地移动到终点 $\mathbf{e}$ 。否则，球体会与这个平面相交，需要计算交点以及相交的确切时间[32]。假设球体与平面第一次接触的时间为 t ， t 的计算方程如下：

$$t = \frac{s_c - r}{s_c - s_e} \tag{25.15}$$

此时球心位于 $\mathbf{c} + t\mathbf{v}$ 。在这个碰撞点上，一个简单的碰撞响应是，围绕平面法线，对速度矢量 $\mathbf{v}$ 进行反射，并使用这个矢量 $(1-t)\mathbf{r}$ 来对球体进行移动，其中 $1-t$ 是从碰撞时间到下一帧之间的剩余时间， $\mathbf{r}$ 是反射向量。

25.11.2 球体/球体

对两个移动的球体 A 和球体 B 进行相交测试，等价于一条射线针对一个静止球体进行测试——这是一个令人惊讶的结果。我们通过两个步骤来说明这种等价性。首先，利用相对运动原理使得球体 B 静止。然后，我们借用一个视锥体/球体相交测试（章节 22.14.2）中的技术。在这个测试中，球体会沿着视锥体的表面进行移动，从而形成一个更大的视锥体。通过将视锥体以球体半径长度向外延伸，使得这个球体本身可

以缩小为一个点。而在这里，我们将一个球体移动到另一个球体的表面上，使用同样的方法得到一个新的球体，这个新球体的半径是两个原始球体半径的和。

因此，将球体 A 的半径加到球体 B 的半径上，我们得到了一个新的球体半径。此时的情况是，球体 B 是静止的，并且半径更大，而球体 A 则是一个沿着直线进行移动的点，即一条射线，如图 25.25 所示。

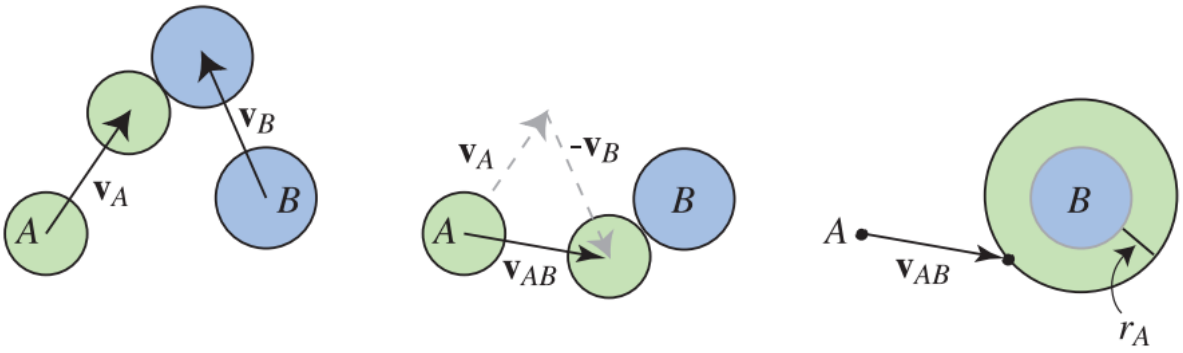


图 25.25：左侧展示了两个球体之间的移动和碰撞。在中间，通过在两个球体中减去球体 B 的速度，从而使得球体 B 其静止。请注意，球体在碰撞点上的相对位置保持不变。在右侧，将球体 A 的半径 r_A 添加到球体 B 上，然后球体 A 再减去自身的半径，这使得运动中的球体 A 变成了一条射线。

这个基本的相交测试已经在章节 22.6 中介绍过了，在这里我们直接给出最终结果：

$$\left(\mathbf{v}_{AB} \cdot \mathbf{v}_{AB}\right) t^2 + 2\left(\mathbf{l} \cdot \mathbf{v}_{AB}\right) t + \mathbf{l} \cdot \mathbf{l} - \left(r_A + r_B\right)^2 = 0 \qquad (25.16)$$

在方程 25.16 中， $\mathbf{v}_{AB} = \mathbf{v}_A - \mathbf{v}_B$ ， $\mathbf{l} = \mathbf{c}_A - \mathbf{c}_B$ 。其中 $\mathbf{c}_A$ 和 $\mathbf{c}_B$ 是球体的中心。对方程 25.16 进行整理，并写出一元二次方程的形式，相应的系数 a 、 b 和 c 为：

$$\begin{aligned} a &= \left(\mathbf{v}_{AB} \cdot \mathbf{v}_{AB}\right), \\ b &= 2\left(\mathbf{l} \cdot \mathbf{v}_{AB}\right), \\ c &= \mathbf{l} \cdot \mathbf{l} - \left(r_A + r_B\right)^2, \end{aligned} \qquad (25.17)$$

这里回顾一下一元二次方程的基本形式，并将各个系数代入其中：

$$at^2 + bt + c = 0 \qquad (25.18)$$

方程 25.18 中的两个根，可以通过一次计算直接得到：

$$q = -\frac{1}{2}\left(b + \text{sign}(b)\sqrt{b^2 - 4ac}\right) \tag{25.19}$$

当 $b \geq 0$ 时， $\text{sign}(b)$ 为 +1，否则为–1。那么这两个根就是：

$$\begin{aligned} t_0 &= \frac{q}{a}, \\ t_1 &= \frac{c}{q}. \end{aligned} \tag{25.20}$$

这种求解二次方程的形式并不是教科书中通常所介绍的，但是 Press 等人指出，它在数值上会更加稳定[77]。

我们假设球体在开始时不重叠（这可以通过一个静态的球体与球体相交测试确定）。在 $[0, 1]$ 范围内， $[t_0, t_1]$ 中的最小值就是第一次相交的时间。将 t 值代入，可得：

$$\begin{aligned} \mathbf{p}_A(t) &= \mathbf{c}_A + t\mathbf{v}_A, \\ \mathbf{p}_B(t) &= \mathbf{c}_B + t\mathbf{v}_B \end{aligned} \tag{25.21}$$

对方程 25.21 进行计算，我们可以获得每个球体在第一次接触时的位置。这个测试与前面所提到的射线与球体测试的主要区别在于，这里并没有对光线的方向 $\mathbf{v}_B$ 进行归一化。

25.11.3 球体/多边形

球体和平面的动态相交十分简单，可以直接进行可视化。球体与平面的动态相交测试，也可以像球体与球体的相交测试一样，被转换为另一种形式。也就是说，运动中的球体可以收缩为一个运动的点，从而构成一条射线；而平面则被扩展为一个厚度为球体直径的平板。这两个测试中所使用的关键思想，都是计算两个物体的 Minkowski 和。球体与球体的 Minkowski 和是一个更大的球体，其半径相当于二者的半径之和。

球体与平面的和是一个平面，并在每个方向上都以球体的半径进行加厚。实际上，任何两个体积都可以通过这种方式加在一起，尽管有时候结果难以进行描述。对于球体与多边形的动态测试而言，我们的想法是根据球体与多边形的 Minkowski 和，来对一条射线进行测试。

我们不会在这里深入介绍他的方法，但是请注意，这个球体与多边形的相交测试，等价于一条射线（由沿着直线移动的球体中心进行表示）对球体和多边形的

Minkowski 和进行测试。在这个和表面中，顶点会变成半径为 r 的球体；边缘会变成半径为 r 的圆柱体；多边形本身会被复制，然后向上移动 r ，同时向下移动 r ，从而将自身封闭，图 25.26 展示了这一点。这与视锥体与球体相交测试时，对视锥体的扩展是一样的（章节 22.14.2）。因此，所提出的算法可以被认为是，针对这个体积的各个部分，对一条射线进行测试：首先，对面向射线的多边形进行测试，然后对代表边缘的圆柱体进行测试，最后对代表顶点的球体进行测试。

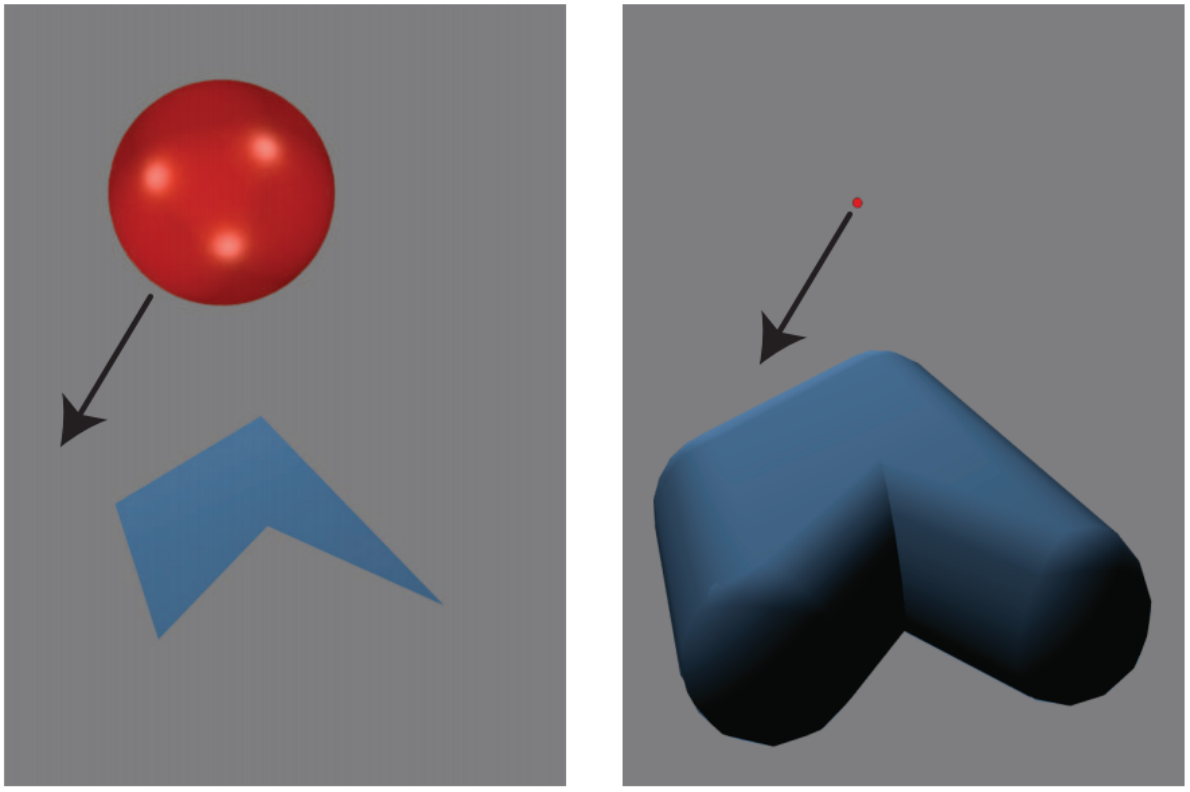


图 25.26：左图展示了一个正在向多边形运动的球体。在右图中，球体被收缩为了一个点，而多边形则进行了“膨胀”，并对这个“膨胀”版本的多边形进行射线检测。这两个相交测试实际上是等价的。

想想这个蓬松（puffy）的物体，你就会明白为什么对一个多边形最有效的测试顺序是：面（多边形）、边、顶点。在这个蓬松的物体中，面对球体的多边形并不会被物体膨胀后的圆柱体和球体所覆盖，因此对其首先进行测试，将会给出最近的可能交点，同时无需进一步测试。类似地，虽然由边缘形成的圆柱体会部分覆盖球体，但是反过来，球体只覆盖了圆柱体内的一部分，因此这里先对边缘进行测试，最后对顶点进行测试。

射线击中圆柱体的内部，等价于找到这个运动球体最后击中的相应边缘上的点，这个点实际上我们并不关心。圆柱体外部的最近交点（如果存在的话），总是要比最近的球面交点更近。因此，找到与圆柱体的最近交点就足以结束整个测试，而无需再去检

查顶点膨胀后的球体。在处理射线与这个蓬松物体的时候，相比于处理原始的球体和多边形，思考要进行的测试顺序则要容易得多（至少对我们来说）。

从这个膨胀的物体模型中我们可以看出，对于具有凹面的多边形，任何凹面位置上的顶点，都不需要针对运动球体来进行测试，因为对于这样的一个凹面顶点，它所形成的球体从外部是不可见的。利用相对运动原理和 Minkowski 和，将运动球体转换为一条射线，可以推导出动态球体与物体相交的高效测试。

静态的相交测试也可以通过对 Minkowski 和进行测试得到，例如：如果发现球体的中心就位于这个膨胀模型的内部，那么球体与三角形就会在它们的初始位置处发生相交。与之相关的 Minkowski 差，是一种更加通用的、描述任意物体包含性测试的方法（章节 25.3.2）。取差值相当于从一个物体中减去另一个物体，对于被减去的物体，其坐标值会被消去（即为 0，位于原点处）。如果两个物体相交，那么则说明它们的 Minkowski 差中包含了原点，这个点代表了一个存在于两个物体中的位置。Gregorius [36]讨论了如何利用 Minkowski 差和物体的高斯贴图，来对分离轴测试的使用进行优化，并提供了一些见解。

还有一些更加复杂的动态测试，例如三角形与三角形之间的动态相交测试。Van Waveren [91]基于顶点/多边形和边/边的碰撞测试，对多面体之间的动态碰撞检测进行了讨论。Catto [14]对保守推进（conservative advancement）的概念进行讨论，所谓保守推进，就是找到两个物体之间的安全距离，并将物体移动得更近，然后再次进行测试。但是，使用这个安全间隔将会使得收敛时间变长。她提出了一种三角形与三角形相交的双边推进算法，这是一种改进的高效求根方法。Shellshear 和 Ytterlid [82]为三角形、线段和顶点之间的距离查询提供了 SSE 优化的代码。PQP（Proximity Query Package）库提供了网格物体重叠，以及两个模型之间最小距离的相应代码。

25.11.4 动态分离轴方法

章节 22.2 中的介绍了分离轴测试（separating axis test，SAT），这种方法有助于检验凸多面体（例如 box 和三角形）之间的相互关系。这种类型的测试也可以扩展到动态查询中[11, 20, 25, 37, 80]。

请记住，SAT 方法会对一组坐标轴进行测试，从而查看这两个物体在这些坐标轴上的投影是否会发生重叠。如果所有轴上的投影都发生了重叠，那么这两个物体也会发生重叠。动态求解问题的关键是：需要将运动物体的投影间隔以 $(\mathbf{v} \cdot \mathbf{a})/(\mathbf{a} \cdot \mathbf{a})$ 的速度（详见方程 4.62），在轴 $\mathbf{a}$ 上进行移动[11]。同样地，如果在所有被测试的轴上都发

生了重叠，则这两个动态物体也会发生重叠，否则就没有发生重叠。图 25.27 展示了动态 SAT 和静态 SAT 之间的差异。

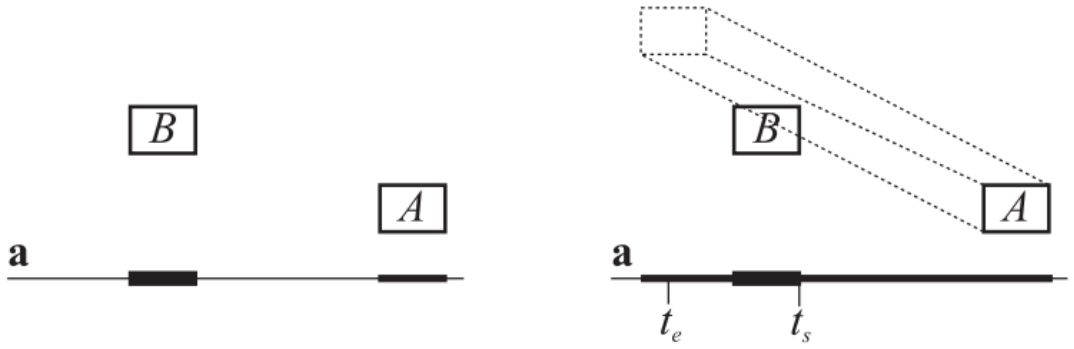


图 25.27：左侧：轴 $\mathbf{a}$ 的静态 SAT。物体 A 和物体 B 在这个轴上不会发生重叠。右侧：动态 SAT 的图解。物体 A 在发生移动，并且在移动过程中，会对物体 A 间隔在轴 $\mathbf{a}$ 上的投影进行追踪。在这里的例子中，两个物体会在轴 $\mathbf{a}$ 上发生重叠。

Eberly [20]使用了 Ron Levine 所提出的一个想法，同样也计算了物体 A 和物体 B 之间相交的实际时间。这是通过计算它们刚刚开始发生重叠的时间 t_s，以及它们停止重叠（因为这个间隔已经“穿过”了彼此）的时间 t_e 来完成的。A 和 B 之间碰撞开始的时间，发生在所有轴上的最大 t_s 处；同样，重叠结束的时间，发生在所有 t_e 值中最小的位置处。可以对 $t = 0$ 时刻进行检测，如果发现间隔没有发生重叠，并且向着相反的方向运动，那么可以不进行后续的测试，即早期的拒绝优化。此外，如果在任何时候，最大的 t_s 会大于最小的 t_e，那么说明这两个物体不会发生重叠，因此测试也可以终止。这与章节 22.7.1 中的射线与 box 相交测试类似。Eberly 为凸多面体之间的广泛测试提供了相应的代码，包括 box 与 box、三角形与 box、三角形与三角形。Gregorius [37]提出了球体、胶囊体、凸包和网格之间的动态相交测试算法。

补充阅读和资源

请参阅本书的网站 realtimerendering.com，以了解该领域中的最新信息和免费软件。CD 的最佳资源之一是 Ericson 的《Real-Time Collision Detection》一书[25]，其中包含了很多代码。在 van den Bergen 的碰撞检测书籍[9]中，对 GJK 和该书附带的 SOLID CD 软件系统进行了特别关注。Van den Bergen 为游戏程序员提供了一个关于物理的有价值演讲[10]，Catto 则提供了一个对 GJK 的愉快介绍[13]。Teschner 等人[87]对用于变形物体的 CD 算法进行了调研和综述。Schneider 和 Eberly 在他们关于几何工具的书[80]中，提出了计算不同图元之间距离的算法。

有关空间数据结构的更多信息，详见章节 19.1。除了 Ericson 的书[25]之外，Samet 所撰写的书籍[78]是一本关于空间数据结构且非常全面的参考著作。

Millington [68]、Erleben 等人[26]的书籍，以及 Eberly 的书籍[21]是碰撞响应领域中的综合指南。

Chapter 26 Real-Time Ray Tracing 实时光线追踪

> Sylvia Plath——“I wanted change and excitement and to shoot off in all directions myself, like the colored arrows from a Fourth of July rocket.”

> 西尔维娅·普拉斯——“<我最腻味的就是永恒的安全感，或者当个射箭的出发点>。我想要改变，想要兴奋，想要自己向四面八方发射，就像独立日火箭上的彩色箭头一样。”（美国自白派诗人；31 岁时自杀；1932—1963）

> 译者注：本章节中引用的参考文献编号是单独列出的，注意不要与第 1 章-第 24 章的参考文献相混淆。

与基于光栅化的技术相比（这是本书中大部分内容的主题），光线追踪（ray tracing）是一种更直接受到光物理学启发的方法，因此，它可以生成更加真实的图像。在 1999 年的第一版《Real-Time Rendering》中，我们的梦想在 2007 年到 2024 年之间，以每秒 12 帧的平均帧率来渲染《虫虫危机（A Bug's Life，ABL）》，从某种意义上来说，我们的预测是正确的。ABL 只会在几个真正需要的镜头中使用光线追踪，例如：水滴中的反射和折射。然而，由于近年来 GPU 的不断进步，已经使得通过光线追踪来实时渲染游戏场景成为可能。例如：这本书的封面（译者注：Nvidia 的星球大战 demo）就展示了一个以每秒 20 帧的速度，使用全局光照进行渲染的场景，它开始与电影故事片（feature-film）的图像质量相当。光线追踪将会彻底改变实时渲染。

在其最简单的形式中，光栅化和光线追踪的可见性确定，可以使用一个双 for 循环来进行描述。光栅化为：

```C++
for (T in triangles )
    for (P in pixels )
        determine if P is inside T
```

另一方面，光线追踪可以使用以下方式进行描述：

```C++
for (P in pixels )
    for (T in triangles )
        determine if ray through P hits T
```

某种意义上来说，这两个算法其实都很简单。然而，想要让其中任何一种算法运行得更快，我们所需要的代码和硬件，要比一张名片（译者注：章节 11.2 中也提到了这个典故）所能容纳的多得多。

> Paul Heckbert 在上世纪 90 年代的名片背面，写下了一个简单的递归光线追踪器代码。[34]

一些光线追踪器使用了空间数据结构（例如 BVH），它们具有一个重要特征，即追踪一根光线的运行时间为 $O(\log n)$，其中 n 是场景中的三角形数量。虽然这是光线追踪一个十分具有吸引力的特性，但是很明显，光栅化的复杂度也可以比 $O(n)$ 更好，因为 GPU 中有相应的遮挡剔除硬件，渲染引擎中也会使用视锥体剔除、延迟渲染和许多其他的优化技术，避免对每个图元都进行完整处理。因此，在大 O 表示法中，对光栅化算法的运行时间进行估计是一件十分复杂的事情。此外，GPU 中的纹理单元和三角形遍历单元非常快，并且在这几十年的时间里，已经针对光栅化进行了很多优化。

重要的区别在于，光线追踪可以向任何方向发射光线，而不仅仅是从眼睛或者光源等单一点上发射光线。正如我们将在章节 26.1 中所看到的，这种灵活性使得递归渲染反射现象和折射现象成为可能[89]，并且能够对渲染方程（方程 11.2）进行充分计算，这样做可以使得图像看起来更好更真实。光线追踪的这一特性也简化了内容制作流程，因为它需要的艺术家手动干预更少[20]。当在使用光栅化的时候，艺术家经常需要对他们的创作进行调整，以便能够与所使用的渲染技术一起工作。然而，在使用光线追踪的时候，图像中的噪声可能会变得十分明显，例如：对面光源进行采样时，当着色表面具有光泽时，当环境贴图被积分时，以及当使用路径追踪时，都会发生这种情况。

也就是说，想要使得实时光线追踪成为实时应用中唯一的渲染算法，很可能还需要几种其他的技术（例如降噪），才能使得图像看起来足够好。降噪技术尝试基于智能图像平均（ intelligent image averaging）来去除噪声（章节 26.5）。在短期内，光栅化和光线追踪的巧妙组合是有望实现的，也就是说，光栅化技术并不会很快消失。从长远来看，随着处理器变得更加强大，光线追踪的规模问题也会变得更好，也就是说，所能提供的计算能够和带宽越大，我们可以通过增加每像素的样本数量

（samples per pixel，spp）和光线的最大递归深度，从而可以通过光线追踪生成更好的图像。例如：由于涉及计算困难的间接光照，图 26.1 中所展示的图像，在每个像素上使用了 256 个样本。另一幅使用高质量路径追踪的图像如图 26.6 所示，其中每个像素的样本数量范围为 1 到 65536。

图 26.1：这是一个十分复杂的场景，其中存在大量的间接照明渲染，每个像素使用了 256 个样本以及 15 次的光线深度，场景中包含一百万个三角形。不过，当对图像进行放大的时候，可能仍然会在图像中看到一些噪声。场景中的一些物体由透明的塑料材质、玻璃和一些具有光泽的金属表面组成，这些材质是很难使用光栅化方法进行渲染的。

在深入研究光线追踪所使用的算法之前，我们建议您参考几个相关的章节。第 11 章的主题是全局光照，其中提供了围绕渲染方程（方程 11.2）的相关理论，并在章节 11.2.2 中，对光线追踪和路径追踪进行了基本解释。第 22 章的主题的相交测试，对于光线追踪而言，光线与物体的相交测试是必不可少的。空间数据结构可以用于对光线追踪中的可见性查询进行加速，我们在章节 19.1.1 中，以及第 25 章关于碰撞检测的相关话题中进行了描述。

26.1 光线追踪基础

回顾一下方程 22.1，一条光线被定义为：

$$\mathbf{q}(t) = \mathbf{o} + t\mathbf{d} \tag{26.1}$$

其中 $\mathbf{o}$ 是光线原点， $\mathbf{d}$ 是归一化的光线方向， t 是沿着光线的距离。请注意，在这里我们使用 $\mathbf{q}$ （而不是 $\mathbf{r}$ ）来代表光线方程，其目的是与下面所使用的右向量 $\mathbf{r}$ 区分开来。光线跟踪可以使用两个函数来进行描述，即 $\text{trace}()$ 和 $\text{shade}()$ 。核心的几何算法是 $\text{trace}()$ ，它负责找到光线和场景图元之间的最近交点，并通过调用 $\text{shade}()$ 来返回光线的颜色。在大多数情况下，我们希望找到 $t > 0$ 的交点；对于构造实体几何（constructive solid geometry，CSG）而言，我们通常也需要找到负距离的交点（位于光线背后的相交点）。

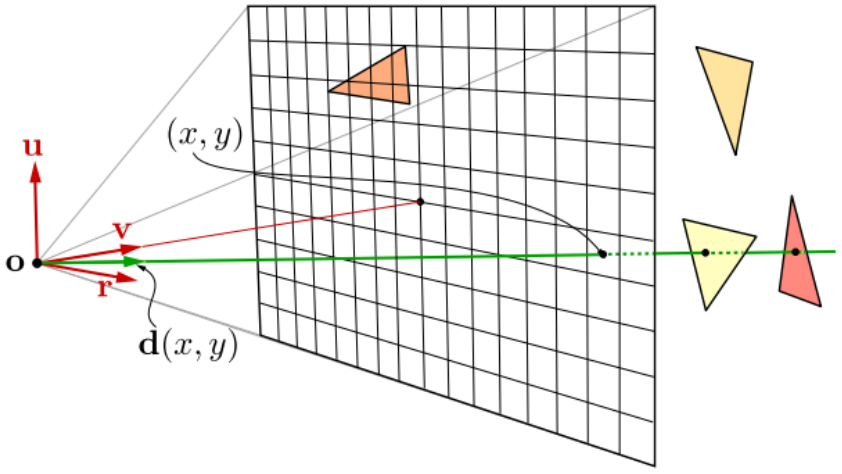


图 26.2：光线由原点 $\mathbf{o}$ 和方向 $\mathbf{d}$ 进行定义。光线追踪可以从观察点，通过每个像素构建并发射一条（或者多条）光线。图中展示的光线碰到了两个三角形，如果三角形是不透明的话，那么只有第一个三角形是值得关注的。请注意，向量 $\mathbf{r}$ （右）、 $\mathbf{u}$ （上）和 $\mathbf{v}$ （观察）用于构建样本位置 (x, y) 处的方向向量 $\mathbf{d}(x, y)$ 。

为了获得像素的颜色，我们会发射一条穿过像素的光线，计算像素的颜色并作为它们结果的加权平均值，这些光线被称为眼睛光线（eye ray）或者相机光线（camera ray）。相机的设置如图 26.2 所示。给定一个整数像素坐标 (x, y) ，其中 x 轴在图像中指向右， y 轴指向下，相机位置为 $\mathbf{c}$ ，相机的坐标系为 $\{\mathbf{r}, \mathbf{u}, \mathbf{v}\}$ （right，up，view），屏幕的分辨率为 $w \times h$ ，则眼睛光线 $\mathbf{q}(t) = \mathbf{o} + t\mathbf{d}$ 可以按照如下方法进行计算：

$$
\begin{aligned}
\mathbf{o} &= \mathbf{c}, \\
\mathbf{s}(x,y) &= af\left(\frac{2(x+0.5)}{w}-1\right)\mathbf{r} - f\left(\frac{2(y+0.5)}{h}-1\right)\mathbf{u} + \mathbf{v}, \\
\mathbf{d}(x,y) &= \frac{\mathbf{s}(s,y)}{\|\mathbf{s}(s,y)\|}
\end{aligned}
\tag{26.2}
$$

其中归一化光线方向 $\mathbf{d}$ 受 $\tan(\phi/2)$ 影响， ϕ 是相机的垂直视场角， $a = w/h$ 是相机的长宽比。请注意，这里的相机坐标系是一个左手系，即 $\mathbf{r}$ 指向右侧， $\mathbf{u}$ 是指向上方， $\mathbf{v}$ 指向远离相机的图像平面方向，即与图 4.5 中所展示的设置类似。注意， $\mathbf{s}$ 是一个用于归一化 $\mathbf{d}$ 的临时向量。由于 $(0.5, 0.5)$ 才是浮点中心[33]，因此将整数位置 (x, y) 加上 0.5，才是每个像素的中心。如果我们想要在一个像素内的任何位置发射光线，我们将会使用浮点值来表示像素的具体位置，并且不会添加这个 0.5 的偏移。

在最简单的实现中， $\text{trace}()$ 将会循环遍历场景中的 n 个图元，并将光线与其中的每个图元进行相交测试，并维护满足 $t > 0$ 的最近交点。这样做的复杂度为 $O(n)$ ，除了使用很少量的图元之外，这个速度慢得令人无法接受。为了达到每条光线 $O(\log n)$ 的复杂度，我们会使用一些空间加速结构，例如 BVH 或者 k-d 树。章节 19.1 中描述了如何使用 BVH 来对射线进行相交测试。

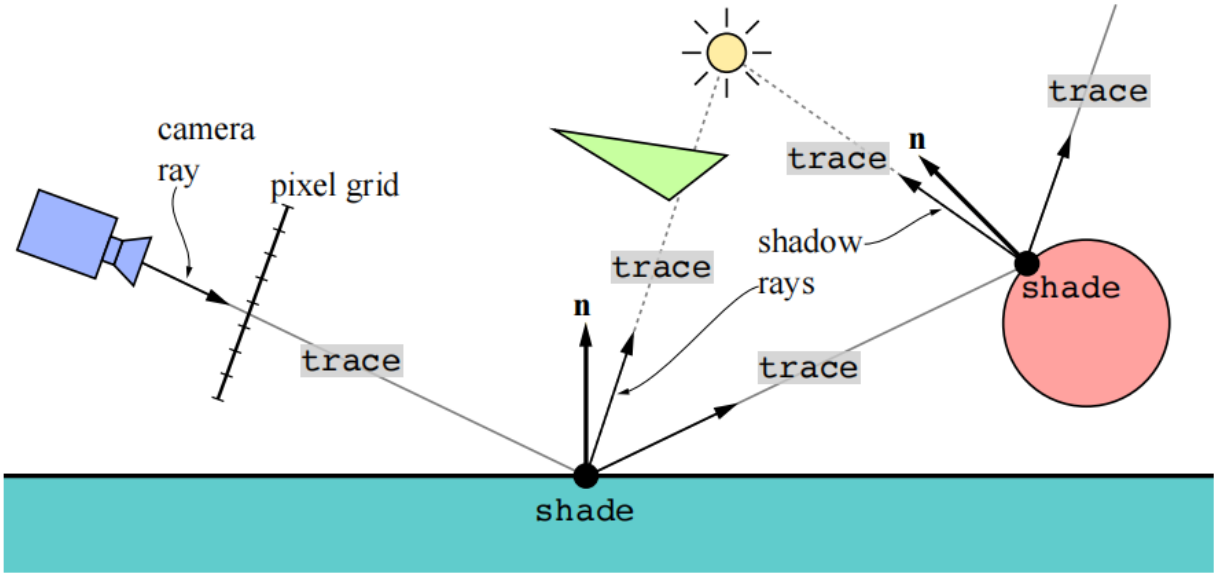


图 26.3：创建一个穿过像素的相机光线，并对 $\text{trace}()$ 进行第一次调用，并在该像素中内开始光线追踪的过程。这条光线击中了一个地面，地面的法线为 $\mathbf{n}$ 。然后在第一个交点处调用 $\text{shade}()$ ，因为 $\text{trace}()$ 的目标就是找到光线的颜色。光线追踪的强大之处在于，当我们计算该点的 BRDF 时， $\text{shade}()$ 可以通过调用 $\text{trace}()$ 来进行辅助。在这个例子中，这是通过向光

源发射阴影光线（shadow ray）来完成的，在图中的情况下，这条阴影光线被一个三角形挡住了。另外，我们假设此时的表面是一个镜面，因此我们会发射一条反射光线，这条反射光线击中了一个圆。在第二个交点处，我们再次调用 shade() 来计算着色。同样地，在这个新的交点处，我们还可以再次发射阴影光线和反射光线。

使用 trace() 和 shade() 来描述光线追踪器是很简单的。使用方程 26.2 来构建一条从相机位置指向像素内位置的光线。这条光线被送入 trace()，它的任务是找到沿着这条光线返回的颜色或者 radiance（第 8 章）。它首先会找到沿着光线最近的交点，然后再使用 shade() 来计算该点的着色，我们在图 26.3 中说明了这个过程。这个概念的强大之处在于，我们可以通过重新调用 trace()，来对需要评估 radiance 的 shade() 进行计算，例如：在 shade() 中重新构建并发射光线，并重复调用 trace()，这些新光线可以用于计算阴影、递归反射和折射、漫反射光线计算等。我们使用术语光线深度（ray depth），来表示沿着光线路径递归发射的光线数量。眼睛光线的光线深度为 1，而在图 26.3 中，对于光线击中红色圆圈的第二个 trace()，所对应的光线深度为 2。

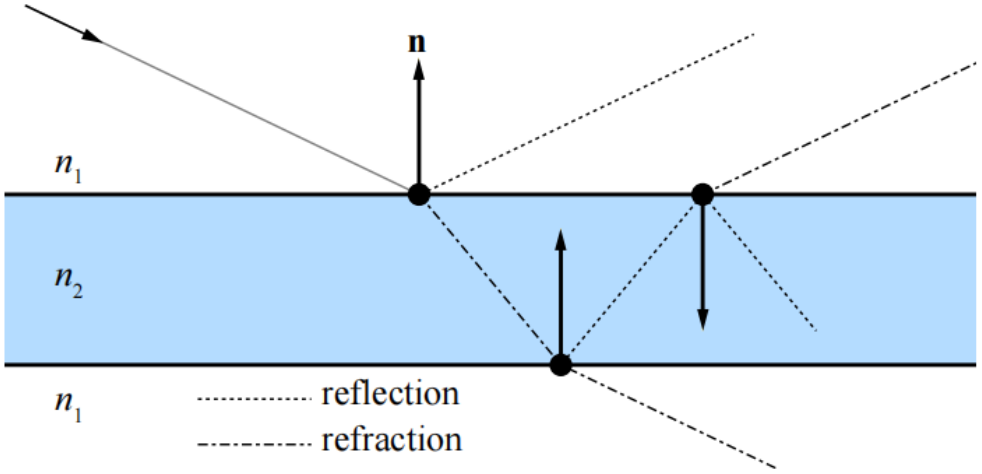


图 26.4：左上角入射光线击中了一个折射率为 n_2 表面，这个折射率要大于光线所经过介质的折射率 n_1，即 $n_2 > n_1$。在每个交点（黑色圆圈）处都会产生一条反射光线和一条折射光线。

这些新光线的一个用途是，确定当前着色点是否处于光源的阴影中，这样做可以产生阴影效果。我们还可以用眼睛光线和法线 $\mathbf{n}$，在交点处计算反射向量，并在这个方向上发射一条新的光线，这样可以在表面上产生反射效果，并且可以递归完成。同样的过程可以用来生成折射光线。完美的镜面反射、完美的折射以及尖锐的阴影，通常会被称为 Whitted 光线追踪[89]。有关如何计算反射光线和折射光线的信息，详见章节 9.5 和章节 14.5.2。请注意，当一个物体的折射率与光线此时所处介质的折射率不同时，光线有可能会被同时反射和折射，如图 26.4 所示。这种递归是基于光栅化的方

法难以解决的问题，我们只能通过使用各种方法来近似实现这些效果，从而获得光线追踪效果的一个子集。光线投射（Ray casting）是指测试两点之间或者某个方向上可见性的思想，可以用于其他的图形（或者非图形）算法，例如：我们可以从一个交点发射一些环境光遮蔽光线，从而获得该效果的准确估计。

下面的伪代码使用了函数 $\mathrm{trace}()$ 、 $\mathrm{shade}()$ 和 $\mathrm{rayTraceImage}()$ ，其中 $\mathrm{rayTraceImage}()$ 是通过每个像素来创建眼睛光线的函数。这些简短的代码片段展示了 Whitted 光线追踪器的整体结构，它可以作为许多渲染变体的基础，例如：路径追踪。

```GLSL
rayTraceImage ()
{
    for (p in pixels )
    color of p = trace (eye ray through p) ;
}

trace (ray )
{
    pt = find closest intersection ;
    return shade (pt);
}

shade ( point )
{
    color = 0;
    for (L in light sources )
    {
        trace ( shadow ray to L);
        color += evaluate BRDF ;
    }
    color += trace ( reflection ray );
    color += trace ( refraction ray );
    return color ;
}
```

Whitted 光线追踪无法提供全局光照的完整解决方案。除了镜面反射之外，在任何方向反射的光线都会被忽略，直接光直接使用点来进行表示。为了充分计算渲染方程（如方程 11.2 所示），Kajiya [41]提出了一种被称为路径追踪（path tracing）的方

法，这是一种正确的解决方案，能够生成具有全局光照的图像。一种可能的方法是，计算眼睛光线的第一个交点，然后向不同方向上发射许多光线，从而计算该点的着色，例如：如果击中了一个漫反射表面，那么我们可以在交点的整个半球方向上发射光线。然而，如果在每条光线的交点上都重复这个过程，那么需要进行计算的光线数量会爆炸。Kajiya 意识到，可以使用基于蒙特卡罗的方法来追踪一条射线，生成穿过场景的路径，并在一个像素上对几条这样的路径光线进行平均。这就是路径追踪方法的工作原理，详见图 26.5。

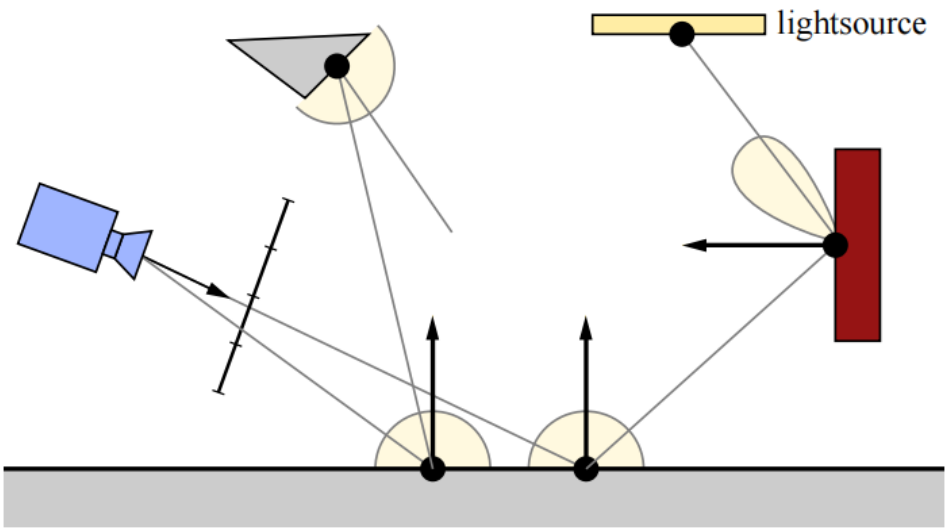


图 26.5：展示了路径追踪的原理，每个像素会发射两条光线。其中所有浅灰色表面都被假定是漫反射的，而右侧的深红色矩形则具有一个光泽 BRDF。在每次命中漫反射表面的时候，会在法线周围的半球上产生一条随机射线，并继续对其追踪。由于该像素的颜色是两条光线 radiance 的平均值，因此漫反射表面会在两个方向上进行计算，其中一个到达三角形，另一个到达矩形。随着光线数量的增加，渲染方程的计算会变得越来越好。

路径追踪的一个缺点是，需要使用许多光线才能使得图像收敛。为了让方差减半，我们需要发射四倍的光线，如图 26.6 所示。

图 26.6：上图：使用路径追踪的完整图像，每个像素使用了 65536 个样本进行渲染。下面的放大图像从左到右分别是：每像素使用 1 个样本，16 个样本，256 个样本，4096 个样本和 65536 个样本，它们渲染的是完全相同场景。请注意，即使在每个像素上使用了 4096 个样本，最终生成的图像中也有一些噪声。

其中的 $\mathrm{shade}()$ 函数总是会由用户进行实现，因此可以使用任何类型的着色，就像是光栅化渲染管线中的顶点着色器和像素着色器一样。 $\mathrm{trace}()$ 中进行的遍历和相交测试，可以在 CPU 上进行实现，也可以使用 GPU 上的计算着色器进行实现，或者使用 DirectX 和 OpenGL 来进行实现。又或者，我们可以使用一些光线追踪 API，例如 DXR。这是下一小节中的主题。

26.2 光线追踪着色器

光线追踪现在已经紧密集成到了实时渲染 API 中，例如 DirectX [59, 91, 92]和 Vulkan。在本小节中，我们将介绍不同类型的光线追踪着色器，它们已经被添加到了这些 API 中，因此可以与光栅化管线一起使用。这种组合的一个例子是，我们可以首

先使用光栅化来生成 G-buffer（第 20 章），然后从这些场景命中点处发射光线，来产生反射效果和阴影效果[9, 76]。我们称之为延迟光线追踪（deferred ray tracing）。

光线追踪着色器会被分配给 GPU，类似于计算着色器（章节 3.10），即在网格（像素）上进行划分。在本小节中，我们将遵循 DXR 中的命名习惯[59]，DXR 是 DirectX 12 中添加的光线追踪，其中共有五种类型的光线追踪着色器[59, 78]：

1. 光线生成着色器（ray generation shader）
2. 最近命中着色器（closest hit shader）
3. 未命中着色器（miss shader）
4. 任意击中着色器（any hit shader）
5. 相交着色器（intersection shader）

光线的定义使用方程 26.1 加上区间 $[t_{min}, t_{max}]$，这个区间定义了接受交点的光线部分，如图 26.7 所示。程序员可以向光线添加一个负载（payload），这是一个数据结构，用于在不同的光线追踪着色器之间发送数据，例如：一个光线的负载可以包含一个 $\mathrm{float4}$ 来表示 radiance，一个 float 来表示到命中点的距离，用户还可以添加任何需要的内容。然而，保持光线的负载较小，可以获得更好的性能表现，因为较大的负载可能会使用更多的寄存器。

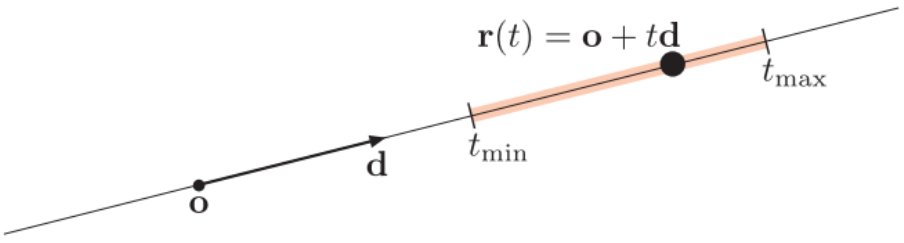


图 26.7：由原点 $\mathbf{o}$、方向 $\mathbf{d}$ 和区间 $[t_{min}, t_{max}]$ 定义的光线。只有位于这个区间内的交点才会被找到。

光线生成着色器（ray generation shader）是光线追踪的起点，它可以像计算着色器一样进行编程，并且能够调用一个新的函数 $\mathrm{TraceRay}()$，它与章节 26.1 中所描述的 $\mathrm{trace}()$ 函数相类似。通常来说，光线生成着色器会针对屏幕上的所有像素都执行一次。$\mathrm{TraceRay}()$ 中空间加速结构的快速遍历，是由驱动程序通过 API 进行提供的。可以定义一个与不同着色器相连接的光线类型，例如：对于标准光线（standard ray），通常会使用一组特定的着色器；而对于阴影光线（shadow

ray），则可以使用更加简单的着色器。对于阴影效果而言，光线可以更加高效地进行追踪，因为我们通常可以在光线间隔（从命中点到光源的范围）中找到任何交点时就立即停止。

对于标准光线而言，第一个正交点是必需的，这样的光线是由光线生成着色器进行发射的。当找到最近交点的时候，会执行一个最近交点着色器（closest hit shader）。在这个着色器中，用户可以实现章节 26.1 中的 $\mathrm{shade}()$，例如：阴影光线测试，反射，折射和路径追踪等。如果光线没有击中任何物体，则会执行一个未命中着色器（miss shader）。这对于生成 radiance 值，并通过射线负载发送回来是十分有用的。它可以是静态的背景颜色、天空颜色，也可以是在环境贴图中查找生成的颜色值。

任意命中着色器（any hit shader）是一个可选的着色器，当场景中包含透明物体或者 alpha 测试纹理的时候可以进行使用。每当在光线间隔中发生命中时，都会执行这个着色器，例如：着色器代码可以在纹理中执行查找，如果这个样本是完全透明的，那么遍历就应当继续进行，否则便可以停止。无法保证这些测试的执行顺序，因此着色器代码可能需要执行一些局部排序，才能得到正确的混合结果。这个任意命中着色器可以同时用于标准光线和阴影光线。与光栅化一样，在镂空纹理的边界周围使用更加紧密的多边形（章节 13.6.2），可以帮助减少这个着色器被调用的次数。

当光线在空间加速结构中击中某个包围盒时，会执行相交着色器（intersection shader）。因此，它可以用于实现自定义的相交测试，例如：针对分形景观（fractal landscape）、细分曲面和解析曲面（例如球体和圆锥）等。

除了光线生成着色器之外，未命中着色器和最近交点着色器都可以调用 $\mathrm{TraceRay}()$ 来生成新的光线。除了相交着色器之外的所有着色器，都可以对光线的负载进行修改。所有光线追踪着色器都可以输出到 UAV 中，例如：光线生成着色器可以将发送到相应像素中的光线颜色输出到 UAV 中。

在结合光栅化和光线追踪的领域中，还有许多创新和研究要做，而且在如何利用实时图形 API 新添加的东西（new addition）方面，也有很多创新和研究要做。Andersson 和 Barre-Brisebois [9]提出了一种混合渲染管线（hybrid rendering pipeline），它将这两种渲染范式结合在一起。首先，使用光栅化渲染一个 G-buffer，使用计算着色器来完成直接光照和后处理。直接阴影和环境光遮蔽可以使用计算着色器或者使用光线追踪来完成。而全局照明，反射，透明和半透明（transparency & translucency）等效果，是使用纯光线追踪来完成的。随着 GPU 的不断发展，性能瓶颈也将会发生移动，因此一个普遍建议是：

Use raster when faster, else rays to amaze.
当需要速度时使用光栅化，当需要质量时使用光线追踪。

与往常一样，记得实际测量性能瓶颈所在的位置（详见第 18 章）。另外，$\mathrm{TraceRay}()$ 还可以作为一个作业生成机制，也就是说，一个着色器可以使用 $\mathrm{TraceRay}()$ 来生成多个作业，从而计算一个组合结果，例如：这个特性可以用于自适应光线追踪（adaptive ray tracing），在这个技术中，会向具有高方差的像素区域发送更多的光线，其目标是以相对较低的成本来提高图像质量。除此之外，$\mathrm{TraceRay}()$ 可能还有许多在 API 设计期间没有考虑到的用途。

26.3 顶层和底层加速结构

DXR 的加速结构对于用户来说基本上是不透明的，但是有两个层次结构是可见的，它们被称为顶层加速结构（top level acceleration structure，TLAS）和底层加速结构（bottom level acceleration structure，BLAS）[59]。BLAS 中包含一组几何图形，它可以被认为是场景的组成部分。TLAS 中包含一组实例，每个实例都指向一个 BLAS，如图 26.8 所示。

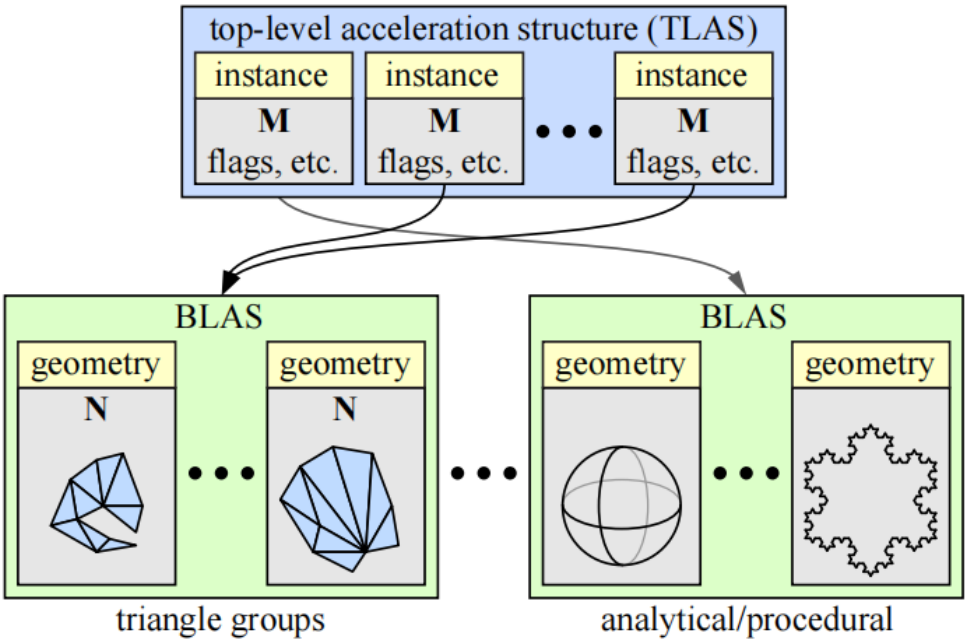


图 26.8：图中展示了顶层加速结构（TLAS）是如何连接到一组底层加速结构（BLAS）的。每个 BLAS 中都可以包含几组图元，每个图元中只包含了三角形或者程序化的几何图形。每个几何图形和实例，都可以通过 3×4 矩阵 $\mathbf{N}$ 或者矩阵 $\mathbf{M}$ 进行转换。请注意，对于相应的实例

或者几何图形，每个矩阵都是唯一的。TLAS 中包含一组实例，这些实例可以指向一个 BLAS。

BLAS 中既可以包含三角形类型的几何图形，也可以包含程序化生成的几何图形。前者直接包含一组三角形，后者则与一个相交着色器关联，使用这个着色器可以实现自定义的相交测试，例如：这可以是对球体或者圆环的解析光线测试，或者是一些程序生成的几何图形。

在图 26.8 中，所有矩阵 $\mathbf{M}$ 和 $\mathbf{N}$ 的大小都是 3×4，即任意的 3×3 矩阵再加上一个平移向量（第 4 章）。其中矩阵 $\mathbf{N}$ 用于在构建过程开始的时候，对相应几何图形的底层加速数据结构（例如 BVH 或者 k–d 树）进行一次转换。另一方面，矩阵 $\mathbf{M}$ 可以每帧进行更新，因此可以用于一些比较轻量的动画。

对于任意动画的几何物体，其中的三角形可能会被添加或者删除，因此必须每帧重建 BLAS。在这种情况下，即使是矩阵 $\mathbf{N}$ 也可以被更新。如果我们只需要更新顶点的位置，那么可以在 DXR API 中请求一个更快的数据结构更新。这样的更新通常会降低一些性能，但是在几何物体只移动了一点点的情况下可以很好地工作。一个合理的方法是在可能的情况下，使用这些成本较低的更新，并每 n 帧进行一次重建，以便将这个成本分摊到多个帧中。

请注意，与光栅化相比，在使用光线追踪的时候，几何物体的分组通常应该不同。在第 18 章中我们提到，对于光栅化中的几何物体，我们通常会按材质参数进行分组，以便在像素着色期间利用着色器的一致性来提高性能表现。当使用空间局部性来进行分组的时候，用于光线追踪的加速数据结构会有更好的性能，如图 26.9 所示。如果光线追踪的几何物体是按照材质而不是空间局部性进行分组的，会使得性能受到很大的影响。

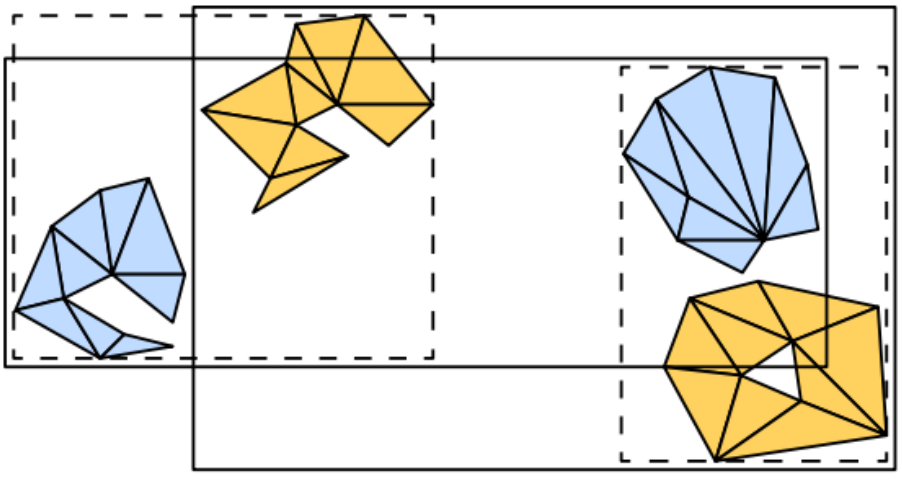


图 26.9：为了对光栅化渲染进行优化，我们经常会对几何图形进行按材质的分组，这里使用三角形网格的颜色来代表不同的材质，对应的分组框使用实线进行绘制。对于光线追踪而言，最好是将空间上彼此接近的几何图形分为一组，相应的分组框使用虚线进行绘制。

26.4 一致性

在软件和硬件的性能优化中，一个最重要的思想就是在执行过程中利用一致性（coherency，或者叫做连贯性和相干性）。我们可以通过在给定计算的不同部分之间，对结果进行重用来节省工作量。在当今的硬件中，无论是时间消耗还是能量消耗，最昂贵的操作都是内存访问，它要比简单的数学运算慢上几个数量级。评估硬件操作成本的一个好方法是，考虑比特在电路中完成操作所需要的物理距离，这个距离越大，成本就越高。在大多数情况下，性能优化的重点是利用内存一致性（即缓存），以及围绕内存延迟来对计算进行调度。GPU 本身可以被看作是一个处理器，它显式约束着其所运行程序的执行模型（数据并行，独立的计算线程），以便能够更好地利用内存一致性（章节 23.1）。

在本章节的介绍中，我们讨论了如何将光线追踪和光栅化用于屏幕像素（相机光线）“首次命中”的可见性，并将屏幕像素视为场景几何结构的不同遍历顺序。虽然排序在算法复杂度方面的影响不大，但是每种排序都有实际的影响。在使用光栅化和光线追踪的时候，我们有一个双重的 for 循环，其中最内层的循环是大多数计算发生的地方（除非它非常小）。由于在内部循环中的迭代是相邻进行的，因此它们是通过在迭代之间重用数据，以及利用内存访问局部性（缓存优化）来减少计算量的最佳候选。

光栅化的内部循环位于给定物体表面的像素上，这些表面上的点很可能会表现出高度的计算一致性（coherent computation）：它们可能会使用相同的材质进行着色，使用相同的纹理，甚至会对附近位置上的纹理（内存）进行访问。如果我们必须计算大量相机像素的可见性，我们可以轻松地以一种空间一致的顺序来遍历这些位置，例如：在屏幕上的小 tile 中进行遍历。这样做可以确保内部循环的的工作量高度一致（章节 23.1）。请注意，一致性不仅只存在于可见性问题上。通常我们会在知道哪些表面是可见的之后，再开始进行渲染，这是工作量最集中的地方，会使用大量的计算能力来计算材质属性，以及它们与场景光照之间的相互作用。光栅化之所以特别快，不仅仅是因为它可以高效地计算哪些物体覆盖了哪些像素，而且还因为随后的着色计算可以自然地以一种利用一致性的方式进行排序。

相比之下，对于外部循环中的给定光线，一个简单的光线追踪器会对内部循环中的所有场景图元都进行迭代。m 个像素和 n 个物体会导致 $O(mn)$ 的复杂度，无论我们如何避免这个双重循环的整体开销，当在沿着单条光线对一个渲染图元列表进行遍历的时候，我们几乎没有利用一致性来进行加速。

因此，现代光线追踪器中的大多数性能优化，都涉及如何在光线可见性查询中，以及在随后的着色计算中“找到”这种一致性。我们当然可以说光栅化在默认情况下就是一致的，但是也会受限于特定的可见性查询，即相机的视锥体。在使用光栅化技术的时候，大部分工作都涉及如何对这个查询函数进行扩展，从而来模拟各种效果。相比之下，光线追踪在默认情况下是十分灵活的，我们可以在任意点的任意方向上来查询可见性。然而，这样做会导致计算的不一致（连贯性），使得它在现代硬件架构上的效率很低，因此大部分的工程努力，都花在了如何以一种一致（连贯）的方式来组织可见性查询上。

随着光线查询的灵活性不断增加，我们可以渲染光栅化无法实现的效果；同时，通过利用一致性我们仍然能够保持较高的性能表现。阴影效果就是一个很好的例子，对阴影追踪光线，可以让我们更加精确地模拟面光源的效果[35]。阴影光线只需要与几何物体相交，在大多数情况下都不需要对材质进行评估。这些特性降低了访问不同物体时的开销。与着色光线相比，我们需要计算的是：光线是否击中了交点与光源之间的任何物体。因此，我们可以避免在交点处计算表面法线，避免检索实体物体的纹理，并且可以在发现第一个（实体）命中后就立即停止追踪。此外，阴影光线通常也是高度一致的。

对于屏幕上附近区域的像素而言，它们具有相似的起点，并且可能会指向相同的光源。最后，阴影贴图（章节 7.4）无法按照屏幕像素的精确频率来对光源的可见性进行采样，这会导致欠采样或者过采样。在后一种情况下，增加阴影光线的灵活性甚至可以带来更好的性能表现。光线通常来说是更加昂贵的，但是通过避免过采样，我们可以执行更少的可见性查询。这也就是为什么阴影贴图是光线追踪在游戏中的第一个图形应用[90]。

26.4.1 场景一致性

当我们考虑到三维场景中图元之间的距离时，它们就会陷入自然的空间关系中。当我们考虑渲染所需要的着色工作时，这些关系并不一定保证计算的一致性，例如：一个物体在空间位置上可能会靠近另一个物体，但是二者使用了完全不同的材质、纹理以及最终的着色算法。在光线追踪器中，用于加速物体遍历的大多数算法和数据结构，同样也可以适用于光栅化器，如章节 19.1 中所述。然而，这些数据结构在光线追踪器

中，要比在光栅化器中更加重要。当我们在追踪光线的时候，对场景物体的遍历属于内部循环中的一部分。

大多数光线追踪器和光线追踪 API，都会使用某种形式的空间加速数据结构。来对光线的可见性查询进行加速。在许多情况下，包括当前版本的 DXR，这些技术会在幕后实现并作为黑盒功能进行提供，对于用户都是不透明的。出于这个原因，如果您只是专注于了解基本的 DXR 功能和相关技术，那么可以跳过本小节关于一致性话题的其余部分。然而，如果您想了解关于光线追踪的性能问题，尤其针对大型场景的光线追踪，那么这一小节就十分重要。如果您知道您的系统依赖于某种特定的空间结构，那么了解与该方案相关的优势和成本，可以帮助您提高渲染引擎的效率。

在可见性计算中，创建利用场景一致性（scene coherency）的数据结构，对于实时渲染而言特别具有挑战性，因为在大多数情况下，场景在动画下都是逐帧变化的。事实上，尽管我们之前提到过，光线追踪器的外部循环在可见性查询方面具有更大的灵活性，但是光栅化器的外部循环实际上可以更加自然地处理动画场景，以及程序化生成的几何图形和核外（out-of-core）几何图形（存储量过大，无法一次性放入内存中）。在光栅化渲染方案中，空间数据结构通常会以相对简单的形式进行呈现，其中一个原因就是因为光栅化的循环结构。

空间数据结构背后的思想是，我们可以在各个空间分区内来组织几何图形，从而将场景中彼此接近的物体组织在同一个空间内。一种实现这种划分的简单方法是，将整个场景细分到一个均匀网格中，并在每个立方体（体素）中存储一个与之相交的图元列表。然后，根据光线的方向，从光线的原点位置开始，对光线上的每个单元格进行遍历，即可完成光线对整个场景的遍历。这种遍历方式与保守的直线光栅化算法实际上是相同的，其中的区别在于，这里发生在三维空间中。该方法的基本思想是在 x 、 y 、 z 方向上，找到与下一个体素之间的距离，并取其中最小距离所对应的方向，并沿着射线移动到该体素上。图 26.10 中最左侧的插图，展示了光线是如何访问均匀网格中的单元格的。然后对这三个值进行更新，并使用新的最小距离，来移动到下一个体素上。在遍历过程中，每当我们发现一个非空单元格的时候，我们都需要根据单元格中包含的所有图元，来对光线进行相交测试。一旦我们在单元格内找到了一个交点，那么网格中的遍历就不需要继续进行了。对于阴影光线（任意命中）而言，我们可以直接停止，因为我们只想知道是否存在遮挡即可；但是对于标准光线而言，我们则需要对单元格内的所有图元都进行测试，并选择其中最接近的那个图元，因为我们不仅需要遮挡信息，还需要找到那个最近的交点。Havran [31]的论文提供了一个很好的综述。

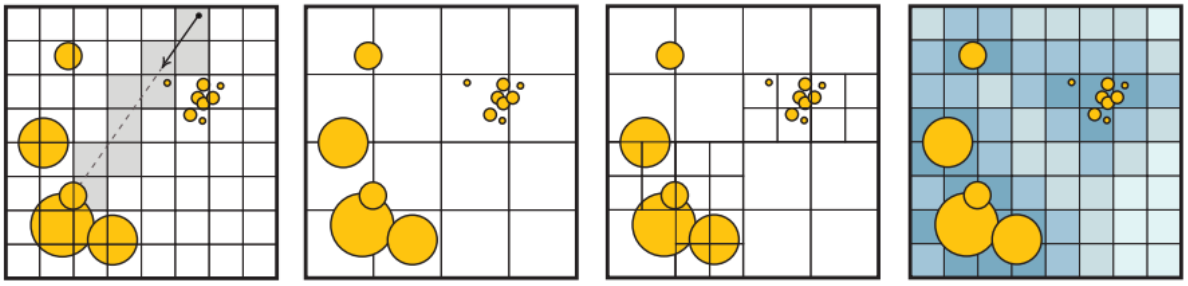


图 26.10：从左到右分别是：一个精细的均匀网格以及被光线穿过的单元格；一个粗糙的均匀网格；一个两级网格；一个嵌入邻近云信息（proximity cloud information）的均匀网格，其中的较暗部分代表了存在距离单元格很近的物体，较亮部分代表了距离该单元格最近的物体，与单元格的距离也比较远。

由于场景中可能会包含小而详细的物体，在一些区域中可能会包含许多微小的图元，而在另一些区域中则包含较大且粗糙的图元，因此固定的网格尺寸可能并不适用于所有地方。这种情况被称为“体育场中的茶壶（teapot in a stadium）”问题[27]，它是指一个复杂的茶壶（注意力的焦点）落入了单个单元格中，因此无法从加速结构中获益。尽管朴素的均匀网格可以进行快速构建，并且遍历方式也十分简单，但是目前在大部分光线追踪中都很少进行使用。有一些改进网格效率的变体结构，这些变体会更加实用。网格可以以一种分层的方式进行嵌套，根据需要使用更加高级的大单元格，来包含更加精细的单元格。两层的嵌套网格在 GPU 上并行构建的速度特别快[42]，并且已经在早期的动画 GPU 实时光线追踪 demo 中进行了成功应用[80]。

哈希表可以用来创建一个无限的虚拟网格，其中只有被实际占用的单元格才会将数据存储在内存中（章节 25.1.2）。另一种策略是在空单元格中，存储该单元格到最近非空单元格之间的距离，这个系统被称为邻近云（proximity cloud）[14]。在遍历过程中，这些存储在单元格中的距离，可以让我们在直线步进（line marching）的过程中，跳过那些保证为空的单元格。最近，不规则网格（irregular grid）方法[64]详细阐述了高效跳过空白空间的想法。在光线追踪的动画场景中，这些方法已经被证明了，它们能够与最先进的空间加速方法相竞争。图 26.10 展示了这些网格的一些变体。

如果我们将分层网格的想法发挥到极致，我们可以想象拥有尽可能低分辨率的网格；在每个轴上有两个单元格来作为顶层数据结构，并将每个非空单元格递归分割为另一个 $2 \times 2 \times 2$ 的单元格，这个结构实际上就是一棵八叉树，我们在章节 19.1.3 中进行了讨论。更进一步，我们可以想象，在分层数据结构的每个层次上，我们都使用一个平面来将单个单元格分割为两个单元格。如果这个平面的选择是任意的，那么就构成了一棵二叉 BSP 树；如果这个平面被约束为是轴对齐的，那么就构成了一棵 k-d 树

（章节 19.1.2）。如果我们在数据结构中的每个层次上，都使用一对轴对齐的平面，而不是使用单个轴对齐的平面，那么我们将会得到一个有界区间层次（bounded interval hierarchy，BIH）树[84]，它具有与之相关的快速构建算法。

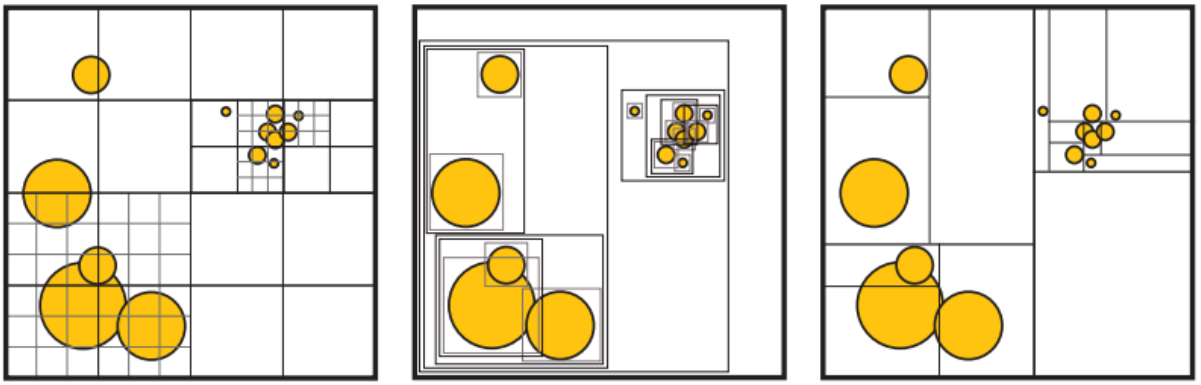


图 26.11：一些流行的空间细分数据结构。从左到右分别是：层次网格；使用 AABB 的 BVH；k-d 树。

今天最流行的光线追踪加速结构是层次包围体结构（bounding volume hierarchy，BVH），我们在章节 19.1.1 中进行了介绍，如图 26.11 所示。例如：在英特尔的 Embree 内核[88]、AMD 的 Radeon-Rays 库[8]、NVIDIA 的 RT Cores 硬件[58]和 OptiX 系统[62]中，都使用了 BVH。

空间数据结构的属性

空间数据结构的设计范围很大。我们可以有更深的层次结构，更深的层次结构意味着需要更多的间接遍历，但是可以更好地适应场景中的几何物体；较浅的层次结构虽然不太灵活，但是在内存中的存储会更加紧凑。我们可以有严格的细分方案，这种方案易于构建，每个节点中只需要存储很少的数据；我们还可以使用更具表现力的方案，这种方案在如何划分空间方面具有许多自由度。例如：BVH 方案可能在创建之前，就已经知道了具体的内存开销，因此可以进行更少的细分，并且能够更好地跳过空白空间。然而，它们构建起来会更加复杂，可能需要更多的存储空间来对每个节点进行编码。

一般来说，空间数据结构的权衡（trade-off）如下：

- 结构质量（construction quality）。
- 构造速度。
- 更新速度，用于动画场景。
- 运行时遍历效率。

其中结构质量可以大致转换为，我们必须要遍历多少图元和多少单元格，才能找到一个光线交点。构造速度和遍历速度通常取决于给定的硬件。更加复杂的是，所有这些数据结构都允许使用多种遍历算法和构造算法，以及能够对节点进行不同方式的编码（使用不同的压缩方案和内存布局）。此外，空间细分方式中的任何自由度，也意味着存在不同的启发式方法来指导如何对空间进行细分。

如果不事先指定这些参数，那么只对数据结构的性能进行讨论是存在一定误导性的。在实践中，对于静态场景和动态场景而言，最先进的数据结构是不同的；对于动态场景而言，其数据结构的构建算法必须要在严格的时间限制下运行，以免构建过程的开销要比渲染时光线追踪所节省的时间更多。在硬件方面，基于 CPU 和 GPU（高度并行）的算法之间存在着显著差异。基于 GPU 的最佳实践仍在不断发展，因为相关的 GPU 架构是最近才出现的，并且随着时间的推移，已经发生了许多的变化[46]。

最后，我们需要进行追踪的光线细节也十分重要，某些结构在相干光线（coherent ray，例如：相机光线，阴影或者镜面反射）下的表现最好；而其他结构对于非相干、随机散射光线（通常用于漫反射全局光照或者环境光遮蔽方法）的容忍度则要更高。

考虑到上述这些因素，值得注意的是，历史上最先进的光线追踪性能，是通过 k-d 树，或者 BVH（使用 AABB）的某种变体来实现的[83]。从理论上来说，二者之间的主要区别在于，k-d 树将空间划分为不相交的单元格，而 BVH 中的节点通常会发生重叠。这意味着 BVH 遍历必须在找到一个交点，并且树中没有其他未检查的包围体时才能停止。然而，k-d 树在发现一个图元交点时便可以立即停止遍历，因为 k-d 树可以强制执行严格的、从前往后的遍历顺序。然而 k-d 树的这种理论优势，并非总是能够实现。例如：BVH 可以更加高效地跳过空白区域，并且能够更加紧密地拟合图元，从而能够更快地找到交点，来对无法提前停止遍历的问题进行弥补[83]。

在实践中，我们对一些用于电影制作[21, 67]和用于交互式渲染[8, 58, 62, 88]的渲染器进行了调查，我们没有找到任何目前使用 k-d 树的渲染器。目前我们调查的所有系统，都以某种形式依赖于 BVH 来进行通用的光线追踪。

> 在默认情况下，2012 年左右的 Brazil 渲染器在场景中使用了三维和四维的（运动模糊）k-d 树。[28]

对于特定类型的图元或者算法而言，使用其他结构可能会更加高效。例如：点云和光子映射会使用三维 k-d 树来存储样本；八叉树和网格结构可以用于体素数据。

BVH 通常能够很好地拟合场景，并且具有快速且高质量的构造算法，它还可以很容易地处理动画场景，尤其是当动画场景表现出良好的时间一致性时。此外，正如我们将在下一小节中所看到的那样，构造一棵拟合紧凑且深度较浅的 BVH 树是可能的，它可以使用更少的内存和更少的带宽，来实现良好的场景划分，这是高性能遍历数据结构的关键属性。

构造方案

详细介绍用于光线追踪空间数据结构中的所有算法变体和排列组合，超出了本章节的讨论范围，但是我们可以介绍其中的一些关键思想。有关这些主题的更多信息，详见章节 19.1 和第 25 章。

这些空间数据结构的构造算法可以分为物体划分（object partitioning）方案和空间划分（space partitioning）方案。其中物体划分考虑在空间上彼此接近的物体或者图元（例如单个三角形），并将它们聚集在数据结构的同一个节点中。这是一个“自上而下”的过程，我们会在每个步骤中，决定如何将场景物体划分成子集合；或者使用一个“自下而上”的过程（第 25 章），通过迭代来对场景物体进行聚类。相比之下，空间划分会决定如何将空间划分为不同的区域，并将物体和图元分布到数据结构的最终节点中，这些结构通常是“自上而下”进行构建的。空间划分的构建过程通常要慢得多，很难用于实时渲染，但是它们可以更加高效地投射光线。

空间划分是构造 k–d 树的最明显方法，但是同样的原理也适用于 BVH 的构造，例如：Stich 等人[62, 77]的 split BVH 方案同时考虑了物体划分和空间划分，并允许一个给定物体被多个 BVH 的叶子节点所引用。与常规的 BVH 方案相比，这样做可以显著降低光线的发射成本，同时仍然要比构建一个纯空间划分的结构更快，如图 26.12 所示。

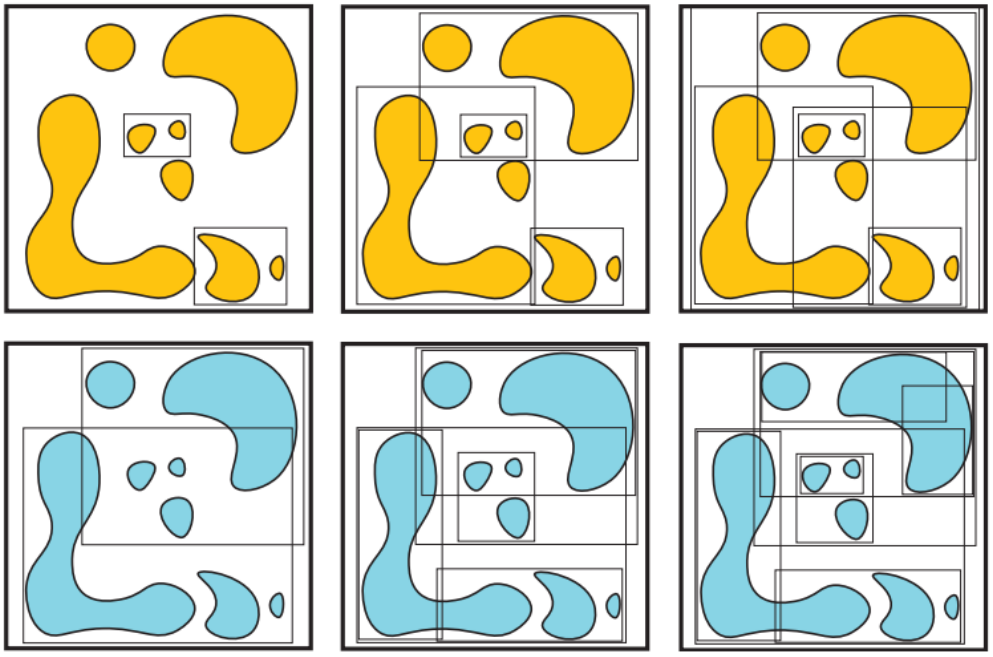


图 26.12：第一行：自下而上、物体划分方案的 BVH 构建阶段。第二行：自上而下、空间划分方案的 BVH 构建阶段。

无论使用哪种方案，都必须在构建过程中的每一步做出选择。对于自下而上方案，我们必须决定要将哪些图元进行聚合；而对于自上而下方案，我们必须决定要将用于细分场景的空间分割位置放置在哪里，如图 26.13 所示。能够最小化总光线追踪时间的方案就是最佳的选择，这反过来又取决于所使用的遍历算法的细节，以及要确定可见性的光线集合。在实践中，想要准确地评估这些选择会对光线追踪产生怎样的影响是不可能的，因此必须采用一些启发式的方法。

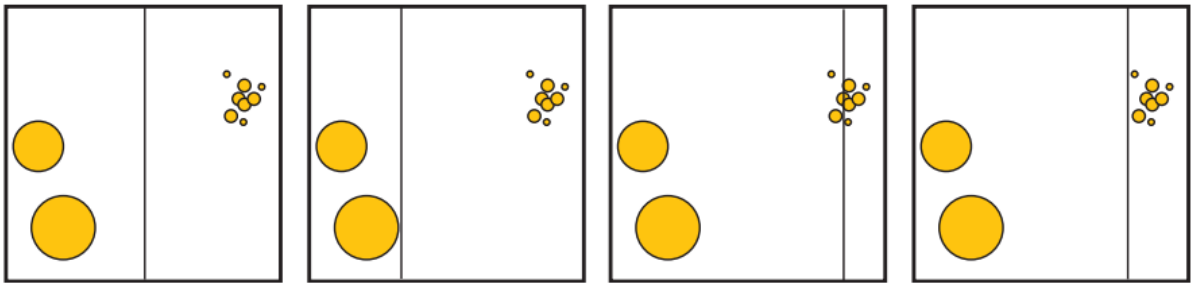


图 26.13：垂直分割平面的不同选择方案。从左到右分别是：中间分割；分割较大的图元；中位数分割，从而产生两个具有相同图元数量的节点；SAH 优化的分割，它可以将大多数图元所处节点的面积进行最小化，从而将命中昂贵节点的概率进行最小化。

对构造质量进行近似的最常用方法就是表面积启发式（surface area heuristic，SAH）方法[53]（章节 22.4）。它定义了如下的成本函数：

$$\frac{1}{A_{\text{root}}}\left(C_{\text{node}}\sum_{x\in I}A_x + C_{\text{prim}}\sum_{x\in L}P_x A_x\right),$$

其中 A_x 是节点 x 的表面积； P_x 是给定节点中的图元数量； I 和 L 是树中内部节点和叶子节点（非空）的集合； C_{node} ， C_{prim} 是对节点和图元平均相交成本（即平均时间）的估计（详见章节 25.2.1）[5, 26]。SAH 可以是包围体、单元格或者其他体积的表面积，这个表面积与一条随机射线击中它的概率成正比。这个方程对图元层次结构的加权概率成本进行了求和，所计算出的成本是一个对该结构效率的合理估计。

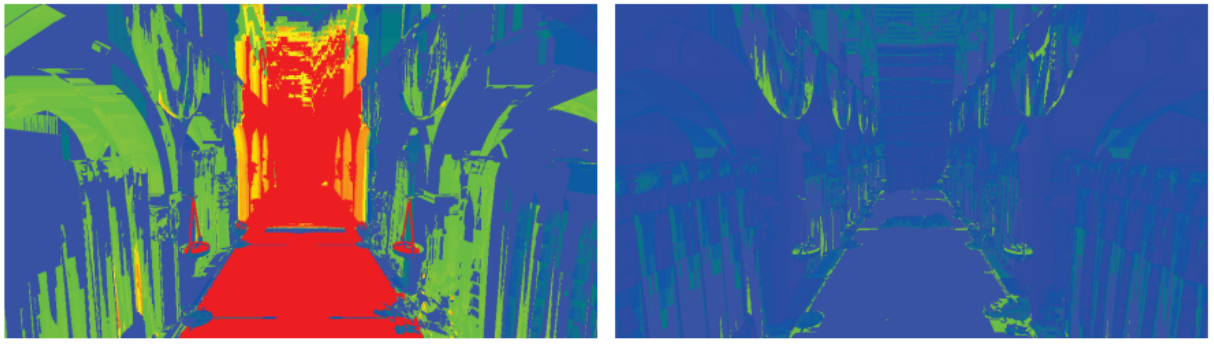


图 26.14：这个热力图（Heat map）代表了在 Sponza 中庭场景中，首次命中的相机射线在每个像素上所遍历的 BVH 节点数量。其中的红色区域需要 500 多个遍历步骤。左图使用了中位数分割（median-cut）启发式方法进行构造的 BVH，右图则使用了经过 SAH 优化的 BVH 构建方法。

当调整其中的常数时，SAH 与追踪随机长射线的实际成本能够很好地相关，并在实践中表现良好，如图 26.14 所示。但是这些假设并不总是成立的，有时候可能会有更好的启发式方法，特别是如果我们事先知道追踪光线的分布，或者可以对场景中追踪光线的特定分布进行采样时。对于完整构建的空间数据结构，SAH 给出了对光线追踪成本的估计，并可以用于在构建过程中对选择进行修正。SAH 优化的结构会在一组可能的选择中，奖励那些能够产生包含许多图元的小节点的选择，如图 26.13 所示。

在实践中，构建 SAH 最优结构的算法可能会很慢，因此需要使用进一步的近似方法。对于 k-d 树而言，一种近似的 SAH 策略是，对固定（少量）划分平面数量的启发式成本进行评估，并在每层的选择中给出最具有最高效结果的平面。这种称为 binning 的策略也可用于自上向下的快速构建 BVH 结构[86]。

对于动画场景而言，构造算法的时间约束是非常严格的，因此可以牺牲一定的树结构质量，来换取更快的构建速度。对于空间划分方案而言，可以使用中间切割[84]，沿

着场景中的最长轴，将空间在中点进行分割；或者是在旋转循环的 x ， y ， z 轴序列中，将空间在中点进行分割。

物体划分方案可以更快。Lauterbach 等人[47]引入了线性层次包围体（linear bounding volume hierarchy，LBVH）构造方法，在该方法中，会使用空间填充曲线来对场景中的图元进行排序，这些方法在章节 25.2.1 中进行了解释，如图 25.7 所示。这样的曲线具有在空间中定义位置排序的性质，在曲线排序顺序中，相邻点在三维场景中的位置也可能会彼此靠近。即使在 GPU 等高度并行的处理器上，这种物体排序也可以高效地计算，并用于对相邻的包围体进行聚类，从而自下而上地构建层次结构。2010 年，Pantaleoni 和 Luebke [23, 63]提出了一种被称为分层线性层次包围体（hierarchical linear bounding volume hierarchy，HLBVH）的改进方法，提高了构造速度和构造质量。在他们的方案中，BVH 的顶层节点是使用 SAH 优化的方式进行构建的，而底层节点的构建方式则类似于原始的 LBVH 方法。

使用 BVH 的一个优点在于，该结构所需的最大内存是事前知道的，因为所需要的聚类节点数量有限制的[86]。然而，从零开始构建空间结构，其时间复杂度在最好的情况下，与图元数量也是线性相关的。每帧都对 BVH 进行重建，可能最终会成为渲染性能的一个重大瓶颈，尤其是在给定帧中并没有对许多光线进行追踪的情况下。另一种方法是避免对 BVH 进行重新构建，并针对场景中的运动物体，对空间数据结构进行一定地“调整（refit）”。在 BVH 的情况下，这种调整操作特别容易实现。首先，使用物体当前的几何图形，针对包含动画图元的叶子节点，重新计算这些叶子节点的包围体。然后再对这个叶子节点的父节点进行检查，如果它不再能够包含这个叶子节点，则将这个父节点进行展开，并在链上对爷爷父节点进行测试。这个过程会一直持续到祖先节点不需要再进行修改位置，或者是到达 BVH 的根节点为止。检查父节点和叶子节点时的另一个选择是，始终根据其子节点的包围体，来将自身的包围体最小化。这将会提供一个性能更好的树结构，但是相应地也将花费更多的时间。

这种调整方法是十分快速的，但是如果在动画期间有一些物体发生了较大的位移，酱会导致空间数据结构的质量下降，这是因为虽然物体在层次结构中仍然聚集在同一个节点内，但是彼此不再靠近，因此相应的包围体会随着时间的推移，逐渐发生膨胀。为了解决这个缺点，可以使用一些迭代算法，来对树结构进行旋转，并逐步提高 BVH 的质量[40, 44, 94]。

目前，最先进的并行 GPU BVH 构建器是 treelets 方法[40]，该方法会从一个快速但低质量的 BVH 开始，并对其拓扑结构进行优化。它可以构建出一棵高质量的树，其质量与 split BVH 方法相当，但是在现代硬件上的处理速度为每秒数千万个图元，这只比 HLBVH 慢一点点。

在 DXR 中所使用的两级层次结构，在动画场景中也经常使用。正如我们章节 26.3 中所看到的，如果场景中的物体是刚性移动的，那么这些层次结构允许通过动画的矩阵变换，来快速进行重建。在这种情况下，只需要重建顶层结构即可，而不需要对开销巨大的底层结构进行逐物体的更新。此外，如果物体的动画是非刚性的，但是并不会发生显著的位移（例如：树叶在风中摇曳），那么仍然可以在 BLAS 中使用重整策略，从而避免完全的重建（章节 25.7）。然而，两级层次结构的一个问题在于，它们的质量通常不如一次性为整个场景构建的空间数据结构。例如：想象一下，现在有许多物体彼此靠近，每个物体都有自己的 BLAS，但是这些 BLAS 的包围体会在 TLAS 中发生重叠。当一条光线穿过多个 BLAS 重叠的空间区域时，必须要对每个 BLAS 都进行遍历；而为整个场景构建的单一的、统一的空间数据结构，则不会遇到这个问题。为了改善这一问题，Benthin 等人[10]提出了“重新编织（re-braiding）”的两层结构思想，允许不同物体的树结构进行合并，从而提高光线遍历的性能。

遍历方案

与空间数据结构的构造方案类似，人们对光线的遍历算法也进行了大量研究。

将一条光线与一个层次数据结构进行求交，也是树结构遍历的一种形式。从根节点开始，会根据将场景划分为子空间的结构，来对光线进行相交测试。一条光线可能会与多个子空间相交，因此需要访问树结构的多个分支，例如：对于一个二叉 BVH，对于给定的树节点，一条光线可能会与 0、1 或者 2 个子节点所对应的包围体相交。在 k-d 树中，每个节点都会与一个平面相关联，这个平面会将空间划分为两部分。如果一条光线与这个平面相交，并且交点位于该节点的边界内部，那么这条光线就需要对两个子空间进行访问。因此，一般来说，每当需要考虑与光线相交的多个子空间时，我们必须决定首先遍历哪个子空间。当没有找到任何一个交点，或者我们需要多个交点的时候，还需要一种方法来进行回溯，并访问其他尚未进行相交测试的子空间。

通常，我们会相对于光线方向，来对节点的子空间进行从前到后的排序，即首先会遍历最近的子空间，然后按照前后顺序，将其他子节点依次压入栈中。如果需要进行回溯操作，则从栈中弹出一个子节点，并从这个节点继续进行遍历。如果我们一次只会追踪一条光线，那么维护这个栈的成本是微不足道的。然而在 GPU 上，通常我们会同时并行遍历数千条光线，每条光线都需要维护一个属于自己的栈结构，这样做会产生很大的内存流量（memory traffic）开销。

对于 k-d 树以及其他总是在不相交子空间中划分场景的空间数据结构而言，如果我们愿意支付在到达叶子节点之后，从根结构重新开始遍历的代价，那么想要实现一个无

栈（stackless）的光线追踪十分简单[22, 37]。如果我们总是会首先遍历最近的子空间，并且当我们到达叶子节点但是却无法找到一个光线交点时，我们可以简单地将光线原点移动到与叶子边界最远的交点处，然后对光线进行再次追踪，就好像它是一个全新的光线一样，如图 26.15 所示。

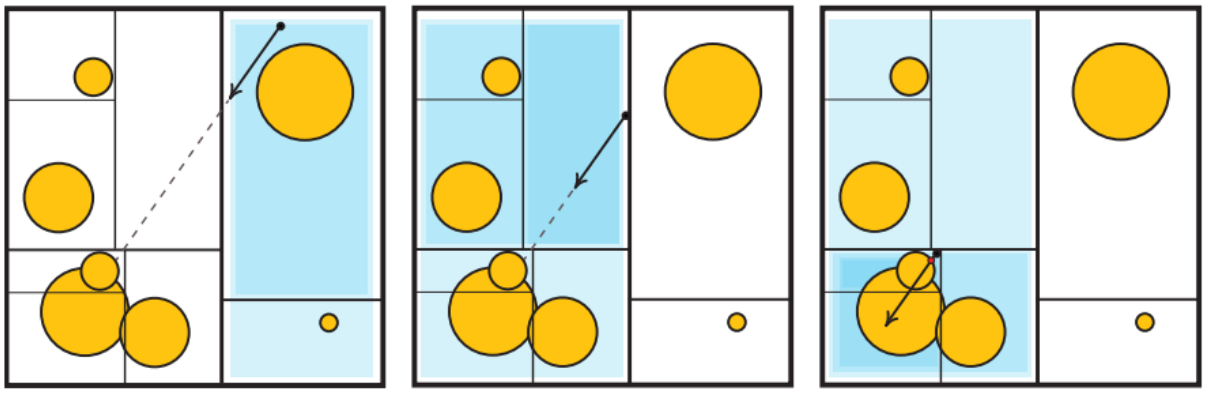


图 26.15：无栈的 k-d 树遍历方案，从左到右依次会进行不同的重启操作。每当到达一个叶子节点，但是没有找到交点的时候，光线就会被“缩短”，并将光线的原点移动到叶子节点的边界之外。

这种策略也称为光线缩短（ray shortening），它并不适用于 BVH，因为 BVH 中的节点可能会发生重叠。将光线原点推进到子树中的节点，可能会完全错过层次结构中应当进行遍历的包围体。Laine [45]建议在每棵树层级中只保留一个 bit，而不是维护一个完整的栈结构，这个 bit 会编码一个路径，从而将二叉树分成已经处理过的节点与仍然待求交的候选节点。这个方法允许在 BVH 中重新开始求交测试，前提是每当我们沿着树结构向下遍历的时候，都能为一条光线计算出一致的节点遍历顺序。

如果我们允许在树结构中存储额外的信息，即如果光线错过了给定的节点，则它会指向下一步需要进行遍历的节点，就有可能避免重新进行遍历。这些指针被称为绳索（ropes），同时适用于 BVH 和 k-d 树，但是它的存储成本很高，并对所有光线都强制执行固定的光线遍历顺序，也就是说，它不允许按照从前向后的顺序来遍历子节点。Hapala 等人[30]，以及随后的 Afra 和 Szirmay-Kalos [2]开发了一种算法，同时使用了存储在树中的指针以及一个较小的逐光线数据结构，从而允许二叉 BVH 的无栈遍历，它能够与基于栈的解决方案保持相同的遍历顺序，并使用回溯来替代完全重启。

在实践中，这些无栈方案可能并不总是要比基于栈的方法更快，即使在 GPU 上也是如此[4]，因为它们需要额外的工作来进行重新启动或者回溯操作，同时增加了空间数据结构的内存大小，并且会在某些情况下具有更差的遍历顺序。Binder 和 Keller [11]设计了一种具有恒定执行时间的无栈回溯算法，这个算法的性能表现首次在现代

GPU 上优于基于栈的遍历方法。最近，Ylitie 等人[93]提出了利用压缩方案来执行基于 GPU 的栈遍历，在很大程度上避免了由于 GPU 逐光线堆栈记录（bookkeeping）所带来的内存流量开销。这种遍历方法是在一个较宽的 BVH 上进行的，每个节点可以有两个以上的子节点。作者实现了一种高效的近似排序方案，从而确定从前往后的遍历顺序。此外，节点包围体自身会以压缩形式进行存储，这是另一种利用场景一致性的方法。

26.4.2 光线和着色的一致性

即使光线追踪可以计算任意光线的可见性，但是在实践中，大多数渲染算法都会生成光线集合，这些光线集合展示出了不同程度的一致性。最简单的例子就是，使用光线来确定场景中的哪些部分，是针孔相机可以通过屏幕像素看见的。这些光线都具有相同的原点、相机位置，并且只会在所有的可能方向上，跨越一个有限范围的立体角。类似地，用于计算来自无穷小光源的阴影光线也是一致的。即使我们考虑那些从表面反射回来的光线（例如反射光线），它们也会保留部分一致性。例如：想象一下，现在相机发射了两条光线，这两条光线分别对应了屏幕上两个相邻的像素，它们命中了场景中的一个表面，并在完美镜面反射的方向上进行反弹。在通常情况下，这两条光线会击中场景中的同一个物体，这意味着两个命中点在空间中彼此靠近，并且还可能具有相似的表面法线。因此，这两条反射光线将会具有相似的起点和方向，如图 26.16 所示。

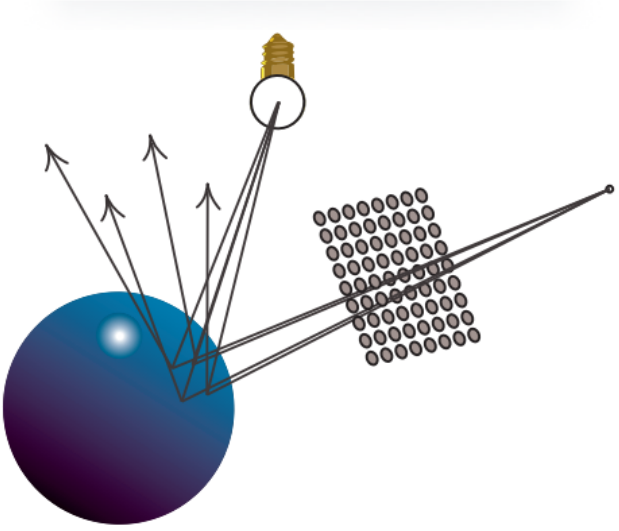


图 26.16：光线从相机中传播出来，击中表面后会产生更多的光线，从而生成阴影效果和反射效果。请注意，所有这些新光线仍然具有相当的一致性，这些光线彼此相似，即具有相似的原点和方向。

当我们需要从给定的表面点对半球的出射方向进行随机采样时，就会产生非相干（没有一致性）的光线。例如：如果我们想要计算环境光遮蔽和漫反射全局光照效果，就会发生这种情况。对于具有光泽（glossy）的反射光线而言，这些光线会更加一致（聚集在一个较小的立体角内）。一般来说，对于我们需要进行追踪的光线集合，以及我们需要计算着色的命中点上，我们可以假设光线追踪确实会表现出一定程度的一致性，我们可以尝试利用这种光线一致性，来对渲染算法进行加速。

利用光线一致性的最早想法之一，就是将光线聚类在一起，例如：我们可以使用一组平行光线（称为光线束 ray bundle）来与场景图元进行相交测试，而不是仅仅使用单条光线。可以利用 GPU 的光栅化器来执行光线束与场景图元的相交测试，使用正交投影来将每个物体的位置和法线存储到离屏缓冲区中。这些缓冲区中的每个像素，都捕获了单条光线的相交数据[81]。另一个早期的想法是将光线分组为不同的锥形[7]，或者是较小的截锥体（frusta，beam）[32]，用于代表具有相同原点的、无限数量的光线，这些光线只会跨越一组有限的方向。Shinya 等人[71]在他们称为 pencil 跟踪的系统中，对光束追踪（beam tracing）的思想进行了进一步的探索和推广。在这个方案中，一个“pencil”被定义未包含了可变的光线原点和方向，例如：如果一条光线的方向允许以一定的角度进行变化，那么这样定义的一组光线就构成了一个实心锥体（solid cone）[7]。

这种方案假定了大面积的连续性。当发现物体的边缘或者大曲率区域时，pencil 就会变成一组更紧密的 pencil，或者使用单独的光线来捕捉这些特征。在立体角上来计算材质和光照的积分，通常也是十分困难的，由于这些限制因素，因此与 pencil 相关的方法没有被广泛使用。一个值得注意的例外是体素化几何的锥形追踪，这是一种最近提出的、在 GPU 上用于近似间接全局照明的技术（章节 11.5.7）。

一种更加灵活的、利用光线一致性的方法是，将光线组织成较小的阵列（被称为数据包 packet），然后对这些光线进行一起追踪。与刚才的 pencil 相类似，这些光线集合有时需要进行拆分，例如：如果它们需要进入 BVH 不同分支的话，如图 26.17。然而，由于这些 packet 只是一种数据结构，而不是几何图元，因此拆分起来要容易得多；我们只需要创建更多的 packet，其中每个 packet 都使用原始光线的一个子集即可。packet 追踪[85]允许我们对空间数据结构进行遍历，并且能够以并行的方式来计算一小组光线与物体的交点，因此它非常适合 SIMD 计算。在实际的实现中[88]，会使用固定大小的 packet，并根据处理器 SIMD 指令的宽度进行调整。如果分组中不存在某些光线（例如在拆分之后），那么则会使用一些标志来屏蔽相应的计算。

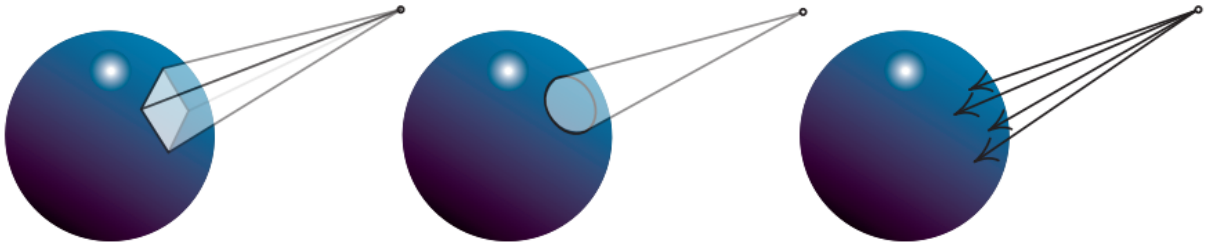


图 26.17：从左到右，球体分别与：光束，锥形、包含四条光线的 packet 相交。

虽然 packet 追踪可以是非常高效的，甚至最近已经被应用在了要求苛刻的 VR 应用中[38]，但是它仍然对光线的生成方式施加了限制。此外，如果光线十分发散的话，那么就需拆分出很多的 packet，这会对它的性能产生影响。在理想情况下，即使是非相干的光线追踪，我们也想要利用现代的数据并行架构。

packet 追踪的思想是利用 SIMD 指令，并行地将多条光线与单个图元进行求交。然而，我们当然可以使用相同的指令，并行地将单条光线与多个图元进行求交，这样就不需要维护 packet 了。如果我们的空间层次结构是使用具有高分支因子的浅层树来构建的（而不是较深的二叉树），那么我们可以通过将单条光线与多个子节点同时进行测试，来进行并行遍历。这些数据结构可以通过将二叉结构部分扁平化来进行构建，例如：如果我们将一个二叉 BVH 的每个其他层级都进行折叠，那么就可以构建一个编码相同节点的 4 叉 BVH。由此产生的数据结构被称为多层次包围体（multi-bounding volume hierarchy，MBVH），有时也被称为浅 BVH（shallow BVH）或者宽 BVH（wide BVH）[15, 19, 87]。

光线追踪的一些应用，例如路径追踪及其变体，这些算法在单条光线上进行工作，无法很容易地生成具有一致性的 packet。但是，如果我们观察生成图像所追踪的所有光线，这些限制也并不意味着我们将无法找到任何程度的光线一致性，即光线一致性总是存在的。场景中的许多光线可能会从相似的原点，沿着相似的方向进行运动，即使我们不能对它们进行明确地追踪。在这些情况下，我们可以对要计算可见性的光线进行动态排序，从而创建出可以进行一致处理的光线组。这些想法是由 Pharr 等人[65]首创的，他们称之为“记忆相干光线追踪（memory-coherent ray tracing）”，这是一种使用空间细分结构中的节点，来存储批次光线的系统。我们并不会对每条光线都在空间结构中进行遍历，并直到找到一个交点；而是会根据节点中所包含的图元，对每个节点中的光线（如果有的话）进行测试，并且会将那些没有命中的光线传播到对应的邻居节点中。因此，会以广度优先的顺序，来对空间细分层次结构进行遍历访问。

通过使用量化的光线方向和光线原点来计算哈希值，可以避免将光线显式存储在场景的空间数据结构中。然后，我们可以维护一个仍然需要进行处理的光线队列，并根据散列键值来对它们进行排序。这个队列相当于在位置和方向的五维空间上创建一个虚拟网格，然后将这些光线分组到这个网格的单元格中。维护光线队列并对其进行动态排序的想法，被称为光线流式追踪（ray stream tracing）或者光线重排序（ray reordering），并已成功地在 CPU 和 GPU 上进行了应用[46, 82]。

最后，还需要注意的是，许多渲染应用程序是由着色操作所主导的，而不是由可见性操作主导的，这就是如今基于光栅化实时渲染的情况。虽然对相干光线进行追踪，也有助于着色计算的一致性，但是并不能保证着色计算一定是一致的。两条光线可能会在场景中命中彼此接近的表面点，但是这两个着色点可能会分别属于不同的物体，它们需要使用完全不同着色器和纹理。这对于依赖宽 SIMD 单元，并在 wavefront（章节 23.2）上同步执行相同指令的 GPU 而言，尤其具有挑战性。对于给定 GPU wavefront 中的光线，如果它们需要使用不同的着色逻辑，那么就必须采用动态分支，这样会导致 wavefront 发散；以及需要使用更大的着色器，更大的着色器通常意味着需要使用更多的寄存器，并导致较低的占用率。

可以对排序进行扩展，从而来解决着色一致性的问题。我们不仅可以在求交的时候使用队列和光线重排序；还可以让光线在命中物体之后，也存储在队列中。然后这些队列可以根据与命中点相关的材质，再次进行排序，然后着色器就可以在具有一致性的批次中进行着色计算。将材质计算与可见性分离的想法被称为延迟着色（deferred shading），这里在光线追踪系统和光栅化系统（章节 20.1）中使用了相同的术语，在光栅化系统中，我们通过屏幕空间的 G-buffer 实现这种解耦。

延迟着色已经成功应用在了离线、产品级的电影路径追踪渲染器中[18, 48]，它会对数百万条光线进行排序（甚至是核外光线）；以及使用较小的队列，来用于 CPU 和 GPU 的光线追踪[3, 46]。然而，对于实时渲染而言，我们必须记住，即使是在能够通过重新排序来利用一致性的系统中，这些排序操作也会增加显著的开销。此外，如果同一时刻运行的光线数量太少，那么它们之间很可能没有值得利用的一致性。此外，当光线击中场景图元的时候，对材质着色器的计算可能会生成新的光线，这可能反过来又会触发其他着色器的递归执行。因此，在计算新生成光线的结果之前，当前着色器的计算可能还需要暂停。这种行为对着色器的执行顺序进行了限制，减少了动态恢复一致性的机会。

在实践中，我们必须注意，尽量使用那些能够利用光线一致性和着色一致性的渲染算法。即使我们需要使用非一致性的光线，仍然存在一些方法能够最小化应用程序的发散，例如：在环境光遮蔽和其他全局光照效果中，需要对屏幕上每个像素的出射半球

方向进行采样。由于这个过程的成本可能会很高，因此一种常见的技术是，我们只在每个像素上采样少量几个样本，然后使用双边滤波（bilateral filtering）来对最终结果进行重建。这通常会导致相邻像素之间的采样方向出现明显不同，即是不一致的。一种优化的实现是，确保采样方向在少量像素上以规则间隔进行重复，并对追踪进行排序，从而使得所有具有相似方向的像素能够被同时处理[43, 50, 68]。

着色计算可能是发散现象的主要来源，甚至影响要比光线追踪更大，因为我们可能需要根据所击中的物体，执行完全不同的着色器程序。在理想情况下，我们希望避免交错执行复杂的着色计算和光线追踪过程，例如：我们可以预先计算着色，并检索光线命中的缓存结果。其中一些策略已经用在了离线、产品级的路径追踪[21]以及交互式的光线追踪[61]中，但是仍然需要更多的研究来推进实时光线追踪技术的发展。一般来说，任何光线追踪应用程序都需要考虑权衡。如果所有光线都相同，那么通常我们可以发射少量的稀疏光线，并利用降噪技术来生成最终图像。然而，对于稀疏且不相干的光线，它们的处理速度较慢，因此有时候发射更多具有较高相干性的光线，总体上的速度会更快。

26.5 降噪

使用蒙特卡罗路径追踪进行渲染，会产生带有噪声的图像，如图 26.6 所示，这些噪声是我们不希望出现的。降噪（denoise）算法的目标是输入一个含有噪声的图像以及其他可选的辅助图像数据，并从中生成一个尽可能与真实图像类似的新图像。在本小节中，我们会以一种非正式的方式来使用“类似（resemble）”一词，因为相较于充满噪声的图像区域，稍微模糊的图像区域在视觉的效果上会更好。降噪对于实时光线追踪而言尤其重要，因为我们通常只能负担得起每个像素发射少量几条光线，这意味着渲染出来的图像可能会充满噪声。例如：图 26.22 中的 PICA PICA 图像使用了每像素大约 2.25 条光线进行渲染[76]。降噪的概念如图 26.18 所示。由于我们可以在降噪器上添加一个反馈回路（feedback loop），如图 26.18 所示，因此时域抗锯齿（章节 5.4.2）可以被认为是一种基本的降噪算法。

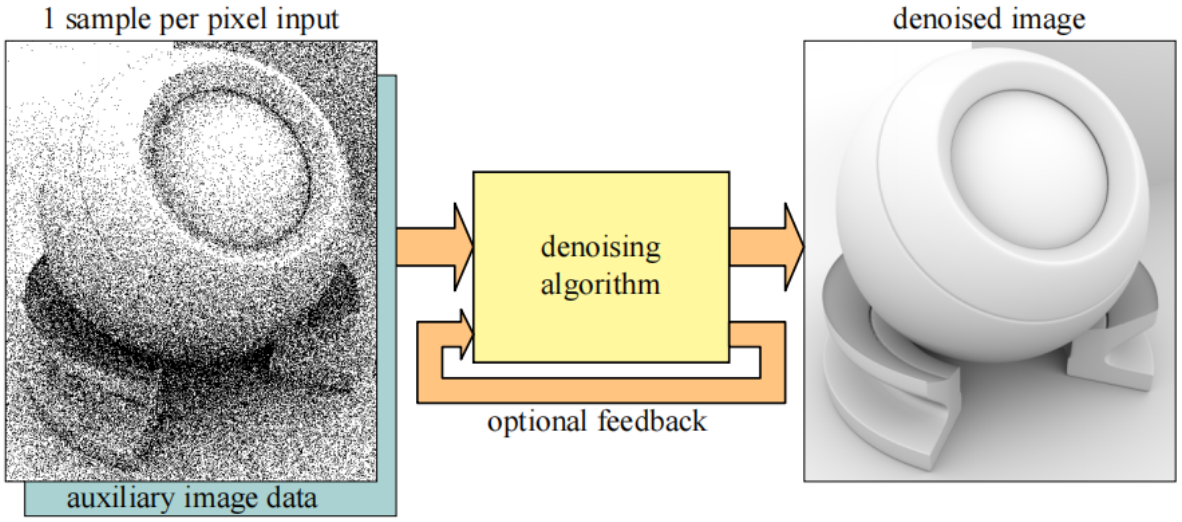


图 26.18：左边是使用光线追踪来计算环境光遮蔽的图像，每个像素都会发射一条遮挡光线。降噪算法会使用这个图像以及可选的辅助图像数据，从中生成降噪图像，如右图所示。其中的辅助图像数据包括：每个像素的深度分量、法线、运动向量，以及着色点到遮挡物的最小距离等。请注意，降噪算法可能会将它的一些输出数据，反馈到下一帧的降噪过程中。

大多数降噪技术可以使用一种简单的方式进行表示，即当前像素周围颜色的加权平均值，我们将当前像素的标量值记为 p 。那么这个加权平均值可以表示为[12]：

$$\mathbf{d}_p = \frac{1}{n}\sum_{q\in N}\mathbf{c}_q w(p,q) \tag{26.3}$$

其中 $\mathbf{d}_p$ 是像素 p 降噪后的颜色值， $\mathbf{c}_q$ 是当前像素周围（包括 p ）的噪声颜色值，$w(p,q)$ 是权重函数。该方程使用了 p 附近的 n 个像素，这个像素集合称为 N ，这个区域通常是正方形的。我们还可以对权重函数进行扩展，让它能够使用来自前一帧的信息，例如 $w\left(p,p_{-1},q,q_{-1}\right)$ ，其中下标 -1 代表了来自前一帧的信息，例如：如果需要的话，权重函数可以访问法线 $\mathbf{n}_p$ 和前一帧中的对应颜色值 $\mathbf{c}_{p-1}$ 。图 24.2、图 24.3、图 26.19、图 26.20 和图 26.21 展示了降噪的例子。

在使用基于光线跟踪算法来进行真实感的实时渲染中，降噪已经成为了一个重要课题。在本小节中，我们将会简要介绍一些重要的工作，并介绍一些可能有用的关键概念。我们推荐将 Zwicker 等人[97]的调研来作为了解更多信息的良好出发点。接下来，我们将重点介绍适用于低样本数量的降噪算法和相关技巧，所谓低样本数量，即每个像素只有一个或者少量几个样本。

为了创建一组没有噪声的渲染目标，我们可以渲染一个 G-buffer（第 20 章），来作为降噪的辅助图像数据，这是十分很常见的[13, 69, 76]。光线追踪可以用来生成噪声

阴影、光泽反射和间接光照等效果。一些方法中所使用的另一个技巧是，将直接光照和间接光照分开进行降噪处理，因为它们具有不同的属性，例如：间接光照通常是相当平滑的。为了增加降噪过程中所使用的样本数量，通常还会使用某种时间积累或者时域抗锯齿（章节 5.4.2）。另一个很好的近似是，对有时被称为无纹理光照（untextured illumination），或者光照和纹理分离的东西进行过滤[96]。为了解释这是如何进行工作的，让我们回顾一下渲染方程（方程 11.2）：

$$L_o(\mathbf{p}, \mathbf{v}) = \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v}) L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{26.4}$$

为了简单起见，这里省略了自发光项 L_e。同时这里我们只处理漫反射项，尽管类似的过程也可以应用于其他项。接下来，我们计算一个反射率项 R，它本质上就是一个漫反射着色项再乘以纹理（在表面被纹理化的情况下）：

$$R \approx \frac{1}{\pi} \int_{\mathbf{l} \in \Omega} f(\mathbf{l}, \mathbf{v})(\mathbf{n} \cdot \mathbf{l})^+ d\mathbf{l} \tag{26.5}$$

然后无纹理光照 U 可以表示为：

$$U = \frac{L_o}{R} \tag{26.6}$$

也就是说，这里的纹理项已经被分离了，因此 U 应当只包含光照信息。因此，渲染器可以使用渲染方程来计算 L_o，并执行一次纹理查找和漫反射着色来获得 R，这样我们就获得了无纹理光照 U。然后我们可以将这一项降噪为 D，然后最终的着色约等于 DR。这样就避免了在降噪算法中对纹理进行处理，这是十分有利的，因为纹理中通常会包含高频内容，例如边缘等。Heitz 等人[35]所做的工作也类似于这个无纹理照光照的技巧，他们将最终图像拆分成一个噪声的阴影项，以及面光源的解析着色项；并对阴影项进行降噪处理，最后再将图像进行重新组合。这种类型的拆分通常被称为比率估计器（ratio estimator）。

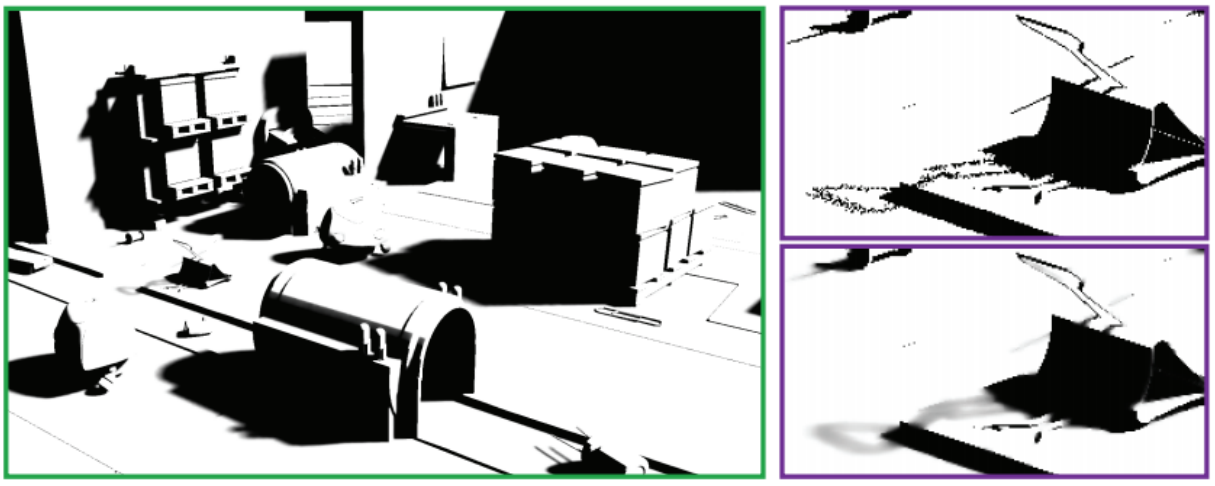


图 26.19：左侧：过滤后的阴影项。右上角：对面光源使用单个样本的放大阴影项。右下角：对右上角图像进行降噪后的结果。阴影会在预期的地方会变得更加平滑，而接触阴影则会更加尖锐。

可以使用时空方差引导滤波（spatial-temporal variance-guided filtering，SVGF）[69]来实现对软阴影的降噪，例如 SEED [76]就使用了这个方法，如图 26.19 所示。SVGF 最初的目的是：针对每个像素一个样本（one-sample-per-pixel，1spp）的路径追踪图像进行降噪。即使用 G-buffer 来获得第一次的可见性，然后在第一次和第二次命中场景的时候，发射一条次级光线和阴影光线。其总体思路是使用时间积累（章节 5.4.2）来增加有效的样本数量，并使用空间中的多 pass 模糊处理[16, 29]，其中模糊核的具体尺寸由噪声数据的方差估计所决定。

随着所使用样本数量的增加，方差可以进行增量计算，从而包含更多的样本。首先，计算差值的平方和：

$$s_n = \sum_{i=1}^{n} (x_i - \bar{x}_n)^2 \tag{26.7}$$

其中 $\bar{x}_n$ 是前 n 个数字的平均值。方差为：

$$\sigma_n^2 = \frac{s_n}{n} \tag{26.8}$$

现在，假设我们已经使用了 $x_1, \ldots, x_n$ 计算了 s_n，现在我们又获得了一个心样本 x_{n+1} 我们想要将其包含在方差计算中。可以使用下列方程来更新求和：

$$s_{n+1} = s_n + (x_{n+1} - \bar{x}_n)(x_{n+1} - \bar{x}_{n+1}) \tag{26.9}$$

然后再使用方程 26.8 来计算 σ_{n+1}^2 。在 SVGF 中，会使用这样的方法来估计随着时间变化的方差，但是如果检测到了一个解除遮挡（disocclusion）（译者注：即上一帧被遮挡的表面，这一帧出现在了画面中），这会使得时间方差变得不再可靠，此时会切换到使用空间估计。对于软阴影的降噪，Llamas 和 Liu [51, 52]使用了一个可分离的交叉双边滤波器（章节 12.1.1），该滤波器的权重和半径都是可变的。

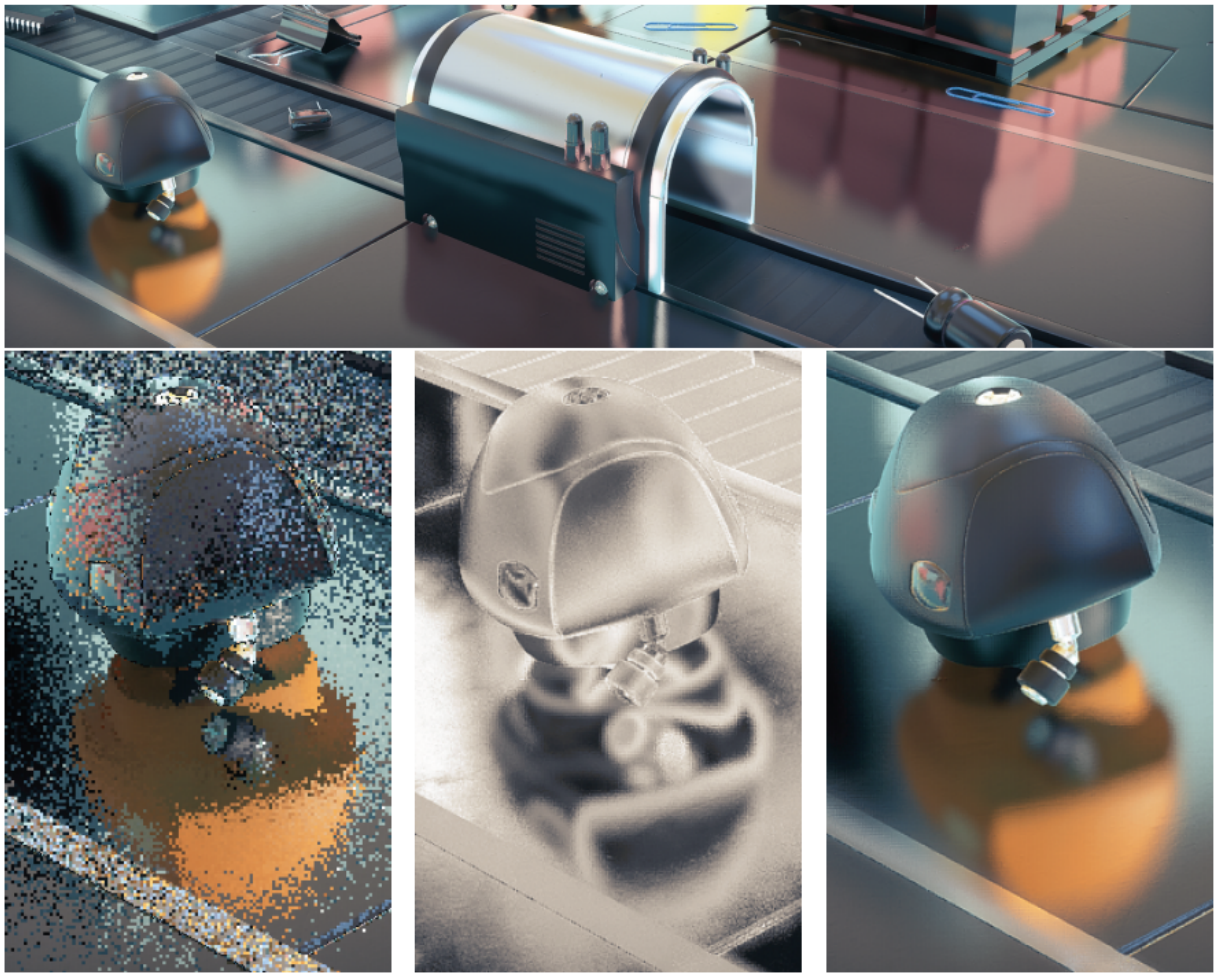


图 26.20：上图：降噪之后仅展示反射项的截图。左下角：降噪之前，在每 2×2 像素上，使用 1 条反射光线（译者注：即在屏幕半分辨率下进行光线追踪）。底部中间：展示了方差项，像素越亮，代表方差越大，从而会使用尺寸更大的模糊核。右下角：全分辨率下的降噪反射图像。

Stachowiak 提出了一个完整的反射降噪管线[76]，图 26.20 展示了一些示例。首先渲染一个 G-buffer，并从中发射光线。为了减少追踪光线的数量，只会在每 2×2 像素的反射点处，发射 1 条反射光线和 1 条阴影光线；然而，会在全分辨率下进行图像重建。这里的反射光线是随机的，并进行了重要性采样。这里的“随机（stochastic）”意味着，随着更多随机生成光线的加入，最终会收敛到正确的结果。“重要性采样（importance sampled）”意味着会对最终结果预期更加有用的方

向发射光线，例如：向着 BRDF 的峰值方向发射光线。然后使用类似于屏幕空间反射[75]（章节 11.6.5）的方法，来对图像进行过滤，同时上采样到完整的图像分辨率，这个滤波器也是一个比率估计器[35]。这个技术会与时间累积、一个双边清理 pass 以及最后的 TAA 相结合。Llamas 和 Liu 提出了一种基于各向异性滤波核的不同反射解决方案。

为了开发滤波方法和自适应采样技术，Metha 等人[54, 55, 56]采用了一种基于傅里叶分析（Fourier analysis）的光线传输理论方法[17]。他们开发了轴对齐的滤波器，相较于更加准确的剪切滤波器（sheared filter），它的计算速度要更快，这样做可以带来更高的性能表现。有关更多信息，请参阅 Metha 的博士论文[57]。

对于环境光遮蔽（AO）效果，为了提高效率，Llamas 和 Liu [51, 52]使用了一种使用可分离的交叉双边滤波器（章节 12.1.1）来实现轴对称滤波核[54]。这个滤波核的尺寸是由在追踪 AO 射线时，发现到物体的最小距离来决定的。当遮挡物较近时，滤波器的尺寸就较小；当遮挡物较远时，滤波器的尺寸就较大。这种关系可以为较近的遮挡物提供更加明显的阴影，而当遮挡物距离较远时，可以提供更加平滑、更加模糊的阴影效果，如图 26.21 所示。

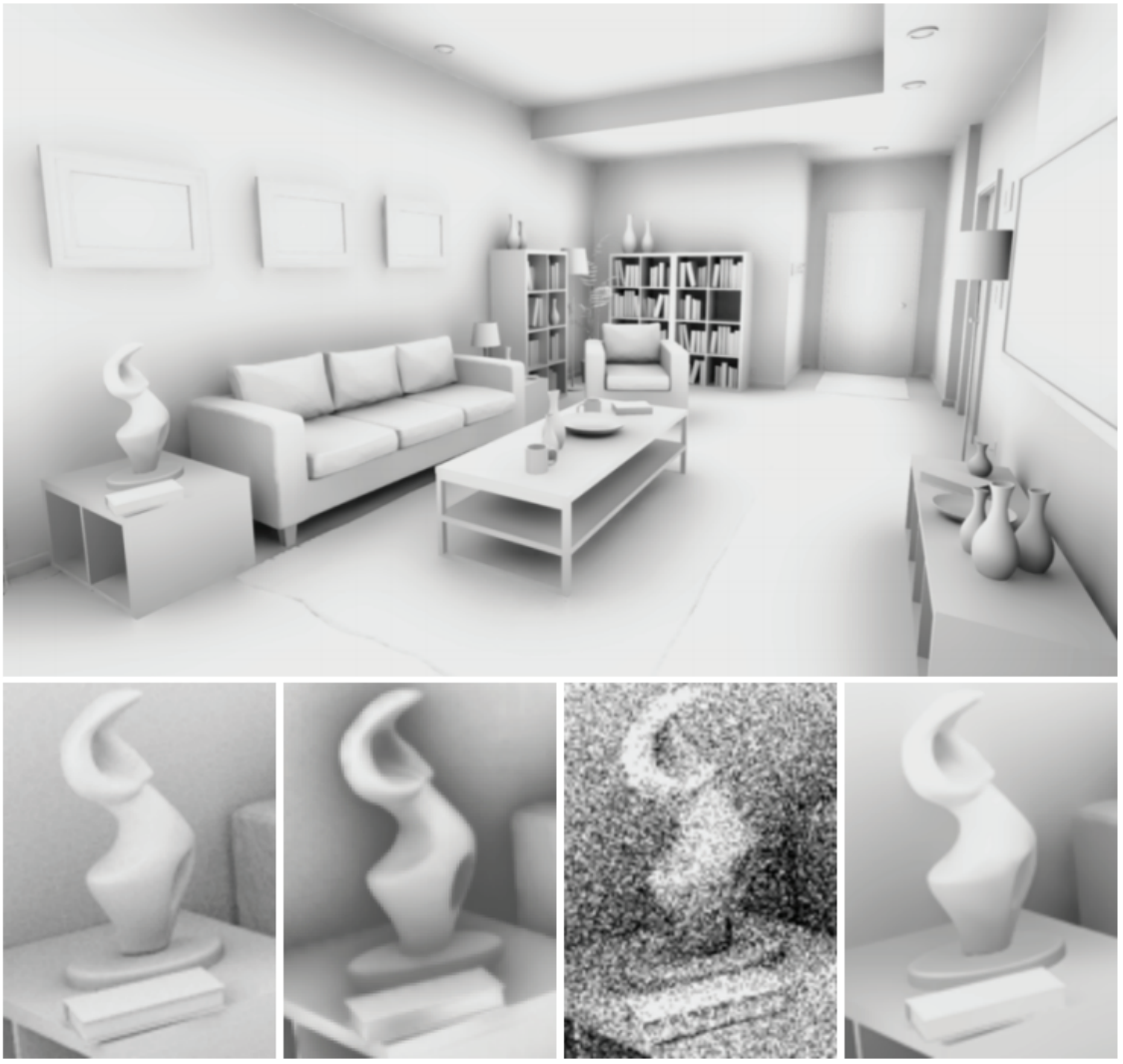


图 26.21：顶部图像中的环境光遮蔽效果，使用每像素一条光线来进行光线跟踪，然后再进行降噪。底部是放大后的图像，从左到右分别是：ground truth；屏幕空间环境光遮蔽（SSAO）；光线追踪环境光遮蔽，每帧中的每像素使用一个样本；对每像素一个样本的光线追踪结果进行降噪。降噪后的图像无法捕获所有较小的接触阴影，但是仍然要比屏幕空间环境光遮蔽更接近 ground truth。

有几种方法可以对全局光照效果进行降噪[13, 51, 55, 69, 70, 76]。对于不同类型的效果，也可以使用专门的过滤器，这就是寒霜引擎和 SEED [9, 36, 76]的研究人员所采用的方法，如图 26.22 所示。

图 26.22。PICA PICA 项目的最终图像，它结合了光栅化和光线追踪，然后使用了几个降噪滤波器。

寒霜引擎的实时光照贴图预览系统[36]，依赖于基于方差的降噪算法。光照贴图中的纹素会存储累积下来的样本贡献，并对它们的方差进行追踪。当新的路径追踪结果出现，并在呈现给用户之前，会根据每个纹素的方差，来对光照贴图进行局部模糊处理。这种基于方差的模糊类似于 SVGF [69]，但它并不是分层的，从而避免了属于不同网格的光照贴图元素相互泄漏。在进行模糊处理的时候，只有来自相同光照贴图元素的样本，才会使用逐纹素的索引被模糊在一起。为了获得无偏的收敛结果，原始的光照贴图将会保持不变。

另一种选择是在纹理空间中来执行降噪[61]。它要求所有表面位置都具有唯一的 uv 空间值，然后在表面命中点的纹素处进行着色。纹理空间中的降噪可以简单地对具有相似法线的纹素，对其着色结果进行平均，例如在一个 13×13 的区域中。Munkberg 等人是这样设计的，如果 $\cos\theta > 0.9$，就将该纹素纳入平均处理，其中 θ 是待降噪纹素上的法线与待包含纹素上法线之间的夹角。在纹理空间执行降噪的其他一些优点是，在纹理空间中的时域平均是十分直接的，可以通过在一个较粗的层级上计算着色，从而降低着色成本，并且着色计算可以平摊在若干个帧上。

到目前为止，我们一直假设场景中的物体是静止的，并且摄像机也没有发生移动。但是，如果包含了运动情况，并且图像在渲染的时候使用了景深效果，那么深度和法线中也可能会存在噪声。对于这些情况，Moon 等人[60]开发了一种各向异性滤波器，

可以逐像素地计算世界位置样本的协方差矩阵（covariance matrix），并使用这个协方差矩阵来估计每个像素上的最佳滤波参数。

通过使用深度学习算法[24, 25]，可以利用游戏引擎或者其他渲染引擎生成的大量数据，并使用这些数据来训练神经网络，从而生成降噪图像。这个想法是首先建立一个卷积神经网络（convolutional neural network，CNN），然后使用噪声图像和无噪声图像来对网络进行训练。如果做得好的话，这个网络将会学习到大量的权重，它可以在随后的推理步骤中，在不需要知道任何额外的无噪声图像数据的前提下，使用这些权重来对噪声图像进行降噪。Chaitanya 等人[13]在编码管线（encoding pipeline）的所有步骤中，都使用一个带有反馈回路的卷积神经网络。添加这些是为了增加时域稳定性，从而减少动画期间的闪烁。即使没有干净的参考数据，也可以使用（不相关的）噪声图像来对降噪器进行训练[49]。这样做可以使得训练过程更加简单，因为不需要生成 ground truth 图像。

很明显，降噪是真实感实时渲染中的一个重要主题，并且它将继续是，在这一领域中将会继续进行更多的研究。

26.6 纹理过滤

我们在章节 3.8 和章节 23.8 中介绍过，光栅化会对 2×2 一组（称为 quad）的像素进行着色处理。这样做是为了估计纹理足迹（texture footprint），计算梯度信息，并在 mipmap 纹理硬件单元中进行计算和使用。对于光线追踪而言，情况就有所不同了，光线通常是相互独立发射的。我们可以想象这样的一个系统，当每条眼睛光线与三角形平面相交的时候，还会有另外两条光线与这个三角形平面相交，其中第一条光线会在水平方向上偏移一个像素，第二条光线会在垂直方向上偏移一个像素。在许多情况下，使用这种技术可以生成准确的纹理滤波器足迹，但是它仅仅适用于眼睛射线。然而，如果现在相机正在观察一个反射表面，并且反射光线击中了一个具有纹理的表面，此时会发生什么呢？在这种情况下，理想的做法是执行一个过滤的纹理查找，这里所说的过滤，需要同时考虑反射的性质和光线传播的距离。这个做法同样适用于折射表面。

Igehy [39]为这个问题提供了一个复杂的解决方案，他使用了一种叫做光线微分（ray differential）的技术。每条光线都需要存储：

$$\left\{ \frac{\partial \mathbf{o}}{\partial x}, \frac{\partial \mathbf{o}}{\partial y}, \frac{\partial \mathbf{d}}{\partial x}, \frac{\partial \mathbf{d}}{\partial y} \right\} \tag{26.10}$$

这些附加数据会与光线一起进行存储。回顾一下，$\mathbf{o}$ 是光线原点，$\mathbf{d}$ 是光线方向（方程 26.1）。由于原点 $\mathbf{o}$ 和方向 $\mathbf{d}$ 都包含三个分量，因此上面的光线微分需要 $4 \times 3 = 12$ 个额外的数字来进行存储。当发射眼睛光线时，$\partial\mathbf{o}/\partial x = \partial\mathbf{o}/\partial y = (0, 0, 0)$，因为此时光线是从单个点发射的。然而，$\partial\mathbf{d}/\partial x$ 和 $\partial\mathbf{d}/\partial y$ 将会模拟每条光线通过像素时的扩散程度。当光线从一个点转移到另一个点时，需要对光线微分进行更新。此外，当光线微分被反射和折射时，Igehy 推导了光线微分会如何发生变化的方程；并且对于具有插值法线的三角形，Igehy 还推导了微分法线的方程。

另一种更加简单的方法是基于锥形追踪，它由 Amanatides 在 1984 年首次提出[7]。这项工作的重点在于对几何图形进行抗锯齿，但他简要地提到，光线锥（ray cone）也可以用于光线追踪中的纹理过滤。Akenine Moller 等人[6]提出了一种与 G-buffer 一起实现光线锥的方法，该方法还考虑了第一次命中时的曲率。滤波器的足迹取决于到命中点的距离、光线的扩散角度、命中点的法线和曲率等。然而，由于这个曲率信息只能在第一次命中时提供，因此较深的反射可能会出现锯齿。他们还提出了几种与 G-buffer 渲染相结合的光线微分方法，并对这些方法进行了比较。

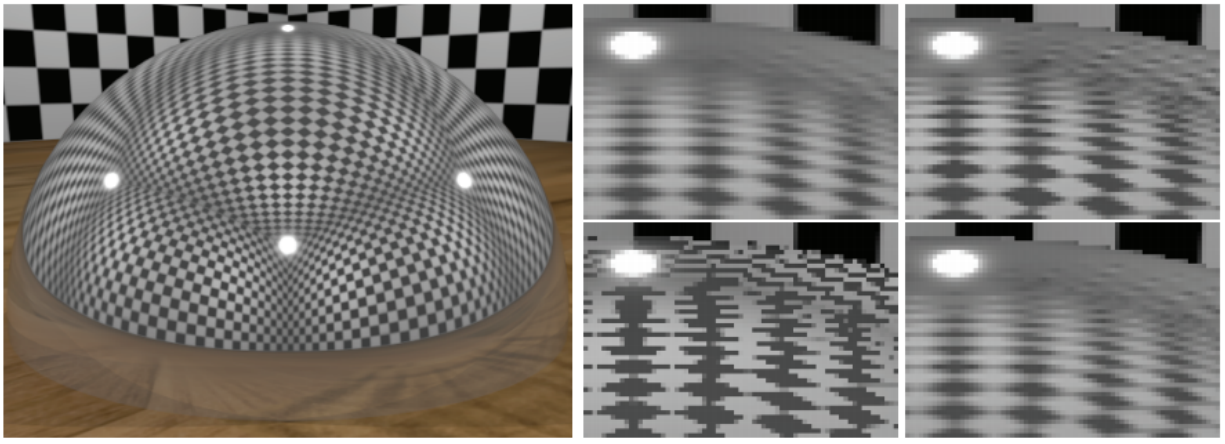


图 26.23：房间里有一个反射半球，墙上和天花板上有棋盘格图案，地板是木头材质。在右边，展示了在几种不同的纹理滤波方法下，同一区域的放大结果。顶部中间： ground truth 渲染结果，每个像素使用了 1024 个样本，每个样本都使用了一个双线性的纹理查找。底部中间：总是通过双线性过滤来访问第 0 层级的 mipmap（即没有 mipmap）。右上角：使用光线锥。右下角：使用光线微分。

图 26.23 展示了四种不同的纹理过滤方法。使用第 0 层级的 mipmap（即没有 mipmap）会导致严重的锯齿。光线锥为这个场景提供了稍微尖锐的结果，而光线微分则提供了更加接近的 ground truth 结果。光线微分通常能够更好地估计纹理足迹，但有时也可能会过度模糊。根据他们的经验，光线锥可能会发生过度模糊，也可

能会发生模糊不足。Akenine-Moller 等人提供了用于光线追踪纹理过滤的光线微分实现和光线锥实现。

26.7 推测

由扩展 API（章节 26.2）提出的新类型着色器，允许复杂的形状求交和材质表示。对追踪进行光线的相关能力，导致了它在网格渲染中的明显应用，例如：表面光照，阴影，反射，折射和路径跟踪全局光照等。

跳出常规思维（thinking outside the box），这些功能可以实现在 API 设计阶段没有考虑到的新情况。在不久的将来，我们将会看到开发者们究竟能够想出什么令人兴奋的东西。光线生成着色器会成为新的计算着色器吗？本小节将对此进行讨论，并一些可能性进行探索。

在降噪算法的帮助下（章节 26.5），新的光线追踪功能应该可以实时渲染更加高级的表面，以及更加复杂的光照。其中一个改进就是使用光线追踪来计算场景中的所有反射，而不是使用屏幕空间反射，例如在游戏《战地 5》中。这样做可以产生更好的接地物体（grounded object），并可以对任何形状网格上的镜面反射和遮挡情况进行改善。诸如屏幕空间反射这样的技术之所以会有效，部分原因是因为对假设进行了简化，例如：表面上的反射效果是围绕主反射方向轴向对称的。当反射样本在 BRDF 半球上的分布更加复杂时，光线追踪能够给出更加精确的结果。

次表面散射是另一种可以使用光线追踪来更好模拟的相关现象。第一个版本的次表面散射已经通过使用新的 API 特性实现了，通过对网格内部的散射光线进行追踪，在纹理空间中对来自多个方向的样本进行时域累积，并使用一些降噪算法[9]。总体而言，阴影、间接光照和环境光遮蔽也将受益于光线追踪，从而使得任何网格在场景中看起来更加接近真实结果[35, 51]。

参与介质的应用（第 14 章）在实时应用程序（例如游戏）中变得越来越重要。体渲染未来将会如何发展，也许会与通常基于体素和光线步进的方法有所不同，这值得密切关注。可能会出现使用光线追踪，并依赖 Woodcock 跟踪（tracking）[95]进行重要性采样，并结合一些降噪过程的新方法。

类广告牌的渲染方式（第 13 章）在实时应用中应用广泛。为一个视图渲染粒子物体，这本身就是一个挑战。为了获得正确的透明度效果，我们需要对透明物体进行排序；对于较大的粒子而言，过度绘制也是一个问题，并且还需要考虑光照性能和光照质量之间的权衡。由于粒子通常都会面向相机，因此在反射中进行渲染也是一个挑战。当一个粒子可以被来自任意方向的光线命中时，它应当如何在路径追踪框架中进

行对齐呢？它是否应当具有一个新的表示方法，并利用相交着色器，始终将每个广告牌与入射光线对齐？最重要的是，由于粒子固有的动态特性，因此这些动画粒子将不得不在每一帧的加速结构中，来对它们的表示结构进行更新（章节 26.3）。这种更新可能会发生在几个大粒子上，或者是许多小粒子上，例如：烟雾或者火花，对于这些粒子效果而言，如何对空间加速结构进行优化，从而对世界进行快速追踪是一项挑战。

与阴影采样类似，投射光线的能力也为更加精确的求交和可见性查询打开了大门，例如：可以更加精确地处理粒子碰撞问题。通常的屏幕空间近似存在分辨率依赖性的问题，并且在某些情况下，对前景深度层厚度的计算可以防止粒子落在这一层的背后。整个刚体物理系统可以在 GPU 上，使用光线投射进行实现吗？此外，光线追踪还可以用于查询两个位置之间的可见性。这种能力对于音频混响模拟（audio reverb simulation）、游戏玩法（gameplay）和 AI 系统中元素与元素之间的可见性可能会非常有价值。

光线追踪 API 添加了一种新的着色器类型，我们在前面没有提到，它就是可调用着色器（callable shader）。它能够以可编程的方式，从一个着色器中生成其他的着色器工作，这个功能以前只有在 CUDA 中才能实现[1]。这种类型的着色器目前还不可用，只能在光线追踪着色器设置中进行使用。根据这个着色器的实现方式以及它的性能，这个功能很可能会成为一个非常有价值的通用新工具，特别是如果它可以用于着色器的其他阶段的话，例如计算。举个例子：一个可调用着色器可以移除许多在引擎中生成的着色器排列组合（shader permutation，译者注：感觉类似于着色器变体），从而提高性能，例如：这里的每个排列都代表了一组针对场景设置进行优化的后处理着色器（post-process shader）。在可调用着色器中包含可选的设置依赖代码（setting-dependent code），可以减少由着色器排列所带来的内存占用，这个内存占用是不可忽略的，如果我们必须要处理 5 个性能关键的设置选项，那么比起使用 $2^5 = 32$ 个着色器，我们现在只需要分配一个着色器即可，并根据需要来调用 5 个子着色器。它也可以使用由艺术家创建的 shader graph，这不仅仅适用于今天通常所做的材质定义，而且可以以一种更加模块化的方式，应用于渲染链中的每个部分：应用于透明网格的贴花体积、光照功能、参与介质材质、天空、以及后处理等。

让我们以透明网格上的贴花体积为例，它们通常是由艺术家创作的 shader graph 来进行定义的。通过对与贴花体积相交的 G-buffer 像素内容进行修改，可以直接将它们应用于延迟上下文中的不透明网格上（章节 20.2）。在前向渲染或者光线追踪过程中，它需要一个复杂的着色器，这个着色器需要能够对项目中由艺术家创作的、所有不同类型的贴花 shader graph 进行计算。这种巨大的着色器是不切实际的。通过

使用可调用着色器，可以为每个与世界空间位置相交的贴花体积调用对应的着色器，并让它对着色所考虑的材质进行修改。使用这个功能将取决于具体实现和用法：一个着色器能够被调用，并返回任意数据结构吗？可以同时调用多个着色器吗？着色器参数是如何进行传递的？它们必须通过全局内存进行传递吗？派生着色器是否可以被限制在一个计算单元上，以便能够通过共享内存进行通信吗？这会是一种“即发即弃（即没有返回值）”的调用吗？

图 26.24：小酒馆的外部场景，使用路径追踪渲染器生成，并具有景深效果。使用 DXR 实现。

毫无疑问，这些问题的答案将会推动进一步的创新，并在不久的将来实现。最后，图 26.24 展示了一个使用 DXR 进行渲染的结果，另一个结果就是本书封面。前途一片光明（The future looks bright）！

补充阅读和资源

请参阅本书的在线网站 realtimerendering.com，从而了解该领域中的最新信息和免费软件。Shirley 关于光线追踪的迷你书籍（光追周末系列）[72, 73, 74]，是对不同阶段光线追踪的优秀介绍，现在提供了免费的 pdf。产品级光线追踪的最佳资源之一，就是是 Pharr 等人[66]的《Physically Based Rendering》，现在也是免费的。Suffern 的书籍《Ray Tracing from the Ground Up》[79]相对较早，但是内容广泛，并对实现方面的问题进行了讨论。对于 DXR 的介绍，我们推荐 Wyman 等人[91]的 SIGGRAPH 2018 课程以及 Wyman 的 DXR 教程[92]。想要了解更多有关产品级

渲染中的路径追踪，请参阅 Fascione 等人[20, 21]最近的 SIGGRAPH 课程。在 ACM TOG 的特刊[67]中，有着更多关于路径跟踪和其他产品级渲染技术的文章。Zwicker 等人[97]在一份调研中详细讨论了降噪技术，尽管这份资料来自于 2015 年，它并不会涵盖最新的研究进展。