世界著名计算机教材精选

PEARSON Prentice Hall

计算机图形学

（OpenGL版）（第3版）

Francis S Hill, Jr.
Stephen M Kelley 著

胡事民 刘利刚
刘永进 张松海 等译

COMPUTER GRAPHICS USING OpenGL

Third Edition

清华大学出版社

会引起错误。我们现在投影一条线段,它的一端在眼睛前面,另一端在眼睛后面。

图 7.23 中再一次从上向下看摄像机。在视点前面的点 A 以一种合理的方式投影到点 A'。在另一端的位于眼睛后面的点 B 同样投影到 B',它似乎位于了视平面错误的一边。考虑点 C 从 A 移动到 B 时,画出它是怎么投影的。当 C 向着视点平面往回移动时,它的投影点沿着视平面越来越向右滑动。当 C 接近视点所在的平面时,它的投影点到了无穷远处,当 C 在视点后面运动时,它的投影点"绕过无穷远"从相反的方向出来了【Blinn96】。如果我们在算法中试图绘制出这样的一条直线,那么结果可能是一片混乱。所以一般来说直线中比近平面更靠近视点的那一部分会在投影之前就被裁剪掉。

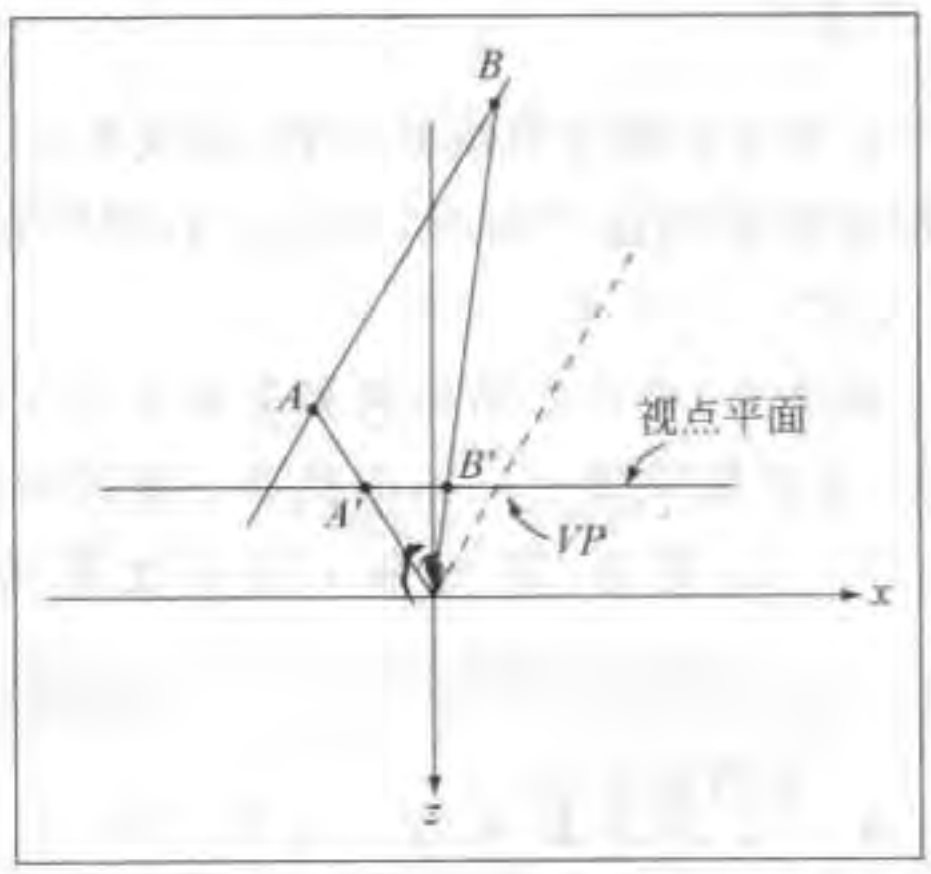


图 7.22 灭点的几何示意图

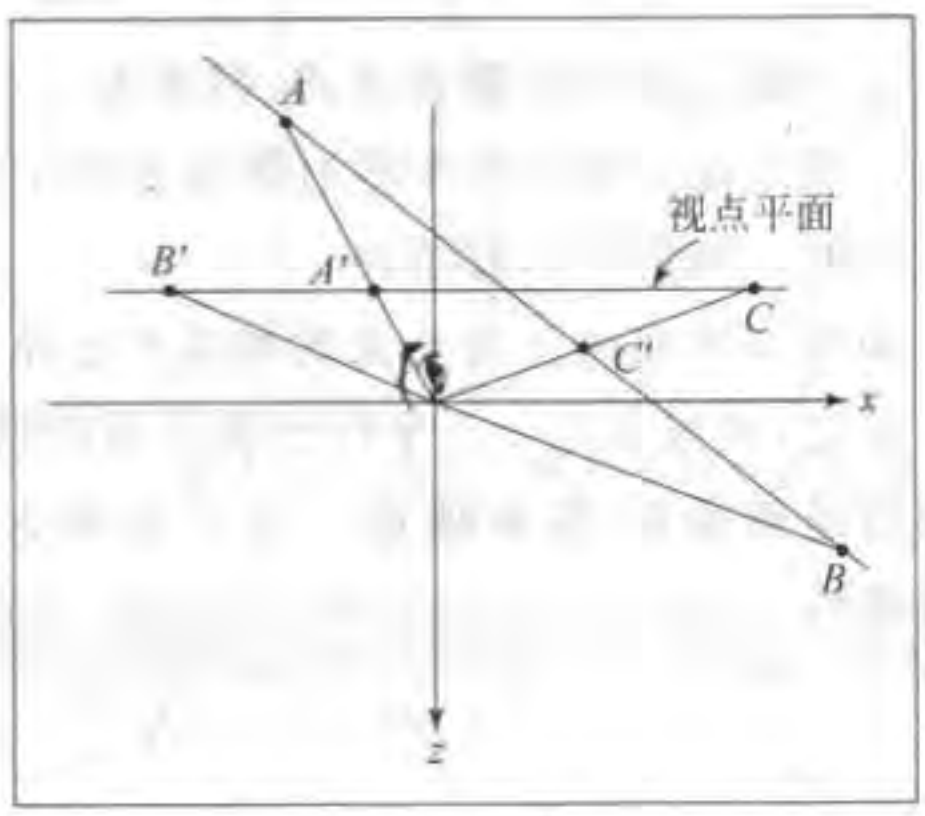


图 7.23 直线 AB 的投影,其中点 B"在视点后"

例 7.4.3 透视中的经典水平平面

观察灭点的一个好办法是观察透视情况下的直线网格,如图 7.24 所示。这里网格线处在 xz 平面上,且均为单位长度间隔。眼睛位于平面上方一个单位处,在(0,1,0),沿着 $-\mathbf{n}$ 的方向看,其中 $\mathbf{n}=(0,0,1)$。和往常一样我们取 $\mathbf{up}=(0,1,0)$,N 定为 1。

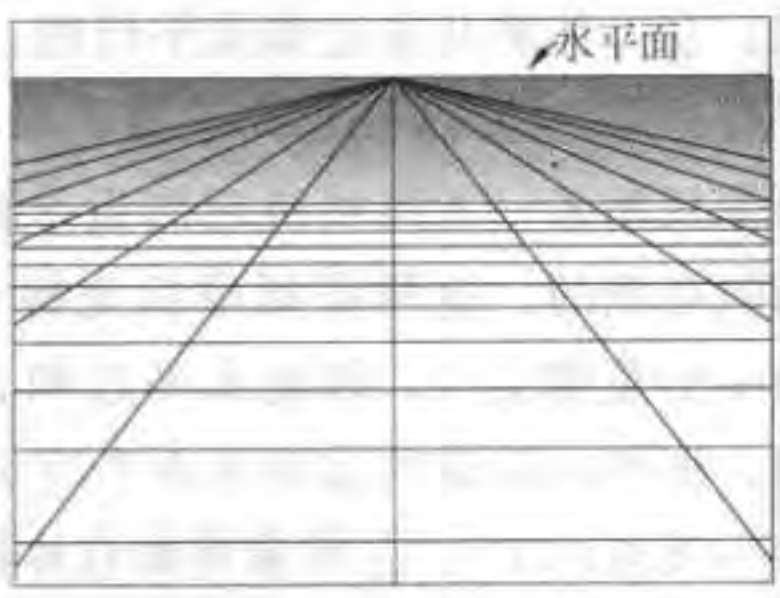


图 7.24 在 xz 平面的水平栅格的透视效果

在视点坐标系下,x 不变的网格线的参数表达式为 $(i,0,t)$,其中 $i=\cdots,-2,-1,0,1,2,\cdots$,$t$ 从 0 到 ∞。通过等式(7.4),第 i 行直线投影到 $(-i/t,2/t)$,是一条经过灭点(0,0)的直线。所有这些直线如预期一样都相交于这个灭点。

在视点坐标系下,z 不变网格线的参数表达式为 $(t,0,-i)$,其中 $i=1,2,\cdots,N$,t 从 $-\infty$ 到 ∞。这些直线投影到 $(-t/i,-2/i)$,这些都是水平直线。它们的投影直线都是平行的,因为网格线本身平行于视平面,越远的直线相互靠得越近,这为我们提供了一个生动的透视缩短的实例。很多遥远的线没有画出来,因为它们已经密得看不清楚了。地平线由所有的 z 是很大的负值的线组成;它位于 $y=0$ 处。

观察很长的平行线时的异常情况

透视投影应该是我们观察方式的合理模型。但是有很多异常情况，这主要由于我们的眼睛没有平坦的观察屏幕。这个问题发生在非常长的物体上面，例如你向上看一对平行的电话线，如图 7.25(a)所示。

从透视视角，如果我们把视平面设置得和电话线平行，我们知道图像将会显示两条平行直线(见图 7.25(b))。但是我们看到的却是不同的情况：电线好像是弯曲了，并且两边都集中于灭点(见图 7.25(c))。实际上这个异常几乎是不可见的，因为窗口或者你的眼睛把观察到的区域限制在一个合理的区域(为了看到电线不同的部分，你需要向上向下翻转你的眼睛，这当然会旋转视平面)。

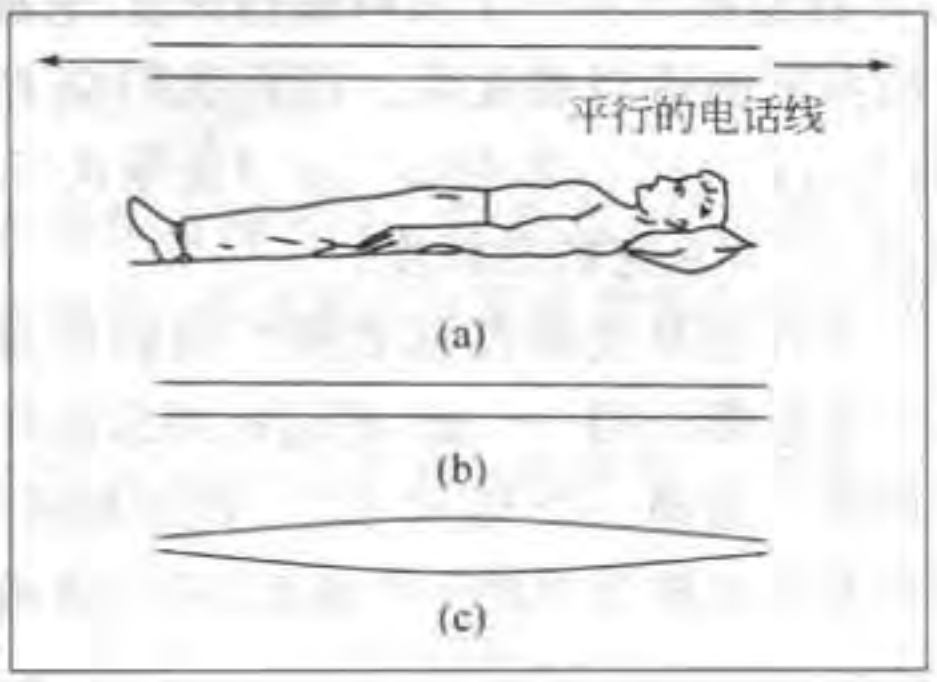


图 7.25 非常长的两条平行线的观察效果

练习

练习 7.4.3 直线投影成直线：参数形式

证明等式(7.5)中的参数表达式是一条直线。提示：对于 x 分量，把分母除到分子上得到 $-A_xN/A_z+Rg(t)$，其中 R 依赖于 A 和 $\mathbf{c}$ 的 x 分量，但不依赖于 y 分量，同样 $g(t)$ 是 t 的函数，不依赖于 x 或者 y 分量。y 分量重复上述过程，得到 $-A_yN/A_z+Sg(t)$ 具有相似的性质。讨论这个为什么是一条直线的参数表示(虽然当 t 变化时点的移动速度并不恒定)。

练习 7.4.4 推导水平网格的结果

推导例 7.4.3 中水平网格投影的参数表示形式。

7.4.3 在图形绘制管道中加入透视投影

我们需要图形绘制管道把物体的顶点投影到近平面上，然后再映射到视口中。在通过模型视点矩阵以后，顶点就变为在摄像机坐标系中的表示，同时等式(7.4)展示了我们所要计算的恰当的投影值。我们需要作裁剪，然后把未被裁剪的点映射到视口。但是除此之外，我们还需要做一些额外的工作。

增加伪深度

在进行投影时我们会丢弃深度信息——也就是说点离眼睛的距离。但是我们一定不能完全丢弃这些信息，否则后面无法进行隐藏面消除。

在视点坐标下，一个点到眼睛的实际距离是 $\sqrt{P_x^2+P_y^2+P_z^2}$，对感兴趣的每一个点计算这个值是烦人和耗时的。我们真正想要知道的只是一种距离的度量方法，使得当两个点投

影到近平面上同一个点时能够告诉我们那一个点更近一些。图 7.26 展示了位于同一条从眼睛出来的直线上的两点 P_1 和 P_2,因此它们投影到同一个点。我们必须能够检验到底 P_1 是遮挡 P_2 还是 P_2 遮挡 P_1。因此对于投影的每一个点 P 我们都计算一个值叫做伪深度,它提供一个 P 的深度的适当的度量。以后我们说 P 点投影到(x^*, y^*, z^*),其中(x^*, y^*)是等式(7.4)提供的值,z^* 就是它的伪深度。

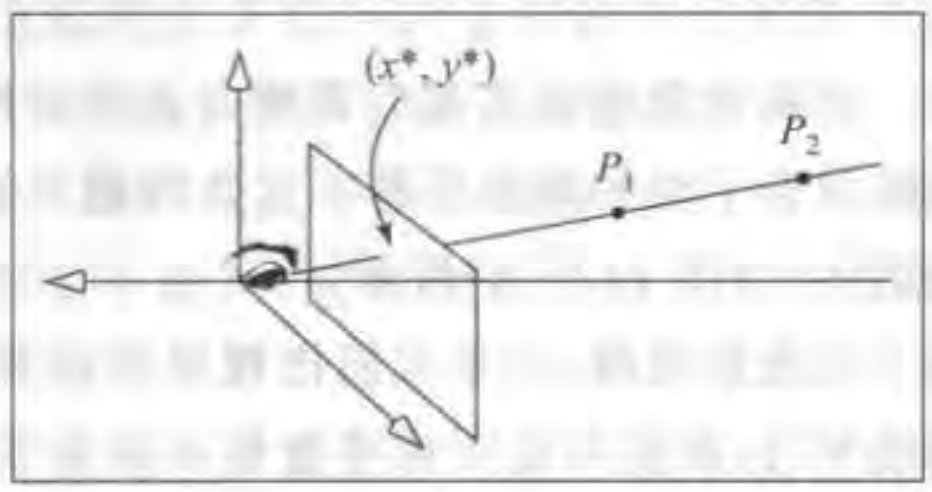


图 7.26 P_1 和 P_2 哪个点更近?

怎样选择伪深度函数呢?我们注意到如果两个点投影到同一个点,更远的点总是有一个更大的负值 P_z,因此我们使用 $-P_z$ 作为伪深度的值。选择一个与 x^* 和 y^* 具有相同分母 $-P_z$ 的函数是很协调和高效的,因此我们尝试使用具有这样分母的一个函数,同时该函数的分子与 P_z 成线性关系,我们说 P 投影到:

$$(x^*, y^*, z^*) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z+b}{-P_z}\right) \tag{7.7}$$

选择常量 a 和 b。尽管很多 a 和 b 都很好用,但是我们应该选择它们使得伪深度处于 -1 和 1 之间(后面会看到这个选择为什么好)。由于当点沿着 z 轴向下移动得更远时深度不断增加,我们决定当 $P_z=-N$ 时伪深度为 -1,当 $P_z=-F$ 时伪深度为 1。在这种情况下,我们很容易计算出 a 和 b,得到:

$$a=-\frac{F+N}{F-N},\quad b=\frac{-2FN}{F-N} \tag{7.8}$$

图 7.27 绘制出了伪深度与 $-P_z$ 的相对关系。它从近平面上点的值 -1 增长直到远平面上点的值 1。当 P_z 靠近 0 时(此时点就在眼睛前),伪深度接近 $-\infty$。对于在眼睛后的点,伪深度是一个很大的正值。但是我们会裁剪掉所有比近平面还近的点,因此这种灾难性的行为不会出现。

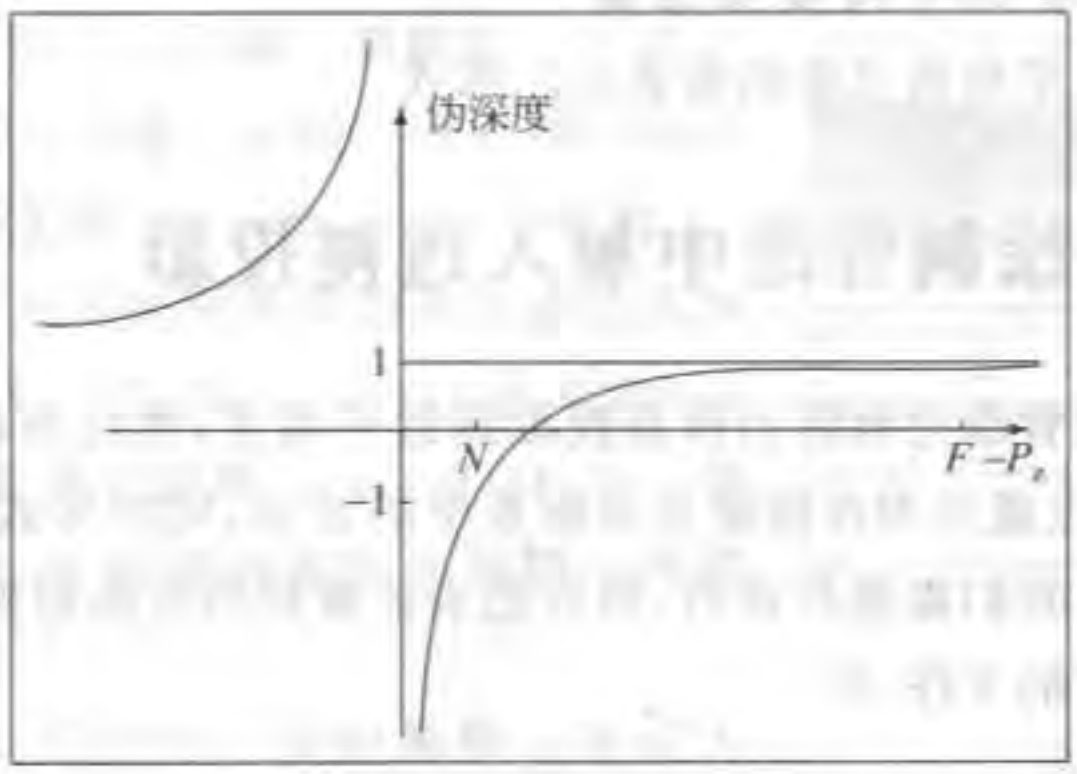


图 7.27 伪深度随着 Pz 增加

另外我们也注意到当$(-P_z)$越来越接近 F 的时候,伪深度变化的速度却越来越慢。由于计算机做的是有限精度的代数运算,所以当进行隐藏面消除的时候,要区分出两个点的伪深度就会产生问题:尽管两点距离视点的实际深度并不相同,但是它们的伪深度值却相同。

留意到按照等式(7.7)中定义,当 P_z 越往负方向变化那么伪深度就越往正方向变化。

这看上去是十分合理的，因为当 P_z 沿负方向越走越远时，距离视点的深度越来越大。

例 7.4.4　当($-P_z$)越来越接近 F 时，伪深度变化得越来越慢

假设 $N=1$，并且 $F=100$。这样就有 $a=-101/99$ 和 $b=-200/99$，由此可以计算出伪深度：

$$\text{pseudodepth}\Big|_{N=1,F=100}=\frac{101P_z+200}{99P_z}$$

这样 $P_z=-N$ 时伪深度等于 -1，$P_z=-F$ 时伪深度等于 1。但是当 P_z 越接近 $-F$ 时，伪深度变化的越来越慢。比如 P_z 分别取 97、98 和 99，那么可以计算出伪深度取值相应为 1.041028、1.040816 和 1.040608。

通过少量的代数运算(参见练习)就可以看出当 N 远小于 F 时，伪深度有如下近似等式：

$$\text{psedodepth}\approx 1+\frac{2N}{P_z} \tag{7.9}$$

再一次可以看出当($-P_z$)接近 F 的时候，伪深度变化得越来越慢。但是当 N 取较大的值时，这种变化速度是可以加快的。所以 N 应该尽可能地取较大的值(当然取值不能任意的大，以至于把距离照相机最近的物体都裁剪掉了)。

使用齐次坐标

在等式(7.7)中我们为什么要在每一项中添加同一个分母？下面我们将要讲到，这可以使得图形绘制管道中截至目前的所有操作步骤都使用矩阵乘法来完成，处理起来更加一致和高效。(一些图形卡上的芯片可以使用硬件来完成点与矩阵相乘的操作，运算速度相当快!)这样做还为高效可靠的裁剪步骤提供了条件。

这个新方法要求我们使用齐次坐标来表示点。因为在齐次坐标下使用 modelview 矩阵对顶点实行变换很简单，事实上我们以前已经这样做过。但是我们不会仅仅局限于此，而是要进一步探讨齐次坐标表示的概念，以发掘它新的能力。特别地，矩阵不仅可以表示仿射变换，还可以表示**透视变换**。

截止到目前，我们所说的点 $P=(P_x,P_y,P_z)$ 的齐次坐标形式为 $(P_x,P_y,P_z,1)$，而向量 $\mathbf{v}=(v_x,v_y,v_z)$ 的齐次形式是 $(v_x,v_y,v_z,0)$。也就是在普通坐标形式后面添上 1 或 0 就可以得到相应的齐次坐标形式。这样我们就有可能使用坐标系作为一组基来表示感兴趣的点和向量，并且能够用矩阵来表示仿射变换。

现在我们把上述思想进行扩展，定义点 $P=(P_x,P_y,P_z)$ 有一族齐次表示 (wP_x,wP_y,wP_z,w) 这里 w 可以取不为 0 的任意值。例如点(1,2,3)可以表示为(1,2,3,1)、(2,4,6,2)、(0.003,0.006,0.009,0.001)、(−1,−2,−3,−1)等等。如果某人给你一个齐次形式表示的点，比如(3,6,2,3)，并且要你说出这个点是什么，这样只需要用它除以其最后一个分量得到 $\left(1,2,\frac{2}{3},1\right)$，然后再把最后一个分量抹去：点的普通坐标形式为 $\left(1,2,\frac{2}{3}\right)$。这就是说：

- 要想把点的普通坐标转化为齐次坐标，在其后面添上 1。
- 要想把点的齐次坐标转化为普通坐标，需要用最后一个分量去除所有的分量，然后

再把第四个分量抹去。

把一个点的所有分量同等程度地放缩后依然表示的是这个点,这个额外的性质正是"齐次"名字的由来。到目前为止,我们处理的都是最后的分量等于 1 的情形。

在练习中我们会继续研究齐次坐标,但是现在我们把精力集中于在点变换过程中齐次坐标的变化规律。请回忆仿射变换矩阵的第四个行向量总是等于(0,0,0,1)。因此我们用这样的矩阵 M 乘上点 P 的齐次坐标形式,有 $MP=Q$(回顾等式(5.24)),正如下面的例子所示

$$\begin{bmatrix} 2 & -1 & 3 & 1 \\ 6 & 0.5 & 1 & 4 \\ 0 & 4 & 2 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{bmatrix} = \begin{bmatrix} wQ_x \\ wQ_y \\ wQ_z \\ w \end{bmatrix}$$

Q 的最后一个分量总不会改变:依然等于 w。因此我们可以用通常的方法把 Q 变回到普通坐标形式。

但是如果我们把第四个行向量(0, 0, 0, 1)进行变动的话,那么就会产生一些新情况。考虑一个重要的例子,它的第四个行向量是(0, 0, −1, 0),(这与我们将要讲到的"投影矩阵"很相似):

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{(投影矩阵 —— 版本 1)} \tag{7.10}$$

其中的 N、a、b 为任意值。用它乘上一个齐次坐标表示的点,其中 w 是任意值:

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{bmatrix} = \begin{bmatrix} wNP_x \\ wNP_y \\ w(aP_z+b) \\ -wP_z \end{bmatrix}$$

它对应了哪一个普通坐标下的点?先除以第四个分量,然后再将它抹去,可得

$$\left(N\frac{P_x}{-P_z},\ N\frac{P_y}{-P_z},\ \frac{aP_z+b}{-P_z}\right)$$

这就是我们使用等式(7.7)得到的结果。因此使用了齐次坐标后矩阵乘法也可以表示透视操作。当然我们还必须把结果除以第四个分量,这一个步骤也称为**透视除法**。

第四个行向量不是(0,0,0,1)的矩阵执行的并不是仿射变换。它代表的是范围更广的一类变换,称为**透视变换**。它属于变换操作而非投影。投影会降低点的维数,把它变为一个 3 元组或 2 元组,而透视变换把 4 元组依然变成 4 元组。

请思考一下,把非零值比如(A,B,C,D)添加到等式(7.10)中矩阵的第四行,会在代数运算方面产生什么效果。当你用矩阵乘以(P_x,P_y,P_z,1)(或它的倍数),结果中的第四项为 $AP_x+BP_y+CP_z+D$,它是与 P 的每一分量线性相关的。当做完透视除法后,这一项会出现在点的分母中。这样的分母也正是我们想要的,它在几何上产生的效果就是往一般平面上的透视投影,我们在练习中会证明这一点。

因此透视变换把一个 3D 点 P 变为另一个 3D 点 P',相应的等式为

$$(P_x, P_y, P_z) \rightarrow \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z+b}{-P_z}\right) \quad \text{“透视变换”} \tag{7.11}$$

那么投影这一部分操作在哪里执行呢？在图形绘制管道中这个点的前两个分量被用于绘制：计算点在屏幕坐标系下的绘制位置。被剥离开的第三个分量用于深度测试。如果要计算点在屏幕上的位置，就要忽略掉第三个分量，这可以通过把它置为 0 实现：

$$\left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z+b}{-P_z}\right) \rightarrow \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, 0\right) \quad \text{“投影”} \tag{7.12}$$

这也正是我们在第 5 章中曾经做过的，那里我们放置好照相机进行 3D 场景绘制的时候要对点实施正交投影（即与视平面垂直）。后面我们还会详细研究正交投影。现在我们可以总结出：

（透视投影）=（透视变换）+（正交投影）

这种把透视投影分解为一个特殊变换外加一个（平凡）投影的做法被证明是十分有用的，这不仅体现在算法方面，而且还有助于理解每个点在通过图形绘制管道时实际上都经历了哪些操作。OpenGL 把变换操作的步骤和投影操作的步骤分离开来。事实上，它还会在这两个步骤中插入裁剪、透视除法，以及一个附加映射的操作。

最后还要强调一点。当我们打算显示一个网格模型的时候，必须把几千个甚至几百万个顶点发送到图形绘制管道。显然，如果我们只让每个顶点进行一次矩阵乘法操作而非一系列的多次矩阵乘法操作，那么效率就会大大的提升。所以把一个顶点所要经历的所有矩阵乘法合成为一个单一的矩阵会显著地提升效率。这也就是 OpenGL 所做的：它一次把所有要用到的矩阵乘起来合成为一个单一矩阵，然后再用这个合成矩阵去乘以每个顶点。

下面我们会更深一步地了解上述过程中变换操作的部分。

透视变换依据等式(7.11)把 3D 点 P 变成另一个 3D 点，为投影做好准备。把它理解为 3D 空间的变形并且了解一个形状如何变形到另一个形状是十分有帮助的。更重要地，它具有保直线性和保平面性，也就是直线被变换到直线，平面被变换到平面，多边形面片被变换到另一个多边形面片。透视变换还具有保持包含关系的性质，因此如果点 a 在一个物体内，那么变换后的点也在变换后的那个物体内（为了保持这些性质，我们要选择好一个适当的伪深度函数）。在练习中会证明上述性质。

特别地，我们想要看一看照相机的视景体是如何被变换的，因为如果我们打算在变形的空间内进行裁剪，那么就要使用变形的视景体。透视变换在这一点表现得十分出众：变形后的视景体是可以进行简单高效裁剪操作的理想形体！图 7.28 说明了视景体和其他的一些形体是如何被变换的。位于 $z=-N$ 处的近平面被映射到 $z=-1$ 处的平面 W'，并且远平面被映射到 $z=+1$ 的平面。把顶面 T 倾斜，将其映射到水平面 T'，使得它与 z 坐标轴平行。底面 S 变为了水平面 S'，两个侧面也都平行于 z 坐标轴。照相机的视景体被变换为一个平行六面体！

透视变换的几何性质

如果平面中的直线或者平行于近平面或者通过视点，那么这些平面的映射关系就很容易证明。请仔细考虑下面的命题：

命题：通过视点的直线被映射为平行于 z 坐标轴的直线。

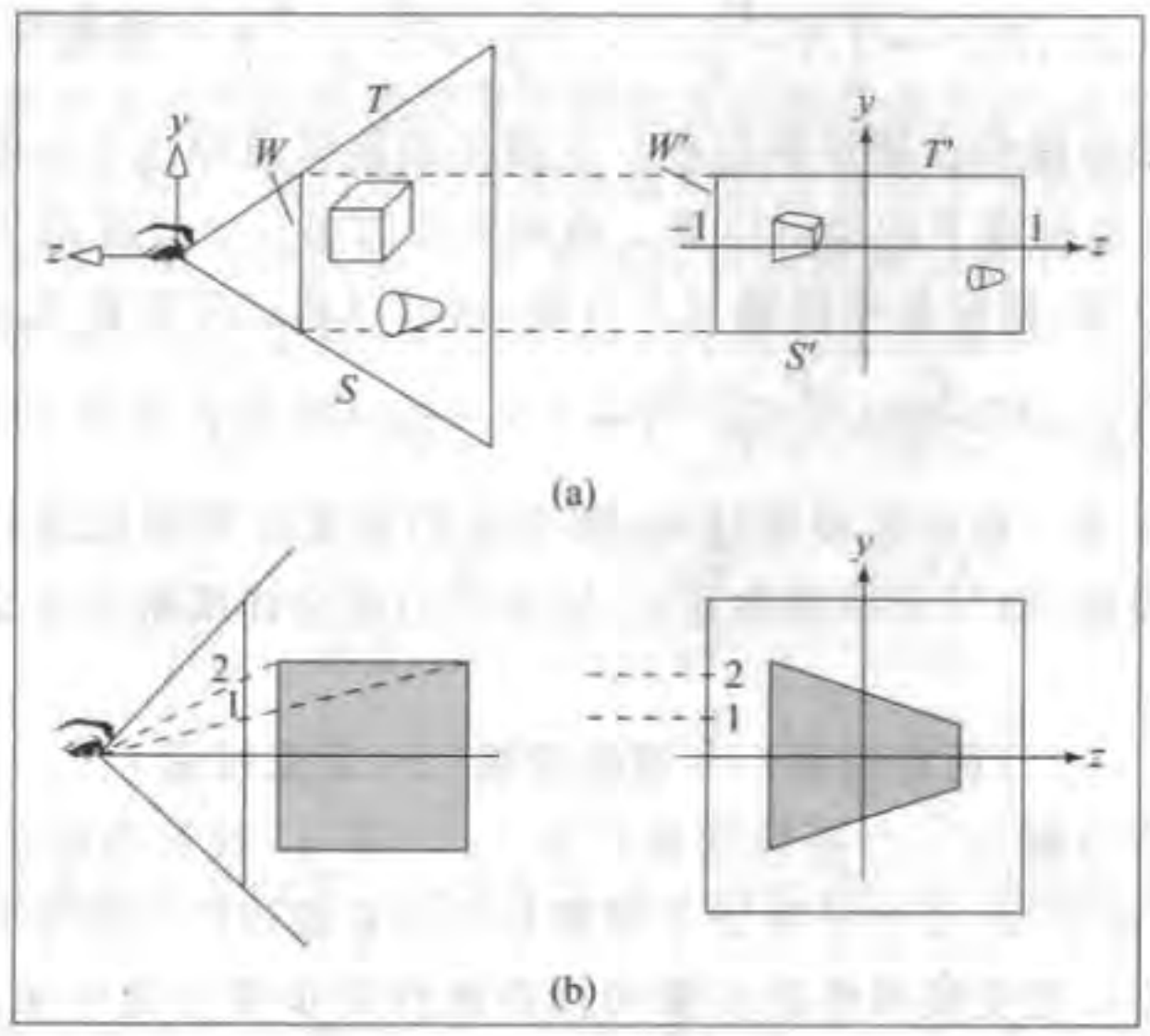


图 7.28 使用透视变换对视景体进行变形

证明：这种直线上的所有点都被投影为视平面上的同一个点，不妨设为(x^*, y^*)。因此直线上的点被变换到点(x, y, z)，其中 $x=x^*$，$y=y^*$，并且 z 取遍 -1 和 1 之间的所有伪深度值。

命题：与 z 坐标轴垂直的直线映射后依然垂直于 z 坐标轴

证明：这种直线上的所有点都具有相同的 z 坐标值，所以它们映射后的点也具有相同的伪深度值。

有了以上事实，就很容易得出变形后的视景体的精确形状和大小。

透视变换也把物体，比如说图中的积木块，变形为其他的形状。图 7.28(b)中一个积木块被投影到了近平面。假设其正面的顶边被投影到 $y=2$，并且背面的顶边被投影到 $y=1$。当对这个积木块实施变换后，它变成一个截断锥体：它正面的顶边位于到 $y=2$，背面的顶边位于 $y=1$。越接近近平面变得越大，越远离近平面变得越小。根据投影规律，变换后的物体它的背面要比正面更小一些。变换后物体的 x 和 y 坐标就是原始物体被投影后的 x 和 y 坐标。当你对变换后物体实行正交投影时会用到这些坐标。简单地说：

物体经透视变换变形后再用正交投影进行绘制，其结果等同于原始物体直接进行透视投影。

因此所有的物体都遵循透视投影的规则被适当地变形为依透视规律缩小的形体。然后再用正交投影进行绘制就可以得到正确的结果。

下面我们深入地研究一下视景体变换后的具体形状和大小。

深入研究变换视景体与正则视景体映射

我们打算给变形前后的视景体分别标出尺寸数值。考虑顶面，如图 7.29 所示它在 $z=-N$ 处通过点$(left, top, -N)$。因为组成顶面的直线通过视点以及近平面中 y 坐标值为 top 的点，所以它必须被变换为平面 $y=top$。同样，

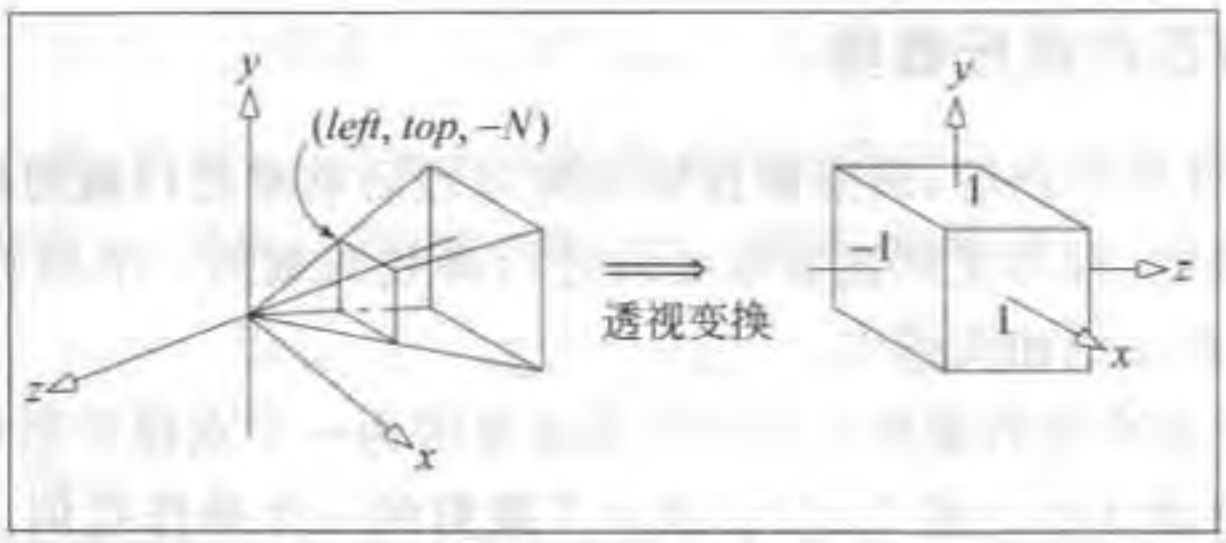


图 7.29 透视变换的细节

- 底面被变换为平面 $y=bott$。
- 左侧面被变换为平面 $x=left$。
- 右侧面被变换为平面 $x=right$。

现在我们对变换后的视景体有了更准确的认识：它是一个平行六面体，它的尺寸大小与照相机的性质有关。我们将会看到，这是一个用于裁剪的绝佳形体，因为它的各个面与坐标面平行。但是如果它的尺寸不依赖于特定的照相机就更好了。OpenGL 在透视变换中又加入了伸缩和平移操作使得上述的平行六面体变成了一个**正则视景体**，它是一个正立方体，每一维的坐标都从 −1 到 1。由于在 x 维和 y 维实行了不同程度的拉伸，把任意场景都变换为一个固定体积的空间，这就势必使形状产生了失真，但是这种失真可以在最终的视口变换后得到消除。

变换后的视景体的 z 维本来就是从 −1 到 1，所以只需要在其他两个维度上进行伸缩操作。因此在 x 和 y 方向加入伸缩和平移操作就可以把平行六面体变换为正则视景体。我们先在 x 方向平移 $-(right+left)/2$，在 y 方向平移 $-(top+bott)/2$。然后在 x 方向乘上尺度因子 $2/(right-left)$，在 y 方向乘上尺度因子 $2/(top-bott)$。当这些矩阵乘法操作完成后，我们就得到了最终的矩阵形式：

$$R=\begin{pmatrix} \frac{2N}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bott} & \frac{top+bott}{top-bott} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad \text{（透视矩阵）} \tag{7.13}$$

这个矩阵被称为**透视矩阵**，它除了完成透视变换之外，还完成了伸缩和平移操作，把照相机的视景体变换为正则视景体。当使用函数 glFrustum(left, right, bott, top, N, F)的时候，OpenGL 生成的(要与当前矩阵进行相乘)正是这个矩阵。回想我们以前倒是经常使用函数 gluPerspective(viewAngle, aspect, N, F)，因为它的参数更加直观一些。它建立的也是同一个矩阵，但是事先要用如下等式计算出 top、$bott$ 等变量的值

$$top = N\tan\left(\frac{\pi}{180}viewAngle/2\right)$$

$$bott=-top,\quad right=top*aspect,\quad \text{以及 } left=-right$$

使用视景体对面片进行裁剪

回想一下图 7.14 中的内容,顶点被投影矩阵处理后就要进行裁剪操作。这种操作是在变形后的空间内完成的,因为正则视景体可以进行高效的裁剪。下面我们开始研究它的机制,并且深入到裁剪算法的相关细节。

在已被变形的空间中进行裁剪之所以行得通是因为一个点位于照相机的视景体中当且仅当它也位于正则视景体中。图 7.30(a)展示了裁剪的一个操作范例。三角形的三个顶点是 v_1、v_2 和 v_3。顶点 v_3 位于正则视景体 CVV 之外。裁剪操作作用在边上:它首先对边 v_1v_2 进行裁剪但是此边全体都位于 CVV 内部。然后它裁剪边 v_2v_3 并且纪录下新的顶点 a,这是边离开 CVV 的位置。最后它裁剪边 v_3v_1 并且记录下此边进入到 CVV 的位置。经过上述过程原始的三角形就变成了顶点为 v_1v_2ab 的四边形(后面我们还会看到,除了计算新顶点的位置,图形绘制管道还会计算这些新顶点的颜色和纹理参数)。

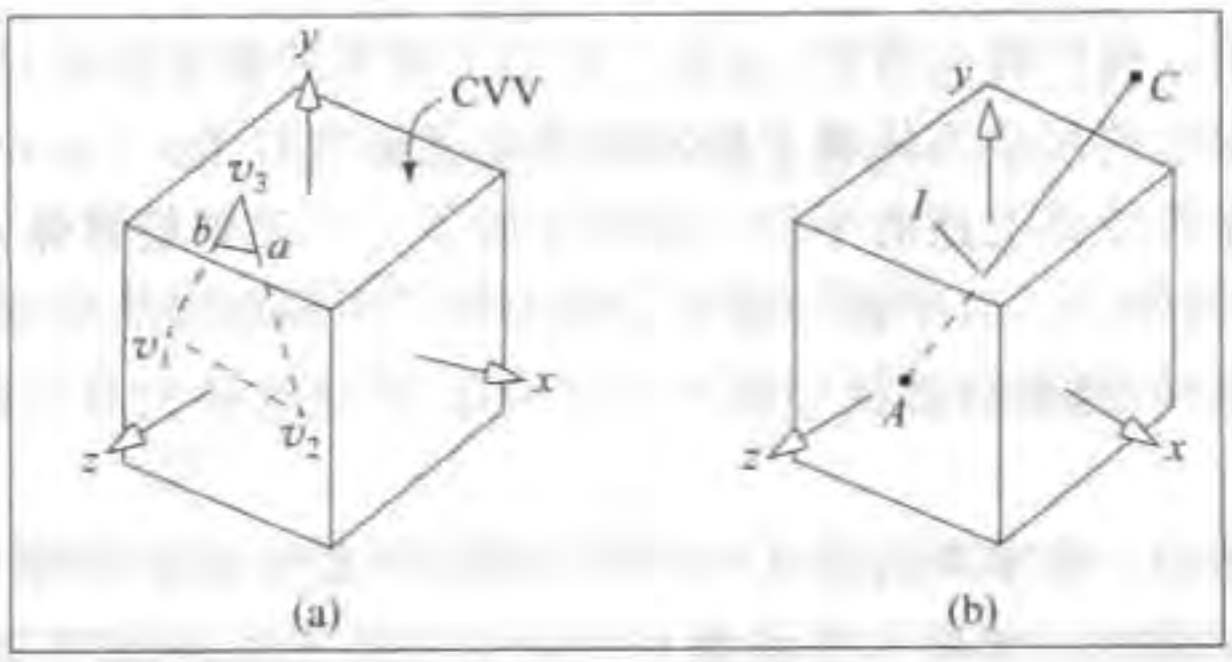


图 7.30 用正则视景体进行裁剪

裁剪的问题基本就是使用 CVV 对线段进行裁剪的问题。在 4.8.3 小节中我们已经讨论过这样的一个算法,Cyrus-Beck 裁剪算法。我们这里要开发的裁剪算法与它很相似,但是当然用于 3D 而不是 2D 的环境。

实际上,它是处理的 4D 情形。我们的裁剪操作作用在 4D 齐次坐标空间,称之为"裁剪坐标",如图 7.14 所示。这比看上去的还要简单,并且可以很好地区分出一个点是在视点之前还是在视点之后。

在图 7.30(b)中假设我们打算用 CVV 裁剪线段 AC。这就是说给出齐次坐标表示的两个点 $A=(a_x,a_y,a_z,a_w)$和 $C=(c_x,c_y,c_z,c_w)$,我们要判断出这条线段的哪一部分位于 CVV 内。如果线段和 CVV 的边界相交,我们需要计算出交点 $I=(I_x,I_y,I_z,I_w)$。

像 Cyrus-Beck 算法一样,我们把 CVV 看成是六张无限延伸的平面,然后逐个地对每个平面考虑给定的边与它形成的位置关系。我们把边表示为参数形式 $A+(C-A)t$。A 点的参数为 0,C 点的参数为 1。针对 CVV 的每个面,我们首先测试 A 和 C 是否位于该面的同一侧:如果是的话就没有必要计算边与面的交点了。如果两点分属不同的两侧,那么我们计算出交点的位置然后裁剪掉位于平面外部的那一部分。

所以我们必须能够判断出一个点是在一张平面的内部还是外部。例如,考虑平面 $x=-1$。如果对于点 A 有

$$\frac{a_x}{a_w} > -1 \quad \text{或者} \quad a_x > -a_w \quad \text{或者} \quad (a_w + a_x) > 0 \tag{7.14}$$

则点 A 位于平面的右侧(内部)。当不等式的两端同时乘上一个负项时,不等号的方向必须改变。但是我们处理的 a_w 只取正值(参见练习)。同样,如果有

$$\frac{a_x}{a_w} < 1 \quad \text{或者} \quad (a_w - a_x) > 0$$

则 A 位于 $x=1$ 的内部。

Blinn【Blinn96】把这些值称为点 A 的"边界坐标",并且他列出了我们要用到的六个这样的值,如图 7.31 所示。

边界坐标	齐次坐标	裁剪面
BC_0	$w+x$	$x=-1$
BC_1	$w-x$	$x=1$
BC_2	$w+y$	$y=-1$
BC_3	$w-y$	$y=-1$
BC_4	$w+z$	$z=-1$
BC_5	$w-z$	$z=1$

图 7.31 边端点的边界坐标

我们依次为点 A 和点 C 计算上述的六个值。如果一个点的所有六个值都为正,那么该点就位于 CVV 内部。如果都为负,则该点位于外部。如果两个端点都位于内部,我们称这种情况为"平凡的接受",在 3.3 节中的 Cohen-Sutherland 算法中处理过相同的情况。如果点 A 和点 C 都位于同一个平面的外侧(相应的边界坐标 BC 为负),那么这条边必定全部位于 CVV 之外。

平凡的接受:两个端点都位于 CVV 之内(所有 12 个 BC 都为正)。

平凡的拒绝:两个端点都位于 CVV 的同一个面的外部。

如果不是上述的两种情况,那么我们就必须单独使用每一张平面对线段 AC 进行裁剪。和 *Cyrus-beck* 算法一样,我们需要记录下**候选区间**(CI)(参见图 4.45),它是由 t 值组成的区间,边只有在这一部分取值才可能位于 CVV 内部。我们知道它的相反情形:如果 t 在 CI 外取值,那么可以肯定它不在 CVV 内部。CI 的取值范围从 $t=t_{in}$ 到 t_{out}。

我们使用每一张面轮流地对边进行测试。如果相应的边界坐标符号相反,那么我们可知边与平面相交于某一 t_{hit},然后需要把它计算出来。若边是进入方向的(当 t 值增加时由平面外部进入到平面内部)那么我们更新 $t_{in} = \max(\text{old}t_{in}, t_{hit})$,这样做是因为进入到 CVV 时的 t 不可能早于 t_{hit}。同样的,如果边是退出方向的,我们更新 $t_{out} = \min(\text{old}t_{out}, t_{hit})$。若在某一时刻 CI 变成了空区间($t_{out} > t_{in}$),那么我们就可以把整个边都裁剪下去然后"提前退出"算法程序,节省不必要的计算。

很容易计算边与平面相交时的 t 值。把边用参数表示为齐次坐标形式:

$$\text{edge}(t) = (a_x + (c_x - a_x)t, a_y + (c_y - a_y)t, a_z + (c_z - a_z)t, a_w + (c_w - a_w)t)$$

例如计算边与平面 $X=1$ 的交点,那么当 $A+(C-A)t$ 的 x 坐标为 1 时有:

$$\frac{a_x + (c_x - a_x)t}{a_w + (c_w - a_w)t} = 1$$

从中可以很容易解出 t,

$$t = \frac{a_w - a_x}{(a_w - a_x) - (c_w - c_x)} \tag{7.15}$$

留意到 t_{hit} 只取决于两个边界坐标。与其他平面的相交也有类似的等式。

很容易把上面所说的内容写成程序,参见图 7.32 所示的伪代码。它基本上就是 Liang-Barsky 算法【Liang84】,然后再用 Blinn【Blinn96】的一些建议进行了细化。程序 clipEdge(Point4 A, Point4 C)需要的参数是两个齐次坐标表示的点(分量是 x、y、z 和 w),当 AC 与 CVV 无交点时结果返回 0,其他情况返回 1。它同时也对 A 和 C 进行修改,这样当程序结束时,A 和 C 就是裁剪后的边的端点。

```
int clipEdge(Point4& A, Point4& C)
{
    double tIn=0.0, tOut=1.0, tHit;
    double aBC[6], cBC[6];
    int aOutcode=0, cOutcode=0;
    <..计算 A 和 C 边界坐标 BC..>
    <..生成 A 和 C 的 outcode..>

    if((aOutcode & cOutcode) !=0)                    //平凡的拒绝
            return 0;
    if((aOutcode | cOutcode)==0)                     //平凡的接受
            return 1;

    for(int i=0; i <6; i++)                          //每个面都进行裁剪
    {
        if(cBC[i=<0)                                 //退出方向:C 在外部
        {
            tHit=aBC[i]/(aBC[i]-cBC[i]);
            tOut=MIN(tOut,tHit);
        }
        else if(aBC[i] <0)                           //进入方向:A 在外部
        {
            tHit=aBC[i]/(aBC[i]-cBC[i]);
            tIn=MAX(tIn, tHit);
        }
        if(tIn >  tOut) return 0;                    //CI 为空,提前退出
    }
    // update the end points as necessary
    Point4 tmp;
    if(aOutcode !=0)                                 //A 在外部:tIn 被修改过
    {//计算 A 新的值,(但暂时不修改它)
        tmp.x=A.x+tIn * (C.x-A.x);
```

图 7.32 边的裁剪程序

```
        tmp.y=A.y+tIn * (C.y-A.y);
        tmp.z=A.z+tIn * (C.z-A.z);
        tmp.w=A.w+tIn * (C.w-A.w);
    }
    if(cOutcode !=0)                          //C 在外部: tOut 被修改过
    {    //更新 C(使用 A 的原始值)
        C.x=A.x+tOut * (C.x-A.x);
        C.y=A.y+tOut * (C.y-A.y);
        C.z=A.z+tOut * (C.z-A.z);
        C.w=A.w+tOut * (C.w-A.w);
    }
    A=tmp;                                    //现在更新 A
    return 1;                                 //边的一部分在 CVV 中
}
```

图 7.32（续）

程序计算每个端点的六个边界坐标并且把它们存储在 aBC[]和 cBC[]中。为了效率起见，它还为每个点构造了一个 outcode 变量用于保存此点的六个边界坐标的符号。如果 aBC[i]>0（A 位于第 i 个面内部）则 A 的 outcode 的第 i 位置 0 否则置 1。有了这些，当 aOutCode 和 cOutCode 都为 0 就是“平凡的接收”。当两个 outcode 逐位进行“与”操作结果非 0 时就是“平凡的拒绝”。

在循环语句中轮流使用每个面对边进行测试，BC 中最多只能有一个为负（为什么?）。如果点 A 有一个负的 BC 值，那么此边在交点处一定是进入方向的；如果点 C 有一个负的 BC 值，那么此边在交点处一定是退出方向的。（Blinn 的方法要稍微快一些，他使用了一个掩码对 outcode 变量的位进行测试。每次 tIn 和 tOut 更新后，如果 tIn 大于了 tOut 那么就提前退出程序。

当测试过所有平面后，tIn 和 tOut 中的一个或者全部会被修改。如果 tIn 被改动了那么把 A 更新为 $A+(C-A)tIn$，如果 tOut 被改动了那么把 C 更新为 $A+(C-A)tOut$。

Blinn 建议在处理之间提前计算好每个点的 BC 值和 outcode 值。这样当一个顶点为多个边的端点时（经常发生这种情况）就不需要再重复计算这些值了。

我们为什么要用正则视景体进行裁剪?

我们现在已经看出使用正则视景体进行裁剪十分方便，所以对物体实施裁剪之前先将其变换到裁剪坐标就有很大的价值。CVV 的两个重要特性包括：

1. 它无须设定参数：算法不需要额外的信息来描述裁剪视景体。它使用的值无非就是−1 和 1。所以程序代码可以进行高度优化获取最大的效率。

2. CVV 的平面与坐标轴是对齐的（在透视变换执行之后）。这就意味着我们只需要使用一个坐标值就可以判断点位于平面的哪一侧，比如 $a_x>-1$。如果平面不是与坐标轴对齐的话，就还需要做开销比较大的点积运算。

我们为什么要使用齐次坐标进行裁剪,而不是在透视除法之后进行裁剪?

这并不是完全必要的,但是这样做可以使裁剪算法更加简洁快速。做完透视除法之后会破坏掉一些信息:如果你知道了 a_x 和 a_w 的值,当然也就知道了它们的符号。但是如果只给出 a_x/a_w 的商,那么你只能了解到 a_x 和 a_w 是否同号以及它们之间的相对比例关系。使用齐次坐标的值并且裁剪掉比近平面离视点更近的点,这样那些位于视点背面的点就自动被排除掉了,例如图 7.23 中的 B。

还存在一些古怪的情形必须要在齐次坐标下进行裁剪,这在【Blinn96, Foley90】中有所描述。它们牵扯到物体的一些特殊变换或某些曲面的构造问题,即使原始点(a_x, a_y, a_z, a_w)位于视点之前,它的第四个分量也取负值。我们在这里讨论的物体建模问题不会遇到这种情况。我们可以得出结论,使用齐次坐标进行裁剪,即使通常并不是必须的,但是这可以使算法更加快速和简单,而且几乎不产生什么开销。

执行完裁剪操作,最后执行透视除法(如图 7.14 所示),然后三元组(x, y, z)就要经历视口变换。我们后面会讨论,这种变换对 x 和 y 值实行尺寸变换以及平移操作,以便它们可以被恰当地放置在视口之中,同时对 z 分量(伪深度)也做一些小的调整使之更加有利于深度测试。

视口变换

我们已经看到,透视变换把场景都变换到正则立方体中,如图 7.33。如果照相机视景体的宽高比(也就是近平面上的窗口的宽高比)等于 1.5,那么透视变换把物体放到宽高比为 1 的窗口后就会产生明显的失真。但是视口变换通过把正方形映射到宽高比为 1.5 的视口可以把这种失真消除掉。

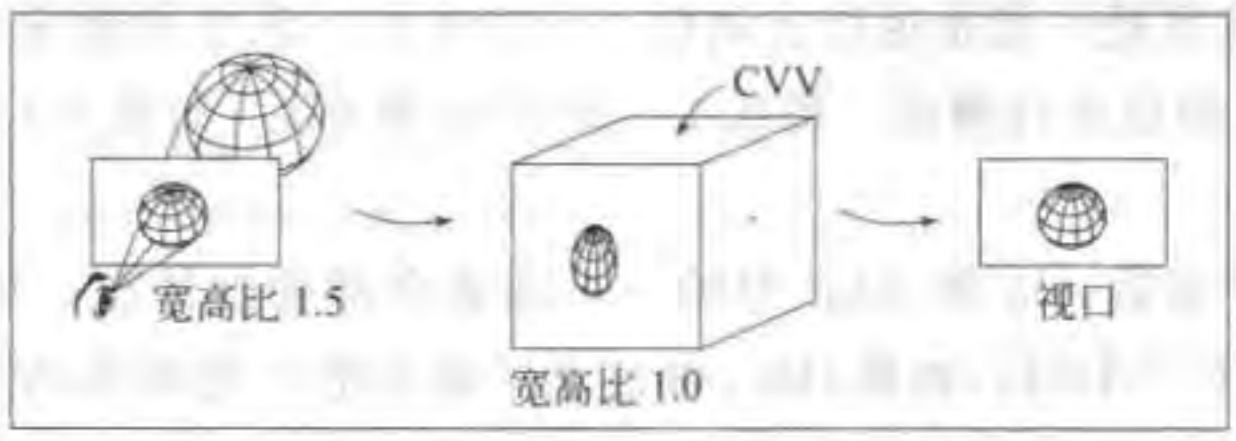


图 7.33 视口变换恢复了原始的宽高比

我们以前经常使用 OpenGL 函数 glViewport(x, y, wid, ht)。它指定视口左下角的屏幕坐标为(x, y),视口的宽等于 wid 个像素,视口的高等于 ht 个像素。因此它指定的视口宽高比等于 wid/ht。视口变换同时也把伪深度由区间[−1, 1]变换到区间[0, 1]。

重新温习图 7.14,它揭示了完整的图形绘制管道。在 OpenGL 中,每个点 P(通常是多边形的一个顶点)要经历如下步骤:

- 通过给 P 添加分量 1 将其**扩展**为齐次 4 元组;
- 用这个 4 元组乘上**矩阵 modelview**,得到视点坐标系下的 4 元组;
- 再将点乘上**投影矩阵**,得到裁剪坐标下的 4 元组;
- 对以此点为端点的边实施**裁剪操作**;

- 进行隐藏面消除操作(参见 8.4 节)；
- 执行**透视除法**，得到一个 3 元组；
- 在视口变换过程中 3 元组被乘上一个矩阵：得到的结果(sx,sy,dz)被用于绘制和深度计算。(sx,sy)是点在屏幕上的显示位置；dz 是用某种尺度表示的原始点距离照相机视点的深度。

练习

练习 7.4.5　点 P 的投影在何处？

假设视平面在照相机坐标系中的方程为 $Ax+By+Cz=D$。请证明任意一点 $P(P_x,P_y,P_z)$在平面上的投影可用齐次坐标表示为

$$P'=(DP_x,DP_y,DP_z,AP_x+BP_y+CP_z)$$

练习 7.4.6　伪深度的近似形式

证明当 N 远小于 F 时，等式(7.9)可以很好的逼近伪深度 $a+b/(-P_z)$，这里的 a 和 b 分别由等式(7.8)给出。

练习 7.4.7　无穷远处的点的齐次坐标

当 w 变得越来越小时，请考虑齐次坐标表示的点(x,y,z,w)的性质。比如当 $w=0.01$ 时，它为$(100x,100y,100z,1)$；当 $w=0.0001$ 时，它为$(10000x,10000y,10000z,1)$，以此类推。点会沿着方向(x,y,z)"趋向于无穷"。事实上，用$(x,y,z,0)$表示的点被称为"无穷远点"。齐次坐标的优点之一就是这种理想点也可以拥有一个完美的有限表达形式：在某些数学推导过程中，这可以消除许多棘手的特例情形。例如，两条直线总是会相交，即使它们相互平行【Ayers67，Semple52】。但是针对一些其他情况，齐次坐标就并不很胜任。两个齐次坐标表示的点的差是什么？

练习 7.4.8　透视变换如何影响直线和平面？

我们必须要证明透视变换具有保平面性和保包含性。

a. 请说明为什么证明了如下情形上述命题就可以成立：直线上一点 P 位于 A、B 两点之间，那么变换后的 P'位于变换后的 A 和 B 之间。

b. 请证明透视变换生成的点 P'确实具有上述性质。

c. 请证明通过视点的每个平面都被映射为平行于 z 坐标轴的平面。

d. 请证明垂直于 z 坐标轴的平面映射后依然垂直于 z 坐标轴。

e. 请证明透视变换保持深度的相对关系。

练习 7.4.9　变换后的视景体的细节

证明变形后的视景体的尺寸可以通过先前练习中的五个点给出。你可以使用先前练习中证明的结论。

练习 7.4.10　证明投影矩阵的最终形式

投影矩阵基本上就是等式(7.10)的形式，但是还需要加上平移和伸缩变换。如果用 M 表示等式(7.10)中的矩阵，T 表示平移矩阵，S 表示伸缩矩阵，请证明矩阵的乘积 STM 就是等式(7.13)中所给出的。

练习 7.4.11 视点后面的点如何变换?

如果透视变换把视点变到负无穷远处,那么原先位于视点后面的点会如何变换?请考虑这样一条直线 $P(t)$,当 $t=0$ 时它的起点位于视点之前,当 $t=1$ 时直线位于视点之后。

a. 找出它的齐次坐标形式

b. 经历过透视变换之后,写出它的参数形式

c. 用几何的观点对上述情形进行解释。请特别指出第四个分量的几何含义。【Blinn78】中有关于此现象的重要论述。

7.5 生成立体视图

我们先简单地旁述一下如何使用照相机得到一个场景的立体视图。立体视图可以使图像表达得更加明晰;当它被正确地观看时,可以让观察者对图像产生深度感,这样不仅真实有趣,更重要的是可以减少图像的视觉歧义(比如哪些线段是在另一些线段的前方)。本书所有的立体图像都是利用下面的方法生成的。

到目前为止,我们所说的照相机也许更应该称为"独眼"照相机;在希腊神话中 Poseidon 的儿子 Polyphemus 就是一位独眼怪人。如果要想体验一下这种照相机的局限性,你可以闭上一只眼睛在房间里来回走动并且试着做一些简单的工作。人们与生俱来的眼-脑立体感系统可以提供大量的视觉深度信息。我们希望让计算机产生的图形图像也获得此种能力。

要想制作立体视图,一左一右的两幅图像要分别用略有差异的照相机生成,如图 7.34 所示。所有照相机的 lookAt 点都是同一个,但是视点位置各不相同。在图 7.34(b)中,显示器里并排显示了两个视口。左图像在左侧的视口,右图像在右侧的视口。

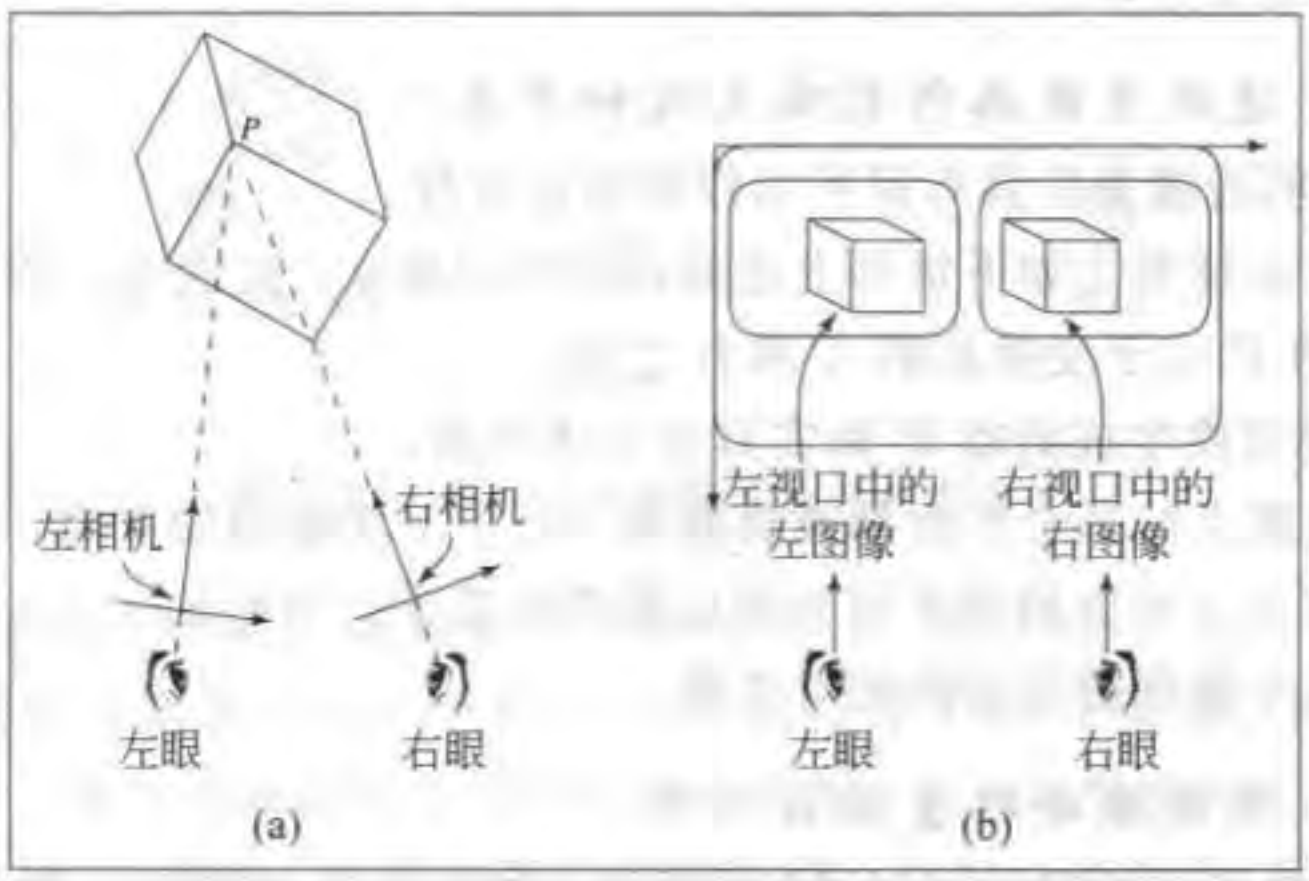


图 7.34 生成立体视图

要想观看立体视图,用你的左眼盯住左图像而右眼盯住右图像。当你方法得当的话,两幅图像就会融为一体并且产生深度信息。这也许需要做一定的练习(前言中描述了一种练习的方法)。

图 7.35 展示了第 6 章提及的 Bucky 球的立体线框视图。两幅图像有很明显的不同,并

且如果只观察其中的某一幅图像就会产生明显的视觉歧义(哪些边在前,哪些边在后?)。但是立体视图就可以消除了这些边的歧义,使得图像表达得简单明了。

图 7.36 是仓房的立体视图。图 7.36(a)中的照相机被转动了 40°。注意到如果没有了立体效果,就很难判别出仓房的朝向。图 7.36(b)是仓房一个角落的特写,可以明显地看出其严重的透视失真。

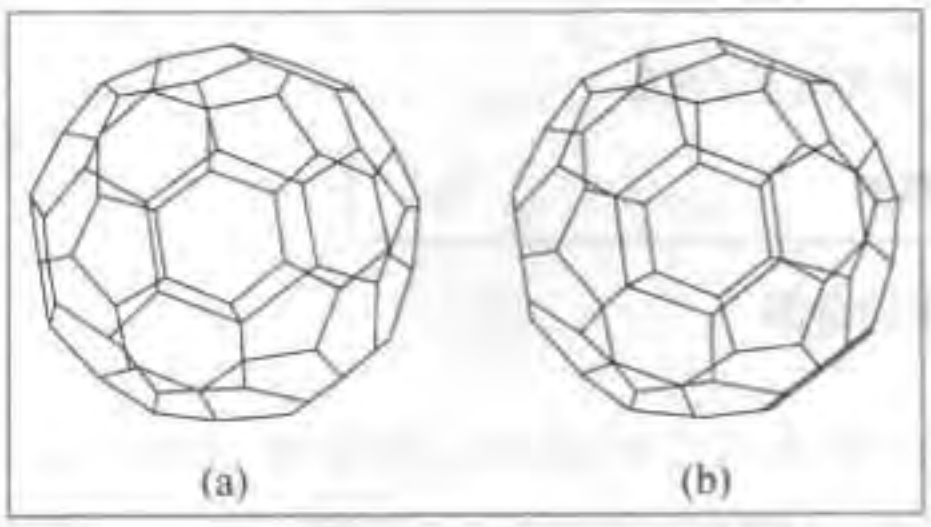


图 7.35 Bucky 球的立体视图

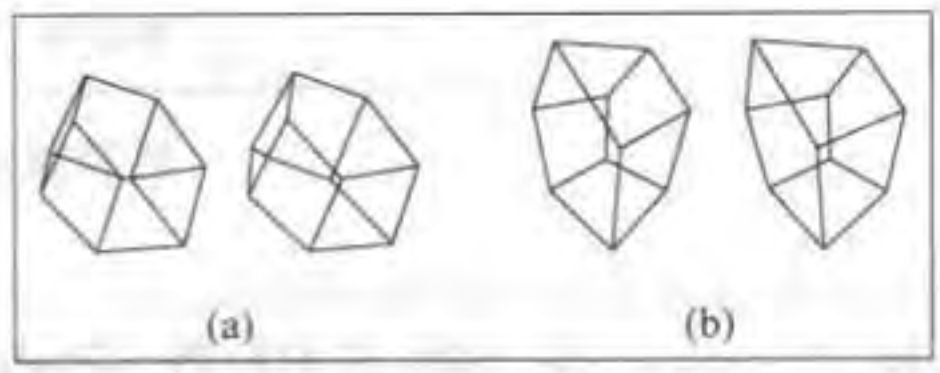


图 7.36 仓房视图特写

为了放置好两台照相机,我们必须要决定左右视点的位置。一个简单的方法是先基于单个 LookAt 点和一个初始视点放置好一台照相机,如图 7.37 所示。选择好 **up** 向量,这就构造好一台"独眼"照相机,它具有三个方向**u**、**v**、**n**。

在初始视点处,沿着 $-\mathbf{u}$ 和 **u** 方向分别偏移距离 D 放置左右视点。D 的选择取决于具体应用中的度量单位。如果所有的长度和距离都用英寸表示,那么用户就应该相应地把照相机放在距离 LookAt 点的若干英寸处。人类两眼大概的距离是 3 英寸,所以 D 最好设为 1.5。如果使用"米"作为度量单位,那么两眼间的距离也应该用"米"来表示。对于一些情形,它的场景比较虚幻怪异,并且不具有内在的尺度,那么为了获得立体效果就要进行一些实验。案例 7.2 是一个生成立体视图的研究项目。

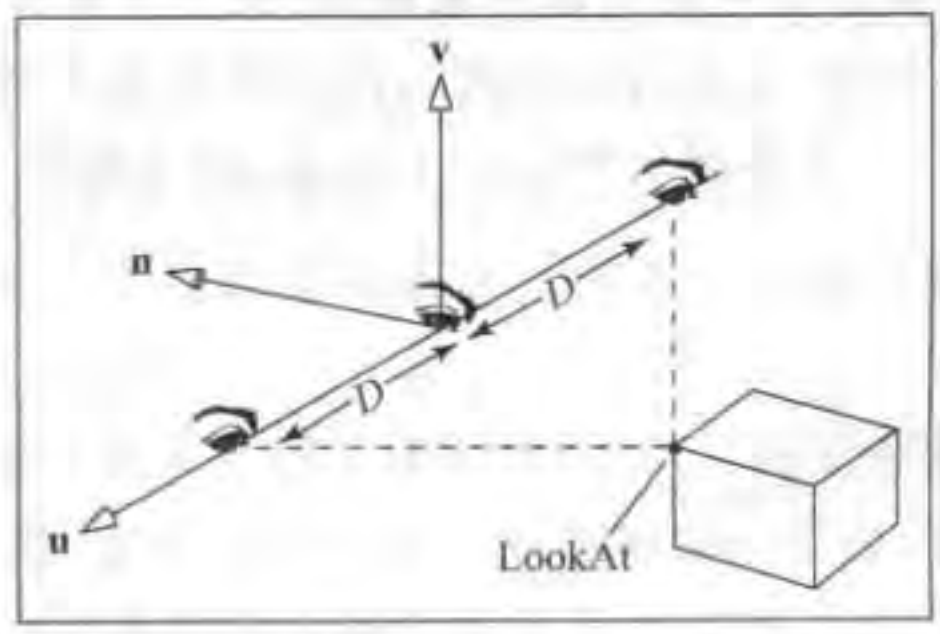


图 7.37 为生成立体视图而放置两个视点

7.6 投影的分类

我们已经探讨过平面投影的一些基本思想,它是把所有的点以某种方式投射到一张平面上。我们在第 5 章看到了平行投影,而在本章又看到了透视投影。在艺术、建筑以及工程绘图领域要涉及到许多投影的特例,下面我们分析一下它们的特点,并且看一看它们之间的关系。

如图 7.38 可以很自然地把平面投影组织成树型结构。在这棵树中,投影类型的每一个子节点都是其父节点的一种特例。在平行投影与透视投影之间产生了第一层重要的分类。我们先来探讨透视投影的各种类型。

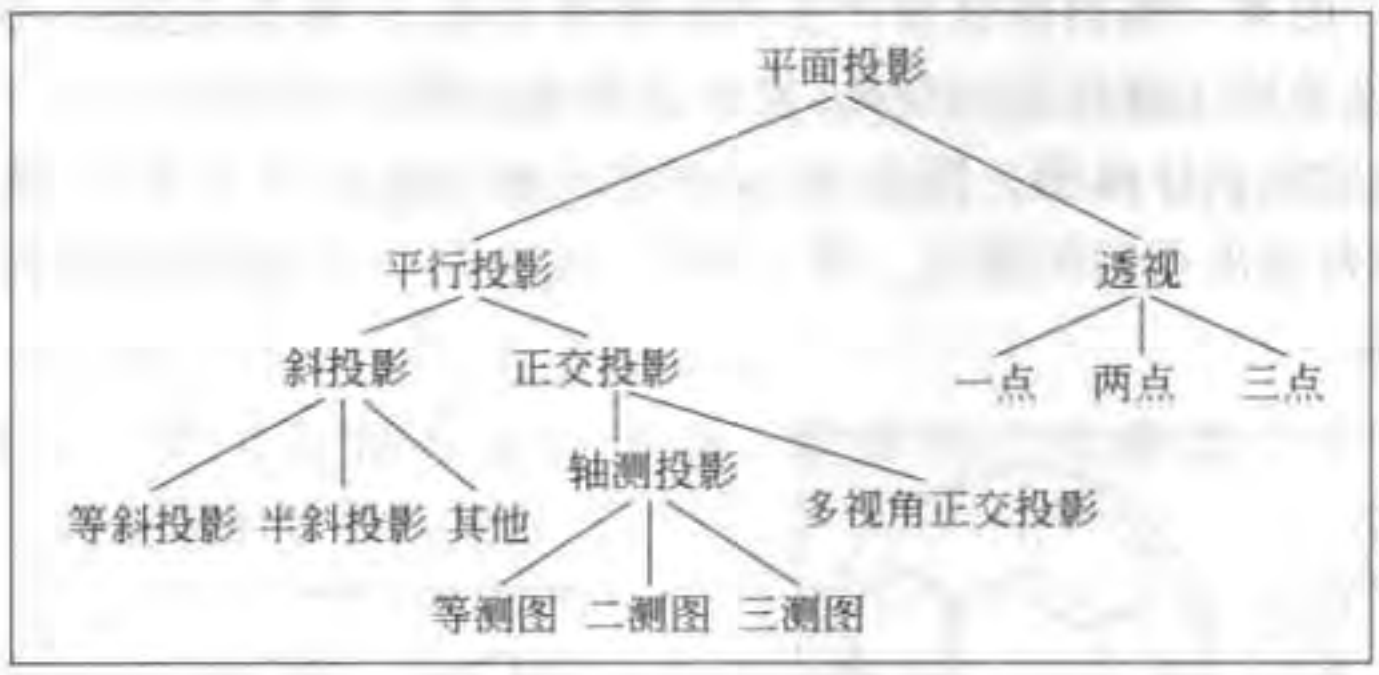


图 7.38 常用投影的分类

7.6.1 一点、两点以及三点透视

透视投影恰好可以分成三类：一点、两点以及三点透视。分类依据的是照相机的朝向与世界坐标系的相对关系。名字来源于观察图 7.39 中单位立方体的情形。这个单位立方体紧靠坐标系的第一象限，并且一个角点位于原点处。更重要的是，它的边与世界坐标系的轴对齐，在本讨论中我们称世界坐标系的轴为**主轴**。主轴的单位方向向量是 **i**、**j** 和 **k**。同样，三个平面 $x=0$、$y=0$ 以及 $z=0$ 被称为**主平面**，立方体的六个面与它们是对齐的。

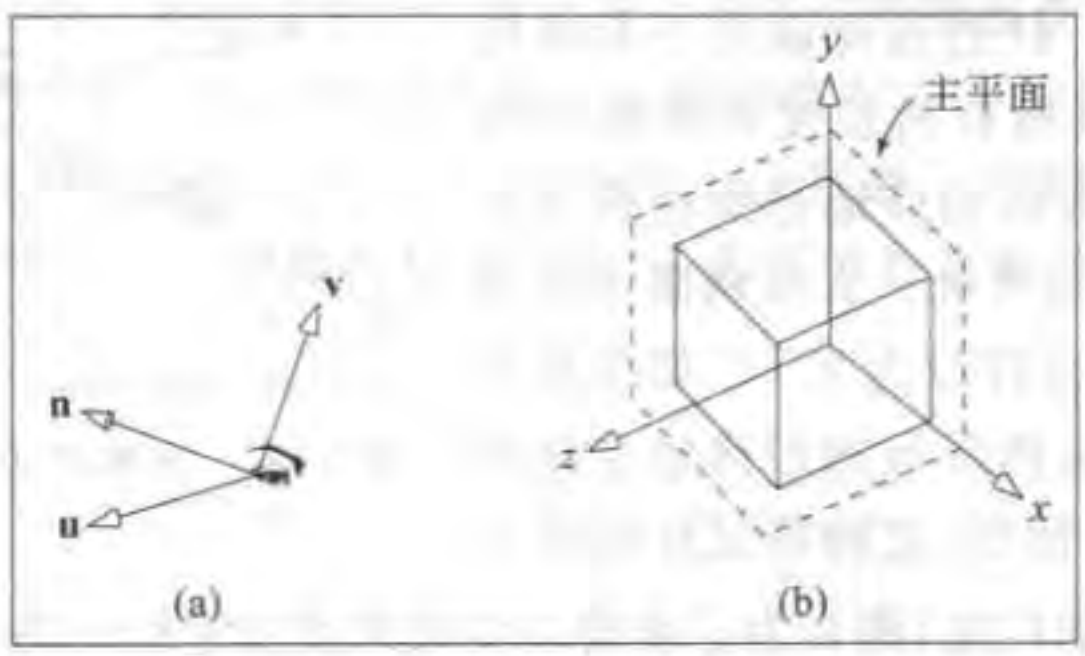


图 7.39 单位立方体、主轴以及主平面

相对于坐标系，照相机可以有无数种朝向方式。对于其中的一些情形，照相机的 $\boldsymbol{n}$ 轴与某个主轴是垂直的。通常，对透视投影的分类可以依据主轴所产生的**有限远处的灭点**数量。回想一下，当某条直线与 **n** 垂直时，其灭点在无穷远处；否则其灭点在有限远处。所以我们同样也可以计算与 **n 不垂直**的主轴数量。这其实也是**穿透**照相机的视平面的主轴数量。(为什么?)

1. **一点透视**：只有一条主轴产生有限远处的灭点。也就是 **n** 只与 **i**、**j**、**k** 三个方向之中的一个不垂直。但由于 **n** 与其他两个方向垂直，所以它垂直于一张主平面。n_x、n_y、n_z 三个分量中的两个必须为 0。

图 7.40(a)是一张一点透视图，其中的照相机视平面与 xy 坐标平面平行。立方体的几条延伸线在有限远处汇聚成一个灭点。这里，照相机的 **n**=(0,0,1)。对于照相机坐标系，延伸线的方向 **c**=(0,0,−1)，所以通过等式(7.6)可得灭点位于(0,0)。另一方面，与 x 轴

和 y 轴平行的直线其灭点位于无穷远处。

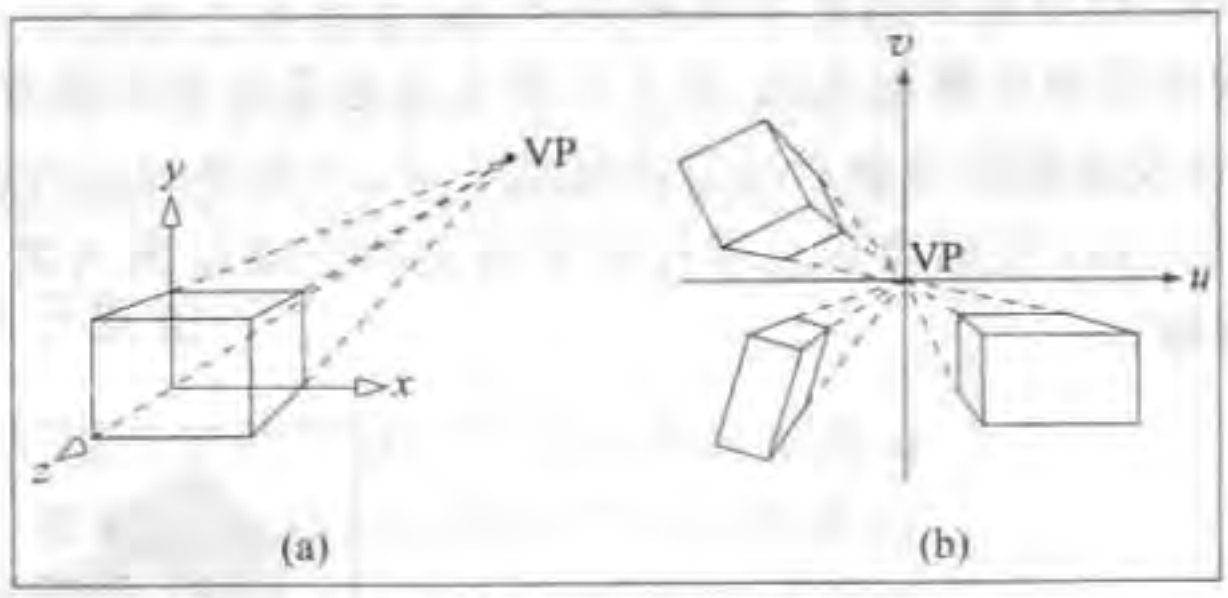


图 7.40 一点透视图

灭点的位置与照相机相对于立方体的位置无关。图 7.40(b)是用一点透视法画出的若干块积木。每一块积木都有其自身的主轴，并且在图中每一块积木的前端面与视平面平行。所有的延伸线汇聚成同一个灭点(0，0)。

重新观察图 7.24 中水平面上的两套网格线。网格线平行于主轴(世界坐标系的坐标轴)。还有另一套垂直的网格线并没有画出来，它与世界坐标系的 y 坐标轴平行。这张图看上去像是一点透视图，因为在水平面上只有一个灭点。但是，如果照相机是向下瞄准的，那么它就是我们将要讨论的两点透视。单从这张图你无法区分到底是哪一种透视(如果你被告知水平面被投影到 $y=0$，那么就可以肯定照相机是水平放置的并且为一点透视)。

2. **两点透视**：恰好两个主轴的灭点在有限远处。因此照相机的 **n** 方向不与这两个轴垂直；它只垂直于另一个轴。**n** 的三个分量中有一个必须为 0。

图 7.41(a)是一个立方体的两点透视图：由于轴 **i** 和 **k** 穿透了视平面，所以有两个有限远处的灭点。照相机按照图 7.41(b)所示的位置放置，它的 **n** 方向与 z 坐标轴成 θ 度角，因此可知 $\mathbf{n}=(\sin(\theta),\cos(\theta))$。这里的 **n** 垂直于 **j**，所以垂直方向的主轴其灭点位于无穷远处。不难计算出有限远处的灭点位置(参见练习)。

如果用两点透视表达图 7.24 中初次见到的网格线场景会是一件很有意思的事情。在图 7.42 中，视点依然位于 $y=1$ 处，其朝向为水平方向，但是照相机向左转动了一定角度使得 $\mathbf{n}=(0.74,0,0.67)$。现在通过在水平面上把那两套直线进行延伸，可以得到水平面上的两个灭点，并且两个灭点之间间隔很远。(如何数值计算出灭点?)由于远处的线相互交织在一起十分拥挤，为了清晰起见这里并没有绘出。

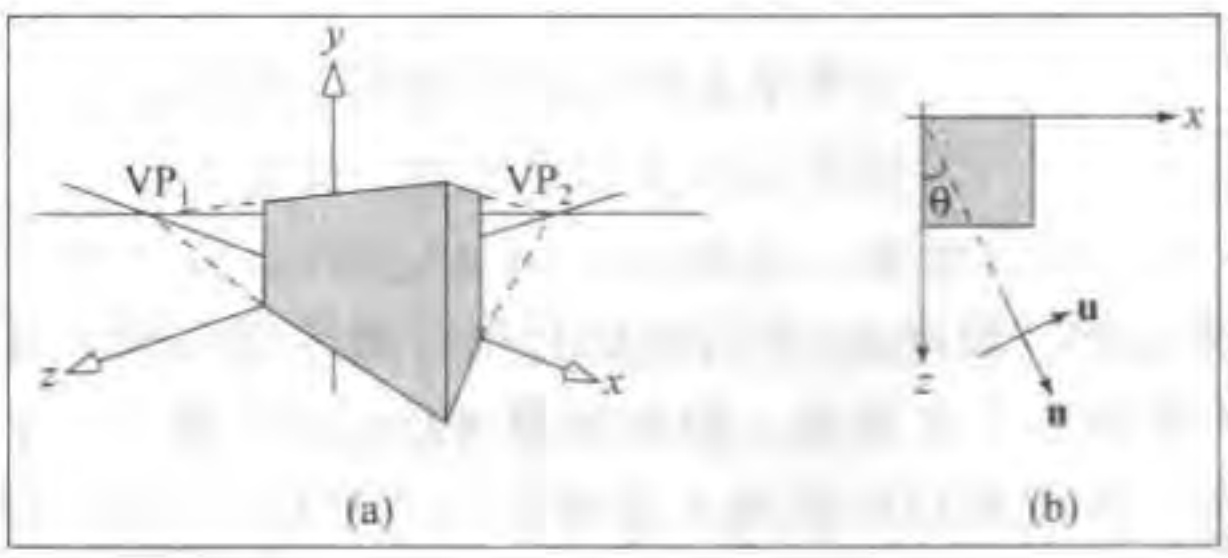


图 7.41 两点透视图

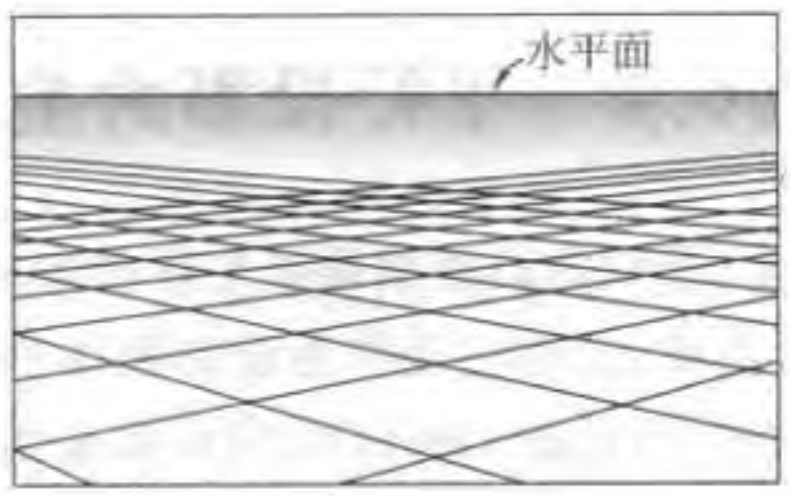


图 7.42 无穷网格线的两点透视图

3. **三点透视**：所有的三个主轴其灭点都位于有限远处：三个主轴都穿透视平面。由于 **n** 不与任何一个轴垂直，所以它所有的分量均不为 0。

艺术家经常在透视绘画中使用灭点，它可以突出表达某些特征或者增加戏剧般的效果。图 7.44 中用三点透视表达的建筑场景可以说明这一点。(画中的三个灭点位于何处?)著名的荷兰艺术家 M. C. Escher 凭借奇异且带有视觉错觉的三维绘画闻名于世，他创作的一些作品就使用了三点透视。

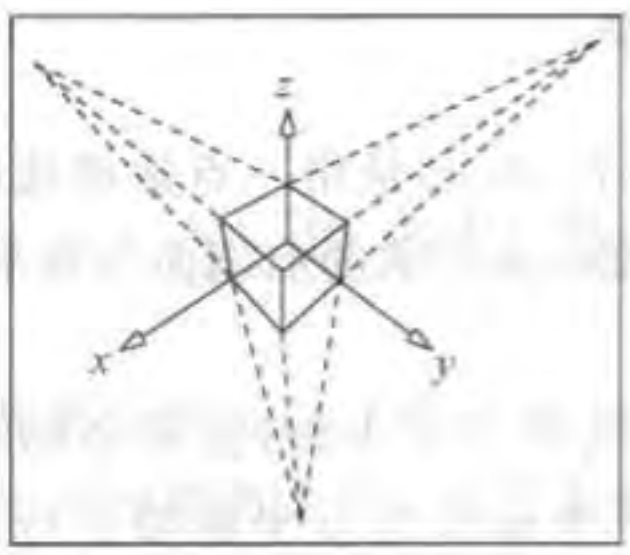


图 7.43 三点透视

图 7.44 三点透视范例

练习

练习 7.6.1 生成一点透视

在应用程序中，为放置理想的照相机我们需要选择好 eye 和 look 的值，此时 **n** 就是 eye−look。选择三套 eye 和 look 值，生成单位立方体的一点透视图。

练习 7.6.2 灭点的位置

用手绘出单位立方体和世界坐标系轴的一点透视图，其中 eye=(2, 3, 4)。

练习 7.6.3 计算灭点

在图 7.41(a)中，对于任意的值 θ 计算图中两个灭点的位置。

练习 7.6.4 生成两点透视

请选择三套照相机的 eye 和 look 值，然后分别生成单位立方体的有趣的两点透视图。对于每个例子请计算灭点的实际位置。

7.6.2 平行投影的类型

下面我们继续研究平行投影的分类。到目前为止，我们仅仅在第 5 章简要地提及过平行投影，那里它是作为绘制物体的一种简单方式出现的。利用函数 glOrtho()建立一个视景体，然后让 OpenGL 完成其余的工作。现在我们稍微深入地研究一下平行投影的性质，并且区分出日常使用中平行投影的各种类型。

我们知道在透视投影中场景里所有的点 P 都沿着**投影线**投射到视平面上，各条投影线

汇聚于视点位置：每条投影线由点 P 自身和视点决定，所以有各种各样的投影方向。相反，作为平行投影，它的所有投影线都是同一个方向，不妨设为 $\mathbf{d}$。图 7.45 展示了两个将要被投影的点。点 P 投影到视平面上的点 p，点 Q 投影到点 q，它们的投影方向都为 $\mathbf{d}$。

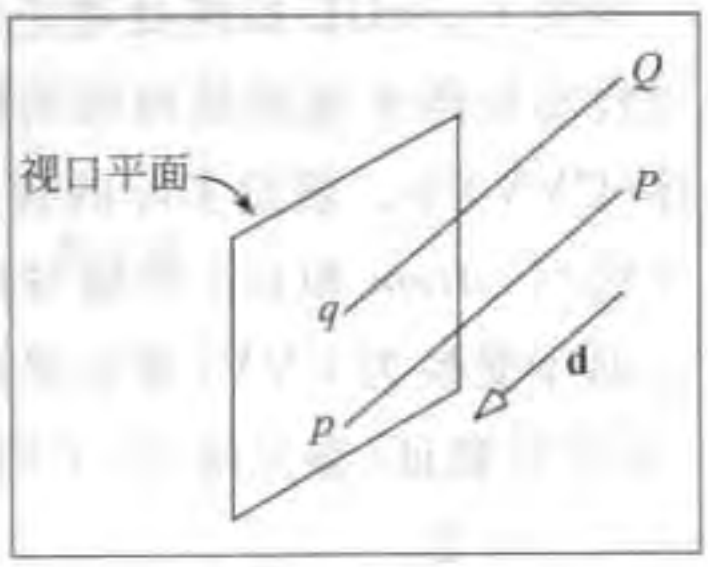


图 7.45　平行投影

平行投影的若干理论

点 P 的投影点 p 位于何处？假设平面的法向量是 $\mathbf{n}$ 并且平面通过点 B。那么由点 P 出发沿着方向 $\mathbf{d}$ 形成的射线与平面的交点就是点 p。先前我们已经多次做过这样的计算。如果射线的参数形式为 $P+\mathbf{d}t$，那么把它代入平面方程 $\mathbf{n}\cdot(P-B)=0$，然后求出 t。然后再把这个 t 值代回 $P+\mathbf{d}t$ 得到交点：

$$p = P + \mathbf{d}\frac{\mathbf{n}\cdot(B-P)}{\mathbf{n}\cdot\mathbf{d}} \tag{7.16}$$

这与我们在透视投影中得到的表达式有很大的不同；特别的，它不会因距离而产生透视缩短的效果：分母里没有出现包含点 P 的项（另外请注意可以用 $-\mathbf{d}$ 替换 $\mathbf{d}$ 而不影响得到的投影结果）。

我们专门使用熟悉的照相机模型来看一看上述等式到底可以揭示出什么信息。假设我们被置于照相机坐标系中，所以 $\mathbf{n}=(0,0,1)$，并且投影平面是 xy 坐标平面那么 $B=(0,0,0)$。（这比投影到近平面要稍微简单一些。）因此可以求得点 p（请验证）：

$$p = \left(P_x - d_x\frac{P_z}{d_z}, P_y - d_y\frac{P_z}{d_z}, 0\right) \tag{7.17}$$

平行投影可以划分为两个主要类型，如图 7.46 所示。

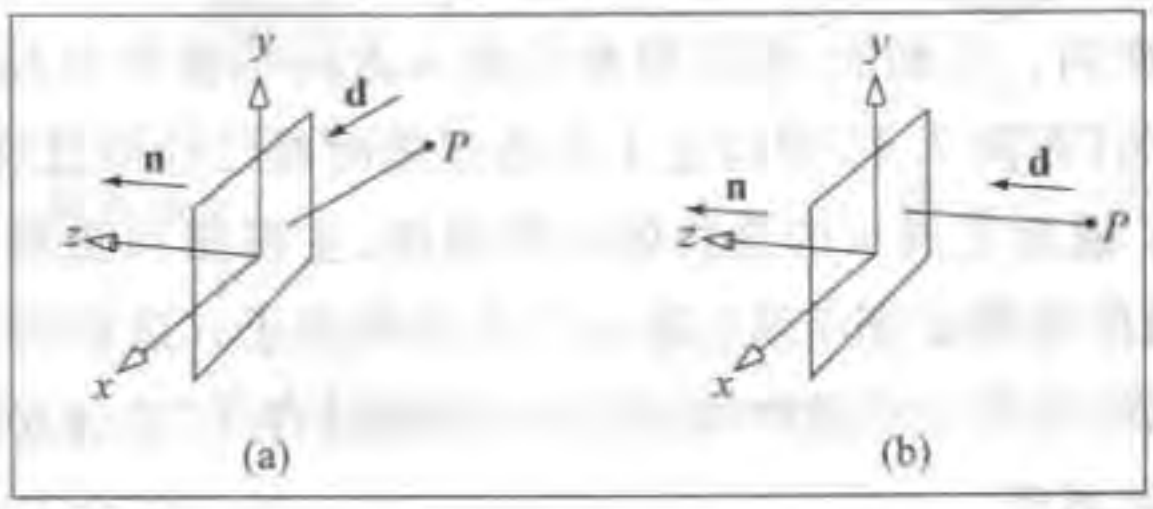


图 7.46　在照相机坐标系中的平行投影

(a) $\mathbf{d}$ 取一般的值；(b) $\mathbf{d}$ 与 $\mathbf{n}$ 平行

1. 斜投影：投影方向 $\mathbf{d}$ 与 $\mathbf{n}$ 不平行。
2. 正交投影：投影方向 $\mathbf{d}$ 与 $\mathbf{n}$ 平行。

图 7.46(a)中的方向 $\mathbf{d}$ 取的是一般情形并不平行于 $\mathbf{n}$（当然我们不会让 d_z 等于 0）。图 7.46(b)是 $\mathbf{d}$ 平行于 $\mathbf{n}$ 的情形，所以这里的 d_x 和 d_y 等于 0。下面我们仅讨论正交投影，因为在日常应用中斜投影并不经常使用。

正交投影

在这种情况下 d_x 和 d_y 都等于 0。那么此时的 $p=(P_x,P_y,0)$。因此正交投影无非就

是抹去点 P 在照相机坐标系下的第三个坐标分量。

研究 OpenGL 如何处理正交投影是一件很有意思的事情。回忆以前的内容,OpenGL 使用投影矩阵来描述照相机的视景体。特别地,投影矩阵指明了如何把顶点变换到正则视景体(CVV)中。假设实际的视景体在 x 轴方向从 l 到 r($left$ 和 $right$ 的缩写),在 y 轴方向从 b 到 t($bottom$ 和 top 的缩写),在 z 轴方向从 n 到 f($near$ 和 far 的缩写)。为了把这个平行六面体变换为 CVV,我们必须进行平移和放缩操作使得 CVV 在每个维度上都从 -1 到 1。很容易验证(参见练习)下列矩阵

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(用于正交投影的 OpenGL 投影矩阵)} \quad (7.18)$$

可以把上述视景体变换为 CVV。此矩阵可以由函数 glOrtho()生成。如果你用它乘上点 P(表示成齐次坐标形式),事实上就得到了点 p,OpenGL 会对它进行裁剪测试并且映射到视口。请注意 OpenGL 并不像我们前面讲到的那样把 z 分量设置为 0。相反,它进行的是变换操作而非投影。最终它还是会进行真正的投影操作,分离出 z 分量,使用 x 和 y 分量在屏幕坐标系上进行绘制,z 分量的作用是进行深度测试。

正交投影的分类

在前面的分类图表中我们可以看到正交投影有很多类型。不同类型的分类依据是照相机在世界坐标系内的朝向。特别的,根据照相机的 **n** 方向与世界坐标系中坐标轴的关系可以给各种类型进行命名(在图 7.40 中讨论 k 点透视的时候已经做过这种尝试)。

多视角正交投影:通常它展示的是物体的俯视图、主视图和侧视图。**n** 分别于 **k**、**i** 和 **j** 平行,然后绘制出想要的物体。图 7.47 是一个展示的例子。这种视图十分适合于工程制图,因为你可以直接从绘图纸上测量物体的尺寸,尤其适合于"立方形"的物体,因为它的面可以与世界坐标系的轴对齐。

轴测视图:在轴测视图中,为了使物体形状的 3D 感表达得更强,要对 **n** 进行一定的选择。通常在此种视图中,**n** 并不与任何主轴平行;更确切地说,要调整好朝向使得(立方形)物体三个相邻面都可见。选择物体中的哪些面作为可见面取决于面的重要程度。除此之外,其中的一个主轴通常是垂直方向的。物体中的平行线在视图中当然依然平行,只不过线会有一定程度的**缩短**。在图 7.48 中 **n** 与 x 坐标轴成 α 角度。

与视平面的法线成 α 弧度的单位线段在视图中的长度为 $\sin(\alpha)$,所以可知**缩短因子**等于 $\sin(\alpha)$。

依据有多少个主轴被同等程度地缩短,可以把轴测图分为三类:

(1) 等测图("同等程度"):所有的主轴都被同等程度地缩短。

(2) 二测图("两种程度"):有两个主轴被同等程度地缩短。

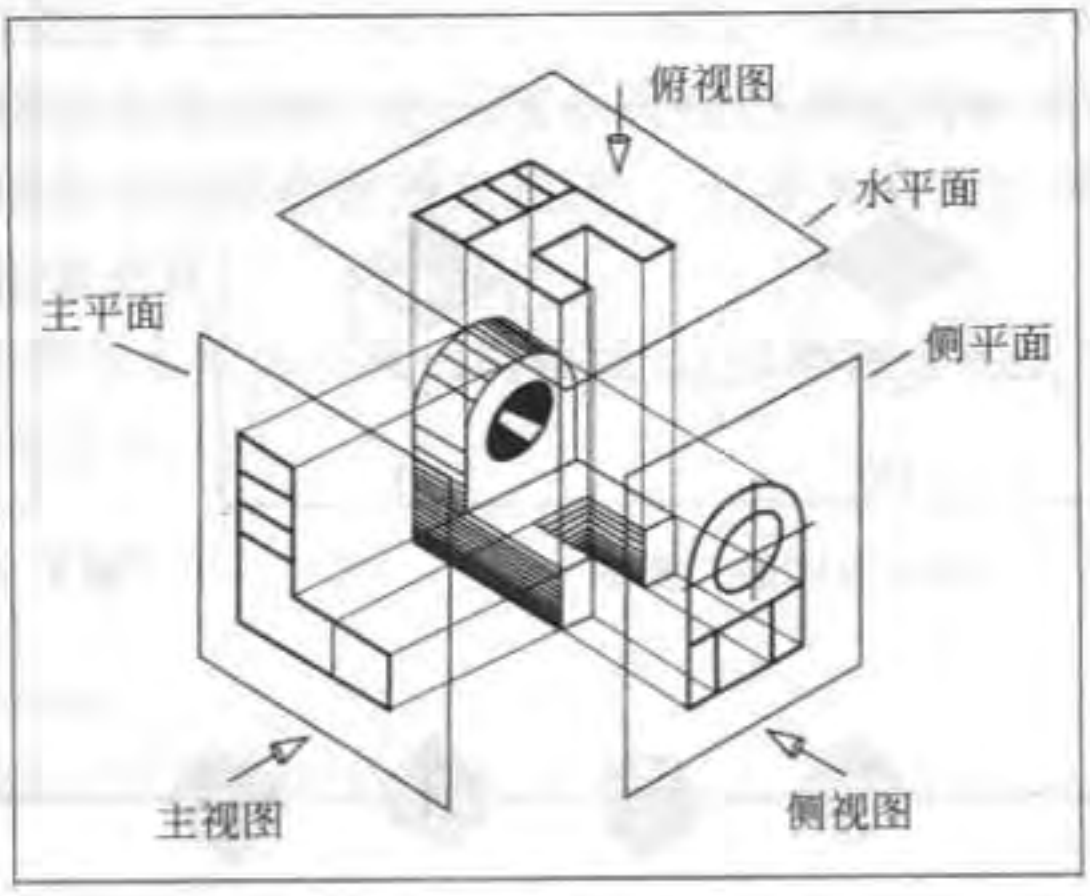


图 7.47　多视角正交投影

(3) 三测图("三个程度")：三个主轴的缩短程度各不相同。

等测图：图 7.49(a)是立方体的一个等测图，它是沿着立方体八个对角线方向中的一个观察得到的。所有的三个主轴被同等程度地缩短。从图 7.49(b)可以看出透明立方体的等测图是一个包含三条对角线的正则六边形。

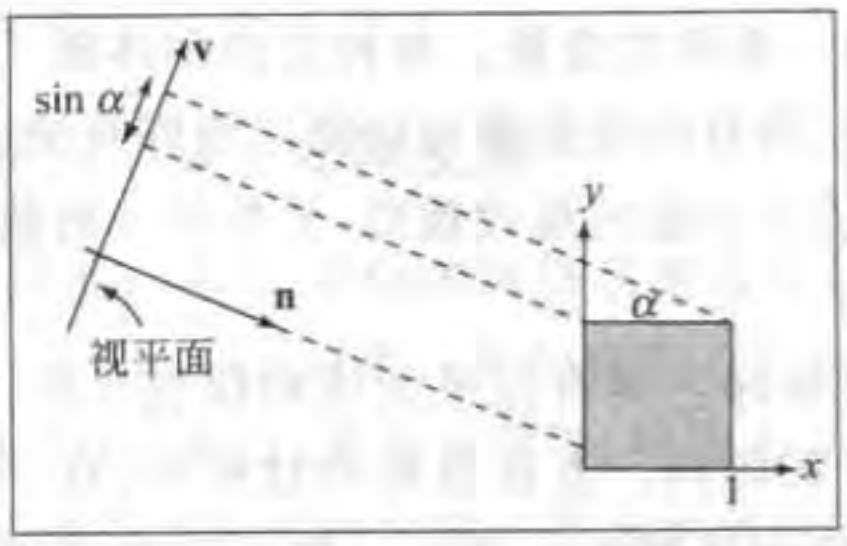


图 7.48　缩短因子

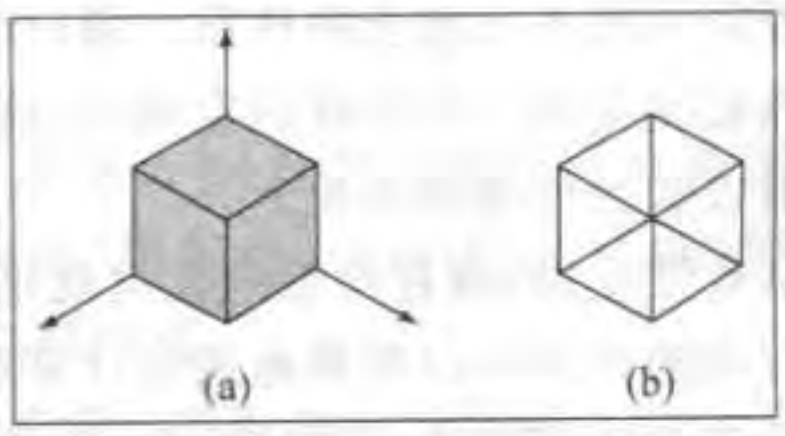


图 7.49　立方体的等测图

获得等测图的条件是 $n_x = \pm n_y = \pm n_z$。在等测图中主轴投影之间的夹角都相等。主平面内线段的长度可以从绘图中直接测量得到，把这个长度乘上一个固定的值(是多少?)就可以获得那条线段的真实长度。

二测图：当只有两条轴与 **n** 夹相同角度时，那么得到的视图为**二测图**。对于二测图，必须有两个方向余弦相同，因此有 $n_x = \pm n_y$，$n_x = \pm n_z$ 或 $n_y = \pm n_z$。图 7.50 展示了一系列的二测图；其中的第三个图几乎就是等测图。对于每一个视图都有 $n_x = n_y$，并且取值都是负值(n_z 在第一张图中等于多少？在最后一张图中又等于多少?)。不同的朝向反映了各种面不同的要强调的程度。

三测图：最后，如果三个主轴与 **n** 的夹角各不相同，那个生成的视图为**三测图**。图 7.51 是三测图的一个例子，其中 **n** 的分量在选择上几乎是任意的。当朝向选择恰当的话，三测图看上去会十分自然。

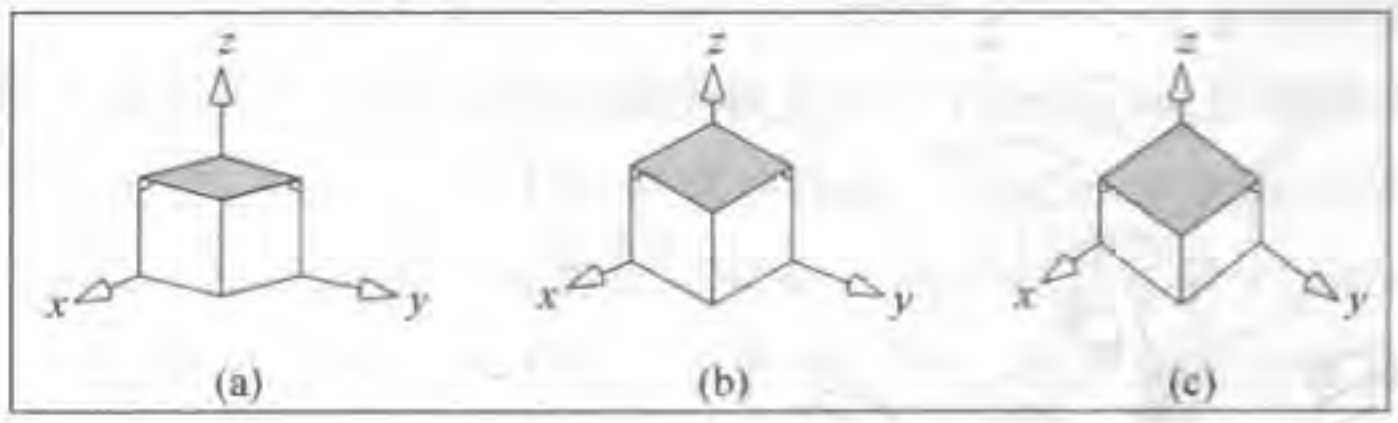


图 7.50 一些立方体的二测图

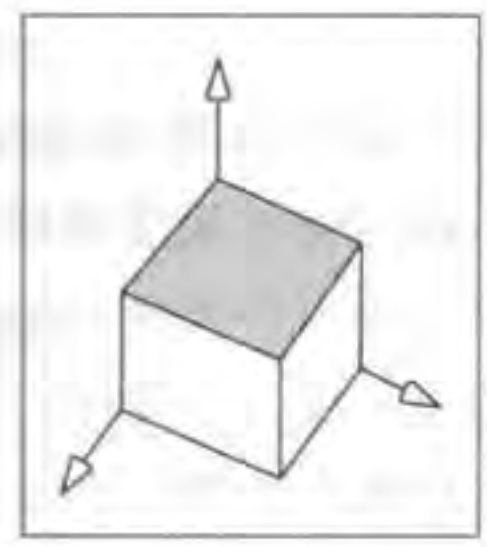


图 7.50 一些立方体的二测图

本章小结

本章中我们学习了如何在程序中定义并控制一个“观察系统”,这样我们就能够生成从不同视点观察 3D 场景所得到的图像,并且还可以在动画中让照相机游走于场景之中。一个视点外加一个视景体就建模构造出一台照相机,它拥有自己的坐标系。一般地,它与针孔照相机很相像,把点投影到一张视平面上。视景体内的点被投影到视平面,而落在视景体之外的点则被裁剪掉。

观察过程要用到的许多数学知识都牵扯到点在不同空间中的几何变换问题。OpenGL 所使用的图形绘制管道对进入绘制管道的顶点实施一系列的变换。每种变换都体现了照相机和投影机制各自独立的性质。通过使用齐次坐标,所有的变换都可以使用矩阵乘法完成。最终点被映射到一个特殊的空间中,这个空间十分适合于边的高效裁剪,另外单一的裁剪算法就可以满足任何照相机。

从本质上讲,透视投影的几何就是某种相似性,幸好它具有保平面性和保包含性,所以整条直线的投影可以简单地通过计算线上两点的投影得到。点在投影的过程中,有一个变换要使用点的深度进行除法操作,这样就使得远处的物体看上去更小一些。因为要用到空间坐标除以一个公共除数这样一个除法操作,所以这种变换就十分适合于采用齐次坐标形式,这里的除数就是第四个分量。于是使用矩阵乘法后面再跟一个除法操作就可以实现这个变换,并且一个投影与另一个投影之间不同的几何细节可以通过矩阵的参数辨别出来。

现在整条图形绘制管道就可以表达为:先进行一次矩阵乘法,接着是裁剪操作,然后进行透视除法,最后通过一次矩阵乘法变换到屏幕坐标系。第一个矩阵通常分成 modelview 矩阵和投影矩阵两部分,这样就可以单独指定出照相机的位置和朝向而其视景体无关。我们前面已经看到 modelview 矩阵包含两个变换:一个变换是把场景中的物体变换到合适的位置和尺寸,另一个变换反映了场景中的照相机位置和朝向。

案例分析

案例 7.1 使照相机在场景中进行游走

(难度:Ⅱ)编写一个允许照相机在场景中进行游走的程序。用户使用键盘来控制照相

机。每按一次键，照相机就沿着三个方向中的一个移动一段距离，或是绕着某一个轴旋转一定角度（与图 7.13 中调用的函数类似），然后场景就在照相机的新视点处重新绘制一次（你来决定哪些键用来移动照相机而哪些键用来旋转）。还要允许用户使用其他的一些键来改变可视角度以及照相机的宽高比。

如果你像图 7.24 那样在水平面上画出**网格线**，这样绘制出的图形就更易于理解。这很容易通过下列的循环语句实现：

```
for(int x=-100; x<100; x++)
{
    glBegin(GL_LINES);
      glVertex3d(x,0,100);glVertex3d(x,0,-100);
    glEnd();
}
```

它绘制了 200 条 x 坐标线。也可以类似绘制出 z 坐标线。增加一个控制键使得用户可以选择网格线的显示与隐藏。

OpenGL 提供了许多图元用来构建有趣的场景。请使用第 5 章所介绍的一些物体，例如球面、锥面、十二面体和茶壶，构造生成这样的场景。请绘制出它们的线框图。稍后我们还会讨论如何绘制它们的实体图，这需要建立好光源并且给物体附上材质。请参考第 5 章中的例子。

案例 7.2　立体视图

（难度：Ⅱ）扩展你的网格观察程序使得它可以生成图 7.34 描述的物体立体视图。要构造三个 Camera 对象。先用标准方法构造出一个“独眼”照相机，然后使用“独眼”照相机的相关信息（视点、方向 **u** 等等）构造出左视点和右视点照相机。

并排在屏幕的两个视口里显示照相机生成的相应图像。要锻炼你的眼睛可以分别盯住两幅图像，这样才能获得立体效果。

还要允许用户能够在场景中移动左右照相机（两台照相机是固定在一起的）获得更佳视图。请试着找出两台照相机之间的最佳距离，这样观看近处的和远处的物体都可以获得十分自然的效果。

案例 7.3　可以极大提升效率的背面剔除操作

（难度：Ⅱ）当我们渲染一个网格物体时，它的每一个面片都被输送到图形绘制管道，其中的顶点要经历变换操作，最后面片被绘制出来。但是对于物体的任意一种朝向，都存在许多背离观察方向的面片。它们被称为**背面**，并且没有必要被输送到图形绘制管道，因为它们都是不可见的：会存在比它们更接近视点的面片把它们挡住。图 7.52(a)是经典的仓房模型。对于任意一个照相机位置，仓房的面片中都存在背面：图中标出了两个。

我们说一个面片是背面，如果它是背离视点方向的。这也就是说它的外法向方向 **m** 与视点与面片任意一点 P 所成向量的夹角小于 90°（参见图 7.52(b)）。若两个向量的夹角小于 90°，那么它们之间的点积为正。所以针对一个面片是否是背面可以有如下的测试：

F 是一个背面，如果 $(P-eye)\cdot\mathbf{m}>0$

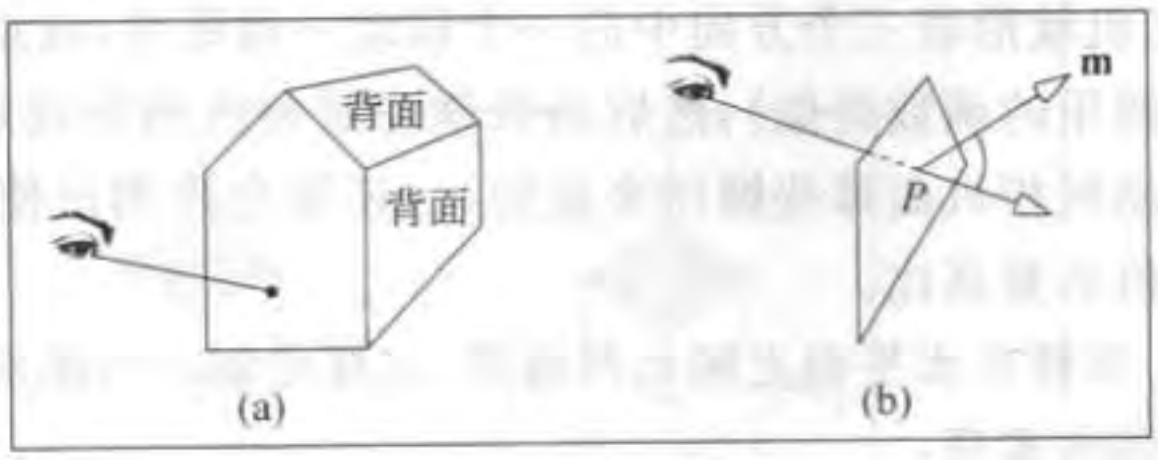


图 7.52

(a) 仓房的背面;(b) 背面的定义

这个测试的开销并不大。所以可以先测试一个面片是否是背面,如果是就将其丢弃,这样就可以显著提升绘制复杂网格的速度。

修改图 6.15 中 Mesh::draw()的代码,让它只绘制网格物体的前面,跳过那些被测试为背面的面片,这可以通过插入语句 if(isBackFace(f,...))continue 实现。开发出函数 int isBackFace(f,...)的代码,当 f 是背面时它返回 1,否则返回 0,只把那些必要的参数传递给它。开发一个可以基于任意视点绘制网格的应用程序,比较一下背面剔除法使用与不使用时的速度。

进一步阅读

贯穿整章都提到过,"OpenGL Programming Guide"【Woo04】为建立和控制 3D 图形观察系统提供了一个清晰的介绍。Foley 等著的【Foley03】涉及到了更多的细节。"Jim Blinn's Corner-A Trip Down the Graphics Pipeline"【Blinn96】中的若干文章针对投影、视角变换、裁剪方法提供了清晰有趣的论述。Angel【Angel03】中也含有照相机修正和观察系统方面很好的描述。

第 8 章

几何体的真实感渲染

本章学习目标

- 增加三维场景绘制的真实感
- 探讨光在物体表面的反射性质
- 在有光源的环境下渲染多边形网格
- 探讨如何使多边形网格的渲染结果看上去平滑
- 为物体表面增加纹理
- 为场景中的物体绘制阴影

预览

8.1 节讲述了增加三维物体渲染的真实感的必要性。

8.2 节介绍了计算机图形学中几种不同的着色(或者光照)模型,以及计算环境光、漫反射和高光对物体最终颜色贡献的工具。另外这节还讲述了在 OpenGL 中,如何建立光源,如何描述面片的材质属性,和渲染多边形网格时 OpenGL 图形管线如何运行。

8.3 节侧重于渲染多边形网格表示的三维模型。介绍了平面着色、Gouraud 着色和 Phong 着色方法。8.4 节介绍了一种简单的基于深度缓存的隐藏面消隐技术,这也是 OpenGL 所采用的方法。适当的隐藏面消隐能够显著地增加结果的真实感。

8.5 节介绍了在物体表面绘制纹理(比如图片,剪切画或者标签)的方法。通过绘制纹理能够使得物体看上去就像表面贴上了标签一样,或者让人觉得就是用木头、砖等真实材料制成的一样。本章也会介绍纹理映射的工作流程。8.5.4 节展示了使用 OpenGL 为物体添加纹理的一个完整程序。8.5.6 节讨论了在反光的曲面上映射纹理,使得物体看上去就像置身于所在场景中一样,基于这种原理的很多方法能够使得三维场景看上去更加真实。

8.6 节介绍了阴影的绘制技术,并且和 OpenGL 所用(功能很弱)的方法进行了比较。8.7 节介绍了凹凸贴图,它可以产生物体表面粗糙或者凹凸的不平的效果,从而使得程序员不用花很大力气去修改模型的真实表面数据(比如一个网格数据)来实现粗糙和凹凸不平的效果。

本章结尾有一些案例分析,对以上提到的内容作更深入的介绍,供有兴趣的读者钻研和实现。

8.1 概　　述

在前面的章节中,我们开发了一些工具进行物体的网格建模,通过操作照相机拍摄这些模型生成照片。现在我们想增加一些工具使得这些物体看上去更加有趣和真实。第 5 章的

一些例子使用了一些 OpenGL 函数渲染出闪闪发光的茶壶和有光源场景中的球体,但是没有涉及到这些实现背后的理论依据。这章我们就要介绍**渲染**一幅三维物体图像所涉及的知识。实际上这就是计算图像中每个像素看上去应该是什么颜色的问题。这很大程度上取决于不同的**着色模型**(**shading model**)。着色模型就是试图描述光源发出的光线和场景中的物体是如何相互作用的。因为实现上的限制,我们通常不会模拟有关光散射和反射的全部物理原理,因为这是非常复杂的,而且会导致出现很长的软件开发周期(一个开发过程中最耗时的部分)和运行起来非常慢的算法。但是有一些近似的模型已经被提出,它们能够很好地提供不同级别的真实感效果。

为了介绍涉及的基本问题,我们开始将由浅入深地介绍实现各个级别真实感效果的技术,然后介绍如何把这种技术应用到实际中去,同时看看 OpenGL 是如何替我们完成这些繁重的工作的。

在各种级别的真实感技术中,处于最底端的是**线框**(**wireframe**)渲染。图 8.1 是用线框方法渲染的一幅 540 个立方体混乱摆放的效果图。每个物体只有边被画了出来,同时你的视线能够透过物体看到后面的东西。这让我们几乎不可能看出来这是什么东西。(立体的图像只对我们有一点点帮助)。

图 8.2 没有绘制在面后部而被遮挡的边,使得结果有了明显的改善。我们可以把这种方法称作"线框加隐藏面消隐"技术(真正准确的说法是:可见线框渲染技术)。虽然只有边被画出来,但是每个物体看上去都有立体感,而且能很容易地看出来每个物体从哪开始到哪结束。注意图中的某些边只画出了一部分,因为它们斜插入遮挡面的后面从而被截断。

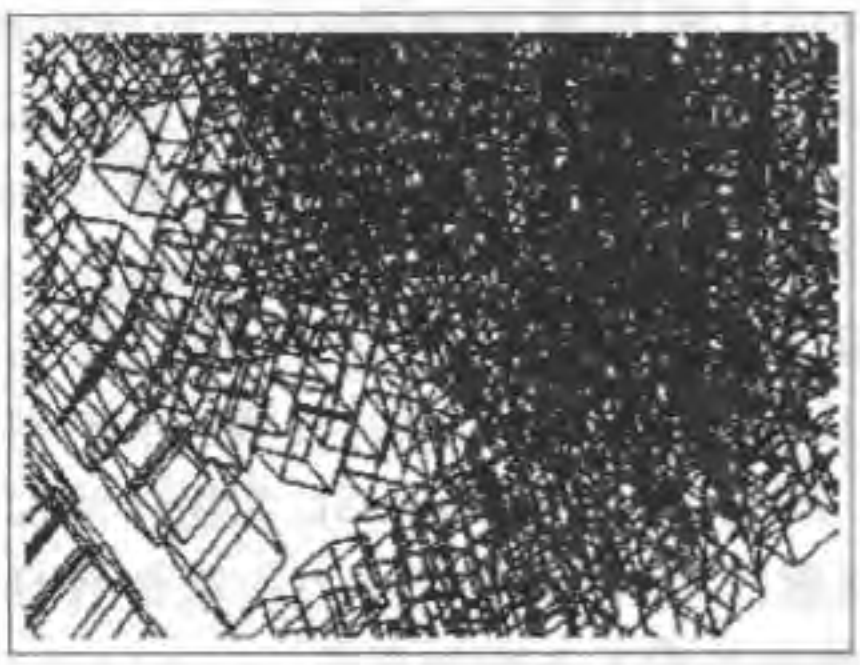


图 8.1 使用线框渲染的场景

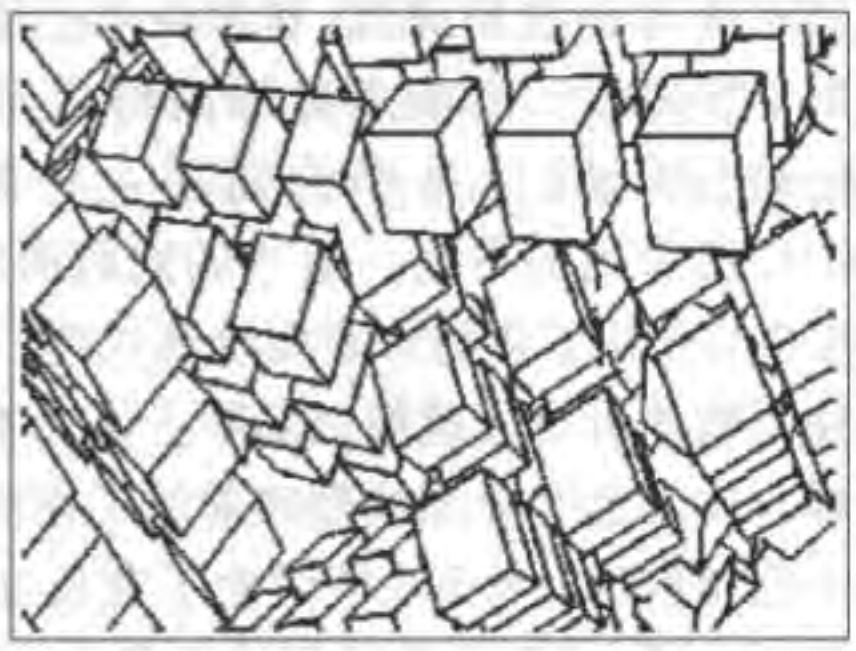


图 8.2 边可见部分的线框渲染

为满足读者的好奇,这里提示一下,这张图是使用 OpenGL 并且打开深度缓存功能绘制出来的——对于每一个立方体,用 drawMesh()画白色的面,再用 drawEdges()画黑色的边。具体的过程我们已经在第 6 章中讨论过。

下一步我们把真实感效果提高一个层次,希望渲染出的物体象是处于有若干光源照明的场景中。根据物体表面性质和光源的位置,每个物体不同的部分反射光的多少也不同。渲染这样的场景需要有一些自动的方法计算每个面上各个部分的颜色,这个方法就是着色模型。

图 8.3 是一个多边形网格场景:权架(见例 5.6.3),在一个圆柱和圆台上的多面球体。图 8.3(a)是**线框**显示的结果,图 8.3(b)是着色模型显示的结果(附加隐藏面消隐)。面向光源的面(光源处于物体的上方)比背向光源的面更亮(请仔细观察这个差异)。这幅图还展示了平着色(flat shading):每个面上反射的光量只根据其上某点计算一次。因此同一个面上

所有点的灰度是一样的。

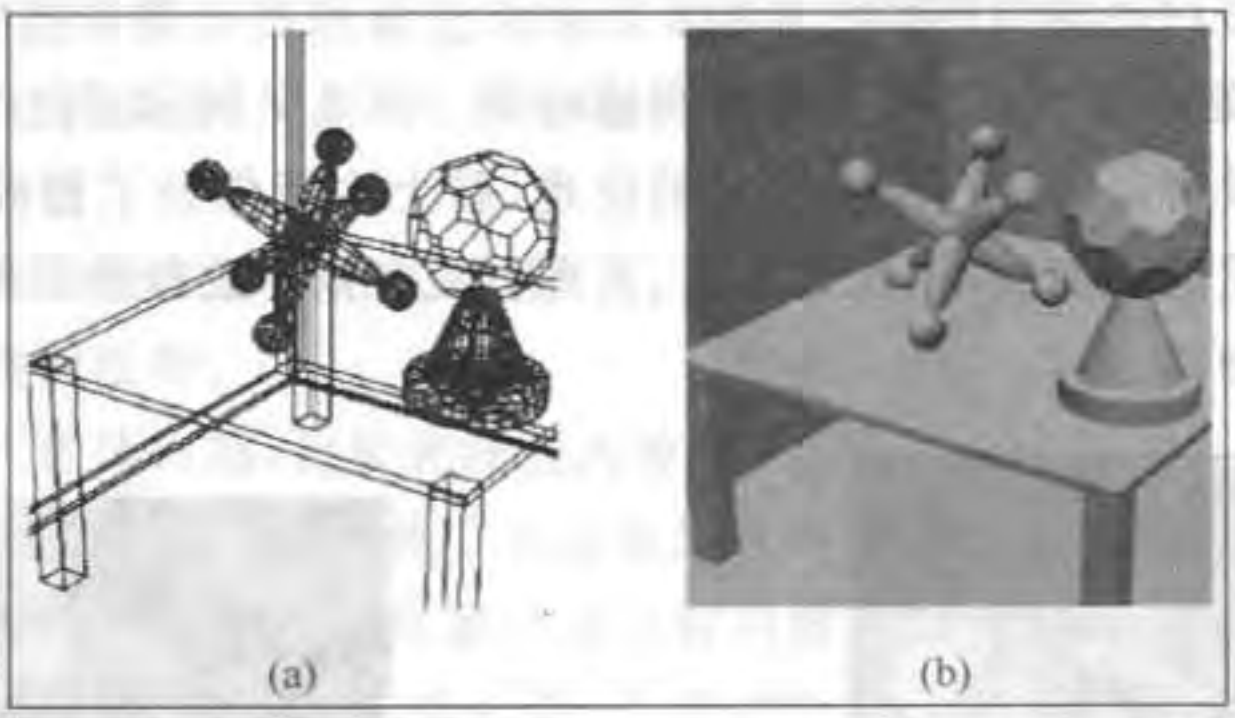


图 8.3 使用着色模型的网格

(a) 线框示图；(b) 平着色

更复杂一些，我们应该体现出颜色的使用。颜色这部分会在 8.2.6 节介绍。

在第 6 章我们讨论过使用网格来逼近平滑曲面的物体。这样的网格模型渲染的结果也应该是平滑的，更像平滑的曲面而不是一个一个的多边形。在图 8.4 中，权架上的球和圆柱是平滑的物体，而多面球体和桌子不是。图 8.3(b)使用的是平面明暗处理，而图 8.4 是同一场景使用**平滑着色**(**smooth shading**)的渲染结果。这里同一个面上不同点的灰度不同是通过插值实现的，这就是 **Gouraud 明暗处理**(**Gouraud shading**)。这样通过不同灰度值使得本来平滑的物体看上去更加平滑(虽然不是完全的平滑)，多边形的边界消失了，从而给我们这种平滑的感觉，而不是感到曲面是由一些小面片组成的。因为多面球体本身就不是一个平滑曲面的物体，所以即使使用了平滑的渲染方法，也会显现出一个个单独的可见表面。我们会在第 8.3 节介绍 Gouraud 明暗处理。

高光(highlight)可以使得物体看上去更加有光泽。图 8.5 展示了加入**镜面反射光**(**specular light**)后的效果。有关高光的讨论放在了 8.2 节。越有光泽的物体应该看上去越明亮，但是越多局部的高光就越会给人一种像塑料材质的感觉。

图 8.4 使用平滑着色的场景

图 8.5 加入镜面反射高光(specular highlight)的效果

阴影是另外一个能够增加图像真实感的渲染效果。图 8.6 展示了在图 8.5 同样的场景

中适当增加了阴影后的渲染效果。其中物体的阴影投射到了与其相邻的其他物体上。我们将在第 8 章的 8.6 节讨论实现阴影的算法。

在物体上增加纹理可以显著地提高真实感效果。图 8.7 展示了把不同纹理绘制到不同表面后的效果。场景背后的墙壁贴上重复的纹理,看上去就像贴了瓷砖的墙一样。使用纹理可以产生不同材质的表面效果,比如木头、大理石或者铜。纹理的另外一个用途是把图片像转印一样包裹在物体上。

图 8.6 带有阴影的场景

图 8.7 映射到表面的纹理

此外还有一些技术能够增加渲染的真实感。在第 12 章我们将更加深入地研究**光线跟踪**(ray tracing)。虽然光线跟踪需要的计算量非常大,但是算法本身易于实现并且能够渲染出许多效果,比如阴影、镜面反射和光线穿透透明物体。

在本章将介绍一些能够渲染出不同级别真实感效果的方法。我们首先从图形学中经典的照明模型开始,它能够使得物体看上去像处于一个或者多个光源的场景之中,同时还会介绍如何绘制多边形网格,使得效果看上去像是平滑的曲线曲面。其次我们会学到一种隐藏面消隐方法,看它如何在实际的渲染过程中发挥作用,这种方法也是 OpenGL 所使用的消隐方法。一般而言,我们将看到这样一个事实:为了实现更真实的效果,选择使用一种渲染模型而不是另外一个模型,总是要付出一定的代价——程序运行得更慢。不过当你使用 OpenGL 编写应用程序的时候,这点或许重要或许不那么重要。

8.2 渲染模型

真实表面上光的反射机制是非常复杂的,受很多因素的影响。有些因素是和几何有关的,例如光源、观察者和物体表面法向的相对方向。另外一些因素与物体表面的性质有关,比如粗糙程度和颜色。

渲染模型描述了光如何在物体表面发生散射和反射。本节我们将会看到一些简单的渲染模型,这些模型假定使用黑白光。这里**黑白**光指只有明暗不同而没有颜色,实际上就是灰度。因此使用一个数值就可以描述它:亮度。我们将会看到如何计算从物体各个部分到达相机的光的亮度。这些计算方法对于普通的光几乎是一样的,除了把红、绿、蓝三个分量分

别单独计算以外。

图形学中常用的渲染模型都假设把为场景照明的光源分为两种：点光源和**环境**光源(ambient light)。这些光源会"照射"到不同物体的表面。入射光和表面的相互作用可以分为三种：

- 一些被物体表面吸收而转化成热能。
- 一些在物体表面反射。
- 一些折射进入物体内部，比如光线进入玻璃。

如果所有的光都被吸收了，那物体看上去是黑色的，这就是物理中所谓的**黑体**。如果所有的入射光都被传导出去，那么物体只能通过反射的效果(或者**折射**的效果，如果把相机置于物体背光面)来看到，关于折射我们将在第 12 章讨论。

这里我们只关心在表面被反射或者散射的这部分光。一部分这样的光正好入射到眼睛，使得我们能够看到物体。到达眼睛的光量的多少取决于物体表面的朝向、光源和观察者。我们假设入射光的反射可以分为两种：**漫反射**(diffuse scattering)和**镜面反射**(specular reflection)。

- **漫反射**发生时入射光浅浅地穿透物体表面然后向各个方向均匀散射。因为散射光和表面有强烈的相互作用，所以光的颜色通常被表面材质的属性所影响。
- **镜面反射**更像镜子和光的作用那样具有很强的方向性。入射光不穿透物体表面，而是直接在外表面反射。结果是产生高光，并且是表面看上去有光泽。在刻画镜面反射的最简单的模型中，反射光颜色和入射光相同，这个性质使得材质看上去趋近于塑料。在镜面反射的更复杂的模型中，颜色在高光的不同区域会有变化，这种模型更好地逼近了金属表面的光泽。我们对这两种镜面反射模型都会进行探讨。

更多的表面是以上两种反射的结合，根据表面特性的不同比如粗糙程度和材质类型，两种反射所占比例也不同。我们可以说从物体表面反射向某一方向的全部反射光是漫反射分量和镜面反射分量之和。对于我们关心的表面上的每一个点，我们需要计算到达眼睛的反射光的每个分量。下面将要介绍的算法就是来完成这个过程。

8.2.1　影响反射光的几何因素

我们需要找到用来计算漫反射和镜面反射分量的三个向量。图 8.8 画出了用来计算从 P 点到眼睛的光量所需要的这三个关键向量($\mathbf{s}$、$\mathbf{m}$ 和 $\mathbf{v}$)。

(1) 法向量 $\mathbf{m}$ 指向 P 点所在表面的法方向。

(2) 向量 $\mathbf{v}$ 从 P 点指向观察者的眼睛($\mathbf{v}$ 是观察者英文 viewer 的首字母)。

(3) 向量 $\mathbf{s}$ 从 P 点指向光源($\mathbf{s}$ 是光源英文 source 的首字母)。

这三个向量之间的角度是计算光强的基本因素。这些角度通常都是在世界坐标系下计算，因为一些变换(比如投影变换)不能保持夹角不变。

网格物体的每个面片拥有两个面。如果物体是实心的，两个面通常称为里面和外面。人的眼睛只能够看到外面(除非眼睛可以进入到物体内部!)，我们计算光的贡献也只考虑外面。但是对于有些物体，比如图 8.9 中的这个敞开的盒子，眼睛有可能看到盒盖的内表面。

能够看到取决于这个面的法向 $\mathbf{m}_2$ 和视点方向 $\mathbf{v}$ 的夹角。如果角度小于 90°则这个面可以被看到。因为这个角度的余弦值正比于两个向量的点乘 $\mathbf{v}\cdot\mathbf{m}_2$,所以只有当 $\mathbf{v}\cdot\mathbf{m}_2>0$ 时眼睛能够看到这个面。

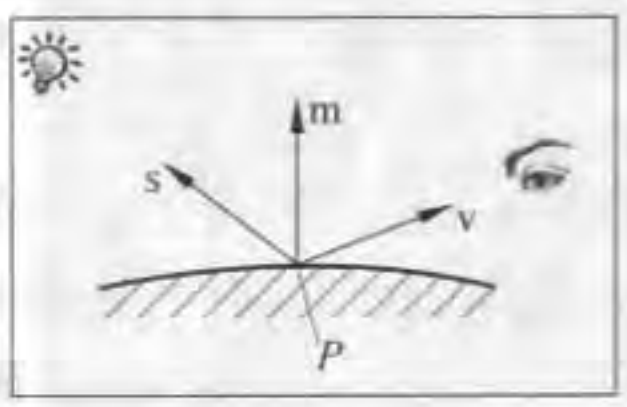


图 8.8 计算反射光时的重要向量

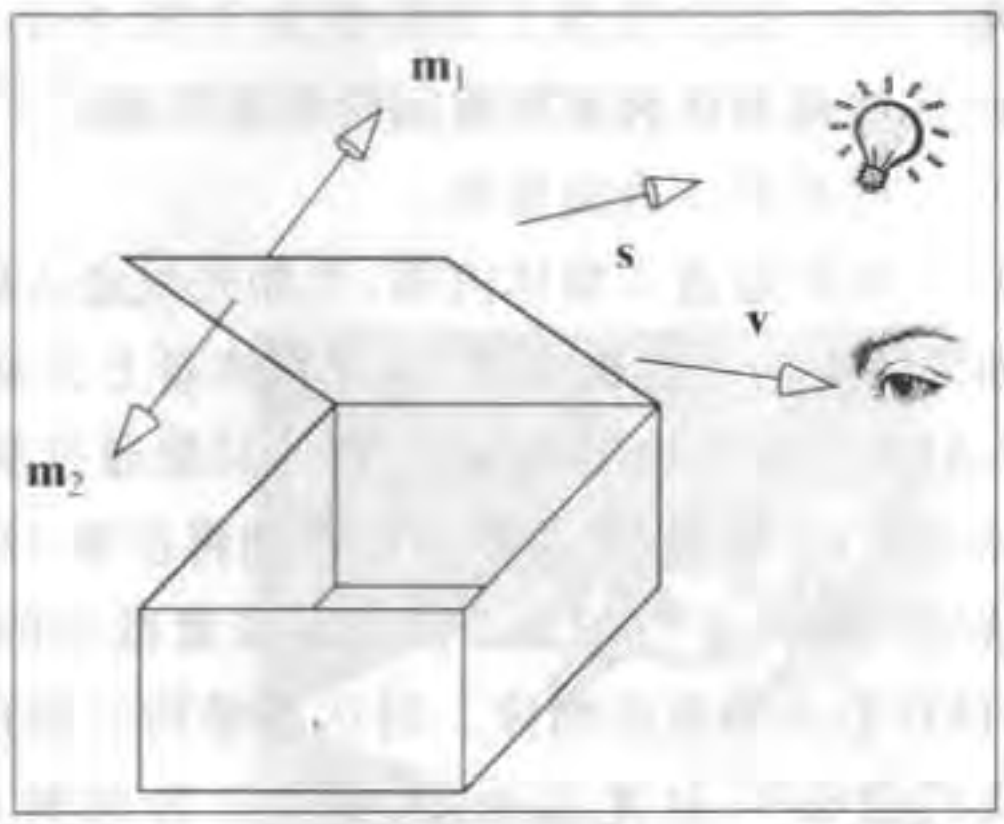


图 8.9 只为面片的一个面计算光照

对于给定面片的某个面我们将推导渲染模型。如果这个面背向视点,则没有任何反射光的贡献。在实际的应用中,必须告知渲染算法所计算的光的贡献来自面片的哪一面或者两面都计算。我们将看到 OpenGL 支持这一点。

8.2.2 如何计算漫反射分量

假设光从一个点光源投向一个面片(曲面的一个小面)。一部分光从这面向各个方向均匀的漫反射出来。一部分漫反射光到达了眼睛,把这部分光的强度设为 I_d。那么 I_d 和 $\mathbf{m}$、$\mathbf{v}$、$\mathbf{s}$ 方向的关系是什么呢?

漫反射的一个重要特点是它独立于视点方向。这点也经常被称为**全方位散射**(omnidirectional scattering)。因为散射向各个方向是均匀的,所以面片 F 相对于视点的朝向没有意义。因此,I_d 与 $\mathbf{m}$ 和 $\mathbf{v}$ 的夹角无关(除非 $\mathbf{v}\cdot\mathbf{m}<0$ 时 I_d 等于 0)。另外一方面,照亮面片的光量和面片相对光源的朝向相关:光量正比于面片能被看到的面积,也就是面片投影的面积。

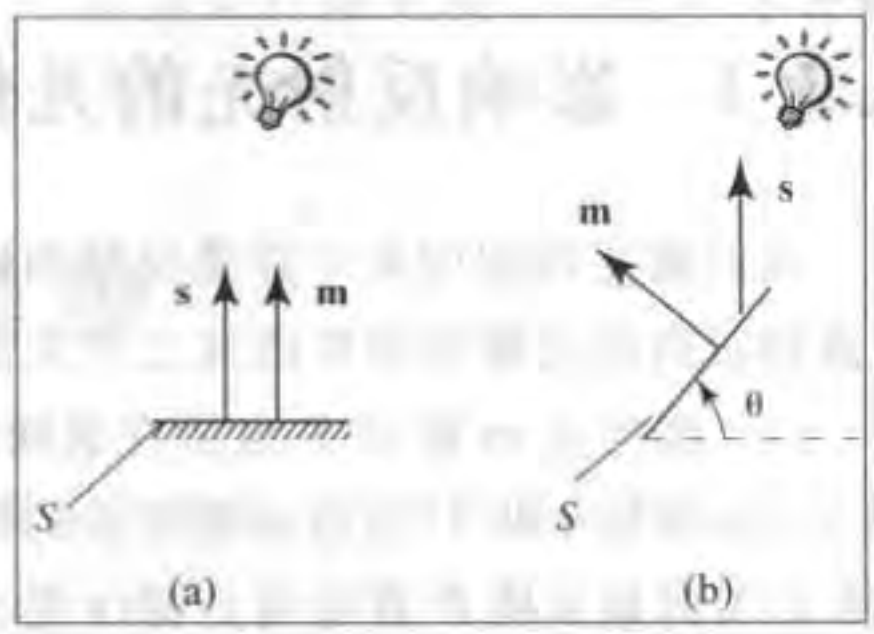


图 8.10 亮度取决于面片的投影面积

图 8.10(a)斜线的部分是一个点光源照在面 S 上,$\mathbf{m}$ 和 $\mathbf{s}$ 平行。图 8.10(b)中面旋转了一个角度 θ。投影的面积只有原先的 $\cos(\theta)$,所以 S 的亮度也成比例的下降。这种亮度和表面朝向的关系常被称作 **Lambert 法则**【Johann Heinrich Lambert, 1782-1777】。需要注意的是,当 θ 接近于 0 的时候,亮度随角度变化很小,这是因为余弦函数在这附近变化很小。但当 θ 接近 90°时,亮度快速地下降到 0。

我们知道 $\cos(\theta)$ 是单位化后 $\mathbf{s}$ 和 $\mathbf{m}$ 的点积,因此我们可以得到计算漫反射分量的强度

等式：

$$I_d = I_s \rho_d \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}$$

式中，I_s 是光源的亮度，ρ_d 是漫反射系数。需要注意的是如果面片背向视点，式子中的点积为负，这时我们希望为 0。所以一个更加准确的漫反射分量计算等式是：

$$I_d = I_s \rho_d \max\left(\frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}, 0\right) \tag{8.1}$$

图 8.11 显示了一个球体在不同光照下的漫反射效果，入射光来自上方，反射系数依次是：0、0.2、0.4、0.6、0.8 和 1。需要注意的是当 ρ_d 等于 0.0 的时候，球体是全黑的。底部的阴影部分（这里点积为负）也都是黑的。

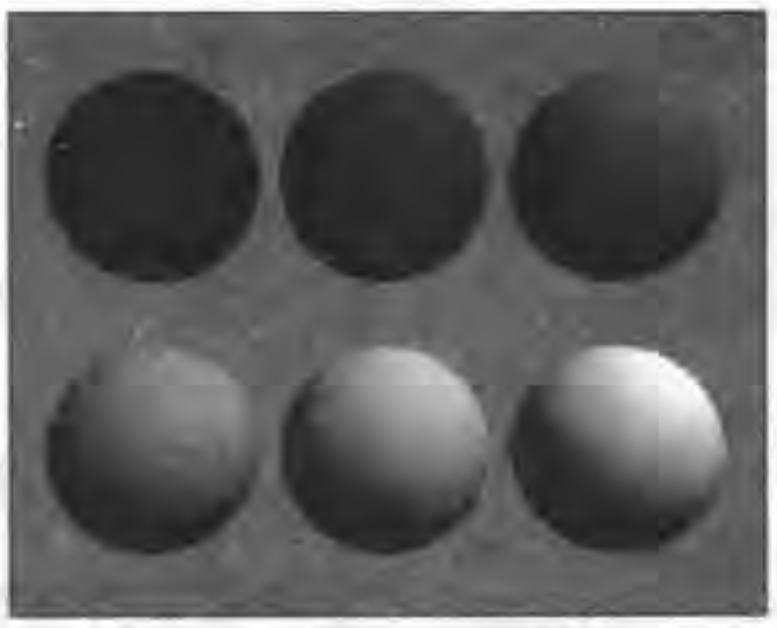


图 8.11　漫反射光下一些拥有不同反射系数的球体

在真实的世界中，漫反射背后的过程要比我们这个简单的模型复杂得多。反射系数 ρ_d 和入射光的波长（颜色）、夹角 θ 和不同的表面物理特性有关。但是为了简化和减少计算时间，渲染图像时这些效果都忽略了。为每个表面选择“合理”的 ρ_d 值，有时需要通过比较结果图片和真实观察效果之间的差别反复试验确定。

在某些渲染模型中距离对效果的影响被考虑其中，虽然这点有一定程度的争议。图 8.10 中表面 S 的反射光强度与 S 到光源的距离的三次方成反比。但是实验结果显示用这种方法生成的图片有夸张的深度效果。另外的一点是，有时候为了方便光源建模，会假设把光源放置在“无穷远”的位置。在这种情况下使用上述反比定理会使得光完全衰减！这里模型中的一个问题是，为了简单我们把光源用点光源建模，但是许多场景的真实照明来自周围物体的反射，这点很难用模型描述（这个效果用环境光分量描述）。所以结果就并不奇怪了，因为在一个不真实的模型基础上严格地附加物理定律最后只能导致不真实的结果。

引进距离项对于大多数图片来说只能增加很小很小真实感。一些过程的确强制光强与视点到物体的距离成反比，但是这不是基于物理原理。实验这个效果虽然很有意思，而且在 8.2.8 节中会看到 OpenGL 提供了一些接口，但是在下面的介绍中我们将不考虑距离因素。

8.2.3　镜面反射

真实物体的散射光并不是各向均匀的，所以镜面反射分量也需要加到渲染模型中。镜面反射能引起高光，当物体有光泽的时候这能够显著的增加渲染图片的真实性。在这节我们将讨论由 Phong【Phong75】提出的一个镜面反射光的简单模型，它非常容易实现。并且由 Phong 镜面反射产生的高光能给物体塑料一样的外表，所以当你打算渲染有光泽的塑料或玻璃的物体时 Phong 模型是不错的选择。对于拥有闪亮金属表面的物体，虽然可以通过在 OpenGL 中精心的选取某些颜色参数粗略的逼近，但是 Phong 模型并不适用于此。一些更复杂的镜面反射光模型对闪亮的金属有更好的表现。但因为 OpenGL 的渲染过程不直

接支持这些模型,所以我们在后面第 12 章光线跟踪的部分再具体介绍它们。

图 8.12(a)给出了光线从光源出发遇到物体表面后向不同方向反射的示例。在我们当前讨论的 **Phong 模型**中,反射光能量最大的方向是完全镜面反射(见第 4 章)的方向 **r**,这个方向反射角和入射角 θ 相等。如果表面是完全的镜面,这个方向也是全部光线反射的方向。附近其他的方向上反射光快速衰减,我们在图中用反射向量的相对长度表示。图 8.12(b)中我们用雷达圈常见的"光束模式"来表示这点。P 点到光束外边缘的长度表示在这个方向上散射光的相对强度。

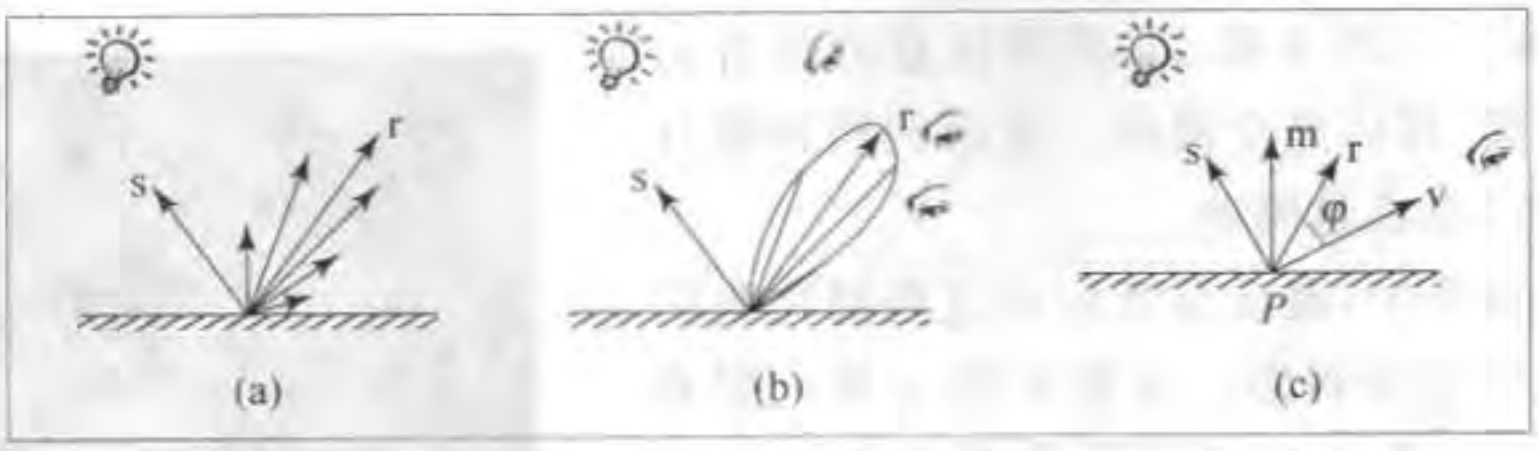


图 8.12 有光泽物体的镜面反射

图 8.12(c) 画出了如何量化这种光束模式。我们从第四章知道完全反射方向 **r** 依赖于 **s** 和表面法向 **m**:

$$\mathbf{r} = -\mathbf{s} + 2\frac{(\mathbf{s}\cdot\mathbf{m})}{|\mathbf{m}|^2}\mathbf{m} \quad \text{(镜面反射方向)} \tag{8.2}$$

对于很光滑但还不是完全镜面的表面,反射光强度随着 **r** 和 **v** 的夹角 ϕ 的增加而衰减。真正的衰减量是一个关于 ϕ 的复杂函数,但是在 Phong 模型中它被表示为 ϕ 余弦值的 f 次幂,也就是说$(\cos(\phi))^f$,其中 f 的值通常在 1~200 之间通过实验确定。

图 8.13 画出了不同 f 值下光强度随 ϕ 值变化的函数。可以发现 f 值越大,反射光越集中于 **r** 方向,也就是接近镜面。理想的镜面可以通过 $f=\infty$ 表示,但是完全反射通常用另外的方式表示,这将在第 12 章介绍。

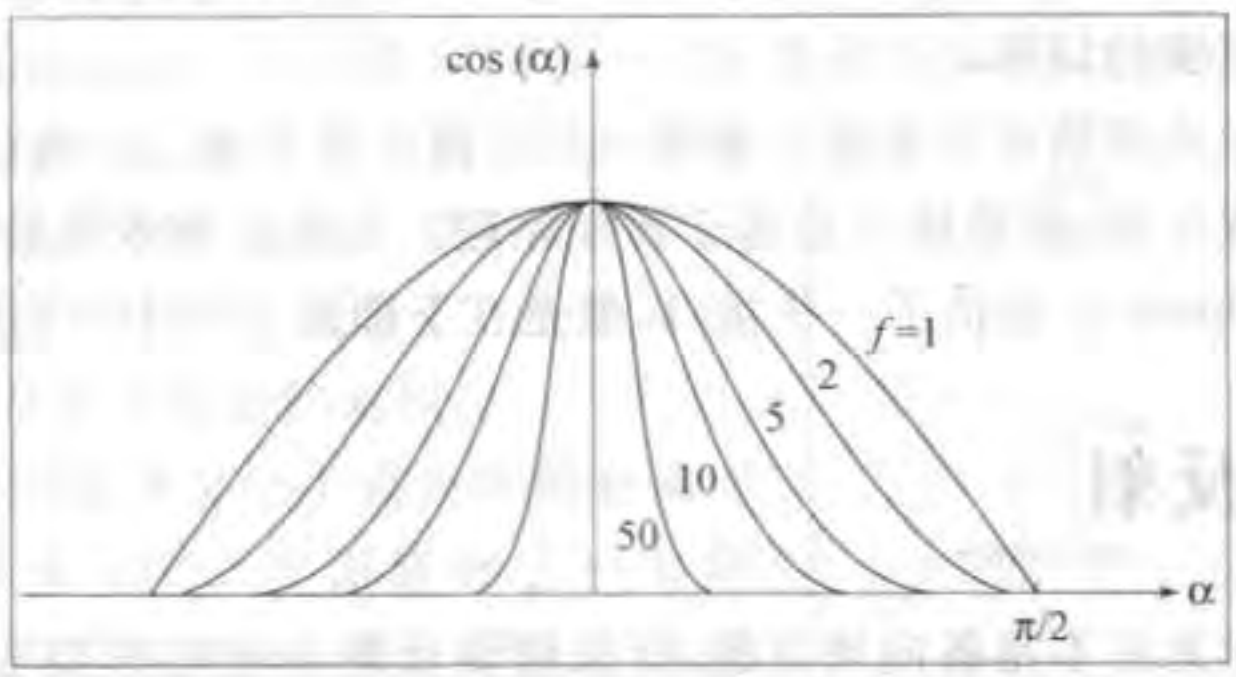


图 8.13 镜面反射光随角度的衰减

用 **r** 和 **v**(单位化后)的点积等价的替换 $\cos(\phi)$,镜面反射的光强度 I_{sp} 可以表示为

$$I_{sp} = I_s\rho_s\left(\frac{\mathbf{r}}{|\mathbf{r}|}\cdot\frac{\mathbf{v}}{|\mathbf{v}|}\right)^f \tag{8.3}$$

其中新的参数 ρ_s 是**镜面反射系数**。和其他在渲染模型中的参数一样,它通常也是通过实验的方法确定(和漫反射一样,如果 $\mathbf{r}\cdot\mathbf{v}$ 为负则 I_{sp} 为 0)。

我们可以看看这个方法计算镜面反射分量的复杂度。它包含了若干点乘、向量的单位化和指数函数。这就自然地让我们去寻找一个简单一些的方法。

通过半角向量加速

通过等式(8.3)计算镜面反射的方法效率比较低，因为它首先需要计算出向量 **r**，并且进行单位化。使用半角向量可以在一定程度上改进这个不足。在实际的应用中 **r** 用另外的一项代替从而加速计算，这个方法由 Blinn【Blinn77】首先提出。为了避免使用 **r** 和 **v** 夹角的余弦值，需要首先计算出 **s** 和 **v** 的半角向量 $\mathbf{h}=\mathbf{s}+\mathbf{v}$，如图 8.14 所示。如果表面法向是沿 **h** 方向，则观察者能看到最强的反射光。因此 **m** 和 **h** 的夹角 β 可以用来表示观察者看到的反射光强度的衰减。β 和 ϕ 不相等(事实上如果共面的变化 **s** 和 **v**，β 可以是 ϕ 的两倍，见习题)，但是这个差别可以通过选取不同的指数 f 来弥补。(无论如何，这个镜面反射项不是基于物理原理，所以这种通过调整 f 产生满意结果的做法是可以接受的)因而在一般的应用中镜面反射都使用基于 $\cos(\beta)$ 的方法，即使用 **h** 和 **m** 的点积：

$$I_{sp} = I_s\rho_s \max\left(0, \left(\frac{\mathbf{h}}{|\mathbf{h}|}\cdot\frac{\mathbf{m}}{|\mathbf{m}|}\right)^f\right) \quad \text{\{调整后的镜面反射分量\}} \tag{8.4}$$

值得注意的是通过这个变化就不需要计算反射向量 **r**，从而节省了计算时间。另外，如果光源和视点都非常远，则 **s** 和 **v** 对于物体上不同的表面都是不变的，所以(对于每个面)只需计算一次 $\mathbf{h}\cdot\mathbf{m}$ 就可以了。

图 8.15 比较了一个球体反射不同强度的镜面反射光。从上到下镜面反射系数 ρ_s 依次是 0.25、0.50 和 0.75，从左到右指数 f 依次是 3、6、9、25 和 200(对于所有的球体环境光和漫反射系数都分别是 0.1 和 0.4。光源在所有的例子中都是位于同样的位置)。

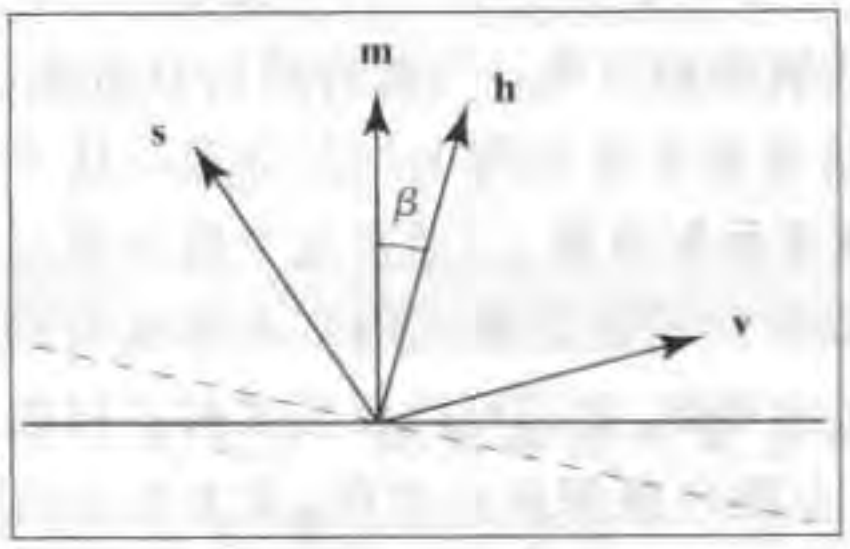


图 8.14　半角向量

图 8.15　光泽表面的镜面反射

事实上，镜面反射的物理机制要比 Phong 模型的描述复杂得多的多。一个更加真实的模型会假定镜面反射系数依赖于入射光波长 λ(比如颜色)和入射角度 θ_i(图 8.10 中向量 **s** 和 **m** 的夹角)。这种模型试图描述光如何从不同的材质(比如锌和锡铅合金)中反射，它的效果将在介绍等式(8.9)时做简要的讨论。我们同样会在第 12 章探讨彩虹和三棱镜的原理时解决这些问题。

练习

练习 8.2.1　半角向量

根据图 8.14 中的几何关系，证明当相关向量共面时，可能出现 $\beta=2\phi$ 的情况。

练习 8.2.2 一种镜面反射加速

Schlick【Schlick94】计算镜面反射项的另外一种方法替代指数函数的使用。令 D 表示等式(8.3)中的点积 $\mathbf{r}\cdot\mathbf{v}/|\mathbf{r}||\mathbf{v}|$。Schlick 提出用下式代替 D^f：

$$\frac{D}{f-fD+D}$$

这个式子的计算速度更快。画出在不同的 f 值下，比较两个函数在 D 在区间[0,1]上的函数图像。另外特别要注意 D 在 1 附近时的情况，因为这是反射光最强的区域。

8.2.4 环境光的作用和人类的视觉感知

通过简化现实中光线和物理表面的反射过程，我们得到了漫反射和镜面反射分量的计算方法。与简单的在线框上填充颜色相比，这种依赖于视点位置、模型和光源位置计算颜色的方法极大地增强了图像的真实感。

为了追求一个简单的反射模型，我们远离了场景渲染的完美结果。比如我们得到的阴影，颜色显得过于深和生硬。为了柔化这些阴影，我们可以增加第三个光分量，称为环境光。

如果只考虑漫反射和镜面反射分量，物体表面任何背向点光源的阴影部分一点光都接收不到，所以只能画为黑色！但在现实的经验告诉我们不是这样。我们身边常见的场景都会接受一些间接光源的照射。这些光从环境中的光源发出后经过周围不同物体的多次反射，最后到达物体，比如透过窗户中射入的光、荧光灯发出的光，还有其他类似的。但是想用模型精确地描述这种光是非常吃力的，会大量增加计算量。

环境光源和环境反射

为了解决阴影完全黑色的问题，我们可以假设环境中均匀地分布着昏暗的背景光，称为**环境光**。这种环境光的光源并不处在某个位置，并且光在各个方向上均匀分布。这个光源用光强度 I_a 表示。给模型的每个面片赋予一个**环境光反射系数** ρ_a(通常这个值和漫反射系数 ρ_d 相等)，则可以简单的把 $I_a\rho_a$ 这项加上原来漫反射和镜面反射光用来表示从面片上 P 点到达视点的全部光。I_a 和 ρ_a 的值通常也是经过实验得到，我们可以反复实验不同的值选择看上去效果最好的。环境光太小会使得阴影看上去颜色很深而且生硬；太大会使得图像看上去像浸泡过一样——平淡缺少变化。

图 8.16 画出了在漫反射光的基础上添加不同强度的环境光后，球体模型的渲染效果。每个例子中漫反射光源强度和环境光源强度都是 1.0，漫反射系数都是 0.4。从左到右环境光反射系数依次是 0.0、0.1、0.3、0.5 和 0.7。只有添加了最恰当的环境光量，原来球体底部生硬的阴影才会被柔化从而看上去更加真实。另一方面过多的环境光会过于影响阴影效果。

图 8.16 环境光的效果

8.2.5　如何综合光的贡献

我们现在可以把漫反射、镜面反射和环境光三种光的贡献加起来得到从 P 点到达视点总的光量 I：

$$\begin{aligned} I &= \text{环境光} + \text{漫反射光} + \text{镜面反射光} \\ I &= I_a\rho_a + I_d\rho_d \times lambert + I_{sp}\rho_s \times phong^f \end{aligned} \tag{8.5}$$

其中：

$$lambert = \max\left(0, \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right) \quad phong = \max\left(0, \frac{\mathbf{h} \cdot \mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right) \tag{8.6}$$

I 的值与三种光源强度和反射系数有关，同时也和点 P、视点和光源的相对位置有关。这里我们给予漫反射光源强度和镜面反射光源强度不同的名称 I_d 和 I_{sp}，这是因为在 OpenGL 中你可以分别单独设置这两个值。这点我们在后面会看到，在实际使用中，这两个值通常相等。

为了更加深入地理解光强度 I 和 P 点位置的关系，请再次看一看图 8.10。I 是根据面片上不同位置的点 P 计算出来的。环境光分量在整个面片上没有变化；**m** 对于面片上的所有点 P 值都相同，但是 **s** 和 **v** 的方向却依赖于 P。（比如，$\mathbf{s}=S-P$，其中 S 是光源的位置。那 **v** 是如何依赖于 P 和视点的呢？）如果光源的位置相对很远（常见的情况），**s** 因 P 的改变只会有很小的变化，所以漫反射分量对于不同的 P 点变化也会很小。特别是当 **s** 和 **m** 几乎平行的时候更是这样，因为当夹角很小的时候 cos() 的值变化很小。对于远处的光源，不同点 P 的半角向量 **h** 也是变化很小。另外一方面，如果光源里面片很近，不同的 P 点对应的 **s** 和 **h** 会有很大的变化。这时镜面反射项在整个面片上变化的非常明显，所以明亮的高光只出现在面片的一小部分上面。同样的效果也会出现在视点和面片距离很近的时候——因为引起了 **v** 方向的大范围变化，还有就是指数 f 值很大的时候也是这样。

练习

练习 8.2.3　视点距离对效果的影响

（基于直觉）描述 P 点在面片上变化时各种光的分量如何变化，(a) 当视点距离面片很远和 (b) 当视点离面片很近时各是怎样？

8.2.6　增加颜色

我们可以很直接地将这个渲染模型推广到有颜色的表面反射彩色光的情况。虽然这个推广很简单，但是最后的效果很好且很合理。

第 11 章有更多的关于颜色本质的背景和具体介绍，这里我们只需知道彩色光是由红、绿、蓝三种光通过不同比例混合组成的。处理彩色光源和彩色的物体表面的时候，我们单独的计算每个颜色分量，简单地将它们加在一起成为最后反射光的颜色。所以等式(8.5)应被使用三次：

$$
\begin{aligned}
I_r &= I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times lambert + I_{spr}\rho_{sr} \times phong^f \\
I_g &= I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times lambert + I_{spg}\rho_{sg} \times phong^f \\
I_b &= I_{ab}\rho_{ab} + I_{db}\rho_{db} \times lambert + I_{spb}\rho_{sb} \times phong^f
\end{aligned} \tag{8.7}
$$

(其中 $lambert$ 和 $phong^f$ 在等式(8.6)中给出)来计算反射光的红、绿、蓝分量。注意我们提到过光源有三种颜色类型：环境光=(I_{ar}，I_{ag}，I_{ab})，漫反射=(I_{dr}，I_{dg}，I_{db})和镜面反射=(I_{sr}，I_{sg}，I_{sb})。通常漫反射和镜面反射的光源颜色相同。另外还应该注意到 $lambert$ 和 $phong^f$ 项与要计算的颜色分量无关(除了通过反射系数来影响)，所以这些项只需要被计算一次。为了使用这个的模型，我们还需要定义 9 个反射系数：

环境光反射系数：ρ_{ar}，ρ_{ag} 和 ρ_{ab}

漫反射系数：ρ_{dr}，ρ_{dg} 和 ρ_{db}

镜面反射系数：ρ_{sr}，ρ_{sg} 和 ρ_{sb}

环境光和漫反射系数依赖于表面本身的颜色。这里"表面的颜色"指当其被白光照射时所反射的漫反射颜色：一个红色的表面在白光照射下就显示出红色。如果在其他一些颜色光的照射下，它可能会显出完全不同的颜色。下面的示例说明了这一点：

例 8.2.1 物体的颜色

如果我们说一个球体的颜色是由 30%的红色、45%的绿色和 25%的蓝色组成的。那么分别设置环境光和漫反射系数为(0.3K，0.45K，0.25K)是合理的，其中 K 是一个比例因子，用来决定球体反射入射光的整体比例。如果这个球体放在红、绿、蓝比例相等($I_r = I_g = I_b = I$)的白光中，则漫反射光强度的三个分量是 $I_r = 0.3KI$，$I_g = 0.45KI$，$I_b = 0.24KI$，所以正和期望的一样，我们可以看到它的颜色是 30%红、45%绿和 25%蓝。

例 8.2.2 一个亮红色物体置于亮绿色光中

假设一个球体的环境光和漫反射系数是(0.8，0.2，0.1)，它在白光中看上去几乎是红色。我们使用一个亮绿色光源 I_s=(0.15，0.7，0.15)，则反射光的颜色是(0.12，0.14，0.015)。这是一个红绿分量几乎相等的混合，最终呈现的颜色浅黄色(我们将在 11 章讨论)。

镜面反射光颜色 因为镜面反射光更像镜子，即反射光的颜色通常和光源颜色相同。例如，平常的经验告诉我们在黄色光中，一个红色的苹果高光部分的颜色是黄色而不是红色。同样的情况也可以在塑料材质的物体上观察到。为了制造塑料表面的镜面高光，等式(8.7)中的镜面反射系数 ρ_{sr}，ρ_{sg} 和 ρ_{sb} 同城都被设置成一样的值 ρ_s，所以反射系数本身都是"灰度"值，不会改变入射光的颜色。设计者可以为一个不太光泽的塑料表面选择 ρ_s=0.5，或者为一个非常有光泽的表面选择 ρ_s=0.9。

不同材质制造的物体

通过精心地选择反射系数，可以使物体呈现特定材质的效果，比如铜、金或者锡铅合金，至少能在效果上做到比较接近。McReynolds 和 Blythe【McReynolds97】建议使用图 8.17 中列出的反射系数。图 8.18 中的球体就是使用了这些系数渲染后的结果(在配套的网站上，有一个彩色版本，其中的球体相当真实地还原了相应材质的外观)。值得注意的是镜面反射系数拥有不相等的红、绿、蓝分量，所以镜面反射光不是简单地和入射光一样。但是

McReynolds 和 Blythe 提醒使用者注意，由于 OpenGL 的渲染算法结合了 Phong 镜面反射分量，所以最终的视觉效果并不是完全的真实。我们将在第 12 章里回顾这个问题并且介绍更加真实的 Cook-Torrance 渲染方法。

材料	环境：ρ_{ar}、ρ_{ag}、ρ_{ab}	扩散：ρ_{dr}、ρ_{dg}、ρ_{db}	反射：ρ_{sr}、ρ_{sg}、ρ_{sb}	指数：f
黑塑料	0.0 0.0 0.0	0.01 0.01 0.01	0.50 0.50 0.50	32
黄铜	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974
青铜	0.2125 0.1275 0.054	0.714 0.4284 0.18144	0.393548 0.271906 0.166721	25.6
铬	0.25 0.25 0.25	0.4 0.4 0.4	0.774597 0.774597 0.774597	76.8
铜	0.19125 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.17622 0.06014	12.8
金	0.24725 0.1995 0.0745	0.75164 060648 022648	0.628281 0.555802 0.366065	51.2
白腊	0.10588 0.058834 0.113725	0.427451 0.470588 0.541176	0.3333 0.3333 0.521569	9.84615
银	0.19225 0.19225 0.19225	0.50754 0.50754 0.50754	0.508273 0.508273 0.508273	51.2
抛光银	0.23125 0.23125 0.23125	0.2775 0.2775 0.2775	0.773911 0.773911 0.773911	89.6

图 8.17　常见材质的参数

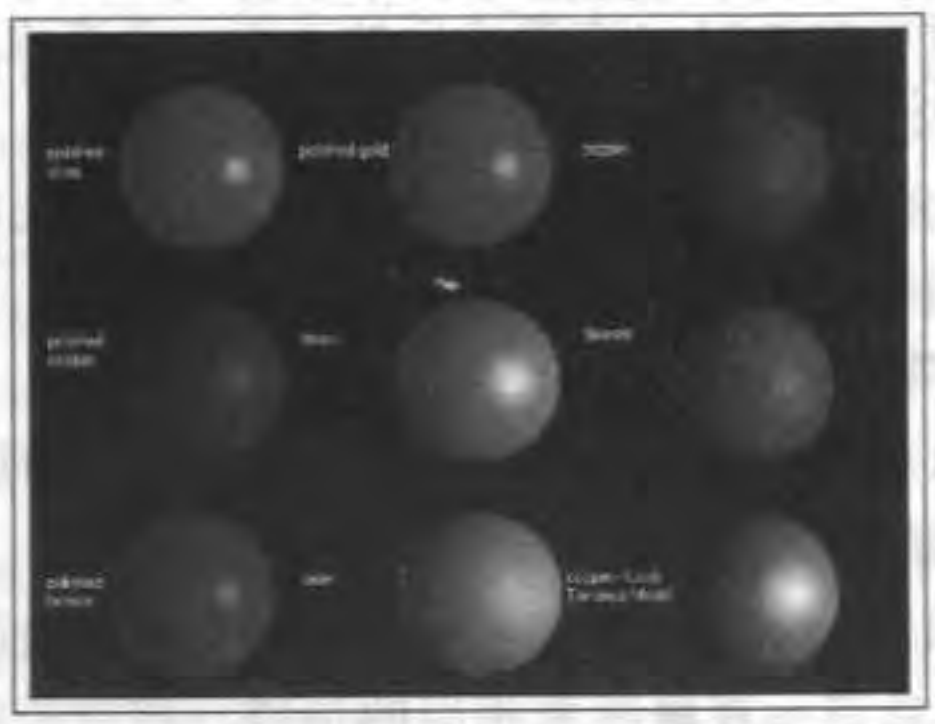


图 8.18　不同材质的光泽球体

8.2.7 渲染和图形管线

着色位于图形管线的哪一步？又是如何完成的？图 8.19 再次给出了流水线。其中关键的一点是网格的顶点与其相应的顶点法向在顺着流水线前进，并且所有的着色计算都是在顶点上完成的。(回顾 Mesh 类中的 draw()方法传递每个顶点与其对应的法向给 OpenGL，见图 6.15。)

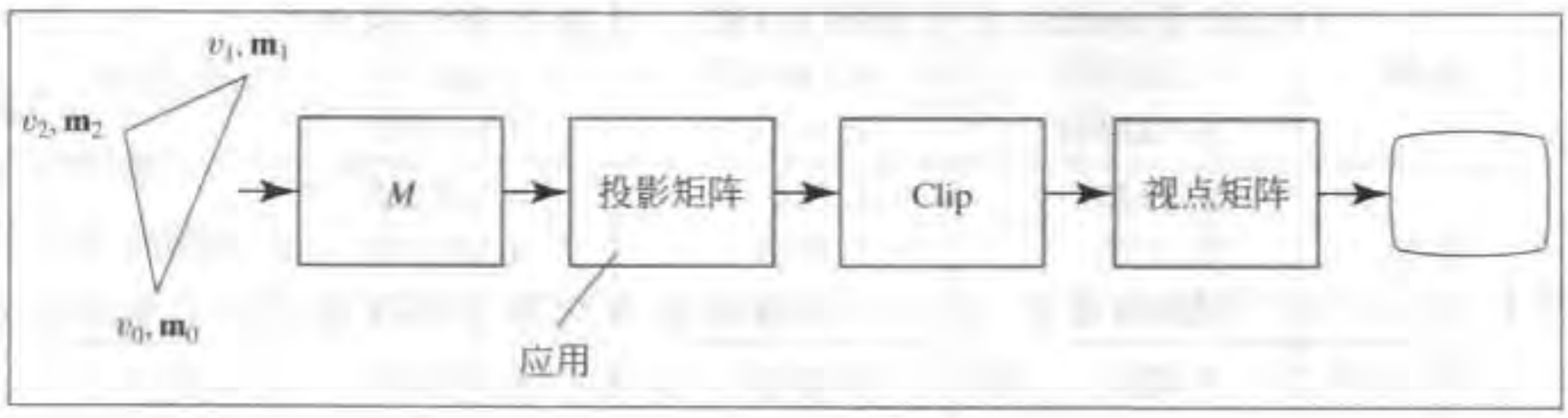


图 8.19 修正后的图形管线

图中画出了一个顶点是 v_0, v_1, v_2 的三角形被渲染的过程。顶点 $\mathbf{v}_i$ 对应的法向是 $\mathbf{m}_i$。这些量通过如下调用传送给流水线：

```
glBegin(GLPOLYGON);
    for(int i=0; i<3; i++)
    {
      glNormal3f(m[i].x, m[i].y, m[i].z);
      glVertex3f(v[i].x, v[i].y, v[i].z);
    }
glEnd();
```

调用 glNormal3f()设置了“当前法向量”，这个法向被应用于随后的通过 glVertex3f()设置的全部顶点。当前法向量直到另外一个 glNormal3f()调用的时候才会改变。所以上面的代码中每一个顶点都被设置了一个不同法向。

顶点通过模型视点矩阵(modelview matrix)$\boldsymbol{M}$ 变换到相机(视点)坐标系。法向量也会变换，但是向量的变换和顶点的变换不同。回顾在 6.5.3 节的介绍，通过矩阵 $\boldsymbol{M}$ 变换表面上的顶点，原来表面上点对应的法向 $\mathbf{m}$ 在变换后的表面上会变成 $\boldsymbol{M}^{-T}\mathbf{m}$，其中 $\boldsymbol{M}^{-T}$ 是 $\boldsymbol{M}$ 的逆矩阵再转置。OpenGL 会自动而正确地处理法向量。

在后面的小节中我们会看到，OpenGL 允许你设定不同的光源和它们的位置。光源也是物体，并且光源的位置同样会被模型视点矩阵所变换。

所有通过模型视点矩阵变换后的量都是处于相机坐标系下。此时应用等式(8.7)表达的渲染模型，这样每个顶点就会被添加上了颜色。计算颜色要知道向量 $\mathbf{m}$、$\mathbf{s}$ 和 $\mathbf{v}$，这些量在流水线处理的这个时刻都是已知的(请相信这一点)。

在随后的处理流水线上，伪深度值被计算出来，顶点也会通过投影矩阵(projection matrix)进行变换。颜色信息则一直跟随着相应的顶点。随后的剪裁过程是在齐次坐标系下完成的，这步可能会改变一些顶点。图 8.20 就给出了这样一个例子：三角形的顶点 v_1 被

剪裁掉，增加了两个新的顶点 a 和 b，三角形变成了四边形。新顶点的颜色也必须要计算，因为这是渲染时需要的。

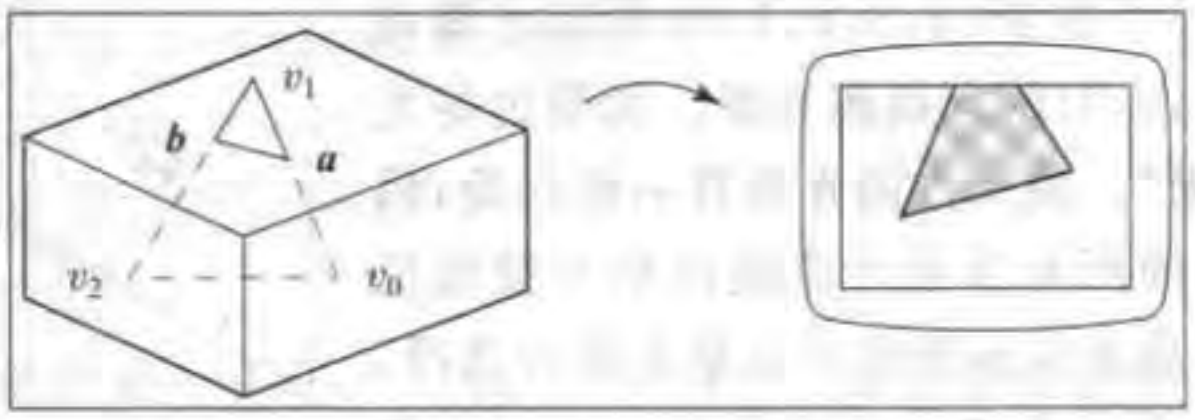


图 8.20　用视锥剪裁一个多边形

新顶点的颜色通常经过插值计算得到，就像在 Gouraud 明暗处理时做的一样。例如，假设 v_0 的颜色是(r_0, g_0, b_0)，v_1 的颜色是(r_1, g_1, b_1)。如果点 a 位于从 v_0 到 v_1 的 40%的位置，则 a 的颜色是(r_0, g_0, b_0)的 60%和(r_1, g_1, b_1)的 40%的混合。这可以用等式表示：

$$a\text{ 点颜色} = (\mathrm{lerp}(r_0, r_1, 0.4), \mathrm{lerp}(g_0, g_1, 0.4), \mathrm{lerp}(b_0, b_1, 0.4)) \tag{8.8}$$

其中我们用使用了函数 lerp()(线性插值 liner interpolation 的简称)，这个函数在第 4 章介绍动画插值帧的时候也用到过。

定义为：

$$\mathrm{lerp}(G, H, f) = G + (H - G)f \tag{8.9}$$

它的值等于 G 和 H[1] 以 f 为比例插值的结果。

顶点最后通过视窗变换(view port transformation)映射到屏幕坐标(screen coordinates)(同时附带着 0 到 1 之间的伪深度值)。这个四边形最终被渲染出来(在隐藏面消隐之后)，如图 8.20 所示。我们将要对实际的渲染步骤做更多的介绍。

8.2.8　在 OpenGL 中使用光源

OpenGL 提供了许多函数用来设置光源和设置物体表面材质属性。理解并掌握其中所有的变化和细节会比较乏味，所以我们在这只介绍基本的用法。我们会讨论如何在场景中设置不同类型的光源。下一节中我们再看一看设置物体表面反射属性的方法。

创建光源

OpenGL 允许你定义八个光源，依次用 GL_LIGHT0、GL_LIGHT1 等命名。每个光源可以设置不同的属性，也可以被开启或关闭。每个属性都有默认的赋值。例如，建立一个位置在世界坐标系下(3,6,5)的光源，可以使用[2]：

```
GLfloat myLightPosition[]={3.0, 6.0, 5.0, 1.0};
glLightfv(GL_LIGHTO, GL_POSITION, myLightPosition);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHTO);
```

数组 myLightPosition[](这个数组也可以使用其他你喜欢的名字)指定了光源的位置。这个位置连同光源名 GL_LIGHT0 一起传递给 glLightfv()来设置光源 GL_LIGHT0。

一些光源，比如桌上的台灯，位于场景中，相比而言另外一些光源，比如太阳，处于无限远的位置。OpenGL 允许你通过使用齐次坐标设置这两种类型光源的位置：

$(x,y,z,1)$：位置在(x,y,z)的局部光源

$(x,y,z,0)$：位于(x,y,z)方向上无限远的光源

图 8.21 画出了一个位于(0,3,3,1)的局部光源和一个"位于"方向(3,3,0,0)的远距离光源。无限远的光源也经常称为"**方向光**"。使用方向光源有一些好处，因为方向 s 对于场景中的所有顶点计算漫反射和镜面反射时都是不变的。但是方向光源也不总是正确的选择：一些视觉效果只有在使用靠近物体的局部光源时才能实现。

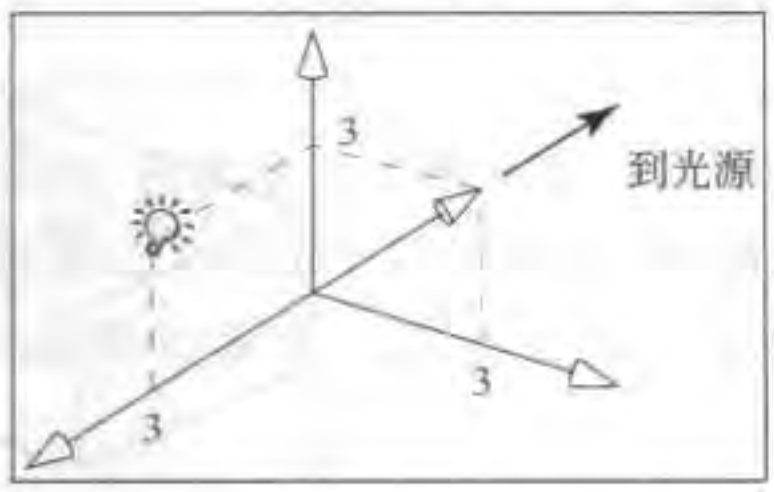


图 8.21 局部光源和无限远处的光源

你也可以非常直观地设置不同颜色的光源。OpenGL 允许你为一个光源发出的三种不同光——环境光、漫反射和镜面反射，设置不同的颜色。一个光源发出环境光听上去可能有些奇怪。它只是根据等式(8.7)来处理：一个整个场景中全局的全方位的光源。将它和一个光源联系在一起的好处是这样可以在应用过程中方便地打开和关闭它。(OpenGL 也提供一个真正的环境光，不与任何光源绑定在一起，我们将在后面有关"光照模型"中介绍。)

光源的颜色使用数组来储存，然后传递给 glLightfv()。

```
GLfloat amb0[ ]={0.2, 0.4, 0.6, 1.0};          //define some colors
GLfloat diff0[ ]={0.8, 0.9, 0.5, 1.0};
GLfloat spec0[ ]={1.0, 0.8, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, amb0);        //attach them to LIGHT0
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

颜色使用所谓的 **RGBA** 格式，也就是红、绿、蓝和"透明度(alpha)"。透明度只当在屏幕上混合两种颜色的时候使用。我们在第 9 章讨论这个。对于我们现在的应用它通常设为 1.0。

光源不同的参数都有默认的数值。对于所有的光源：

默认环境光=(0,0,0,1)；　←可能最昏暗：黑色

对于光源 LIGHT0：

默认漫反射=(1,1,1,1)；　←可能最亮：白色

默认镜面反射=(1,1,1,1)；　←可能最亮：白色

同时对于其他光源的漫反射和镜面反射默认都是黑色。

探照灯

光源默认都是点光源，即发出的光均匀地射向各个方向。但是 OpenGL 允许你把它们设置为探照灯，这样它们发出的光就集中在一些特定的方向。图 8.22 画出了一个朝向方向 **d**，"切角"为 α 的探照灯。

在圆锥之外的点无法看到光。对于圆锥之内的顶点 P，到达 P 点的光强以因子 $\cos^{\epsilon}(\beta)$ 衰减，其中 β 是 **d** 与光源和 P 点连线的夹角，epslon 是用户指定的光源根据角度衰减的指数。

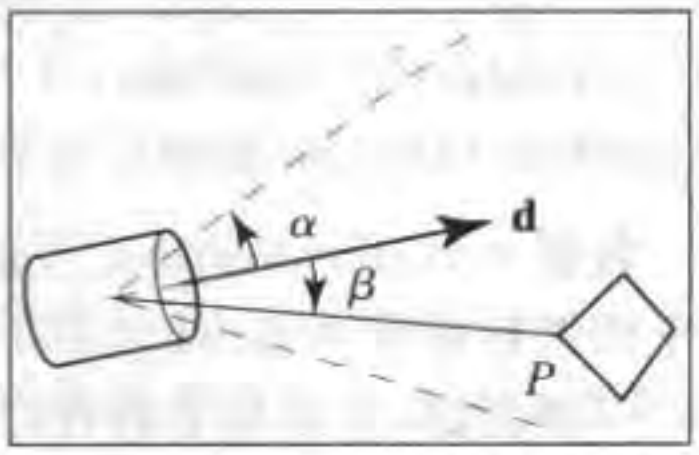


图 8.22 OpenGL 探照灯的属性

设置探照灯的参数，使用 glLightf()设置单个值，使用 glLightfv()设置一个向量：

```
glLight(GL_LIGHTO, GL_SPOT_CUTOFF, 45.0);
glLight(GL_LIGHTO, GL_SPOT_EXPONENT, 4.0);
Glfloat dir[]={2.0, 1.0, - 4.0};
glLight(GL_LIGHTO, GL_SPOT_DIRECTION, dir);
direction vector, dir;
```

这些参数的默认值是 $\mathbf{d}=(0,0,-1)$，$\alpha=180^\circ$，$\varepsilon=0$，这些参数使得光源是一个全方向的点光源。

光随距离的衰减

OpenGL 还允许你设置光随光源发出的距离以多快的速度衰减。虽然我们已经说过这种依赖关系不是很重要，但通过实验不同的衰减速率来仔细地调整图像也还是有趣的。OpenGL 中位置光源的强度衰减是根据如下衰减因子：

$$atten=\frac{1}{k_c+k_lD+k_qD^2} \tag{8.10}$$

其中 k_c、k_l 和 k_q 是系数，D 是光源到顶点的距离。这个表达式非常通用，它使你能够实现关于光源距离的常数、线性和二次函数(反平方定律?)。这些参数被如下函数调用控制：

```
glLightf(GL_LGIHT0, GL_CONSTANT_ATTENUATION, 2.0);
```

同样对于 GL_LINEAR_ATTENUATION 和 GL_QUADRATIC_ATTENUATION。参数的默认值是 $k_c=1$、$k_l=0$ 和 $k_q=0$，也就是默认没有衰减。

光照模型

OpenGL 通过设置三个参数来指定光照模型的一般规则。这些参数传递给不同版本的 glLightModel 函数。

a. 全局环境光颜色

你可以在场景中建立一个全局环境光源，它独立于任何一个普通的光源。建立这个光源，制定它的颜色可以使用：

```
GLfloat amb[]={0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
```

这样就设置了漫反射光源的颜色为(0.2,0.3,0.1)。默认的值是(0.2,0.2,0.2,1.0)，所以这个漫反射光源始终存在，除非你有意改变它。这使得即使你没有调用任何光照函数，场景中的物体也可以被看见。

b. 视点是局部还是无限远?

在第 8.2.3 节我们介绍过 OpenGL 计算镜面反射光使用“半角向量”

$$\mathbf{h}=\mathbf{s}+\mathbf{v} \tag{8.11}$$

对于网格上的不同顶点对应的 **s** 和 **v** 真实的方向一般不同(画图说明这点)。如果光源是方向光，则 **s** 是常数，但是 **v** 对于不同的顶点仍然变化。如果 **v** 对于所有的顶点也是常

数,则渲染速度能够增加。在默认的情况下,OpenGL 使用 v=(0,0,1),这是在相机坐标系下 z 轴的正方向。你可以强制渲染管线使用对于每个顶点真实的 v 值:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

c. 多边形的两面是否都适当地着色?

模型中的每个多边形面片都有两个面。建模的时候,我们一般把它们想象成"内表面"和"外表面"。列出面片各个顶点的排列顺序时,惯例上是按照从物体外面观察的逆时针方向。大多数的网格模型表示的物体都是封闭空间的固体,这就能很好地定义里面和外面。对于这样的物体,相机只能够看到它们外表面(假定相机不会放置在物体内部)。在适当进行隐藏面消隐之后,从视点处看每个面片的内表面肯定会被更近的面片遮挡住。

OpenGL 中没有里面和外面的概念。它只能区分"正面"和"背面"。一个面是**正面**,如果它的顶点从视点看过去是按照逆时针顺序(CCW)列出的。图 8.23(a)画出了从视点观察到的一个立方体,我们假定建模使用了逆时针顺序的惯例。箭头表明了每个面片的顶点传递给 OpenGL 时的顺序(在 glBegin(GL_POLYGON);…;glEnd()之间)。对于一个空间封闭的物体,所有的可见面片因此都是正面。同时 OpenGL 会适当而正确地画出它们。OpenGL 也会画出背向面片,但是它们最终被一些更近的正面所遮挡。

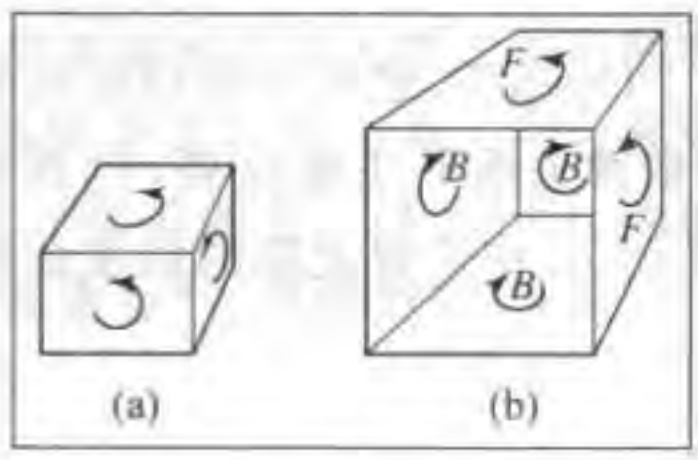


图 8.23 OpenGL 中正面的定义

在图 8.23(b)中情况就不一样了,盒子的一个面被移去。箭头依旧列出了各个面片的顶点传递给流水线时的顺序。这次,三个可见的面片是背面。在默认的情况下,OpenGL 不会对这些面片适当地着色。为了强制 OpenGL 适当地对背面着色,使用:

```
glLightModeli(GL_LIGHT_MDOEL_TWO_SIDE, GL_TRUE);
```

这样 OpenGL 会对所有背面的法向取反,这样它们就指向了观察者,并且可以适当地计算颜色。用 GL_FALSE(默认值)替换 GL_TRUE 就可以关闭这个功能。

注意,OpenGL 画的面片不会投阴影,所以背面会收到光源同样的光,即使可能有其他的面片位于它和光源之间。

光源的移动

回顾光源通过模型视点矩阵的变换和普通的顶点是一样的。因此光源位置也可以通过适当地使用 glRotated()和 glTranslated()重置。数组 position 通过调用 glLightfv(GL_LIGHT0, GL_POSITION, position)设置的位置会被模型视点矩阵改变,这个效果是在 glLightfv()调用的时候完成的。所以为了通过变换改变光源位置,并且还要独立于相机的运动,要把光源位置变换的命令置于一对压栈/退栈之间,例如:

```
void display()
{  GLfloat position[ ]={2, 1, 3, 1};                    //initial light position
    <..clear color and depth buffers..>
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
```

```
    glPushMatrix();
    glRotated(...);     glTranslated(...);                //move the light
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPopMatrix();
    gluLookAt(...);                                       //set the camera position
    <..draw the object..>
    glutSwapBuffers();
}
```

另外一方面，为了使光源随着相机一起移动，使用：

```
GLfloat pos[ ]={0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, position);          //light at (0,0,0)
gluLookAt(...);                                       //move light and camera
<..draw the object..>
```

这样建立的光源放置在视点的位置(就像矿工安全帽上的灯)，光源会随着相机移动。

8.2.9　在 OpenGL 中使用材质属性

你只能通过光从物体表面反射看到光源的效果。OpenGL 提供了设定在等式(8.7)中出现的各种反射系数的方法。这些参数通过调用函数 glMaterial 的不同版本进行设置，并且还可以单独地对正面和背面的参数进行设置(见关于图 8.23 的讨论)。例如，

```
GLfloat myDiffuse[ ]={0.8, 0.2, 0.0, 1.0};
glMaterialfv(GL_FRONT, GL_DIFFUSE, myDiffuse);
```

为随后所有正面设置了漫反射系数($\rho_{dr}, \rho_{dg}, \rho_{db}$)=(0.8,0.2,0.0)。反射系数通过 RGBA 格式的四元组设置，这点和颜色一样。第一个传递给 glMaterialfv()的参数可以选择：

- GL_FRONT：为正面设置反射系数。
- GL_BACK：为背面设置反射系数。
- GL_FRONT_AND_BACK：为两面同时设置反射系数。

第二个参数可以选择：

- GL_AMBIENT：设置环境光反射系数。
- GL_DIFFUSE：设置漫反射系数。
- GL_SPECULAR：设置镜面反射系数。
- GL_AMBIENT_AND_DIFFUSE：设置环境光和漫反射系数为相同的值。这个参数方便了一般的使用，因为环境光和漫反射系数通常都会设置成一样的值。
- GL_EMISSION：设置表面的辐射颜色。

最后的一个参数是设置面片的**辐射颜色**(**emissive color**)，就是面片发出“辉光”的颜色，它独立于所有的光源。

把所有的放在一起

我们现在扩展等式(8.7)把 OpenGL 实际计算光的所有附加贡献添加进来。全部的红色分量这样计算：

$$I_r = e_r + I_{ar}\rho_{ar} + \sum_i atten_i \times spot_i \times (I_{ar}^i\rho_{ar} + I_{dr}^i\rho_{dr} \times lambert_i + I_{spr}^i\rho_{sr} \times phong_i^f) \tag{8.12}$$

对于绿色和蓝色分量的表达式类似。辐射光是 e_r，I_{ar} 是全局环境光，光照模型中介绍过它们。求和式说明所有光源的全局光、漫反射和镜面反射的贡献最后是要相加在一起。对于第 i 个光源，$atten_i$ 是等式(8.10)中的衰减因子，$spot_i$ 是探照灯因子(见图 8.22)，$lambert_i$ 和 $phong_i$ 是我们熟悉的漫反射和镜面反射的点积。所有这些项对于每个光源都要被重新计算。

注意：如果 I_r 出现了大于 1.0 的值，*OpenGL* 会把它剪裁到 1.0：所有最亮的光分量只能是 1.0。

8.2.10 渲染 SDL 设定的场景

场景描述语言(scene description language，SDL)在第 5 章和附录 3 做过介绍，它支持读取某个材质属性到物体，这样物体就能够被适当地着色。例如，

```
Light 3 4 5 0.8 0.8 0.8 ! bright white light at (3, 4, 5)
background 1 1 1 ! white background
globalAmbient 0.2 0.2 0.2 ! a dark gray global ambient light
ambient 0.2 0.6 0
diffuse 0.8 0.2 0.1 ! red material
specular 1 1 1 ! bright specular spots-the color of the sourse
exponent 20 ! set the Phong exponent
scale 4 4 4 sphere
```

描述了一个包含球体的场景，球体的材质属性是(见等式(8.7))：

- 环境光反射系数：$(\rho_{ar}, \rho_{ag}, \rho_{ab}) = (0.2, 0.6, 0)$
- 漫反射系数：$(\rho_{dr}, \rho_{dg}, \rho_{db}) = (0.8, 0.2, 1.0)$
- 镜面反射系数：$(\rho_{sr}, \rho_{sg}, \rho_{sb}) = (1.0, 1.0, 1.0)$
- 和 *Phong* 模型指数：$f = 20$

光源的漫反射和镜面反射分量的颜色都是(0.8，0.8，0.8)。还有全局环境光(I_{ar}，I_{ag}，I_{ab})=(0.2，0.2，0.2)。

当物体被创建时，当前材质属性被读入到物体的 mtrl 域(参见附录 3 和网上的源代码 Shape.cpp 中 Scene::getObject()的结尾部分)。当一个物体通过自身的 drawOpenGL()方法画其自己的时候，它首先将自己的材质属性传递给 OpenGL(参见 Shape::tellMaterialsGL()),所以当它真的被进行绘制的时候，OpenGL 的当前状态就是这些属性值。

我们在第 12 章介绍光线跟踪的时候，也会类似地使用每个物体的材质数据，查询这些

材质属性并且做适当的渲染。

8.3　平面着色与平滑着色

不同的物体需要不同的着色效果。在第 6 章中我们使用多边形网格来为许多不同的形状建模。对于其中的一些，像谷仓或者多面球体，我们希望在图像中看到每个单独的面片。但是对于另外的一些，像球体或者国际象棋中的卒，我们希望看到用面片逼近的表面，同时不想看到其中的边。

在建模过程中，对于每个面片的每个顶点我们都附加上了法向量。重复一下其中的要点，如果某个面片被看成一个独立的多边形，我们给它所有的顶点都附加上了同样的法向量。该法向量就是该面片所在平面的法方向。另外一方面，如果面片是用来拟合一个潜在的曲面，我们为每个顶点附加被拟合曲面在该点处的法向。

我们现在看看每个顶点上的法向量信息是如何在各种不同的渲染中被使用的。其中最主要的区别是一个方法强调单个的多边形（平面着色），另一个方法通过融合面片来降低边的突出程度（平滑着色）。平滑着色又有两种方法，称为 Gouraud 和 Phong 明暗处理。我们两种方法都要讨论。注意一些作者更喜欢称其为“照明模型（illumination model）”而不是“着色模型（shading model）”。

对于两种着色方法，顶点都是通过图形管线和光照的计算为每个顶点附加上颜色信息，最终面片顶点转化为屏幕坐标，并且面片被逐像素地绘制出适当的颜色。

画一个面片

面片是在多边形填充过程中被着色的。填充一个多边形非常简单，虽然一个精心设计的高效率填充算法可能会非常复杂（见第 9 章）。这里我们只看基本的内容，侧重于每个像素的颜色是如何设定的。

多边形填充程序有时也被称为**填充器（tiler）**，因为它在多边形上一个像素一个像素地移动，同时为每个像素着上适当的颜色，就像人在地板上铺瓷砖一样。具体地讲，多边形内的像素是按照规定顺序被访问的，通常是多边形底部到顶部的扫描线顺序，每个扫描线内是从左到右。

我们假定这里所处理的多边形是凸多边形。只处理凸多边形的情况下，填充算法可以设计得很高效，因为每个扫描线上只有一段连续的像素在多边形中。大多数 OpenGL 的实现都利用了这一点，可以正确地填充凸多边形，但是并不保证对于非凸多边形也能正确地填充。参见练习中有关多边形凸性有关的内容。

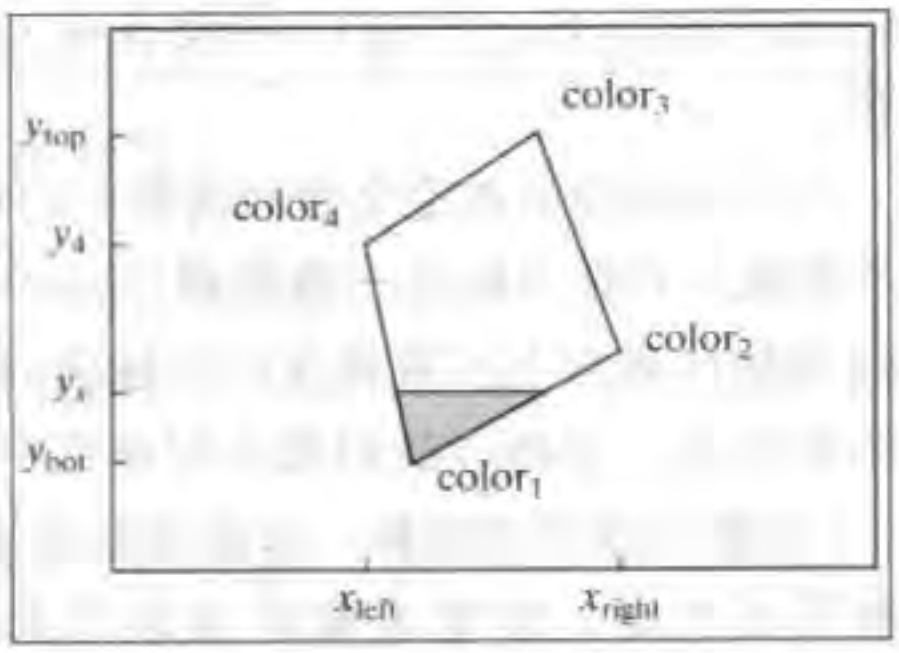


图 8.24　带颜色地填充多边形面片

图 8.24 是一个四边形面片的例子。每个顶点的屏幕坐标也标注了出来。面片的最低和最高点分别是 y_{bot} 和 y_{top}。填充过程先是填充 $y=y_{bot}$ 这行（这种情况下只是一个像素），然后是 $y_{bot}+1$ 行，以

此类推。在处理每个扫描线时,比如在图中的 y_s 这行,这有一个最左边的像素 x_{left} 和最右边的像素 x_{right}。填充算法从左 x_{left} 到 x_{right} 的移动把每个像素应有的颜色画上。所以填充算法的实现就是一个简单的双重循环,伪代码大致如下:

```
for (int y=ybott; y<=ytop; y++)          //for each scan-line
{
    <..find xleft and xright..>
  for (int x=xleft; x <=xright; x++)
      //for each relevant  pixel across this scan-line
    {
        <..find the color c for this pixel..>
        <..put c into the pixel at (x, y)..>
  }
}
```

在双重循环中隐藏面消隐也可以很容易地实现。平面着色和平滑着色的主要区别是每个像素颜色 c 的决定方法。

8.3.1 平面着色和马赫带

当面片是平的(比如谷仓的屋顶)并且光源相距很远的时候,屋顶上不同点的漫反射分量变化很小。等式(8.6)中的 *lambert* 项在面片的各个点上几乎相等,因为从每个点指向光源的向量几乎一样。在这种情况下为凸面片上的每个像素点都使用相同的颜色是合理的。OpenGL 提供了一种渲染模式可以为整个的面片渲染相同的颜色。虽然面片的每个顶点都会在流水线中携带一个颜色,但是绘制算法只会使用其中的一个(通常是面片的第一个顶点的颜色)。所以伪代码中的命令,<为这个像素计算颜色 c>,并不是在双重循环的内部,而是位于内重循环之前(译者注:这里应该是位于双重循环的外面),还要有"把 c 设置成某一个顶点的颜色"。(因为对每个像素使用同样的颜色,所以使得平面着色非常的快)。

OpenGL 中指定平面着色要使用:

```
glShadeModel(GL_FLAT);
```

图 8.25 就是使用平面着色渲染出的多面球体和球体。多面球体上每个单独的面片都像希望的那样清晰可见。而与此相反地,球体应该是一个平滑的物体,但是使用 glShadeModel(GL_FLAT)并没有使其平滑,因为整个面片都只用其中一个顶点的颜色着色。

有一种还没有被完全解释清楚的幻觉,使得面片之间的边看上去比其实际情况更加明显,这种现象在视觉中称为**一侧抑制**(lateral inhibition)。Ernst Mach 首先描述了这种现象:当光强在物体表面上出现梯度的不连续,视觉就会在梯度不连续的地方产生**马赫带**,人就能看到清晰的边。这种边的幻觉不用很集中精神就能被看到。

(观察马赫带的实验:选择多面球体的一个面片和一个边,要求其左边相邻面片比其右边相邻面片暗。你能否看到靠近边并且与边平行的窄带,左边的这个颜色——很奇怪地——要比右边的深?这可能不太容易看出来)这个效应夸大了使用平面着色的渲染结果,

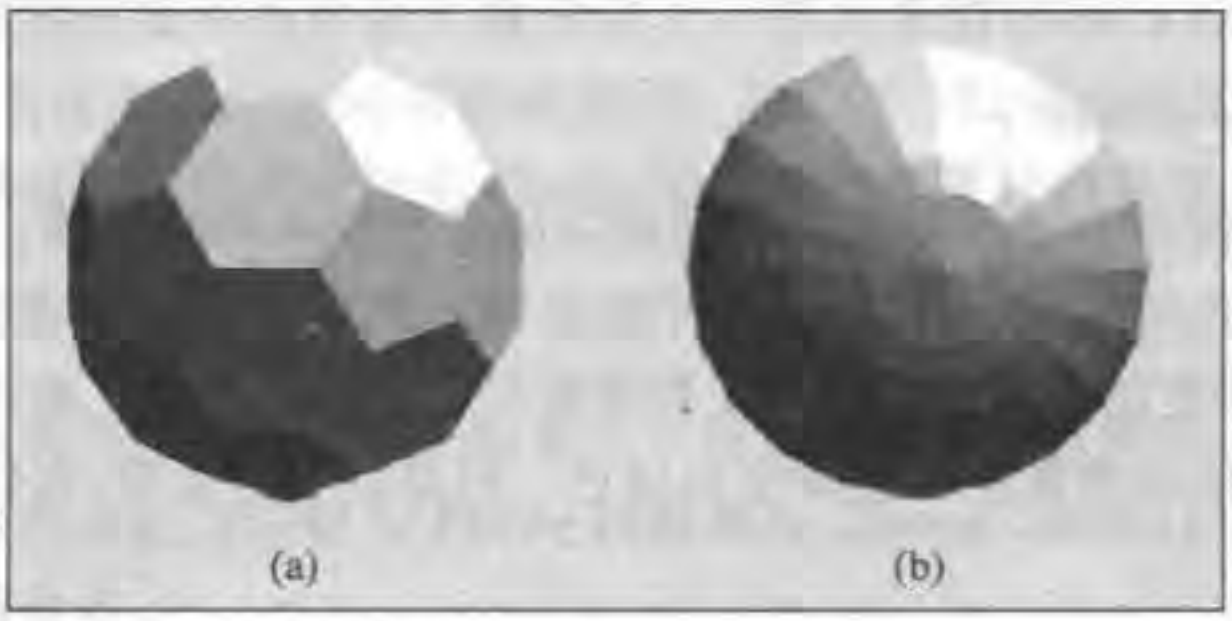


图 8.25　使用平面着色渲染的两个网格
(a) 多面球体；(b) 球体

使其看上去更加像网格模型。马赫带在图 b 中的球体上也能被很清楚地看到。

使用平面着色镜面高光的渲染结果就更差了，还是因为整个面片都是使用一个顶点计算出的颜色。如果恰好一个很强的镜面反射分量处于填充所选取的顶点上，则整个面片都会填充上这个亮度。如果镜面高光没有落在代表顶点上，那整个面片都没有高光。正是这个原因，一般不会在计算着色的时候考虑镜面反射分量。

8.3.2　平滑着色

平滑着色试图通过计算面片上多个点的颜色来弱化面片之间的边。这有两个主要的平滑着色方法，Gouraud 明暗处理和 Phong 明暗处理【Gouraud71，Phong75】。这些技术更准确地应该被称作“Gouraud 和 Phong 插值”，因为它们对于不同的点都是在混合颜色。OpenGL 只使用了 Gouraud 明暗处理，但是我们两种方法都会介绍。

Gouraud 明暗处理对于每个像素计算不同的颜色。对于在 y_s 处的扫描线（见图 8.24），首先通过左边最上方和最下方顶点的插值，找到最左边像素点的颜色 $color_{left}$。对于扫描线 y_s 来说，上方顶点的颜色是 $color_4$，下方顶点的颜色是 $color_3$，所以 $color_{left}$ 通过下式来计算（回顾等式(8.9)）：

$$color_{left} = \mathrm{lerp}(color_1, color_4, f) \tag{8.13}$$

其中 f 是 y_s 在 y_4 和 y_{bot} 之间距离所占的比例。

当 y_s 从 y_4 变化到 y_{bot}，f 值从 0 变到 1。注意等式(8.13)实际上包含三个计算对应红、绿和蓝三个分量。

类似地，再通过最上方和最下方的顶点颜色插值出 $color_{right}$。则填充过程就可以沿着扫描线，在 $color_{left}$ 和 $color_{right}$ 上线性插值得到像素 x 的颜色：

$$c(x) = \mathrm{lerp}\left(color_{left}, color_{right}, \frac{x - x_{left}}{x_{right} - x_{left}}\right) \tag{8.14}$$

为了提高效率，每个像素的颜色采用增量计算。也就是说 $c(x+1)$ 和 $c(x)$ 之间的差是一个常数，所以

$$c(x+1) = c(x) + \frac{color_{right} - color_{left}}{x_{right} - x_{left}} \tag{8.15}$$

Gouraud 明暗处理比平着色要耗费多一些的计算。OpenGL 中设置 Gouraud 明暗处理

使用：

```
glShadeModel(GL_SMOOTH);
```

图 8.26 展示了使用 Gouraud 明暗处理渲染的多面球体和圆球体。多面球体看上去和图 8.25 中用平面着色的结果一样，因为每个面片顶点的颜色都一样，所以插值不做任何改变。但是球体看上去更加平滑了。颜色在相邻面片之间没有突变。面片的边(还有马赫带)都不见了，取而代之的是物体上平滑渐变的颜色。但是沿着外轮廓，你仍然能够看到一些面片的外边缘。

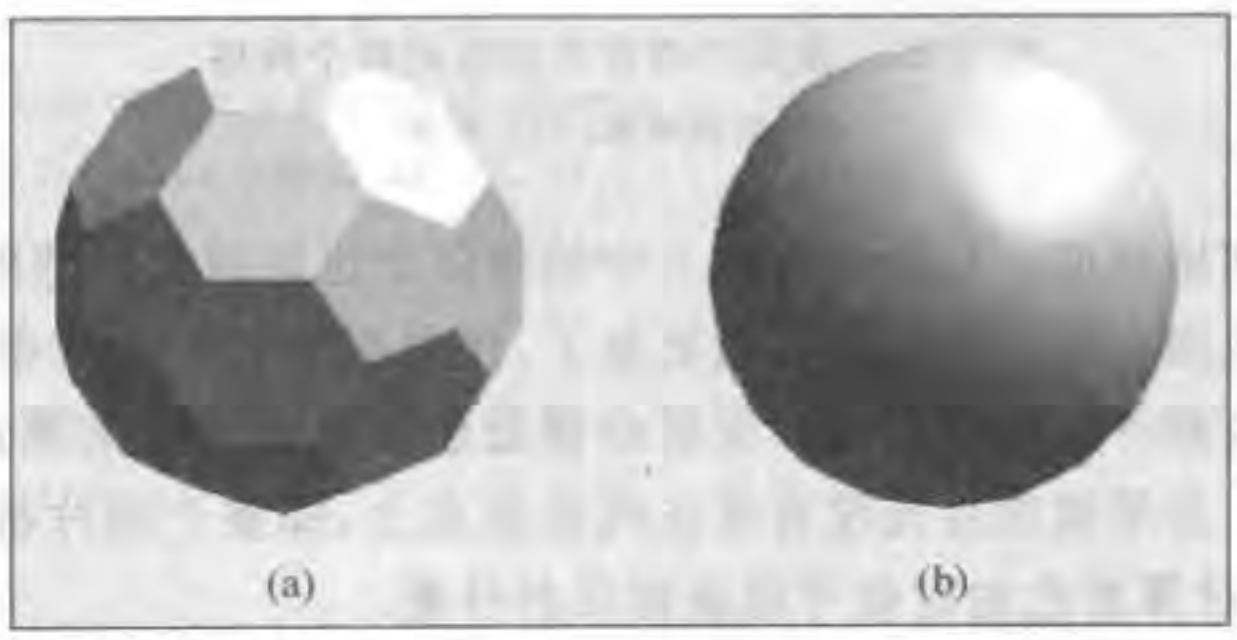


图 8.26 使用平滑着色渲染的两个网格

(a) 多面球体；(b) 球体

为什么使用这个技术后便消失了？图 8.27(a)画出了两个面片 F 和 F'，它们共享一个边而且假定物体想渲染出单独的面片。当渲染 F 的时候，使用了颜色 c_L 和 c_R，当渲染 F' 的时候，使用的是颜色 c'_L 和 c'_R。但是因为 c_R 等于 c'_L，所以在边的附近沿扫描线出现了颜色明显的变化，这也是我们希望的结果。计算 c_L 和 c_R 时所用的法向是通过平均与这个边相邻的面的法向得到的。

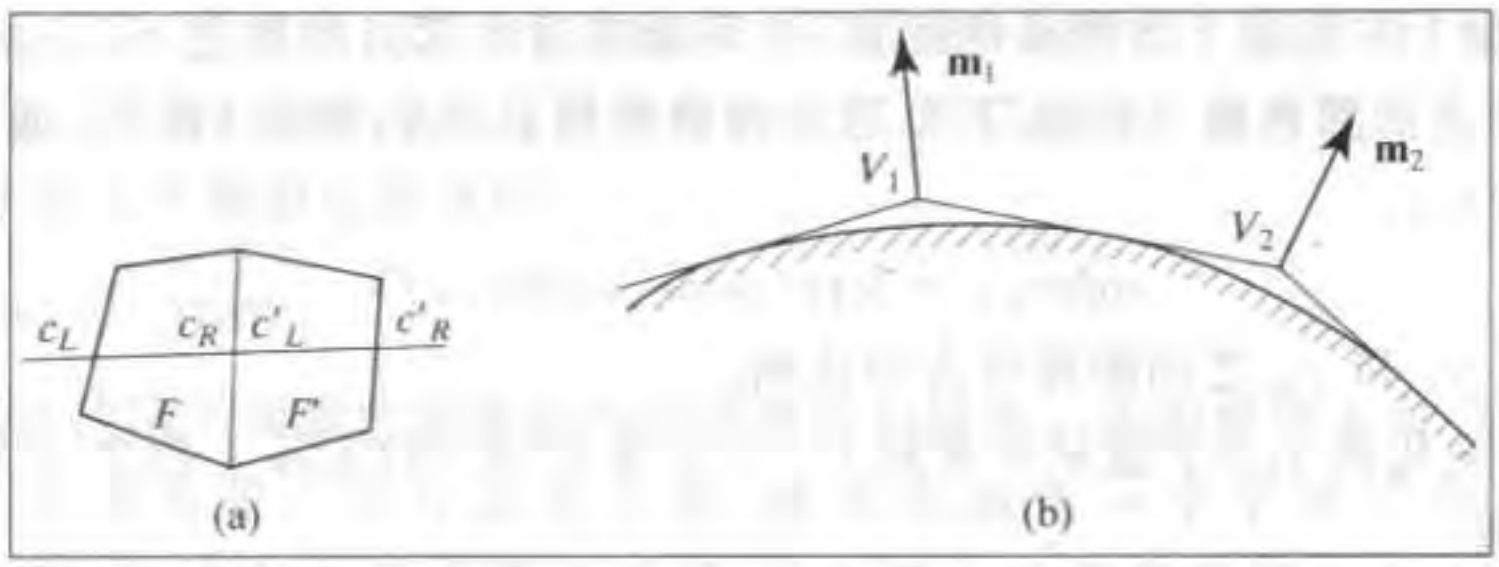


图 8.27 多边形两侧颜色的连续性

(a) 有明显边缘的物体；(b) 平滑曲面的物体

图 8.27(b)解释了这个技术如何显示用网格逼近的潜在曲面，这个时候我们想得到一个平滑物体的图像。图中有一个标注了 V_1 和 V_2 等的多边形表面，同时用这个多边形面片逼近的假象的平滑曲面也画了出来。顶点法向 $\mathbf{m}_1$、$\mathbf{m}_2$ 等的计算要使其垂直于该点所在的假象曲面，这样在每个顶点上用这个法向进行渲染就是正确的，因此计算出的颜色也是正确的。这样产生的顶点颜色就非常平滑，这只是一个简单的基于插值的数学定律而不是物理定律。

因为颜色是通过插值而不是在每个像素点上计算出来的，Gouraud 明暗处理不能很好地处理高光。因此，当使用 Gouraud 明暗处理时，等式(8.12)中光强的镜面反射分量经常被去掉。高光可以通过下面将要介绍的 Phong 明暗处理来更好的重现。

Phong 明暗处理

通过更好地逼近面片上像素处的法向量可以得到更加真实的效果——特别是对于高光和有光泽的物体。这种着色方法称为 **Phong 明暗处理**(Phong Shading)，以其发明者 Phong Bui-tuong【Phong75】命名。

计算 Phong 明暗处理时，我们要找到面片上每个点处的法向量，然后再在那里使用着色模型计算颜色。我们计算每个像素点处的法向量的方法是插值多边形顶点的法向量。

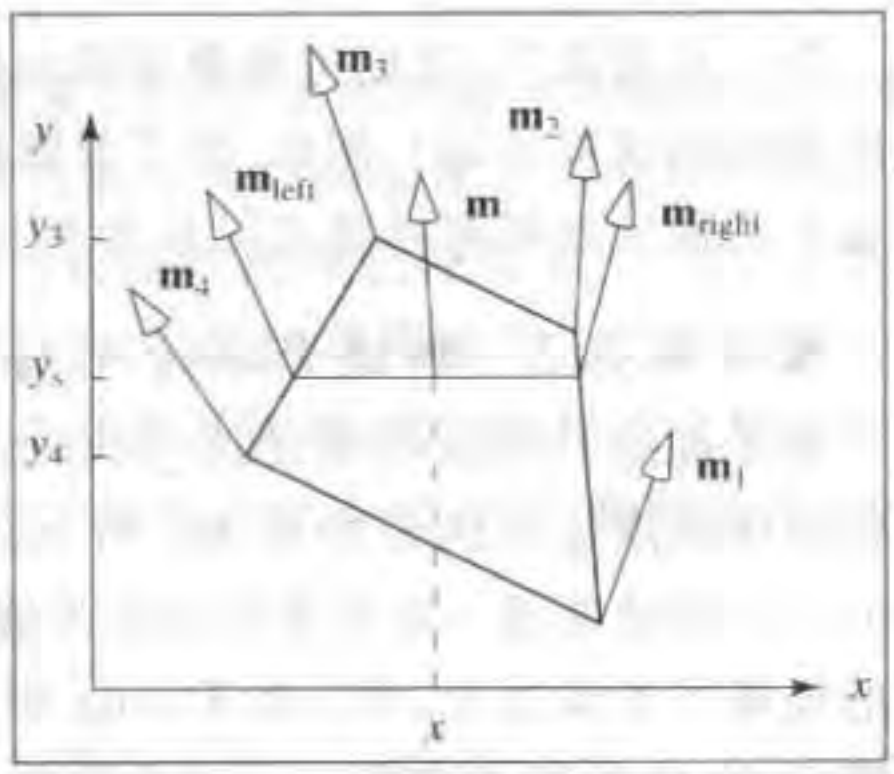


图 8.28　法向插值

图 8.28 画出了一个投影后的面片，四个顶点处标注了法向量 $\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$ 和 $\mathbf{m}_4$。对于所示的扫描线 y_s，向量 $\mathbf{m}_{left}$ 和 $\mathbf{m}_{right}$ 通过线性插值得到。

插值后的向量必须要标准化为单位向量，因为它要被用在渲染等式中。一旦 $\mathbf{m}_{left}$ 和 $\mathbf{m}_{right}$ 已知，它们就可以被插值得到扫描线上每个位置 x 处的法向量。这个向量被单位化后就可以在着色计算中使用，以得到这个像素的颜色。

图 8.29 对比 Gouraud 和 Phong 明暗处理的效果。因为各点之间法向量方向的变化非常平滑，也更接近于潜在的平滑曲面，所以和 Gouraud 明暗处理相比，Phong 明暗处理产生的高光更加真实，最终的渲染结果也更真实。

图 8.29　对比

(a) Gouraud 明暗处理；(b) Phong 明暗处理

Phong 明暗处理的主要缺点是每个像素上的计算量，因此 Phong 明暗处理所用的时间可能是 Gouraud 明暗处理的 6～8 倍。一些加速这个处理的方法也相继被提出来【Bishop86，Claussen90】。

OpenGL 不能够设置使用 Phong 明暗处理，因为它只在模型视点变换后的每个顶点上使用一次光照模型。法向量这个信息并不会传递到投影变换和投影剪裁步骤之后的渲染这

步。但是,在 8.5 节我们将看到一种逼近 Phong 明暗处理的方法,它使用环境贴图技术将高光纹理贴到物体上。

练习

练习 8.3.1 填充面片

完成多边形填充算法的操作细节,多边形顶点$(x,y)=(23,137),(120,204),(200,100),(100,25)$,对于扫描线 $y=136$, $y=137$ 和 $y=138$。写出每种情况下 x_{left} 和 x_{right} 的值。

练习 8.3.2 *Gouraud* 明暗处理中保留边

在一些情况下,我们可能希望画出模型中特定的褶皱和边。讨论如何通过选择顶点的法向量控制这样的边?例如,为了保留图 8.27 中的面片 F 和 F'之间的边,顶点法向应该是什么?其他的技巧和问题可以在参考文献中找到,如【Rogers85】。

练习 8.3.3 加速 *Phong* 明暗处理的篱笆明暗处理(*fence shading*)

为了加速 Phong 明暗处理,Behrens【Benhrens94】提出了一种插值方法,在每个扫描线上仍然用顶点法向插值得到 $\mathbf{m}_L$ 和 $\mathbf{m}_R$,然后计算这个最左和最右像素点的颜色,随后像 Gouraud 明暗处理一样沿着扫描线插值这两个颜色。这种称为"篱笆明暗处理"的方法大幅度的提高了渲染速度,而且和 Phong 明暗处理相比高光的处理效果只是略微下降。描述一下图 8.28 中的顶点法向 $\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$ 和 $\mathbf{m}_4$ 的方向,使得

a. 篱笆明暗处理产生和 Phong 明暗处理一样的高光

b. 篱笆明暗处理和 Phong 明暗处理产生的高光非常不一样

练习 8.3.4 *Phong* 明暗处理算法

在逐像素着色代码上做必要的修改,使其加入 Phong 明暗处理。假设每个面片的顶点法向量已知。同时讨论使用 OpenGL 的平滑着色算法如何逼近 Phong 明暗处理。提示:增加模型中的面片数量。

8.4 增加隐藏面消隐

如果有足够的内存作为**深度缓存**(depth buffer)(也称为"z-缓存","z-buffer"),在我们前面讨论的渲染过程中加入隐藏面消隐是非常简单的。因为它能够很容易地融入我们讨论的渲染机制,所以我们到这步才介绍它。

8.4.1 深度缓存方法——OpenGL 使用的方法

深度缓存(或者 z-buffer)算法是一个最简单最容易实现的隐藏面消隐算法,它也是 OpenGL 所使用的方法。它的主要局限是需要大量的内存,还有经常绘制那些会被随后的较近物体遮挡的物体(所以绘制较远物体的时间就被浪费了)。

图 8.30 画出了一个深度缓存和与其对应的帧缓存(frame buffer)。对于屏幕上每个像

素 p[i][j]，深度缓存一个 b 比特的量 $d[i][j]$。b 的值通常在 12～30 的范围内选取。

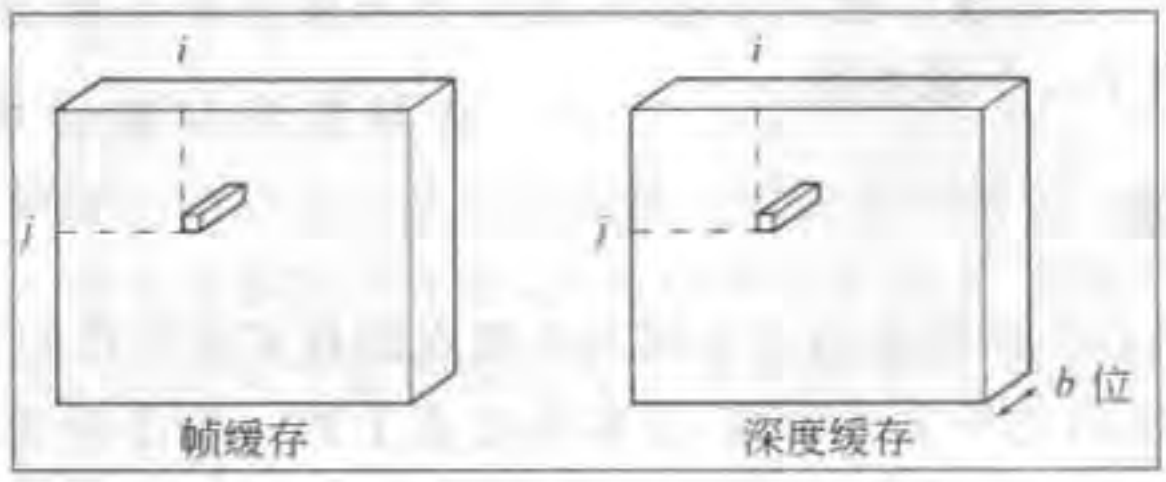


图 8.30　深度缓存示意

在渲染过程中，深度缓存值 $d[i][j]$ 存储这个像素（当前）遇到的最近物体的伪深度。随着面片扫描线上的像素一个个地被处理，一起测试当前面片的伪深度是不是小于该点深度缓存中存储的伪深度值 $d[i][j]$。如果小于，则用当前面片的颜色替代 $p[i][j]$，并且用这个更小的伪深度替换 $d[i][j]$ 中旧的深度值。面片可以以任意顺序绘制。如果远一些的面片先被绘制，该面片的一些像素将被随后更近面片的颜色所替代，因此渲染这个较远面片的时间就浪费了。需要注意的是这个算法对于任何形状的物体都有效，包括曲面在内，因为它寻找最近的表面是基于逐点的测试。

综上所述

- 对于每个像素点 ij，$p[i][j]$ 保存屏幕上该点目前最近面片的颜色。
- $d[i][j]$ 保存屏幕上该点目前最近面片的伪深度值。

数组 $d[][]$ 初始值是 1.0，这是伪深度值可能的最大值。帧缓存初始值是背景颜色。

计算每个像素的伪深度值

我们需要一个快速计算每个像素伪深度的方法。回顾面片的每个顶点 $P=(P_x,P_y,P_z)$ 传入图形管线，经过一系列变换，每个顶点在视窗变换后可用的信息是三元组。这是下式的一个缩放和平移后的结果（见等式(7.2)）：

$$(x,y,z)=\left(\frac{P_x}{-P_z},\frac{P_y}{-P_z},\frac{aP_z+b}{-P_z}\right)$$

其中第三个分量就是伪深度值。常数 a 和 b 的选择方法，是使第三个分量对于近平面上的点 P 等于 0，对于远平面上的点 P 等于 1。为了更高的效率，我们希望能够增量地计算每个像素的伪深度值，这也是使用插值的方法，类似我们用等式(8.15)计算颜色时所做的。

图 8.31 中是一个正在沿扫描线 y 填充的面片。标注了各个点的伪深度值。顶点处的伪深度值 d_1、d_2、d_3 和 d_4 是已知的。我们要计算扫描线 y_s 上 d_{left} 的值 lerp(d_1，d_4，f)，和 d_{right} 的值 lerp(d_2，d_3，h)。我们还要通过 lerp(d_{left}，d_{right}，k)计算扫描线上像素(x，y)的伪深度值 d(h，k 的值是什么?)。问题是这样算出来的三维面片上相应点的伪深度值是"真"的么。

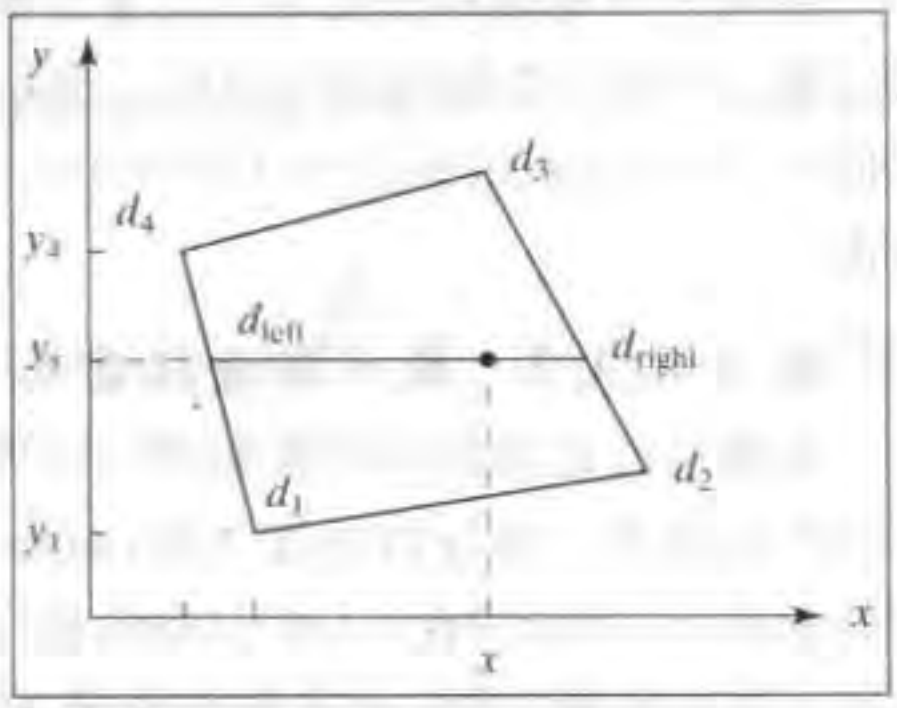


图 8.31　伪深度值的增量计算

答案是不完全地有效。如果原始的三维面片是平的,则计算就是对的,因为投影变换保证变换后还是平的。但是回顾以前介绍过的,这个伪深度值并不是随物体距视点的真实距离线性的变化,而是趋于一个渐近线。

远处的深度压缩

回顾 7.4 节,一个点的伪深度值不是随其距视点的真实值线性的变化,而是趋于一个渐近线。这意味着当深度值比较大的时候,真实深度值上的一个小的变化映射到伪深度值后变为极小的变化。因为表示伪深度值的比特数有限,两个接近的深度值有可能会被映射为相同的伪深度值,这就可能在进行 d<d[x][y]比较的时候导致错误。使用更多的比特来表示伪深度值可以解决这个问题,但是需要更多的内存。而且对于近平面距离视点比较远的这种可能情况,这样做的效果也很小。

OpenGL 支持一个深度缓存并且使用上述算法实现隐藏面消隐。你必须在初始化显示模式的时候指明使用深度缓存:

```
glutInitDisplayMode(GLUT_DEPTH| GLUT_RGB);
```

并且启用深度测试:

```
glEnable(GL_DETPH_TEST);
```

渲染每一幅图像之前要初始化深度缓存:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);          //清屏
```

其他许多的隐藏面消隐方法可以在本书的配套网站上找到,可以作为深入阅读的资料。和本算法相比,那些算法复杂得多,并且必须被仔细地检查以保证对所有的情况都正确。

练习

练习 8.4.1 补充

补充从已知的顶点伪深度值计算 d_{left}, d_{right} 和 d 值的细节。

练习 8.4.2 深度值编码

假设为深度缓存中的每个元素分配 b 个比特。这 b 个比特用来记录从 0 到 1 之间的伪深度值。0 和 1 之间的值可以用二进制数 $.d_1d_2d_3\cdots d_b$ 表示,其中 d_i 是 0 或者 1。例如,伪深度值 0.75 可以被编码为 0.1100000000…这是不是一个使用 b 比特的方法?从正反两方面讨论。

练习 8.4.3 减少深度缓存的大小

如果没有足够的内存来实现一个完全的深度缓存,我们可以一片一片地生成图像。深度缓存用来为一部分扫描线工作,对每一部分图像反复调用算法。比如,在一个 512×512 的显示屏上,只能分配 64 个扫描线相应的深度缓存,所以要反复执行 8 次算法。每次所有的面片都被扫描,计算相应扫描线覆盖区域的面片深度,并且和当时的深度缓存比较。当然,所有的面片的 8 次扫描使得这个算法的运行很慢。假设一个场景有 F 个面片,每个面

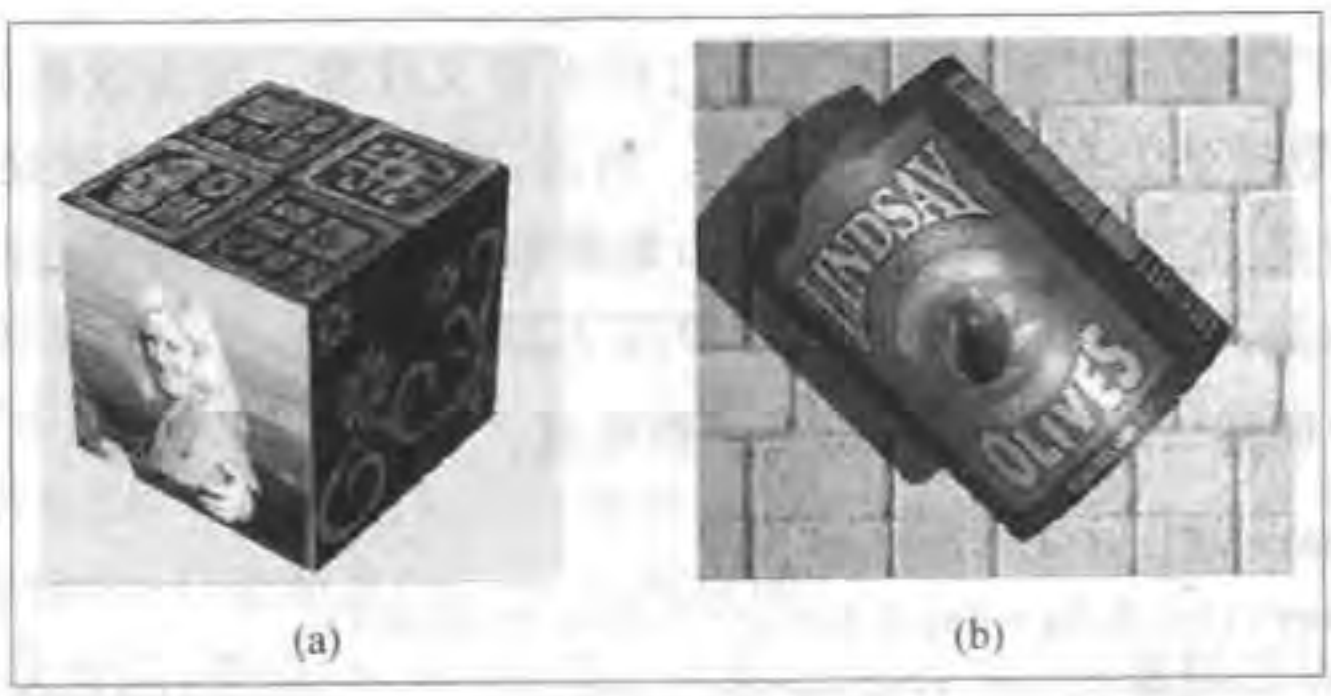
片平均覆盖 L 个扫描线。当分配 $nRows/N$ 扫描线的内存给深度缓存时，计算这个方法将多用多少时间？

练习 8.4.4　单扫描线深度缓存

上一个练习中把帧缓存的分块方法可以被推广到极限的情况：深度缓存只存储一行扫描线的深度。这看上去需要更多的计算量，因为每个面片因为"更新"需要被处理多次，每个扫描线都需要处理一次。讨论对于这种情况算法将如何修改，并且估计和一个全屏深度缓存相比这将要多用多长的时间？

8.5　为面片添加纹理

图像的真实感连同其对于观察者的吸引力，能够通过为网格物体的各个表面贴纹理而被显著增强。图 8.32 给出了一些例子。在图 8.32(a)中，一些图片被贴在立方体的表面上。在图 8.32(b)中，一个标签被弯曲地贴在圆柱罐上，背后的墙看上去也是由砖块砌成的。



(a)　(b)

图 8.32　物理映射示例

(a) 纹理映射到立方体表面；(b) 纹理包裹在一个罐头上，还有贴在罐头后面的墙上

基本的技术始于在"**纹理空间**(texture space)"中定义的一组纹理函数，比如图 8.33 所示的那些。纹理空间中的参数按习惯以 s 和 t 命名。纹理就是一个函数 texture(s,t)，对于 s 和 t 在 0 到 1 之间的取值产生一个颜色或者亮度值。

我们要看如何把纹理贴到物体的各个点上，但是首先我们先看看不同纹理的来源。

不同的纹理来源

纹理最普遍的来源是位图(矩阵一样的像素数组)和生成函数。图 8.33(a)是一个位图的例子，对应的图 8.33(b)是一张通过一些函数计算生成的(人造)图片。

位图纹理

位图经常来源于位图表示的图片(比如数字照片、数码相机图片、剪切画或者用一些程序生成的图片)。另外，互联网也提供了数以万计的有趣纹理。这种纹理由二维数组组成，

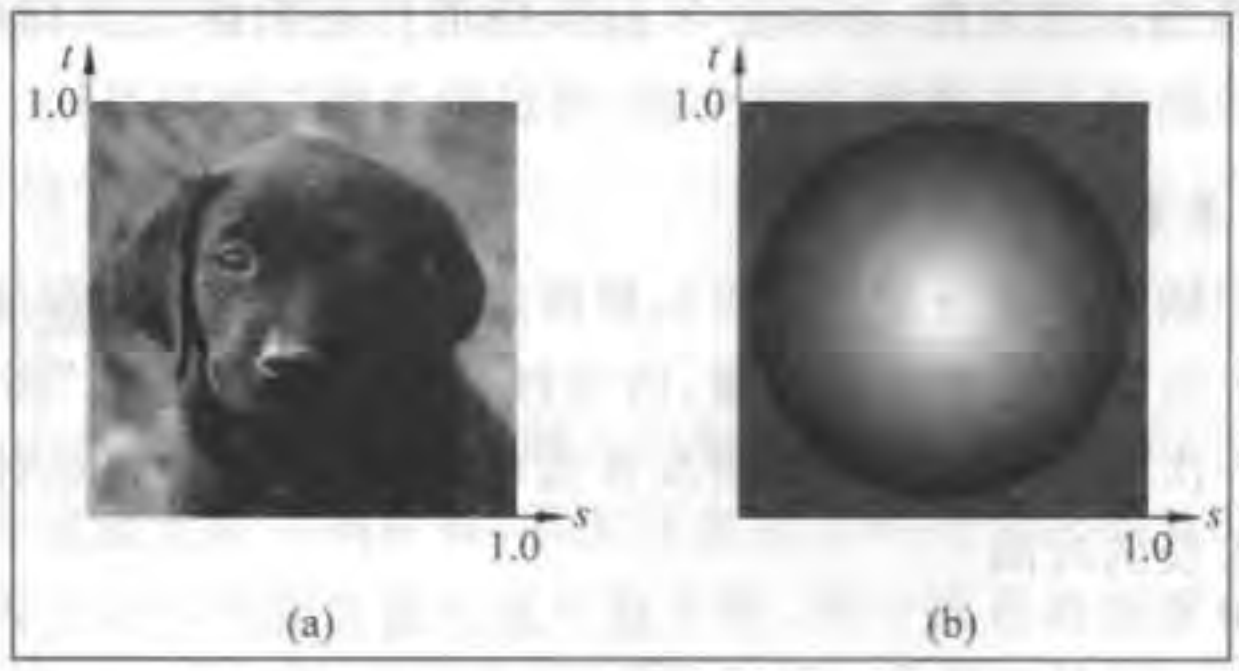


图 8.33 纹理示例

(a) 图片纹理;(b) 生成的纹理

记为 txtr[r][c],存储着颜色值(经常称为"**纹理元素(texels)**")。我们的第一个任务是把数组 txtr[r][c]中的值适当地和 0 到 1 之间的 s、t 值联系起来。假设数组 *txtr*[][]有 c 列和 r 行。

纹理映射的过程化生成

除此之外,我们可以通过数学函数或者过程来定义纹理。因为这些函数通常都由编程人员创建,所以函数描述上有很大的自由度。例如,图 8.33(b)中的球处于图片中心,圆心位于二维坐标的$(s,t)=(0.5,0.5)$,半径是 0.5,所以从点(s,t)到其中心的距离是:

$$\sqrt{(s-0.5)^2+(t-0.5)^2}$$

图 8.31(b)中的球纹理可以通过以下函数生成:

```
float fakeSphere (float s, float t){
    float r=sqrt((s-0.5) * (s-0.5)+(t-0.5) * (t-0.5));
    if (r>0.3) return 1-r/0.3; else return 0.2;}        //0.2 is the background
```

这个函数值域从位于中心的 1(白色)到位于(顺着轴线)边缘的 0(黑色),组成一个圆。在处理函数中的值 0.3 使得球内最暗的亮度能和背景亮度平滑地融合。练习中给出了另外一个模仿国际象棋棋盘的纹理。任何输入是各个 s 和 t 值的函数都可以被称为纹理:平滑融合和卷曲的颜色,Mandelbrot 集合(见附录),固体的线框绘制等等。

我们后面会看到有多种方法来使用 texture(s,t)的值:如果面片发光可以被用来当面片的颜色;可以被用来当做环境光、漫反射或者镜面反射系数去调和面片反射光的比例;可以用来改变表面的法向量而使物体产生凹凸的外表。

练习

练习 8.5.1 经典的棋盘纹理

图 8.34 给出了一个棋盘,它是由明暗相间的 5×5 的方格组成,亮度是 0(黑色)和 1(白色)。

为这个纹理写出函数 float texture(float s, float t)(参见练习 2.3.1)。

在有了纹理函数之后,下一步是如何把它适当地映射到物体表面,然后从相机中看到

它。图 8.35 中的例子展示了所有的问题。这里一个简单的纹理样例被映射到两个不同的物体：一个平面多边形和一个圆柱。对于每个物体，都需要有个变换，称为 T_{tw}（从纹理坐标到世界坐标）。这个变换把纹理(s, t)的值映射到物体表面的点(x, y, z)上。相机从某个角度拍摄场景，产生最后看到的结果。我们把点从三维坐标变换到屏幕的变换称为 T_{ws}（从世界坐标到屏幕坐标），则物体表面上一点(x, y, z)被看到的像素位置是$(s_x, s_y)=T_{ws}(x, y, z)$。所以综合起来，纹理上的值$(s^*, t^*)$最后到达像素是$(s_x, s_y)=T_{ws}(T_{tw}(s^*, t^*))$。

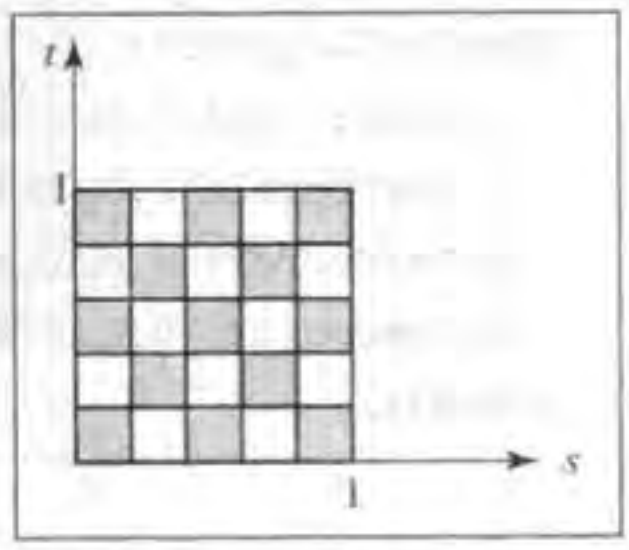


图 8.34　经典的棋盘纹理

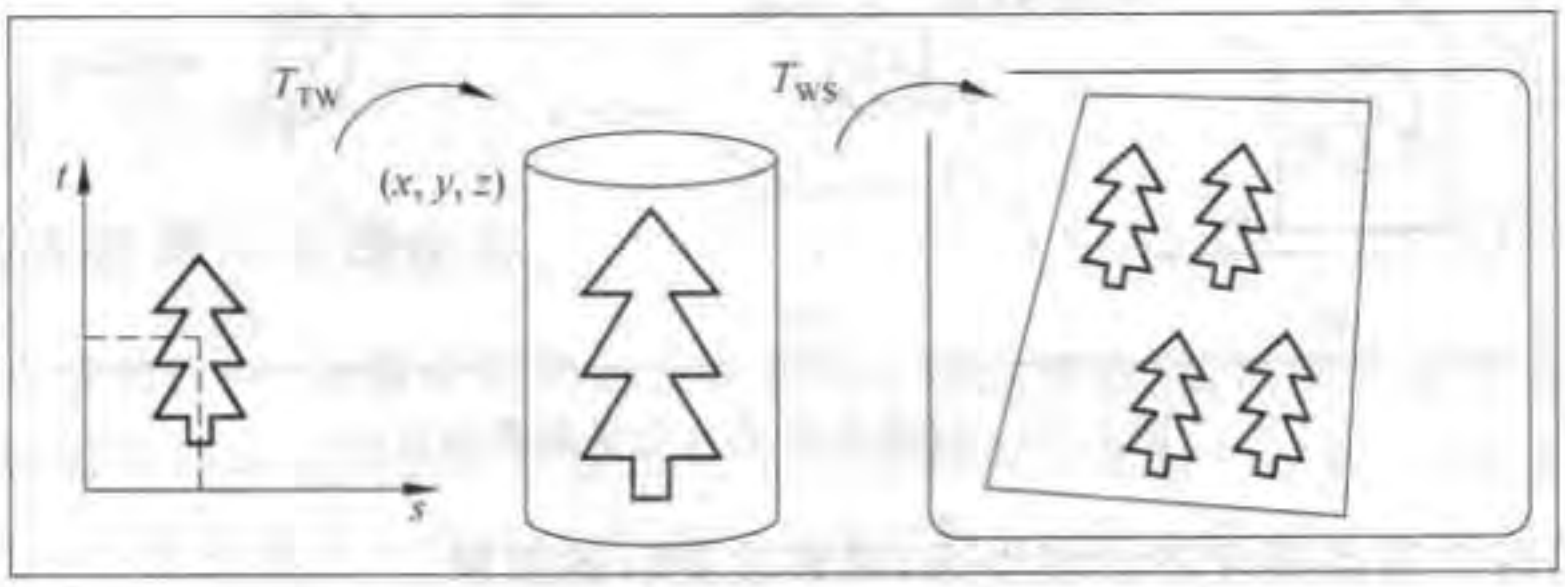


图 8.35　纹理贴在一个圆柱体和一个平面上

渲染过程实际上是另一个途径：对于像素(s_x, s_y)这有一系列的问题：

a. 从(s_x, s_y)看到的最近曲面是什么？回答这个问题需要解决隐藏面消隐问题，这样才能决定哪个纹理才是相关的。

b. 曲面上哪个点(x, y, z)和(s_x, s_y)是对应的？

c. 哪个纹理坐标(s, t)是和点(x, y, z)对应的？

所以我们需要反变换，类似$(s, t) = T_{tw}^{-1}(T_{ws}^{-1}(s_x, s_y))$，这样给出像素的坐标就能够得到$(s, t)$的坐标。反变换依据曲面的不同可能很容易得到也可能很难得到。

8.5.1　在平面上贴纹理

我们首先看看最重要的情况：映射纹理到一个平面。这是一个建模任务。然后我们在看过纹理是如何真正地被渲染出来的以后，解决视角的问题。最后我们讨论把纹理映射到更加复杂的形状表面。

把纹理贴到平面上

因为纹理空间本身是平面，所以把纹理贴到平面上是最简单的。图 8.36 给出了一个（在五边形中的）纹理图像映射到一个平面多边形（另外的一个五边形）F。我们必须指定 F 上每个点在纹理上的对应点。在 OpenGL 中，指定纹理空间中的点 $P_i=(s_i, t_i)$和面片上的顶点 V_i 的对应关系，使用函数 glTexCoord2f()。函数 glTexCoord2f(s,t)设置当前的纹理坐标为(s,t)，这个坐标就和随后定义的顶点联系到一起。通常每个 glVertex2f()调用前都要调用 glTexCoord2f()，这样每个顶点都有一对不同的纹理坐标。例如，为了定义一个四边形并且在

其上设置一个纹理,我们通过如下程序传给 OpenGL 4 个纹理坐标和 4 个三维顶点:

```
gBegin(GL_QUADS);        //define a quadrilateral and position our texture on it
  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, 2.5, 1.5);
  glTexCoord2f(0.0, 0.6); glVertex3f(1.0, 3.7, 1.5);
  glTexCoord2f(0.8, 0.6); glVertex3f(2.0, 3.7, 1.5);
  glTexCoord2f(0.8, 0.0); glVertex3f(2.0, 2.5, 1.5);
glEnd();
```

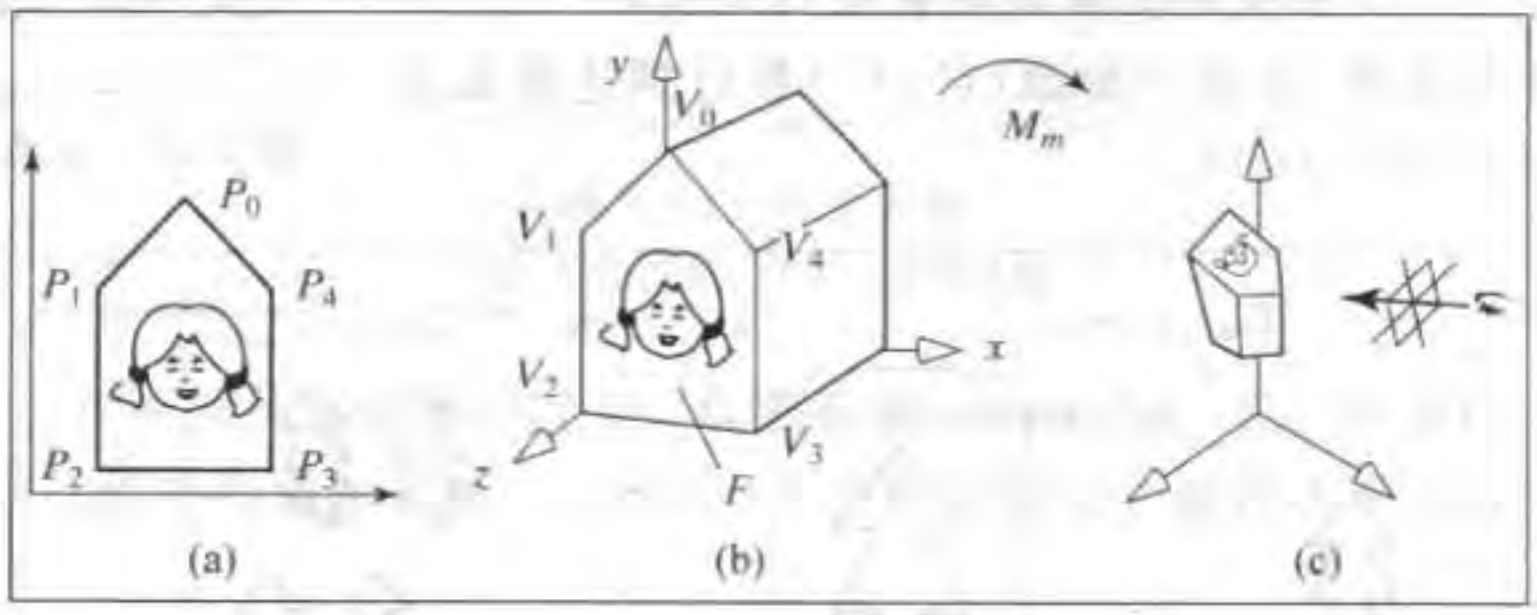


图 8.36 纹理映射到一个平面多边形

改变上面四个顶点相应的纹理坐标,观察改变后的效果。

为每个 V_i 设定 P_i 等价于在纹理空间定义了一个多边形 P,它拥有和 F 一样多的顶点。通常 P 和 F 的形状也是一样的:在 P 中的这部分纹理被无扭曲地贴到整个 F 上面。当 P 和 F 有相同的形状时,这个映射无疑是仿射变换:缩放,有可能伴随着旋转和平移。

图 8.37 给出了一个普通的情况,矩形纹理的四个角对应到三维场景中面片的四个角。(纹理坐标(s, t)赋值到对应的三维面片的拐角上。)在这个例子中,纹理是一个 640×480 像素的位图,并且被贴到长宽比为 640/480 的矩形上,所以看上去没有扭曲(注:纹理坐标 s 和 t 的变化范围依旧是 0 到 1)。图 8.38 使用的纹理坐标产生了反复平铺纹理的效果。为了做到这一点,一些纹理坐标的值处于[0,1]区间之外。当渲染器遇到 s 和 t 的值处于单位正方形之外的时候,比如 s=2.76,它忽略掉整数部分而只使用小数部分 0.76。这样面片上使用(s, t)=(2.6,3.77)为其纹理的点所帖的纹理是 texture(0.6, 0.77)。默认情况下 OpenGL 就是这样平铺纹理的。如果需要,它也可以替代设置使用"剪裁"纹理值的方法,参见练习。

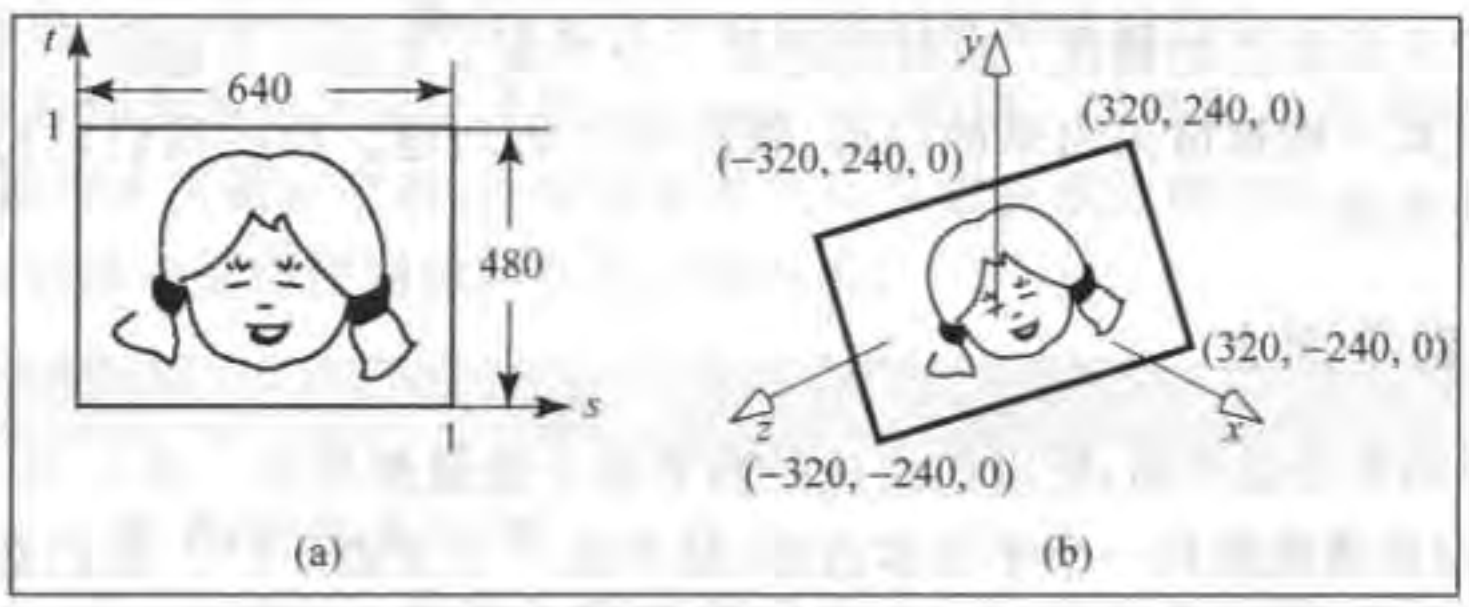


图 8.37 正方形映射到矩形

这样一个纹理坐标(s, t)跟随着面片的顶点沿流水线传递。下一节我们会介绍,F 中的点将被填充上 P 内对应的纹理值,而且使用插值计算所得到的纹理坐标(s, t)值。这个插

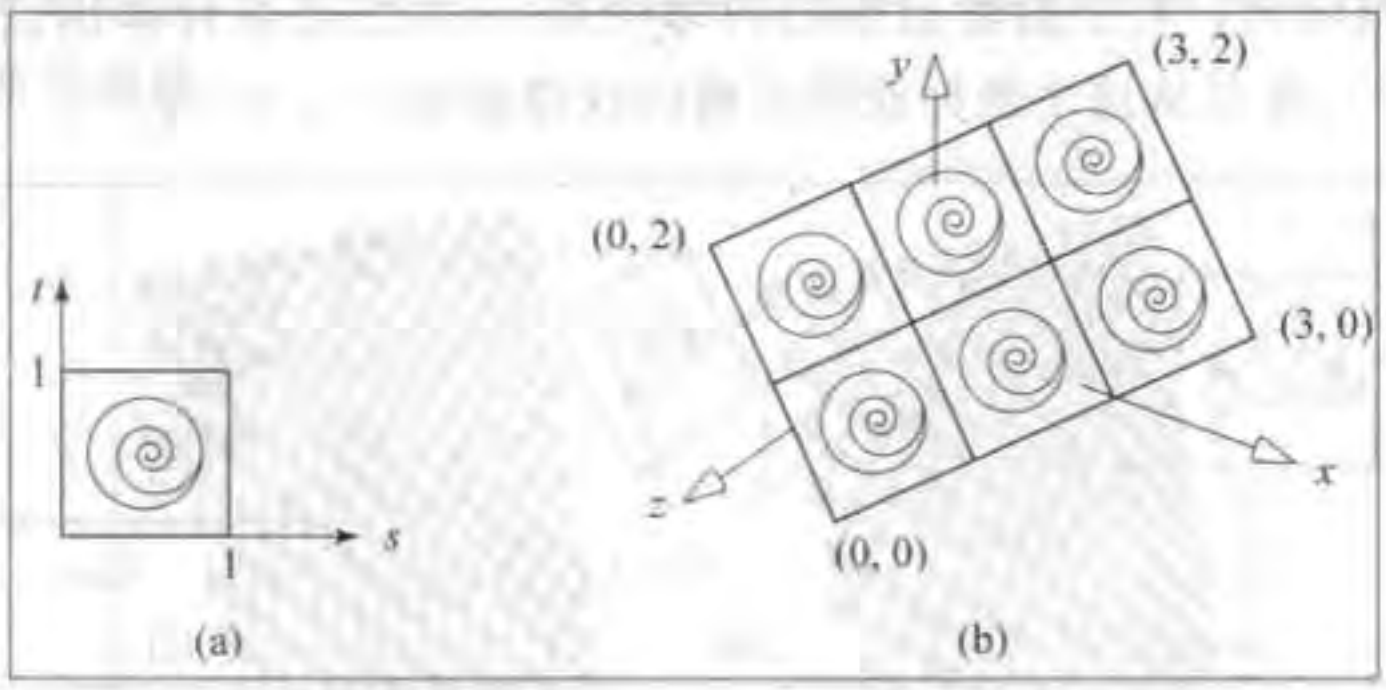


图 8.38　产生反复出现的纹理

值过程将在下一节介绍。

为网格模型添加纹理坐标

回顾第 6 章的代码，网格模型拥有三个列表：顶点列表、法向量列表和面片列表。我们必须在其中添加纹理坐标列表以存储不同的顶点对应的坐标(s_i, t_i)。为这目的我们添加一个数组，每个元素是类 TxtrCoord 中的一对浮点数，类定义是

```
Class TxtrCoord { public: float s, t; };
```

数组存储了所有网格的纹理坐标对。这有几种不同的处理模型纹理的方法，每种方法都要指明模型中纹理信息是如何组织的。最重要的两种方法如下。

1. 网格模型是由一些平面片组成，并且每个面片应用不同的纹理。这样每个面片只有一个单独的法向值，但拥有其自己的纹理坐标列表。所以每个面片的数据组织可以是：

- 面片顶点的数量
- 面片法向的索引
- 顶点索引的列表
- 纹理坐标索引的列表

2. 网格表示一个潜在的平滑物体，并且单个纹理包裹着这个物体(或者其一部分)。这样每个顶点都对应一个指定的法向量和一个特定的纹理坐标。因此使用一个顶点、法向和纹理坐标索引的列表就可以了。每个面片的数据组织可以是：

- 面片顶点的数量
- 顶点索引的列表

练习中会更加深入地探讨这两种类型网格所需要的数据结构。

8.5.2　渲染纹理

渲染面片 F 中的纹理和 Gouraud 明暗处理相似：渲染器在面片中一个像素一个像素地移动。对于每个像素它必须决定对应的纹理坐标(s, t)，访问纹理并且设置像素为适当的纹理颜色。我们将看到必须非常小心地寻找坐标(s, t)。

图 8.39 展示了相机拍摄帖有纹理的面片 F，并且正在进行渲染的过程。扫描线 y 正在

从 x_{left} 到 x_{right} 地被填充。对于沿着扫描线的每个点 x,我们必须计算出面片上正确的坐标(用 $P(x, y)$ 表示),并且从这个坐标获得正确的纹理坐标(s^*, t^*)。

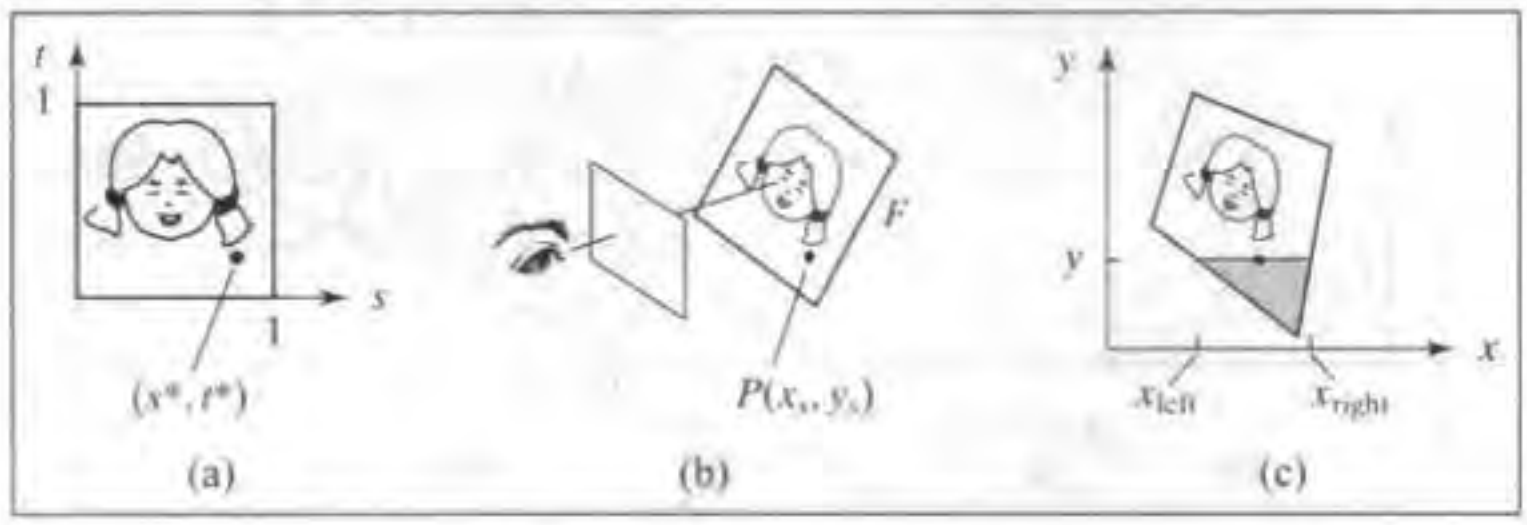


图 8.39 从视点看到的头像

在建立纹理到物体的映射之后,我们就知道面片 F 上每个顶点的纹理坐标,例如图 8.40 中的标注。自然的想法是为每个扫描线使用增量方法快速地计算(s_{left}, t_{left})和(s_{right}, t_{right}),然后在扫描线上随着位置移动插值这些值。但我们必须小心:当我们在扫描线 y 上从 x_{left} 到 x_{right} 移动,简单地使用增量方法从 s_{left} 变到 s_{right} 是不行的。因为在投影后的面片上相等的间隔并不对应于三维面片上相等的间隔。

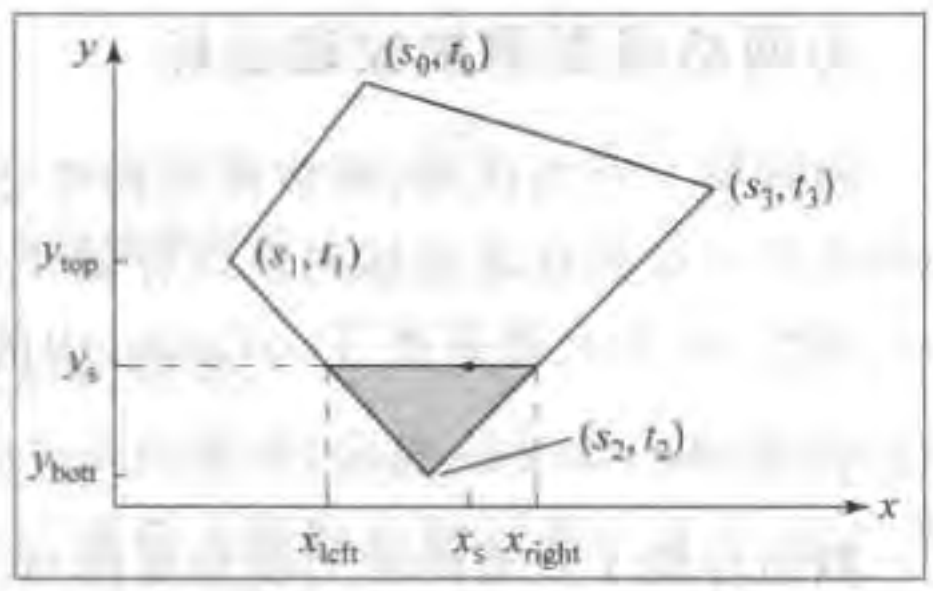


图 8.40 纹理坐标的增量计算

图 8.41 说明了这个问题。图 8.41(a)是一个左边比右边更接近观察者的面片 F。图 8.41(b)是这个面片在屏幕上的投影 F'。在扫描线 $y=170$ 上,我们等距地标出 F' 上的点代表面片上连续像素的位置。这些标记点对应到真正面片上的位置在图 8.41(a)中标出。可以看出这些点在距离 F 的远端会靠得更近,这就是立体视觉中简单的近大远小现象。

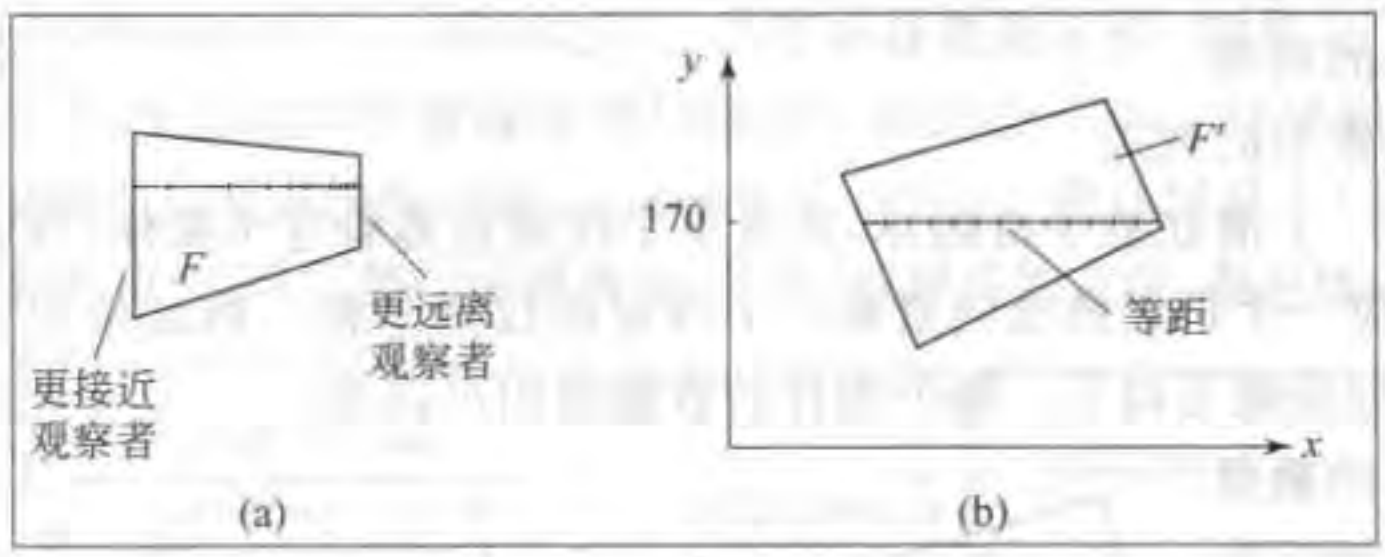


图 8.41 线性插值点的间距

如果我们使用简单的线性插值,等距地划分 s 和 t 的间隔来计算纹理坐标,我们就在错误的位置进行了纹理采样,得到一个图像扭曲的结果。图 8.42 说明了这样简单地映射棋盘纹理到一个矩形上会得到什么结果。图 8.42(a)中使用的线性插值产生了纹理上明显的扭曲。这个扭曲在多边形旋转的动画中会显得特别的混乱,因为纹理动态出现卷绕和拉伸。图 8.42(b)中使用了正确的插值,棋盘看上去就像其应该的那样。在动画中,这个纹理就会像紧紧地附着在旋转的面片上一样。一个完整的程序可以在网上(RotatingCube.cpp)找到,展示了一个 6 个面帖有不同纹理的旋转立方体。图 8.32 是这个旋转立方体程序的一个

截图。因为有正确的插值，所以每个面都没有扭曲变形。我们下面将要讨论如何写出正确的代码来实现正确的插值。

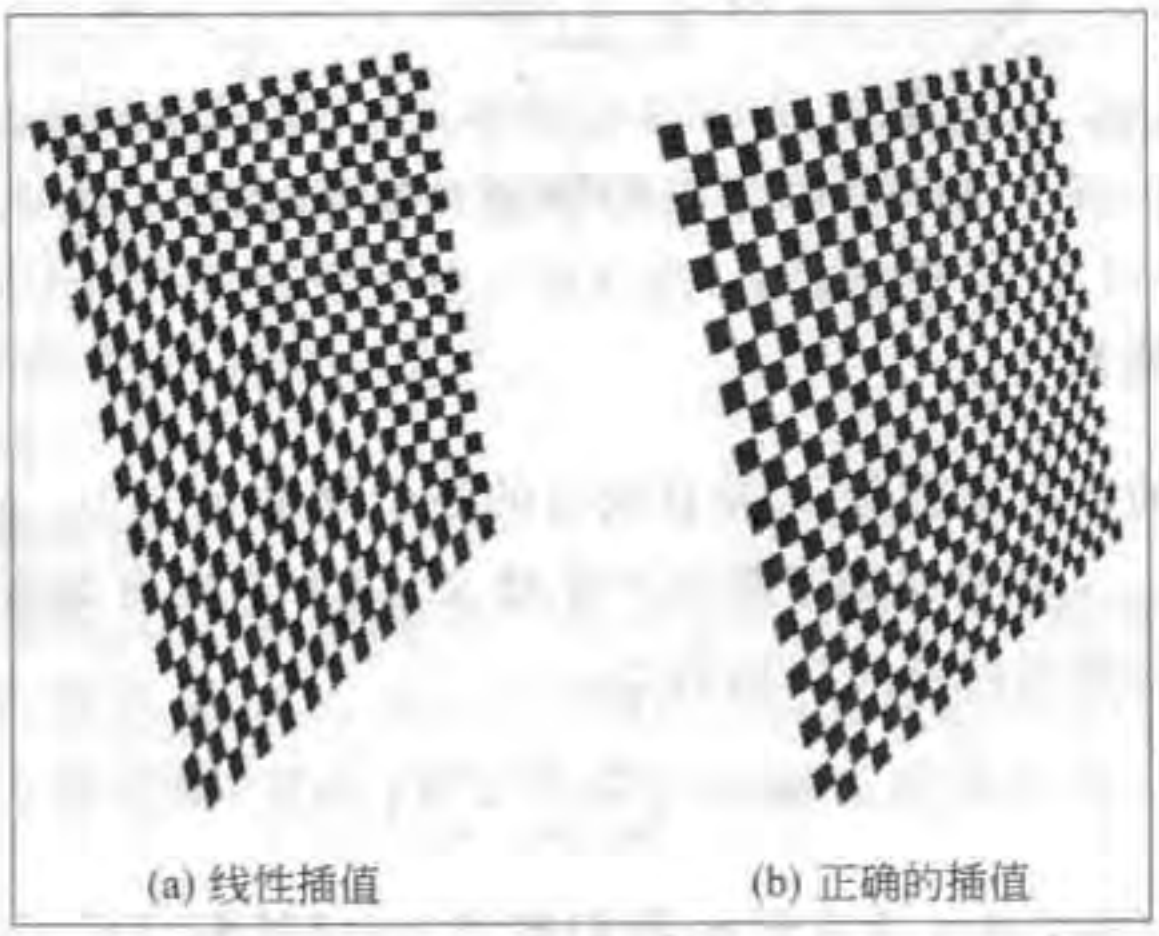


图 8.42 线性插值和正确插值的渲染结果

一些研究正确插值方法的文章已经被发表。Heckbert、Moretn【Heckbert91】和 Blinn【Blinn92】描述了一个基于仿射变换和投影变换本质的简洁推广。Segal 等人【Segal92】使用基于线段参数表达的更加代数化的推导得到了同样的结果。我们在此介绍后一种推导。

图 8.43 给出了我们将要分析的情况。我们知道仿射和投影变换保持直线性，所以在视点坐标系内直线 L_e 投影为平面坐标系下的直线 L_s。相似地，我们希望画在直线 L_t 上的纹理元素，在纹理坐标下是沿着直线 L_t 分布并且映射为 L_s。关键的问题是：如果我们在屏幕直线 L_s 上移动相等的距离，在纹理空间上纹理元素将沿 L_t 怎样移动？

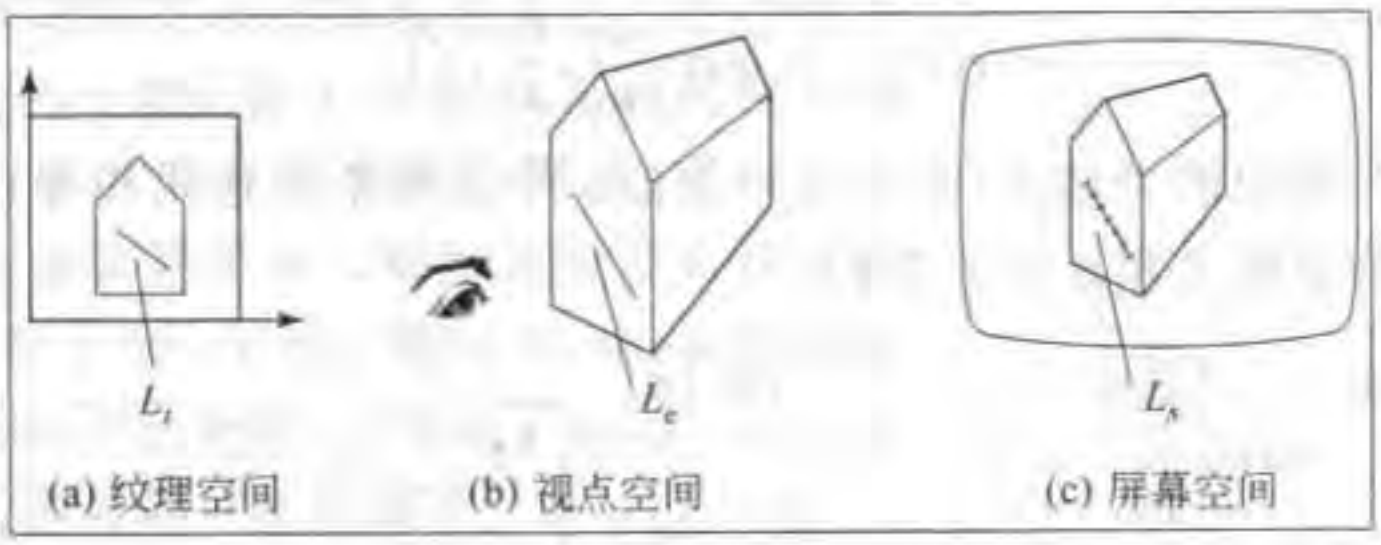


图 8.43 直线从一个空间映射到另一空间

我们在后面推导了一个一般的结果，总结如何进行插值：所做的都是和立体视觉分割的效果有关。然后我们把这个结果和图形管线中执行的变换联系起来，看看为了完成正确的纹理映射所需要的额外步骤在哪里。

图 8.44 画出了三维中的直线 AB 经过矩阵 M 的变换后变为三维中的直线 ab（M 可以表示一个仿射变换，或者是更一般的透视变换）。点 A 映射到 a，B 映射到 b。考虑点 $R(g)$ 位于由 A 到 B 中比例为 g 的位置。它被映射到位于由 a 到 b 中比例为 f 的某点 r(f)。我们将会看到比例 f 和 g 的值并不相等。问题是当 f 值从 0 变到 1，g 值将如何变化？这也就是沿 ab 的运动如何对应沿 AB 的运动？

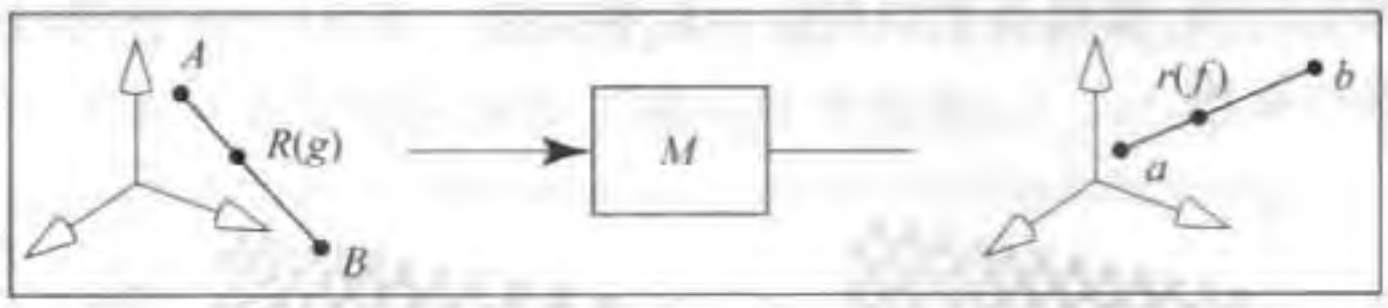


图 8.44 对应直线的运动如何变换?

推导 g 和 f 的关系

我们假定坐标 a 的齐次坐标是 $\tilde{a}$,并且设它的各个分量 $\tilde{a}=(a_1, a_2, a_3, a_4)$。(我们使用下标 1、2、3、4 替代 x、y 等等是为了防止产生歧义,因为这里有太多的"x, y, z"了)。所以通过对 $\tilde{a}$ 进行透视变换的除法可以得到点 a:

$$a = \left(\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4}\right)$$

因为 M 映射 $A=(A_1, A_2, A_3)$ 到 a,我们知道 $\tilde{a}=M(A, 1)^T$,其中 $(A, 1)^T$ 是分量为 A_1、A_2、A_3 和 1 的列向量。类似地,$\tilde{b}=M(B,1)^T$(仔细检验这些关系)。为了保持简洁我们使用符号 lerp(),我们已经定义了 $R(g)=\text{lerp}(A, B, g)$,这点映射到 $M(\text{lerp}(A, B, g), 1)^T=\text{lerp}(\tilde{a}, \tilde{b}, g)=(\text{lerp}(a_1, b_1, g), \text{lerp}(a_2, b_2, g), \text{lerp}(a_3, b_3, g), \text{lerp}(a_4, b_4, g))$(检验这些关系)。这是点 $r(f)$ 的齐次坐标版本 $\tilde{r}(f)$,所以能通过透视变换的除法恢复真实的 $r(f)$ 的分量。为了简洁,我们只写出第一个分量 $r_1(f)$:

$$r_1(f) = \frac{\text{lerp}(a_1, b_1, g)}{\text{lerp}(a_4, b_4, g)} \tag{8.16}$$

但是因为定义 $r(f)=\text{lerp}(a, b, f)$,我们有第一个分量 $r_1(f)$ 的另外一个表达式:

$$r_1(f) = \text{lerp}\left(\frac{a_1}{a_4}, \frac{b_1}{a_4}, f\right) \tag{8.17}$$

$r_2(f)$ 和 $r_3(f)$ 相应的表达式(是什么?)类似。令这两个表达式相等,再做一些代数变换可以得到我们希望的 f 和 g 的关系:

$$g = \frac{f}{\text{lerp}\left(\frac{b_4}{a_4}, 1, f\right)} \tag{8.18}$$

因此点 $R(g)$ 映射到了 $r(f)$,但是 g 和 f 不相等。g 只有在 $f=0$ 和 $f=1$ 时与 f 相等,随后 g 随 f 因除以一个和比例 b_4/a_4 有关的分母而变化。如果 a_4 等于 b_4 则 g 和 f 相等(检验这个关系)。图 8.45 给出了对于不同的 b_4/a_4 的值,g 随 f 的变化关系。

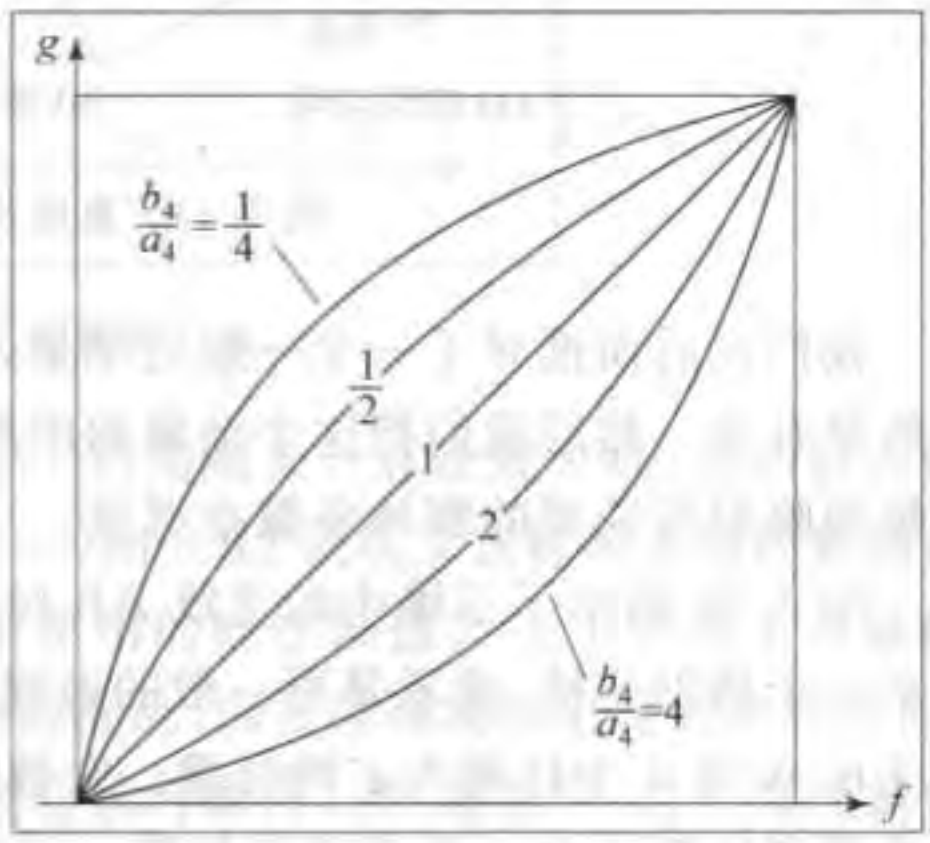


图 8.45 g 和 f 的关系

这样我们来到最后一步,看看 $r(f)$ 映射到三维面片上的点 $R(g)$ 在哪里。把等式(8.17)简写成 $R(g)=A(1-g)+Bg$,然后代数化简得到第一项(请自行推导):

$$R_1 = \frac{\mathrm{lerp}\left(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f\right)}{\mathrm{lerp}\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)} \tag{8.19}$$

对于分量 R_2 和 R_3，有相似的等式(其中的分母和 R_1 的相同)。这是个关键的结果。它说明了三维中哪个点(R_1, R_2, R_3)和屏幕坐标系中位于两点 a、b 之间(比例为 f)的点对应。所以任何三维面片顶点上的随其线性变化的属性值(比如纹理)就可以依旧线性地变化。

我们关心的变换矩阵 M 的两种分类是：

- 变换是仿射变换。
- 变换是透视投影变换。

a. 当变换是仿射变换，a_4 和 b_4 都等于 1(为什么?)，所以上述的等式可以简化。f 和 g 一致并且上面提到的 R_1 等于 $\mathrm{lerp}(A_1, B_1, f)$。我们可以直接总结这点为：

事实：如果 M 是仿射变换，直线 ab 上等距的间隔对应到直线 AB 上依旧是等距的间隔。

b. 当 M 表示从视点坐标系到剪裁坐标系的透视投影变换时，第四个分量 a_4 和 b_4 不再是 1。我们在第 7 章推导过矩阵 M，它的基本形式在等式(7.10)中给出：

$$M = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其中 c 和 d 是使伪深度值正确的常量。对于这个矩阵，向量 $M(A,1)^T$ 是什么？是 $\tilde{a} = (NA_1, NA_2, cA_3 + d, -A_3)$，关键的部分是 $a_4 = -A_3$。这也就是在相机坐标系下点在 z 轴方向上的位置——也就是点在视点前方的深度。

所以 a_4 和 b_4 的相对大小在线段近大远小的透视中处于核心位置：它们分别代表了 A 和 B 沿相机视平面法向的"深度"。如果 A 和 B 有相同的深度(比如它们所在的平面平行于相机视平面)，则沿线段没有透视扭曲，所以 g 和 f 实际上是一样的。图 8.46 中，从视点发出连接视平面上等距点的光线(以等距 f 增加)，到达原始三维面片上对应的就是不等距的点。对于这个例子，A 比 B 更近，导致 $a_4 < b_4$，所以 g 的增量会随着 A 到 B 而变大。

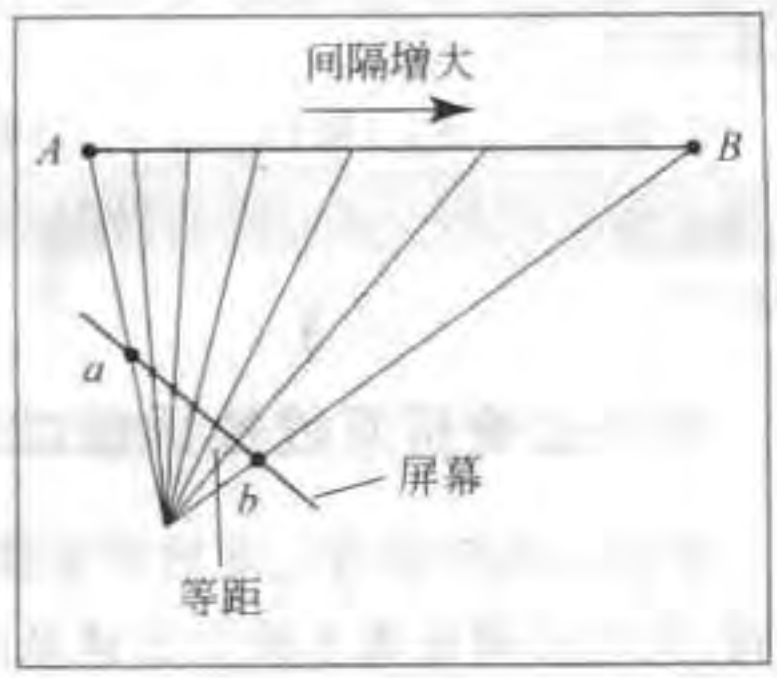


图 8.46　a_4 和 b_4 的值点的深度有关

增量渲染

我们现在把所有的这些放到一起，来为要被渲染的面片上的每个点计算正确的坐标(s, t)。图 8.47 给出了一个正在被渲染的谷仓模型的面片。面片左边端点是 a 和 b。面片在扫描线 y 上的范围是从 x_{left} 到 x_{right}。我们需要分别为 x_{left} 和 x_{right} 找到相应的纹理坐标(s_{left}, t_{left})和(s_{right}, t_{right})，这样它们就可以在扫描线上被插值。考虑要计算扫描线 y 上 s_{left} 的值 $s_{\mathrm{left}}(y)$。我们知道点 a 的纹理坐标 s_A 和点 b 的纹理坐标 s_B，因为这两个值是随顶点 A 和 B 经流水线传递过来的。如果扫描线 y 的位置在 y_{bott} 和 y_{top} 之间的比例是 f(也就是 $f = (y-$

$y_{bott})/(y_{top}-y_{bott})$),则从等式(8.19)中我们知道正确的纹理坐标是:

$$S_{left}(y)=\frac{\mathrm{lerp}\left(\frac{S_A}{a_4},\frac{S_B}{b_4},f\right)}{\mathrm{lerp}\left(\frac{1}{a_4},\frac{1}{b_4},f\right)} \tag{8.20}$$

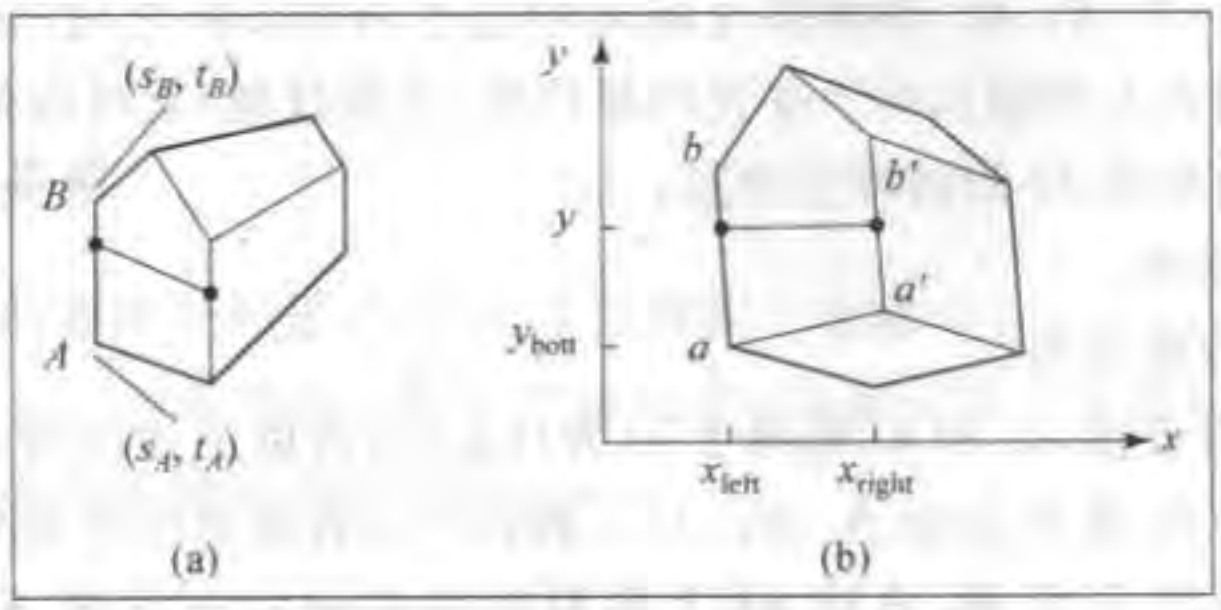


图 8.47 渲染面片上的纹理

相应地我们也能求出 t_{left} 的纹理坐标。注意 s_{left} 和 t_{left} 的式子中有相同的分母——$1/a_4$ 和 $1/b_4$ 的一个线性插值。分子是纹理坐标除以 a_4 和 b_4 后的线性插值。这有时也称作"理性线性绘制(rational linear rendering)"【Heckbert91】或者"双曲线插值(hyperbolic interpolation)"【Blinn92】。当 f 增加的时候为了高效的计算(s,t),我们需要存储以下的值 s_A/a_4, s_B/b_4, t_A/a_4, t_B/b_4, $1/a_4$ 和 $1/b_4$,因为随着像素的变化这些值不会改变。分子和分母项都可以找到基于 y 的增量计算方法,就像我们在 Gouraud 明暗处理(参见等式(8.15))中做的。但是为了求出每个 y 对应的 s_{left} 和 t_{left},等式中的除法是必须要做的。

(s_{right}, t_{right})也是用类似的方法计算出来的。它们的分母是基于点 a' 和 b' 所产生的 a'_4 和 b'_4 的值。

一旦(s_{left}, t_{left})和(s_{right}, t_{right})被找到,扫描线就可以被填充了。对于 x_{left} 到 x_{right} 中的每个点 x 所对应的 s 和 t 值,同样地再次使用双线性插值计算得到(求 x 的 s 值的表达式是什么?)。

为什么使用双线性插值以及这样计算的代价多大?

使用双线性插值正确的渲染纹理意味着什么?并且剪裁这步是否需要改进?我们将会看到,我们必须在流水线上传递某些附加的信息,并且计算一些不同于上面看到的值。

图 8.48 给出了流水线的一个改进。不同的点都标注了在那一步可用的信息。每个顶点 V 附带着纹理对(s,t)和一个顶点法向。顶点通过模式视点矩阵(法向乘以这个矩阵的逆的转置)变换,产生视点坐标系下的顶点 $A=(A_1, A_2, A_3)$和法向 $\mathbf{n}'$。计算着色时使用这个法向,产生颜色 $\mathbf{c}=(c_r,c_g,c_b)$。纹理坐标(s_A,t_A)(等于(s,t))一直跟随着顶点 A。顶点 A 随后经过透视变换,产生 $\bar{a}=(a_1, a_2, a_3, a_4)$。纹理坐标和颜色 $\mathbf{c}$ 并没有改变。

如第 7 章的讨论,处理完根据**视椎**(view volume)的剪裁。正像图中所提到的,这步有可能使一些顶点消失同时生成其他的顶点。比如当创建了一个顶点 D,我们必须决定它的位置(d_1,d_2,d_3,d_4),并且还有相应的颜色和纹理坐标。依据剪裁算法的原理,位置 d_i 通过线性插值得到:$d_i=lerp(a_i,b_i,t)$,$i=1,\cdots,4$,对于某个 t。请注意第四个分量 d_4 也是这样

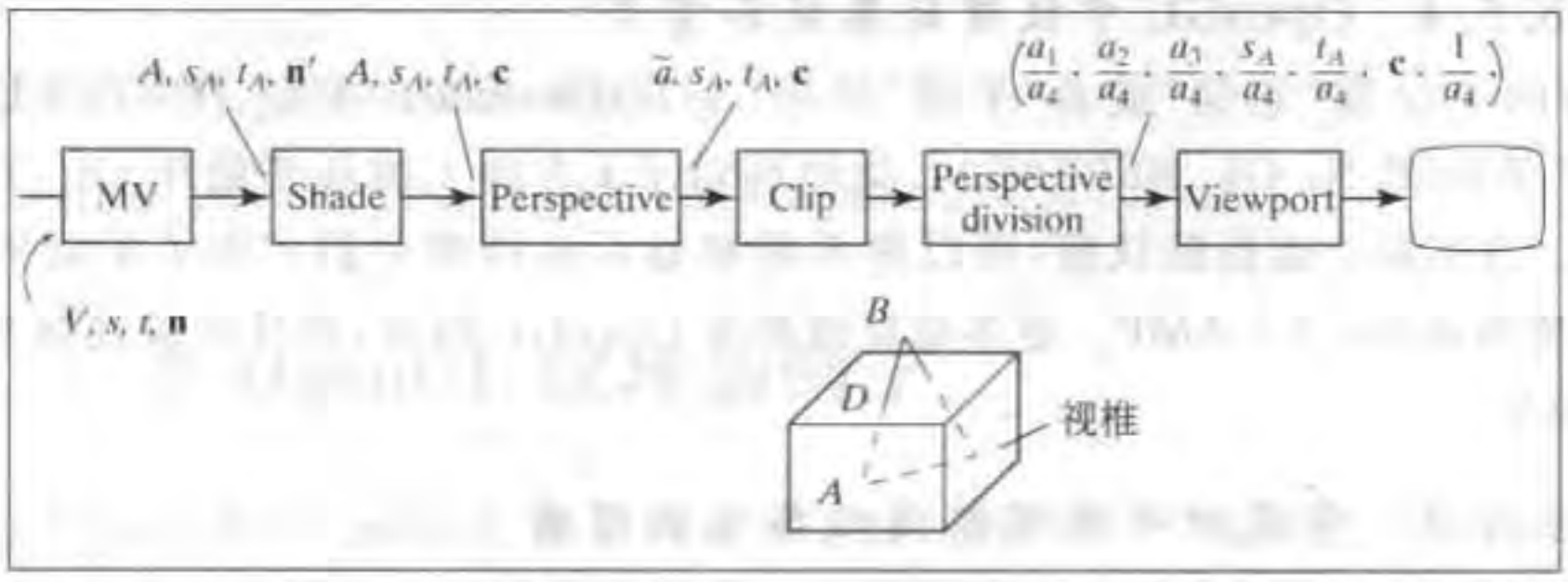


图 8.48　包含双线性插值的图形管线

形成的。很自然地，在这里同样使用线性插值来生成颜色分量和纹理坐标(这样做的原因在练习中有讨论)。因此，在剪裁之后，面片仍然由一些附带着颜色和纹理坐标的顶点组成。对于点 A，这些信息可以存为一个数组$(a_1, a_2, a_3, a_4, s_A, t_A, \mathbf{c}, 1)$。最后的一项 1 是被添加进去的，我们将在下一步用到它。

现在做完了透视除法之后，为了做双曲线插值我们需要一些项，比如 s_A/a_4，$1/a_4$ 和 t_A/a_4(见等式(8.20))，为了做双曲线插值我们把数组中的每一项都除以 a_4 得到了数组$(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$。(我们也会除以颜色分量以获得更加真实一些 Gouraud 明暗处理。参见练习)。前三项$(x, y, z) = (a_1/a_4, a_2/a_4, a_3/a_4)$表示了在标准化设备坐标系上点的位置。第三个分量是伪深度值。前两个分量将会通过视窗变换缩放和平移。为了简化符号，我们将继续称其屏幕坐标为(x, y, z)。

所以最后渲染器对于每个要渲染面片的顶点收到了数组$(x, y, z, 1, \cdots)$。现在使用等式(8.20)中的双线性插值渲染纹理就很简单了：每个顶点所需要的值 s_A/a_4 和 $1/a_4$ 都已知。

练习

练习 8.5.2　带纹理的网格模型的数据结构

讨论表示以下两种类型的网格模型所需要指定的数据类型：

a. 每个表面应用不同的纹理。

b. 一个纹理被卷绕包裹于整个网格。

写出每种数据类型所需要的模板，当网格表示一个立方体时给出每种类型表示下的一个数据样例。

练习 8.5.3　伪深度值的计算是正确的

证明在 8.4 节宣称的在计算伪深度值时使用线性插值(而不是双曲线插值)是正确的。假设点 A 投影到 a，点 B 投影到 b。用线性插值我们为投影后的点计算其伪深度值 $\mathrm{lerp}(a, b, f)$，把它作为这个点的第三个分量。只有当这样得到的结果等于插值点 $\mathrm{lerp}(A, B, g)$(对于某个适当的 g 值)投影后的伪深度值时，我们上面的做法才是对的。证明实际情况下这种做法是对的(提示：对被投影点的第三个分量使用等式(8.16)和等式(8.17))。

练习 8.5.4 OpenGL 中纹理的卷绕和剪裁

在 s 方向上设置"卷绕"或者"平铺"使用：glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)。类似地对于 t 方向上的环绕使用 GL_TEXTURE_WRAP_T。这实际上也是默认值，所以你不需要显示地指明它们。为了关掉平铺把 GL_REPEAT 替换成 GL_CLAMP。更多信息请参考 OpenGL 网站，并且实验观察 OpenGL 不同设置的效果。

练习 8.5.5 剪裁时对纹理做线性插值的理由

用视锥剪裁面片时经常会产生新的顶点。我们必须为每个点设定纹理坐标。假设新点 V 形成在顶点 A 到顶点 B 中间比例为 f 的位置。另外假设 A 点纹理坐标是 (s_A, t_A)，B 点类似。证明假设纹理被贴在一个平的面片上，设定 V 的纹理坐标为 $(\mathrm{lerp}(s_A, s_B, f), \mathrm{lerp}(t_A, t_B, f))$ 是合理的。

练习 8.5.6 双曲线插值的计算量

比较纹理坐标的双曲线插值和线性插值所需要的计算量。假设乘除法所需要的计算量 10 倍于加减法。

8.5.3 什么是纹理调和

纹理映射得到的数值在渲染计算中如何使用？为了产生不同的视觉效果，我们将看到三种常见的使用方法。我们简单地使用等式(8.5)中计算亮度灰度值的情况来学习。对于彩色情况只是同样的计算，分别应用于红色、绿色和蓝色分量。

1. 创建一个发光体

这是最简单的计算方法。设置可见的亮度 I 等于该点的纹理值：

$$I = \mathrm{texture}(s,t)$$

(或者再乘以某个常量)。物体看上去像发光或者荧光：小的纹理值表示发出很少的光，大的纹理值表示更多的光。不需要其他的光照计算。

(对于彩色光，红、绿、蓝分量分别地赋值：例如，红色分量 $I_r = \mathrm{textrue}_r(s,t)$)

为了让 OpenGL 这样贴纹理，设定：

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

2. 调和反射系数画纹理

我们先前知道一个物体的颜色就是其漫反射光的颜色(当处于白光中)。因此我们可以通过变化漫反射系数，也许还有环境光反射系数，来使得纹理看上去就像被画在物体表面上一样。我们称为纹理函数逐点地调和反射系数。这样对于适当的 s 和 t 值，我们替换等式(8.5)为：

$$I = \mathrm{texture}(s,t)[I_a\rho_a + I_d\rho_d \times lambert] + I_{sp}\rho_s \times phong^f$$

因为 Phong 镜面反射光的颜色是光源的颜色而不是物体的颜色，所以高光不依赖于

纹理。

为了让 OpenGL 这样贴纹理，设定：

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

8.5.4　一个 OpenGL 纹理的例子

为了说明如何调用 OpenGL 提供的纹理工具，我们用一个应用程序来做说明，这个程序显示一个旋转的立方体，立方体的每个面上都贴有不同的图片。图 8.49 是这个程序所实现的动画的截图。纹理映射到每个面片使用双曲线插值。

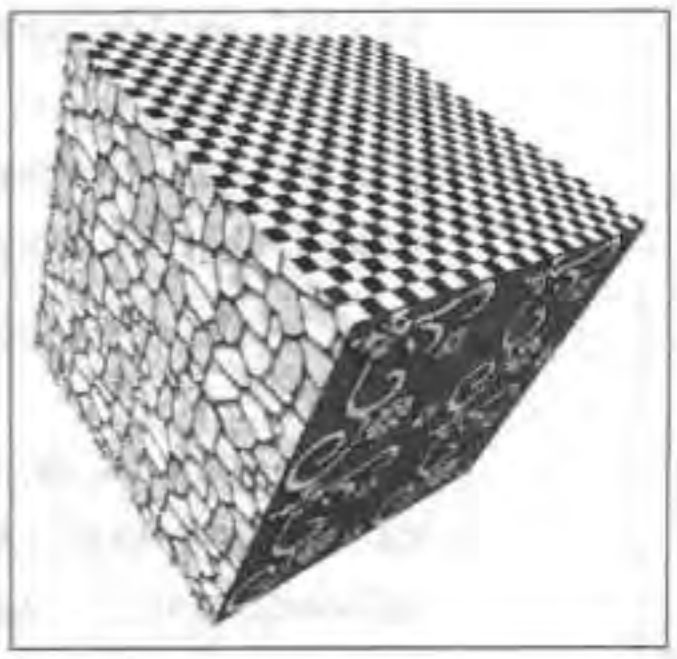


图 8.49　演示程序生成的贴有纹理的立方体

图 8.50 给出了这个程序的代码。它使用了一些 OpenGL 函数建立纹理并且把它们贴到立方体的面上。这里还列出了许多不同的参数可供映射纹理的时候使用。图中给出的参数可以很好地工作，小心调整一些参数（参见 OpenGL 的手册）可以改善图像质量或者提高速度。我们这里只介绍基本的关键步骤。

添加纹理的第一个任务是在内存中创建纹理的**像素图像**（pixmap）。OpenGL 实用的纹理存储在**像素图像**（pixel maps，或者简称为 pixmaps）中。这点将在第 9 章有深入的讨论，并且类 RGBpixmap 的设计就是为了提供建立和操作纹理图像的工具。这里我们把一个纹理图像看成一个简单的像素值的数组，每个像素值都是一个存储红、绿、蓝颜色值的三元组。

```
#include "windows.h"
#include "GL/gl.h"
#include "GL/glu.h"
#include "GL/glut.h"
#include "RGBpixmap.h"

//##########################GLOBALS##########################
RGBpixmap pix[6];     //make six (empty) pixmaps
float xSpeed=0, ySpeed=1,  xAngle=0.02, yAngle=0.02;
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<myinit>>>>>>>>>>>>>>>>>>>>>>>>>>>>.
void myInit(void)
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);          //background is white
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);

    pix[0].makeCheckerboard();                  //make pixmap procedurally
    pix[0].setTexture(2001);                    //create texture
```

图 8.50　旋转的纹理立方体程序

```
    pix[1].readBMPFile("Mandrill.bmp");          //make pixmap from image
    pix[1].setTexture(2002);                     //create texture

    pix[2].readBMPFile("Mandrill.bmp");          //make pixmap from image
    pix[2].setTexture(2003);                     //create texture

    pix[3].makeCheckerboard();                   //make pixmap from image
    pix[3].setTexture(2004);                     //create texture

    pix[4].readBMPFile("red.bmp");               //make pixmap from image
    pix[4].setTexture(2005);                     //create texture

    pix[5].makeCheckerboard();                   //make pixmap from image
    pix[5].setTexture(2006);                     //create texture


    glViewport(0, 0, 640, 480);                  //set up the viewing system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 640.0/ 480, 1.0, 30.0);//set camera shape
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslated(0.0, -0.2, -4);                 //move camera back
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<display>>>>>>>>>>>>>>>>>>>>>>>>>>
void display(void)
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glPushMatrix();
    glRotated(xAngle, 1.0,0.0,0.0);
    glRotated(yAngle, 0.0,1.0,0.0); //rotate

    glBindTexture(GL_TEXTURE_2D,2001);   //top face: 'fake' checkerboard
    glBegin(GL_QUADS);
    glTexCoord2f(-1.0, -1.0); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(-1.0, 2.0); glVertex3f(-1.0f, 1.0f,  1.0f);
    glTexCoord2f(2.0, 2.0); glVertex3f( 1.0f, 1.0f,  1.0f);
    glTexCoord2f(2.0, -1.0); glVertex3f( 1.0f, 1.0f, -1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2002);   //right face: mandrill
    glBegin(GL_QUADS);
```

图 8.50 (续)

```
    glTexCoord2f(0.0, 0.0); glVertex3f(1.0f, -1.0f,  1.0f);
    glTexCoord2f(0.0, 2.0); glVertex3f(1.0f, -1.0f, -1.0f);
    glTexCoord2f(2.0, 2.0); glVertex3f(1.0f,  1.0f, -1.0f);
    glTexCoord2f(2.0, 0.0); glVertex3f(1.0f,  1.0f,  1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2003);      //left face: mandrill
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0f, 1.0f,  -1.0f);
    glTexCoord2f(2.0, 0.0); glVertex3f(-1.0f, 1.0f, 1.0f);
    glTexCoord2f(2.0, 2.0); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(0.0, 2.0); glVertex3f(-1.0f,  -1.0f,  -1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2004);      //bottom face: 'fake' checkerboard
    glBegin(GL_QUADS);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0, -2.0); glVertex3f(1.0f, -1.0f,  -1.0f);
    glTexCoord2f(-2.0, -2.0); glVertex3f( -1.0f, -1.0f,  -1.0f);
    glTexCoord2f(-2.0, 1.0); glVertex3f( -1.0f, -1.0f, 1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2005);      //box front: red
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 2.0); glVertex3f(1.0f, 1.0f, 1.0f);
    glTexCoord2f(2.0, 0.0); glVertex3f(1.0f, -1.0f, 1.0f);
    glTexCoord2f(0.0, -2.0); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(-2.0, 0.0); glVertex3f(-1.0f, 1.0f, 1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2006);   //box back: 'fake' checkerboard
    glBegin(GL_QUADS);
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0f, 1.0f, -1.0f);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0f, -1.0f, -1.0f);
    glEnd();

    glFlush();
    glPopMatrix();
    glutSwapBuffers();
    glutPostRedisplay();
}
```

图 8.50（续）

```
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<spinner>>>
```

然后为每个对象读入所需的纹理图像。最后把每个对象传递给 OpenGL 来定义纹理。

生成函数纹理

我们首先使用方法 makeCheckerBoard()创建一个棋盘纹理。棋盘的样子为我们所熟悉而且非常容易创建，同时方方正正的几何形状也使其成为测试正确性的好纹理。程序为 pix[0]生成棋盘像素图像使用：pix[0].makeCheckerboard()。方法本身的实现如下：

```
void RGBpixmap:: makeCheckerboard()
{   //make checkerboard pattern
    nRows=nCols=64;
    numPixel=new RGB[3*nRows*nCols];
    if (!numPixel) {cout <<"out of memory!"; return;}
    long count=0;
    for(int i=0; i <nRows; i++)
        for(int j=0; j <nCols; j++)
        {
            int c=(((i/8)+(j/8))%2)*255;
            numPixel[count].r=c;                    //red
            numPixel[count].g=c;                    //green
            numPixel[count++].b=0;}                 //blue
        }
}
```

它创建了 64×64 的数组，每个像素都是 RGB 的三元组。OpenGL 一直到版本 1.5 都要求纹理像素图像的宽和高都必须是 2 的整数次幂。OpenGL 2.0 已经去除了这个限制。像素图像在内存中的布局就是一个长的字节数组：从下到上一行接一行，每行中从左到右。每个像素都被赋值为(c,c,0)，其中每 8 个像素 c 值在 0 和 255 中跳变一次(我们在练习 2.3.1 中使用了类似的跳变方法)。棋盘的两个颜色是黑(0,0,0)和黄(255,255,0)。函数保存了像素图像中第一个像素的地址，它在后面将会传递给 glTexImage2D()来为 OpenGL 创建真正的纹理。

一旦生成了像素图像，我们必须把它和一个单独的正整数值绑定到一起(作为它的名字)，这样在 OpenGL 中就能够准确地引用它。我们随意地为示例中的 6 个纹理命名为 2001,2002,…,2006。通过执行一些 OpenGL 的调用来创建纹理，我们把这些调用封装到方法中：

```
void RGBpixmap::setTexture(GLuint textureName)
{
    glBindTexture(GL_TEXTURE_2D, textureName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
                 GL_UNSIGNED_BYTE, pixel);
}
```

通过调用 glBindTexture()为要生成的纹理绑定名字。正像我们将要看到的，当这个调

用执行之后,它就会设定这个纹理为活动的纹理。

调用 glTexParameteri()指定了,当纹理需要被放大或者缩小时,使用坐标距离像素中心最近的那个纹理元素来填充像素。这样做速度快但会导致走样。我们将在第 9 章更深入地讨论图像滤波和反走样。最后,调用 glTexImage2D()将纹理图像和当前的纹理联系起来。这个调用描述了纹理是由 RGB 字节三元组组成的二维图像,还给出了宽和高以及位图首字节(像素)的内存地址。

从已有图像生成纹理

OpenGL 不支持读取图像文件并且在内存中创建像素图像。方法 readBMPFiel()给出了一个简单的读取 BMP 图像到像素图像的方法,具体实现可以在本书配套网站上找到。例如:

```
pix[1].readBMPFile("mandrill.bmp);
```

读取文件 mandrill. bmp 并且创建像素图像 pix[1]。

一旦像素图像被建立,就可以用 pix[1]. setTexture()函数把像素图像传递给 OpenGL 以生成纹理。

纹理映射必须通过 glEnable(GL_TEXTURE_2D)启用。另外,还需要使用 glHint(GL_PERSPECTIVE, GL_NICEST)要求 OpenGL 正确的(使双曲线插值)渲染纹理,这样当面片相对观察者旋转的时候,纹理才会正确的贴在面片之上。

纹理的创建、启用和设定插值方法只需在初始化过程中做一次。随后每次显示的时候,纹理都会被正确的处理。在 display()中,立方体通过 xAngle 和 yAngle 被旋转,并且画出其六个面。这只需要简单的为每个面绑定正确的纹理,并且在 glBegin()/glEnd()中对应好面片顶点的三维坐标和其纹理坐标,就像我们在代码中展示的一样。

当立方体的渲染(在屏幕外)完成了,glutSwapBuffers()被调用使得新的帧变为可见。控制动画是使用回调函数(callback function)spinner()作为"空闲函数"。当系统空闲的时候——没有需要相应的用户输入,spinner 就会被自动的调用。它小小地改变立方体的旋转角度然后再次调用 display()。最终实现的动画效果就是立方体在旋转,这样各个面片就会反复的出现在视野中或者转出视野。

8.5.5 在曲面上卷绕纹理

我们已经看到了如何把纹理贴到平面上。现在我们将看看如何在曲面上卷绕纹理,比如一个啤酒罐或者一个象棋子。我们假定物体是网格模型,它是由大量小的平的面片组成。正如 8.5.1 节末尾提到的,网格的每个顶点有其相应的纹理坐标(s_i, t_i),虽然我们可能不知道它们。所以主要的问题是为网格的每个顶点找到适当的纹理坐标(s, t)。

我们给出映射纹理到圆柱状物体上和球状物体上的例子,看看对于每种情况都是如何处理的。

例 8.5.1 在罐头上包裹标签

假设我们想把一个标签裹在一个圆柱体上,如图 8.51 所示。我们自然地会想到圆柱坐

标系。标签位于方位角 θ_a 到 θ_b、z 轴 z_a 到 z_b 之间。圆柱体是用多边形网格表示，所以它的表面是矩形的条带，见图 8.51(b)。对于每个面片的顶点 V，我们必须找到合适的纹理坐标 (s_i, t_i)，保证纹理上正确的"条带"被映射到表面。

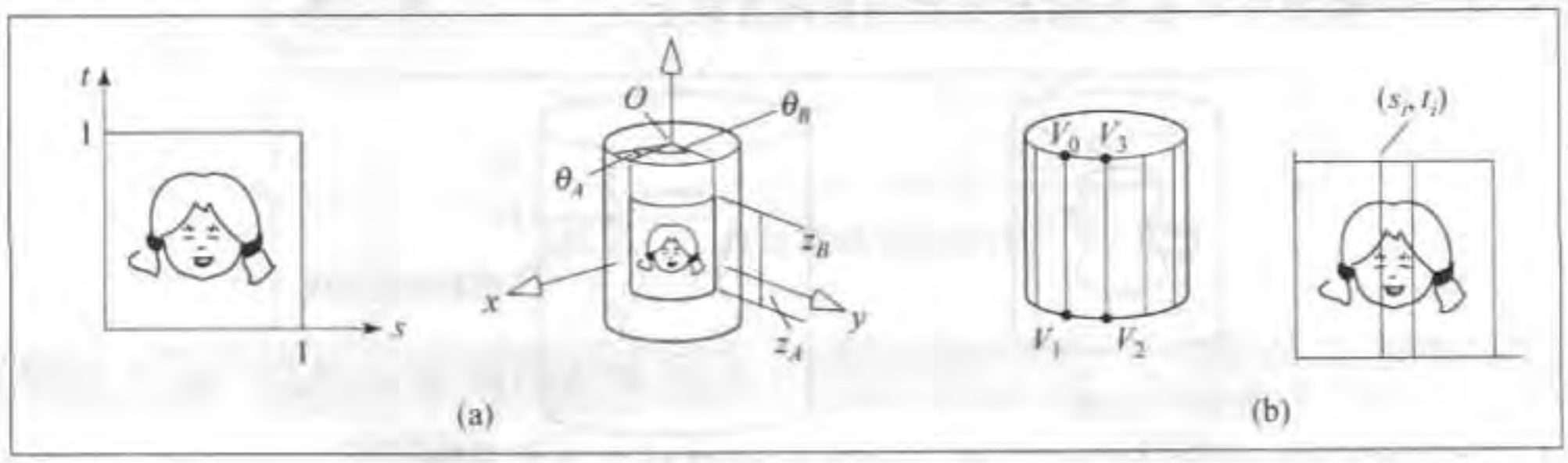


图 8.51　将标签包裹在圆柱上

该几何体足够简单以至于解决方法可以直观地得到。这里 (s,t) 和圆柱体上一点的方位角、高度 (θ,z) 之间存在一个直接的线性关系：

$$s = \frac{\theta - \theta_a}{\theta_b - \theta_a}, \quad t = \frac{z - z_a}{z_b - z_a} \tag{8.21}$$

因此如果有 N 个面片围绕成圆柱体，第 i 个面片左边的方位角是 $\theta_i = 2\pi i/N$，则它左上角顶点的纹理坐标 $(s_i, t_i) = ((2\pi i/N - \theta_a)/(\theta_b - \theta_a), 1)$。其他三个顶点的纹理坐标用类似的方法可以得到。将每个面片的顶点和 (s,t) 值联系起来只需要简单地在建模过程中添加一些代码即可(见练习)。

当物体不是一个简单的圆柱体时，这个过程会变得更加复杂。我们后面将看到如何把纹理映射到更一般的旋转曲面上。

例 8.5.2　把标签收缩包裹到一个旋转曲面上

回顾第 6 章，旋转曲面是通过一个特征曲线 $(x(v), z(v))$ 定义的，如图 8.52(a)，曲面的结果——一个花瓶，是通过参数 $P(u,v) = (x(v)\cos u, x(v)\sin u, z(v))$ 给出。物体的曲面是通过一系列沿着轮廓线分布的面片表示的(见图 8.52(b))。所以给定的面片 F_i 有四个顶点 $P(u_i, v_i)$，$P(u_{i+1}, v_i)$，$P(u_i, v_{i+1})$，$P(u_{i+1}, v_{i+1})$。我们需要为这些顶点找到对应的纹理坐标 (s,t)。

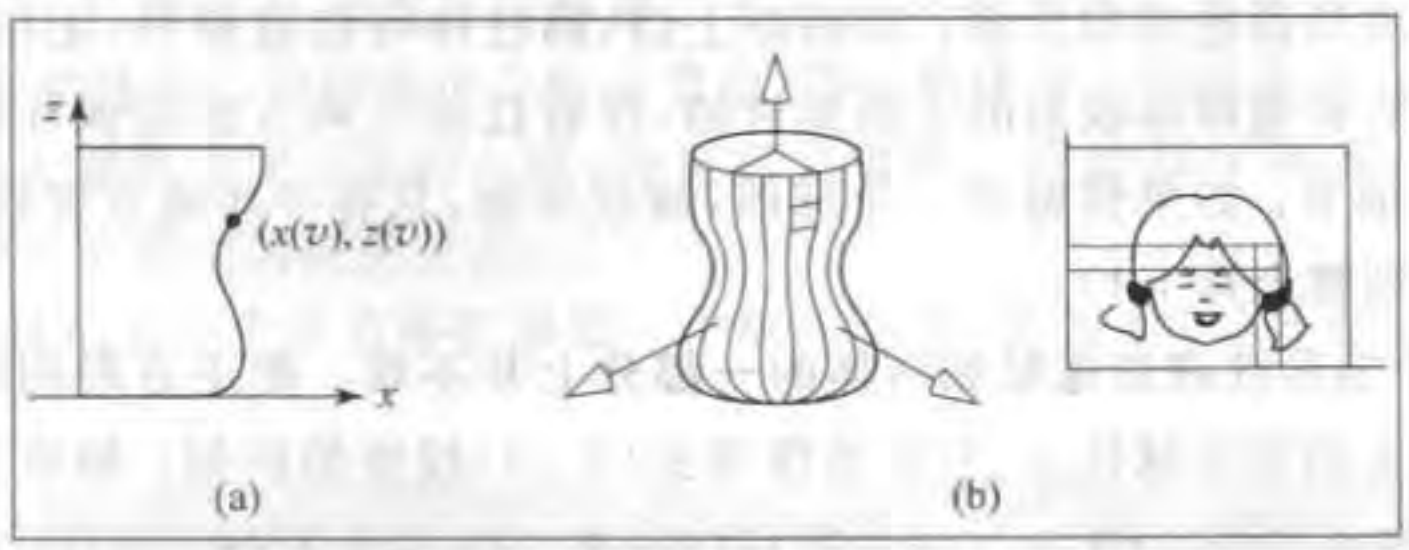


图 8.52　将标签包裹在花瓶上

和上个示例一样，一种自然的做法是根据等式(8.23)使 s 和 t 线性的随 u 和 v 变化。这等价于把纹理卷绕贴在一个假想的包裹着花瓶的橡胶圆柱体上(见图 8.53(a))，然后让

圆柱体收缩,这样纹理就被水平和竖直地分割直到它们接触到花瓶的表面。这种方法称作"收缩包裹",由 Bier 和 Sloane【Bier86】提出,文章中他们还讨论了把纹理映射到不同类型形状上的几种方法。可以把收缩映射看成沿着假想圆柱体的法向(见图 8.53(b)):纹理上的顶点 P_i 对应的模型点 V_i 位于从 P_i 出发的法向方向上。

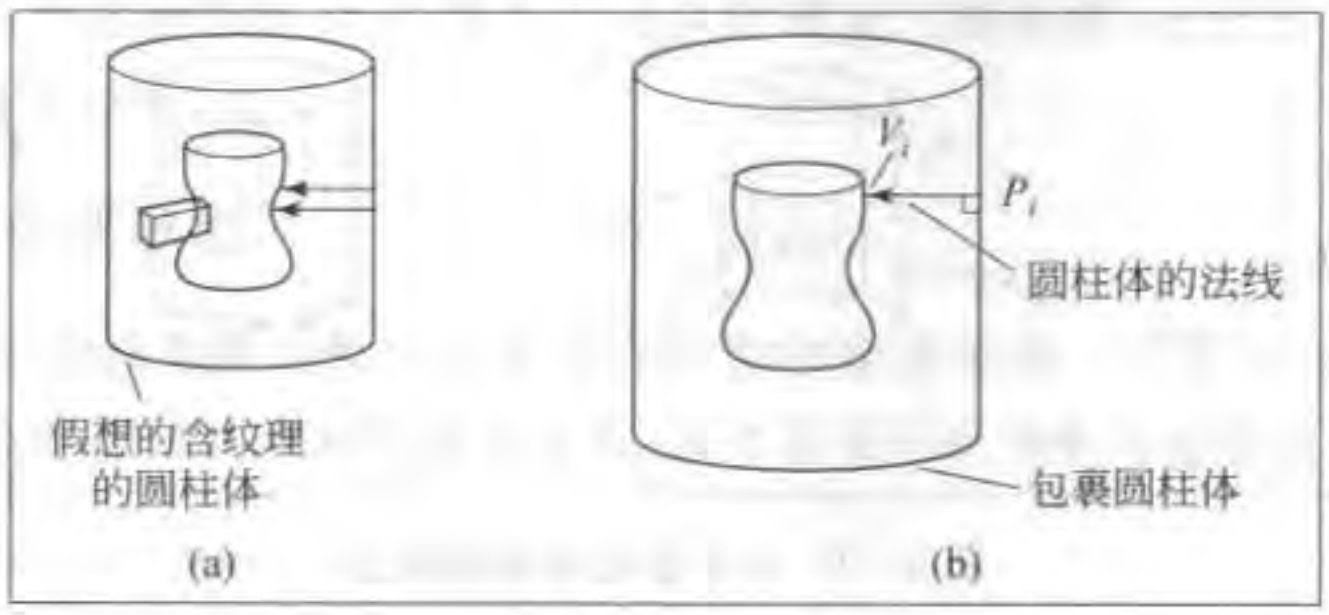


图 8.53 将纹理收缩包裹到花瓶上

收缩包裹对圆柱形状的物体有效,虽然如果特征曲线是复杂的形状,纹理的图像有可能被扭曲。

Bier 和 Sloane 提出了联系假想圆柱上的纹理点和物体顶点的几种其他方法。图 8.54 给出了其中的两种。

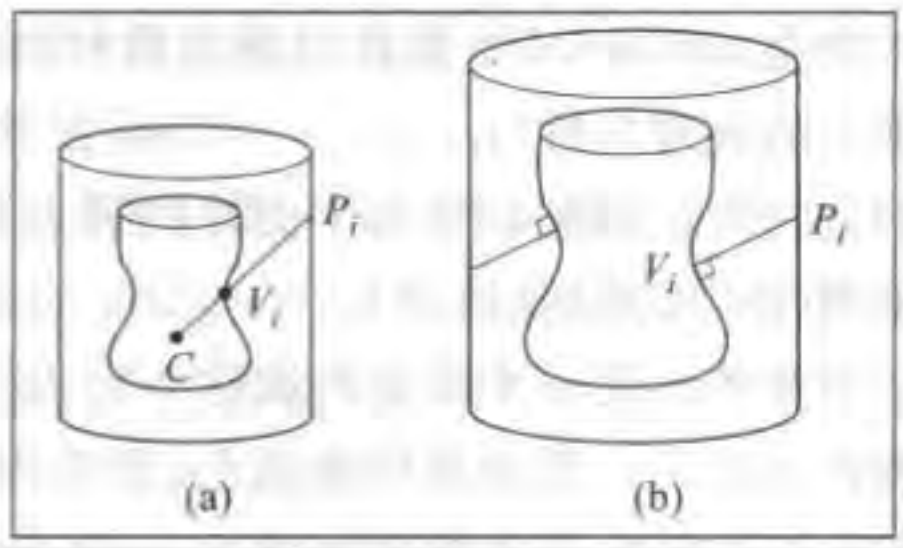


图 8.54 从假想圆柱上到物体的不同映射方法

图 8.54(a)中,一条直线从物体的中心点 C,通过顶点 V_i 和圆柱体相交于 P_i 点。图 8.54(b)中,使用了 V_i 点处的物体表面法向:P_i 是从 V_i 出发沿这个法向与圆柱的交点。要注意的是,这三种联系纹理和物体顶点的方法可能会产生非常不同的结果,因为效果依赖于物体的形状(见练习)。设计者需要根据物体的形状和纹理的属性选取最合适的方法(对于象棋棋子的兵使用哪种方法合适?)。

例 8.5.3 映射纹理到球体上

映射矩形纹理到圆柱体很简单:从拓扑上讲,圆柱体可以被剪开,无扭曲地展成平面。球体却不同。像所有做球体映射的人所知道的,没有任何一种方法能保证够把球体展成平面后保持所有的细节:如果你剪开一个球体,铺成平面,总有一些地方要被严重的拉伸(想象一个棋盘映射到整个球上!)。

但是,把一个矩形纹理图像贴到球体的**一部分**上并不难。把正方形的纹理映射到经度 θ_a 到 θ_b 维度 ϕ_a 到 ϕ_b 的部分球体上,只需要像等式(8.23)线性的映射:如果顶点 Vi 位于(θ_i, ϕ_i),其对应的纹理坐标$(s_i, t_i) = ((\theta_i-\theta_a)/(\theta_b-\theta_a), (\phi_i-\phi_a)/(\phi_b-\phi_a))$。图 8.55 给出了一个图片贴到部分球体上的结果,只能看到少量的扭曲。

图 8.55(b)说明了如何把纹理覆盖到整个球体上:把八个三角形纹理映射到球体的八个部分。

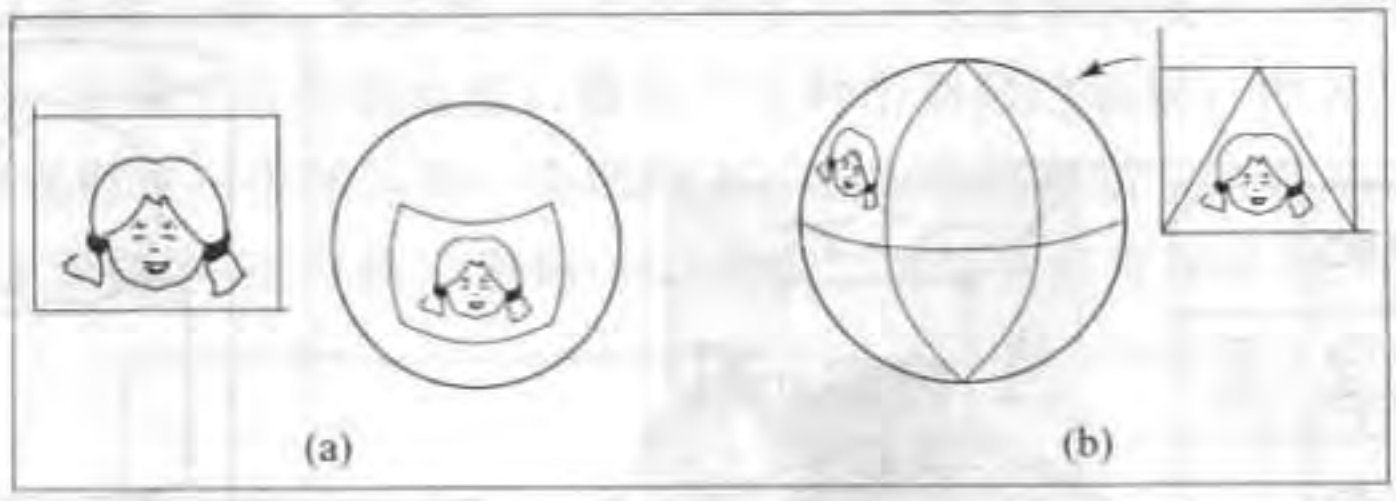


图 8.55　将纹理映射到球体

例 8.5.4　映射纹理到球状物体上

我们在前面的几段中介绍了为圆柱状物体添加纹理。但是一些物体更像球体而不是圆柱。图 8.56(a)给出了一个多面球体，它的表面是五边形和六边形。我们可以设计一些五边形和六边形的纹理，并且手工把它们贴在每个面上，但是对于一些场景我们更希望用一个纹理包裹住整个多面球体。

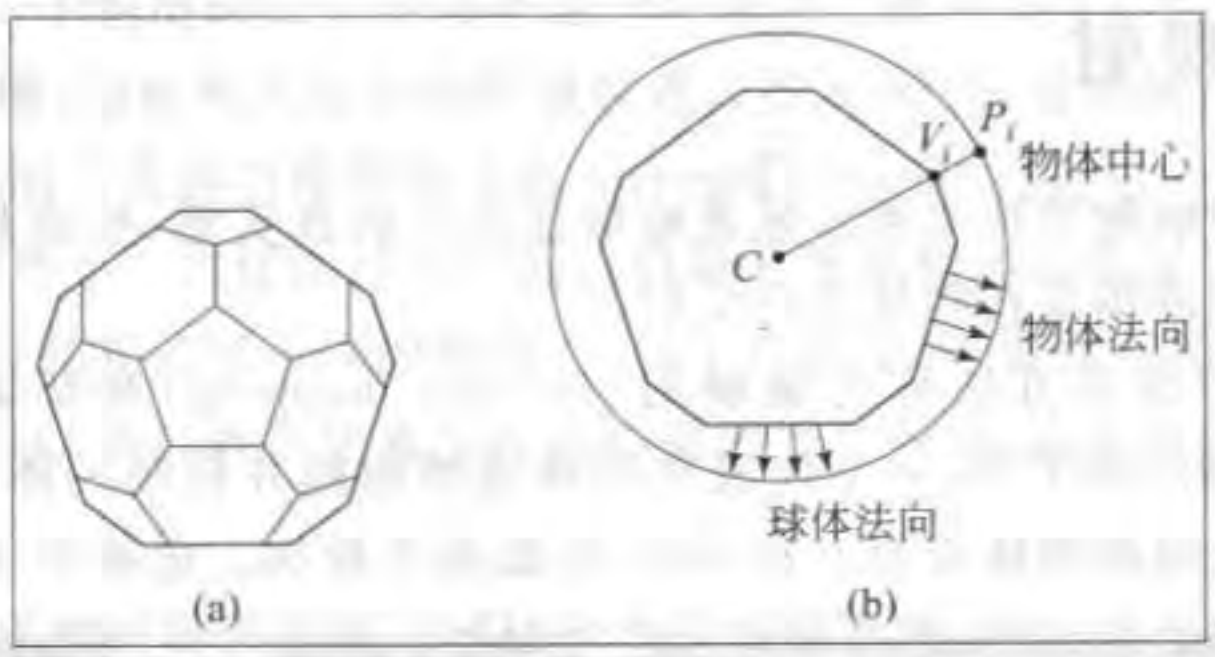


图 8.56　球状物体

自然地，我们可以在球状物体外包裹一个拥有纹理的假想球体(而不是一个假想圆柱)，使用上面讨论的某个建立关联的方法进行贴图。图 8.56(b)给出了被这样一个球体包裹的多面球体。这里列出了三种联系纹理点 P_i 和物体顶点 V_i 的方法：

- 物体中心：P_i 在物体中心 C 到顶点 V_i 的连线上。
- 物体法向：P_i 是从 V_i 出发沿面片法向与球体的交点。
- 球体法向：V_i 是从 P_i 出发言沿球面上 P_i 点法向与物体的交点。

(**问题**：如果物体中心和球体中心恰巧重合，那么物体中心和球体法向两种方法是否一样?)物体中心法大多数情况下更好一些，并且容易实现。正如 Bier 和 Sloane 所说的，其他两种方法经常产生不希望的渲染结果。

图 8.57(a)显示了一个立方体扩展到一个纹理图上，图 8.57(b)显示了该纹理包裹该立方体的情况。

练习

练习 8.5.7　如何联系 P_i 和 V_i

图 8.58 中的旋转曲面 S 由一个圆柱体和其上的一个球体组成。整个物体被一个拥有棋盘纹理的圆柱体包围。画出使用以下三种对应方法得到的最终贴图结果：

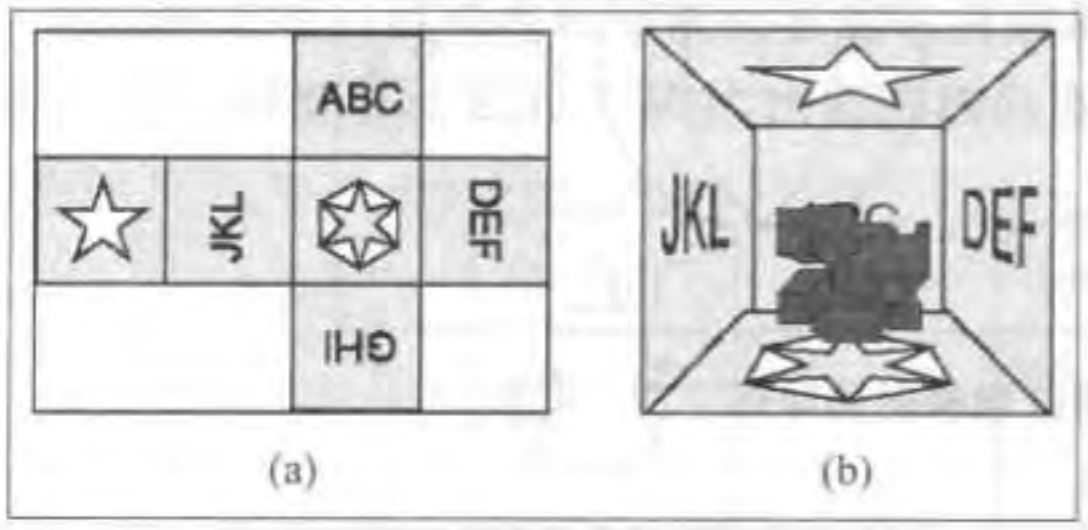


图 8.57 使用包围立方体

图 8.58 假想圆柱体包裹的旋转曲面

- 收缩包裹
- 物体中心
- 物体法向

8.5.6 反射映射

一类被称为**反射映射**的技术可以显著地增强图片的真实感,特别是在动画中。基本的想法是看到物体周围的场景在物体上的反射。

两种主要的反射映射方法称为**铬映射**(chrome mapping)和**环境映射**(environment mapping)。在**铬映射**的例子中,一个代表周围环境的粗糙并模糊的图像被反射在物体上,就像你可能在电镀上铬的物体表面上看到的,比如拖车栓钩。电视中大量出现的闪亮的字母和标志在空间中移动的动画,都是使用了铬映射加上偶尔出现的高光后产生的奇特效果。图 8.59 给出了这样一个例子。图 8.59(a)是一个铬映射的例子;整个图像清晰呈现了一个闪亮的机器人坐在一个凳子上。机器人上有暗和亮的区域,但是我们并不能看出来反射本身的图像,反射只是提供了围绕在物体周围的大概的情况。

图 8.59 举例

(a) 铬映射;(b) 环境映射

在**环境映射**(首先由 Blinn 和 Newell【Blinn76】提出)的例子中,我们在物体的反射中可以看到一个可辨识的周围环境的图像。从这样的反射中,我们得到了更多的视觉线索,特别是当物体移动的时候。每个人想必都看过这个经典的照片,一个宇航员在月球上行走,在他

的面罩上反射出了月球飞船。相似地，在电影中你有时候会看到演员带着墨镜的特写（比如黑暗武士，译者注：星球大战中的角色），墨镜中反射出周围的场景。图 8.60 所给的两个例子中，一个餐馆被反射在一个球体和一个圆环上。餐馆的纹理被包裹在一个巨大的球体上，这个球体内部包含着物体，这样纹理坐标(s,t)对应于球体的经度和纬度。

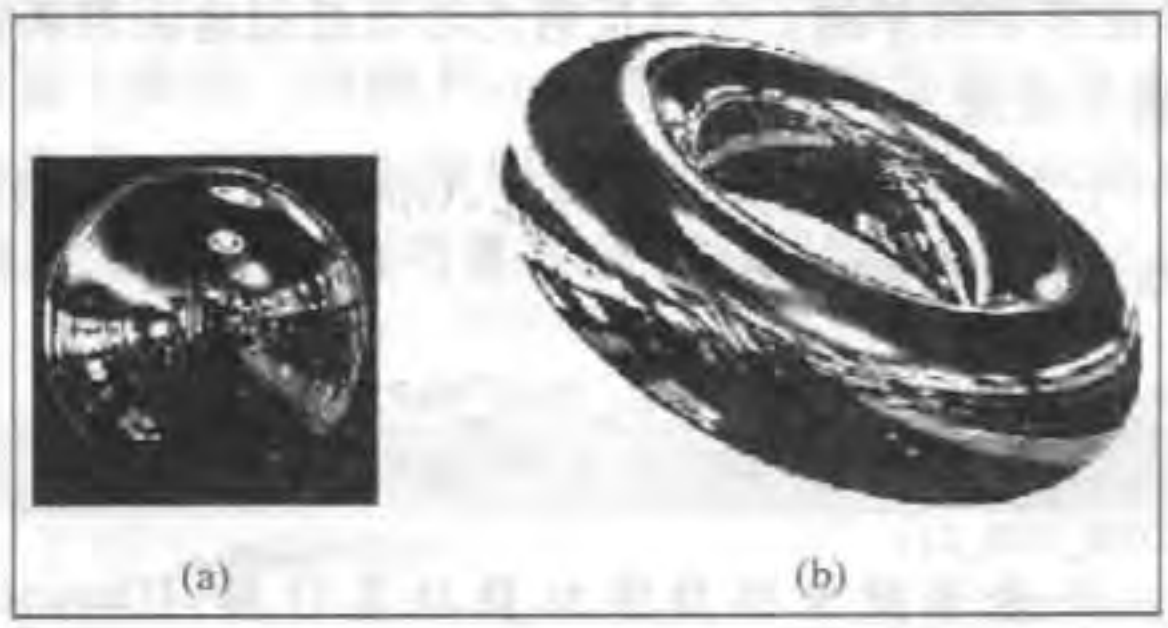


图 8.60　环境映射举例

图 8.61 使用了一个包围的立方体而不是一个球体。图 8.61(a)给出了这个环境，它由 6 张从室内观察到的墙、地板和天花板的图像组成。图 8.61(b)给出的一个光泽物体反射了房间的不同部分。包围立方体的使用首先由 Greene【Greene86】提出，一般来讲它的效果的扭曲将会比包围球的更小。六张图片可以通过渲染 6 张从物体的位置分别向不同方向观察的图片生成（当然这时候先去掉中间的物体）。对于每张图片，要摆好相机的位置，设好窗口的大小。另外，纹理可以通过数字化真实相机拍摄的照片获得，真实相机从各个主要方向拍照房间或场景。

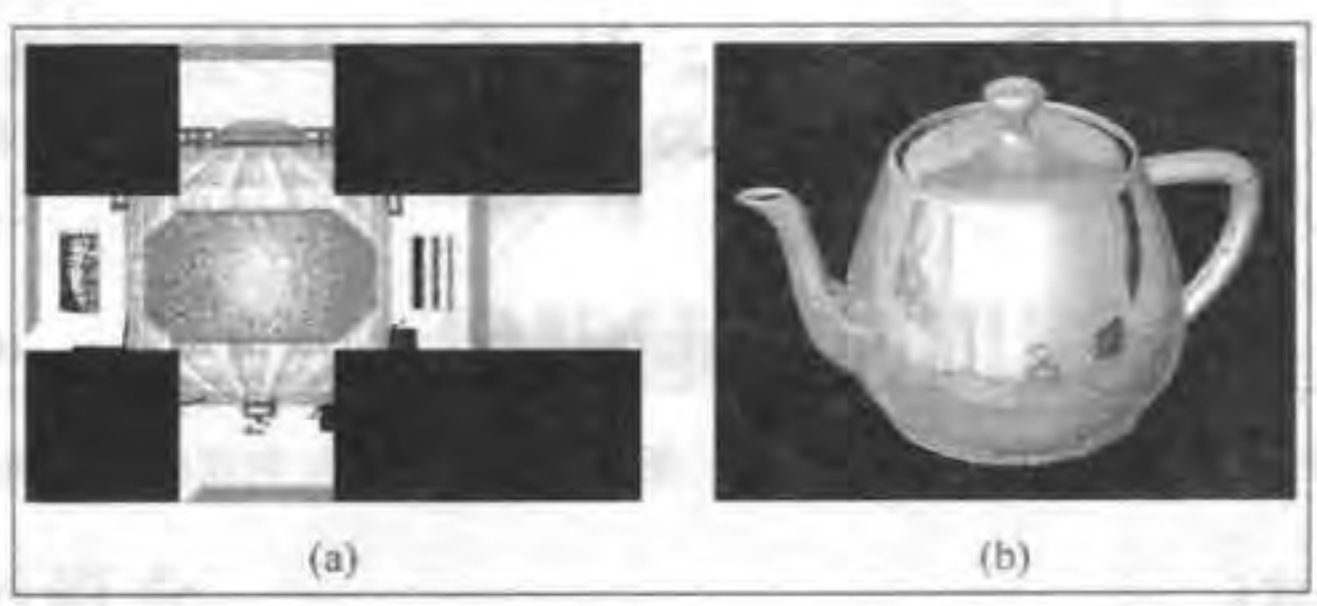


图 8.61　基于包围立方体的环境映射

在带光泽物体移动的动画中，铬映射、环境映射与法向纹理映射有很大的区别。前者反射的图像会在移动的物体上流动，而法向纹理贴图将会被固定在物体上和其一起移动。如果一个带光泽的球体旋转一个固定角度，法向纹理贴图也会随着球体旋转一个角度，而反射映射确保持不变。

如何实现铬映射和环境映射

环境映射是如何工作的？你看到的光泽物体上的点 P 是环境中正好在 P 点正确反射到你眼睛的颜色。为了找到这个方向，跟踪从视点到 P 的光线，并且计算反射光的方向，再

沿着这个光线找到与它(在包围立方体或者球体上)相交的纹理。图 8.62 画出了一个光线从视点出发射向点 P。如果这个光线的方向是 $\mathbf{u}$ 并且 P 点的单位法向是 $\mathbf{m}$,我们根据等式(8.2)可以知道反射光的方向是 $\mathbf{r}=\mathbf{u}-2(\mathbf{u}\cdot\mathbf{m})\mathbf{m}$(回顾第 4 章)。反射光沿 $\mathbf{r}$ 方向前进直到与有纹理的假想曲面相交。如果假设光泽物体位于正中心,并且和包围立方体或球体相比非常小,则这个计算是非常简单的。这样反射光可以近似地认为从物体中心射出,它的方向 r 可以被直接用来索引纹理。

对于纹理包裹在一个大的包围立方体的情况,OpenGL 提供了进行这种近似环境映射的工具。通过设定 s 和 t 的映射模式实现:

```
glTexGenf(GL_S,GL_TEXTURE_GEN_MODE, GL_CUBE_MAP);
glTexGenf(GL_T,GL_TEXTURE_GEN_MODE, GL_CUBE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

这样当顶点 P 和其单位法向 $\mathbf{m}$ 传入流水线时,OpenGL 计算一个适合的纹理坐标(s,t)用来索引在包围立方体上的纹理。这只在物体面片的每个顶点上计算一次,绘制面片时依旧为中间的其他点插值计算纹理坐标(s,t)。

OpenGL 是如何快速计算出正确的纹理坐标(s,t)的呢?如图 8.63(a)所示,它首先(使用上面的等式)找到(在视点坐标系中的)反射方向 $\mathbf{r}$,其中 $\mathbf{u}$ 是(视点坐标系中的)从视点指向物体上顶点 V 的单位向量,$\mathbf{m}$ 是 V 点的法向。

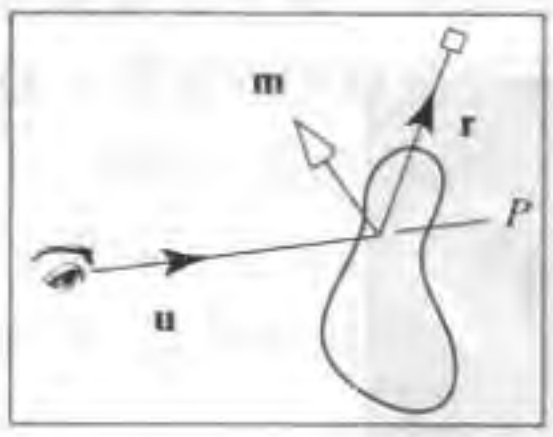


图 8.62 计算反射光的方向

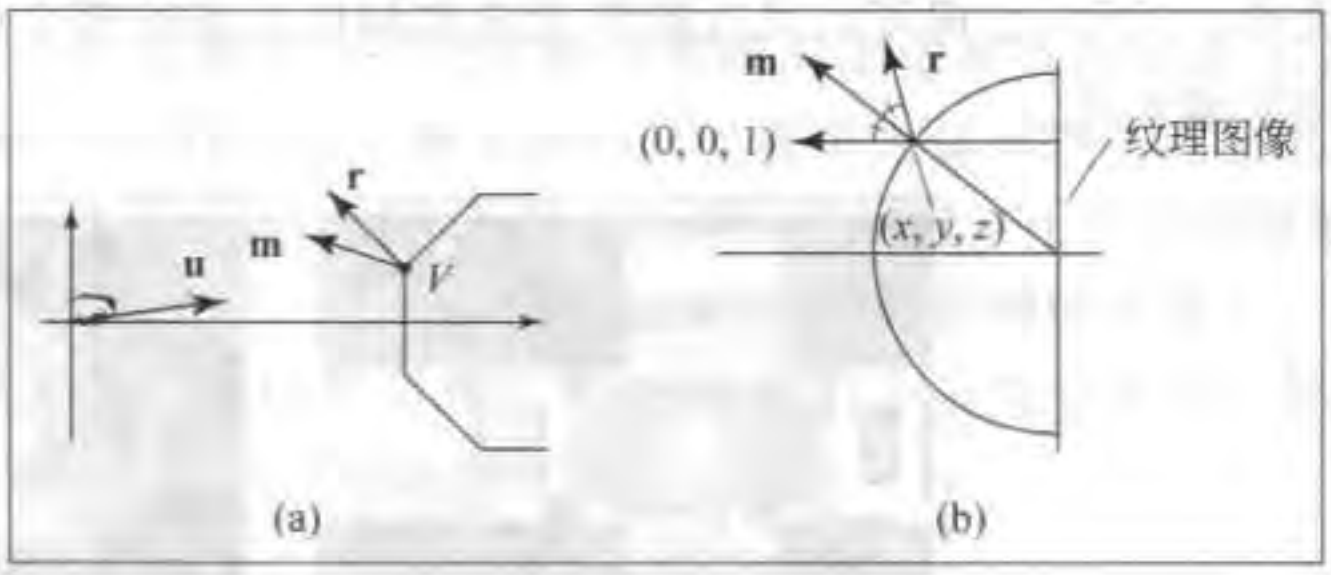


图 8.63 OpenGL 中纹理坐标的计算

然后它使用等式:

$$(s,t)=\left(\frac{1}{2}\left(\frac{r_x}{p}+1\right),\quad \frac{1}{2}\left(\frac{r_y}{p}+1\right)\right) \tag{8.22}$$

其中 p 是一个神奇的缩放因子 $p=\sqrt{r_x^2+r_y^2+(r_z+1)^2}$。这项的推导放在练习中。我们还必须预计算一个纹理表示所在的环境。这个纹理相当于把一个完全反射的球体(译者注:原著中是立方体)置于环境中,视点从无限远处观察球体上的图像【Haeberli93】。这样就把视点背后的半球环境映射到了纹理中央的圆,视点前方的半球环境映射到围绕这个圆的圆环(画图说明这点)。如果视点变化了位置,这个纹理必须要被重新计算。

回顾在前言中我们介绍过 OpenGL 中**扩展**的概念;OpenGL 中实现的许多扩展可以使得环境映射变得非常简单。

使用环境映射模拟高光

OpenGL 中可以使用反射映射在物体上产生镜面反射高光。可以创建拥有一个集中的高亮度的斑点的纹理。反射映射把这个高光画到物体表面使它看上去像环境中某处有一个真正的光源。这样产生的高光可以比 Phong 镜面反射项附加 Gouraund 明暗处理产生的高光更集中更富于细节。回顾 Phong 模型项只会在面片顶点处计算，很容易丢掉落在两个顶点之间的高光。反射映射使用的纹理坐标(s,t)是在顶点形成然后进行插值的，所以即使顶点索引的纹理坐标恰巧落在亮点的周围，这个亮点也会在面片上恰当地被渲染出来。

练习

练习 8.5.8　OpenGL 的计算环境映射纹理坐标的方法

推导等式(8.22)。图 8.36(b)给出了(视点坐标系下)相关的向量。视点位于方向(0,0,1)的远处。球体的半径是 1，位于 z 的负半轴。假设光从方向 r 射过来，击中球体上的点(x,y,z)。球面上这点的法向是(x,y,z)，并且沿方向 **r** 过来的光正好反射到方向(0,0,1)。这意味着法向必须等分 **r** 和(0,0,1)，或者说必须是这两个向量相加后的方向，所以$(x,y,z)=K(r_x,r_y,r_z+1)$，对于某个 K：

a. 证明当 $K=1/p$ 时，法向量是单位长度。其中 p 在等式(8.22)中给出。

b. 证明因此$(x,\ y)=(r_x/p,r_y/p)$；

c. 假设当前纹理图像坐标 x 和 y 的范围是 -1 到 1。证明为什么我们希望在点(x,y,z)反射后所看到的等于纹理图像在(x,y)的值。

d. 证明，现在纹理坐标的范围是 0～1——也就是 *OpenGL* 使用的，我们希望在(x,y)看到的值在纹理图像上的坐标(s,t)就是等式(8.22)给出的。

8.6　给物体添加阴影

8.6.1　阴影简介

阴影使图像看上去更真实。由日常经验可知，一个物体投射阴影到另一个物体的方式，对于判断它们的位置提供了重要的视觉线索。图 8.64 包括了两个图像，悬浮在平面上的一个立方体和一个球体。图 8.64(a)的图没有阴影，就不可能看出立方体和球体悬浮在平面上多远的距离。作为对比，图 8.64(b)可以看到阴影，对判断物体的位置提供了有用的提示。一个阴影能够传达很多信息；近似于第二次观察物体(从接近光源的视点的角度)。

正如我们将要看到的，有一些在图像中生成阴影的方法。不幸的是，本章中所使用的工具只适用于点光源投影到平面上时产生的阴影。在第 12 章中，光线跟踪工具对于任何光源投射到任何形状表面上的情况，都能计算出准确的阴影形状。

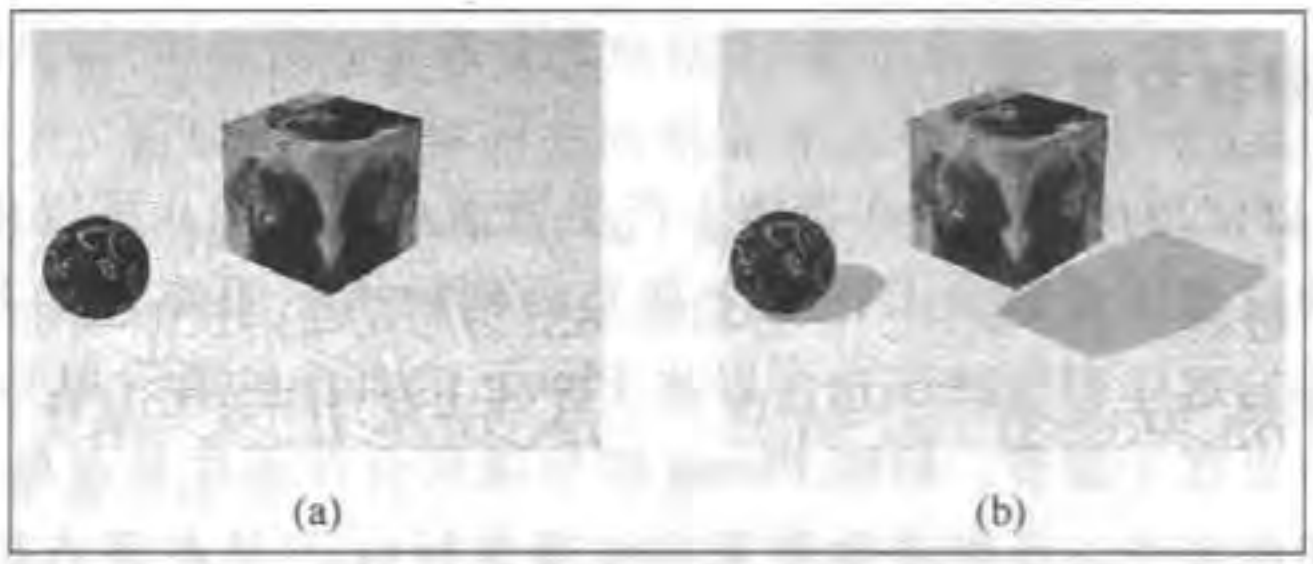


图 8.64 阴影的效果

(a) 悬浮的立方体和球体,没有阴影;(b) 有阴影的效果

作为纹理的阴影

这种技术显示了通过一个点光源投射到平面的阴影。问题是如何计算出阴影的形状。图 8.65(a)显示了一个盒子投射阴影到地板上。把光源作为投影中心,阴影的形状由盒子的每个面到地板平面的投影决定。事实上阴影是 6 个面投影的集合。图 8.65(b)显示了其中两个面的叠和:顶面投影到 top',前面投影到 $front'$(勾画出其他四个面的投影,可以看出它们的集合就是所需要的阴影)。

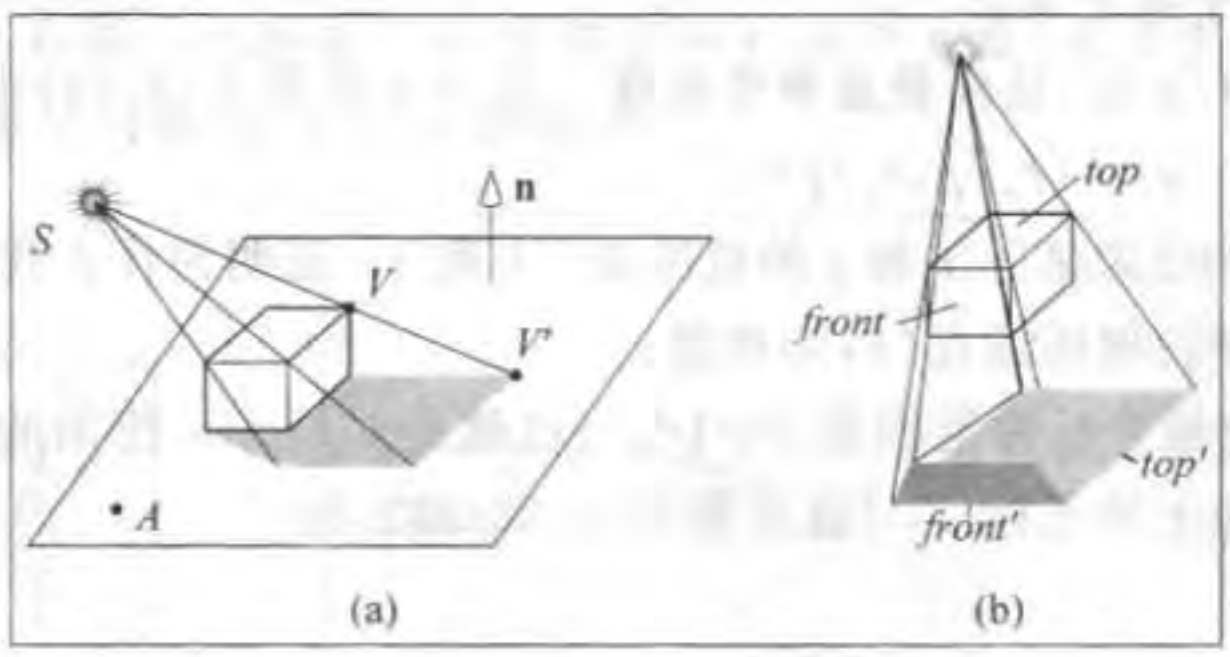


图 8.65 计算阴影的形状

这是画阴影的关键。在使用环境光、漫反射、和镜面反射光绘制平面后,只采用环境光重新绘制 6 个面在平面上的投影,就能得到阴影的正确形状和色彩。最后画盒子。(如果盒子接近平面,盒子的一部分将挡住部分阴影。)

构建"投影"面

通过 F 生成新面 F',投影它的每个顶点到平面上。我们需要一种计算出这些顶点在平面上位置的方法。假设,如图 8.65 所示,平面经过点 A,其法向量为 $\mathbf{n}$。认为投影顶点 V,产生点 V'。这里的数学原理是熟悉的:点 V' 是在 S 的光源发射的光线经过 V 到达平面的点。就像练习中所做的,这个点是:

$$V' = S + (V - S)\frac{\mathbf{n} \cdot (A - S)}{\mathbf{n} \cdot (V - S)} \tag{8.23}$$

这个练习说明怎样在同一坐标系中写成一个 V 次矩阵,这对于绘制引擎有利,就像支持矩阵乘法的 OpenGL。

练习

练习 8.6.1　阴影形状

假设一个立方体悬浮在一个平面上。这个立方体的阴影是什么形状的？如果点光源位于(a)直接在顶部上面？(b)沿着立方体的主对角线(看作是各向同性的视图)？勾画出悬浮在平面上的一个球体和一个圆柱体在各种位置的光源中的阴影。

练习 8.6.2　制作阴影面

(a) 证明：从 S 点的光源发出的光线经过顶点 V 到达平面 $\mathbf{n}\cdot(P-A)=0$ 在点 $t^*=\mathbf{n}\cdot(A-S)/\mathbf{n}(V-S)$。

(b) 证明：这就是等式(8.25)给出的点 V'。

练习 8.6.3　它等价于一个矩阵乘法

(a) 证明：V'在等式(8.23)中的表达式能够被写成矩阵乘法：$V'=M(V_x,V_y,V_z,1)^T$，这里 M 是一个 4×4 的矩阵。(b) 用 A,S 和 $\mathbf{n}$ 表达 M。

8.6.2　利用阴影缓冲区绘制阴影

另一种完全不同的绘制阴影的方法是采用深度缓冲区完成隐藏面的移除。对于每个光源，它使用一个辅助的二次深度缓冲区，称为**阴影缓冲区**。这需要大量的存储空间，但这种方法并不仅限于投射阴影到平面上。

这种方法是基于隐藏在光源后的任何点都一定位于阴影中的原则。在另一方面，如果没有物体位于点和光源之间，那么这个点不在阴影中。阴影缓冲区包含视点在光源处的场景的“深度图片”：它的每个元素记录了从光源到最近物体的连接方向的距离。

绘制分如下两个步骤。

1. 阴影缓冲区装载。

阴影缓冲区每个元素初始化为 1.0，这是可能的最大伪深度。然后，假设照相机定位在光源，扫描转换场景中的每个面，但是只检测面上的点的伪深度。阴影缓冲区的每个元素记录下能被看到的最小伪深度。

更特殊的情况时，图 8.66 显示了位于光源的“人眼照相机”看到的场景。假设点 P 在从光源到阴影缓冲区“像素”$d[i][j]$ 的光线上，金字塔上的点 B 也在这条光线上。如果金字塔在，则 $d[i][j]$记录的是到 B 的伪深度；如果金字塔不在，则 $d[i][j]$记录的是到 P 的伪深度。

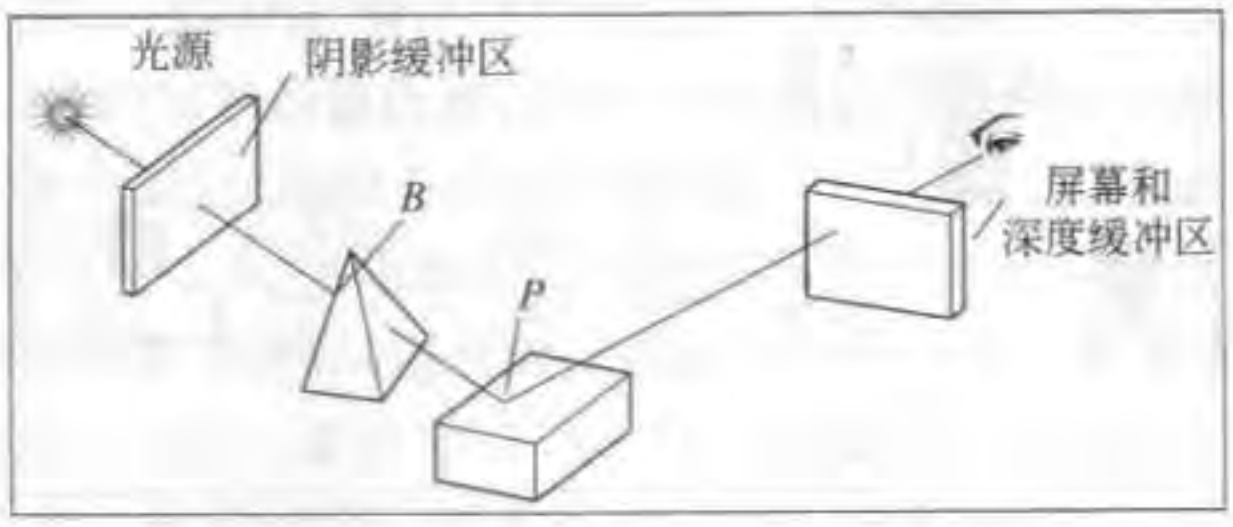


图 8.66　使用阴影缓冲区

注意阴影缓冲区的计算是与眼睛的位置无关的,因此在一个动画中若只有眼睛移动,阴影缓冲区只需要被装载一次。但是只要物体相对于光源移动,阴影缓冲区就必须被重新计算。

2. 绘制场景。

绘制场景中的每个面通常采用人眼相机。假设相机穿过 $p[c][r]$看到 P 点。当绘制 $p[c][r]$时,我们必须要找到:

- 从光源到 P 的伪深度 D;
- 在深度缓冲区被测试的索引位置$[i][j]$;
- 储存在深度缓冲区中的值 $d[i][j]$。

如果 $d[i][j]$小于 D,则点 P 在阴影中,$p[c][r]$只需要利用环境光设置。反之则点 P 不在阴影中,$p[c][r]$要使用环境光、漫反射光、和镜面反射光设置。

这些步骤怎样实现?如练习中所描述的,人眼相机视平面的每个点对应于光源相机视平面的一个点。对于每个屏幕像素,找出光源到 P 的伪深度,也包括索引$[i][j]$计算出存储在阴影缓冲区的最小伪深度。

练习

练习 8.6.4 计算来自光源的伪深度

假设矩阵 M_c 和 M_s 分别将场景中的点 P 变换到人眼相机视平面和到光源相机视平面上合适的(3D)点:

a. 叙述如何建立一个光源相机和怎样找到结果矩阵 M_s。

b. 转换:给定人眼相机视平面上的点(x,y),找到在光源相机视平面上的位置(i,j)和深度。

c. 给定(i,j),怎样确定光源相机上的索引$[i][j]$和 P 的伪深度。

练习 8.6.5 扩展光源

本章我们只考虑点光源的情况。而现实中更多的是各种类型的其他光源。如图 8.67(a)所示,这样的光源投射更复杂的阴影,有一个任何光源看不到的全影区,和一个稍亮点的部分光源可见的半影区。在图 8.67(b)部分,一个炽热的半径为 2 的球形光源照射在单元立方体上,从而投射一个阴影到墙面 W 上,准确地画出墙上的全影和半影。如你所想的,绘制阴影的算法因为扩展的光源变得复杂。参阅【Watt92】,可获得完整的介绍。

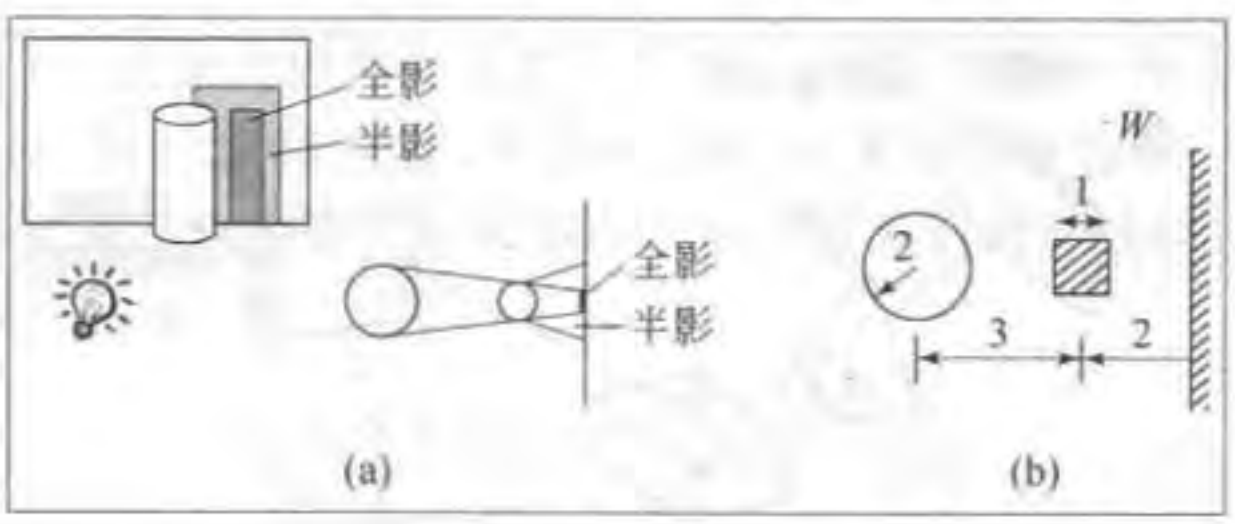


图 8.67 扩展光源的全影和半影

接下来我们将介绍图像制作的辐射度方法，并与光线跟踪算法作对比。这里我们只给出辐射度方法的简要介绍，因为大量需要解释的细节超出了本书的范围。

8.6.3　辐射度的简要介绍

回顾一下，环境光包含从物体的每个面上反射的光分量的计算，可称为“包罗万象”。它试图包括场景中许多光束在各种表面上产生的多重反射。一种可行的绘制真实感改进方法是形成一个精确的关于到达每个表面的光能量与离开表面光能量的对比的模型。然而由于跟踪每条光线的代价太大了，所以把它们合并成一个环境光分量。环境光是一种假想的光属性，现实生活中不存在。图 8.68 展示了一个光线跟踪的场景的例子：图 8.68(a)展示了一个用辐射度方法生成的场景；相对的是，图 8.68(b)采用光线跟踪，而没有用辐射度。

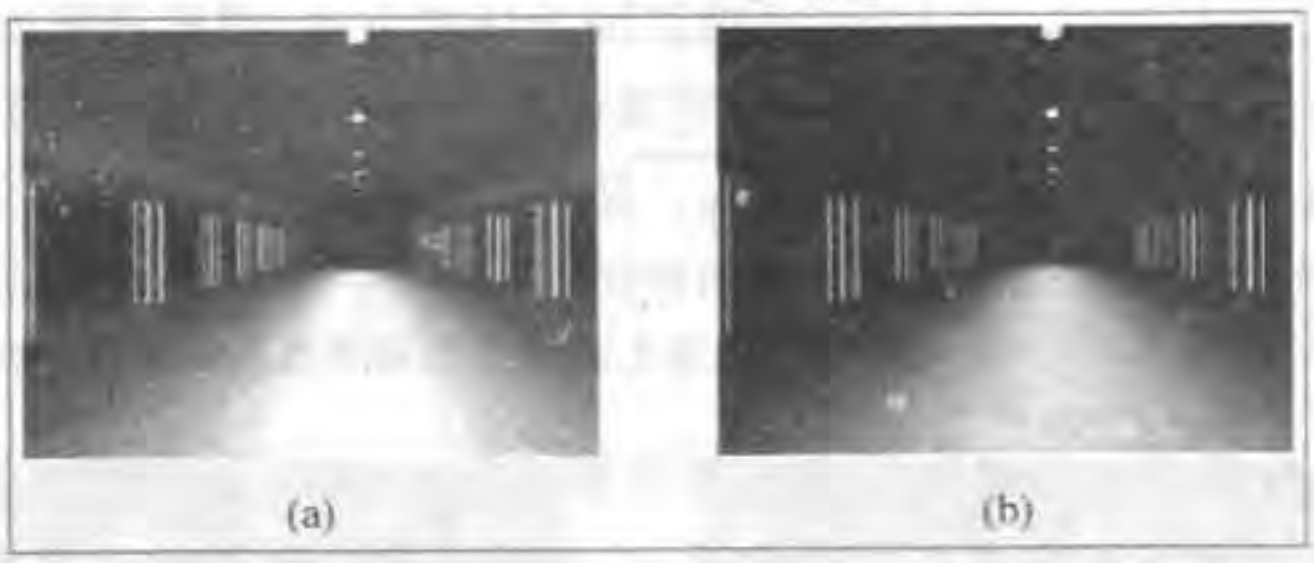


图　8.68

(a) 用辐射度；(b) 没有用辐射度

第 12 章我们将看到采用光线跟踪算法能显著提高真实感的水平，可以生成一些特殊的效果。特别是当辐射度取代光线跟踪会制造出更高等级的真实效果，但代价是增加了计算量。

一些资源可在本书网站上获取。也有一篇 Paul Nettle[18] 写的文章，对辐射度的算法及其复杂度作了精彩总结。

8.7　OPENGL2.0 和着色语言(GLSL)

回顾前言，有一个用 *OpenGL* 提高核心功能的机制：

- **扩展**—这允许对 OpenGL 做出改变和提高。那些想测试和提高 OpenGL 的公司，定义和创造扩展，例如新的算法集，函数，和有权使用新硬件的能力。

例如，可编程的着色部分(不要与阴影混淆)是一个扩展，它提供程序员一些直接调用图形卡的函数。在 OpenGL 2.0 中，可编程着色扩展成为 OpenGL 的核心。这些新功能的加入，允许程序员直接控制着色操作，避开固定的着色函数管道。最复杂的着色方法包括凹凸贴图和 3D 纹理，但是 OpenGL 着色语言(GLSL)使得程序员能轻易实现这些工作的大部分。图 8.69 是一个简化的 OpenGL 管道外加 GLSL 的简单版本。管道开始先输入多边形(顶点)的集合，接着是**顶点处理器**。顶点处理器运行**顶点着色代码**。顶点着色代码是处理

输入顶点数据的代码;数据包括位置,颜色,法线,等等。通过顶点着色代码,一个程序可完成如下任务:

- 通过视图矩阵和投影矩阵控制顶点位置转换;
- 转换法向量,如果需要的话将它们归一化;
- 生成并转换纹理坐标;
- 对每个顶点应用光照模型(环境光,漫反射,和镜面反射);
- 计算颜色。

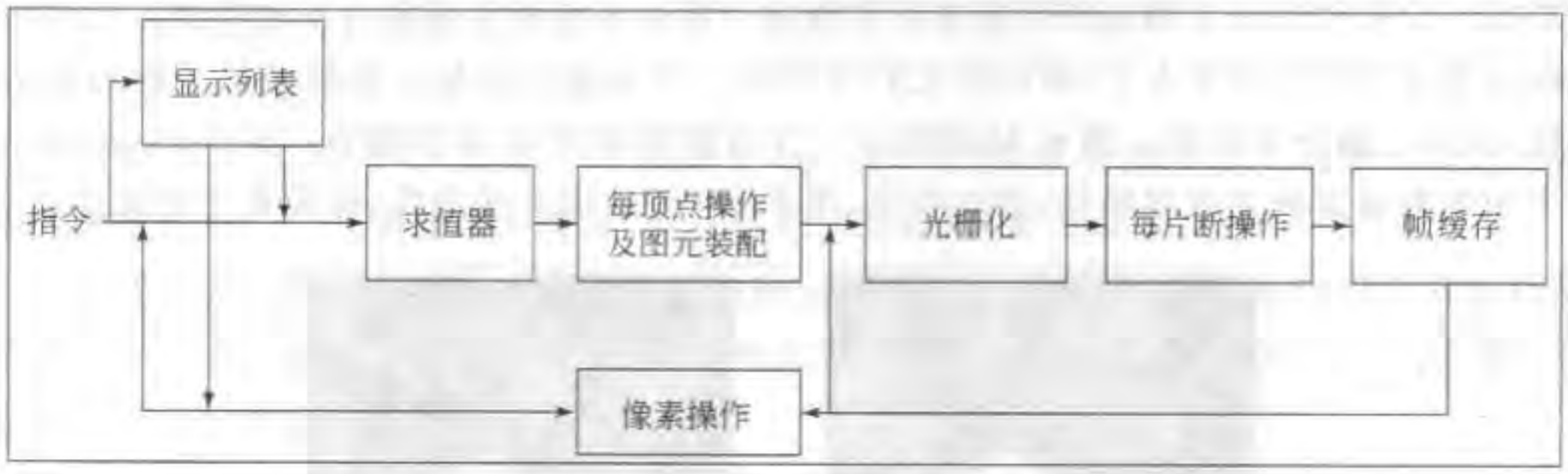


图 8.69 OpenGL 管道,加上顶点处理器和像素处理器

注意:当程序员设计一个定制的顶点着色器时,OpenGL 的顶点处理器的固定功能性被取代。如果定制的顶点着色器被激活的话,它将负责顶点处理的所有需要功能。

一旦这些顶点被转换成视平面,图元就被光栅化,并形成片断。例如,一个图元就像一个点可能覆盖屏幕上 1 个像素,或者一条线可能覆盖屏幕上 5 个像素。由点产生的片断包含一个窗口的坐标和深度信息,及与之相关联的属性包括颜色,纹理坐标,深度,等等。光栅化也确定了片断上的像素值。每个这样的属性值通过对顶点上对应属性值插值得到。一个跨越 5 个像素的线图元生成 5 个片断,这些片断的属性通过线段端点之间的对应值插值而得到。一些图元从不产生任何片断,反之另一些图元产生大量的片断。关于 OpenGL 管道在图 8.69 中"光栅化"阶段的输出,是一个片断流(这里的管道实际上是指 OpenGL 怎样操作的建议)。

8.7.1 凹凸贴图

凹凸贴图技术首次是被 Jim Blinn【Blinn88】提出的。它是一种给物体添加复杂纹理的高效方法,不需要改变物体的基本几何结构。例如,图 8.70 显示了一幅桔子粗糙表面的放大视图。然而,桔子的真实几何结构仍是一个光滑的球体,只是在球体上绘制了纹理。在每一点,纹理扰动了表面的原始法向量,从而扰动了在镜面高光计算中起重要作用的法方向。

重要的是要注意到物体的几何形状仍是一个简单的球形(很容易建模),与此相反的是扰动纹理可以是一个图像或者其他的随机纹理源,正如我们前面讨论的那样。

图 8.71 显示了另一个例子,表面看上去像是雕刻了数字,"1914—1918"。但是,物体实际上是一个光滑的矩形,刻痕通过扰动法向量产生。可以想象如果通过改变原始几何体(可能是个复杂的网格)来取得同样的效果将是多么的复杂。

图 8.70　桔子的凹凸贴图

图 8.71　平面表面上的浮雕

凹凸贴图是怎样制作的呢？给一个原始的扰动贴图，我们打算用它来扰动物体表面的法向量，我们必须一个一个像素地计算它对真实表面法向的影响。如 Blinn 所描述的那样，法向扰动要求重复使用偏导数，并在它们之间叉积。

这些计算超出了本书的范围。幸运的是某些 OpenGL 的扩展对凹凸贴图会有所帮助，使程序员免于原始的数学计算（偏导数，等等）。在 www. paulsprojects. net 有 Paul Baker 关于这方面的精彩指南。图 8.72 举了一个圆环的例子，分别采用和不采用它在指南中仔细讲解的凹凸贴图的技术。

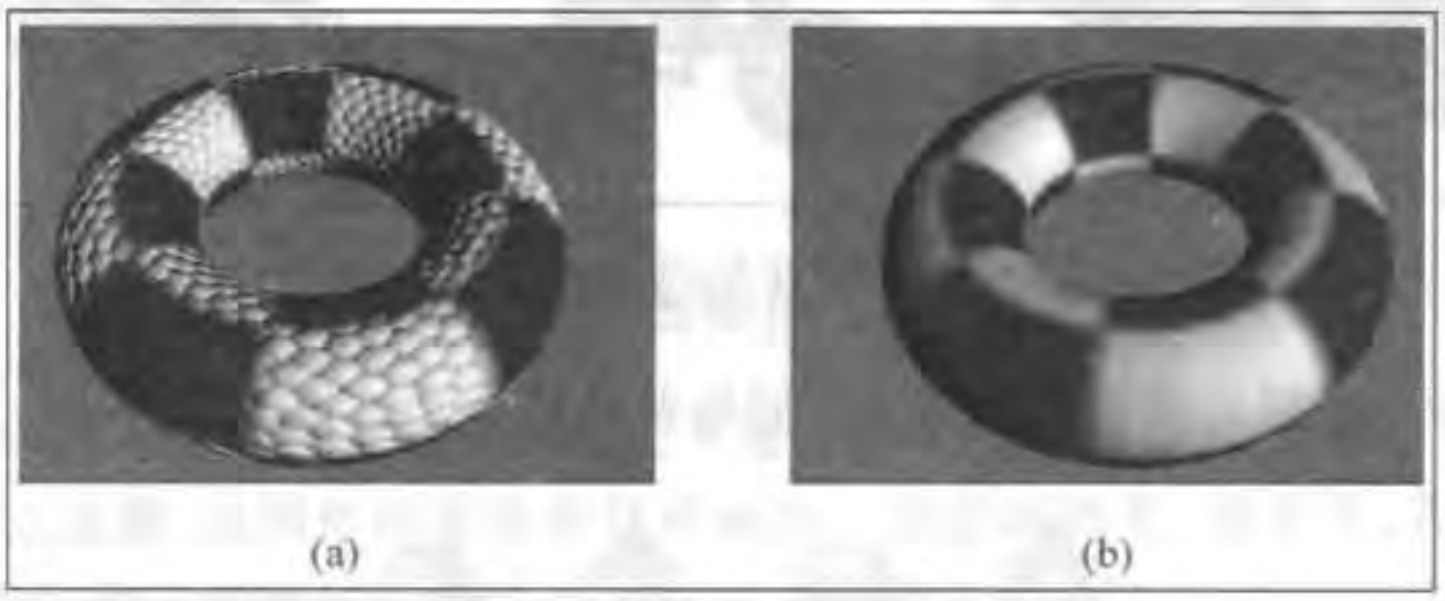


图　8.72

(a) 圆环的凹凸贴图例子；(b) 没有凹凸贴图的圆环

另一篇关于凹凸贴图的精彩文章是 NVIDIA 公司的 Mark Kilgard 写的，他原从事于 GLUT 的应用后来继续 OpenGL 的研究。Mark 的文章，"A Practical and Robust Bump-Mapping Technique for Today's GPUs"（能够在 www. nvidia. com 中找到），是一篇涉及凹凸贴图及其原理的精彩文章。

8.7.2　非真实感绘制

本书的重点是场景的真实感绘制，以求最终的图像看上去尽可能的真实。但在有些情况，真实不是图像最想得到的效果。相反，有人也许想强调一幅图像中的特殊信息，或者故意忽略一些绘制成本高和用户不感兴趣的细节。例如，有人想要制作一个类似卡通的发动机图片，如图 8.73 所示。这幅图像给人的印象是它像是手工用画笔画出来的。

在另外一些情况下有人也许想制作非常详细和准确的某一机器的技术图或一幅蓝图，不需要考虑着色、阴影、和反锯齿。这种技术被称为**笔墨绘制**（也可称作雕刻）。图 8.74 显

示了笔墨绘制帆船的画。

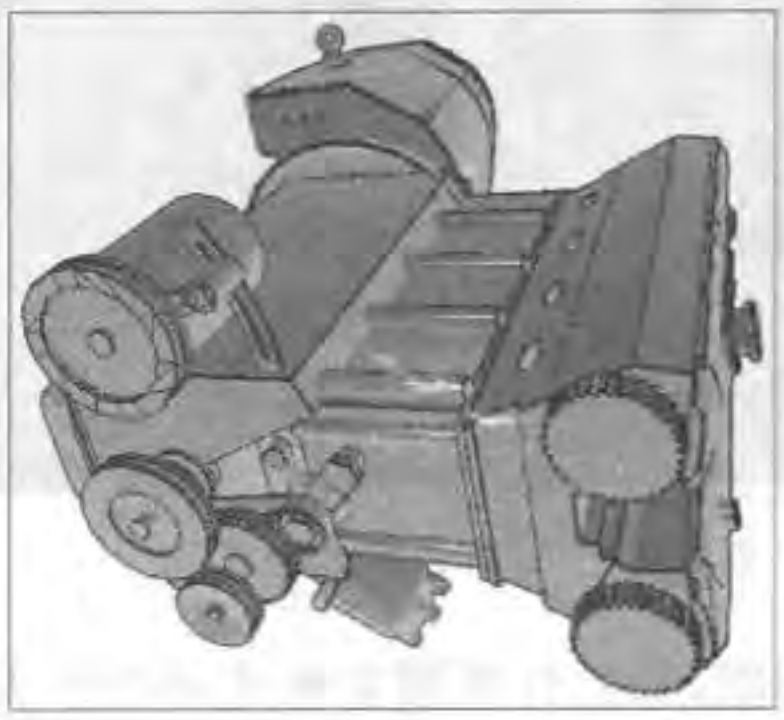


图 8.73 一个近似卡通的发动机图片

图 8.74 帆船的笔墨绘制

图 8.75 举了非真实感绘制的另外 3 个例子。注意这些图像必须非常像它们所代表的东西,尽管它们明显不像物体的真实照片。

图 8.75 拖拉机、摩托车、飞机的非真实感绘制

查阅本书的网站,可以找到关于非真实感绘制的资源和链接。

本章小结

计算机图形学从发展之初就对绘制 3D 场景的真实感有着很高的要求。物体线框表示可以被非常快速的绘制,但场景很难分辨,特别是几个物体相互重叠的情况。当用颜色填充面片和面消隐后,真实感有了很大的提高,但是这种方法绘制的图像在有光源照明的情况下依然不能给场景中的物体以突出印象。

所需要的是着色模型,以描述光线在表面的反射规律,这取决于表面的属性以及它与光源和摄像机的位置关系。光反射的物理原理是非常复杂的,所以程序员想出了一些近似方法和技巧,在大部分情况下在效果和速度上都可以接受。漫反射分量的模型是最接近现实物理效果的,但当考虑越来越多的因素时它变得极端复杂。高光反射根本不是基于物理原则的,但它们对于闪亮物体上的高光可以做恰当的近似。环境光非常的抽象,是避开处理物体之间多重反射的捷径,使得我们可以回避复杂的光线追踪计算。

即使是简单的着色模型也包含一些参数比如反射系数、表面粗糙度的描述和光源的色彩,OpenGL 提供了许多设置这些参数的方法。对于设计者,很少有如何选择这些参数取值的指导;往往通过反复试验直到最终达到满意的图像绘制效果来确定它们的取值。

在本章我们重点讨论多边形网格模型,最基本的任务就是绘制一个多边形。多边形的面

非常简单，用少量的数据就可以描绘，比如顶点位置、顶点法线、表面颜色和材质。另外还介绍了一个给多边形填充色彩的高效算法，特别是对凸多边形的情况。算法还能利用一个多边形的平整度来修改增加样式的深度值，使得深度-缓冲器隐藏面消隐算法更简单和有效。

当一个网格模型用来近似成一个原本平滑的表面时，表面上显现的边线效果是我们不希望看到的。Gouraud 和 Phong shading 提供了绘制平滑表面的方法（除了轮廓线）。Gouraud shading 速度快，但不能如实地重现高光；Phong shading 效果更真实但计算代价高。进一步研究 OpenGL 体系，Review Board（ARB）设计了一种着色语言来制作更真实的图像。更多细节可参阅 www.opengl.org。

场景的真实感程度可以通过在物体表面贴纹理得到极大的提高。纹理可以使物体看上去就像用某种材料比如砖块或木头制作的，标签或者其他外形也可以贴到表面上。环境贴图带给观众一个环绕闪亮物体环境的印象，这使得场景更真实，特别是在动画中。纹理贴图必须要认真制作，然而，需要采用正确的插值和反锯齿方法（如在第 9 章中所述）。

本章进一步讨论了物体阴影制作的效果。这是一个复杂的课题，现已经发展出了许多技术。通过更精细的技术比如光线跟踪可以得到更强的真实感。第 12 章阐述了这些技术的主要思想。

案例分析

案例 8.1　利用 OPENGL 制作着色物体

（难度：Ⅱ）设计一个程序：一个照相机飞越空间观看各种多边形网格物体。然后，在场景中设置一个点光源，给网格设置各种材质属性。包括环境光、漫射光、和镜面反射光分量。提供一个按键，能够在线框图和光滑着色图之间切换。

案例 8.2　纹理绘制

（难度：Ⅱ）升级案例 8.1 的程序，在网格物体表面绘制纹理。编写一个程序：能够读一个 BMP 图像文件，并把它贴到一个 OpenGL 纹理物体上。试验放五种不同的图像纹理和一种程序纹理到一个立方体的各个面，然后设置一个动画旋转立方体。提供一个按键，让用户能够在线性插值和校正插值之间切换来绘制纹理。

案例 8.3　升级 SDL 使其包含纹理

（难度：Ⅲ）SDL 场景描述语言不包括给物体表面指定纹理的方法。关键词纹理经常出现在 SDL 中，但它不做任何工作只是出现在文件中。认真研究 Scene 和 Shape 类的代码，可参阅本书网站，设计一种语法比如

```
texture giraffe.bmp p1 p2 p3 p4
```

从存储图像中制作纹理（这儿 giraffe.bmp），把它贴到后来定义物体的某一面上。确定纹理需要多少参数，和它们怎样使用。扩展 drawOpenGL()到 2 或 3 个形状，这样便于它准

确地对要求的物体贴图。

进一步阅读

要想更深入地了解最近的 OpenGL 着色语言,可阅读 Randi J. Post【Post02】所著的"OpenGL Shading Language"。还可阅读另一本最近的关于顶点和片断着色的书 Matt Pharr【Pharr05 所著的"GPU Gems 2"。另外,在图形学的经典著作 Jim Blinn 的"A Trip Down the Graphics Pipeline"【Blinn96】和"Dirty Pixels"【Blinn98】中,提供了众多的文章详细地介绍了阴影绘制和纹理绘制的双曲线插值的相关问题。

第 9 章

光栅显示工具

本章学习目标

- 讲述像素映射的概念和针对像素映射的操作
- 开发像素映射的复制、缩放、旋转工具
- 讨论各种绘制模式，例如 XOR 模式
- 开发图像合成的工具
- 介绍定义和处理区域的方法
- 讲述 Bresenham 直线绘制算法
- 构建区域填充工具——特别是多边形区域填充
- 讨论走样的产生和反走样算法
- 介绍用来产生更多颜色的抖动和误差扩散工具

预览

9.1 节和 9.2 节回顾了作为图像存储与处理的基本对象的像素映射，并讲述了针对像素映射的几种操作。9.3 节讲述像素映射合并的方法。像“异或”这种特殊的绘制模式也将在本节讨论。

9.4 节讲述 Bresenham 直线绘制算法。9.5 节到 9.7 节详细讲述像素映射中区域的描述和使用颜色或图案的区域填充算法。9.8 节讨论在光栅设备上显示图像时固有的走样（或锯齿）现象，和针对其开发的改善视觉效果的技术。

9.9 节介绍了抖动和误差扩散方法使得光栅显示器看起来比实际拥有更多的颜色。案例分析对上述主题进行了详尽讨论，并提出了一些重要的程序设计项目。

9.1 概　述

在本章中我们详细介绍如何从图元形成图像，以及如何处理图像以获得各种视觉效果。这里提到的图像由像素阵列组成，通常在光栅显示器上显示。

生成图像的方法主要有两种：

1. 扫描（和数字化）一幅已经存在的照片或者电视图像。
2. 用程序计算像素值，例如场景绘制。

在前面的章节中，我们主要关注于第二种生成方法：先生成如直线和多边形这样的图元，然后考虑如何为每个图元设置颜色和纹理坐标。利用 OpenGL，我们只需把它们发送到

"视口",这些图元就会显示在屏幕上。我们掩盖了为直线或多边形内部的每个像素确定颜色这一关键步骤,这件事由 OpenGL 来完成。

这种通过帧缓冲区一块区域的顶点位置和颜色等高级信息确定区域内部像素颜色的过程称为**扫描转换**(scan conversion)或**光栅化**(rasterization)。图 9.1 是图形绘制流水线的后端部分,光栅化就是其中一个过程。只要顶点经过视点变换后仍然可见,OpenGL 便将他们组合成图元(通过 glBegin (GL_POLYGON)或类似语句决定),然后对图元进行光栅化,即确定图元内部的所有像素的属性。在本章中我们将详细讲述这一过程是如何实现的,并介绍直线绘制以及使用颜色和纹理进行多边形填充的方法。

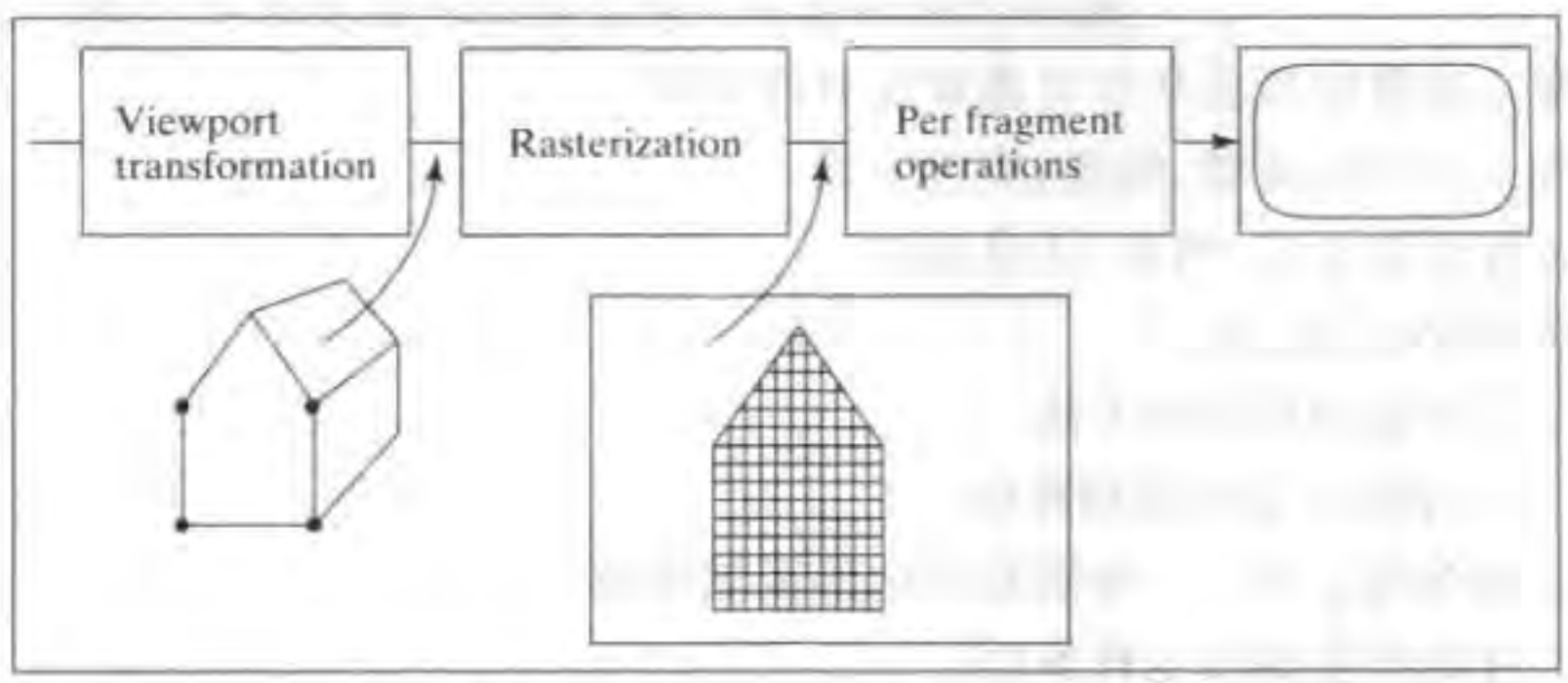


图 9.1 图形绘制流水线中的光栅化过程

事实上,光栅化过程生成的并非简单的像素,而是生成片段(fragment)。片段由颜色、深度和纹理坐标对组成。在把片段作为简单像素值写入帧缓冲区之前要对其进行一系列的操作和测试(见图中的"逐个片段操作")。我们将详细讲述各种可能的操作,并讨论它们的作用。

我们还要介绍在图像生成后如何对其进行各种处理,这包括基于一幅现存图像中(或要合并的几幅图像中)相关像素值计算新的像素值的操作。OpenGL 把这些操作放在图像处理流水线的"逐个片段操作"部分执行。

第 1 章介绍了光栅图像和一些相关的显示设备,在那里图像以像素映射(pixmap,pixel map 的缩写)的形式存储在内存中——一个矩形的数值数组。像素映射可以存储在计算机内存的任意区域,并且可以从一个地方复制到另一个地方。当像素映射复制到帧缓冲区后,扫描控制器将像素值转换为彩色亮度,像素映射就在显示设备上显示出来了。此时我们在屏幕上看到的实际上是帧缓冲区中所存数据的一个"画像"。因此,"像素"一词通常既指像素映射中保存的数值,也指显示器上这些亮点本身。

在前面章节中讲到像素映射中的每个像素值是定长的,而表示一个像素数据的比特长度被称为**颜色深度**(color depth)。如果一个像素具有 b 比特的颜色深度,那么它可以取 2^b 种不同的数值,因此最多可以表示 2^b 种不同的颜色。如果 $b=1$,则只有两种可能的颜色,例如黑与白。但是由于造价降低,现在很多视频显示器使用 32 位颜色深度,因此拥有 2^{32}(多达 8.1×10^{16})种可能的颜色。

光栅图像和光栅显示设备在处理空间和颜色数据时都是离散的。这种离散性,加上程序可以直接对内存像素操作的能力,导致一类特殊的图像生成和处理工具的出现。

9.2　处理像素映射

本节详细介绍像素处理的工具(包括在屏幕上,即处理帧缓冲区数据,也包括不在屏幕上,即在常规内存中),并描述这些处理过程产生的视觉效果。我们还会介绍 OpenGL 为处理像素映射提供了哪些工具。

9.2.1　对像素映射的操作

首先概括一下对像素映射都有哪些操作。在后续小节中,我们讨论这些操作如何实现,以及它们在不同情况下的效果如何。

绘制图像

通过绘制到帧缓冲区进行渲染的操作会改变显示设备上可见的内存映射[①]。当使用 OpenGL 绘制场景时,写操作会直接在帧缓冲区的像素映射上进行。

复制内存映射

像素映射可以从一段内存复制到另一段内存中。图 9.2 显示了四种复制操作,图中给出了四个可见的像素映射和它们的不可见副本。复制(copy)操作将图像从显示设备的一处复制到另外一处。读取(read)操作将显示图像的一部分复制到常规内存。绘制(draw)操作将常规内存的像素映射复制到帧缓冲区显示出来。还有一类操作称为内存复制(memCopy),它将一幅图像在常规内存中从一处复制到另外一处。

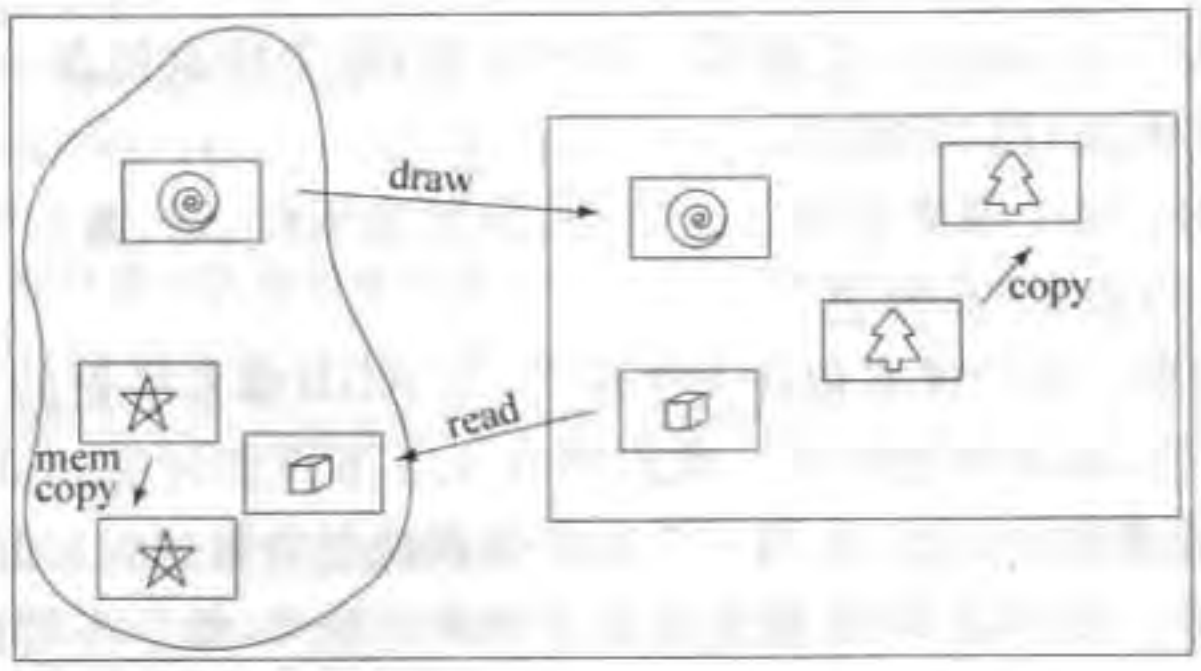


图 9.2　像素映射各种不同的复制操作

OpenGL 提供了一些函数来支持这些复制操作:

- glReadPixels();←读取帧缓冲区中的一个区域到常规内存。
- glCopyPixels();←在帧缓冲区中把一块区域复制到另外一个位置。
- glDrawPixels();←将给定的内存映射绘制到帧缓冲区中。

---

① 帧缓冲区内存有时也称为"屏幕上"内存,因为这些像素值和屏幕上显示的内容关系紧密。

缩放和旋转内存映射

放大、缩小和旋转图像有时是很必要的。根据缩放因子和旋转角度的不同,实现这些操作可能很简单,也可能相当复杂。

稍后我们会详细讨论。

比较两个内存映射

另一个重要的任务就是比较两个像素映射的差异。例如,通过比较两幅 X 光图像可以发现肿瘤是否发生变化。通常两个像素映射是逐个像素比较,同时在对应的像素对上执行某些数学操作。

在内存映射中表示一个区域并着色

一些像素映射包含的是易于识别的物体,例如圆、多边形和某些方面看上去一致的像素组成的区域(参考心脏、脑部、下颌的 X 光图像)。我们需要描述像素映射中的这些区域,并使用更高级、更抽象,或符号化的形式对它们进行表示。另外,我们还经常需要使用单一颜色或图案,为这些区域的内部像素进行着色,这被称为**区域填充**(region filling)。

9.2.2 对像素映射有用的数据类型

为像素映射的操作而设计一些类是很自然的。一个像素映射具有特定的行数和列数,其中每个像素又保存着一些特定的数据。下面是一些重要的像素映射实例:

- 位图(bitmap):每个像素占一位(bit),因此每个像素不是"开"就是"关"。
- 灰度级(grayscale)像素映射:每个像素占一个字节(byte),表示 0~255 的灰度级别。0 代表黑色,255 代表白色。
- LUT 索引(LUT indices):正如第一章的介绍,每个像素包含一个数字,这个数字是一个颜色查找表(LUT)的索引。
- RGB 像素映射:每个像素包含 3 个字节,分别表示红、绿、蓝三个颜色分量。通常认为这种方式可以表示"真彩色"。
- RGBA 像素映射:每个像素包含 4 个字节,与 RGB 像素映射的不同之处在于第4 个字节是 alpha 值,表示不透明度。我们将在 9.3 节详细讨论 alpha 通道的使用。

为了方便地操作像素映射,我们使用一个类将像素映射的数据封装起来,并提供一些成员方法来管理和操作图像。我们以 RGB 像素映射为例讲述细节,然后将其扩展到其他类型。为了充分利用 OpenGL 的图像处理工具,像素映射的数据格式应与 OpenGL 的格式保持一致[①]。

我们首先定义 RGB 类型,来保存一个像素值的 RGB 三元组:

```
struct RGB {unsigned char r, g, b; };
```

图 9.3 显示了 RGBpixmap 类的定义。我们遵循 OpenGL 的格式,将像素映射表示为一维的像素数组,像素的顺序从底到顶按行排列,每行之内按从左到右顺序排列。

---

① 实际上 OpenGL 提供了一系列方法来封装内存中的像素映射数据。我们在这里使用最常用的格式。

```
class RGBpixmap {
private:
    int nRows, nCols;                          //像素映射的尺寸
    RGB * pixel;                               //像素数组
public:
    RGBpixmap() { nRows=nCols=0; }
    RGBpixmap(int r, int c)                    //构造器
    {
        nRows=r;
        nCols=c;
        pixel=new RGB[r * c];
    }
    void setPixel(int x, int y, RGB color)
    {
        if (x>=0 && x<nCols && y>=0 && y<nRows)
            pixel[nCols * y+x]=color;
    }
    RGB getPixel(int x, int y)
    {
        return pixel[nCols * y+x];
    }
    //***在当前光栅位置(CRP)绘制像素映射
    void draw() {
        glDrawPixels(nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
    }
    //***将矩形区域像素值读入像素映射
    void read(int x, int y, int wid, int ht) {
        nRows=ht;
        nCols=wid;
        pixel=new RGB[nRows * nCols]; if (!pixel) exit(-1);
        glReadPixels(x, y, nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
    }
    //***复制部分显示区域回显示器上
    void copy(int x, int y, int wid, int ht) {
        glCopyPixels(x, y, wid, ht, GL_COLOR);
    }
    //***将 BMP 文件读取到像素映射中
    int readBMPFile(char * fname);
    //***将像素映射写入 BMP 文件
    void writeBMPFile(char * fname);

    //…其他内容…
};
```

图 9.3　RGBpixmap 类定义

图中定义了一些方法。默认构造函数创建一个空的像素映射，另一个构造函数创建一个 r 行 c 列的像素映射。setPixel()和 getPixel()的作用分别是设置和读取一个特定的像素值。

draw()、read()和 copy()方法直接通过 OpenGL 函数实现。

- draw()方法将像素映射复制到帧缓冲区,使得映射的左下角放在"当前光栅位置",这个变量可以通过 glRasterPos2i(x, y)来设置。
- read()方法执行反方向的复制,将一块矩形的帧缓冲区数据复制到像素映射中。区域的左下角位于"当前光栅位置"pt,wid 和 ht 指定区域的大小。read()首先分配保存这些像素所需的内存,由 glReadPixels()来完成真正的复制操作。
- copy()方法在帧缓冲区内将一个区域复制到另外一个位置,效果相当于依次执行了一次 read()和 draw(),只是没有生成中转的像素映射。被复制的区域左下角在(x, y),大小由 wid 和 ht 决定。复制到的新位置仍在帧缓冲区中,左下角在当前光栅位置。这是 9.3.4 节将要介绍的 bitblt 操作的一种形式。

利用 readBMPFile()和 wrtieBMPFile()可以很容易的创建和存储像素映射。readBMPFile()将保存在 BMP 文件中的图像读入像素映射,同时为其分配所需的内存。wrtieBMPFile()创建一个 BMP 文件保存像素映射。这两个方法的代码可以从图书公司的网站上获取。也可以从 http://nehe.gamedev.net 获取 loadBMP()函数,该网站还提供了一些例程,但与像素映射的函数稍有不同。

这个类虽然简单但是功能强大,下面的例子将说明如何使用这个类。

例 9.2.1 处理像素映射的测试平台

图 9.4 给出了一个使用 pixmap 类的应用程序,使用鼠标和键盘控制图像映射的读取和绘制。使用这个应用程序进行实验可以获得很多信息。这个程序开始运行时创建了两个像素映射 pic[0]和 pic[1],然后为其加载了两幅 BMP 图像。其中一个图像显示在初始的光栅位置。单击鼠标左键调用 draw()方法在鼠标点击位置再次绘制图像。如果按住鼠标左键在屏幕上拖动,像素映射则随着鼠标运动位置在屏幕上绘制出来。按下 s 键,在两种像素映射间切换。按下 r 键,调用 read(),读取屏幕上一块 200×200 大小的区域的内容,写入 pic[0],覆盖掉原来的内容。单击鼠标右键清屏。

```
//必须的头文件
RGBpixmap pic[2];                                   //创建两个(空的)全局像素映射
int screenWidth=640, screenHeight=480;
IntPoint rasterPos(100,100);
int whichPic=0;                                     //当前显示的像素映射
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<myMouse>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myMouse(int button, int state, int mx, int my)
{     //左键单击设置当前光栅位置
    if (button==GLUT_LEFT_BUTTON && state==GLUT_UP)
    {
        rasterPos.x=mx; rasterPos.y=screenHeight-my;
        glRasterPos2i(rasterPos.x, rasterPos.y);
        glutPostRedisplay();
    }
    else glClear(GL_COLOR_BUFFER_BIT);        //右键清屏
}
```

图 9.4 一个像素映射操作的测试程序

```
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<mouseMove>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void mouseMove(int x, int y)
{    //随着鼠标移动设置当前光栅位置
    rasterPos.x=x; rasterPos. y=screenHeight-y;
    glRasterPos2i(rasterPos.x, rasterPos.y);
    glutPostRedisplay();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<myReshape>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myReshape(int w, int h)
{
    screenWidth=w; screenHeight=h;
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<myDisplay>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myDisplay(void)
{
    pic[whichPic].draw();          //在当前光栅位置绘制图像
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<myKeys>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myKeys(unsigned char key, int x, int y)
{
    switch(key)
    {
    case 'q': exit(0);
    case 's': whichPic=1-whichPic; break;          //切换图像
    case 'r': pic[0].read(0,0,200,200);break;      //屏幕抓图
    }
    glutPostRedisplay();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<main>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(30, 30);
    glutCreateWindow("Experiment with images");
    glutKeyboardFunc(myKeys);
    glutMouseFunc(myMouse);
    glutMotionFunc(mouseMove);
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glClearColor(0.9f, 0.9f, 0.9f, 0.0);           //背景色
    glClear(GL_COLOR_BUFFER_BIT);
    pic[0].readBMPFile("CokeCan2.bmp");            //创建一个像素映射
    pic[1].readBMPFile("Mandrill.bmp");            //创建另一个像素映射
    glutMainLoop();
}
```

图 9.4（续）

例 9.2.2 在屏幕上写文本

在光栅显示器中绘制字符的一种常用方法是为每个字符创建各自的像素映射。图 9.5 给出了一些例子。这些像素映射提供了字符的"图画",它通常定义在某一特定尺寸的像素阵列内,如 12 像素×8 像素,这些像素取值为 0 或 1。因此如果点阵尺寸较大,字符的形状容易控制,可以获得更加美观的字体。

程序初始化时,可以从硬盘加载各种字体到屏幕外内存,随时准备使用。当绘制字符串时,每个字符依次通过 draw()函数绘制到帧缓冲区的适当位置,同时根据该字符的宽度递增 x 的位置。对于按比例间隔的字体,每个字符都有一个特有的宽度(通常"i"间隔最小,"W"间隔最大)。这样字符显示就获得了最美观的效果。

例 9.2.3 窗口卷动

对于类似文字处理软件的应用程序中,屏幕上经常布满文字,一行接着一行。如图 9.6 所示,为了给屏幕底下新建的一行腾出位置,上方所有文字必须向上卷动一行(当然最上一行文字也随之消失)。为此,可以调用 copy()方法将除第一行之外的所有行的矩形,向上复制一行,并覆盖原来的内容。然后调用 draw()方法绘制一个空行到最下一行。

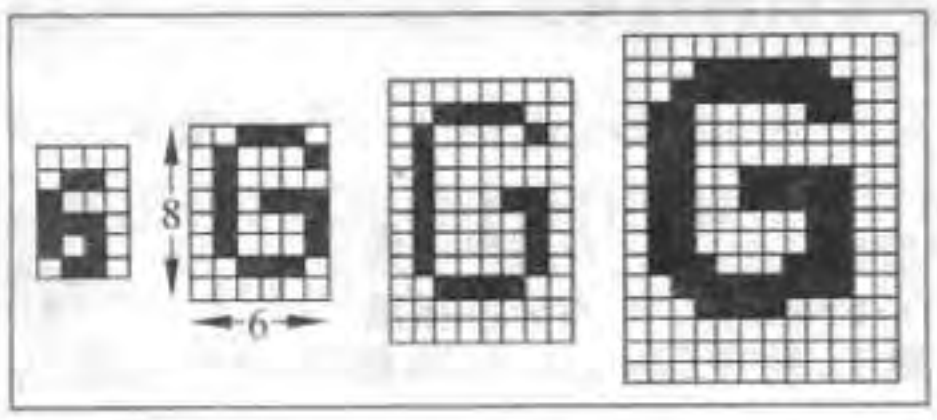


图 9.5 字符可以由二进制点阵构成:大的点阵可以构造更美观的字体从左到右点阵的大小分别是 4×6、6×8、8×12、12×16

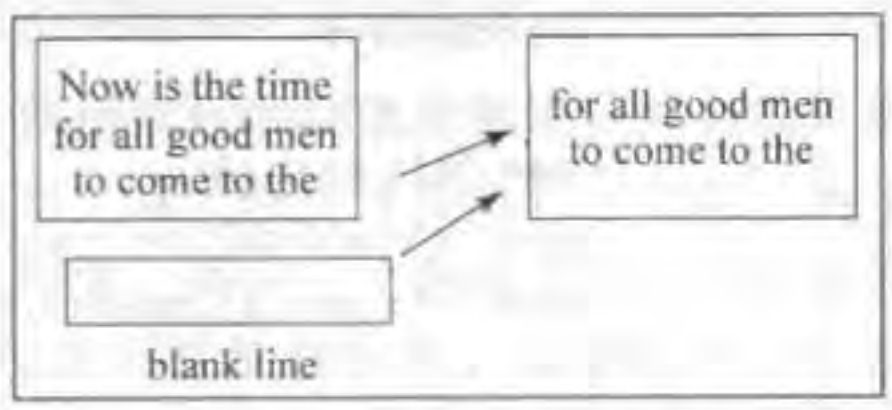


图 9.6 窗口内文字卷动

例 9.2.4 弹出菜单、下拉菜单和对话框

正如第 2 章所述,当今人们与计算机交互大多通过图形用户界面(GUI),它提供了多种菜单和对话框,用户通过鼠标选择要执行的动作。

当菜单被激活时,会弹出到视图中,暂时覆盖了部分屏幕。图 9.7(a)显示了屏幕被下拉菜单覆盖的结果。当用户做出了选择后,菜单消失,被菜单覆盖的区域就会重新出现(图 9.7(b))。当然,被遮挡的部分必须事先保存到常规内存,稍后才能被恢复。菜单覆盖区域恢复的流程如下:

1. 用户选择菜单标题。
2. 程序计算将要覆盖的矩形。
3. 调用 read(),将该区域复制到屏幕外内存。
4. 调用 draw(),在合适的位置绘制菜单。
5. 用户移动鼠标指针,点击需要的菜单项。
6. 程序将屏幕外内存中的像素映射复制到原来的位置,从而恢复被覆盖区域。

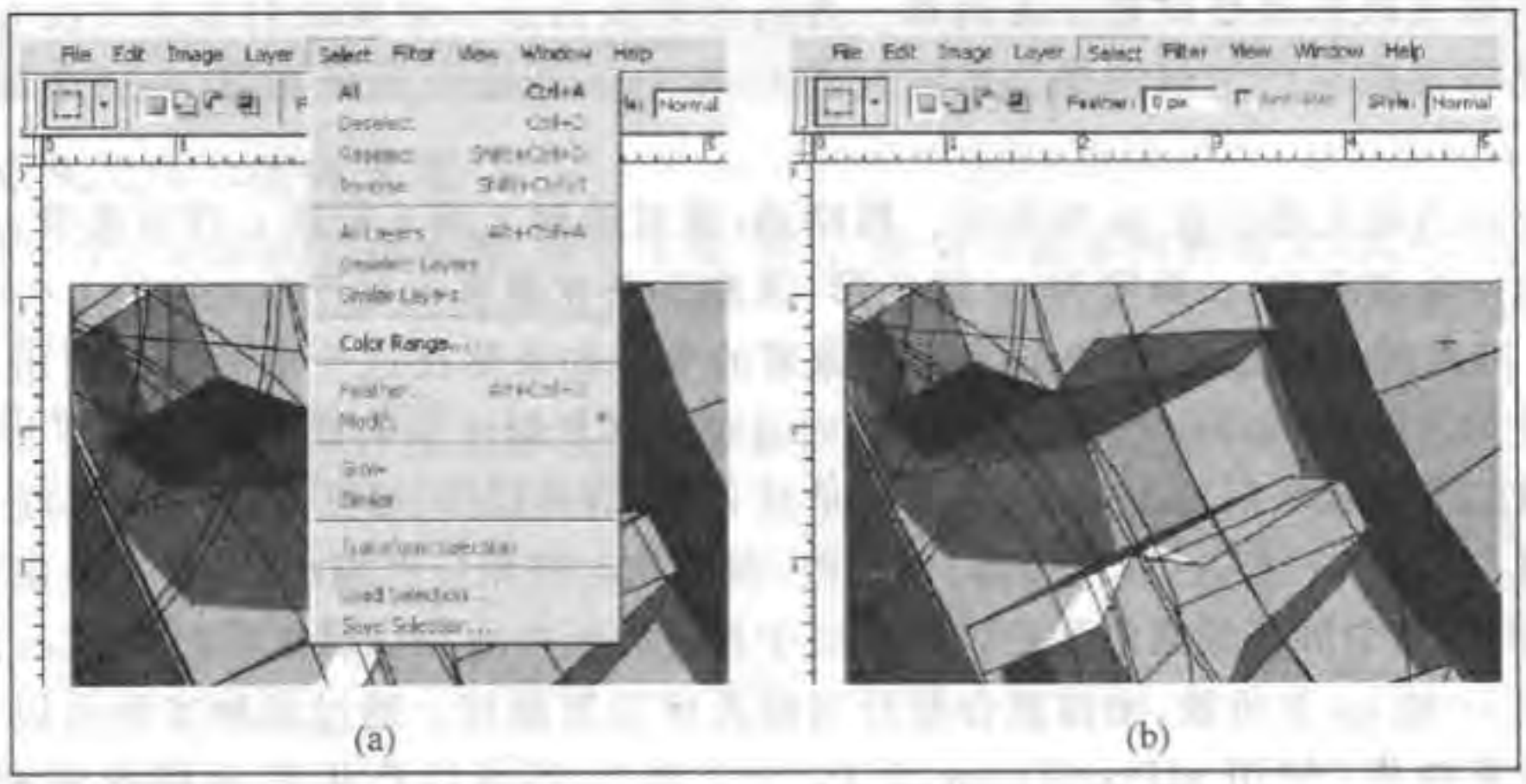


图 9.7　下拉菜单

9.2.3　图像缩放和旋转

图像的缩放和旋转操作是经常被用到的。例如，一个人造卫星拍摄的欧洲的图像可能是以 1600×1200 的像素映射格式显示，而我们要将其与以前扫描过 1540×1800 像素的图像进行比较。在比较之前，需要将小的那个图像放大，使两幅图像对应的部分能够正确的重叠起来。类似地，我们可能要比较一组数字化的指纹，但是发现一些指纹图像和其他图像比起来有轻微的旋转。为了对比不同的指纹，我们必须旋转图像使他们对齐。下面介绍一些简单的缩放和旋转操作。

以某个系数 s 对一个像素映射进行缩放的概念就是生成一个新的像素映射，其 x 和 y 方向的像素个数均为原来的 s 倍。用摄影术语说，如果 s 大于 1，就说这个像素映射被放大(enlarged)了；否则称这个像素映射被缩小(reduced)。当然像素本身的大小是不变的。当 s 是整数时，很容易通过像素复制(pixel replication)来放大像素映射。图 9.8 显示了一个字符位图两倍放大的例子。小位图中每个像素在大位图中生成一个 2×2 的像素阵列。对于其他整数 s，每个像素生成一个 $s\times s$ 的像素阵列。

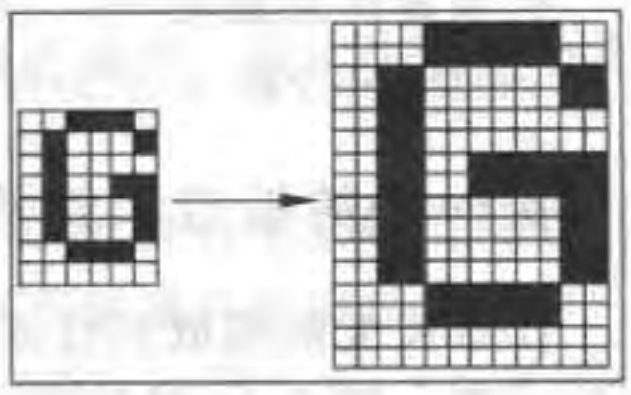


图 9.8　字符的两倍放大

以某个整数系数(即 $s=1/n$，n 是整数)缩小像素映射的过程也是很直观的。例如，对于 $s=1/3$ 的情况，可以通过在原始像素映射中每 3 行和每 3 列中保留一个像素来建立缩小的像素映射。按这种方式对原像素映射进行的“采样”会丢失信息，因此缩小的图像可能比原始图像的质量差很多。通常计算这 9 个像素的均值，并将其放在保留的位置上作为代表像素，会得到更好的效果。我们会在 9.7 节讨论包括这种思路的反走样技术。

OpenGL 提供了一个简单的机制来进行图像的缩放，这些图像使用 glDrawPixels()和 glCopyPixels()绘制(因此 draw()和 copy()函数可以看到缩放的效果)。函数

```
glPixelZoom(float sx, float sy);
```

为后续的像素映射函数设置 x 方向和 y 方向的缩放因子。像素映射本身并没有缩放;缩放的是从像素映射产生的图像。缩放因子 sx 和 sy 可以是任意浮点数,包括负值,默认值是 1.0。

缩放以当前光栅位置 pt 为基准。粗略讲,像素映射上第 r 行第 c 列的像素,在屏幕上被绘制成一个宽度为 sx 高度为 sy 的矩形,其左下角在屏幕上处于($pt.x+sx*r$, $pt.y+sy*c$)。准确的说法是,屏幕上任意一个像素的中心如果落在这个矩形内就使用该像素点的颜色表示。例如,如果 sx 是 2,sy 是 3,则通过像素复制生成的要绘制在屏幕上的像素映射的宽度是原像素映射宽度的 2 倍,高度是其 3 倍。它的左下角位置保持不变,还是在当前光栅位置。如果 sx 或 sy(或两者都)是小数,图像就会被相应的缩小,并且有一定程度的质量损失,因为每个屏幕像素的颜色仅仅取决于其中心所在的"矩形"像素的颜色。

如果 sx 或 sy 是负数,图像就会相对当前光栅位置翻转。通过这种方法可以显示一个倒置的像素映射(使用 glPixelZoom(1.0, −1.0)),还可以产生其他特殊的视觉效果。图 9.9 是测试平台的一个截图,并排显示了一幅图像的四个缩放后的结果(使用缩放系数分别为 sx=1.0, −1.0, 0.5, 1.5)。生成每幅图像时首先设置 sx,然后执行 glPixelZoom(sx, 1); glutPostRedisplay()。

图 9.9 通过修改 x 方向缩放尺度,生成一个像素映射的多个不同版本

1/4 周旋转

把像素映射旋转 90°、180°和 270°的过程非常简单:只要生成一个新的像素映射,然后使用 getPixel()和 setPixel()将像素从一个映射复制到另外一个。参见练习。

更一般的缩放和旋转

任意尺度的缩放和任意角度的旋转情况会更加复杂。假定我们要生成某个像素映射的变换结果。图 9.10 显示了一个像素映射 S(源),我们要对其施加变换 T,T 可能是一个混合了旋转和缩放的仿射变换。变换的结果是像素映射 D(目标)。现在的问题是如何正确计算 D 中每个像素 p 的颜色值。最简单的方法是对 D 中的每个像素 p,寻找 S 中覆盖了 $T^{-1}(p)$ 的像素的颜色,这个颜色就作为 D 中像素 p 的颜色值。然而仅仅使用这样的"采样"方法可能产生严重的锯齿,变换结果的质量可能会很差,这也是反走样部分要解决的问题之一。

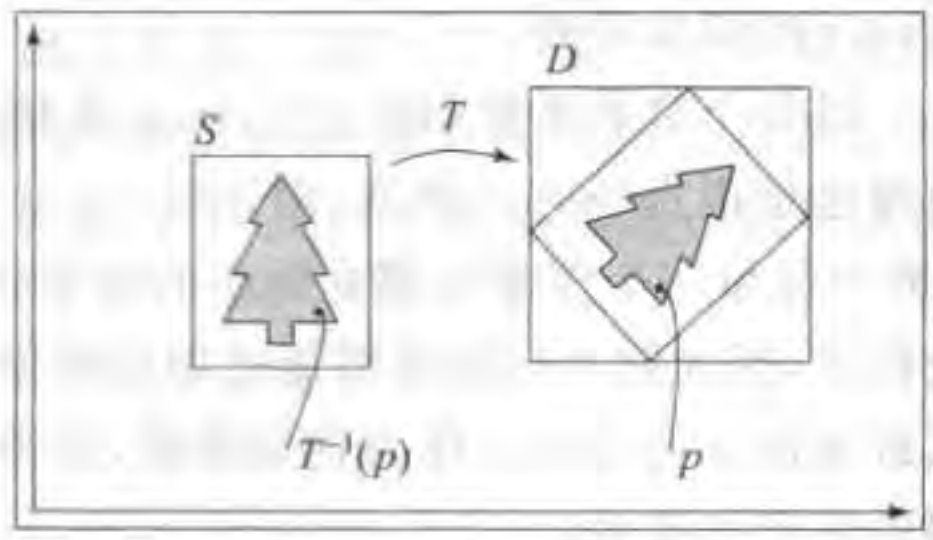


图 9.10 计算变换后的图像

练习

练习 9.2.1　设计字符'R'

设计一个 8×16 的二值图像，显示字符'*R*'。使用 1 和 0 两种模式，其中 0 表示白色，1 表示黑色。

练习 9.2.2　设计字符

在 r 行 c 列的点阵内设计最好看的字符：a、A、r、R、& 和 3。按照下列点阵尺寸进行设计：

a. $(r, c)=(6, 4)$；

b. $(r, c)=(10, 8)$；

c. $(r, c)=(14, 12)$。

练习 9.2.3　缩小字符

一种把矩形像素映射中定义的字符缩小的方法是把阵列的行数和列数分别缩小为原来的一半，缩小的像素映射中每个像素必须模拟原来单元中对应的 2×2 的像素阵列。给出一个算法，从 12×10 的像素映射生成 6×5 的像素图案。一种方法是如果 2×2 的像素阵列中有两个或更多的像素是 1，就把对应的新像素设置为 1。这个规则是不是最好的呢？

练习 9.2.4　旋转 90°

考虑一个正方形的像素映射，我们使用数组 $A[i][j]$表示，i 和 j 的范围是 $0\sim N-1$。给出 A 逆时针旋转 90°时形成的像素映射 $B[col][row]$的表达式。

9.3　合并像素映射

在某些情况下我们需要通过合并两个像素映射生成一个新的像素映射。这个操作可以用于屏幕上移动鼠标、比较两幅图像、让一幅图像渐变到另一幅图像等。我们先看几个有重要实际意义的例子。

像素映射合并操作通常是逐像素(pixelwise)的——也就是说，在新旧像素映射的对应像素执行某种操作。明确地说，两个像素映射 A 和 B 根据下式合并形成第三个像素映射 C。

$$C[i][j] = A[i][j] \otimes B[i][j] \quad \text{对于每一个 } i, j$$

其中⊗表示某种操作。下面是几种不同的操作的示例：

- 计算两幅图像的平均值。这里⊗表示取 A 的一半和 B 的一半的和：

$$C[i][j] = \frac{1}{2}(A[i][j] + B[i][j])$$

- 计算两幅图像的差，用来确定两幅图像的差别。这里⊗表示取减：

$$C[i][j] = A[i][j] - B[i][j]$$

- 寻找一幅图像比另一幅图像亮的位置。这里⊗意味着"前者大于后者"：

$$C[i][j] = A[i][j] > B[i][j]$$

如果 A 中某一像素比 B 中对应像素亮，则 C 中对应像素取值为 1，否则取值为 0。

计算两幅图像平均值的一般形式是计算两幅图像的加权平均(weighted average)。对

于某个小数 f,像素映射 A 的权重为$(1-f)$,B 的权重为 f,则:

$$C[i][j]=(1-f)A[i][j]+fB[i][j]$$

举例来说,如果 $A[i][j]$的 RGB 分量是(14,246,97),$B[i][j]$的 RGB 分量是(82,12,190),则当 $f=0.2$ 时,有 $C[i][j]=(27,199,115)$。可以通过修改 setPixel()函数实现两个 RGBpixmaps 的加权平均。

例 9.3.1 从一幅图像渐变到另一幅图像

加权平均的一个有趣应用是两幅图像的渐变。开始的时候图像 A 完全显示。随着时间流逝,A 逐渐淡出,同时图像 B 逐渐出现在 A 上,直到最后显示内容只有图像 B。如果使用 t 代表时间,随着 t 从 0 平滑过渡到 1,在时刻 t 显示的图像为:

$$A(1-t)+Bt \tag{9.1}$$

这个过程非常类似于第四章介绍的"插值"。图 9.11 显示了 $t=0$、0.25、0.5、0.75、1 等五个时刻显示的图像。案例分析 9.2 讨论了一种利用 OpenGL 的 alpha 通道功能实现两幅图像渐变的方法。9.3.2 节将详细介绍 alpha 通道。

图 9.11 两幅图像的渐变

9.3.1 读-改-写循环

在把像素映射 D 和 S 合并形成一个新像素映射 C 的过程中,有一种特殊情况就是 C 和 D 相同,即合并的结果又放回到 D 中。这个合并过程可以表示为 $D=D\otimes S$:像素映射 D 中的像素与 S 中的对应像素合并,得到的结果放回到 D 的该像素位置。这个过程被称为读-改-写循环,因为 D 中的像素首先从内存中读出,然后与 S 中对应像素合并而发生改变,最后结果写回 D。图 9.12 用符号说明了 D 本身是帧缓冲区时,这个循环的工作流程。合并过程就是依次对 D 中每个像素应用这个循环的过程。采用一些硬件配置能使这个操作效率非常高。后面将会看到,OpenGL 提供了一个指令可以完成整个像素映射的合并操作。

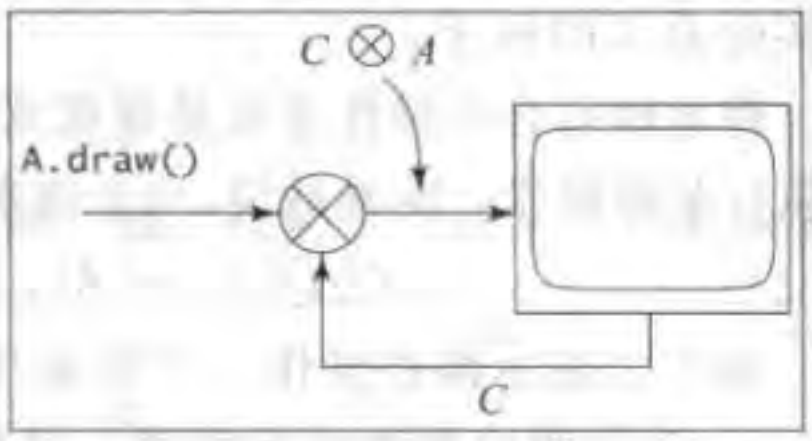


图 9.12 应用于帧缓冲区的读-改-写循环

9.3.2 alpha 通道和图像融合

两幅图像加权求和实际上是图像融合(blending)或合成(compositing)的一个特例。融合操作可以在一幅图像上绘制另外一幅半透明的图像,其关键是为 RGB 颜色模型添加第四

个分量——alpha 值。参照前面表示像素的数据类型，我们将 RGB 类型扩展为 RGBA 类型：

```
struct RGBA {
    unsigned char r, g, b, a;
};
```

alpha 值 a 的取值范围是 0～255，通常表示像素的“不透明”程度：值为 0 表示完全透明，值为 255 表示完全不透明。需要注意的是，RGBA 类型的像素映射需要的内存是同样的 RGB 类型像素映射的 4/3。像素映射中所有 alpha 值放在一起的集合通常称为 alpha 通道。

后面将会看到，alpha 分量最常见的用法是作为一个在 0～1 之间的缩放系数，因此实际使用的值是一个小数，$a/255$。

例 9.3.2　半透明图像覆盖

图 9.13 给出了一个在背景图像 D 上覆盖一个由面具和龙组成的图像 S 的例子。当源图像 S 创建时，龙所在像素的 alpha 值设为 255(完全不透明)，面具所在像素的 alpha 值设为 128(半透明)，所有其他像素的 alpha 值设为 0。当 S 和 D 融合时，结果显示出龙在背景 D 上，同时有一些背景颜色从面具中“显露出来”。

图 9.13　把图像 S 合成到图像 D 上

alpha 融合是一个源像素和目标像素加权平均，并写回目标像素的过程，也就是一个读-改-写循环。用表达式表示就是 $D=aS+(1-a)D$，其中 a 是源像素的 alpha 值，为某个 0～1 之间的小数。但实际上每个像素的 alpha 值可能都是不同的，而且 S 和 D 都有红、绿、蓝三个分量，因此更精确的写法是这样的，以第 j 行第 i 列上的最终目标像素的绿色分量为例：

$$D[i][j]\cdot g = aS[i][j]\cdot g + (1-a)D[i][j]\cdot g \tag{9.2}$$

其中 $a=\frac{S[i][j].a}{255}$。特别注意的是，a 值与 i 和 j 有关，即不同像素的 alpha 值也是不同的。红色和蓝色分量融合的表达式也是类似的。

要在程序中实现这种融合操作，我们首先要把前面介绍的 RGBpixmap 类扩展成 RGBApixmap(参见练习)，然后添加 blend()方法，通过一个读-改-写循环实现这种覆盖功能。像素映射 S 和 D 的融合便可以使用下面的代码完成：

```
D.draw();                    //和以前一样使用不透明方式绘制 D
S.blend();                   //使用 S 的 alpha 值,与 D 进行加权平均
```

OpenGL 提供了工具,使得 blend()更容易实现。只需设置一个"融合模式",指定源缩放系数和目标缩放系数。要把 a 设置为源 alpha 值,只需用如下方式指定:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

该命令设置源缩放系数为源本身的 alpha 值,设置目标缩放系数为 1-源 alpha 值。图 9.14 给出了 blend()方法的一个简单实现。

```
void RGBApixmap::blend()
{
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);              //启用融合
    draw();                          //绘制像素映射,并与目标融合
}
```

图 9.14 使用源图的 alpha 通道融合源图像和目标图像

例 9.3.3 模拟色度键:将某种颜色强制设为透明色

我们在电视上经常看到天气预报员站在一幅地图前讲解各种天气状况。实际上,这个预报员是站在一幅蓝色的背景前(图 9.15),同时一幅单独的气象图在同步的播放。电视信号在人的图像与地图图像之间切换:当线扫描中遇到蓝色就切换到地图的信号,否则显示人的画面。

图 9.15 在电视中使用的色度键

我们可以利用像素映射模拟这种效果,通过指定某种颜色的 alpha 值为 0 ,使源像素映射中这种颜色透明。下面程序在一个像素映射中扫描,如果当前像素的颜色与指定的颜色一致,就把它的 alpha 值设为 0,否则设为 1。

```
void RGBApixmap::setChromaKey(RGB c)
{
    long count=0;
    for(int row=0; row<nCols; row++)
        for(int col=0; col<nRows; col++)
        {
            RGBA p=pixel[count];
```

```
            if(p.r==c.r && p.g==c.g && p.b==c.b)
                pixel[count++].a=0;
            else pixel[count++].a=255;
        }
}
```

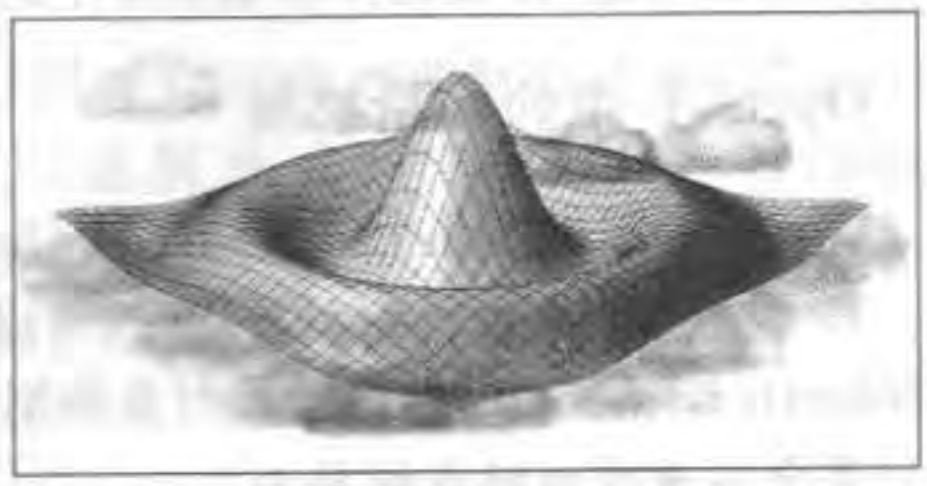


图 9.16　使像素映射的背景透明

图 9.16 显示了像素映射 S，它有单一背景颜色 c。应用 setChromaKey(c)方法可以使得该像素映射所有颜色为 c 的像素的 alpha 值为 0。

例 9.3.4　用画笔绘制

在画图程序中，鼠标起到了画笔的作用。用户可以为每条“笔划”设置颜色样本。这个功能可以通过 blend()函数来实现，将画笔的像素映射的 alpha 值设为 10%左右，并与目标图像融合。每画一笔，就有一些画笔的颜色融合到目标图像上(参见练习，了解如何估算每次下笔时新颜色增加的数量)。接近画笔中心像素的 alpha 可以比画笔边缘处像素的 alpha 设置得高一些，使得中心的颜色增加的快一些。

例 9.3.5　光标管理

移动鼠标时，鼠标的光标也在屏幕上掠过。光标移动到每一个位置，它都会遮住一部分显示的内容，而当鼠标移开，这部分内容也必须恢复回来。

实现这一任务的处理过程与显示下拉菜单类似，正如图 9.17 所示。在屏幕上绘制光标之前，先生成一个大致能够覆盖光标的矩形区域的图像副本(图中的像素映射 $Pix1$)。然后把光标的像素映射在此处与图像融合。在光标的像素映射中，1 表示光标箭头部分，0 表示其余部分。因此，当融合发生时，箭头看上去是浮在背景图像之上。当光标移动到一个新的位置时，有三件事情发生：

(1) 调用 draw()方法绘制 $Pix1$(alpha 值为 1)，覆盖到当前光标位置，也就是恢复原来的图像。

(2) 调用 read()方法将要被光标遮挡的部分($Pix2$)读入屏幕外的内存。

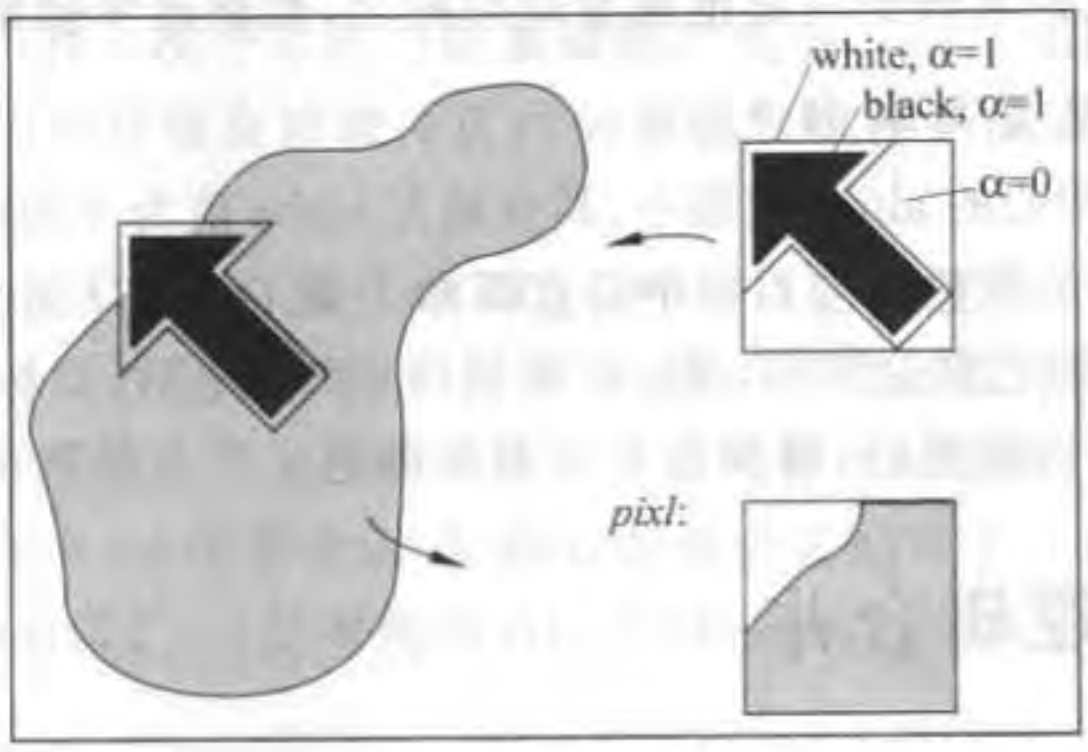


图 9.17　管理移动光标

(3) 在新的鼠标位置对光标和背景图像进行融合。

要注意的是,即使光标移动非常小的距离,也要进行这个过程。

OpenGL 中的可用扩展

OpenGL 提供了几个设置和使用 alpha 通道的工具。

1. **设置 alpha 值** 绘制场景时,可以显式的设置后面要绘制的图元的 alpha 值。使用 glColor4f(r, g, b, a)指定这些对象的第四个颜色分量,a 的取值范围为 0.0(完全透明)到 1.0(完全不透明),缺省值是 1.0。

我们也可以给三维物体设置"半透明材质",此时绘制这个物体的所有像素都具有一个特定的 alpha 值。这可以通过为物体每个定点的漫反射系数指定一个 alpha 值来实现:

```
glMaterialfv(GL_FRONT, GL_DIFFUSE, ref1) ;
```

其中 ref1 是一个四元组,它给出漫反射分量的颜色。例如 ref1[4]={0.5,0.3,0.8,0.4},将后面定义的顶点的 alpha 值设置为 0.4。

2. **融合设置方法** 我们已经介绍了使用 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)生成源像素映射和目标像素映射的加权平均。通过为 glBlendFunc()选取不同的参数,OpenGL 允许为源缩放系数和目标缩放系数设置不同的值,这些参数包括 GL_ZERO、GL_ONE、GL_DST_ALPHA、GL_SRC_COLOR 等。

例如,glBlendFunc(GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA)设置了源像素和目标像素的权值分别为 α_D 和 $1-\alpha_D$,其中 α_D 是目标像素的 alpha 值,而非源像素的。glBlendFunc(GL_DST_COLOR, GL_ZERO)设置了用目标像素的相应分量调整(或乘)源像素的每个颜色分量。

练习

练习 9.3.1 扩展 RGBpixmap 类使其包含 alpha 分量

调整 9.2.2 节的 RGBpixmap 类定义,使像素映射支持 alpha 通道。添加 blend()方法和 setChromaKey()方法,并通过以下途径进行测试:(a)将一个 BMP 图像读取到像素映射中(如果图像具有单一颜色背景,测试效果更好);(b)使用 setChromaKey()方法设置某一颜色为透明色;(c)将此像素映射与其他像素映射融合,观察透明效果。

练习 9.3.2 画笔应用新颜色

考虑一个靠近笔刷中心的 alpha 值很小、其余地方 alpha 值为 0 的画笔的像素映射。每一次使用这个画笔时,画笔的颜色 C 与目标中任意图像 D 融合,则 D 变为 $\alpha C+(1-\alpha)D$。重复使用目标图像上画笔的颜色就会加深。(a)证明使用 i 次画笔后,目标颜色为 $D_i=\alpha C+(1-\alpha)D_{i-1}$;(b)计算使用 8 次画笔后,源颜色 C 在目标颜色中所占的比例。

9.3.3 像素的逻辑合并

还有一些合并像素的方法是将每个像素值简单的看作比特位的集合来处理,而不关心比特位所表示的数值。同以前一样,像素映射还是逐个像素合并,但每个像素是逻辑地

(logically)合并，逐位运算，例如使用或(OR)、与(AND)、异或(EXCLUSIVE OR)等运算来实现像素映射的合并。

举例说明，像素 A 的 RGB 颜色分量为(21,127,0)，写成二进制形式是(00010101,01111111,00000000)。假定 B 的颜色分量的二进制形式是(01010101,11110000,10000101)。如果对它们进行或操作，结果 C 在下面给出：

A：(00010101,01111111,00000000)
B：(01010101,11110000,10000101)
C：(01010101,11111111,10000101)　←A 和 B 的或运算结果

如果 A 和 B 上下对齐的位中有一个(或者都)是 1，则 C 中相应的位也是 1。如果对 A 和 B 通过异或运算(EXCLUSIVE OR 或者 XOR)合并，就得到

A：(00010101,01111111,00000000)
B：(01010101,11110000,10000101)
C：(01010101,11111111,10000101)　←A 和 B 的或运算结果

这种情况下，如果 A 和 B 对应位上恰好有一位是 1，即只有一个而不是两个都是 1，那么 C 中相应位也是 1。颜色 $C=A$ XOR B 依赖于 A 和 B，但结果并不像与和或运算那样直观(参见练习)。如果 B 是黑色(0,0,0)，那 $C=A$ XOR B 结果是什么呢？如果 B 是白色(255,255,255)，结果又是什么呢？异或运算有一个有趣的性质：应用两次异或运算等于根本没进行任何运算。也就是说，如果 $C=A$ XOR B，然后将 C 与 B 再次进行异或，即(A XOR B)XOR B，这个结果还是 A！(参考练习)。后面在实现"橡皮筋"(rubber band)时会充分利用这个性质。

像 C++ 这样的语言可以直接支持按位进行的逻辑运算，可以在程序中直接完成这样的操作。例如，对 A 和 B 的绿色分量 OR 运算，可以简单的执行 $C \cdot g = A \cdot g | B \cdot g$。类似地，对于红色分量的 XOR 运算，则执行 $C \cdot r = A \cdot r \wedge B \cdot r$。

其他运算还包括"与"(AND)、"非"(NOT)，以及这些运算的组合。如果对两个量进行 AND 运算，用 C++ 语言表示绿色分量的运算为 $C \cdot g = A \cdot g \& B \cdot g$，只有当 A 和 B 对应位都为 1 时，$C \cdot g$ 的相应位才是 1。如果对某个值进行非(或补)运算，用 C++ 语言表示是 $C \cdot g = \sim A \cdot g$，当且仅当 A 中的位为 0 时，C 中的对应位才为 1。

我们可以使用 C++ 这种逻辑运算对两个像素映射进行逐个像素的合并。不过更幸运的是，OpenGL 提供了一种一次合并两个像素映射的机制。一旦选择并启用了一种逻辑运算，后面所有绘制都对当前存储在帧缓冲区内的源像素执行一次读-改-写的循环来应用这种逻辑运算。

执行命令 glEnable(GL_COLOR_LOGIC_OP)可以启用逻辑运算。使用 glLogicOp()选择当前的运算模式，可用的参数如图 9.18 所示，图中 S 表示源像素，D 表示目标像素。默认值是 GL_COPY，这个参数表示使用源像素简单地代替目标像素。

图 9.19 显示了两个 8×8 像素映射 A 和 B 逻辑合并的例子。这里像素值与颜色的关系是：0 表示黑色，1 表示白色。(如果使用 GL_AND 和 GL_EQUIV，得到的像素映射是什么样的呢？)

下面我们介绍 XOR 绘制模式的经典应用——"橡皮矩形"(rubber rectangle)，并介绍如何使用OpenGL实现这个应用。它很好的利用下面这个原理：在XOR模式下对一个物

参 数 值	结 果	参 数 值	结 果
GL_CLEAR	0	GL_OR	S 或 D
GL_COPY	S	GL_NAND	(S 和 D)的非
GL_NOOP	D	GL_NOR	(S 或 D)的非
GL_SET	1	GL_XOR	S XOR D
GL_COPY_INVERTED	非 S	GL_EQUIV	(S XOR D)的非
GL_INVERT	非 D	GL_AND_INVERTED	S 的非和 D
GL_AND_INVERT	S 或非 D	GL_OR_INVERTED	S 的非或 D
GL_AND	S 和 D		

图 9.18 OpenGL 中可用的逻辑运算

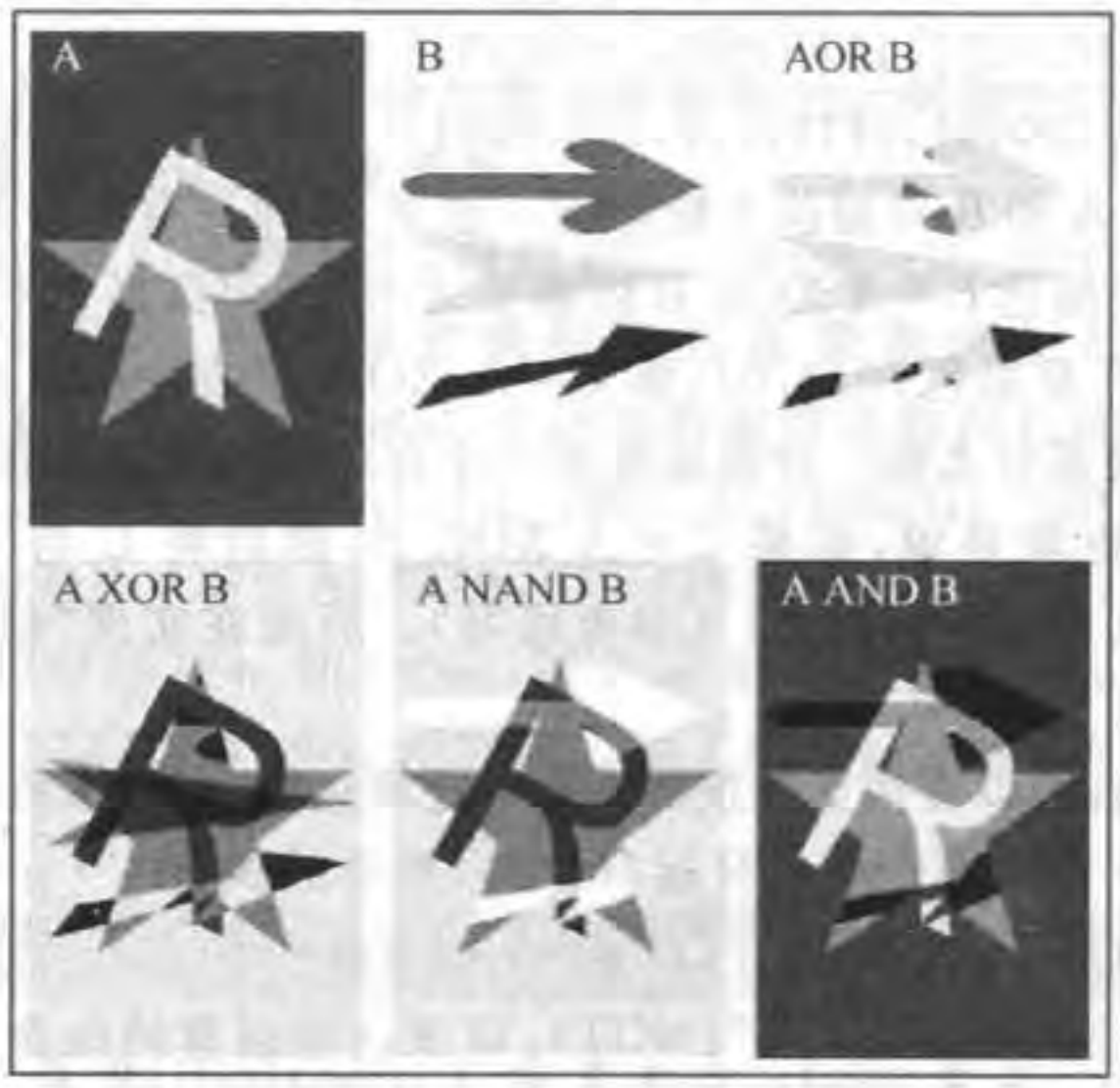


图 9.19 各种逻辑运算的效果

体连续绘制两次就会擦去这个物体,恢复成原来的图像。练习中还会介绍其他几个例子。

例 9.3.6 绘制"橡皮矩形"

当用户希望用鼠标在屏幕上绘制精确的图形时,"橡皮矩形"和"橡皮筋"能够提供很有价值的反馈。图 9.20 显示了几个例子,要绘制图 9.20(a)中的橡皮矩形,用户先把鼠标移动到要绘制的矩形的一个角点处(这个点称为中枢点(pivot point)),然后按下鼠标。当用户按住鼠标拖动时,就会看到一个弹性的矩形,它的一角始终固定在中枢点,对角位于鼠标指针当前位置。这个矩形变化时,它看起来就像掠过屏幕上绘制的图像而不改变它们。用户可以来回调整矩形,直到精确地覆盖住想要的区域。当鼠标键抬起以后,最终的矩形就在屏幕上保持可见了,它的坐标可供程序以后绘图使用。"橡皮筋"(图 9.20(b))是一条从中枢点延伸到鼠标位置的弹性直线,并随着鼠标移动拉长或缩短。实际上它仅仅是"橡皮矩形"的一个变种,通过绘制矩形的对角线就可生成。

橡皮矩形(或橡皮筋)需要不停地擦除和重绘,即使位置有一点点变化。而且不能用绘制背景色的方法擦除这个矩形,这会破坏线下面的图像。解决方案是利用 XOR 绘制模式。

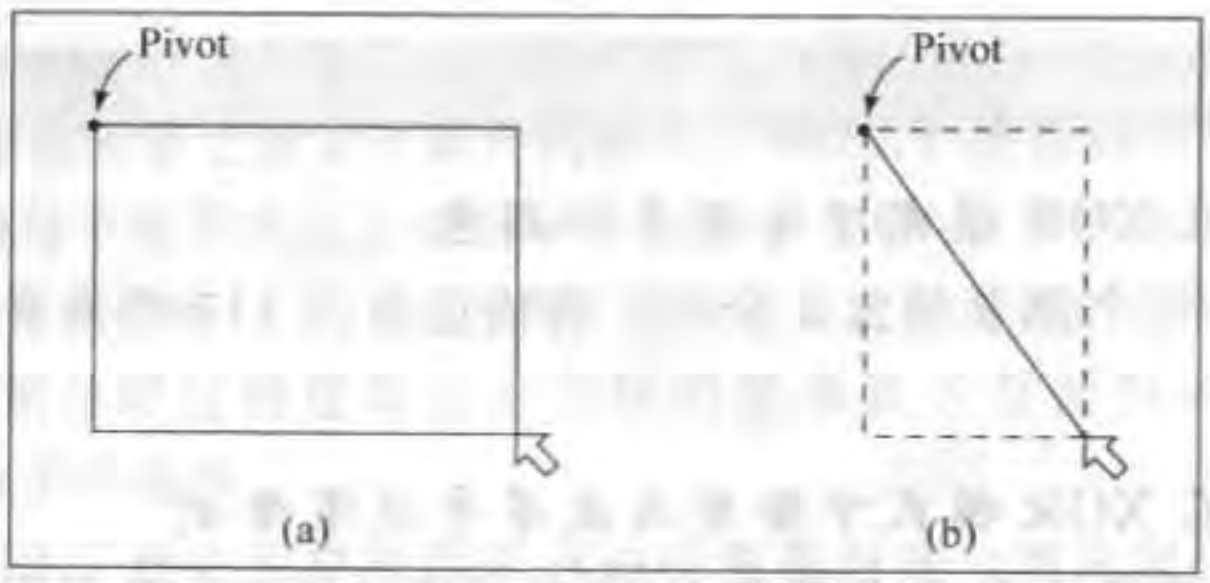


图 9.20　橡皮矩阵和橡皮筋

要擦去这个矩形，只须将其再绘制一次，同时原来的图像也会恢复。

图 9.21 给出了创建和管理橡皮矩形的鼠标回调程序。这些回调例程按照通常的方式在 main()中注册：调用 glutMouseFunc(myMouse)使 myMouse()成为鼠标按下或抬起事件的回调函数，调用 glutMouseMotionFunc(mouseMove)使 mouseMove()成为鼠标拖动 move 事件的回调函数。定义了一个全局的矩形对象 rr，并在两个回调函数中引用。初始矩形(大小为 0)在 myMouse()中建立。绘制矩形时不要使用黑色(全 0)作为矩形的绘制颜色！(为什么?)

值得注意的是，在 XOR 绘图模式下这个橡皮矩形的颜色并不是一致的。矩形上要绘制的每个像素的颜色是矩形的真实颜色和背景该处像素颜色进行 XOR 运算后的结果。这种效果通常并不令人讨厌，实际上是非常有用的，这使得矩形下的任何颜色的物体都能清楚的看到。

```
IntRect rr;                                    // 全局矩形变量

//<<<<<<<<<<<<<<<<myMouse>>>>>>>>>>>>>>>>
void myMouse(int button, int state, int mx, int my)
{
if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
    {
            glEnable(GL_COLOR_LOGIC_OP);       //启用逻辑运算
            glLogicOp(GL_XOR);                 //设置当前运算模式为 XOR
            rr.left=rr.right=mx; //set the pivot
            rr.top=rr.bott=screenHeight ?my;
    }
    if(button==GLUT_LEFT_BUTTON && state ==GLUT_UP)
        glDisable(GL_COLOR_LOGIC_OP);          //禁用逻辑运算
}
//<<<<<<<<<<<<<<<<mouseMove>>>>>>>>>>>>>>>>
void mouseMove(int mx, int my)
{
        rr.draw();                             //清除旧的矩形，仅在 XOR 模式有效
        rr.right=mx;                           //设置新的对角点
        rr.bott=screenHeight-my;               //翻转 y 坐标
        rr.draw();                             //绘制新的矩形
}
```

图 9.21　橡皮矩形的鼠标回调例程

练习

练习 9.3.3 在 XOR 模式下绘制多种颜色

假定帧缓冲区的每个像素包含 3 个位。请描述值为 110 的像素与各种可能像素值 XOR 的结果。

练习 9.3.4 在 XOR 模式下绘制两次等于没有绘制

对于任意颜色的像素 A 和 B,证明在 XOR 模式下绘制两次 B 得到的是没有改变的 A,也就是说(A XOR B) XOR B 的结果是 A。

练习 9.3.5 对称运算符

我们注意到,OR 运算符是对称的,即 A OR B 和 B OR A 是相等的。那么 16 个运算符$\oplus$中还有哪些在 $a\oplus b=b\oplus a$ 意义下是对称的?

练习 9.3.6 原地交换两幅图像

证明两个像素映射 A 和 B 可以通过执行 3 次 XOR 运算交换位置。当这个过程结束时,A 中的是原来 B 的像素值,B 中是 A 原来的像素值:

A=A XOR B;

B=A XOR B;

A=A XOR B。

练习 9.3.7 反转模式绘制

使用反转模式绘制图案也可以用来擦除一条线并恢复原来的像素值。一个 b 位的帧缓冲区有 $N=2^b$ 可能的像素颜色值,范围从 0 到 2^b-1。使用反转模式绘制像素的意义是将当前像素值 d 替换为 $f(d)=N-d$。需要注意的是这个函数是"循环"的:绘制两次就可以恢复原来的数值,因为 $f(f(d))=N-f(d)=N-(N-d)=d$。说明如何使用这种技术进行橡皮筋的绘制,并说明当橡皮筋跨越不同颜色的像素时,橡皮筋上的颜色是什么,是否和 XOR 模式下一样?对于一位($b=1$)的平面,反转模式与 XOR 模式有什么区别?

9.3.4 BitBLT 操作

可以把 draw()、read()和 copy()方法集成为单个函数,并在其中添加一些能够显著提高其能力和适用性的功能。这个操作被称为 BitBLT(发音为"bitblit"),意思是位块传输(bit boundary block transfer),还可以被称为 raster op【Inga78, Newell79】。当每个像素超过 1 位时,有时也把 BitBLT 称为 pixelBLT。这里我们把这两种情况都用 BitBLT 表示。

BitBLT 有时使用软件实现,但由于这种操作已经非常普遍,因此出现了专门设计的 VLSI 芯片可以高速执行 BitBLT(每秒处理几亿像素)。只需向 BitBLT 芯片中写入几个参数,该芯片就可以接管过来,并直接与系统总线交互,进行数据传输。

BitBLT 操作的定义

BitBLT 有几个不同的版本,它们在一些细节上有所差别。BitBLT 最简单的形式是把

源矩形(source rectangle)的像素复制到高度和宽度相同的目标矩形(destination rectangle)中。源和目标可以是在屏幕上或者屏幕外内存中。BitBLT 处理器以一种简单的方式描述这两个矩形：它存储每个矩形的左上角的 x 坐标和 y 坐标，以及矩形的高度和宽度(以像素为单位)。处理器来处理所有(x，y)对与内存对应地址之间所需的变换。

BitBLT 使用前面介绍过的逻辑合并方法把源像素 S 复制到目标矩形 D 中：$D=D\otimes S$。其中$\otimes$可以有多种选择。

有些版本的 BitBLT 允许用户对源矩形中的图像与某个预先定义好的像素映射进行“预混合”，这个预先定义好的像素映射被称为半色调图案(halftone pattern)。然后把这个预混合后的图像作为源 S 复制到目标。半色调图案保存在某个大小的像素映射中，比如 16×16 像素块，它可能是用 0 和 1 表示的棋盘图案。如果模板的大小小于源矩形，就把它复制拼接起来，直到填满矩形。对源像素映射 S 的选择有 4 种：

- 对源和半色调图案进行 AND 运算。
- 只使用源(忽略半色调图案)。
- 只使用半色调图案。
- 纯黑色。

多数 BitBLT 处理器还要维护一个裁剪(clipping)矩形。在绘制目标的每个像素之前，它们的坐标要先与裁剪矩形的边界比较，只有落在边界内部的像素才显示出来。裁剪过程可以在处理器内由硬件快速实现。裁剪矩形有效地限制了目标中实际绘制的范围。

9.4　自己实现直线绘制：Bresenham 算法

大多数图形环境都有绘制直线的内建工具。每种环境都至少有某种形式的 line()或者 lineto()函数。OpenGL 当然提供了更多的工具。因此我们通常不太关心这些工作的原理和细节——这些工具如何确定直线两个端点之间哪些像素要被绘制。

但是直线绘制是计算机图形学中最基本的算法，掌握它可以有助于了解上述程序的运行过程。实际上，程序员在开发或优化新的商业图形包时，仍然需要编写这些程序。

我们先从一种直观但是效率非常低的方法开始，以便提出我们的主要目的。然后再提出一种速度快得多的方法，这就是著名的 Bresenham 直线绘制算法。

假定我们要设置一些像素值，使得两个整数坐标(a_x，a_y)和(b_x，b_y)之间出现一条直线。图 9.22 显示出沿着理想直线(a_x，a_y)和(b_x，b_y)有某些像素被设置为“开”。如果这些像素被有效的选择出来，就可以在屏幕上显示出一条直线的样子(尽管不可避免的会有一些“锯齿”)。

根据一些基本代数知识，理想直线应满足下面的方程：

$$y = m(x - a_x) + a_y \tag{9.3}$$

其中 x 在 a_x 和 b_x 之间变化，m 是直线的斜率，有

$$m = \frac{b_y - a_y}{b_x - a_x} \tag{9.4}$$

例如，取(a_x，a_y)=(23，41)、(b_x，b_y)=(125，96)，则 m=55/102=0.5392。斜率只对非竖直的直线有意义，即 a_x 与 b_x 不同的时候。直线是水平或竖直时，可以很清楚确定哪些像

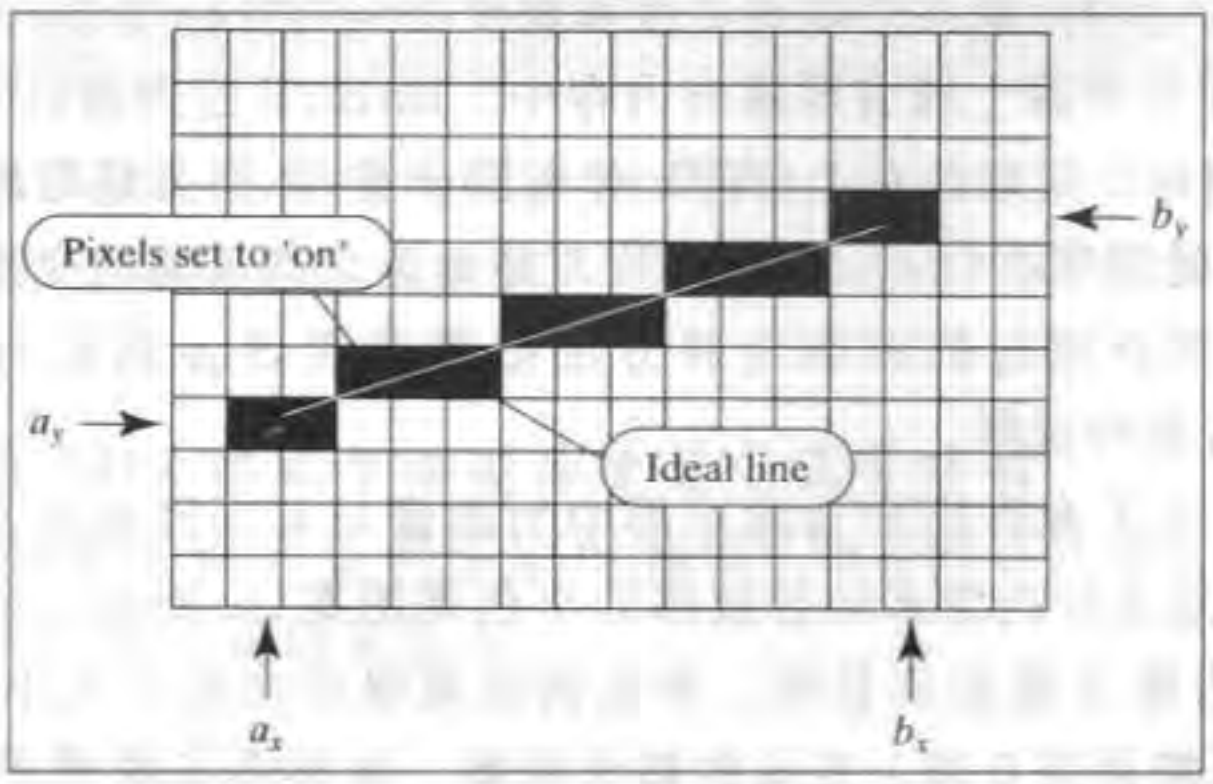


图 9.22 绘制一条直线段

素要被打开。对于其他类型的直线我们就需要一种算法来确定。

一种简单但效率低下的方法就是,沿着 x 方向从 a_x 增加到 b_x,递增的步长为一个像素。在每一步计算 $m(x-a_x)+a_y$,并取最接近此值的整数。在 a_x 小于 b_x 的情况下,可以使用以下代码实现:

```
float y=a.y                          //初始数值
for (int x=a.x; x<=b.x; x++, y+=m)
    setPixel(x, round(y));
```

其中 setPixel(x, y)和以前一样,把当前颜色写入帧缓冲区第 x 行第 y 列的像素中。本算法中需要对 y 值取最接近的整数值,这是一个非常费时的操作。

9.4.1 Bresenham 直线绘制算法

Bresenham 算法与前述方法相比有很大优势,它避免了浮点运算和舍入操作。它背后的思想也非常重要,其他类型的图元,如圆和椭圆的绘制算法也都应用了同样的思路。Bresenham 算法是增量算法的经典例子,它利用前一个像素的信息来计算当前像素的位置[①]。该算法只使用整数运算并且不用任何乘法运算,生成像素的内层循环紧密而且高效。

Bresenham 算法有几种变体,它们的计算方法略有不同。我们介绍的版本被称为中点算法(midpoint algorithm)。对于直线,该算法与 Bresenham 算法产生同样的像素,而且还可以扩展到更复杂的图形绘制算法,如圆和椭圆。

和以前一样,假定给定直线的两个端点(a_x,a_y)和(b_x,b_y),要确定直线上两端点之间像素的最优序列。

为了简化讨论,我们首先考察 $a_x<b_x$(也就是说 b 点在 a 点右侧),同时直线斜率在 0 和 1 之间的情况(稍后我们将取消这个限制)。为方便起见,我们定义线段在 x 方向和 y 方向上的长度,即 W 代表"宽度",H 代表"高度":

$$W = b_x - a_x$$

① 这种算法有时也称为 DDA 算法,由一种使用增量方式求解微分方程的机械设备而得名。

$$H = b_y - a_y \tag{9.5}$$

根据我们前面的假设可以得到，W 和 H 都为正，且 $H<W$。因此，当 x 从 a_x 增加到 b_x，对应的 y 值也从 a_y 增加到 b_y，但 y 增加的速度慢。当 x 的步长为一个单位时，y 值的最优结果一定是有时保持不变，有时增加 1。中点算法能够快速地确定到底会发生哪一种情况。（y 永远不会减少或者增加幅度大于 1，为什么？）

根据第 4 章的介绍，我们知道通过(a_x, a_y)和(b_x, b_y)的理想直线方程为：

$$-W(y-a_y)+H(x-a_x)=0$$

对于所有落在理想直线上的点(x, y)，代入方程的左侧始终为 0。

我们为这个表达式起个名字以备后用。事实上先把这个表达式乘以 2 再命名更为重要（稍后会看到，这样做是为了避免在关键公式中出现一个麻烦的系数 1/2）。现在我们定义下面函数：

$$F(x,y) = -2W(y-a_y)+2H(x-a_x) \tag{9.6}$$

$F(x, y)$的重要性质就是，它的符号表明了(x, y)是在理想直线的上方还是下方：

- 如果 $F(x,y)<0$，则(x, y)在直线上方。
- 如果 $F(x,y)>0$，则(x, y)在直线下方。

注　推理提示：假设(x, y)在直线上，那么我们知道 $F(x,y)=0$。把 y 向上稍微调整，并且保持 x 不变，这使得函数 F 的数值变小。因此把 y 增加到直线上方会使 F 变为负值。

练习

练习 9.4.1　Bresenham 算法实验

对于两个端点(3, 7)和(9, 11)之间的直线段，有

$$F(x,y) = (-12)(y-7)+(8)(x-3)$$

直线上的点，例如(7, 29/3)满足 $F(x,y)=0$。而点 $A=(4, 4)$和 $B=(5, 9)$处函数 $F(x,y)$的符号是什么，它们与直线的相对位置关系是什么样的？

答案：A 处在直线下方，F 值为 44。B 处在直线上方，F 值为-8。

现在如何确定哪些像素在直线上呢？图 9.23 给出了一些接近理想直线的像素。位于格子交叉点的圆圈表示像素的中心。假设我们已知对应 p_x 的最优 y 值是 p_y，现在要确定下一个 x 即 p_x+1 对应的最优 y 值。我们希望知道直线在 p_x+1 处是距离点 $L=(p_x+1, p_y)$更近，还是距离点 $U=(p_x+1, p_y+1)$更近。图中只是给出了理想直线上一个可能位置，它可能低于 L 或者高出 U 么？（为什么？）。

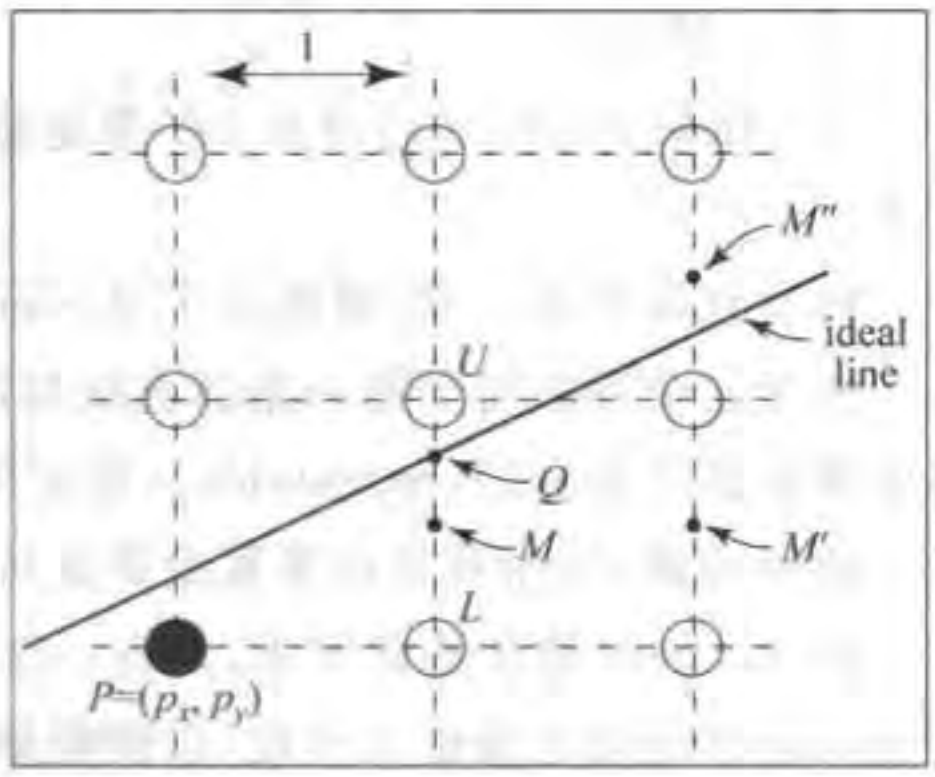


图 9.23　中点算法推导图示

我们可以通过判断直线在 U 和 L 的中点 M 的上方还是下方来确定选择像素 U 还是像素 L。$M=(p_x+1, p_y+1/2)$。通过计算 M 点处的函数 F 的符号，我们可以确定理想直线与 M 的相对关系：

- 如果 $F(M_x,M_y)<0$,则 M 在理想直线上方,所以选择 L。
- 如果 $F(M_x,M_y)>0$,则 M 在理想直线下方,所以选择 U。

因此,如果 $F(M_x,M_y)>0$,要选择的像素比前一像素高一个点,我们增加 y 值,否则就不增加 y 值。剩下的就是代数问题了:如何快速计算 $F(,)$,关键是根据这个函数每一步的增量来递增地计算函数值。

我们计算出 M 点的函数值 $F(M_x,M_y)$以备后用:

$$F(M_x,M_y)=-2W(p_y+1/2-a_y)+2H(p_x+1-a_x) \tag{9.7}$$

当 x 从 p_x+1 移动到下一个 x 值 p_x+2 时,考虑函数 $F(\ )$的变化。如图 9.23 所示,M 可能是 M',也可能是 M''。如果在前一步 y 没有增加,那么结果是 $M'=(p_x+2,p_y+1/2)$;如果前一步 y 增加了 1,那么结果就是 $M''=(p_x+2,p_y+3/2)$。

情况 1:如果前一步计算 F 是负值(因此这一步中 y 值不增加),则

$$F(p_x+2,p_y+1/2)=-2W(p_y+1/2-a_y)+2H(p_x+2-a_x)$$

减去(9.7)式,得到与 $F(M_x,M_y)$的差异:

$$F(p_x+2,p_y+1/2)=F(M_x,M_y)+2H \tag{9.8}$$

情况 2:如果前一步计算 F 是正值(此步 y 增加 1),则

$$\begin{aligned}F(p_x+2,p_y+3/2)&=-2W(p_y+3/2-a_y)+2H(p_x+2-a_x)\\&=F(M_x,M_y)-2(W-H)\end{aligned} \tag{9.9}$$

不管在哪种情况下,测试量的增量都是常数的:如果不增加 y,增量为 $2H$;如果增加 y,则增量为 $-2(W-H)$。

现在剩下的唯一问题是这个过程如何开始。当 $x=a_x$ 时,我们知道 $y=a_y$。因此 M 的第一个值是 $M=(a_x+1,a_y+1/2)$,而

$$\begin{aligned}F(x,y)&=-2W(a_y+1/2-a_y)+2H(a_x+1-a_x)\\&=2H-W\end{aligned} \tag{9.10}$$

注:如果前面没有把函数加倍,这个结果就会是 $H-0.5W$,这样就无法做到对所有量都使用整数值。

总结 把 F 初始化为 $2H-W$,把 x 初始化为 a_x,y 初始化为 a_y。然后对于每一步:

(1) 将(x,y)处的像素设置为需要的颜色。

(2) x 增加 1。

(3) 如果 $F<0$,通过增加 $2H$ 来更新 F;否则将 y 增加 1,并通过增加 $-2(W-H)$来更新 F。

将所有综合在一起,就得到了 Bresenham 算法(在前面所述的约束条件下)。

由于这个算法对于每一条新直线都需要重新初始化,并且只依赖于直线的端点数据,因此该算法是可重复的(repeatable),即使用不同的颜色重新绘制直线每个像素都可以完全覆盖。换句话说,使用背景色重复绘制就可以完全擦去原来绘制的直线。

Bresenham 算法非常简单,只有一个内部循环,在循环内也只有几次加法和几次比较。它可以很容易用汇编语言实现,以获得最大速度。通常也有一些特殊用途的图形硬件实现此算法以获得更高的运行速度。

```
bresenham(IntPoint a, IntPoint b)
{//约束条件: a.x<b.x, 同时 0<H/W<1
    int y=a.y, W=b.x-a.x, H= b.y-a.y;
    int F=2*H-W;                    //当前误差项
    for(int x=a.x; x<=b.x; x++)     //内层循环
    {
        setPixel(x, y);
        if(F<0)
            F+=2*H;                 //为下一个像素更新函数值
        else{
            y++;
            F+=2*(H-W);
        }
    }
}
```

图 9.24　中点算法(带约束条件下)

例 9.4.1　示例

观察一个具体的例子中 F 的变化,非常有助于理解算法。令 $(a_x,a_y)=(4,1)$,$(b_x,b_y)=(16,4)$,则有 $W=12$,$H=3$。因为这条直线的斜率是 1/4,所以期望的结果应该是每个四步左右 y 值递增一次。F 初始化为 -6。每次 x 增加,都要做下面两件事情中的一件:如果 F 是负数,就把它加 6;否则把它减 18,同时 y 递增 1。

得到的结果序列(可以执行 setPixel(x, y)来观察)如下:

x:	4	5	6	7	8	9	10	11	12	13	14	15	16
y:	1	1	2	2	2	2	3	3	3	3	4	4	4
F:	−6	0	−18	−12	−6	0	−18	−12	−6	0	−18	−12	−6

图 9.25 显示了中点算法应用于此例的情况,图中给出了结果直线和 F 的变化。从图中可以很清楚看到直线的"锯齿"出现在每个短的水平线段跳到下一个高一点的水平线段。

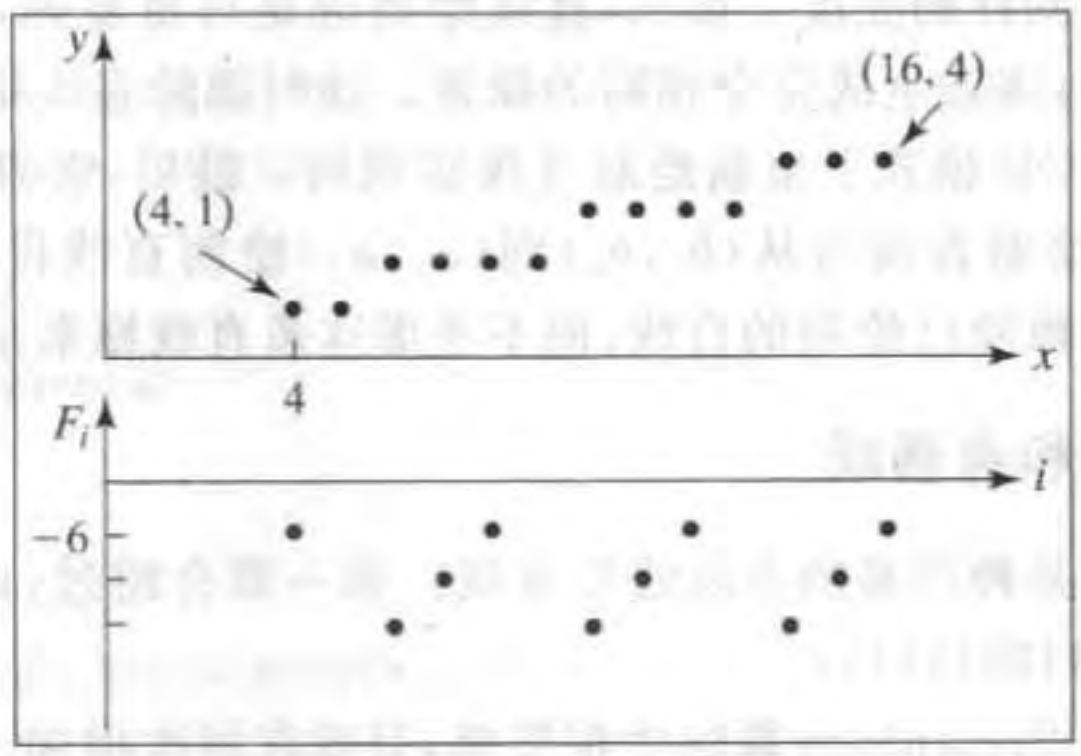


图 9.25　中点算法应用示例

去掉对 Bresenham 算法的限制

前面讲述的算法只能处理 $a_x<b_x$,且斜率在 0 和 1 之间的情况。通过添加处理各种情况的代码,其余的情况也不难处理。

当 $a_x>b_x$ 时得到同样直线 中点算法在每一步都会在两种情况中选择一个,把像素设置在中点的正确的一侧。因此我们只须调整图 9.24 的算法,实现在 $a_x>b_x$ 情况下的中点算法,这不需要很大的改动(参见练习)。需要注意的一个细节是:图中的代码分为 $F<0$ 和 $F\geqslant 0$ 两种情况,其中 $F=0$ 的情况与 $F>0$ 合并在了一起(此时 y 增加)。所以我们必须确保相反方向跟踪直线时,要把 $F=0$ 包含在 y 增加的情况中。

另一种方法可以通过检查是否有 $a_x>b_x$,如果是就交换 a_x 和 b_x,并交换 a_y 和 b_y,然后使用图 9.24 的算法。这种方法永远不会绘制"向左"的直线,而是重新定义直线的端点,使它不必绘制这样的直线。然而如果要用这种算法绘制折线段,这种技术就并不十分成功。因为交换顶点会损失折线中端点的自然顺序,而在把折线绘制成点线或虚线时,这个顺序是非常重要的。

斜率大于 1 的直线 这种情况下,只需交换 x 和 y 的角色,将 y 从 a_y 到 b_y 递增,并且使用同样的方法测试并决定何时增加 x(此时 x 的变化比较慢)。

斜率为负数的直线 如果斜率在 0 和 -1 之间,H 自动具有正确的符号。在 x 上递增并使用同样的测试,但另一个变量不再是递增,而是递减。如果斜率小于 -1,则用 $-W$ 代替 W,交换 x 和 y 的角色,就像处理斜率大于 1 的情况一样。

水平线和竖直线 这种情况在图形中经常出现,因此必须提高对这些直线的检测效率(例如使用代码 if(a. x= = b. x)…)和使用简化的绘制算法。对于竖直线,改进是有限的,因为大部分时间用在 setPixel()上。对于水平线,可以使用一个函数沿着扫描线一次写入多个位,从而显著提高其绘制速度。

参见练习了解如何实现这些一般化过程。

绘制出的直线应具有的性质

我们需要直线绘制算法应具有哪些性质呢?首先,直线应该尽可能的直,并且确实穿过给定的两个端点。第二,这条直线应该尽量"平滑"并且在直线各个部分具有一致的亮度,不同斜率的直线应该具有同样的亮度。第三,直线绘制应是可重复的,也就是说,如果使用同样的端点绘制直线两次,应该生成完全相同的像素。这对擦除直线很重要,因为擦除就是通过使用背景颜色或在 XOR 模式下重新绘制直线实现的。最后,它应该与绘制的方向无关,即从(a_x,a_y)到(b_x,b_y)绘制直线与从(b_x,b_y)到(a_x,a_y)绘制直线得到的结果应完全一样。这样应用程序可以完全擦除已绘制的直线,而不考虑这条直线原来是怎么绘制的。

用图案绘制直线和点画线

我们有时需要绘制某种图案的点画线和虚线。第一章介绍过,虚线的图案以位序列的形式存储,比如 0011111100111111。

把这种图案集成到 Bresenham 算法中很简单,只要在每次增加 x 的同时递增一个指向图案的指针,然后使用指向相应位的比特值设置 setPixel()所使用的当前绘制颜色。对于长直线,从图案末尾折回起始处循环递增地重复使用图案。

当使用某种图案的虚线绘制折线时，虚线的图案需要在线段与线段间保持连续。为此，需要把这个图案和指向图案的指针设置为 Bresenham 算法的全局变量，这样在算法连续调用时能够不断访问和更新这些变量，在算法结束时保证其数据不丢失。参见练习，了解关于图案绘制的其他问题。

OpenGL 使得使用点画线绘制直线和多边形变得简单。glEnable(GL_LINE_STIPPLE)命令为后面绘制的所有直线启用了点画线绘制模式。点画线模式可以通过 glDisable(GL_LINE_STIPPLE)命令关闭。一旦点画线模式启用，后面绘制的直线就按照 glLineStipple(factor, pattern)命令定义的图案绘制方式绘制。参数 factor 是 int 类型，它决定了每个图案重复的次数。例如，factor=2 表示图案的每个比特位使用 2 次以后才切换到下一个比特位。factor 的范围是 1～256。图 9.26 显示了三条直线，factor 分别为 1、2 和 3。

图 9.26　使用不同 factor 的三条点画直线

参数 pattern 是一个 gluByte 类型的数组，使用 0 和 1 的串(1 表示像素开，0 表示像素关)表示图案。为了简化，程序员通常使用十六进制方式表示图案的元素，每个十六进制数值可转换为 4 个比特位。例如一个图案的十六进制表示是 0x0000，等同于二进制的 0000000000000000，即 16 个像素均为黑色。图案 0x0101(二进制为 0000000100000001)可以生成一个点画线。图案 0x0F0F(二进制为 0000111100001111)可以生成一个较宽的虚线。图 9.27 给出了“OpenGL Red Book”【Woo04】中一个完整的程序。

```
#include<GL/glut.h>
#include<stdlib.h>

#define drawOneline(x1,y1,x2,y2) glBegin(GL_LINES); \
    glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

void init(void)
{
    glClearColor(1.0,1.0,1.0.0,0);                //使用白色不透明背景
    glShadeModel(GL_FLAT);
}

void display(void)
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);

/* 选择黑色绘制直线 */
```

图 9.27　一个完整的演示点画线绘制的程序【Woo04】

```
    glColor3f(0.0,0.0,0.0);

/* in first row, 3 lines, each with a different stipple  */
    glEnable(GL_LINE_STIPPLE);

    glLineStipple(1,0x0f0f);            /*  点画线  */
    drawOneline(50.0, 125.0, 150.0, 125.0);
    glLineStipple(3, 0x0f0f);           /* 虚线 */
    drawOneline(250.0, 125.0, 350.0, 125.0);

    glDisable(GL_LINE_STIPPLE);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, (GLdouble) h);
}

void keyboard(unsigned char key, int x, int y)
{
    //ESC 键退出
    swith(key){
        case 27:
            exit(0);
            break;
    }
}

int main(int argc, char**argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GULT_SINGLE | GLUT_RGB);
    glutInitWindowSize(400,150);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

图 9.27 (续)

多边形点画显示

OpenGL 也为多边形点画显示提供了相应机制。多边形的点画显示与直线类似。首先也要使用 glEnable(GL_POLYGON_STIPPLE)来启用多边形点画模式；然后使用 glPolygonStipple(bug)语句定义图案，该图案由一个十六进制数值的数组构成。bug 是我们定义的图案名称，如图 9.28(b)所示。9.28(c)显示的是一个半色调图案，程序员经常用它来达到一种平滑的灰色效果。

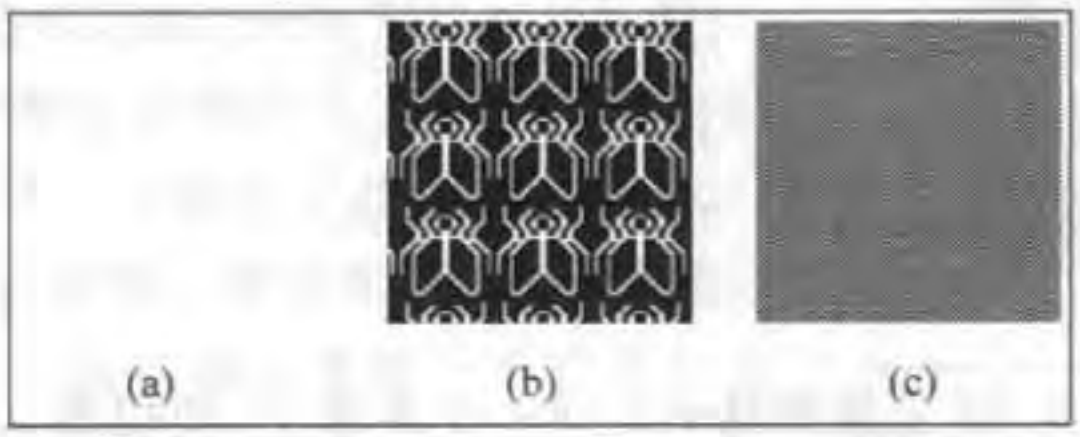


图 9.28　三个矩形

(a) 无图案；(b) "bug"图案；(c) 半色调图案

bug 和半色调图案由下面的数组定义：

```
Glubyte bug[]={
    0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
    0X03, 0X80, 0X01, 0XC0, 0X06, 0XC0, 0X03, 0X60,
    0X04, 0X60, 0X06, 0X20, 0X04, 0X0C, 0X30, 0X20,
    0X04, 0X18, 0X18, 0X20, 0X04, 0X0C, 0X30, 0X20,
    0X04, 0X06, 0X60, 0X20, 0X44, 0X03, 0XC0, 0X22,
    0X44, 0X01, 0X80, 0X22, 0X44, 0X01, 0X80, 0X22
    0X44, 0X01, 0X80, 0X22, 0X44, 0X01, 0X80, 0X22
    0X44, 0X01, 0X80, 0X22, 0X44, 0X01, 0X80, 0X22
    0X66, 0X01, 0X80, 0X66, 0X33, 0X01, 0X80, 0XCC
    0X19, 0X81, 0X81, 0X98, 0X0C, 0XC1, 0X83, 0X30
    0X07, 0XE1, 0X87, 0XE0, 0X03, 0X3F, 0XFC, 0XC0
    0X03, 0X31, 0X8C, 0XC0, 0X03, 0X33, 0XCC, 0XC0
    0X06, 0X64, 0X26, 0X60, 0X0C, 0XCC, 0X33, 0X30
    0X18, 0XCC, 0X33, 0X18, 0X10, 0XC4, 0X23, 0X08
    0X10, 0X63, 0XC6, 0X08, 0X10, 0X30, 0X0C, 0X08
    0X10, 0X18, 0X18, 0X08, 0X10, 0X00, 0X00, 0X08
};
Glubyte halftone[]={
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
```

```
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
};
```

如果程序员想把图 9.28 三幅图显示在同一个屏幕上，使用图 9.29 给出的代码片段可以完成。9.28(a)使用熟悉的函数 glRectf(...)进行单一颜色绘制，然后启用点画模式，并使用 gluPolygonStipple(bug)设置 bug 为当前图案；然后使用另一个 glRectf 调用绘制。最后把当前图案切换到半色调图案，并再次绘制矩形。

```
glRectf (25.0, 25.0, 125.0, 125.0);          //白色矩形
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(bug);
glRectf (125.0, 25.0, 225.0, 125.0);         //苍蝇图案
glPolygonStipple(halftone);
glRectf (225.0, 25.0, 325.0, 125.0);         //半色调图案
glDisable(GL_POLYGON_STIPPLE);
```

图 9.29 一个多边形点画显示的代码片段

练习

练习 9.4.2 去掉 Bresenham 算法的约束

把 Bresenham 算法的变体应用于以下情况：(a)$a_x > b_x$；(b)直线斜率大于 1；(c)直线斜率为负。把各种情况的处理方法合并成一个单独的算法，使其可以处理任意线段，尽量提高程序的效率。

练习 9.4.3 数值例子

对于端点为(8,23)和(21,11)的直线，应用 Bresenham 算法，给出 x、y、F 值序列。证明算法从另一个点开始时得到的结果相同。

练习 9.4.4 按图案绘制

扩展 Bresenham 算法，使其能够绘制 16 位图案的直线。进行进一步的调整，使得绘制折线时该图案从一条线段接到另一条线段时保持连续不中断。

9.5 定义和填充像素区域

光栅显示器能够显示纯色或图案填充(fill)的像素区域。在这里区域(region)一词的意思是以某种方式彼此“相邻”或由某种共同性质联系起来的一组像素。图 9.30 显示了一些使用不同颜色和图案填充的区域。

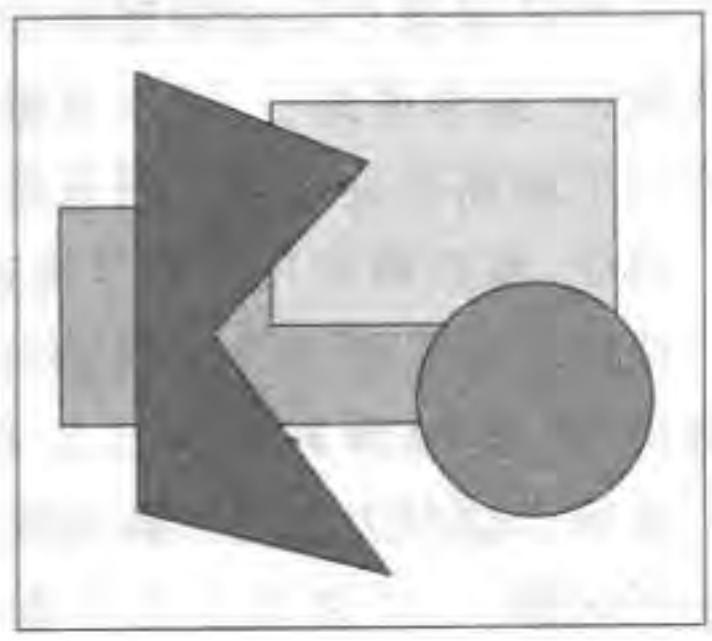


图 9.30　一些使用图案绘制的区域

现在许多绘图程序都允许用户交互的生成一幅图像。所有这些程序都包含一个填充工具，这个工具允许用户单击一个区域，然后选择一种图案，这个区域随之马上被新的图案填充。下面让我们来看一下它是如何实现的。

9.5.1 区域定义

有多种定义区域的方法。这些方法的主要区别是：“像素定义”还是“符号表示”。

- “像素定义的”(pixel-defined)区域是用像素映射中的实际像素描述的。对一个区域 R 的描述可能要列出 R 内部的所有像素：(34,12)、(34,13)、(34,14)等。或者可以定义 R 为由所有像素值为 77，且按某种方式与像素(43,129)连通的像素组成。连通的概念需要被仔细定义。想知道区域 R 的样子，需要仔细查看像素映射，观察在这个定义下，哪些像素属于 R。不管怎么说，列举区域的所有像素都是令人厌烦的，并且容易出错。
- “符号表示”这种描述方式并不列举像素，而是提供区域 R 中所有像素具备的某种性质。这样的描述通常比纯粹的枚举像素更高级，或者说更抽象。下面是一些符号描述区域的方法：
  - 所有到给定点(23, 47)的距离比到点(12, 14)、(22, 56)、(35, 45)中任何一个距离都小的像素。
  - 以(5, 23)为圆心，8 为半径的圆形内部的所有像素。
  - 所有位于以(32, 56)、(120, 456)、(345, 1129)、(80, 87)为顶点的多边形内部的点。这种“多边形定义”(polygon-defined)区域是一种非常重要的形式，我们将在后面详细讨论。

根据要研究的区域描述方法不同，程序员可以选择完全不同的数据结构和算法对区域进行处理和分析。

我们不会对像素定义的区域进行过多的关注，而是着重讨论符号定义区域，特别是多边形定义区域的各种处理方法。

9.6 处理符号定义的区域

一些区域填充算法要逐个像素地读取帧缓冲区中的像素,以确定每个像素的颜色,从而在区域中"探索道路"。可以设想,如果填充算法处理的是一个用更高级的更符号化的描述定义的区域而不是靠像素列举描述的区域,那么这个填充算法的效率会高很多。

因此,我们研究几种符号描述区域的方法,这几种方法各有优缺点。在图形学中使用最为广泛的方法是把一个区域定义为一个多边形的内部,这也是 OpenGL 所采用的方法。我们将详细介绍这种方法。

这种方法分为两类:第一类是用矩形的集合表示多边形,第二类用定义边界的路径确定一个区域。

9.6.1 矩形定义的区域

这种方法把一个区域描述为一组矩形。这些矩形可以小到由单个像素构成,大到整个像素映射。这个区域可以有空洞甚至有不连通的小区域。

图 9.31 中显示了一个简单的例子,其中每个像素的值是黑色或白色。我们把这个区域分解为一组对齐的矩形,这种表示正好是找到的所有矩形的列表(用眼睛快速扫视这个图形可以发现约有 11 个矩形满足这个要求,但是有多种分解方式)。

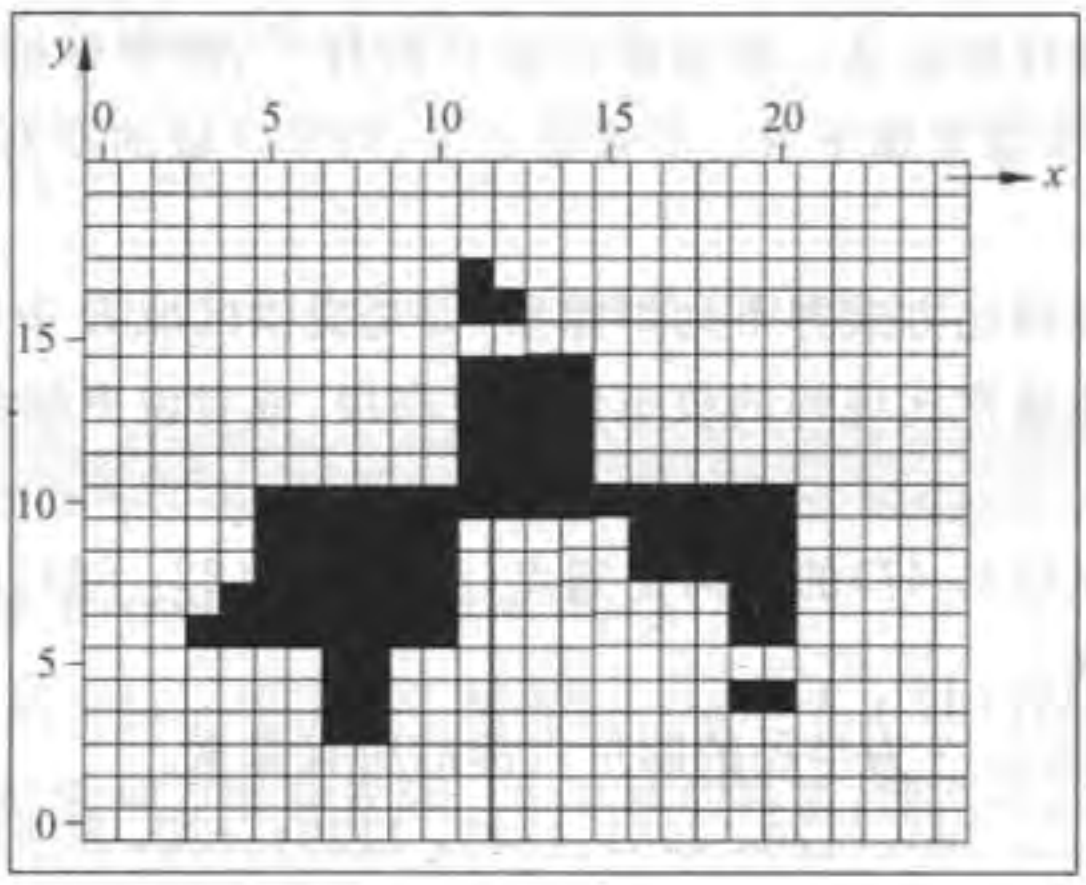


图 9.31 用矩形表示的区域

我们介绍一种规范的将像素映射分解为矩形列表的方法。该方法简单有效,它可以快速的从像素映射构建形状(shape),也可以快速填充这个形状描述的区域。

该方法从像素映射的最上端开始,一行一行地向下扫描来识别矩形。只要某个扫描线中的像素串集合与上一行扫描线中的像素串集合不同,就会出现一组新的矩形。其中,像素串(run)表示同一扫描线上相同颜色的相邻的像素集合。

像素映射的这种表示方法有时也称为它的形状(shape)表示【Steinhart92】。这样的表

示方法可以描述像素映射中的任意像素集合，包括不连通区域组成的集合。

当区域具有横向跨度一致性和扫描线一致性时，用这种方法表示最紧凑。横向跨度一致性(span coherence)是一条扫描线内的属性，指的是沿着一条扫描线多数像素具有相同数值。扫描线一致性(scanline coherence)是扫描线之间的属性，指的是一条扫描线的图案与相邻扫描线的图案类似。如果区域具有高度的一致性，那么用形状表示时，平均每个像素需要的数据更少。

缩放和平移区域

当我们想对区域进行一些处理时，"形状"表示法也很有用，例如：

- **区域平移**　假设把一个用形状表示法表示的区域在 x 方向平移 30 个像素，y 方向平移 55 个像素，只需遍历它的列表，把所有矩形的 x 值增加 30，y 值增加 55。
- **区域扩大一倍**　首先平移这个区域，使得第一个像素串位于原点(0，0)上，然后把数据结构中的所有值扩大一倍。最后再把这个区域平移回所需的位置上。

9.6.2　路径定义的区域

很自然会想到使用区域的边界来定义这个区域，这个边界通常是按照某种顺序排列的路径。描述路径的方法有很多，下面是其中一些：

- 通过数学公式。例如 $(x-122)^2+(y-36)^2=25$ 确定了一个直径为 10 像素的圆形路径。
- 通过折线。通过像素位置序列 $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ 可以确定一条折线路径。如果这条路径是封闭的，即 $(x_1, y_1)=(x_n, y_n)$，就确定了一个多边形。
- 通过相邻像素序列。链码(chain code)是一种特别值得注意的表示邻接像素路径的方法。先为路径指定一个起始像素，比如(34，67)。然后记录从像素到像素移动的序列，比如"向上、向右、向下"等等。为可能的移动方向集合编码的方式有很多。

图 9.32 中显示了两种编码方式。第一种方法(图 9.32(a))记录 4 个"罗盘"方向的移动，每个方向与 0 到 3 中的一个数字关联。于是这样的移动序列定义了一个连续的路径。图中的路径 A 由它下面的数字序列表示，该序列从右下角开始记录。这条路径恰好是封闭的，但是一般说来，路径并不一定是封闭的(序列中的 0、1、2、3 的个数满足什么条件时，路径是封闭的?)。值得注意的是，由于使用了 4 个可能的方向，所以每个方向可以只用两个比特位表示。

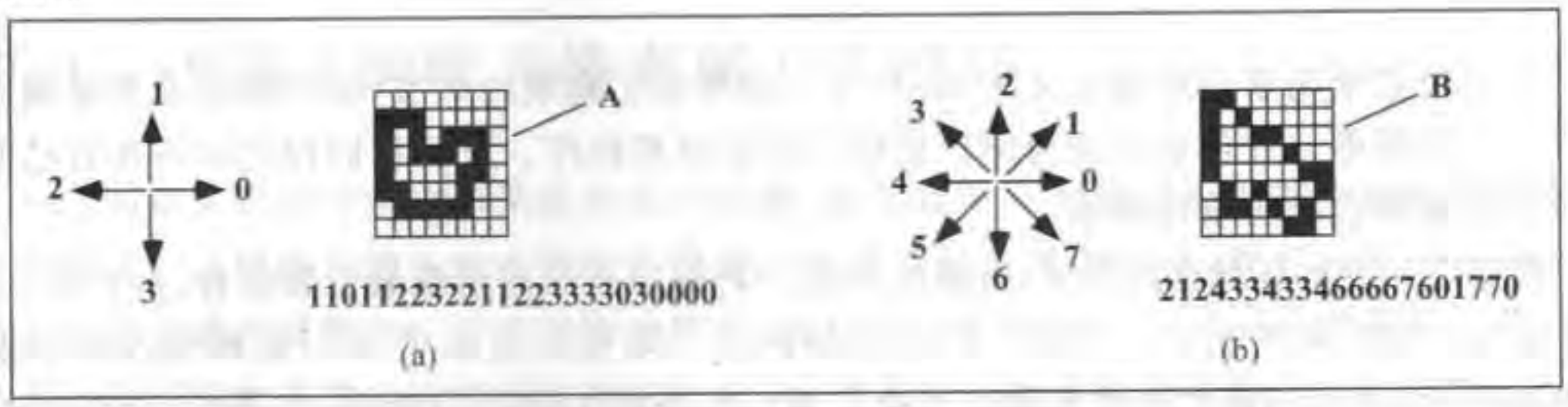


图 9.32　通过链码定义的路径

图 9.32(b)中给出的方法允许有 8 种可能的方向。其中,"1"表示路径中的下一个像素在上面一行偏右一个像素。另外 3 个对角线方向的表示方法与此类似。(图中所示序列的起始点在哪里?)链码与第 3 章中的相对绘图很类似,我们会在附录 5 中进一步讲述。

图 9.33 显示了一种用来存储路径的数据结构。最开始的两个元素给出路径的起始点坐标,接下来是要移动的步数,然后就是移动序列。

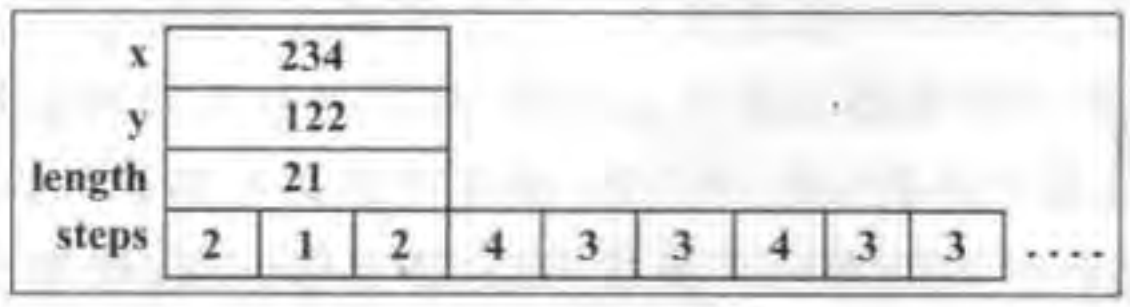


图 9.33 适用于链码编码路径的数据结构

链码能够为某些路径和区域提供非常简洁的表示。(对于 8-方向链码,每个方向需要用几个比特位表示?)此外,还可以很方便地执行某些处理过程:

(1) 平移路径:非常简单,只需改变起始点坐标即可。

(2) 以系数 k 放大路径:将每个符号重复 k 次。

(3) 把路径逆时针旋转 90°:对于 4-方向链码,只需把每步的数值增加 1,如果是 3 就增加为 0。8-方向链码的处理也是同样道理。

而另一方面,有些操作处理起来会变得困难。例如填充一个用链码表示的区域就很复杂,这是因为从链码确定扫描线上的像素串并不容易。所以最好是先把链码表示变换为"形状"表示。

练习

练习 9.6.1 链码路径

编写代码,使用 3 比特移动编码表示一个对齐的正方形,并选择合适的起点使得链码总长最小。

练习 9.6.2 旋转的正方形的链码

重复练习 9.6.1,表示一个旋转 45°的正方形。如果不能精确的表达,就寻找最佳的近似。

9.7 填充多边形定义的区域

人们经常使用多边形来定义区域,并且已经开发了高效的算法,使用纯色或图案填充多边形。有些图形软件包也已经为用户提供了高效填充程序。下面我们讨论如何在自己的软件包中实现多边形的填充操作。

假定要填充的区域多边形 P,由描述为指定 P 的顶点序列的像素位置集合,$p_i=(x_i,y_i)$,$i=1\cdots N$。图9.34 给出了一个有 7 个顶点的例子。填充多边形 P 可以在帧缓冲区中逐条扫描线进行,通过扫描线与边界的交点决定填充每条线上的适当部分(如图 9.34 中扫描线 $y=3$ 的情况)。因为边界对之间的像素串一定在 P 内部,可以用适当颜色对其进行填充。

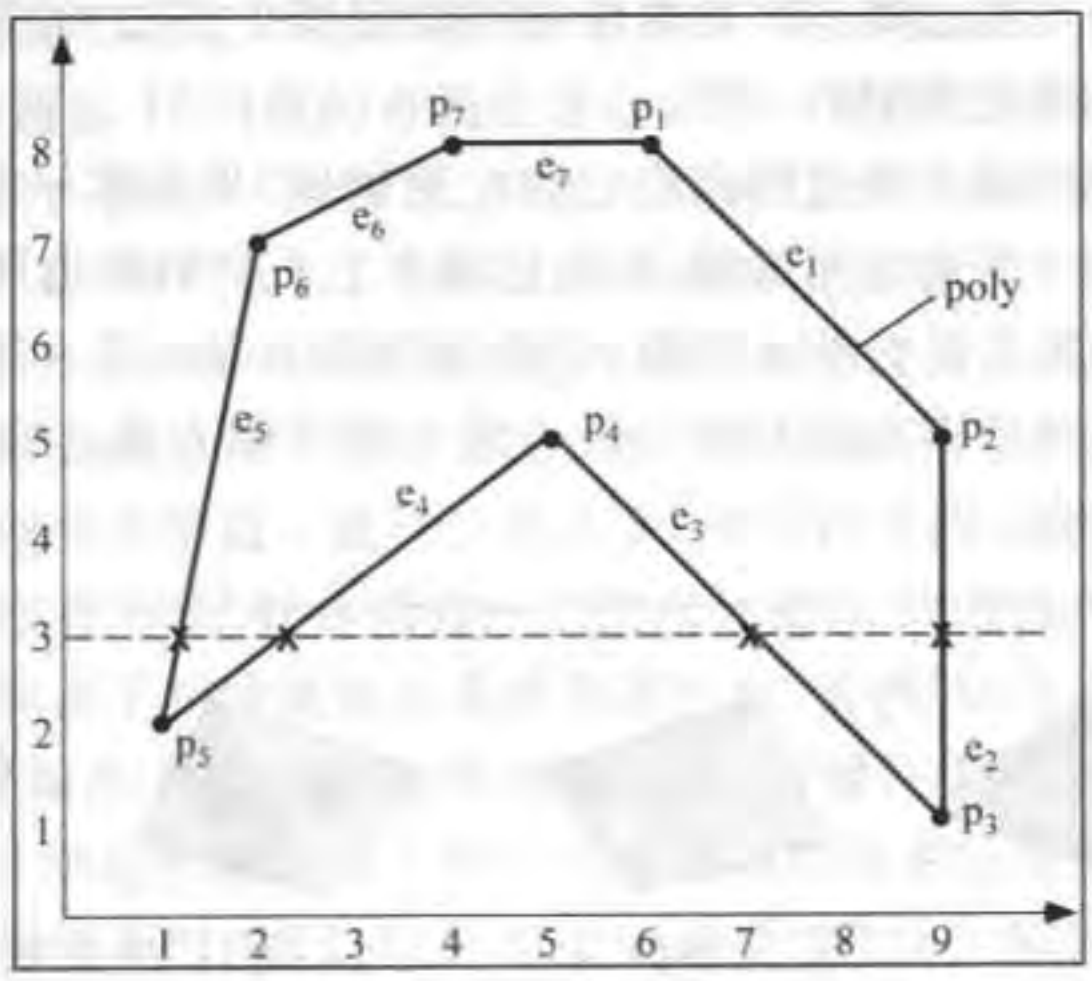


图 9.34　填充扫描线上的适当部分

下面的伪代码给出了填充的过程：

```
for (每条扫描线 L)
{
    计算 L 与 P 的所有边的交点
    按 x 大小为这些交点排序
    填充所有交点对之间的像素串
}
```

例如，图中的扫描线 $y=3$ 与 4 条边 e_2, e_3, e_4, e_5 相交。对这 4 个交点的 x 值舍入取整（取整的方法后面会介绍），然后排序得到序列 1，2，7，9。接着填充了两个像素串：第一个从第 1 列到第 2 列，第二个从第 7 列到第 9 列。

需要注意的是，为排序后的边交点配对的过程中使用了内部-外部测试（inside-outside test）。沿着扫描线移动，在每个交点处或者是进入 P 或者是离开 P，也就是说“内部性”（insideness）会在交点处发生改变（内部性有时也称作奇偶性（parity），也可以说在每个交点处奇偶性发生变化）。如果是进入多边形，那么要填充接下来的像素；如果是离开多边形，接下来的像素就不应该填充。如果整个多边形 P 都在扫描线起始处的右侧，那么奇偶性的初始值为“出”。算法充分利用了横向跨度的一致性，扫描线上与一个内部像素相邻的几个连续像素常常也是内部像素。这样可以通过最少的计算填充整个像素串。

9.7.1　边界上的哪些像素属于多边形

一个场景包含几个多边形是很常见的事情，而且其中几个多边形可能会彼此相邻共用一条公共边。这样的多边形称为彼此邻接的。如果不认真考虑的话，我们的算法可能会把公共边上的像素先设置为一个多边形的颜色，然后又设置为另一个多边形的颜色。像这样把一条边绘制两次可能会导致混乱的视觉效果：如果应用程序恰好是在 XOR 模式下绘图，那么这条边可能会呈现出奇怪的颜色。如果绘制结果是传送给自动记录照相机，那么绘制

两次可能导致这条边的亮度过高。因此算法必须确定哪个多边形"拥有"这条边,这样每条邻边才会只属于这两个多边形中的一个。

一种效果较好的规则是令多边形拥有它的左侧边(如果是水平边,就是下面的边)。所以如图 9.35 所示,当两个多边形相邻时,公共边属于它右侧的多边形,因此只会用这个多边形的颜色绘制一次。如果公共边是水平的,它将使用它上方的多边形的颜色绘制。图 9.35 用简单的三角形显示了两个多边形共有一条边的 4 种可能方式。每种情况下,公共边都应使用多边形 B 的颜色绘制。

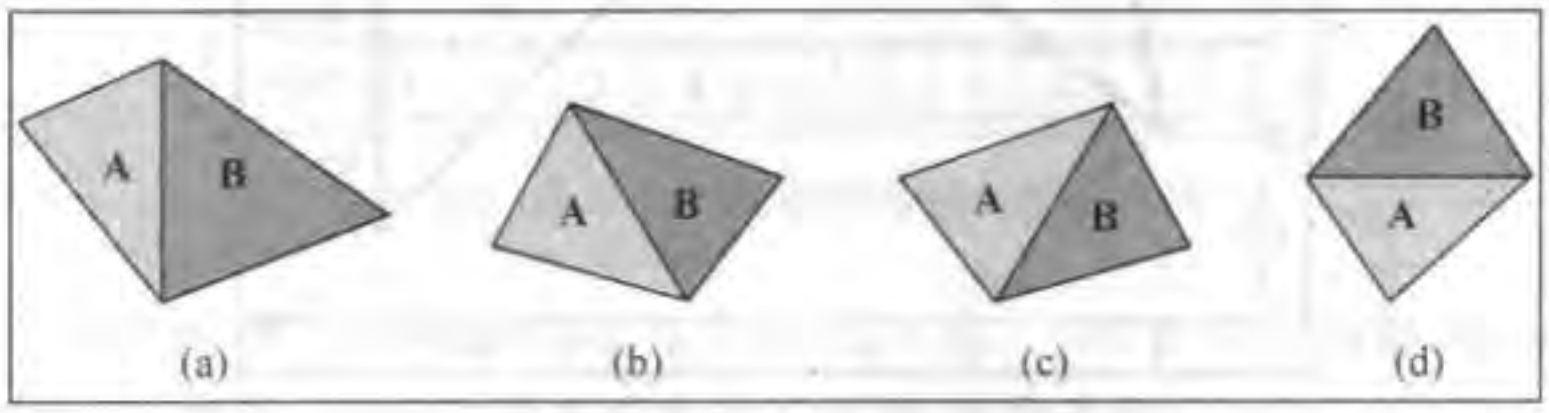


图 9.35 属于多边形 B 的公共边

如何在执行"填充所有交点对之间的像素串"这个命令时应用这条规则呢?图 9.36 给出了一个有 9 个顶点的多边形 P 的例子。中心在 P 内部的像素已经着色,但是哪些中心在边界上的像素应该着色呢?由于左侧和下面的边属于 P,所以这些边上的像素要着色,而那些位于右侧和上面边上的像素不着色。

因此在沿着扫描线填充两个交点之间这部分的过程中,如果最左边的像素位于边界上,那么填充的部分应包括这个像素,而不包括最右边的像素。这样就不会影响那些属于多边形 P 右边或上边的其他多边形的任何像素(参看图 9.36)。

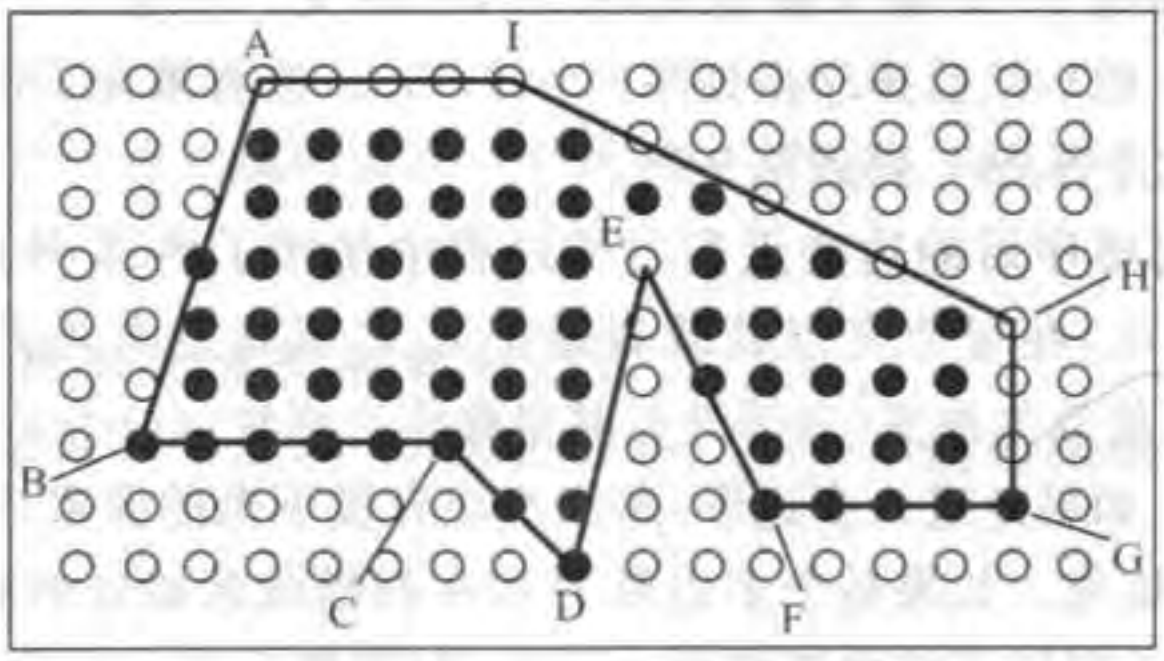


图 9.36 填充内部像素之间的部分

由于扫描线与边界交点一般在两个整数之间,所以要对其上取整或下取整,以得到填充部分正确的起点和终点。例如,假设一个像素串的左交点为实数值 xLeft,右交点为 xRight。我们要分别计算这个像素串中的起始像素 xFirst 和终止像素 xLast。根据前述规则,xFirst 是大于等于 xLeft 的最小整数。类似地,xLast 是严格小于 xRight 的最大整数。

处理扫描线相交于边界端点的情况

经常会出现一条扫描线直接穿过一条边的端点的情况(因为边的端点为整数)。为了获得正确的奇偶性,需要把这类穿越当作相交来处理。

例如在图 9.36 中，穿过顶点 H 的扫描线“看到了”两个交点（一个是与边 GH 的交点，一个是与 HI 的交点），所以 H 两侧的奇偶性看起来是一样的。但是这样会使我们将 H 的右侧填充，这显然是错的。另外一种情况，多边形拥有位于下面的水平边线 BC，所以我们希望在 B 处看到一个交点（为什么？）。但多边形不拥有上面的水平边 AI，所以希望在 A 处的交点数为偶数个。有这么多种不同的情况似乎需要相当复杂的规则集合！

然而，有一种简单的规则适用于各种情况：在交点计算中，忽略扫描线与边界的上面端点产生的交点，并完全忽略水平边。这样在图 9.36 中可以看到，正如我们所希望的，顶点 A 和 I 处没有出现交点，在 B 和 H 处各只有一个交点。图 9.37 中显示了扫描线求交时可能会遇到的其他情况，并标出了扫描线通过多边形顶点时“看到”的交点数目。在每种情况下，对于交点右侧的奇偶性都是正确的数值（验证一下）。需要注意的是，这个规则产生了一个不合理的效果：图 9.36 中处于两条边上端点处的 E 点的像素没有画出。

综上，我们对填充算法中“计算 L 与 P 所有边的交点”这一部分进行调整，使其包含以下改进：

- 计算扫描线与 P 的所有边的交点；
- 丢弃与水平边的交点和与任意边上端点的交点。

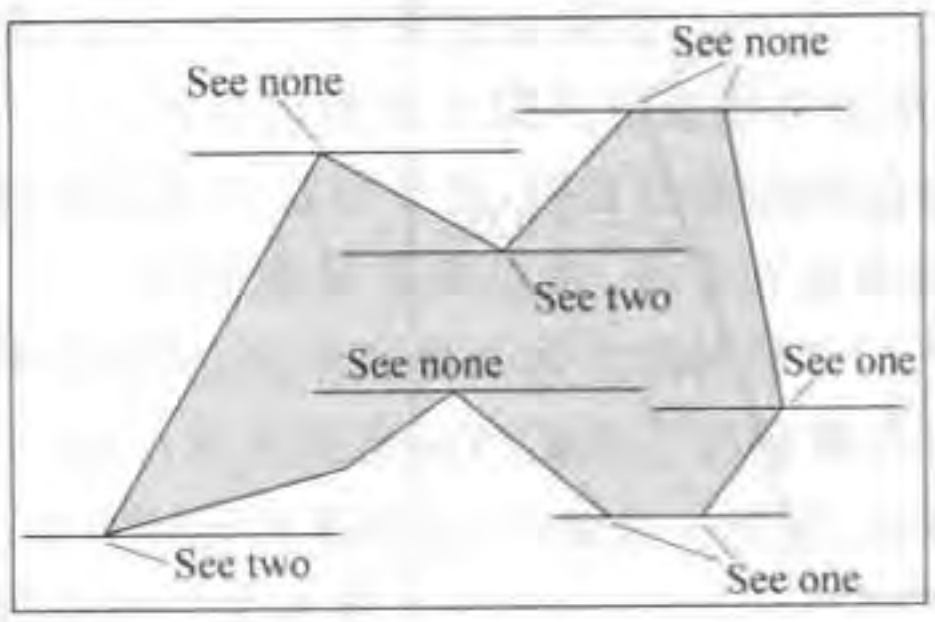


图 9.37　在多边形边界上看到的交点个数

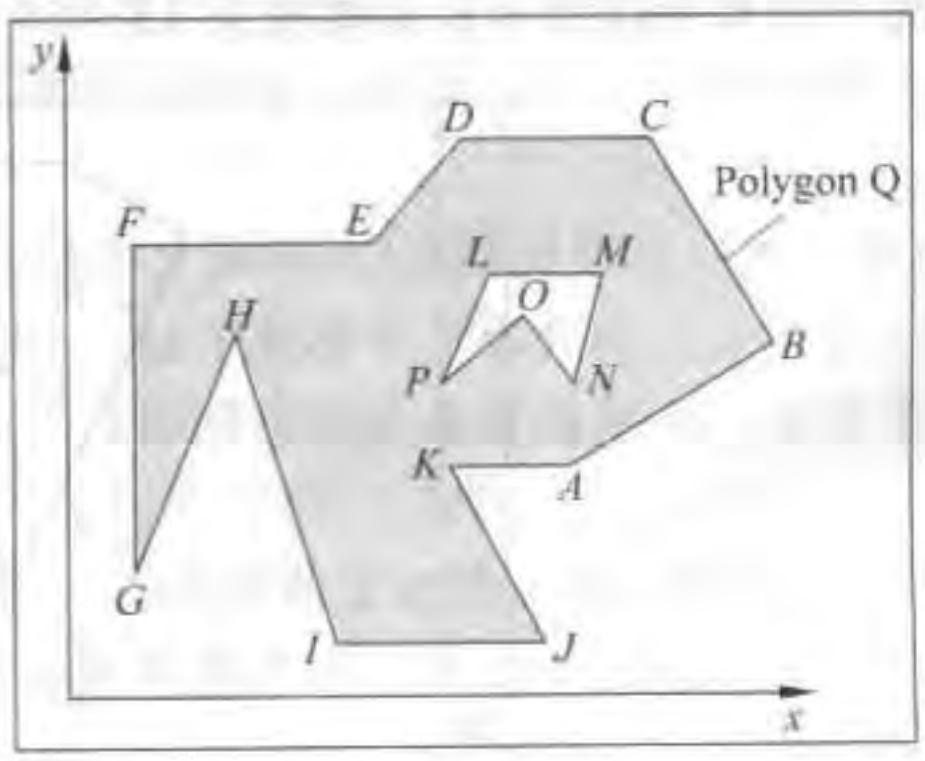


图 9.38　要填充的多边形示例

练习

练习 9.7.1　对方法进行测试

考虑图 9.38 中多边形 Q 的例子，这个多边形有如下 16 个顶点。

A：(52,30)，B：(74,43)，C：(60,60)，D：(38,60)，E：(30,50)，

F：(10,50)，G：(10,28)，H：(22,41)，I：(33,10)，J：(50,10)，

K：(39,30)，L：(40,44)，M：(54,44)，N：(46,34)，O：(45,42)，

P：(33,34)。

最后 5 个点构成了 Q 的一个空洞。画出这个多边形，然后判断前面我们讲述的方法能否正确填充这个多边形。

9.7.2 改进算法性能

要改进前一节中算法的性能,我们需要寻找其中最耗时的部分。这部分就是大量的扫描线与边交点的计算。为了减轻这个负担,可以建立并维护一个简单的列表以便于快速地确定交点。这个列表称为活性边表(Active-Edge List,AEL),有了它算法便能够充分利用边一致性(edge coherence)。边一致性包含两部分内容:

- 与扫描线 y 相交的边中有许多也与扫描线 $y+1$ 相交;
- (直线的一条性质)从一条扫描线到下一条扫描线,交点的 x 值增量相等。

在填充过程中,可以很简单的参考 AEL 确定要在每条扫描线上填充的像素串。AEL 中保存着当前扫描线(正在被填充的扫描线)与所有边的交点的 x 值。x 值按顺序保存,所以根据奇偶性规则,前两个 x 值确定了第一个像素串,接下来的两个值确定第二个像素串,依此类推。

例如,假定正在填充图 9.39 所示的多边形,已经到达扫描线 $y=50$。这条扫描线与多边形的四条边有交点。这些交点的 x 值很容易计算,按顺序分为 45、56.66、70 和 100。根据这个结果可以很容易地填充从 45~56 和从 70~99 这两段区间。

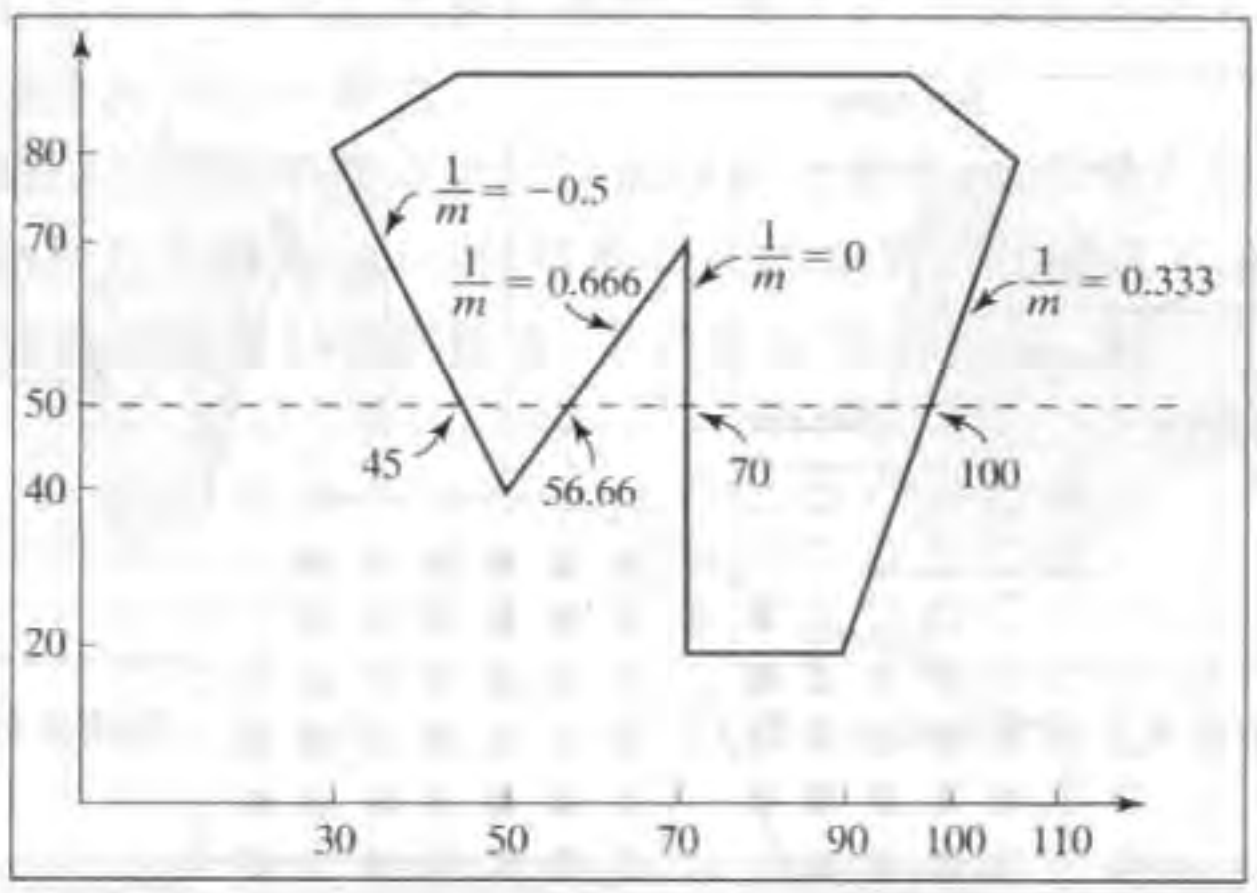


图 9.39 填充多边形示例

图 9.40 中提出了在这种情况下 AEL 的一种链表形式,图中显示了按顺序保存的当前的四个交点以及其他一些辅助数据,这些辅助数据能够使这种算法快速更新 AEL,以便于下一条扫描线使用。

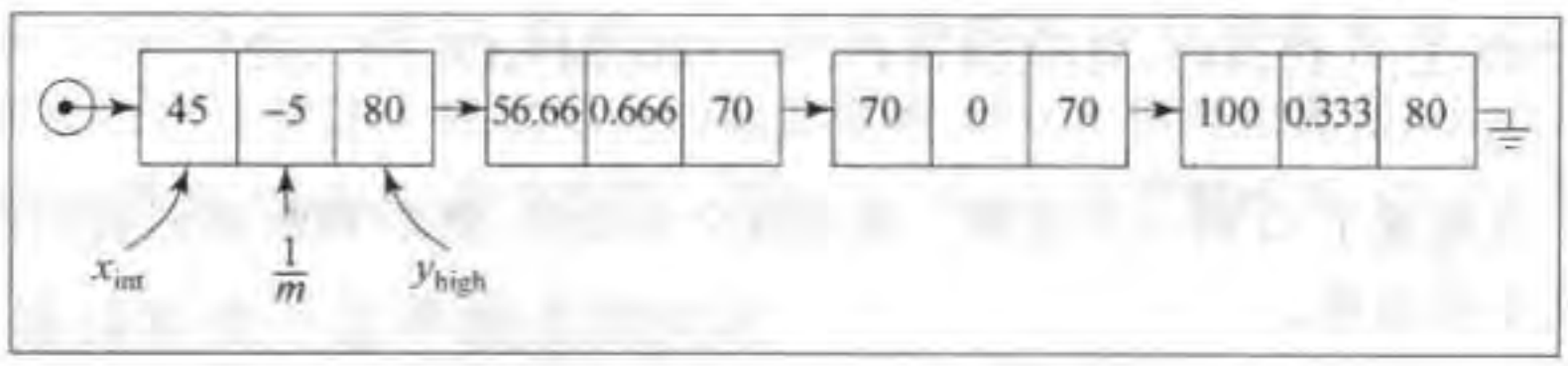


图 9.40 扫描线 y=50 的 *AEL*

确切的说，对于每条与当前扫描线相交的边，AEL 中都包含下面 3 项数据：

(1) 这条边与当前扫描线交点的 x 值 x_{int}。

(2) 这条边斜率 m 的倒数 $1/m$。

(3) 这条边的上端点 y 值 y_{high}。

其中第二项用于确定扫描线 $y=51$ 与各条边交点的位置。需要注意的是，如果一条边的斜率是 m，那么 y 向上移动一个单位将导致交点的 x 值增加 $1/m$（验证这一点）。因此每条边的 $1/m$ 值直接存储在该边的 AEL 中（如图 9.40 所示）。（例如，最左侧的边在 x 方向有20 个单位的跨度，在 y 方向有 40 个单位的跨度。因此其斜率的倒数为 20/40。检验图中给出的其他数据）。于是只需对 x_{int}加上 $1/m$ 就得到了这条边与其上方下一条扫描线的交点，这样就充分利用了边一致性。

需要注意的是，增加 $1/m$ 并舍入正是 Bresenham 算法沿着一条线一步一步移动时所做的事情。实际上，在此处使用 Bresenham 方法（根据某个误差项的符号递增，然后更新这个误差项）计算交点可以使得效率达到最高。

在填充过程中移动到下一条扫描线时，除了沿着一条边递增之外，还可能发生另外几种情况：

(1) 新的扫描线可能刚好超出（在其之上）AEL 表示的一条边。AEL 中保存着每条边的上端点 y 值，这使得这种情况很容易确认。如果新的扫描线 y 的值超出了一条边的上端点值，就从 AEL 中删去这条边。

(2) 如果扫描线 y 值与多边形中的某些边的下端点 y 值相等，这就说明它遇到了新边。此时通过查找另一个表，将这些新边的信息加入 AEL。这个表将在下面介绍。

(3) 对于非简单多边形，如果两边相交，那么边交点的顺序可能会发生颠倒。如果发生这样的情况，记录列表必须重新排序。

所以，填充完当前扫描线上的像素串之后，递增 y，同时更新交点的 x 值，必要的话还要从 AEL 中删除某些边，添加另一些边，并将交点按 x 值重新排序。下一条扫描线也可以按照同样的方式处理。

边表

处理下一条扫描线时，为了确定要添加哪些边，一种方法是检测 P 的每个顶点，以确定当 y 增加时遇到了哪些边。但是这种方法效率不高。我们可以提前获取 P 中每条边的信息，然后将其巧妙的放置在一个边表（Edge Table，ET）中。在更新 AEL 时，利用 ET 可以快速获取所需的信息。

边表由一个链表的数组 edgetable[]构成。对应每一条扫描线有一个链表。图 9.41 给出了图 9.38 中多边形 Q 的 ET 中的一部分。边表可以在最初遍历多边形以去掉平行边和处理其他边时构造。

每条扫描线的链表中包含 Q 中所有其下端点在这条扫描线上的边的信息（这种方法有效地根据其下端点的 y 值对 Q 的所有边进行了排序；有一种说法是 ET 为多边形的边执行了一次桶排序（bucket sort））。可以看到，实际上 ET 中的许多表都是空的。

每条边的表示方法与 AEL 中边定义的类型相同，并写入正确的 y_{high}和 $1/m$ 字段，x_{int}字段的值为该边底端点的 x 值。

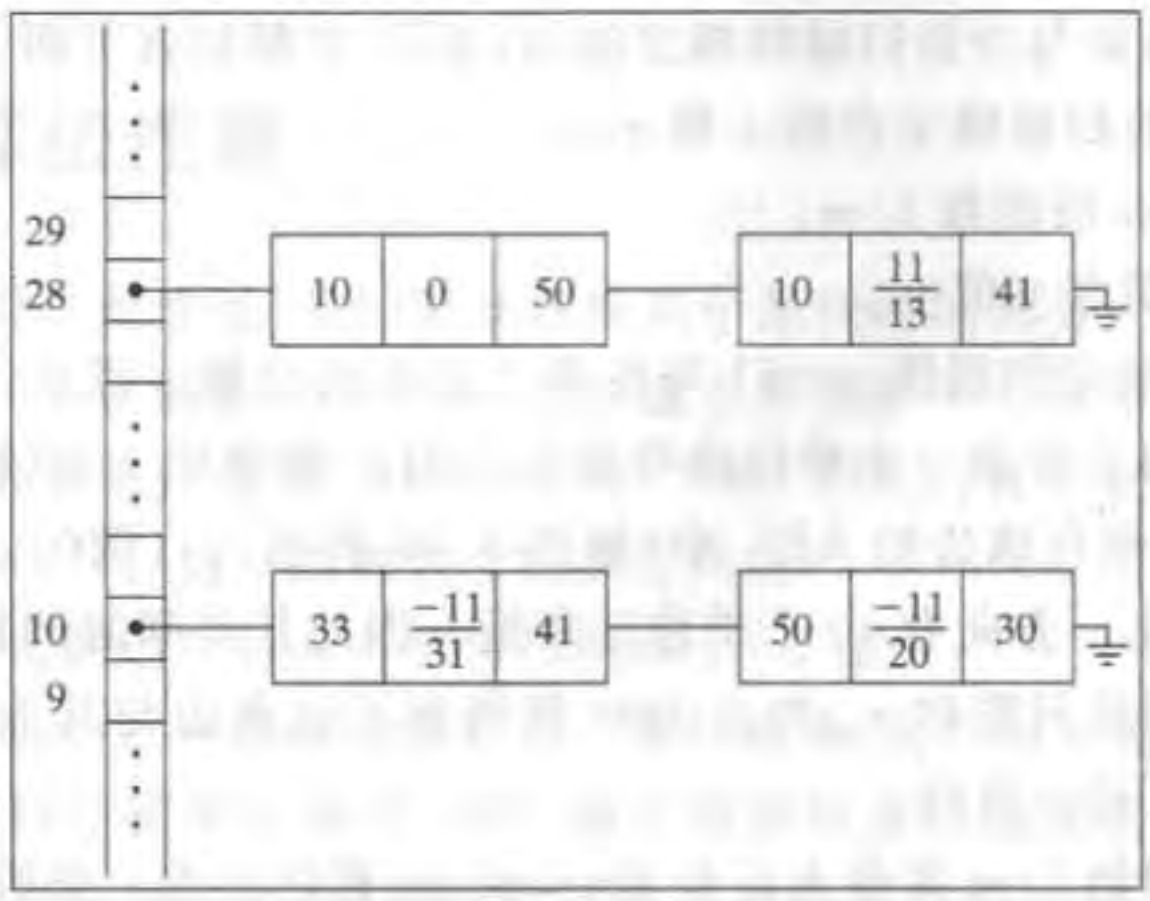


图 9.41 表示多边形 Q 的边表

更新 AEL 时,利用边表可以很容易确定在 AEL 中要添加哪些边。对于位于第 y 行的新的当前扫描线,edgetable[y]保存的边都需要添加到 AEL 中。x_{int}值自动包含最初的交点 x 值,其他字段也被正确地加载。因此要在 AEL 中添加新记录,只需调整少量几个指针即可。在填充像素串之前,对 AEL 按照 x_{int}的升序重新排列,以维护正确的奇偶性。

图 9.42 给出了整个算法的框架。AEL 开始为空,然后算法从 $y=0$ 开始依次填充每条扫描线。开始 y 值随着主循环快速递增,直到在 AEL 中添加第一批边,在这之前都不进行绘制。案例分析 9.7 给出了这个算法的更多细节。

```
AEL=NULL;
for(y=0; y<=maxRow; y+1)
{  // APE 最初为空
    <将 edgetable[y]中的所有边加到 AEL 中>
    if( AEL!=NULL)                          //判断是否有边处理?
    {
      根据 xInt 值对 AEL 排序
      使用 AEL 信息沿着 y 填充像素值
      将 AEL 中满足 yupper==y 的记录删除
      用斜率的倒数更新 xInt 的值
    }
}
```

图 9.42 多边形填充算法框架

Lui 等人【Lui00】最近提出了一个非常高效的多边形填充算法,该算法使用一个基于"三元组"的数据结构来描述多边形的每条线段(三元组由一个 y 值、一个线段左侧端点的 x 值和一个线段右侧端点的 x 值构成)。

练习

练习 9.7.2 表填充算法

表填充算法能够快速填充某一类多边形。首先,对多边形上的每条边进行扫描转换(可

以使用 Bresenham 算法)，生成沿着每条边的(x,y)对。然后将其与两个数组 min[y]和 max[y]作比较测试，这两个数组中保存对于每个 y 值目前所遇到的最小 x 值和最大 x 值。如果 $x<$min[y]，就把 min[y]更新为该 x 值，max[y]的处理方法类似。最后对于每条扫描线 y，通过绘制从 min[y]到 max[y]的像素串来填充这个多边形。由于不需要排序，所以这个算法的速度很快。

请问多边形要满足什么几何条件才能确保使用该算法正确填充？针对该算法，给出几个能够正确填充的多边形的例子，再给出几个不能正确填充的多边形的例子。

练习 9.7.3　栅栏填充算法

对多边形的栅栏填充(fence-fill)方法利用了对图像中像素求"补"或求"反"这个概念。对于一个二值图像来说，求补就是把值为 1(白色)的像素设置为 0(黑色)，反之亦然。求补的一个基本性质是求补两次就恢复原来的值。

该填充算法的流程是这样的：建立一个垂直的"栅栏"——它可能通过要填充的多边形的某个顶点。把所有像素设置为 0。然后对于多边形的每条边，在通过这条边的每条扫描线上，对从该边到栅栏之间的所有像素求补。一条扫描线上位于多边形之外的所有像素会被求补偶数次，因此值为 0。那些位于多边形内部的像素被求补了奇数次，所以值为 1(要被填充)。这个算法与栅栏的位置无关。

在纸上画出几个多边形，手算使用栅栏填充方法填充这些多边形。证明这种方法对于多个多边形和有空洞的多边形也适用。如果将这种方法应用于一条线，而不是有着真实内部和外部的多边形，会产生什么效果？

9.8　走样和反走样技术

前面的讨论中多次出现"锯齿"这个词，它是"走样"(aliasing)的一种形式。而走样是光栅显示的一种固有性质。产生走样现象的原因是像素本质上的离散性，也就是说，像素是以某一固定尺寸的矩形阵列的形式出现在显示器上的。图 9.43 显示了字符'a'的走样图像。

以图 9.44(a)所示的黑色矩形为例，如果这个矩形覆盖了大量的像素，它的边界尽管还是有锯齿，但是看起来会相对平滑一些。但是如果这个矩形只覆盖了少数几个像素，如图 9.44(b)所示，那么"锯齿"就会非常明显，以至于能够扰乱视线。在本图中，像素设置为黑色的依据是该矩形是否覆盖了一个特殊的点，即像素的中点。使用像素的中点对矩形进行采样，看它是否在矩形上，这是一种非常有效的方法。以这种采样为依据，整个像素区域的颜色可以设置为黑色或白色两种颜色。

图 9.43　字符'a'的走样图像

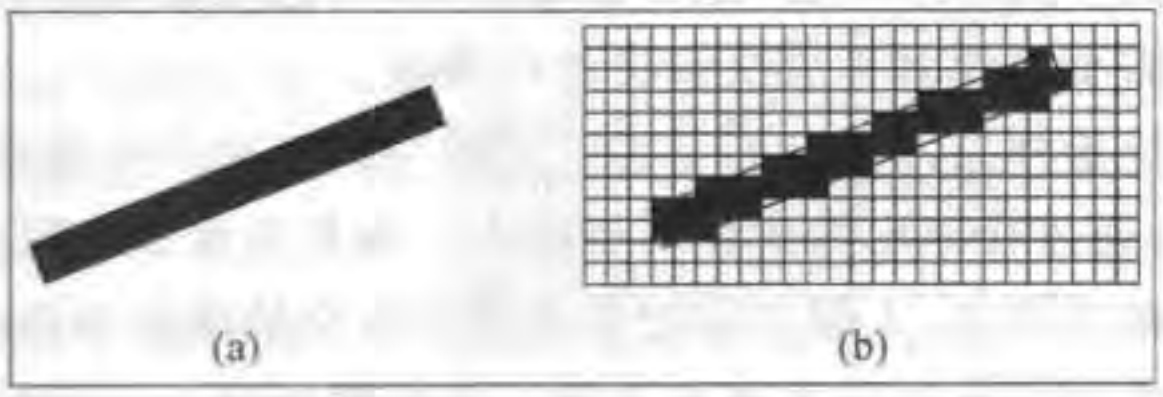


图 9.44　一个矩形的走样现象

(a) 图形覆盖了较多像素；(b) 图形覆盖像素相对较少

但这种采样策略可能会导致小物体(比如 3D 场景中远处的物体)完全消失(见图 9.45(a))。如果这个物体处于像素的中心之间,就完全不会被显示。图 9.45(b)演示了动画中这样的物体如何会一会儿出现、一会儿消失地闪烁。该物体可能在某一帧中覆盖了一个像素的中心,而到了下一帧就不覆盖了,这样产生的结果是无法让人接受的。

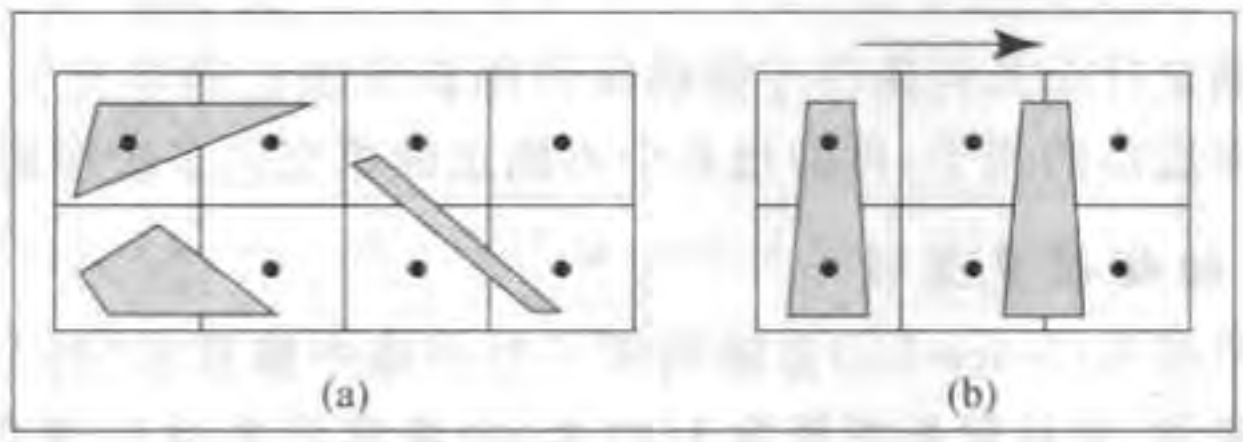


图 9.45 小物体由于走样而消失

粗略地说,如果对一个快速变化的信号采样频率过低,所得样本表示的会是低频变化的信号:原始信号的频率看起来被较低的"走样"频率所代替。图 9.46(a)显示了一个快速变化的方波,以图中黑点标出的等间距采样。如果只从这些采样结果看(图 9.46(b)),这个信号看上去是它的走样结果:是一个频率较低的方波。

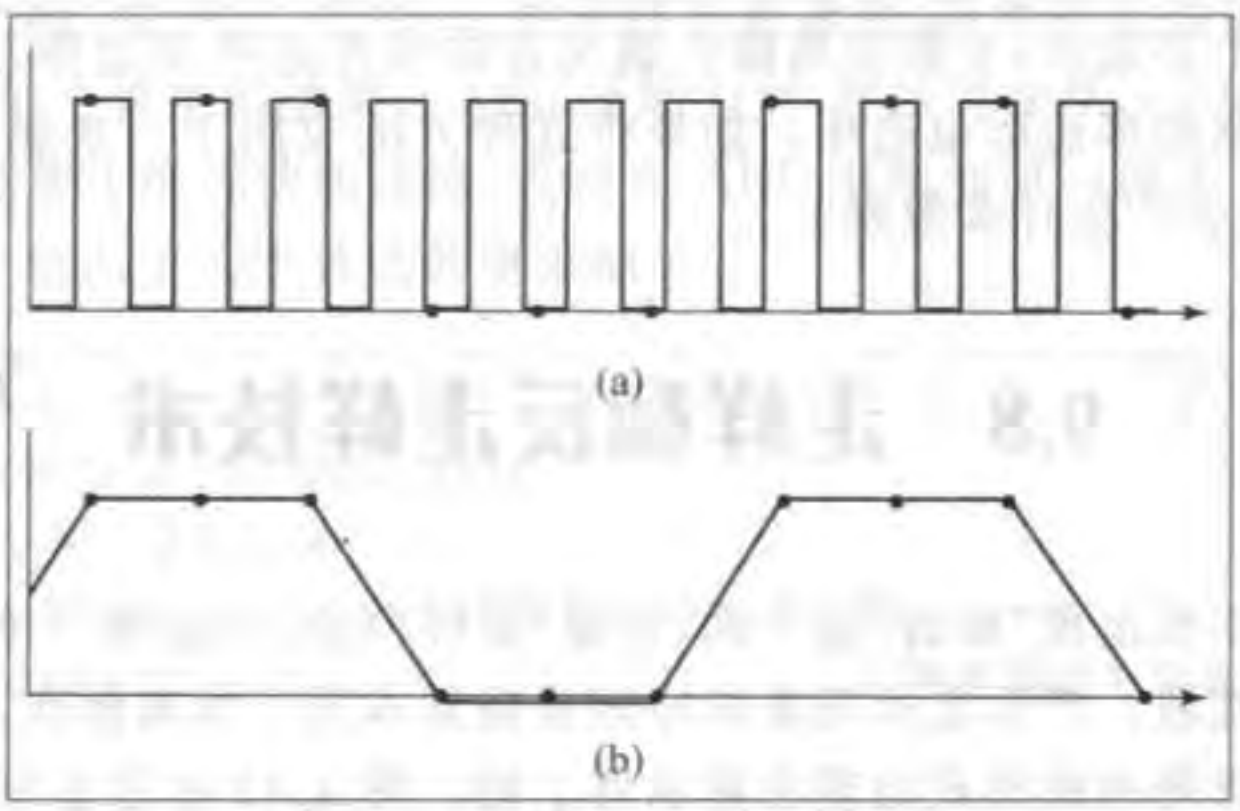


图 9.46 采样频率过低使得信号看上去走样

9.8.1 反走样技术

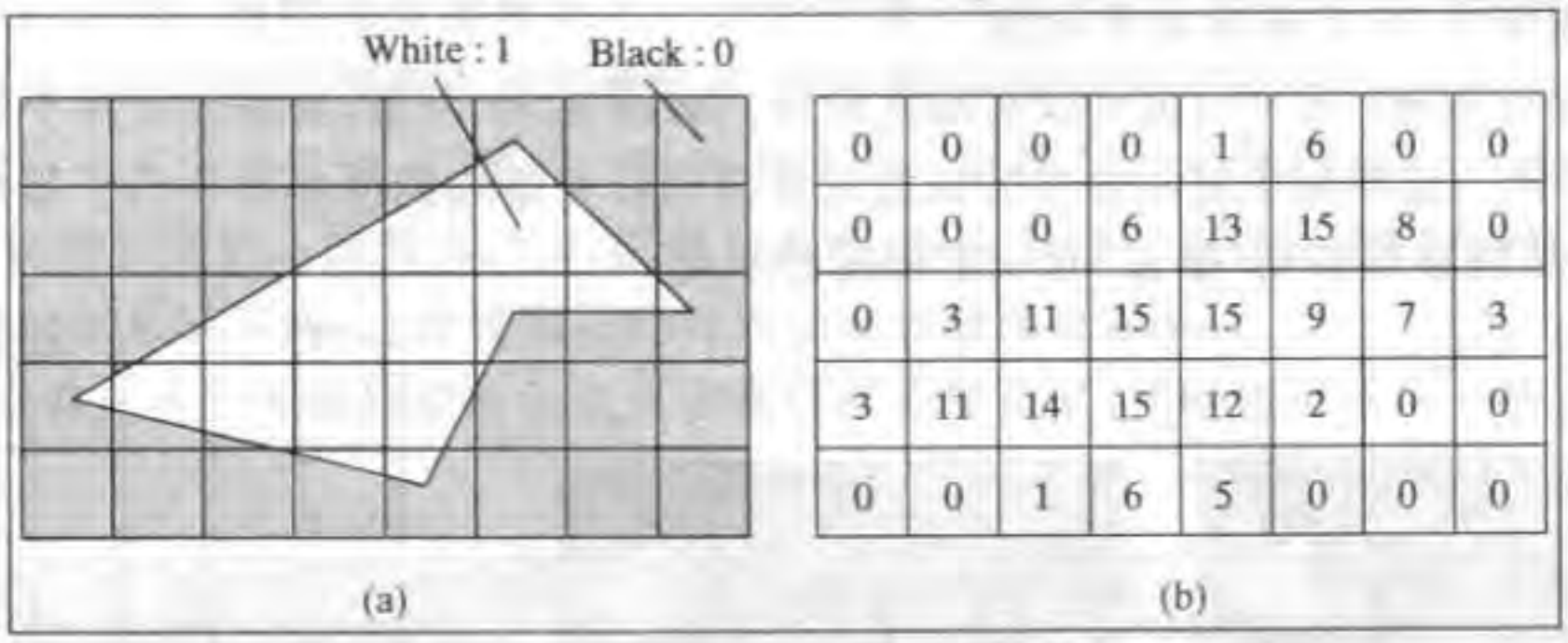
如何降低由于采样不足而产生的走样现象呢?采用分辨率更高的显示设备,配以更好的绘制算法,对解决问题有所帮助,因为这可使锯齿相对物体会更小一些,但仍存在。所以我们需要寻找其他方法解决走样现象。

反走样技术涉及到某种形式的"模糊"来产生更平滑的图像。对于在白色背景中的黑色矩形,通过在矩形的边界附近掺入一些灰色像素,可以柔化从黑到白的尖锐变化。从远处观察这幅图像时,人眼能够把这些缓和变化的暗影融合在一起,从而看到了更加平滑的边界。

常用的反走样方法有 3 种:即前置滤波(prefiltering)、过采样(supersampling)和后置滤波(postfiltering)。

前置滤波

前置滤波技术根据物体的覆盖率(coverage)计算像素的颜色。覆盖率是指某个像素区域被物体覆盖的比例。考虑图 9.47(a)所示的在黑色背景上的白色多边形的扫描转换结果。假定亮度值 0 表示黑色,1 表示白色。把这个多边形放在方格线中,其中每个正方形的中心对应显示器上一个像素的中心。被多边形覆盖了一半的像素的亮度值赋为 1/2;被覆盖了三分之一的像素赋值为 1/3;依此类推。如果帧缓冲区的每个像素有 4 个比特位,那么 0 表示黑色,15 表示白色。一个像素如果被多边形覆盖了四分之一,它应该赋值为(1/4)15,舍入之后结果为 4。图 9.47(b)给出了根据每个像素的覆盖率计算得到的值。(如果只使用像素中心对多边形采样,当多边形覆盖像素中心时取值 15,否则为 0,则该像素阵列的值会是什么?)



0	0	0	0	1	6	0	0
0	0	0	6	13	15	8	0
0	3	11	15	15	9	7	3
3	11	14	15	12	2	0	0
0	0	1	6	5	0	0	0

图 9.47　利用像素区域被覆盖的比率

当然,求出每个像素覆盖率的几何计算可能非常耗时。但是现在已经有很多有效的方法被研究出来,例如 Pitteway 和 Watkinson 提出的方法【Pitteway80】,还有最近 Xiaolin Wu 提出的算法【Wu91】。这些算法都采用增量的思路计算每个像素的覆盖率,并且都使用整数计算。

综上所述,前置滤波对扫描转换后的各种物体的详细几何形状进行操作,并根据这个物体对每个像素区域的覆盖率计算每个像素的平均亮度。对于非多边形的形状,前置滤波的计算代价可能非常大,所以我们需要寻找其他反走样的方法。

过采样

既然走样是由于对物体采样点数过少,那么我们可以尝试对每个像素多点采样来降低走样现象。这种技术称为过采样(supersampling),即对场景进行超过显示像素数目的亮度采样,然后取几个样本的平均值来形成要显示的像素值。

图 9.48 给出了一个双倍采样的例子。该物体(这里是一个倾斜的条状物)在 x 方向和 y 方向的采样密度都是显示像素密度的 2 倍。图中的正方形表示显示出来的像素,'x'表示场景的采样位置。

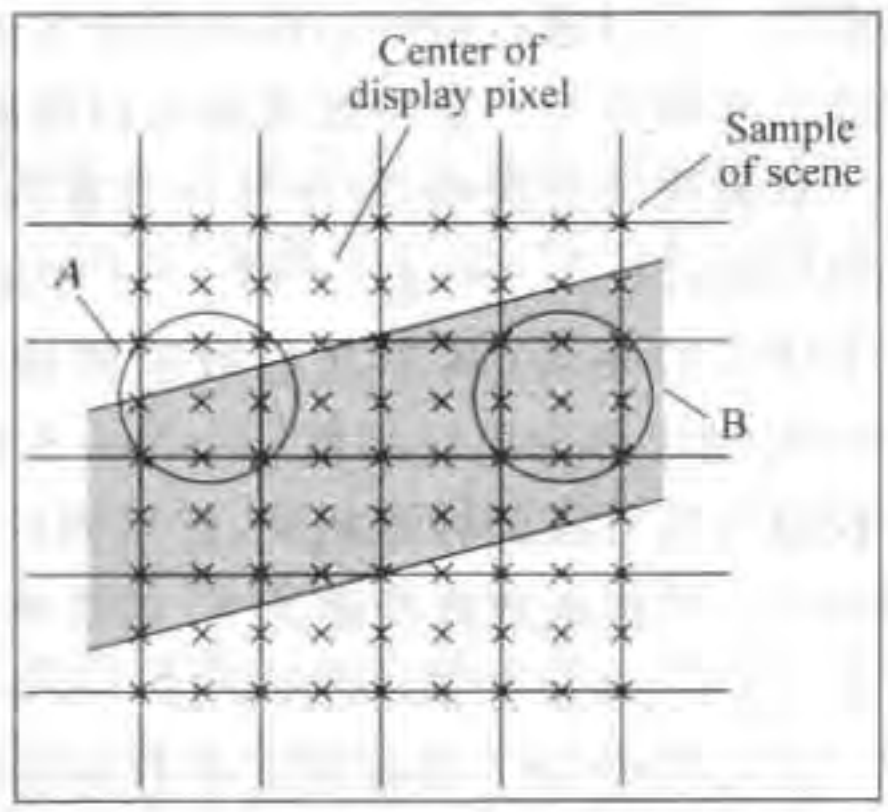


图 9.48　采用过采样进行反走样

每个最终显示的像素值可以通过 9 个"相邻"样本点的平均值构成,即该显示像素的中心位置的样本点以及围绕它的 8 个样本点。可以看到,有些样本点是被几个像素重复利用了。(哪些样本点?)中心在 A 的显示像素的数值基于条状物内部的 6 个样本和背景的 3 个样本。因此它的颜色设置为条状物颜色的 2/3 加上背景颜色的 1/3。处在 B 点的显示像素数值基于 9 个条状物内部的样本,因此它的颜色设置为条状物的颜色。

图 9.49 左图显示了一个分辨率为 300×400 像素的场景,场景的"锯齿"很明显,特别是在物体的侧面处。图 9.49 右图显示了双倍采样得到的结果。对于同样的场景,以 600×800 的分辨率采样,300×400 显示像素中的每个像素取过采样数据中相应的 9 个相邻样本点的均值。虽然有一定程度的模糊,但可以看到"锯齿"被明显地柔化了。

一般来讲,过采样方法对于每个显示像素在 x 方向和 y 方向各计算 N_s 个样本,然后取一定数量的相邻样本的平均值作为每个显示像素的值。例如,对于 $N_s=4$ 的过采样,每个显示像素需要求 16 个样本的平均值。

不使用过采样($N_s=1$)也可以实现反走样。如图 9.50 所示,对场景在每个显示像素的 4 个角点采样。这样每个显示像素的亮度设置为在其 4 角取到样本的均值。虽然没有采用过采样,仍然能够观察到"锯齿"有一定程度的柔化。

图 9.49 使用两个不同的采样数绘制的物体
左侧为未进行反走样操作效果;右侧为双倍采样效果

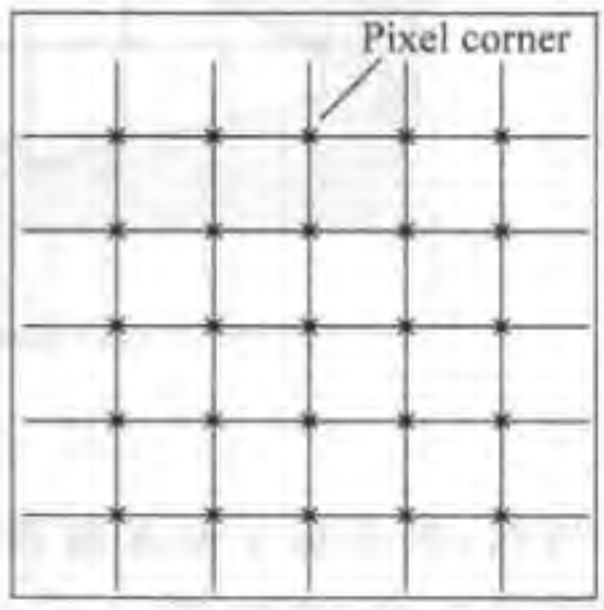


图 9.50 采用角采样的反走样方法

后置滤波

双倍采样方法取 9 个相邻样本的平均值作为显示像素的亮度值,每个样本具有同样的重要性。可以通过给中心样本以更大的权重,给 8 个相邻样本较小的权重来改善这种方法造成的模糊效果。这种改变还可以使更多的相邻样本加入到均值计算中来。

后置滤波方法中,每个显示像素的数值是场景中一个适当规模的相邻样本集合的加权平均(weighted average)。图 9.51 中给出了双倍采样时的情况。图中每个数值表示场景中一个样本的亮度,其中灰色数值的位置表示显示像素的中心。表示权重的正方形模板(mask)也可称为窗口函数(window function)依次覆盖每个灰色像素。然后把窗口中的每个权值与其对应的样本相乘,把得到的 9 个乘积相加就是这个显示像素的亮度值。例如,当图中所示的模板放在亮度为 30 的样本上时,加权平均的结果为:

$$(30)/2+(28+16+4+42+17+53+60+62)/16=32.625$$

舍入后为 33。该模板将中心的权重设为其 8 个相邻像素权重的 8 倍,而权重之和总是为 1。

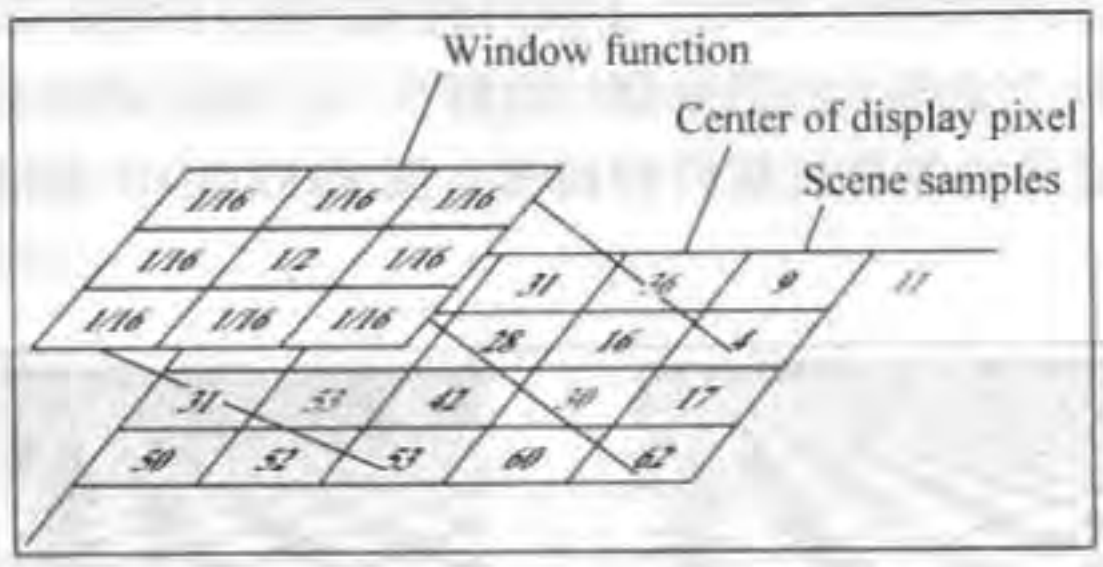


图 9.51　对以副图形进行后置滤波处理

值得注意的是，前面介绍的过采样仅仅是后置滤波的一种特例，只不过所有的权值都是 1/9。从信号处理领域引入的采样和滤波理论为研究作为后置滤波器的各类不同的窗口函数提供了分析方法。有时还会使用更大一些的模板，比如 5×5，甚至 7×7 的模板，这些模板考虑了距离中心样本范围更大的邻居样本，能够提供更好的平滑效果。

后置滤波法可以在任意 N_s 值的过采样中使用。如果 $N_s=4$，使用 5×5、7×7 甚至 9×9 模板都是比较合适的。如果 $N_s=1$，就像角采样，可以采用中心权值比较大的 3×3 模板。根据所渲染的场景的不同，这种模糊可能有效果也可能没有效果。

第 12 章将讨论与光线跟踪有关的更多高级反走样技术，同时也可以从一些联机资料获得相关信息。

练习

练习 9.8.1　过采样

对于 $N_s=1$ 的情况，画出一个与图 9.50 类似的像素栅格图，在每个显示像素中心位置防止'X'，在每个像素的四角放置's'表示场景的采样点。

练习 9.8.2　多边形反走样

设帧缓冲区像素的取值 0～15。对一个顶点为(1,3)、(6,7)、(15,4)、(11,15)和(1,8)的白色多边形的每个边进行扫描变换同时进行前置滤波，计算结果像素的值。假定背景为黑色，使用边界填充算法用白色填充这个多边形。如果多边形的边上有一个空洞，那么这个填充算法是否可能失败？如果可能，应该如何弥补这个缺陷？

练习 9.8.3　角采样

对于一个有 R 行 C 列的光栅显示器，使用角采样时，一个场景要计算多少个样本？也就是说，这个光栅显示器有多少个"角"？把这种采样方法与"像素中心"采样法和双倍采样法进行对比。

9.8.2　纹理的反走样

第 8 章介绍了如何把纹理映射到各种表面上，以获得更强的真实感效果。因为通常总希望纹理能够真实地反映某种图案和图像，所以映射的纹理对走样效应更加敏感。另外，纹理本身通常也是像素映射的形式，而显示出来的像素和像素映射中的像素通常有着复杂的

几何关系。我们将介绍几个绘制纹理时会出现的特殊问题,并详细讲述几种解决方案。

图 9.52(a)给出了一个走样的经典示例:绘制在一个倾斜表面上的纹理。远处的棋盘方格已经严重地走样,呈现出各种波纹状的图案。图 9.52(b)中的图像是经过反走样处理的,结果有本质的改善。

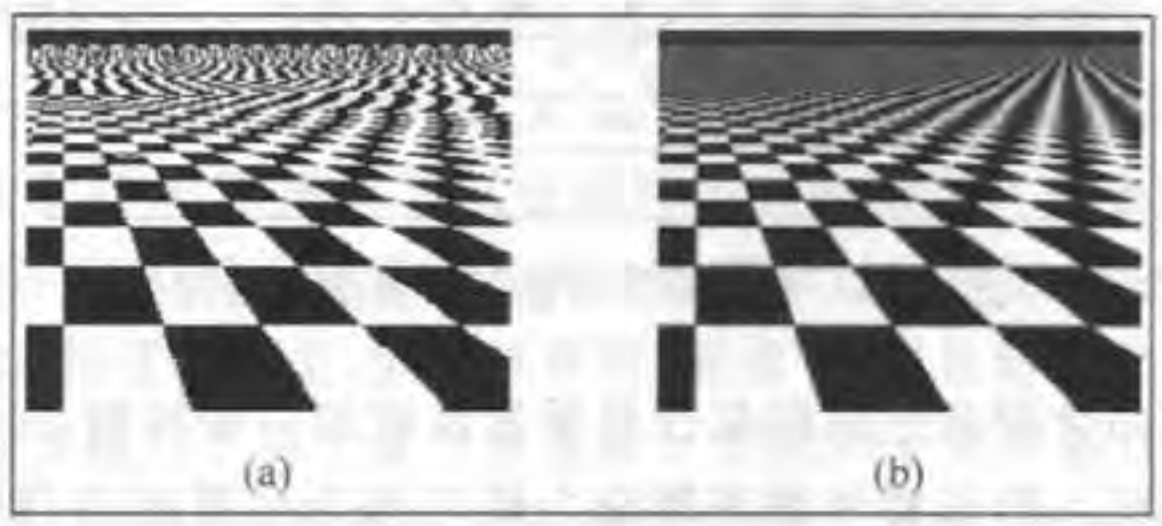


图 9.52 走样和用反走样技术改进的示例

回顾一下以前的内容,纹理定义为纹理空间中的一个纹理函数 texture(s,t),在最终绘制到显示器之前会进行一系列复杂的映射。而渲染任务却是反方向进行工作,对每个给定的显示像素坐标(x,y),计算它在纹理函数 texture()中对应的颜色。

图 9.53 显示了一个正被渲染的特定像素(x,y)和纹理空间中对应的纹理坐标(s^*,t^*)。为方便起见,我们用 $T()$表示从像素空间到纹理空间的所有映射,于是有$(s^*,t^*)=T(x,y)$。(这个映射的各个分量已经在第 8 章中介绍)。

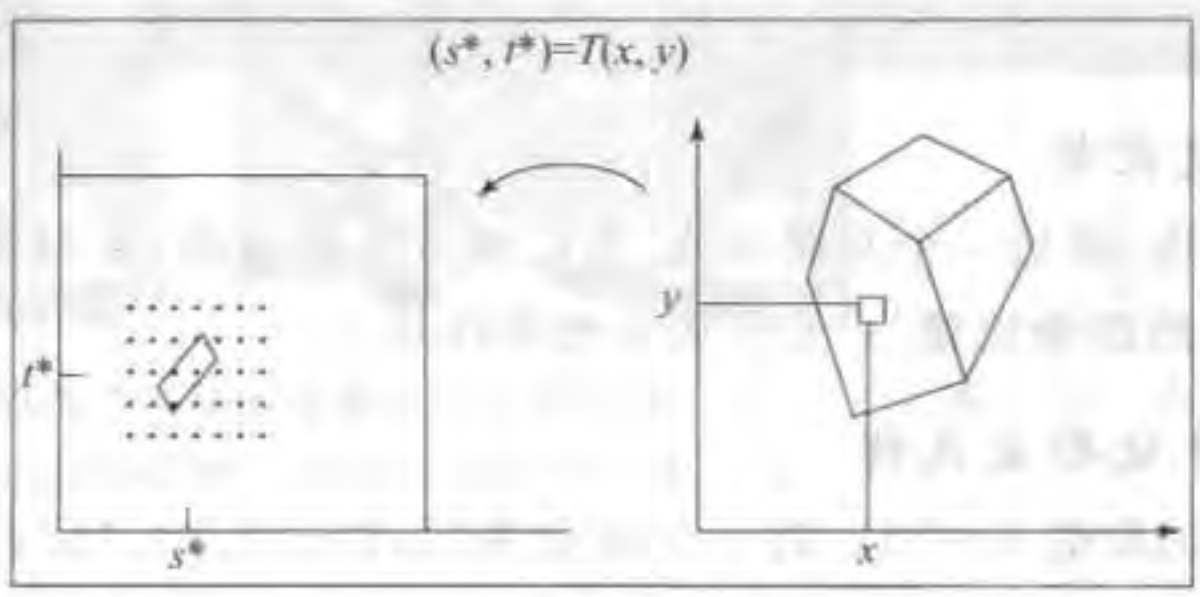


图 9.53 纹理渲染过程走样的原因

由于像素不是一个点而是有一定面积的,因此我们必须认真考虑怎样把以(x,y)为中心的方形像素映射到纹理空间。图 9.53 中也以四边形的形式画出了这个像素的映射[①]。后面我们把这样的四边形称为屏幕像素的纹理四边形。想像纹理空间由这样的纹理四边形覆盖,每个纹理四边形都来自于一个屏幕像素。纹理四边形的大小和形状取决于 $T()$的性质,要计算起来代价很高。如果纹理函数 texture(,)在纹理四边形内部有变化,而屏幕像素却只用一个采样结果 texture(s^*,t^*)着色,那么就会丢失重要信息,出现走样现象。

为了减小走样效果,我们为每个屏幕像素着色时,应使用位于对应的纹理四边形内部的若干颜色均值作为该像素的颜色值。图 9.54 显示了叠加在定义 texture(s,t)的像素映射 txtr[][]中各个纹理元素(texel)之上的一个纹理四边形。如果有一种简单的方法可以计算

① 如果显示的 3D 表面是曲面,那么四边形的边也是弯曲的。

每个纹理元素位于这个纹理四边形之内的面积，就很容易求出平均值：按照面积对每个 txtr[r][e]计算加权求和，然后除以这个四边形的面积即可。这个过程类似于前面介绍的用于反走样的前置滤波技术。不幸的是，这样的计算严重影响性能，因为很难计算出每个纹理元素被四边形覆盖的面积。

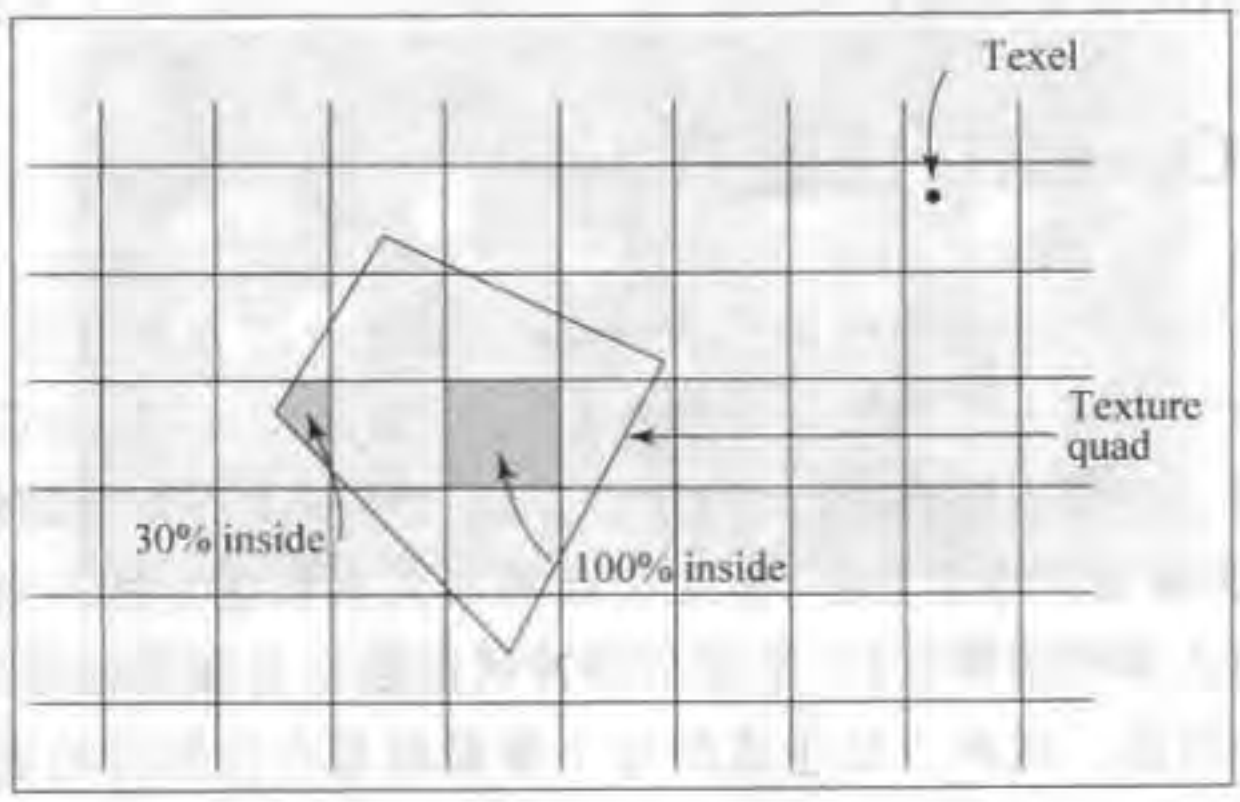


图 9.54　计算用面积加权平均的颜色

Heckbert 调研了研究人员提出的各种近似前置滤波的方法【heckbert86】，发现一种有效的方法——椭圆加权平均(EWA)滤波。如图 9.55 中表明，这种方法把每个屏幕像素想像成被一个圆对称的滤波器函数覆盖。图中的同心圆表示不同的权重等级(就像地形图)，并把这个滤波器函数映射到纹理空间中。在纹理空间中，这些权重等级变成一系列的椭圆，并能够大致表示纹理四边形的形状。我们将滤波函数采样，并保存在查找表(LUT)中，记录椭圆内部的不同位置的权值，以此对椭圆内的纹理元素进行加权平均。这个过程可以完全通过增量方法实现，对每个像素只需要少量的计算，所以效率很高(利用了滤波器函数从一个空间到另一个空间的可控变换)。图 9.52(b)就是用 EWA 滤波器渲染的向后倾斜的棋盘图案。

另一种方法称为随机采样，在【Watt&Watt92】中有详细介绍。该方法避免了困难的几何计算，通过在纹理元素中随机采样求均值来计算纹理颜色。图 9.56 显示了与某个屏幕坐标相对应的以(s^*,t^*)为中心的纹理四边形。我们并不是用这个纹理四边形本身，而是在(s^*,t^*)周围的纹理元素中随机采集一定数量的样本，最终纹理颜色就是这一些采集的样本点颜色的均值：

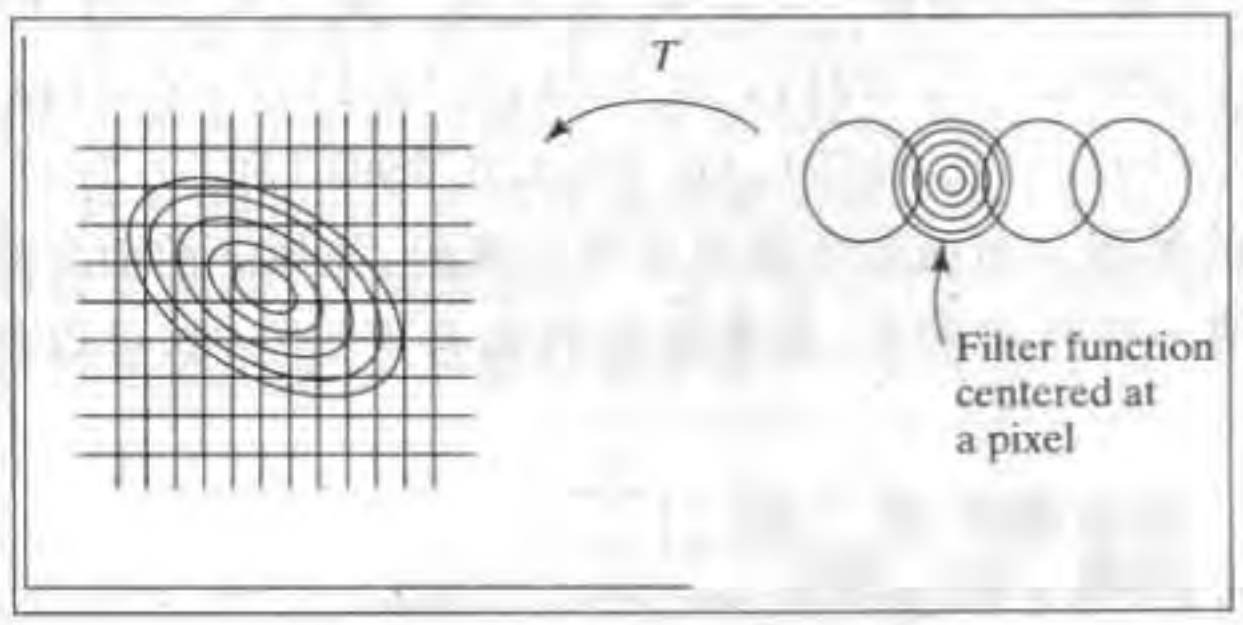


图 9.55　椭圆加权面积滤波

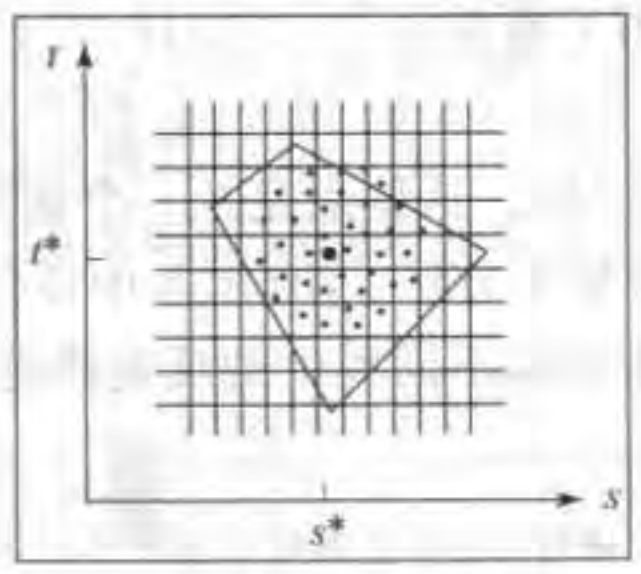


图 9.56　采用随机采样方法的反走样

$$\text{average} = \frac{1}{N}\sum_{k}\text{texture}(s^* + \alpha_k, t^* + \beta_k) \tag{9.11}$$

其中 α_k 和 β_k 为小的随机量,可以很容易的使用随机数生成器生成。它们的分布可以根据纹理四边形的大致尺寸进行调整。

在第 12 章中我们还会看到随机采样作为一个强大的工具应用于光线跟踪。

9.8.3 使用 OpenGL 反走样

OpenGL 提供了一些执行反走样操作的工具。最简单的方法是采用一个累积缓冲区(accumulation buffer),这是一个与帧缓冲区类似的附加存储区,OpenGL 可以生成累积缓冲区并对其进行绘制。类似于随机采样法,这种方法在略微不同视点的位置(相差只有一个像素的几分之一)多次绘制同一个场景,然后把结果加入累积缓冲区。当把所有这些位置经过扰动的绘制结果加入累积缓冲区后,把累积缓冲区的数据复制到帧缓冲区中,显示结果就是经过反走样处理的图像。这种方法生成的每个像素都是在投影后的场景中该像素邻域颜色的均值。

下面的代码显示了当使用摄像机对 3D 场景中拍照时反走样是如何实现的。首先在程序运行时创建累积缓冲区[①],并将其初始化为全 0(使用 glClear(GL_ACCUM_BUFFER_BIT))。然后把这个场景绘制 8 次,每次绘制之前把摄像机根据存储在数组 jitter[]中的位移量,在 x 方向和 y 方向上"平移"一个微小的距离(参见第 5 章和第 7 章)。使用 glAccum(GL_ACCUM, 1/8, 0)语句,把每次新绘制结果的数值乘以 1/8 后逐个像素地加到累积缓冲区中。绘制完第 8 次之后,使用 glAccum(GL_RETURN, 1.0)把累积缓冲区复制到帧缓冲区中。下面是完整的代码:

```
glClear(GL_ACCUM_BUFFER_BIT);                          //清空累积缓冲区
for (int i=0; i <8; i++)
{
    cam.slide(f * jitter[i].x, f * jitter[i].y, 0);   //平移摄像机
    display( );                                        //绘制场景
    glAccum(GL_ACCUM, 1/8, 0);                         //加入到累加缓冲区
}
glAccum(GL_RETURN, 1.0);                               //把累积缓冲区数据复制到帧缓冲区
```

jitter 向量中包含 8 个点,这些点的 x 值和 y 值在 $-0.5 \sim 0.5$ 之间。头文件 jitter.h[②]中使用了数值是(−0.3348,0.4353),(0.2864,−0.3934),(0.4594,0.1415),(−0.4144,−0.1928),(−0.1837,0.0821),(−0.0792,−0.3173),(0.1022,0.2991)和(0.1642,−0.0549)。这些点是从一个圆形对称的概率分布函数中随机选择出来的,类似于前面介绍过的 EWA 方法。jitter.h 中还包括其他一些抖动向量,有更短些的也有更长些的,可以用它们来尝试不同程度的反走样处理。

---

① 通过在初始化函数中加入 GL_ACCUM 参数来创建累积缓冲区:glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH)。

② 可以在 GLUT 软件包中获得。

图 9.57 给出了用两种方法渲染的一个 3D 场景：左图中的场景没有使用反走样技术；右图显示了在累积缓冲区中取 8 个场景“抖动(jittered)”绘制结果的均值得到的改进效果。可以看到经过反走样处理后的“锯齿”明显减少了。因为对于每一帧场景都要绘制 8 次，所以这种方法会降低性能。

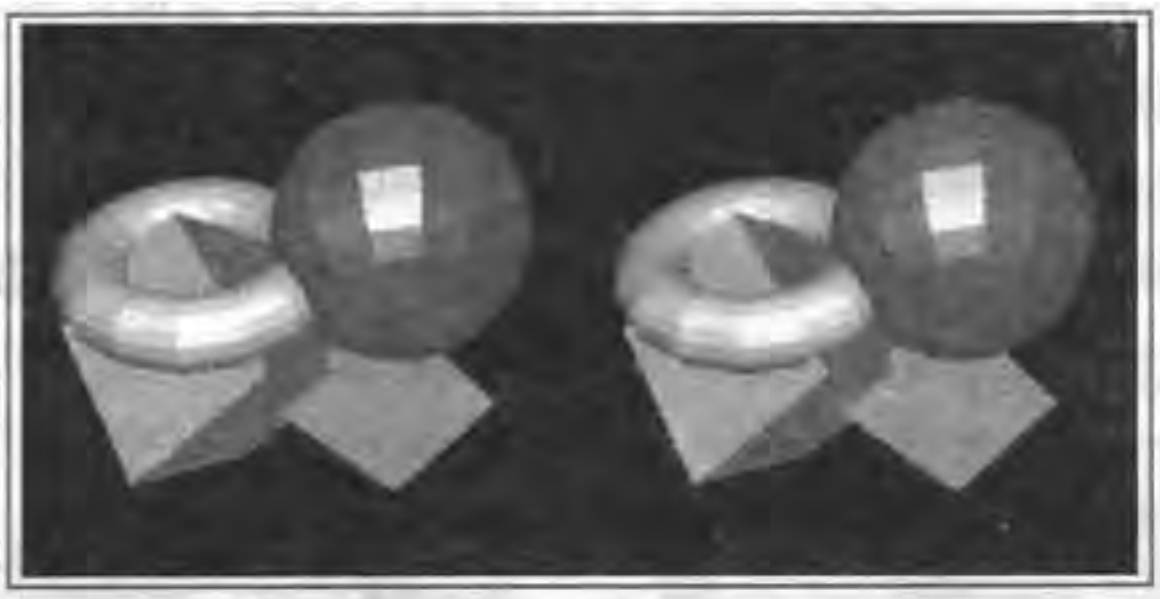


图 9.57　未采用和已采用反走样技术渲染的场景

9.9　生成更多的明暗效果和色彩

早期的图形系统中，显示设备是非常昂贵的，特别是那些支持真彩色的显示器。当然在当今，这些显示器不管是工作用途还是家用，支持真彩色(每个像素占 32 比特位)已是稀松平常，且价格也便宜了很多。因此，在 20 世纪 80 年代末，大量研究人员就研究出一些算法，使得每像素一比特的图像看上去具有每个像素多种明暗的效果。

本节我们会简要介绍其中的一两种方法，包括该问题的历史，以及人的感知系统与图像的交互方式。

一种方法称为半色调(halftone)，这种方法以空间分辨率换取色彩分辨率【Ulichney87，Knuth87】。采用了这种方法的一个熟悉的例子就是报纸。虽然只用了黑色的油墨，报纸的图像看上去却有很多灰度等级。这种效果是通过使用紧密排列的、大小不一的黑色油墨块获得的。多数墨块都较大时其覆盖的区域就显得颜色较深，因为这里深度的平均值更高一些。而墨块都较小时看起来就较浅。眼睛能够把墨块合并，来感受局部区域内的平均深度。然而，报纸图片的空间分辨率要比照片小得多，因为报纸图片是由独立的墨块组成，而这些墨块不可能达到任意小。

在计算机图形学中，数字半色调(digital halftone)或图案法(patterning)采用了点的阵列而不是大小不同的墨块。图 9.58(a)给出了一个例子，使用 2×2 的点阵列(每个点的值为 1 或 0)模拟有 5 个可能亮度级的块。为了讨论方便，我们采用打印业习惯，用 0 级表示白(没有油墨)，4 级表示黑。这个想法之所以能够实现，是因为眼睛看到的是每个 2×2“小块”的平均亮度，从而能够感受到 5 个亮度级。

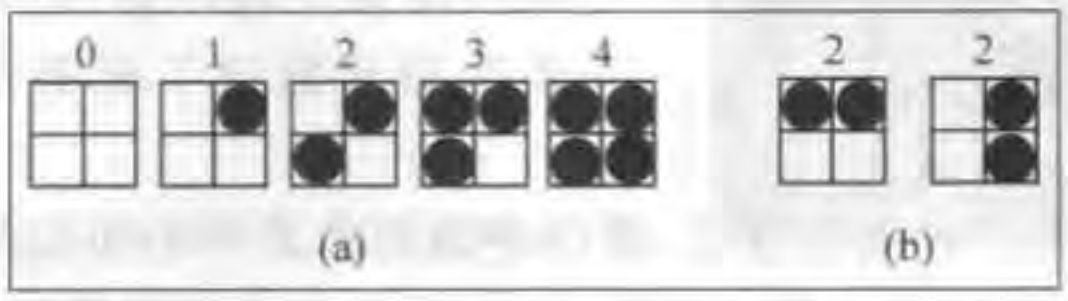


图 9.58　2×2 图案

为了对前面的讨论有直观的理解,考虑一个 100×100 像素的原始灰度图像,亮度值范围是从 0～4。而我们只有一台二值显示器。为了显示这个灰度图像的内容,我们可以使用 200×200 像素显示该图像,然后给每个 2×2 的像素小块设置合适的图案,用来表示灰度级 0,…,4。这又是一个用空间分辨率换取亮度分辨率的例子。

选取图 9.58(a)中黑色元素的位置时,要尽可能做到"不规则",从而避免图像中产生条纹或其他不自然的现象。相反的,如果只把图 9.58(b)中两种图案之一用于表示灰度级 2,那么在某些图案中,图像上可能会出现水平或垂直的条纹。

图 9.59 给出了两幅图:左侧是 256 级灰度图。右侧左图在二值显示器上使用 2×2 图案表示的结果。从图中可以清楚地看到 5 级灰度产生的效果。

图 9.59 使用 2×2 图案化示例

左图:256 级灰度显示没有图案化;右图:2×2 图案化图像在二值显示器上的结果

还可以使用更大的单元尺寸产生更多的灰度等级。图 9.60 显示了使用 3×3 单元的图案,这样能够获得 10 个灰度级。一般 $n \times n$ 个单元 0 和 1 的图案可以产生 n^2+1 种灰度等级(为什么?)。图案化方法在原始图像的分辨率低于所使用显示设备的分辨率时是最适合使用的。

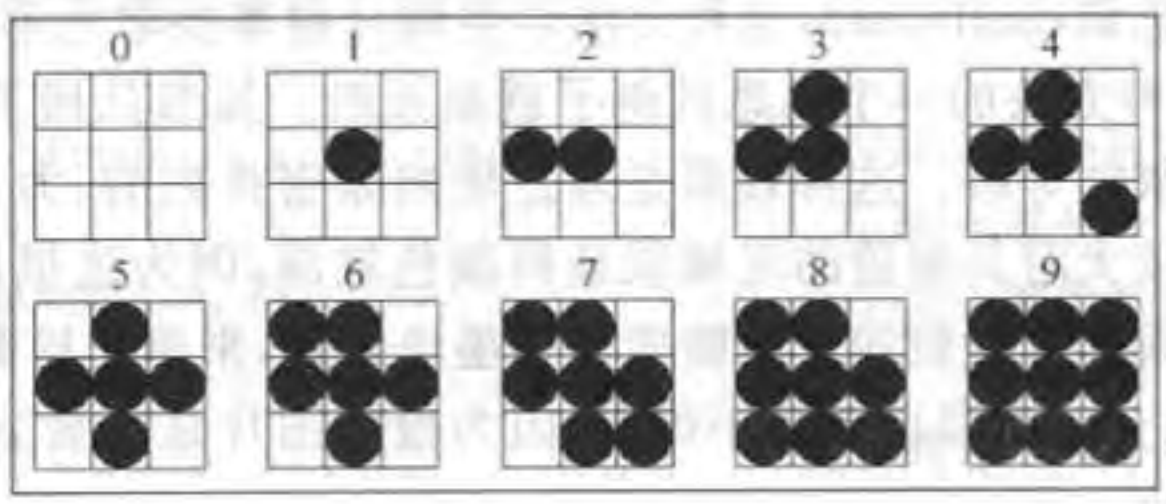


图 9.60 3×3 图案

请参阅【Knuth87】获取数字半色调的更多信息。

前面我们简要介绍了使眼睛看上去比实际更多灰度级的半色调技术。下面我们简单介绍另一种技术:**误差扩散**(error diffusion)

9.9.1 误差扩散

误差扩散提供了另外一种阈值方法,使得多级的像素映射可以显示在只支持较少颜色的显示设备上(虽然现在这种显示器已不多见)。这里再次假定原始像素映射中的每个像素

亮度范围为 0～255，要用适当的方法把每个像素替换为 0 或 1。

如果一个像素的亮度为 A，简单的阈值法就是，如果 $A<128$ 就将其替换为 0，如果 $A\geqslant 128$ 就替换为 1。只要 A 不是 0 或 255，这种方法会产生显示值相对于"真实"值的误差。例如，如果 $A=42$，就会把这个像素设置为 0，这个值比起 42 来太小了。如果 $A=167$，就会把这个像素设置为 1(最高亮度，对应像素值 255)，这个值又太大了，比真实值大了 $255-167=88$。如何处理这个误差呢?

误差扩散方法通过从像素映射中的某些相邻像素减去不可避免的误差来补偿这些误差。我们把产生的误差分成几份传送给还没有经过阈值化的相邻像素，当这些像素阈值化时，就使用调整以后的新值与阈值相比较。按照这种方式，误差在图像中"扩散"开来，以保持正确的平均亮度值。

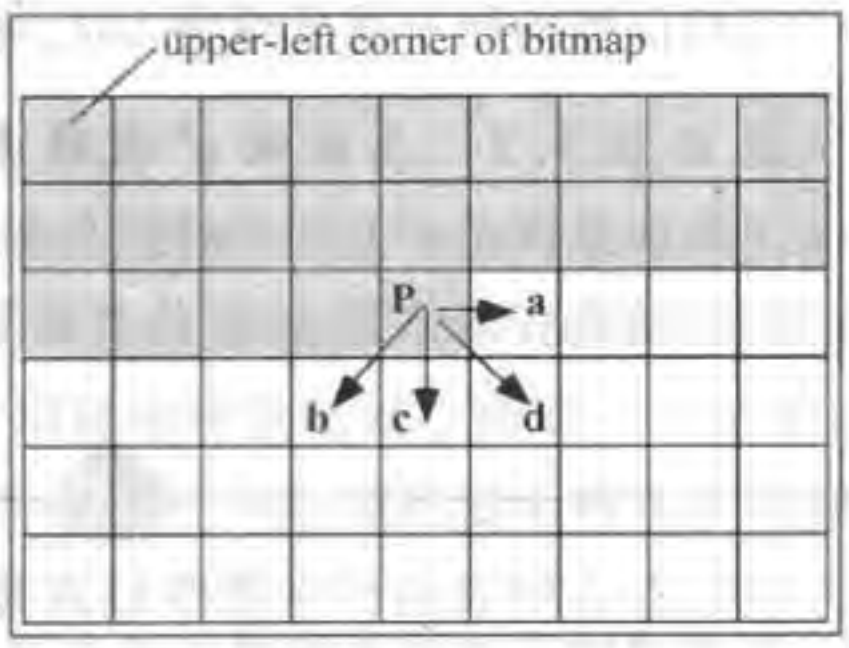


图 9.61　误差扩散过程

图 9.61 显示原(多级灰度)像素映射的一部分。我们假定仍使用通常的顺序从上到下，从左到右处理。假设灰色的像素是已经处理过的。像素 p 刚刚与 128 比较过，并设置为 0 或者 1。

设 A 为该点的像素值。如果 A 小于 128，则显示的像素值为 0，同时误差 E 是 $-A$(显示的亮度值太低了)。如果 A 大于等于 128，则显示的像素值为 1，这时误差 E 是 $255-A$(显示的亮度值高了 $255-A$)。

现在把二值化产生的误差分为几份传给像素 a、b、c 和 d(如图 9.61 所示)。这些像素的值变为：

$$\begin{aligned} a &= a - f_a E \quad \{adjust\ pixel\ to\ the\ right\} \\ b &= b - f_b E \quad \{adjust\ pixel\ to\ lower\ left\} \\ c &= c - f_c E \quad \{adjust\ pixel\ to\ below\} \\ d &= d - f_d E \quad \{adjust\ pixel\ to\ lower\ right\} \end{aligned} \tag{9.12}$$

其中 f_a，f_b，f_c，f_d 是常量。一种常用的比例选择是 $(f_a, f_b, f_c, f_d)=(7/16, 3/16, 5/16, 1/16)$。这些数值相加等于 1，也就是说产生的误差全部传递到相邻像素中，从而保持了局部的亮度均值不变。

当处理到扫描线最末端时，传递给 a 和 d 的误差就不要传送给下一条扫描线的开始(为什么?)，可以把它们简单忽略掉，也可以将所有误差传递给 b 和 c。经验表明最好在处理下一条扫描线时反方向处理，即第一条从左向右，第二条就从右向左处理。这种蛇形扫描的方式被称为蛇纹光栅模式。

图 9.62　经过误差扩散处理后的一幅像素映射

图 9.62 给出了一幅经过误差扩散的 512×512 像素映射在一台过时的显示器上显示的效果。这个误差扩散方法使用了蛇纹光栅和等式(9.12)中的系数。

将此技术扩展，使其支持两个以上颜色的显示设备是很容易的。对每个像素计算最接近

的显示灰度级,把生成的误差按照刚才介绍的方法传递出去。对于彩色图像,可以分别对三个颜色分量进行独立的误差扩散。

Knuth 将抖动技术和误差扩散技术合并为一种称为"点扩散"的方法,这种方法只比单独每种方法稍微复杂一点,但是能够产生更好的效果。算法的细节可以参看【Knuth87】。

练习

练习 9.9.1 手算

对一幅图像最上面一行扫描线执行误差扩散处理,这行扫描线上的所有像素值为 130,下一行扫描线的所有像素值为 132。讨论误差扩散后的结果。

练习 9.9.2 其他的扩散误差量

说明如果误差扩散处理时把全部误差都传送给右边像素得到的效果。再说明如果把一半误差传送给右边像素,一半传送给下面像素得到的效果。

本章研究了使用光栅显示设备显示图像时出现的问题和一些解决方法。光栅显示的基本性质是离散性,即显示出来的图像是以许多离散行和列方式排列的发光像素组成的,每个像素发出的颜色是几个离散的颜色值之一。图像在三个维度上都是离散的——水平方向、垂直方向、颜色值。但是由于像素点的排列非常紧密,而且(通常)有许多种颜色值,所以眼睛-大脑系统会融合临近的像素点而看到它们的"平均"颜色,因此能够把发光点的阵列融合为可以识别的图案。

这种融合使得光栅显示器具有一种基本属性,即可以使区域看上去像是用纯色或图案填充的,基于线绘制的设备如绘图仪很难达到这一点。我们讨论了几种填充区域的方法,特别关注了用多边形定义的区域的填充。

离散性既有好处也有坏处。空间离散性(水平的和竖直的)导致了走样现象的发生("锯齿"):一条对角的直线看起来是有锯齿的,这样观看起来会很不舒服。所以开发了很多反走样技术来改善走样的视觉效果。过分的颜色离散化——比如只有两种颜色值的显示设备——会产生不自然的"色带"或"色岛",而且丢失了原始图像中的许多信息。所以人们又研究出抖动技术,充分利用了人眼融合邻近点这个特点,使眼睛"看到"比实际更多的颜色。

光栅显示还有另外一个基本性质——用数值表示像素,而数值可以保存在内存中。光栅显示器可以看作是一个巨大内存区域对外开放的一个窗口,这个内存区域就是像素映射。显示系统将数字映射为颜色,使得像素映射可以被人们感知。计算机对数值操作的便捷特性打开了一扇通往新技术的大门。光标可以在鼠标的操纵下在屏幕上扫过,窗口可以卷动,图像可以在屏幕上内存和屏幕外内存之间来回切换等。

现代光栅显示器的特点就是可以处理非常多的像素,有时甚至数百万。这使图像能够产生丰富的细节,但是也增加了执行某些操作的时间。复制大量像素值通常要由通过特制的硬件来实现,比如 bitBLT 芯片。这也促使人们开发出了更高效的算法以改进性能。其中许多算法都利用了一致性这个贯穿于图形学中的概念。比如,横向跨度一致性

(span coherence)是指扫描线上许多相邻的像素常常具有同样的颜色；还有扫描线一致性(scan line coherence)，是指相邻扫描线上的像素常常是类似的。

本章讨论的各种技术都是在像素级上操作，以提高光栅显示器上的图像效果。这些技术或者是利用了离散性的优势，或者是弥补了离散性的缺点。因为一幅图像中的像素个数如此众多，所以对于每种算法来说效率都很重要。

案例分析

案例 9.1　读取和显示 BMP 图像文件

(难度系数：Ⅱ)编写一个应用程序，给出 RGBpixmap 类的方法 draw()、read()、copy()、readBmpFile()和 writeBmpFile()的细节，并允许用户读取存储在 BMP 文件中的图像，并利用 OpenGL()显示出来。另外程序还应允许用户用鼠标在显示器上指定一个矩形，并将该矩形内部的像素映射保存到 BMP 文件中。如果用户按下 f 键，就把显示的像素映射围绕其水平中心线反转，显示出上下颠倒的图像。按 v 键就围绕图像的垂直中心线反转。选择多个 BMP 图像进行实验。

案例 9.2　用 OpenGL 融合两幅图像

(难度系数：Ⅱ)9.3 节介绍过，很容易让一幅图像渐变为另一幅图像。图 9.12 中还给出了一个示例。如果这两幅图像分别存储在像素映射 A 和 B 中，那么只需对一系列 t 值绘制这两幅图像的加权平均 A(1－t)＋Bt 即可。

本案例使用 OpenGL 的 alpha 融合功能和图 9.15 中的 blend()方法形成加权平均，基本步骤如下：

a. 读取两个图像文件生成两个大小相同的像素映射 *A* 和 *B*

b. 设置适当的融合函数 glBlendFunc()

c. 清屏

d. 将 *t* 设为值分别为 0，0.2，0.4，0.6，0.8，1

e. 设置 *A* 的 alpha 值为 t

f. 绘制 *B*，完全不透明

g. 绘制 *A*，与背景融合

h. 暂停以观察融合的图像

需要为图像设置一个“全局”alpha 值，即把每个像素的 alpha 值设置为同一个值。这很容易通过在 RGBpixmap 类中添加一个方法 setAlpha(float alpha)实现：

1. 编写方法 setAlpha(float alpha)。只需遍历像素映射中的所有像素值，把其中每个像素的一个分量设置为最大值(255)的 alpha 倍。

2. 以示例 9.3.2 为范例，编写一个应用程序实现两幅图像 *A* 和 *B* 的渐变，使得按 d 键时开始从 *A* 到 *B* 的渐变，按 b 键时开始从 *B* 渐变到 *A*。通过几对图像验证这个程序。

案例 9.3 处理"shape"数据结构

(难度系数:Ⅲ)9.6.1 节介绍了一个用矩形集合表示区域的数据结构。本案例分析中,要求读者开发出实际处理这种结构的程序:

1. 生成一个 shape。编写例程:

```
void pixmap2shape(RGBpixmap& pixmap, shape& shape, Color3 color);
```

该例程从像素映射生成一个 shape 数据结构。这个 shape 结构精确地表示了这个像素映射内颜色为 color 的所有像素组成的所有区域。

2. 从 shape 中生成像素映射。这是前面操作的逆操作,生成一个像素映射,其中包含一个 shape 中描述的全部区域。

```
void pixmap2shape(shape& shape, RGBpixmap& pixmap, Color3 fore, Color3 back);
```

这个例程生产一个足够大的像素映射,使之可以覆盖 shape 描述的区域。这个像素映射用颜色 fore 填充所有 shape 内的像素,其余的所有像素设置为 back 颜色。

案例 9.4 一般的多边形填充

(难度系数:Ⅲ)编写例程:

```
short fillPoly(IntPolyArray& poly);
```

该例程填充多边形 poly。如果多边形是退化或病态的,或者填充过程不成功则返回-1,否则返回 0。按照 9.7 节提出的算法编写,并使用边表和活性边表来提高效率。

测试多边形可以简单的随机生成。用户输入多边形的顶点个数,顶点坐标由程序随机产生,并确保多边形闭合。多边形生成以后,便可进行填充。提高要求:使用图案填充多边形。

案例 9.5 误差扩散

(难度系数:Ⅱ)编写例程:

```
errorDiffuse(IntRect r);
```

该例程对矩形 r 内部的屏幕图像执行方程 9.2.2 所述的误差扩散算法。感兴趣的矩形由案例分析 9.1 中的方法确定。观察误差扩散前后的图像差别。该结果在仅支持少数亮度或颜色的显示器上观察效果会更明显。例程使用扇形模式对相邻扫描线进行扫描。通过几幅图像验证这个程序。

进一步阅读

Jim Blinn 在 Jim Blinn'Corner, Dirty Pixels【Blinn98】中针对本章的主题给出了一个十分通俗的论述,其中包括反走样、抖动和合成。Ulichney 的 Digital Halftoning【Ulichney87】针对抖动给出了一个透彻的论述,同时 Knuth 也对此问题提出了很广泛的看法【Knuth87】。"A New Polygon Based Algorithm for Filling Regions"【Lui00】是 Lui 等人发表的一个区域填充的新方法。OpenGL 红宝书也针对如何控制程序以生成本章中讨论的各种效果进行了详细阐述【Wu04】。

第10章 曲线和曲面设计

本章学习目标

- 开发用于表示和设计曲线的工具
- 描述常用曲线的重要数学性质，例如光滑性
- 研究贝塞尔曲线和B样条曲线的数学性质
- 开发用于设计贝塞尔、B样条和NURBS曲面的工具

预览

10.1节介绍曲线参数化表示的一些重要性质，提出用于度量这些曲线光滑性的方法。10.2节讨论用多项式和有理多项式来表示的曲线类型。

10.3节介绍交互式曲线设计的主要方法，用户通过鼠标交互指定一些控制顶点，利用曲线生成算法预览曲线，然后编辑控制顶点来改变曲线的形状。重点讨论插值控制顶点的曲线和逼近控制顶点的曲线之间的区别。

10.4节介绍贝塞尔曲线，10.5节讨论其性质，这些性质在计算机辅助设计(CAD)中得到广泛应用。10.6节讨论贝塞尔曲线的局限性并且寻找曲线设计的更好的方法，主要介绍分段多项式。从而引出样条函数的讨论。

10.7节介绍B样条，10.8节讨论它们的一些有用的性质。10.9节描述非均匀有理B样条曲线(NURBS)。10.10节讨论一些插值控制顶点的曲线的计算方法。

10.11节处理复杂曲面形状设计的问题，并且把第6章的方法推广到基于贝塞尔，B样条和NURBS曲线的曲面构造。主要涉及将两个曲面片无缝拼接的问题，并且用经典的茶壶模型作为混合曲面片的例子。另外还给出了一些其他曲面设计的例子。

案例展示了用于描述和绘制多种曲线和曲面的工程，此外还介绍了ElliptiPool游戏和绘制贝塞尔，B样条和NURBS曲线和曲面的工程。

10.1 概　　述

到目前为止，我们主要讨论了两种简单的几何物体：由点和直线组成的多点体(Polypoints)和由面集合组成的网格(回顾第6章)。我们想以一种有组织的方式来描述和表达出现在计算机图形学和计算机辅助设计程序中的更加丰富的形状类型。

某些形状会特别引起人们的兴趣，因为它们从审美角度看上去非常愉悦或者它们表示了自然界中一些实际存在的物体的外形。有些形状是专为某种功能需求需要仔细计算得

到,例如如何设计机翼的扫掠方式将给出飞机的最大提升力。还有其他的一些形状,例如汽车的挡泥板曲线,需根据其特定的功效,制造的简易性及用户实际需求来组合设计。

某些形状,例如对数螺旋和行星绕太阳运行的路径,有简明的数学公式使得它们易于分析,但是当我们想写个程序来绘制它们时,这些公式却帮助不大。因此我们需要从一种表示形式转换成另一种更适合于特定任务的表示形式的方法。其他的一些形状可直接由给定的数据来表示,而不是由数学表达式来表示。

10.1.1 作为运动轨迹的参数曲线

曲线参数化表示的重要应用源于描述物体随时间移动的轨迹。例如,当设计一个动画,场景中摄像机的轨迹需要在任意时刻确定。图 10.1 显示了一个在场景中移动的摄像机。它在 t 时刻位于位置 $P(t)$。设计者选择一个合适的函数 $P(t)$使得摄像机按要求移动,或许在 $t=0.1$ 时拍摄快照#1,在 $t=0.2$ 时拍摄快照#2,等等。摄像机的视线方向在任意时刻也都必须被指定(见练习)。

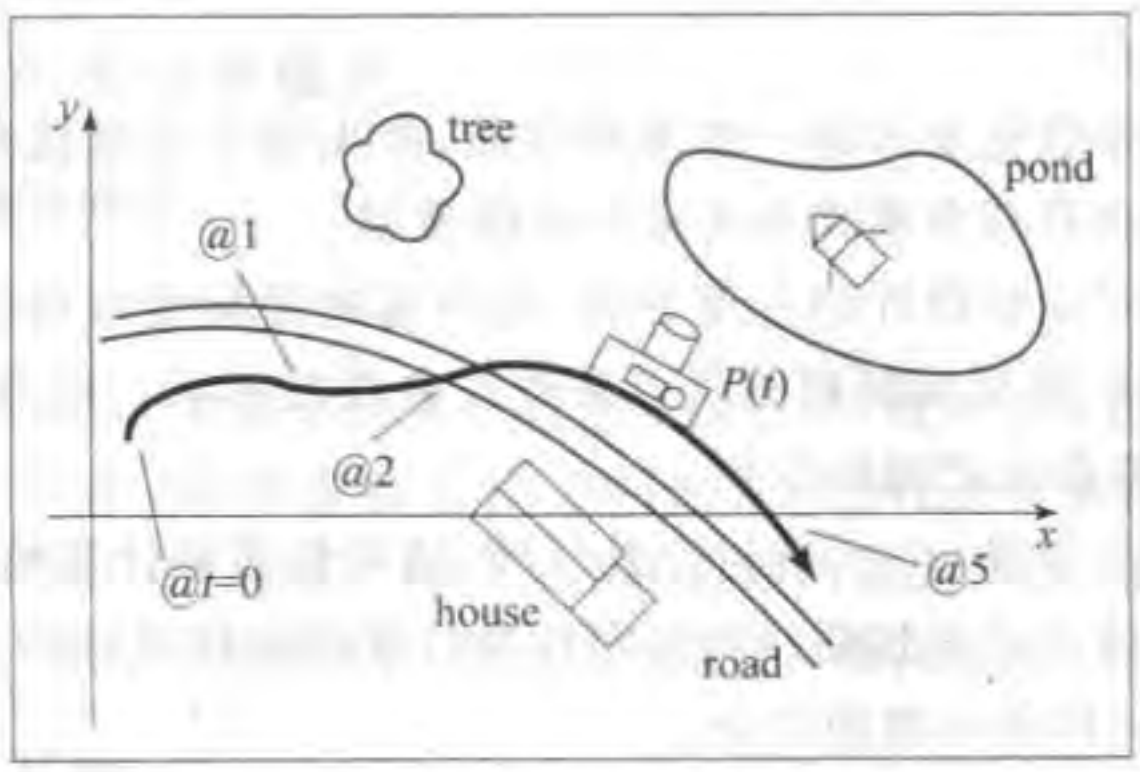


图 10.1 在动画中指定摄像机的位置

除了使得摄像机在指定的时间处于指定的位置,设计者必须确保摄像机沿着 $P(t)$前进时光滑地移动,当动画被回放时没有任何扰乱性的抖动。如下面我们所考虑的,这对摄像机的速度施加了某些条件。其他的物体也可能在动画中移动:在图 10.1 中汽车可能会沿着道路移动,小船可能会改变它的航线,一个人也可能从房子里出来。如在第 6 章中讨论的,人的胳膊和腿可能也会有它们自己的运动。所有这些物体的运动必须用一个适当的方法来描述。

通常,动画在 3D 场景中生成,其中摄像机沿着 3D 的路径运动且每个快照都是通过把场景投射到摄像机的胶卷上形成影像。

10.1.2 运动的光滑性

把参数 t 当成是时间的变量,当 t 增加时把点 $P(t)$看作是沿曲线运动。很自然要问在任意时刻点 $P(t)$沿曲线运动的速度是什么。速度 $\mathbf{v}(t)$是描述 $P(t)$经过曲线时的速率和方向的向量。通过初步的计算,对于 2D 中的曲线有

$$\mathbf{v}(t)=\frac{dP(t)}{dt}=\left(\frac{dx(t)}{dt},\frac{dy(t)}{dt}\right) \tag{10.1}$$

例如，由 $P(t)=(W\cos(t),H\sin(t))$ 给出的椭圆在 t 时刻的速度由 $\mathbf{v}(t)=(-W\sin(t),H\cos(t))$ 给出。$\mathbf{v}(t)$ 的大小通常被称为在 t 时刻的速率。图 10.2 显示了在椭圆上一些点的速度。当 t 改变时，大小和方向都会改变。（哪个位置的速率最大?）

曲线 $P(t)$ 在 $t=t_0$ 时刻的切线可以由含有参数 u 的参数形式 $L(u)$ 给出，如下所示。在 $u=t_0$ 时它明显通过 $P(t_0)$ 且在方向 $\mathbf{v}(t_0)$ 上运动。因此它由下式给出

$$L(u)=P(t_0)+\mathbf{v}(t_0)u \tag{10.2}$$

这些公式使得在各种应用中计算和画出曲线的切线变得很直观。

曲线的**法线**方向也可以在每个点处计算。它在所关心的点处被定义为垂直于切线的方向。因此如上在时刻 t_0 有切线方向 $v(t_0)$，法向方向是第 4 章中向量 v 的任意倍数。用 K 表示常量倍数，法向方向为：

（注意：倍数 K 符号的选择将会区分开**内向法向**和外向法向，在实际分析中根据需要改变 K 的符号。）

$$\mathbf{n}(t_0)=\mathbf{v}^{\perp}(t_0)=(-dy/dt,dx/dt)\mid t=t_0 \tag{10.3}$$

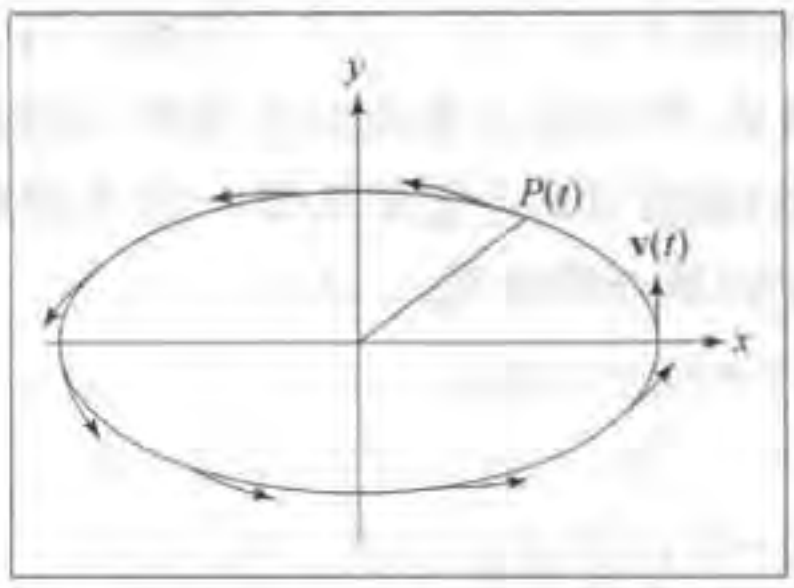


图 10.2　一条曲线上 $P(t)$ 的速度

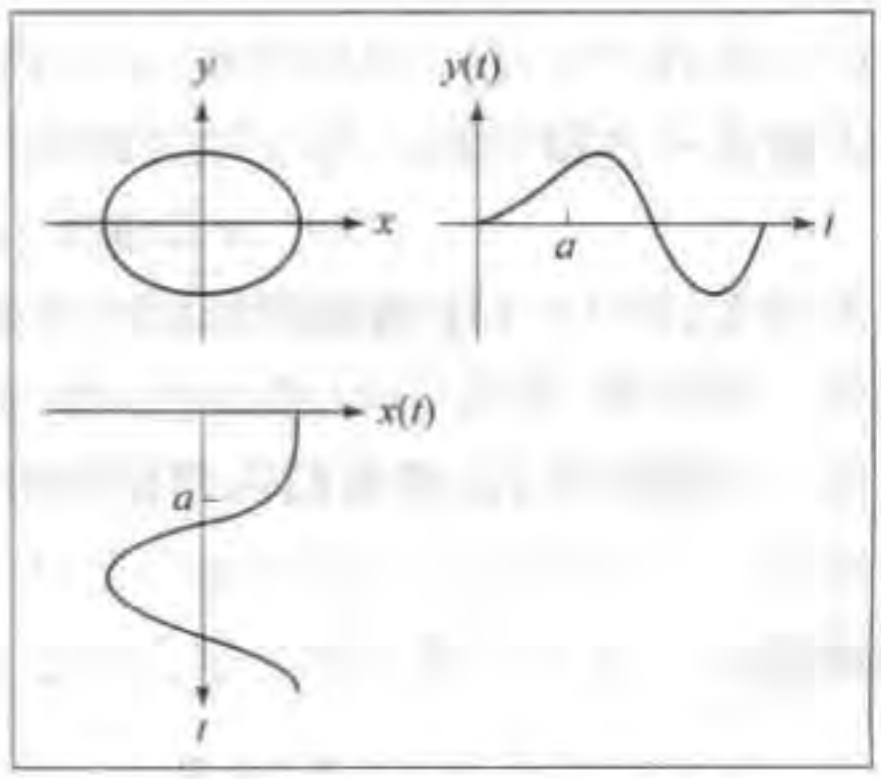


图 10.3　运动速度突然改变的轨线

因此上述示例的等式(10.1)的椭圆，作为例子，其法向向量为 $(-H\cos(t),-W\sin(t))$ 或其任意倍数。这是一个内向法向：乘以 -1 可以得到外向法向。在圆的特殊例子中，$\mathbf{n}(t_0)$ 是 $P(t_0)$ 自身的倍数，因此法向与从 0 到点 $P(t_0)$ 的径向向量方向相同（注意对于椭圆不是这样）。

当曲线 $P(t_0)$ 被用来描述一个物体例如像摄像机如何实时移动时，问题产生：物体沿着轨线是颠簸地移动还是光滑地移动？沿着给定的路线物体可以无穷种方式移动，开始，停止，加速，暂时在一部分曲线的折回，等等。如果我们只绘制曲线，则运动无关紧要：所有的轨线具有相同的图像。但是用参数函数来表示摄像机的运动，情况就会变得非常复杂：沿着轨线运动的细节被记录在一系列得到的快照中。突然的加速或跳跃通常是不希望出现的。

作为例子，考虑一个沿着椭圆轨线移动的摄像机，但有一个替换的参数表示：在 $t=a$ 时刻运动的速度突然被加速 3 倍：

$$P(t)=(x(t),y(t))=\begin{cases}(W\cos(t),H\sin(t)) & 0<t<a\\ (W\cos(3t-2a),H\sin(3t-2a)) & a<t<\dfrac{2(\pi+a)}{3}\end{cases} \tag{10.4}$$

图 10.3 展示了这种情况。轨线依然是椭圆，但是在 $t=a$ 时刻 $x(t)$ 和 $y(t)$ 开始以 3 倍的速度加速变动，所以导数在 $t=a$ 时刻有个不连续点。在这个时刻，时间轴好像以 3 倍比

例被压缩了。

参数曲线的形状不受这个跳跃影响,但是沿曲线点 $P(t)$ 的运动的本质在 $t=a$ 时刻前后是有显著不同的。

练习

练习 10.1.1 沿着椭圆上点的速度公式

根据等式(10.4)中的曲线给出在 a 时刻前后的一阶导数公式 $P'(t)=(x'(t),y'(t))$。速度相同吗?速率相同吗?

$$P'(t)=(x'(t),y'(t)) \tag{10.5}$$

当我们研究曲线和摄像机运动的光滑性,很容易想到描述曲线的不同种类的光滑性。我们将对两种光滑性有兴趣,它们分别被称为**参数连续性**和**几何连续性**。

参数连续性

我们说曲线 $P(t)$ 在 t 时间间隔 $[a,b]$ 内处处 k 一阶参数连续如果在 $[a,b]$ 内曲线所有点有直到 k 阶的一连续导数。为了简洁的表达,我们说:

$$P() \text{ 在}[a,b]\text{ 内 } k \text{ 阶光滑} \tag{10.6}$$

作为例子,图 10.4 中的椭圆处处 0 阶光滑,因为函数 $P(t)$ 自身是处处连续的。但是它不是处处一阶光滑,因为 $P'(t)$ 在 $t=a$ 有一个不连续点(除了 $t=a$ 它是处处一阶光滑的)。为了避免抖动的动画,通常我们强调摄像机的运动必需的是一阶光滑。

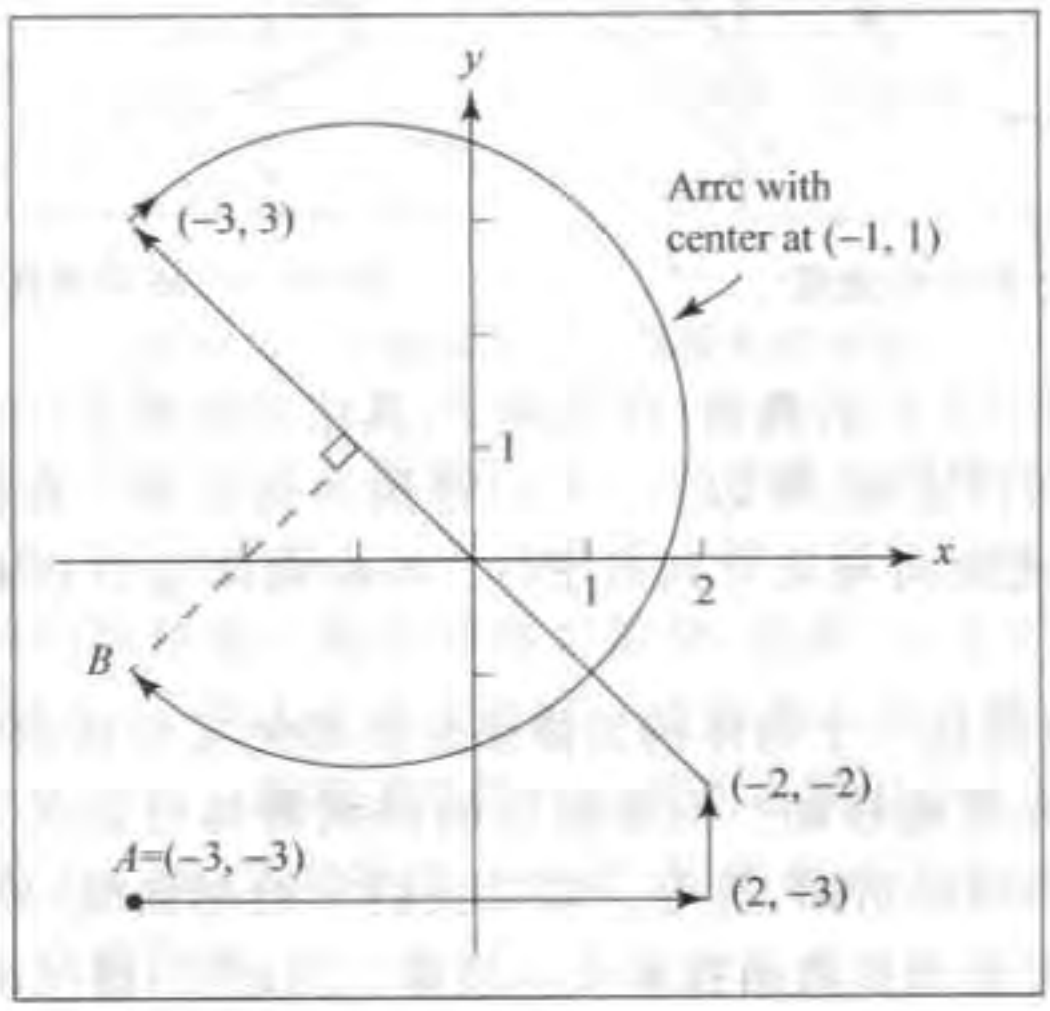


图 10.4 什么是参数表示?

有连续的速度和加速度的曲线是二阶光滑的。注意 $[a,b]$ 上二阶光滑函数必定是一阶和 0-阶光滑的。但一阶光滑函数未必是二阶光滑的。

回顾在第 3 章中讨论的三种类型的螺线(阿基米德螺线、对数螺线和等角螺线)。我们需要研究这些曲线的光滑阶。例如,对于对数螺线,其参数形式为 $f(\theta)=Ke^{a\theta}$,显然为 0-阶光滑的,且其速度向量为:

$$f'(\theta)=(-Ke^{a\theta}\sin(\theta)+Ke^{a\theta}\cos(\theta)ae^{a\theta}$$

$$f'(\theta) = Ke^{a\theta}\cos(\theta) + Ke^{a\theta}\sin(\theta)ae^{a\theta})$$

因为上述表达式中所有的项都是连续的，所以对数螺旋是一阶光滑的。图 10.5 显示了对数螺线的图像，其外形很像鹦鹉螺的外壳轮廓线。

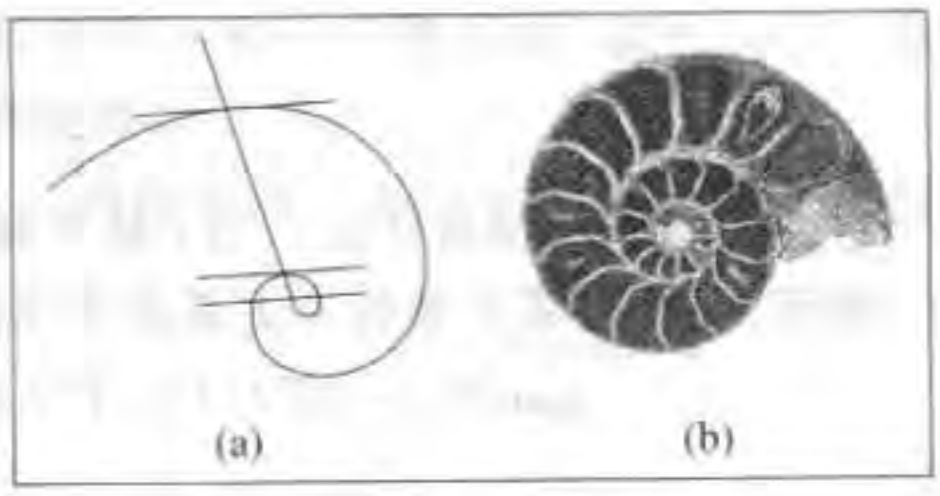


图 10.5　对数螺线和鹦鹉螺

在这节中，我们基于曲线的视觉光滑性而非沿着曲线的运动光滑性，讨论另一种光滑性的不同次数。

- 如果曲线在一个区间是连续的，我们称它为 0 阶光滑。
- 如果曲线在一个区间存在一阶连续导数，我们称它为一阶光滑。
- 如果曲线在一个区间存在一阶及二阶连续导数，我们称它为二阶光滑。
- 如果曲线在一个区间存在一阶、二阶及三阶连续导数，我们称它为三阶光滑。

几何连续性(G^k 连续)

一般地，曲线的几何连续性要求曲线的各阶导数向量的方向是连续的，即使它们在速率上有不连续点。等式(10.4)中椭圆除了在 $t=a$ 之外有连续导数。

等式(10.4)中的例子还表明，正是这个奇怪的参数化导致曲线的这种类型的光滑性得不到满足(在 $t=a$)。这是容易理解的。为了讨论这种连续性，我们需要如下的一些定义。

- G^0Z 连续和 0 阶光滑相同：它仅仅表示 $P(t)$ 关于 t 在区间$[a,b]$上连续。
- 在$[a,b]$上 G^1 连续表示对于区间$[a,b]$上每个 c 有 $P'(c-)=kP'(c+)$其中 k 为常数。因此在此区间上任一点(c)的速度向量的长度可能以某个数量 k 的倍数跳跃变化，但是其方向是连续的。
- 在$[a,b]$上 G^2 连续表示一阶和二阶导数有连续方向：对区间$[a,b]$上任一点 c 有 $P'(c-)=kP'(c+)$和 $P''(c-)=mP''(c+)$，其中 k 和 m 为常数。

注意：从这些定义中可知，曲线的几何连续性实际上是指曲线的速率或速度向量的长度(和它们的导数)是连续的。

练习

练习 10.1.2　绘制椭圆的切线

写一个程序画出在参数值 t 由图 10.3 生成的切于椭圆的短线段。

练习 10.1.3　圆锥曲线的切线和法线

在任意值 t 处计算抛物线和双曲线的切线和法向量的表达式。

练习 10.1.4　求参数表示

求图 10.4 所示曲线的参数表示。曲线在 $t=0$ 时刻始于点 A 且在所示的整个形状上以常速率移动，在 $t=20$ 时刻到达点 B。

练习 10.1.5　圆的另一种参数表示

除了由时间变换产生的表示的多样性，某些曲线允许在特征上有很大差别的简单表示，

例如下面这个。证明下列形式在 t 从 0 变化到无穷时生成圆的一部分。

$$x(t) = a\frac{1-t}{1+t}$$

$$y(t) = 2a\frac{\sqrt{t}}{1+t} \tag{10.7}$$

它的圆心和半径是什么？当 t 从 0 到 1 变化时生成圆的哪部分？

提示：回顾对某个参数 b 关系式 $T=\tan(b/2)$。然后

$$\sin(b) = 2T/(1+T^2) \text{ 和 } \cos(b) = (1-T^2)/(1+T^2)。 \tag{10.8}$$

10.2 用多项式描述曲线

多项式是基本的数学对象且在计算机图形学中经常使用，是因为它们能有效计算且具有很好的性质。在本章的剩余部分我们将关注多项式的特殊形式：这里我们研究由简单多项式构成的隐式形式和参数形式的相互关系。

首先介绍一个概念：一个 t 的 **L 次多项式**是一个下式给出的函数

$$a_0 + a_1 t + a_2 t^2 + \cdots + a_L t^L \tag{10.9}$$

其中常数 $a_0, a_1, \cdots a_L$ 是它的**系数**，每个都与 t 的某个次幂对应。多项式的**次数**是当 t 升幂时的最高次幂的指数。为此要成为 L 次多项式，我们强调 a_L 不等于 0。**阶**是多项式中的系数个数(这里是 $L+1$)，它比次数大 1。

一次多项式曲线

我们已经研究过线性多项式，知道 $x(t)$ 和 $y(t)$ 的线性参数形式生成一条直线，并且对应的隐式形式对于 x 和 y 是线性的。例如，参数表示为 $P(t)=a_0+a_1 t$ 的曲线是一条在时刻 0 通过 a_0 且在时刻 $t=1$ 通过 a_0+a_1 的直线。注意关于 $P(t)$ 的式子实际上是两个式子，一个是 $x(t)$ 另一个是 $y(t)$。当然，在 3D 中会有第三个式子是 $z(t)$。

二次多项式曲线

我们自然要问哪些曲线形状可以用二次多项式 $x(t)$ 和 $y(t)$ 表示：

$$x(t) = at^2 + 2bt + c$$

$$y(t) = dt^2 + 2et + f \tag{10.10}$$

其中 a、b，等等是常数。答案是简单的：对于任意选择的常数 $a, b, \cdots, f$，这条曲线总是一条抛物线。因此没有办法用这个形式来表示一条椭圆或双曲线。可以试着画一下如下两种情况时的抛物线的形状：

- 情况 #1：$a=b=0, c=d=-2$，和 $e=f=-6$
- 情况 #2：$a=b=3, c=d=8$，和 $e=f=4$

二次隐式形式

通过另一种方式，我们来看二次隐式形式，也就是，关于 x 和 y 的二次多项式。回想解

析几何中一般的二次隐式形式为：

$$F(x,y) = Ax^2 + Cy^2 + Dx + Ey + F \tag{10.11}$$

（注：某些作者写这个二次隐式形式时可能会有关于 x 和 y 的乘积项，如 $+Bxy$。事实上，通过一个适当的坐标轴的旋转变换，这项可以被去掉。）

其中 $A,C,\cdots,F$ 为常数。假定 A 和 C 不全为 0，否则生成一条退化曲线。同样回想起 $F(x,y)=0$ 描述的曲线形状是一个圆锥曲线。所表示的特殊的圆锥曲线可以通过检查系数 A 和 C 的符号来简单地确定，如下：

如果 $AC > 0$，它是一个椭圆　(10.12)

如果 $AC = 0$，它是一个抛物线

如果 $AC < 0$，它是一个双曲线　(10.13)

因此如果 A 或 C 有一个为 0 但不都为 0 时它是抛物线，如果 A 和 C 有相同符号它是椭圆，且如果 A 和 C 有相反符号它是双曲线。

它描述的圆锥曲线依赖于离心率[①]的值，ε，如图 10.6 所示。

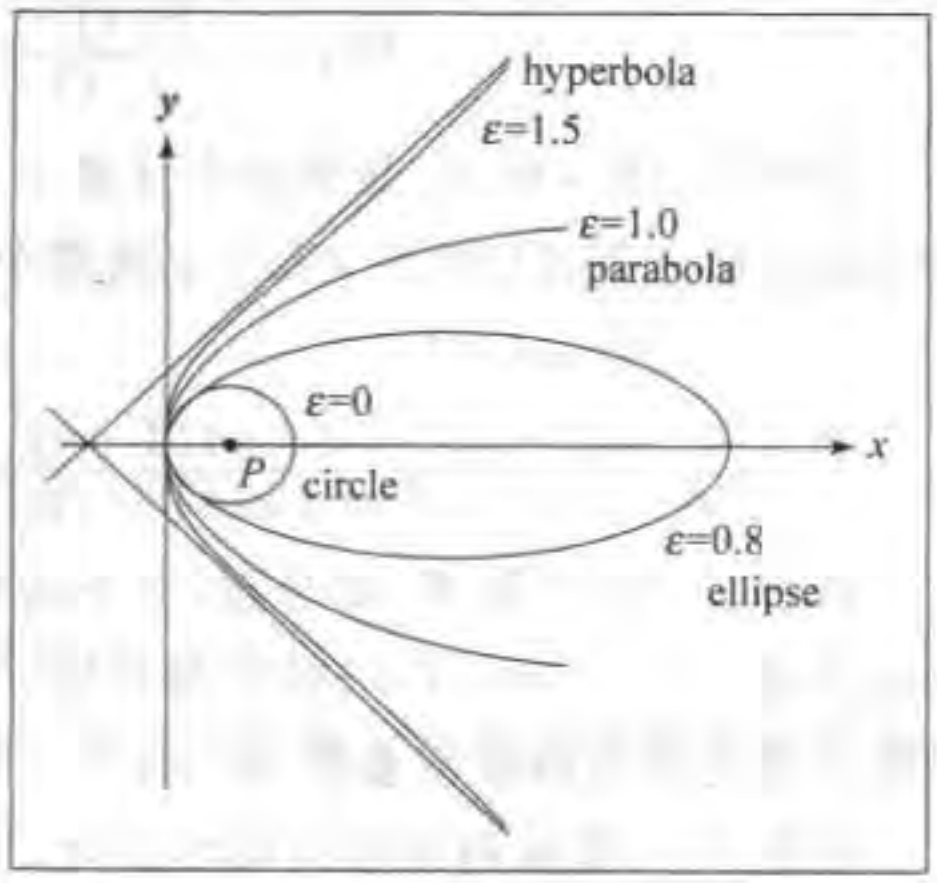


图 10.6　具有不同离心率的二次曲线

练习探究了为了画这批曲线的可能的参数化选择。注意在抛物线之外，我们无法为圆锥曲线找到多项式参数化形式。为此我们必须等到下一小节和**有理**多项式的介绍。

三次多项式曲线和更高次多项式曲线

参数形式到隐式形式与隐式形式到参数形式

如果我们把自己限制到由一次和二次多项式参数化的曲线，我们看到这种情形是很好理解的。然而，当多项式是更高次时事情变得比较复杂。在早期的曲线设计研究中，Sederberg 等【Sederberg85】证明对于给定的多项式函数 $x(t)$ 和 $y(t)$ 总有可能找到其隐式形式，但是只有对一次或二次的隐式形式才总能找到其参数形式。

本章中关于贝塞尔和 B 样条曲线我们主要的工具是三次多项式，并且我们将看到它为曲线设计提供了一个强有力的方法。

但是这些方法将不会从一个隐式形式出发且试图寻找其参数化形式，而是从由设计者设定的控制顶点集合开始且应用某个特定的算法沿曲线产生一些点。因此设计者如果有必要可以编辑控制点的位置且重新查看曲线。在许多方式中，与纯粹的数学方法相比，这是一种自然的曲线设计方法，因为它比较形象，当程序生成形状时允许设计者查看曲线设计的进程。

在结束当前对曲线设计的解析方法研究之前，我们先简要地预览将在后面研究的**有理多项式**函数类（如我们将要看到的，它们是 NURBS 的基）。下面对有理多项式的简要介绍将说明一些点的定义如何确定一个曲线的形状。它还产生了一个重要的结果：圆锥曲线可以用二次多项式的比精确表示。

---

① 作为感兴趣的一点，地球轨道的离心率是 0.0167，冥王星的是 0.25。

有理参数形式

考虑参数化,其中 $x(\cdot)$ 和 $y(\cdot)$ 被分别定义为两个多项式的比。线性情形在练习中探究过。等式(10.14)显示了由两个二次多项式的比组成的参数形式。我们看这里的二次多项式的情形且关注特殊的参数化形式:

$$P(t)=\frac{P_0(1-t)^2+2wP_1t(1-t)+P_2t^2}{(1-t)^2+2wt(1-t)+t^2} \tag{10.14}$$

其中 P_0,P_1,和 P_2 是平面中任意三点。它们在本书中被称为**控制顶点**,因为它们控制曲线的形状。等式(10.14)实际上是两个式子,是如下曲线的简化形式:

$$\begin{aligned}&(x(t),y(t))\\&=\left(\frac{x_0(1-t)^2+2x_1wt(1-t)+x_2t^2}{(1-t)^2+2wt(1-t)+t^2},\frac{y_0(1-t)^2+2y_1wt(1-t)+y_2t^2}{(1-t)^2+2wt(1-t)+t^2}\right)\end{aligned} \tag{10.15}$$

其中 x_0 和 y_0 是 P_0 的分量,对于另外两个点类似。分子中二次多项式的系数是控制顶点的分量。$x(\cdot)$ 和 $y(\cdot)$ 的分母相同。它们也是二次的但不依赖于控制顶点。然而,它们依赖于将要讨论的**权重参数**,即 w。

注意 $P(t)$ 是控制顶点的线性组合。如我们在 4.5.2 节中所看到的,为了使 $P(t)$ 成为一个有意义的点,它必须是这些点的仿射组合。

注意如果我们在等式(10.15)取 $t=0$,右端项简单地趋于 (x_0,y_0),因此这个曲线通过称为点 P_0。类似的,它在 $t=1$ 通过点 P_2。对于中间的 t 值 $P(t)$ 以一种复杂的方式依赖于所有三点。

图 10.7(a)图示了三个示例的控制点以及曲线当 t 从 0 增加时如何从 P_0 形成和当 t 到达 1 时如何在 P_2 点结束。图中询问中间的形状是什么样的?图 10.7(b)说明了结果:曲线是二次曲线段的一种且其类型依赖于 w 的值:

- 如果 $w<1$,它是一个椭圆。
- 如果 $w=1$,它是一个抛物线。
- 如果 $w>1$,它是一个双曲线。 (10.16)

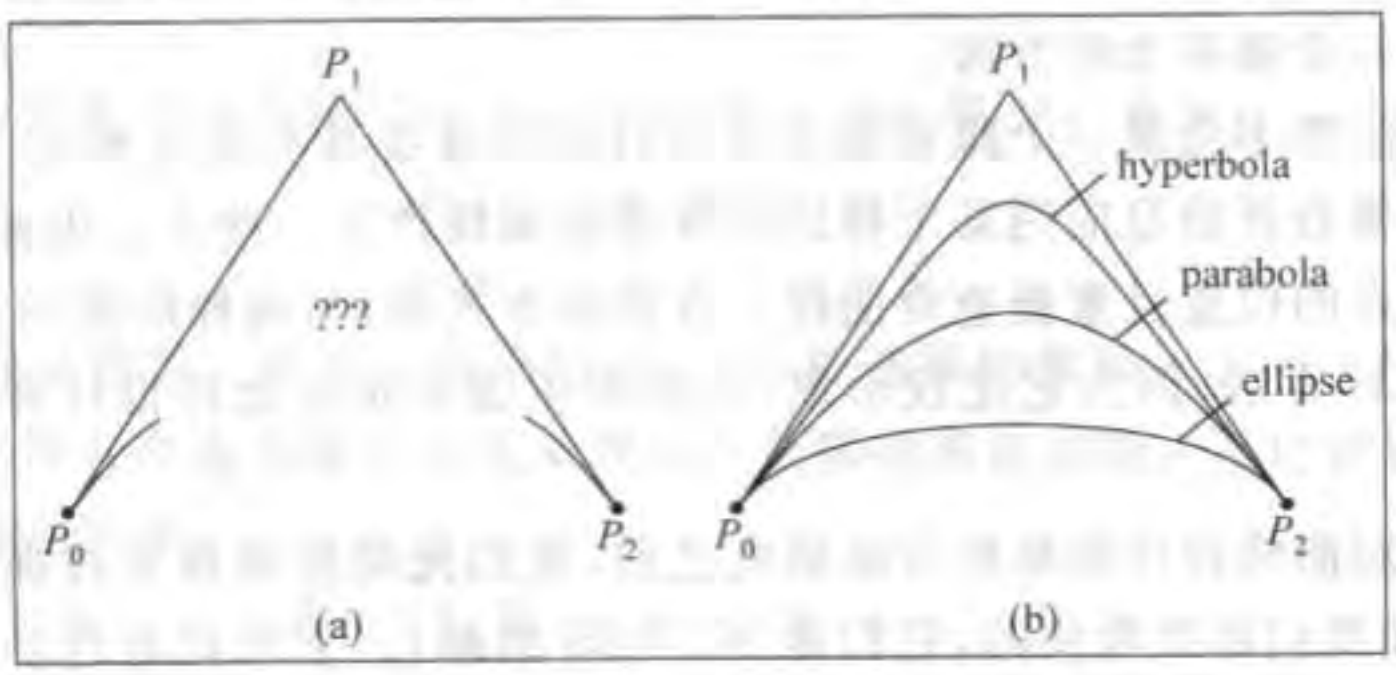


图 10.7 利用二次有理多项式生成圆锥曲线

因此我们有个参数化的生成圆锥曲线的方法,不用求助于之前使用的三角函数。练习说明了怎样生成这些曲线的另一半和怎样用这个参数形式生成圆。

练习

练习 10.2.1　退化的二次方程式

给出等式(10.11)中系数 $A,B,\cdots,F$ 的例子使得曲线 $F(x,y)=0$ 是：

(a) 不存在(例如，没有点满足 $F(x,y)=0$)；

(b) 单个点；

(c) 一条直线；

(d) 两条平行线。

练习 10.2.2　线性有理参数化

讨论由下列参数形式生成的曲线类：

$$x(t)=\frac{a+bt}{e+ft}$$

$$y(t)=\frac{c+dt}{g+ht} \tag{10.17}$$

其中 $a,b,\cdots,h$ 为常数。能否生成非直的曲线？给出一些例子说明这类曲线有多丰富。

练习 10.2.3　是直的吗？

从图 10.7 中看出如果 P_0、P_1 和 P_2 位于一条直线上，整个曲线应该是一条直线。证明这是否是真的。

练习 10.2.4　用有理二次多项式画圆锥曲线

图 10.8(a)显示了一个内切于等边三角形的圆。圆的三分之一可以利用等式(10.14)的参数表示基于所示的点 P_0，P_1，和 P_2 来绘制。因此整个圆可以被当作三个圆弧来画，每段用三个点来生成。可以证明【Farin90】对于圆要用的适当的权重 w 为 $w=\cos(a)$，其中 a 是角 $P_2P_0P_1$。对于等边三角形这个角是 60°。

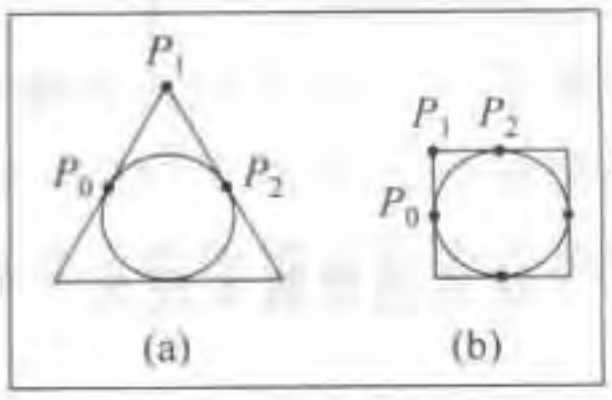


图 10.8　圆可看作三段或四段圆弧

(a) 对于五个 t 值画出等式(10.7)中点 $P(t)$，使得这些点沿圆弧从 P_0 到 P_2 接近均匀分布。

(b) 这与用($\cos(\cdot)$，$\sin(\cdot)$)的样本点画圆相比是否是更有效的方式？讨论。

(c) 对于 10.8(b)中所示四个圆弧重复上述(a)和(b)的工作，在这种情况下角 $P_2P_0P_1$ 是什么？

10.3　关于交互式曲线设计

目前为止所讨论的曲线是基于相对简单的数学公式。现在我们想扩大任务，研究适合于某个特定目标的更加复杂的曲线。特别地，我们希望开发一些允许设计者通过简单地指定控制顶点的小集合来生成大量形状的工具。

例如，假定设计者希望生成图 10.9(a)中曲线的计算机表示。这条曲线可能是汽车挡

泥板,涡轮机叶片或电钻外壳设计的一部分。或者它也可能是正在拍摄快照的摄像机从场景中掠过的轨线。目的是以某种形式捕获曲线的形状允许它任意重建,按要求调整形状和大小,将这些信息发送给机器用于自动切割和造型,等等。

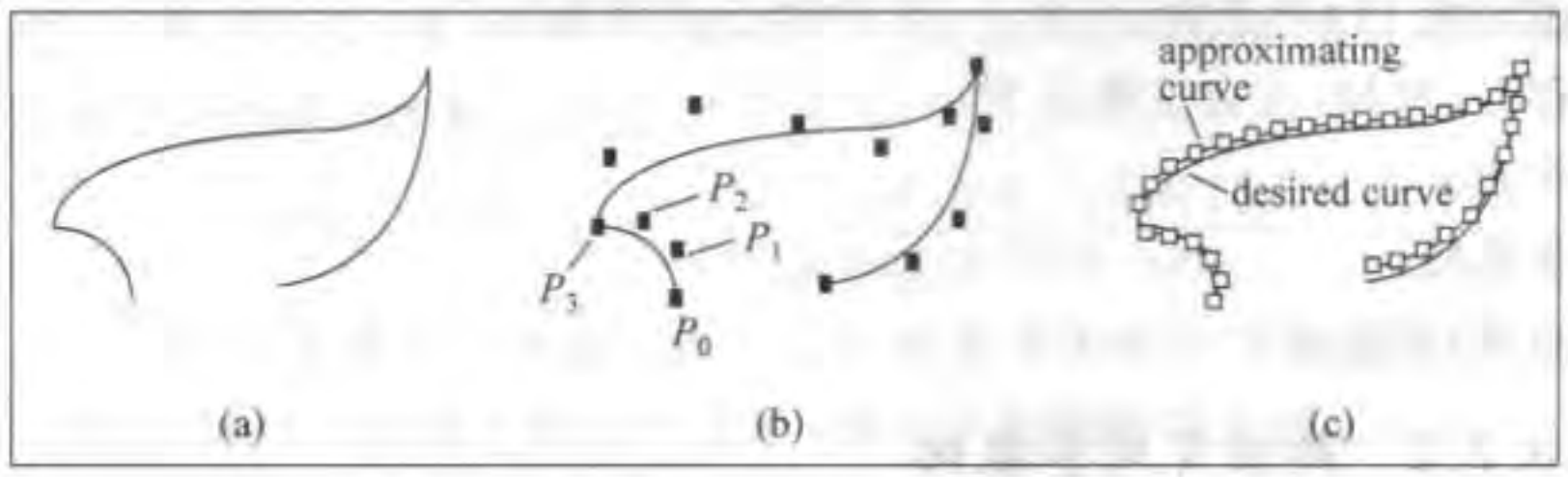


图 10.9 一个曲线设计流程

为了输入曲线,设计者把曲线的草图录制到绘图板上,然后沿着曲线移动光标在一组接近曲线的**控制顶点** $P_0, P_1, \cdots$ 处点击,如图 10.9(b)所示。控制顶点序列通常被叫做**控制多边形**[①]。设计者根据丰富的经验和对曲线生成算法特征的深入理解来输入控制顶点-这个算法将在后面用来根据数据点重新生成曲线。

如图 10.10 所示,算法的作用是对于任意给定的 t 值生成点 $P(t)$。算法的输入数据是一起确定点 $P(t)$所在曲线的控制顶点集合。

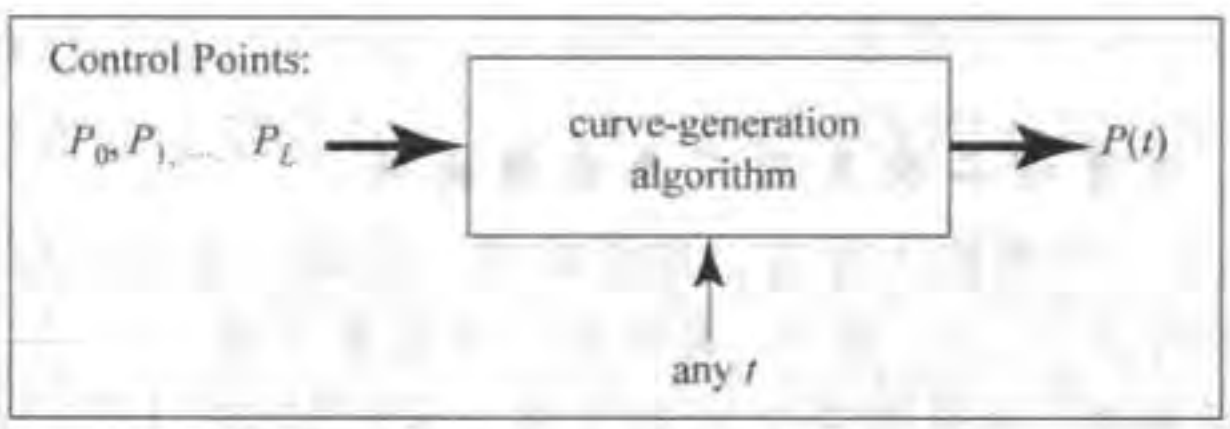


图 10.10 曲线生成算法

算法通常被实现为一个函数

```
Point2 CurvePt(double t, RealPointArray pts)
```

对于某个区间内任意 t 值返回一个点。为了绘制这条曲线用户可以选择一系列 t 值,在每个 t 值处计算 CurvePt(),然后用线段把这些点连接起来形成一个多边形折线。图 10.9(c)显示了一些小正方形的集合,表示给定图 10.9(b)中控制顶点序列由一个合理的算法可能返回的点。

由这些点所定义的折线很有希望能逼近当设计者输入控制顶点时头脑中的原始曲线。如果生成的曲线不能对原始曲线提供合适的近似,设计者将编辑控制顶点,大概用鼠标以这种和那种方式移动控制顶点且通过多次调用 CurvePt()来重新生成曲线。这个迭代过程继续进行直到设计者满意为止。因此交互设计由以下步骤组成:

(1) 确定初始控制顶点。

(2) 使用算法生成曲线。

---

① 控制顶点序列通常指定一个多边形折线形而非一个封闭的多边形,但是我们将继续使用"多边形"

(3) 如果对曲线满意,停止。

(4) 调整某些控制顶点。

(5) 回到步骤 2。

插值与逼近

图 10.11 区分了主要的两类曲线生成算法。图 10.11(a)显示了生成**插值**控制顶点的曲线 $P(t)$ 的算法:$P(t)$ 精确地通过这些控制顶点且在这些点中间形成光滑的曲线。图 10.11(b)使用了生成逼近控制顶点的曲线 $R(t)$ 的算法。$R(t)$ 依次被每个控制顶点所吸引,但实际上并不经过所有控制顶点。从随后各种算法的学习中我们将会看到每种算法都有优点和缺点。

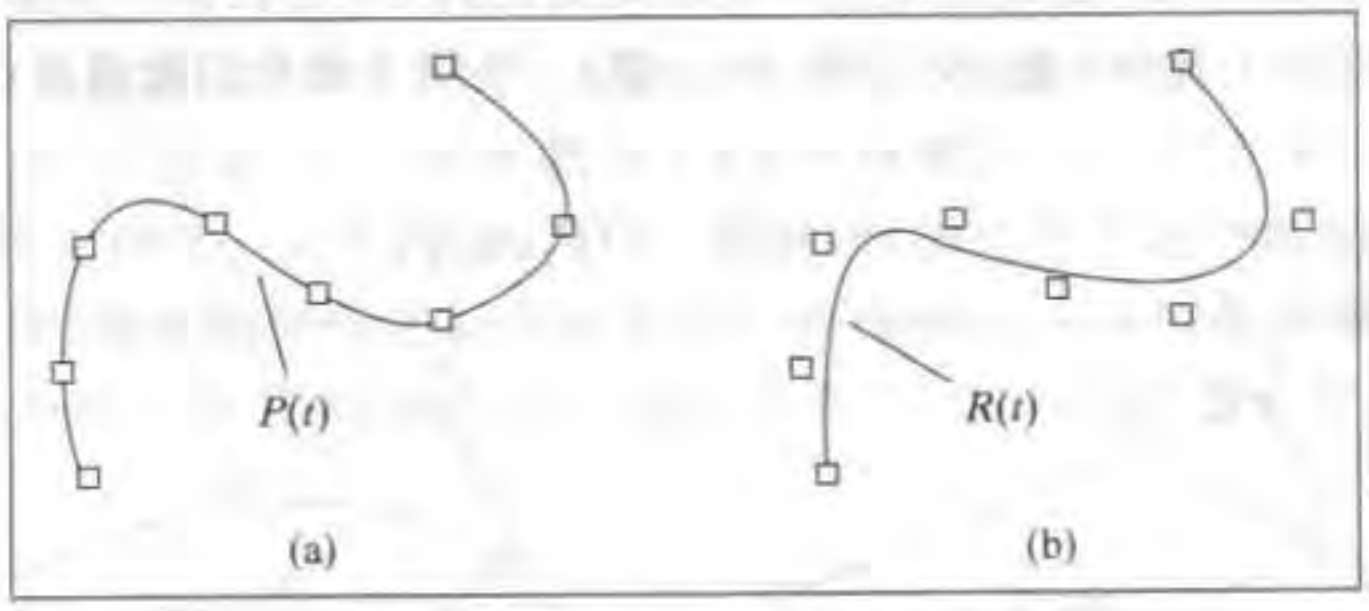


图 10.11　插值和逼近曲线生成算法

迭代曲线设计的流程是计算机辅助设计(CAD)领域中一个重要方面且当设计一个要加工的物体时经常被使用。

下面几节我们将推出一系列曲线和曲面设计的技术。在许多可以采用的方法中选择时,重点放在用贝塞尔和 B 样条的交互曲线设计。这些曲线族在 CAD 应用中已经变得很流行。我们的介绍将会比较简要,但是我们会提供足够的细节使读者能够编写程序来执行交互曲线设计和绘制所设计的物体。

我们首先介绍几种曲线逼近的技术,主要研究贝塞尔曲线和 B 样条曲线。尽管设计者似乎总是希望得到的曲线能够插值控制顶点,但是我们将看到逼近方法是有优点的。接下来我们再研究如何调整算法确保对控制顶点的插值。

10.4　用贝塞尔曲线进行曲线设计

我们以简洁精致的生成贝塞尔曲线的**德卡斯特里奥**算法开始,这个算法是 CAD 的基础。贝塞尔曲线由 Paul de Casteljau 于 1959 年发明,后来又由 Pierre Bézier 在 1962 年左右独立发明。这些算法都成为两个汽车公司(雪铁龙和雷诺)中作为汽车车身形状设计的 CAD 系统的重要部分。

10.4.1 德卡斯特里奥算法

德卡斯特里奥算法用一系列点 $P_0, P_1, P_2, \cdots$,对于 0 到 1 之间的每个 t 值为点 $P(t)$ 构造一个定义好的值。因此这个算法提供了一种由一组点生成一条曲线的方法。改变这些点就改变了这条曲线。构造过程基于一系列熟悉的易于实现的内插步骤(回顾第 5 章)。因为内插是一个很稳定的过程,对其生成的曲线可能会推出许多有价值的性质。

在 3 点之间内插生成抛物线

从三个点 P_0、P_1 和 P_2 开始,如图 10.12(a)所示。在 0 和 1 之间选择某个 t 值,比如 $t=0.3$,把点 A 放在从 P_0 到 P_1 的线段上占两点距离的比例为 t 的位置。类似地,把点 B 放在从端点 P_1 到 P_2 的线段上比例 t 处(使用同一个 t 值)。从第 5 章我们知道新点可以被表示为

$$A(t) = (1-t)P_0 + tP_1$$
$$B(t) = (1-t)P_1 + tP_2 \tag{10.18}$$

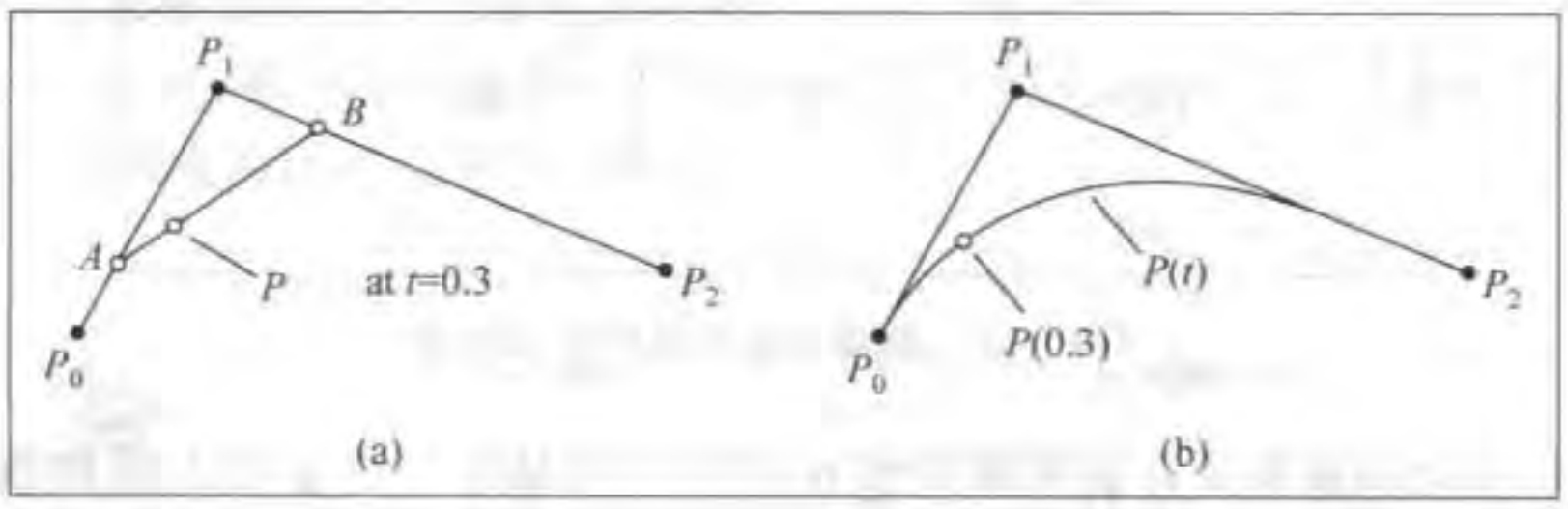


图 10.12 基于三点的德卡斯特里奥算法

现在对这些点重复线性插值步骤(还使用同一 t 值)。寻找位于从 A 到 B 线段上比例为 t 处的点 $P(t)$:

$$P(t) = (1-t)A + tB \tag{10.19}$$

如图 10.12 所示。例如,对 $t=0.5$,$P(0.5)$ 就是给定的三点的"中点的中点"。如果对于 0 和1 之间的每个 t 值执行这个过程,曲线 $P(t)$ 将被生成,如图 10.12(b)所示。这条曲线的参数形式是什么?通过直接把等式(10.18)代入等式(10.19),我们得到

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \tag{10.20}$$

它可以被表示为第 5 章中函数 Tween()项的形式(见练习)。

$P(t)$ 的参数形式是 t 的二次多项式,所以从 10.2 节中知道这条曲线是抛物线。即使 t 从 $-\infty$ 到 ∞ 变化,这条曲线依然是抛物线。显然,在 $t=0$ 处曲线通过点 P_0,在 $t=1$ 处曲线通过点 P_2。(为什么?)因此我们得到一种定义好的过程能够基于给定的三点生成一条光滑的抛物线。

如果使用的控制顶点多于三个将会怎样?最常用的贝塞尔曲线族是那些基于四个控制顶点的。图 10.13(a)显示了如何对点 P_0、P_1、P_2 和 P_3 应用德卡斯特里奥算法。对于给定的 t 值,点 A 位于从 P_0 到 P_1 的线段上比例为分数 t 的位置上,点 B 和点 C 类似。然后点 D 位于从 A 到 B 线段上分数 t 处,E 点类似。最后,要求的点 P 位于从 D 到 E 线段上分

数 t 处。如果对0 到 1之间的每个 t 值都进行计算，就形成了从 P_0 开始，被 P_1 和 P_2 吸引，终止于 P_3 的曲线$P(t)$。这就是由四点确定的贝塞尔曲线。图 10.13(b)显示了不同位置的四点所确定的贝塞尔曲线。

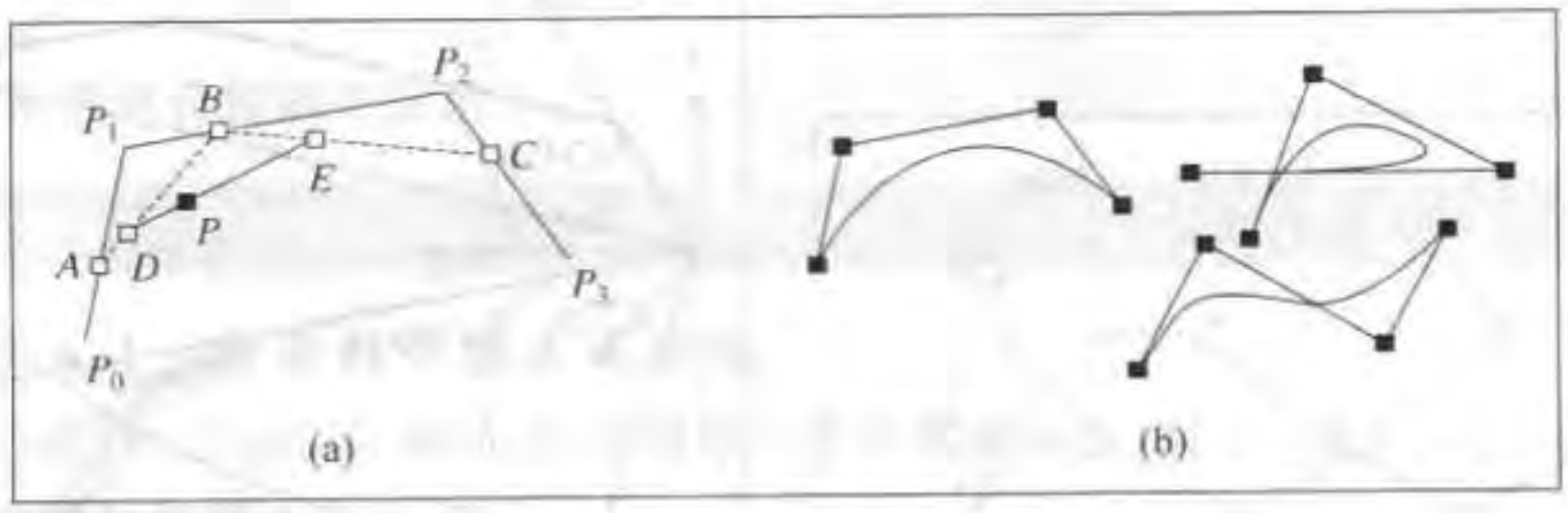


图 10.13　基于四点的贝塞尔曲线

很容易证明(见练习)基于四点的贝塞尔曲线有如下参数形式：

$$\mathbf{P}(t) = \mathbf{P}_0(1-t)^3 + \mathbf{P}_1 3(1-t)^2 t + \mathbf{P}_2 3(1-t)t^2 + \mathbf{P}_3 t^3 \tag{10.21}$$

它是 t 的三次多项式。从等式(10.21)注意到 $P(t)$包含了四个控制顶点，每项都被一个三次多项式加权，然后把加权之后的项相加。这里所包含的项都被看作 **Bernstein 多项式**。这四个三次 Bernstein 多项等式(它们的阶是多少？它们的系数是多少？)是：

$$\begin{aligned} B_0^3(t) &= (1-t)^3 \\ B_1^3(t) &= 3(1-t)^2 t \\ B_2^3(t) &= 3(1-t)t^2 \\ B_3^3(t) &= t^3 \end{aligned} \tag{10.22}$$

三次 Bernstein 多项式很容易被记为展开表达式 $a(t)=(1-t+t)$到三阶所得到的项，这个表达式显然对所有的 t 值都简单地为 1。现在展开 $a^3(t)$然后按照$(1-t)$和 t 的各次幂合并得到的项。

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

这立即产生 Bernstein 多项式的一个重要性质：对于任意的 t 值这些多项式加起来和为 1。(为什么？)

$$\sum_{k=0}^{3} B_k^3(t) = 1 \tag{10.23}$$

为此等式(10.21)中 $P(t)$显然是点的仿射组合，所以把加权后的点加起来是合理的。(回顾第 5 章中添加点的问题。)

图 10.14 画出了 t 从 0 到 1 变化时四个三次 Bernstein 多项式的形状。可以看到这些多项式随 t 光滑地起伏。接下来我们将看到它们有多么“光滑”。

图 10.15 从几何上解释了等式(10.21)中点 $P_0,\cdots,P_3$ 是如何被融合在一起形成 $P(t)$。把这些点看作是从原点出发的向量(因此我们把 P_0 写为 $\mathbf{p}_0$，其他类似)且令 $t=0.3$。则等式(10.21)变为

$$\mathbf{p}(0.3) = 0.343\mathbf{p}_0 + 0.441\mathbf{p}_1 + 0.189\mathbf{p}_2 + 0.027\mathbf{p}_3$$

图中四个向量被加权且加权结果用平行四边形法则加起来形成向量 $\mathbf{p}(0.3)$。可以看到，在 $t=0.3$ 处做主要贡献的是 $\mathbf{p}_0$ 和 $\mathbf{p}_1$。虽然第三项和第四项的权值很小，但是这些项对 $\mathbf{p}(0.3)$依然有影响因为向量 $\mathbf{p}_2$ 和 $\mathbf{p}_3$ 相对较大。

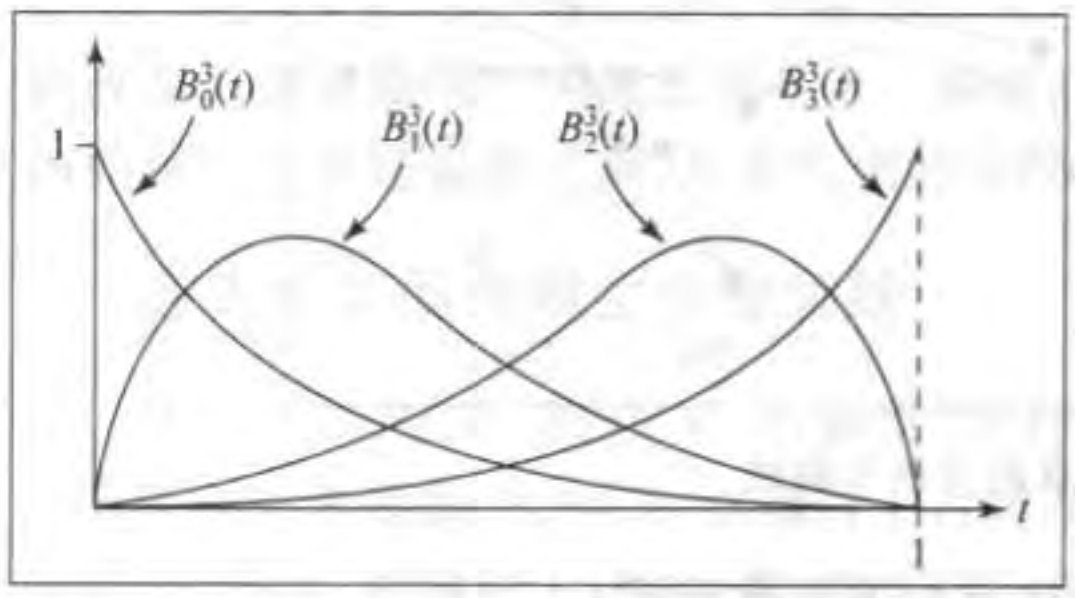


图 10.14 三次 Bernstein 多项式

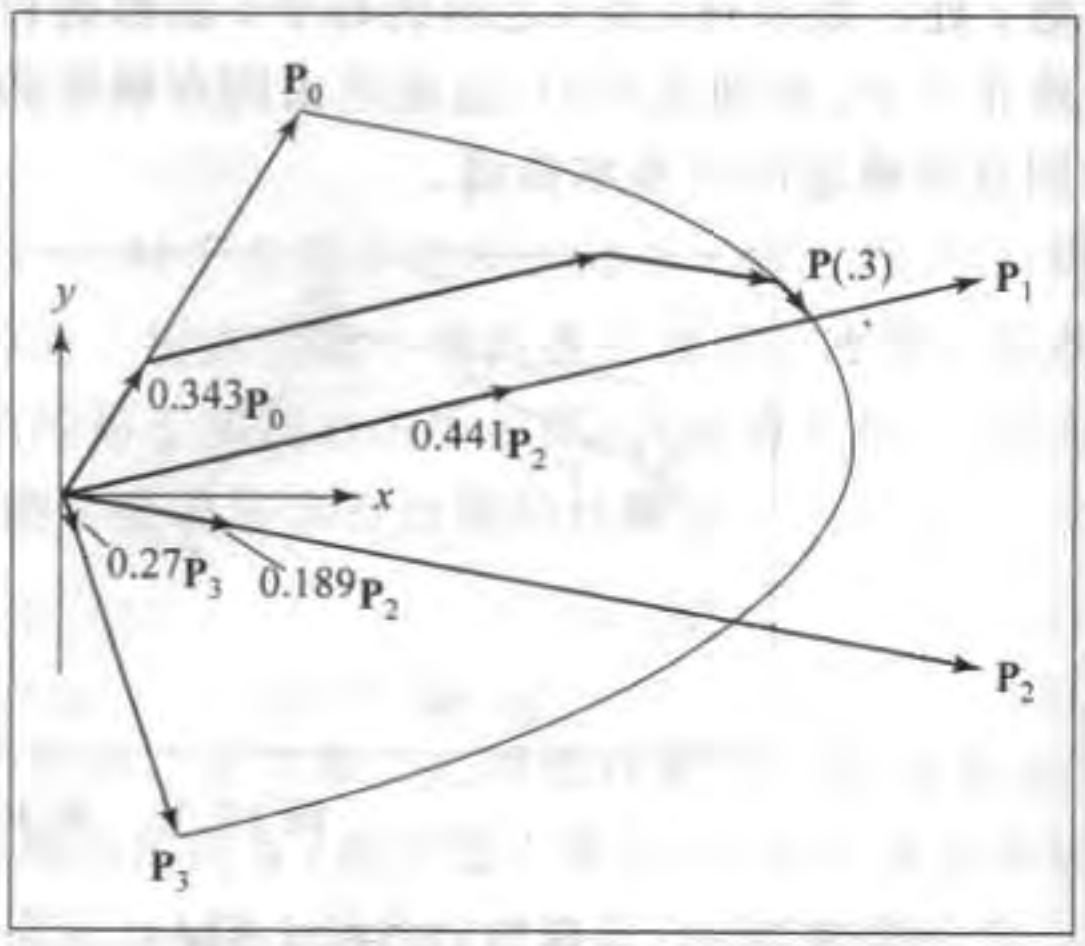


图 10.15 用 Bernstein 多项式混合四个向量

随着 t 的变化,这四个向量的权值也发生变化,且 $\mathbf{p}(t)$转移到不同的位置。试着看出在 $t=0.5$时各个向量的权重以及在形成向量 $\mathbf{p}(0.5)$时各个向量的贡献。

把德卡斯特里奥算法扩展到任意多个点

我们已经看到德卡斯特里奥算法利用内插,当使用三个点时生成二次参数表达式,当使用四个点时生成三次参数表达式。这个算法可以自然地推广到使用 $L+1$ 个控制顶点 P_0, $P_1,\cdots,P_L$的情形。对于每个 t 值,可以连续类似计算多次,每次计算都是通过插值前一次计算生成的相邻点得到:对于 $i=0,\cdots,L$ 有

$$\mathbf{P}_i^4(t) = (1-t)\mathbf{P}_i^3(t) + t\mathbf{P}_{i+1}^3(t)$$

$$\cdots$$

$$\mathbf{P}_i^L(t) = (1-t)\mathbf{P}_i^{L-1}(t) + t\mathbf{P}_{i+1}^{L-1}(t) \tag{10.24}$$

$\mathbf{P}_i^k(t)$中上标 k 表示第 k 次计算。这个过程以 $\mathbf{P}_i^0(t)=P_i$ 开始,结束于最后的贝塞尔曲线 $P(t)=P_i^L(t)$。得到的贝塞尔曲线可以用 Bernstein 多项式表示为

$$P(t) = \sum_{k=0}^{L} P_k B_k^L(t) \tag{10.25}$$

其中第 k 个 L 次 Bernstein 多项式被定义为[①]

$$B_k^L(t) = \binom{L}{k}(1-t)^{L-k}t^k \tag{10.26}$$

这里$\binom{L}{k}$是**二次项系数函数**,由下面给出

$$\binom{L}{k} = \frac{L!}{k!(L-k)!} \quad 对\ L \leqslant k \tag{10.27}$$

① 熟悉概率论的读者会注意到这种形式与二次项概率分布非常相似。

如果 $L<k$，这项值被定义为 0。每个 Bernstein 多项式都被看作是 L 次的。如以前，Bernstein 多项式是展开$[(1-t)+t]^L$ 所得到的各项，因此我们确保

$$\sum_{k=0}^{L} B_k^L(t) = 1 \quad \text{对所有的 } t \tag{10.28}$$

且 $P(t)$是各点合理的仿射组合。

练习

练习 10.4.1　用图形计算生成曲线

对于三个点(0,0)、(2,4)和(6,1)应用德卡斯特里奥算法，对于 t 值 0、0.2、0.4、0.6、0.8 和 1 分别在绘图计算器上画出 $P(t)$。

练习 10.4.2　二次曲线必是平面曲线

证明过三个(不同的)点存在唯一的一条二次曲线。回想三个(非共线的)点确定一个平面，证明它们所确定的二次曲线不会偏离这个平面。因此抛物线永远不会偏离一个平面。作为额外的收获，证明三次曲线可能是非平面的。

练习 10.4.3　由 Tween()表示的贝塞尔曲线

回顾 4.5.3 节中的函数 Tween()。它用两个点作为参数，且对于给定的 t 值返回这些点的内插值。证明等式(10.20)中 $P(t)$可以被写为 Tween(Tween(P0,P1,t), Tween(P1,P2,t),t)，或者两个内插值的内插值。写出三次贝塞尔曲线的类似形式。

练习 10.4.4　三次贝塞尔曲线

通过写出类似于前一个练习中的连续的内插值，证明等式(10.21)给出的贝塞尔曲线的参数形式是正确的。

练习 10.4.5　建立贝塞尔曲线的直观图

利用等式(10.21)，当四个控制顶点为(2, 3)、(6, 6)、(8, 1)和(4, −3)时在时间 $t=0.2,0.5,0.9$计算 $P(t)$的位置。如图 10.15 所示，画出对最终的 $P(t)$值做贡献的加权后的向量的草图。

练习 10.4.6　四次和五次贝塞尔曲线

写出 $L=4$ 情形下等式(10.26)中的 Bernstein 函数。证明这些函数是展开$[(1-t)+t]^4$ 所得到的各项。对 $L=5$ 重复这个过程。

练习 10.4.7　Bernstein 多项式的递推关系

证明 n 阶 Bernstein 多项式总可以由 $n-1$ 阶 Bernstein 多项式利用下式推出

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \tag{10.29}$$

其中 $B_0^0(t)=1$，且当 j 不在 $0,\cdots,n$ 的范围内时 $B_j^n(t)=0$。

提示： $\begin{pmatrix} n \\ i \end{pmatrix} = \begin{pmatrix} n-1 \\ i \end{pmatrix} + \begin{pmatrix} n-1 \\ i-1 \end{pmatrix}$.

练习 10.4.8 贝塞尔曲线插值两端点

证明一般的 L 次贝塞尔曲线通过两个最外面的控制顶点：$P(0)=P_0$ 和 $P(1)=P_L$。通过在 $t=0$ 和 $t=1$ 处计算 Bernstein 多项式的值来证明，并证明在这两个端点处除一项外其他所有项都为 0。

10.5 贝塞尔曲线的性质

贝塞尔曲线具有一些重要的性质使其很适合于 CAD。我们将在后面看到这些性质也适合于 B 样条。学习这些性质及其证明提供了对贝塞尔曲线更深入的理解。

端点插值

基于控制顶点 $P_0,P_1,\cdots,P_L$ 的贝塞尔曲线 $P(t)$ 通常并不通过或插值所有控制顶点。但是我们已经看到，它总是插值 P_0 和 P_L。这是一个很重要的性质，因为输入一系列点的设计者精确地知道贝塞尔曲线从哪里开始到哪里终止。

仿射不变性

为了缩放贝塞尔曲线，改变其方向或其位置以备后用，需要其满足一个仿射变换。假定我们希望利用仿射变换 T(T 在 $2D$ 情况下用 3×3 矩阵表示，在 $3D$ 情况下用4×4矩阵表示)把等式(10.25)中贝塞尔曲线上点 $P(t)$变换到新点 $Q(t)$。因此 $Q(t)=T(P(t))$。看起来对于任意给定的 t 值为了计算 $Q(t)$我们必须先计算 $P(t)$，然后对它进行变换，对于每个新的 t 值重复这个计算过程。实际上并不需要这样。我们只需变换控制顶点(一次)，对任意 t 在同样的 Bernstein 形式中用这些新的控制顶点来重新生成变换后的贝塞尔曲线！即：

$$Q(t)=\sum_{k=0}^{L}T(P_k)B_k^L(t) \qquad (10.30)$$

仿射不变性就是变换后的曲线与基于变换后的控制顶点的曲线是相同的。

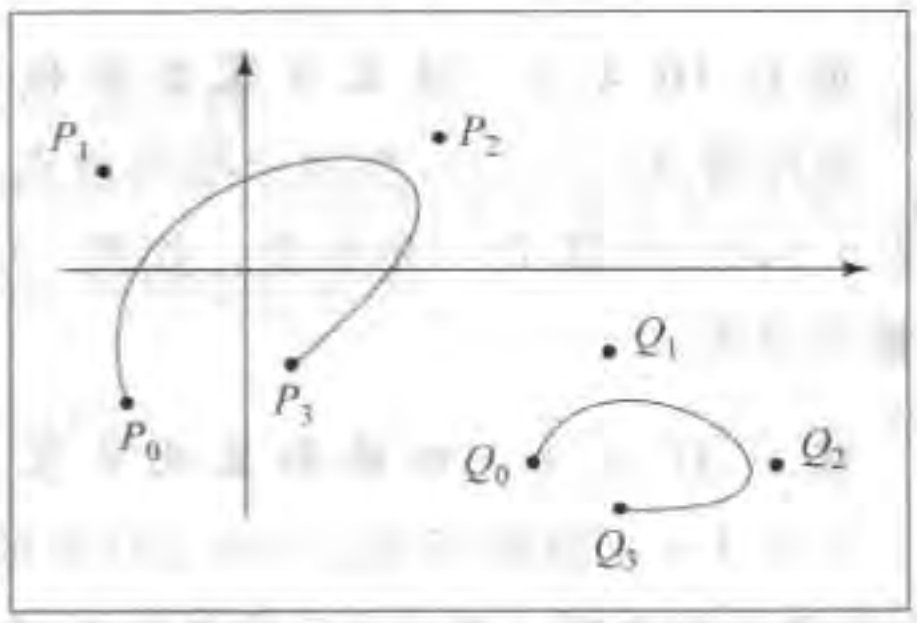


图 10.16 显示仿射不变性

图 10.16 显示了一条基于四个控制顶点 $P_0,\cdots,P_3$的贝塞尔曲线。这些点被旋转，伸缩和平移到新的控制顶点 Q_k，且由它们确定的贝塞尔曲线也画出来了。这条曲线与变换原始的贝塞尔曲线所得的结果相同。

贝塞尔曲线具有仿射不变性的原因很简单：它们是由点的仿射组合形成，且从 5.2 节中我们知道仿射变换保持仿射组合。

凸包性

设计者常常依赖的另一个性质是贝塞尔曲线，$P(t)$，不会超出其凸包。回顾第 5 章中点集 $P_0,P_1,\cdots,P_L$ 的凸包是所有这些点的凸组合的集合，即由下式给出的所有点的集合

$$\sum_{k=0}^{L} \alpha_k P_k \tag{10.31}$$

其中每个 α_k 是非负的，且它们的和为 1。

等式(10.25)中的 $P(t)$ 对每个 t 都是它的控制顶点的凸组合，因为 Bernstein 多项式不会为负数且其和为 1。因此贝塞尔曲线上的每个点都是其控制顶点的凸组合，所以贝塞尔曲线必定位于其控制顶点的凸包内。

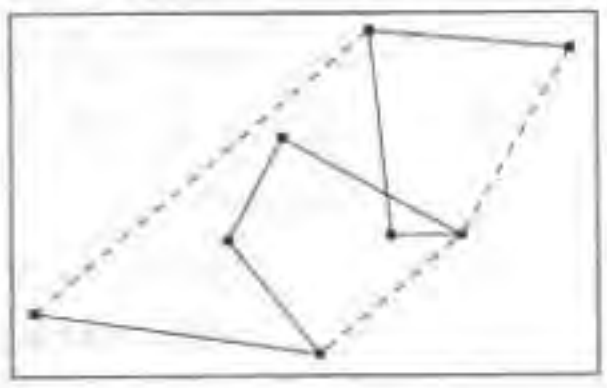


图 10.17　利用凸包性

凸包性也可由以下事实推出：曲线上的每个点都是两个内插点的内插值，且两个点的内插值形成它们的凸组合。图 10.17 阐明了设计者如何利用凸包性质。即使图中的 8 个控制顶点形成了一个锯齿状的控制多边形，设计者也知道贝塞尔曲线会在两个端点之间光滑移动，而且永远不会超出凸包的范围。

贝塞尔曲线的导数

因为当曲线关于 t 的导数有不连续点时曲线上会出现尖角或突变，所以我们需要研究等式(10.25)中 $P(t)$ 的各种导数。

对于贝塞尔曲线可以证明其一阶导数为

$$\mathbf{p}'(t) = L \sum_{k=0}^{L-1} \Delta P_k B_k^{L-1}(t) \tag{10.32}$$

其中

$$\Delta P_k = P_{k+1} - P_k \tag{10.33}$$

(见练习)速度曲线是建立在一组新的控制顶点 ΔP_k 上的另一条贝塞尔曲线。我们简单地对原始控制顶点两两进行差分来形成速度曲线的控制向量。从形式 $B_k^{L-1}(t)$ 注意到求导把曲线阶数降低一阶。例如三次贝塞尔曲线的导数是二次贝塞尔曲线。贝塞尔曲线的光滑性在练习中讨论。

生成并绘制贝塞尔曲线

假定我们希望建立一个应用程序使其通过一系列控制顶点绘制一条贝塞尔曲线。如何组织这样的应用程序呢？等式(10.25)中贝塞尔曲线 $P(t)$ 将用一条近似折线形来绘制。在密集分布的 t 值上对函数 $P(t)$ 采样－比如 $t_i = i/N$ 对于 $i = 0, 1, \cdots, N$－且用直线连接点 $P(t_i)$。唯一的问题是在每个要求的 t 值处计算 $P(t)$，我们假定用如下的函数来执行：

```
Point2 bezier(RealPointArray poly, double t);
/* 根据控制多边形计算贝塞尔曲线上点的位置 P(t) */
```

这个程序使用存储在 poly 中的点数组(称为控制多边形)对于给定的 t 值计算等式(10.25)的结果并返回所得到的点。注意 Bernstein 多项式的次数存储在 poly.num 中。程序的实现留到练习中。图 10.18 显示了一个完整的可执行程序其中用户可以指定附加的控制顶点。每组三个点指定后，另一条贝塞尔曲线就被画出来了。这个程序的深层扩展将在案例分析中讨论。

```
#include<windows.h>
#include<math.h>
#include<gl/Gl.h>
#include<gl/glut.h>

int SCREEN_HEIGHT=480;
int NUMPOINTS=0;
class Point {
public:
    float x, y;
    void setxy(float x2, float y2) { x=x2; y=y2;}
    cont Point & operator (const Point &rPoint) {
        x=rPoint.x;
        y=rPoint.y;
        return * this;
    }
};
Point abc[3];
void myInit() {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0, 0.0, 0.0);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIndentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
void drawDot(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}
void drawLine(Point p1, Point p2) {
    glBegin(GL_LINES);
    glVertex2f(p1.x, p1.y);
    glVertex2f(p2.x, p2.y);
    glEnd();
    glFlush();
}
Point drawBezier(Point A, Point B, Point C, double t) {
    Point P;
    P.x=pow((1-t), 2) * A.x+2 * t * (1-t) * B.x+pow (t, 2) * C.x;
    P.y=pow((1-t), 2) * A.y+2 * t * (1-t) * B.y+pow (t, 2) * C.y;
    return P;
}
```

图 10.18　一个完整的贝塞尔曲线应用程序练习

```
void myMouse (int button, int state, int x, int y) {
    if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN) {
      abc[NUMPOINTS].setxy((float) x, (float)(SCREEN_HEIGHT-y));
      NUMPOINTS++;
      drawDot(x, SCREEN_HEIGHT-y);
      if(NUMPOINTS==3) {
          glColor3f(1.0, 0.0, 1.0);
          drawLine(abc[0], abc[1]);
          drawLine(abc[1], abc[2]);
          Point POld=abc[0];
          for (double t=0.0; t<=1.0; t+=0.1){
              Point P=drawBezier(abc[0], abc[1], abc[2], t);
              drawLinw(POld, P);
              POld=P;
          }
          glColor3f(1.0, 0.0, 0.0);
          NUMPOINTS=0;
      }
    }
}
void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

int main(int argc, char * agrv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | CLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("Bezier Curve");
    glutMouseFunx(myMouse);
    glutDisplayFunc(myDisplay);
    myInit();
    glutMainLoop();
        return 0;
}
```

图 10.18（续）

练习

练习 10.5.1　画出速度

在绘图计算器上或绘图纸上画 4 个点，画出对应的贝塞尔曲线，仔细的计算它的速度和加速度向量关于 t 的函数。

10.6 寻找更好的混合函数

似乎贝塞尔曲线为曲线设计提供了终极工具。通过在平面上适当地放置控制顶点就能够生成各种各样的光滑曲线。

但在后面将看到,在曲线设计中贝塞尔曲线自身并未提供足够的灵活性。一个问题是所用的 Bernstein 多项式的次数与控制顶点的个数相关:基于 $L+1$ 个控制顶点的贝塞尔曲线是 L 次多项式的组合。高次多项式计算起来代价很高,且易有数值舍位误差。我们希望设计者可以随意选取任意多个控制顶点——甚至 40 个或更多。

10.6.1 局部控制问题

一个更加重要的问题是贝塞尔曲线并未提供足够的对曲线形状的局部控制。图 10.19 给出了使用五个控制顶点生成一条贝塞尔曲线(实线)的情形,其在 $t=1$ 附近偏离了所期望的曲线(虚线)。

为了纠正这个偏离,用户需要向上移动 P_2 和 P_3 以使贝塞尔曲线更接近所期望的曲线。但是,如图 10.19 所示,这也影响了曲线的前半部分,迫使它偏离所期望的曲线。

问题是任意一个控制顶点的改动都会改变整个曲线。这起因于 Bernstein 多项式的本质(回顾图 10.14):在整个区间[0,1]上每个都是"活动"(意思是不为 0)的。函数值不为 0 的区间通常叫做它的**支撑**。因为每个 Bernstein 多项式在整个区间[0,1]上都有支撑,且曲线是这些函数的混合,所以每个控制顶点对 0 到 1 之间 t 值处的曲线都有影响。因此调整任何控制顶点都会影响曲线各处的形状,缺少局部控制。

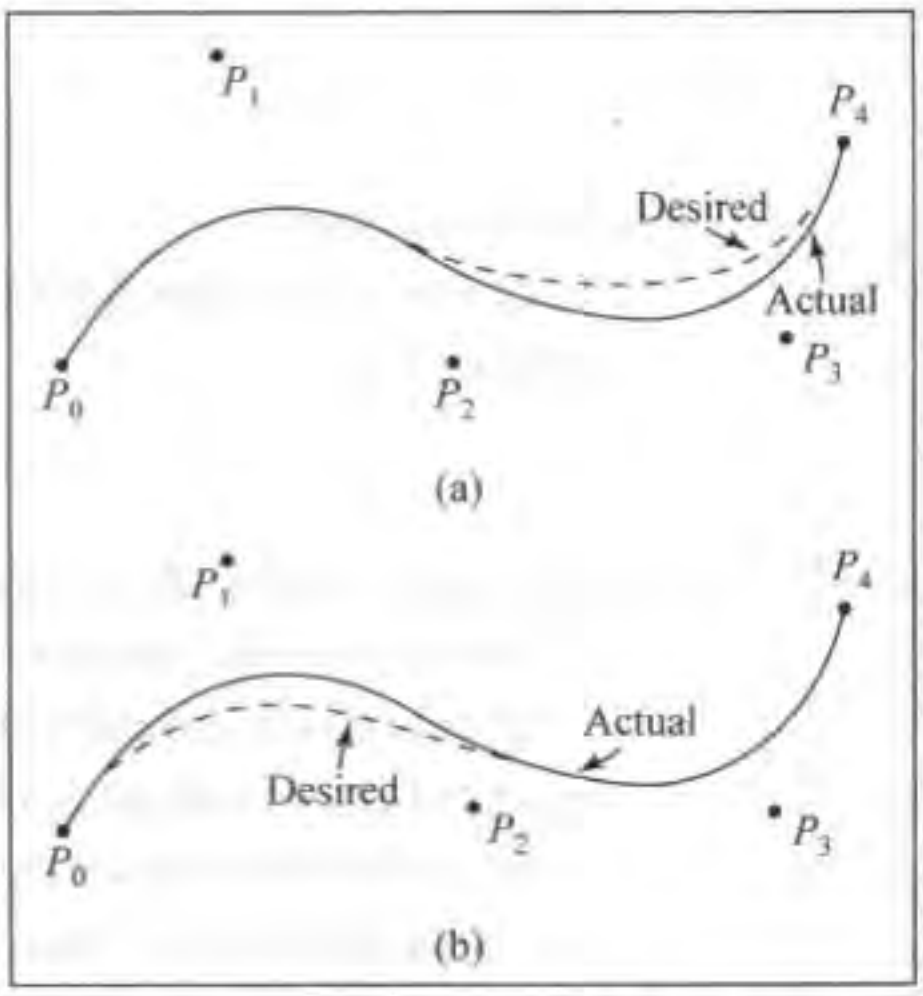


图 10.19 编辑曲线的某些部分

把上述情况与图 10.20 中给出的一组有用的猜想的混合函数相比较。图中显示了六个混合函数,$R_0(t), R_1(t), \cdots, R_5(t)$(后面将定义①),每个函数只在区间[0,1]的一部分上有支撑。例如,$R_0(t)$的支撑是[0,.25],而 $R_3(t)$的支撑是[.25,1]。事实上,对于任意 t 值这些混合函数中活动的不会超过 3 个。

考虑使用这些混合函数,基于六个控制顶点 $P_0, P_1, \cdots, P_5$ 建立一条曲线 $V(t)$。我们使用与贝塞尔曲线相同类型的参数形式:

$$V(t) = \sum_{k=0}^{5} P_k R_k(t) \tag{10.35}$$

① (预览):这些实际上是后面要研究的二次 B 样条。

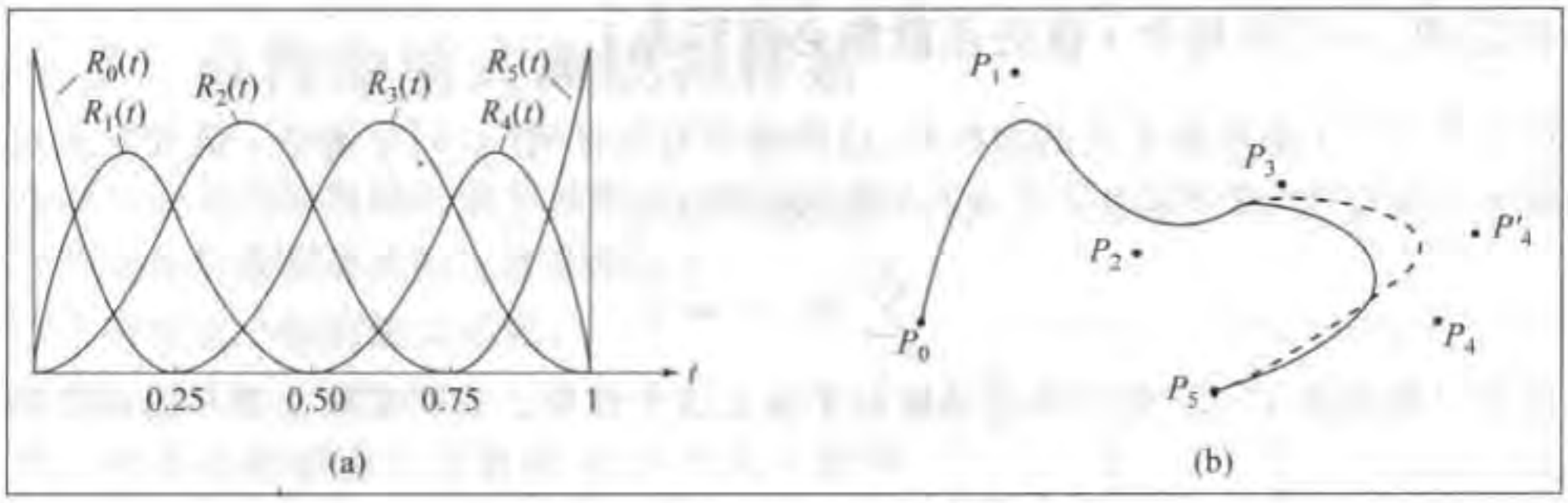


图 10.20　具有集中支撑集的混合函数

(a) 比较有希望的具有集中支撑集的猜想的混合函数集；

(b) 一个控制顶点调整前(实线)后(虚线)的示例曲线

但寻找了一组较好的函数 $R_k(t)$。图 10.20(b)显示了基于一组示例控制顶点的曲线。在每个 t 值处位置 $V(t)$所依赖的控制顶点不会超过 3 个。特别地，在$[0.75,1]$中的 t 值处只有点 P_3，P_4，和 P_5 控制曲线的形状。如果单个控制顶点 P_4 被移动到 P'_4，只有曲线中用虚线表示的那部分会发生变化。因此这组混合函数确实使得控制顶点具有某些局部控制力。

10.6.2　混合函数集的性质

图 10.20 中混合函数的形状经过了调整只是为了说明局部控制的性质。但是它们事实上是基于曲线设计中常用的函数。它们的详细性质将在后面描述，但是我们在这里对它们进行深入研究是为了对混合函数集应该具有的性质有个印象。

和往常一样，我们假定要使用的曲线生成算法依照下式混合控制顶点 $P_0,\cdots,P_L$：

$$V(t)=\sum_{k=0}^{L}P_kR_k(t),\quad t\text{ 在}[a,b]\text{ 区间内} \tag{10.36}$$

其中混合函数 $R_0(t),\cdots,R_L(t)$具有某些性质使其能够生成更好的曲线或者能使设计过程更加直观。混合函数应该具有如下的这些性质：

- 易于计算且数值稳定；
- 在区间$[a,b]$中每个 t 值处和为 1；
- 具有小的支撑集来提供局部控制；
- 插值由设计者选定的某些控制顶点；
- 足够光滑。

我们分别考虑各个性质。

函数集应该易于计算且数值稳定

为了快速生成曲线我们希望混合函数计算起来很简单。我们还希望它们受数值舍位误差的影响最小。这些考虑引导我们选择多项式作为混合函数，选择次数尽可能低且能连续生成所需要的曲线的多项式。其他类型的函数，例如正弦和余弦函数，其代价过高而不能使用。

在区间$[a,b]$中每个 t 值处函数集必须和为 1

$V(t)$在每个 t 值处都是点的加权和,这使得只有在区间$[a,b]$中每个 t 值处它是点的仿射和时才有意义(否则它不是一个点)。因此我们坚持

$$\sum_{k=0}^{L} R_k(t) = 1 \tag{10.37}$$

你可以看到图 10.20 中的候选函数似乎满足这个性质。我们实际上要用的函数确实和为 1。

函数集应该具有小支撑集用于局部控制

之前我们已经讨论过:为了达到局部控制我们希望每个混合函数集中在某些 t 值上,只在区间$[a,b]$的一小部分上有支撑。

函数集应该插值某些控制顶点

设计者可能希望 $V(t)$通过一些控制顶点而只被另外一些控制顶点所吸引。图 10.20 中的形状只插值了第一个和最后一个控制顶点。这是显然的因为在 $t=0$ 和 $t=1$ 处除了一个混合函数其他的都为 0,迫使剩下的那个为 1(为什么)使得能够插值。(为什么?)我们将在不久后看到怎样提供一个机制来调整混合函数使得另外的一些控制顶点被插值。

函数集应该有足够的光滑性

设计者通常希望对于任意控制顶点集 $V(t)$都是一条光滑曲线。典型地,为了生成所期望的曲线形状的光滑性 $V(t)$至少应该是一阶光滑的,可能甚至是二阶光滑的。$V(t)$的光滑性依赖于混合函数的光滑性。特别地,如果每个混合函数在$[a,b]$上是一阶光滑的,则 $V(t)$在$[a,b]$上也是一阶光滑的。

注意图 10.20 中的混合函数似乎在其内部是光滑的。起始和结束的平稳也是很重要的。图 10.21显示了一个混合函数及其一阶导数。

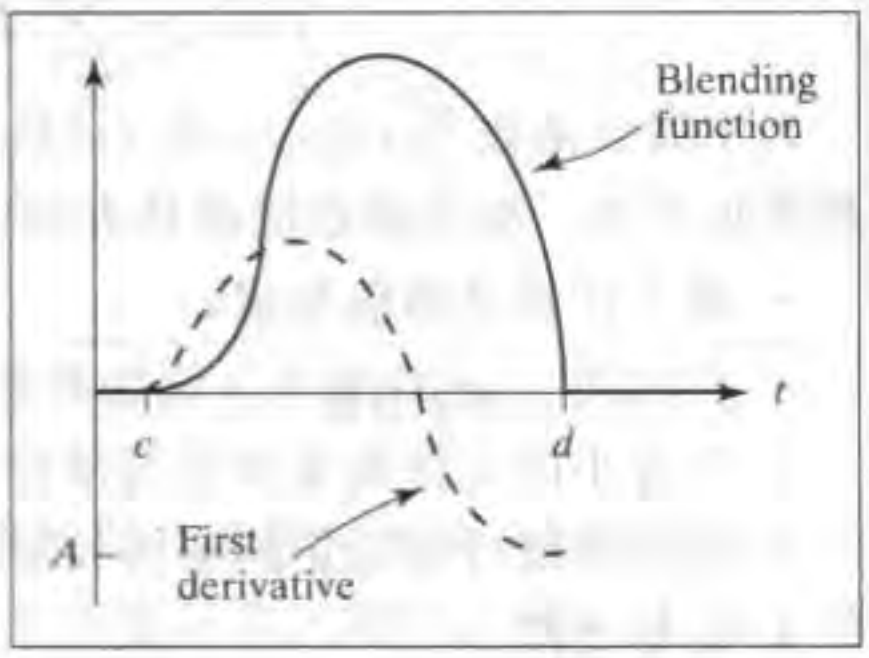


图 10.21 一个候选混合函数及其导数

函数的导数在起始处 $t=c$ 从 0 连续变化。但它的导数在结束处 d 不连续,从一个负数突变到 0。这个函数在除了 $t=d$ 处处是一阶光滑的。使用这个混合函数的曲线在 $t=d$ 不会是一阶光滑的。因此要求混合函数在其内部光滑,且在起始和终止处导数为 0。

图 10.20 中某些形状在起始和终止处导数为 0,而其他的不是这样。那些起始点和终止点在区间[0,1]的形状满足起始和终止点处导数为 0,所以 $V(t)$在区间(0,1)内是一阶光滑的。在区间[0,1]的端点处导数不连续是可以接受的因为我们永远不会使用超过 0 和 1 的 t 值。

10.6.3　分段多项式曲线和样条

我们开始寻找好的候选混合函数，比如低次多项式。例如，是否存在具有图 10.22 所示形状且满足我们所有要求的三次多项式？

为了研究这个我们定义函数：

$$R(t) = at^3 + bt^2 + ct + d$$

然后看是否能够选到参数使 $R(t)$ 及其一阶导数在 $t=0$ 和 $t=1$ 处都为 0。这导出关于系数的四个条件：

$$R(0) = d = 0$$
$$R(1) = a + b + c + d = 0$$
$$R'(0) = c = 0$$
$$R'(1) = 3a + 2b + c = 0$$

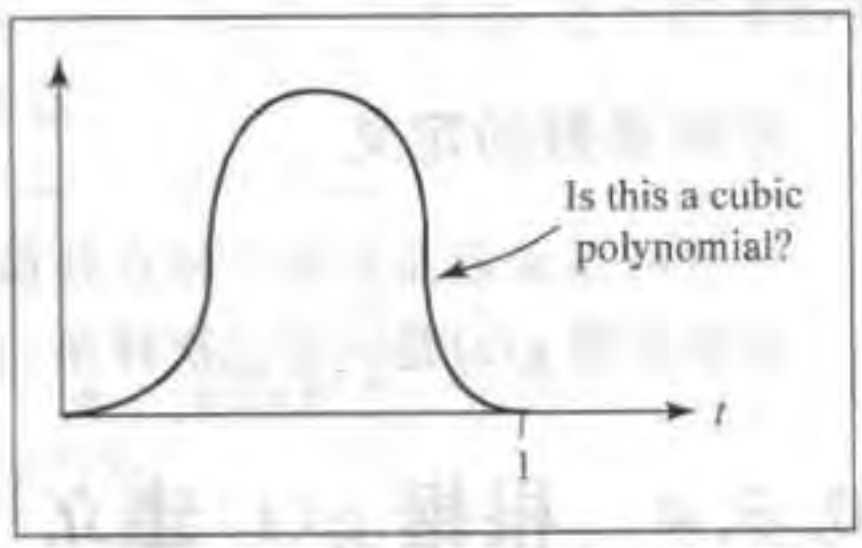


图 10.22　能否找到这样的三次多项式

不幸的是，这些条件使得 $a=b=c=d=0$，所以不存在这样的形状。也就是说三次曲线不具有足够的灵活性来实现我们所需要的混合（练习中会问四次多项式是否满足这个要求）。

为了获得更多的灵活性我们试着把几个低次多项式分段连接起来。这样在不同的 t 分段区间上由不同的多项式定义的曲线称为**分段多项式**。

图 10.23 给出的示例形状 $g(t)$ 帮助建立几个术语。我们看到 $g(t)$ 由三个多项式**段**，$a(t)$、$b(t)$ 和 $c(t)$ 组成，如下定义

$$a(t) = \frac{1}{2}t^2$$

$$b(t) = \frac{3}{4} - \left(t - \frac{3}{2}\right)^2$$

$$c(t) = \frac{1}{2}(3-t)^2 \tag{10.38}$$

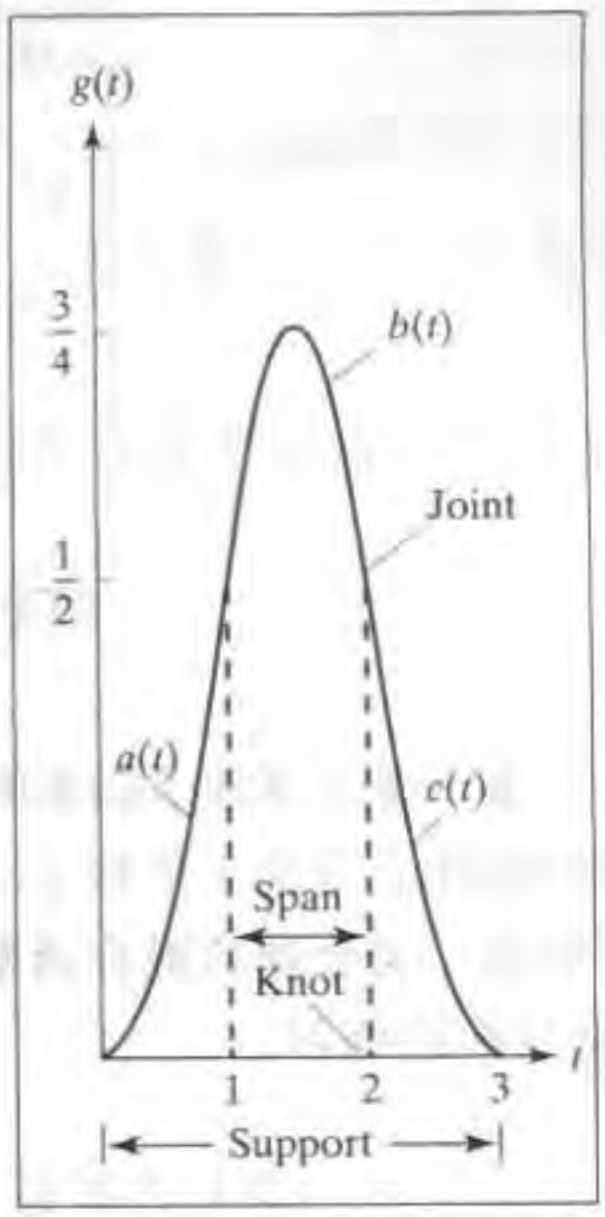


图 10.23　分段多项式成分

$g(t)$ 的支撑区间是 $[0,3]$；$a(t)$ 定义在区间 $[0,1]$，$b(t)$ 定义在区间 $[1,2]$，$c(t)$ 定义在区间 $[2,3]$。一对分段多项式相遇的点叫做**连接点**，发生相遇时的 t 值叫做**结点**。本例中有四个结点：0、1、2 和 3。

再一次强调：

- 结点是分段相遇时的 t 值；
- 连接点是相邻的分段函数值。

$g(t)$ 在其支撑区间上是否处处连续呢？因为它是由多项式定义的，在每个区间内部它肯定是连续的，因此我们只需检查这些分段在连接点的情况。从等式(10.37)中这很容易检查：$a(1)=b(1)=1/2$ 和 $b(2)=c(2)=1/2$。

进一步，$g(t)$ 的导数是处处连续的，所以 $g(t)$ 在 $[0,3]$ 上是 1-光滑的。为此，注意到导

数在每段区间内部必定是连续的(为什么?),所以只需在结点处检查它的连续性。直接计算表明 $a'(1)=b'(1)=1$ 和 $b'(2)=c'(2)=-1$。因此当我们从一个多项式段移到下一个时,斜率不会发生突变。

二阶导数是不连续的,但是在结点(哪个?)处的两个值是否发生跳跃?

这里的形状 $g(t)$ 是**样条函数**的一个例子,分段多项式函数具有足够的光滑性。特别地,我们有

样条函数的定义

M 次样条函数是在每个结点处都是 $M-1$ 阶光滑的 M 次分段多项式。

显然示例 $g(t)$ 是一个二次样条:它是一个二次分段多项式且有处处连续的一阶导数。

10.6.4 根据 $g(t)$ 建立一组混合函数

如何将上述的样条函数 $g(t)$ 作为混合函数来表示曲线呢?一种方法是使用平移后的 $g(t)$,其中每个混合函数 $g_k(t)$ 是用某个数量平移基本形状 $g(t)$ 形成的。图 10.24 显示了由将 $g(\cdot)$ 平移一个整数位置所形成的 7 个混合函数 $g_0(t),\cdots,g_6(t)$:

$$g_k(t) = g(t-k), \quad 其中\ k = 0,1,\cdots \tag{10.39}$$

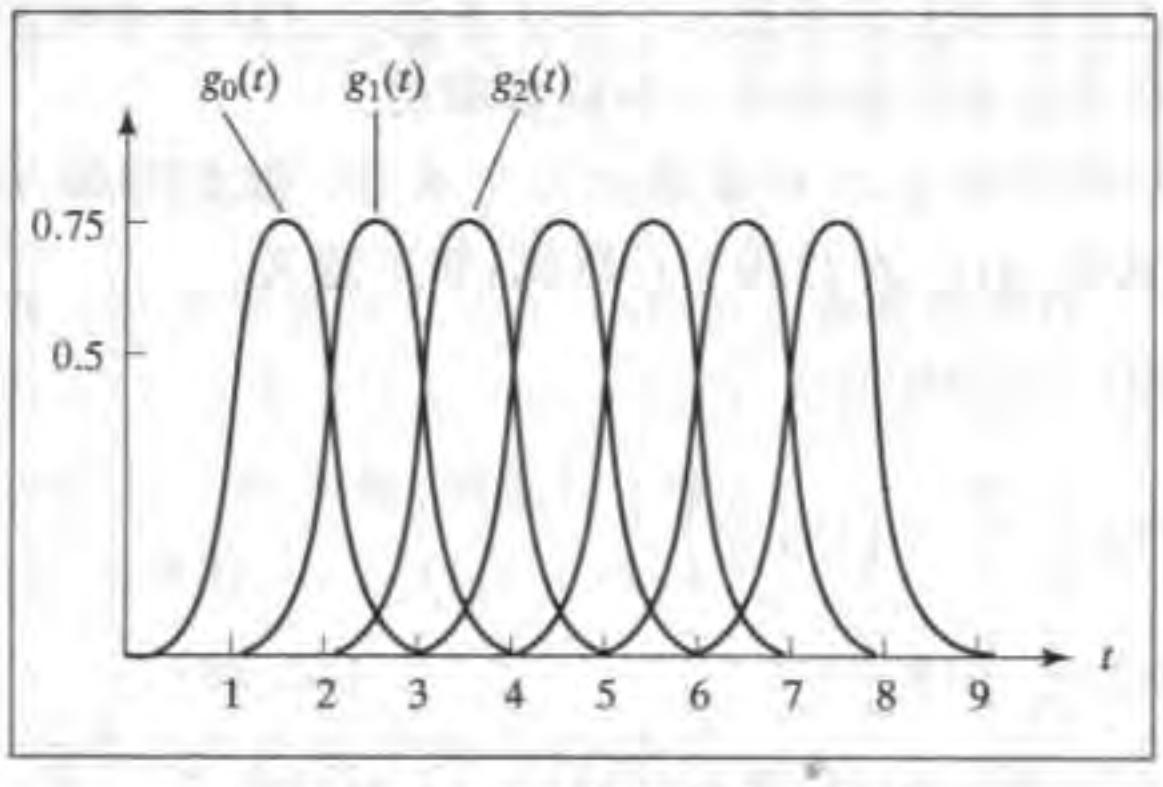


图 10.24 表示样条曲线的混合函数

由于 $g(\cdot)$ 各个版本的结点都出现在整数位置,所以平移曲线使其结点叠加在下个曲线的结点上等价于平移 $g(\cdot)$。只有当这些平移后的版本在任意 t 值处和为 1 时,它们才能形成一组合理的混合函数。然而对于 2~7 之间的所有 t 值这些函数确实满足这一点!(见后面的练习。)

$$\sum_{k=0}^{6} g(t-k) = 1 \quad t\ 在[2,7]内 \tag{10.40}$$

这似乎看起来很神奇,但是我们将看到这是我们最终得到的混合函数的一般属性。这里的关键是把每个函数平移一个整数,适当地把这些形状连接起来使其和为 1。

设计者选定七个控制顶点且利用如下算法生成曲线:

$$V(t) = \sum_{k=0}^{6} P_k g(t-k) \tag{10.41}$$

只能用 2 和 7 之间的 t 值(为什么?)。在这个范围内,注意到在任意 t 值处恰有三个混合函数是活动的,所以对曲线形状有很好的局部控制。另外注意到在特殊时刻 $t=2$, 3,…,7处只有两个函数是活动的且其值都为 0.5。因此在这些 t 值处 $V(t)$将位于其中两个控制顶点所形成线段的中点上。

图 10.25 给出了七个控制顶点和所得到的曲线的一个示例。曲线在 $t=2$ 处从 P_0P_1 的中点起始,且光滑地波动依次通过控制多边形各个边的中点。随着 t 的增加,各个混合函数起起落落,且对曲线的主要影响从一个点传递到另一个点。

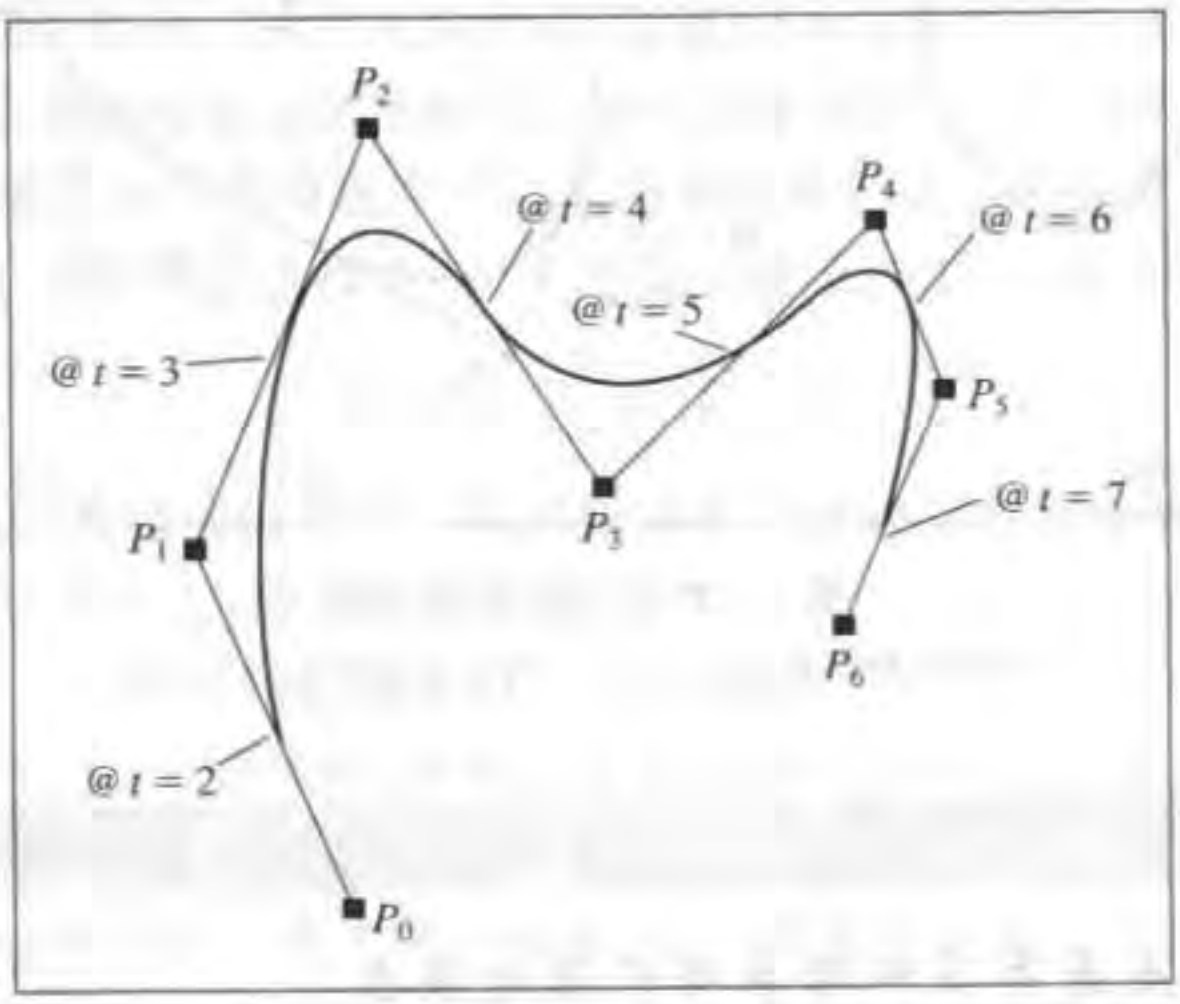


图 10.25　利用 $g(\cdot)$的平移设计曲线

基于这组混合函数的曲线具有哪些性质?

- 设计者对曲线形状具有很好的局部控制,因为每个混合函数的支撑区间的长度都不超过 3。
- 设计者放置控制顶点时可以利用曲线必定通过控制多边形各边中点的这一特点。因此这个算法具有某些直观的几何性质。
- 因为每个混合函数都是 1-光滑的,所以整个曲线是 1-光滑的。
- 除了 P_0 和 P_6 曲线上任何控制顶点都不会被插值。
- 所有多项式都是二次的,所以它们计算起来既快速又稳定。多项式的次数不依赖于控制顶点的个数。这种方法适用于任意多个控制顶点。

刚描述的曲线生成算法可以用下面的程序很容易地实现

```
Point2 curvePt(double t, RealPointArray pts)
```

对于每个输入的 t 值返回点 $V(t)$。这个将在案例 10.4 中更加深入讨论。

例 10.6.1　扩展到绘制封闭曲线

不难扩展这种方法使其能够生成如图 10.26(a)所示的封闭曲线。图 10.26(b)显示了图 10.25 中曲线的封闭版本。为了实现这个目标,你所需做的就是通过复制控制顶点 P_0 和 P_1,从而增加了两个额外的控制顶点,使得方程(10.41)的和式中添加两项。我们知道在

$t=7$ 前面的曲线通过线段 P_5P_6 的中点。因此在 $t=8$ 时曲线通过 P_6P_7 的中点,且在 $t=9$ 时曲线通过 P_7P_8 的中点,于是封闭了曲线。另一种方法在练习中给出。

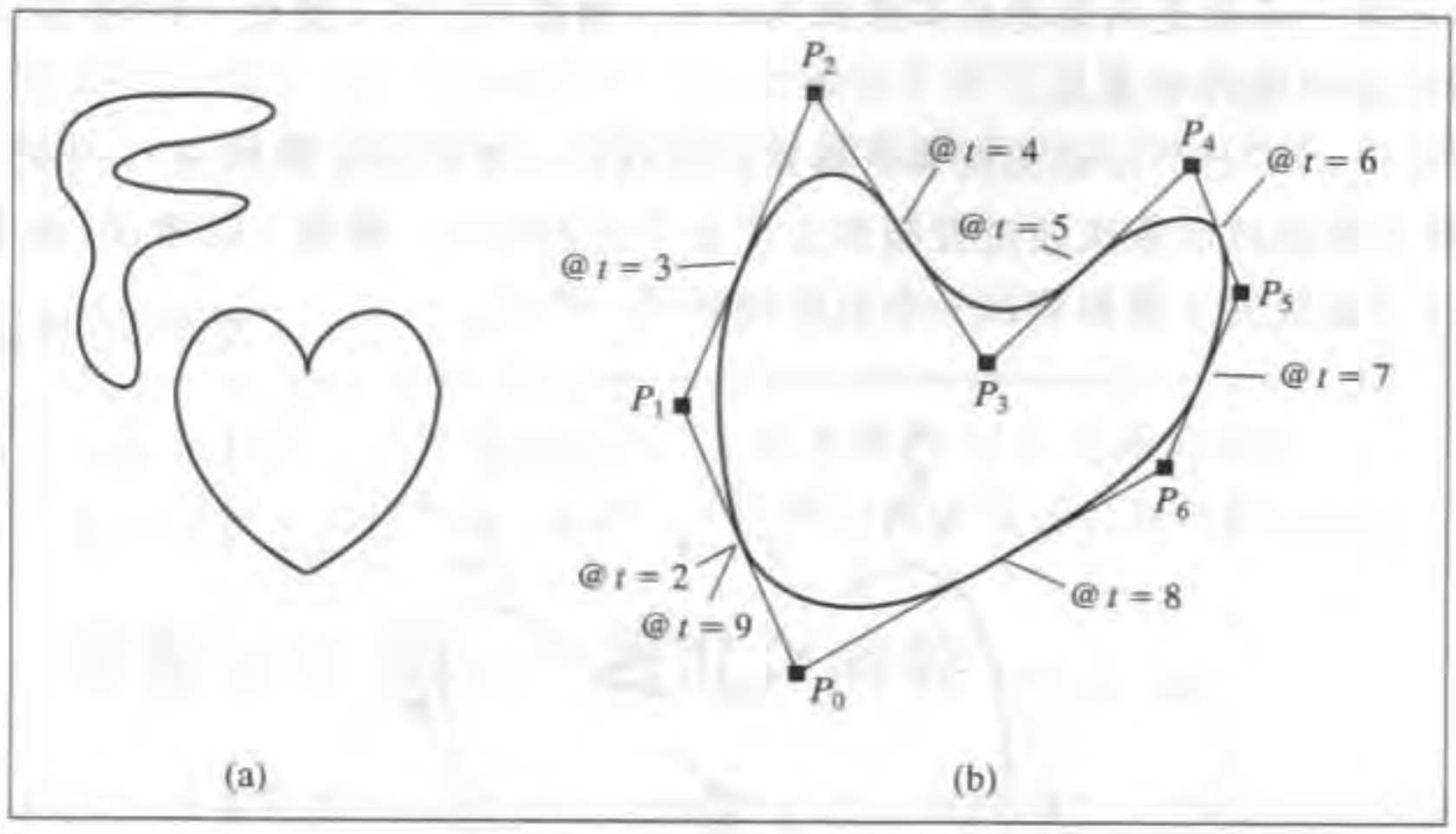


图 10.26 生成封闭曲线

(a) 封闭曲线的例子;(b) 一种生成封闭曲线的方法

练习

练习 10.6.1 是否存在四次多项式混合函数?

考虑四次多项式 $r(t)=at^4+bt^3+ct^2+dt+e$。是否存在这样的 a、b、c、d 和 e 使得函数通过 0 且其一阶导数在 $t=0$ 和 $t=1$ 处都等于 0? 如果存在,找到合适的值且画出曲线的草图。

练习 10.6.2 证明 g(·)的平移版本在任意 t 值处和为 1

考虑平移的和 $g(t)+g(t-1)+g(t-2)$,其中 $g(t)$是等式(10.38)中定义的分段多项式。证明对于 2 和 3 之间的任意 t 值这个和总为 1。

练习 10.6.3 Bernstein 多项式是样条吗?

证明 Bernstein 多项式确实是样条。形成每个多项式需要多少个分段多项式? 结点位于何处? 这个样条的次数是多少? 是否足够连续可微?

练习 10.6.4 试验二次样条曲线的直观性

以某种复杂的模式画出 12 个控制顶点,然后画出由等式(10.38)的二次样条函数生成的曲线的粗略草图。利用曲线通过控制多边形各边中点这个事实。

练习 10.6.5 没有额外控制顶点的情况下生成封闭曲线

证明对等式(10.41)做稍微变动

$$V(t)=\sum_{k=0}^{6}P_k g((t-k)\bmod 7) \tag{10.42}$$

随着 t 从 0 到 7 变化时能够生成基于七个(各不相同的)控制顶点的封闭曲线。不需要控制顶点的副本。取模函数 mod 把混合函数有效地折叠到区间[0,7]内部,使这些函数在

这个区间的不同部分活动。为了了解这种方法是怎么实现的，画出 $L=4$ 时当 t 从 0 变化到 5 对于 $k=0,1,2,3,4$ 的 5 个函数 $g((t-k)\bmod(L+1))$。

10.6.5　样条曲线和基函数

上面这种包含了二次函数 $g(t)$ 平移的方法似乎给我们提供了一种很好的曲线设计工具。为什么还要深入研究呢？问题是我们需要对曲线形状有更多的控制：它必须更加弯曲且光滑度高于一阶。这使得我们转向研究三次多项式。我们还希望设计者能够指定哪些控制顶点被插值。如果有单个算法能够包含上面所描述的所有设计方法——包括贝塞尔曲线，就好了。

因此我们想寻找满足前面期望表中列出的所有性质的更一般的混合函数族。我们继续使用同样的基于 $L+1$ 个控制顶点和 $L+1$ 个混合函数 $R_0(t),\cdots,R_L(t)$ 的参数形式：

$$P(t)=\sum_{k=0}^{L}P_kR_k(t) \tag{10.43}$$

我们依然要求 $P(t)$ 是点的仿射和。我们继续使用分段多项式作为混合函数，但现在它们定义在更一般的结点序列上，称为**结点向量**，即 $\mathbf{T}$：

$$\text{结点向量}:\mathbf{T}=(t_0,t_1,t_2,\cdots) \tag{10.44}$$

也就是结点值是 $t_0,t_1,\cdots$ 的序列，且假定是非递减的：$t_i\leqslant t_{i+1}$。某些结点可能具有相同的值但仍然给予不同的名字。T 中结点的名字将在下面讨论。

图 10.27 说明了这种情形。每个混合函数 $R_k(t)$ 都是在时刻 t_k 之前为 0，在结点向量中的几个分段区间上非 0，然后又回到 0 的混合函数。我们还要求每个 $R_k(t)$ 都是样条函数，于是这个函数在其支撑集的所有 t 处具有一定的光滑性。

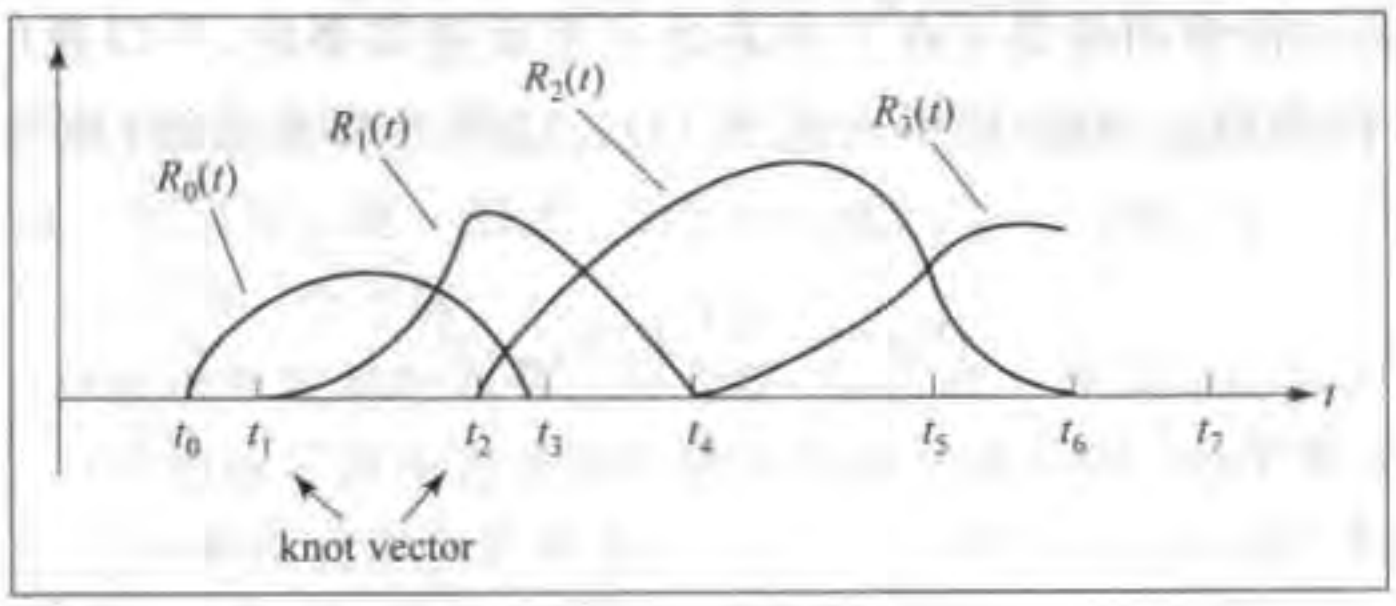


图 10.27　结点向量和混合函数的推广

注意贝塞尔曲线和 $g(t)$ 的平移公式都满足这个更一般的形式：

- 贝塞尔：有一个从 0 到 1 的区间，所有的 Bernstein 多项式都是 L 次的，在 $t=0$ 处有 $L+1$ 个结点且在 $t=1$ 处有 $L+1$ 个结点。Bernstein 多项式是样条，因为它们具有连续的 L 阶导数。
- $g(t)$ 的平移：结点值是整数 $0,1,\cdots,L+2$；每个平移函数 $g(t-k)$ 都是一个样条，由三个二次多项式组成且覆盖长度为 3 的区间。

因为每个 $R_k(t)$ 都是一个分段多项式，所以整个曲线 $P(t)$ 是用控制顶点加权的分段多项式的总和。例如，在某个区间曲线可能由下式给出

$$P(t)=P_0(3t^2-4t+2)+P_1(8t^2-7.3t-7.99)+\cdots \tag{10.45}$$

在不同的分段区间内,它由不同的多项式之和给出,但是我们知道所有分段都能够使曲线连续。这样的曲线称为样条曲线【Farin88】[①]。

现在提出的问题是:给定一个结点向量,是否存在某组混合函数使其能够生成所有定义在这个结点向量上的样条曲线?这样一组函数称为样条基,意味着任何样条曲线通过选取适当的控制多边形都可以于等式(10.43)的和相匹配。

答案是存在很多这样的函数族,但是只有一组特别的基函数其混合具有最小的支撑集并因此提供最大的局部控制。它们就是 B 样条,其中'B'源于单词 basis。

10.7 B 样条基函数

我们希望能用一种直观的方式去定义 B 样条混合函数 $R_k(t)$,这样有利于计算机实现。虽然文献给出了很多定义 B 样条的方法,但是还存在一个公式可以定义任意阶的 B 样条函数。它是递归定义的,容易编程实现,而且不存在数值问题(其他方法具有更高的计算效率,见练习)。

每个 B 样条函数都基于某个阶数 m 的多项式。如果 $m=3$,多项式的阶数为 3,度数为 2,所以它定义 2 次 B 样条。如果阶数 $m=4$,也就是度数为 3 或者立方。这是两种最重要的情况,尽管公式允许我们定义任意阶的 B 样条。

10.7.1 B 样条函数的定义

让 B 样条函数的阶数明确显示在它的表达式里面是需要的,所以我们用 $N_{k,m}(t)$ 来表示第 k 个 m 阶 B 样条混合函数,而不再用 $R_k(t)$。这样 B 样条曲线(见等式(10.43))就可以表示为:

$$P(t) = \sum_{k=0}^{L} P_k N_{k,m}(t) \tag{10.46}$$

给定以下几个条件:

- 节点向量 $\mathbf{T}=(t_0,t_1,t_2,\cdots)$。
- $(L+1)$个控制顶点 P_k。
- B 样条函数的阶 m。

则 B 样条函数 $N_{k,m}(t)$可以这样得到:

$$N_{k,m}(t) = \left(\frac{t-t_k}{t_{k+m-1}-t_k}\right)N_{k,m-1}(t) + \left(\frac{t_{k+m}-t}{t_{k+m}-t_{k+1}}\right)N_{k+1,m-1}(t) \tag{10.47}$$

其中 $k=0,1,\cdots L$。它是一个递归式即如何通过两个 $m-1$ 阶 B 样条函数来定义 m 阶 B 样条函数。要想完备这个定义,还需要给定一初始条件,它是一个区间上值恒为 1 的函数:

---

① 注意 10.6.3 节中定义的样条函数和样条曲线的区别。样条函数只是具有某种程度光滑性的分段多项式。样条曲线是使用分段多项式混合函数得到的点的仿射组合。在结点处样条曲线一定是连续的但不一定具有连续的导数。

$$N_{k,1}(t)=\begin{cases}1 & t_k<t\leqslant t_{k+1}\\ 0 & \text{其他}\end{cases} \tag{10.48}$$

注意到所有这些函数的和在任意点 t 处均为 1，所以把它们表示成点的线性组合是完全合理的。

例 10.7.1　线性 B 样条

当节点向量均匀分布如 $T=(t_0=0,t_1=1,t_2=2,\cdots)$ 时，第一个 2 阶 B 样条函数 $N_{0,2}(t)$ 会是什么形状呢？在这个条件下，等式(10.47)变为：

$$N_{0,2}(t)=\frac{t}{1}N_{0,1}(t)+\frac{2-t}{1}N_{1,1}(t) \tag{10.49}$$

图 10.28(a)给出了线性系数 t 与一阶 B 样条函数 $N_{0,1}(t)$ 相乘以及 $2-t$ 与 $N_{1,1}(t)$ 相乘的图，它们相加得到一个三角形峰线。注意到与此相同的，二阶 B 样条函数 $N_{0,2}(t)$ 是 $N_{0,1}(t)$ 与 $N_{1,1}(t)$ 的仿射组合，所以，当 $0\leqslant t\leqslant 1$ 时 $N_{0,2}(t)=t$；当 $1\leqslant t\leqslant 2$ 时 $N_{0,2}(t)=2-t$；其他情况下 $N_{0,2}(t)=0$。

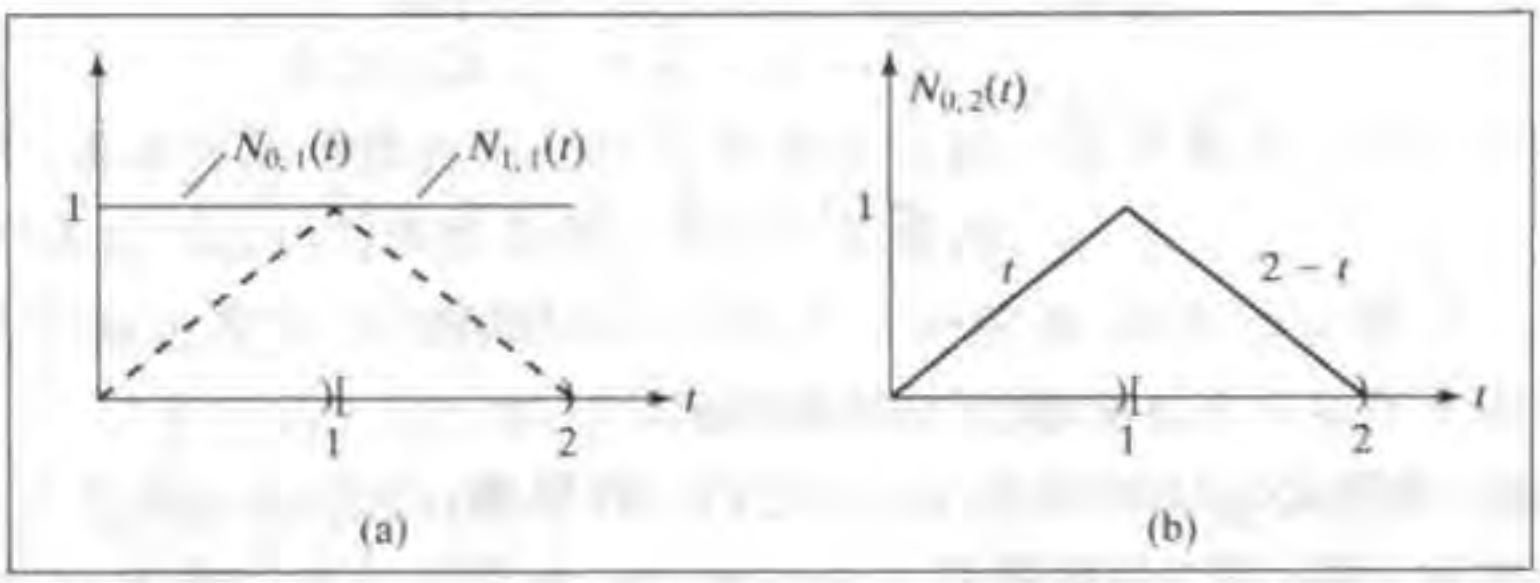


图 10.28　线性 B-样条函数的构造

其余 B 样条函数的构造类似，比如 $N_{1,2}(t)$ 也是一个三角峰线起始于 $t=1$ 结束于 $t=3$，只要将 $N_{0,2}(t)$ 平移一下即可。更一般地，当节点向量均匀分布时，每一个二阶 B 样条函数都只是第一个即 $N_{0,2}(t)$ 的一个平移。

注意到线性 B 样条曲线就是控制多边形本身(但是如果节点向量不均匀分布是不是也这样呢?)由于线性 B 样条除了简单直线没有提供任何其他东西，所以它一般不会被用在曲线设计中，但是它可以用来构造高阶 B 样条。

例 10.7.2　二次 B 样条

假如我们想确定在均匀节点条件下二次($m=3$)B 样条函数 $N_{i,3}(t)$ 的形状，我们只需看 $N_{0,3}(t)$ 的形状就可以了，因为其他都只需将它平移即可。由等式(10.47)得出：

$$N_{0,3}(t)=\frac{t}{2}N_{0,2}(t)+\frac{3-t}{2}N_{1,2}(t) \tag{10.50}$$

上述等式中两项的第一项是一上升系数与第一个三角峰线相乘，第二项是一下降系数与第二个三角峰线相乘，见图 10.29(a)，一系数与三角峰线相乘得到两条抛物线，它们相交形成一个角点，但是当上面等式(10.50)中两项相加时，角点消失得到一条连续曲线 $N_{0,3}(t)$，见图 10.29(b)。

我们可以由等式(10.50)来确定二次 B 样条函数 $N_{0,3}(t)$ 每一部分的代数形式，其结

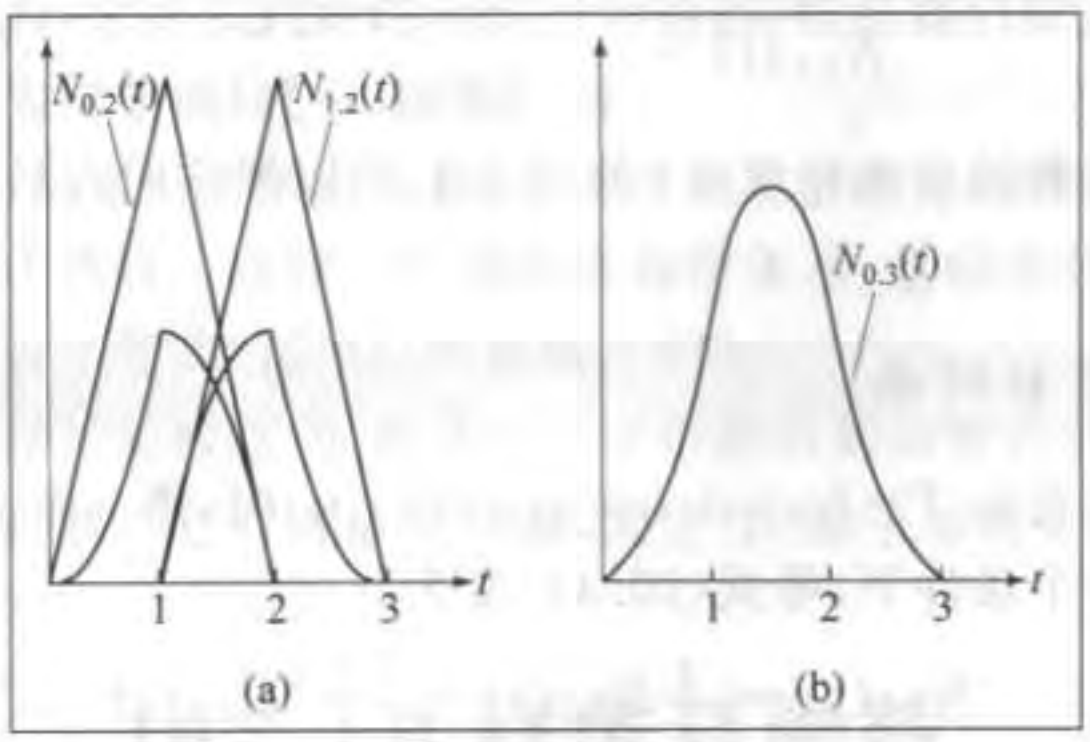


图 10.29 第一个二次 B 样条的形状

果为：

$$N_{0,3}(t)=\begin{cases}\frac{1}{2}t^2 & 0\leqslant t\leqslant 1\\ \frac{3}{4}-(t-\frac{3}{2})^2 & 1\leqslant t\leqslant 2\\ \frac{1}{2}(3-t)^2 & 0\leqslant t\leqslant 1\\ 0 & \text{其他}\end{cases} \tag{10.51}$$

它取决于四个节点 0、1、2、3，其支撑区间为[0，3]。

比较 $N_{0,3}(t)$ 与等式(10.39)中的 $g(t)$，它们一样精确，因为 $g(t)$ 也是 B 样条函数。它的一阶导数存在且连续，但二阶导数就不是这样了。在均匀节点向量下，二次 B 样条确实是一曲线。

当节点向量均匀分布时，其他二阶 B 样条的形状是很容易确定的，因为第一个二次 B 样条函数是其他二次 B 样条的一个简单平移，而且等式(10.47)中每一项系数的不同之处仅在节点向量上，也说明了这一点。事实上，此结论对均匀节点条件下的任意阶 B 样条都是成立的，因为：

$$\text{若节点 } t_k=k,\quad \text{则 } N_{k,m}(t)=N_{0,m}(t-k) \tag{10.52}$$

当节点向量均匀分布时，此形式可以直接替换到等式(10.46)中。

事实上，即使节点不均匀分布，等式(10.47)中所有二次 B 样条的和在任意点 t 处均为 1，详见练习。

例 10.7.3 三次 B 样条

三次 B 样条是最常用的，图 10.30(a)给出了 $N_{0,4}(t)$ 的形状，它的形式与前面类似(见练习)。$N_{0,4}(t)$ 关于 $t=2$ 对称，而且它可以写成：

$$N_{0,4}(t)=\begin{cases}u(1-t) & 0\leqslant t\leqslant 1\\ v(2-t) & 1\leqslant t\leqslant 2\\ v(t-2) & 2\leqslant t\leqslant 3\\ u(t-3) & 3\leqslant t\leqslant 4\\ 0 & \text{其他}\end{cases} \tag{10.53}$$

$$u(t)=\frac{1}{6}(1-t)^3, v(t)=\frac{1}{6}(3t^3-6t^2+4) \tag{10.54}$$

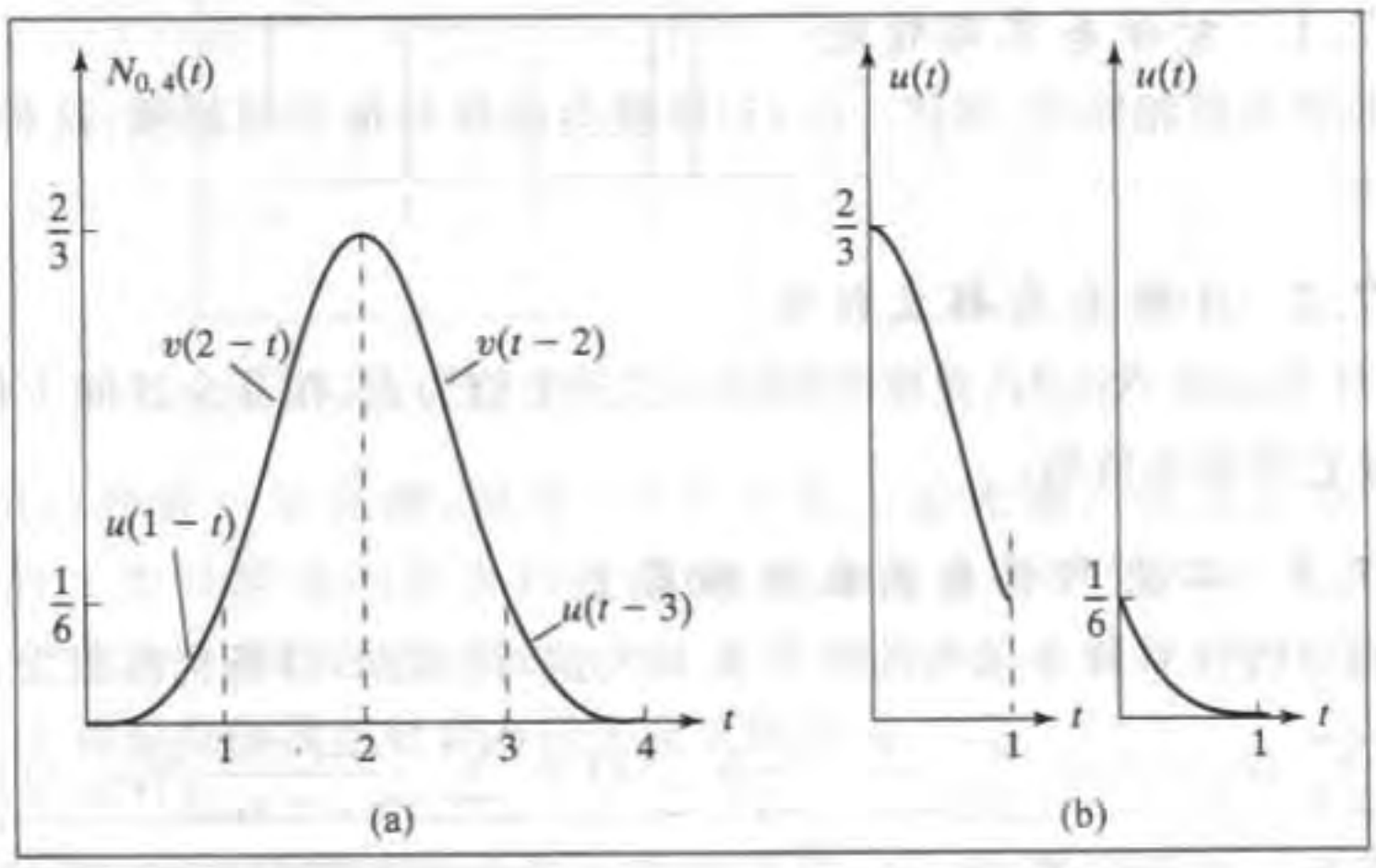


图 10.30　均匀节点下的三次 B 样条

通过直接计算求得的导数表明三次 B 样条曲线的一阶导数及二阶导数都是处处连续的，所以三次样条曲线在均匀节点条件下是二阶光滑的。

基于这里的结论我们还可以得出（见练习），一般地，B 样条函数 $N_{k,m}(t)$ 只在区间 $[t_k, t_{k+m}]$ 上为正，其余区间上均为零，它在整个区间上是非负的。

我们还应注意到除数为零的情况，即等式(10.47)的两项中的分母都可能因为选择特殊的节点向量而变为零，但是当这种情况发生时，其对应的低阶函数即 $N_{k,m-1}(t)$ 和 $N_{k+1,m-1}(t)$ 也总是相应地为零，所以任何一项若它的分母为零，我们便认为这一项为零。

由于 B 样条函数 $N_{k,m}(t)$ 是递归形式定义的，所以它很容易编程实现，容易计算它在给定节点向量条件下任意点 t 处的函数值，图 10.31 给出的代码就是等式(10.47)的实现。

```
float bSpline(lnt k, int m, float t, float knot []
{
      float denom1, denom2, sum=0.0;
      if(m==1)
        return(t>=knot[k] && t<knot[k+1]);        //1 or 0
      //m exceeds 1.. use recursion
      denom1=knot[k+m-1]-knot[k];
      if (denom1 != 0.0)
        sum=(t-knot[k]) * bSpline(k, m-1, t, knot)/denom1;
      denom2=knot[k+m]-knot[k+1];
      if (denom2 != 0.0)
        sum+=(knot[k+m]-t) * bSpline(k, m-1, t, knot)/denom2;
      return sum;
}
```

图 10.31　计算 B 样条函数

练习

练习 10.7.1 分母为零的情况

当两个相邻节点值相同时,等式(10.47)中就会出现分母为零的项,这种情况下我们将此项直接视为零。

练习 10.7.2 B 样条局部支柱性

一般地,B 样条函数 $N_{k,m}(t)$ 只在区间$[t_k, t_{k+m}]$上值为正,在其余区间上值为零,也说明了在任意点 t 处它都是非负的。

练习 10.7.3 二次 B 样条函数的和为 1

我们已经知道线性 B 样条函数的和为 1,其实这对二次 B 样条仍然成立。我们想证明

$$\sum_{k=-\infty}^{\infty} N_{k,3}(t) = \sum_{k=-\infty}^{\infty} \frac{t-t_k}{t_{k+2}-t_k} N_{k,2}(t) + \sum_{k=-\infty}^{\infty} \frac{t_{k+3}-t}{t_{k+3}-t_{k+1}} N_{k+1,2}(t)$$

在任意点 t 处值为 1,应用完等式(10.47)后将第二项中的 k 变化为 $k-1$,然后两项合并为一项,再利用一下线性 B 样条函数和为 1 的条件,就可以得到二次 B 样条函数的和在任意点 t 处为 1。

练习 10.7.4 三次 B 样条函数的计算

对均匀节点下的三次 B 样条,验证等式(10.53),并计算这个三次 B 样条的一阶及二阶导数,并证明它们处处连续。

练习 10.7.5 周期性 B 样条曲线

证明下面基于 $L+1$ 个控制顶点及 m 阶 B 样条函数的曲线

$$P(t) = \sum_{k=0}^{L} P_k N_{0,m}((t-k) \bmod (L-1)) \tag{10.55}$$

是封闭的,因此它关于 t 具有周期性。

10.7.2 如何利用节点向量中的多重节点

到现在为止我们还只是讨论了均匀节点下的 B 样条函数。通过改变相邻两个节点之间的距离,曲线设计者更容易控制最终曲线的形状。

核心问题是当两个相邻节点离得很近时,B 样条混合函数会是什么形状呢?图 10.32 给出了节点向量为 $\mathbf{T}=(0,1,2,3,3+\varepsilon,4+\varepsilon,\cdots)$时几个一阶 B 样条函数的形状,这里 ε 是一个很小的正数。

这时每一个分段多项式在区间$[3,3+\varepsilon]$的部分被挤成一小段,每一个 B 样条混合函数不再是其他同阶函数的一个简单平移。若将 ε 设为零,这一小段将不存在,而且节点 $t=3$ 变为多重节点,重数为 2。

图 10.33 给出了二重节点下 B 样条混合函数的形状,其中两个线性 B 样条函数是不连续的(图 10.33(b)),一个二次 B 样条函数在重节点 $t=3$ 处一阶导数是不连续的(图 10.33(c))。一般地,在重节点处 i 阶光滑曲线退化为 i−1 阶光滑曲线。三次 B 样条曲

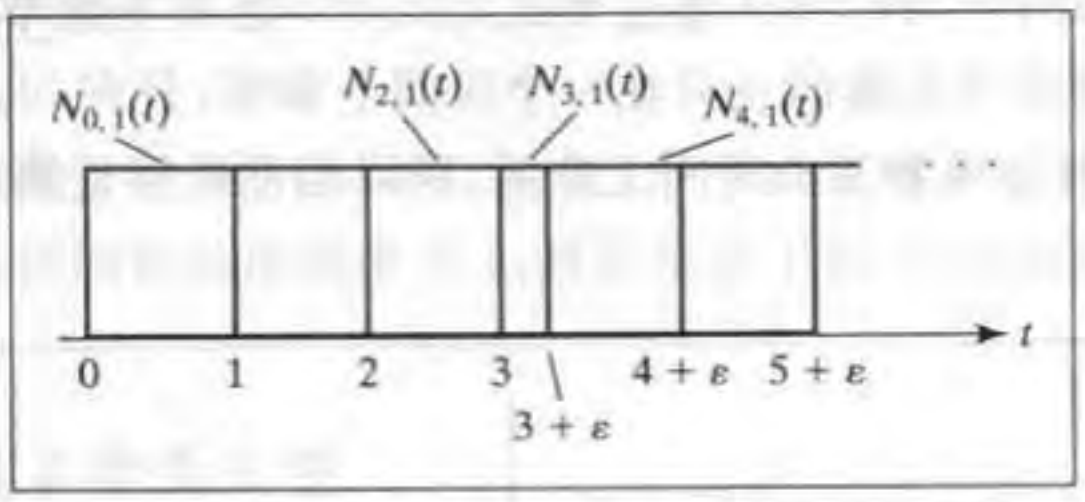


图 10.32　两个相邻节点距离很近时的 B 样函数

线(图 10.33(d))处处一阶光滑，但在 $t=3$ 处非二阶光滑。但是注意到如果我们用图 10.33(c)中的二次 B 样条函数去构造曲线，则曲线将会插值控制顶点 P_2，因为函数 $N_{2,3}(t)$在 $t=3$ 处值为 1，而其他函数在 $t=3$ 处值均为 0。一般地，当 t 接近一个重数大于 1 的节点时，在相应控制顶点处会有很多重要的性质。

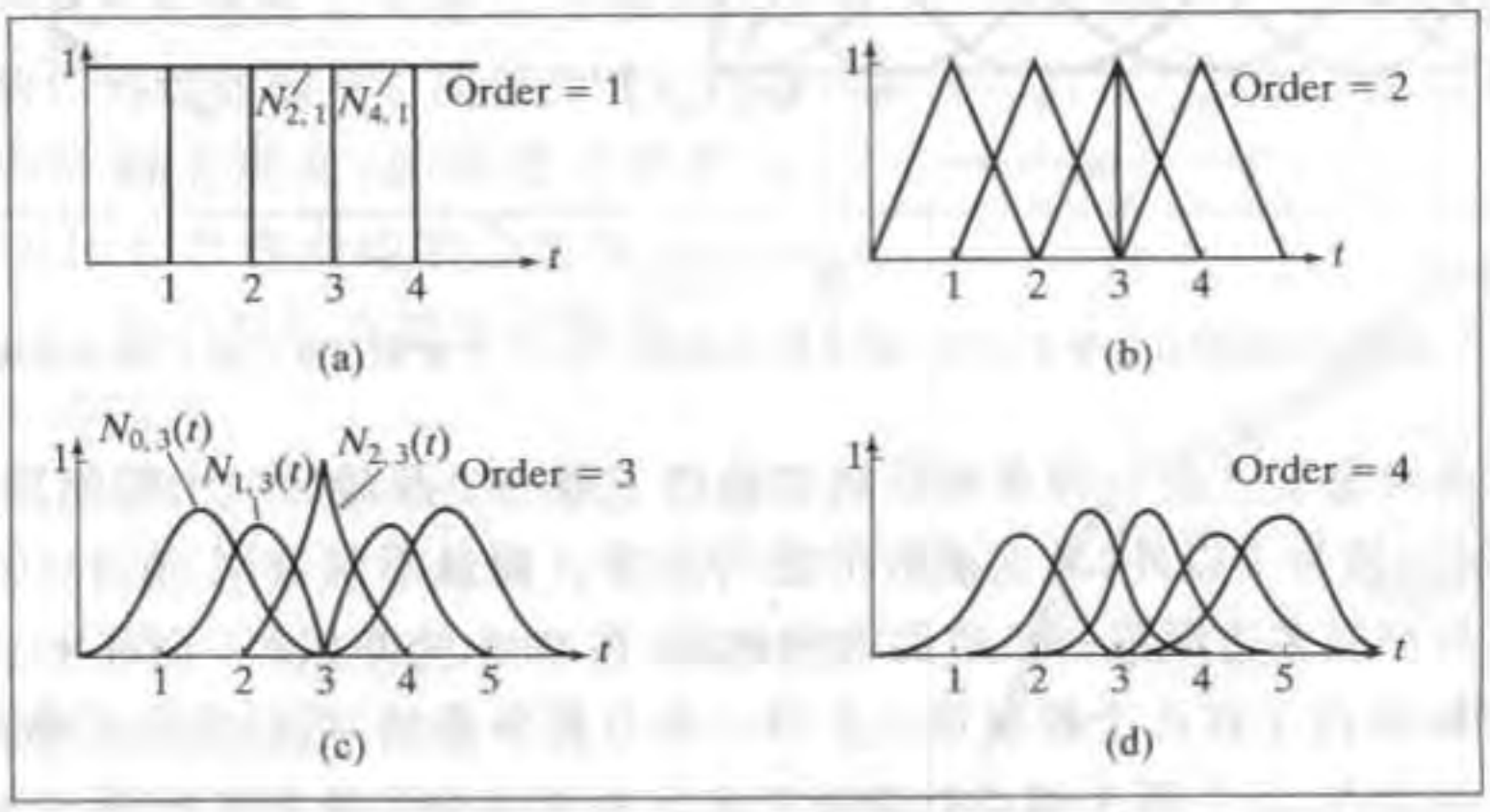


图 10.33　二重节点下 B 样条的形状

另外，二次样条在三重节点处不连续，三次样条在三重节点处一阶导数不连续，但是它们仍然插值其中一个控制顶点。通过调整每一个节点的重数，曲线设计者可以很好得控制曲线的形状。

不过当节点向量包含重节点时，无论等式(10.47)还是图 10.31 中的代码部分都不需要改变，至于等式(10.47)我们已经提到会出现分母为零的情况，图 10.31 中的代码也无需修改，因为它会自动处理这种情况的。

10.7.3　非封闭 B 样条曲线：标准节点向量

在曲线设计中，一类特殊的节点向量已成为标准。在此节点向量下，曲线插值第一个和最后一个控制顶点，使得设计者很容易预测曲线的大致位置。

m 阶 B 样条曲线的标准节点向量为首末两个 m 重节点，其余节点距离相差 1，我们先看一个例子再看这个节点向量是怎么提出的。假设有八个控制顶点，我们用三次($m=4$)B 样条函数来构造曲线，则标准节点向量为：

$$\mathbf{T}=(0,0,0,0,1,2,3,4,5,5,5,5)$$

B 样条函数 $N_{0,4}(t),\cdots,N_{7,4}(t)$ 通过等式(10.47)定义在此节点向量上,见图 10.34(a)。$N_{0,4}(t)$,$N_{7,4}(t)$ 都是不连续的且只在一个区间上非零,只有 $N_{3,4}(t)$,$N_{4,4}(t)$ 在四个区间上非零,其余都是在两个或者三个区间上非零,所以当曲线接近第一个和最后一个控制顶点时形状被严重扭曲。

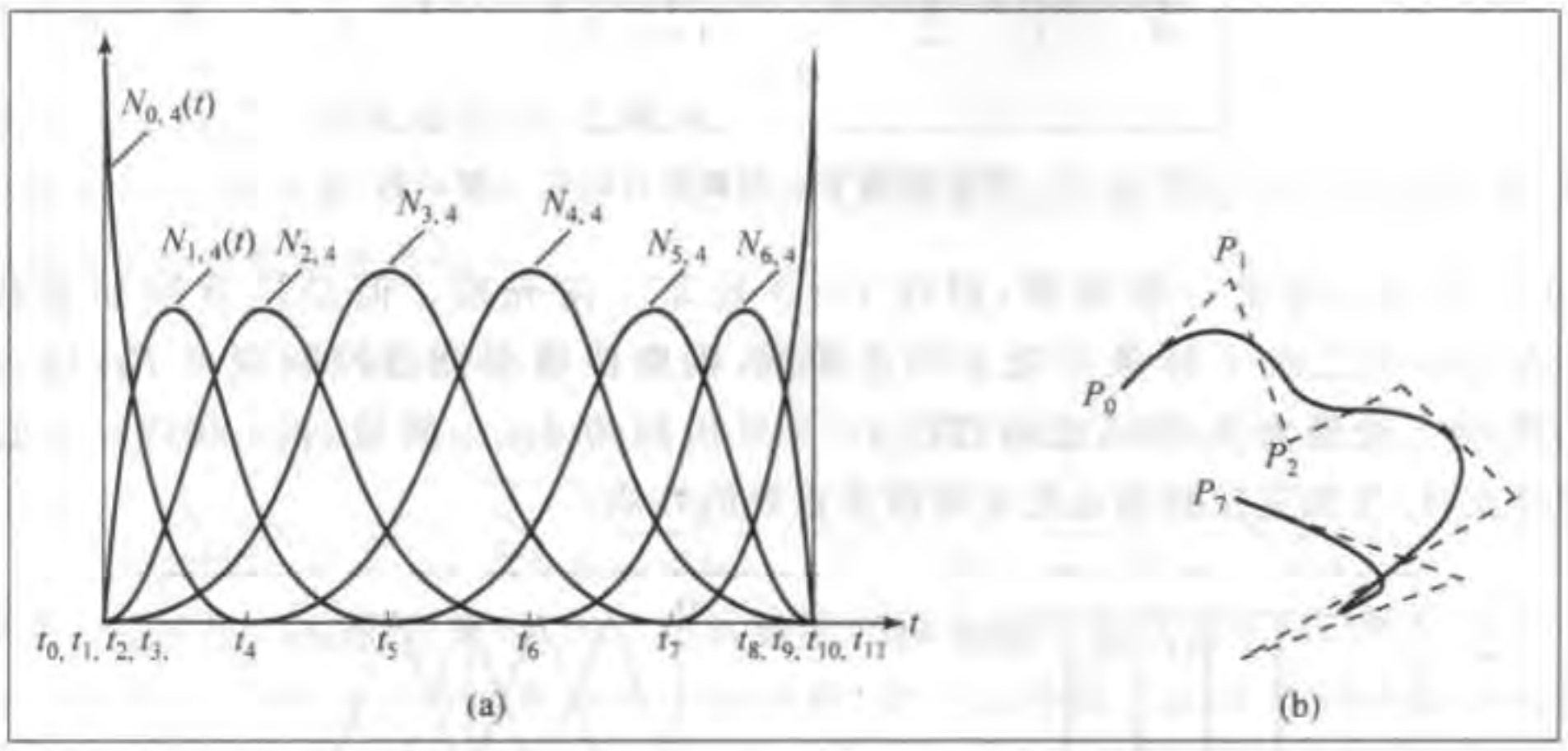


图 10.34

(a) 定义在标准节点向量下 8 个三次 B 样条函数;(b) 八个控制顶点下的 B-样条曲线

但是此节点向量下这些 B 样条函数确实插值了第一个及最后一个控制顶点。比如 $t=0$ 处,除 $N_{0,4}(t)$ 的值为 1 以外,其他函数的值均为零。而且不难证明,在 $t=0$ 处,B 样条曲线的切线方向与控制多边形第一条边的方向相同,在 $t=5$ 处亦如此(见练习)。

图 10.34(b)给出了在八个控制顶点下的一条 B 样条曲线,可以看出它确实插值第一个和最后一个控制顶点,且在两个端点处切线方向与控制多边形的方向相同。

注意到当控制多边形自交时,B 样条曲线也会出现自交。

$L+1$ 个控制顶点下的 m 阶 B 样条曲线的标准节点向量应满足如下条件:

(1) 首先要有 $L+m+1$ 个节点,记为 $t_0,\cdots,t_{L+m}$。

(2) 前 m 个节点即 $t_0,\cdots,t_{m-1}$ 的值均为零(第一个 m 阶 B 样条函数从 $t=0$ 开始定义)。

(3) 节点 $t_m,\cdots,t_L$ 的值从 1 开始依次增加 1,直到 $L-m+1$(最后一个 m 阶 B 样条函数 $N_{L,m}(t)$ 从 $t_L=L-m+1$ 开始定义,只在一个区间上值非零)。

(4) 最后 m 个节点即 $t_{L+1},\cdots,t_{L+m}$ 的值均为 $L-m+2$。

有了上述规则,给定值 m 和 L,就很容易编写程序来产生一个标准节点向量,代码如下:

```
void buildKnots(int m, int L, double knot[])
{
    // 建立 L+1 个控制顶点下的 m 阶 B-样条曲线的标准节点向量
    int i;
    if(L<(m-1)) return;                    // 控制顶点太少
    for(i=0; i<=L+m; i++){
        if (i<m) knot[i]=0.0;
        else if (i<=L) knot[i]=i-m+1;      // i 不小于 m
```

```
        else knot[i]=L-m+2;}                    // i 大于 L
}
```

注意代码中的判断条件取决于 m 和 L 的值，对于给定的阶数 m 必须有一定数目的控制顶点使得曲线的定义区间的长度至少为 1，所以便有了如下的约束条件：阶数 m 不能超过控制顶点的个数 $L+1$。

贝塞尔曲线也是 B 样条曲线

前面我们通过两种方式引入了贝塞尔曲线，即：德卡斯特里奥算法和 Bernstein 多项式。现在我们给出第三种方式。贝塞尔曲线也是一特殊的 B 样条曲线，这是因为当 $m=L+1$ 时定义在标准节点向量上的 B 样条函数其实是一 Bernstein 多项式，即：

$$N_{k,L+1}(t) = B_k^L(t), \quad k = 0,\cdots,L$$

我们需要注意的是当阶数 m 增加到 $L+1$ 时，标准节点向量会发生什么样的变化（见练习），前面 m 个节点的值为 0，后面 m 个节点的值为 1，t 只在区间[0,1]上变化。例如，如果 $L=5$，$m=6$，我们得到标准节点向量为 $\mathbf{T}=(0,0,0,0,0,0,1,1,1,1,1,1)$，这样每个分段多项式只在一个区间上定义，且多项式的阶 $m=L+1$，这就是为什么 B 样条函数会变为 Bernstein 多项式。事实上，我们也可直接通过等式(10.47)得到 Bernstein 多项式。

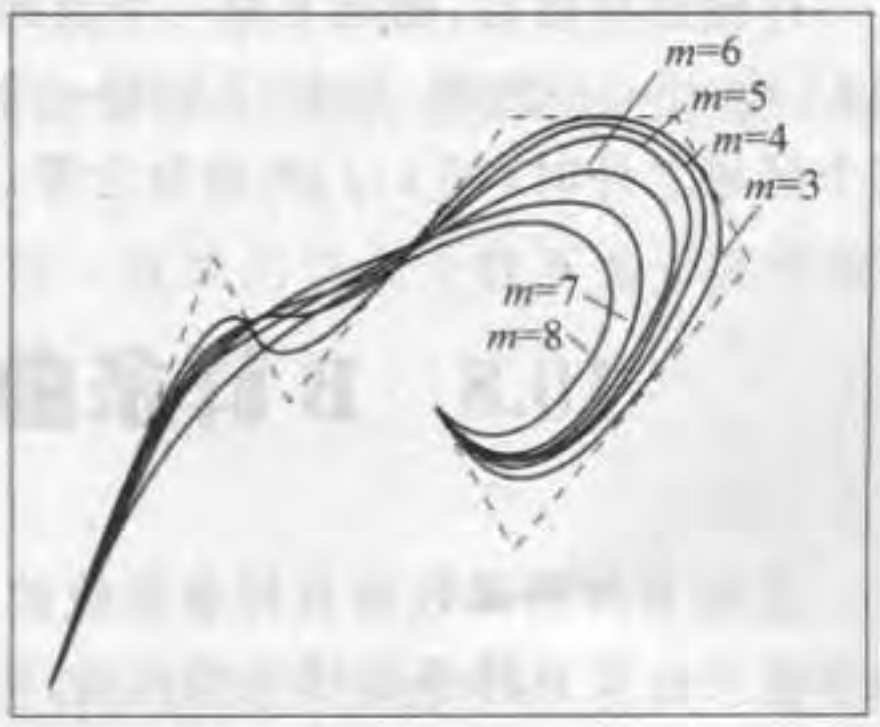


图 10.35　不同阶数的 B 样条曲线

我们最初引入 B 样条曲线的原因是贝塞尔曲线不能很好地对曲线进行局部控制与调整。当 B 样条函数的阶数增加 1 时，它的非零区间就增加 1 个，曲线局部控制就变得不那么灵活，当 m 增加到 $L+1$ 时，就出现了 B 样条曲线的情况，失去了局部控制灵活性。图 10.35 给出了阶数逐渐增加时 B 样条曲线的变化。图中有八个控制顶点，所以当阶数 $m=8$ 时，曲线就变成了贝塞尔曲线。图中的曲线都是由代码 10.31 产生的。（$m=2$ 时的曲线是？）

练习

练习 10.7.6　二次 B 样条曲线的标准节点向量

八个控制顶点下的三阶 B 样条曲线的标准节点向量为 $\mathbf{T}=(0,0,0,1,2,3,4,5,6,6,6)$。

练习 10.7.7　当阶数 m 增加时的标准节点向量

当阶数分别为(a)$m=3$；(b)$m=4$；(c)$m=5$；(d)$m=6$；(e)$m=7$ 在七个控制顶点下的标准节点向量分别是？

练习 10.7.8　标准节点向量下的二次 B 样条曲线

通过等式(10.47)，计算出在标准节点向量下二次 B 样条函数的显式表达式（分段多项式的形式）

练习 10.7.9 三次 B 样条

计算出前四个标准节点向量下的三次 B 样条函数,并求出曲线在 $t=0$ 时的导数,看它与控制多边形的第一条边的方向是否相同?

练习 10.7.10 导出 Bernstein 多项式

当 $m=L+1=4$ 时,通过等式(10.47)求得的 B 样条函数就是 Bernstein 多项式。

练习 10.7.11 B 样条的高效计算

当绘制一条 B 样条曲线时,需要产生许多点,也就是要计算出 $N_{k,m}(t)$ 在很多不同的 t 点处的值。用图 10.58 递归的方法计算会导致许多多余的计算量即低阶 B 样条函数的值。这里我们提供一个高效的方法,每个 $N_{k,m}(t)$ 的值只需计算一次,然后把它们存为数组,比如 $N[k,i]$ 中保存 $t=i\Delta t$ 处计算得到的 $N_{k,m}(t)$ 的值,其中 Δt 可以按照需要设定。这样就只需要在 B 样条函数的非零区间上计算即可,而且只需要计算几种不同型形状的 $N[k,i]$,因为其他都是某个函数平移得到的。那么若有 55 个控制顶点,三次 B 样条函数有多少种不同的形状呢?

存储这些值后,曲线上每一个点的坐标都可以通过访问数组的某个值得到:$P(t_i)$ 是各项 $C[k]N[n,j]$ 的和,其中 $C[k]$ 是一个控制顶点,需要选择适当的 n,j 的值。注意当 j 处于某个区域之外时,$N[k,j]$ 的值全为零,就不需要再访问这个数组了。

10.8 B 样条曲线在设计中的重要性质

总结 B 样条函数和 B 样条曲线的基本性质是很重要的。我们应该知道贝塞尔曲线的很多重要性质 B 样条曲线也都具备。

(1) m 阶 B 样条函数事实上就是一 m 阶分段多项式,且它们是 $m-2$ 阶光滑的,也就是说它们的 $m-2$ 阶导数是连续的。它们形成一组样条基,就是说任何样条曲线都可以表示为它们的线性组合。在所有的样条基中,B 样条基是局部支柱性最好的,也就说 B 样条基的非零区间是最短的。

(2) B 样条混合函数 $N_{k,m}(t)$ 的非零区间为 $[t_k,t_{k+m}]$,起始于 t_k,终止于 t_{k+m}。函数集 $\{N_{k,m}(t)\mid k=0,1\cdots,L\}$ 的非零区间为 $[t_0,t_{m+L}]$。

(3) 由等式(10.55)可以得到一条 $L+1$ 个控制顶点下的封闭 B 样条曲线(等式(10.55)是在节点均匀分布条件下的)。

(4) 当节点取为标准节点向量时,B 样条曲线会插值第一个及最后一个控制顶点,且在两个端点处的切线方向分别与控制多边形的第一条边及最后一条边的方向相同。

(5) B 样条函数 $N_{k,m}(t)$ 在任意点 t 处都是非负的。且对任意的 $t\in[t_0,t_{m+L}]$,均有

$$\sum_{k=0}^{L} N_{k,m}(t) = 1 \tag{10.56}$$

前面我们已经证明了低阶的情况,高阶的情况可以由等式(10.47)归纳得到。

(6) B 样条曲线具有仿射不变性。要想变换一条 B 样条曲线,只需同样变换所有控制顶点即可,这是因为 B 样条曲线是所有控制顶点的仿射组合。

(7) 根据性质 5，B 样条曲线是它的所有控制顶点的凸组合，所以整条曲线落在控制顶点集的凸包内。但是注意到在任意点 t 处，只有 m 个 B 样条函数是非零的，所以在 t 处，曲线落在 m 个连续的活动控制顶点组成点集的凸包内。图 10.36 给出了在标准控制顶点下的一个二次 B 样条曲线，在任意点 t 处最多只有 3 个控制顶点是活动的，所以其凸包是三角形。随着 t 的逐渐增加，$P(t)$ 渐进地跳出一个三角形凸包进入下一个，因为 B 样条函数是轮流活动的。请问图中曲线进入及离开阴影凸包的 t 值分别是多少？

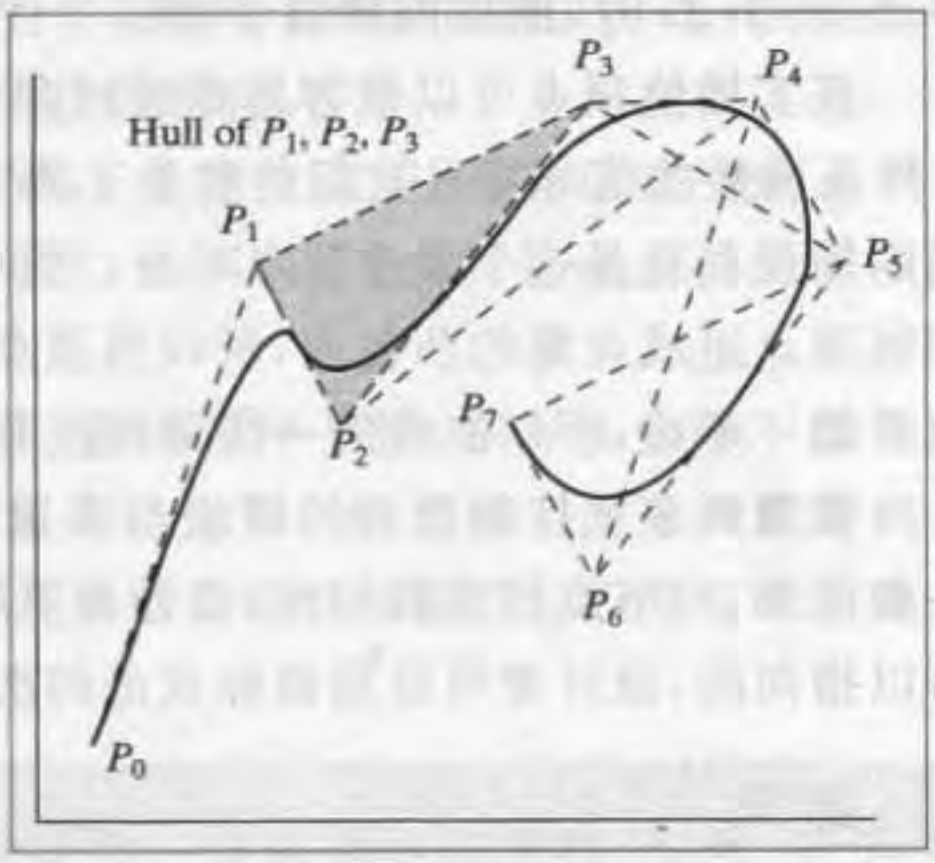


图 10.36　二次 B-样条曲线的凸包

m 个控制顶点组成点集的凸包比所有控制顶点组成点集的凸包要小得多，所以与贝塞尔曲线相比，B 样条曲线被限制在了一个更小的区域，这个性质使得曲线设计者更容易局部调节控制曲线的形状。

(8) B 样条曲线可以呈现直线型：如果 m 个连续控制顶点共线，它们的凸包是一条直线，所以曲线呈直线型。

(9) B 样条曲线具有变差缩减性：B 样条曲线与任一直线的交点个数不多于其控制多边形与该直线的交点个数【Farin88】。

10.8.1　多重控制顶点

曲线设计者还可以利用多重控制顶点更有效地调节 B 样条曲线的形状，所谓多重控制顶点就是使几个连续控制顶点重合于一点，这会使得曲线更加逼近控制多边形。

图 10.37 给出了三次 B 样条曲线的例子。当顶点 D 为单重顶点即控制顶点为 A、B、C、D、E、F、G 时，曲线呈现出通常三次 B 样条的形状；当顶点 D 为二重顶点即控制顶点为 A、

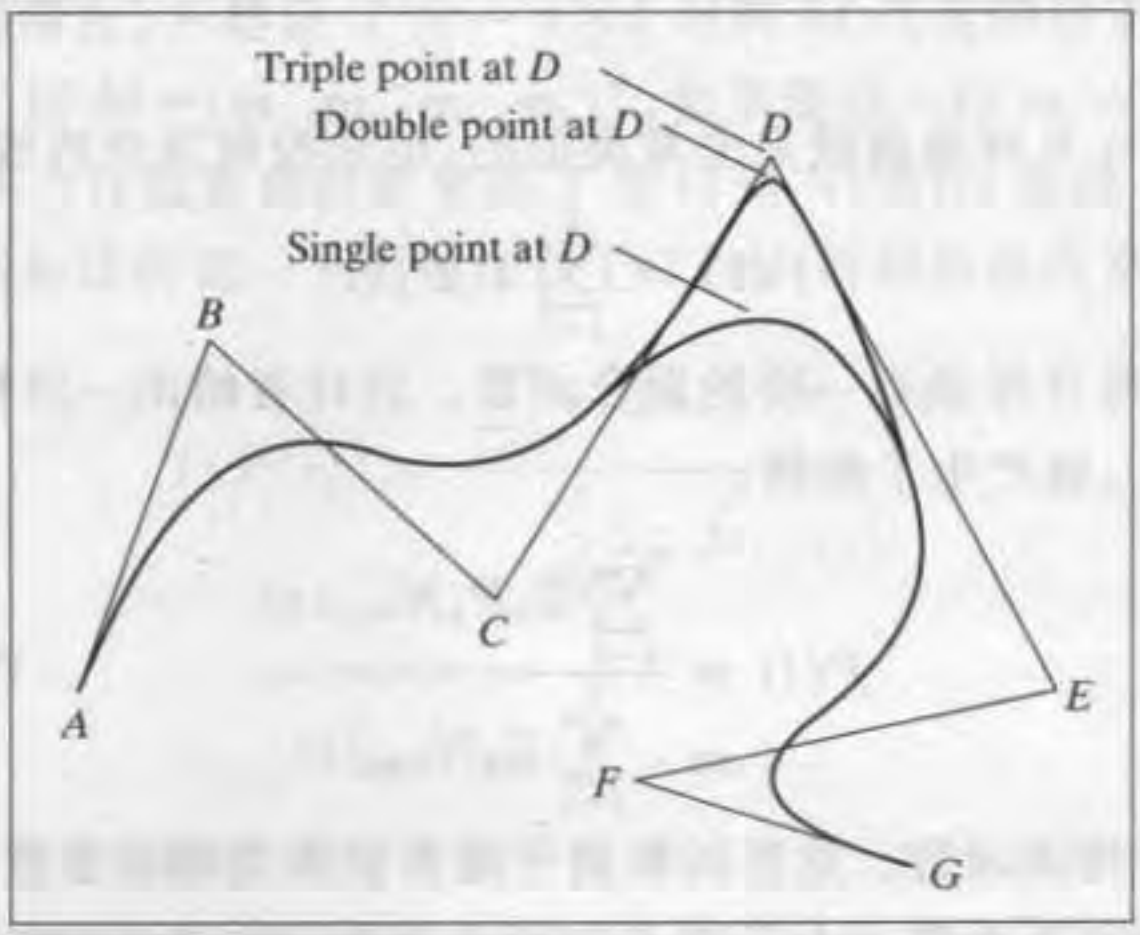


图 10.37　多重控制顶点下的 B-样条曲线

B、C、D、D、E、F、G 时,曲线更加靠近 D 点;当顶点 D 为三重顶点即控制顶点为 A、B、C、D、D、D、E、F、G 时,曲线则插值 D 点。

其实插值顶点可以很容易地通过图 10.34(a)来解释,注意到在节点 t_5,t_6 处,只有三个 B 样条函数的值非零且它们的和为 1,所以如果这三个相应的控制顶点重合为一点,那么它们的加权和就是这个多重顶点本身。更一般地,一个三次 B 样条曲线会被限制在四个连续控制顶点组成点集的凸包内,所以当顶点 D 为三重顶点时,包含 D 的凸包就变成了控制多边形的一条边,所以曲线的一段落在控制多边形的那两条边上,插值了多重顶点 D。

注意到多重控制顶点的用法与多重节点的用法是不同的,尽管它们的效果是一样的。一般说来,与节点的重数相比,设计者更容易增加控制顶点的重数,因为控制顶点是可见且可以指向的,设计者可以用鼠标点击的次数来指定控制顶点的重数。

练习

练习 10.8.1 凸包性质

画出几种八个控制顶点组成的控制多边形,假设二次 B 样条函数定义在标准节点向量下,画出其连续凸包,并计算出曲线进入和退出每个凸包的 t 值。然后大致勾画出 B 样条曲线,确保它通过其控制多边形内部边的中点。

练习 10.8.2 多重控制顶点下的二次 B 样条曲线

解释双重控制顶点在二次 B 样条曲线中的作用。先画出一个没有多重顶点的控制多边形,画出它的每个相应凸包,大致勾画出 B 样条曲线。然后逐次增加其中一个控制顶点的重数后重复上述过程。

10.9 有理曲线和非均匀有理 B 样条曲线

在 10.2 节我们大致介绍了贝塞尔曲线的有理参数形式,知道了圆锥曲线(如椭圆,双曲线,抛物线)可以由有理二次贝塞尔曲线(等式(10.14))精确产生,这里我们详细介绍有理 B 样条曲线。

有理 B 样条曲线与 B 样条曲线是非常类似的,也是控制顶点的线性组合,即:

$$P(t) = \sum_{k=0}^{L} P_k P_k(t) \tag{10.57}$$

但是有理 B 样条使用与 B 样条不一样的混合函数。设计者给出一组权因子$\{\overline{\omega_0},\overline{\omega_1},\cdots,\overline{\omega_L}\}$和控制顶点 $P_0,\cdots,P_L$,就产生了曲线:

$$P(t) = \frac{\sum_{k=0}^{L} \overline{\omega}_k P_k N_{k,m}(t)}{\sum_{k=0}^{L} \overline{\omega}_k N_{k,m}(t)} \tag{10.58}$$

其中 $N_{k,m}(t)$是 k 阶 B 样条函数。这里的权因子通常被称为形状参数,且通常被设为非负以保证等式(10.58)中分母不为零。由于节点向量通常是非均匀分布的,所以此时的曲线被称为非均匀有理 B 样条(即:NURBS)。

观察等式(10.58),$P(t)$的分子是所有控制顶点的线性组合。在第 4 章我们已经知道,为了使得它有意义,我们需要把它表示成控制顶点的仿射组合,所以同除以分母,使得所有系数的和为 1,这样 $P(t)$就表示成了所有控制顶点的仿射组合。

由等式(10.58)我们还可以看出,如果其中的权因子是完全相同的,分母就变成了一个常数(为什么?),$P(t)$就退化为最初的 B 样条曲线等式(10.46)了,也就是说当权因子不同时,$P(t)$就是等式(10.46)的推广。

我们再来看一下这里的混合函数是怎么得来的。像练习中一样,在齐次坐标下给每个控制顶点一个权因子,即给顶点 P_k 权因子 ω_k,然后将其转化为一般坐标下的形式就得到了 NURBS 曲线,NURBS 曲线通过控制顶点的权因子在高维空间中保留了它们原有的性质。

NURBS 曲线的两大重要优点

1. 通过合适地选择控制顶点和权因子,$P(t)$就成为一条精确圆锥曲线,见例 10.9.2。这不同于非有理 B 样条曲线,它只能逼近而不能精确得到圆锥曲线。

2. NURBS 曲线在投影变换下保持不变,所谓投影变换我们已经在第七章中讨论过了,它是仿射变换的推广。仿射变换矩阵的第四行固定为(0,0,0,1),但是投影变换矩阵的第四行更一般化。一般的 B 样条曲线具有仿射变换不变性,所谓仿射变换就是旋转,伸缩,平移以及它们的线性组合,而 NURBS 曲线在更宽范围的变换下有不变性。

这就是说如果想得到 NURBS 曲线在某个方向上的投影,只需要先得到每个控制顶点在此方向上的投影,然后通过等式(10.57)得到新的曲线就是原曲线在此方向上的投影(当然权因子也要做相应的调整),这样比找出曲线上每一个点的投影要简单得多。但是相应地 B 样条就不能这样,因为它只在仿射变换下具有不变性。

投影不变性

这里我们来详细介绍一下 NURBS 曲线的投影不变性。投影不变性就是当一个 NURBS 曲线通过一个 4×4 矩阵变换时,其结果仍然是一 NURBS 曲线,它的控制顶点是原曲线的控制顶点通过上面 4×4 矩阵变换得到的,当然这里的权因子要作适当调整。

我们来考虑 3D 的情况,不妨设 T 为一 4×4 矩阵 M 代表的投影变换,$\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$、$\mathbf{m}_4$ 分别代表 M 的每一行,即 $M=(\mathbf{m}_1,\mathbf{m}_2,\mathbf{m}_3,\mathbf{m}_4)^T$,如果最后一行 $\mathbf{m}_4=(0,0,0,1)$,则 T 就是仿射变换。所谓投影不变性就是将投影变换 T 作用到 NURBS 曲线 $P(t)$的每一个点上即 $T(P(t))$与将它作用到曲线的每一个控制顶点上 $T(P_k)$得到的曲线是相同的【Pieg191】。

$$T(P(t))=\frac{\sum_{k=0}^{L}\overline{\omega}_k T(P_k)N_{k,m}(t)}{\sum_{k=0}^{L}\overline{\omega}_k N_{k,m}(t)} \tag{10.59}$$

其中权因子调整为:

$$\overline{\omega}_k=\omega_k(\widetilde{P}_k\cdot\mathbf{m}_4) \tag{10.60}$$

$\widetilde{P}_k=(P_x,P_y,P_z,1)^T$ 为控制顶点 P 的齐次坐标形式,齐次坐标为 1。注意到权因子的调整依赖于控制顶点的位置,即矩阵 M 的第四行,所以如果 T 是一仿射变换,则 $(P_k\cdot\mathbf{m}_4)=1$,权因子不用调整。

练习

练习 10.9.1 一般曲面如何通过 NURBS 曲线产生?

讨论以下曲面如何通过 NURBS 曲线产生?

a. 圆柱

b. 圆锥

c. 圆盘

d. 球

详细参考案例 10.10。

练习 10.9.2 当权因子完全相同时会如何?

当 NURBS 曲线的权因子完全相同时,它就退化一 B 样条曲线,这时 NURBS 曲线被一矩阵 M(M 是一个 4×4 矩阵)变换后是否还是一 NURBS 曲线? 是否还是一 B 样条曲线?

练习 10.9.3 在齐次坐标下产生 NURBS 曲线

对一 B 样条曲线,我们给曲线的每一个控制顶点 P_k 加上一个权因子 ω_k,则曲线在齐次坐标下表示为:

$$\widetilde{R}(t)=\sum_{k=0}^{L}\omega_k\widetilde{P}_kN_{k,m}(t) \tag{10.61}$$

这里 $\widetilde{P}_k$ 表示顶点 P_k 的齐次坐标形式(即:$\widetilde{P}_k=(P_x,P_y,P_z,1)^T$)。

a. 分别写出 $\widetilde{R}(t)=(x(t),y(t),z(t),\omega_k)^T$ 的每一个组成部分即 $x(t),y(t),z(t),\omega_k$,并注意观察 ω_k 的形式。

b. 除以 ω_k,把上述 $\widetilde{R}(t)$ 转化为一般坐标下的形式,得到:

$$P(t)=\frac{\sum_{k=0}^{L}\omega_kP_kN_{k,m}(t)}{\sum_{k=0}^{L}\omega_kN_{k,m}(t)} \tag{10.62}$$

c. 可以看出它与等式(10.57)或等式(10.58)的形式是完全相同的。

例 10.9.1 怎样得到圆锥曲线?

对于等式(10.14)的特殊情况,令 $m=3$(二次 B 样条),并选择合适的节点向量使得B 样条为 Bernstein 多项式(如何选择?),然后令权因子 $\omega_0=\omega_2=1,\omega_1=\omega$,这样就可以得到等式(10.14)。

例 10.9.2 怎样得到一个精确的圆?

由 NURBS 曲线可以通过多种方式精确得到一个圆【Pieg189】,我们在这里给出怎样通过七个控制顶点下的二次 B 样条得到圆。图 10.38(a)给出了控制顶点 $P_0,\cdots,P_6$ 围成一个

正方形，圆就围绕着这个正方形产生，权因子的值也在图中标出，它们要么为 1，要么为$\frac{1}{2}$，节点向量为 $T=\left\{0,0,0,\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{3}{4},1,1,1\right\}$，注意这里 $t=\frac{1}{2}$为双重节点。图 10.38(b)给出了七个基函数，其中所有 $R_k(t)$的共同分母的曲线图中用虚线画出。本节后面练习给出了确定每个基函数在非零区间内实际等式的过程，并验证在所有 t 处均有$x^2(t)+y^2(t)=1$。

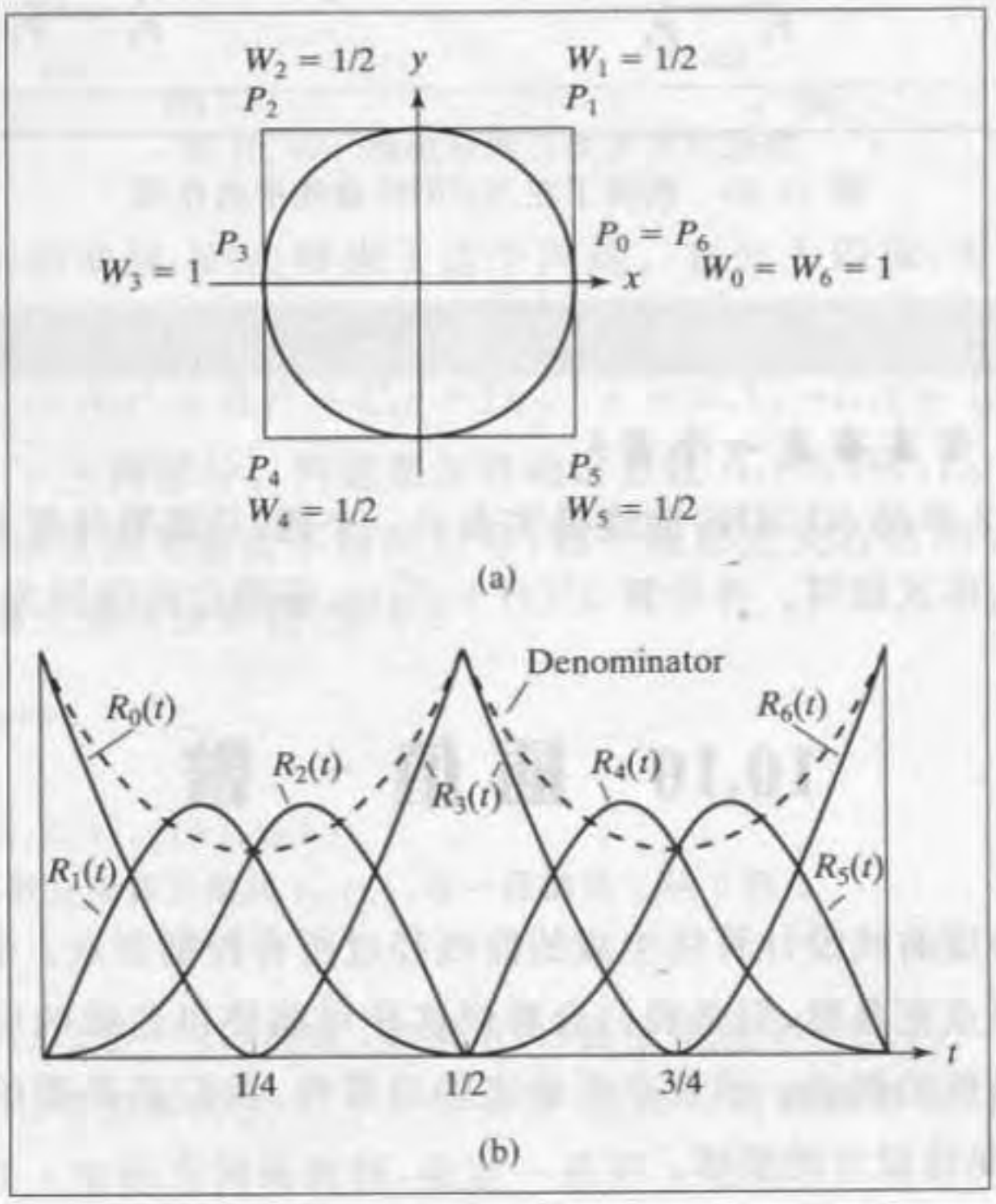


图 10.38　通过多重控制顶点调整曲线

(a) NURBS 曲线得到圆；(b) 对应的基函数

例 10.9.3　权因子的作用

下面图 10.39(a)给出了 6 个控制顶点下的三次 B 样条曲线(在均匀节点下)，它是通常情况下的非有理 B 样条(等价于权因子相同时的 NURBS 曲线)。图 10.39(b)显示出了当权因子不一样时曲线发生的变化，即曲线更接近于权因子较大的控制顶点(当权因子为零时会怎样呢?)。

NURBS 曲线的普遍性和灵活性使得它在曲线设计中越来越流行。由于它包含了 B 样条曲线这种特殊情况，所以一个单一的曲线算法可以用于生成较宽范围的曲线族，包括圆锥曲线。所以设计者只需一个算法即可，而不需要一个包含多种算法的库。

在曲线设计中还存在许多其他方法，其中一些可以让用户更好的控制曲线的形状。比如用户可以改变节点之间的距离或者增加控制顶点等来控制曲线的形状。读者如果对此感兴趣可以上网搜索关于 B 样条和 NURBS 曲线的其他资源。

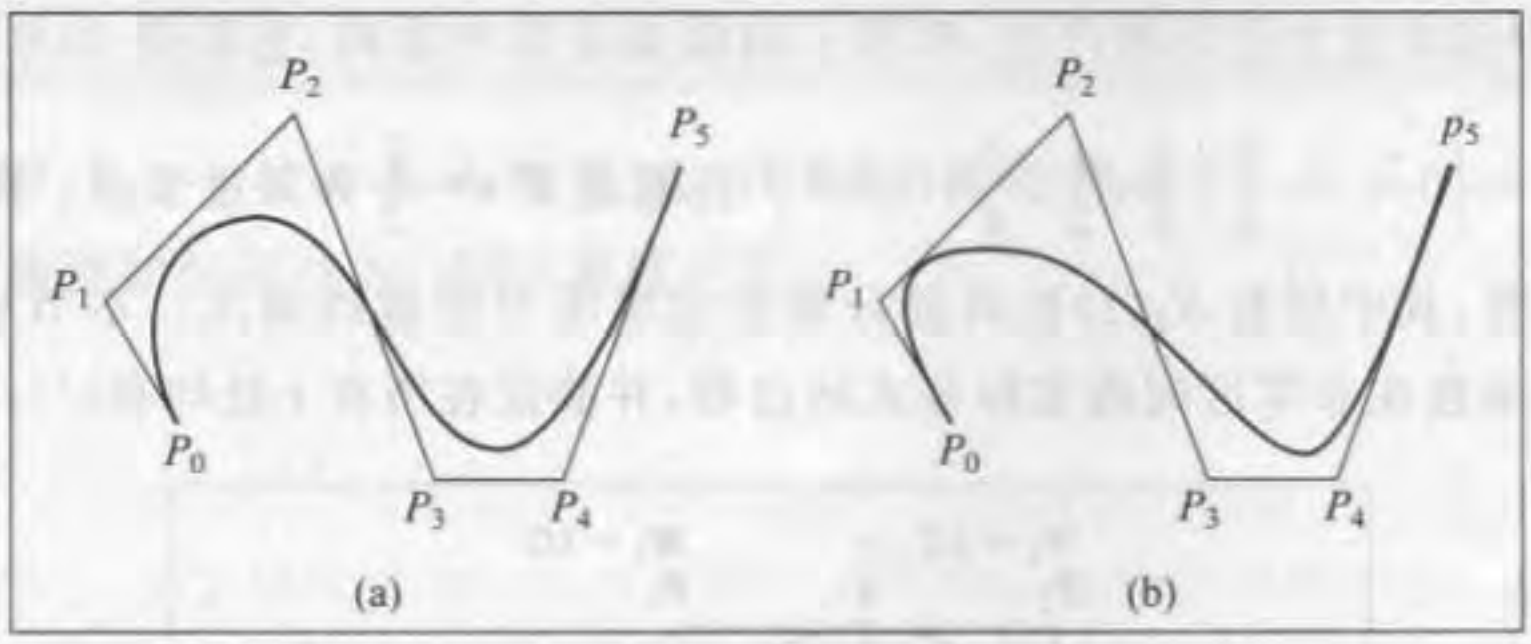


图 10.39 权因子在 NURBS 曲线中的作用

练习

练习 10.9.4 它是否是一个圆?

证明例子 10.9.2 中的 NURBS 曲线确实表示一个圆,只需要计算出在每个节点区间上基函数的多项式表达形式即可。并计算 $x^2(t)+y^2(t)$,证明它的值恒为 1。

10.10 插值一瞥

有时候设计者希望曲线设计算法生成的曲线经过所有控制顶点。表面上好像这样比让曲线逼近每个控制顶点更自然,但是我们会看到这样可能使得曲线的形状变得很局限或者控制顶点有不希望出现的摆动。尽管存在着这些局限性,我们还是要给出各种情况下的插值算法,满足现实中曲线设计的要求。

设计者已经有了一种通过 B 样条得到的插值算法,即:给定每一个控制顶点足够大的重数,就可以使得得到的 B 样条曲线插值它们。

在案例 10.6 中我们将看到另外一种插值方法,它也是建立在 B 样条基础上,设计者给出一系列点,此算法将会产生另外一系列顶点,以此作为控制顶点产生的 B 样条曲线将会插值设计者最初给出的点。

这一节我们大致介绍一下曲线插值算法,即给定一系列点,如何得到一曲线使得它经过这些点。这是一个很早就提出的问题,我们在这里仅介绍一部分算法,Farin【Farin88】详细讨论了曲线插值这个问题。

10.10.1 利用分段三次多项式插值

这里我们介绍如何利用分段三次多项式来插值。我们只用分段三次多项式是因为它虽然简单但是足够灵活有效。

问题:图 10.40(a)解释了这个基本问题,即:用户给定一系列控制顶点 $P_0, P_1, \cdots, P_L$(这里 $L=4$),得到一条足够光滑的曲线依次经过这些点。

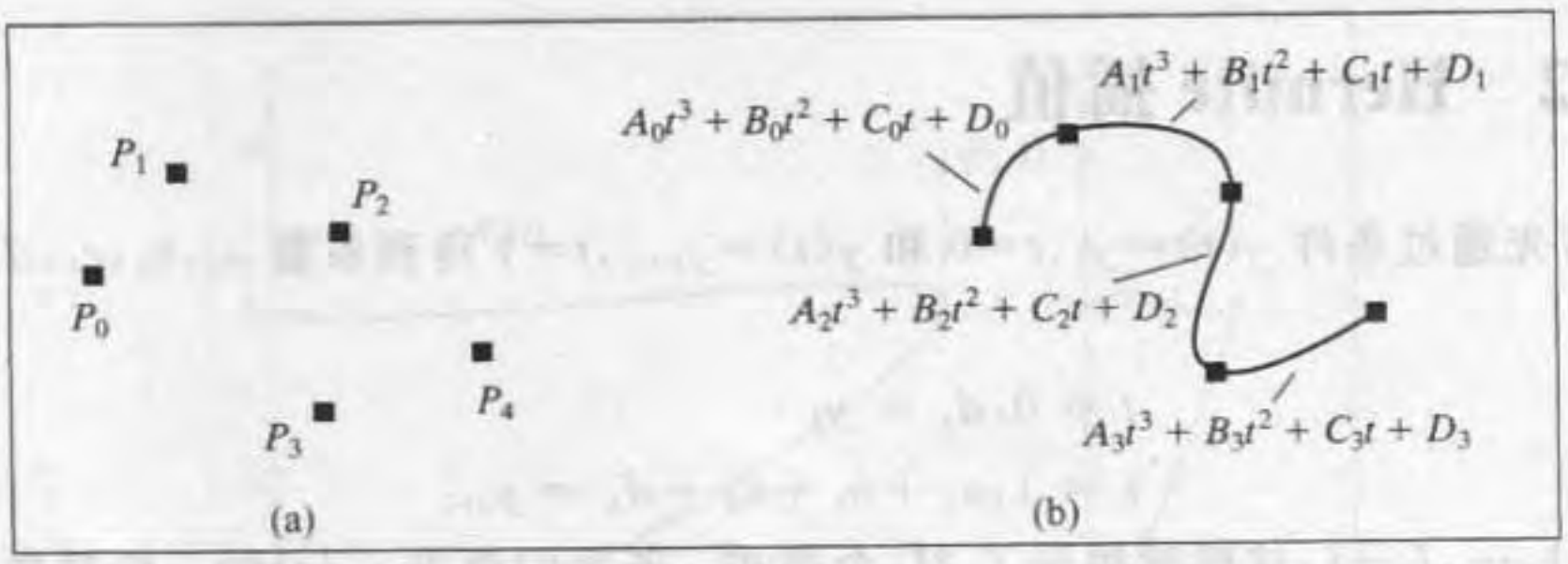


图 10.40　利用分段三次多项式插值

图 10.40(b)所示的曲线 $R(t)$ 解决了这个问题。它分为四段，每段都是一个三次多项式：

$$R_k(t) = A_k t^3 + B_k t^2 + C_k t + D_k, \quad k = 0,1,\cdots L, t \in [0,1] \tag{10.63}$$

当然每段都包含 x、y 两部分。问题是怎样确定系数 A_k, B_k, C_k, D_k 呢？

这里我们为了简单采用与前面不同的记号：每一段都定义在区间 $t \in [0,1]$ 上，这样画这条曲线时，就需要独立画出这四段，如下：

```
for(k=0; k<L; k++)
{
    求出系数 A[k],B[k],C[k],D[k]
    根据求得的系数分段画出曲线 Rk(t),每一段都是 t 从 0 到 1
}
```

$R_k(t)=(x(t),y(t))$ 是一 2D 曲线，如果 $R(t)$ 插值顶点 $P=(x,y)$，则 $x(t)$ 插值 x，$y(t)$ 插值 y。所以我们可以考虑如何使 x、y 分别插值，这样问题就被简化，且形象化了。

图 10.41(a)就以 y 坐标为例来解释，给定一系列 y 值即 $y_0, y_1, \cdots y_L$，我们要得到一个分段三次多项式 $y(t)$ 经过这些 y 值且一阶光滑(有时候还会要求二阶光滑，也就是说是一样条)。

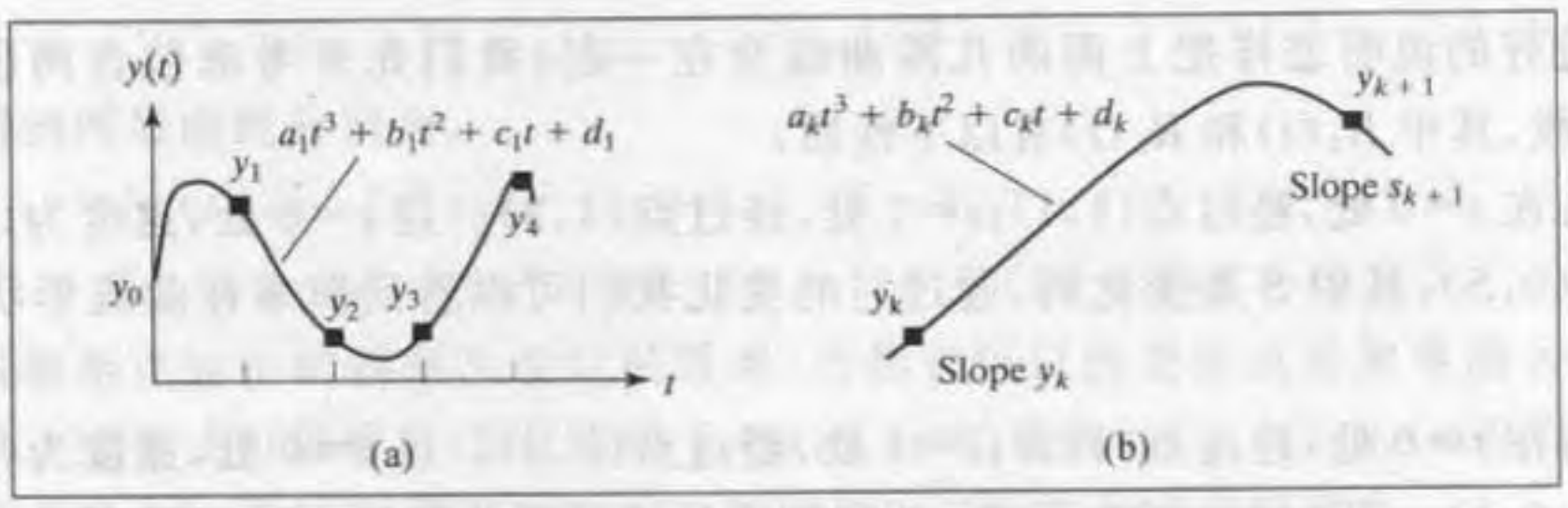


图 10.41　插值曲线的 y 部分

图 10.41(b)画出了曲线的第 k 段，它表示为：

$$y_k(t) = a_k t^3 + b_k t^2 + c_k t + d_k, \quad k = 0,1,\cdots L, t \in [0,1] \tag{10.64}$$

我们用 s_k 来表示它的一阶导数，即 $y_k'(0)=s_k$，有时候 s_k 是给定的，有时则是通过曲线需要满足的其他性质计算出来的。再问一次，系数如何确定？

10.10.2 Hermite 插值

我们首先通过条件 $y(t)=y_k, t=0$ 和 $y(t)=y_{k+1}, t=1$ 得到系数 a_k, b_k, c_k, d_k 应满足以下条件：

$$t=0: d_k=y_k$$
$$t=1: a_k+b_k+c_k+d_k=y_{k+1}$$

其中 $k=0,1,\cdots,L-1$，这样就得到了 $2L$ 个等式。若我们要求 $y_k(t)$ 的一阶导数 $y_k'(t)$ 在端点处等于给定的值，即满足条件 $y_k'(t)=s_k, t=0$ 和 $y_k'(t)=s_{k+1}, t=1$，因为 $y_k(t)$ 的导数为 $y'_k(t)=3a_kt^2+2b_kt+c_k$，得到：

$$t=0: c_k=s_k$$
$$t=1: 3a_k+2b_k+c_k=s_{k+1}$$

其中 $k=0,1,\cdots,L-1$，又得到了 $2L$ 个等式。这样对 $4L$ 个未知系数我们就得到了 $4L$ 个等式，而且通过给定每一段端点处的一阶导数使得曲线在每一段接点处保持一阶光滑。

通过上面 $4L$ 个等式我们可以直接求解得到系数 a_k, b_k, c_k, d_k（通过 y_k, s_k 表示）：

$$a_k=s_{k+1}+s_k-2(y_{k+1}-y_k)$$
$$b_k=3(y_{k+1}-y_k)-2s_k-s_{k+1}$$
$$c_k=s_k$$
$$d_k=y_k$$

其中 $k=0,1,\cdots,L-1$。这样就确定了每一段 $y_k(t)$，同样可以确定 $x_k(t)$，就可以画出分段三次多项式曲线 $R_k(t)$ 了。

给定每一个 y_k 点的斜率 s_k，曲线的坐标就可以完全确定，方法有多种。我们首先看到了 s_k 在曲线形状中所起的作用，接下来我们来研究怎样使得所得曲线二阶光滑，引出了三次样条。

例 10.10.1 正切和斜率

为了较好的说明怎样把上面的几段曲线合在一起，我们先来考虑包含两段 $R_0(t)$ 和 $R_1(t)$ 的曲线，其中 $R_0(t)$ 和 $R_1(t)$ 有以下性质：

$R_0(t)$：在 $t=0$ 处，经过点(1,1)；$t=1$ 处，经过点(4,3)。且 $t=0$ 处，速度为(1,0)；$t=1$ 处，速度为(0,S)，其中 S 是变化的，通过它的变化我们可以看到斜率在曲线形状中所起的作用。

$R_1(t)$：在 $t=0$ 处，经过点(4,3)；$t=1$ 处，经过点(0,3)。且 $t=0$ 处，速度为(0,S)；$t=1$ 处，速度为(0,1)。图 10.42 给出了 $S=1$ 这种特殊情况下的结果，不仅画出了原始曲线，还分别画出了曲线的四个组成成分 $x_0(t), x_1(t), y_0(t), y_1(t)$。

仔细观察上面两段曲线是怎样合为一段的，我们可以看到两段曲线是光滑过渡的，因为它们的速度是连续的。这是因为 $x(t)$ 和 $y(t)$ 都是连续的，且在接点处有连续的斜率，所以曲线是一阶光滑的。由于 $x(t)$ 的两段在接点处斜率都是零，所以它在接点处是水平地从左端过渡到右端的，$y(t)$ 也是光滑地从左到右的，在接点处斜率都为 1。

为了更清楚地了解每一部分的性质，我们给等式(10.63)以具体的数据，先以 y 一来

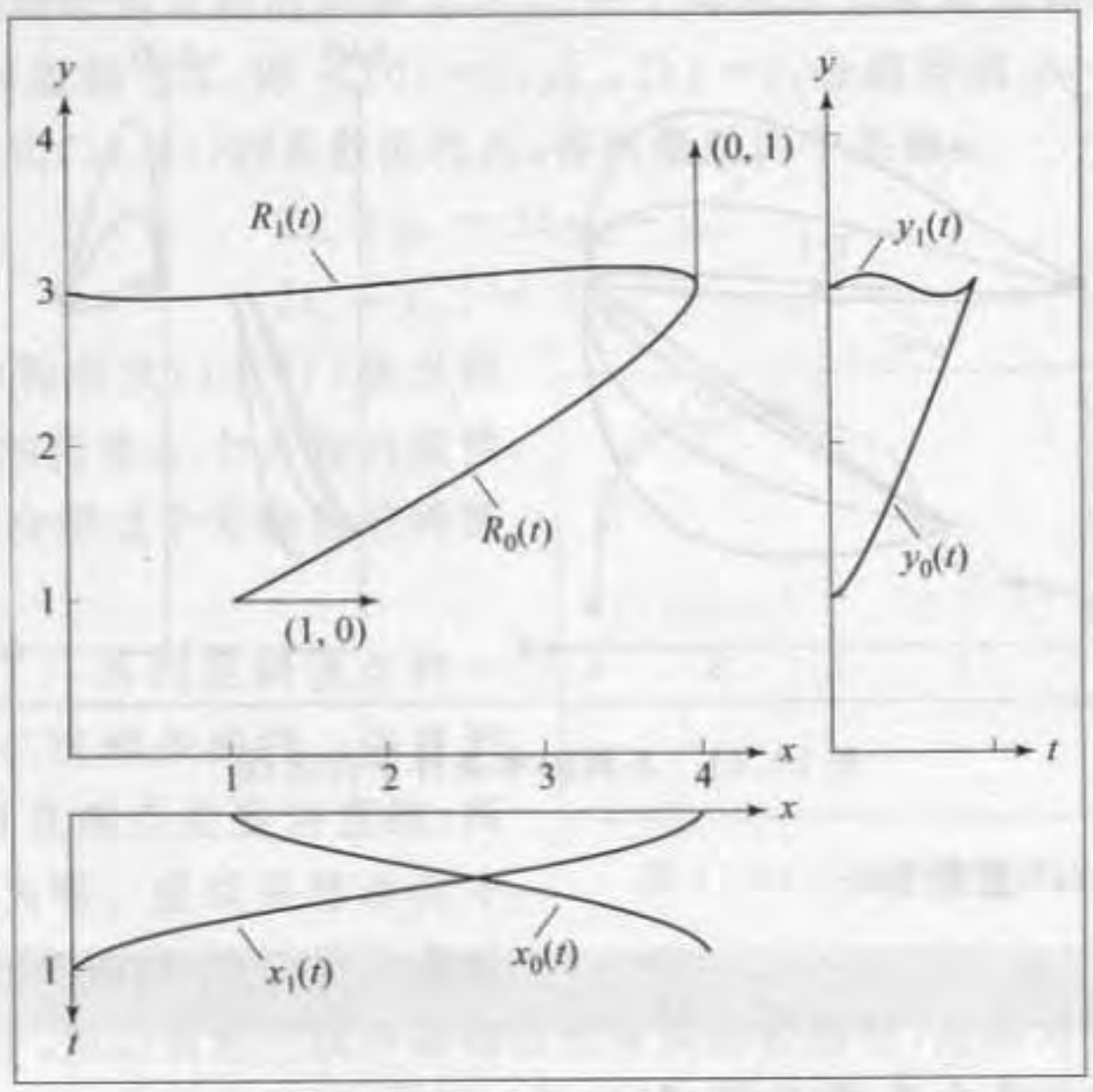


图 10.42　两段三次曲线的合成

说明：

第一段：$y_0=1, y_1=3, s_0=0, s_1=S$，得到三次多项式：

$$y_0(t)=(S-4)t^3+(6-S)t^2+1$$

第二段：$y_1=3, y_2=3, s_1=S, s_2=1$，得到三次多项式：

$$y_1(t)=(S+1)t^3-(2S+1)t^2+St+3$$

对于 x 一同样有：

第一段：$x_0=1, x_1=4, s_0=1, s_1=0$，得到三次多项式：

$$x_0(t)=-5t^3+7t^2+t+1$$

第二段：$x_1=4, x_2=0, s_1=0, s_2=0$，得到三次多项式：

$$x_1(t)=8t^3-12t^2+4$$

这样得到两段曲线分别为：

$$R_0(t)=-5t^3+7t^2+t+1,\quad (S-4)t^3+(6-S)t^2+1$$

$$R_1(t)=8t^3-12t^2+4,\quad (S+1)t^3-(2S+1)t^2+St+3$$

仔细观察接点处 y 的斜率 S 变化的效果，当然它仅仅改变接点处速度的大小，而不改变方向。从上述等式可以看出，S 仅影响 $y_0(t)$，$y_1(t)$ 的系数，对 $x_0(t)$，$x_1(t)$ 没有任何作用。图 10.43 给出了多个不同的 S 下函数 $y(t)$ 的图像以及整条曲线的形状。

从图上可以看出，斜率 S 越大，曲线在接点处的速率就越大。为了适应这个速率的变化，曲线需要在接近及离开接点时凸出以便在垂直方向上能快速通过接点，这也说明了三次曲线的“弹性”程度：因为没有足够的自由度，不能任意快的弯曲，所以会调节整个区间上的形状以满足端点处的斜率约束。

当 $S=0$ 时，曲线在接点处产生一个角点（见练习），$R_0(t)$ 和 $R_1(t)$ 在接近接点时速度都会变为零，所以即使它们的速度在接点处突然改变方向也仍然是连续变化的，所以曲线是一

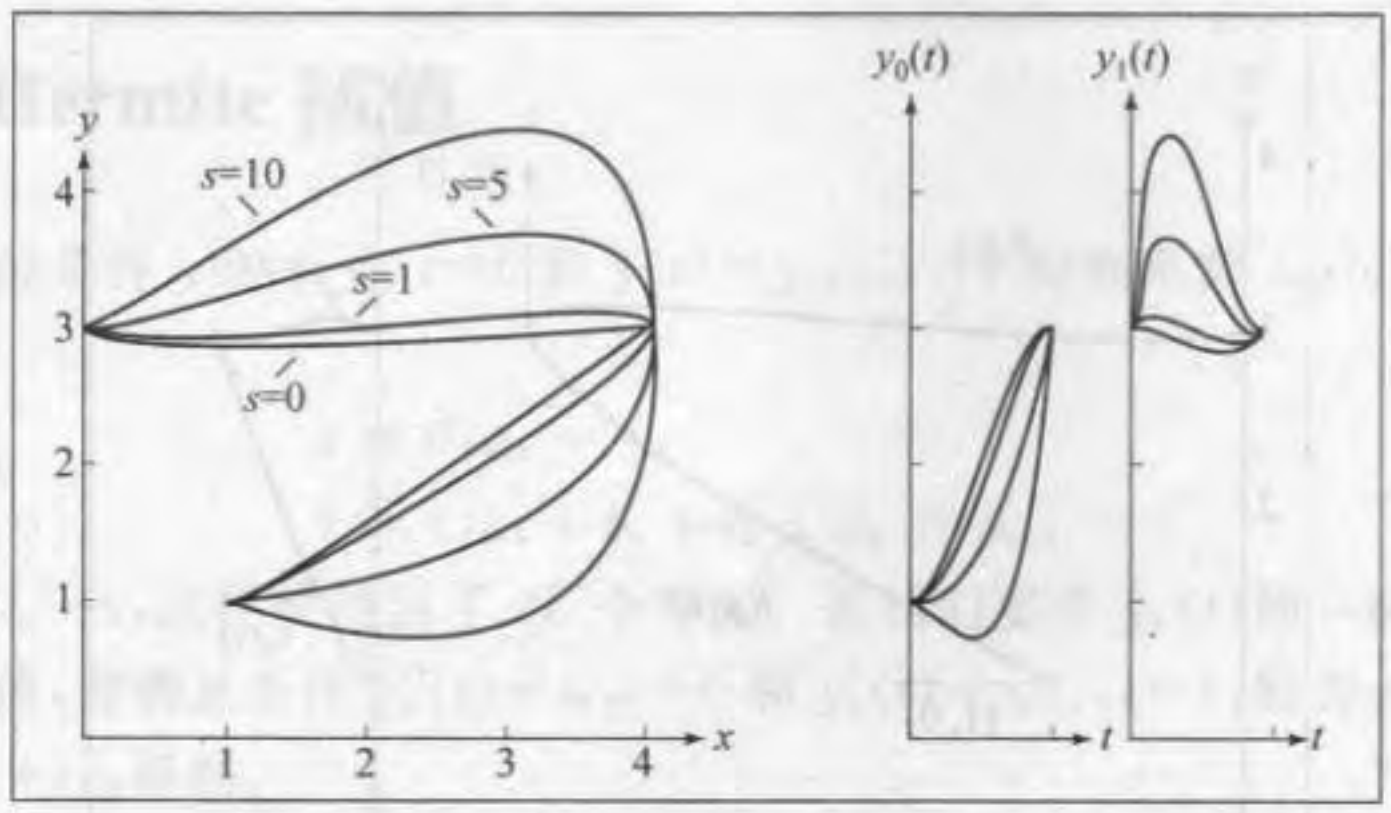


图 10.43 不同斜率条件下的曲线

阶连续的(问它是否 G^1 连续?)。

练习

练习 10.10.1 速度为零的情况

证明若三次曲线的两段在端点处速度均为零,则曲线就变成了直线。

练习 10.10.2 观察曲线

试写一个小程序来产生图 10.41 和图 10.42 中的曲线,然后改变 S 的值看曲线有什么变化。可以如下这样来定义函数:

```
Point2 y(float t, float S, int seg)
{
    //return y(t)
}
```

对 $x(.)$同样。

练习 10.10.3 环

若令 $S=-1$,则图 10.42 中的曲线将会出现一个环。

10.10.3 自然三次样条

上节我们已经知道,只要给定曲线在每个控制顶点处的斜率 s_k,就可以确定曲线的形状,称为 Hermite 插值。但是如果我们指定特殊的 s_k,就可以使得 $y(t)$的二阶导数在每一个内部点处连续(在端点处导数连续没有意义),其中 $k=1,2,\cdots,L-1$。每一段 $y_k(t)$的二阶导数为 $y_k''(t)=6a_kt+2b_k$,所以让 $y_{k-1}''(1)$和 $y_k''(0)$相等,得到:

$$6a_{k-1}+2b_{k-1}=2b_k,\quad k=1,2,\cdots,L-1 \tag{10.65}$$

用等式(10.63)的系数值代入并简化,得到关于斜率的条件:

$$s_{k-1}+4s_k+s_{k+1}=3(y_{k+1}-y_{k-1}),\quad k=1,2,\cdots,L-1 \tag{10.66}$$

可以看出相邻控制顶点的斜率具有上式的关系才能保证曲线在内部点处是二阶连续

的，但我们需要指定两个端点处的斜率 s_0, s_L。一个经典的方法是这样确定它们的，就是让两个端点处的二阶导数都为零，即 $y_0''(0)=0, y_{L-1}''(1)=0$，分别得到：$b_0=0, 3a_{L-1}+b_{L-1}=0$（为什么？）。也用等式(10.63)的系数值代入，得到最后两个条件：

$$\begin{aligned} 2s_0 + s_1 &= 3(y_1 - y_0) \\ 2s_L + s_{L-1} &= 3(y_L - y_{L-1}) \end{aligned} \tag{10.67}$$

将等式(10.66)和等式(10.67)联立得到一个关于斜率 s_k 的包含 L 个方程的线性方程组，后面练习将介绍这个方程组的特殊性质及它的求解。

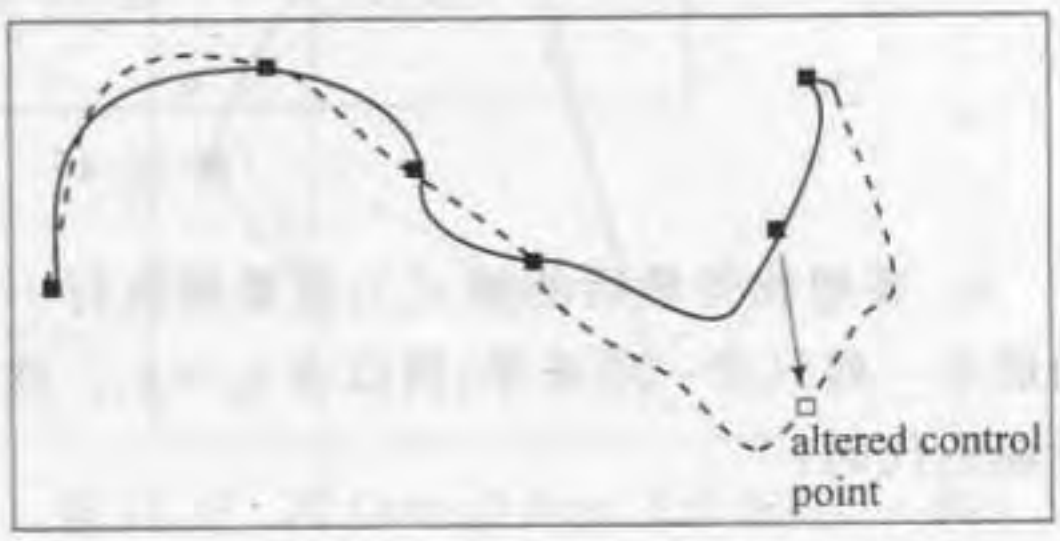


图 10.44　插值控制顶点的自然三次样条

图 10.44 给出了一系列控制顶点和一条插值它们的自然三次样条曲线。注意到曲线处处连续，而且在端点处变为直线，所以端点处二阶导数为零。虚线是移动其中一个控制顶点后得到的曲线，可以看出曲线的位置处处都移动了，所以自然三次样条曲线没有局部控制性，是因为它不仅固定了端点处的一阶导数，还要使得二阶导数连续，所以一个控制顶点的变化会引起整条曲线的变动。

练习

练习 10.10.4　求解自然三次样条中关于斜率的线性方程组

将由等式(10.66)和等式(10.67)组成的线性方程组写成矩阵的形式，比如当 $L=4$ 时，我们有：

$$(s_0, s_1, s_2, s_3, s_4)\begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} = 3(y_1 - y_0, y_2 - y_0, y_3 - y_1, y_4 - y_2, y_4 - y_3)$$

系数矩阵是三对角矩阵，即只有对角线及对角线两侧的值非零，这样的线性方程组可以分两步求解，方法如下：

a. 首先对这个方程组执行向前消去法，以消去上面一斜线上的 1。具体步骤为：首先第一个方程两边同除以 2，使其第一项系数为 1；其次从第二个方程开始，依次减去上一个方程的适当倍数，以消去其中的 1；然后缩放这个方程式使其对角线上的系数为 1。这样就把这个方程组转化成了“下三角”形式（即主对角线上面都为零）：

$$(s_0, s_1, s_2, s_3, s_4)\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ v_0 & 1 & 0 & 0 & 0 \\ 0 & v_1 & 1 & 0 & 0 \\ 0 & 0 & v_2 & 1 & 0 \\ 0 & 0 & 0 & v_3 & 1 \end{pmatrix} = (q_0, q_1, q_2, q_3, q_4) \tag{11.68}$$

其中的 $v_i, q_i, i=0,1,2,3,4$ 容易计算出来的。

b. 图 10.45 给出了前面向前消去法的执行代码，其中使用数组 $v[]$ 和 $q[]$。

```
v[0]=0.25;
q[0]=3.0*(y[1]-y[0])*v[0];
for(i=1; i<L; i++)
{
    v[i]=1.0/(4.0-v[i-1]);
    q[i]=(3.0*(y[i]-y[i-1])-q[i-1])*v[i];
}
```

图 10.45 向前消去法

c. 要想获得最后的解 $s[]$,需要再执行一个反向代入步骤,注意到等式(11.79)中矩阵的最后一列只有一项非零,所以有 $s_4=g_4$。然后依次向前代入每一个方程即可得到各个 s_i,下面是代码:

```
s[L]=q[L];
for(i=L-1; i>0; i--) s[i]=q[i]-v[i]*s[i+1];
```

10.10.4 三次样条插值中斜率的计算

对于三次样条插值中斜率的确定,除了上一节介绍的方法外,还有很多其他方法,这里我们只大致介绍目前最常见的一种,**Catmull-Rom** 方法。

我们仍然要求曲线在内部点一阶光滑,但是不再要求二阶光滑,主要是希望将条件放宽可以使得曲线设计者能够更好的局部控制曲线的形状。

上节的方法是通过曲线的二阶导数连续来确定它在控制顶点处的斜率,相反这里我们通过它的相邻控制顶点的位置来确定。

Catmull-Rom 方法是很简单的,就是让点 P_k 的速度向量 $P'(t_k)$ 与它的前一个及后一个顶点的位置相关,如图 10.46 所示,简单的说,就是与顶点 P_{k-1} 到 P_{k+1} 的向量成比例,这样使得曲线在顶点 P_k 的切线平行于它的前一个顶点到后一个的方向,即:

$$\mathbf{P}'(t_k)=m(P_{k+1}-P_{k-1}) \tag{10.69}$$

其中 m 是一常数(我们下面会来分析系数 m 的作用,通常情况下取 $m=1/2$)。按照以前的方式,只需要让函数 $y(t)$ 的斜率 $s_k=m(y_{k+1}-y_{k-1})$,对函数 $x(t)$ 也同样。

这样我们就确定了曲线在每一个内部控制顶点的斜率 s_k,进而确定了等式(10.32)中当 $k=1,2,\cdots,L-2$ 时三次多项式的系数,但是我们还需要两个条件来确定曲线在两个端点处的斜率即 s_0,s_L,可以通过使得端点处的二阶导数为零来得到等式(10.35),其他方法见练习。图 10.47 给出了通过此方法确定的一条曲线,注意到曲线在端点处是伸直的正是因为二阶导数为零。

增加张力控制

为了设计者可以更好的控制曲线在每一个接点处的形状,Catmull-Rom 方法还引入了张力参数 v_k,这可以使设计者改变等式(10.69)中的参数 m,进而在不改变 $\mathbf{P}'(t_k)$ 方向的前提下调整它的大小。

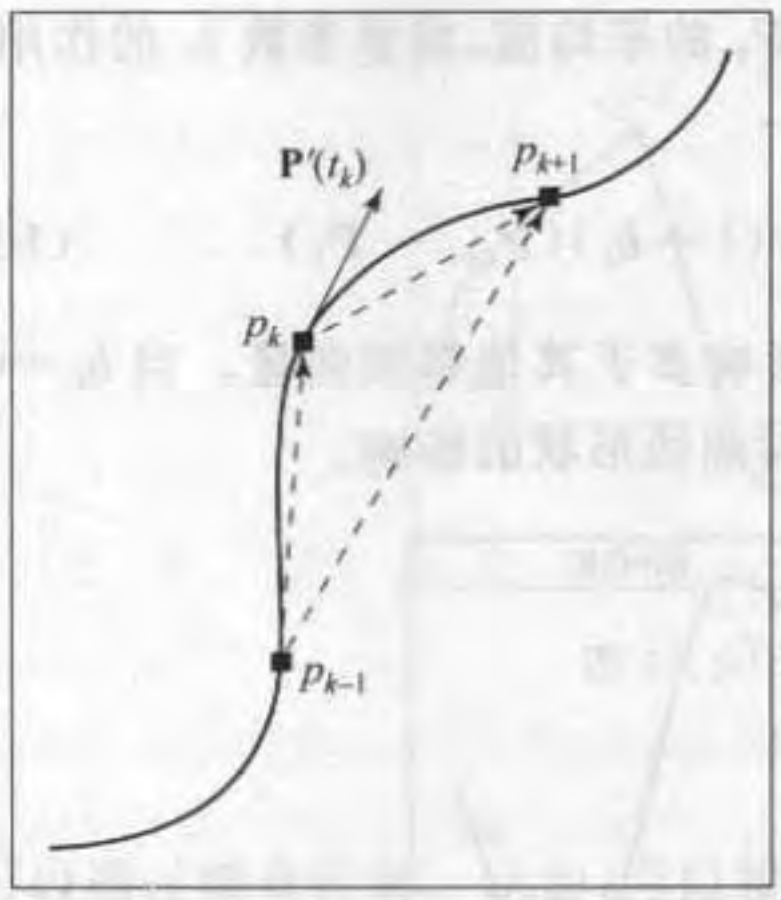


图 10.46　通过两个相邻顶点确定斜率

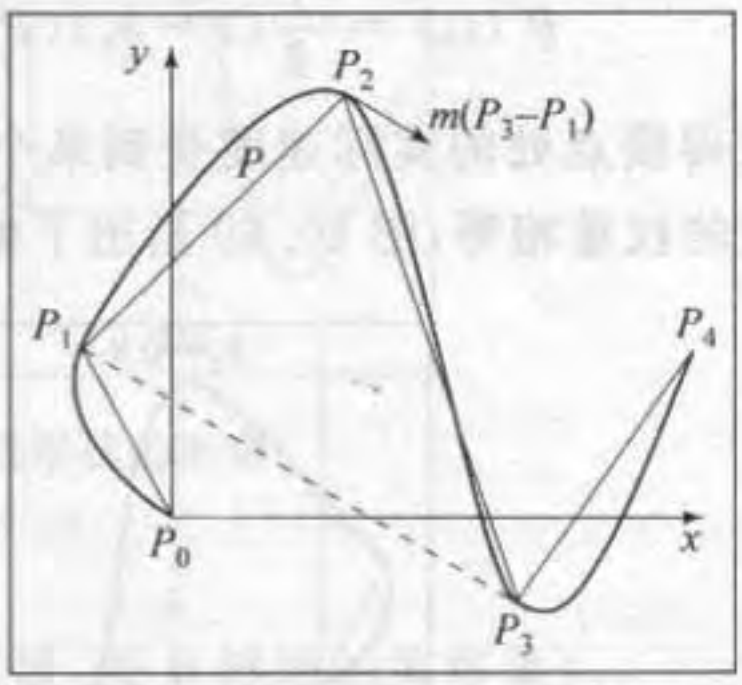


图 10.47　用 Catmull-Rom 方法得到的一条插值曲线($m=1/2$)

这样第 k 个控制顶点的速度向量为：

$$\mathbf{P}'(t_k)=\frac{1}{2}(1-v_k)(P_{k+1}-P_{k-1}) \tag{10.70}$$

其中 $k=1,2,\cdots,L-1$。(这样斜率 s_k 的值会有什么变化?)

参数 v_k 通常被控制在区间[$-1,1$]上，如当 $v_k=0$ 时，对应着等式(10.69)中 $m=\frac{1}{2}$。图 10.48 给出了顶点 P_2 处的张力参数对曲线形状的影响，图 10.48(a)中曲线的张力参数是 1，所以在接点处速度为零，而且在接近接点处曲线变直(为什么?)。图 10.48(b)中曲线的张力参数是 -1，所以曲线在接点处"松弛"。

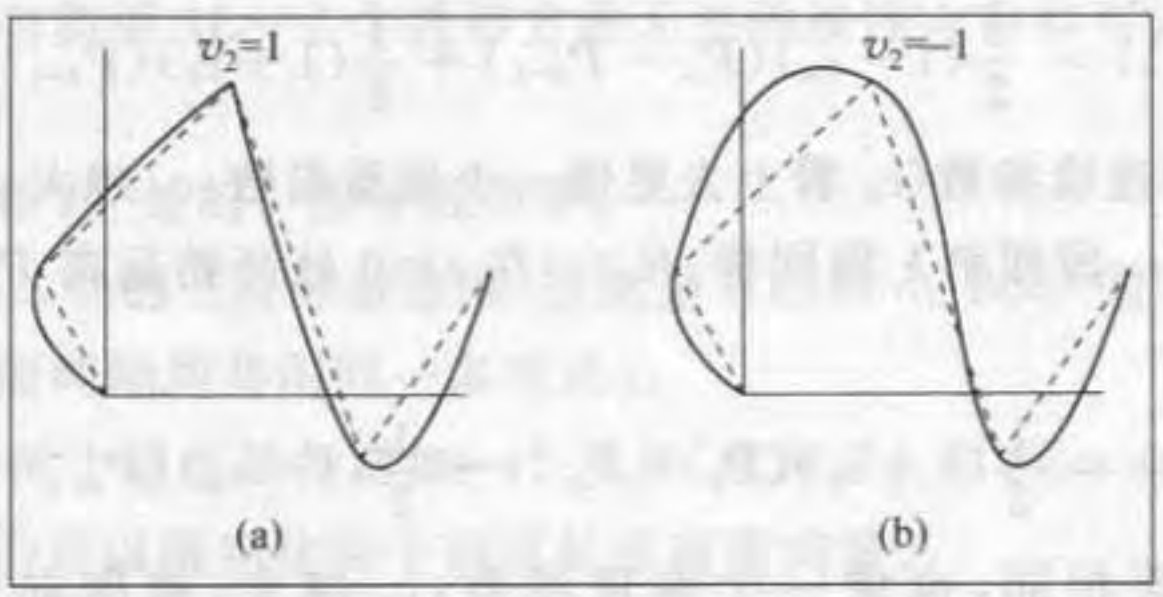


图 10.48　张力参数对曲线形状的影响

(a) 张力参数=1；(b) 张力参数=-1

增加偏重控制

经典曲线 Kochanek-Bartels 样条就是额外增加了一个参数以更好的控制曲线的形状【Kochanek84】，其中之一就是"偏重"。我们可以将等式(10.69)改写一下$\left(m=\frac{1}{2}\right)$：

$$\mathbf{P}'(t_k)=\frac{1}{2}(P_k-P_{k-1})+\frac{1}{2}(P_{k+1}-P_k) \tag{10.71}$$

即表示成与它相邻的两个向量 P_k-P_{k-1}，$P_{k+1}-P_k$ 的平均值，偏重参数 b_k 的作用就是使得这两项不再平均，即：

$$\mathbf{P}'(t_k)=\frac{1}{2}(1-b_k)(P_k-P_{k-1})+\frac{1}{2}(1+b_k)(P_{k+1}-P_k) \tag{10.72}$$

这使得接点处的实际速度受到某个邻接向量的影响多于其他邻接向量。当 $b_k=0$ 时，两个向量的权重相等，图 10.49 给出了偏重参数变化对曲线形状的影响。

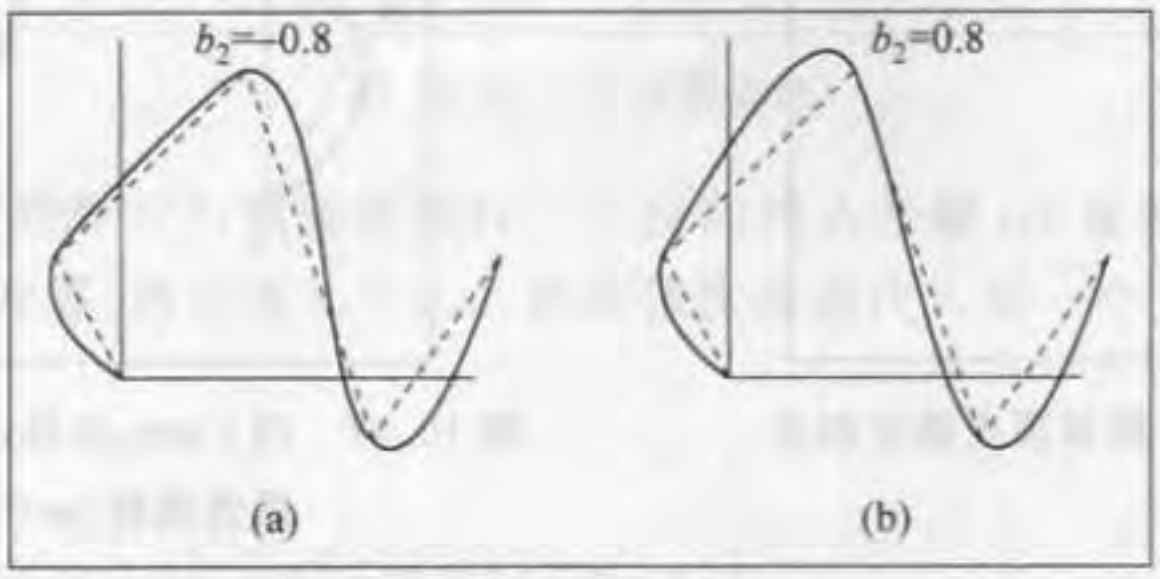


图 10.49 偏重参数对曲线形状的影响

(a) 偏重参数=-0.8；(b) 偏重参数=0.8

增加连续性控制

Kochnaek 和 Bartels 引入另一参数，设计者通过这个参数，可以控制接点处之前的速度不同于接点之后的，而不同于以前让速度向量在接点处连续。以接点 P_k 来说，它的前一段即第 $k-1$ 段，$R_{k-1}(t)$(见等式 10.59)在 $t=1$ 处到达 P_k，我们在此处设置它的速度向量为：

$$\mathbf{R}'_{k-1}(1)=\frac{1}{2}(1-c_k)(P_k-P_{k-1})+\frac{1}{2}(1+c_k)(P_{k+1}-P_k) \tag{10.73}$$

这里的 c_k 就称为连续参数(c_k 看上去更像一个偏重参数：c_k 越大，速度向量越接近向量 $P_{k+1}-P_k$)。对 P_k 后一段即第 k 段同样，$R_k(t)$ 在 $t=0$ 处开始远离 P_k，我们在此处设置它的速度向量为：

$$\mathbf{R}'_k(0)=\frac{1}{2}(1+c_k)(P_k-P_{k-1})+\frac{1}{2}(1-c_k)(P_{k+1}-P_k) \tag{10.74}$$

这里的 c_k 与前面相同，也像一个偏重参数：c_k 越大，速度向量越接近向量 P_k-P_{k-1}。如果 $c_k=0$，这两个速度向量相同，曲线在接点处一阶光滑。当 c_k 不等于零时，这两个向量具有不同的大小和方向。图 10.50 给出了这个连续参数对曲线形状的影响。

张力、偏重、连续性这三个参数可以同时被应用到实际中(见练习)，设计者可以在接点处分别设置这三个参数的值。典型应用中设计者遵守以下步骤：

1. 用鼠标点击确定初始控制顶点。

2. 检查这样得到的曲线，符合条件就停止。

3. 不符合条件的话，编辑控制顶点，调整每个控制顶点的张力，偏移量和连续性参数(设计者可以点击一个控制顶点，将它拖动到新的位置。然后在这个顶点调整它的各个参

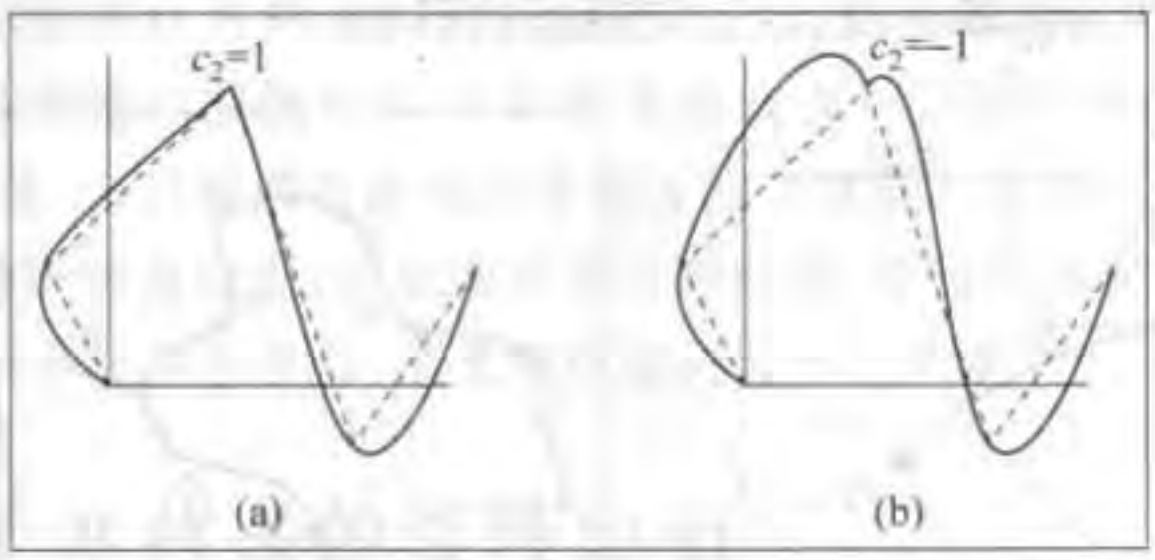


图 10.50　连续参数对曲线形状的影响

(a) $c_2=1$；(b) $c_2=-1$

数，可以通过键盘控制。比如，可以按 b 键减少偏移量，按 B 键增加偏移量）。

4. 返回到第二步。

练习

练习 10.10.5　综合的 Kochnaek-Bartels 样条

综合考虑张力、偏移量和连续性三个参数的影响，来写出 Kochnaek-Bartels 样条 $\mathbf{R}'_{k-1}(1)$ 和 $\mathbf{R}'_k(0)$ 的等式。

练习 10.10.6　选择端点条件

前面在 $L+1$ 个控制顶点上三次插值的时候，我们需要确定 $4L$ 个系数可，其中 $4L-2$ 个可以通过插值条件和斜率约束来确定，有 2 个系数待定。等式(10.66)是通过要求两个端点的二阶导数为零来确定这 2 个系数的，但是还有很多其他条件可以用来确定它们，只需要有两个关于系数的且与前面 $4L-2$ 个线形方程无关的线性方程即可，以下几个条件均满足此要求：

a. 两个端点 P_0 和 P_L 处的一阶导数为零。

b. 顶点 P_1 和 P_{L-1} 处的三阶导数连续（这就是有名的 de Boor 的“非节点”约束，它使得曲线的前面两段和后面两端均共用同一多项式）。

c. 在控制多边形两个端点处各增加一个顶点（记为 P_{-1} 和 P_{L+1}），曲线仍然从 P_0 开始，P_L 结束，但是这样我们可以得到这两个顶点处的速度向量。

练习 10.10.7　通过控制顶点的抛物线

为了计算顶点 P_k 处的斜率，我们可以先求出经过点 P_{k-1}，P_k，P_{k+1} 的抛物线，然后据此计算出点 P_k 处的速度，对于任意这样的三个顶点，都可以求出速度的表达式。

10.10.5　交互指定切向量

一个 CAD 绘图程序可能需要给设计者提供可以控制切向量的可视“控制柄”，如图 10.52 所示。用户指定控制顶点后，就会有一条初始插值曲线，这时用户可以通过拖动鼠标来限定某个顶点处的切向量方向，然后就会产生一条新曲线取代原来的曲线。这样的可

视化交互让用户可以快速编辑曲线,以达到理想的形状。

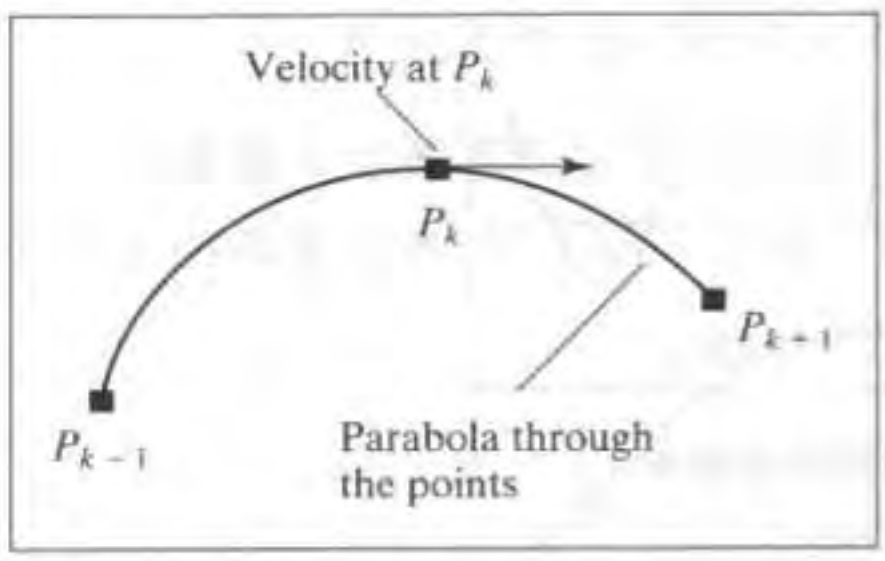


图 10.51 由经过三个相邻顶点的抛物线来确定中间顶点处的速度

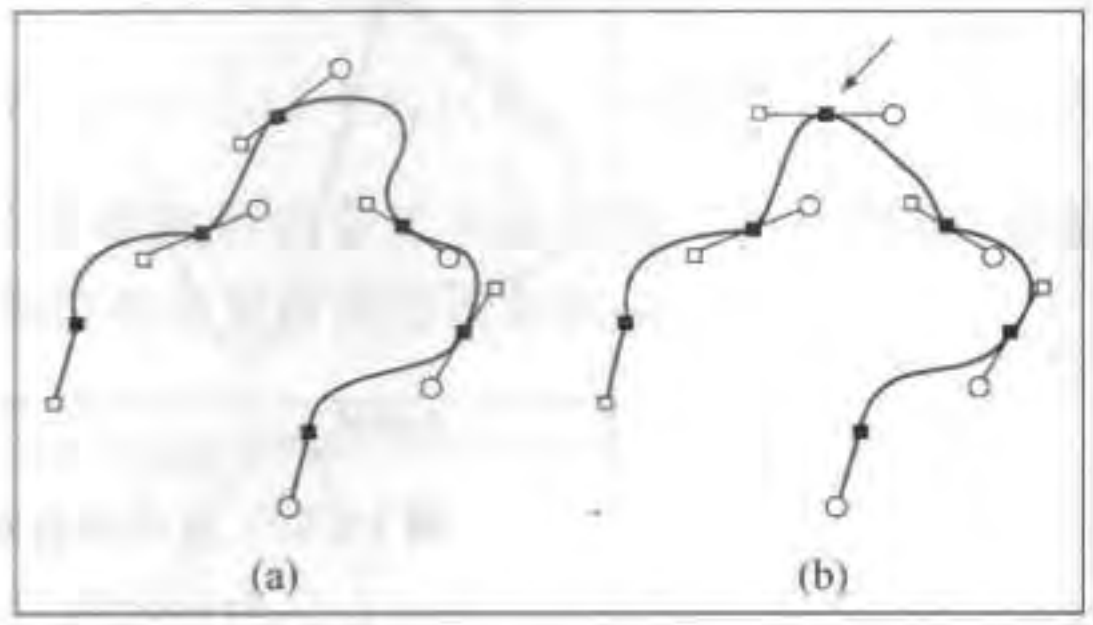


图 10.52 三次曲线段的交互设计

10.11 自由曲面造型

迄今为止,我们已经研究了如何表示和构造二维及三维上的曲线,那么显然对这些思想的推广可以得到曲面的构造方法。在第 6 章我们介绍了一系列不同类型的曲面,如直纹面、双线性曲面片、Coons 曲面片以及旋转曲面等。现在我们来研究如何用贝塞尔曲线和B样条曲线来设计生成这些曲面,从而为这些曲面的构造提供强有力的工具。

10.11.1 基于 B 样条的直纹面

直纹面是一类非常便于使用的曲面。回忆一下 6.5.6 节中的内容,直纹面由两条末端曲线 $P_0(u)$和 $P_1(u)$决定,且每一个 u 所确定的两点用直线相连后就生成了相应直纹面。因此直纹面的参数表达式就是两条曲线上相应点的线性插值(或内插)。为方便起见,重复等式(6.35),我们有

$$P(u,v) = (1-v)P_0(u) + vP_1(u) \tag{10.75}$$

这里我们把 $P_0(u)$和 $P_1(u)$扩展为 B 样条(或贝塞尔)曲线。图 10.53 显示了末端曲线 $P_0(u)$和$P_1(u)$为三次贝塞尔曲线的直纹面。$P_0(u)$的 4 个控制顶点为 $P_0^0, P_1^0, P_2^0, P_3^0$,$P_1(u)$的 4 个控制顶点为 $P_0^1, P_1^1, P_2^1, P_3^1$。根据贝塞尔曲线的等式(10.25),可以把这个曲面写作:

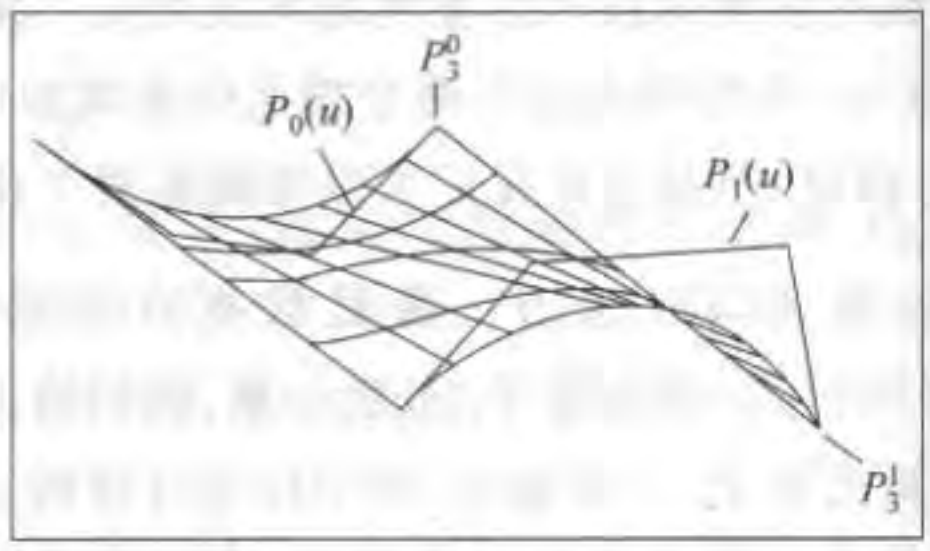


图 10.53 基于贝塞尔曲线的直纹面

$$P(u,v) = \sum_{k=0}^{3}((1-v)P_k^0 + vP_k^1)B_k^3(u) \tag{10.76}$$

现在来看看这个曲面的特性,u 线是连接两条对边贝塞尔曲线上相应两点的直线,v 线

是贝塞尔曲线，它的控制顶点为两条对边贝塞尔曲线上相应控制顶点的内插$(1-v)P_k^0+vP_k^1$。上述的两条末端贝塞尔曲线可以用 B 样条曲线或 NURBS 曲线代替，甚至它们可以定义在不同的节点向量上，只要两条曲线的参数 u 定义在同一个区间即可。

这样，所有比较特殊的直纹面，比如圆锥面和圆柱面，都可以被简单的表达。例如，如果 $P_0(u)$是 $P_1(u)$在空间中的一个平移，那么直纹面会成什么形状？

10.11.2　基于 B 样条的旋转曲面

回忆 6.5.7 节中的内容，轮廓线绕着轴线旋转扫描形成的扫掠面就是旋转曲面。若 $C(v)=(X(v),Z(v))$绕 z 轴扫掠，生成的旋转曲面其参数表达式为等式(10.77)。我们假设轮廓线 $P(u)=(x(u),y(u))$在 x 轴和 y 轴上有分量，那么在它绕着 y 轴扫掠后生成的旋转曲面其参数表达式为：$P(u,v)=(x(u,v),y(u,v),z(u,v))=(x(u)\cos(v),y(u),x(u)\sin(v))$。

$$P(u,v)=(X(v)\cos(u),X(v)\sin(u),Z(v)) \tag{10.77}$$

为方便起见，轮廓线通常用贝塞尔曲线或 B 样条曲线表示。我们选择 $L+1$ 个控制顶点(X_k,Z_k)来构造曲线：

$$(X(v),Z(v))=\sum_{k=0}^{L}(X_k,Z_k)N_{k,m}(v) \tag{10.78}$$

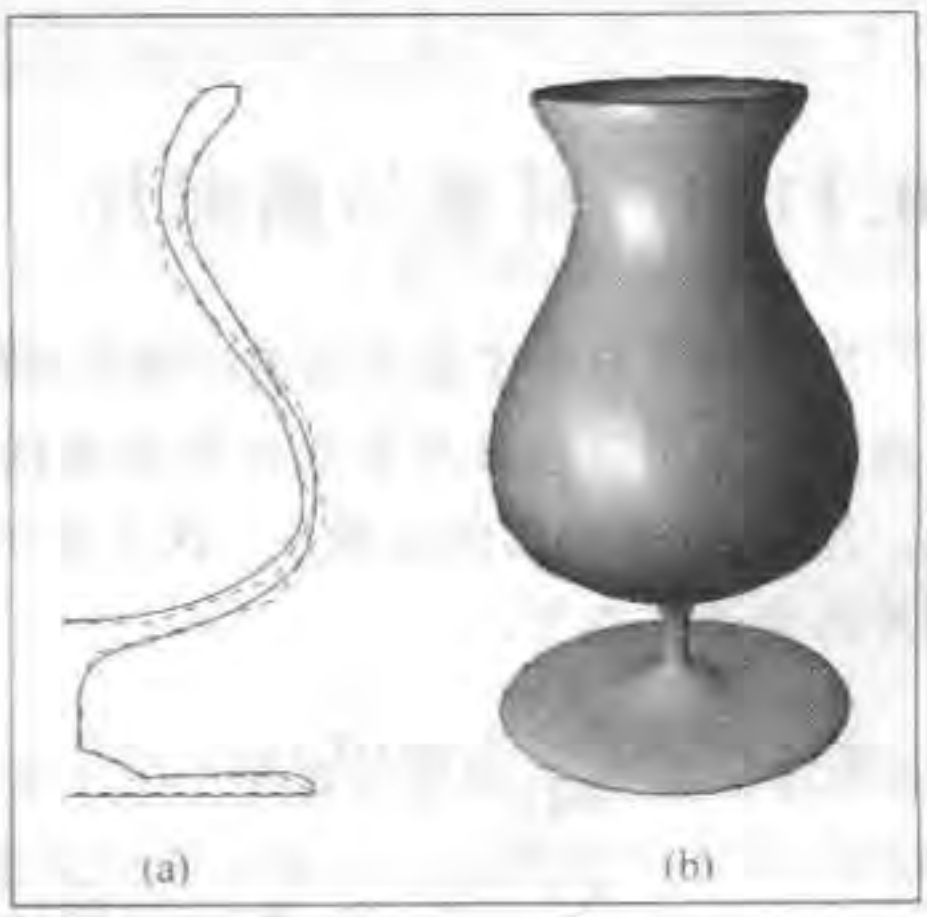


图 10.54　轮廓线为 B 样条曲线

(a) 高脚玻璃杯的轮廓线；

(b) 旋转曲面生成的高脚玻璃杯

图 10.54(a)中用三次 B 样条曲线定义了一个高脚玻璃杯的轮廓线，图 10.54(b)显示了由此生成的旋转曲面(在第 6 章中介绍了旋转曲面的网格对象，则这个网格的绘制可以应用 Mesh::draw()方法)。

例 10.11.1　经典的茶壶

轮廓线也可以用贝塞尔曲线表示。图 10.55(a)显示了最初由 Martin Newell【Blinn96】设计的茶壶壶体的轮廓线。此轮廓线由 3 条贝塞尔曲线组成，其 10 个控制顶点如图 10.56 所示。第一段贝塞尔曲线的控制顶点为点 0、1、2、3，第二段为点 3、4、5、6，第三段为点 6、7、8、9。注意到每条曲线最后一部分与下一段曲线第一部分共线(译者注：应为“每条曲线段控制顶点的最后一部分与下一段曲线控制顶点的第一部分共线”)，这个条件保证了不同的贝塞尔曲线合成一条曲线时满足导矢向量(译者注：应为“向量方向”)连续(见 10.1.2 节)。

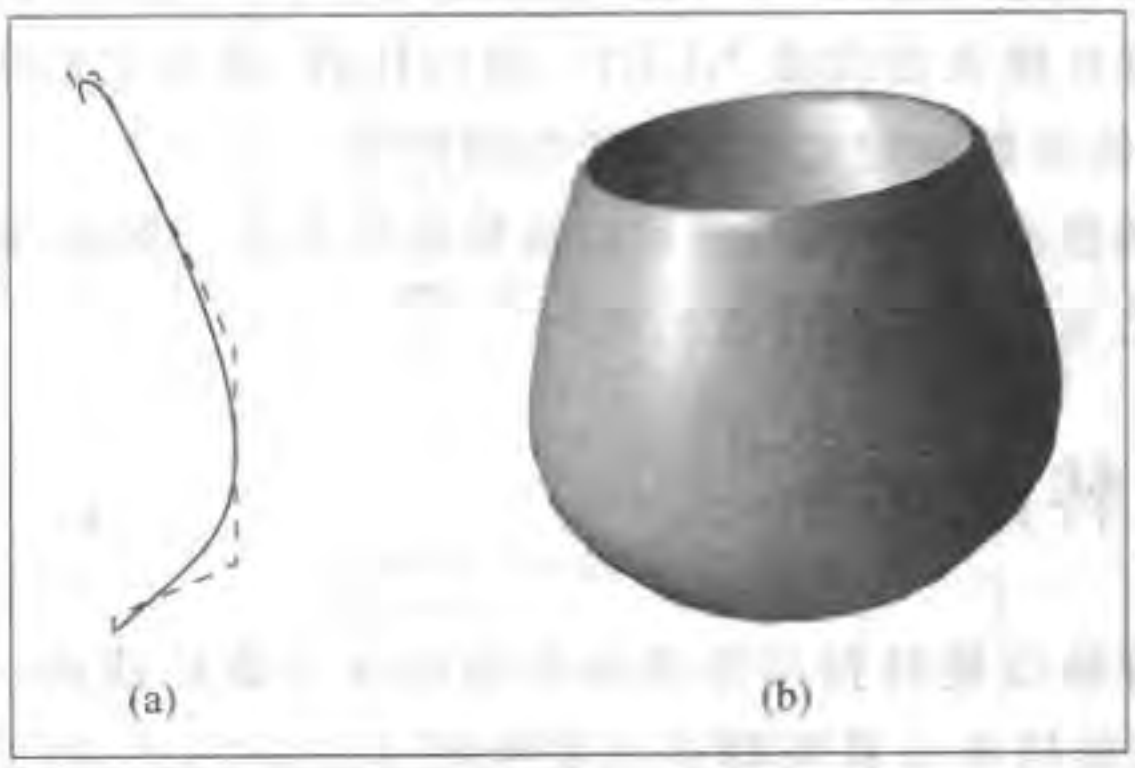


图 10.55 茶壶壶体轮廓线的贝塞尔曲线表示

(a) profile of body; (b) the teapot

i	x	z
0	1.4	2.25
1	1.3375	2.38125
2	1.4375	2.38125
3	1.5	2.25
4	1.75	1.725
5	2	1.2
6	2	0.75
7	2	0.3
8	1.5	0.075
9	1.5	0

图 10.56 茶壶壶体轮廓线数据点

10.11.3 贝塞尔曲面片

图 10.57 显示了基于两条贝塞尔曲线的直纹面。为了设计时更加灵活,等式(10.76)中线性的 u 线可以用贝塞尔或 B 样条曲线代替。贝塞尔曲面片的 u 线和 v 线都是贝塞尔曲线。比如,如果曲面的 u 线是二次贝塞尔曲线,而 v 线是三次贝塞尔曲线,那么此贝塞尔曲面片的表达式如下:

$$P(u,v)=\sum_{k=0}^{3}P_k(v)B_k^3(u)$$
$$=\sum_{k=0}^{3}\left(\sum_{i=0}^{2}P_{i,k}B_i^3(v)\right)B_k^3(u) \quad (10.79)$$

这里,u、v 值在 0 到 1 之间变化。图 10.57 给出了一个例子。

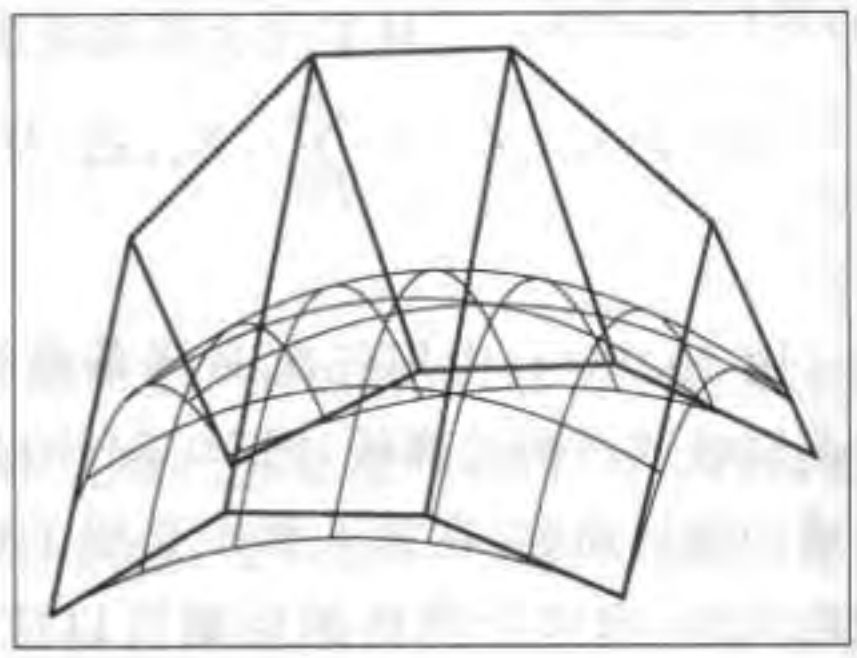


图 10.57 贝塞尔曲面片的一个实例

每一条 v 线都是关于 u 的贝塞尔曲线,它的 4 个控制顶点都位于一条二次贝塞尔曲线上。(这些 u 线该如何描述?)这 12 个控制顶点一起构成了**控制多面体**,它们决定了一个贝塞尔曲面片的形状。贝塞尔曲面片的表达式如等式(10.80):

$$P(u,v)=\sum_{k=0}^{L}\sum_{i=0}^{M}P_{i,k}B_i^M(u)B_k^L(v) \quad (10.80)$$

通常这个控制多面体一般就是由$(M+1)(L+1)$个顶点组成的控制网格。为了生成一个贝塞尔曲面片,设计者必须仔细设置各个顶点的位置来决定曲面的形状。

图 10.58 显示了一个**双三次**贝塞尔曲面片(即 L 和 M 都是 3)及其控制多面体。关于贝塞尔曲面的更多细节,可参看【Rogers90】。ATI(www.ati.com)和 NVIDIA(www.nvidia.com)也都提供关于贝塞尔曲面建模的既有意思又有深度的讨论。

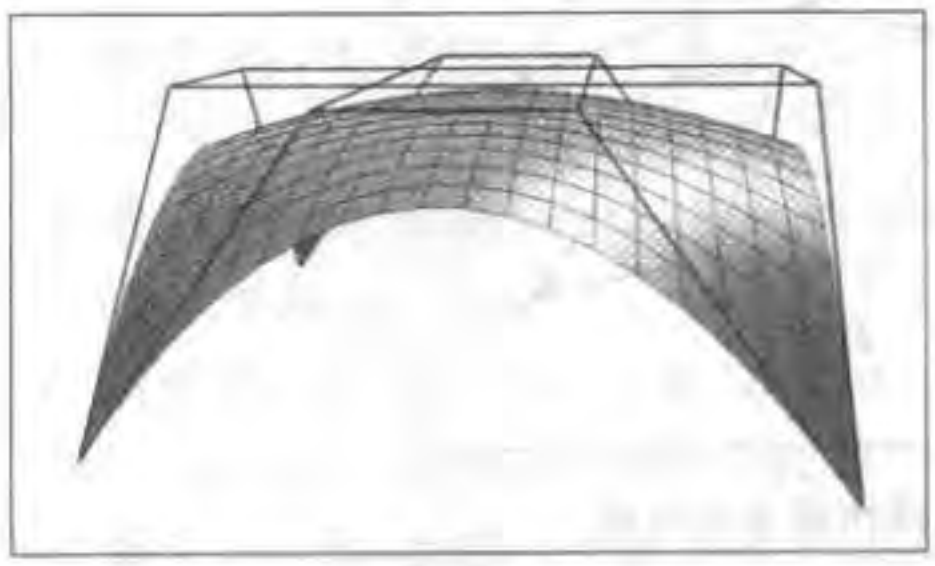


图 10.58　双三次贝塞尔曲面片及其控制多面体

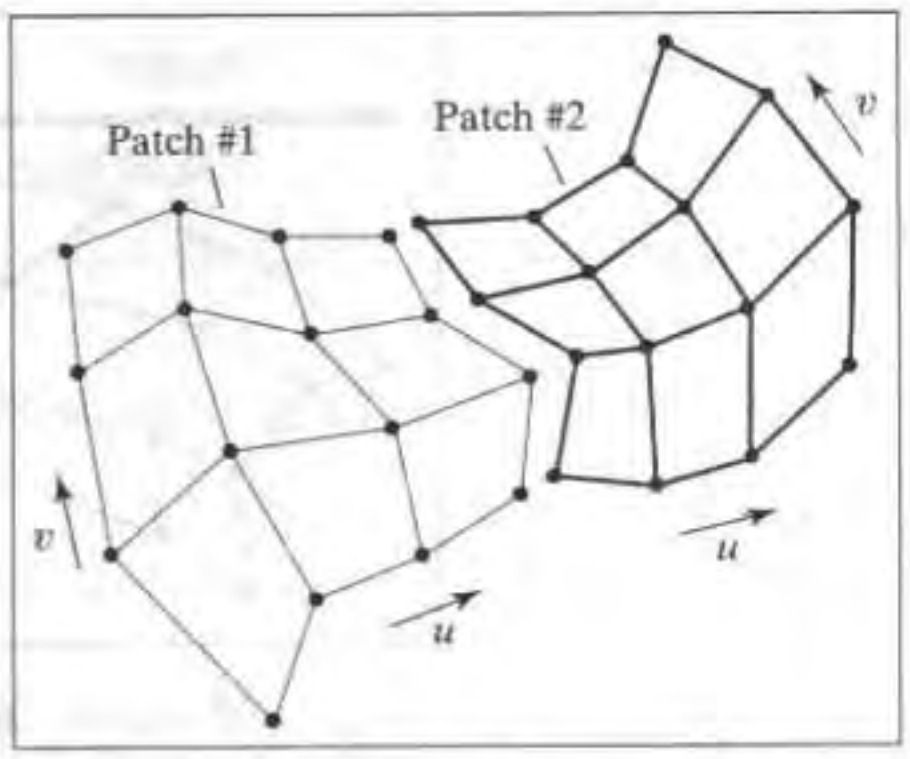


图 10.59　两个连续过渡的贝塞尔曲面片

练习

练习 10.11.1　贝塞尔曲面片的法线

把贝塞尔曲面片的参数表达式代入等式(6.25)，得到一个复杂的关于贝塞尔曲面法向量的表达式。请从形式上尽可能的进行化简。问法向量是否在整个(u,v)上连续变化？

10.11.4　贝塞尔曲面片的拼接

设计人员在对复杂外形进行造型时，也许需要用到多个贝塞尔曲面片，并且它们在公共边界上能够光滑拼接。图 10.59 显示了两个用于定义贝塞尔曲面片的控制多面体，分别为黑色和灰色。对两个控制多面体都应用等式(10.80)，当 u 和 v 都从 0 变化到 1 时就生成了相应的贝塞尔曲面片。这时它们只有控制网格不同而已。

为了两个曲面片能够无缝拼接，两个相应控制多面体必须满足怎样的条件？事实上很简单，只要两个曲面片的边界重合，即只需两个控制网格在边界处的点重合。这是由于边界上的曲线形状只由控制多面体上的相应边界点决定(可考虑等式(10.80)在 $u=0$ 时的情况)。所以设计者只需取两个曲面片的边界有相同的控制多边形即可。

相比较而言，在连接处保持切线(译者注：应为“切线方向”)连续更为困难(但是一旦切线方向连续，那么也就保证了曲面法线方向在连接处连续——这是为什么？)一个充分条件如图 10.60 所示：两个控制多面体中两条相交且交点在边界上的边，比如 E 和 E1，必须共线。但设计者很难做到这一点，更多难度稍低的要求详见 Faux 和 Pratt【Faux79】。

例 10.11.2　茶壶把手的设计

我们已知茶壶的壶身和壶盖是基于贝塞尔曲线的旋转曲面，而茶壶的把手和壶嘴，每一个却都是由四个贝塞尔曲面片构成的。这里我们来示范把手是如何构造的【Blinn96】，壶嘴的构造详见案例 10.8。

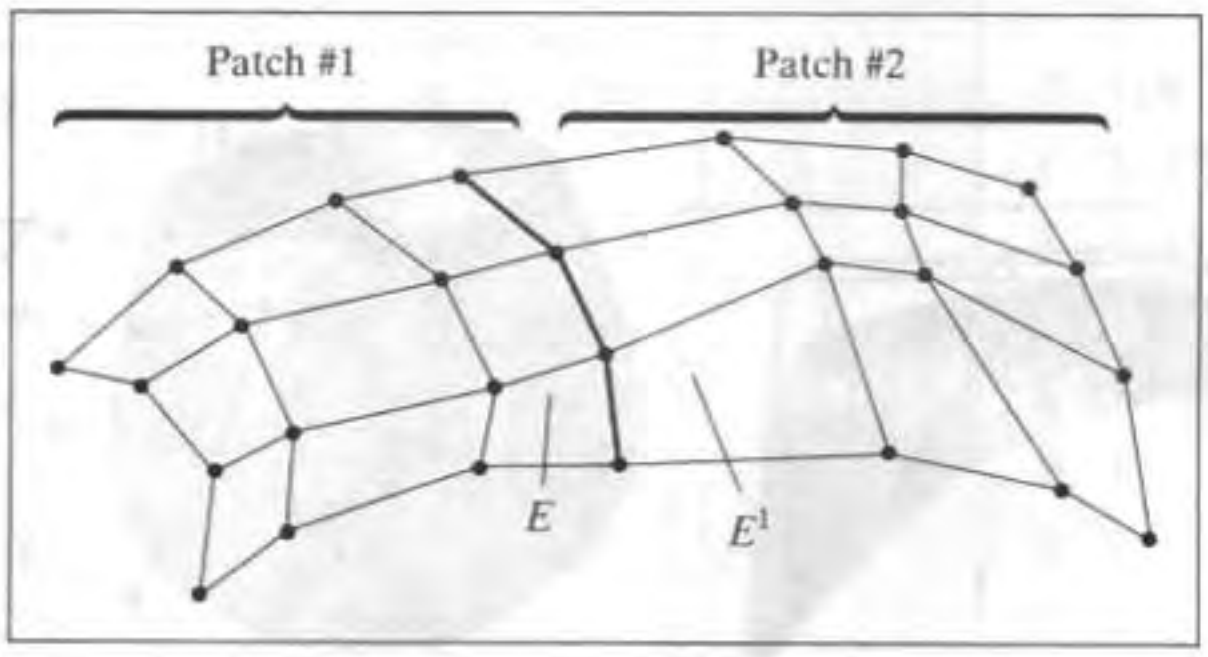


图 10.60 在连接处保持切线方向连续

图 10.61(a)从三维的视角显示了茶壶的把手,图 10.61(b)显示了它的横截面,以及横截面处相应的控制多面体。

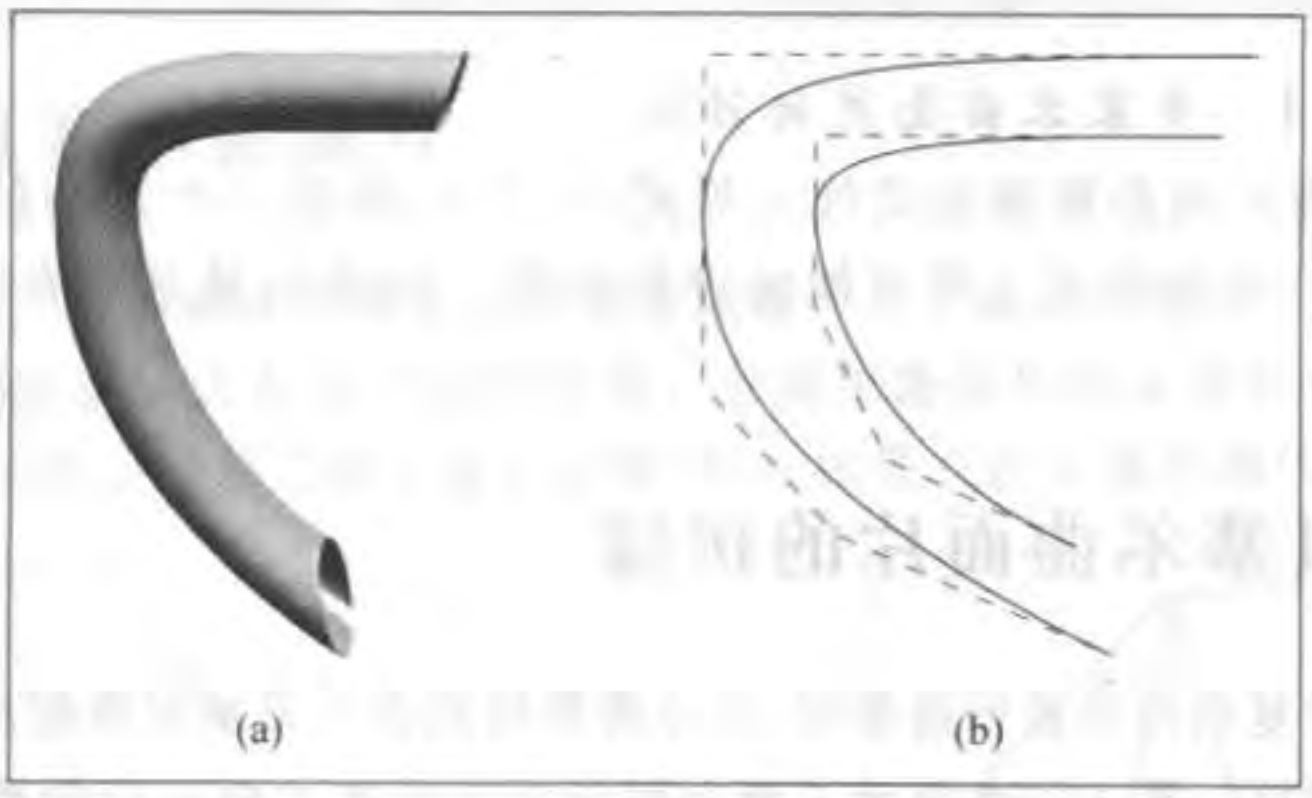


图 10.61 茶壶把手的设计

把手曲面是关于 xz 平面对称的,在 xz 平面的 y 轴正方向一侧有上下两个曲面片,同时类似镜面反射,xz 平面的 y 轴负方向一侧也有上下两个曲面片。对于 y 轴正方向一侧的上曲面片的控制多面体,是由 4 个竖立在 xz 平面上的矩形组成,这些矩形延伸到正八分圆 0.3 个单位距离处。对于第一个矩形,在 $y=0$ 和 $y=0.3$ 时使用 A_0 点各一次,同样,$y=0.3$ 和 $y=0$ 时使用 B_0 点各一次。当 $A_0=(-1.65, 1.875)$ 和 $B_0=(-1.5, 2.1)$ 时,我们得到四个控制顶点(−1.65,0.0,1.875),(−1.65,0.3,1.875),(−1.5,0.3,2.1),(−1.5,0.0,2.1)。

整个在 y 轴正方向一侧的上曲面有 16 个控制顶点:

- 基于 A_0,B_0 点的:(−1.6,0.0,1.875),(−1.6,0.3,1.875),(−1.5,0.3,2.1),(−1.5,0.0,2.1)
- 基于 A_1,B_1 点的:(−2.3,0.0,1.875),(−2.3,0.3,1.875),(−2.5,0.3,2.1),(−2.5,0.0,2.1)
- 基于 A_2,B_2 点的:(−2.7,0.0,1.875),(−2.7,0.3,1.875),(−3.0,0.3,2.1),(−3.0,0.0,2.1)

- 基于 A_3，B_3 点的：(−2.7，0.0，1.65)，(−2.7，0.3，1.65)，(−3.0，0.3，1.65)，(−3.0，0.0，1.65)

同样，整个在 y 轴正方向一侧的下曲面有 16 个控制顶点：

- 基于 A_3，B_3 点的：(−2.7，0.0，1.65)，(−2.7，0.3，1.65)，(−3.0，0.3，1.65)，(−3.0，0.0，1.65)
- 基于 A_4，B_4 点的：(−2.7，0.0，1.425)，(−2.7，0.3，1.425)，(−3.0，0.3，1.2)，(−3.0，0.0，1.2)
- 基于 A_5，B_5 点的：(−2.5，0.0，0.975)，(−2.5，0.3，0.975)，(−2.65，0.3，0.7875)，(−2.65，0.0，0.7875)
- 基于 A_6，B_6 点的：(−2.0，0.0，0.75)，(−2.0，0.3，0.75)，(−1.9，0.3，0.45)，(−1.9，0.0，0.45)

案例 10.8 中要求读者使用这些数据点写一个程序，从不同视角表示这个茶壶。

10.11.5　B 样条曲面片

在曲面设计中，为了得到更好的局部可控性，我们可以用 B 样条函数来代替 Bernstein 多项式。其等式为

$$P(u,v)=\sum_{i=0}^{M}\sum_{k=0}^{L}P_{i,k}N_{i,m}(u)N_{k,n}(v) \tag{10.81}$$

这里 $N_{i,m}(u)$ 和 $N_{k,n}(v)$ 是由等式(10.47)定义的 B 样条基函数(可以有不同的阶数)。通常我们为这两个 B 样条形式选择标准节点向量，这样就可以插值控制多面体的角点了。如果控制顶点反复出现或者有如同等式(10.55)那样的周期性，则曲面封闭(关于 u 或 v 封闭或两者皆封闭)。控制多面体由 $(L+1)(M+1)$ 个控制顶点组成，u 和 v 的取值范围从 0 到其相应节点向量中节点最大值。我们还是选择三次 B 样条(即 $m=n=4$)。由于控制顶点的数目不受限制(顶点数并不像贝塞尔曲线那样影响多项式的阶数)，它可以对极其复杂的曲面进行很好的造型。如前文所示，为了设计出需要的曲面，设计者必须设置相应的节点向量和控制多面体。

图 10.62 显示了 B 样条曲面的一个例子。它在某一个参数上具有周期性(见习题)，而且控制多面体和 u 线 v 线清晰可见。

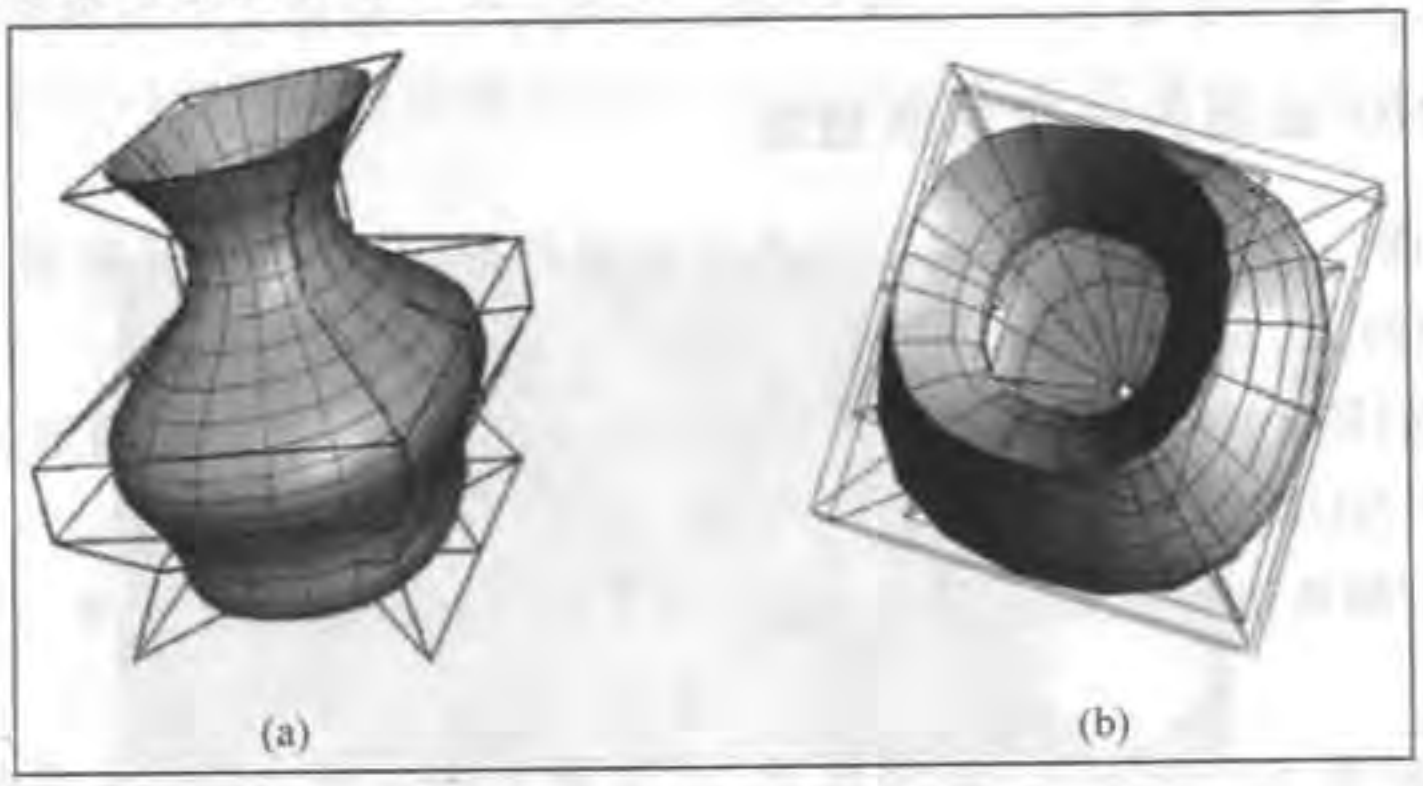


图 10.62　B 样条曲面的实例

练习

练习 10.11.2 周期 B 样条曲面

修改等式(10.81)使得曲面在参数 u 上具有周期性。你会发现等式(10.55)有助于此。那么对于这里的两类 B 样条,怎样的节点向量才有用?

10.11.6 NURBS 曲面

在 10.9 节中,我们简要地介绍了有理参数 B 样条曲线。如同等式(10.57)所示,这类曲线的形状同时取决于一组控制顶点和一组权因子 ω_i。设计者通过设置这些参数来得到所需曲线。

如同等式(10.81)那样处理 B 样条曲线,我们同样可以把 NURBS 曲线扩展成 NURBS 曲面,【Rogers00】使用如下形式:

$$P(u,v)=\frac{\sum_{i=0}^{M}\sum_{k=0}^{L}\omega_{i,k}P_{i,k}N_{i,m}(u)N_{k,n}(v)}{\sum_{i=0}^{M}\sum_{k=0}^{L}\omega_{i,k}N_{i,m}(u)N_{k,n}(v)} \tag{10.82}$$

这里 $P_{i,k}$ 是**控制多面体**的控制顶点,$\omega_{i,k}$ 是权因子,$N_{i,k}$ 是 B 样条函数。如同 B 样条曲线,如果所有的权因子相等,则这个曲面就可以化简为等式(10.81)中的 B 样条曲面(请验证)。

如同前面所讨论的,NURBS 曲面有两个主要优点:

1. 选择适当的控制顶点和权因子,$P(u,v)$ 可以精确表示二次曲面。这与非有理 B 样条曲面形成对比,后者只能逼近二次曲面。后面我们会举例构造圆锥曲面。

2. NURBS 曲面具有**投影不变性**。这个不变性即若要得到一个 NURBS 曲面的投影曲面,只需计算原曲面控制顶点的投影点,适当调整相应权因子,然后应用等式(10.82)即可。与此相比,非有理 B 样条曲面只有仿射不变性,而没有投影不变性。

由于 NURBS 曲面表现出来的一般性和灵活性,它在曲线曲面设计中有着广泛的应用。

基于 NURBS 曲面的一般曲面造型

NURBS 曲面一族在设计中提供了种类繁多的外形,其中包括了前面章节中讨论的曲面类型。实例如图 10.63 所示。

- 拉伸曲面(图 10.63(a))。棱柱外形由关于 u 的 NURBS 曲线确定,直线边界由关于 v 的一阶 NURBS 曲线决定。
- 直纹面。两条边界曲线是关于 u 的 NURBS 曲线,母线是关于 v 的一阶 NURBS 曲线。
- 旋转曲面(图 10.63(b))。轮廓线是 xz 平面上的一条 NURBS 曲线,鉴于 NURBS 曲线可以精确表示圆锥曲线,则用平行于 $z=0$ 的平面截得部分为圆弧。

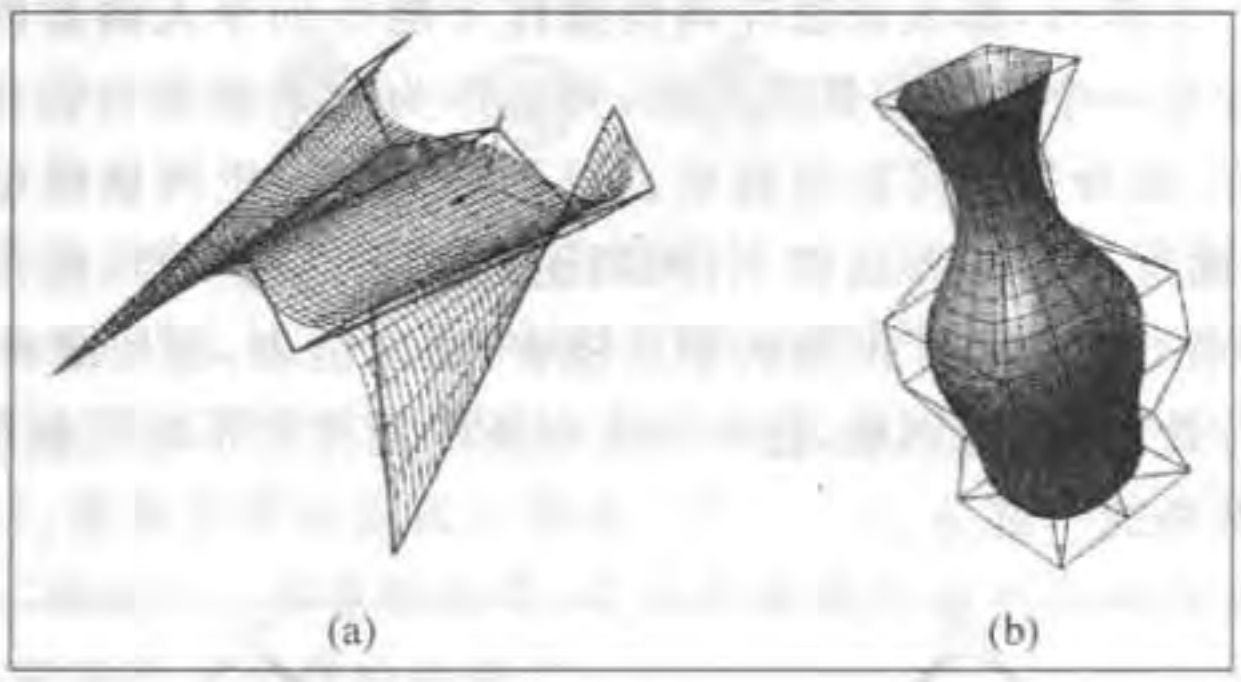


图 10.63　NURBS 曲面的实例

- 二次曲面。旋转曲面若以圆锥曲线为其轮廓线，则生成二次曲面。若对它进行仿射变换，可使其发生扭曲而不再成圆对称。

我们在此强调仅仅依靠 NURBS 设计工具所展现的灵活性，就足以对以上所有类型曲面进行造型。案例 10.10 展示了在设计和造型这些曲面时所遇到的细节。

许多用于设计和处理曲面的其他技术也已经有所发展。关于这些技术具体实例及细节的资源和链接可以在本书所属网站上找到。

练习

练习 10.11.3　一般曲面造型

讨论如何用 NURBS 曲面对以下曲面进行造型：(a)圆柱；(b)圆锥；(c)平面；(d)球面。更多细节参见案例 10.8。是否可以对任意的 Coons 曲面片进行造型？

10.11.7　细分曲面

对于一些网格，尤其是从第三方得到的以及能够在线使用的网格，它们只提供粗糙的几何模型，对一些应用而言缺少细节。比如一些地形模型如果面片数量不足，就不能遮盖个别的表面，因而也不容易区别光滑波动的山丘(从远处望去)和山脉上陡峭的山峰(从飞机上近距离观察)。或者你希望手工制成一个光滑的国际象棋棋子，比如象或者兵，如图 10.64 所示，但是所能使用的锉刀却只能锉成很少的面片(如图中最左边模型)，从而不能令人感到满意。

图 10.64　网格线的两层改进，以及明暗显示模式下的棋子

如果得到这样一个锉刀,那么该怎样调整操作才能得到令人满意的模型?细分网格面片从而得到新的面片是一个极有前景的方法,并且作为一个课题它仍在被研究中。如同我们接下来描述的那样,细分过程可以很简单。对于图 10.65 里网格模型中的一个特定的三角形面片,一个增加面片的简单方法如下:两两连接三条边的中点,使得一个三角形生成四个三角形(谨记第 6 章介绍的关于网格的顶点链表,法向链表,面片链表等数据结构,如何利用它们构造算法进行处理)。特别地,注意到生成新的面片并不需要新的数据——改进后的网格实质上是原始网格的一部分。

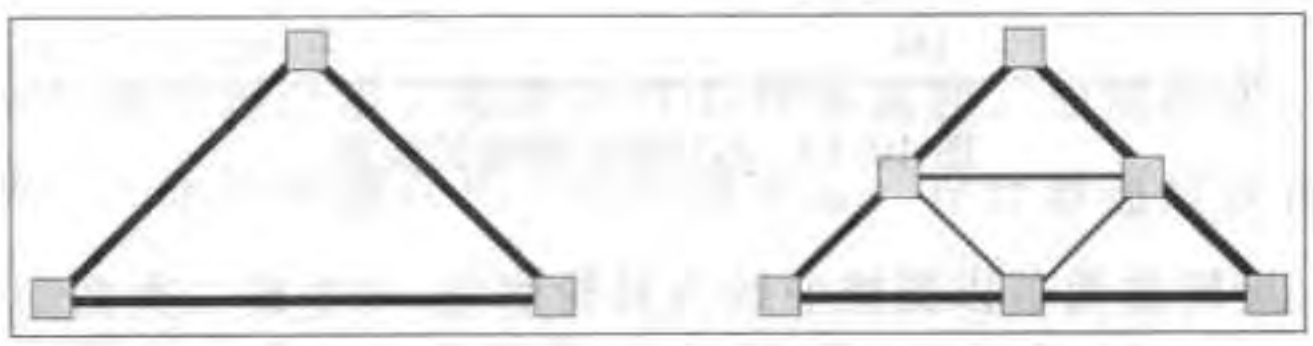


图 10.65 左侧:原始网格;右侧:经过一次细分后的网格

由一个面片生成四个面片,称为"单层改进"。显然把这个过程直接继续下去可得到更多层次的改进。图 10.66 显示了线框显示模式下一个女性躯干模型的三层改进,其中左侧模型的面片清晰可见。

图 10.66 在线框显示模式下的三层改进

图 10.67 中在明暗显示模式下同样的三层改进。此时从左至右各个模型在质量上的差别十分明显。

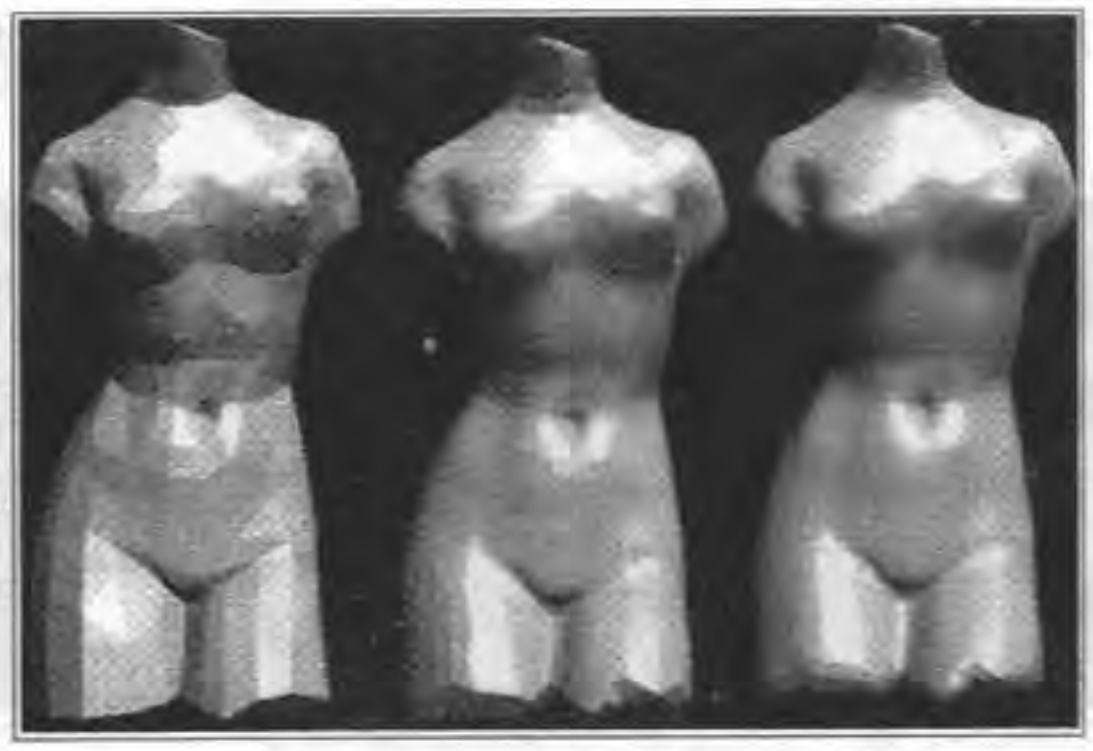


图 10.67 在明暗显示模式下的三层改进

本章小结

本章介绍了几类表示和处理复杂曲线曲面的技术。这类曲线主要有两个用处，一个是用于图形应用程序中的绘制，另一个是对物体的移动进行建模，比如动画中的照相机。大多数的曲线设计是基于在第 3 章首次出现的参数表达形式。在处理和绘制在特定点自交的或是垂直移动的曲线时，参数方程比隐式方程更自然。

我们已经了解了曲线的一些重要性质，尤其是曲线的速率和曲线上每点的法向量。另外，还给出了曲线的光顺性这个微妙的定义。

我们举了一系列曲线的例子，如圆锥曲线，多项式曲线，以及按比例混合的多项式曲线。

我们还讨论了如何基于一组控制顶点来构造平滑变化的曲线，这种方法在计算机辅助几何设计(CAGD)中很常见。设计者可以设置一组数据点，混合生成曲线，从而控制曲线的形状。必须强调插值点的曲线和逼近点的曲线是有区别的。无论哪一种情况，几个控制顶点和算法可以构造曲线上无限个点，每个点对应参数 t 的一个值。

由于贝塞尔曲线的简洁性，它们最先被介绍。这类曲线来自迭代的 de Casteljau 内插过程，这使得它们有许多直观的性质。这类曲线有几类我们所需要的性质，使得它的形状可被预测，因此设计者在设置控制顶点的位置时即可大致确定曲线的形状。

贝塞尔曲线虽然应用在许多设计环境中，但由于作为基础的 Bernstein 多项式是定义在整个区间上的，使得贝塞尔曲线失去了局部可控性。另一个缺点就是随着控制顶点个数的增加，多项式的阶数也随之增加。这会导致曲线不能随心所欲地变化形状，并且计算代价较大和稳定性较差。

因此我们介绍了一类内容更为丰富的基于样条的混合函数。所谓样条即是分段多项式的组合，同一类型的曲线在各点有相同的连续阶。B 样条作为一类特殊的基函数，可以生成任何样条，在造型中应用最为广泛。因此它们为设计者提供了强有力的局部可控性，同时还具有像贝塞尔曲线那样所期望的性质。当 B 样条多项式的阶数增加到与控制顶点个数相同时，B 样条就是 Bernstein 多项式。

NURBS 曲线有更多的方法来控制外形。尽管它更为复杂，但是却使 B 样条成为它的一个特殊形式，而且能够精确表示圆锥曲线。支持 NURBS 算法的 CAD 系统能够使设计者运用一个统一的工具来生成各种不同形状的曲线。

我们讨论了用于插值而不仅仅是逼近控制顶点的曲线算法。我们着重介绍了分段三次多项式曲线，而且研究了当曲线不仅插值而且在各个连接点处有预定速率时，各个系数需要满足的条件。这个速率可由用户指定，或者由为得到在连接处更大的光顺性的约束指定，或由相邻控制顶点的局部几何信息指定。设计者也可以利用额外的参数，比如张力和偏移量，来更加有力地控制曲线的最终外形。

B 样条也可用于插值，但是它通常只能插值首末端点而不是所有控制顶点。这里有一个技巧，即计算一组新的控制顶点，使得由此生成的 B 样条曲线恰好通过所有原始控制顶点。

我们扩展了对曲线设计技术以设计不同类型的曲面，包括直纹面，旋转曲面，二次曲面。我们也讨论了使用贝塞尔，B 样条，有理 B 样条函数来设计曲面。可以把贝塞尔曲面片的生

成想象成一条形状不断变化的贝塞尔曲线在空间中的扫掠。这条移动中的贝塞尔曲线上的每一点其轨迹本身也是一条贝塞尔曲线。若几个贝塞尔曲面片的控制多面体的边界满足特定条件,则这几个贝塞尔曲面片就可以拼接起来。我们也讨论了 B 样条曲面片,它给设计者提供了更大的灵活性。由于 B 样条曲面多项式的阶数不随着控制顶点的增加而增加,所以可以生成非常复杂的曲面。

我们最后也扩展到了 NURBS 曲面族,设计者可以通过改变一组权因子来改变曲面片的形状,从而得到更大的灵活性。NURBS 曲面具有仿射不变性和投影不变性。此外,可以特殊化 NURBS 曲面来生成其他类型的曲面,因此设计者可以运用一种 NURBS 曲面算法来构造多种不同类型的曲面。

曲面细分法可以改进简单的和有组织结构的网格模型。利用这种方法对网格进行优化,可以生成新的面片,从而使网格模型在渲染时显得更加光滑。

本章只涉及到最基础的曲面设计方法,它们也已经发展出许多新的技术,大型计算机辅助设计和 3D 模型软件包中常常包含这些技术。设计者可以从中选择,反复调整,直到得到所需结果。一些外形,比如飞机的机翼和帆船的船体,是根据科学原理,美学,直觉,经验等各方面的综合进行造型的。

案例分析

案例 10.1 几类有趣的参数曲线

(难度等级:Ⅱ)**A. 椭圆的一般形式** 椭圆的参数表达方程可用一个正弦函数和一个余弦函数表达。通过叠加几条不同速度的椭圆曲线可以得到一组有趣的曲线。用这种方法计算谐波之和,与第 3 章的傅立叶级数曲线相似,只是这里是在二维上进行。我们先从两项开始,然后再考虑一般情况。设这簇曲线表达式如下:

$$x(t) = X_1\cos(2\pi t) + X_2\cos(2\pi kt)$$
$$y(t) = Y_1\sin(2\pi t) + Y_2\sin(2\pi kt) \tag{10.83}$$

每个等式的第一项表示椭圆,后面添加的是第二个椭圆,其速度是第一个的 k 倍。当 t 从 0 变到 1,第一个椭圆绕了一圈,另一个绕了 k 圈。如果 k 是整数,那么生成的图形是完全闭合的。编程画出这样的周期图形,其输入变量为 X_1, X_2, Y_1, Y_2, k。把更多的项添加到 $x(t), y(t)$ 中,可得到更一般的结果。

B. 圆的渐开线 抓住缠在扫帚柄周围绕圈的细线上的一点。拉紧细线绕着扫帚柄转动,使之展开。P 点所经过的螺旋型轨迹就是一条圆的渐开线。圆(扫帚柄)和 P 的连线就是圆的切线。显然这条切线总是与螺旋线垂直。此外,螺旋线上的各圈是相互平行的,且间距相等(间距多少?)通过旋转这个图形(或选择细线上不同的点 P)可以得到一族这样的螺旋线。每条螺旋线与圆所有的切线正交。曲线的参数方程如下:

$$x(t) = \cos(2\pi t) + 2\pi t\sin(2\pi t) \quad y(t) = \sin(2\pi t) - 2\pi t\cos(2\pi t) \tag{10.84}$$

编程来画出圆的渐开线。

C. 其他正弦类型曲线　对前面练习中的 $x(t)$，$y(t)$ 进行轻微的调整也会生成完全不同的形状。例如，图 10.68 中的曲线（感谢 Mount Holyoke 学院的 Robert Weaver 教授）可由下面等式得到：

$$x(t) = \cos(t) + \sin(8t)$$
$$y(t) = 2\sin(2\pi t) + 7\sin(7t) \qquad (10.85)$$

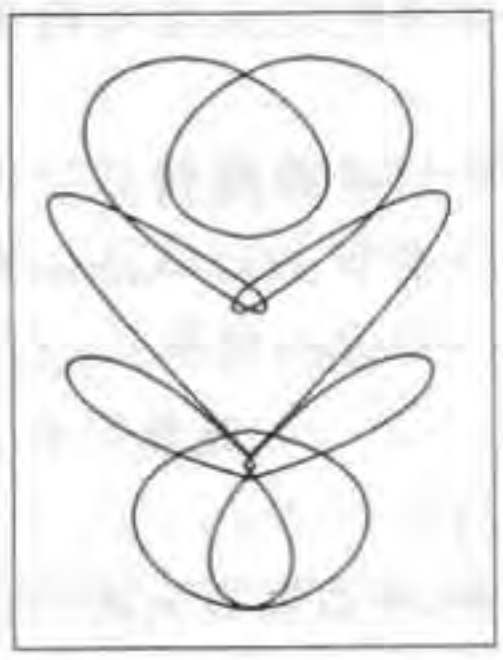


图 10.68　形状古怪的曲线

编程生成这条曲线，并且尝试一下其他变动。若要生成封闭的（周期的）曲线，这些三角函数必须满足怎样的条件？

D. Lissajous 曲线　如果允许两个正弦函数的频率不同，那么将会生成椭圆的一个变体：

$$x(t) = \cos(2\pi Mt + angle) \quad y(t) = \sin(2\pi Nt) \qquad (10.86)$$

这里 M 和 N 是新的频率，*angle* 是两个分量之间的"相位偏移"。这些生成的形状被称为"Lissajous 曲线"，在测试闭合电路时偶尔会出现在示波器上。编程显示 Lissajous 曲线，其输入参数为 M，N 和 *angle*。在实验中选择曲线上的点并大幅度增加 t 的值，观察其形状的变化。如果在曲线绘制完成后互相交换 x 和 y 变量的值，将可以观察到有趣的对称性。

案例 10.2　椭圆形台球桌

（难度等级：Ⅲ）在案例 4.4 中，我们研究了光线如何在多边形房间的墙壁上进行反射。可以考虑其他形状的房间又会怎样。被称为"Elliptipool"的椭圆形台球桌于 1964 年在美国开始销售【Gardner71，Steinhaus69】。如图 11.65 所示，通过跟踪光线在椭圆形房间内的反射情况，可以模拟出椭圆形台球桌。

给定初始光线 $S+\mathbf{c}t$，其起始点 S 在椭圆内。计算光线与椭圆的交点 P 以及反射光线方向 $\mathbf{r}$。从 S 到 P 绘制一条直线用于跟踪这条反射光线。然后把下一条光线设为 $P+\mathbf{r}t$，并重复这个过程。

不妨假设椭圆方程如下：

$$F(P) = \left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 - 1 = 0$$

且假设给定光线为 $S+\mathbf{c}t$。

光线与椭圆的交点：

如果光线与椭圆相交，那么一定存在一个 t 值使得点 $S+\mathbf{c}t$ 在曲线 $F(P)=0$ 上。即 t 满足条件 $F(S+\mathbf{c}t)=0$。所以对于椭圆我们有如下方程：

$$\left(\frac{S_x + c_x t}{a}\right)^2 + \left(\frac{S_y + c_y t}{b}\right)^2 - 1 = 0$$

这是一个关于 t 的二次方程。二次方程很容易求解，可能有零个，一个或两个解。

- 无解：光线不与椭圆相交。
- 一个解：光线与椭圆相切。
- 两个解：光线进入椭圆后又离开椭圆。

如果光线从房间内部出发,那么两个解一正一负(为什么?)。我们使用正值解,记为 t_{hit}。

我们还需找到反射光线在交点处的方向。在等式(10.3)中令 $x(t)=a\cos(t)$,$y(t)=b\sin(t)$即可得到其法向量$(-b\cos(t),-a\sin(t))$,若用 x,y 的形式可写作$(-bx/a,-ay/b)$。既然只有方向才是重要的,那么经过放缩可化简为$(-b^2x,-a^2y)$。由于光线是从椭圆的内壁反射回来,所以需要向内的法向量。通过观察图 10.69,可得当 x 和 y 都为正时,向内的法向量其 x 和 y 分量都是负的,则法向量有如下形式:

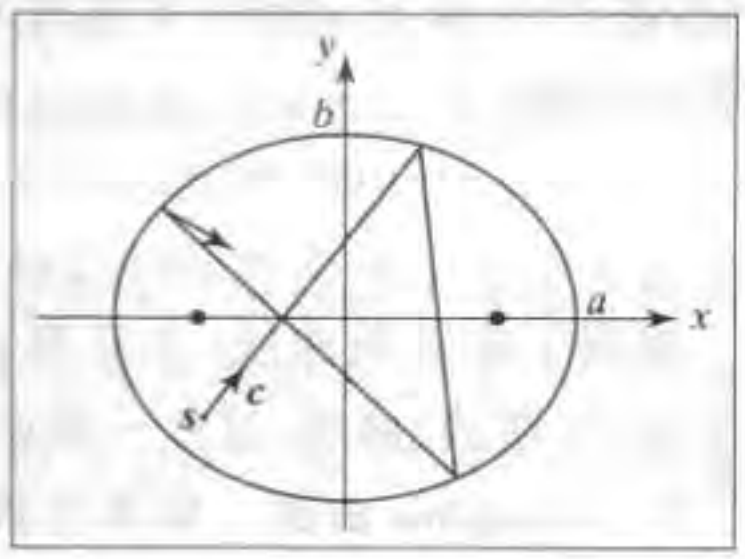


图 10.69 椭圆形台球桌的模拟

$$\mathbf{n}=(-b^2x,-a^2y) \qquad (10.87)$$

它确定了向内法向量的方向。把它代入等式(4.27),可以得到反射光线 r。

对椭圆形台球桌的模拟揭示了一些有趣的现象:光线的路径只有三种【Steinhaus69】。

- 如果光线通过任意一个焦点,则经过反射它会通过另一个焦点。这显然是由于椭圆的反射性质:从一个焦点发出的光线经过椭圆反射后会通过另一个焦点。这样这条光线会一直反复交替地通过两个焦点。几次反射后,它经过的路径就会和 x 轴难以区分了。
- 如果对于光线其初始路径不在两个焦点之间,那么它以后将再也不会经过两个焦点之间了。如图 10.70 所示,它的路径与一个较小椭圆相切,这个小椭圆和原始椭圆有相同的焦点。
- 如果光线的初始路径在两个焦点之间,那么它经过路径将不会终止,而且这条路径与焦点的距离不小于与拥有同椭圆一样焦距的双曲线的距离,如图 10.71 所示。

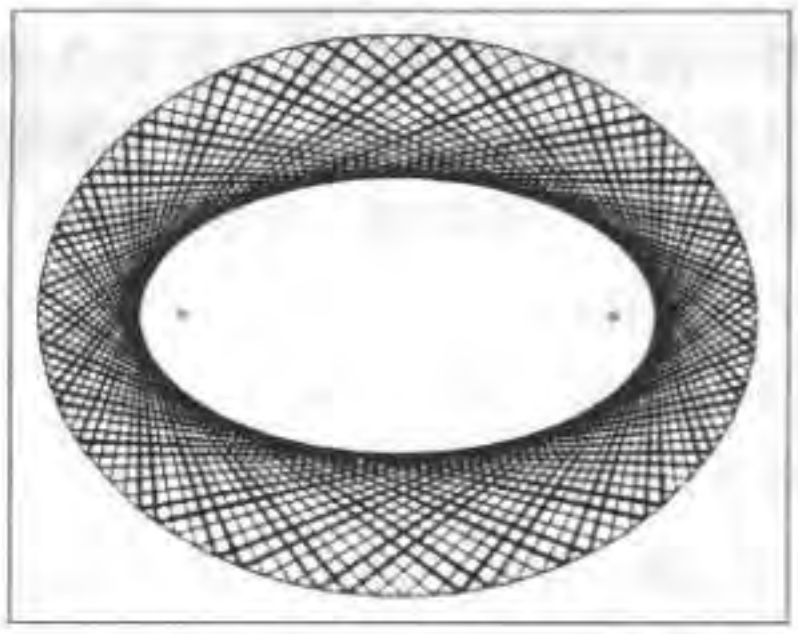


图 10.70 光线与第二个椭圆相切

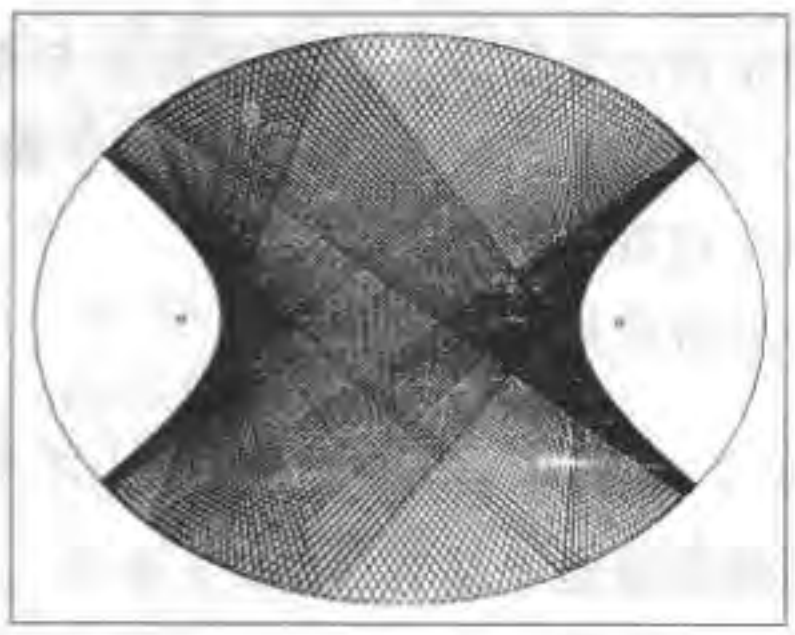


图 10.71 光线与一条双曲线相切

编写并执行一段程序来模拟椭圆形台球桌。用户在某些地方指定光线的初始位置和方向,然后对光线的多次反射进行跟踪。要允许用户指定光线通过一个焦点。(注意此时数学上的舍入误差会产生一些不可预料的影响,如何消除它?)偶尔改变路径的颜色,这样即使显示器被已有路径填满,当前路径仍可被区别显示。在实验中使用包括圆形在内的不同离心率的椭圆形台球桌。

(选做)通过在椭圆形台球桌内放置圆或者其他障碍物(椭圆?)来增强实验。这时是否会有可辨认的图案出现?

案例 10.3　贝塞尔曲线

(难度等级：Ⅱ)编写一个程序来绘制一条贝塞尔曲线，其一系列控制顶点由用户操作鼠标输入。计算曲线上参数值间隔密集的点，并用直线段依次连接。实验中可用不同数量的控制顶点。注意随着控制顶点个数的增加，多项式阶数也随之增加，计算就会变慢。

案例 10.4　二次样条的生成

(难度等级：Ⅱ)等式(10.41)表示了利用混合函数 $g(t-k)$ 的加权之和生成的曲线：

$$V(t) = \sum_{k=0}^{6} P_k g(t-k)$$

这里是由等式(10.38)定义的二次样条函数。编写并执行一段程序来绘制上式中的曲线 $V(t)$，其中的一系列控制顶点 P_k 由用户操作鼠标输入。此时可以编写一个辅助函数 double g(double t)来返回 g(t)的值。特别的，当 t 在区间[0，3]外时返回 0。

案例 10.5　建立样条曲线编辑器

(难度等级：Ⅲ)设计并执行一段程序，使得用户能够操作鼠标生成一个控制多边形 **P**。要求程序能够绘制由 **P** 定义的 B 样条曲线。这个程序可以执行以下命令，用户只需按下键盘上适当的键即可执行这些命令("b"代表 begin，"d"代表 delete，等等)：

- b)egin：(开始一个新的控制多边形 **P**)。
- d)elete：(删除 **P** 中离鼠标所指位置最近的点)。
- m)ove：(把鼠标所指的点拖到新的位置)。
- r)efresh：(绘制 **P**)。
- o)rder('1',…,'9')：(绘制阶数给定的基于 **P** 的样条曲线)。
- c)losed('−1',…,'−9')：(绘制基于 **P** 的封闭 B 样条曲线)。
- e)rase：(清屏)。
- q)uit：(退出程序)。

1. (选做)多个控制多边形　扩充程序的功能，使得它每次可以显示 10 个不同的控制多边形，并且能对每一个进行任意编辑。

2. (选做)能用 B 样条得到圆吗？　选择共圆的四点或八点，试验基于这些点的封闭 B 样条对圆的逼近可以到何种程度？设计一种合理的标准来评估曲线和圆之间的数值误差，并尝试不同的配置来确定最佳的曲线。

3. (选做)B 样条的变换　扩充程序的功能，使得用户可以对曲线进行仿射变换(可以通过预先设置的菜单进行)，并得到一个新的控制多边形。基于新控制多边形的 B 样条就是仿射变换后的 B 样条。

案例 10.6　用 B 样条插值控制顶点

(难度等级：Ⅲ)基于 B 样条混合函数和标准节点向量的曲线只能插值首末控制顶点。但是，通过对控制顶点的预处理，可以使 B 样条曲线插值所有控制顶点。在预处理中，根据原有的控制顶点产生新的控制顶点。根据新控制顶点生成的 B 样条曲线将通过所有给定

的原始控制顶点。

我们通过一个特定的例子来揭示其核心思想。用三次 B 样条函数来插值 6 个点 $y_0,\cdots,y_5$,这些点对应的 t 值间距相等,如图 10.72(a)所示。

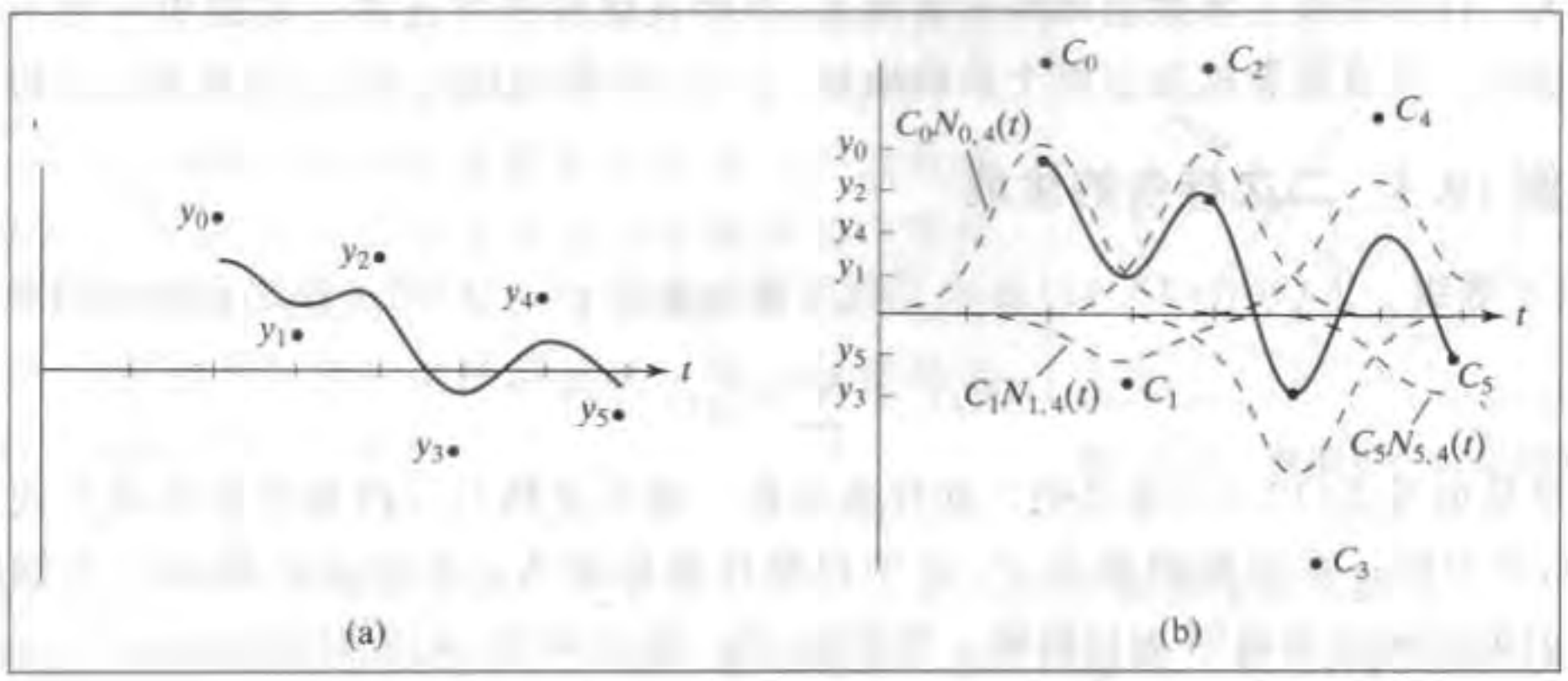


图 10.72 试着用三次 B 样条插值 6 个控制顶点

我们使用等距节点而不用标准节点向量,这样 $N_{0,4}(t)$ 在 $t=0$ 处开始,在 $t=2$ 处"凸起",$N_{1,4}(t)$ 在 $t=1$ 处开始,在 $t=3$ 处"凸起",依次类推。因此我们试着在 $t=2$ 时插值 y_0,$t=3$ 时插值 y_1,依次类推。

如图 10.72(b)所示,下面的等式

$$y(t)=\sum_{i=0}^{5}y_iN_{i,4}(t)$$

不经过所有控制顶点。这是由于 B 样条函数叠加后和式中各项互相影响,导致 $y(t)$ 值不能达到数据点的值。为了纠正这个错误,我们用一组新的值 $c_0,\cdots,c_5$ 来代替 y_i,这样基于如下等式的曲线确实能够插值 y_i,如图 10.72(b)所示。

$$p(t)=\sum_{i=0}^{5}c_iN_{i,4}(t) \tag{10.87}$$

通过求解一个线性方程组,就可以得到这组正确的 c_i 值。插值 6 个点满足条件 $p(2)=y_0,p(3)=y_1,p(4)=y_2,\cdots p(7)=y_5$。由于 t 在取整时,B 样条函数只能在 0,1/6,4/6上取值。所以这 6 个条件有如下形式:

$$\begin{aligned}4c_0+c_1&=6y_0\\c_0+4c_1+c_2&=6y_1\\c_1+4c_2+c_3&=6y_2\\\cdots&=\cdots\\c_4+4c_5&=6y_5\end{aligned} \tag{10.88}$$

这些等式几乎与等式(10.65)和等式(10.66)相同,所以可用 10.10.3 节中的方法求解。

选择一个合适的 L,上述方法可以立刻推广到任意个数的控制顶点 y_0, $y_1,\cdots,y_L$。编写一个函数 void adjust(double y[], double c[], int L),用给定的数组 y[]生成数组 c[]。

插值点 $p_i=(x_i,y_i)$时,对 x 分量和 y 分量各做一次预处理,得到两个数组 x_new[]和 y_new[]。这时插值曲线如下:

$$P(t) = \sum_{i=0}^{L} W_i N_{i,4}(t)$$

这里 W_i=(x_new[i], y_new[i])。实例见图 10.73。

编写一个程序，允许用户操作鼠标输入($L+1$)个控制顶点，由此绘制一条三次 B 样条插值曲线。尝试用不同的 L 进行实验。

(选做)扩展这个程序的功能，使得生成的插值曲线为闭合曲线。那么，为此等式(10.88)需要作何修改？

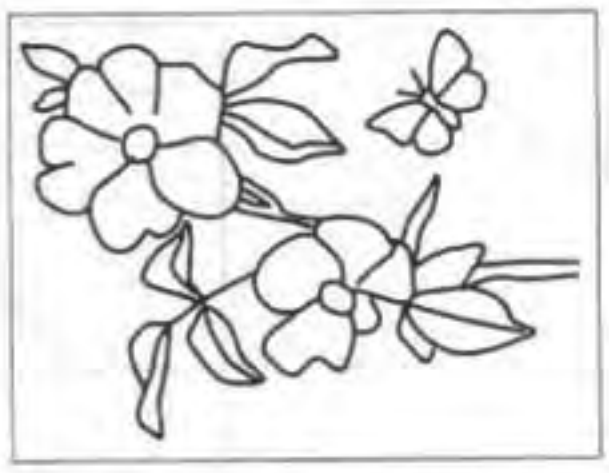


图 10.73　用 B 样条在 2D 上插值的实例

案例 10.7　用三次多项式插值

(难度等级：Ⅲ)编写一个程序，允许用户操作鼠标输入($L+1$)个控制顶点，由此绘制一条基于三次多项式的插值曲线。根据事先指定的张力值，偏移量，连续性，用 Kochanek-Bartels 方法设置内部连接点的速率。对于剩下的两个三次系数的条件，可以强制两个端点处二阶导数为 0。

允许用户使用键盘调整每个内部控制顶点的张力值，偏移量，连续性。比如，当用户单击控制顶点选定后，按“v”键张力将减少一个小的定量，或者按“V”键增加。同样的，按“b”或“B”键调整偏移量，按“c”或“C”键调整连续性。

案例 10.8　经典的茶壶

(难度等级：Ⅱ)用 OpenGL 编程，从不同视角绘制经典的茶壶。不要用 GLUT 版本中的茶壶：用曲面片造型一个属于自己的茶壶。

茶壶有 4 个主要部分组成。**壶体**是一张旋转曲面，其轮廓线由 xz 平面上 3 条贝塞尔曲线组成，详见 10.10.2 节。**把手**由 4 张贝塞尔曲面片组成，详见 10.10.3 节。**壶盖**也是一张旋转曲面，其轮廓线由 2 条贝塞尔曲线组成，如图 10.74，其数据点见图 10.75。

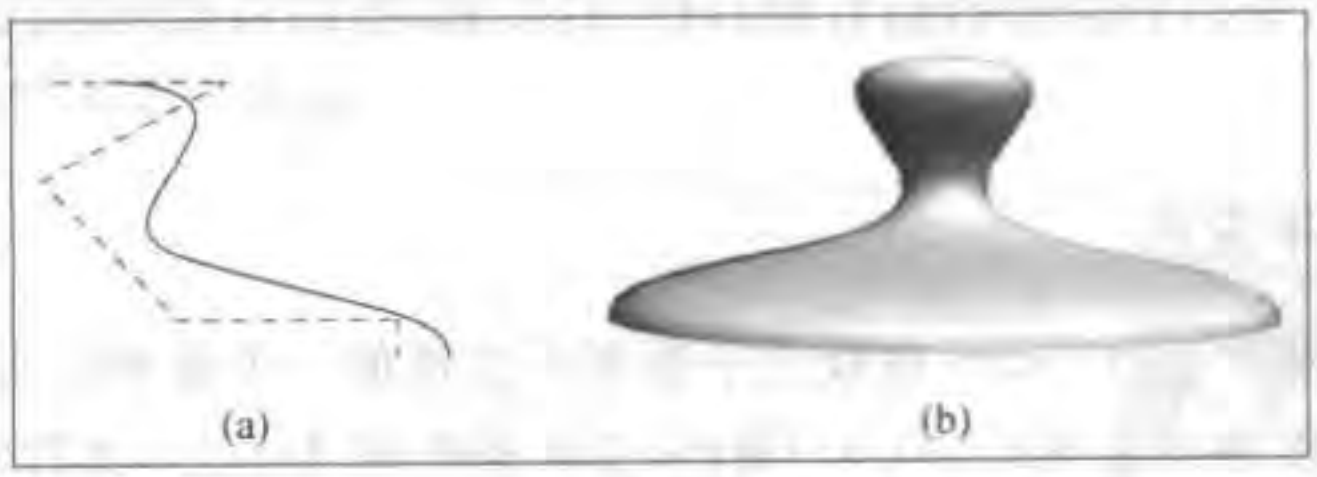


图 10.74　茶壶的壶盖

i	x	z
0	0	3
1	0.8	3
2	0	2.7
3	0.2	2.55
4	0.4	2.4
5	1.3	2.4
6	1.3	2.25

图 10.75　茶壶壶盖轮廓线的数据点

壶嘴与把手类似，也是由 4 张贝塞尔曲面片组成。图 10.76(a)显示了它的横截面，以及横截面处相应的控制多面体。

与把手类似，壶嘴曲面也关于 xz 平面对称。整个 y 正方向一侧的上曲面片有以下 16 个控制顶点：

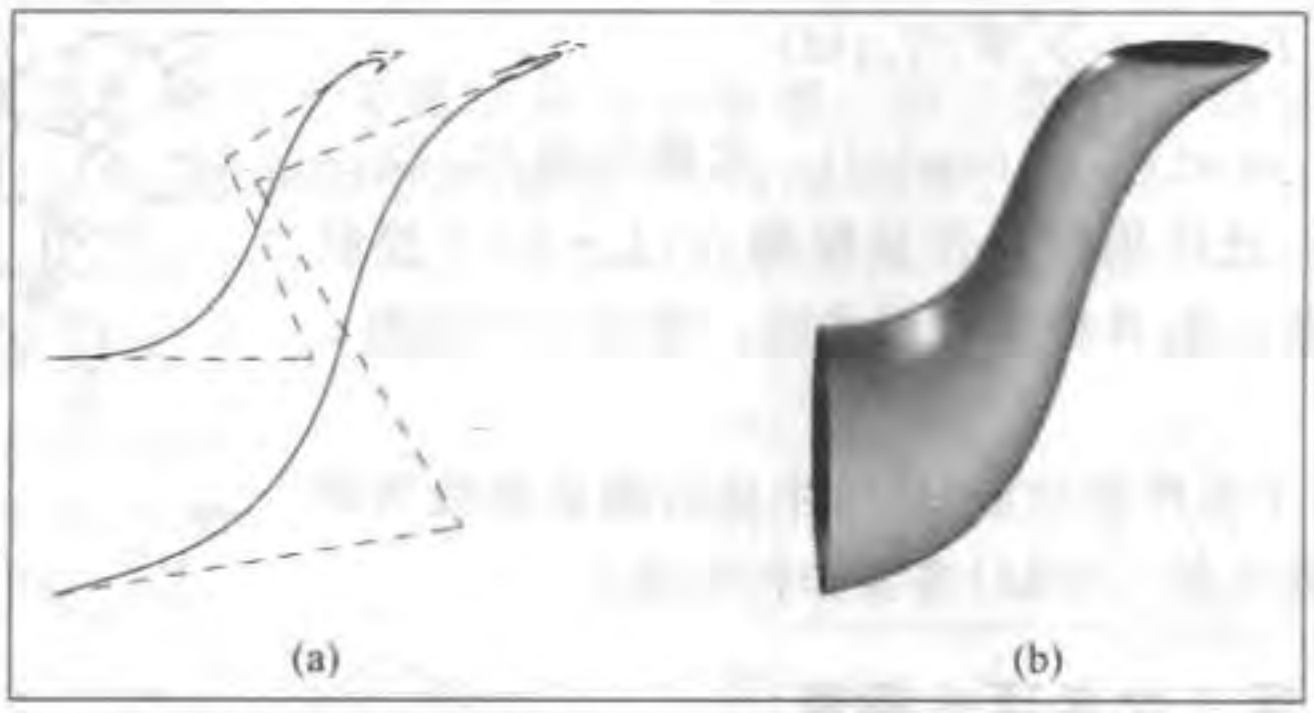


图 10.76 茶壶壶嘴的设计

- 基于 C_0, D_0: (1.7, 0.0, 0.45),(1.7, 0.66, 0.45),(1.7, 0.66, 1.275),(1.7, 0.0, 1.275)
- 基于 C_1, D_1: (3.1,0.0,0.675),(3.1,0.66,0.675),(2.6,0.66,1.275),(2.6,0.0,1.275)
- 基于 C_2, D_2: (2.4,0.0,1.875),(2.4,0.25,1.875),(2.3,0.25,1.95),(2.3,0.0,1.95)
- 基于 C_3, D_3: (3.3,0.0,2.25),(3.3,0.25,2.25),(2.7,0.25,2.25),(2.7,0.0,2.25)

在 y 轴正方向一侧的下曲面有 16 个控制顶点:

- 基于 C_3, D_3: (3.3,0.0,2.25),(3.3,0.25,2.25),(2.7,0.25,2.25),(2.7,0.0,2.25)
- 基于 C_4, D_4: (3.525,0.0,2.34375),(3.525,0.25,2.34375),(2.8,0.25,2.325),(2.8,0.0,2.325)
- 基于 C_5, D_5: (3.45,0.0,2.3625),(3.45,0.1,2.3625),(2.9,0.1,2.325),(2.9,0.0,2.325)
- 基于 C_6, D_6: (3.2,0.0,2.25),(3.2,0.15,2.25),(2.8,0.15,2.25),(2.8,0.0,2.25)

案例 10.9 NURBS: 投影不变性

(难度等级: Ⅰ)A. 这个案例提供了 NURBS 曲线关于投影不变性的一个逐步证明。其中的细节即具体步骤。设 NURBS 曲线由等式(10.57)给出,变换矩阵 M 为 4×4,其行向量分别是 $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4$。

a. 经过变换 T,等式(10.57)在齐次坐标下的曲线 $\widetilde{P}(t)$ 成为如下的 $M\widetilde{P}(t)$:

$$M\widetilde{P}(t) = \sum_{k=0}^{L} \omega_k (\widetilde{P}_k \cdot \mathbf{m}_1, \widetilde{P}_k \cdot \mathbf{m}_2, \widetilde{P}_k \cdot \mathbf{m}_3, \widetilde{P}_k \cdot \mathbf{m}_4)^T N_{k,m}(t)$$

这里 $\widetilde{P}_k = (P_x, P_y, P_z, 1)^T$ 是齐次坐标下的 P_k。

b. 在一般坐标下,上式即为:

$$T(p(t)) = \frac{\sum_{k=0}^{L} \omega_k (\widetilde{P}_k \cdot \mathbf{m}_1, \widetilde{P}_k \cdot \mathbf{m}_2, \widetilde{P}_k \cdot \mathbf{m}_3)^T N_{k,m}(t)}{\sum_{k=0}^{L} \omega_k (\widetilde{P}_k \cdot \mathbf{m}_4) N_{k,m}(t)} \tag{10.89}$$

c. 显然每一个控制顶点 P_k 变换为：

$$T(P_k) = \left(\frac{\widetilde{P}_k \cdot \mathbf{m}_1}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_2}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_3}{\widetilde{P}_k \cdot \mathbf{m}_4} \right)^T$$

d. 用以上控制顶点和一组权因子，比如 v_k，生成的 NURBS 曲线如下：

$$\frac{\sum_{k=0}^{L} v_k \left(\frac{\widetilde{P}_k \cdot \mathbf{m}_1}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_2}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_3}{\widetilde{P}_k \cdot \mathbf{m}_4} \right)^T N_{k,m}(t)}{\sum_{k=0}^{L} v_k N_{k,m}(t)}$$

e. 如果 v_k 取等式(10.60)中的 $\bar{\omega}_k$，那么上式就和等式(10.59)相同。

B. 为什么 B 样条没有投影不变性　说明前面的推导步骤中哪一步对 B 样条不成立。

C. 扩展到曲面　用与上面类似的步骤，证明 NURBS 曲面也具有投影不变性。

案例 10.10　NURBS 曲线的绘制

（难度等级：Ⅱ）

a. 编写并执行一个应用程序，用于绘制 NURBS 曲面片的 u 线和 v 线。它的核心内容是一个计算曲面 $P(u,v)$ 上点的函数，其原型可能是：

```
Point3 nurbsPoint(Point3 P[][],          //控制顶点的矩阵
        int L, int M,                    //控制顶点个数为(L+1)(M+1)
        float w[],                       //权因子向量
        float knot[],                    //节点向量
        int m, int n,                    //B 样条的阶数
        float u, v);                     //参数 u, v 的值
```

b. 用图 10.77 平面视角中的控制多面体进行实验，这里的 0、a 或 b 表示各点在 xy 平面上方的高度。图 10.77(b)显示了在选择特定权因子以及高度 a 和 b 后所生成的“穹顶状”NURBS 曲面。

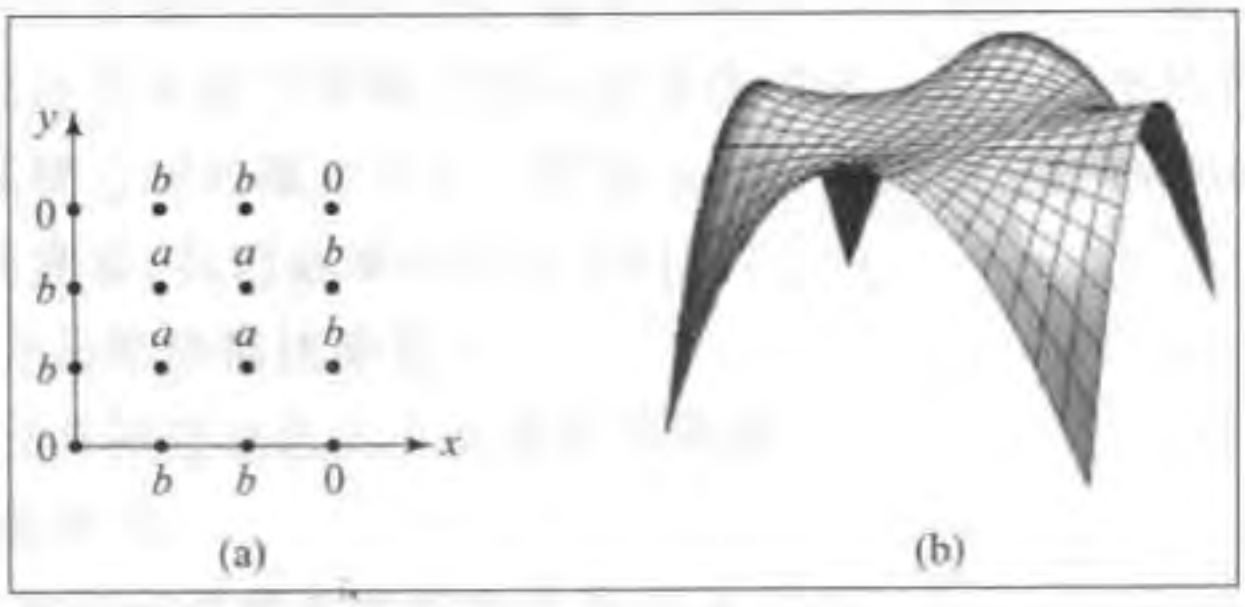


图 10.77　设计一个屋顶形状

在用户指定 a 和 b 后，按照上述方法构造和绘制曲面片。试用以下类型的权值：

i. 所有权因子相等。

ii. 高度为 0 的权因子为 W,其余点权因子为 sW,其中 s 值由用户输入。

iii. 其他感兴趣的权因子组合。

以上组合能否生成一个接近于半球的屋顶曲面?怎样调整才能使屋顶曲面更接近于半球?

c. 构造拉伸曲面和直纹面。尝试用 NURBS 曲面构造拉伸(线性扫描)曲面。以 NURBS 曲线 $C(u)$ 为轮廓线,沿着着垂直于轮廓线所在平面的方向扫掠,就可得到拉伸曲面。直线侧边是由 v 上的一阶 NURBS 曲线构成。再构造直纹面,其两条边界曲线都是 u 上的 NURBS 曲线,母线是 v 上的一阶 NURBS 曲线。

进一步阅读

Faux 和 Pratt【Faux79】在早年写了一本十分优秀的书,介绍了曲线曲面设计中的数学基础。Rogers 和 Adams【Rogers90】的书提供了大量在曲面设计中用到的方法,Farin【Farin90】也仔细地介绍了曲面中用到的数学。Bartels【Bartels87】讨论了曲线曲面设计中种类繁多的样条,以及它们各自不同的性质。由 Bloomenthal【Bloomenthal97】编辑的书中几个关于曲线曲面设计的章节很精彩,包括一章 Blinn 所写的关于 2 阶曲面和一章 Bajaj 所写的隐式曲面。Rogers【Rogers00】从历史的观点很好地介绍了 NURBS 曲线曲面。还可以看最近 N. Bunnel 在"GPU Gems2"一书中的"Adaptive Tesselation of Subdivision Surfaces with Displacement Mapping"以及 R. Pajarola 与 J. Rossignac【Pajarola00】在期刊"IEEE CG&A"中的 Compressed Progressive Meshes,这些都是关于细分曲面的。

第11章 颜色理论

本章学习目标

- 研究颜色的性质及其数值描述
- 分析颜色描述的一些标准
- 定义和使用各种颜色空间
- 学习几种减少图像颜色种数的方法
- 研究对颜色查找表进行编程的方法

预览

11.1 节介绍人类复杂的颜色视觉系统，及如何用可靠且可复制的数值方法描述颜色。11.2 节讨论颜色的匹配过程，及如何用三原色的线性组合来表示任何颜色，此外还将说明如何选择三原色最合适。11.3 节介绍 CIE 标准色度图的主要思想，并说明其在颜色计算中的应用，另外还将讨论色域的概念。11.4 节介绍几种不同的颜色空间，并给出几个工具用于在不同颜色空间之间转换颜色。11.5 节介绍几种颜色量化算法，颜色量化是指在不降低视觉质量的条件下减少一幅图像的颜色数。

11.1 概　述

前面章节中介绍的光照模型通过分别计算红、绿、蓝三原色分量从而得到最终颜色。这种方法是可行的，下面将证明事实上这和人眼的工作原理有关。此外，这也是和现今大多数普通显示设备混合不同数量的内置红、绿、蓝色从而得到任意颜色的过程一致。

但是颜色的形成过程比这个简单方法远远复杂的多。颜色的形成依赖于光线的物理辐射和人眼——大脑系统之间的微妙交互。更深入的研究请参见【Feynman63】。从开发计算机图形应用程序的角度看，我们必须回答以下问题：

- 如何用数值方法准确描述颜色？
- 这些方法和日常描述颜色的方法有什么关系？
- 如何比较颜色异同？
- CRT 显示器和打印机能显示的颜色范围是多少？
- 怎样加载颜色查找表才能生成所需颜色？
- 当某个设备能显示的颜色数较少，比如说只有 256 色时，应该如何处理颜色范围？

为了开发能正确显示颜色的程序,需要方便的工具来描述和控制颜色,故本章将着手处理这个棘手的问题。

和长波、红外线和 X 射线一样,光本身也是一种电磁现象。所谓光,就是波长[①]在很窄的带宽范围,即可见光谱,内的电磁波。图 11.1 展示了可见光谱(对人类而言)在整个电磁波谱内的位置,图中同时还显示了一些其他常见电磁波的光谱。电磁波振动频率 f 向右边增大,而波长 λ 向左边增大[②]。人眼可见光的波长大约在 400nm～700nm(纳米)之间。

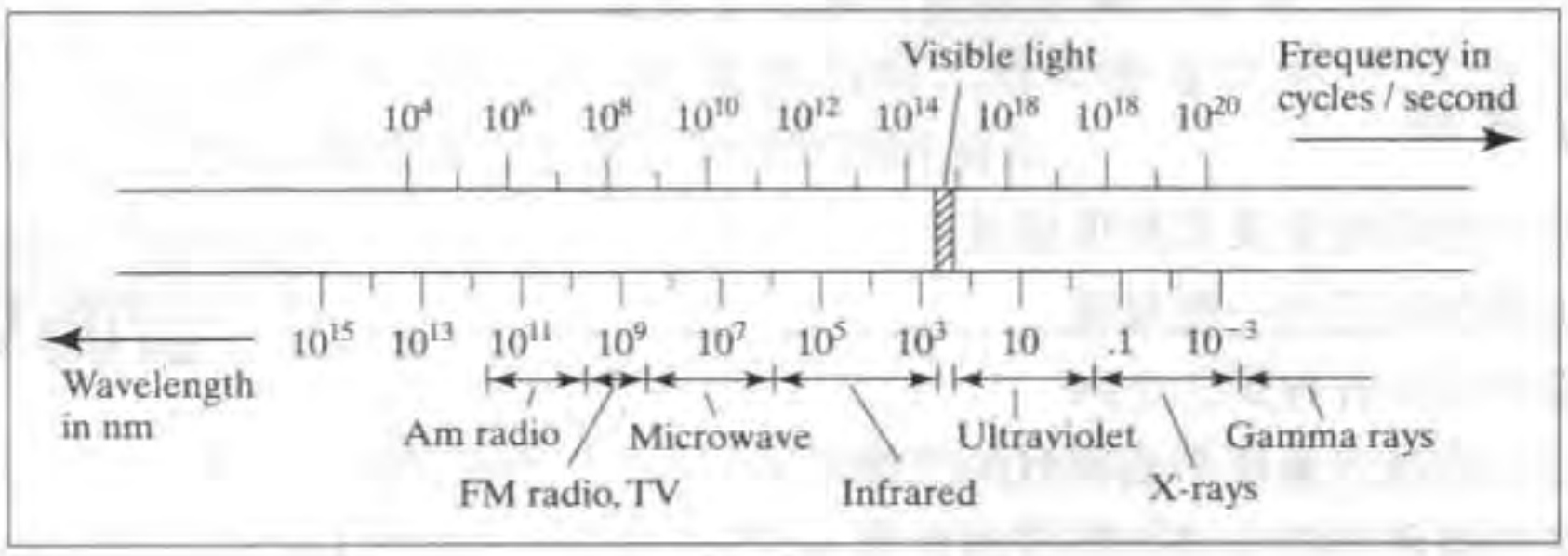


图 11.1 电磁波谱

11.1.1 眼睛:人类颜色感知的生理学基础

视网膜是人眼中感受光的一层薄膜。它位于眼球后部,包含两类神经感知细胞:视锥细胞和视杆细胞。

视锥细胞是对颜色敏感的细胞,每个视锥细胞感应某种特定颜色:红色、绿色或蓝色。根据三色刺激理论(将在后面分析说明),我们看到的颜色是视锥细胞对红、绿和蓝光的相对响应的结果。人眼大概能区别 200 种强度的红色、绿色或蓝色。每个眼睛有 600 万～700 万个视锥细胞,主要集中在视网膜上称为中央凹的一小块区域内。每个视锥细胞有它自己的神经细胞,从而允许眼睛分辨很小的细节。为了看清物体的细节,人眼将直视目标,从而使物体成像于中央凹。

相反,视杆细胞既不能辨别不同颜色,也不能看清细节。视网膜上围绕中央凹集中了 7500 万～15000 万个视杆细胞。此外,许多视杆细胞只连接到单个神经细胞,因而妨碍它们辨别微小细节【Gonzalez 和 Wintz87】。那么视杆细胞有什么作用?它们对低强度光线特别敏感,可以看到暗淡光线下视锥细胞看不到的物体。因此,晚上最好在稍远处注视物体,从而使物体成像于视网膜的中央凹之外。虽然丢失了细节和颜色,但至少看到了物体的大致轮廓。事实上,人眼在暗淡光线下对周围景象的视觉敏感度对人类的进化是很重要的。

有些光源,例如激光,发射单一波长的光,也称"纯光谱"光。人眼感知 400nm 光为紫色,620nm 光为红色,其他纯光谱色的波长在这两个极值之间。图 11.2 显示了一些纯光谱

① 波长是电磁波在一个振动周期内传播的距离。

② 波长 λ 和频率 f 成反比:$\lambda = v/f$,其中 v 为光在相应介质中的传播速度。在空气(或真空)中 v=300000 千米/秒;在玻璃中的速度为空气中的 65%。

色的光谱密度 $S(\lambda)$(单位波长功率)及其视觉颜色的常用名称。

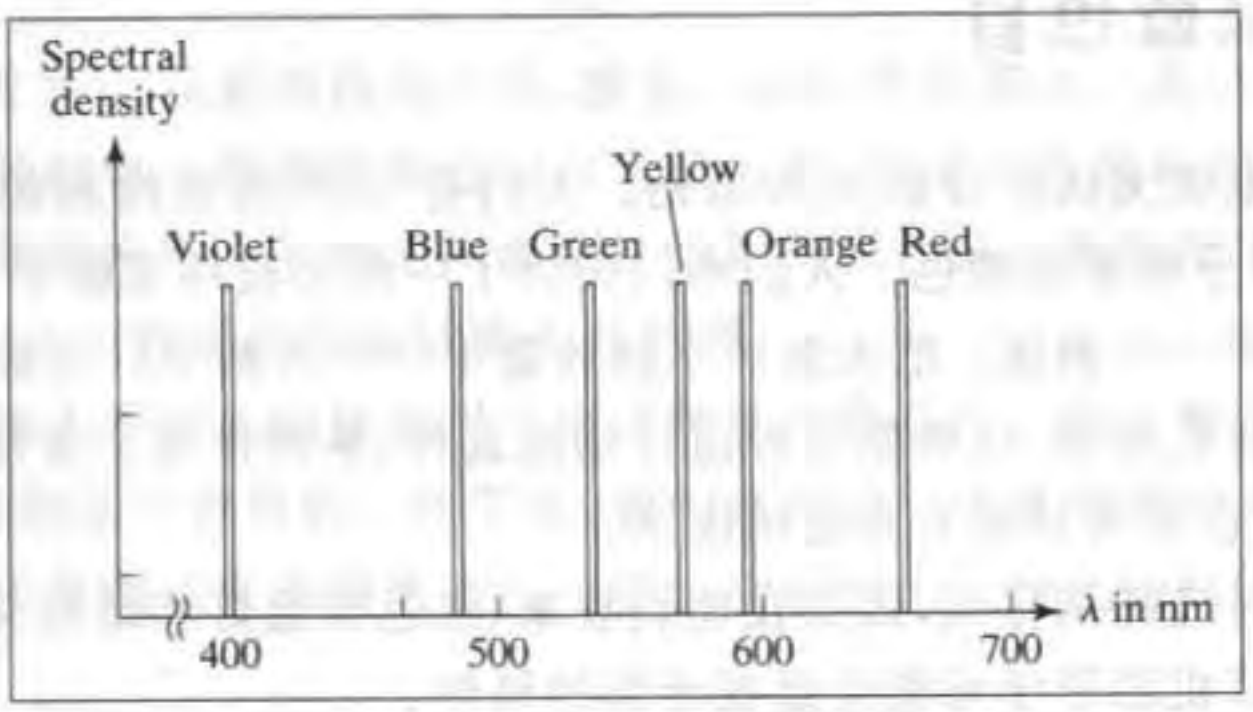


图 11.2 一些纯光谱色的光谱

大多数光源的光不是单一波长的,它们包含某一连续范围内的不同波长,相应每种波长的总能量也互不相同。它们的光谱密度集合(或称"频谱")覆盖一个波长带。光在任何波段范围内的总能量为此波段内光谱密度曲线下(和横坐标轴围成)的面积。图 11.3 展示了几种常见颜色的光谱密度曲线。

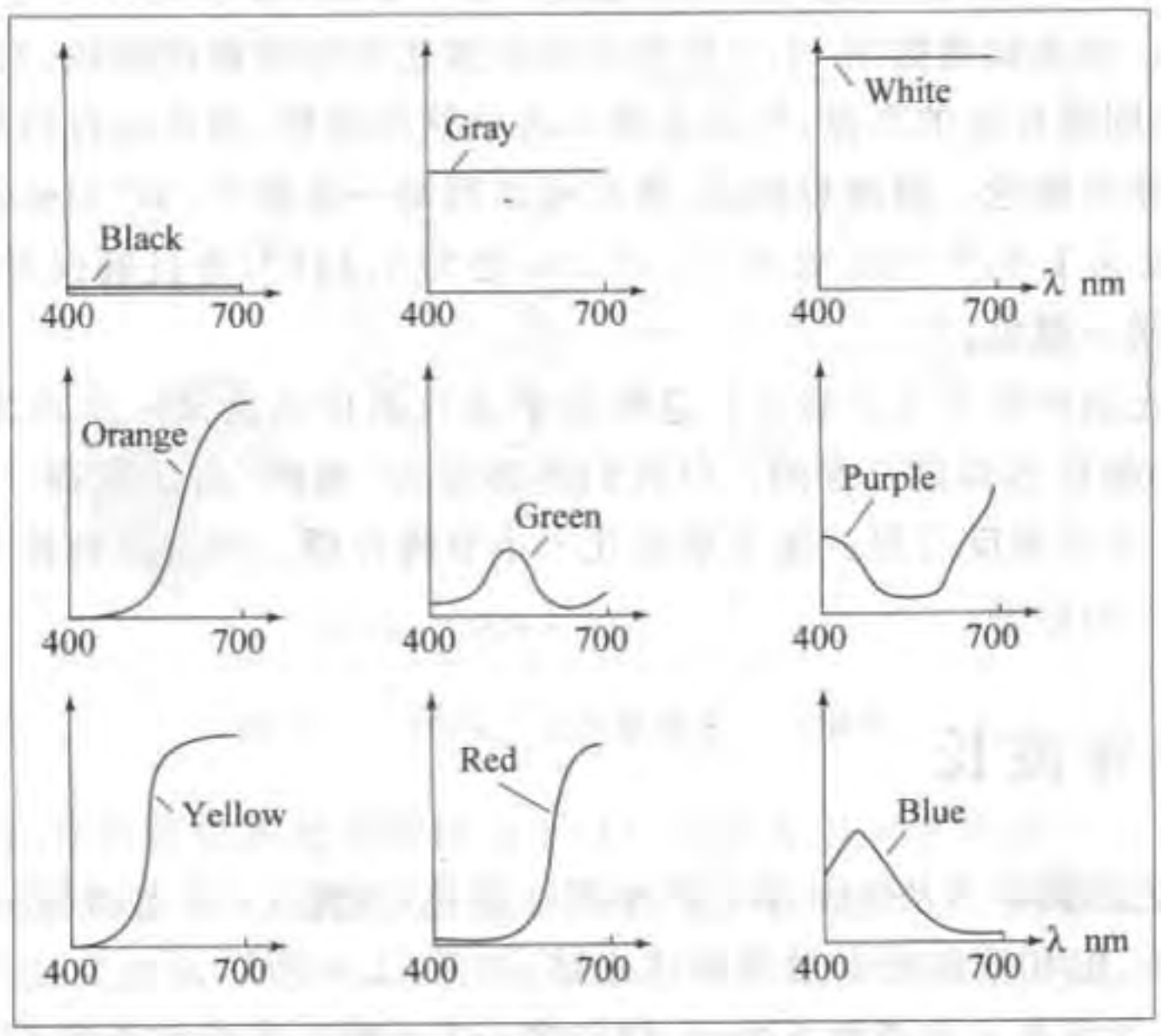


图 11.3 一些典型光线的光谱及其视觉颜色

每条曲线标注有人眼看到该种光时感知到的颜色。注意到白光在所有频率处包含近似等量的能量,而红光的能量集中于长波长处。灰色光也有较"平坦"的光谱密度,但是其强度较低。这些例子都突出了用数值方法描述颜色的困难之一:许多不同的光谱密度函数在人眼看来有相同颜色。例如,给定任何颜色样本,可以和很多不同的光谱密度形状相匹配,也就是说把它们并排放在一起时,样本颜色和其中任一种光谱密度对应的颜色是人眼无法分辨的。

11.1.2 红绿蓝色盲

有些人的视觉系统无法区分红光和绿光。大约有 10%的男性和相对少得多的女性是“色盲”,即他们无法分辨某些颜色。人们专门设计了一些实验用来确定被测试者是否色盲。本书主页展示有这样一个测试:把大量的小圆放置于一个大圆中,(你能看见数字吗?)小圆分别用稍微不同的颜色绘制,这些颜色经过仔细地选择,从而使有正常视觉的人可以看到大圆中的数字图案,而色盲者只能看到随机图案。

色盲者过马路时应特别仔细,因为在他们看来,红色交通灯和绿色交通灯是一样的!同样,在他们看来,计算机图形学可能会是很奇怪的科学。

11.2 颜色描述

假设要在电话里精确地描述一种颜色,所有描述必须都是口头的。对方可能是染料生产商或出版公司的生产经理。仅仅说“亮紫蓝色”是不够的,必须确保对方精确领会我们对颜色特征的描述。如果知道如图 11.3 所示的目标颜色的频谱密度曲线,就可以试着描述曲线在近似 12 个不同波长处的高度,但是这种方法显然很糟糕,而且太过精确,因为许多不同频谱曲线生成同样的颜色。最理想的话,我们可以列举一些数字,如“目标颜色在 400nm 处为 3.24,500nm 处为 1.6,600nm 处为 85,680nm 处为 1.117”,并且确保对方根据此描述能精确地重新生成同一颜色。

那么需要从光谱中取出多少数字?这些数字又代表什么含义?值得注意的是,答案是 3 个数字:人眼的颜色感知是三维的。但我们还需要对“编码”方法取得一致意见:如何将颜色映射为数字,及其相反过程。接下来的几个小节将介绍一些常用转换方法,然后讨论当前国际标准所包含的思想。

11.2.1 主导波长

一种描述颜色的简单方法是列举能产生同一颜色(视觉上)的光谱范围。这种方法用 3 个数字:主导波长、饱和度和光通量来描述光谱,如图 11.4 所示。

在此例图中,光谱在主导波长 620nm 处包含一个峰值。主导波长的位置规定了颜色的色调,此例中为红色。此外还有一定量白光,表现为矩形“底座”,降低红色的饱和度,使之呈现粉红色。

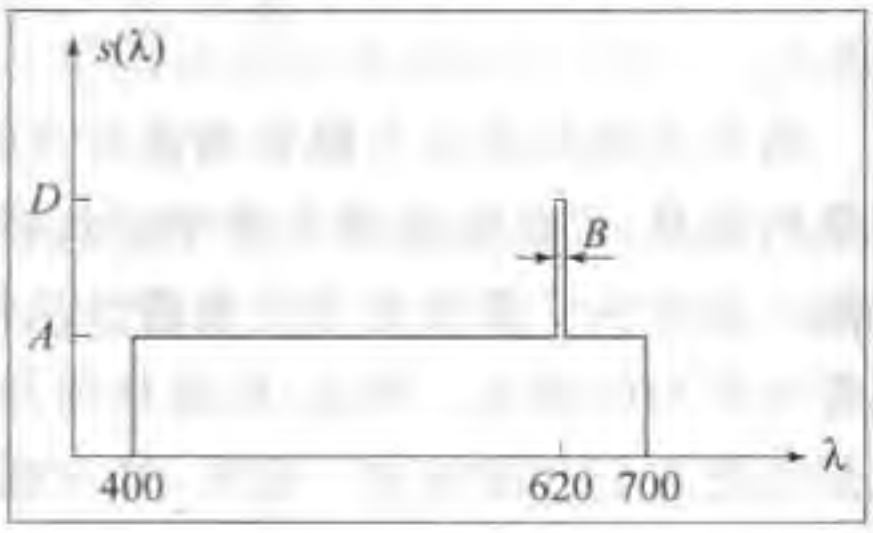


图 11.4 一种有主导波长的颜色的光谱

光的总能量,即光通量,由整个光谱曲线下的面积 L 给出。由于这里所说的各种光谱都是长方形状的,所以这个计算很简单:$L=(700-400)A+DB$。光的饱和度(或称纯度)定义为光的主导成分的光通量占总光通量的百分比【Billmeyer81】:

$$纯度 = \frac{(D-A)B}{L} \times 100\% \tag{11.1}$$

若 $D=A$，则纯度为 0，视觉颜色为白色，没有一点红色的痕迹。若 $A=0$，则一点白色都没有，人眼看到的是纯红色。柔和的颜色包含大量白色，称之为不饱和的。当两种颜色只有色调不同时，人眼大概能辨别 128 种不同颜色。当两种颜色只有饱和度不同时，人眼大概能辨别 20 种不同颜色，这个数字具体还依赖于其色调。

饱和度，光通量和主导波长的概念对于描述颜色非常有用。但是对于具体的一种颜色，如何准确度量这些值却是不容易的。接下来，我们将介绍一些更有效的方法用于描述颜色。首先，我们需要方法用来测试两种颜色是否一样。这就引出了颜色的匹配问题，这是描述颜色的基础。

11.2.2　颜色感知和匹配

常用描述颜色的方法是将其与一系列标准颜色样本比较，从中找出最匹配的颜色。人们设计了许多这类标准颜色集合，并广泛用于染料和印刷业【Munsell41】。如图 11.5 所示，也可以通过适当地组合一些测试光，然后和目标颜色做比较。

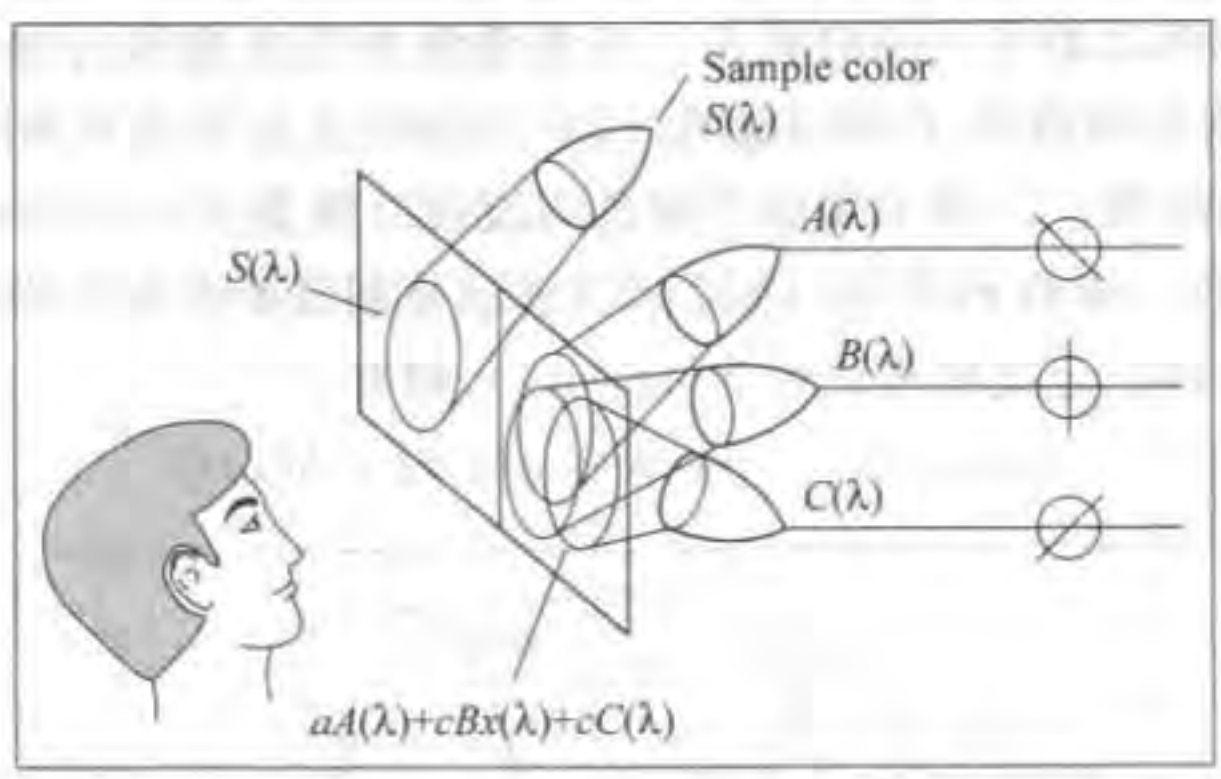


图 11.5　用测试色的叠加来匹配颜色

如上图所示，记样本颜色的光谱密度为 $S(\lambda)$，它被投射至屏幕的一个区域。三种光谱密度分别为 $A(\lambda)$、$B(\lambda)$和 $C(\lambda)$的测试光在屏幕的另一区域叠加。观察者调节测试光的强度(a、b 和 c)，直到测试颜色 $T(\lambda)=aA(\lambda)+bB(\lambda)+cC(\lambda)$和样本颜色无法区别，即使两个光谱 $S(\lambda)$和 $T(\lambda)$差别很大也没关系。这时就可以说样本颜色由 a、b 和 c 单位的 3 个测试颜色组成。在什么意义下，这么说是有道理的？

人脑有着非同寻常的颜色叠加代数机制【Feynman63】。若 $S(\lambda)$和 $P(\lambda)$两个光谱形状，在人眼感知下是同一颜色，我们记这一事实为 $S=P$。现在给这两个颜色同时添加第三种颜色 N，即把光谱 $N(\lambda)$同时叠加至前两者的光谱。实验证明，新生成的两种颜色仍然是人眼无法区别的！

除了符号＝表示两种颜色不可分辨之外，我们还定义符号＋，用 $S+N$ 表示光谱 $S(\lambda)$和 $N(\lambda)$相加产生的颜色。因此前述实验结果可以写为：

$$若\ S=P,\quad 则\ N+S=N+P \tag{11.2}$$

同样可以定义颜色的缩放,即对其光谱密度或总体亮度进行缩放。若 S=P,则对任何(正)标量 a 有 $aS=aP$。从而对两种颜色 A 和 B 进行线性组合是有意义的,例如 $T=aA+bB$,其中 a 和 b 为标量(回忆第四章中对向量的类似操作)。因此可以设计实验验证颜色的向量代数。在这个实验中,我们把颜色当作向量,对其进行加减、缩放、分解成不同分量等各种操作。

如前所述,关于人眼颜色感知系统的另一个重要事实是它是三维[①]的:任何颜色 C 都可以由三种原色,如红、绿和蓝色按下式叠加构造:

$$C = rR + gG + bB \tag{11.3}$$

其中 r、g 和 b 三个数分别描述每种原色在 C 中所占数量。符号 R、G 和 B 分别表示红、绿和蓝色,它们经常被作为三原色讨论。选用红、绿和蓝色作为三原色的原因是:人眼的视锥细胞对这三种颜色特别敏感。当然,等式(11.3)对其他三原色也适用,只要任一原色不是另外两种原色的线性组合即可。

给定一组三原色:R、G 和 B,任何其他颜色 $C=rR+gG+bB$,可以用三维空间中的一个点 (r,g,b) 表示。例如,若 R、G 和 B 分别对应红、绿和蓝色,则(0,1,0)表示单位亮度的纯绿色,(0.2,0.3,0.5)表示黄色。如果把每个分量增加一倍,则得到的颜色色调不变而亮度加倍。

有人做过一些实验测试人眼如何匹配颜色。其中特别有趣的是,对 R、G 和 B 进行线性组合得到(看起来)和纯光谱色相同的颜色。纯光谱色是完全饱和的单色,其能量集中于单一波长 λ。(用主导波长的话说,此纯光谱色 100%饱和,且主导波长为 λ)。图 11.6 显示了对大量测试者的试验结果。其中采用的三原色为波长分别为 700nm、546nm 和 436nm 的纯单色红色、绿色和蓝色。函数 $r(\lambda)$,$g(\lambda)$ 和 $b(\lambda)$ 表示分别需要多少这样的红、绿和蓝色来匹配波长为 λ 的纯光谱色。记此纯光谱色为 $mono(\lambda)$,则有:

$$mono(\lambda) = r(\lambda)R + g(\lambda)G + b(\lambda)B \tag{11.4}$$

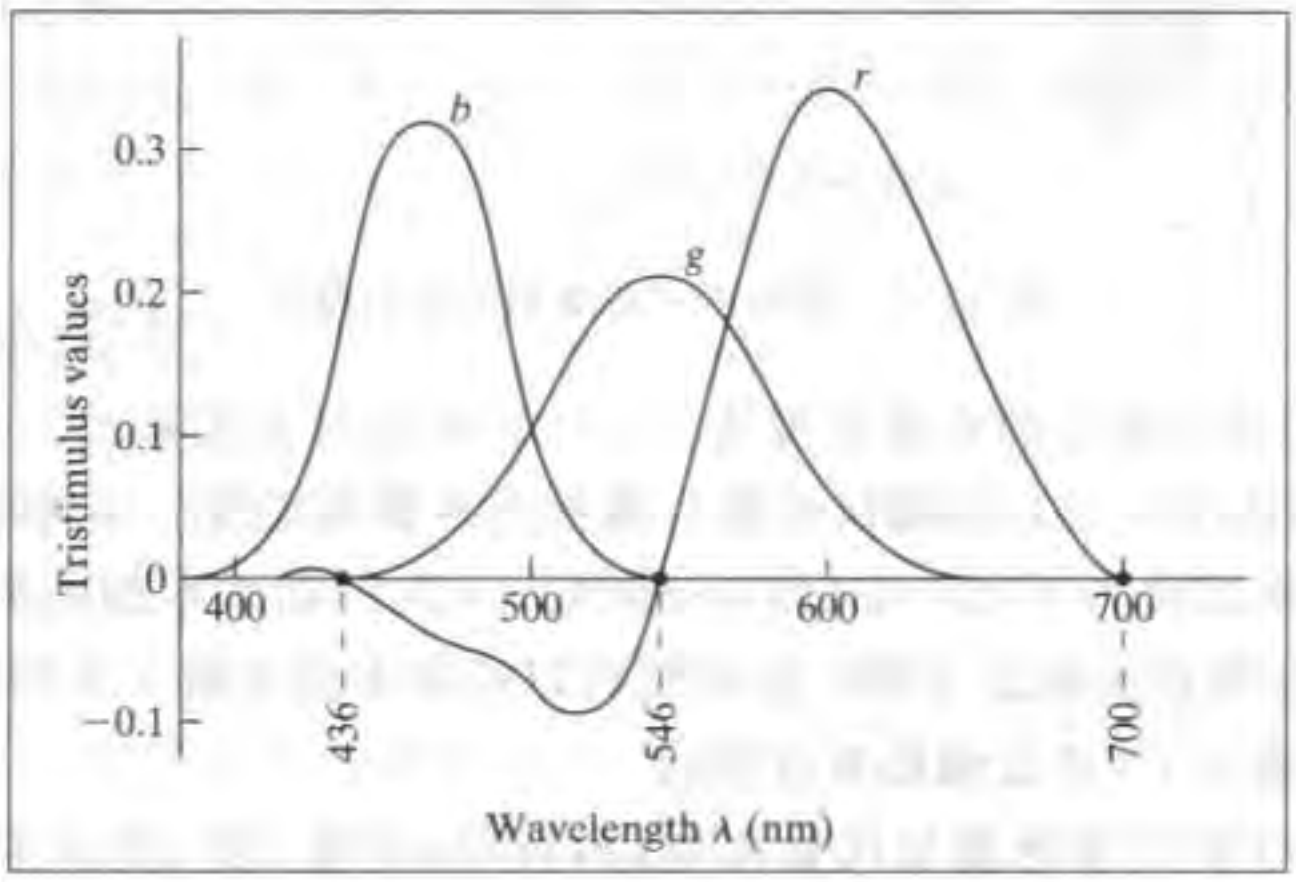


图 11.6 RGB 三原色的颜色匹配函数

例如,纯橙色 $mono(600)$ 在一般人看来和 $0.37R+0.08G$ 合成的颜色是一样的。显然,两者的光谱是不同的,但是两者看起来却完全相同。

① 换种说法就是:任何四种颜色总是线性相关的,即其中任一种颜色总是可以表示为另外三种的线性组合。

特别要提一点：对于前面使用的原色 R、G 和 B，数值 r、g 和 b 中某些可能必须是负值，才能使等式(11.4)成立！例如 $r(\lambda)$ 在 $\lambda=520$ 处是负值。由于我们不能去除尚不存在的颜色，那么颜色线性组合等式中的负号，例如 $C=0.7R+0.5G-0.2B$，代表什么物理意义？幸运的是，如果我们将上述等式重写为 $C+0.2B=0.7R+0.5G$，则前述矛盾消失了。这是说 C 本身不能由三原色的正叠加来构造，但是颜色 $C+0.2B$ 可以和 R 与 G 的适当的凸线性组合匹配。当我们取任何可见色 R、G 和 B 作为三原色时，这种现象就会发生。大部分颜色可以由三原色构造(使用正系数 r、g 和 b)，而其他颜色则不行，必须将其中一种原色放在等式的左边。简单地说，原因是两种颜色相加的和是不饱和色，故不可能由两种颜色的叠加来构造高度饱和的颜色。这对任何纯光谱色特别明显，因为它们本身是完全饱和的。

将颜色匹配等式的系数进行归一化(统一缩放使其和为 1)是很有用的，故定义：

$$\bar{r}(\lambda)=\frac{r(\lambda)}{r(\lambda)+g(\lambda)+b(\lambda)},\quad \bar{g}(\lambda)=\frac{g(\lambda)}{r(\lambda)+g(\lambda)+b(\lambda)},$$

$$\bar{b}(\lambda)=\frac{b(\lambda)}{r(\lambda)+g(\lambda)+b(\lambda)}$$

则显然有 $\bar{r}(\lambda)+\bar{g}(\lambda)+\bar{b}(\lambda)=1$。这些相对权重被称为 $mono(\lambda)$ 的色度值。它们给出了匹配单位亮度 λ 波长的颜色所需要的三原色的量。固定亮度不变，则可以只用两个数值，例如 $(\bar{r}(\lambda),\bar{g}(\lambda))$，来描述颜色，因为总是可以根据 $\bar{b}(\lambda)=1-\bar{r}(\lambda)-\bar{g}(\lambda)$ 确定 $\bar{b}(\lambda)$。

令 λ 在可见光谱内变化，我们可以绘制出三维点 $(\bar{r}(\lambda),\bar{g}(\lambda),\bar{b}(\lambda))$ 的轨迹，如图 11.7 所示。由于经过了归一化，曲线上所有点位于 $r+g+b=1$ 平面内。注意到，当 λ 取某些值时，曲线上的点的一些坐标为负值，即此曲线不完全位于第一卦限内。接下来讨论的 CIE 标准，对此曲线进行一些变换，从而使它完全位于第一卦限内，即三个坐标值处处为正。

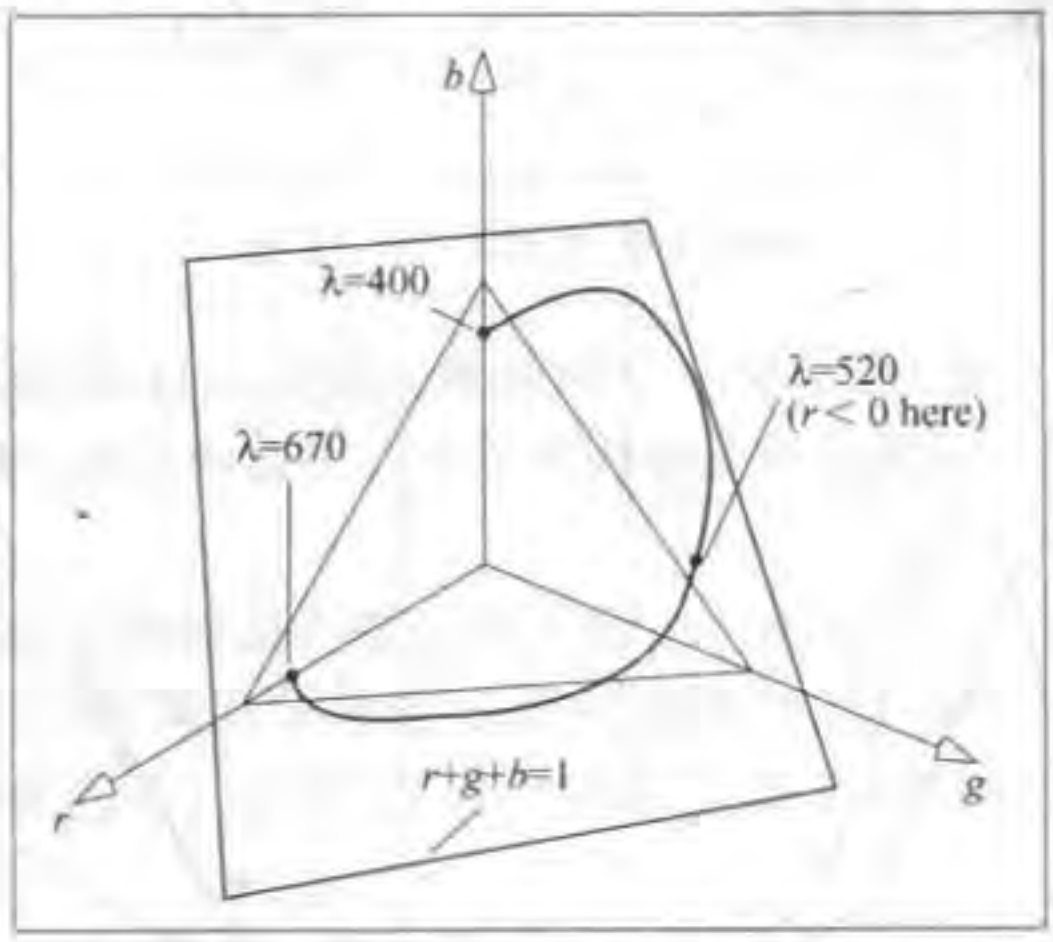


图 11.7　RGB 三原色的纯光谱色曲线

11.3　CIE 标准

怎样才能精确地描述颜色，并且使任何人都同意？由于颜色感知是三维的，只要对三原色达成一致意见，就可以用适当的三元数如 (r,g,b) 来描述颜色。那么该采用哪三种原色？

遗憾的是,使用任何可见色作为原色,都会导致描述有些可见色需要负系数。

为了解决这个难题,国际照明委员会(CIE)于 1931 年制定了 CIE 标准。CIE 标准定义了三种特殊的"过饱和"原色 X、Y 和 Z。CIE 标准经常被称为 **XYZ 模型**。其中的三原色不对应任何真实存在的可见色,但它们却有个性质:所有可见色可以表示为它们的凸线性组合。它们由类似图 11.6 所示的颜色匹配函数定义。

图 11.8 展示了 CIE 标准采用的颜色匹配函数。其概念和等式(11.4)是一样的:波长为 λ 的单色光和三原色的某个特定的线性组合相匹配:

$$mono(\lambda) = x(\lambda)X + y(\lambda)Y + z(\lambda)Z \tag{11.5}$$

注意到三个函数在任意 λ 值处均大于零,所以 $mono(\lambda)$ 总是三原色的凸线性组合。

那么 X、Y 和 Z 是怎么确定的?它们是通过对等式(11.4)中的颜色匹配函数即 $r(\lambda)$、$g(\lambda)$、$b(\lambda)$ 进行仿射变换来定义的,$x(\lambda)$、$y(\lambda)$ 和 $z(\lambda)$ 都是这些曲线的特殊的线性组合。为了方便起见,要求 $y(\lambda)$ 和**光效函数**有同样形状,**光效函数**即人眼对单位辐射亮度不同波长的单色光的反应函数。这就使得任意颜色包含的 Y 原色的量和它的总体亮度相等。

再次将色度值归一化以保持单位亮度是很有用的,因此我们定义:

$$\bar{x}(\lambda) = \frac{x(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)}, \quad \bar{y}(\lambda) = \frac{y(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)},$$

$$\bar{z}(\lambda) = \frac{z(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)}$$

显然有 $\bar{z}(\lambda)=1-\bar{x}(\lambda)-\bar{y}(\lambda)$。图 11.9 展示了参数曲线 $\mathbf{s}(\lambda)=(\bar{x}(\lambda),\bar{y}(\lambda),\bar{z}(\lambda))$,它完全位于 xyz 空间的第一卦限内。

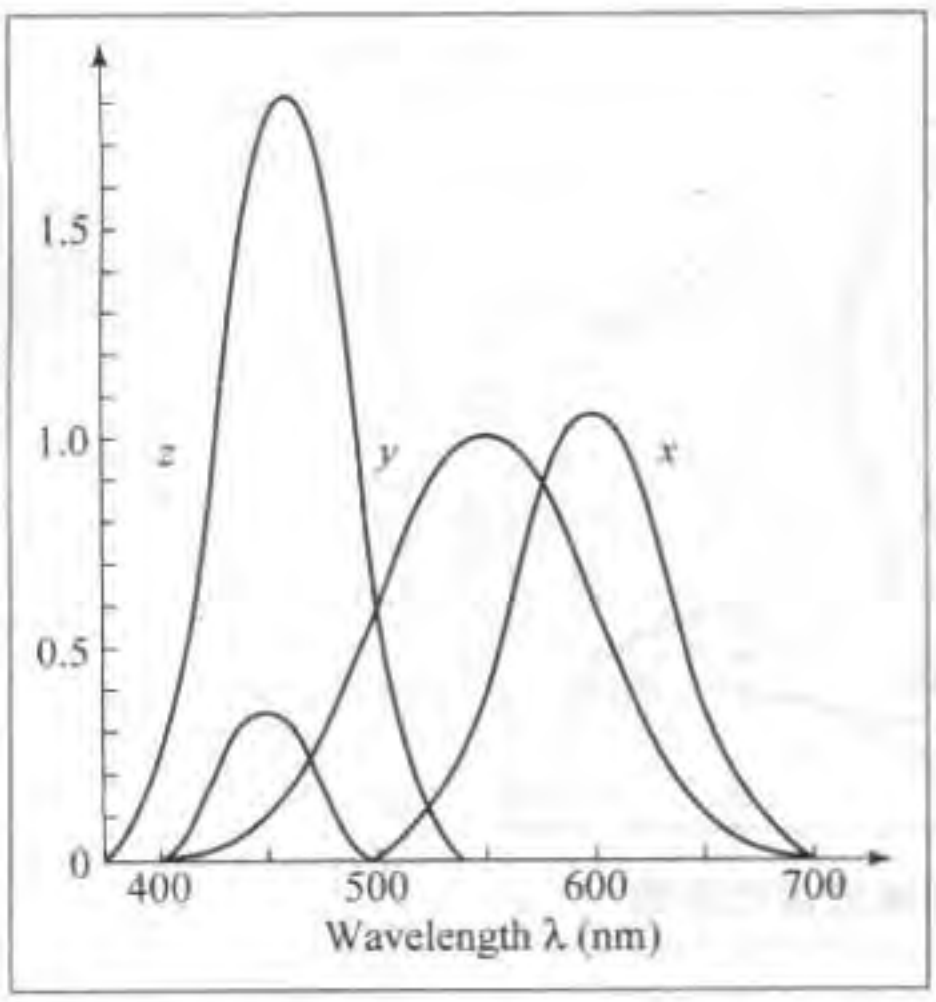


图 11.8 X、Y、Z 三原色的颜色匹配函数

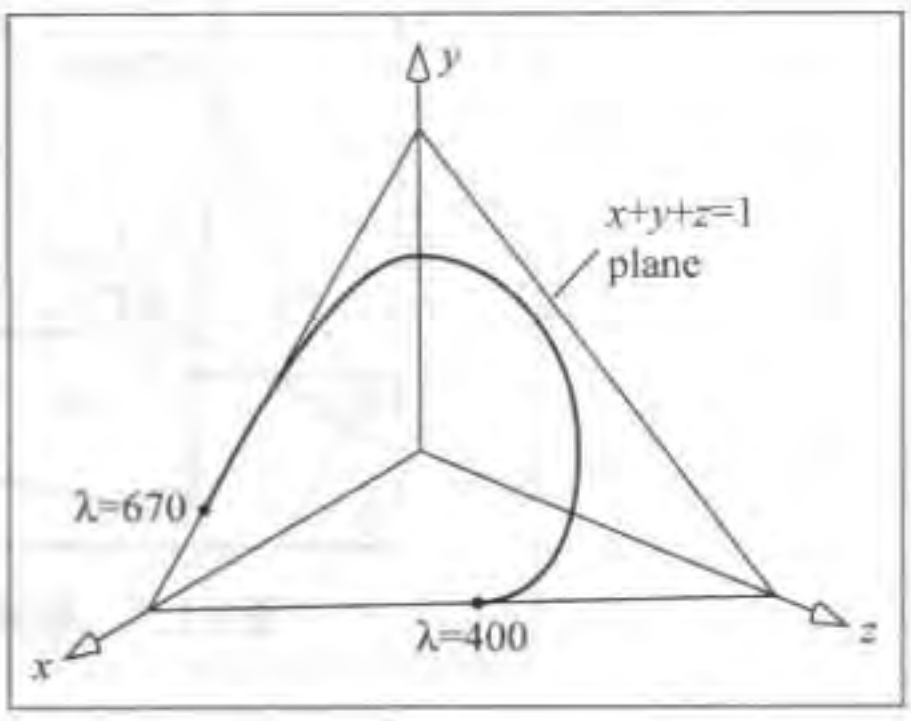


图 11.9 构造 CIE 标准

这三种原色的特定性质及该仿射变换的计算细节,可以从各种参考书中找到,如【Billmeyer81】,【Conrac85】。但是即使不知道这些,也可以理解和使用 CIE 标准。

11.3.1 构造 CIE 色度图

虽然光谱颜色曲线 $s(\lambda)$ 是三维的，但是它位于平面 $x+y+z=1$ 内，所以可以很容易用二维图形表示它的形状，而且可以打印出来用于查阅。描述一种（单位亮度）颜色只需要指定 x 和 y 即可，因为给定 (x,y) 可以很容易的计算 z（如何计算？）。

所谓标准 **CIE 色度图**，就是图 11.10 中所示的曲线 $s'(\lambda)=(\bar{x}(\lambda),\bar{y}(\lambda))$。

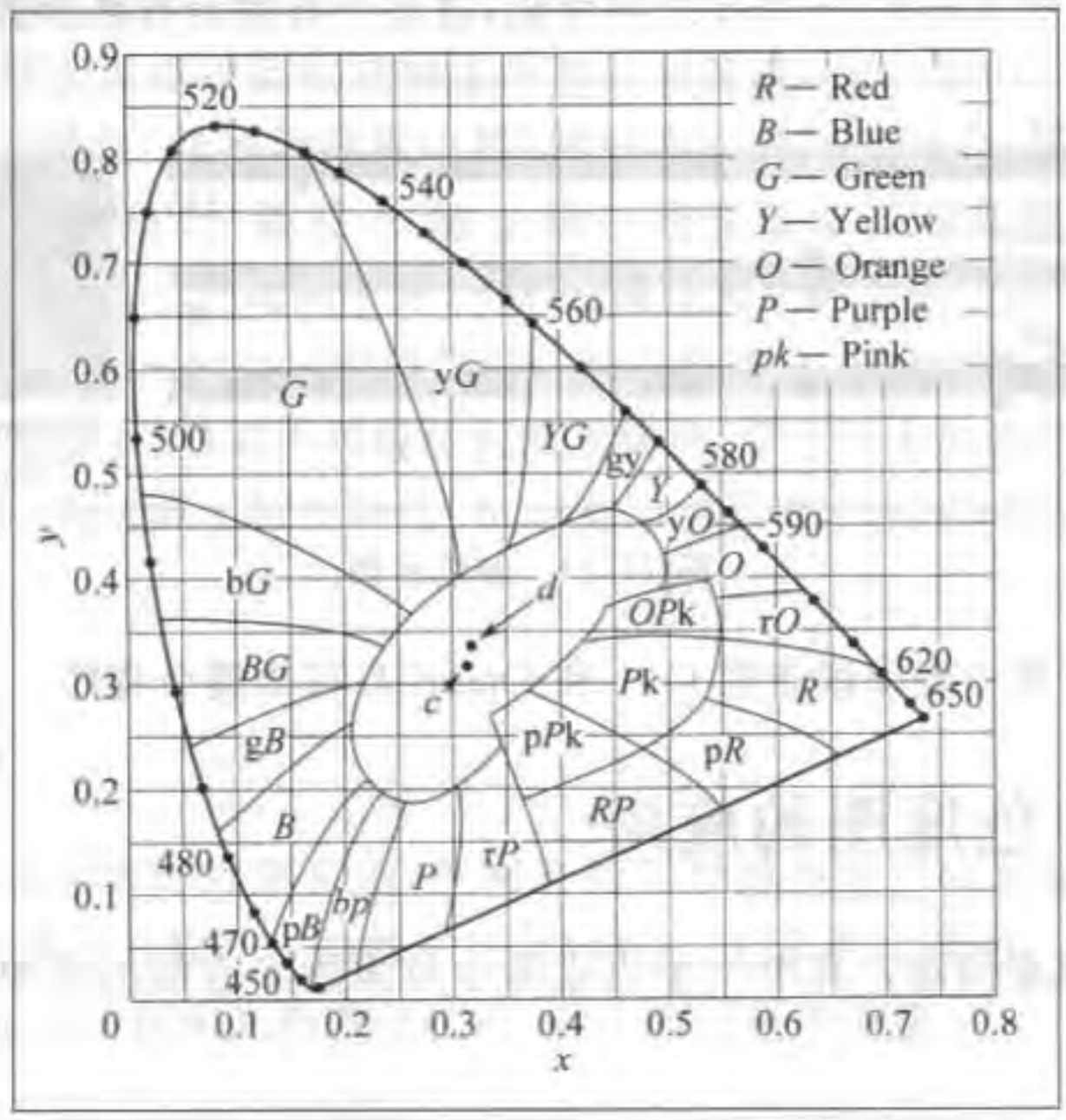


图 11.10 CIE 标准色度图

（想象沿 z 轴方向查看图 11.9 中的三维曲线的正交投影）这个图展示了纯光谱色的马蹄形轨迹，其中标签为相应颜色的波长。马蹄形内部是所有其他可见色，马蹄形以外区域不对应任何可见色。

图中各个区域分别用常用的颜色描述词汇标记。例如（0.6，0.3）点附件看起来是红色。遗憾的是，图中相同的距离并不对应相同的视觉色差。例如，区域 G 内的微小位置变化只会引起微小的视觉色差。相反，区域 B 和 Y 内很小的位置变化就会导致很大的视觉色差。

CIE 色度图定义了一些特殊点。位于 $(x,y)=(0.310,0.316)$ 的 c 点是被称为 **C 光源**的白色，它被认为是“完全不饱和”色。它经常被作为基准白色用于校准某些图形显示器。C 光源的颜色类似于正午多云的天空。位于（0.313，0.329）的 d 点是理想黑体加热到“白热”温度，即 6504K 时发射出的光。这是比 C 光源稍微带一点绿的颜色。许多其他颜色，如白炽灯的钨丝、月光、红热的铁等，在特定温度下发射出的光都经过了准确度量（【Conrac85】和【Rogers85】）。

CIE 色度图的重要价值在于提供了一个描述颜色的世界标准。人们设计了专门仪器用

来生成马蹄形内部任意点(x,y,z)表示的颜色,因而还可以通过仔细匹配来度量任意颜色。人们还设计出设备用来给指定颜色样本自动计算(x,y,z)。接下来可以看到,这个图还可用来对颜色进行一些重要计算。

DIY:颜色控制测试

接下来我们介绍一个程序,可以用鼠标直接控制颜色的 r、g、b 值,同时可以看到最后的颜色。图 11.11 展示了这个程序使用的颜色表,同时还有三个滑块用于调节 r、g 和 b 的值。缓慢调节每个滑块,使其值在 0～255 之间变化,这是个有趣而有启示意义的过程。

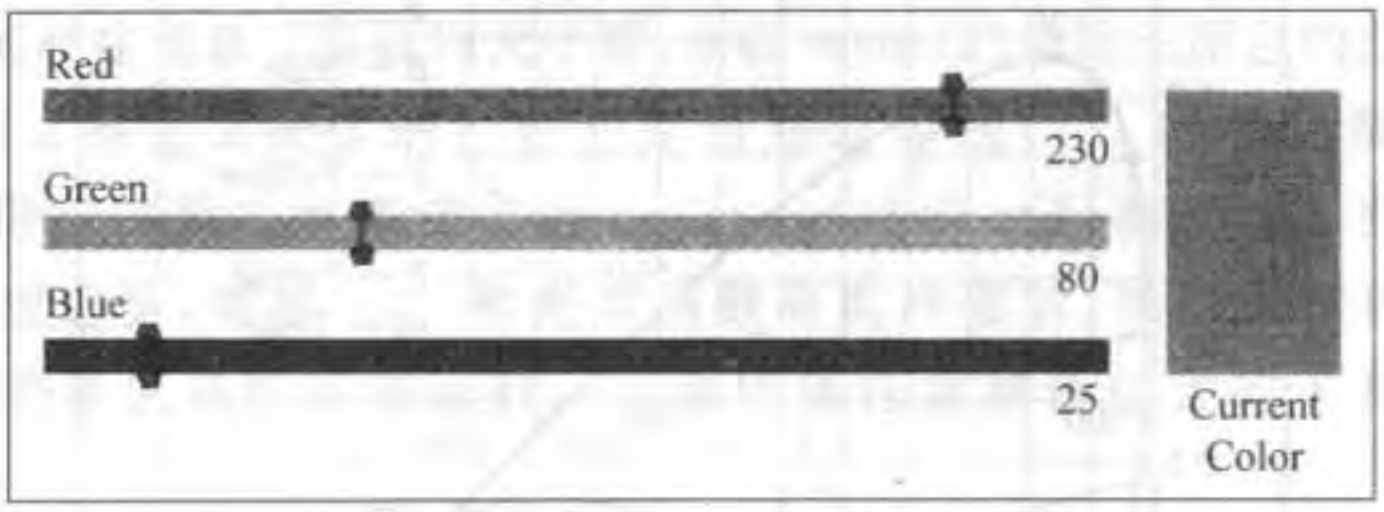


图 11.11 颜色示例

见练习 11.3.1,其中要求读者用 C++ 和 OpenGL 写出整个程序。

11.3.2 CIE 色度图的应用

CIE 色度图有很多用途。其中一些用途源自对该图中直线的简单解释,就像图 11.12 中所提示的。

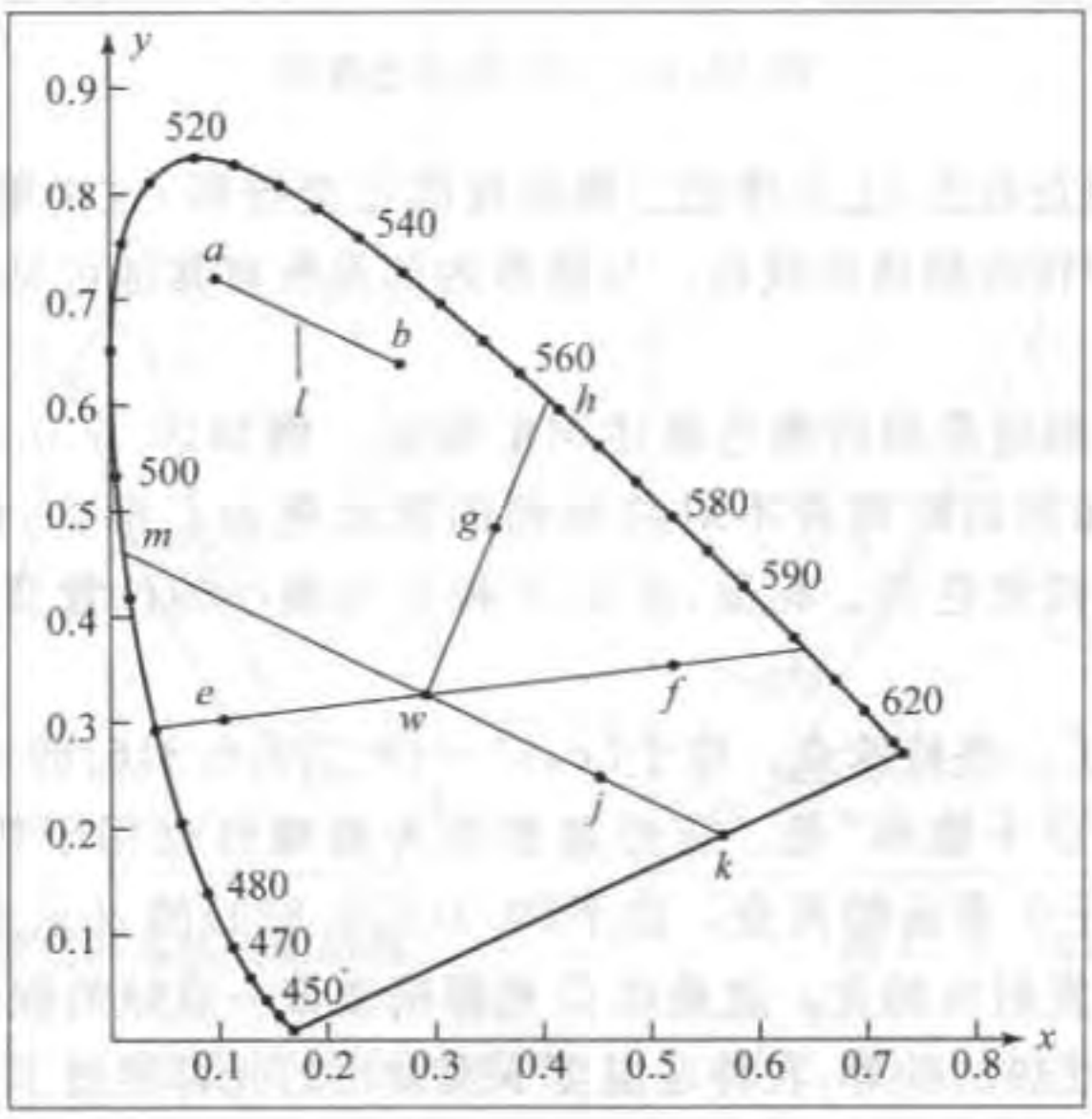


图 11.12 CIE 色度图的应用

考虑两个颜色 a 和 b 之间的线段 l。l 上任意点都是 a 和 b 的凸组合(回顾第 4 章),其形式为 $(\alpha)a+(1-\alpha)b$,其中 $0\leqslant\alpha\leqslant1$。每个点都代表合法的颜色(其 x、y 和 z 分量值和为 1,为什么?),因此可以断言,线段上任意点(并且只有这些点)的颜色可以通过在屏幕上叠加一定量的颜色 a 和 b 来产生。

当两种颜色相加,和为白色时,我们说这两个颜色是互补色(和白色的选择有关)。所以在图 11.12 中 e(蓝-绿)和 f(桔-粉红)关于 w 互补,因为一定量的这些颜色相加产生白色 w。图 11.13 列出了一些常见的互补色对,后面将进一步讨论。

红色—青色
绿色—洋色
蓝色—黄色

图 11.13　互补色对

CIE 色度图也可用于度量给定颜色的主导波长和纯度,例如图中的 g。g 一定是某纯光谱色(位于马蹄形边界)和标准白的线性组合。为了确定具体是哪种光谱色,只要从 w 向 g 画一条直线得到和马蹄形的交点(这里是 h 点),然后测量 h 处的波长——此例中为 564nm,微带黄色的绿色。类似地,饱和度或称纯度即为两个距离的比值:gw/hw。j 点的颜色没有主导波长,因为延长线段 wj 和马蹄形的交点 k 位于所谓的紫色线上,而它不对应任何纯光谱色(紫色线上的颜色是由红色和紫色组成的)。在这种情况下,先计算 j 的互补色(在 m 处),然后用互补色的波长加后缀 c 表示主导波长:498_c。

11.3.3　色域

CIE 色度图在定义色域时特别有用,色域是指某个设备能产生的颜色的范围。例如,CRT 显示器只能显示其荧光粉产生的基本红、绿和蓝三原色的组合色。图 11.14 显示了一种常见的彩色 CRT 显示器的荧光粉颜色红色、绿色和蓝色的位置。

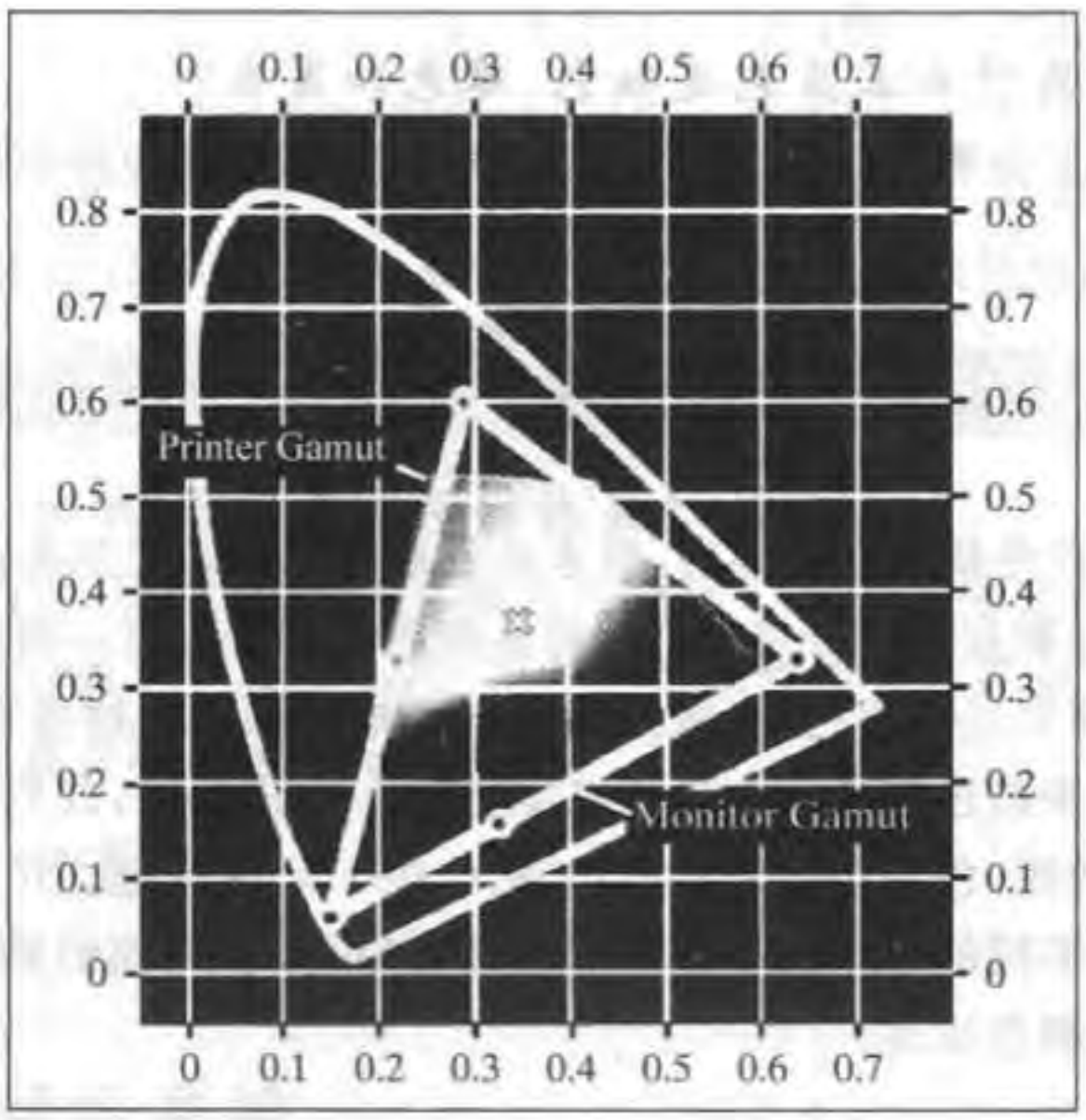


图 11.14　普通 CRT 显示器的色域

图 11.15 显示了某个 CRT 显示器的三原色的具体位置(见【Stone88】)。

原色	x	y
红	0.628	0.330
绿	0.258	0.590
蓝	0.1507	0.060

图 11.5 普通 CRT 显示器的 CIE 坐标

由图可见,这三个点定义了一个三角形区域。三角形内部任何点的颜色是这三原色的凸线性组合,所以都能显示[①]。三角形以外的颜色不属于此显示器的色域,所以无法显示。白色属于此色域,反映了一个众所周知的事实,就是适当数量红、绿、蓝相加产生白色。

同时显示的还有一个彩色打印机的色域。由于颜色在白纸上的形成机制,颜色不是直接的加法系统(见 11.4.2 节),所以其色域不是简单的三角形。这个色域比 CRT 显示器的要小,因此有些显示器能显示的颜色,这个打印机不能产生。另一方面,打印机的色域包含显示器的色域之外的一些点,因此有些颜色打印机可以产生,CRT 显示器却不能。

注意到无论如何选择三原色,即使选择为马蹄形边界上的点,三角形色域总是不能覆盖所有可见色,因为马蹄形总是会凸出到它内部任何三点构成的三角形之外。红色、绿色和蓝色是三原色很自然的选择,因为它们在 CIE 图中距离比较远,从而其色域可以覆盖色度图的大部分面积。相反,比如取黄色、青色和洋红色作为三原色,色域就会小得多(还能产生白色吗?)。

练习

练习 11.3.1 控制颜色表

以图 11.11 为指导,用 C++ 和 OpenGL 开发一个程序,允许用户调节样本颜色的 r、g 和 b 三分量的值。

练习 11.3.2 为什么三原色是红色、绿色和蓝色?

从物理学和哲学上说明为什么人眼中的视锥细胞对红、绿和蓝色最敏感。

11.4 颜色空间

CIE 对颜色的描述是精确而标准的,但未必是最自然的。尤其是在计算机图形学中,最自然的方法是对红、绿和蓝色进行组合得到目标颜色。有些人更习惯于用色调、饱和度和亮度来描述颜色,而画家则经常用色泽,浓淡和色调来描述颜色。这些都是**颜色模型**的例子,即选择 3 个"描述符"来描述颜色。如果可以度量这 3 个描述符,我们就可以用三元数来描述颜色,例如(色泽,浓淡,色调)=(0.125,1.68,0.045)。这就建立了一个用于描述颜色的三维坐标系统。选择不同的坐标架导致不同的**颜色空间**,因此我们需要算法实现在不同颜色空间之间互相转换颜色描述。

---

① 美国通信标准委员会(NTSC)定义了 CIE 坐标下的标准三原色:红色=(0.670,0.330,绿色=(0.210,0.710),蓝色=(0.140,0.080)【Rogers85】。

11.4.1 RGB 颜色空间

RGB(红、绿、蓝的简称)颜色模型描述颜色为适当定义的红、绿和蓝三原色的凸线性组合,如等式(11.3)所示。若限制 r、g 和 b 三个数值都在 0 和 1 之间变化,则所有可定义的颜色位于图 11.16 所示的立方体内部。

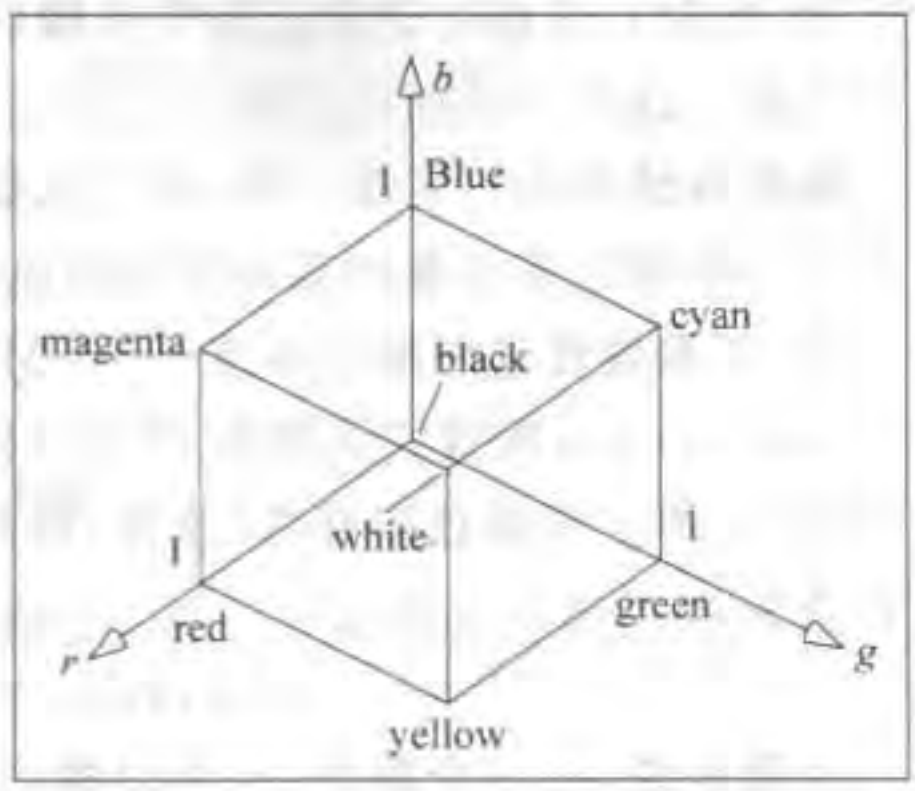


图 11.16 RGB 颜色立方体

和 CIE 色度图不同,RGB 模型不对颜色亮度进行归一化。靠近(0,0,0)的点较暗,而远离它的点较亮。例如(1,1,1)对应纯白色。对计算机图形学而言,这个颜色空间是最自然的,因为颜色描述如(0.3,0.56,0.9)可以直接翻译为颜色查找表(LUT)中的值。注意到标记洋红色的角落表示红色加蓝色可以得到洋红色,标记黄色和青色的角落可以同样解释。在体对角线上的颜色是互补色。

当归一化至单位亮度时,这里定义的所有颜色在 CIE 色度图上都有对应位置,只要提供合适的红、绿和蓝色分量值。(这个色域是什么形状?)不难将 CIE 坐标(x,y,z)描述的颜色转化至(r,g,b)空间,反之亦然。由于 R、G、B 三原色都是 CIE 标准中 X、Y、Z 三原色的线性组合,所以线性变换即可完成上述任务。这个变换由 R、G、B 三原色及白色的定义确定。对于图 11.14 中给出的三原色和图 11.10 中 d 点指定的白色,要求的线性变换如下(详见【Rogers85】):

$$(r,g,b)=(x,y,z)\begin{pmatrix} 2.739 & -1.110 & 0.138 \\ -1.145 & 2.029 & -0.333 \\ -0.424 & 0.033 & 1.105 \end{pmatrix} \tag{11.6}$$

显然,用此矩阵的逆,即可从 RGB 颜色空间变换到 XYZ 颜色空间。

练习

练习 11.4.1 从 RGB 颜色空间转化至 CIE 空间

计算前述矩阵的逆,从而得到 RGB 颜色空间到 CIE 空间的变换。CIE 空间的一个重要性质是所有颜色可以表示为 X、Y、Z 三原色的凸线性组合。这会导致逆矩阵有何性质?你计算得到的逆矩阵是否满足这个条件?

练习 11.4.2 计算分量

推导计算等式(11.6)中矩阵的各个元素并计算其逆。

11.4.2 加/减色系统

到目前为止,我们讨论的都是由有色光各分量的叠加来得到新颜色,这是一个加色过

程。**加色系统**用一定量的三原色的和,通常为红、绿和蓝,来表示一种颜色 D:$D=(r,g,b)$。加色系统可以用任何三种颜色作为三原色,但是由于红、绿、蓝在 CIE 图上相距较远,它们提供了较大的色域。

当用颜色消除法表示颜色更自然时,我们用减色系统来描述颜色。当光线在物体表面上反射(散射),或在半透明介质中传输时(例如使用摄影滤光器时),有些颜色就会被介质吸收而去除。这是一个减色过程。

减色系统和加色系统一样,用三元数表示一种颜色 D,但是 3 个数中每一个指定需要从白色中去除相应多少量的某种特定颜色(相应原色的互补色),从而生成颜色 D。为了说明这一点,考虑最普通的减色系统——CMY 系统,它使用减法三原色:青、洋红和黄色。如果 $D=(c,m,y)_{CMY}$,则我们说颜色 D 可以由白色去除 c 份青色的补色(即红色),m 份洋红的补色(绿色),和 y 份黄色的补色(蓝色)得到。因此,可以立即得到下述 RGB 系统和 CMY 系统的关系:

$$(r,g,b)_{RGB}=(1,1,1)-(c,m,y)_{CMY} \tag{11.7}$$

也就是说,在一个颜色中,可以通过增加 y 来减少蓝色分量 b,因为 y 指定了从白色中去除黄色的补色的数量。

图 11.17 举例说明了减色过程。三个玻璃片在 CMY 系统中被描述为 $(0.4,0.5,0.2)_{CMY}$。当 RGB 加色系统中的白色光 $(1,1,1)_{RGB}$,透过青色玻璃片后,40%红色成分被吸收,产生了青色光 $(0.6,1,1)_{RGB}$。当光又透过洋红色玻璃片后,50%绿色成分被吸收,生成颜色 $(0.6,0.5,1)_{RGB}$。(这是什么颜色?)最后,当这个光透过黄色玻璃片后,20%蓝色成分被吸收,产生颜色 $(0.6,0.5,0.8)_{RGB}$。

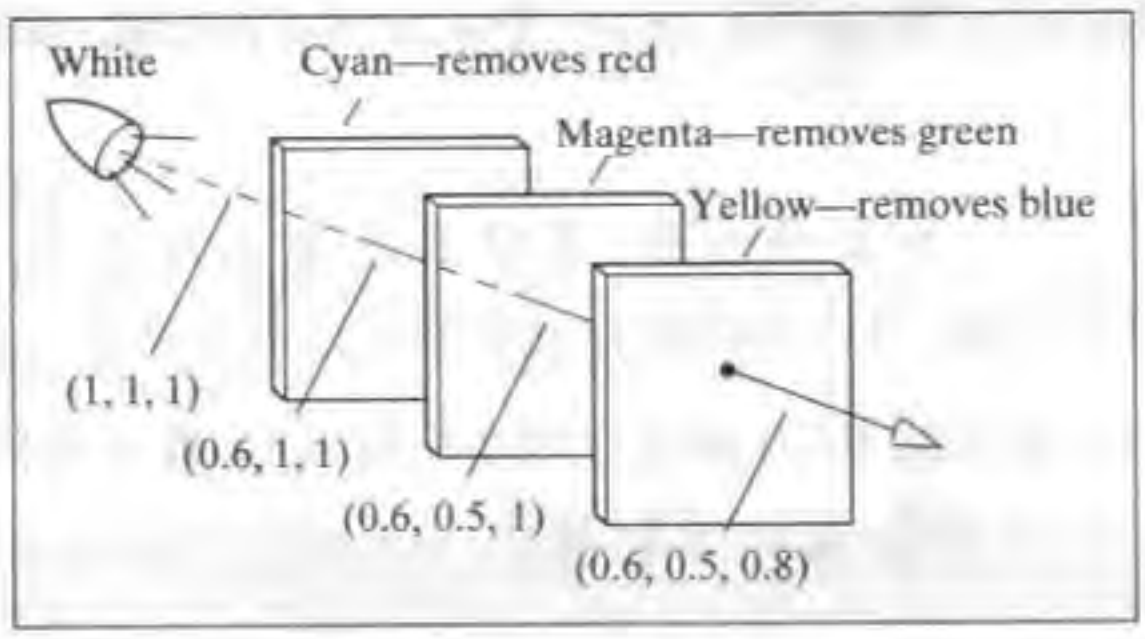


图 11.17 减色过程

现在有色表面的散射可以类似地描述。例如在三色打印过程中,青、洋红和黄色颜料和透明颜料混合在一起。添加任一种颜色都会导致相应的互补色减少。例如,若洋红色颗粒和无色颜料混合,则此颗粒会去除白光中的绿色成分而只反射红色和蓝色成分。通过 CMY 三原色的混合,此减色系统在颜色硬拷贝设备中用来产生目标颜色。

图 11.18 显示了加/减色系统的三原色及其相互作用。图 11.18(a)中,红、绿和蓝色光线共同照射到白色表面上。当这些光线重叠时,它们就会组合形成新的颜色。例如,红色和绿色相加形成黄色。当三个都重叠时,白色就形成了。

图 11.18(b)展示了相反的情况。每个圆是通过加入图示颜色的墨水形成的。在白光下观察此图,有个圆盘呈现黄色,因为其颜料从白色中减去了其补色(蓝色)。当黄色和青色

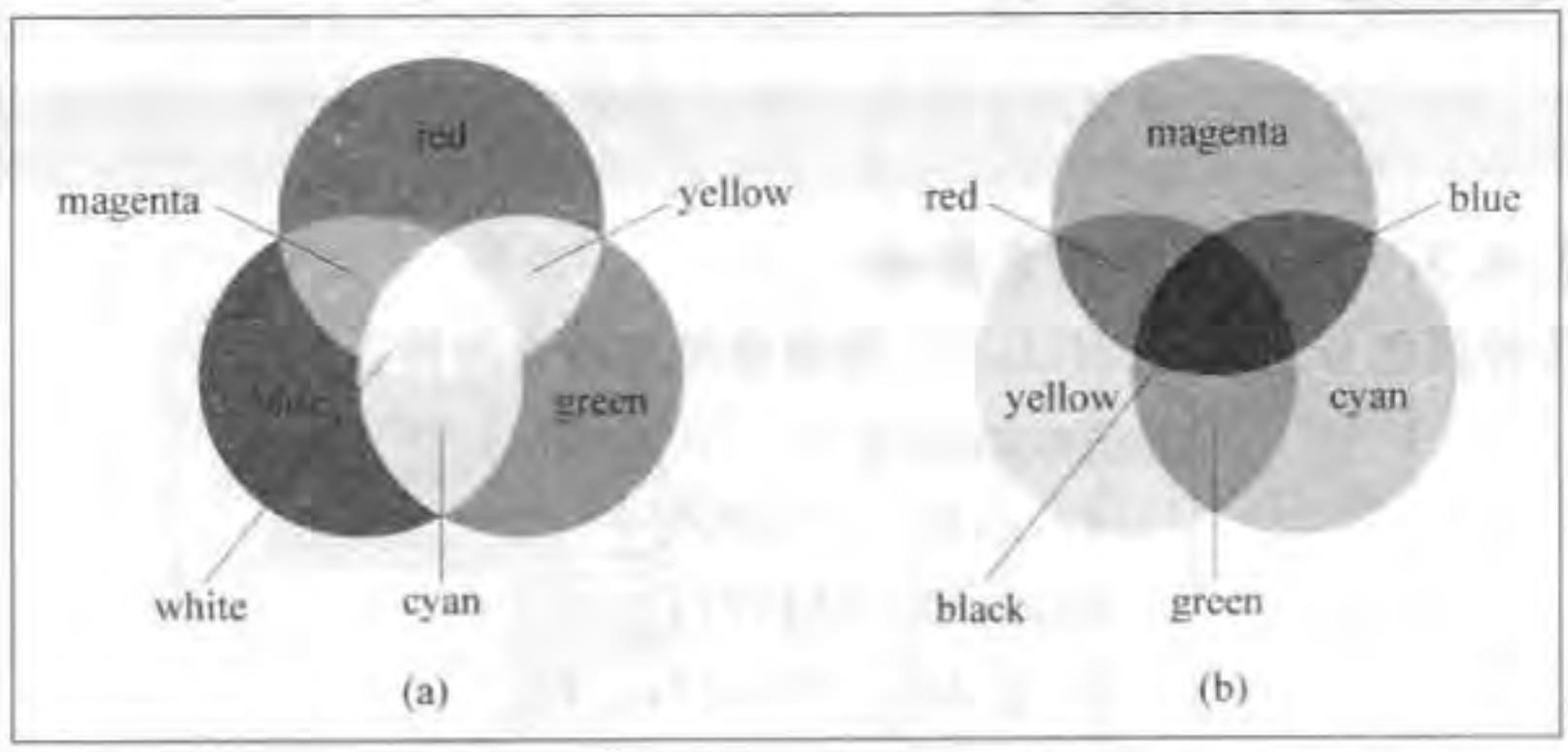


图 11.18 加/减色系统

颜料混合时，我们看到了绿色，因为蓝色和红色成分都已去除。中心部分为黑色是因为所有颜色成分都已去除。

11.4.3 HLS 颜色模型

一个更自然的颜色模型用色调(H)、亮度(L)和饱和度(S)来描述颜色，因为这些量是人眼容易感知和区分的。此模型由将 RGB 立方体扭曲为底部相连的两个六棱锥得到，如图 11.19 所示。

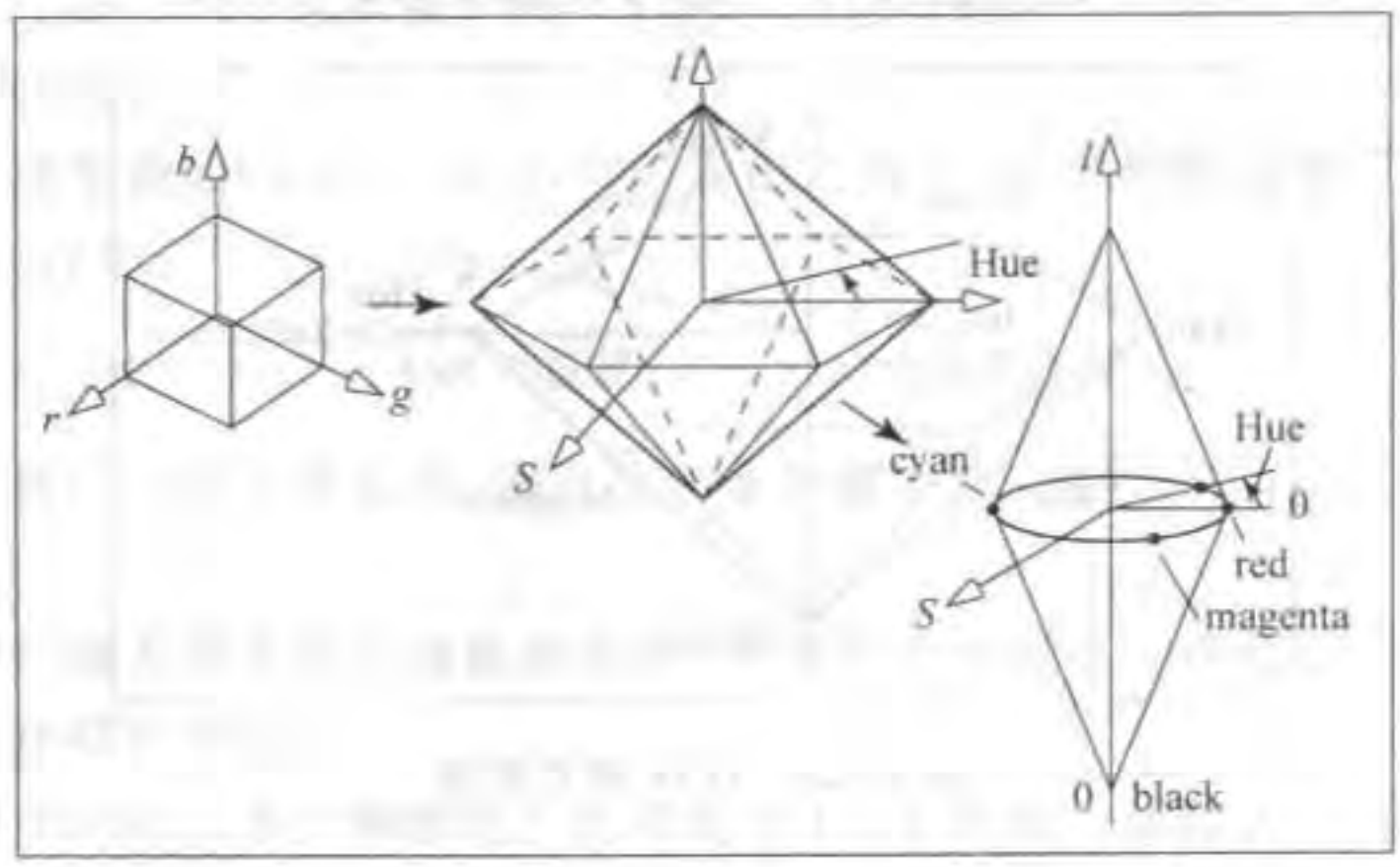


图 11.19 把 RGB 系统变形为 HLS 系统

沿从(1,1,1)到(0,0,0)的体对角线看 RGB 正方体，我们看到六个主要色调(红、绿、蓝及其互补色)分别位于六边形的顶点。故色调可以和 0～360 的角度关联。习惯上 0 度对应红色。当 RGB 三个分量都在 0 和 1 之间变化时，亮度相应在 0 和 1 之间变化。这刚好和 RGB 立方体中白色到黑色的体对角线相对应。一种颜色的饱和度，大致上为其到 RGB 立方体的体对角线的距离，映射到 HLS 锥中为其到亮度中轴的径向距离

HLS 颜色空间是基于 RGB 空间的变形，这引导我们寻找一种算法把颜色从一个系统转化至另一个系统。这里的变形非常复杂，因此我们不必坚持精确的几何变换。把颜色从 RGB 空间转换至 HLS 空间的算法最关心对 R、G、B 分量的最大值和最小值的转换。该算

法提供了一个有用且可逆的转换工具。

练习

练习 11.4.3 完成一些颜色转换

对下列几种颜色分别计算(H,L,S),并解释结果的合理性:

$$(r,g,b)=(0.2,0.8,0.1)$$
$$(r,g,b)=(0,0,0.8)$$
$$(r,g,b)=(1,1,1)$$
$$(r,g,b)=(0,0.7,0.7)$$

练习 11.4.4 HLS-to-RGB 转换

写一个算法 HLS-to-RGB(),将颜色从 HLS 空间转换至 RGB 空间。要求提供前述 RGB-to-HLS 变换的反变换:对 RGB-to-HLS 变换得到的 H、L、S 值应用此变换能恢复原来的 R、G、B 值。

练习 11.4.5 HSV 颜色模型

另一种颜色模型,色调(H)、饱和度(S)和值(V)系统也是基于对 RGB 立方体的变形,但它是单锥体,而非双锥体,如图 11.20 所示。和 HLS 系统一样,在此模型中,色调也被映射为角度,六边形变形为圆,饱和度的含义也是一样的。值 V 记录颜色的亮度,取值范围为 0~1,如图所示。写一个算法把颜色从 RGB 空间转换至 HSV 空间。

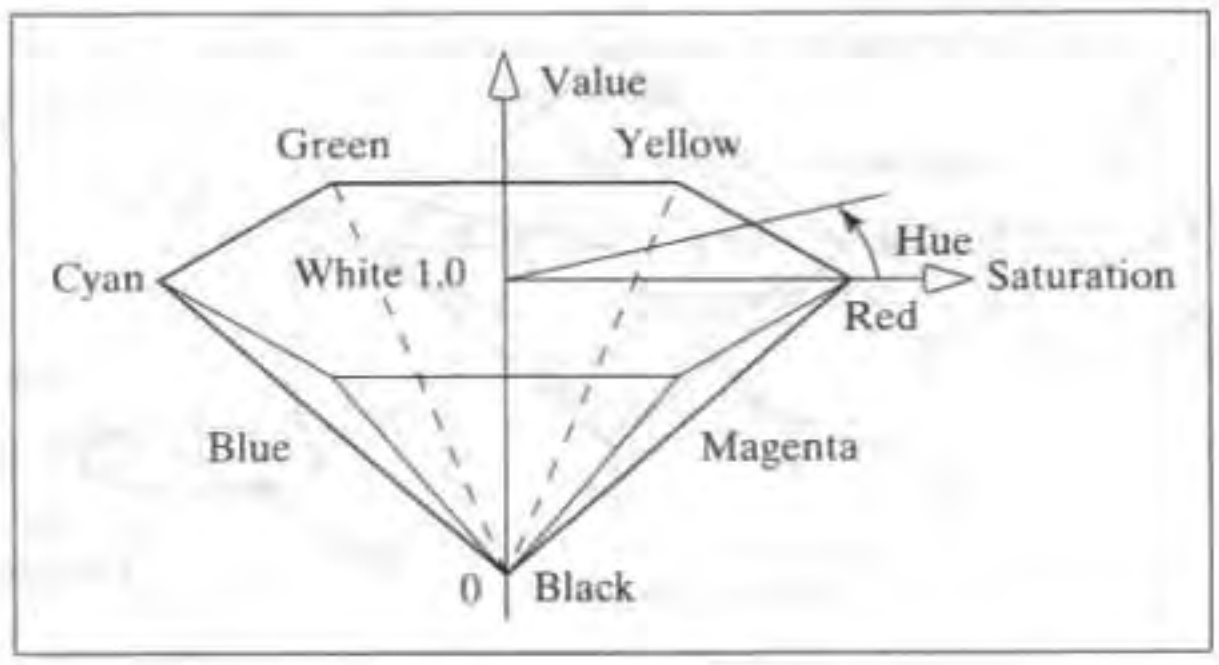


图 11.20 HSV 颜色模型

11.5 索引颜色和颜色查找表

有些图形系统用间接方法关联像素值和颜色。它们使用颜色查找表(简称 LUT),像素值和最后颜色之间的关联可由编程确定。图 11.21 展示了一个简单的例子。这里色深为 6,但是每个像素的 6 位数值在驱动 CRT 之前需要经过一个中间步骤。这些像素值是一个表格的索引,这个表格有 64 个单元格,LUT[0],…,LUT[63]。(为什么正好是 64 项?)例如,若像素值为 39,LUT[39]中存储的值就被用来驱动 DAC(数模转换器),而不是直接用

数值 39 驱动。如图所示，LUT[39]中为 15 位数 01010 11001 10010，其中前 5 位(01010)用来驱动“红色 DAC”，中间 5 位驱动“绿色 DAC”，最后 5 位驱动“蓝色 DAC”。

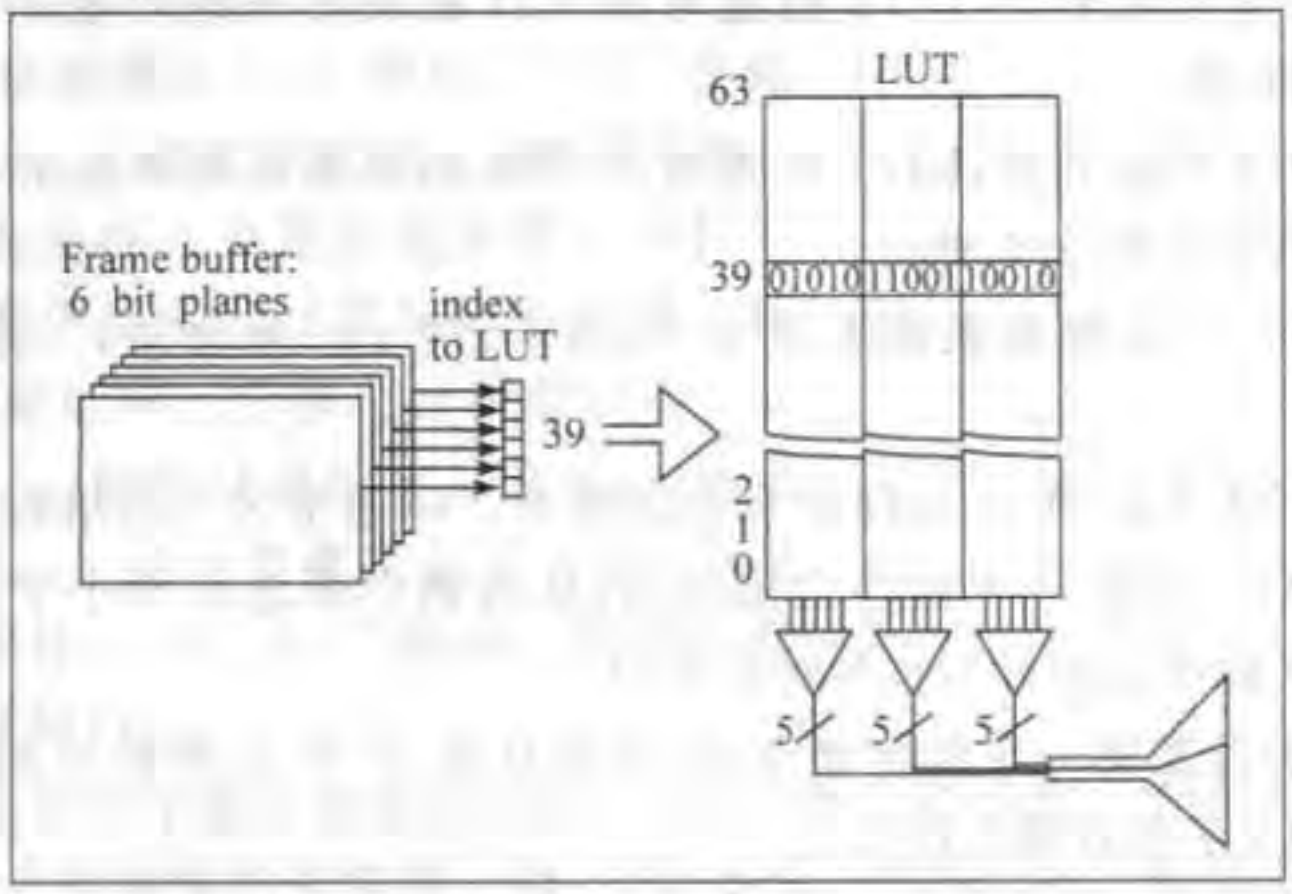


图 11.21 一个采用 LUT 的彩色显示系统

每个 LUT[]单元格可由程序用某些操作系统编程接口如 setPallette()设置。例如，指令：

```
setPalette(39, 17, 25, 4);
```

将设定 LUT[39]为 15 位数值 10001 11001 00100(因为 17、25、4 的二进制数值分别为 10001、11001、00100)。

为了使某个像素例如(x,y)=(479,532)发出此颜色，可以用预先定义的 drawDot()把数值 39 存入帧缓存中：

```
drawDot(479, 532, 39);          //设置(479,532)处像素值为 39
```

每次帧缓存被“扫描”至显示器，读出这个像素值为 39，接着 LUT[39]中的数值被用来驱动 DAC。

上述可编程性极大地便利了颜色的选择，但是自然要牺牲一些代价：程序(或程序员)必须计算出要使用哪种颜色！

这个系统最多能显示多少种颜色？此系统中 LUT 的每一项包含 15 位，故每项可以设为 2^{15}=32K=32 768 种颜色之一。系统可显示的 2^{15} 种颜色集合被称为调色板，故我们称这个显示器“有 32K 色调色板”。

问题是每个像素值在 0～63 范围内，所以 LUT 一次只能存储 64 种不同颜色。因此这个系统一次最多只能显示 64 种不同颜色。这里的“一次”是指描扫完整个帧缓存的过程——类似于 1/60 秒左右。在扫描一幅图像的过程中，LUT 的内容是不能改变的，所以整个扫描过程使用固定的 64 种调色板颜色。虽然程序可以在两次连续扫描的极短间隔内改变 LUT 的内容，但是通常 LUT 的内容在多次扫描过程中都保持不变。

概括地说，假设某光栅显示系统色深为 b(则其帧缓存有 b 个位平面)，每个 LUT 单元格位宽为 w，那么这个系统最多可以显示 2^w 种颜色，一次最多 2^b 种颜色。

例 11.5.1

(1) 一个有 $b=8$ 个位平面,LUT 位宽为 $w=12$ 的图形系统可显示 4096 种颜色,一次可显示其中任何 256 种。

(2) 一个有 $b=8$ 个位平面,LUT 位宽为 $w=24$ 的图形系统可显示 $2^{24}=16\ 777\ 216$ 种颜色,一次可显示其中任何 256 种。

(3) 若 $b=12$,$w=18$,则此系统可显示 $256K=262\ 144$ 种颜色,一次可显示 $2^{12}=4096$ 种颜色。

位平面数 b 和 LUT 位宽 w 之间没有强制关系。w 通常为 3 的倍数,从而每个 DAC 由同样多位驱动($w/3$)。另外,b 永远不会超过 w,从而调色板总是不小于系统一次能显示的颜色数。(为什么永远不会设计 $w<b$ 的系统?)

注意到 LUT 本身需要一定的存储空间,虽然只有 2^b 个 w 字长空间。例如,若 $b=12$,$w=18$,则需要 9 216 字节存储 LUT。

光栅显示系统采用 LUT 的动机是什么?一般程序都要尽量减少内存占用。增大 b 会大量增加帧缓存需要的内存,主要是因为图像中像素太多。大量的内存占用会显著增加整个系统的代价。

11.6 颜色量化

渲染场景时,会产生大量浮点颜色三元数(I_r,I_g,I_b),每个三元数对应图像中一个像素。第 9 章讨论了把这种颜色三元数转化为二进制的方法,从而可以传输至显示器或存档为文件以备后用。一种常用方法是把(三元数的)每个浮点分量转化为一个字节,从而图像就可以用"真彩色"——24bit/pixel(位每像素)来表示。扫描仪将照片数字化时,通常也生成真彩色图像文件。

但是有两个重要问题必须解决:

- 有些设备(如 PDA 或 GPS 系统)不能处理 24 位颜色。虽然不断下降的内存价格使得普通工作站的显示器有足够的色彩分辨率,但还是有许多设备色彩分辨率有限。它们可能只能处理 5 位的红、绿或蓝色,或者它们使用只有 256 项的颜色查找表(例如飞机场或超市的自助服务机)。
- 真彩图像可能很大,需要很大的磁盘空间和很长的网络传输时间

因此我们需要一些方法,减小彩色图像大小并减少其中包含的颜色数。例如我们需要这样一种方法,它能选择真彩图像中"最好"的 256 色,并对任何其他颜色,从这 256 色中找到"最接近的"一个代替。此过程通常称为**颜色量化**。

问题描述

给定 N 个颜色三元数,找到能最好地代表初始(N 种)颜色的 K(K 一般比 N 小得多)种颜色。对每种初始颜色,找到其"最接近"的替代色。有需要的话,将原始颜色替换为相应的最佳替代色的索引 $0,1,\cdots,K-1$。

这里有许多问题要解决，例如“最好”和“最接近”是什么意思？另外，由于 N 如此之大，任何好算法都必须足够高效。

我们接下来讨论四种减少图像颜色数的方法。在每种方法中，假设图像用 RGB 颜色三元数[①]存储于文件中——浮点三元数或者整数三元数都可以。

很多时候会先进行一个预处理步骤。即扫描图像文件确定图像中像素三元数的取值范围。特别地，此扫描过程要计算三个颜色分量的极值：红色分量的 r_{min} 和 r_{max}，绿色分量的 g_{min} 和 g_{max}，及蓝色分量的 b_{min} 和 b_{max}。这些数值定义了 RGB 空间中的一个平行六面体，如图 11.22 所示。则所有颜色全体就位于此区域内。

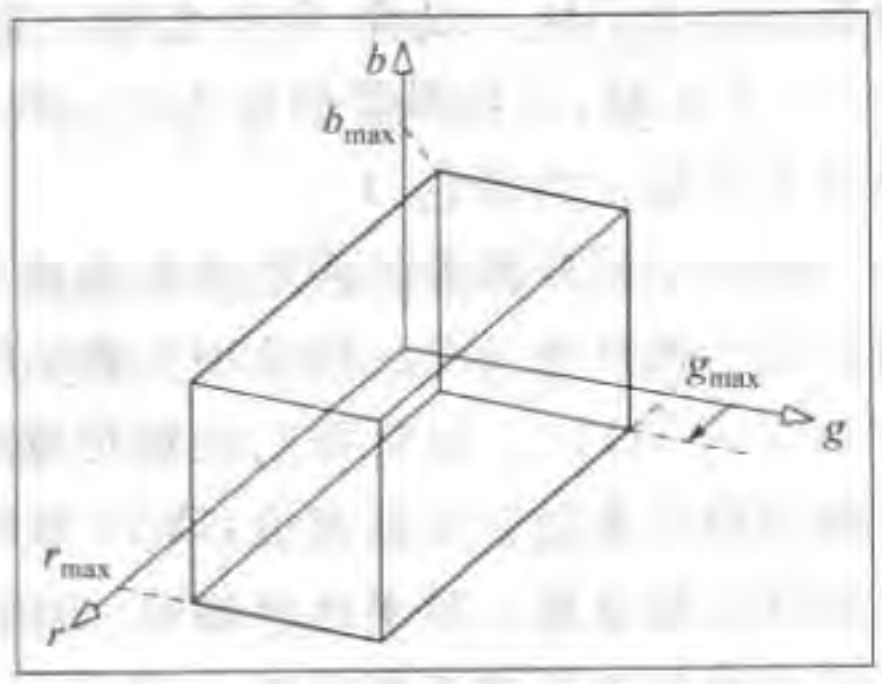


图 11.22　位于一个颜色块内的颜色全体

设想有大量点散布于此六面体的不同区域内，每个点代表一种颜色。有些颜色可能会重复很多次，有些可能紧紧的聚集在一起，可能还有一些孤立点，在这个六面体的一个很大区域内都没有相邻点。这就导致了很多问题：如果许多点聚集在很小的区域内，我们是否应该用该区域中心替代区域内所有点？如果有一种颜色重复出现多次，但是没有比较近的相邻点，我们是否该用高精度来表示它？

11.6.1　均匀量化

最简单的方法是沿坐标轴把颜色块分成一些互不重叠的子块，并选择每个子块接近中心的颜色值作为该子块的代表色。

那么如何细分颜色块？这里我们介绍一个最简单的方法。如图 11.23 所示，分别沿三个颜色轴将颜色块切片。这样就可以独立的选择代表色的红、绿和蓝色分量。例如，若

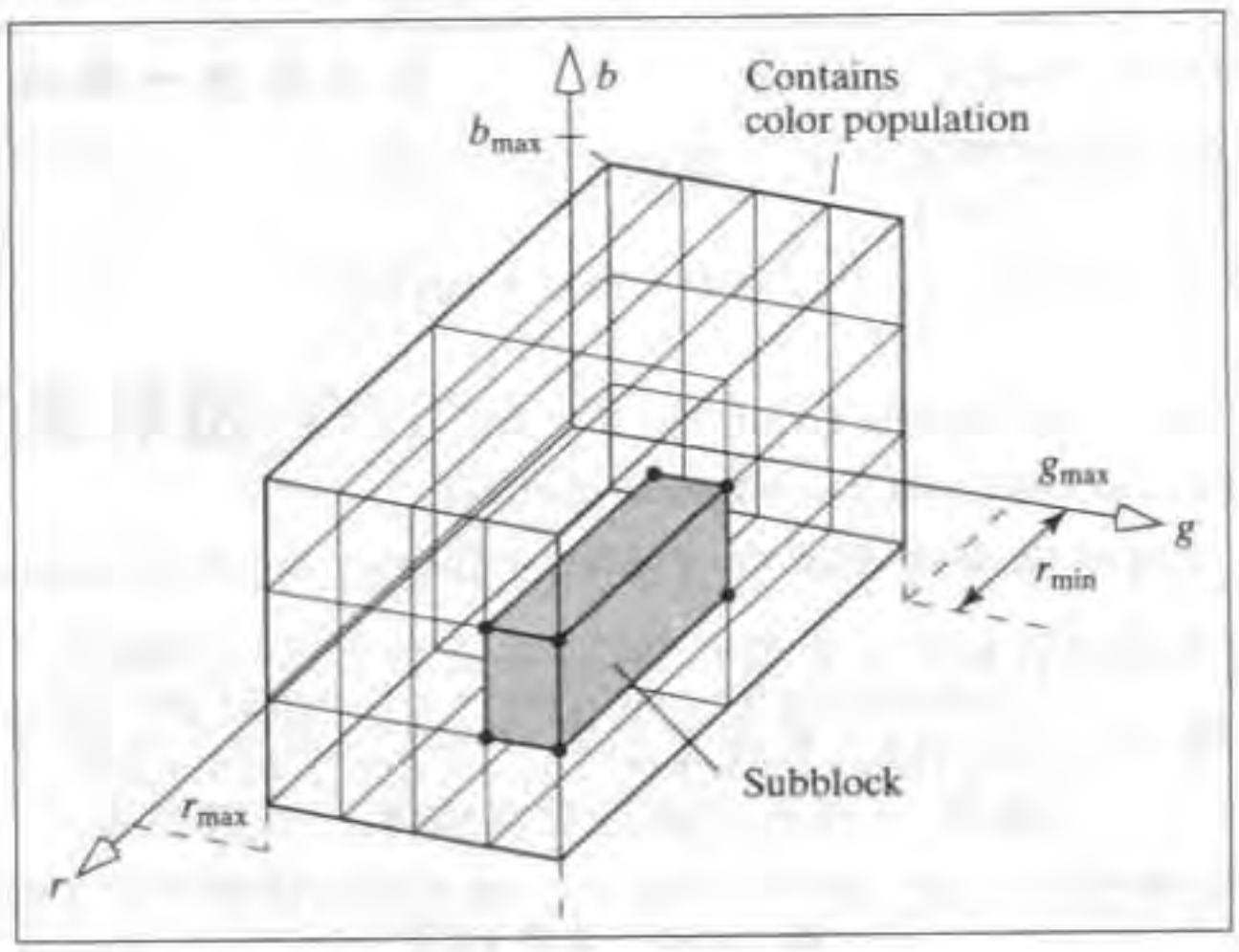


图 11.23　切分颜色块

① 为了改进量化结果，也可以先把 RGB 颜色转化至其他颜色系统，如 HSV 或 CIE，然后在新系统中进行量化。

$K=256$,则红色范围 $r_{min},\cdots,r_{max}$ 可分成 8 片,绿色也分为 8 片,而蓝色分为 4 片,从而正好得到 256 个子块。或者,我们也可以把红色和蓝色都分成 6 片,而绿色分成 7 片,从而产生 252 个子区域。(如果要使用 LUT,则有 4 个条目可以留空不用,或者它们可以用来存储边界或文本标记的颜色。)

还有个同样重要的问题是如何选择代表色,它们的数值必须可以用 LUT 记录。例如假设每个条目有 4 个比特位用于表示绿色分量,从而允许绿色的强度范围为 0～15(二进制为 0000～1111)。如果我们决定把绿色分成 6 块,那么该从 0～15 之间选择哪 6 个数值?注意到对大多数显示器来说,第 15 级绿色的实际亮度可以通过调节其前置面板上的控制旋钮来独立地设置。对于这种终端,必须对相对强度进行编码。下面的例子将帮助说明不同因素之间是如何相互影响的。

例 11.6.1

考虑一种典型情况:LUT 有 $K=256$ 个条目,每个条目分别用 4 位表示红色、绿色和蓝色的强度。扫描三元数图像文件后,发现 r_{min}、g_{min}、b_{min} 都很接近 0,而 $r_{max}=4.0$,$g_{max}=7.0$,$b_{max}=3.0$,那么该如何将颜色三元数映射为 LUT 值?

我们决定用 7 个代表描述绿色成分,用 6 个代表描述红色和蓝色成分。为了简单起见,我们在 16 个候选 LUT 值中尽可能均匀的选择代表色。对于红色和蓝色,6 个值分别设为 0、3、6、9、12 和 15 即可满足要求。对于绿色,我们不能达到完全等间隔的分割,但是把值设为 0、3、5、8、11、13 和 15 是个合理选择。

接下来,如何把这些值存储至 LUT 条目?创建一个整数数组 int r[6]存储 6 个红色值,类似地创建数组 g[]和 b[]。以 r[]为例,它保存 6 个数值 0、3、6、9、12 和 15。在下面的源代码中,三重嵌套的循环遍历了这三个数组中颜色的所有可能组合,并对每种组合生成一个 12 位数值并存储于 LUT[]数组:

```
#define numRed 6
#define numGreen 7
#define numBlue 6
#define pack(r,g,b) (256*(r)+16*(g)+(b))

for(rd=0; rd <numRed; rd++)
    for(grn=0; grn <numGreen; grn++)
        for(blu=0; blu <numBlue; blu++)
        {
            index=numRed * numGreen * rd +numBlue * grn +blu;
            LUT[index]=pack(r[rd], g[grn], b[blu]);
        }
```

图 11.24 装载 LUT

宏定义 pack 将红色值乘以 256,即将其左移至 LUT 条目的最高 4 位,对绿色进行类似位移处理。例如,当 rd、grn、blu 分别为 3、2、5 时,对应 LUT 中索引为 133,由 $r[3]=9$,$g[2]=5$ 和 $b[5]=15$ 三个数根据要求组合得到的数为 2399,对应 16 进制数为 $95f$。

当重新扫描图像文件时，三元像素值序列(r_i, g_i, b_i)中每个r_i分别和 6 个红色代表色比较，找出最接近的一个：找到j_r使得r_i离$r[j_r]$最近。例如，若$r_i=8.23$，则最接近的值为 9，故$r_i=8.23$。同样地计算j_g和j_b，然后可以计算索引值为$j=42*j_r+6*j_g+j_b$。

均匀量化方法有其缺陷：不考虑颜色的总体分布，并且扫描原始图像文件时没有统计任何信息（除了r_{max}、g_{max}和b_{max}三个数值）。然而这种方法可适用于某些图像，特别是当每个像素的比特位数较大且 *LUT* 提供的调色板也很大时。

图 11.25 显示的是一幅原始图像，图 11.26 显示了对它进行均匀量化的结果（真彩色版本可从此书主页找到）。注意到大量单色带出现在原图像中一些有颜色梯度的区域（但是变化很平缓）：许多相邻的近似颜色由单一代表色替代，从而导致这些区域颜色单一，缺少平滑过度。

图 11.25　本书作者的原始照片

图 11.26　量化至 256 色后的作者照片

练习

练习 11.6.1　试验一些颜色值

在例 11.5.1 中，试说出 LUT 的前 20 项的内容。这个排列是否有序？LUT[29]中存储的是什么？LUT[231]呢？

11.6.2　流行度算法

流行度算法【Heckbert82】和均匀量化方法相比至少考虑到有些颜色会经常在图像中出现。因此此算法给这些颜色以较大优先级，即使可能很多最流行的颜色会紧凑在一块。

它的基本方法是构造一个表格记录每种颜色在图像中出现的次数，然后对这个表格进行排序。那么，排序后的表格的前 K 种颜色就是最流行的 K 种颜色。

因为表示真彩色的 3 字节整数有很多种可能取值（2^{32}种），所以我们首先将红、绿和蓝色每个值截短为 5 位以简化问题，这样就只剩下 $2^{15}=32K$ 种可能的颜色值。于是分配一个长度为 32K 的数组并将所有元素初始化为 0。扫描图像，每读入一个颜色，就将数组中相应元素增一。读完所有 N 种颜色后，对这个数组进行排序，然后取 K 个最流行颜色为代表色。

接着重新扫描图像，将每个颜色用流行度表格中最接近的代表色$(r,g,b)[i]$替换。通

常采用最小平方距离作为"最接近"的度量：

求 i 使得 $(rd-r[i])^2+(grn-g[i])^2+(blu-b[i])^2$ 达到最小值

这是个代价昂贵的操作。

11.6.3 中值切分算法

中值切分算法由 Heckbert【Heckbert82】首先提出。这个算法把色块分割为 K 个子块，每个子块包含大致相同的颜色数。图 11.27 显示了二维情况下的基本步骤。(假设颜色只有红色和绿色成分。)在图 11.27(a)中，原始颜色立方体被首先沿最长的一维细分：找到中值 r_1 使两个子块分别包含 $N/2$ 种颜色。接着，类似地处理两个子块——在最长一维的中值处切割为两部分。持续这个过程，直到分割成 K 个子块。每个子块的代表色取为其中心。图 11.27(b)显示了一个经过九次切分的例子。

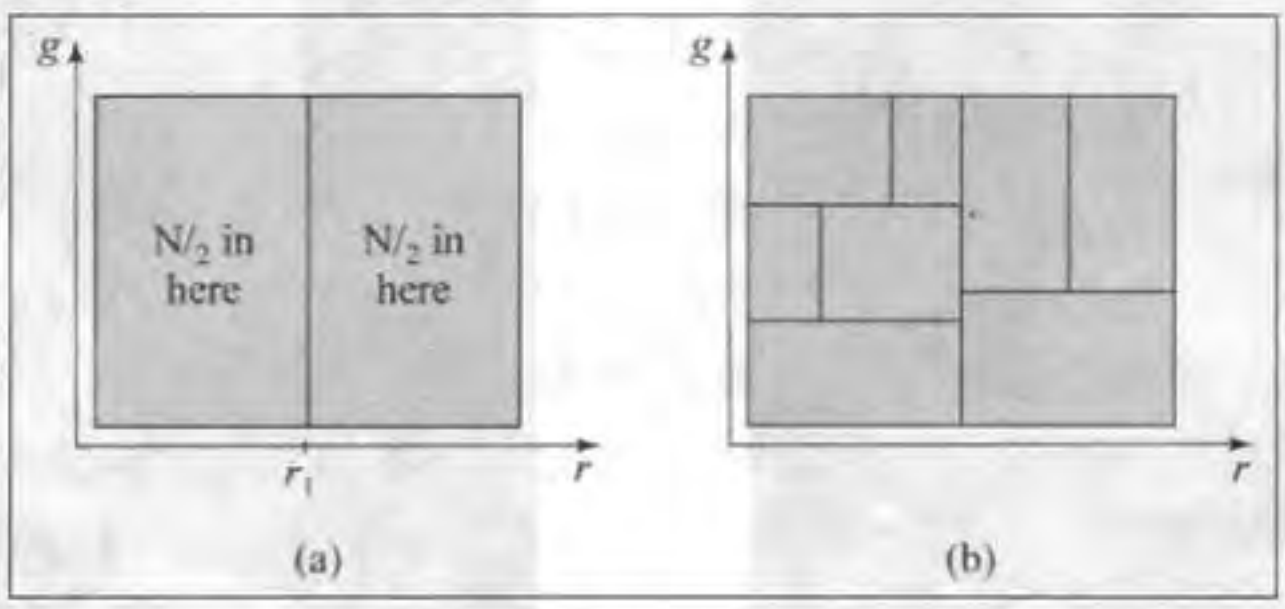


图 11.27 中值切割过程(对于二维颜色)

进行切分过程的一个简单方法是使用子块队列。定义合适的数据结构，存储色块维数及指向其包含的颜色数组的指针。每次出列一个子块，在最长维的中值处把它切分成两个子块，再把两个子块都加入队列等待下次处理。

当我们得到 K 个子块后，将其分别用 $0,1,\cdots,K-1$ 标记，然后重新读取整个图像，检查每个颜色位于哪个子块。Heckbert【Heckbert82】提出了高效算法实现此过程。

图 11.28 展示了图 11.25 经过中值切割算法处理，并把颜色数减少至 256 时的结果。仍然可以看到一些单色带。

图 11.28 用中值切割算法量化至 256 色后的图像

11.6.4　八叉树量化

Gervautz【Gervautz90】提出了一种基于八叉树(一种树形数据结构,其中每个节点最多有 8 个子节点)的方法,用来有效地将图像颜色数减少至 K 种。事实上此方法读取彩色图像,并建立一个颜色八叉树,准确记录颜色直至 K 种不同颜色被加入。然后,每读入一种新颜色,就将其加入八叉树中,但是紧接着又"简化"八叉树使它再次只包含最多 K 种不同颜色。简化过程强制一些颜色被合并到一起(变为它们的代表色),并用相应的近似颜色代替被合并的颜色。当图像中 N 种颜色全部读入后,八叉树包含 K(或略少)种代表色。然后再次遍历此八叉树,并赋给每个代表色一个索引值。

接着再次遍历图像,每遇到一种颜色,就把它插入八叉树中并很快的找到位置,或者正好位于树的底部(被叶节点精确描述),或者位于其最佳替代色所在的某些中间节点。然后返回此节点索引,说明它应该替换为哪种颜色。

如何利用八叉树结构使上述过程有效执行,并且不占用过多内存,这是个很有意思的问题。图 11.29 展示了八叉树的一部分:每个节点包含加入此树的颜色的一些相关信息,及 8 个指向其子节点的指针。

当一个颜色被插入八叉树时,其存在性通过"增加"一条直到第 8 级的路径来记录其存在。图 11.30展示了一个例子,表中显示了红、绿和蓝色值(二进制)。检查(r,g,b)三元数的每一位,分别从 R、G 和 B 的最高位开始,这里最高位分别为 101,它代表二进制的 5,所以路径指向 5 号子节点。若 5 号子节点尚不存在,则创建一个新节点并连接至此位置。继续此过程,处理第二高位三元数(110),路径指向 3 号子节点。经过 5 次迭代之后,检查颜色的最低位(110),路径指向 6 号子节点。如果此处有一叶节点(因为之前图像中已出现过此颜色),则将该颜色的计数(节点的一个域)增一。否则创建一个颜色计数为 1 的叶节点。注意到八叉树的深度不超过 8,插入一个颜色的过程是很快的:只需要不超过 8 次的子节点索引计算。

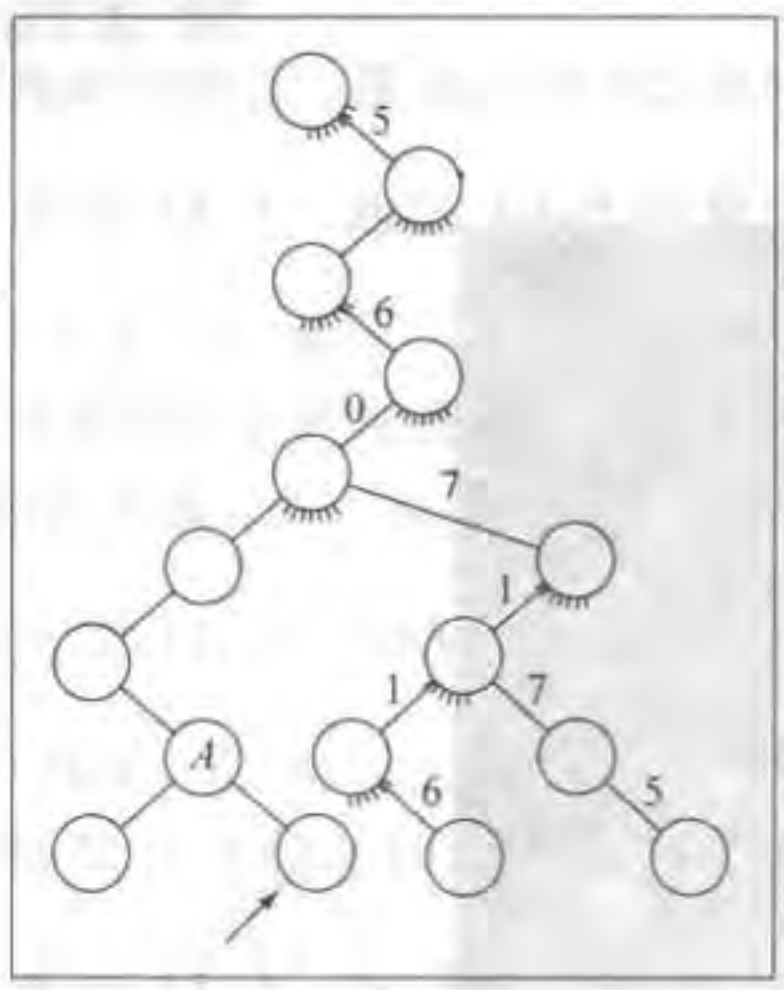


图 11.29　一个包含 4 种颜色的八叉树

R=	1	0	1	0	1	0	0	1
G=	0	1	1	0	1	0	0	1
B=	1	1	0	0	1	1	1	0
child:	5	3	6	0	7	1	1	6

图 11.30　从根节点到颜色叶节点的路径示例

每个深度为 8 的节点精确代表一种颜色(精度为 8 位),而中间节点代表颜色子块。例如,一个 3 级节点表示所有 R、G、B 分量的高三位相同,而其余位为任意值的颜色。

当八叉树包含颜色数超过 K 时,树就要被简化。从尽可能低的深度寻找一个"可简化"节点(至少有 2 个分支)。(图 11.29 中 A 节点就是一个可简化节点。)为了简化八叉树,先删除可简化节点的子节点(释放空间用于存储新节点),并将该简化节点标记为叶节点。此节点已经包含其子节点中出现颜色的汇总信息。每次八叉树简化过程减少的颜色种数比删除的子节点数小 1(为什么?)。现在,新的颜色又可以被插入八叉树中了。

如案例 11.7 中所详细描述的,每个节点要跟踪一些信息的变化,包括:

```
long numTimes;                    //这个子块中出现的颜色个数
long rSum, gSum, bSum;            //颜色值的累加和
int isLeaf;                       //1,如果此节点为叶节点
```

每个颜色(R,G,B)被插入八叉树时,更新其路径经过的所有节点的信息:

```
numTimes++;                       //这个子块又多了一种颜色
rSum+=R;                          //红色值总和
gSum+=G;
bSum+=B;
```

因此当简化一个节点时(其子节点被删除),它已经包含关于其子树可能发生任何变化的所有信息。稍后扫描八叉树确定每个叶节点的索引时,叶节点的代表色取为平均颜色:

$$代表色 = \left(\frac{rSum}{numTimes}, \frac{gSum}{numTimes}, \frac{bSum}{numTimes}\right)$$

最后,当再次从图像中读入 N 个颜色时,跟踪每个颜色的路径直至找到叶节点,检索得索引值。由于每条路径不超过 8 个节点,故此过程是很快的。

Gervautz 和 Purgathofer 关于颜色量化提出了一些建议,并估计了每种方法需要的计算资源。他们认为用八叉树方法得到的结果在质量上可以和中值切割方法相媲美,而且前者的处理时间和内存占用要低得多。

图 11.31 是图 11.25 经过八叉树量化,颜色数减少至 256 时的结果。仍然可以看到一些单色带,但是再现的颜色明显要比图 11.26 好得多。

图 11.31 用八叉树量化至 256 色后的图像

本章小结

人眼——大脑系统感知颜色的过程是很复杂的。一种光的颜色可以通过描述其光谱密度函数来精确指定，但是这种方法用于交流颜色是很不方便的。另外，人眼看起来一样的颜色，它们的光谱密度可以是不同的。事实上，我们对颜色的感知是三维的，所以我们需要一种方法来用三元数描述颜色。

CIE 标准提供了一个精确描述颜色的方法。任何具有单位亮度的颜色 S，可由 2 个数值(x,y)确定。这个表示法是由于如标准中定义，可以通过对适量的 X、Y 和 Z 进行相加来产生颜色 S。确切的说，$S=xX+yY+(1-x-y)Z$。其中特殊的超饱和三原色实际上是人眼不可见的，选择它们是为了使任何纯光谱色（单一波长光）都可以用正系数 x 和 y 来组合生成。CIE 色度图提供了一个描述颜色和计算给定颜色的组成成分的世界通用标准。CIE 色度图也用于显示色域，色域可以通过列举三原色的任意正线性组合得到。

有些算法可以把一个颜色系统描述的颜色转换至其他颜色系统，所以我们可以在 CIE 坐标和更熟悉的 RGB 坐标之间互相转换。其他颜色空间，如色调、饱和度和亮度，也经常被使用。

计算一幅彩色图像时（例如根据有光照的三维场景），只有绘制完成后，才知道要绘制的颜色范围，所以边绘制边显示图像是很困难的。一种较好的方法是，绘制时把所有颜色三元数存档至一个文件，然后扫描此文件确定颜色的总体分布。一旦知道颜色的总体分布，就可以在颜色三元数和显示设备可以处理的数值之间建立合适的对应。这其中可能包括要选择适当的颜色并装入 LUT。由于有些设备能显示的颜色数很有限，所以经常需要减少图像中的颜色数。人们提出了几种不同方法用于颜色量化，这些方法在程序实现的复杂度、计算效率、最后产生图像的质量等各方面都有不同改进。

案例分析

案例 11.1　绘制 RGB 颜色空间

（难度：Ⅱ）编写一个程序显示图 11.16 中 RGB 颜色立方体的一个“切片”。用适当的颜色显示切片上所有点。要求用户可以用键盘确定切平面位置（即确定平面 $aR+bG+cB=d$的系数 a、b、c 和 d）。

案例 11.2　HSV-RGB 转换

（难度：Ⅱ）编写一个带 GLUI 菜单的程序，包含第 2 章中的微调输入旋钮，允许用户用鼠标选择 H、S 和 V 的值，然后程序显示特别着色的彩色长方形中相应的 RGB 代表色。

案例 11.3　均匀颜色量化

（难度：Ⅲ）编写一个程序试验均匀颜色量化算法。从文件中读取一幅图像，然后在屏

幕上并排显示图像的 2 个版本：左边为全彩色版本，右边为量化版本。用户确定不同的红、绿和蓝色种数(例如分别为 8、7 和 4 种)，然后程序计算量化后的颜色并重新计算要显示的图像。

案例 11.4 流行度颜色量化

(难度：Ⅲ)编写一个程序试验流行度颜色量化算法。从文件中读取一幅真彩图像，然后用户确定允许使用的不同颜色种数 K。接着程序扫描图像，将 RGB 三元数量化为 15 位，计算颜色直方图，将数组排序，并得到 K 个最流行颜色。最后再次扫描图像，用 K 个代表色中最匹配的一个显示每个像素。屏幕上并排显示两个版本的图像：左边为全彩色版本，右边为量化版本。

案例 11.5 中值切割颜色量化

(难度：Ⅲ)编写一个程序试验中值切割颜色量化算法。代码可从网上或者【Lindley92】等书中获取。从文件中读取一幅真彩图像，然后用户确定允许使用的不同颜色种数 K。接着程序扫描图像，计算由中值切割算法确定的 K 个代表色。最后再次扫描图像，用 K 个代表色中最匹配的一个显示每个像素。屏幕上并排显示两个版本的图像：左边为全彩色版本，右边为量化版本。

案例 11.6 八叉树颜色量化

(难度：Ⅲ)编写一个程序试验八叉树颜色量化算法。可以自己实现算法，也可以像前一实例一样获取代码。从文件中读取一幅真彩图像，然后用户确定允许使用的不同颜色种数 K。接着程序扫描图像，建立一个有 K 个代表色的八叉树，并遍历此树给每个叶节点一个索引值。最后再次扫描图像，找到每个像素在八叉树中的索引，并用相应颜色显示。屏幕上并排显示两个版本的图像：左边为全彩色版本，右边为量化版本。

一个有用的八叉树节点数据结构如下：

```
class Node{
    public:
    int isLeaf;                          //值为 1 若此节点为叶节点
    int whichLevel;                      //此节点的深度
    int index;                           //LUT 索引，在 Pass 1 中确定
    int numChildren;                     //此节点的子节点数
    long numTimes;                       //这个子块中出现的颜色个数
    long Rsum, Gsum, BSum;               //颜色值的累加和
    Node * nextCand;                     //在此深度进行简化的候选节点
    Node * child[8];                     //此节点的 8 个子节点
};
```

一些主要函数及主要流程如下：

```
insertTree(Node& node, BYTE rgb[])
{ //将 24 位颜色 rgb 插入子树
    if( node==NULL )
```

```
    makeNewNode(node);                    //构造并初始化一个新节点
if( node.isLeaf )
{
    node.NumTimes++;                      //代表的颜色数增一
    addColors(node, rgb);                 //颜色值累加
}
else
{
    childIndex=4*rgb[0]+2*rgb[1]+rgb[2];
        insertTree(childIndex, rgb);
    }
}
```

如何寻找可简化节点？ 维护一个数组 reducible[]，它有 7 个栈，每个栈对应树的一层。reducible[i]包含节点指针，指向树中第 i 层可简化节点(其包含子节点数超过 1)。每次向八叉树中插入一种颜色时，沿一条特定路径到达一个叶节点，可能还会建立新的节点。在经过的每个节点处(除了最后的叶节点)，将子节点数增一并测试此数值，如果恰为 2，则此节点变为可简化节点，一个指向它的指针被压入相应的栈中。

当八叉树必须被简化时，从 $i=7$ 开始减小 i 值逐个测试这些栈：找到第一个非空栈 reducible[i]，栈头指向一个可简化节点，将此节点从栈中弹出并简化。

```
void reduceTree(void)
{
    thisNode=findReducibleNode();
    thisNode.isLeaf=1;
    numLeaves-=(thisNode.numChildren-1);
    freeChildren(thisNode);
}
```

进一步阅读

Feynman【Feynman63】关于颜色的章节写的很精彩，是介绍颜色理论的经典教材。Bill Meyer【Meyer81】全面讨论了颜色技术。Rogers【Rogers90】也完整地介绍过颜色理论及其在计算机图形中的管理。Hall 的“Illumination and Color in Computer Generated Imagery”【Hall88】更详细地介绍了颜色的产生和转换。

第12章 光线跟踪基础

本章学习目标

- 认识光线跟踪的基本概念
- 理解光线跟踪的数学和算法基础
- 建立并渲染包含球体、圆锥体、圆柱体、凸多面体以及其他实体的场景
- 创建包含光线折射和透明度处理的高真实感图像
- 建立处理 3D 实体纹理和位图的工具
- 对场景中物体的表面添加纹理，使之更加生动逼真
- 学习如何创建物体的 3D 实体纹理，如木质纹理和大理石纹理等，并学习如何光线跟踪这些物体
- 利用构造实体几何(CSG)方法创建更多几何形状的物体，并学习如何光线跟踪由这些物体构成的场景

预览

12.1 节介绍光线跟踪技术，12.2 节介绍在执行光线跟踪算法前需要设定的照相机的位置和场景的几何信息。12.3 节给出了光线跟踪的一个实例。12.4 节介绍了一些方便光线跟踪的基本几何形状，并引出了光线跟踪算法的核心技术，就是一条光线如何和物体求交。在该节中还讨论了如何将光线变换到物体的一般坐标系中。

12.5 节介绍了一些面向对象编程的思想和方法，并设计了一些类和程序来构成一个光线跟踪器，还讨论了不同类对象之间的相互作用关系。一个基本的类是 polymorphism，在该节中将讨论类 polymorphism 的设计不仅避免了大量代码的编写，而且使得程序更加可靠和灵活。该节实现了一个简单但完整的光线跟踪器。12.6 节考虑光线如何与扩展集合中的几何物体求交，如圆台和凸多面体等。

12.7 节利用第 8 章中提出的渲染模型建立了一个全彩色的光线跟踪器，该光线跟踪器能处理物体表面的环境光、漫反射光以及镜面反射光等。12.8 节讨论如何在物体表面绘制纹理，其中包括有图片生产的 2D 纹理，以及如木质纹理和大理石纹理等 3D 纹理。关于光线跟踪的反走样问题将在 12.9 节中处理。12.10 节讨论如何显著地加速光线跟踪算法。我们可以利用包围盒或其他包围区域技术避免大量的耗时的光线求交计算。

12.11 节描述如何生成二级光线来加强光线跟踪器的真实感。该节还描述了一个生成真实阴影的方法。12.12 节描述了具体生成二级光线的方法来模拟有光泽物体表面的反射光线，以及穿过透明物体的折射光线。

12.13 节扩充了可以光线跟踪的物体集合，并通过构造实体几何(CSG)方法来生成新形

状的物体；新的适合任意形状物体的光线跟踪方法也随之给出。在本章的结尾给出了许多案例分析，通过这些案例分析能测试光线跟踪器的效果，并引导我们对光线跟踪器进行改善。

12.1 前　言

在第 6～8 章中，我们介绍了由多边形网格构成的场景的渲染方法，包括一个描述（近似描述）了光线如何在多边形表面反射的渲染模型。另外，Gouraud 和 Phong 插值渲染方法防止了多边形网格模型表面的面片化现象。因此渲染得到的图像显示过渡光滑的表面（轮廓边界除外），即使模型的表面由离散面片组成。

光线跟踪提供了一个渲染场景的强劲方法。图 12.1 显示了光线跟踪的基本想法。设想屏幕缓存为由空间中的像素排列成的数组，人眼透过每个像素点观察场景。对于屏幕缓存上的每一个像素点，考虑一个问题：人眼透过这个像素点会观察到什么？我们可以设想场景中的 P 点将会发射一条光线穿过该像素点到达人眼，于是像素点的颜色就设为该光线的颜色。

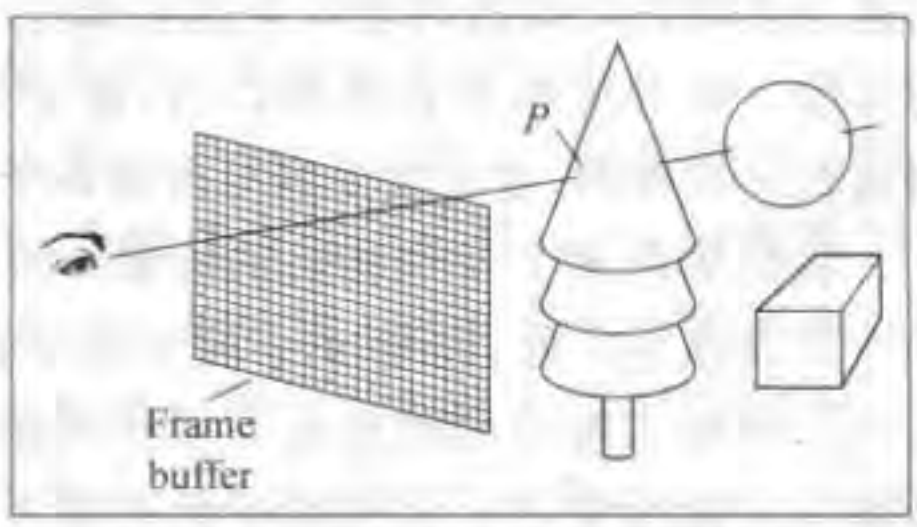


图 12.1　通过屏幕上一个像素点观察场景中的一点

实际上光线跟踪的过程和上面描述的过程有些区别，这一点常常困扰很多学生，我们将在本章中详细讨论。光线被设定为由眼睛发射出来，并穿过每个像素点的中心进入场景。光线进入场景后，检测场景中每个物体和光线的相交情况。如果与物体有相交，记录第一个和光线相交的物体，并记录相交的点。这里我们引进参数 t 来关联抽象的时间描述和光线的位置，这样一来，我们就可以说“光线现在在这里”或者“光线先到达这一点”。这里，我们应该看到光线跟踪的一个显著性质：它自动解决了隐藏面剔除问题，因为与光线相交的第一个曲面总是更接近人的眼睛，许多远处的曲面被自动忽略了。在给场景设定光源后，前面描述过的渲染模型将运用到第一个光线与场景相交的点，并计算该点处的环境光、漫反射光和镜面反射光。最后计算得到的颜色设为该像素点的颜色。

因为我们跟踪光线在场景中的传播路径，所以一些有趣的视觉效果很容易添加进来，例如阴影、光线的反射和折射效果等，渲染出用其他方法很难得到的高真实感的场景。光线跟踪的另一个优点是它可以处理更多类型的几何物体而不仅仅是多边形网格模型。几何实体可以由一些基本的几何形状构造得到，如球体、圆锥体和圆柱体等。这些基本几何形状都可以用数学公式精确表示出来，当然我们不要求每个形状都一定要写出它的数学表达式，但是如果需要的时候我们可以利用这些数学表达式。在把几何物体添加到场景中之前，我们也可以对它作用一个仿射变换来改变它的大小和方向，使得最后得到的场景更加复杂。在描述场景中的物体时，我们将会看到本章中提供的**场景描述语言**（Scene Description Language）是多么有用。

在本章中，我们将会描述生成复杂场景的高质量渲染图像的光线跟踪算法基础。我们采用渐进式的学习过程，因此在几个小节后我们就能了解到如何通过少量的编程得到一个

简单场景的渲染图像。其他复杂的算法将逐个介绍来建立更加高级的光线跟踪技术。为了激励读者更加深入地学习和实现本章中的算法,我们提供了一些精美的插图(如图 12.52)。从这些插图可以看出,光线跟踪技术可以渲染出极具真实感的图像。

12.2 建立光线跟踪的几何场景

为了跟踪穿过某个屏幕像素的光线的路径,我们需要一种光线的简单表示形式。参数表示法是目前已知的最易用的表示光线的方法。为了便于参考,我们将需要将第 6 章和第 7 章中涉及取景过程的内容集中到一起。需要事先说明的是,为了精确描述光线跟踪的几何信息我们需要设定大量的参数。大量的参数往往让人难以接受,所以我们建议读者在开始的时候最好能在纸上记下一些关键的参数。到对这些参数熟悉以后,你会发现很多参数都是相互关联的,这样就更容易理解各个参数的意义。

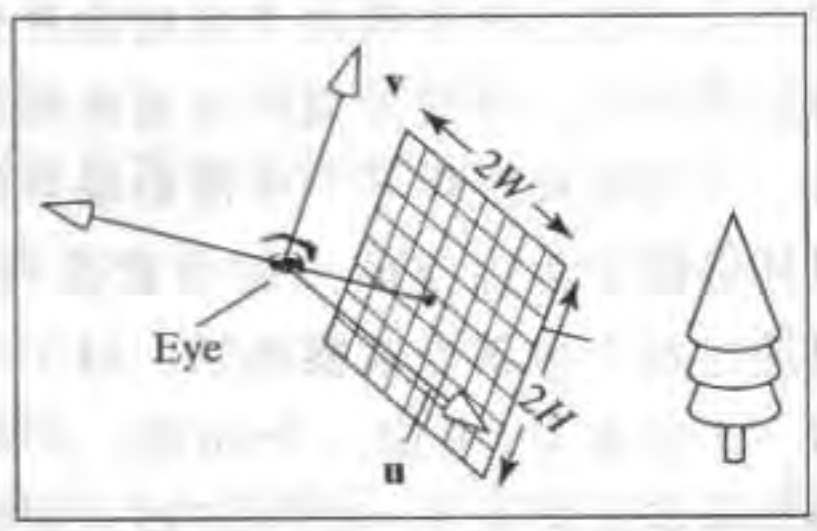


图 12.2 设定光线跟踪的相机位置

我们采用第 7 章中相机的设置:如图 12.2 所示,视点设在点 *eye* 处,向量 **u**、**v** 和 **n** 确定相机三个轴的方向。视域四棱锥的前截面与视点的距离为 N,投影屏幕与前截面重合。相机的形状和第 7 章中描述的相同:如图 12.3 所示,相机的视角为 θ,前截面上的投影屏幕的宽高比为 *aspect*。于是投影屏幕在 v 方向上的范围为 $-H$ 到 H,**u** 方向上的范围为 $-W$ 到 W,其中 H 和 W 由下面的等式给出:

$$H = N \times \tan(/2)$$
$$W = H \times aspect \tag{12.1}$$

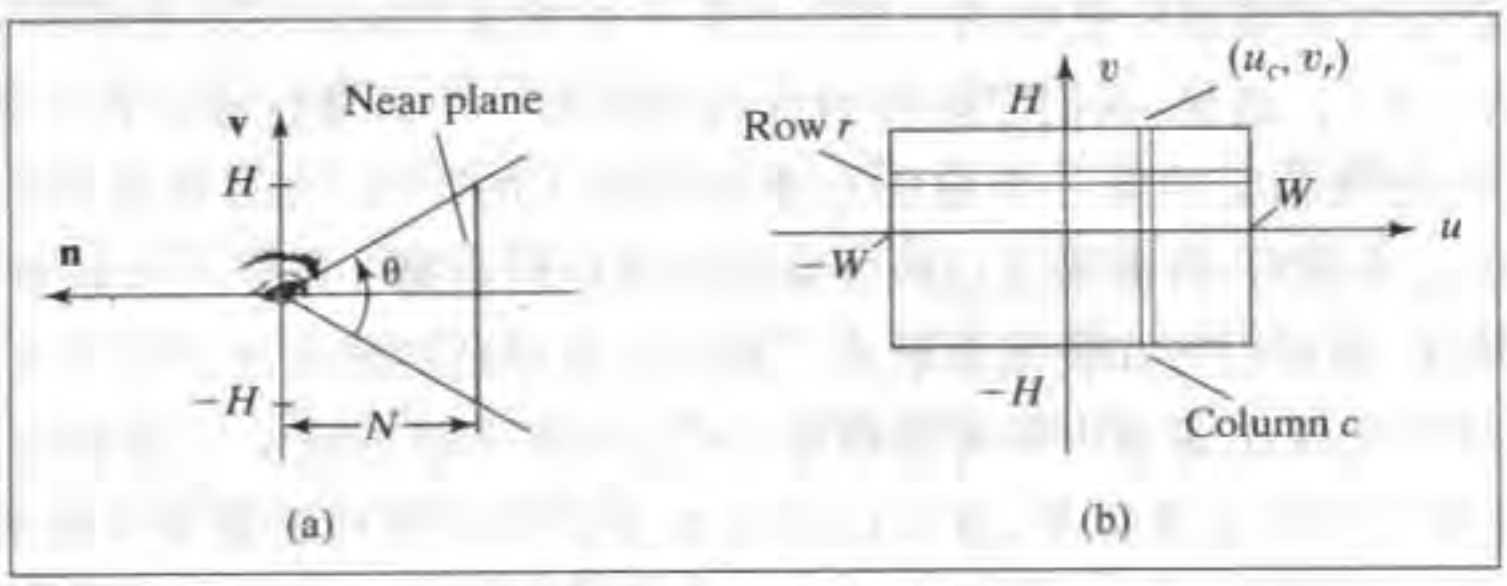


图 12.3 相机的形状

(a) 相机的形状;(b) 屏幕上的像素点

在第 7 章中我们曾提到视口变换决定了前截面上的点和视口上像素点之间的一一对应关系,这里我们继续采用这样的对应关系,但没有显式作用视口变换。事实上,我们可以简单地认为视口和前截面上的投影屏幕窗口重合,这样一来,人眼就通过视口上每个像素点来观察场景。假设视口上有 $nCols \times nRows$ 个像素,考虑第 r 行第 c 列上的像素,这里 r 的范围为 0 到 $nRows-1$,c 的范围为 0 到 $nCols-1$,以后我们称该像素为"第 rc 个像素"。一般情况下,我们按从下到上,从左到右的顺序遍历所有视口上像素,如图 12.3(b) 所示。有一

个问题是：每个像素点的坐标是多少？

如图 12.3 所示，设每个像素的左下角的坐标为(u_c, v_r)，则

$$u_c = -W + W\frac{2c}{nCols}, \quad 其中\ c = 0,1,\cdots,nCols-1$$

$$v_r = -H + H\frac{2r}{nRows}, \quad 其中\ r = 0,1,\cdots,nRows-1 \tag{12.2}$$

为了用数学公式表示一条光线在 3D 空间的位置，我们需要知道光线经过前截面上的像素的坐标。实际上只要知道该像素点离视点 *eye* 的距离就可以了。从 *eye* 点出发，计算在 **u**、**v** 和 **n** 方向上需要移动多少距离才能到达该像素的左下角点。读者们可能会觉得光线的路径应该经过每个像素的中心点，而不是左下角点，这样的想法当然是合理的，但是一般像素都非常小，这两种方法的结果没有明显区别。我们设在 **n** 的反方向上移动距离 N，在 **u** 方向上移动距离 u_c，在 **v** 方向上移动距离 v_r，则光线经过的像素左下角点的三维坐标由下式给出：

$$eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v} \quad \leftarrow 像素左下角点在三维空间中的坐标 \tag{12.3}$$

沿着这条光线发射方向上的每一点，我们可以关联一个时间参数 t。随着时间 t 的增加，光线上的点距离视点越来越远。我们设定 $t=0$ 时光线在视点的位置，$t=1$ 时光线到达像素的左下角点。光线跟踪中涉及到的基本计算就是当 t 在 0 到 1 之间变化时确定光线的位置，特别是要找出场景中有哪些物体和光线相交。光线沿着一条射线的方向匀速向前传播，我们要确定光线的传播路径是否和场景中的物体相交。如果用伪代码表示的话，光线传播路径的表达式可以写成：

$$r(t) = eye(1-t) + pixelcorner\ t \tag{12.4}$$

把 *pixelcorner* 的坐标代入上式，我们得到第 rc 光线参数表示形式：

$$r(t) = eye(1-t) + (eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v})t \tag{12.5}$$

稍作计算就会发现，该光线当 $t=0$ 时在视点位置，当 $t=1$ 时在第 rc 个像素的左下角点处。对于一条光线来说，它的起点和传播方向是我们比较关心的，所以对等式(12.5)作简单处理我们得到经过第 rc 个像素的光线如下式：

$$r(t) = eye + \mathbf{dir}_{rc}t$$

其中

$$\mathbf{dir}_{rc} = -N\mathbf{n} + W\left(\frac{2c}{nCols} - 1\right)\mathbf{u} + H\left(\frac{2r}{nRows} - 1\right)\mathbf{v} \tag{12.6}$$

在本章中，我们将会用上面的等式表示一条光线，而且在用程序实现光线跟踪算法是会直接利用到上述等式，所以读者有必要对光线的表达式深刻理解。

用上述等式表示的这簇光线有一个重要的性质，那就是当 t 不断增加时光线的位置离视点越来越远。你可以计算一下向量 $r(t)-eye$ 的长度来验证这条性质。如果光线分别在 t_a 和 t_b 时刻与两个物体相交，那么离视点近的物体与光线相交时对应的 t 值比较小。因此，在跟踪每条光线并检查它与场景中物体的相交情况时，我们只要记录光线第一次和物体相交的时间就可以了，光线通过的像素也最终根据这个记录来赋予颜色值。这样一来，我们自动解决了图形学中多年以来的一个重要问题，那就是“隐藏面剔除问题”，远处的曲面被自动忽略了。

练习

练习 12.2.1 补充算法细节

详细推导等式(12.1)～等式(12.6),证明第 rc 条光线的传播方向确实如文中所述。计算当 $t=2$ 和 $t=-1$ 时光线的位置。

练习 12.2.2 关于光线的数值计算

假设相机的设置是 $eye=(0,0,-1)$,$\mathbf{u}=(1,0,0)$,$\mathbf{v}=(0,1,0)$和 $\mathbf{n}=(0,0,-2)$,视角 θ 为 30°,前截面的宽高比为 1.5。又设 $nRows=480$,$nCols=640$,请给出第 rc 条光线的参数表达式,这里 $r=100$,$c=200$。

练习 12.2.3 像素点的位置?

请写出第 rc 个像素的中心点的坐标等式,而不是像素的左下角点坐标。

练习 12.2.4 关于光线的渐进计算

由等式(12.6)可知,如果按扫描线的顺序遍历所有光线,那么当前光线的方向可以由前一条光线的方向经过一个向量加法得到:请将 $\mathbf{dir}_{r,c+1}$ 写成关于 $\mathbf{dir}_{rc}$ 的形式。

12.3 光线跟踪过程纵览

我们首先给出一个光线跟踪器的基本框架,描述其中涉及到的一些基本计算。然后,我们介绍如何实现光线跟踪算法中的关键部分。

图 12.4 给出了实现光线跟踪器的基本步骤。需要光线跟踪的场景一般由一些几何物体与光源构成,比如场景中会包含球体、圆锥体、长方体和圆柱体等,每个几何物体都有确定的形状、大小和位置信息等。我们用一定的形式描述这些几何物体,并把它们存放到一个物体列表中。我们还需要如上一节所述对场景设定相机参数,然后对每一个像素都建立一条

```
<定义场景中的物体和光源>
<设定相机位置>
for(int r=0; r<nRows; r++)
  for(int c=0; c<nCols; c++)
  {
   <1. 建立第 rc 条光线>
   <2. 求出场景中所有与第 rc 条光线相交的物体>
   <3. 找出离视点最近的与光线相交的物体>
   <4. 计算光线与物体的交点坐标,并得到物体表面在该点的法向>
   <5. 计算在交点处人眼看到的光线的颜色>
   <6. 第 rc 个像素被赋予上述颜色>
  }
```

图 12.4 光线跟踪器构架的伪代码

从视点出发穿过该像素左下角点的光线，这里涉及到计算第 rc 条光线的发射方向 $\mathbf{dir}_{rc}$。整个光线跟踪的过程已经在图 12.4 中用伪代码的形式给出。

图 12.4 中的步骤 3～5 是之前没有提到过的，我们将在以后几节中详细介绍。我们先检测第 rc 条光线是否与物体列表中的物体相交，如果有相交，则记录光线 $r(t)$ 到达物体表面时的时间 t。当物体列表中的所有物体都已经被检测过，那么与光线相交时对应的 t 最小的物体离视点最近。计算该物体与光线的交点坐标，并计算物体表面在交点处的法向量。于是，在交点处人眼看到的光线的颜色就存储为像素的颜色。

图 12.5 显示了由一些圆柱体、球体以及三个圆锥体组成的简单场景，其中雪人主要由球体组成(观察雪人的帽子是由那些基本几何形状组成的?)，该场景中也包含了两个光源。注意到场景中的物体是可以相互贯通的，当把场景渲染成图像时，我们只关心最外层的曲面。

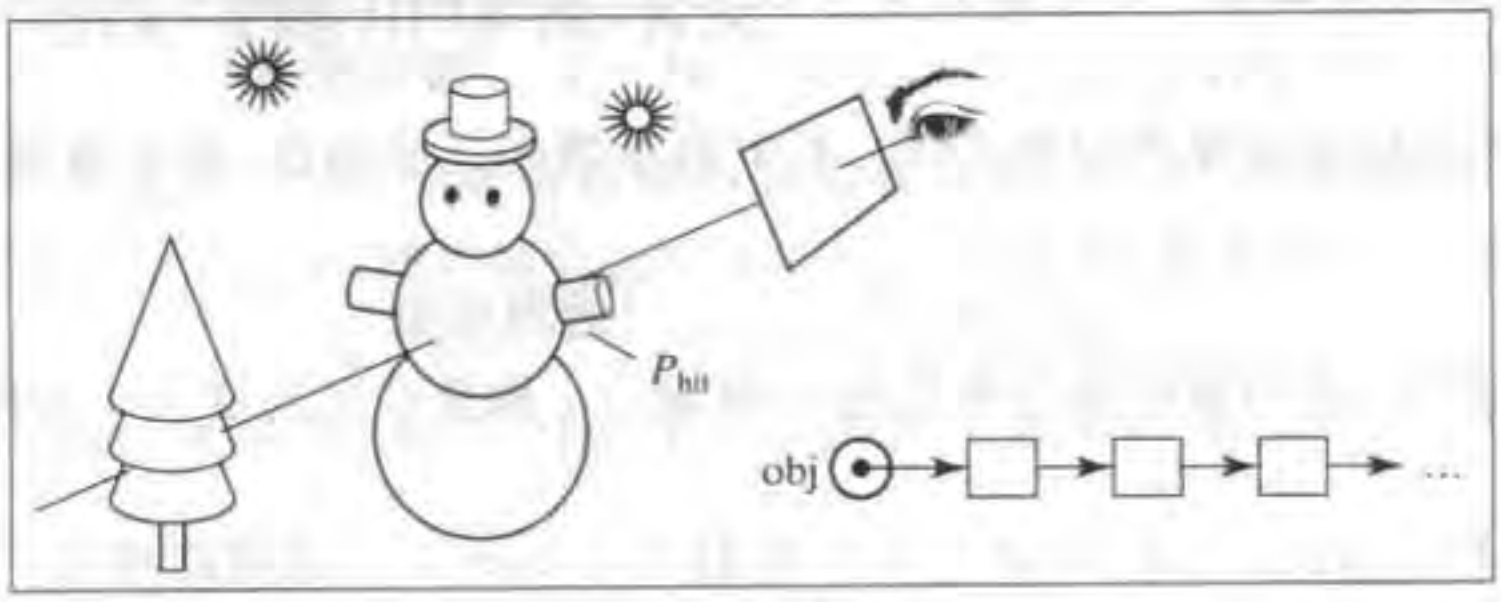


图 12.5　光线跟踪一个场景

场景中所有物体的描述都被保存在一个物体列表中，如图 12.5 所示。图中所画的光线与一个球体、一个圆柱体以及两个圆锥体相交，与其他的物体都不相交。在这个例子中，容易发现圆柱体是最先与光线相交的物体。设交点为 P_{hit}，则计算 P_{hit} 点的坐标只要把光线到达 P_{hit} 点时的时间 t_{hit} 代入光线的表达式(12.6)就可以了：

$$P_{hit} = eye + \mathbf{dir}_{r,c} t_{hit} \quad \langle 交点 \rangle \tag{12.7}$$

(请务必理解上述等式，因为它会在本章中多次出现。)对于某些像素，穿过它的光线可能与场景中的任何物体都不相交，那么这些像素的颜色就设为场景的背景颜色。

经过上面的描述，我们容易知道光线跟踪中的基本计算就是寻找光线与给定物体的交点。在本章中，我们将用一定的篇幅来介绍求交算法，并讨论如何使算法更加快速有效。

12.4　光线与物体求交

回顾一下第 5 章，我们曾建立了一个场景类 Scene，它能读入一个用 SDL 语言编写的文件并建立场景中的物体列表。在光线跟踪时，我们仍然用这个类来建立场景：

```
Scene scn;                    //实例化一个场景对象
Scn.read("myScene.dat");      //读入一个 SDL 场景文件
```

场景中的物体被创建后都被存放在物体列表中，每一个物体都是通用形状的一个实例，

并附加一个仿射变换决定物体的大小、方向以及在场景中的位置等。这里的通用形状指球体、圆锥体等,(后面我们将讨论关于每类形状的通用的精确意义)。在第 6 章中,我们介绍了一些通用形状,图 12.6 显示了我们将光线跟踪的通用形状,如球体、圆锥体、圆锥体、正方体、正方形以及四面体等。

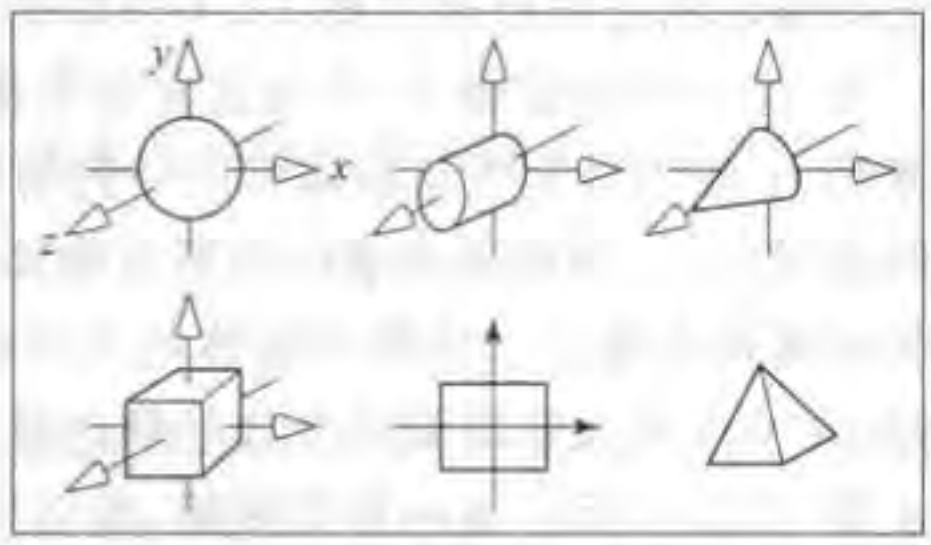


图 12.6 一些光线跟踪的基本通用形状

在第 5 章中我们利用 OpenGL 绘制物体(对每一类形状我们编写了 drawOpenGL()绘制方法),在本章中我们将利用光线跟踪的方法绘制物体。这样就涉及到光线与场景中物体的求交问题。如果我们对每个形状采用第 6.5.1 节中介绍的隐式表示方法,那么会使求交计算简单很多。比如,**通用球面**是一个半径为 1 中心在原点的球面,它的隐式表达式为:

$$F(x,y,z) = x^2 + y^2 + z^2 - 1 \quad \text{〈通用球面〉} \tag{12.8}$$

如果我们采用更为简单的记号 F(P),其中的参数为三维的点,那么通用球面的隐式表达式可以写成:

$$F(P) = |P|^2 - 1 \quad \text{〈通用球面〉} \tag{12.9}$$

如图 12.6 所示,通用圆柱面是半径为 1,纵轴与 z 轴重合且长度为 1 的圆柱面,它的隐式表达式为:

$$F(x,y,z) = x^2 + y^2 - 1 = 0, \text{且 } 0 < z < 1 \quad \text{〈通用圆柱面〉} \tag{12.10}$$

在实际场景中,每个通用形状都经过一个仿射变换得到一个新的形状,这时它的隐式表达式与通用形状的隐式表达式有很大的不同。但是,很快我们就会发现在计算光线和物体的交点时,只要求解光线和通用形状的交点就可以了。因此,通用形状的隐式表达式在光线跟踪算法中显得十分重要。

那么,我们如何求解一条光线和一个隐式表达式为 F(P)通用形状的交点呢?假设光线的起点为 S,传播方向为 **c**。采用这样的符号比之前我们采用的符号起点 *eye* 和方向 $\mathbf{dir}_e$ 更为简洁,于是光线的表达式由下式给出:

$$r(t) = S + \mathbf{c}t \tag{12.11}$$

我们容易知道:物体表面上所有的点满足 $F(P)=0$;且当光线上某一点 $r(t)$ 落在物体表面时,光线和物体相交。于是,点 $r(t)$ 在物体表面上的充要条件是 $F(r(t))=0$。假设在时间 t_{hit} 时,光线和物体相交,那么 t_{hit} 可以通过求解下面的等式得到:

$$F(S + \mathbf{c}t_{hit}) = 0 \tag{12.12}$$

光线跟踪算法的大部分计算都集中在如何高效地求解上述等式。下面我们将介绍当物体具有比较简单的表面形状时,如平面或球面等,上述等式的求解方法。

12.4.1 光线与通用平面求交

在场景中经常需要用一张平面来表示房间的地板和墙壁等,它们往往具有比较均一的颜色,或者被贴上一些纹理,如黑白棋盘纹理或扫描的图片等。

通用平面是指 xy—平面,或 $z=0$ 的平面,所以它的隐式表达式是 $F(x,y,z)=z$。当光线 $S+ct$ 与该平面相交时满足 $S_z+c_zt_h=0$,这是一个关于 t_h 的线性等式,它的解为:

$$t_h = -\frac{S_z}{c_z} \tag{12.13}$$

如果 $c_z=0$，光线和平面平行，此时光线和平面没有交点（除非 S_z 也为 0，这时光线嵌入到平面，也看不到任何反射光线）。否则，光线和平面的交点为 $P_{hit}=S-\mathbf{c}(S_z/c_z)$。

例 12.4.1　求解光线和平面的交点

请问光线 $r(t)=(4,1,3)+(-3,-5,-3)t$ 与通用平面的交点坐标是什么？

解：

利用等式(12.13)，求得 $t_h=-3/-3=1$；则交点坐标为 $S+\mathbf{c}=(1,-4,0)$。检验发现该点确实落在 $z=0$ 的平面上。

12.4.2　光线与通用球面求交

给定光线的表达式 $S+\mathbf{c}t$ 和通用球面的隐式表达式(12.9)，如何求解它们的交点呢？把 $S+\mathbf{c}t$ 代入等式 $F(P)=0$，我们得到 $|S+\mathbf{c}t|^2-1=0$，或者（参考等式(4.10)）[①]

$$|\mathbf{c}|^2t^2+2(S\cdot\mathbf{c})t+(|S|^2-1)=0 \tag{12.14}$$

这是关于 t 的二次等式：$At^2+2Bt+C=0$，其中 A、B 和 C 是

$$\begin{aligned} A &= |\mathbf{c}|^2 \\ B &= S\cdot\mathbf{c} \\ C &= |S|^2-1 \end{aligned} \tag{12.15}$$

我们用一元二次方程的求根公式求解上述等式：

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2-AC}}{A} \tag{12.16}$$

如果判别式 B^2-AC 的值为负，那么等式有实数解，光线和球面无交点。如果判别式的值为零，那么光线在球面上某一点擦过球面，和碰触到球面的时间为 $-B/A$。如果判别式的值为正，那么光线在两个时间点 t_1、t_2 和球面相交，分别对应等式(12.16)中的“+”和“-”（关于数值计算中的精度问题将会在练习中讨论）。

求解光线和平面或球面的交点问题相对比较简单，因为最终我们只要求解一个线性或者一元二次等式就可以了。在处理光线和其他基本形状求交问题时，往往导出的等式不易求解，我们将在练习中进一步讨论这个问题。

例 12.4.2　求解光线和通用球面的交点

请问光线 $r(t)=(3,2,3)+(-3,-2,-3)t$ 与通用球面的交点是什么？

解：

利用等式(12.15)得，$A=22$，$B=-22$ 和 $C=21$；代入等式(12.16)得，$t_1=0.7868$ 和 $t_2=1.2132$；于是两个交点分别是 $S+ct_1=(3,2,3)(1-0.7868)=(0.6393,0.4264,$

---

① 注意，为了表达简洁，我们并没有严格遵守符号的用法。S 是点而不是向量，所以严格来说我们不能用它来做点积。$|S|^2$ 是 $S_x^2+S_y^2+S_z^2$ 的简略表示方法。

0.6396)和 $S+ct_2=(3,2,3)(1-1.2132)=(-0.6393,-0.4264,-0.6396)$。容易验证发现这两个点确实落在通用球面上,与原点的距离为 1。读者还会发现两个交点原点对称,这是因为光线经过原点而且通用球面的中心落在原点上。

12.4.3 光线与仿射变换后的物体求交

在构造场景时,设计者会对每个物体施加一个仿射变换 T,来确定物体在场景中的大小、方向和位置。对物体施加仿射变换后,我们之前得到的等式和结果会产生什么样的变化?如果仿射变换后物体的隐式表达式变得非常复杂,那么等式(12.12)就很难去求解,这是我们不愿看到的。下面我们将证明,这种情况不会发生。

图 12.7 显示了变换 T 把通用球面 W' 映射为一个椭球面 W。光线 $S+\mathbf{c}t$ 何时和 W 相交呢?假设我们已经求得变换后的物体 W 的隐式表达式 $G()$,那么只要求解等式 $G(S+\mathbf{c}t)=0$ 来求出碰撞时间。

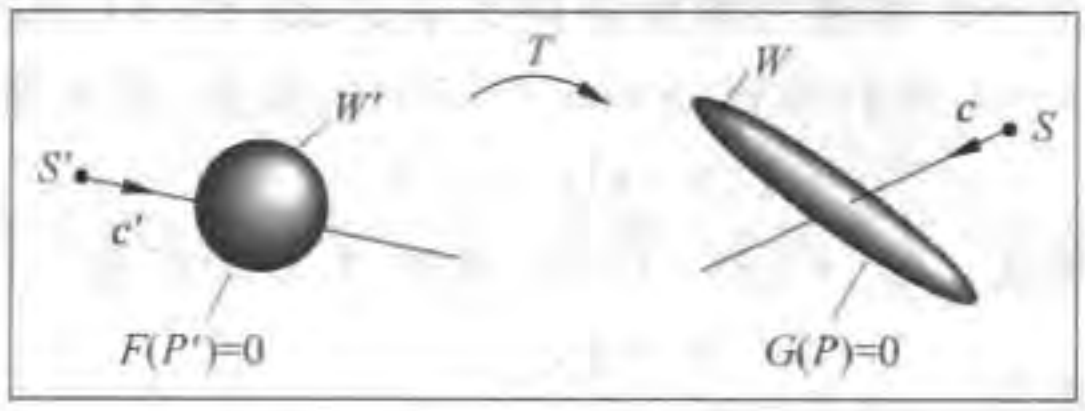


图 12.7 光线与一个椭球面相交

这样一来,问题就转化为求解变换后物体的隐式表达式。在第 6.5.3 节中我们看到,如果通用物体的隐式形式为 $F(P)$,那么变换后物体的隐式形式为 $F(T^{-1}(P))$,其中 T^{-1} 为 T 的逆变换(也就是说,如果 T 的变换矩阵为 M,那么 T^{-1} 的变换矩阵为 M^{-1})。于是,我们必须通过求解下面等式得到碰撞时间 t:

$$F(T^{-1}(S+\mathbf{c}t))=0 \tag{12.17}$$

这个等式看起来有点凌乱,但实际上它的几何意义非常简单:求解逆变换光线 $T^{-1}(S+\mathbf{c}t)$ 与原来的通用物体的碰撞时间!因为仿射变换 T 是线性的,所以逆变换光线为:

$$T^{-1}(S+\mathbf{c}t)=(T^{-1}S)+(T^{-1}\mathbf{c})t$$

在图 12.7 中,我们用 $S'+\mathbf{c}'t$ 表示逆变换光线(明显,逆变换光线还是一条光线,它的起点为 S',传播方向为 $\mathbf{c}'$)。假设仿射变换 T 的矩阵为 M,采用齐次坐标表示,我们得到逆变换光线的参数表达式:

$$\tilde{r}(t)=M^{-1}\begin{bmatrix}S_x\\S_y\\S_z\\1\end{bmatrix}+M^{-1}\begin{bmatrix}c_x\\c_y\\c_z\\0\end{bmatrix}t=\tilde{S}'+\tilde{c}'t \quad \text{〈逆变换光线〉} \tag{12.18}$$

这里 M^{-1} 在一个等式中与一个点相乘,在另一个等式中与一个向量相乘。点 S' 的坐标由 $\tilde{S}'$ 舍弃 1 得到;向量 c' 的坐标由 $\tilde{c}'$ 舍弃 0 得到。

因此,我么可以求解逆变换光线与原物体的交点,来代替求解光线与变换后物体的交点。场景物体列表中的每一个物体都关联了一个仿射变换,求解光线 $S+\mathbf{c}t$ 与变换后物体

的交点需要下面几个步骤：

(1) 求出逆变换光线 $S'+\mathbf{c}'t$。

(2) 求出逆变换光线与通用物体的碰撞时间 t_h。

(3) 把碰撞时间 t_h 代入等式 $S+\mathbf{c}t$ 得到实际的交点坐标。

这个方法的优点是我们只要设计代码实现光线与通用物体的求交算法就可以了。这是个双赢的结果：程序员只要编写更简单的代码；计算机只需更少的计算。我们在构造场景时可以随意对物体施加任何仿射变换来得到更为复杂有趣的场景，而不用担心在光线跟踪时需要编写更为复杂的代码。对物体的变换转化为对光线的变换，由此带来的好处主要是得益于光线是一条直线。

例 12.4.3　求解光线和变换后物体的交点

假设椭球面 W 是由通用球面变换得到，用 SDL 语言描述如下：

```
translate 2 4 9
scale 1 4 4
sphere
```

可知，球面先经过一个放缩然后平移变换为椭球面。它的仿射变换和逆变换矩阵分别为(请验证)：

$$M=\begin{bmatrix}1&0&0&2\\0&4&0&4\\0&0&4&9\\0&0&0&1\end{bmatrix},\quad M^{-1}=\begin{bmatrix}1&0&0&-2\\0&\frac{1}{4}&0&-1\\0&0&\frac{1}{4}&-\frac{9}{4}\\0&0&0&1\end{bmatrix}$$

请求出光线$(10,20,5)+(-8,-12,4)t$ 与 W 的交点。

解：

逆变换光线为$(8,4,-1)+(-8,-3,1)t$(请验证)。利用等式(12.15)得到$(A,B,C)=(74,-77,80)$。于是判别式的值为 9，光线和物体有两个交点。从等式(12.16)我们得到碰撞时间为 1.1621 和 0.9189。把比较小的碰撞时间代入光线的表达式就得到所求的交点坐标，$(10,20,5)+(-8,-12,4)0.9189=(2.649,8.97,8.67)$。你将如何验证该点确实落在变换后的物体上？

练习

练习 12.4.1　求交点

请求出中心在$(1,2,1)$，半径为 5 的球面和光线$(3,5,8)+(-4,-2,-6)t$ 的碰撞时间以及交点坐标。

练习 12.4.2　它和平面相交吗？

已知一个平面由 SDL 描述如下：translate 4 5 6 rotate 90 1 0 0 plane；请问光线$(10-t,8-2t,3+t)$于何时何处与该平面相交？

12.5 一个光线跟踪器的应用

有了前面介绍的光线与物体的求交算法,我们现在可以根据图 12.4 描述的框架构造一个光线跟踪器。我们将会利用场景类 Scene 和第 5 章中介绍的 SDL 语言,因为它们是描述复杂场景的有用工具。我们假定场景已经给定,场景中的物体保存在一个物体列表中,而且每一个物体都关联一个仿射变换。

一个光线跟踪器需要处理一些相互影响的对象,比如:对场景取景的照相机;场景的投影屏幕;从视点发射出来并穿过场景的光线;以及包含许多几何物体和光源的场景。在用面向对象的思想实现光线跟踪器时,我们要精心设计哪些类是需要的,每个类又要具有哪些功能,以及每个类要具有什么样的属性来支持它的功能。这里我们采用自上而下的设计方法,当然你也可以采用其他方法来解决这个问题。

在我们的设计方法中,我们把光线跟踪的任务分配给照相机,于是我们在已有的 Camera 类里添加一个方法:

```
Void Camera::raytrace(Scene& scn, int blockSize);
```

我们传递给照相机两个参数,一个需要光线跟踪的场景对象,另一个是 blockSize,它的意义我们下面会介绍。该方法产生从视点出发并经过每个像素左下角点进入场景的光线,并求出人眼在每条光线上看到的颜色值,然后把求得的颜色值赋给相应的像素。

我们将用 OpenGL 来实现像素的绘制,并在 raytrace()中设定模型视图和投影矩阵来直接在视图上绘制,具体的过程将在下面叙述。这样一来,程序主循环中 display()方法就显得比较简单:它只要清除屏幕并告诉相机类对象 cam 去光线跟踪场景就可以了,如下面代码所示:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);          //清除屏幕
    cam.raytrace(scn, blockSize);          //光线跟踪场景
}                                          //适当地设置像素的颜色
                                           //用正确的颜色填充该像素
```

读者在设计自己的光线跟踪器时,对上面代码的微小改变可能会带来很大的变化。在前面我们已经实现了用 OpenGL 函数来渲染 SDL 语言描述的场景,所以在用光线跟踪算法渲染场景前我们不妨给出场景的一个粗糙预览图。场景的预览图可以快速地被渲染出来,它能让用户知道相机的取景正确并且场景中的物体都在它们合适的位置上。无论是相机或者是场景的设置有问题,用户都可以进行调整,然后再次查看预览结果。如果所有的设置都正确,那么光线跟踪的渲染结果将和场景的预览结果吻合。因此,通过对比预览结果,用户可以很明显地发现光线跟踪器中的错误。

为了添加对场景的预览,可以把 display()方法简单扩展为:

```
void display(void)
```

```
{
    //清除屏幕并重置深度缓存
    glClear(G L_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    cam.drawOpenGL(scn);
      //绘制场景的预览结果
    cam.raytrace(scn, blockSize);
      //用光线跟踪的渲染结果覆盖预览结果
}
```

图 12.8　在预览场景上绘制光线跟踪结果

图 12.8 显示了光线跟踪的一个中间结果：首先绘制场景的一个粗糙的预览结果，在预览结果上再绘制光线跟踪的结果，在该图中只有场景的下半部分已经用光线跟踪绘制。这里需要强调的是，光线跟踪的结果和预览场景完全重合。

整合：raytrace()的构成

Camera 类中的 raytrace()方法要实现图 12.4 中描述的光线跟踪器的框架。对于每一个在第 r 行第 c 列上的像素，需要产生一条从视点出发并穿过第 rc 个像素的左下角点进入场景的光线。因此，我们需要构造一个光线类 Ray，它的开头部分代码如图 12.9 所示。Ray 类需要一个属性来保存光线的起点，以及另一个属性保存光线的发射方向，以后我们将会添加 Ray 类的其他属性。另外有两个方法用于设置光线的起点和发射方向。

```
class Ray{
public:
    Point3 start;
    Vector3 dir;
    void setStart(Point3& p){start.x=p.x; etc..}
    void setDir(Vector3& v){dir.x=v.x; etc..}
    //其他属性和方法
};
```

图 12.9　Ray 类的开头部分代码

图 12.10 描述了 raytrace()方法的框架：它设置光线的起点为视点，然后对应于每一个像素计算并设置第 *rc* 条光线的发射方向(根据等式(12.3))。Scene 类对象 scn 将计算沿着发射光线返回的颜色的值：crl=scn. shade(theRay)。方法 shade()做了核心的工作：它生成一条射入场景的光线，求出光线与场景中物体的交点，计算沿着发射光线返回的颜色值。在本章的后面部分，我们将具体实现方法 shade()的功能。

绘制用于预览的像素块

参数 blockSize 决定了在每一步绘制时像素块的大小。在实现光线跟踪器时，绘制像素块而不是每个像素点是一个简单的加速方法。此时得到的渲染图像比较粗糙但渲染速度很快：我们仅跟踪穿过像素块左下角点的光线；如果跟踪穿过屏幕上每个像素的光线的话，图

```
void Camera::raytrace(Scene& scn, int blockSize)
{
    Ray theRay;
    theRay.setStart(eye);
    //设置 OpenGL 显示简单的预览结果
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,nCols,0,nRows);          //整个投影屏幕设为视口
    glDisable(GL_LIGHTING);               //于是 glColor3f()能正确着色
    //开始光线跟踪
    for (int row=0; row<nRows; row+=blockSize)
    for (int col=0; col<nCols; col+=blockSize)
    {
        计算光线的发射方向
        theRay.setDir(<direction>);       //设置光线的发射方向
        Color3 clr=scn.shade(theRay);     //计算颜色值
        glColor3f(clr.red,clr.green.clr.blue);
        glRecti(col,row,col+blockSize,row+blockSize);
    }
}
```

图 12.10 raytrace()的框架

像将由逐个像素渲染得到,渲染速度比较慢。

图 12.11(a)显示了一个简单的例子来说明像素块的构成。屏幕上实际的像素个数是 16 行 32 列,这里设置 blockSize 的值为 4,于是每个块由 16 个像素组成。计算穿过该块的左下角点的光线的颜色值,那么块中所包含的 16 个像素都被设置为该光线的颜色值。此时得到的渲染图像是分辨率为 4×8 的光栅化结果,是光线跟踪的全分辨率渲染结果的一个粗糙近似,但是它的渲染速度很快。如果粗糙的渲染图像是正确的,那么用户可以设置 blockSize 的值为 1 来得到场景的全分辨率渲染结果。图 12.11(b)显示了设置不同的块的大小时一个简单场景渲染结果,对应的块大小分别为 4、2 和 1。第一种设置的渲染速度是最后一种设置的 16 倍。注意到当块的大小为 4 时比块的大小为 1 时得到渲染图像的效果粗糙很多,但是地球仪和地板上的纹理还是清晰可见的。

注意到 raytrace()中设置了 OpenGL 的模型视图矩阵和投影矩阵来正确绘制像素块。模型视图矩阵被设为单位阵,投影矩阵也只是简单地把窗口缩放为视口大小。这样设置使得 OpenGL 的绘制过程显得更加明晰,于是我们可以利用 glRecti()方法直接将一个正方形绘制到视口上。这里,我们假设在程序开始运行的时候,已经利用 glViewport(0,0,nCols,nRows)方法将视口设置为整个投影屏幕。OpenGL 的光照状态应该关闭,否则 glColor3f()不能得到正确的着色结果。

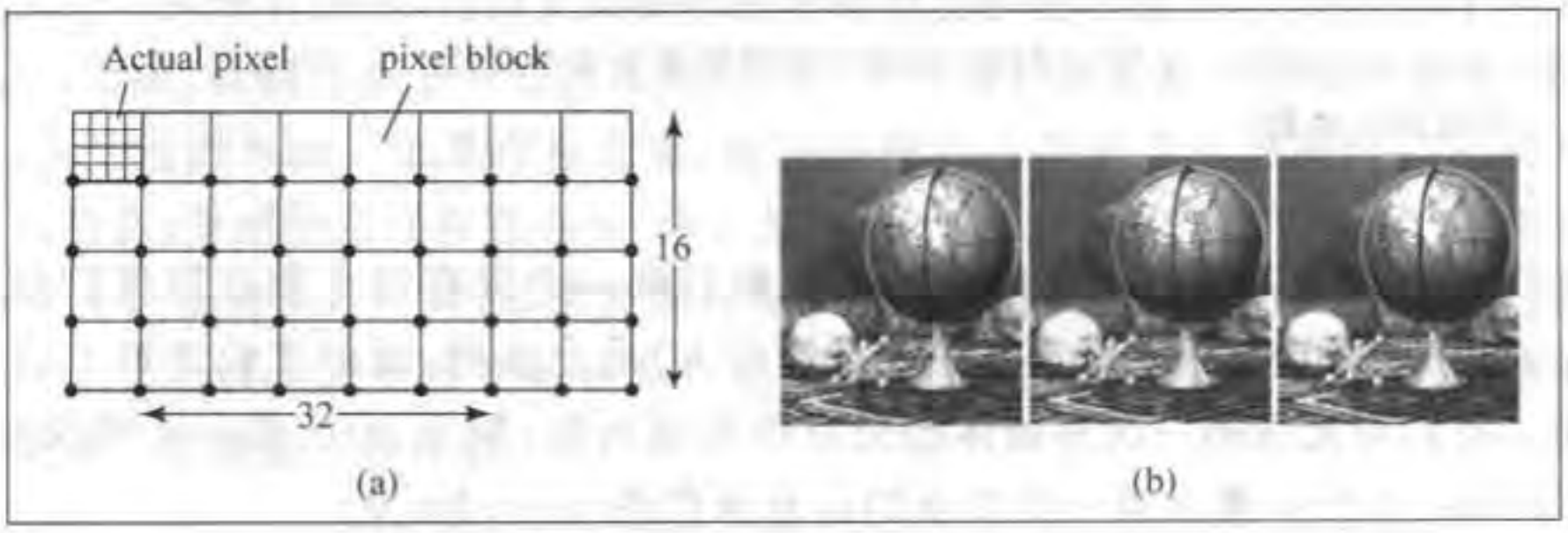


图　12.11

(a) 设置像素块的大小用于预览光线跟踪结果；(b) 块的大小分别为 4、2 和 1 时光线跟踪的结果

shade(ray)的基本构成

光线跟踪器的核心内容在 shade()方法内，图 12.12 中用伪代码描述了它的算法框架。它的第一部分工作是判断光线是否和物体相交，如果是，找出相交的物体。为了完成这个工作，它需要一个 getFirstHit(ray, best)，该方法将光线与物体第一次相交的数据保存到名字为 best 的相交记录中。我们将在下面更详细地介绍 getFirstHit()和相交记录的结构。一旦光线的相交信息已经得到，shade()将计算光线的颜色值。如果没有物体和光线相交，那么 shade()简单地返回背景的颜色。如果光线确实和一个物体相交，那么 shade()将根据第 8 章中介绍的经典渲染模型计算光线颜色的各个组成部分，其中包括物体发射出的光线(如果物体是发光体)，以及环境光线、漫反射光线和镜面反射光线。另外，shade()还考虑有光泽的物体表面的反射光线，以及穿过透明物体的折射光线。

```
Color3 Scene::shade(Ray& ray)
{       //返回该光线的颜色
    Color3 color;                          //返回的综合颜色
    Intersection best;                     //用于保存目前为止最佳的碰撞记录
    getFirstHit(ray, best);                //填充"best"记录
    if (best.numHits==0)
        return background;
    color.set(the emissive color of the object);
//添加更多的构成光线
    color.add(ambient, diffuse, and specular components);
    color.add(reflected and refracted components);
    return color;
}
```

图 12.12　shade()的算法框架

getFirstHit()寻找最早和光线相交的物体，并把相关信息记录在相交记录 best 中。我们用 Intersection 类来实现相交记录，它的结构由下面的伪代码描述：

```
class Intersection{
public:
```

```
    int numHits;        //有正的相交时间的交点的数目
    HitInfo hit[8];     //交点列表-以后可能需要多于 8 个交点
    …其他成员函数…
};
```

它有两个属性域-有正的相交时间的交点数目和一个保存每个交点信息的数组。我们认为有正的相交时间的交点(位于光线起点的前方)是合法的,而对于位于视点后面的交点不感兴趣。我们对光线第一次和物体的交点特别感兴趣,如果 inter 是一个相交记录,而且 inter. numHits 大于 0,那么第一个交点的信息保存在 inter. hit[0]。

为什么我们要保存光线与物体的所有交点信息,而不是仅仅保存第一个交点信息?这样做一个很大的好处是能渲染布尔物体(见第 12.12 节)。为了能处理布尔物体,我们必须保存光线与物体的所有交点信息,所以为了以后处理方便我们先要花精力保存所有需要的信息。一般来说,视点在所有物体的外部,而且一条光线和物体相交两次:一次相交时进入物体,另一次相交时从物体中出来。在这种情况下,inter. numHits 等于 2,inter. hit[0]描述光线进入物体的位置,inter. hit[0]描述光线从物体中出来的位置。但是有些物体,如图 12.13 中所示的圆环和布尔物体,与光线有多于两个交点。图 12.13(a)所示的物体与光线有四个交点,所以 inter. numHits 为 4,我们把交点信息保存在 inter. hit[0],……, inter. hit[3]。图 12.13(b)中的物体(假设是透明的)与光线有八个交点,但是视点在物体内部,只有第 6 次、第 7 次和第 8 次相交在正的时间发生(所以 inter. numHits 为 3)。

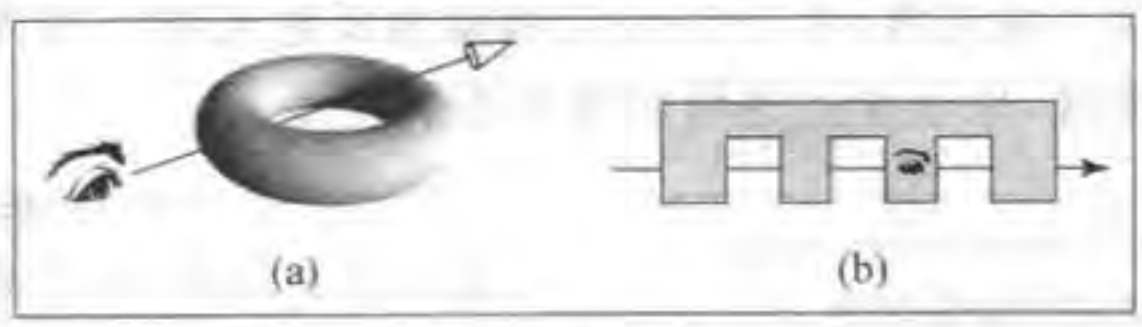


图 12.13 物体与光线有多个交点,有些相交在负的时间发生

每个交点的信息保存在相交记录类 HitInfo 中,它的结构如下所示:

```
class HitInfo{                          //每个交点的信息
    public:
    double hitTime;                     //相交时间
    GeomObj * hitObject;                //相交的物体
    bool isEntering;                    //光线是进入还是出来
    int surface;                        //相交于哪个表面
    Point3 hitPoint;                    //交点的坐标
    Vector3 hitNormal;                  //交点处曲面的法向
    …其他成员变量和方法…
};
```

一个相交记录包含交点的多个属性:相交时间,一个指向相交物体的指针,光线在交点处是进入物体还是从物体中出来,交点的坐标,以及物体表面在交点处的法向。所有这些属性在光线跟踪过程中都是非常重要的。

图 12.14 中用伪代码描述了 getFirstHit()的算法框架。它用指向物体的指针 pObj 遍

历了物体列表中的所有物体，检测光线是否和物体相交。为了检测相交情况，它调用了每个物体的 hit()方法，我们将在下一节的开头介绍这个方法。如果存在有效的交点，每个 hit()返回 true，否则返回 false。如果存在交点，该方法建立一个相交记录来保存光线与该物体的所有交点信息，并把信息保存在 inter 中。然后 getFirstHit()比较 inter 中的第一个交点时间和 best 中记录的交点时间，如果发现有更早的交点时间，那么把 inter 中的数据复制到 best 中。best.numHits 的值被初始化为 0，这样当 pObj 遍历物体列表时，第一个有效的相交记录的交点个数将被正确计数。

```
void Scene::getFirstHit(Ray& ray, Intersection& best)
{
    Intersection inter;                     //实例化相交记录类
    best.numHits=0;                         //还没有交点
    for(GeomObj * pObj=obj; pObj!=NULL; pObj=pObj->next)
    {    //检测场景中的每一个物体
        if(!pObj->hit(ray, inter))          //光线是否和 pObj 相交?
            continue;                       //无交点：检测下一个物体
        if(best.numHits==0 ||               //best 中还没有交点信息
            inter.hit[0].hitTime<best.hit[0].hitTime)
            best.set(inter);                //复制 inter 到 best
    }
}
```

图 12.14　getFirstHit()函数

注意到 getFirstHit()把复杂的光线与物体的求交计算交给了每个物体的 hit()方法。我们将为每种类型物体实现一个 hit()方法。为了提高效率，我们将充分利用每种通用物体的形状特征设计适当的 hit()方法。这是 C++ 语言**多态性**特征的一个典型应用，利用多态性可以简化代码，并且使程序更加稳定高效。多态性特征允许我们为每种物体编写 hit()方法时用同一个函数名“hit”。编译器不知道调用哪个 hit()，因为它甚至不知道物体列表中物体的类型是什么。实际上正确的 hit()将在程序运行时被系统调用，从而得到正确的求交结果。

物体类中的 hit()方法的主要任务是对给定的一条光线建立一个相交记录，并把光线和该物体的所有交点信息填充到相交记录中。下一节我们将实现球体的 hit()方法。

12.5.1　光线和球体的求交程序

图 12.15 显示了 Sphere 类的 hit()方法。它首先利用球体的逆变换将光线 r 变换到球体的通用坐标系，xfrmRay()方法根据等式(12.18)实现了这个变换(见练习)。然后计算等式(12.12)中的系数 A、B 和 C，以及判别式 B^2-AC。如果判别式的值为负，此时等式没有实数解，可知光线和物体没有交点。于是 hit()方法的返回值为 false，inter 变量也将不会被使用到。

```
bool Sphere::hit(Ray &r, Intersection& inter)
{
    Ray genRay;                          //用于生产泛型光线
    xfrmRay(genRay,invTransf,r);
    double A, B, C;
    A=dot3D(genRay.dir, genRay.dir);
    B=dot3D(genRay.start, genRay.dir);
    C=dot3D(genRay.start, genRay.start)-1.0;
    double discrim=B*B-A*C;
    if(discrim<0.0)                      //没有交点
        return false;
    int num=0;                           //目前的交点个数
    double discRoot=sqrt(discrim);
    double t1=(-B-discRoot)/A;           //第一次相交时间
    if(t1>0.00001)                       //交点是否在视点前方?
    {
        inter.hit[0].hitTime=t1;
        inter.hit[0].hitObject=this;
        inter.hit[0].isEntering=true;
        inter.hit[0].surface=0;
        Point3 P(rayPos(genRay, t1)); //交点坐标
        inter.hit[0].hitPoint.set(P);
        inter.hit[0].hitNormal.set(P);
        num=1;                           //一个有效交点
    }
    double t2=(-B+discRoot)/A;           //第二次相交时间
    if(t2>0.00001)
    {
        inter.hit[num].hitTime=t2;
        inter.hit[num].hitObject=this;
        inter.hit[num].isEntering=false;
        inter.hit[num].surface=0;
        Point3 P(rayPos(genRay, t2)); //交点坐标
        inter.hit[num].hitPoint.set(P);
        inter.hit[num].hitNormal.set(P);
        num++;                           //另一个有效交点
    }
    inter.numHits=num;
    return (num>0);                      //true 或者 false
}
```

图 12.15 Sphere 类的 hit()函数

如果判别式的值为正，由等式(12.16)得到两个交点时间。我们记值较小的交点时间为 t_1，值较大的交点时间为 t_2。如图 12.16 所示，这里有三种情况。当 t_1 和 t_2 的值都大于零时，球体在视点前方；当 t_1 的值为负 t_2 的值为正时，视点在球体内部；当 t_1 和 t_2 的值都为负时，球体在视点的后方。

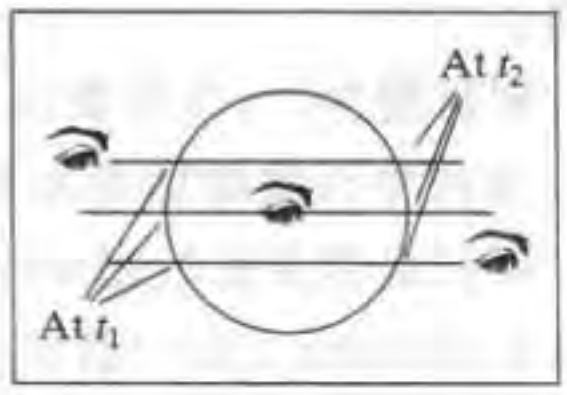


图 12.16　哪些交点在视点前方？

如果 t_1 的值严格大于 0，那么 t_1 就是有效的第一次相交时间，我们把交点信息保存在 inter.hit[0]中，并设变量 num 的值为 1，显示已经有一个交点存在。如果 t_2 的值也为正，那么第二个交点的信息保存在 inter.hit[num]中。(如果第一个交点时间为负，第二个交点时间可能为正也可能为负，相应的交点信息被自动保存在 inter.hit[0]中。)

在保存交点信息时，我们利用了球体的特征。比如说，我们知道球体是凸物体，光线必然先由一个交点进入球体，然后从另一个交点出来。变量 surface 的值设为 0，因为球体只有一个表面：我们以后将处理光线和多个表面相交的情况。在通用坐标系下，光线和球体的交点坐标也保存在 hitPointfield 中。根据定义可知，交点始终落在光线上位于给定的交点时间处，交点的坐标由下面方法得到(见练习)：

```
Point3 rayPos(Ray& r, float t);      //返回光线在 t 时刻的位置
```

计算球体表面在交点处的法向也非常简单：因为球面上点的法向由球心指向该点，法向的坐标和该点的坐标一致。

练习

练习 12.5.1　把场景中的光线转换到通用坐标系中

hit()方法中调用 xfrmRay()方法来把光线变换到球体的通用坐标系中，请写出 xfrmRay()方法的实现细节。请特别注意对点做变换和对向量做变换的区别。

练习 12.5.2　rayPos()方法

请实现文中提到的 rayPos()方法，它的输入参数为一条光线和一个时间，返回为在给定时刻光线的位置。

练习 12.5.3　光线何时与球面相切

讨论当光线和球面相切时，我们应该计算和保存哪些值。

练习 12.5.4　关于计算的复杂度

光线和球体求交需要多少计算量？在光线和球体求交计算中，需要进行多少次乘法、除法和开根号运算？

12.5.2　一个完整的光线跟踪器

现在我们手上已经有足够的工具来为由球体和椭球体组成的场景构建一个简单的光线跟踪器。在处理更复杂的场景前，我们要先确认前面介绍的光线跟踪器的各个部分在整合

到一起后仍能正常工作。在这个过程中的每个部分都没有浪费,因为所有的工具都在以后用到。

到目前为止,如果要设置物体的颜色,我们只能设物体为发光物体(回顾第 8 章),在 SDL 文件中设置物体的发光颜色:emissive 0.3 0.6 0.2。

我们很容易修改图 12.12 中的方法 Scene::shade()让它仅处理发光物体的发射光线:只要删除 color.add()语句,该语句添加其他组成光线:

```
Color3 Scene::shade(Ray&ray)
{
    Color3 color;
    Intersection best;
    getFirstHit(ray, best);
    if(best.numHits==0) return background;
    Shape * myObj=(Shape * )best.hit[0].hitObject;    //相交物体
    color.set(myObj->mtrl.emissive);
    retrun color;
}
```

图 12.10 中方法 Camera::raytrace()唯一需要修改的部分是计算相应于每个像素块的光线的发射方向[①]。等式(12.2)可以帮助我们计算经过第 ik 个像素块的坐下角点的光线的相关参数。案例分析 12.1 将更详细的讨论一个球体场景的光线跟踪器实现细节。

12.6 光线和其他基本几何体求交

我们需要实现其他形状类的 hit()方法。所有的 hit()都是相似的:光线首先被变换到物体的通用坐标系,然后计算光线和通用物体的角点。我们只需要处理一下每个物体和光线求交时的计算细节。

12.6.1 与正方形求交

正方形是经常用到的通用物体之一。通用正方形落在 $z=0$ 的平面上,x 和 y 的范围都是 -1 到 1。它的隐式表达式为 $F(P)=P_z$,$|P_x|\leqslant 1$ 且 $|P_y|\leqslant 1$。它可以变换到空间中任何位置上的平行四边形,所以经常用来表示场景中的墙壁和窗户物体等。hit()首先计算光线是否和通用正方形相交(见等式(12.10)),然后再检测交点是否落在正方形内,如图 12.17 所示。

① 注意,为了提高效率,一些变量的计算可以从内循环中去除。

```
//<<<<<<<<<<<<<<<<<<<与正方形求交>>>>>>>>>>>>>>>>>>> .
bool Square::hit(Ray &r, Intersection& inter)
{
    Ray genRay;                                          //用于生成泛型光线
    inter.numHits=0;                                     //初始化
    xfrmRay(genRay, invTransf, r);
    double denom=genRay.dir.z;                           //分母
    if(fabs(denom)<0.0001) return false;                 //光线和平面平行：无交点
    double time=-genRay.start.z/denom;                   //相交时间
    if(time<=0.0) return false;                          //交点落在视点后方
    double hx=genRay.start.x+genRay.dir.x*time;          //交点的 x 坐标
    double hy=genRay.start.y+genRay.dir.y*time;          //交点的 y 坐标
    if((hx>1.0) || (hx<-1.0)) return false;              //x 不在范围内
    if((hy>1.0) || (hy<-1.0)) return false;              //y 不在范围内
    inter.numHits=1;                                     //有一个有效交点
    inter.hit[0].hitObject=this;
    inter.hit[0].hitTime=time;
    inter.hit[0].isEntering=true;
    inter.hit[0].surface=0;
    inter.hit[0].hitPoint.set(hx,hy,0);
    inter.hit[0].hitNormal.set(0,0,1);
    return true;
}
```

图 12.17　Square 类的 hit()函数

12.6.2　与锥形柱体求交

图 12.18 显示了一个通用锥形柱体和几条光线。该柱体的侧面是一个“无限长围墙”的一部分，它在 $z=0$ 处的半径为 1，在 $z=1$ 处有个小于 1 的半径 s。以后将“无限长围墙”简称为围墙。由等式(6.30)可知，锥形柱体的侧面的隐式形式为

$$F(x,y,z)=x^2+y^2-(1+(s-1)z)^2,\quad 0<z<1$$

如果 $s=1$，这个形状变成通用圆柱体；如果 $s=0$，则变成通用圆锥体。我们实现锥形柱体的 hit()方法，自然也就得到了圆柱体和圆锥体的 hit()方法。

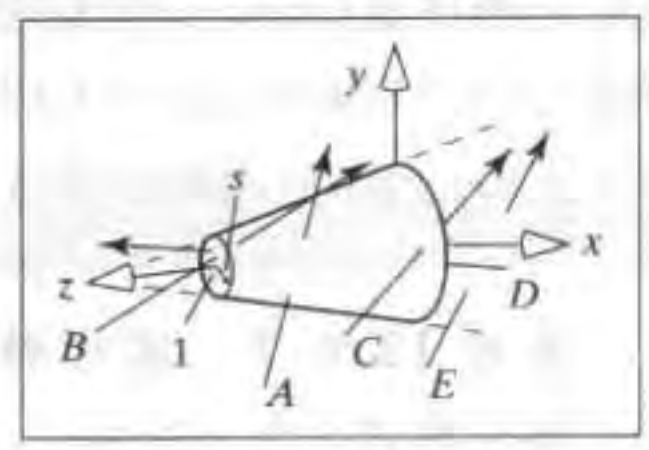


图 12.18　泛型圆台

图中还显示了几条光线，用来说明光线与物体相交或者不相交时的不同情况。光线 A 与柱体(简便起见，我们通常把锥形柱体简称为“柱体”)的侧面相交两次，而光线 B 先与围墙相交，然后从柱体的上底面进入柱体，最后从侧面穿出。光线 C 首先与围墙相交，然后从柱体的下底面穿出，而光线 D 从柱体的下底面进入，然后从上底面穿出。光线 E 虽然和围墙相交，但它处于柱体的外部，所以和柱体没有交点。图中没有显示与围墙完全分离的

光线。

由于存在如此多的情况,我们需要一个好的方法来避免多次的 if(),.else 检测。解决的办法是:按照任何一种方便的顺序来区分交点,而且将交点保存在 inter.hit[]列表中而不去考虑它们的顺序。最后一步,如果 inter.hit[]中有两次命中顺序颠倒,就交换这两项的位置。事实上这很容易做到,因为我们已经有一个列表来保存交点记录。

单个的相交关系测试是简单的。为了决定光线是否和围墙相交,我们只要把 $S+ct$ 代入锥形柱体的隐式形式,得到二次等式 $At^2+Bt+C=0$。容易知道,

$$\begin{aligned} A &= c_x^2 + c_y^2 - d^2 \\ B &= S_x c_x + S_y c_y - Fd \\ C &= S_x^2 + S_y^2 - F^2 \end{aligned} \tag{12.19}$$

其中 $d=(s-1)c_z$,且 $F=1+(s-1)S_z$。对于圆柱体和圆锥体,这些参数又变成怎样的形式?

我们使用的系数和球体情况时使用的系数形式相同。如果判别式 B^2-AC 的值为负,则光线在围墙外。如果判别式的值为非负,那么光线和围墙相交,且相交时间可以通过求解这个二次等式得到。为了检测光线是否和柱体相交,就要求出交点坐标的 z 分量。仅当 z 分量落在 0 和 1 之间时,光线和柱体的侧面相交。

为了检测光线和柱体下底面的相交情况,我们求解光线和平面 $z=0$ 的交点。假设交点坐标为$(x,y,0)$,那么当 $x^2+y^2<1$ 时交点落在下底面内。相似地,为了检测光线和上底面的交点,我们求解光线和平面 $z=1$ 的交点。假设交点坐标为$(x,y,1)$,那么当 $x^2+y^2<s^2$ 时交点落在上底面内。

一个柱体有多个表面,接下来我们想知道交点落在哪个表面上。例如,我们可能会给侧面和底面贴上不同的纹理。我们采用如下的编号方法:侧面编号为 0,下底面编号为 1,上底面编号为 2。然后,在相交记录的 surface 字段中保存相应的曲面编号。

将上述想法整合到一起并编写光线与锥形柱体的求交代码并不困难,但还是有些繁琐。处于简洁考虑,我们把细节留在练习里,要求读者补充完整。

还有一个信息我们需要得到,那就是交点处的法向量。如第 6 章中所述,可以简单得到柱体侧面在点(x,y,z)处的法向量为$(x,y,-(s-1)(1+(s-1)z))$。上下底面的法向量分别为$(0,0,1)$和$(0,0,-1)$。hit()将为 HitNormal 字段附上相应的值。

练习

练习 12.6.1 实现锥形柱体的 hit()方法

验证等式(12.19)中二次等式系数的正确性。将 TaperedCylinder::hit(Ray& ray, Intersection& inter)的代码补充完整。

练习 12.6.2 实现圆锥体的 hit()方法

如果你想编写一个专门用于 Cone 类的 hit()方法,那么它与 TaperedCylinder 类中的 hit()方法会有什么区别?说明如果要提高效率,可以修改哪几行代码。

练习 12.6.3 关于计算的复杂度

在求解光线与正方形或锥形柱体的交点时,分别需要进行多少次的加、减、乘、除运算?

12.6.3　与立方体(或任何凸多面体)求交

在多种图形环境中,凸多面体都被认为是非常有用的。我们也在本书中多处讨论到凸多面体的应用(例如第 6 章中对柏拉图立方体的讨论)。因为凸多面体是用它的边界平面来表示的,所以很容易实现光线与它求交的算法。我们将在本节的后面讨论这个问题。

通用立方体作为凸多面体的一个特例,应该引起我们的特别关注。它以原点为中心,顶点坐标为(±1,±1,±1),包含了所有+1 和−1 的所有 8 种组合。因此它的边与坐标轴平行,而且六个面落在图 12.19 所示的平面上。为了表述方便,我们给每个平面指定了一个名字。每个平面的名字根据它与观察者的位置关系指定,设观察者的位置在点(0,0,10),则名字为 top 的平面就位于立方体的上方,其他平面的位置以此类推。图中也给出了每个平面向外的法向量,以及平面中具有代表性的一个点 spot。

平面	名称	等式	向外法向	spot 点
0	top	y=1	(0,1,0)	(0,1,0)
1	bottom	y=−1	(0,−1,0)	(0,−1,0)
2	right	x=1	(1,0,0)	(1,0,0)
3	left	x=−1	(−1,0,0)	(−1,0,0)
4	front	z=1	(0,0,1)	(0,0,1)
5	back	z=−1	(0,0,−1)	(0,0,−1)

图 12.19　构成泛型立方体的六个平面

通用立方体之所以重要,有两个原因:

(1) 通过对立方体作用仿射变换,我们可以在场景中添加各种各样的有趣的"盒子"。然后在光线跟踪过程中,通过逆变换将每条光线变换到通用立方体的坐标系中,这样我们就可以利用光线与通用立方体的求交程序快速求出交点。

(2) 通用立方体可以作为场景中其他通用形状的**包围盒**:每个通用形状都内切于一个通用立方体。后面我们将会看到,在检测光线与物体的相交情况前先检测光线与物体包围盒的相交情况,将会大大提高程序的效率,特别是在光线与物体的求交计算很费时的情况下,提速的效果更加明显。如果光线与物体的包围盒不相交,那么光线必然与物体也不相交。因此,拥有一个光线与通用立方体的高效求交算法将会带来巨大的好处。

通用立方体的求交算法

在第 4.8.3 节中我们描述过的 Cyrus-Beck 算法,它处理 2D 的直线在凸窗口中的裁剪问题。光线与立方体的求交过程实质上就是 Cyrus-Beck 算法。该算法也可用于第 8 章中描述的直线与相机视景体的裁剪问题。它的基本思想是:立方体的每个平面把空间分为了"内"半空间和"外"半空间;光线上的一个点在立方体内,当且仅当它位于立方体的所有内半空间内。因此,光线与立方体的求交问题就转换为确定一个时间段的问题,在这个时间段内光线落在立方体所有平面的内侧。

设立方体为 P。对 P 的每个平面依次检测光线,计算光线进入或是离开平面内半空间

的时间。我们跟踪一个"候选"时间段 CI：该时间段基于目前为止的检测结果，在该时间段内光线有可能落在立方体内部。这个区间是 t_{in} 和 t_{out} 之间的部分，即 $CI=[t_{in},t_{out}]$。在时间段 CI 内，光线可能位于指定的半空间内；或者说，在时间段 CI 外，光线肯定不在半空间内。如果光线是进入指定的半空间且 $t<t_{in}$，或者光线是从内半空间中出来且 $t>t_{out}$，那么光线在半空间相关的平面的外侧。因为 P 的每个平面都要检测，所以我们对该区间进行"砍断"，或者是增大 t_{in} 或者是减小 t_{out}。在任何时候如果 CI 为空，那么光线必定与物体没有交点，提前结束光线与其他平面的相交检测。如果检测完 P 的所有平面后，CI 非空，那么说明光线在 t_{in} 时刻进入物体，在 t_{out} 时刻离开物体。

为了使这一过程更加形象，图 12.20 描述了一个光线在 $t=3.6$ 时进入通用立方体，在 $t=4.1$ 时离开。在这个例子中，检测结束后 CI 的结果为[3.6,4.1]，且光线确实和立方体相交。为了简便起见，图 12.21 给出了这一过程的一个 2D 版本，就是找出光线与正方形的交点。检测过程会因为立方体各个面的检测顺序不同而有所变化，但最终检测结果都是一样的。我们假设正方形的边的检测顺序与图中标的序号一致，由 1 号到 4 号。CI 初始化为 $CI=(-\infty,+\infty)$。光线从第一条边穿出的时间为 2.9，所以可知在 $t=2.9$ 以后光线必然不在正方形内，于是 CI 变成 $CI=(-\infty,2.9]$。检测第 2 条边可知，光线在更早的时候，$t=2.7$，就离开了正方形，所以 t_{out} 变为 2.7。检测第 3 条边可知，光线不可能在前进入正方形，所以 t_{in} 变成 1.6。最后检测第 4 条边，得到 t_{in} 为 1.8。于是，CI 的最终结果为[1.8,2.7]。

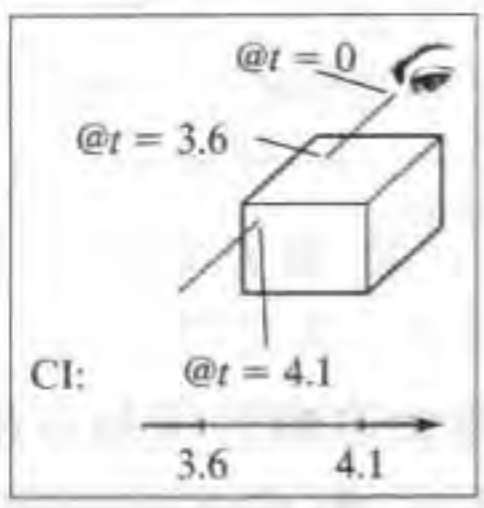


图 12.20 一条光线穿过泛型立方体

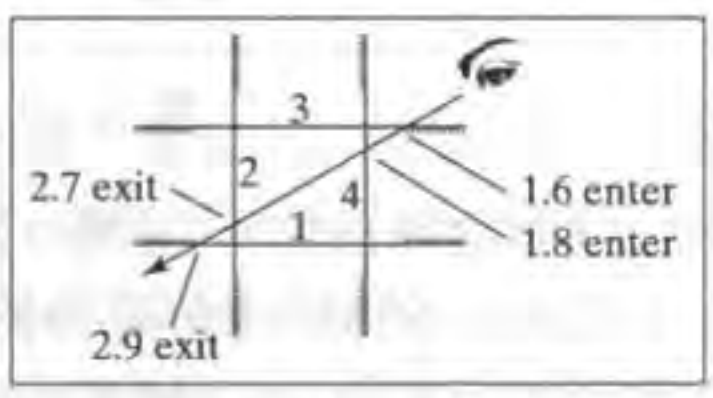


图 12.21 二维情况

在上述的每一步中，t_{in} 和 t_{out} 的值都是按照下面的伪代码进行调整：

```
初始化 tin为-∞，tout为∞ (对每个 CI)
if(光线在 thit进入)
        tin=max (tin, thit)
else if(光线在 thit离开)
        tout=min (tout, thit)
```

进一步具体化，令光线为 $S+\mathbf{c}t$，待检测的平面经过点 B 且具有向外的法向量 $\mathbf{m}$。该平面的隐式形式为 $F(P)=\mathbf{m}\cdot(P-B)$，所以当光线和平面相交时 t 满足 $\mathbf{m}\cdot(S+\mathbf{c}t-B)=0$，可得

$$t=\frac{numer}{denom}$$

其中

$$\begin{aligned} numer &= \mathbf{m}\cdot(B-S) \\ denom &= \mathbf{m}\cdot\mathbf{c} \end{aligned} \tag{12.20}$$

如果 $denom>0$，说明光线进入平面的外半空间（因为此时 **m** 和 **c** 的夹角小于 90°）；如果 $denom<0$，说明光线进入平面的内半空间。如果 $denom=0$，说明光线与平面平行，此时 *numer* 的值决定了光线是完全位于内半空间还是外半空间，正如第 4.8.3 节所述。图 12.22 归纳了所有这 4 种情况。

位置	条件
进入内半空间	$denom<0$
进入外半空间	$denom>0$
完全位于内半空间	$denom=0$，$numer>0$
完全位于外半空间	$denom=0$，$numer<0$

图 12.22　光线与平面内半空间的相交情况

所有这些思想对 3D 和 2D 情况都适用。对于通用立方体，*numer* 和 *denom* 的值可以很快计算得到：其中的点积计算得到很大地简化，因为在立方体的通用坐标系中，向量 **m** 有两个分量都是 0。图 12.20 中的平面产生了下列值（请检验它们）：

平面	*number*	*denom*
0	$1-S_y$	c_y
1	$1+S_y$	$-c_y$
2	$1-S_x$	c_x
3	$1+S_x$	$-c_x$
4	$1-S_z$	c_z
5	$1+S_z$	$-c_z$

图 12.23 给出了 Cube 类的 hit()方法的伪代码，它实现了光线和立方体的交点检测算法。和其他的 hit()方法一样，光线先被变换到物体的通用坐标系中，然后所有的检测都在光线和通用物体间进行。在一个循环语句中逐个检测六个平面，并适当改变 numer 和 denom 的值。CI 由最初的（$-\infty$，$+\infty$）砍断为最终的（tIn，tOut），除非在某次检测后它为空，此时说明光线和立方体没有交点。因为确定哪个平面和当前的 tIn 和 tOut 相关联是非常重要的，所以我们把这个信息保存在变量 SurfIn 和 SurfOut 中。

当所有的平面都经过测试，就可以确定相交时间 tIn 和 tOut。如果 tIn 为正，那么就将 tIn 时刻的交点信息加载到 inter.hit[0]中，将 tOut 时刻的交点信息加载到 inter.hit[1]中。如果只有 tOut 为正，就将该时刻的交点信息加载到 inter.hit[0]中。

与光线相交的平面的法向量可以通过一个辅助方法 cubeNormal(i)得到，该方法返回如图 12.20 所示的平面的向外法向量。例如 cubeNormal(0)返回(0,1,0)，cubeNormal(3)返回（−1,0,0）。参见本节后面的练习。

任意凸多面体的求交算法

拓展 hit()方法来适合任意凸多面体是很直接的。假设凸多面体有 N 个面，第 i 个平面包含点 B_i，且具有（向外）法向量 $\mathbf{m}_i$。除了利用等式(12.20)计算 *numer* 和 *denom* 外，立方体的 hit()基本不需要做任何修改。此时，需要两个费时的点积计算，而 for 循环变为：

```
bool Cube::hit(Ray& r, Intersection& inter)
{
    double tHit, numer, denom;
    double tIn=-100000.0, tOut=100000.0;        //正负无穷大
    Ray genRay;
    int inSurf, outSurf;                        //6个面中的哪个面
    xfrmRay(genRay, invTransf, r);
    for (int i=0; i<6; i++)
    {
        switch(i)                               //待检测的平面
        {
        case 0: numer=1.0-genRay.start.y; denom=genRay.dir.y; break;
            case 1, case 2, case 3, case 4 similarly
        case 5: numer=1.0+genRay.start.z; denom=-genRay.dir.z; break;
        }
        if(fabs(denom)<0.00001)                 //光线和平面平行
        {
            if(numer<0) return false;           //光线在外部
            else;                               //光线在内部,不改变 tIn 和 tOut 的值
        }
        else                                    //光线不和平面平行
        {
            tHit=numer/denom;
            if(denom>0){                        //射出
                if(tHit<tOut){                  //一个更早的射出时刻
                    tOut=tHit; outSurf=i;
                }
            }
            else {                              //denom 的值为负:进入
                if(tHit>tIn){                   //一个更迟的进入时刻
                    tIn=tHit; inSurf=i;
                }
            }
        }
        if(tIn>=tOut) return false;             // 不相交-提早结束
    }                                           //for 循环语句结束
    int num=0;                                  //仍未有有效的交点
    if(tIn>0.00001)                             //第一次相交在视点前方吗?
    {
        inter.hit[0].hitTime=tIn;
        inter.hit[0].surface=inSurf;
        inter.hit[0].isEntering=1;              //进入
        inter.hit[0].hitObject=this;
        inter.hit[num].hitPoint.set(rayPos
                (genRay.start,genRay.dir,tOut));
        inter.hit[num].hitNormal.set(cubeNormal(outSurf));
        num++;
    }
    inter.numHits=num;                          //在视点前方的交点个数
    return (num>0);
}
```

图 12.23 Cube 类的 hit()方法

```
for (int i=0; i<N; i++)     //对多面体的每个面
{
    numer=dot3(m_i, B_i-S);
    denom=dot3(m_i, c);
    if (fabs(denom)<eps)… 和以前一样
    … 同前 …
}
```

网格物体的求交算法

从第 6 章开始，我们就使用了各种各样的网格物体，而且已经有了一个可以用于任何网格物体的 drawOpenGL()方法，因此自然想到对网格物体进行光线跟踪。我们利用前面介绍的方法实现一个网格物体的 hit()方法。回忆前面的内容，我们用面列表描述 Mesh 对象，每个面又是一些顶点以及顶点处的法向量组成的列表。我们依次取网格的每个面，每个平面都是一个有界平面，并称该平面所在的无界平面为"面平面"(face plane)。因此，我们要跟踪的网格物体的形状就是所有"面平面"的内半空间的交。

图 12.24 给出了 2D 平面上的两个形状，并标记了它们的面平面(直线)。图 12.24(a)中的图形具有凸形状，所以它的面平面和它的边界平面一致。这个图形将会被正确的光线跟踪。图 12.24(b)中的图形具有一个非凸形状，我们用阴影标记了包含在所有面平面内半空间的区域，该区域是光线跟踪器将会绘制的形状。因此，对于非凸形状的物体，光线跟踪器利用 Cyrus Beck 算法将会得到一个错误的结果，Cyrus Beck 算法只适用于具有凸形状的网格物体。

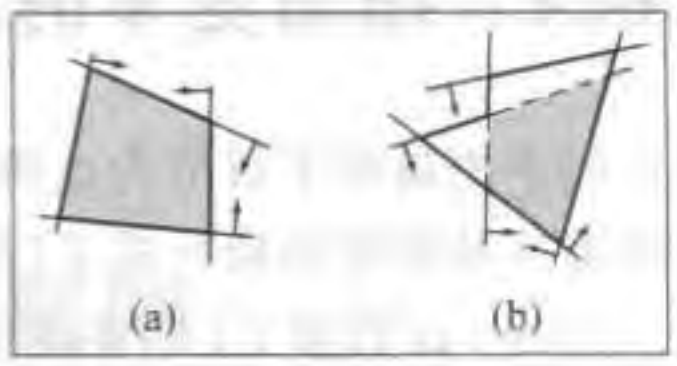


图 12.24　2D 的物体以及它的面平面
(a) 凸形状；(b) 非凸形状

为了实现网格对象的 hit()方法，我们依次为每个面设置两个向量 numer 和 denom，并把面的第 0 个顶点 pt[face[f].vert[0].vertIndex]设置为面的代表点，该点的法向量与面平面的法向量 norm[face[f].vert[0].normIndex]一致。

练习

练习 12.6.4　设置立方体交点处的法向量

编写方法 Vector3 cubeNormal(int which)，该方法返回立方体的面 which 的向外法向量，如图 12.20 所示。最具可读性的代码可能会是使用一个具有 6 种判断条件的 switch 语句，然后将向量设置为 6 个已知法向量中的一个。稍微简单一点的判断并设置法向量的方法如下所示：

```
int m=which/2, n=(which%2)?-1:1;
if(m==0) v.set(0, n, 0);
else if(m==1) v.set(n, 0, 0);
else v.set(0, 0, n);
```

说明这种方法是如何运作的，并利用此方法重新编写方法 cubeNormal()。

练习 12.6.5 四面体的面

根据图 6.27,说明正四面体的面列表中包含的内容。

练习 12.6.6 修改 hit(),使之适合于凸多面体

补充光线与凸多面体求交的 ConvexPolyhedron :: hit ( ) 方法的细节。关于 ConvexPolyhedron 类的细节,请参考本书相关网站上的关于 Shapes 类的说明。

练习 12.6.7 与四面体相交

对于顶点为(1, 0, 0),(−1, 0, −1),(−1, 0, 1)和(0, 1, 0)的四面体,请确定光线(0, 0, 0)+(1, 2, 3)t 在其内部的时间段。

练习 12.6.8 关于计算的复杂度

在对凸多面体的每个面进行交点检测时,需要进行多少次的乘法/除法运算?有时可能只检测了几个平面后,就因为 tIn 超过了 tOut 而提前结束检测。试问,当随机选择一条光线进行相交检测时,平均需要检测多少个平面?试将此计算的复杂度与光线和球体的求交计算的复杂度作比较。

12.6.4 添加更多的基本形状

人们不用局限于目前为止所考虑的通用形状,而可以涉及一些其他类型的形状。我们只要知道这些形状的隐形式 $F(P)$,然后就可以像以前那样,确定光线 $S+\mathbf{c}t$ 在何处与表面相交。用 $S+\mathbf{c}t$ 代替 $F(P)$中的 P,形成了以下对时间 t 的函数:

$$d(t) = F(S + \mathbf{c}t) \tag{12.21}$$

该函数具有以下性质:

- 当光线上的点在物体的外部,函数值为正;
- 当光线上的点在物体的表面上,函数值为零;
- 当光线上的点在物体的内部,函数值为负。

在计算交点的时候,我们寻找满足 $d(t)=0$ 的 t 值,所以与一条光线求交时相当于求解这个等式。对于球面以及其他二次曲面,我们已经知道这个等式是简单的二次等式。

但对于圆环面就不同了。通用圆环面具有如下隐式形式

$$F(P) = \left(\sqrt{P_x^2 + P_y^2} - d\right)^2 + P_z^2 - 1 \tag{12.22}$$

所以等式 $d(t)=0$ 是四次的,这使得求解比较困难,虽然可以在很多数学手册中找到它的显式解。

更一般地,我们考虑函数 $d(t)$的曲线形状。如果光线的起点 S 位于物体的外部,并且发射方向指向物体,那么 $d(t)$的形状与图 12.25 中的形状类似。

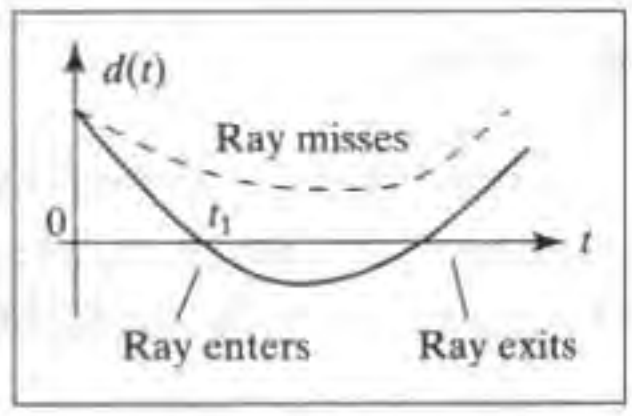


图 12.25 对于给定光线和物体的 $d(t)$函数的形状

因为光线的起点 S 位于物体的外部,所以 $d(0)$的值为正。当光线靠近物体时,$d(t)$的值逐渐减小,并且 $d(t)=0$ 时光线到达物体表面。然后一段时间光线位于物体内部,且 $d(t)<0$。当光线从物体中出来,$d(t)$经过零点,然后逐渐增

大。对圆环这样的物体，光线可能再次进入物体内部（请勾画一个具有这样性质的物体）。相反地，如果光线和物体没有交点（图中用虚线表示的情况），那么 $d(t)$ 减小一段时间后，在未到达零之前，就会一致增大下去。

对于像球面这样的二次曲面，$d(t)$ 是抛物线形状；对于圆环，$d(t)$ 是一个四次曲线形状。对于其他曲面，$d(t)$ 的形状可能会非常复杂，此时就需要利用数值计算的方法来求解等式 $d(t)=0$。

为了找到满足 $d(t)=0$ 的 t 的最小正值，我们需要逐步求解 $d(t)$ 的值，使 $d(t)$ 不断逼近于零。很多数值方法，如牛顿迭代法【Acton70，Conte 和 deBoor80】，都致力于得到更优的 t 值，但是这些方法都需要多次迭代计算，因此也大大降低了光线跟踪的速度。

人们还开发出了一些其他类别物体的光线跟踪方法，特别是分形曲面、旋转曲面和拟圆柱面【Kajiya83】。参见案例分析中对这些物体进行光线跟踪的方法。寻找由超大数量的物体组成的场景的高效光线跟踪算法，仍然是当前图形学研究中的一个课题。

练习

练习 12.6.9　与圆环相交

根据数学手册，确定如何得到一个四次等式的解，并设计必要的步骤来求出光线 $S+ct$ 与圆环等式(12.22)的交点。

12.7　绘制场景的明暗图

设计一个发光物体的光线跟踪器是开始学习光线跟踪算法的好方法，但是当要绘制一幅具有强烈真实感的场景时，你会发现这样的光线跟踪器是远远不够的。为了产生逼真的图像，我们必须确定在交点处反射到人眼的光线的性质。为了推导出整个计算过程，我们将回顾一下第 8 章中讨论的 OpenGL 明暗着色模型，然后讨论如何改进现在的模型以得到更逼真的效果。

OpenGL 在绘制场景时考虑了顶点处的环境光线、漫反射光线和镜面反射光线，并利用 Gouraud 模型计算得到在曲面上光滑变化的漫反射光线。图 12.26 归纳了光线的主要成分。图中，第 rc 条光线与一个椭球体在 P_h 点处相交，光源设在 L 处。我们希望计算有多少光在 P_h 点处返回到眼睛里。计算将涉及到的几个主要向量为 **s**、**v** 和 **m**。向量 **s** 指向光源，所以可以由 $\mathbf{s}=L-P_h$ 式得到；向量 **v** 指向观察者，所以与光线的方向 **dir** 相反；**m** 是 P_h 点在表面上的法向量。

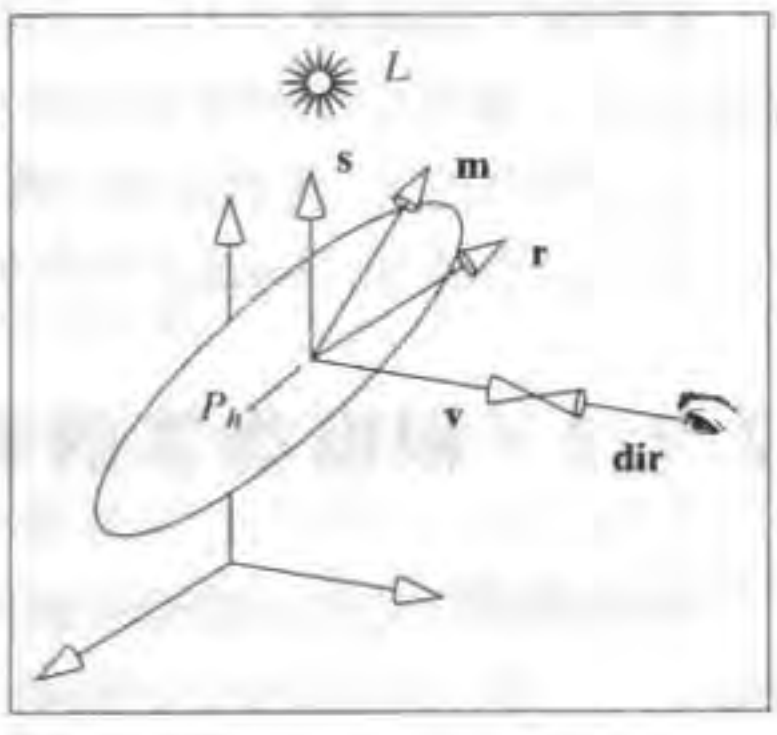


图 12.26　明暗着色模型

正如第 8 章中所述，眼睛观察到的光的总强度由环境光、漫反射光和镜面反射光三部分组成，即

$$I = I_a\rho_a + I_d\rho_d \times lambert + I_{sp}\rho_s \times phong^f \tag{12.23}$$

其中,我们定义

$$lambert = \max\left(0, \frac{\mathbf{s}\cdot\mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right) \quad 和 \quad phong = \max\left(0, \frac{\mathbf{h}\cdot\mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right) \tag{12.24}$$

I_a 是环境光强度,I_{sp} 是光源强度。ρ_a、ρ_b 和 ρ_s 分别是环境光、漫反射光和镜面反射光的反射系数。向量 **h** 是由 **h**=**s**+**v** 定义的中间向量。

如果场景中有多个光源,那么每个光源都会产生漫反射光和镜面反射光线。这些光线都将添加到等式(12.23)中。回顾环境光的定义,它实际上是场景中相邻物体间多重反射光的近似合成。后面将看到,当一个物体足够亮时,就必须跟踪它的经过多次反射后的光线。场景中也有些光线数量多强度弱,很难分别对它们进行光线跟踪,I 中环境光的部分有助于模拟这些光线的效果。

和第 8 章中的做法一样,我们把这个等式推广到彩色光源和物体上。此时,到达眼睛的光线包含红、绿、蓝三种成分,它们分别由下式给出:

$$\begin{aligned} I_r &= I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times lambert + I_{spr}\rho_{sr} \times phong^f \\ I_g &= I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times lambert + I_{spg}\rho_{sg} \times phong^f \\ I_b &= I_{ab}\rho_{ab} + I_{db}\rho_{db} \times lambert + I_{spb}\rho_{sb} \times phong^f \end{aligned} \tag{12.25}$$

其中每组 I 和 ρ 都是相应颜色成分的强度和反射系数。值得注意的是,这 3 种颜色成分使用了同样的项 *lambert* 和 *phong*(以及同样的向量 **s**、**v** 和 **m**),因此对一条光线只需要跟踪一次,而 *lambert* 和 *phong* 也只需要计算一次。

明暗着色计算要求知道交点 P_h 的坐标以及向量 **s**、**v** 和 **m**。下面讨论如何计算 **m**,这将利用到一些信息,这些信息都已经保存在相交记录中。

12.7.1 计算交点处的法向量

向量 **m** 是物体表面在交点处的法向量,如何确定它呢?首先我们在通用坐标系下可以相对容易计算它,然后再把它变换到世界坐标系中。在第 6.5.3 节中,我们曾说明如果物体经过矩阵 M 变换到另一个物体,那么法向量 $\mathbf{m}'$ 将会变换成法向量

$$\mathbf{m} = M^{-T}\mathbf{m}' \tag{12.26}$$

其中 M^{-T} 是矩阵的 M 的逆转置矩阵。于是,通过计算通用物体在交点处的法向量,然后乘以 M^{-T} 就可以得到要求的法向量。

令人高兴的是,我们已经计算并保存了通用坐标系中交点处的法向量;该法向量信息保存在相交记录的 hitNormal 字段中。

12.7.2 根据物体的表面材质为其着色

我们看到的交点处的颜色取决于环境光、漫反射光和镜面反射光成分的合成。反射系数如 ρ_{ar}、ρ_{dg} 等保存在相交记录的 mtrl. ambient、mtrl. diffuse 和 mtrl. specular 字段中。当首次读入 SDL 文件时,这些字段将被加载到内存里。

在实践中一个普遍的做法是将 Phong 镜面反射光设置为与光源一样的颜色。这样只

要把反射系数 mtrl. specular 的红、绿、蓝三个分量设置为相等，例如(0.9，0.9，0.9)，那么用此系数乘以光源的颜色得到的仍然是相同的颜色，只是亮度减弱。这样的设置使物体看起来像有光泽的塑料。

在光线跟踪器中增加环境光、漫反射光和镜面反射光成分并不困难。图 12.27 用伪代码描述了如何修改图 12.12 中的 shade()方法。

```
Color3 Scene::shade(Ray& r)
{
    调用 getFirstHit(r, best)得到第一个交点;
    生成副本 h=best.hit[0];                          //第一次相交记录
    根据 h.hitTime 计算 hitPoint
        Form v=-ray.dir;                             //向着观察者方向
    v.normalize();
    Shape* myObj=(Shape*)h.hitObject;                //指向相交物体
    Color3 color(myObj->mtrl.emissive)①;             //从发光部分开始
    color.add(ambient contribution);                 //计算环境光颜色
    Vector3 normal;
    //将一般法向量转换为世界法向量
    xfrmNormal(normal, myObj->invTransf, h.hitNormal);
    normal.normalize();                              //归一化
    for(each light source, L)                        //累加所有光源
    {
        if(isInShadow(…)) continue;      //如果在阴影中,跳到下一个光源
        Form s=L.pos-hitPoint;                       //由交点指向光源的向量
        s.normalize();
        float mDotS=s.dot(normal);                   //Lambert 系数
        if(mDotS>0.0)                                //交点面向光源
            Form diffuseColor=mDotS*myObj->mtrl.diffuse *L.color;
        color.add(diffuseColor);                     //添加漫反射光部分
        Form h=v+s;                                  //中间向量
        h.normalize();
        float mDotH=h.dot(normal);                   //phong 系数
        if(mDotH<=0)continue;                        //无镜面发射光
        float phong=pow(mDotH, myObj->mtrl.specularExponent);
        specColor=phong*myObj->mtrl.specular*L.color;
        color.add(specColor);
    }
    return color;
}
```

图 12.27　伪代码，在 shade()中添加明暗计算

检查 getFirstHit()方法中得到的最佳相交记录 best，得到光线与物体的第一个交点信息。复制 best. hit[0]到 HitInfo 记录 h 中，交点 hitPoint 的坐标可以利用光线等式和时间 h. hitTime 得到。通用坐标系中交点的法向量保存在 h. hitNormal 中，然后利用等式

① 我们假设 Vector3 类和 Color3 类已经具有了一些简单的构造函数：Vector3 s(A,B) 构造由点 A 指向点 B 的向量；Color3 col(f, d, c)构造一个颜色，它的红色部分为 f * d. red * c. red，蓝、绿部分类似。

(12.26)变换到世界坐标系中。接着,沿着光线返回的各种颜色成分保存在 color 中。这样就求出了放射光线和环境光线部分,同时对于该光源,求出它们所产生的漫反射光和镜面反射光成分并将其添加到 color 中。如果光源在交点处被某个对象遮住,就不用考虑该光源发出发出的光线。在第 12.10 节中我们曾讨论过阴影效果。现在只要删除其中"if (isInshadow (…)) continue;"这一行,就可以不进行任何阴影测试。

根据等式(12.23)确定镜面反射光部分,这要求计算向量 **v** 和 **h** 并利用保存在 myObj->mtrl.specular 中的镜面反射系数(和指数 f)。

图 12.28 显示了一个场景,其中的物体已经用环境光、漫反射光和镜面反射光进行了渲染,但是没有计算阴影。镜面高光使物体看起来像是由有光泽的塑料制成的。

图 12.28 综合环境光、漫反射光和镜面反射光渲染场景

练习

练习 12.7.1 变换法向量

编写一个函数 void xfrmNormal(Vector3& res, Affine4& aff, Vector3& v),用来计算向量 v 和矩阵 aff 的转置的乘积,并把结果保存在 res 中。在光线跟踪器中,我们会调用函数 xfrmNormal(normal, myObj->invTransf, h.hitNormal)。也就是说,该函数利用保存在相交对象中的逆矩阵 myObj->invTransf,对通用坐标系中的法向量 h.hitNormal 进行变换,最后得到世界坐标系中的法向量。

12.7.3 基于物理的明暗着色模型—Cook Torrance 明暗处理

Phong 高光很容易产生,但是总使物体看上去像有光泽的塑料。如果使用 OpenGL 为场景明暗着色,就必须解决 Phong 高光问题。但是如果通过光线跟踪算法渲染场景,就有了多种选择,并且在计算镜面反射高光时可以考虑采用不同的算法。

为了追求更加逼真的视觉效果,许多研究人员已经开发出了更精确的明暗着色模型。这些模型从物理原理出发,分析了光如何从一个实际表面反射,并建立一些数学表达式来表示到达人眼的反射光的强度和颜色。这些模型主要关心物体表面上光能的"守恒":入射光的一部分转化为热能被材料吸收;一部分与表面发生相互作用,变成漫反射光;还有一部分则转化为镜面反射光从表面反射出去。不同材料对入射光的分裂作用是不同的:例如,粗糙的表面比光滑、有光泽的表面产生的漫反射光更多,而镜面反射光更少。

在 Torrance 和 Sparrow【Torr67】以及 Trowbridge 和 Reitz【Trowbridge75】的早期工作中,粗糙表面被概念化地模拟为许多"微平面"(microfacet)的集合,这些微平面有着不同的方向,如图 12.30 所示。入射光线与平均法线方向 **m** 的夹角为 ϕ,当入射光到达不同的微平面时,就会向不同的方向反射出去,其中有一部分光线反射到人眼。

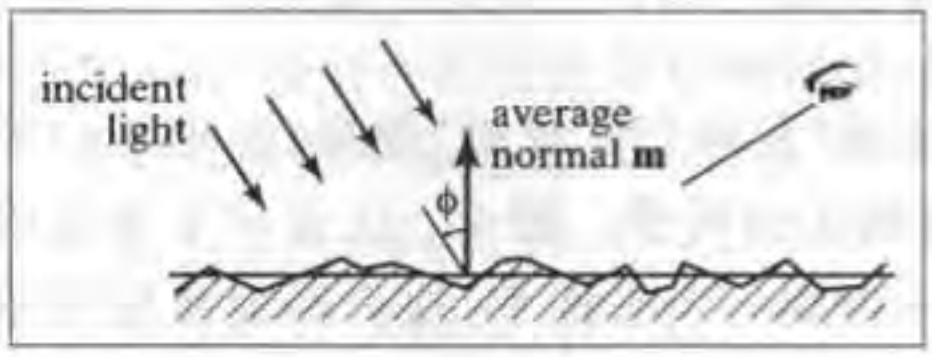


图 12.29　粗糙表面模拟为随机取向的微切面的集合

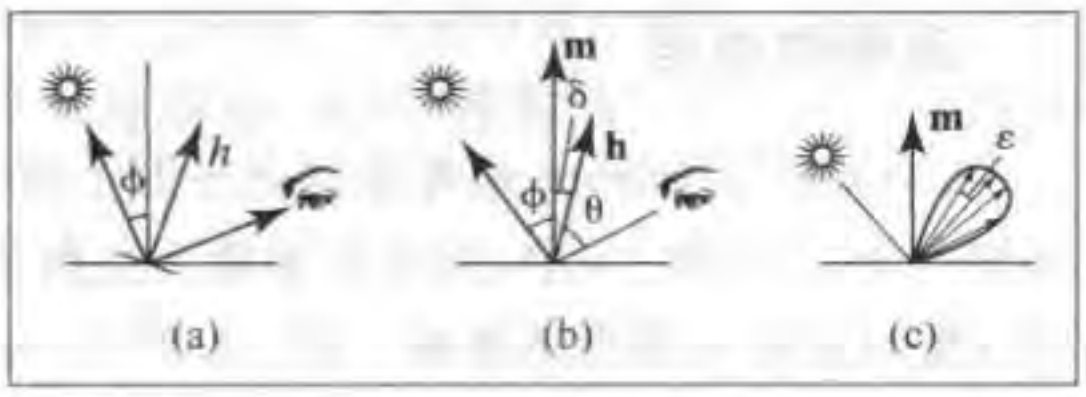


图 12.30　确定方向为 δ 角的微平面所占的比例

Blinn【Blinn77】采用这个模型开发了一种适用于计算机图形学应用的算法，并显示了该算法产生的高光效果与 Phong 高光有很大的不同。Cook 和 Torrance【Cook82】发展和改进了 Blinn 的模型，在反射高光中引入了“颜色偏移”(color shift)，这样就与真实材料上的光反射效果更加匹配。

下面我们介绍 Cook-Torrance 模型的主要部分，并说明如何将它应用于光线跟踪器中。讨论这个模型的 3 个基本方面以及它们对镜面反射光数量的影响。每个方面都试图通过建立相应的算法来准确表达光反射所涉及的物理现象。

微平面法向量的分布

Cook-Torrance 模型假设每个微平面都相当于一个理想的镜面，而且只有那些方向恰好吻合的平面才对特定方向上的反射光线有贡献。如图 12.31(a)所示，只有那些以 $\mathbf{h}=\mathbf{s}+\mathbf{v}$ 为法方向的平面才对在方向 $\mathbf{v}$ 上见到的光有贡献(方向 $\mathbf{v}$ 指向观察者)。因此，有必要知道哪些微平面具有此法方向。人们已经对某些特定材料制成的表面上的微平面法方向的分布情况进行了统计研究。研究得到了一个分布函数 $D(\delta)$，它反映了法方向和曲面的法向量成 δ 角的微平面在所有微平面中占的比例。如图 12.30(b)所示，如果入射角为 ϕ，观察者位于偏离微平面法向量 θ 角的位置，那么只有微平面的法方向和曲面法向 $\mathbf{m}$ 成角度 $\delta=(\theta-\phi)/2$ 时，微平面才能将入射光线反射到观察者(参见练习)。具有此法方向的微平面所占的比例为 $D(\delta)$。

人们已经发现了多种分布函数，Cook 和 Torrance 采用了 Beckmann 分布函数【Beckmann63】：

$$D(\delta)=\frac{1}{4m^2\cos^4(\delta)}e^{-\left(\frac{\tan(\delta)}{m}\right)^2} \tag{12.27}$$

其中参数 m 度量曲面的粗糙程度。特别地，可以取 m 为微平面斜率的均方根。对于相对光滑的表面 m 的值大约为 0.2；而对于粗糙的表面，m 的值大约为 0.6。实验表明，Beckmann 分布是很多粗糙物体表面很好的数学近似。$D(\delta)$在 $\delta=0$ 处有最大值(最大值是多少?)，随着 δ 远离 0，$D(\delta)$的函数值也逐渐减小；m 的值越小，$D(\delta)$的函数值下降越快。

Cook 和 Torrance 将 $D(\delta)$作为不同方向上镜面反射作用的比例因子，其中 δ 为 $\mathbf{h}$ 和 $\mathbf{m}$ 之间的夹角。图 12.30(c)显示了 $m=0.3$ 时反射光强度随观察方向的变化。图中给出的是与图 8.12 类似的“光束方向图”。每个方向上反射光的相对强度用向量的长度表示。在全反射方向上(角 ϕ 处)反射光强度最大，这是因为该平面的法线最可能与 $\mathbf{m}$ 平行。如果偏离了这个理想镜面的方向(例如偏离了 ε)，反射光强度就变小了，这是因为该平面的法线方向偏离向量 $\mathbf{m}$ 角 ε 后，分布函数在此处的值也变小，于是只有较少的平面具有此法线方向(参见本节后面的练习)。

阴影和遮挡

Cook 和 Torrance 还考虑到微平面上将会产生的"阴影"和"遮挡"效果,并把它们都归结到一个"几何项"G 中,用来作为镜面反射光强度的比例因子。图 12.31 显示了平面的几何位置对反射光强度的影响。图 12.31(a)中,光入射到一个平面上,这个平面所处的角度使之能够完全被照亮而且所有的反射光都能从该平面上"逃逸"出去。这种情况下,G 等于 1。图 12.31(b)中,反射光离开平面时,有一部分被相邻平面遮挡住了,此时相应地减小了 G 的值。而在图 12.31(c)中,平面只有一部分被照亮,部分入射光被相邻平面遮挡住了而在平面上产生阴影,此时 G 小于 1。

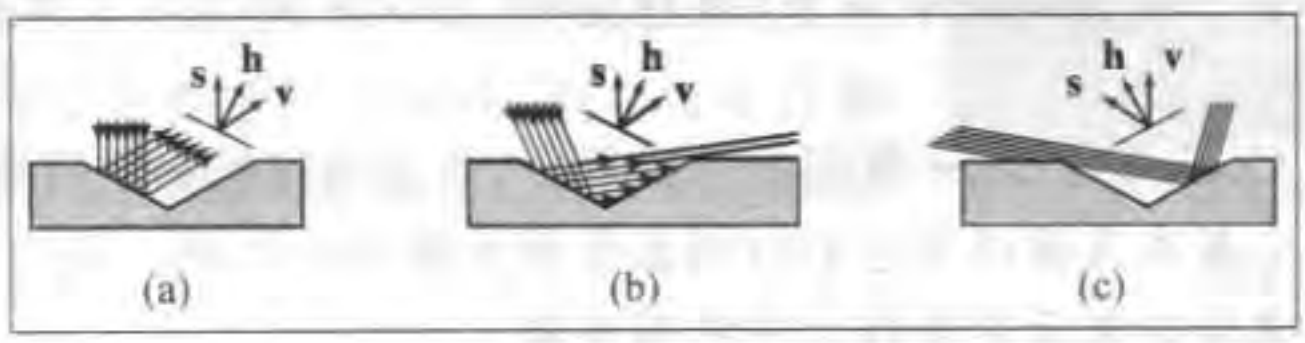


图 12.31 微平面上光的遮挡和阴影

Blinn 引进了几何因子来表示这些几何现象,在这些等式中只有涉及到简单的点积的运算。几何因子取 1、G_s 和 G_m 三者中的最小值:

$$G = \min(1, G_m, G_s) \tag{12.28}$$

其中

$$G_m = \frac{2(\mathbf{m}\cdot\mathbf{h})(\mathbf{m}\cdot\mathbf{s})}{\mathbf{h}\cdot\mathbf{s}}, \quad G_s = \frac{2(\mathbf{m}\cdot\mathbf{h})(\mathbf{m}\cdot\mathbf{v})}{\mathbf{h}\cdot\mathbf{s}} \tag{12.29}$$

分别表示没有被遮挡住的那部分反射光和没有被遮挡住的那部分入射光。

菲涅耳(Fresnel)系数

在 Torrance-Sparrow 模型中的第三个因素也可以影响镜面反射光的数量。微平面并不是理想的镜面,但是它像实际材料那样,只反射部分的光线,其余部分则被表面吸收。被反射的那部分光由菲涅耳系数[①] $F(\phi,\eta)$决定,其中 ϕ 是图 12.30 所示的光线的入射角($\mathbf{m}$ 和 $\mathbf{s}$ 之间的夹角),η 是材料的折射率。在后面的章节 12.11 中,当光线跟踪透明材料[②]时将会用到折射率,但是对折射率的使用其实是从金属这样的不透明材料开始的。

对于折射率为 η 的表面,菲涅耳系数 $F(\phi,\eta)$可以由电磁反射第一定律推导出来。计算等式为

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left\{1+\left(\frac{c(g+c)-1}{c(g-c)+1}\right)^2\right\} \tag{12.30}$$

其中 $c=\cos(\phi)=\mathbf{m}\cdot\mathbf{s}$, $g^2=\eta^2+c^2-1$。注意,中间项 g 由材料的折射率决定。对于许多材料来说,折射率随光波波长的变化非常复杂,这就使得 F 与光波波长的关系也非常复杂。

---

① 菲涅耳(Augustin Jean Fresnel, 1788—1827),法国物理学家,提出很多关于光反射、折射以及偏振现象的基本理论。

② 材料的折射率是光在真空中的传播速度与光在材料中传播速度的比值。它一般和光的波长有关系。在本章后面我们会介绍光的波长和它颜色之间的关系。

图 12.33 显示了对于不同的 η 值，F 相对 ϕ 的变化曲线。从图中可以看出，菲涅耳系数在“垂直入射”处($\phi=0$)的值与材料相关，然后平滑地变化，直到“掠射角”($\phi=\pi/2$)处到达 1.0。菲涅耳系数不会超过 1(为什么?)

- 在垂直入射时，$F(0,\eta)<1$；
- 在掠射时，$F(\pi/2,\eta)=1$。

Torrance 和 Sparrow【Torrance67】在微平面模型中综合考虑了上述因素，得到了一个基于物理原理的镜面反射模型。他们把镜面反射光的强度定义为 F、D 和 G 这 3 项的乘积与 $\mathbf{m}\cdot\mathbf{v}$ 的商，即

$$spec = \frac{F(\phi,\eta)DG}{(\mathbf{m}\cdot\mathbf{v})} \tag{12.31}$$

其中的分母 $\mathbf{m}\cdot\mathbf{v}$(就是 $\cos(\theta)$，参见图 12.31(b))是根据下面的论述推导出来的：以掠射角(θ 大余弦值就小)观察表面与以一个小角 θ 观察表面相比，在给定的立体角范围内看到更多的微平面。看见的微平面的数量越多，产生的光也越多。因此 $\mathbf{m}\cdot\mathbf{v}$ 的值越小，得到光的强度也越大。

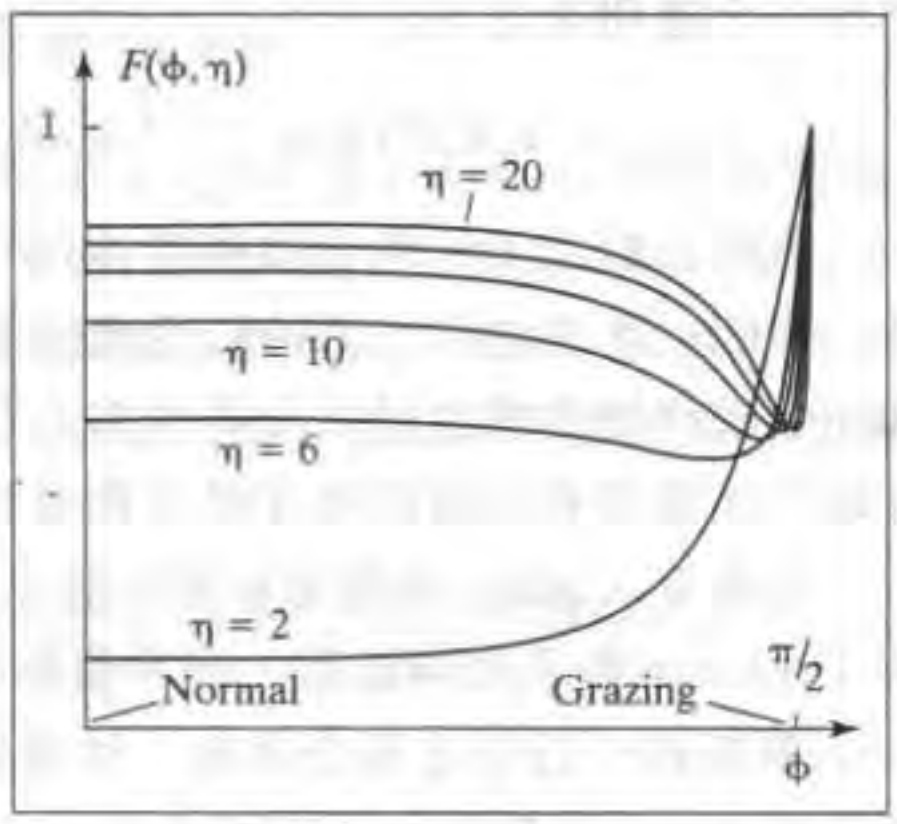


图 12.32　菲涅耳系数与入射角的关系

Blinn【Blinn77】对这个模型做了适当的改进以适合于计算机图形学的应用，并将它与 Phong 高光做了比较。尤其令人感兴趣的是，以不同的入射角 θ 得到的镜面反射光的作用方法也不同。根据 Phong 模型，镜面反射高光在“理想射入”方向上的光强总是最强的，而当视角稍微偏离这个方向之后，反射高光就会骤然下降。图 12.33(a)显示了“光束方向图”，描述了 Phong 高光的光强如何随着角度的变化而减弱。图中显示了两种入射角的光束方位图，无论入射角 ϕ 的大小如何，光束方向图的形状总是一样的。微平面模型的光束方向图见图 12.33(b)(本节后面的练习中推导了光束方向图的表达式)。当 ϕ 较小时，光束方向图的形状类似于 Phong 高光。但是对于大掠射角，不仅光束方向图的形状不一样，而且最大强度的方向也不再是镜面反射的方向！这使镜面反射高光周围的光环与 Phong 高光周围的光环完全不同，这更接近真实材料表面的反射光所观察到的效果。

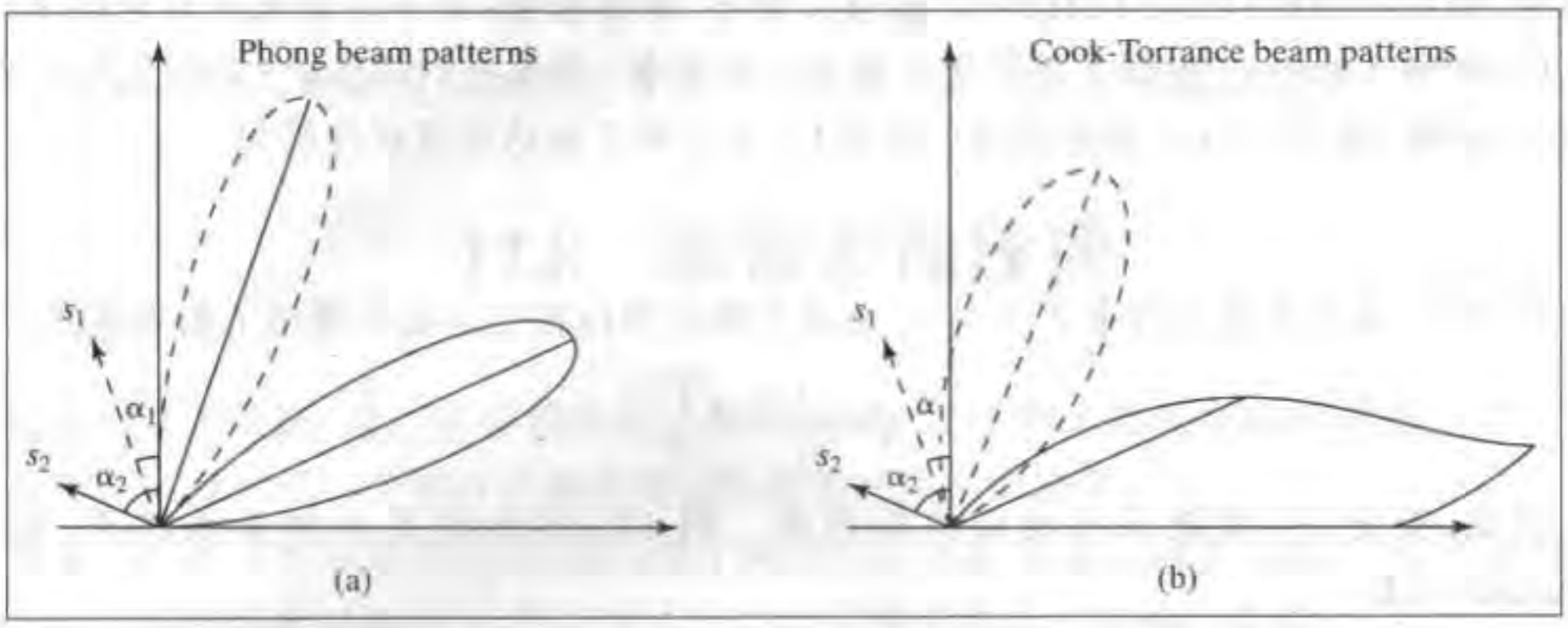


图 12.33　不同入射角的光线产生的镜面反射光的强度比较

Cook 和 Torrance【Cook82】在两个方向上扩展了这个模型。首先,它们考虑了如何将入射光的能量分成环境光、漫反射光和镜面反射光。其次,他们考虑到在实际材料中折射率随入射光波长的变化。这个变化也将引起菲涅耳系数随波长的相应变化,结果就导致镜面反射光随不同的入射角而改变颜色。这种颜色随着入射角度的变化,或称为"颜色偏移",在真实世界中也能被观察到。

如果上述每种现象都有一个表达式恰当表示,那么组合这些等式就可以得到关于此模型的一个通用表达式:

$$I_r = I_{ar}k_aF(0,\eta_r) + I_{sr}d\omega k_dF(0,\eta_r) \times lambert + I_{sr}k_sd\omega\frac{F(\phi,\eta_r)DG}{(\mathbf{m}\cdot\mathbf{v})} \quad (12.32)$$

该式只给出了红色光的强度;对于绿光和蓝光,除了将 η_r 分别换成 η_g、η_b 外,其他形式是一样的。环境光和光源的红色光强度分别为 I_{ar} 和 I_{sr},而环境光、漫反射光和镜面反射光这 3 个成分的贡献也可以分辨出来。I_{ar} 通常占 I_{sr} 的一小部分(如果存在多于一个光源,而且每个光源都有类似的漫反射光和镜面反射光,就是把所有光源的贡献加到一块)。

当垂直入射时,环境光和漫反射光的反射系数取决于菲涅耳系数 $F(0,\eta_r)$,这是 Cook 和 Torrance 做的简单近似。当 ϕ 值在 0 附近变动时,$F(\phi,\eta_r)$ 只有微小的变化,因此只有在 $\phi=0$ 附近这个近似才是合理的。注意到对于不同的波长(它们对应不同的颜色",这个系数是不同的:绿光和蓝光分别由 $F(0,\eta_g)$ 和 $F(0,\eta_b)$ 决定,因为不同颜色光的折射率不同,所以这两个值也不相同。

漫反射项中包含了我们所熟悉的 *Lambert* 项(参见式(12.24)),以及其他两个因子。$d\omega$ 项是光源相对于交点处的立体角(参见图 12.34)。为了简便起见,我们认为这一项对于场景中每一个点都是一个常数,而且通常令它为一个很小的常数,比如 0.0001(从地球观察太阳时,这个角的球面度为 0.000068)。因子 k_d(在镜面反射项中与之相对应的是 k_s)用来表示入射光是如何被分成漫反射光和镜面反射光的。这两个因子体现了材料的特性,且它们的和为 1,$k_d+k_s=1$。镜面反射项就是等式(12.31)中的 *spec* 项,其中菲涅耳系数对波长存在着明显的依赖关系。

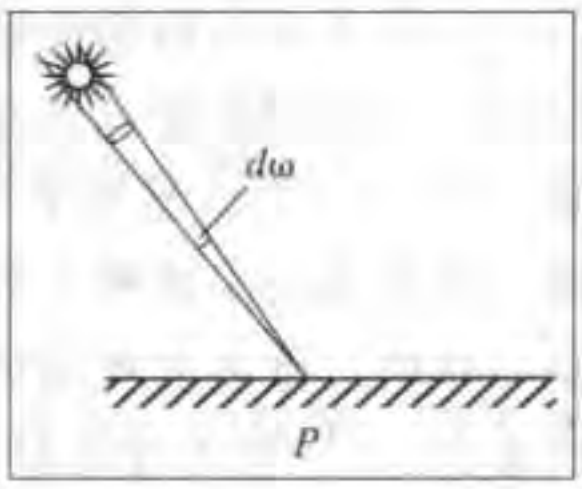


图 12.34 光源所对的立体角 $d\omega$

为了使用这些等式,我们必须要知道不同波长的光在材料中的折射率。通常得不到这个信息,我们得到的只是不同材料的"垂直反射率"的测量值,即不同波长光对应的 $F(0,\eta)$ 值。Cook 和 Torrance 根据下面的理论推导出折射率:如果我们知道 $\phi=0$ 时等式(4.28)中的 $F(\phi,\eta)$ 值,则 $c=1$,$g=\eta$(检验这一结果)。于是整个表达式可以化简为

$$F_0 = \frac{(\eta-1)^2}{(\eta+1)^2}$$

其中 F_0 表示垂直反射率 $F(0,\eta)$。由这个等式可以很容易地推导出 η 的表达式:

$$\eta = \frac{1+\sqrt{F_0}}{1-\sqrt{F_0}} \quad (12.33)$$

因此,给定 F_0 就能求出相应的折射率。图 12.35 给出了 4 种金属的 F_0 测量值【Touloukian】。

	红光的 F_0	绿光的 F_0	蓝光的 F_0
金	0.989	0.876	0.399
银	0.95	0.93	0.88
铜	0.755	0.49	0.095
铁	0.53	0.505	0.480

图 12.35 不同波长上测得的垂直反射率

Cook-Torrance 模型在计算上比 Phong 模型复杂得多，但是为了得到这种模型所提供的更逼真的效果，很多人宁可接受这些复杂的计算。如果要效果进一步逼真，可以采用更多波长的光来渲染场景。虽然只用红、绿、蓝 3 种颜色或多或少都可以与 CRT 显示器的颜色相匹配，但是如果要使计算机产生的图像达到终极的逼真度，那么它们始终受到显示设备的制约。

练习

练习 12.7.2 镜面反射对应的角是多少？

请证明图 12.30(b)中角 δ 必须满足 $\delta=(\theta-\phi)/2$ 才能使位于角 ϕ 的光源发出的光反射到达位于角 θ 处的观察者。

练习 12.7.3 微平面模型的光束方向图

由于入射光和反射光位于同一个平面内，因此等式(12.31)中采用的 G、D、F 等项都可以用入射角 θ 和观察角 ϕ 表示出来。

a. 证明等式(12.28)中的 G 可以表示为

$$G(\phi,\theta)=\min\left[1,\frac{\cos\left(\frac{\theta-\phi}{2}\right)\cos(\phi)}{\cos\left(\frac{\theta+\phi}{2}\right)},\frac{\cos\left(\frac{\theta-\phi}{2}\right)\cos(\theta)}{\cos\left(\frac{\theta+\phi}{2}\right)}\right]$$

b. 证明等式(12.31)中的 $spec$ 可以表示为

$$spec=\frac{F(\phi,\eta)D\left(\frac{\theta-\phi}{2}\right)G(\phi,\theta)}{\cos(\theta)}$$

c. 试画出当 ϕ、粗糙度 m 和折射率取不同值时，$spec$ 随 θ 的变化曲线图。

12.8 添加表面纹理

正如第 8 章所述，通过在各种表面上绘制纹理，可以使计算机生成的图像更加生动逼真。图 12.36 展示了一个带有几种纹理的场景的光线跟踪结果。

第 8 章中涉及的表面都是多边形，每个面用 OpenGL 渲染。对于每个面 F，它的每个顶点对应一组纹理坐标，OpenGL 用纹理图像内对应点绘制面片内每个像素。

我们想知道如何把纹理和光线跟踪过程结合起来。通常使用以下两类纹理。

(1) 图像纹理：将二维图像贴入物体的每个表面。

(2) 体纹理：物体是由某种材料块雕刻得到的。材质本身是有纹理的。光线跟踪显示出物体表面每个点的纹理颜色。

我们先考虑体纹理，因为它处理起来比较简单。我们研究在案例 8.5 中用 OpenGL 绘制多边形表面时，讨论了一种应用体纹理的方法，但是这个方法不是很自然。所幸的是，此方法可以很自然地合并到光线跟踪的框架内。

图 12.36 物体表面带有不同纹理的一个场景

12.8.1 体纹理

体纹理有时也称为三维纹理，它由 Perlin 和 Peachey[Perlin85, Peachey85]同时首先提出。我们把物体看作由纹理材料例如大理石或木头雕刻得到。纹理用函数 $texture(x, y, z)$表示，它在空间内每个点产生一个颜色值(r, g, b)。想象这种纹理是随着坐标变化的颜色或彩墨，我们在不同点(x, y, z)可以(用 X 射线视觉)观察到不同颜色。在此空间内定义一个具有某种形状的物体，将形状外所有材料裁剪掉之后，物体表面上的点(x, y, z)就会显现出来，并具有特定的纹理。

我们先详细解释实例 8.5 中提到的几个有趣的例子，然后介绍怎么在光线跟踪器中加入这类纹理。接下来，我们再介绍一些更丰富的材质，如木纹和大理石。

例 12.8.1 填满整个空间的三维棋盘

想象一个三维棋盘，由红色和黑色立方体在整个空间内交替堆积构成，将其中一个小方块的一个顶点放置在点(0, 0, 0)处，并调整其大小使其对角顶点位于点 $S=(S.x, S.y, S.z)$处。对于其他小方块也采用同样大小(宽为 $S.x$，高为 $S.y$，等等)，并将它们一个挨一个地排列在整个三维空间中。对于这样一个棋盘纹理，很容易写出其表达式：将 $x/S.x$、$y/S.y$ 和 $z/S.z$ 的整数部分相加，再将得到的和关于 2 取模：

$$\text{jump}(x,y,z) = ((\text{int})(x/S.x) + (\text{int})(y/S.y) + (\text{int})(z/S.z))\%2 \quad (12.34)$$

然后，若 $jump()=0$，则令 $texture(x, y, z)$返回黑色；若 $jump()=1$，则返回红色。

图 12.37 展示了由这种体纹理材料构成的一个通用球体和一个通用立方体。同时显示的是有类似纹理的一组相互贯穿的平面和一个逐渐变小的锥形柱面。纹理颜色用于设置表面上的漫反射系数，因此材料的颜色就是纹理的颜色。漫反射成分随着位置(相对于光源)的变化而变化，Phong 镜面反射成分就是光源的颜色。注意到此球体和立方体显然是由实体“小方块”构成的。对比一下，如果将一个二维棋盘图像粘贴到这些物体上，会是什么样？

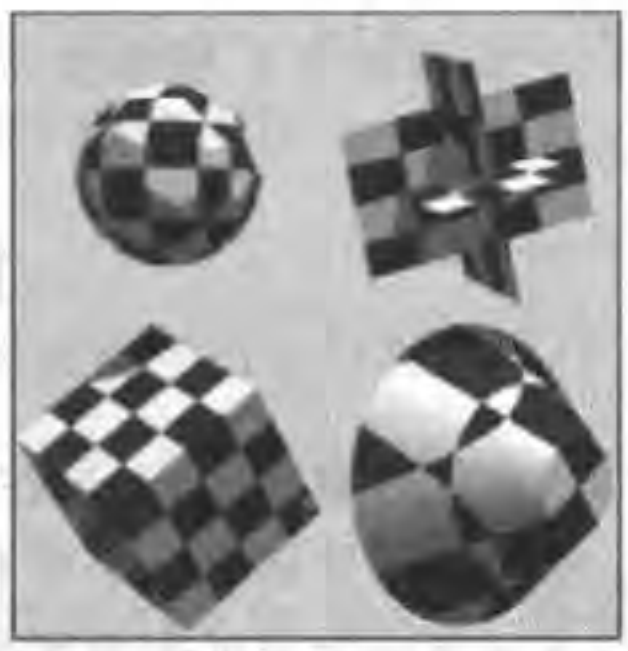


图 12.37 对于一些具有棋盘体纹理的物体的光线跟踪

例 12.8.2　一堆颜色渐变的立方体

也可以将一组同样的立方体堆积起来。在这些立方体内部移动时，可以看到平滑变化且覆盖整个光谱的颜色。立方体所有 8 个角都是黑色，而中心点(0.5，0.5，0.5)为白色，从而在中心和角落之间颜色平滑地变化。为了使红色成分沿着 x 方向上升为 1 再降回 0，可以取 $red=1-|2x-1|$。对于绿色成分和蓝色成分，可以分别得到沿 y 方向和 z 方向的类似表达式。把无限多个这样的立方体在空间中堆积在一起，利用 x、y、z 的小数部分(fraction)，最后可以得到

$$\begin{aligned}\text{texture}(x,y,z) = (&1-|2\text{fract}(x)-1|,\\ &1-|2\text{fract}(y)-1|,\quad 1-|2\text{fract}(z)-1|)\end{aligned} \tag{12.35}$$

其中 fract(x)为 1−(int)x 即 x 的小数部分。想象一下，一个由这种材料制成的半径为 100 的球体会是什么样？

对体纹理对象进行光线跟踪

一旦得到函数 texture()，那么基本上不需要其他任何信息就可以将体纹理添加到光线跟踪器中。纹理可以通过下述几种方法改变从表面某点发出的光线：

1. 光线可以设置成 texture()本身，看起来就像物体本身发出这种光。
2. 纹理可以调整环境光和漫反射系数(回忆第 8 章)，使得：

$$I = \text{texture}(x,y,z)(I_a\rho_a + I_d\rho_d lambert) + I_s\rho_s phong \tag{12.36}$$

其中 texture(x，y，z)在交点(x，y，z)计算。这是纹理最普遍的用途：根据 texture()的波动，物体表面不同点处的固有颜色看起来变得更暗或更亮。这里，高光反射的颜色和光源相同，不受纹理影响。这使得纹理物体表现出发光特性，仿佛它是由颜料制成的。

等式(12.36)中用到的交点(x，y，z)既可以是在通用坐标系下，也可以是在世界坐标系下。不过一般都将它置于通用坐标系中，在这种情况下，物体在旋转或移动到场景中最终位置的过程中，会一直“携带”着它的纹理。在动画中，物体从一帧旋转或移动到另一帧，而纹理却始终牢牢地附在这个物体表面。

另一方面，如果(x，y，z)位于世界坐标系下，则纹理在空间中是固定的。此时当物体在动画中旋转或移动时，纹理将会“扫过”物体，使物体在每个新位置看上去都像是由新材料雕刻出来一样。这个过程将产生非常有趣的视觉效果。

丰富多变的体纹理很容易创建，其中一些还能很真实地模拟实际材质，两个经典的例子就是木纹和大理石。

木纹纹理

不同颜色的同心圆柱形成了木材上的纹理，切开原木时就可以看到相应的圆环。随着点到中心轴的距离发生变化，函数值在两个数值之间来回变动。这种效果可以用以下模函数模拟：

$$\text{ring}(r) = ((\text{int})r)\%2$$

其中，此圆环的中心轴为 z 轴，半径为 $r=\sqrt{x^2+y^2}$。随着 r 由 0 开始增大，rings%2 的值在 0 和 1 之间来回变动。利用函数

$$\text{simple_wood}(x,y,z) = D + A \cdot rings(r/M)$$

可以使纹理在两个预设值 D 和 $D+A$ 之间来回变动。这里仍有 $r=\sqrt{x^2+y^2}$,这就产生了一系列厚度为 M、以 z 轴为中心的同心圆环。

如果对这些圆环进行摆动、扭曲或者旋转[Watt92],就会得到更有趣的结果。为了摆动这些圆环,我们添加了一个分量,它随着(关于 z 轴的)方位角 θ 的变化而变化:

$$\text{rings}\left(\frac{r}{M} + K\sin\left(\frac{\theta}{N}\right)\right)$$

这样一来,即使固定半径不变,这些圆环也会随着 θ 的改变而发生变化,此时围绕 z 轴观察,圆环会里外摇摆 N 次。注意我们在原函数内嵌套了一些别的函数来添加更多效果:*rings*()的参数不只是 r,而是 r/M 与一个正弦函数的和,而这个正弦函数本身的参数是一个角度关于 N 的商。Perlin[Perlin85]充分利用了这种复合函数,创造出许多有趣的视觉效果。利用下面的等式可以进一步在摆动的木纹纹理上增加一种"螺旋效果":

$$\text{rings}\left(\frac{r}{M} + K\sin\left(\frac{\theta}{N} + Bz\right)\right) \tag{12.37}$$

这样就使得正弦函数的相位随着 z 变化,在 z 变化时有效地旋转上述摆动。如果我们想倾斜木纹,使它以 z 轴之外的轴为中心,那么在计算 r 和 θ 之前就要先对它进行一次旋转。例如,我们取 r 为 $\sqrt{x'^2+y'^2}$,其中 $(x',y',z')=T(x,y,z)$ 对应于旋转变换 $T()$。

图 12.38 展示了一些从木纹材料"雕刻"出来的物体,这些木纹材料就是按上述方法定义的。

三维噪音和大理石纹理

像大理石这样的材料,它的纹理非常混乱,如图 12.39 所示。深色材料代表的湍流(turbulence)穿过大理石,并随机地产生螺旋和斑点,使得大理石看上去像是由一些急速搅拌的熔融材料形成的。

图 12.38 从木头雕刻出来的物体

图 12.39 从大理石雕刻出来的物体

我们可以模拟这种湍流。先构造一个"噪音"函数,让它在空间内每个点 (x, y, z) 处产生明显的随机值,然后,可以用一种容易控制的方法搅动这个"噪音场"来形成湍流现象。

噪音场本身很容易编程实现。想象空间每个整数位置(即在点 $(x, y, z)=(i, j, k)$ 处,其中 (i, j, k) 是任意整数 i、j、k 的组合)随机地给定一个值。这种点排列称为整数格

(interger lattice)。图 12.40(a)显示了一个二维的情况，图中对二维整数格中每个点都用 0～1 之间的噪音值标记。例如，点(1，2)的噪音值是 0.653，而点(3，1)是 0.129。图 12.40(b)显示了一个三维整数格。其中每个整数点都对应某个固定的噪音值，例如在(2，2，1)处对应的是 0.7341。

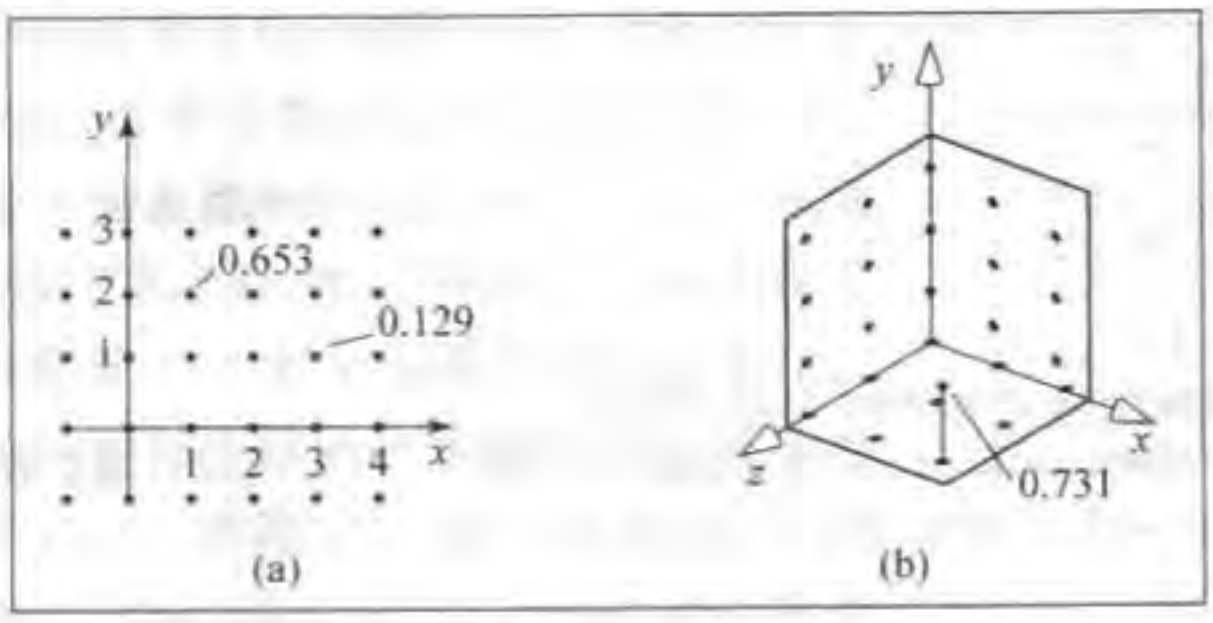


图 12.40　在每个三维整数格点定义噪音值

所有噪音值可以保存在一个巨大的数组 float N[10000][10000][1000]中，但是这样占用的内存之大令人无法接受。而如果只在每次需要时产生一个噪音值，问题就简单多了。为此需要构造一个函数，例如 float latticeNoise(int i, int j, int k)，当给定整数 i、j、k 来指定整数格位置时，这个函数就返回一个明显的随机值。这个函数必须足够高效并且是完全可重复的，也就是说对于给定的(i，j，k)，该函数返回的噪音值总是一样的。在 Texturing and Modeling【Ebert98】这篇优秀的论文中，几位作者介绍了几种适当的方法。这里重点介绍其中一种，这种方法由 Perlin 创立，并由 Peachey 进行了改进。

产生可重复的随机数

主要技巧是在初始化阶段建立一个固定的数组，例如 noiseTable[]，用来存放伪随机噪音值。研究发现，长度为 256 的数组已经足够大，因此就采用这个长度。

接下来，主函数 latticeNoise(i, j, k)只要用可重复的方法索引 noiseTable[]即可。为了确保当 i、j、k 变化时噪音值的变化没有规律，索引函数将(i，j，k)组合有效地散列编码为介于 0～255 之间的数。这只要用另一个数组 index[]就可以很容易地实现，该数组包含 0～255 中所有整数的一个随机排列。Peachey 建议定义下面两个宏：

```
#define PERM(x) index[(x) & 255]
#define INDEX(ix, iy, iz) PERM( (ix)+PERM( (iy)+PERM(iz)))
```

宏 PERM 取 x 的整数部分，再对它和数值 255 执行一个“按位与”操作，只保留 x 的低 8 位，再将此低 8 位值散列编码为 0～255 中某个数。因此，PERM 的值就是 index 数组中的一个数。宏 INDEX 利用 PERM 访问 index 数组 3 次，每次分别根据 ix、iy、iz 选择 index[]中的元素。注意到这个操作是可重复且高效的，期间还有大量的置乱操作。于是，函数 latticeNoise()可以简化为：

```
float latticeNoise(int i, int j, int k)
{
    return noiseTable[ INDEX(i, j, k)];
}
```

开发 Noise 类

如果把刚才提到的所有操作都封装到一个类 Noise 中,将会非常方便,这个类将用来生成大理石及其他类噪音纹理。图 12.41 给出了这个类的声明和定义(也可参见本书网页)。

```
class Noise{
public:
    Noise()                                          //构造函数
    {
        int i;
        index=new unsigned char[256];
        for(i=0; i<256; i++) index[i]=i;             //用索引填充数组
        for(i=0; i<256; i++)                         //置乱
        {
            int which=rand()%256;                    //选择数组中随机位置
            unsigned char tmp=index[which];          //将它们互相交换
            index[which]=index[i];
            index[i]=tmp;
        }
        noiseTable=new float[256];
        for(i=0; i<256; i++) noiseTable[i]=rand()/32767.99;
    }    //构造函数结束
    float noise(float x, float y, float z);
    float noise(float scale, Point3&p);
    float turbulence(float x, float y, float z);
    float marble(float x, float y, float z);
    float marble(float strength, Point3& p);
private:
    float * noiseTable;                              //噪音值数组
    unsigned char * index;                           //伪随机索引
    float mySpline(float x);                         //用于生成大理石纹理
    float latticeNoise(int i, int j, int k)
    {    //返回整数格上的噪音值
        #define PERM(x) index[(x) & 255]
        #define INDEX(ix, iy, iz) PERM( (ix)+PERM( (iy)+PERM(iz)))
        return noiseTable[INDEX(i, j, k)];
    }
};
```

图 12.41 生成可重复的噪音值

这个类中最有趣的方法是 marble(),它返回一个 0~1 之间的亮度值,这个亮度值的大小视位置而定,可以用来模拟大理石上的明暗波纹。稍后研究这个算法的细节。这个类可以用来生成一种带绿色的大理石纹理,只要在光线跟踪的开始阶段通过命令

```
Noise n;                    //构造和创建一个噪音对象
```

构造一个噪音对象,然后再用 n.Marble(x, y, z)获得每个点(x, y, z)处的 texture(x, y, z)值即可。

方法 marble()使用了 noise()和 turbulence()(随后将对它们做简短介绍)以及辅助函数 latticeNoise()。这个类的构造函数创建并填充数组 noiseTable[]和 index[]。利用 C 标准库函数 rand()生成随机值并将其写入 noiseTable[],调整这些值的大小使之介于 0.0 和 1.0 之间。先按顺序将 0～255 中的数赋给数组 index[],然后再将该数组中每个元素依次和另外一个随机选中的元素交换,这样一来,元素的顺序就被"打乱"了。

函数 latticeNoise()在整数格点上产生随机值,我们希望能利用它来构造一个函数 noise(x, y, z),可以在整格点之间(也就是空间任意点处)生成随机值。此外我们还希望这个噪音函数随着 x、y 和 z 的变化而平缓变化。

在这些整数格之间进行简单的线性插值可以得到令人满意的结果[①]。图 12.42 显示了二维下的插值,三维的情况与之类似。这里,我们希望利用 4 个相邻的整数格点处的噪音值,来计算点(x, y)=(0.6, 1.4)处的噪音。首先沿 y=1 和 y=2 对 x 进行插值,得到以下值[②]

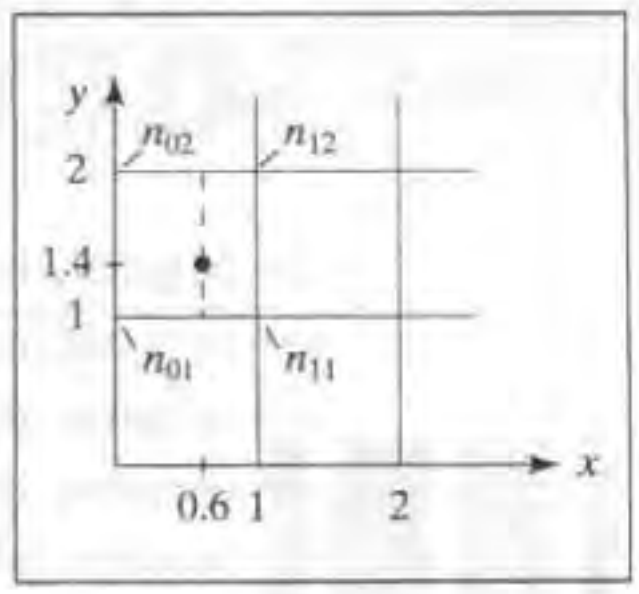


图 12.42 二维下噪音值的线性插值

$$n(0.6,1) = \mathrm{lerp}(0.6, n_{01}, n_{11})$$

$$n(0.6,2) = \mathrm{lerp}(0.6, n_{02}, n_{12})$$

然后对这些值关于 y 插值,得到

$$n(0.6,1.4) = \mathrm{lerp}(0.4, n(0.6,1), n(0.6,2))$$

图 12.43 给出了在三维中实现 noise()函数功能的一种方法。它利用一个额外的参数 scale 来对三维点(x, y, z)进行缩放;在产生湍流时,这个方法是很有用的。首先,将经过缩放后的点在 x、y 和 z 方向上各偏移 1000,使所有分量都为正(因为格点值是随机的,所以这一偏移不会改变所产生的噪音的统计性质)。然后,在这个点的周围 8 个整数格点处生成噪音值。最后,再用 7 个 lerp 函数插值得到噪音值。

```
float Noise::noise(float scale, Point3& p)
{     //对格点噪音进行线性插值
    #define lerp(f, A, B)A+f*(B-A)
    float d[2][2][2];
    Point3 pp;
    pp.x=p.x*scale+10000;          //偏移量不应为负值
    pp.y=p.y*scale+10000;
    pp.z=p.z*scale+10000;
    long ix=(long)pp.x;
    long iy=(long)pp.y;
    long iz=(long)pp.z;
    float tx, ty, tz, x0, x1, x2, x3, y0, y1;
    tx=pp.x-ix;                    //分量
```

图 12.43 在任意点 p 处生成噪音的函数

① Perlin 和 peachey 认为三次插值得到的结果更真实。

② 回忆第 5 章介绍的 lerp()函数:lerp(f, A, B)=A+(B−A)f,f 为小数,表示一个介于 A 和 B 之间的值。

```
    ty=pp.y-iy;
    tz=pp.z-iz;
    float mtx=1.0-tx, mty=1.0-ty, mtz=1.0-tz;

    for(int k=0; k<=1; k++)     //获取8个格点处的噪音值
    for(int j=0; j<=1; j++)
    for(int i=0; i<=1; i++)
        d[k][j][i]=latticeNoise(ix+i, iy+j, iz+k);

    x0=lerp(tx, d[0][0][0], d[0][0][1]);
    x1=lerp(tx, d[0][1][0], d[0][1][1]);
    x2=lerp(tx, d[1][0][0], d[1][0][1]);
    x3=lerp(tx, d[1][1][0], d[1][1][1]);
    y0=lerp(ty, x0, x1);
    y1=lerp(ty, x2, x3);
    return lerp(tz, y0, y1);
}
```

图 12.43(续)

图 12.44(a)显示了函数 noise(20, x, y, 0)对应的图,黑色代表 0.0,白色代表 1.0。图中 x 和 y 的值都是从 −1 变到 1。由于随机数生成的特殊过程,因此噪音场显现出了某些结构,但还不是十分清晰。

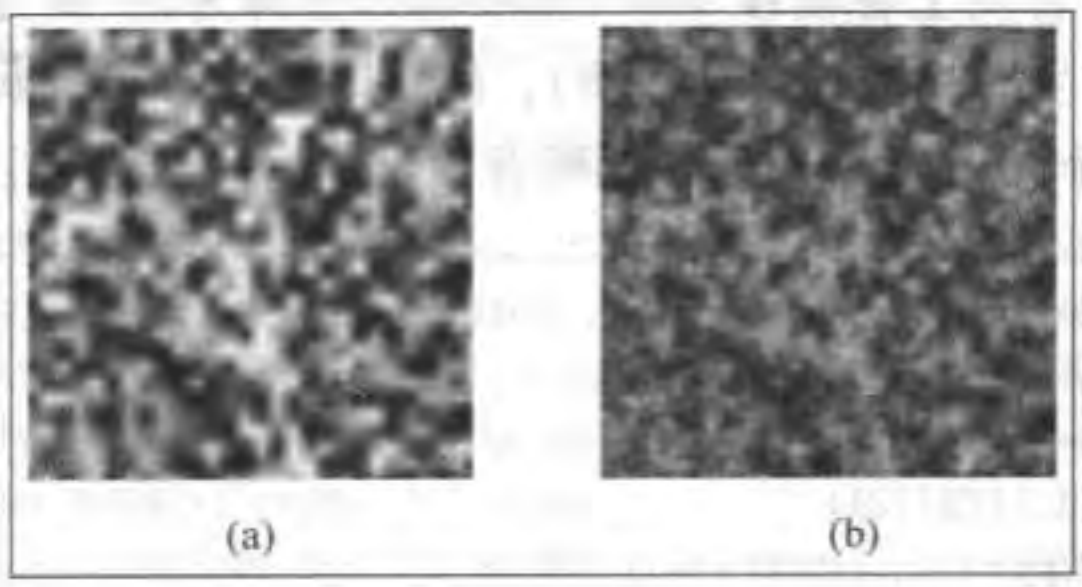


图 12.44

(a) noise()的样图;(b) turb()的样图

扰动

Perlin【Perlin85,Ebert98】设计了一种方法用来产生比前述结果更加有趣的噪音。这种方法的思想是将几个噪音分量叠加在一起。这些分量包括:随着空间中位置的移动函数值发生缓慢波动的分量、波动速度为前述速度 2 倍的分量、波动速度为前述速度 4 倍的分量等。变化速度越快的分量,它的强度越小。函数

$$\text{turb}(s,x,y,z) = \frac{1}{2}\text{noise}(s,x,y,z) + \frac{1}{4}\text{noise}(2s,x,y,z) + \frac{1}{8}\text{noise}(4s,x,y,z) \tag{12.38}$$

中添加了 3 个这样的分量，其中每个分量的强度依次是前一个的一半，而变化速度则依次是前一个的两倍。参数 s 的作用和在 noise()中一样，仍然用于调整距离。图 12.45 提出了一个在二维中观察 turb()的波动情况的方法。想象用 noise(1, x, y, 0)覆盖 xy 平面，对于每个点 $P=(x,y)$，turb(1, x, y, 0)将平面上(x, y)、($2x$, $2y$)和($4x$, $4y$)三个点上的噪音值相加。在 P 点的临近点 P' 处，noise(1,x',y',0)和 noise(1,x,y,0)的值非常接近，但 noise(2,x',y',0)与 noise(2,x,y,0)的值之间差距就比较大，而 noise(4,x',y',0)和 noise(4,x,y,0)差距就更大了。第一个噪音分量表现出来的特征只有第二个分量的一半，第三个分量的四分之一。

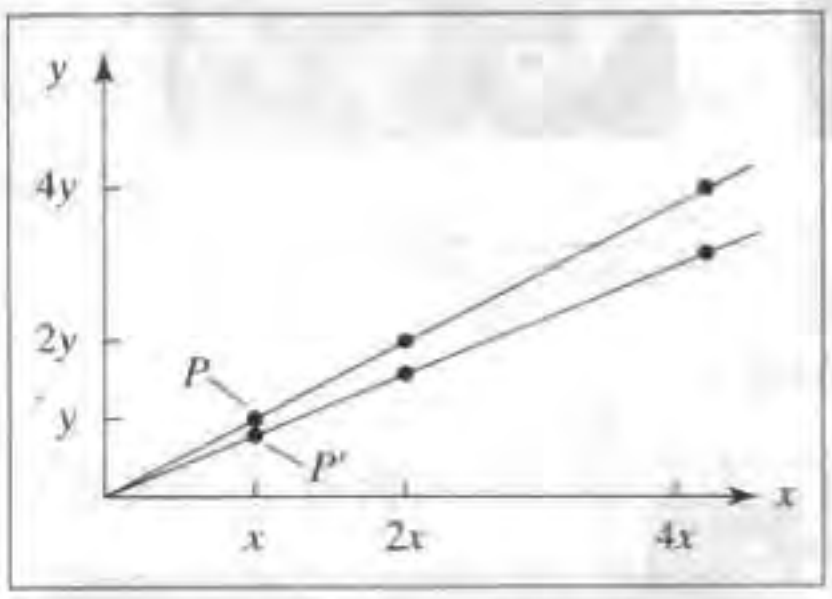


图 12.45　turb()函数

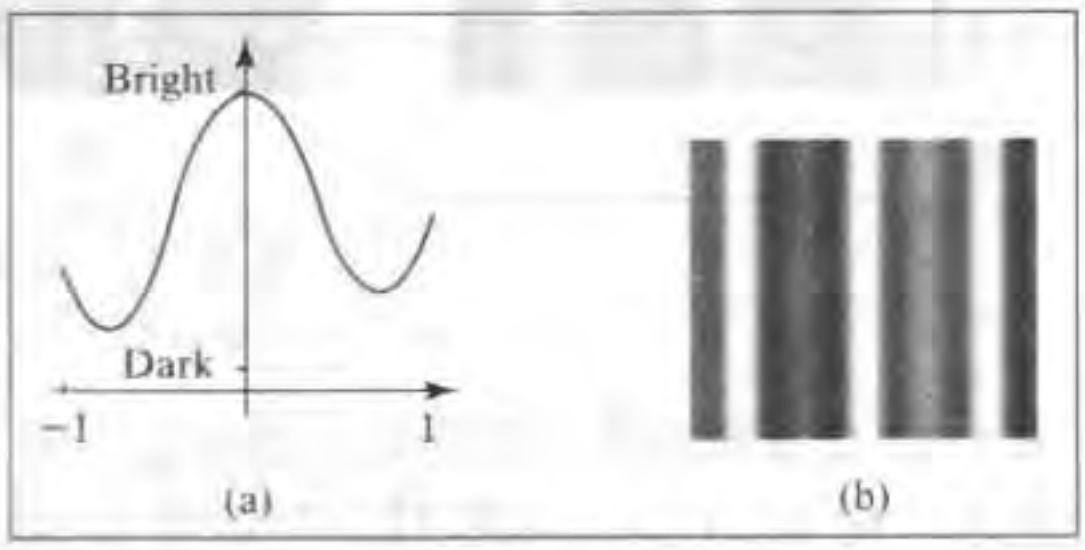


图 12.46　构造大理石纹理

(a) 样条曲线；(b) 未经扰动的大理石纹理

在任何细节尺度上，每个分量的大小都是前一分量的 1/2，而它的频率(波动速度)则是前一分量的两倍。这就形成了附录 5 中介绍的**自相似性**，实际上，在一定条件下，turb()可以近似得到所谓的“1/f 噪音”，各种自然图形中都会表现出这种噪音。图 12.44(b)显示了 $M=3$ 时由图 12.46(a)的 noise()场产生的 turb()函数的图，其中明显有更多细节，而波动则更平缓，更像云雾。类似 turb()这样的函数所产生的扰动可以用来改变外形或纹理的某些性质，使其显得更有真实感，这在后面对大理石纹理的讨论中可以看到。

大理石纹理

大理石材料呈现出明暗相间的脉络，这些脉络有一定规律，但同时也呈现出杂乱而不规则性。根据 Watt【Watt92】的观点，通过给脉络一个平滑的波动(例如在 z 方向上)，再用 turb()函数去扰动它，就可以构造出一种类似大理石的三维纹理。我们先考虑一种在 x 和 y 方向上都是常数，只在 z 方向上发生渐变的纹理：

```
marble(x, y, z)=undulate(sin(z))
```

其中 undulate()是图 12.46(a)所示的样条形函数【Watt92】，当参数从 −1 变到 1 时，这个函数将在某个暗值和某个明值之间变化(本节后面的练习将讨论此样条函数)。把 sin(z)作为 undulate()的参数，就产生了一个 z 方向的周期性“波纹”，这个“波纹”在样条曲线上来回移动，每个周期移动一次，于是就产生了图 12.46(b)所示的亮度波动。由于大理石中颜色的纵向脉络过于规则(这是很自然的)，因此就要调整 sin()的参数使它带有某种波动：

$$\text{marble}(x,y,z) = \text{undulate}(\sin(z + A \cdot \text{turb}(s,x,y,z)))$$

sin()函数的相位在大理石中的不同位置有不同偏移，这样就产生了更逼真的脉络。参

数 s 使不同点处的扰动变得更快或更慢;而参数 A 则改变扰动总量的大小。Watt 证明如果用简单的直线代替 undulate(),所产生的效果就会很差。

图 12.47 显示了一个立方体表面上的大理石纹理。相应的函数是:

$$g = \text{undulate}(\sin(2\pi z + A \times \text{turb}(5, x, y, z)))$$

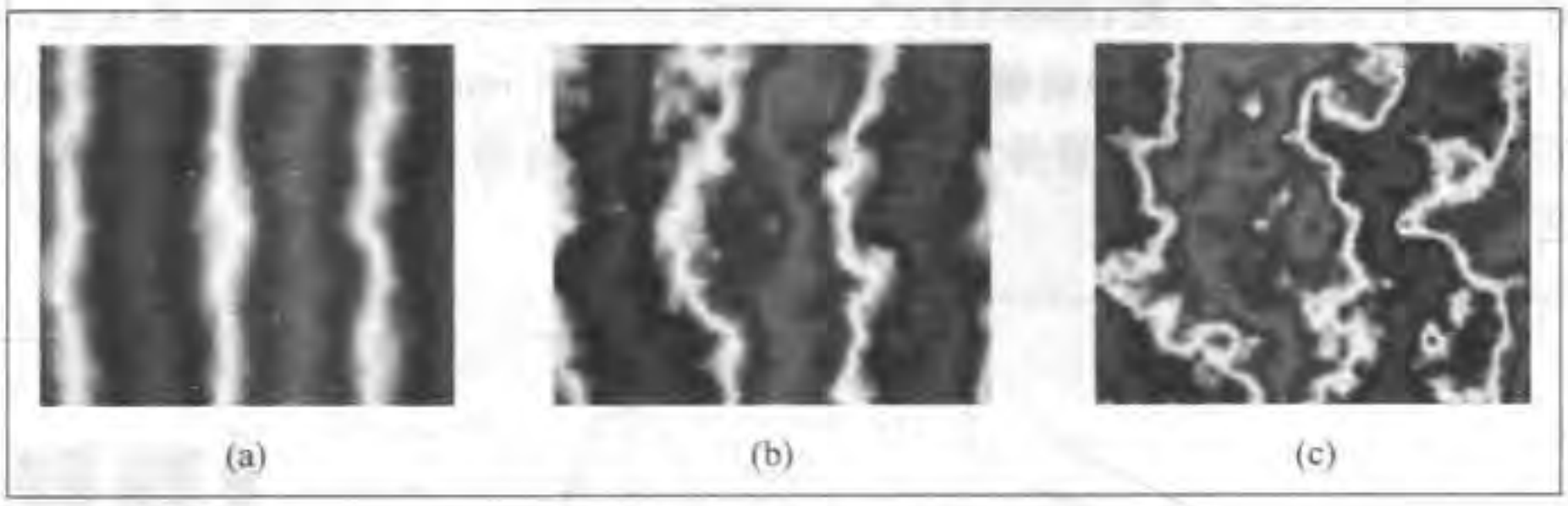


图 12.47 大理石纹理

(a) A=1; (b) A=3; (c) A=6

其中,z 自右向左从 0 变到 1,y 指向上。图 12.47(a)中振幅 $A=1$,因此表现出来的扰动非常小。图 12.47(b)和图 12.47(c)中,A 的值分别是 3 和 6,此时扰动也相应变大了。

可以将 marble()的值作为等式(12.36)的反射系数,调整不同光线命中物体表面不同点后返回的光的总量。最直接的方法是,扩展 marble(),使其分别返回全色光线跟踪器的红、绿、蓝分量。

图 12.48 展示了一个包含大量大理石物体的场景内的光线跟踪,效果相当令人满意。

图 12.48 由大理石物体组成的一个场景

12.8.2 在表面上粘贴图像

第 8 章分析了如何利用 OpenGL 将图像粘贴到多边形表面上。现在我们介绍如何利用光线跟踪器将图像粘贴到任意曲面上。实现此操作的过程很简单,结果也非常好,但是执行时间要比使用 OpenGL 长。因为除了其他额外负荷,每个像素都要单独计算,无法利用扫描行的连贯性。

和第 8 章一样,假设已定义一个二维纹理函数 texture(u, v),其中 u 和 v 分别从 0 变到 1,该函数在每个点(u, v)都产生一个亮度值或颜色。函数 texture(u, v)可以是一种过程纹理(如棋盘或 Mandelbrot 集),也可以是保存在像素图中的图像纹理。假设像素图是一个存储像素值的 $N \times M$ 数组,记为 txtr[][]。那么给定 u 和 v 在 0~1 之间的值后,就可以用下述语句在 txtr 中检索相应的像素:

```
txtr[(int)(u/N)][(int)(v/M)]
```

在一般物体上直接粘贴纹理,比在经过场景坐标变换后的物体上粘贴纹理要简单得多。设计者将纹理坐标和一般物体的位置坐标关联起来,使得当把物体变换到场景中时,纹理也

正确地(并以适当的纵横比)显示在变换后的物体上。

用纹理覆盖表面

我们需要一种方法,把一般物体表面上的点(x, y, z)和纹理坐标(u, v)关联起来。显然对于不同的物体形状,需要使用不同的映射。

例 12.8.3　正方形和平面的纹理

通用平面即 xy 平面,通用正方形就位于此平面内。因此 texture(u, v)的图像平面和通用正方形或平面之间存在一个很自然的关联。如图 12.49 所示,设计者在平面上选择一个窗口($left$, top, $right$, $bottom$),当光线与此窗口相交时,可以很容易计算出应使用纹理中哪个点(u, v)。

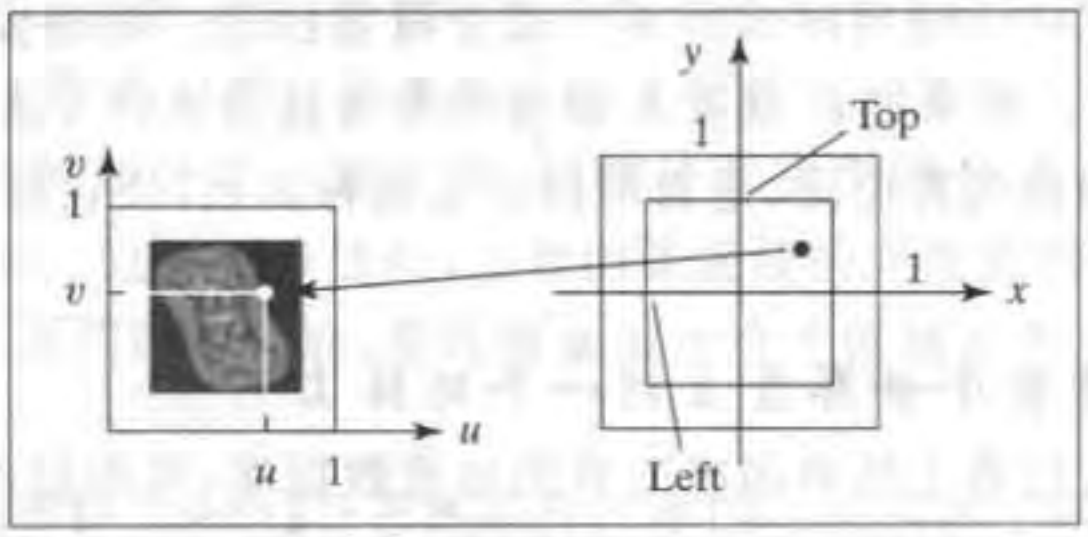


图 12.49　将纹理映射到正方形或平面上

具体地说,给定通用坐标系下的交点(x, y, 0),则相应的纹理坐标为

$$u = \frac{x - left}{right - left}, \quad v = \frac{y - bottom}{top - bottom} \tag{12.39}$$

设计者还必须确定如何处理落在窗口之外的交点。当这种情况出现时,一个常用的方法就是,取为某个预先指定的纹理颜色。

例 12.8.4　柱面上的纹理

在通用圆柱面上覆盖纹理同样简单。回忆 8.5.5 节中关于如何把纹理覆盖到圆柱形曲面上的讨论。通用锥形柱体如图 12.50 所示,在它的表面上有一个指定的"窗口",窗口的方位角从 a_1 扩展到 a_2,在 z 轴方向从 z_1 延伸到 z_2。当一条光线在(x, y, z)处与柱体相交时,容易计算出方位角为,然后即可用下式计算纹理坐标(u, v):

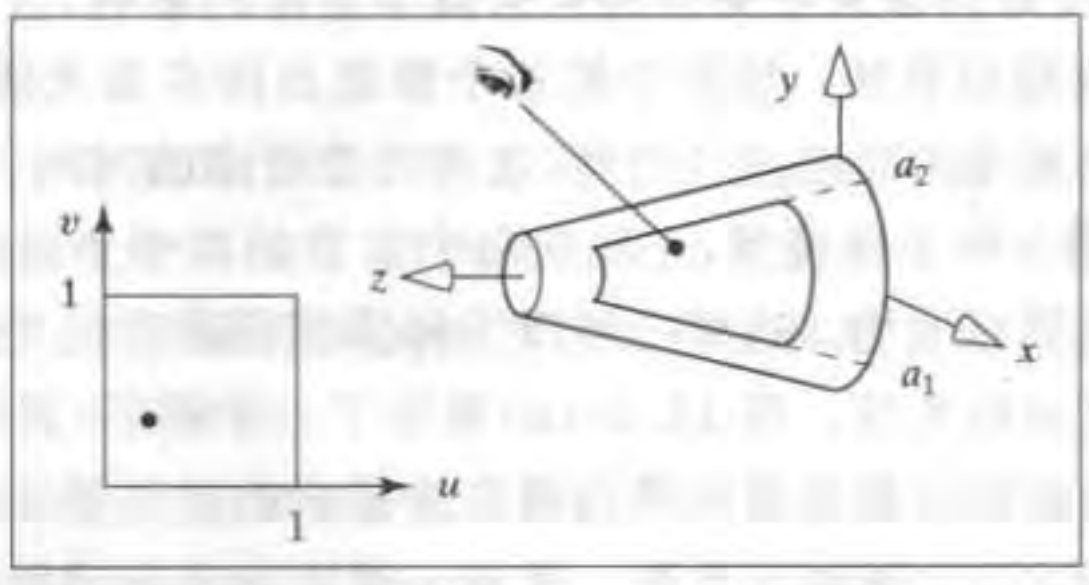


图 12.50　在柱体上覆盖纹理

$$u = \frac{\theta - a_1}{a_2 - a_1}, \quad v = \frac{z - z_1}{z_2 - z_1} \tag{12.40}$$

若交点位于指定窗口之外,设计者须选择用于该处的纹理颜色。注意,如果锥体缩小很快,则映射的纹理图像将发生扭曲。接下来的练习将解决如何将纹理粘贴到柱体的顶部或底部,另外还提到了在球体或圆锥体上覆盖纹理的问题。

练习

练习 12.8.1 将纹理覆盖到柱体的顶部或底部

试详细说明如何将纹理覆盖到一个柱体的顶部或底部。将位图的一部分粘贴上去,使之完全覆盖顶部或底部。

练习 12.8.2 在球面上覆盖纹理

试详细说明如何在一个通用球体的某个部分覆盖纹理。纹理窗口的方位角从 a_1 扩展到 a_2,纬度从 l_1 变到 l_2。如果把 l_1 或者 l_2 设置为非常接近 $\pm 90°$,试分析位图发生扭曲的程度如何。如果用纹理函数代替位图,是否可以定义某种函数,使它即使在接近球的两极也不会发生扭曲?

练习 12.8.3 将整个世界覆盖到一个球体上

假设你有一个关于世界上所有国家边界的地理数据库,其中包含了大量的折线,它们的端点用(经度,纬度)表示。利用此数据库是否可以将世界地图准确地覆盖到一个通用球体上?如果可以,试说明如何实现。

练习 12.8.4 在圆锥上覆盖纹理

试详细说明如何在一个通用圆锥体的侧面的某个部分覆盖纹理。对窗口参数的自然解释是什么?讨论位图在各种情况下会发生什么样的扭曲。说明如何将纹理映射到圆锥的底部。

12.9 光线跟踪算法中的反走样技术

光线跟踪算法是一个点采样的过程——沿着每条单独的光线将场景离散开来。因此,光线跟踪的图像经常出现走样问题,这并不奇怪。

前面学过,可以通过对场景采样更多的点来减少走样的影响,这个通常称为**超采样**。这个技术也可以用于光线跟踪算法:场景中的每个像素点用多条光线来跟踪,将每条光线返回值的平均作为当前像素点的亮度值。当然,这将花费更多的时间。

图 12.51(a)给出了一种采样模型,光线从每个像素的四个角通过。像素的最终颜色取为它四个角的光线颜色值的平均。这样一种反走样算法容易实现并且时间花费较少。超采样可以为每个像素取更多的光线。图 12.51(b)展示了一个例子,其中每个像素采样九条光线。最后将这九条光线返回的颜色值的平均值作为像素的颜色值。

还有几个更有效的方法也被提了出来。Whitted【Whitted80】提出了一种自适应的处理方法,在更需要反走样处理的区域即图像颜色值改变迅速的区域插入更多的光线。

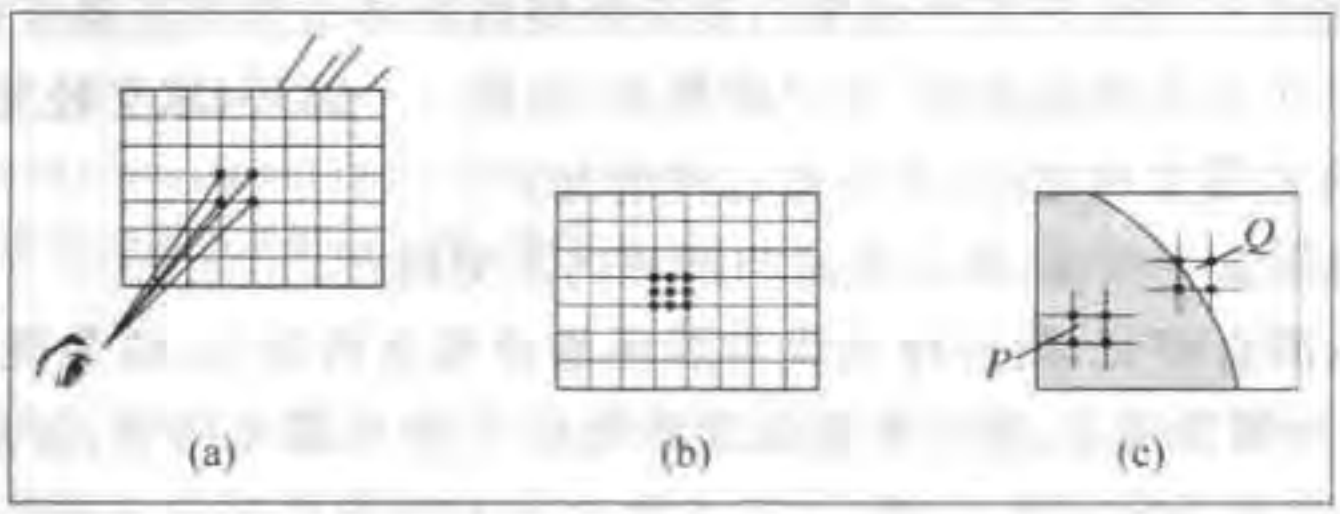


图 12.51　反走样中的超采样技术

在 Whitted 的方法中，从像素的四个角发射四条光线，然后对四条光线的亮度进行平均，这个平均值将与四个单独的亮度值进行比较。如果有一个角上的光线亮度值与平均值相差很大，那么这个像素将被一分为四，接着对亮度相差大的这四分之一块的四个角再采样光线。参见图 12.51(c)。像素 P 的四个角的光线返回了几乎相同的值，这是因为在这个区域，场景的变化并不大。像素 Q 的一条光线返回了与其他三条相差很大的亮度值。因此三条新的光线将从 Q 左下的四分之一块射出，同样计算这个四分之一块的四条光线的平均值，并与每条光线做比较。这样一直迭代，直到四条光线的亮度值都很接近或者迭代到了用户指定的次数为止。在迭代结束之后，最终像素值由这个区域所有光线值的一个加权平均来决定。

另外一种技术是基于【Cook84】分布式采样技术的。它采用了随机采样的形式，这个在前面第 9 章的纹理反走样里面提到过。对每个像素做随机的光线采样，然后对这些光线返回的亮度值做平均。例如，将像素细分成规则的 4 乘 4 的网格。但并不是对这些规则网格点进行采样，而是对抖动的网格点进行光线采样。通过对图像加入噪音的度量来抖动采样点，这些加入的噪音比走样的误差对人眼的视觉影响要小得多。采用抖动采样点的小网格采样比不采用它的采样效果好很多。

图 12.52　台球绘制中的抖动效果

图 12.52 是用随机采样的光线跟踪技术绘制一个台球的场景。特别注意图中的走样现象，特别是在球的边缘。

12.10　包围盒技术

光线跟踪算法是一个重复迭代的过程，它对大量的光线一遍又一遍地进行同样的操作。每条光线要对场景中的每个物体相交，这样就涉及到大量求交的运算。当考虑到阴影、反射和折射时，情况变得更糟。这时就需要能减少待处理物体数量的技术。包围盒技术的运用极大地提高了光线跟踪算法的速度。

一个物体的包围盒是能把这个物体包裹在内的一种形状。它通过快速排除与被包围物体不相交的光线来提高光线跟踪的速度。如果一条光线与某个包围盒不相交，那么它必然与此包围盒内的物体不相交。如果包围盒的形状是简单形状，那么判别这样一个简单形状

与一条光线是否相交比判别与它所包围的复杂外型的物体是否相交要节省很多时间。

图 12.53 是一个长方体包围盒(求交较简单)包围一个圆环(求交较复杂)的例子。当判断一条光线是否与包围盒相交时,有下面三种情况:

(1) 光线与包围盒不相交,那么光线与圆环的求交跳过。

(2) 光线与包围盒相交,则进行光线与圆环是否相交的测试,结果光线与圆环不相交。(这种情况也被称为错误警报;前一半测试预示光线可能与圆环相交,但是事实上它们并不相交)

(3) 光线与包围盒相交,进行光线与圆环是否相交的测试,光线确实与圆环相交。

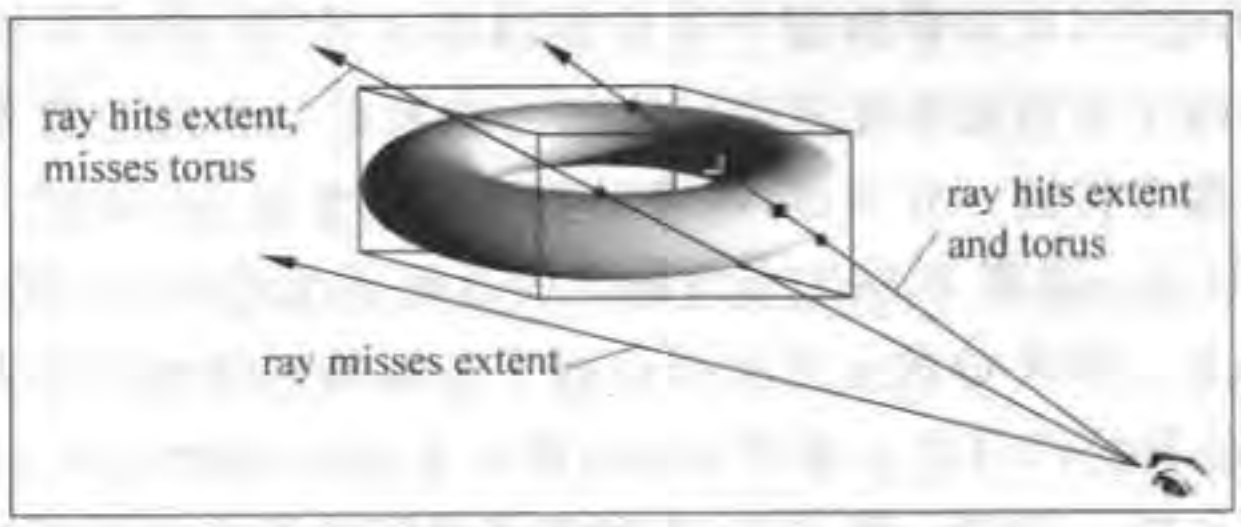


图 12.53 圆环和它的长方体包围盒

如果对物体进行求交测试比对包围盒进行求交测试花费大得多,用包围盒技术省去对物体求交要比维护这样一个求交列表花费少很多。

补充:包围盒技术对时间节省的一点分析

假设测试一条光线与包围盒是否相交花费 T 个时间单位,而一条光线与圆环的求交测试花费 mT 个时间单位。假设绘制这个场景需要测试 N 条光线,其中有比率 f 部分是与包围盒相交的。绘制需要计算 N 次光线与包围盒的交,花费 NT,fN 次光线与圆环的交,花费 $fNmT$。因此总的时间花费是 $NT(1+fm)$。另一方面,如果没有使用包围盒技术,所有的 N 条光线都要跟圆环做求交测试,总的花费则是 mNT。于是得到如下比率:

$$\text{speed ratio} = \frac{\text{cos t without extents}}{\text{cos t with extents}} = \frac{m}{1+f\cdot m} \tag{12.41}$$

例如,如果 $m=20$,$f=1/40$(在一个典型的场景中,每个物体只占整个场景的一个很小部分)。这个比率大概是 13,因此如果禁止包围盒求交,那么总的时间将延长至使用包围盒技术的 13 倍之多。

从上面的分析注意到,f 应该尽可能的小。也就是说,每个包围盒应该越小越好,这样就会有更少的光线与其相交。这跟人们的直觉很相似,包围盒应该尽量紧地包裹住原始的物体。

在光线跟踪算法的哪个部分插入包围盒求交测试比较合适呢? Arvo 和 Kirk【Arvo89】对所有的可能做了个精彩的综述,其中包括一个包围盒包裹一组物体的层次包围盒技术。这里集中考察改进基本的 hit()函数的方法,这样每个物体都将处理到。回忆前面图 12.4 中提到的基本程序 getFirstHit(),当前光线必须对场景中所有物体求交,对物体使用 getFirstHit()方法:

```
for(each object, obj)
{
    if(!obj->hit(ray, inter))continue;
      比较本次求交与当前最好的求交,等等
}
```

应该把包围盒求交测试放到 hit()里面来加速每次单独的求交过程。如果可以在整个求交计算之前快速判断光线与当前物体不会相交,那么整个光线跟踪算法的速度将大幅提高。将包围盒求交测试放到 hit()里面的一个好处是可以调整求交算法适应特殊的几何形状。

有很多方法可以在光线跟踪算法中创建和应用包围盒技术。一些很容易实现,另一些需要更多的程序量但有很好的效果。这里考察少数几个简单的方法,并说明它们是如何符合光线跟踪算法的逻辑思维的。

12.10.1　长方体和球体包围盒

包围盒技术中最常用的两种形状是长方体和球体:

- 球体包围盒:一个完全包裹物体的球体;由表达式(C,r)指定,球心为 C,半径为 r。
- 长方体包围盒:边与坐标轴平行的规则平行六面体。有六个参数指定:$(left,top,right,bottom,front,back)$。

判断一条光线和这些形状是否相交的算法速度是很快。对于圆球,求交有一个对应的二次等式,应该判断其表达式是否为正(参见图 12.15)。而对于边与坐标轴平行的长方体,要判断光线与六个平面是否相交(或者更少,如果足够幸运),然后判断区间(t_{in},t_{out})是否为零(参见图 12.23)。与每个平面的求交是相当快的,因为不需要点积运算。

进行包围盒测试的坐标空间可分为两类,世界坐标和通用坐标。通过坐标变换可以将每个物体放置到场景中。为求光线与物体的交,首先将光线通过坐标逆变换转换到通用坐标中,再求通用坐标中的光线和物体的交。但在这之前,先在世界坐标中对物体进行包围盒测试。如果光线与包围盒不相交,光线与物体的求交就不需要进行,这样就节省了坐标变换的花费。通常也可以在通用坐标空间对物体进行包围盒求交测试,如果光线与包围盒不相交,则可以避免光线与物体的求交运算。

世界坐标中的包围盒可能并不是很适合当前的物体。图 12.54 展示了场景中进行光线跟踪绘制的一个圆锥体。简单起见,图 12.54(b)用二维显示了圆锥体被长方体和圆球体包裹的情形。如果圆锥体细而长,圆球包围盒就会不适合,利用它来进行求交测试会出现很多

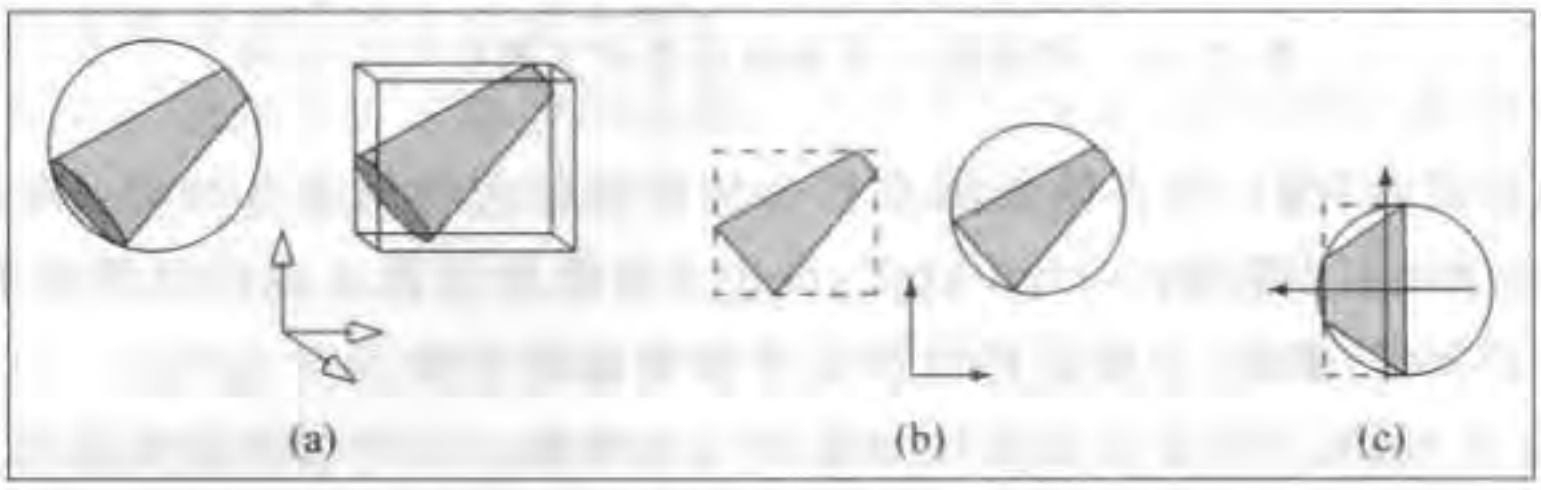


图 12.54　世界坐标和一般坐标中的包围盒

的错误警报(光线与包围盒相交但不与物体相交)。而长方体包围盒却能有效地包裹这样细且长的圆锥体,除非圆锥体旋转之后与坐标轴不平行。后面的联系请你对不同情况的错误警报做一个估计。

通用坐标中的包围盒通常会更适合物体。图12.54(c)展示了通用坐标的圆锥体与它的圆球包围盒和长方体包围盒。圆球包围盒跟物体很适合,长方体包围盒更紧致地包裹了物体。

包围盒求交测试是在世界坐标中进行还是在通用坐标中进行是有一个平衡的。在世界坐标中求交速度比较快,因为它不需要对光线进行坐标逆变换,但是通常包围盒对物体的包裹不够紧致。在通用坐标中求交需要计算通用坐标中的光线,但它的包围盒通常更适合物体。很难说前者与后者哪个更好。同样圆球包围盒和长方体包围盒的好坏也是无法判断。人们通常是在样本场景中对一个或多个方法进行时间测试,然后决定在他们的光线跟踪算法中采用何种方法组合。对一个拥有100个随机放置的圆锥体的场景进行测试的结果是,在世界坐标空间中使用圆球体包围盒会节省27%的光线跟踪时间,使用其他任何组合的时间花费都会比这个高。但是后面会看到,当物体的形状越来越多的时候,结果会很不一样。

图12.55是将包围盒求交测试集成到圆锥体情形的hit()里的伪代码。这里的包围盒是圆球体,并且分别在世界坐标系和通用坐标系中进行测试。在实际的算法中,只需要用到一种测试。函数rayHitsSphereExtent()用来判断光线与圆球包围盒是否相交,如果不相交,程序从这里提前退出hit()函数。这个函数首先被用来与世界坐标中的圆球包围盒worldSphereExtent()求交。如果光线与其相交,对光线做一个坐标变换。函数被第二次调用,用来判断光线是否与通用坐标中的圆球包围盒相交。同样,如果不相交,则退出求交函数,光线与圆锥体的求交将被跳过。

```
bool TaperedCylinder::hit(Ray &r, Intersection &inter)
{
    if(!rayHitsSphereExtent(r, worldSphereExtent)) return false;

    Ray genRay;                          //一般坐标中的光线
    xfrmRay(genRay, invTransf, r);       //花费高的

    if(!rayHitsSphereExtent(genRay, genSphereExtent))
return false;
…光线与包围盒相交;必须做花费很高的光线与景物的求交测试…
}
```

图12.55 圆锥体的圆球包围盒求交测试hit()函数

若先前经验告诉我们,长方体包围盒将会比圆球体包围盒更加有效,那么上面的函数rayHitsSphereExtent()将用rayHitsBoxExtent()函数来代替。也可以将两种包围盒的测试函数都放入到hit()里面,虽然这样会影响求交测试的速度。

这样,为hit()函数选择测试的方法就显得尤为重要。有些选择是明显的。对于一个球体的物体来说,没有必要对其进行通用坐标系下的球体包围盒求交测试(为什么?),类似地,

没有必要对长方体物体进行通用坐标系下的长方体包围盒求交测试。平面没有包围盒，因此函数 Plane::hit()不包含函数过程。为正方形物体建立包围盒是有意义的，而直接对这样的物体求交也很简单以至于用包围盒测试很少或者不会带来效率上的提高。另一方面，对于一个网格物体，它由大量的平面组成，对它直接求交花费很大，所以包围盒求交测试可以带来效率上的巨大提高。

如何实现包围盒求交测试

不同几何形状的物体的包围盒是如何计算的？每个包围盒求交测试是如何执行的？自然的方法是将包围盒的信息储存到它所对应的物体里，因此在 GeoObj()类里加入一些域的信息(参见本书的相关网页)。前面提到的四种包围盒求交方法经常会被用到，可以为每种方法加入一些域信息：

```
SphereInfo genSphereExtent, worldSphereExtent;
Cuboid genBoxExtent, worldBoxExten;
```

类 SphereInfo(参见本书的相关网页)记录了球体的两个信息：center 记录球体包围盒的球心位置，radSq 记录了半径的平方。可以只记录球体的半径信息，但在求交的时候只会用到半径的平方。类 Cuboid 包含了 6 个信息：left、top、right、bottom、front、back。长方体包围盒沿 x 轴从 left 延伸到 right，沿 y 轴从 bottom 延伸到 top，沿 z 轴从 front 延伸到 back。

在光线跟踪之前，物体列表建立之后，列表中的每个物体都储存了对应的包围盒信息。当光线跟踪进行时，一些绑定在 hit( )函数内部的函数 rayHitsSphereExtent( )和 rayHitsBoxExtent()将被调用，如图 12.55 所示。光线与球体包围盒的求交测试很容易实现(见练习)。

考察对应的二次等式的判别式是否为正。计算判别式需要做十一次乘法，因此这个测试也不是完全不花费时间的。

rayHitsBoxExtent()是 Cube 类中 hit()函数里的一个自适应函数，它使用了类 Cuboid 中的长方体包围盒结构。这个函数调整了程序运算的最快速度。

构造圆球体和长方体包围盒

每个给定的物体都需要计算球体和长方体包围盒。通用坐标系下每个物体的包围盒只需要计算一次，但世界坐标系下的包围盒每次都需要重新计算，因为每个物体的不同实例都有不同的仿射变换。仿射变换使得包围盒的计算更加复杂。怎样为一个经过复杂的伸缩和旋转变换的圆锥体构造圆球体包围盒呢？

一个特殊并且直接的方法是使用**点束**。点束是一些点，它们的凸包包裹着物体。空间中固定点集的凸包的直观的想像是：在这些点外面放置一个气球，然后让这个气球向这些点慢慢萎缩，最后得到的多面体便是它对应的凸包。对于一给定点束，很容易可以得到包裹它的一个球体。当然这样的球体也包裹着对应的物体。经过变换的物体可以通过这样得到它的凸包：对点束里的每个点做同样的变换，然后用这些变换过的新点求出变换过的凸包。因为仿射变换保持物体的相对位置不变，所以如果物体 A 在包围盒 B 之内，那么经过变换

的物体 $T(A)$ 一定在经过变换之后的凸包 $T(B)$ 之内。

因此需要对每种形状建立点束。长方体和平面的点束的建立很简单：直接采用它们的顶点。同样，网格也很简单：将其顶点作为点束即可。

对球体和圆锥体建立对应的点束要复杂一些，因为它们由一系列的对称关系所定义，而不是由很多点来定义的。一个合理的做法是求出这样的形状的一个紧紧包裹住它的多面体，然后用这个多面体的顶点来作为其点束。图 12.56(a)里一个圆锥体被一个底面为六边形的棱柱所包裹。为了紧紧包围圆锥的底面圆，棱柱的上下底面六边形的半径必需分别为圆锥上下底面圆半径的 4/3。注意不同的圆锥体实例有不同的六边形棱柱来包裹，这样求解点束的方法也只适用于圆锥体和圆柱体。图 12.56(b)是一个二十面体包围一个圆球的示意，它的 12 个顶点被用来作为圆球的点束。这个二十面体的顶点必须放在离球心大约 1.26 倍半径的地方，这样二十面体可以比较好地包裹圆球体。

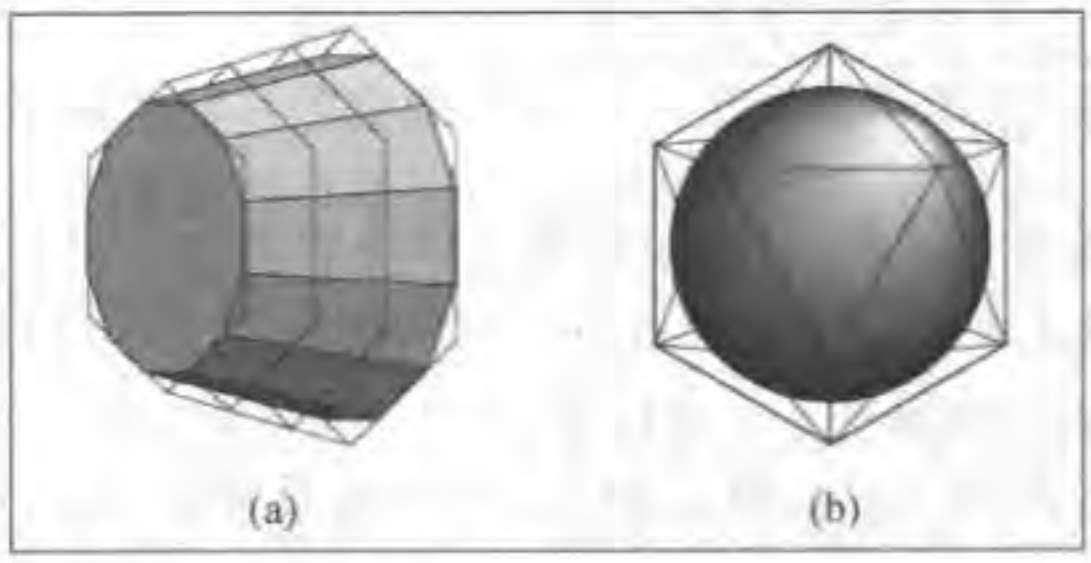


图 12.56 圆锥体和圆球体的点束定义

每种形状的点束可以储存在一个简单的数据结构 PointCluster 中，它记录点束中点的个数，并且用一个数组 pt[]记录点的坐标值。在光线跟踪算法之前的预处理中，每种形状都调用一个对应函数 makeExtentPoints(PointCluster& clust)。

一旦物体的点束建立成功，为这个物体创建长方体和圆球体的包围盒就很方便了。长方体包围盒的三个方向上的边的两个端点分别采用点束里面的点在对应方向上的极大值和极小值，因此可以通过一个单循环来求得。可以通过编写一个程序 void makeBoxExtent (Pointcluster& clust, Cuboid& cub)来分析点束，生成一个 Cuboid 类型数据结构并将其储存到对应物体中。

构建圆球体包围盒稍微复杂一些。点束的中心作为圆球的中心，因为这是一个很好的中心点。计算点束的中心，将点束中的点在三个方向上的分量分别求和，然后除以点的个数。而圆球体包围盒的半径选择为点束中的点到点束中心的最大距离。这样，函数 void makeSphereExtent (Pointcluster& clust, SphereInfo& sph)就很容易编写。

最后，构建世界坐标下的长方体包围盒和圆球体包围盒是很简单的。只需将通用坐标系下点束中的每个点都坐标变换到世界坐标系下，然后用上面同样的方法即可求得。

练习

练习 12.10.1 光线与圆球体包围盒的求交测试

(a) 求证包裹圆柱体(半径为 1)的最小圆球包围盒的球心是(0,0,0.5)，半径为 $\sqrt{1.25}$。

(b) 求证这样的圆球体包围盒可以用隐式 $F(x,y,z)=x^2+y^2+\left(z-\frac{1}{2}\right)^2-1.25$ 来表示。

(c) 证明求解等式 $F(S+\mathbf{c}t)=0$ 可以转换为求解 $At^2+2Bt+C=0$，其中 $A=|\mathbf{c}|^2$，$B=c\cdot S$，$C=|S|^2-1.25$，这里定义 $S'=(S_x,S_y,S_z-0.5)$。

练习 12.10.2　立方体包围盒测试的实现

请在函数 Cube::hit()里面加入函数 bool rayHitsBoxExtent (Ray& ray, Cuboid& cub)用来测试给定的光线是否与用 cub 描述的包围盒相交。这在不改变逻辑的基础上简化了函数 hit()。考察立方体每个面的 numer 和 denom 值是如何依赖于 cub 里面的数据的。例如，令 numer=cub. top-ray. start. y；denom=ray. dir. y。因为立方体的每个面都可能与光线相交，候选的区间(t_{in}，t_{out})不断地被更新，如果此区间长度变为零，函数退出并返回值 true。

练习 12.10.3　圆球体的点束

考察图 5.28 中的二十面体，这个二十面体的顶点要缩放多少倍才可以正好包裹那个圆球体。(缩放因子 $f=1.071$ 够不够?)

练习 12.10.4　创建圆球体和长方体包围盒的程序

编写从给定点束创建圆球体和长方体包围盒的程序：void makeSphereExtent (Pointcluster& clust, SphereInfo& sph)和 void makeBoxExtent (Pointcluster& clust, Cuboid& cub) 。

12.10.2　投影包围盒

光线跟踪中还有另外一种包围盒技术，它使得光线跟踪算法的速度得到很大的提高。与在三维空间中的长方体和圆球体不同的是这种包围盒是屏幕上的一个矩形区域。

物体的**投影包围盒**指的是屏幕上一块包裹物体投影的矩形区域，如图 12.57(a)所示。图 12.57(b)展示了投影包围盒的更多细节。投影包围盒记录四个坐标信息{*left*，*top*，*right*，*bottom*}。

在光线跟踪算法中使用投影包围盒十分简单。每条光线在一个特定的屏幕行列值上穿过，记为(r，c)。如果(r，c)在物体投影包围盒的外面，光线肯定不与物体相交。例如在图 12.57(b)中，如果 c 小于 244 或大于 296，或者 r 小于 237 或大于 319，那么这条光线与物体不相交。

因为投影包围盒与物体的相机和人眼的位置密切相关，它只适用于**眼光线**即从眼睛发出的光线。后面的章节将会介绍如何为阴影、反射光和折射光生成光线，因为这些光线从场景中任意点发出，没有屏幕可以被这些光线投影。但对眼光线，这个技术可以运行得很好，并且极大地提高光线跟踪算法的效率。

与长方体和圆球体包围盒一样，在预处理阶段为每个物体求投影包围盒，并且把它与对应的物体储存在一块。这样需要在 GeomObj 里面加入一个域 IntRect screenExtent 结构。还需要扩展类 Ray 是的光线知道它在哪里与屏幕相交。这里再为其提供一个信息：**递归级**

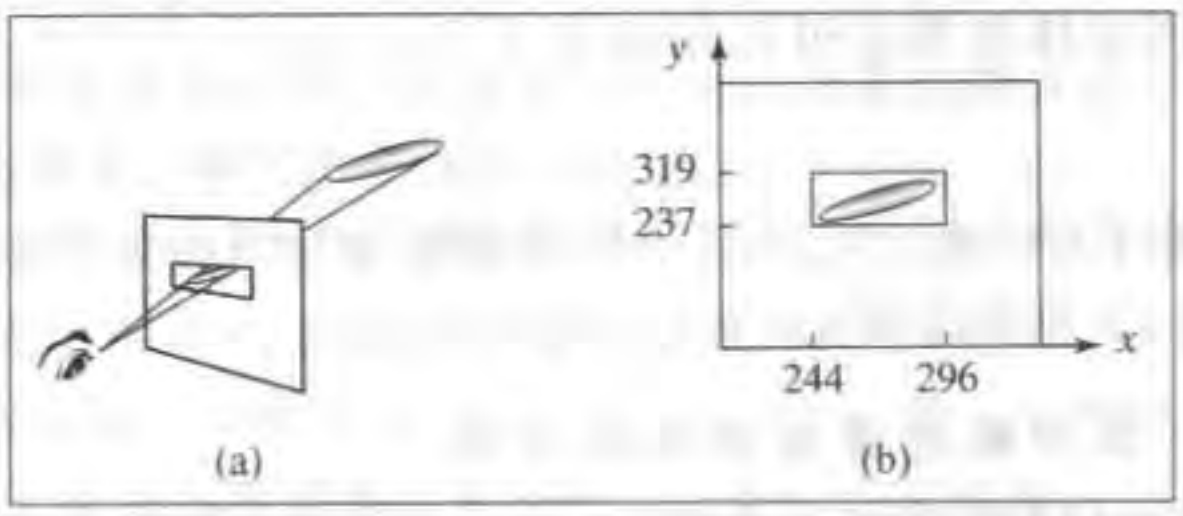


图 12.57 景物的投影包围盒

别。后面将知道,这在反射和折射中占有很大一部分。这里简单用来判断光线是否是眼光线。眼光线的级别是 0。

在长方体和圆球体包围盒测试之前,光线跟踪算法的函数 hit()已开始执行一个新的测试。图 12.58 展示了在类 TaperedCylinder 中这个测试放置的位置。同样的测试被用于除平面外的不同物体。与图 12.55 比较。这个测试检测光线是否为眼光线,如果是,接着检测光线与屏幕的交点是否在与物体储存在一块的投影包围盒之内。如果光线与屏幕的交点在投影包围盒之外,光线与物体不相交,函数 hit()返回 false。这个测试速度很快。

```
bool TaperedCylinder::hit(Ray &r, Intersection &inter)
{
    if(r.recurseLevel==0 &&
r.col<projExtnt.left ||
r.col>projExtnt.right ||
r.row<projExtnt.bottom ||
r.row>projExtnt.top)return false;     //与投影包围盒不相交
    if(!rayHitsSphereExtent(r, worldSphereExtent)) return false;
    求出逆变换,求出一般坐标下的光线
    if(!rayHitsSphereExtent(genRay, genSphereExtent))
return false;
…光线与包围盒相交;必须做花费很高的光线与景物的求交测试…
}
```

图 12.58 带有圆锥体 hit()方法的圆球体包围盒测试

计算投影包围盒

必须在预处理阶段为场景中所有的物体计算投影包围盒。给定物体的点束,相应的投影包围盒很容易计算。通用坐标系下的点束通过坐标变换转换到世界坐标下,和前面的世界坐标下的圆球体、长方体包围盒一样。图 12.59 是变换后的圆锥体的 12 个点的点束。

每个点都投影到相机所对的平面上。给定场景中的一个点 p,可以通过点 p 的坐标和相机的位置信息计算出它的投影 p'。这里记录了投影点 p'的行值 r 和列值 c。

图 12.60 给出了光线跟踪场景中每个物体的投影包围盒。为了调试方便,一些程序员喜欢在光线跟踪算法开始之前求出投影包围盒。这些长方形在最终像素值绘制之后将会消失。

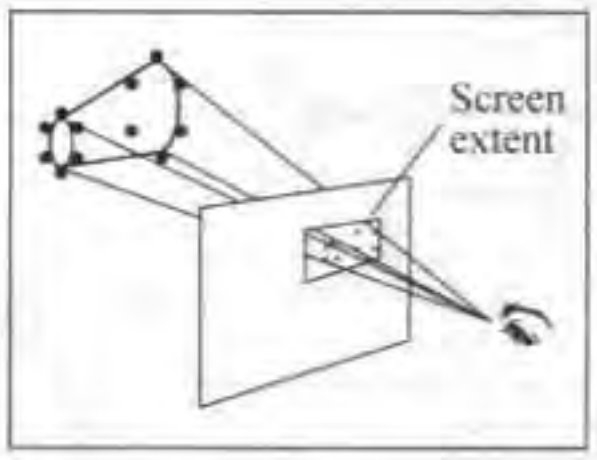


图 12.59　建立投影包围盒

图 12.60　光线跟踪场景中的可见投影包围盒

12.10.3　其他光线跟踪加速方法：二叉空间剖分树技术

我们知道，由于场景中有大量的光线和大量的物体，光线跟踪算法是十分耗时的。很明显，这么多光线与物体中，每条光线和每个物体都必须进行求交测试，这是耗时的关键所在。包围盒技术的运用一定程度上提高了算法的效率，但是对二级光线的帮助并不大。

因此需要其他的方法来加速光线跟踪。加速光线跟踪的一个很好的方法是二叉空间剖分树(BSP)方法【Arvo88】。它为光线跟踪构建了一个多层次的数据结构，通过快速决定哪些物体将不与光线相交排除了大量的求交运算。BSP 树是一种可以快速创建和遍历的数据结构。BSP 树通过某种顺序访问树的每个节点并且做当前节点与光线的求交测试来遍历。在每个节点，如果物体与光线相交，只需通过一个粗糙的求交测试即可判断。

构建 BSP 树

为了图示方便，这里我们通过二维例子介绍 BSP 树。因为树的组织与它所包含的物体的维数无关，所以把 BSP 树从二维推广到三维是很直接的。

图 12.61 展示了平面中的三个多边形和一个处在特定点的眼睛位置。问题是：从光线原点，可以看到什么？如果在三维中想象，一个观察者站在三块高出底面很多的大石柱之前。大石柱的横截面如图 12.61 所示。大石柱的哪些部分是可见的，哪些部分是不可见的？画出这个场景的草图。给每个边赋一个值，标出它的朝外的法向量。我们希望将每条边都插入到一棵树里面，并且满足平面被不重叠的多边形区域所分割开。

为这些边建立二叉树，先取一条边，记为#1。将此边的两端延长，边#1 将平面(多边形所在的)分成两部分，外部(边的朝外的法向量所指的半平面)和内部。我们将边#1 放置到二叉树的根部，如图 12.62(a)所示。接着将其他边按一定顺序都插入进去。考虑边#2，将其与边#1 比较。如果边#2 在边#1 的外部，那么将其放置在边#1 的右边树枝，标记成 outsideOnes。否则，将其放置到左边树枝，标记为 insideOnes。当每条边都插入进二叉树后，通过判断每个节点的走向，从上到下的路径就确定了。新边如果在当前节点的外部，则沿着节点往标记 outsideOnes 的子节点走，否则选择标记 insideOnes 的子节点，直到此新边成为二叉树的一片新叶子为止。

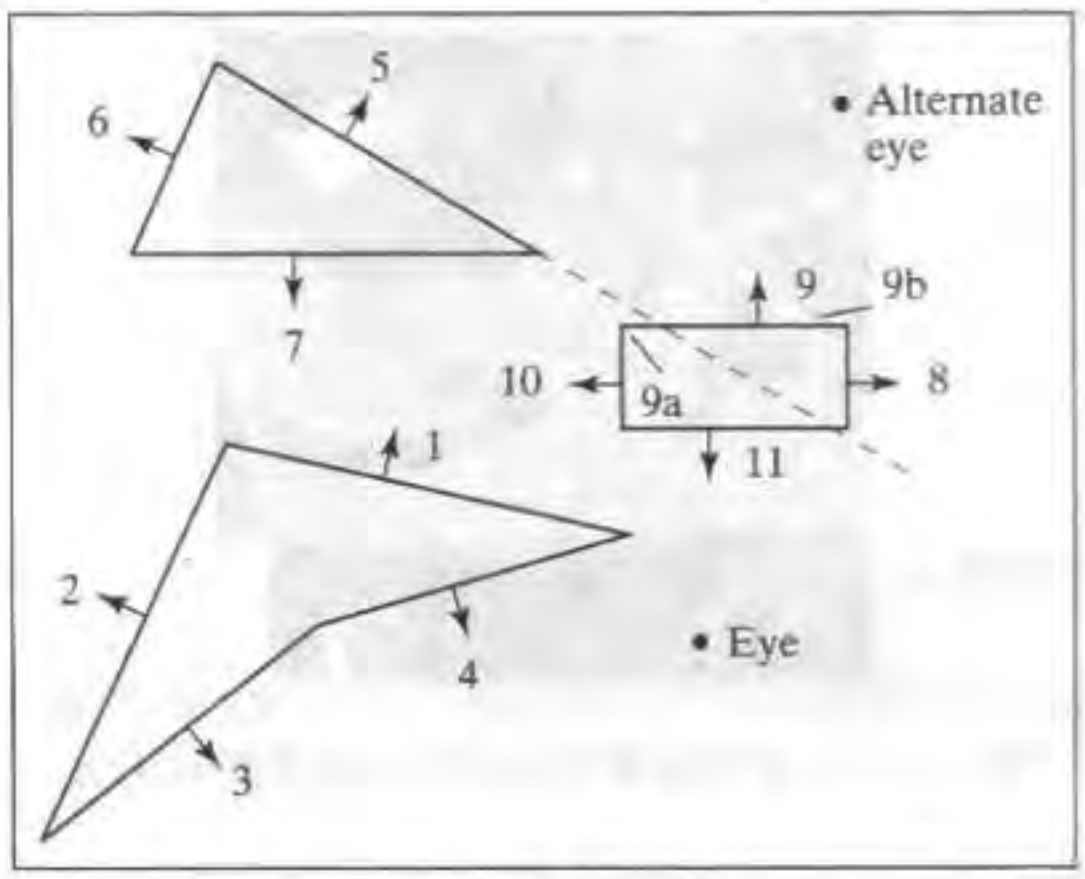


图 12.61 一些将被插入到一棵树结构的二维边

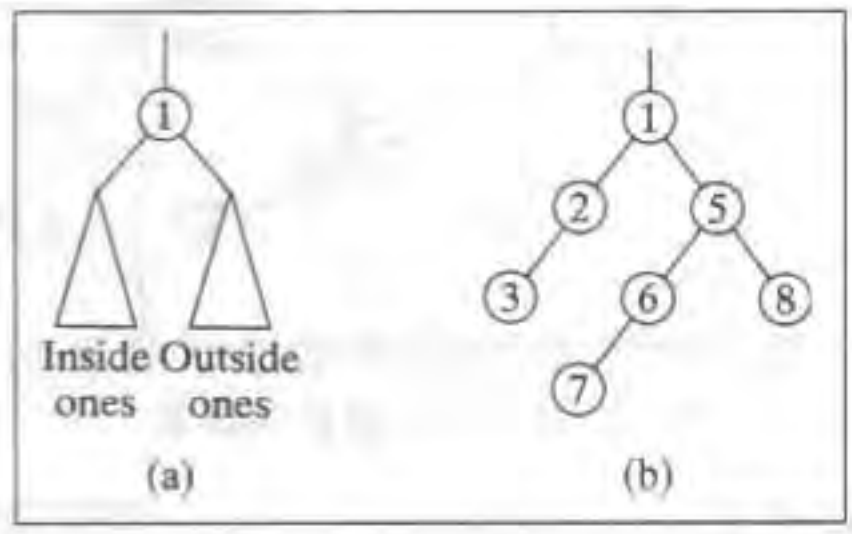


图 12.62 构建 BSP 树

图 12.62(b)给出了二叉树插入八个节点#1，…，#8 之后的情形。边#2 在边#1 的内部半平面，因此它被放置在 insideOnes 树枝。边#3 同时处在边#1 和边#2 的内部，因此它沿着图示方向往下走。边#4 没有在图中画出来，它应该处在图中的那个位置呢？

当我们插入边#9 时，会碰到新的情况。它处在边#1 的外部，但是在边#5 的两侧。这样，它被边#5 **分割**成两段，#9a 和#9b，如图 12.61 所示，每段可以继续往下走。边#9a 处在边#5 的内部，因此它往左边走，并且与边#6 和边#7 做测试判断；边#9b 处在边#5 的外部，边#8 的外部。边#10 和边#11 也按照同样的规则往下走，它们不需要分割成两段。(最终的二叉树是怎样的结构？)

BSP 树用特殊的方法将边列了出来，这样对每个边可以很快速地(通过它在二叉树中的位置)指出它所处的空间位置(例如，边#6 在边#1 的外部和边#5 的内部)。

而边插入的顺序对最终二叉树的形状有着极大的影响。一些顺序的选择方法可以使得最终的二叉树的形状宽并且“浅”(从树的根部到叶子只有很少数量的节点)，另外一些使得最终形状窄并且“深”。浅的树处理起来效率更高。BSP 树的最差的情况是从上到下只有一条单一的链。

三维场景中的 BSP 树

这个方法很直接地就可以推广到一个由三维多边形网格组成的场景中。考虑图 12.63(a)所示的多面体。为它的每个面都标记了数字，并指出了每个面相应的朝外的法向量。图 12.63(b)给出了同一多面体的另一个角度视图。如果按顺序将面#1，#2，…，#8 插入到二叉树，将得到一棵图 12.63(c)所示的树。(一个面按二维的方法往下走：如果它在当前节点面的外部半空间，那么它将往树枝 outsideOnes 方向走，继续直到最后。)面#7 是里面唯一需要分割的：它被面#3 分割成#7a 和#7b，如图中虚线所示。(有没有这样的插入顺序，使得最终没有面被分割的？)

这里我们看到了 BSP 树是如何完美地绘制一幅场景的。

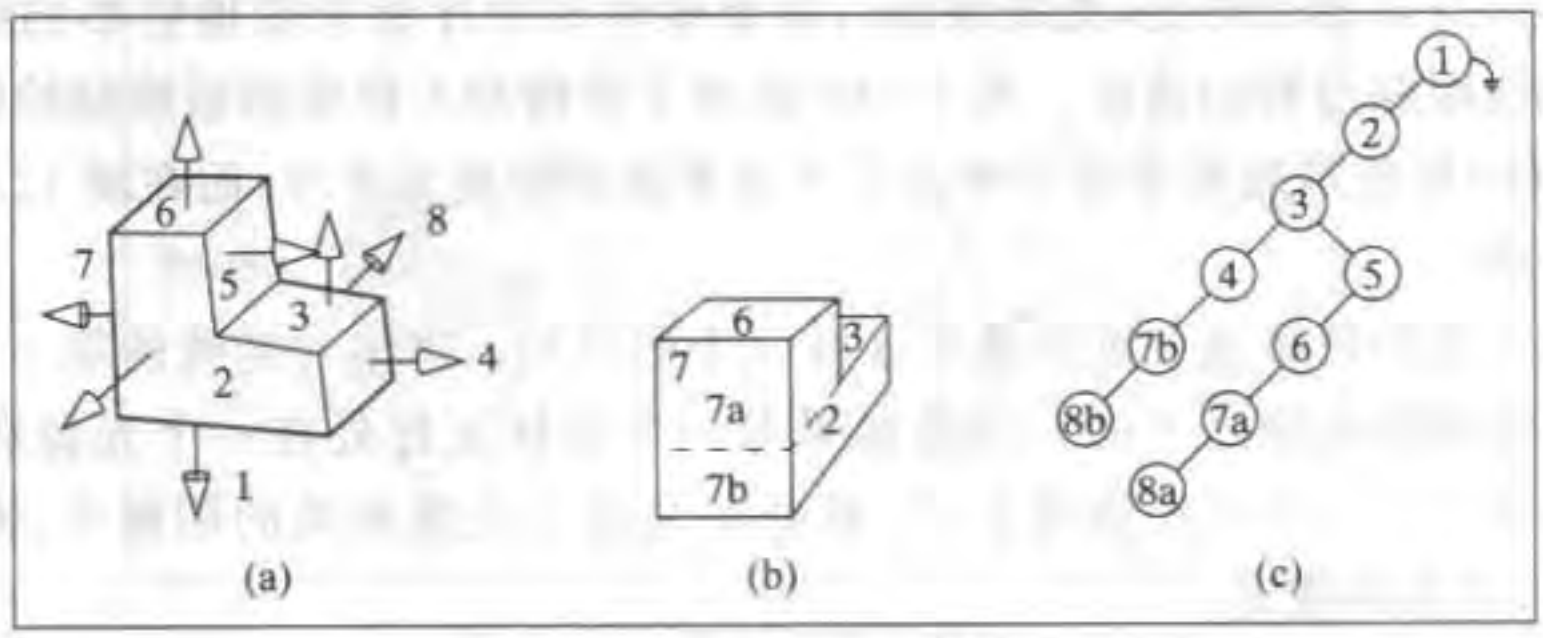


图 12.63　构建 BSP 树

横穿二叉树(用于光线跟踪)

首先应该决定哪个物体最先被光线碰撞。我们知道光线与物体相交时,同一层的节点中总是左边的节点先与光线相交。因此为了找到最小的时间,可以简单地从树根开始跟踪,每次都移动到下一层的左边子节点。最后移动到达图 12.63 所示的标记为 8b 的叶子,这片叶子就是最靠近眼睛的物体。

图 12.63(a)中三维场景的树(图 12.63(c))通过这种方法横穿,可以将此多面体正确地绘制出来。图 12.64(a)中的箭头表示访问图 12.63(c)中的 BSP 树节点的顺序,对应的是图 12.63(a)中多面体的特定朝向。图 12.64(b)给出了光线碰撞物体的升序排列时间的次序。下划线所指的面是不用绘制的面。

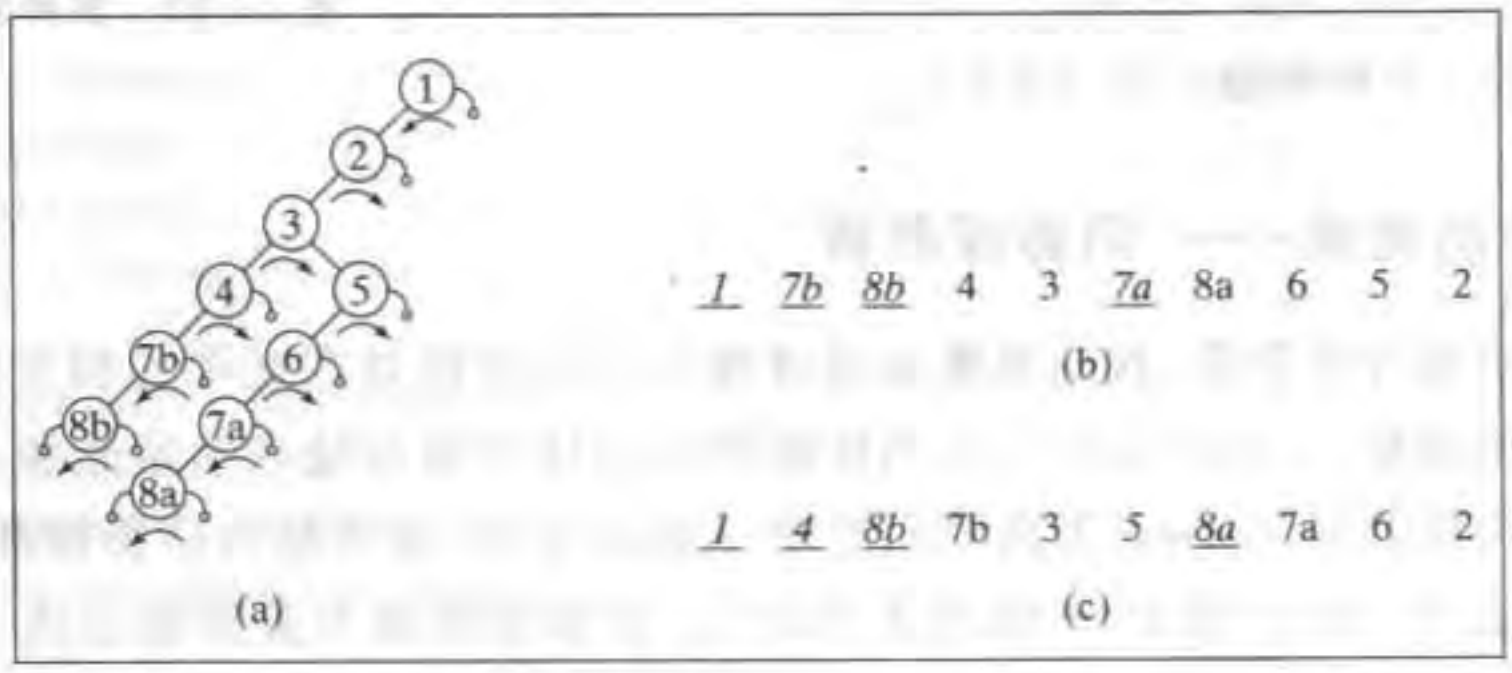


图 12.64　给定眼睛位置,面的绘制次序

有关光线跟踪算法中的 BSP 树技术的详细介绍,请参考"Graphics Gems III"【Kirk92】中的"Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations"和"Ray Tracing with BSP Trees"两篇文章。

12.11　添加阴影

为物体添加阴影使计算机绘制出来的图像更加真实。在第 8 章中我们了解到 OpenGL 提供了一些绘制阴影的基本机制,但是这些工具使用起来很不方便,而且只能在一些特定情

形下产生阴影。与其相反的是,光线跟踪只需要很少的程序量来获得阴影的绘制。不幸的是它降低了光线跟踪过程的速度。图 12.65 给出了使用和不使用阴影绘制的同一场景。注意在图 12.65(b)里很难判断多面体球离它下方平面的距离是多少,而在图 12.65(a)我们一眼就可以看出来。

到现在为止我们计算光亮度是基于这样一个假设的:物体与光线的第一碰撞点 P_h 总是沐浴在各种光源的光照之下的。但是如果有一个物体正好处在一个光源和 P_h 之间,上面的假设就不成立了。在这种情况下 P_h 就处在与这个光源相关的阴影中,漫反射和镜面反射都将消失,只有环境光。

图 12.66 给出了多种阴影的情形。从点 P 可以看到光源 L_1,因此点 P 不在光源 L_1 相关的阴影里面。但 P 处在与光源 L_2 相关的立方体的阴影里。此外,光源 L_3 被物体自身所阻挡,这时 P 处在自己所在物体的阴影中,这被称为**自身阴影**。对这样的场景结构,P 可以直接看到的唯一的光源是 L_1。

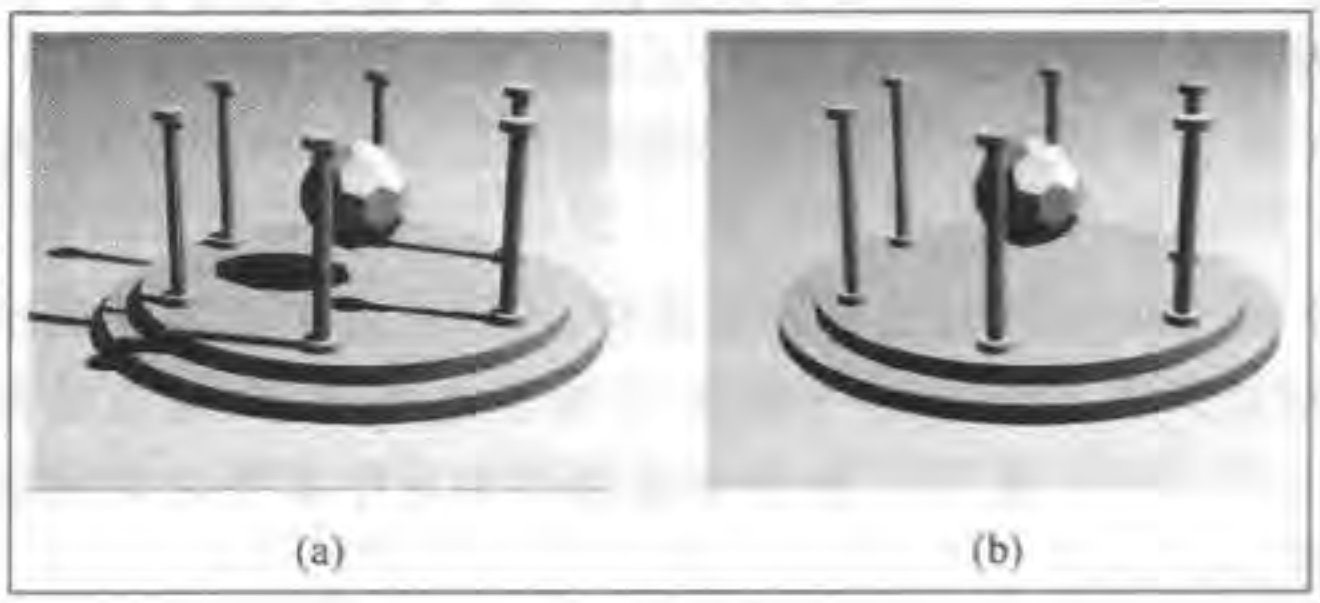


图 12.65

(a) 有阴影的渲染;(b) 无阴影的渲染

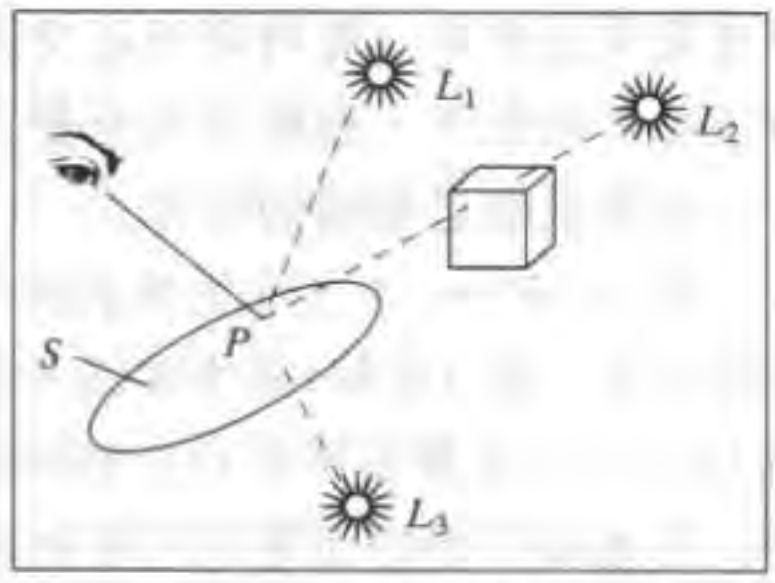


图 12.66 多种阴影情形

使用额外的光线——"阴影探测器"

为了能准确地计算阴影,我们需要知道碰撞点什么时候处在光源的相关阴影中。因此我们需要这样的函数,*isInShadow*(),当任何物体的任何部分处在给定光源与碰撞点之间时,返回 *true*,否则返回 *false*。为此我们使用一条新光线,通常称为**阴影探测器**,它在参数 $t=0$ 时从碰撞点 P_h 发出,在 $t=1$ 时到达光源 L。阴影探测器有参数表达式 $P_h+(L-P_h)t$。为了考察它是否碰撞到任何物体,所有的物体都将被扫描一遍,每个物体都与此光线执行一次求交测试。如果在 $t=0$ 和 $t=1$ 之间有物体相交,isInShadow()返回 true。这看上去要求比较高,但使用光线去探测场景的光线跟踪基本机制使得这变得简单。

直接简单地使用探测器会隐含一个严重的问题,物体自身的阴影。如果阴影探测器从 P_h 出发,那么探测光线与物体总是在位置 $t=0$ 处相交!于是函数 isInShadow()总是返回 true 值,这显然是错误的。一些人试图通过严格限制交点的参数 $t>0$ 这一条件来解决这个问题。但是,如图 12.67(a)所示,如果场景中光源与眼睛分别处在一片长方形或平面的两侧,这个真正的相交情况就会被误判。从 P 出发指向 L_1 的探测器正确地报告了没有交点,但是从 P 指向 L_2 的探测器也将给出没有交点的报告,然而点 P 在长方形自身的阴影之下。为什么这种异常不会在其他形状例如球体和锥体上出现?

一个比较可靠的策略是调整函数 isInShadow()里面的阴影探测器,如图 12.67(b)所

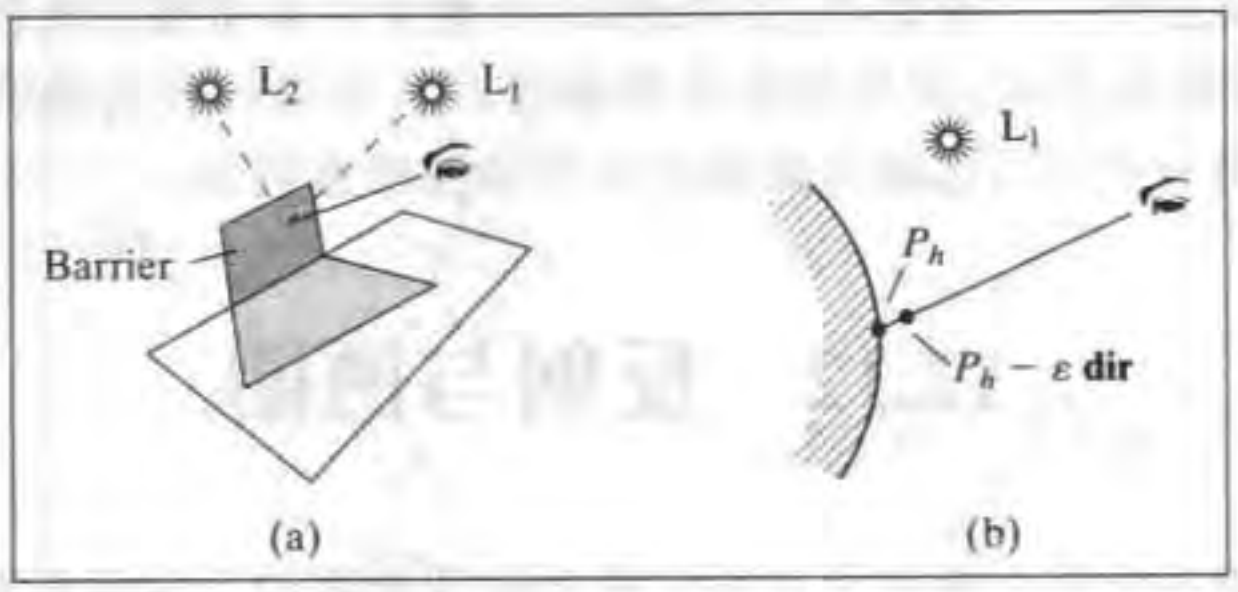


图 12.67　阴影探测器策略

示。让阴影探测器的开始点朝人眼方向移动一个微小的距离。如果光线的方向为 dir，碰撞点为 P_h，阴影探测器被移动到 $P_h-\varepsilon$dir，其中 ε 是一个很小的正数。这将开始点轻微移动到与光线碰撞的物体的靠眼睛侧。在阴影探测器中使用这样的开始点，在物体位置 t=0 时没有交点，问题解决了。

这种方法通过下面的方式嵌入到 Scene::shade(ray)函数中。当函数 getFirstHit()返回最佳的碰撞记录时，碰撞点与此点的法向量都确定了。光线的环境光颜色也确定了下来。接着一条探测光线从开始点 $P_h-\varepsilon dir$ 处发出。它的 recurseLevel 被设置为 1，这样投影包围盒将在不同的函数 hit()中禁止掉。

接着通过表达式 L. pos-feeler. start 为每个光源 L 计算探测方向，并且调用函数 isInShadow (feeler)来检查探测器是否碰撞到物体。如果发生碰撞，当前光源的漫反射和镜面反射的光线亮度计算将跳过。下面的伪代码是阴影相关的程序部分：

```
feeler.start=hitPoint-ε ray.dir;
feeler.recurselevel=1;
color=ambient part;
    for(each light source, L)
    {
        Feeler.dir=L.pos-hitPoint;
        If(isInShadow(feeler)) continue;
        color.add(diffuse light);
        color.add(specular light);
}
```

下面的代码是 isInShadow()函数的一种可能的实现。它简单地从物体列表里查看碰撞，如果有碰撞，则返回 false。如果没有碰撞发现，则返回 true。

```
bool Scene::isInShadow(Ray& f)
{
    for(GeomObj * p=obj; p; p=p->next)
        if(p->hit(f) return true;
    return false;
}
```

它对每个不同类型的物体使用一个简化版本的 hit()函数，它不需要构建碰撞列表。这个版本的 hit()函数与前面我们使用的有三点不同：

(1) 它只接受时间从 0 到 1 之间的碰撞，因为在光源另外一侧的物体不会产生阴影。

(2) 如果检测到这样一个碰撞,它立即返回,不需要计算碰撞的任何数据信息。

(3) 不能使用投影包围盒,因为阴影探测器可以从场景的任何地方发出。因此它必须为每种物体类型使用一些专门选取的球体或长方体包围盒组合。

12.12 反射与透明

光线跟踪算法的一个优点是它可以很简单地处理光线的反射和折射。这使得我们可以构建异常真实的场景,包括镜子、玻璃金鱼缸、透镜等。在到达人眼之前,光线可以在多个光亮的曲面上进行反射,或者同时进行反射与折射。每个这样的处理都需要构建额外的跟踪光线。

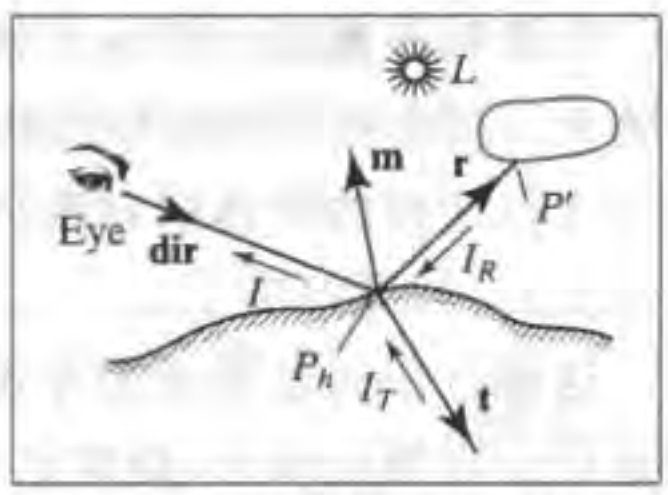


图 12.68 反射光与折射光

图 12.68 是一条从眼睛发出方向为 **dir** 的光线,与一曲面在 P_h 点相交。图示了二维情况的主要因素,这是可以接受的,因为反射与折射的属性使得所有的向量都在同一个平面上。所有的等式都可以推广到三维。当曲面为镜面或透明面或者两者皆有时,到达人眼的光线 I 可能由五个部分组成:

$$I = I_{amb} + I_{diff} + I_{spec} + I_{refl} + I_{tran} \qquad (12.42)$$

前面三项是熟悉的环境光、漫反射光和镜面反射光。漫反射和镜面反射部分由环境中 P_h 点可见的光源所产生的。I_{refl} 是反射光,由从方向 $-\mathbf{dir}$ 入射到点 P_h 的光线 I_R 产生的。方向是由入射角与反射角相等决定的,因此 **r** 由下面的等式给出:

$$\mathbf{r} = \mathbf{dir} - 2(\mathbf{dir} \cdot \mathbf{m})\mathbf{m} \qquad (12.43)$$

(其中我们假设 P_h 点法向量 **m** 已经归一化)。

相似地,I_{tran} 是光线传输的分量,它从光源 I_T 出发,沿着方向 $-t$ 到达点 P_h 其中一部分传输到透明材料内部。然后沿着方向 $-\mathbf{dir}$ 继续传输。折射的方向 **t**,取决于多个因素,有关细节我们下一节再讨论。

因为 I 是多种光的贡献之和,I_R 和 I_T 分别由其自己的五个分量组成:环境光、漫反射光等。如图 12.68 所示,光线 I_R 从 P' 出发到达 P_h 点,并在 P_h 点被人眼看见。为了计算 I_R,我们在点 P_h 构建沿方向 r 的第二条光线,找出与其碰撞的第一个物体,然后按等式(12.42)的重复计算光线分量。同样,I_T 的计算是通过构建一条 r 方向上的光线,求出与其碰撞的第一个物体,然后计算那里的光的贡献,诸如此类。

图 12.69 给出了对一条光线有所贡献的分量的数量是如何增加的。最终到达人眼的光线记为 I;它是很多个分量的组合,由很多个点的反射光和折射光以及很多个环境光、漫反射光和镜面反射分量组成。

如图 12.69(a)所示,I 由三个分量组成,反射分量 R_1、折射出来的分量 T_1 和局部分量 L_1(没有标记出来)。局部分量是 P_h 处的环境光、漫反射与镜面反射之和。局部分量的值取决于实际的光源;它们并不需要通过构建另外的光线来计算。环境光是模拟其他曲面上的漫反射与镜面反射对当前曲面的影响。R_1 是 R_3、T_3 和局部分量 L_3 之和。T_3 是 R_4、T_4 和 L_4 之和。每个贡献分量是另外三项之和,可能会出现无穷项。这样我们建议使用迭代的方法计算光亮度。

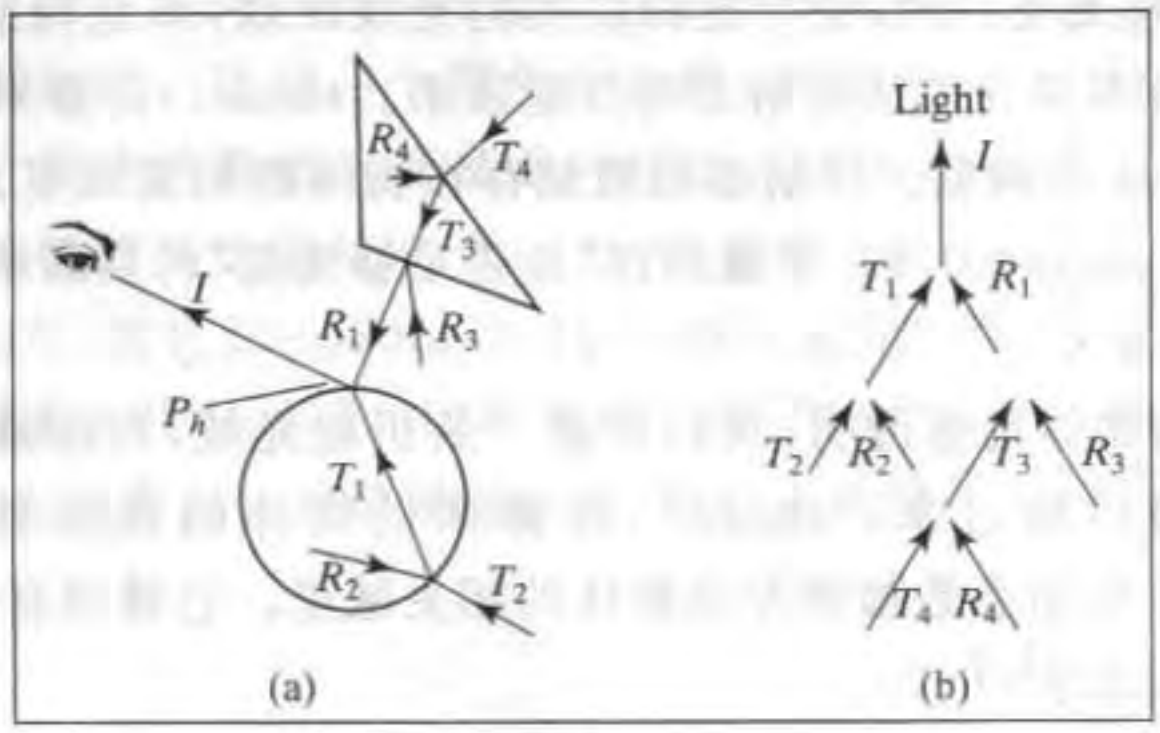


图 12.69　光线树

图 12.69(b)画出了由多个分量组成一棵树，树的左枝是折射分量，右枝是反射分量【Whitted80】。对每个节点，局部分量都必须添加上去，但是简单起见，这里没有画出来。

为了结合这几种视觉效果，函数 Scene∷shade()将被改成可以迭代地调用自己。图 12.70 给出了 shade()函数的结构框架，强调了在图 12.12 的 shade()函数中增加的部分。在正确的条件下，函数 shade()将调用自身两次来累计反射与折射光线的贡献。

```
Color3 Scene::shade(Ray& r)
Int recurseLevel=0;                          //设置迭代等级为 0
{
    计算第一次碰撞,构建碰撞信息 h
    Shape * myObj=(Shape * )h.hitObject; //指向碰撞景物的指针
    Color3 color.set(the emissive component);
    color.add(环境光分量);
    计算在碰撞点的归一化后的法向量 m
    for(每个光源)
        添加漫反射和镜面反射分量
    //下面添加反射与折射分量
    if(r.recurseLevel==maxRecursionLevel)
        return color;                        //不需要再迭代
    if(hit object is shiny enough)           //添加任何反射光
    {
        计算反射方向,构建反射光线 refl
        refl.recurseLevel=r.recurseLevel+1;
        color.add(shininess * shade(refl));
    }
    if(hit object is transparent enough)
    {
        计算折射方向,构建折射光线 trans
        trans.recurseLevel=r. recurseLevel+1;
        color.add(transparency * shade(trans));
    }
    return color;
}
```

图 12.70　函数 shade()的迭代结构

如果碰撞物体足够光亮,可以进行更深层次的光线跟踪,那么构造一条反射光线,运用函数 shade()来计算有多少光线从反射方向反射回来。shade()计算来的物体的光线亮度将通过反射参数 shininess 来调整。反射参数被储存在物体的相关域里。用户在 SDL 文件里指定参数,比如说 shininess=0.8。它被用在"是否足够光亮"的判断中,例如 if(shininess>0.6)。

同样,如果碰撞的物体足够透明,我们构造一条折射光线,利用函数 shade()来计算有多少光线从折射方向传输过来。shade()计算来的物体的光线亮度将通过折射参数 transparency 来调整。折射参数被储存在物体的相关域里。它被用在"是否足够透明"的判断中,例如 if(transparency>0.5)。

因为光线有可能一直不停地构建新的反射和折射光线,所以在迭代的深度上必须加一些限制。考虑这样一个场景,有四面镜子被适当放置以至于一条光线可以永远在它们之间反射而不离开。这样会导致无止尽的迭代,这应该得到控制。为了解决这个问题,我们控制每条光线的迭代深度;在阴影计算中提到的 recurseLevel 结构是每条光线储存结构的一部分。从人眼发出的光线的 recurseLevel 为 0。每当一条新的反射或折射光线被构建,这个值就自动增加。如果光线的迭代层次达到一个限制值 maxRecurseLevel(储存在 Scene 结构的一个域中,在 SDL 文件中被指定为 maxRecursionDepth),就不再产生反射与折射光线。

12.12.1 光线的折射

当一条光线遇到一个透明的物体,一部分光线穿透物体,如图 12.71 所示。如果光线在介质 1 和介质 2 中的传播速度不一样,它的方向将从 **dir** 变成 **t**。向量 **t** 仍处在 **dir** 和法向量 **m** 所在的平面内。如果光线的入射角为 θ_1,由 **Snell 定律**可以得到折射角为 θ_2,见下式【Halliday70,Rogeers92】:

$$\frac{\sin(\theta_2)}{c_2}=\frac{\sin(\theta_1)}{c_1} \tag{12.44}$$

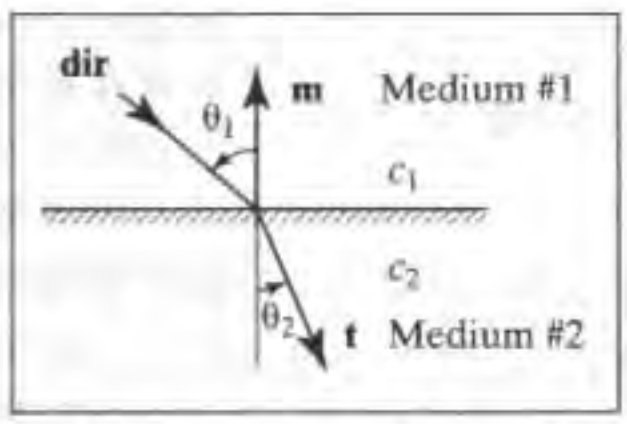


图 12.71 光线在透明介质中的折射

其中 c_1 是光线在介质 1 中的传播速度,c_2 是光线在介质 2 中的传播速度,如图所示。只有比率 c_2/c_1 才是重要的。通常称其为介质 2 关于介质 1 的**折射率**。注意若 θ_1 等于 0,θ_2 也为 0;若光线垂直入射到物体表面,折射光线也垂直表面。

例 12.12.1 求折射角

假设介质 2 为某种玻璃,光在其中的传播速度为真空中的 54%。再假设当前入射光的入射角是 60°。求折射角为多少?

解:

先求出等式(12.44)中的 $\sin(\theta_2)$,由 $c_2/c_1=0.54$,$\sin(\theta_1)=0.866$ 可以求得 $\sin(\theta_2)=(0.54)(0.866)=0.4676$,因此 $\theta_2=27.88°$。光线从传输快的介质进入传输速度慢的介质,传播方向会向表面法向量靠近。

图 12.72 给出了光在不同介质中传播的速度与其在真空(或空气)中速度的比值。这是一个简化的表格,因此不同波长的光在同一介质中传播的速度是不一样的。例如,光在溶化的石英中传播速度比值从 0.680(红光,波长 $\lambda=400\text{nm}$)到 0.685(绿光,波长 $\lambda=520\text{nm}$),到 0.687(蓝光,波长 $\lambda=680\text{nm}$)不等【Halliday97】。这种不同引起了一个常见的现象,当一束白光穿过一块玻璃棱镜时,被分离成像彩虹一样的光谱。

空气:99.97%
玻璃:52.2%～59%
水:75.19%
30%的盐水:72.46%
丙酮和乙醛:73.5%
氯化钠:64.93%
苯:55.5%
蓝宝石:56.5%
钻石:41.33%

图 12.72　不同介质中光的传播速度

临界角

例 12.12.1 中指出光线从速度快的介质进入速度慢的介质($c_2/c_1<1$),传播方向会朝法方向偏移。从等式 12.44 可以知道,因为 $\sin(\theta_2)$,等于 $c_2/c_1\sin(\theta_1)$,小于 $\sin(\theta_1)$,因此 θ_2 一定小于 θ_1。

反过来也成立,当光线进入一个传播速度相对快的介质:光线会偏离介质表面的法方向。当下标 1 和 2 互换的时候,Snell 定律是完全适用的。图 12.73(a)是光线从传播快的介质到慢的,图 12.73(b)是从传播速度慢的介质进入快的。这里两个介质中光线传播的角度是一样的,不同的只是代表它们的符号。图 12.73(c)和 12.73(d)给出了相似的情形,但是大角接近 90°。这时小角接近临界角。当小角(与传播速度慢的介质相关的)变得足够大时,它迫使大角接近 90°。更大的角度是不可能的,因此没有光线传输入相邻的介质中。这被称为全反射。考虑到图 d 的情形,光线从传播慢的介质进入传播快的介质。因为 $\sin(\theta_1)$ 等于 $c_1/c_2\sin(\theta_2)$,当 $\theta_2=90°$或 $\theta_1=\sin^{-1}(c_1/c_2)$时,角 θ_1 取得最大值。例如,当光线从水里传播到空气时,我们有 $c_1/c_2=0.7519$,因此临界角为 48.75°。

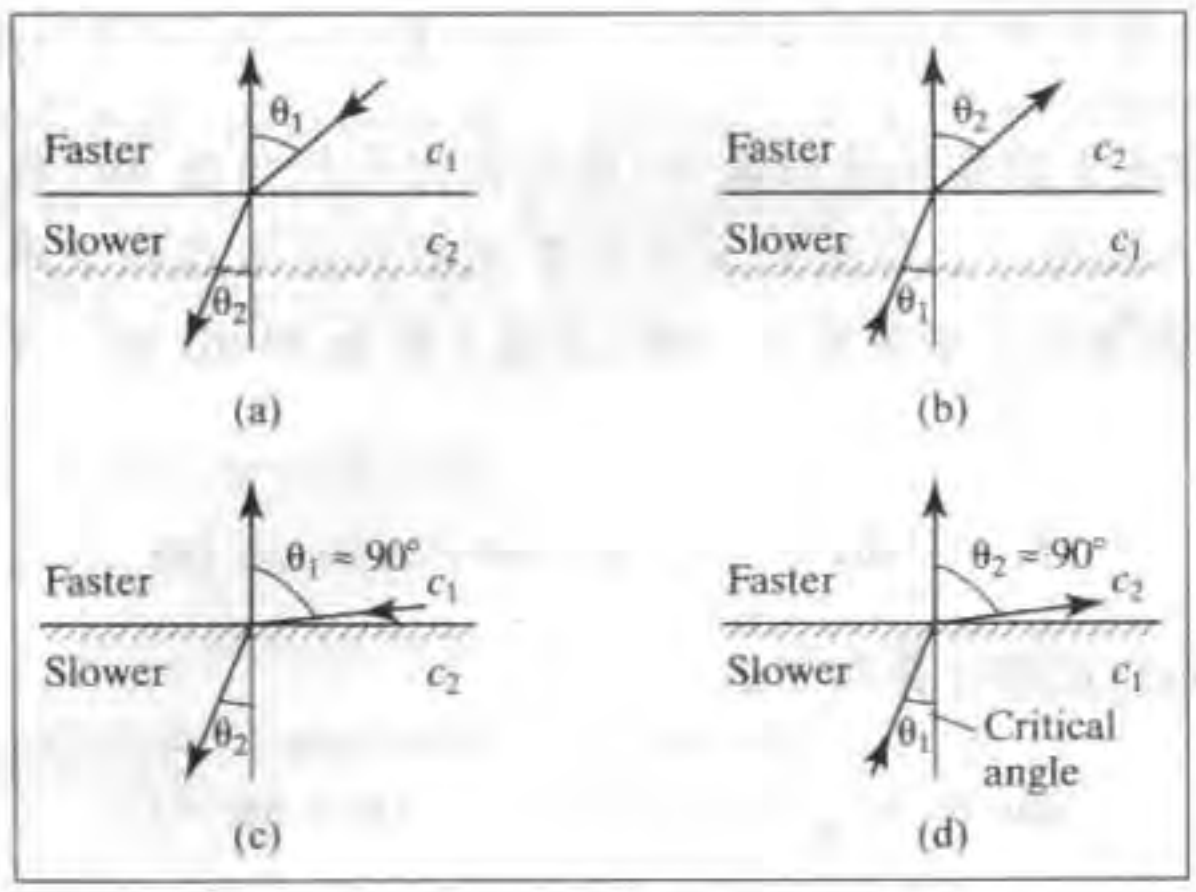


图 12.73　光线折射的对称性与临界角

临界角在一个有趣的情况下出现。当你处在池塘的水下,抬头向上看,你可以看到水面上的地平线以上的整个世界。当你以 48°角仰视的时候,你就可以看见池塘的边岸。另一方面,如果你在一条船上向下看池塘,你可以看到的角度范围是多少?有没有一个限制视角的圆锥体,在其之外你看不到任何东西?(参见本节最后的练习。)

例 12.12.2 彩虹的形状是怎样的?

光线在水中的传播速度随着波长的变化而变化,这就是彩虹的起因。图 12.74(a)所示是悬在空中的球体水滴。一条太阳光从左边入射,在水滴内部有个轻微的折射,在离开水滴之前又经历一个全反射。入射光线与出射光线的夹角大概为 42°。红色光线比蓝色光线折射角度小,因此出射的角度也有所不同。这样红色光线从成千上万个水滴中以同一方向反射出来,蓝色光线却以不同的方向出射。

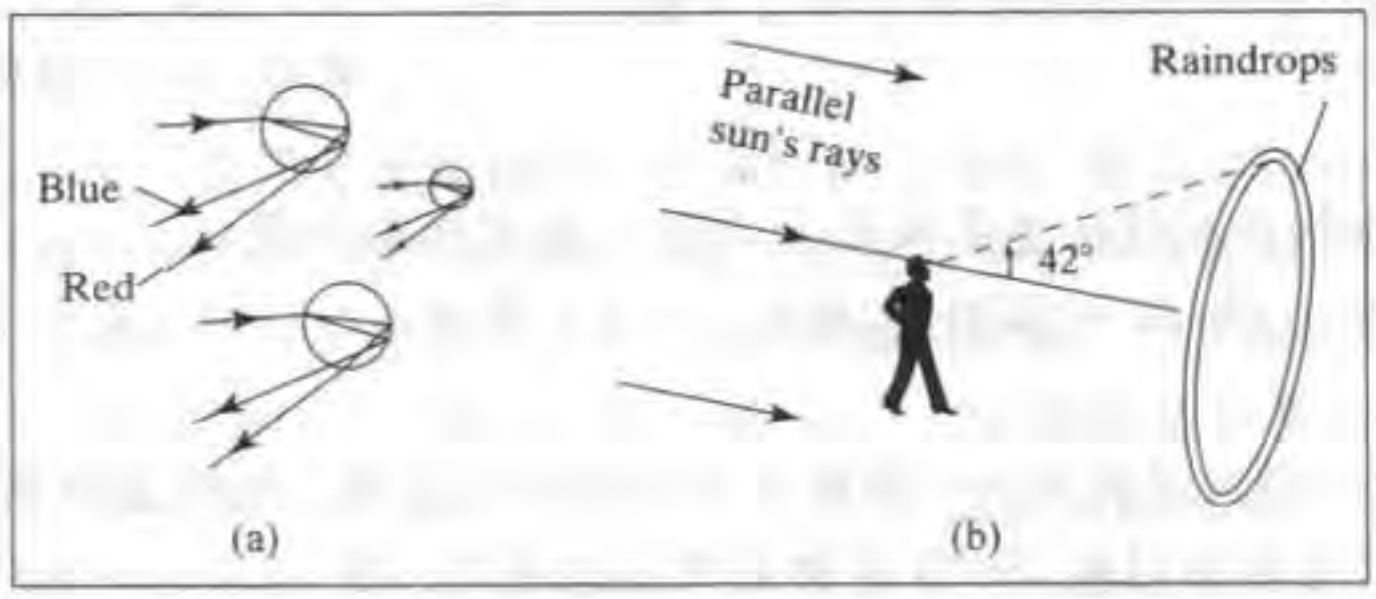


图 12.74 彩虹的起因

在 1637 年笛卡尔利用一个简单的光线跟踪模型来解释彩虹的形状。图 12.74(b)为一个背朝太阳的观察者,朝着由小水滴组成的气雾看去。当观察者的视角偏离太阳光 42°以上,她将看到一些光环。水滴沿着一个特定的角度反射一种特定颜色的光线,当角度有微小的变动时,反射回去的光线颜色也有所不同。因此,彩虹是一系列光线圆环的组合,每个观察者有她自己的彩虹。

计算折射角 t

在光线跟踪中,给定了曲面的法向量 **m**,和入射光线的角度 **dir**,我们必须计算折射角 **t**(见图 12.71)。我们应该在一个与坐标无关的系统中只使用点积来求得,为了可以适用任何 **m** 和 **dir** 方向。就像在练习中提到的一样,向量 **t** 是 **m** 和 **dir** 的一个线性组合(假设它们都被归一化):

$$\mathbf{t} = \frac{c_2}{c_1}\mathbf{dir} + \left(\frac{c_2}{c_1}(\mathbf{m} \cdot \mathbf{dir}) - \cos\theta_2\right)\mathbf{m} \tag{12.45}$$

其中 $\cos\theta_2$ 由 Snell 定律计算得来:

$$\cos\theta_2 = \sqrt{1 - \left(\frac{c_2}{c_1}\right)^2 (1 - (\mathbf{m} \cdot \mathbf{dir})^2)} \tag{12.46}$$

检查 **t** 在这两种情况下是否有正确的值:(a)两个介质中光传播速度一样;(b)入射角为时 0°。

如果等式 12.46 中平方根下的值为负数,就会出现全反射现象,这时,求解 **t** 的等式没有意义。这种情况在临界角时发生。练习中要求我们编写求解 **t** 的函数 transmitDirection()。

因为求解 **t** 只用到点积,它也适用于二维的情形。参见案例 12.8 中有趣的二维折射。

编写光线跟踪程序时,最简单的透明模型是不考虑光线波长对折射角的影响。在这种假

设下，同一条光线被用来跟踪红、绿、蓝的颜色分量。否则的话，每种不同颜色的光线需要不同的光线来跟踪，因为她们有不同的折射方向。这样计算的花费巨大，而且也仅仅是对光线的一个近似，因为精确地模拟折射需要考虑大量的颜色的光线，而不是这样三种主要颜色的光线。

练习

练习 12.12.1　折射的效果

图 12.75(a)给出了在高度 H 的人眼从空气往水里看的场景。一条与人眼水平距离相隔 L 的鱼在水下深度为 D。假设光在空气中传播速度是在水中的两倍，在什么地方人眼可以看见鱼？

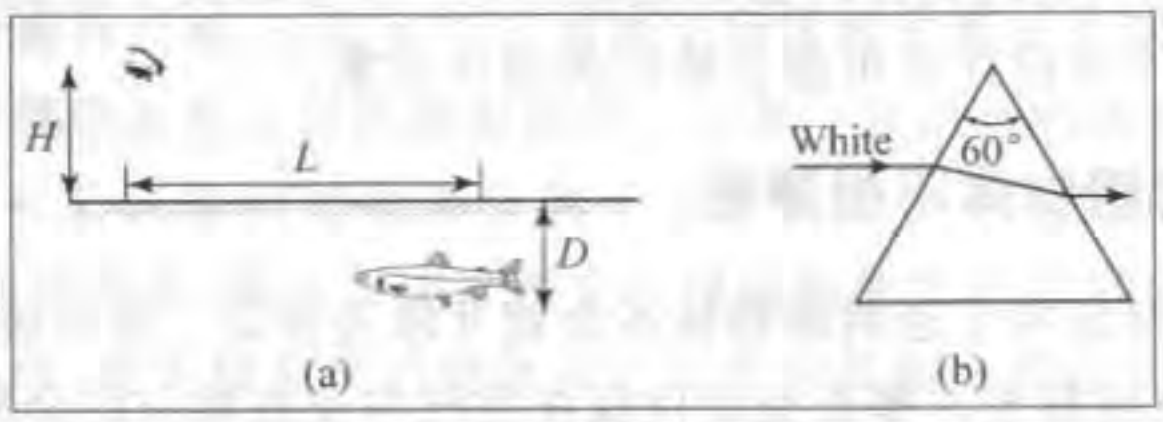


图 12.75　一些折射的图示

练习 12.12.2　画出彩虹

图 12.75(b)给出了一束穿过内角为 60°的棱镜的例子。棱镜是由石英制成的。计算并画出红、绿、蓝三种光穿过棱镜并分离的路径。若光线从一块两边平行的石英厚板穿过，存在光线分离之后的彩虹么？

练习 12.12.3　计算折射光线的方向

图 12.76 给出了类似【Heckbert89】中的一个简单的计算折射方向 **t** 的方法。单位向量 **dir** 和 **m** 唯一确定了一个平面，折射光与反射光也在这个平面之内。因此所有的向量都是单位向量，它们的坐标分量都是与 **m** 垂直或者平行，因此它们的长度可以用角 θ_1 和 θ_2 的正弦与余弦来表示。

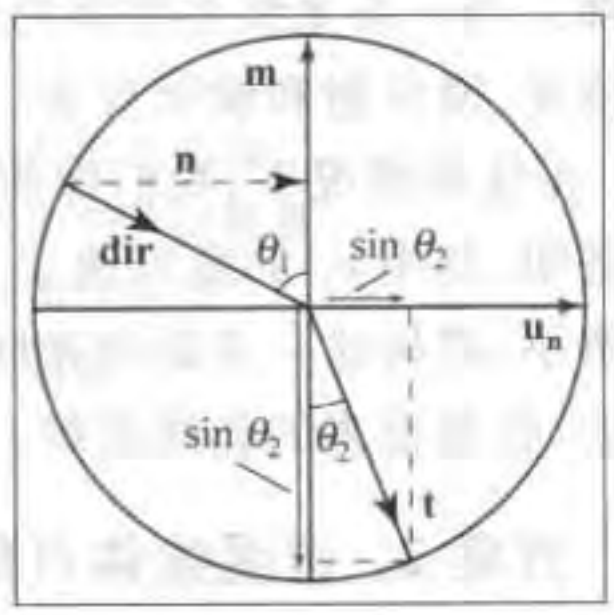


图 12.76　计算折射光线的方向

a. 证明：图中记为 **n** 的向量可表示为：

$$\mathbf{n} = \cos(\theta_1)\mathbf{m} + \mathbf{dir}$$

其长度为 $\sin(\theta_1)$，因此归一化之后的 $\mathbf{u}_n$ 等于 $\mathbf{n}/\sin(\theta_1)$。并用点积证明 **n** 处在 **m** 与 **dir** 所在的平面内，并且与 **m** 垂直。

b. 证明：

$$\mathbf{t} = \sin(\theta_2)\mathbf{u}_n - \cos(\theta_2)\mathbf{m} = \frac{c_2}{c_1}\mathbf{dir} + \left(\frac{c_2}{c_1}\cos(\theta_1) - \cos(\theta_2)\right)\mathbf{m}$$

并证明它与等式(12.45)等价。

c. 用 Snell 定律证明 $\cos(\theta_2) = \sqrt{1 - \left(\frac{c_2}{c_1}\right)^2 \sin^2(\theta_1)}$ 和等式(12.46)。

练习 12.12.4 计算折射方向的程序

给定单位法向量 **m** 和方向 **dir** 以及光线在两个介质中的传播速度,编写计算方向 **t** 的程序 Vector3 transmitDirection (Vector3 m, Vector3 dir, float c1, float c2)。

12.12.2 在函数 shade()中处理折射

当场景中包含透明的物体时,我们必须跟踪光线穿过的介质内部,因此我们可以计算下一次碰撞(光线穿出一种介质或者光线进入另外一种介质)时 c_2/c_1 的值。这可以很简单地做到:为光线添加一个域来储存一个指向它进入的物体的指针。

函数 shade()是如何处理物体内部的光线的呢?答案取决于绘制场景的人有多少自由度。我们可以想像若干个设计者可能会赞同的设计方案。

方案 1:两个透明物体不相渗透

首先假设设计者保证两个透明的物体不会相互渗透相交。没有玻璃大理石放置在另一个里面,没有水存在玻璃杯里。那么光线仅仅在空气中或在单一的介质中传播。

(a) 假设当前 shade()中的光线在所有物体之外,然后碰撞到一个足够透明的物体 A。利用等式(12.45)计算折射光线的折射角 **t**,这里空气中 $c_1=1$,而 c_2 取决于物体 A 的属性。一条包含指向 A 的指针的新光线被构建,它的 recurseLevel 适时地增加。然后 shade()递归地被调用,最终返回一个颜色值。这个颜色值被 A 的透明属性调整过并被添加到累计叠加的颜色里。

(b) 假设,另一方面,当前光线在某物体 A 内部并且碰撞到另外一张曲面。这张曲面一定是 A 的面,光线从 A 出射到空气中。程序可以检查光线是否碰撞的是它所在的物体。因为光线在物体内部,碰撞点的法向量的符号必须逆反一下;我们需要它指向光线传播的物体内部。当一条光线在物体内部时,通常认为它不在光源的照射之下,因此不去计算它的局部环境光、漫反射和镜面反射。A 的内表面被认为足够光亮,可以将一部分光线反射回物体 A。在这种情况下,我们像前面一样构建并计算反射光线。对于折射光线来说,A 显然足够的透明,其中 c_1 的值取决于 A 的属性,c_2 的值被设置成空气的对应值 1。如果入射角比临界角小,将构建一条新的光线(将其指针设置成 NULL,因为它不与任何物体相关)并且跟踪它,收集更多的光线贡献。

方案 2:透明物体可能相互渗透

如果我们允许把场景中透明的物体放置在一起并互相渗透,情况将会变得复杂一些。在图 12.77 的场景内,一个玻璃立方体嵌入了若干个物体:一个石英大理石球体,一个空气填充的圆柱体,一填满水的立方体和一块特别嵌入的玻璃锥体。我们将考察如何为第 12.13 节中提到的这样的**布尔物体**建立模型。光在每个物体里传播速度都不一样。图 12.77(b)所示光线在一系列相交的透明物体中的传播的情形。光线频繁地进出不同物体。用一个列表记录光线进出的物体。沿着光线的路径,物体按照一个复杂的次序在列表中增加或删除。这虽然是一个人为制造的场景,但是它值得用来检验函数 shade()是否可以正确地处理这类问题。光线跟踪算法可以添加这种额外的能力也可以不加。

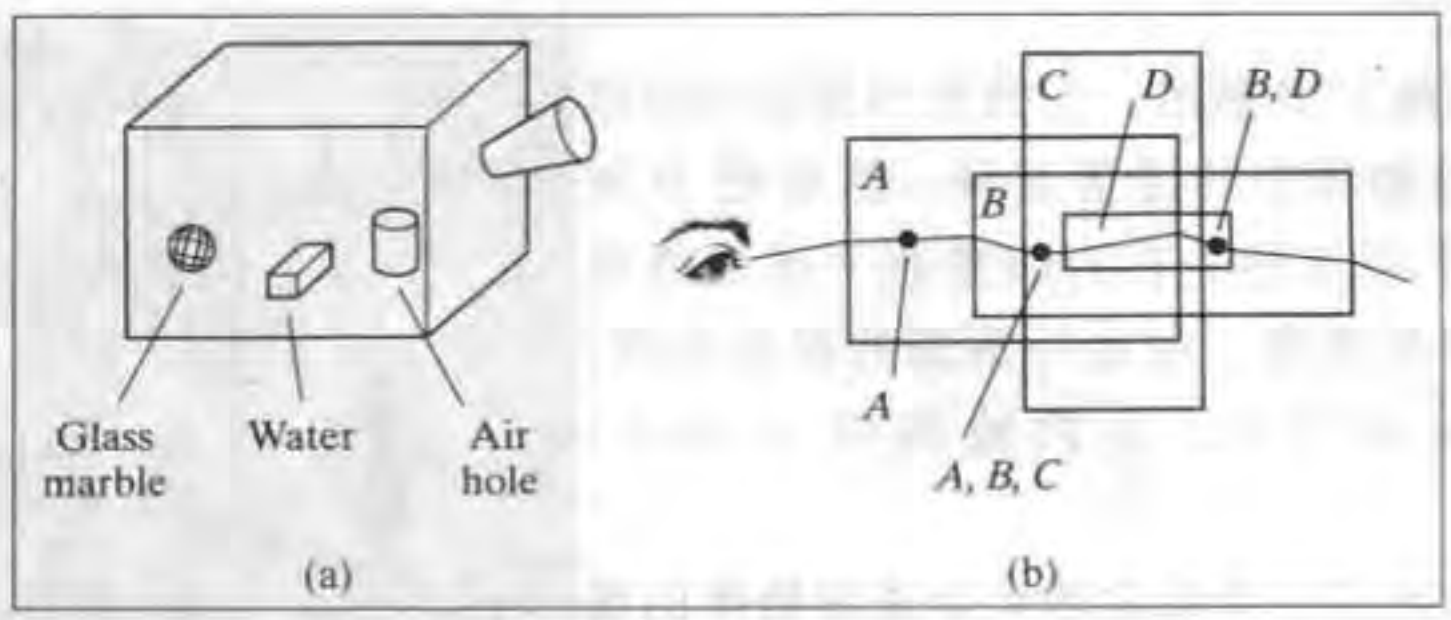


图 12.77　若干个互相渗透的透明景物

同时在两个透明物体内部，空间中的点被两个物体拥有是什么意思呢？在建模的层面，你必须决定想渗透的部分有什么样的材料属性。如果一块绿色的大理石完全包含于一块蓝色的大理石内部，直观上认为相交的部分属于蓝色大理石，给其着上蓝色。但是如果像图 12.77(b)所示的部分相交，那么要决定相交区域的颜色就不是那么明显了。设计者必须给每个物体一个优先级，优先级最高的物体决定每个与其相交的区域的颜色。

一种处理方法是给每个物体都增加一个域来储存优先级信息，并在 Ray 结构中增加一个指向此物体的指针，于是形成了一个指针列表。在任何时刻，光线处于某个物体集合当中，这时的列表实时记录了这个集合。如果列表为空，光线则处于所有物体之外。因此我们必须扩充 Ray 类型，如图 12.78 所示。列表被储存成一个数组而不是一些相互连接的列表是为了在添加物体的时候方便一次复制整个物体。

```
class Ray{
public:
    Point3 start;
    Vector3 dir;
    int recurseLevel;
    int row, col;                          //用于投影包围合
    int numInside;                         //列表中的景物数量
    GeomObj * inside[8];                   //景物指针数组
    Ray(){recurseLevel=numInside=0;}       //构造函数
    …其他方法…
}
```

图 12.78　Ray 类型的扩充

怎样在函数 shade()运用这个列表呢？在任何时刻当前光线都处在一些物体的内部，这被记录在列表中。当它继续碰撞到某个物体的表面时，记为 B，我们对其是入射还是出射 B 相应地采取不同的策略，这可以依靠 isEntering 来决定。

(a) 假设光线入射 B。如果 B 没有足够透明，就不构建折射光线，但如果 B 足够光亮，需要构建反射光线。如果 B 足够透明，使用当前列表里优先级最高的物体的光线传播速度 c_1。如果 B 的优先级别更高，那么 c_2 使用光在其中的传播速度；否则让 c_2 等于 c_1。添加 B 到物体列表。构建一条新的折射光线，将当前的列表(包含 B)复制过去。递归地调用 shade

()函数。

(b) 假设光线从 B 射出。c_1 我们用光在列表里面优先级最高的物体中的传播速度。然后把 B 从列表中删除。而 c_2 使用光在列表里剩下优先级最高的物体中传播的速度。构造一条新的折射光线,将当前的列表复制过去。递归地调用 shade()函数。

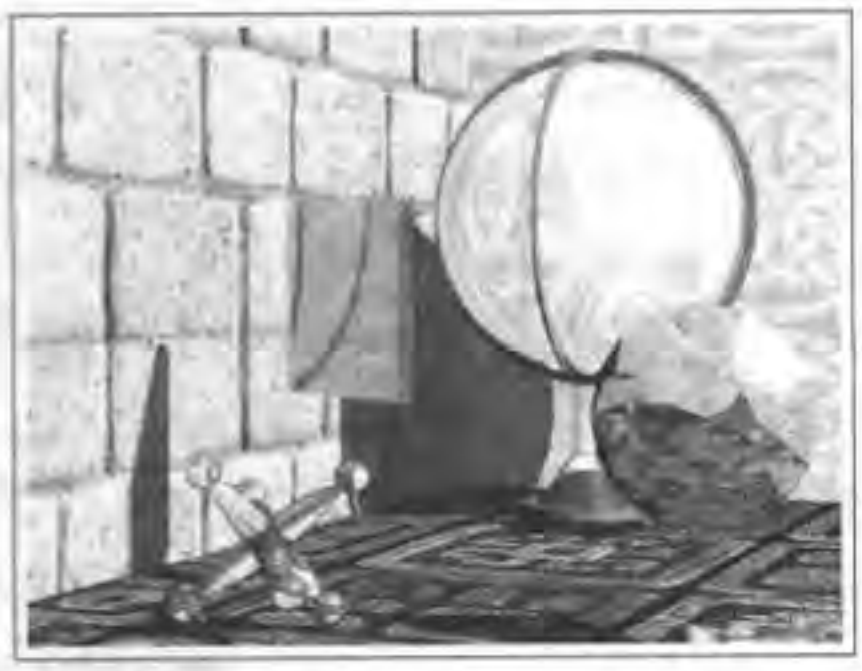


图 12.79 有折射和没有折射的场景

图 12.79 给出了一个包含若干个透明物体的场景。球体和架子是透明的,地板和墙是不透明的。当光线在透明物体中折射时,扭曲现象就会出现。

练习

练习 12.12.5 另外一种设计方案:如果两个透明物体相互渗透,一个包含另外一个

假设建模者规定不允许两个透明物体只有一部分相交:既不是分离,也不是一个包含另一个。用集合的术语来说,它们的交集既不为空,也不是它们中的任何一个。讨论逻辑上 shade()应该如何实现。特别地,说明处理列表过程变得简单:不需要优先级,列表可以被视为堆。讨论实现的细节。

练习 12.12.6 光线从有颜色的玻璃中穿过

讨论光线从桔黄色玻璃球穿过时,折射效果如何实现的。

12.13 复合体:物体上的布尔操作

到目前为止,我们只能光线跟踪有限的几种形状:变换过的球体、平面、锥体等。如果有一个方法可以将这些简单形状组合成一个复杂的物体,并开发适合这种情况的光线跟踪算法,那么将有更多更有趣的场景可以用光线跟踪算法绘制(图 12.80)。

构造实体几何(CSG)就是这样一种方法。任意的复杂形状被定义为一系列简单形状上的布尔**操作**的集合【Ballard82, Mortenson85】。象透镜、透明鱼缸和包含洞的物体之类的这些物体,可以很容易地通过目前已接触过的简单物体的组合来得到。这样的物体被称为**复合体**、**布尔体**或 **CSG** 体。将光线跟踪算法推广到这类物体上很自然:这是光线跟踪算法强大的一个地方,它与 CSG 模型结合的非常好。

我们看例子中的三种布尔操作:**并集**、**交集**和**差集**。图 12.81 给出了两个有球体构造出来的复合体。图 12.81(a)是有两个球体通过**交集**构造的透镜。一个点落在透镜内部,当且仅当它同时落在两个球体内部。球体 S_1 和 S_2 相交,符号上我们通常用 L 记成等式(12.47)。

$$L = S_1 \cap S_2 \tag{12.47}$$

图 12.80 复杂布尔物体的光线跟踪

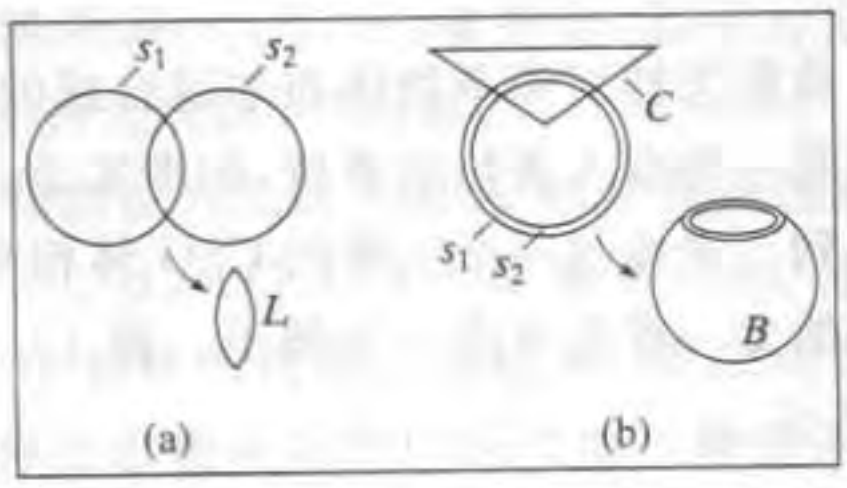


图 12.81 透镜和鱼缸

图 12.81(b)给出了由差集操作得到的一个鱼缸。一个点在集合 A 与 B 的**差集**中，记为 $A-B$，如果它在 A 中不在 B 中。差集相类似于移除物体，切割或雕刻。这个鱼缸由下面的等式(12.48)定义。

$$B=(S_1-S_2)-C \tag{12.48}$$

移除球体 S_1 内部的球体 S_2 里面所有点，可以得到一个内空的球壳。然后将其顶端移除锥体 C 里面的点得到一个开口。

一个点在两个集合 A 和 B 的**并集**中，记为 $A\cup B$，如果它属于集合 A 或集合 B 或两个集合。求两个物体的交集类似于将它们粘合在一起。图 12.82 给出了一个火箭模型，它由两个圆锥体和两个圆柱体通过并集操作得到，数学上表达为等式(12.49)。

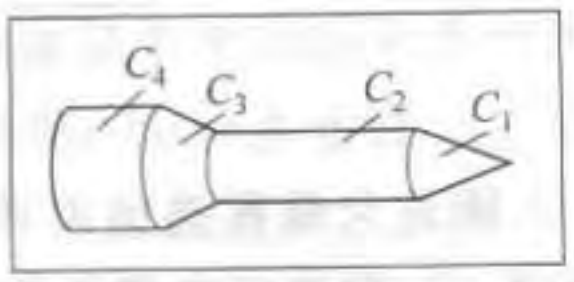


图 12.82 四个简单体的并集

$$R=C_1\cup C_2\cup C_3\cup C_4 \tag{12.49}$$

锥体 C_1 接在柱体 C_2 上。锥体 C_3 一部分嵌入到 C_2 里，另一边拼接在 C_4 上。

练习

练习 12.13.1 分解复合体

求图 12.83 中物体的等式表达式，运用球体、锥体、柱体和规则平行六面体的集合操作来表示。注意同一个物体的表达式可能不一样。

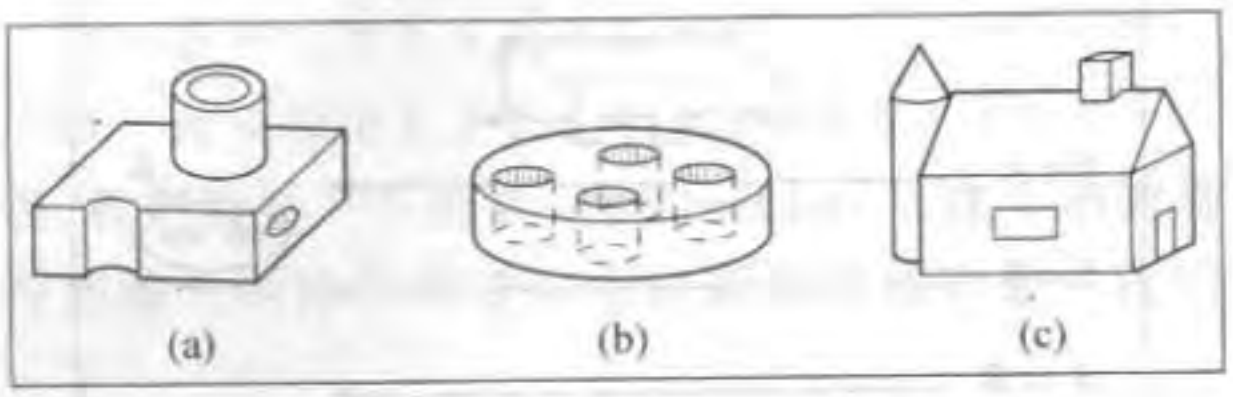


图 12.83 几个由简单形状组成的物体

练习 12.13.2 组合复合体

考虑保龄球类似的物体，球面上有三个便于抓牢的指洞。将其表达成一个球体和几个柱体的 CSG 组合。

12.13.1 CSG 体的光线跟踪

我们怎样对简单物体布尔组合成的物体使用光线跟踪技术呢？第一眼看上去可能会认为这是一件令人害怕的事情，但事实上，归功于光线跟踪过程的本质，这件事情可以很容易地办到。光线与一个三维的 CSG 体相交被简化成一个一维数字列表与一个 CSG 体的一维部分的交。首先考虑一个例子。图 12.84 给出了一条光线在标出的时间入射和出射球体 S_1 和 S_2。

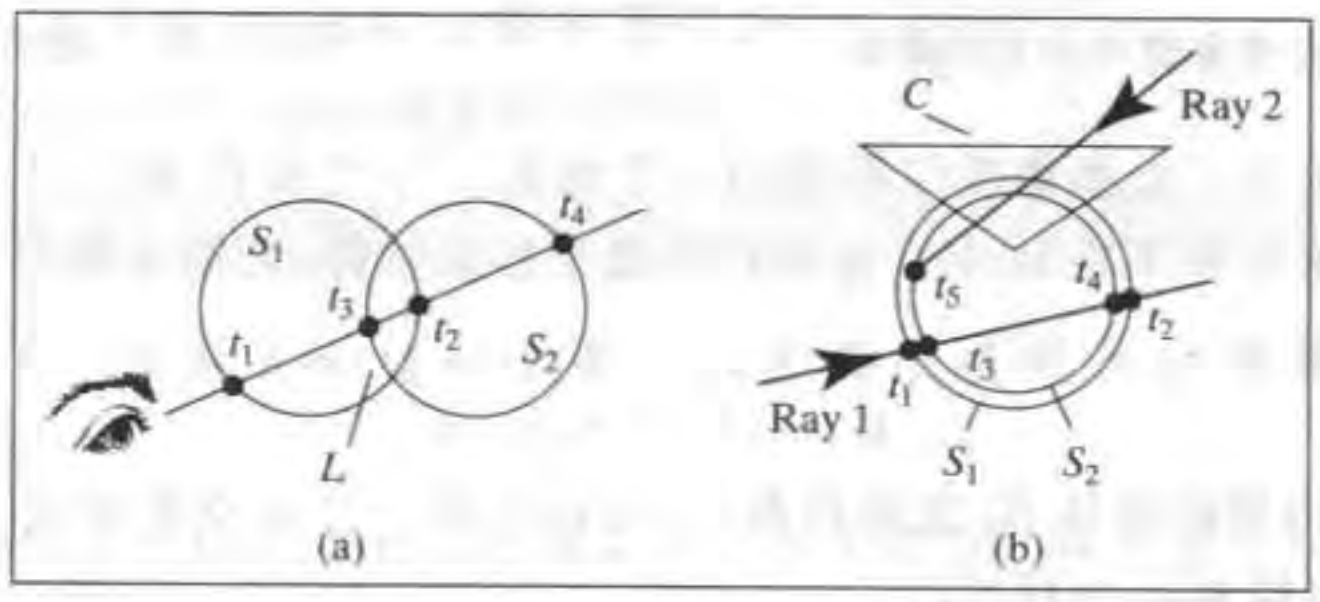


图 12.84 透镜与鱼缸的光线跟踪

因此光线在透镜 L 内部的时间从 t_3 到 t_2，碰撞的时间是 t_3。如果透镜是不透明的，我们将运用熟悉的渲染规则来决定透镜上碰撞点的颜色。如果它是镜面或者是透明的，我们将用合适的角度构造新的光线用于更深的跟踪。

图 12.84(b)中的鱼缸有着类似的情形。一方面，光线 1 首先与鱼缸在 t_1 相碰撞，最先相交的是 S_1 而不是 S_1 和 S_1。另一方面，光线 2 首先与鱼缸在时刻相碰撞。同样最先相交的是 S_2 而不是其他球体和锥体。最先碰撞的是组成鱼缸的分支物体，但不是鱼缸本身。

我们将这些想法组织成一个可以光线跟踪任何复合体的算法【Roth82】。考虑两个物体，A 和 B，还有一条光线。构建一个光线入射和出射 A 的时间列表，并按升序排列。因为物体是固体的，入射和出射时间轮流出现。为 B 构建一个类似的进出时间列表。图 12.85 给出了两个这样的列表。

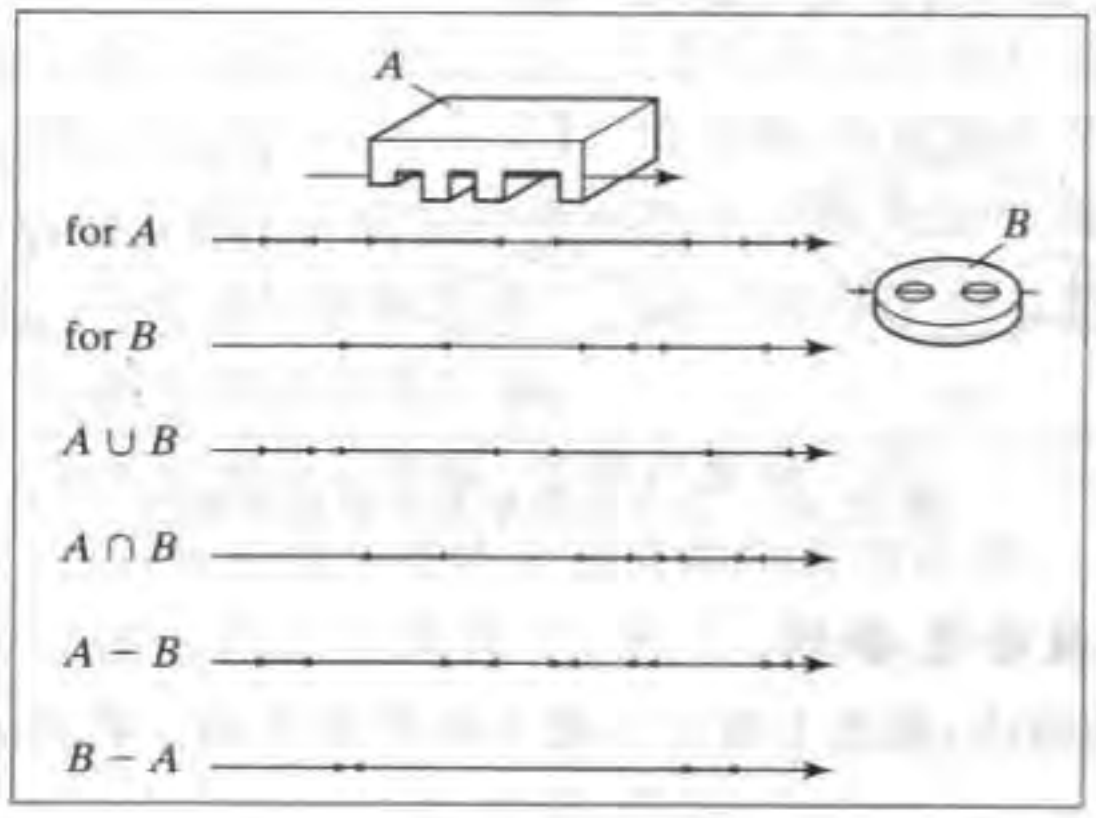


图 12.85 光线与两个物体的 t—值列表

入射的时间和下一个出射时间之间的区间用实线表示；否则用虚线表示。在实线区间当中，光线在物体的内部。如果在某个 t—值集合里光线处在物体的内部，我们则称其为**内部集合**。一个内部集合由一个包含入射与出射时间的列表(t_1，t_2，…)来指定，其中 t_1 是入射时间，t_2 是一个出射时间，以此类推。有时候称这个为光线的 t—列表。

给定一条光线与两个物体 A 和 B 的内部集合，我们想为光线与由 A 和 B 组成的复合体找到内部集合。我们可以构造的四个新物体为 $A \cup B$，$A \cap B$，$A - B$ 和 $B - A$。考虑 $A \cup B$。光线在物体并集内部当且仅当它处于至少一个物体的内部。因此 $A \cup B$ 的内部集合可以简单地定义为原来单个内部集合的并集。同样的思想可以推广到其他三种物体上。$A \cap B$ 的内部集合是原来单个内部集合的交集；$A - B$ 的内部集合定义为 A 的内部集合与 B 的内部集合的差集，同样地定义 $B - A$ 的内部集合(只需交换 A 与 B 的位置)。总的来说，如果我们记 $T(A)$和 $T(B)$为两个物体 A 和 B 的内部集合，如等式(12.50)所示，

$$T(A\ op\ B) = T(A)\ op\ T(B) \tag{12.50}$$

其中 op 为布尔操作 $\cup$，$\cap$ 或 $-$ 之一，那么复合体 $A\ op\ B$ 的内部集合即为 $T(A)\ op\ T(B)$。总之，如上所示，对物体进行布尔操作等价于与列表中的区间做同样的布尔操作。

虽然我们最终仅仅对内部集合的第一项(第一次碰撞)感兴趣，但我们必须在整个构建过程中为内部集合维持整个列表，因为最终的首次碰撞时间可能会被某个中间的列表所改写。值得庆幸的是，我们已经为光线跟踪中构建内部集合做好了准备：hit()方法在一个以时间排列的数组 inter.hit[]里为每种形状记录了光线的所有碰撞行为。从这个方法看来，我们只用到列表里的第一次碰撞时间。尽管如此，当我们开始光线跟踪布尔物体时，我们还需要利用整个列表。

例 12.13.1　构造内部集合

考虑下列两个内部集合的列表：

A_list：1.2 1.5 2.1 2.5 3.1 3.8

B_list：0.6 1.1 1.8 2.6 3.4 4.0

我们将运用上面的规则构造四个新的内部列表(查看它们的准确性)：

$A \cup B$ = 0.6 1.1 1.2 1.5 1.8 2.6 3.1 4.0

$A \cap B$ = 2.1 2.5 3.4 3.8

$A - B$ = 1.2 1.5 3.1 3.4

$B - A$ = 0.6 1.1 1.8 2.1 2.5 2.6 3.8 4.0

为了构造新的内部列表，我们需要一个函数 combineList()，这个函数将一个布尔操作和两个 t—列表作为参数，按照输入的操作构造相应的新的内部列表。我们将在后面讨论这个函数是如何工作的。

复合体的光线跟踪过程概括起来分下面几步，光线跟踪各个分支物体，为每个分支构建内部集合，最后将这些内部集合复合。复合后的列表中最先碰撞的时间就是光线最先碰撞到复合体的时间。接着运用普通的渲染技术来绘制，包括产生第二条光线，如果曲面为光亮面或透明面。

练习

练习 12.13.3 寻找向部列表

假设给定一条光线,物体 C 和 D 有如下 t 一列表:

C_list: 0.8 1.7 2.2 2.9 4.7 5.55

D_list: 1.2 1.5 2.1 2.5 3.1 3.8 4.7 8.3

找出 C 和 D 的四种可能布尔复合体的 t 一列表。

12.13.2 布尔物体的数据结构

在一个程序里我们如何表达类似 $((A\cap B)-C)-D$ 的布尔复合体?因为复合体总是由两个(可能也是复合体)物体复合而成,$Obj_1\ op\ Obj_2$,因此二叉树结构能很好的表示此类物体。图 12.86 给出了与物体 $(((B_1\cap B_2)\cup C_2)\cup((S_1-S_2)-B_3))-C_1)$ 相关的树的结构。每个内部的节点(用圆圈表示)代表一种操作,每片叶子(有矩形表示)代表一种原始的形状。一棵树由若干子树组成,这些子树也由原始形状或子树组成(这个过程无限进行下去)。当一个节点包含差集操作时,这被理解为这个节点构造了(lefttsubtree)-(rightsubtree),与(rightsubtree)-(lefttsubtree)相反。

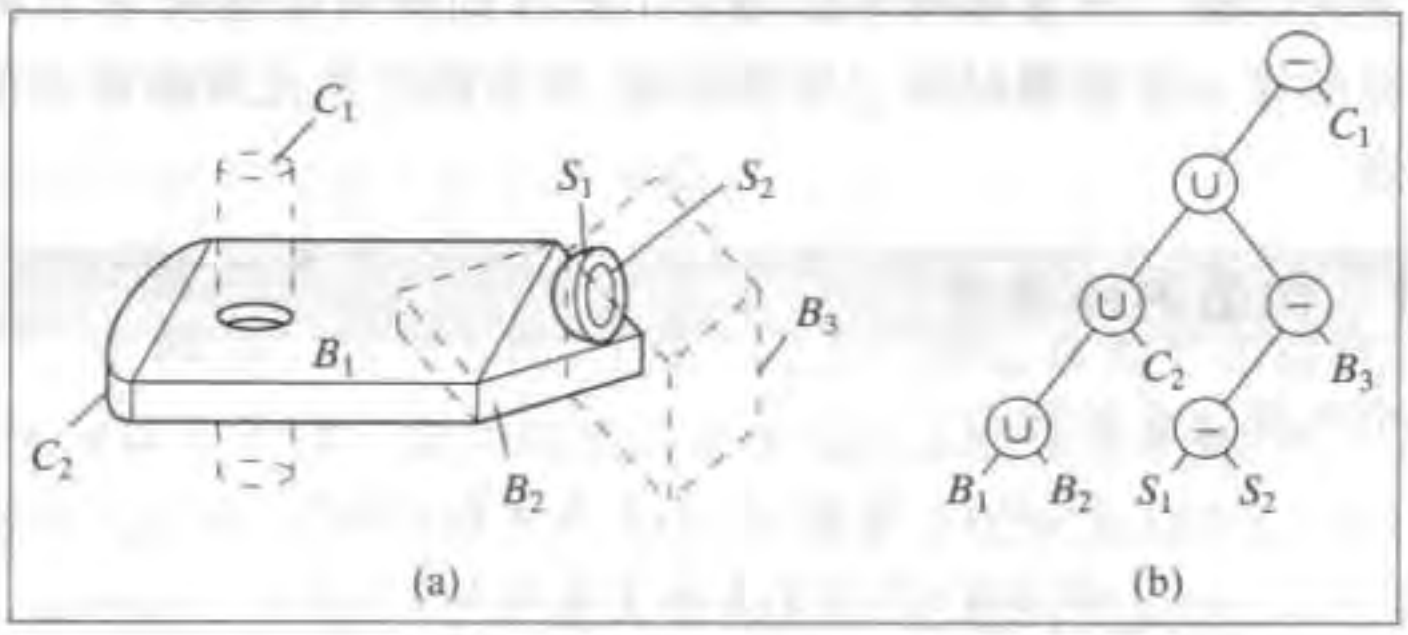


图 12.86 复合体和它的 CSG 树

复合体的表达式可以被重新排列而不改变最终表达的物体形状。每次重新排列将产生不同的树结构。因此一个给定的复合体不是只有一棵唯一的树来表示。

到目前为止,所有的物体类型例如 Sphere 和 Cube 都是由类 Shape 而来,而这个类又是从基本类 GeomObj 而来(见附录 3 的继承关系)。从 GeomObj 得到类 Boolean 看起来是很自然的事情,这样物体列表(一系列 GeomObj 的列表)就可以记录一个布尔体。一个布尔体必须可以记录一棵二叉树,因此我们给它两个指针 left 和 right 指向它的子树。这些指针必须指向 GeomObj 类型,因为一些情况下指针会指向集合图形,另一些情况下会指向布尔树。因此类 Boolean 的结构应该是这样的。

```
class Boolean : public GeomObj{
    Public:
      GeomObj *left, *right;                          //指向子分支的指针
      Boolean(){left=right=NULL;}                     //构造函数
      virtual bool hit(Ray &r, Intersection &inter);
```

```
    …其他方法…
};
```

GeomObj 类型拥有球体和长方体包围盒，布尔体也是，这个很重要，因为对布尔体光线跟踪的花费是很大的，任何加速的技术都是受欢迎的。我们应该为布尔体设计一个特殊的可以判断光线与布尔体相交的 hit()函数，使用多态性来简化代码。实际上，因为 hit()必须对并集、交集和差集做不同的操作，这样将会产生适用于这三种布尔体的不同类：UnionBool、IntersectionBool 和 DifferenceBool。UnionBool 的定义简单的表示如下。

```
class UnionBool: public Boolean{
    public:
      UnionBool(){Boolean();}        //构造函数
      virtual bool hit(Ray &r, Intersection &inter);
      …其他方法…
};
```

其他类的定义类似。每种类型将包含一个适用于不同物体性质的 hit()函数。

布尔体没有自己的仿射变换；只有 Shape 类型有。这样使得一棵布尔树只有在叶子的部分拥有变换。这种情况既带来了好处，也带来了坏处，事实上一些建模系统和光线跟踪算法有不一样的操作。为了理解这里的差别，考虑图 12.87(a)，它给出了一棵布尔体的树，它的每个内部节点和叶子都有自己的变换。每个变换可以影响它的子树中的所有节点。我们将在练习中讨论，这棵树在几何上与图 12.87(b)的树等价，即它们表示的形状是一样的。这些变换将一直被过滤到叶子节点，然后组合成一个新的变换。

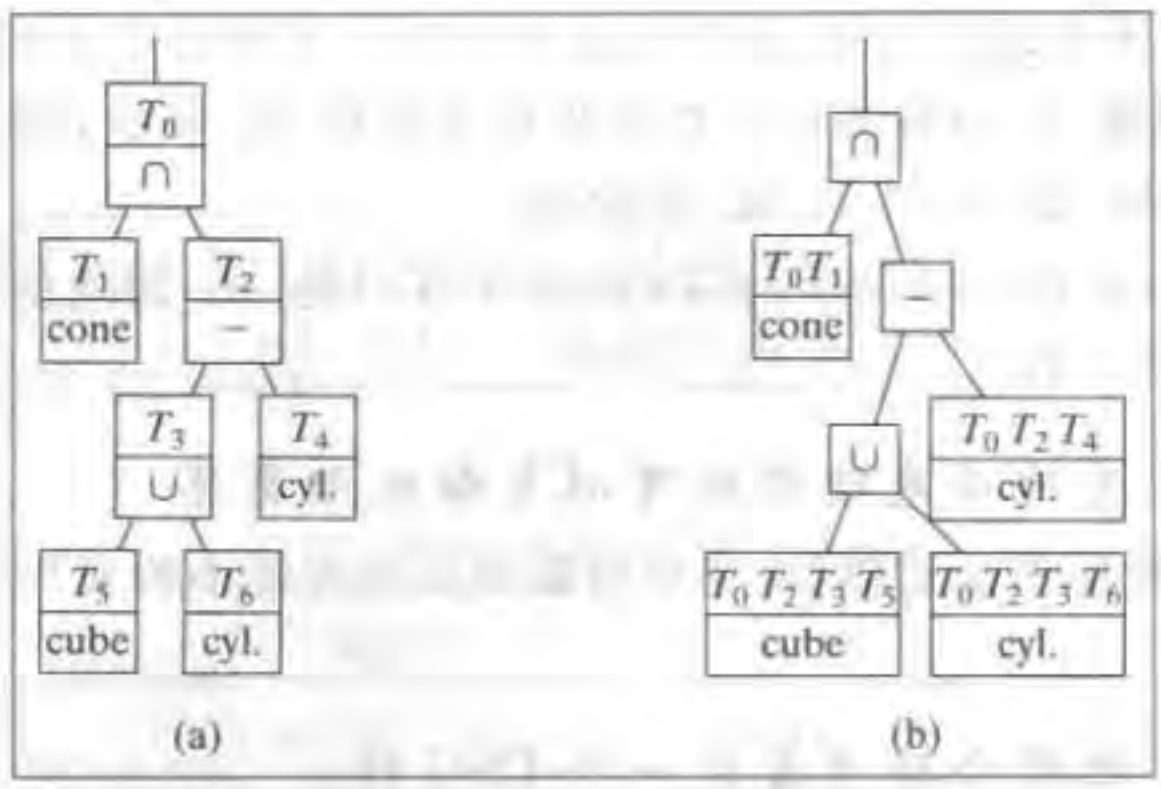


图 12.87　复合体包含变换的内部节点

保持节点中的显式变换的一个好处是，在 CSG 体被构建之后，设计者可以单独使用树中的一个变换以便于调整形状。这也为寻找布尔体的更紧致的包围盒提供了可能。一个重要的弱点是光线跟踪的速度，因为每一条光线必须在树的每个节点，不仅仅是叶子部分，逆变换一次。

在 SDL 文件里指定布尔体的方法是使用下面一种关键词：union、intersection 或 difference，然后指定左边的几何体，再指定右边的几何体(它们自身也可以是布尔体)。例如，一个如图 12.84(b)的鱼缸，由$(S_1-S_2)-C$给出，可以由如下的 SDL 代码指定。

```
!make a fish bowl
rotate -90 1 0 0
difference
    difference
        sphere !outer sphere
        push scale 0.9 0.9 0.9 sphere pop !inner sphere
    push translate 0 0 1.3 rotate 180 1 0 0 cone pop
```

第一个变换将整个鱼缸围绕 x 轴做了一次旋转。然后在左边物体(它本身是两个球体的差集)和锥体之间做差集。因为给予 SDL 的变换将在后面与当前变换做叠加,而当前的变换在形状被构建的同时就放置在了其中,所以变换将自动地往树的下面传播。

练习

练习 12.13.4 重新排列布尔体表达式

因为写出一个复合体的表达式有机动性,可以有不止一棵树来表达它。例如∪和∪等价,因此 $A\cup B=B\cup A$。判断下列等式的正误,并证明。

a. $(A-B)-C=A-(B\cup C)$

b. $(A\cup B)-(A\cup C)=A\cup(B-C)$

c. $(A\cup B)\cap(A\cup C)=A\cap(B\cup C)$

练习 12.13.5 变换是如何在布尔树中向下传播的

考虑图 12.84(a)中的树,每个内部节点都有一种变换。

a. 判断并证明这棵树所表示的几何形状与图 12.84(b)所表达的是否一样。证明过程依赖仿射变换么?

b. 如果图中的变换 T_i 由在各向同性的坐标中矩阵 M_i 表示,图 12.84(b)中立方体的矩阵组合与 $M_0M_2M_3M_5$ 或 $M_5M_3M_2M_0$ 等价吗?

提示:证明 $T(A\ op\ B)=T(A)\ op\ T(B)$其中 $T()$是一一对应的变换,A 和 B 为点集,op 代表∪,∩或−中的一种。

练习 12.13.6 为什么光线跟踪可以自动处理并集

解释为什么光线跟踪两个物体(两物体可能相交也可能不相交)之后的并集等价于光线跟踪它们的并集。

练习 12.13.7 把整个场景看成一个 CSG 体

讨论将整个场景看成一个复合体的好处与不好之处,与使用一个简单复合体列表做比较。

12.13.3 布尔体与光线的交

我们需要开发一个适用于各种布尔体的 hit()方法。它必须可以构建光线与左右子树的内部集合,并且可以恰当地将它们组合起来。

图 12.88 给出了交集操作下的 hit()函数的一个框架结构。包围盒测试判断是否可以提前退出函数(将在下节讨论)。然后左边子树调用合适的 hit()函数,除非光线与此子树不相交,不然构建碰撞列表 lftInter。如果它们不相交,函数 hit()返回 false,这是提前退出,因

为光线要与两棵子树的交集相交必须与它们同时相交。如果相交，我们再构建一个右子树的碰撞列表 rtInter，最后用交集操作将两个碰撞列表组合起来。

```
bool IntersectionBool::hit(Ray &r, Intersection &inter)
{
    Intersection lftInter, rtInter;
    if(光线与包围盒不相交) return false;
    if((!left->hit(r, lftInter)) || (!right->hit(r,
rtInter)))
        return false;                //提前退出
    构建复合列表：将其放置到 inter 内
    return(inter.numHits>0);     //
}
```

图 12.88　交集操作下的 hit()函数的一个框架结构

其代码与类 UnionBool 和 DifferenceBool 的相似。

构建复合 t—列表

构建复合 t—列表是复杂的但是符合逻辑的。考虑两个碰撞列表的例子，L 与 R（分别代表左和右），如图 12.89 所示。为了简单起见，我们记左边的列表为 L[]而不是 leftInter.theHit[]，右边列表相类似。而且，hitObject 和 surface 省略。L 里记录了 8 个碰撞，R 记录了 5 个碰撞。每个列表包含排列过的碰撞时间，结构中第 0 个元素记录了当光线入射或出射物体时，第一次碰撞时间是否为正。因为物体是固体，所以在第 0 个元素之后入射与出射的值交替出现。

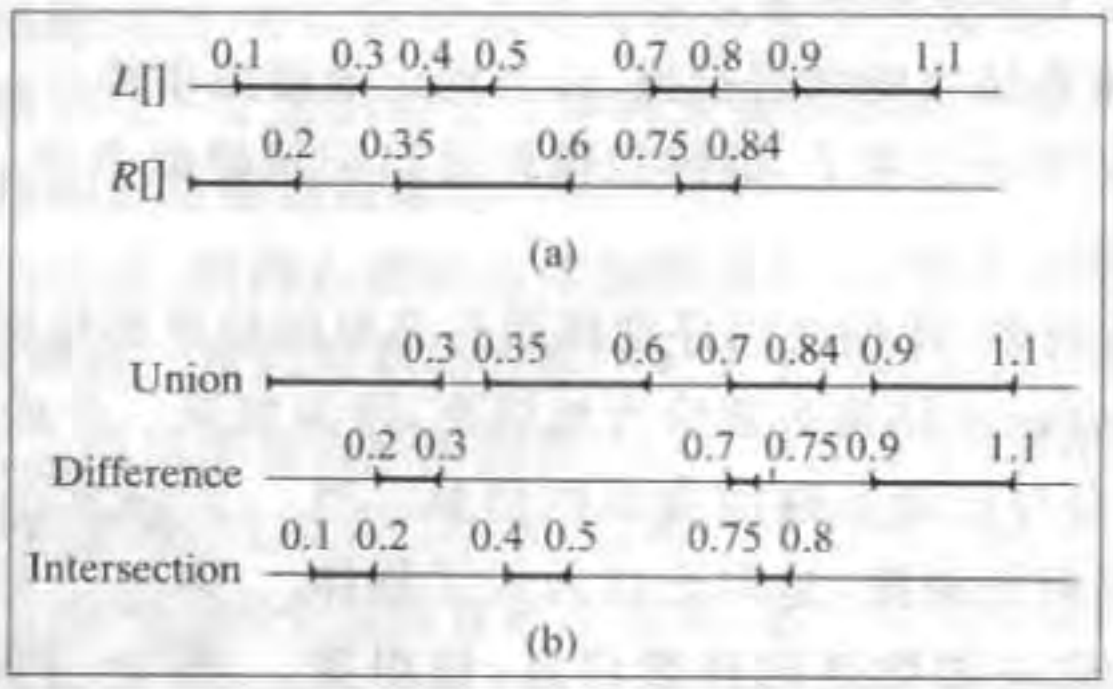


图 12.89　相交区间的例子

复合列表的构建是通过扫描这两个列表来实现的，在每一步中考察两个列表后面的哪一项拥有更小的碰撞时间，并在这个过程中保持光线在物体内部还是外部的跟踪。这些信息将通过不同布尔操作所对应的规则来组合。图 12.89(b)给出了三种不同的布尔组合所对应的复合 t—列表。

考虑一个特定的构建复合列表的例子。假设在差集的操作中，光线在碰撞下一个面之前是处在左边物体的内部和右边物体的外部，因此它处在它们的差集当中。检查每个列表

中的下一个碰撞时间;假设右边列表的下个碰撞时间更小。这意味着在下次碰撞之后,光线将进入右边物体,因此它现在应该在它们差集的外部。在这种情况下,光线的状态从在差集外部改变为在差集的内部。

当光线伴随下列变量从子树中通过时,我们保持跟踪它的状态(无论其现在处在物体的内部还是外部)。

```
bool lftInside;          //值为真,如果光线在左边物体内部;
bool rtInside;           //值为真,如果光线在右边物体内部;
bool combInside;         //值为真,如果光线在复合体内部;
```

这些是光线在碰撞之间的状态——即在下次碰撞之前的状态。lftInside 的初始值依照光线第一次碰撞时是射入还是射出左边物体——即依照 L[0].isEntering 的值。具体的,如果首次碰撞之后光线射入物体(在碰撞之前光线在物体外部),lftInside 的值即为 false;否则 lftInside 的值为 true。因此我们可以将 lftInside 初始化为! L[0].isEntering,rtInside 也类似。然后 combInside 被定义为 lftInside 和 rtInside 的一个逻辑上的组合;如果操作为并集,combInside 为真当且仅当 lftInside 或 rtInside 或两者都为真。这可以很简单地运用逻辑运算操作来实现:combInside=lftInside || rtInside。类似地,如果是交集操作,我们使用 combInside=lftInside && rtInside,如果是差集操作,我们用等式 combInside=lftInside && ! rtInside。

此算法过程不断地扫描列表 L[]和 R[],在每一步寻找较小的碰撞时间,用它来更新 lftInside,rtInside 和 combInside。如果 combInside 的值在某点处改变了,那么最后一次碰撞事件将被添加到列表 C[]中。

当列表 L[]或 R[]已经到达尽头(当它的最后一个碰撞时间被扫描到),其他没有结束的列表有可能不用再考虑:

- 交集时:忽略没有结束的列表;
- 并集时:继续没有结束的列表扫描,适当地记录碰撞事件;
- 差集时:如果左列表结束了,继续扫描并适当记录碰撞事件;如果右列表结束,忽略没有结束的列表。

注意在复合列表的时候,我们必须仔细将所有必要的信息都传递给每个新的 hit[]。特别地,isEntering 和 hitObject 结构必须恰当地填充,因为如果一个碰撞称为第一次碰撞,那么我们将从 hitObject 自身记录下碰撞曲面的信息。当右边物体发生了一个出射的碰撞,这时光线射入了物体自身,我们必须清楚指向左边物体的碰撞信息,使得我们透过洞看过去的是左边的物体。

当物体是透明的而光线在布尔体里面时(例如当光线穿过一个马提尼酒杯),我们也必须确保 CSG 物体的逻辑函数 hit()被正确执行。对于上面不同的物体,即使光线刚刚进入物体,正确的碰撞信息也会被放置在碰撞列表中。

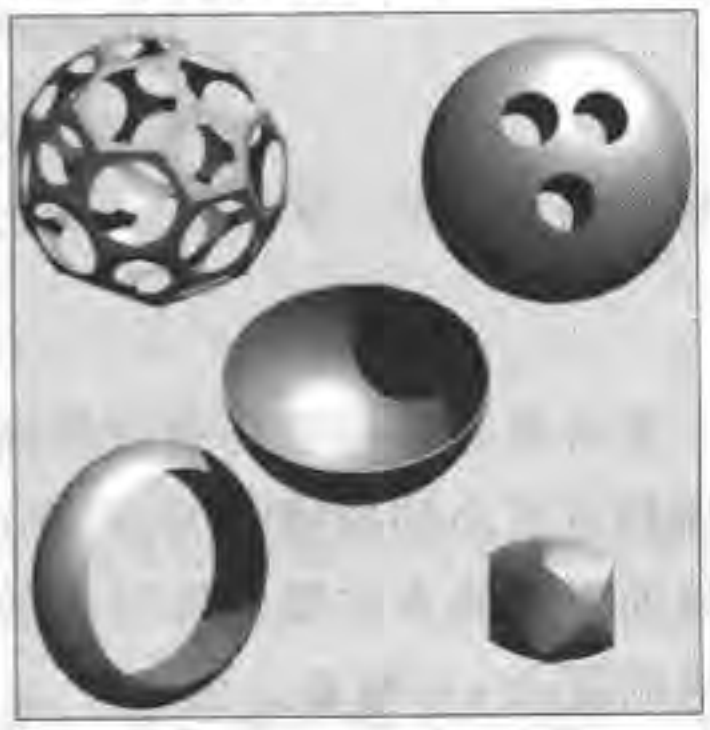


图 12.90 光线跟踪复合物体的例子

图 12.90 给出了用这些技术来光线跟踪多个复合

物体的例子。试着确定给定的布尔物体是由哪些物体构成的。

练习

练习 12.13.8　估计算法结果

利用图 12.88 中的复合列表的算法估计例 12.13.1 中几个例子的运算结果。

练习 12.13.9　编写函数 hit()

为类 UnionBool 和 IntersectionBool 实现对应的 hit()函数。每个函数必须很好地运行在指定的布尔操作上。

12.13.4　为 CSG 物体构建和使用包围盒

还剩下一个问题，那就是为 CSG 物体构建投影、球体、长方体包围盒。在预处理阶段，我们扫描 CSG 物体对应的树，为它的每个节点构建包围盒并且储存在节点自身当中。在随后的光线跟踪中，光线先同遇到的包围盒做求交测试，如果我们可以确定光线与物体不相交，函数就可以提早退出。这样做可以节省复合列表的花费。

在图 12.88 中提到的，包围盒测试总是放置在物体类的 hit()函数当中的。如果光线与当前节点的包围盒相碰撞，像通常一样我们认为它与左右子树相交。

构建长方体包围盒

一个 CSG 节点的合理的长方体包围盒的定义是怎样的呢？这里，当然，指的是世界坐标下的长方体包围盒，因为一个布尔体没有一般的视图。假设一个节点代表物体 $A\ op\ B$，这里 A 和 B 是形状，op 是一种布尔操作。物体的长方体包围盒必须包裹整个物体，使得如果光线与包围盒没有碰撞，我们可以肯定光线与物体也没有碰撞。物体的形状可能很复杂（例如，一个旋转过的圆环与一个歪斜的立方体的交集），因此在没有经过复杂的处理，要自动找到物体最紧致的包围盒是相当困难的。

我们考虑最简单的情况，利用 L 的长方体包围盒 $E(L)$ 和 R 的长方体包围盒 $E(R)$ 来构建 $L\ op\ R$ 的长方体包围盒。我们为不同的操作定义不同的长方体包围盒。

(a) **并集**：定义长方体包围盒为同时包裹 $E(L)$ 和 $E(R)$ 的调整过的长方体。这个等价于 $E(E(L)\cup E(R))$。（问题：这跟 $E(L\cup R)$ 总是一样的吗？这个比分别对 $E(L)$ 和 $E(R)$ 求交更有效吗？如果光线与这两个包围盒都不相交，它一定与 $L\cup R$ 不相交吗？）

(b) **交集**：定义长方体包围盒为 $E(L)$ 和 $E(R)$ 的交集。两个调整过的长方体的交集总是一个调整过的长方体，并且很容易计算它的（*left*，*top*，*right*，*bottom*，*front*，*back*）分量。（问题：如果光线与 $L\cap R$ 的包围盒不相交，它一定与 $L\cap R$ 不相交吗？）如果 L 和 R 的包围盒是紧的，那么这个交集的包围盒也被认为是紧的。

(c) **差集**：简单定义长方体包围盒为 $E(L)$。这是很保守的做法，因为 $E(L-R)$ 有可能比 $E(L)-E(R)$ 小。这样 $E(L)$ 可能会出现很大一块空间，但是做一个更精确的分析会花费很多。

长方体包围盒将递归地求得，因为一棵布尔树的内部节点的长方体包围盒将从它的子

节点的包围盒中求得。如果是简单形状,它的包围盒将不需要递归地构建,我们前面提到过:在世界坐标中求物体的点云,然后构建一个包裹它的调整过方向的长方体包围盒。每种形状类型拥有不同的makeBoxExtent()函数。每个函数都构建一个长方体包围盒并且保存在物体中,以备其他物体的调用。例如,类UnionBool可能用下面的代码。

```
Cuboid Unionbool::makeBoxExtent()
{
        Cuboid lft=left->makeBoxExtent();
        Cuboid rt=right->makeBoxExtent();
        Cuboid tmp;
        tmp.left=min(lft.left, rt.left);          //构建并集
        tmp.top=max(lft.top, rt.top);
        //等等,其他四个分量的定义
        worldBoxExtent=tmp;                       //将其保存到物体中
        return tmp;
}
```

这个处理过程为物体列表中每个物体都调用makeBoxExtent()函数。当一个布尔体调用这个方法的时候,它像每个中间节点一样,构建并在物体中保存长方体包围盒。

构建球体包围盒

世界坐标下的球体包围盒的构建与长方体包围盒的构建类似。像前面讨论的一样,为每个形状求世界坐标下的点云,球体包围盒是最适合这个点云的球体。对于布尔物体,我们必须判断如何构建两个物体的并集、交集和差集的球体包围盒。

差集很简单;用左边物体自身的球体包围盒。它不会非常紧,但是它是适用的。交集可以同样使用,因为为物体交集求真正的球体包围盒相当困难。并集情况可以先将两个物体的点云组合,然后求复合点云的球体包围盒。

构建投影包围盒

为了减少光线跟踪的花费,运用投影包围盒是非常必要的。一个CSG体的每个节点的投影包围盒的构建与其长方体包围盒的构建是一样的。一个节点的投影包围盒由其左右子树的投影包围盒组成。用P(object)来表示object的投影包围盒,按照上面的规则我们有如下表示:

$$P(L \cup R) = P(P(L) \cup P(R))$$
$$P(L \cap R) = P(L) \cap P(R)$$
$$P(L - R) = P(L)$$

像前面一样,每个类有其自身的makeBoxExtent()函数(只用于光线跟踪)。

12.14 光线跟踪与光线投射

光线投射方法始于1992年ID软件公司的游戏Wolfenstein 3D。在Wolfenstein 3D中玩家被放置进一个像迷宫一样的环境中,在这里玩家可以在房间里移动与魔鬼和敌人战斗。

虽然光线跟踪是一种可以产生适用与各种场景的高质量的图像的算法，但它非常费时，因此没有办法适用于实时绘制，例如游戏和实时模拟。而光线投射是一个非常快的技术，它只能处理受到严格几何约束的场景并且产生的图像是斑驳的、不真实的。

图 12.91　使用光线投射算法的游戏 Wolfenstein 3D 的截图

图 12.91 给出了一些 Wolfenstein 3D 的游戏截图，展示了那些可以运用光线投射算法的场景类型。注意斑驳的、粗糙的图像质量和纹理。近些年随着图形硬件的更新，光线跟踪算法的限制变得不重要，光线投射算法基本上被抛弃了。F. Permadi 的网站上有对光线投射算法的详细介绍。

本章小结

光线跟踪算法提供了一个简单的均匀的绘制华丽的真实图像的方法。场景中大量相关光线被跟踪，分析每一条光线最终返回观察者眼睛的光亮度。这个机制为绘制多种视觉效果提供了可能。光线跟踪不但可以模拟环境光、漫反射光和镜面反射光，还可以模拟相关的阴影、镜面反射和透明物体的投射。光线跟踪还为二维和三维场景提供了一种自然的纹理映射方法。

光线跟踪算法的核心是求光线与物体是否相交的程序。为了绘制高质量的图像，巨大数量的光线被构建，大部分光线还要产生若干条二次光线，这样使得大量的时间用于求交测试。因为每条光线与场景中每个物体都要求交，光线跟踪的时间花费与场景中的物体数量成线性关系。包围盒技术简化了这一过程，它可以提前判断大量光线与物体是否相交，这样可以避免大量的无谓的求交计算，于是提高了光线跟踪的速度。

光线跟踪算法自动解决隐藏面的绘制问题，因为一条光线与各种物体的碰撞时间是它们与观察者距离的一个直接的度量。因此拥有最小碰撞时间的物体是最近的一个，它不会被其他物体所遮挡。相同的规则可以用来直接计算阴影。从光线与物体的碰撞点发出一个阴影探测器，它指向光源，每个物体的任何碰撞都做测试。如果一个物体处在碰撞点与光源之间，那么这个点就在与这个光源相关的阴影中。因此，检测这样的阴影相当于检测物体与阴影探测器的交点是否处在 0 和 1 之间。

一些形状容易做光线跟踪，因为求光线与其的交点只需要求解一个线性或二次等式。这一类的典型是平面、球体、圆柱体和圆锥体。场景设计者从普通视图开始，然后用仿射变换将它们变换到一个设计需要的大小、朝向和位置。这样可以将圆球体变成椭球体，将立方体变成平行六面体，等等，这样就扩充了物体的种类。任意凸多面体的光线跟踪也很简单，因为它的内部几乎可以看成一系列半平面的交集。

光线是一类容易处理的，有很好性质的物体，当它经过仿射变换，得到的还是一条光线。因此为了检测哪些物体与它相交，从一个给定的起始点开始产生一些新的光线是不难的。这提供了一个直接的方法来累加光滑曲面的反射和透明物体的折射。实现这样的技术也是很简单的。

为光线跟踪添加跟踪 CSG 体的方法增加了需要渲染的场景的种类。十分复杂的物体

可以被构造和渲染。光线跟踪一个 CSG 体的内在逻辑是很直观的,并且容易实现。

今天,光线跟踪是最强大的绘制计算机合成的图像的方法之一。虽然巨大数量的计算使得它的运行速度很慢,但是它所产生的图像的真实度是其他方法所没有办法达到的。

案例分析

案例 12.1 放射型光线跟踪器

(难度级别:Ⅱ)编写并执行一个应用程序,实现对 SDL 中所描述场景的光线跟踪。该光线跟踪器只对自身发光的物体进行适当处理。不要求其计算环境光、漫反射光和镜面反射光分量。该程序应该至少能够跟踪球体、圆锥体、立方体、正方形和平面。SDL 场景文件中包含的其他形状可以忽略。光线跟踪开始之前利用 Scene::drawOpenGL()方法绘制出预览图。

利用键盘按键来控制各项功能。如 12.5 中讨论的,按键 1、2、4、8 用于设定像素块的尺寸,使得场景在不同分辨率下可以切换;其他按键则用于调整照相机的位置和方向。

还要构建一个 SDL 的场景,其中包含了几个插孔和八角体,并适当调整位置以便光线跟踪。每个插孔有三个互相垂直的长条,这里的长条是指一个小长球体,她的两端各有一个小球。八角体由一些小球组成,它们位于一个八面体的 6 个顶点上,而顶点之间的 12 条边则由小长球体组成。

案例 12.2 一种新型光线跟踪器

(难度级别:Ⅱ——假设实例分析 12.1 中的光线跟踪器已经存在)对其补充改进,是的它能够正确处理环境光、漫反射光和 Phong 镜面高光。它不需要处理反射、折射和阴影效果。它可以正确地跟踪所有的通用形状以及凸多边形网格。和上面一样,也要在光线跟踪之前绘制出场景的预览图。

为了实现进一步控制,请编写代码,使用户可以利用鼠标指定显示器上的矩形区域,然后在对指定的区域进行最大分辨率的光线跟踪(块尺寸=1)。这方便了那些需要快速光线跟踪,并且只需要在一个指定范围内进行最大分辨率光线跟踪的用户。

案例 12.3 在光线跟踪中实现阴影效果

(难度级别:Ⅰ——假设实例分析 12.2 中的光线跟踪器已经存在)请对其添加能显示阴影效果的功能。

案例 12.4 用包围盒加速光线跟踪算法

(难度级别:Ⅱ——假设实例分析 12.3 中的光线跟踪器已经存在)请在光线跟踪中加入球体包围盒、长方体包围盒和投影包围盒,并比较使用每种包围盒所起到加速计算的作用。允许用户使用一个按键来控制包围盒的生效和失效,从而确定每种光线跟踪花费的时间。

案例 12.5　对三维纹理的光线跟踪

(难度级别：Ⅱ——假设实例分析 12.2 中的光线跟踪器已经存在)改进光线跟踪，使其可以支持一些三维纹理显示，包括棋盘格、木头纹理和大理石纹理。用户使用不同的按键来选择纹理，然后再利用所选择的纹理来渲染场景中的所有物体。

确定一种方法，用于扩展 SDL 和方法 Scene::read()，使得 SDL 文件中的参数能够被指定并与随后定义的物体相关联。使用参数来选择物体的纹理，并指定纹理定义中的一些数值参数。例如，SDL 里：

```
parameter 3 12.6 -91
cube
```

可能关联到有关立方体的三个参数，所以将纹理 #3 关联到此物体。

案例 12.6　反走样

(难度级别：Ⅱ——假设实例分析 12.2 中的光线跟踪器已经存在)扩展实例分析 12.2 中的光线跟踪器，使它能够实现反走样处理。对于每条主光线，它将发射 N 条方向略微不同的光线，并对这些光线的返回值取平均。请取不同的 N 值，并在光线方向上添加不同数量的抖动，分别进行实验。

案例 12.7　对其他基本形状的光线跟踪

(难度级别：Ⅲ——假设实例分析 12.2 中的光线跟踪器已经存在)请研究一下 Jim Kajiya 关于对形如棱柱、不规则山脉以及旋转面等复杂形状的光线跟踪的文章。要求至少可以实现对分形山的光线跟踪，并使用关键字 mountain 以及一些用于指定山脉的属性的参数来扩展 SDL。

案例 12.8　一个用于研究折射的二维光线跟踪器

(难度级别：Ⅱ)因为在等式(12.45)中关于折射方向 t 的等式只用到点积，所以它同时适用于二维和三维的情况。编写并执行一个程序，产生类似于图 12.63 所示的图片。当用户给定折射率和入射角时，该程序绘制出入射和折射光线。

案例 12.9　反射光和折射光

(难度级别：Ⅲ——假设实例分析 12.2 中的光线跟踪器已经存在)扩展此光线跟踪器，使其能够正确处理光亮面的反射光线和透明物体的折射光。采用不同的折射率对不同的透明形状进行实验，从而研究折射现象的本质属性。

案例 12.10　光线跟踪物体的布尔组合

(难度级别：Ⅲ——假设实例分析 12.2 中的光线跟踪器已经存在)扩展实例分析 12.2 中的光线跟踪器，使其能够正确跟踪任意复杂的布尔体。创建一个包含此类物体的有趣的场景，包括：

- 内部有个圆柱形孔的物体;
- 一个球面棱镜 $A\cap(B-C)$(由两个球体相交而成);
- 由 $A\cap(B-C)$确定的物体,其中 A、B 和 C 均表示形状。

适当调整它们的位置,使得能够透过其他一些对象中的孔来看到某个物体,并且透过透镜看到某些物体。

进一步阅读

由 Andrew Glassner 编著的"An Introduction to Ray Tracing"【1989】,提供了大量优秀的论文,这些文章研究了用于光线跟踪的基本算法以及加速光线跟踪算法的一些方法。"Graphic Gems III"【Kirk92】提供了一些有用的有关光线跟踪的文章。Watt 和 Watt【Watt92】对光线跟踪以及相关的各种方法进行了更深入和广泛的研究。因特网上(http://www.irtc.org/)的光线跟踪网站搜集了一些光线跟踪的图像,并提供了有关光线跟踪的想法和动机。

附录 1

图形工具：怎样获取和安装 OpenGL

本书中所讨论的大部分代码是基于以 OpenGL 为应用程序接口(API)的。本附录将介绍在使用 OpenGL 构建自己应用程序时，如何获取所需的 OpenGL 工具。

A1.1 获取和安装 OpenGL

对于目前常用的任何一种计算机平台，获得必要的 OpenGL 软件都是直接且免费的。我们将针对各种主要的平台，讨论如何获取该软件。此外，因特网上提供了大量关于 OpenGL 的补充信息。

OpenGL 信息库

http：//www.opengl.org/上提供了大量关于 OpenGL 的一般信息以及软件下载的起始网址。从 http：//opengl.org/documentation/上也可以获得包括一些在线手册在内的许多信息。本书的网站(见前言)上提供了关于 OpenGL 的附加信息和连接。OpenGL Architecture Review Board (ARB)出版的 OpenGL Programming Guide(经常被称作"红宝书")是一本很优秀的关于使用 OpenGL 的必读书目。该书介绍了 OpenGL 软件的获取和安装，目前已经有第 5 版。

读者的需求

无论使用什么系统，你首先得有一个优秀的 C/C++ 编译器并且安装适当的 OpenGL 头文件和库。正如本书所述，为了使用 OpenGL，必须有以下三个库及相关文件：

- OpenGL(基本的 API 工具)；
- GLU(OpenGL 实用程序库)；
- GLUT(OpenGL 实用工具包，一个用于处理视窗系统操作的视窗工具包)。

一般来说，每个库都对应几个相关的文件：一个头文件(.h)、一个库文件(.lib)和在某些系统中的动态链接库文件(.dll)。

添加头文件：把 gl.h、glu.h 和 glut.h 这三个文件放在你编译器的 include 目录的 gl 子目录(你自己新建的)中。你需要在每个应用程序中写下面的包含语句：

```
#include <gl/Gl.h>
#include <gl/Glu.h>
```

```
#include <gl/Glut.h>
```

连接库文件：每个应用程序都将成为项目的一部分被编译和连接。除了自己写的(.c 或.cpp)文件之外，还应该在你的项目中添加适当的 OpenGL 库(.lib)文件以便连接器能够找到它们。

下面我们分别讨论对各种主要类型系统的要求。

Microsoft Windows 95/98/NT

大部分 Windows 机器上已经安装了 OpenGL。在写这本书的时候，Windows XP 机器上都自带了 OpenGL。

Windows 系统中使用 OpenGL 的一个合适的环境是微软的 Visual C++ 6.0 或更高版本。(如果你把应用程序声明为"win32 控制台应用程序"，就没有必要建立一个图形用户界面：键盘输入和文本输出分别通过不同的控制台窗口进行)。

单击 http://www.opengl.org/documentation/spec.html 中"Download OpenGL Specs"来获取基于 Windows 系统的 OpenGL 及其实用程序库的最新信息[①]。这些库也可以通过微软 FTP 站点提供的自解压文件 ftp://ftp.microsoft.com/softlib/mslfiles/opengl95.exe 来获得。其中包括 gl.h、glu.h、glue2.lib、opengl32.lib、glu32.lib 以及 opengl32.dll(不要使用老的.aux 文件)。

glut 库可以通过 http://www.opengl.org/resources/libraries/glut.html 获得。其中包含了 glut.h、glut32.lib 和 glut32.dll(不要使用 glut.lib 和 glut.dll)。下载并解压这些文件之后，将.lib 文件放在合适的目录下，以便将来把它们添加到你的每个工程中。

把.dll 文件放在 Windows/System 目录下(如果它们还未被放在这里)。加入下面的 include 语句将指定文件包括在你的工程中。

```
#include <windows.h>
#include <gl/Gl.h>
#include <gl/Glut.h>
#include <gl/Glut.h>
```

如果你将.h 文件放在了不同于上述的文件夹下，例如"myglut"，包括的指令就应该是 #include <myglut/glut.h>。

Macintosh

Macintosh 下的 OpenGL 主要来源是 http://www.apple.com/macosx/features/graphicsand media/。OS 9 和 X 用户应当注意 OpenGL 库和 GLUT 已经自动安装了！Opengl.org 也有关于如何在更早版本的 Macintosh 平台上运行 OpenGL 的信息。每一个 iMac、iBook、PowerBook、PowerMac G4 和 PowerMac G5 都有内建的 OpenGL 硬件加速。

---

① 翻译这本书时该网页已经无效。可以通过访问 http://www.opengl.org/documentation/specs/来获得相应信息。

UNIX 和 X Windows 系统

每一个 UNIX 工作站都支持 OpenGL。GLX 提供了 OpenGL 和 X Windows 系统之间的连接。所有的 OpenGL 实现都需要用 X。资源和下载信息可以在 http：//www. sgi. com/products/ software/opensouce/glx/上面找到。GLX 把 OpenGL 扩展到了 Linux、FreeBSD 和其他使用流行的 X 系统 XFree86 实现的操作系统。

Mesa3D

Mesa 是一个可以免费获得的类似于 OpenGL 的库。目前，最好的下载地址是 http：//www. mesa3d. org/#Systems。上面附带了说明及每一个可下载版本。另外一个选择是 Windows 下的 Cygwin。Cygwin 是 Windows 下运行的 Linux。Cygwin 允许应用程序员连接到 Windows 的 dll。Cygwin 比 Mesa 运行快，而且允许访问可能在本地 Linux 视频驱动上不能得到的视频硬件。

附录 2

计算机图形学的数学基础

本附录总结并归纳了全书所涉及的各种数学知识。有时还会提供结论的简要推导，不过这些资料主要是为了便于参考。

A2.1 一些关于矩阵及其运算的主要定义

本附录只是回顾了一些矩阵的基本概念和矩阵运算的方法。更加全面的分析可以参考其他书籍(例如，【Birkhoff76】，【Faux79】)。

矩阵是一个由元素组成的矩形数组，这里的元素通常是数字。一个 m 行、n 列的矩阵称为 $m\times n$ 矩阵。例如

$$A=\begin{bmatrix} 3 & 2 & -5 \\ -1 & 8 & 0 \\ 6 & 3 & 9 \\ 1 & 21 & 2 \end{bmatrix} \tag{A2.1}$$

是一个 4×3 的整数矩阵，而 $B=[1.34,-6.275,0.0,81.6]$ 是一个 1×4 的矩阵，也称为四元组或者向量。一般来说，一个 $1\times n$ 的矩阵就是一个行向量，而一个 $n\times 1$ 的矩阵是一个列向量。

按惯例，矩阵中的单个元素用小写字母符号表示并用下标来区分。矩阵 B 的第 ij 个元素表示为 b_{ij}。这代表了第 i 行第 j 个元素。同理，对于等式(A2.1)中的矩阵，$a_{32}=3$。如果一个矩阵行数和列数相等，就称之为方阵。在图形学中，我们经常使用 2×2、3×3 和 4×4 的矩阵。零矩阵和单位矩阵是两个常用的矩阵。零矩阵中的所有元素为零。除了主对角线($i=j$ 对应的 a_{ij})上值为 1 的元素外，单位阵中的其他元素也都是零。因此，3×3 的单位阵如下：

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

A2.1.1 矩阵运算

数字矩阵 B 可以被一个数 s 缩放。B 中的每个元素和 s 相乘。结果矩阵用 sB 表示。以等式(A2.1)中的矩阵 A 为例，我们可以得到

$$6A = \begin{bmatrix} 18 & 12 & -30 \\ -6 & 48 & 0 \\ 6 & 126 & 12 \end{bmatrix}$$

两个具有相同行数和列数的矩阵 C 和 D，称之为具有相同形状。它们可以相加。相加的和 $E=C+D$ 的第 ij 个元素是 C 和 D 中相应元素的和：$e_{ij}=c_{ij}+d_{ij}$。因此

$$\begin{bmatrix} 3 & 2 & -5 \\ -1 & 8 & 0 \\ 6 & 3 & 9 \\ 1 & 21 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 5 & -1 \\ 9 & 8 & -3 \\ 2 & 6 & 18 \\ 4 & 2 & 7 \end{bmatrix} = \begin{bmatrix} 3 & 7 & -6 \\ 8 & 16 & -3 \\ 8 & 9 & 27 \\ 5 & 23 & 9 \end{bmatrix}$$

因为矩阵可以被缩放和相加，所以定义（具有相同形状的）矩阵的线性组合是很有意义的，例如 $2A-4B$。从这些定义可以直接得出关于具有相同形状的三个矩阵 A、B 和 C 的下列结论。

$$A+B=B+A$$
$$A+(B+C)=(A+B)+C$$
$$(f+g)(A+B)=fA+fB+gA+gB$$

通过交换矩阵 M 的行和列就可以构成 M 的转置，记做 M^T。M^T 的第 ij 个元素是 M 的第 ji 个元素。因此等式（A2.1）中矩阵 A 的转置为

$$A^T = \begin{bmatrix} 3 & -1 & 6 & 1 \\ 2 & 8 & 3 & 21 \\ -5 & 0 & 9 & 2 \end{bmatrix} \tag{A2.2}$$

一个行向量的转置是一个列向量。例如，

$$(3,2,-5)^T = \begin{bmatrix} 3 \\ 2 \\ -5 \end{bmatrix}$$

如果一个矩阵和它自己的转置相等，就称该矩阵为对称矩阵。只有方阵才可能是对称矩阵。因此，一个 $n\times n$ 的矩阵 M，如果满足 $m_{ij}=m_{ji}$ 对任意 1 到 n 之间的 i 和 j 成立，它就是对称矩阵。

A2.1.2 两个矩阵相乘

第 5 章开始讨论的变换涉及到了一个向量和一个矩阵相乘以及两个矩阵的乘积。其中，前者是后者的一个特例。

只有当两个矩阵 A 和 B 匹配时，才可以定义他们的积 AB。也就是说第一个矩阵 A 的列数要等于第二个矩阵 B 的列数。因此，如果 A 是 3×5 的，B 是 5×2 的，就可以定义 AB，但不能定义 BA。

A 和 B 乘积 $C=AB$ 中的每一项仅是 A 中某行与 B 中某列的点积。具体来说，积的第 ij 个元素 c_{ij} 是 A 的第 i 行和 B 的第 j 列的点积。因此一个 $n\times m$ 的矩阵和 $m\times r$ 的矩阵的乘积是一个 $n\times r$ 的矩阵。例如，我们可以得到

$$\begin{pmatrix} 2 & 0 & 6 & -3 \\ 8 & 1 & -4 & 0 \\ 0 & 5 & 7 & 1 \end{pmatrix} \begin{pmatrix} 6 & 2 \\ -1 & 1 \\ 3 & 1 \\ -5 & 8 \end{pmatrix} = \begin{pmatrix} 45 & -14 \\ 35 & 13 \\ 11 & 20 \end{pmatrix}$$

例如,由于$(2,0,6,-3)\cdot(2,1,1,8)=-14$,所以 $c_{12}=-14$。一个很相似的例子是行矩阵和列矩阵的乘积,如

$$(4,5)\begin{pmatrix} 2 \\ 3 \end{pmatrix} = 23$$

这个结果是因为$(4,5)$和$(2,3)^T$ 的点积是 23。一个 1×1 的矩阵和另一个 1×1 的矩阵的乘积自然等于简单的将两个数字相乘。

两个方阵乘积的例程已经在本书的网站上公开了。由此可以很容易地扩展到任意两个相互匹配的矩阵的乘积。

我们列举了矩阵乘法的一些有用的性质。假设矩阵 A、B 和 C 相互匹配。那么

$$(AB)C = A(BC)$$
$$A(B+C) = AB + AC$$
$$(A+B)C = AC + BC$$
$$(AB)^T = B^T A^T$$

且

$$A(sB) = sAB$$

其中 s 是一个数。当计算两个矩阵 A 和 B 的乘积的时候,矩阵顺序的不同会导致不同的结果。对表达式 AB 我们称之为"A 左乘 B"或者"A 被 B 右乘"。如果 A 和 B 是相同大小的方阵,在两种顺序中它们都匹配,因此 AB 和 BA 都可以定义,但是两种乘积可能包含了不同的元素。如果 $AB=BA$ 对于两个矩阵成立,就称他们**可交换**(两个对称矩阵总是可交换的吗?)。

向量和矩阵相乘

矩阵相乘的一个特例是其中的一个矩阵是行向量或列向量。在图形学中,经常见到一个列向量 w 被左乘以一个矩阵 M,形式为 Mw。例如,令

$$w = \begin{pmatrix} 2 \\ 5 \\ -3 \end{pmatrix} = (2,5,-3)^T$$

且

$$M = \begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix}$$

那么 $\mathbf{w}$ 和 M 相匹配,可以得到

$$M\mathbf{w} = \begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix} \begin{pmatrix} 2 \\ 5 \\ -3 \end{pmatrix} = \begin{pmatrix} -14 \\ 33 \\ 4 \end{pmatrix}$$

通过上述相同的方法，$M\mathbf{w}$ 的每个元素都是 M 的行和 $\mathbf{w}$ 的点积。同样可以对一个矩阵左乘以一个行向量 $\mathbf{v}$，如

$$\mathbf{v}M = (3,-1,7)\begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix} = (-2,34,71)$$

A2.1.3　矩阵分块

把矩阵分成元素块并给每块命名有时会很便利。例如，

$$M = \begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix} = \left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right)$$

其中各元素块被定义为

$$M_1 = \begin{pmatrix} 2 & 0 \\ 8 & 1 \end{pmatrix},\quad M_2 = \begin{pmatrix} 6 \\ -4 \end{pmatrix},\quad M_3 = (3,2)$$

和 M_4，由单个元素 7 组成。这称为把 M 分块为以上 4 块。需要注意的是，如果一个块在另一个的上方，他们必须具有相同的列数。与此相似，两个块并排时，它们必须具有相同的行数。两个按同样方式进行分块的矩阵（对应块具有相同的形状）可以通过分别相加对应块来计算。对分块矩阵进行转置，可以先对每块分别转置，然后在将块之间的排列进行转置。因此，

$$\left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right)^T = \left(\begin{array}{c|c} M_1^T & M_3^T \\ \hline M_2^T & M_4^T \end{array}\right)$$

同样，矩阵乘法也可以通过将它们的子矩阵按通常的方式相乘得到，只要子矩阵相互匹配：

$$\left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right)\left(\begin{array}{c|c} M_5 & M_6 \\ \hline M_7 & M_8 \end{array}\right) = \left(\begin{array}{c|c} M_1M_5 + M_2M_7 & M_1M_6 + M_2M_8 \\ \hline M_3M_5 + M_4M_7 & M_3M_6 + M_4M_8 \end{array}\right)$$

A2.1.4　矩阵的行列式

每个方阵 M 都有一个与之关联的数字，这个数字被称为**行列式**，记做 $|M|$。行列式描述了特定几何形状的体积，并提供了说明线性变换对物体面积和体积产生的影响的信息。对于 2×2 的矩阵，其行列式很简单，就是两个积的差：

$$|M| = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

如果 M 是一个 3×3 的矩阵，它的行列式具有如下形式

$$|M| = \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix} = m_{11}\begin{vmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{vmatrix} - m_{11}\begin{vmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{vmatrix} + m_{13}\begin{vmatrix} m_{21} & m_{21} \\ m_{31} & m_{32} \end{vmatrix}$$

因此，

$$\begin{vmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{vmatrix} = 294$$

注意到这里的$|M|$是三项之和：$m_{11}M_{11}-m_{12}M_{12}+m_{13}M_{13}$。所以它可以表示成点积的形式：$|M|=(m_{11},-m_{12},m_{13})\cdot(M_{11},M_{12},M_{13})$。$M_{ij}$项是什么？$M_{ij}$被称为矩阵$M$关于元素$m_{ij}$的余子式。由于在求解矩阵逆的过程中会得到余子式，所以可以很方便的定义它。

定义：方阵的每个元素m_{ij}都有一个与之对应的余子式M_{ij}。相应的余子式是把矩阵M中的第i行和第j列去掉所得的矩阵的行列式与$(-1)^{i+j}$的乘积。

注意，当在一行或一列上移动时，$(-1)^{i+j}$的值在1和-1之间交替变化。可以用一个棋盘来形象的表示1和-1在矩阵上的分布。计算任意一个$n\times n$的矩阵M的行列式$|M|$的一般规则如下：

取M中的任意一行，计算该行中每个元素对应的余子式，然后将该行与由余子式构成的n元组进行点积。同样，也可以选择M中的一列并执行相同的规则。(这个规则对于2×2的矩阵成立吗?)

下面是有关行列式的一些有用的性质：

- $|M|=|M^{T}|Z$。
- 如果M中的两行(或两列)相同，那么$|M|=0$。
- 如果M和B都是方阵，那么$|MB|=|M||B|$。
- 如果B是由M中的两行(或列)构成的，那么$|B|=-|M|$。
- 如果B是由M通过把M中的一行(或列)乘以一个常数k得到的，那么$|B|=k|M|$。
- 如果B是由M通过把M中的一行(或列)加上令一行(或列)构成的，那么$|B|=|M|$。

A2.1.5 矩阵的逆

如果$|M|$不等于0，就称$n\times n$的矩阵M是非奇异的。在这种情况下，M有逆矩阵，记为M^{-1}，具有如下性质

$$MM^{-1}=M^{-1}M=I$$

其中I是$n\times n$的单位矩阵。同样，方阵乘积的逆是

$$(AB)^{-1}=B^{-1}A^{-1}$$

用M的余子式来表示M^{-1}的元素是很简单的。记A为M的逆。那么A的第ij个元素

$$a_{ij}=\frac{M_{ji}}{|M|}$$

也就是说我们计算m_{ji}项的余子式，再除以整个矩阵的行列式。仔细的注意这里的下标：计算a_{ij}的时候用m_{ji}的余子式。计算M行列式的一个等价步骤如下：

(1) 建立一个由余子式$c_{ij}=M_{ij}$构成的中间矩阵。

(2) 用C的任何一行和M中对应行的点积求出$|M|$。

(3) 对 C 进行转置得到 C^T。

(4) 将 C^T 中的所有元素用 $|M|^{-1}$[①] 进行缩放得到 M^{-1}。

示例

求 $M=\begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix}$ 的逆矩阵

解：

建立以 M 的余子式为元素构成的矩阵 C：

$$\begin{pmatrix} 27 & -56 & 40 \\ 30 & 14 & -10 \\ -6 & 56 & 2 \end{pmatrix}$$

然后用 $(2,0,6)\cdot(27,-56,40)=294$ 算出 $|M|$。然后对 C 转置，并用 $1/|M|$ 缩放所有的元素得到

$$M^{-1}=\frac{1}{294}\begin{pmatrix} 27 & 30 & -6 \\ -56 & 14 & 56 \\ 40 & -10 & 2 \end{pmatrix}$$

计算乘积 $M^{-1}M$ 和 MM^{-1} 看是否等于单位矩阵来验证这个结果。

正交矩阵

对于有些变换，例如旋转（见第 5 章），相关的矩阵都有一个很容易求解的逆矩阵。如果简单的将一个矩阵转置就可以产生它的逆矩阵，也就是 $M^T=M^{-1}$，就称该矩阵是一个正交矩阵。因此，$MM^T=I$。

如果 M 是正交的，$MM^T=I$ 说明 M 的每一行都是具有单位长度的向量，而且各行相互正交。对 M 的各列上述结论依然成立。（为什么？）例如，如果 M 是 3×3 的，以如下方式对其分块：

$$M=\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \end{pmatrix}$$

那么，三元组 $\mathbf{a}$、$\mathbf{b}$ 和 $\mathbf{c}$ 都是单位长度并且 $\mathbf{a}\cdot\mathbf{b}=\mathbf{a}\cdot\mathbf{c}=\mathbf{b}\cdot\mathbf{c}=0$。

A2.2 向量及其运算的一些性质

A2.2.1 向量的正交、正交点积

正交向量和正交点积只适用于二维向量。正如第 4 章所看到的，向量的正交和正交点积提供了求解两个方程两个未知数的有用的工具。

---

① 原书中为 $|M|^T$，应该是一处小错误。

向量的正交

令向量 $\mathbf{a}=(ax,ay)$。那么,$\mathbf{a}$ 沿逆时针方向旋转得到垂线,或"正交向量",记为 $\mathbf{a}^{\perp}$。定义为 $\mathbf{a}^{\perp}=(-ay,ax)$,满足如下性质:

a. 向量 $\mathbf{a}$ 和 $\mathbf{a}^{\perp}$ 具有相同的长度:$|\mathbf{a}|=|\mathbf{a}^{\perp}|$。

b. 线性性质:$(\mathbf{a}+\mathbf{b})^{\perp}=|\mathbf{a}^{\perp}|+|\mathbf{b}^{\perp}|$ 且 $(A\mathbf{a})^{\perp}=A\mathbf{a}^{\perp}$ 对于任意标量 A 成立。

c. 向量正交两次就得到原向量的负向量:$\mathbf{a}^{\perp\perp}=(\mathbf{a}^{\perp})^{\perp}=-\mathbf{a}$。

正交点积 $\mathbf{a}^{\perp}\cdot\mathbf{b}$

a. 正交点积 $\mathbf{a}^{\perp}\cdot b=a_xb_y-a_yb_x$ 对于任意的二维向量 $\mathbf{b}=(\mathbf{b}_x,\mathbf{b}_y)$[①]成立。

b. $\mathbf{a}^{\perp}\cdot\mathbf{a}=0$。($\mathbf{a}^{\perp}$ 垂直于 $\mathbf{a}$)

c. $|\mathbf{a}^{\perp}|^2=|\mathbf{a}|^2$。($\mathbf{a}^{\perp}$ 和 $\mathbf{a}$ 具有相同的长度)

d. $\mathbf{a}^{\perp}\cdot\mathbf{b}=-\mathbf{b}^{\perp}\cdot\mathbf{a}$。(正交点积是反对称的)

e. $\mathbf{a}^{\perp}\cdot\mathbf{b}$ 可以被写成行列式形式

$$\mathbf{a}^{\perp}\cdot\mathbf{b}=\begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

f. 如果 $\mathbf{a}+\mathbf{b}+\mathbf{c}=\mathbf{0}$,那么 $\mathbf{a}^{\perp}\cdot\mathbf{b}=\mathbf{b}^{\perp}\cdot\mathbf{c}=\mathbf{c}^{\perp}\cdot\mathbf{a}$。

g. 当且仅当存在 $\mathbf{a}$ 到 $\mathbf{b}$ 的 CCW 旋转时 $\mathbf{a}^{\perp}\cdot\mathbf{b}>0$。

h. 如果 $\mathbf{b}$ 与 $\mathbf{a}$ 平行或者反平行,则 $\mathbf{a}^{\perp}\cdot\mathbf{b}=0$。

i. $|\mathbf{a}^{\perp}\cdot\mathbf{b}|$ 是向量 $\mathbf{a}$ 和 $\mathbf{b}$ 确定的平行四边形的面积。

A2.3 球坐标和方向余弦

本节回顾球坐标的概念并总结如何在球坐标和笛卡儿坐标之间进行相互转换。图 A2.1 图示了在球坐标中是如何定义点 U 的。R 是 U 到原点的径向距离,ϕ 是 U 到 xz 平面的夹角,称为点 U 的纬度。θ 是 U 的方位角,xy 平面与经过 U 和 y 轴的平面所成的角。ϕ 的范围是 $-\pi/2\leqslant\varphi<\pi/2$,$\theta$ 的范围是 $0\leqslant\theta<2\pi$。

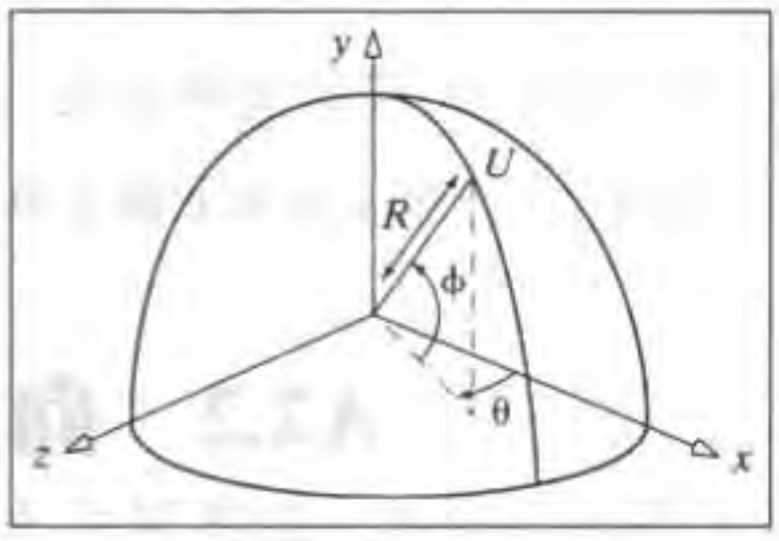


图 A2.1 球坐标的定义

使用简单的三角函数,可以直接算出这些量和 U 的笛卡儿坐标(u_x,u_y,u_z)之间的关系。方程为:

$$u_x=R\cos(\phi)\cos(\phi)$$
$$u_y=R\sin(\phi) \qquad (A2.3)$$

和

$$u_z=R\cos(\phi)\sin(\phi)$$

① 原书为 $\mathbf{b}=(\mathbf{b}_x,\mathbf{b}_y)=a_x\mathbf{b}_y-a_y\mathbf{b}_x$,应该是一处小错误。

也可以反过来用 u_x, u_y, u_z 来表示(R, ϕ, θ)：

$$R = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

$$\phi = \sin^{-1}\left(\frac{u_y}{R}\right)$$

$$\theta = \arctan(u_z, u_x) \tag{A2.4}$$

函数 arctan (,)是两个变量形式的反正切函数，定义如下

$$\arctan(y, x) = \begin{cases} \tan^{-1}(y/x) & \text{如果 } x > 0 \\ \pi + \tan^{-1}(y/x) & \text{如果 } x < 0 \\ \pi/2 & \text{如果 } x = 0 \text{ 且 } y > 0 \\ -\pi/2 & \text{如果 } x = 0 \text{ 且 } y < 0 \end{cases} \tag{A2.5}$$

这个函数可以区分 x、y 都是正数和 x、y 都是负数的情况，不像通常的形式$\tan^{-1}(x/y)$，后者只能得到$-\pi/2$ 到 $\pi/2$ 之间的角。注意 x 是零而 y 不是零的情况，这将导致可怕的除数为零的情况。这种情况被小心地避免了。

例 2.1

假设点 U 和原点的距离为 2，在 xz 平面上偏 60°，并且沿 x 轴反方向。因此 U 在 xy 平面内。所以 U 的球坐标表示为(2,60°,180°)。用等式(A2.3)可以计算 U 的笛卡儿坐标，可得 $U=(-1,1.732,0)$。

方向余弦

上面例子中，点 U 的方向用两个角的形式给出：方位角和纬度。方向通常可以用另一个有用的方法——方向余弦来表示。过原点的直线的方向余弦是其分别与 x、y 和 z 轴三个夹角的余弦。回顾一下，两个单位向量夹角的余弦是由它们的点积给出的。用给定的点 U，我们可以得到位置向量(u_x, u_y, u_z)。从前述的讨论中可以看到这个向量的长度是 R，所以它必须被标准化到具有单位长度的向量 $\mathbf{m}=(u_x/R, u_y/R, u_z/R)$。所以它和 x 轴夹角的余弦可以通过点积 $\mathbf{m} \cdot i = u_x/R$ 给出，后者仅是 $\mathbf{m}$ 的第一个分量。类似的，$\mathbf{m}$ 的第二和第三个分量分别是方向余弦的第二和第三项。记与 x、y 和 z 轴构成的角度分别为 α、β 和 γ，就可以得出从 $\mathbf{0}$ 到 U 这条线的三个方向余弦：

$$\cos(\alpha) = \frac{u_x}{R}$$

$$\cos(\beta) = \frac{u_y}{R}$$

且

$$\cos(\gamma) = \frac{u_z}{R}$$

注意，这 3 个方向余弦是相关的，它们的平方和总是 1。

计算向量 $\mathbf{n}$ 的方向余弦，其中

1. $\mathbf{n}=(1,1,1)$。
2. $\mathbf{n}=(2,3,4)$。

附录 3

有用的类、例程以及 SDL

本附录定义了一些数据类型和算法，可用于开发图形学应用程序。基本的类型是通过各种类描述的，其中有些有完整的定义，另外的一些则只声明了大量的方法而没有给出定义。这有待读者进一步充实。有些类只给出了框架形式，以便说明在实际应用程序中那些可以被进一步改进。大部分的类采用简单的封装方法：为了简明和避免定义大量的读取和访问函数，大部分的数据域被声明为公有而非私有。另外一些优秀的类和与应用程序资料可以在“Graphics Gems”系列中找到，其在线存储库是 http：//tog. acm. org/GraphicsGems/。

本附录所定义的类收集在各种头文件(.h)和源文件(.cpp)中，它们在本书的网站上也能找到。(见前言。)由于某些代码片段可能被改进或者随时间调整，请参考对应的网站。对应的网站上也有一些程序员可能会觉得有用的附加代码。

本书中所描述的类的集合

1. 2D 图形类。这些类提供了对绘制 2D 图的支持，包括 IntPoint、Point2、Polyline、IntRect、Vector2 和 Canvas。

2. RGB 像素图。该类提供了对创建和绘制像素图的支持，包括读入一个图像以及存储为 BMP 格式。

3. SDL。这些类提供了对处理和绘制 3D 场景的支持，包括 SDL 中描述的场景。支持的类包括 Point3、Vector3、Color3、Light、Affine4、AffineStack、Material、GeomObj、Boolean、UnionBool、IntersectionBool、DifferenceBool、Shape、Cube、Sphere、TaperedCylinder、Square、Plane、Face、Mesh、Torus、Teapot 和 Scene。

4. 噪声。这些类中包括对固体纹理创建 3D 噪音和扰动的 Noise 类。

5. 光线跟踪类。包括 PointCluster、SphereInfo、Cuboid、Ray、HitInfo 和 Intersection。

A3.1 2D 图形类

A3.1.1 基本 2D 图形类

```
//graphics2d.h                                  //支持 2D 图形的类的集合
#ifndef _GRAPHICS2D
#define _GRAPHICS2D
#include <string>
```

```
#include <iostream>
#include <fstream>
#include <strstream>
using namespace std;
#include <windows.h>                    //如果使用 xWindows 或其他平台则需要改变这里
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>

//@@@@@@@@@@@@IntPoint class @@@@@@@@@@@@@@@@@
class IntPoint{                         //整型坐标的 2D 点
public:
int x,y;
void set(int dx, int dy){x=dx; y=dy;}
void set(IntPoint& p){ x=p.x; y=p.y;}
IntPoint(int xx, int yy){x=xx; y=yy;}
IntPoint(){ x=y=0;}
};
//@@@@@@@@@@@@@@Point2 class @@@@@@@@@@@@@@@@@
class Point2{                           //实数坐标的 2D 点
public:
float x,y;
void set(float dx, float dy){x=dx; y=dy;}
void set(Point2& p){ x=p.x; y=p.y;}
Point2(float xx, float yy){x=xx; y=yy;}
Point2(){x=y=0;}
};
//<<<<<<<<<<<<<<<<<<<<<<<<PolyLine >>>>>>>>>>>>>>>>>>>>>>>>>>
class PolyLine{                         //一个多边形是一个数①加上点的数组
public:
int num;
Point2 pt[80];                          //在某些情况下可能需要更大的数组
PolyLine(){num=0;}
};
//@@@@@@@@@@@@@@@IntRect class @@@@@@@@@@@@@@@@
class IntRect{                          //边界值为整数的矩形
public:
  int left, top, right, bott;
  IntRect(){left=top=right=bott=0;}
  IntRect(int l, int t, int r, int b)
```

① 原书中是“num”，此处译为“数”，指的是类中的代表多边形中有效点个数的整数变量 num。

```
  {left=l; top=t; right=r; bott=b;}
  void set(int l, int t, int r, int b)
  {left=l; top=t; right=r; bott=b;}
  void set(IntRect& r)
  {left=r.left; top=r.top; right=r.right; bott=r.bott;}
};
//@@@@@@@@@@@@@@Vector2 class @@@@@@@@@@@@@@@@
class Vector2{
public:
float x,y;
void set(float dx, float dy){ x=dx; y=dy; }
void set(Vector2& v){ x=v.x; y=v.y;}
void setDiff(Point2& a, Point2& b)        //设为 a-b 的差
{x=a.x-b.x; y=a.y-b.y;}
void normalize()                          //调整该向量为单位长度
{double sizeSq=x * x+y * y;
if(sizeSq<0.0000001)
{
cerr <<"\nnormalize() sees vector (0,0)!";
return;                                   //对零向量不做任何操作
}
float scaleFactor=1.0/(float)sqrt(sizeSq);
x * =scaleFactor; y * =scaleFactor;
}
Vector2(float xx, float yy){x=xx; y=yy; }
Vector2(Vector2& v){x=v.x; y=v.y; }
Vector2(){x=y=0;}                         //默认构造函数
float dot(Vector2 b)                      //返回和 b 点积的结果
{return x * b.x+y * b.y;}
void perp()                               //perp this vector
{float tmp=x; x=-y; y=tmp;}
float perpDot(Vector2& v)                 //返回和 v 正交点积的结果
{return x * v.x-y * v.y;}
};
//<<<<<<<<<<<<<<<<<<<<<<<Canvas class >>>>>>>>>>>>
//全局 Canvas 对象(见第三章描述)知道如何在世界坐标系中画线①
class Canvas {
private:
Point2 CP;                                //在世界中的当前位置
float CD;                                 //以角度表示的当前方向
public:
float windowAspect;
```

① 原书中此类有错误："class Canvas {"这一块误写为"// Beginning of graphics2d. h"且原书中漏掉了"class Canvas"，根据代码这里不应该是 graphics2d. h 的起始。

```
Canvas(int width, int height, char* title);
void setWindow(float l, float r, float b, float t);
void setViewport(int l, int r, int b, int t);
float getWindowAspect(void);
void lineTo(float x, float y);
void moveTo(float x, float y){CP.x=x; CP.y=y;}
void forward(float dist, int vis);
void initCT()                          //初始化 CT(模型试图矩阵)
{
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}
void rotate2D(double angle)
{
glMatrixMode(GL_MODELVIEW); glRotated(angle, 0.0, 0.0, 1.0);
  }
void translate2D(double dx, double dy)
{
glMatrixMode(GL_MODELVIEW); glTranslated(dx, dy, 0.0);
}
void scale2D(double sx, double sy)
{
glMatrixMode(GL_MODELVIEW); glScaled(sx, sy, 1.0);
}
void pushCT(void)
{
glMatrixMode(GL_MODELVIEW); glPushMatrix();
}
void popCT(void)
{
glMatrixMode(GL_MODELVIEW); glPopMatrix();
}
void ngon(int n, float cx, float cy, float radius);
};①

//graphics2d.cpp----使用#include "graphics2d.h"
//graphics2d.h 中定义的方法
#include "graphics2d.h"
Canvas::Canvas(int width, int height, char* title)
{
char* list;                            //glutInit 的虚列表
int numArgs=1;                         //glutInit 的虚赋值
glutInit(&numArgs, &list);
```

① 原书此处有好几行的缺失，根据和第二版的对比，对此处做了较大的修正。由于修改点太多，所以不一一赘述，有兴趣的读者可以参考原版。

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(width, height);
glutInitWindowPosition(100, 100);
glutCreateWindow(title);
CP.x=CP.y=0.0;
windowAspect=1.0;
}
void Canvas::setWindow(float l, float r, float b, float t)
{
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D((GLdouble)l, (GLdouble)r, (GLdouble)b, (GLdouble)t);
if(t==b) return;
windowAspect=(r-l)/(t-b);
}
void Canvas::setViewport(int l, int r, int b, int t)
{
glViewport((GLint)l, (GLint)b, (GLint)(r-l), (GLint)(t-b));
}
void Canvas::lineTo(float x, float y)
{
glBegin(GL_LINES);
glVertex2f((GLfloat)CP.x, (GLfloat)CP.y);
CP.x=x; CP.y=y;
glVertex2f((GLfloat)CP.x, (GLfloat)CP.y);
glEnd(); glFlush();
}
void Canvas::forward(float dist, int vis)
{
#define RadPerDeg 0.017453393                    //每度的弧度数
float x=CP.x+dist*cos(RadPerDeg*CD);
float y=CP.y+dist*sin(RadPerDeg*CD);
if(vis) lineTo(x, y);
else moveTo(x, y);
CP.x=x; CP.y=y;
}
void Canvas::ngon(int n, float cx, float cy, float radius)
{
#define RadPerDeg 0.017453393                    //每度的弧度数
if (n<3) return;                                 //错误的边数
double angle=0, angleInc=2*3.14159265/n;         //角度增加
moveTo(cx+radius, cy);
for(int k=1; k<=n; k++)
{
Angle+=angleInc;
```

```
lineTo(radius * cos(angle)+cx, radius * sin(angle)+cy);
}
}
//graphics2d.cpp 结束
```

RGBPixmap 类

```
//RGBpixmap.h: 支持使用 RGB 像素映射的类
#ifndef _RGBPIXMAP
#define _RGBPIXMAP
#include <fstream>
//需要定义 IntPoint 和 IntRect 类
typedef unsigned char uchar;
class mRGB{                                    //RGB 这个名字已经被 Windows 使用了
public: uchar r,g,b;
mRGB(){r=g=b=0;}
mRGB(mRGB& p){r=p.r; g=p.g; b=p.b;}
mRGB(uchar rr, uchar gg, uchar bb){r=rr; g=gg; b=bb;}
void set(uchar rr, uchar gg, uchar bb){r=rr; g=gg; b=bb;}
};
//$$$$$$$$$$$$$$$$$$RGBPixmap 类 $$$$$$$$$$$$$$$$
class RGBpixmap{
private:
mRGB * pixel;                              //像素数组
public:
int nRows, nCols;                          //像素映射的维数
RGBpixmap() {nRows=nCols=0; pixel=0;}
RGBpixmap(int rows, int cols)              //构造函数
{
nRows=rows;
nCols=cols;
pixel=new mRGB[rows * cols];
}
int readBMPFile(string fname);             //将 BMP 文件读入该像素映射
void freeIt()                              //从该像素映射中恢复内存
{
delete []pixel; nRows=nCols=0;
}
//<<<<<<<<<<<<<<<<<<<<复制 >>>>>>>>>>>>>>>>>>>>
void copy(IntPoint from, IntPoint to, int x, int y, int width, int height)
{ //把显示器区域的一部分复制回显示器
if(nRows ==0 || nCols ==0) return;
glCopyPixels(x, y, width, height,GL_COLOR);
}
//<<<<<<<<<<<<<<<<<<<<绘制 >>>>>>>>>>>>>>>>>>>>
```

```
void draw()
{ //在当前光栅位置绘制像素映射
if(nRows == 0 || nCols == 0) return;
//通知 OpenGL: 不要在内存中对像素进行 4 字节边界对齐
glPixelStorei(GL_UNPACK_ALIGNMENT,1);
glDrawPixels(nCols, nRows,GL_RGB, GL_UNSIGNED_BYTE,pixel);
}
//<<<<<<<<<<<<<<<<<<读取 >>>>>>>>>>>>>>>>>>
int read(int x, int y, int wid, int ht)
{                                          //读取像素矩形到像素映射
nRows=ht;
nCols=wid;
pixel=new mRGB[nRows * nCols]; if(!pixel) return -1;
//通知 OpenGL: 不要在内存中对像素进行 4 字节边界对齐
glPixelStorei(GL_PACK_ALIGNMENT,1);
glReadPixels(x, y, nCols, nRows, GL_RGB,GL_UNSIGNED_BYTE,pixel);
return 0;
}
//<<<<<<<<<<<<<<<<<从 IntRect 中读取 >>>>>>>>>>>>>>>>>
int read(IntRect r)
{ //读入像素矩形到像素映射表中
nRows=r.top-r.bott;
nCols=r.right-r.left;
pixel=new mRGB[nRows * nCols]; if(!pixel) return -1;
//通知 OpenGL: 不要在内存中对像素进行 4 字节边界对齐
glPixelStorei(GL_PACK_ALIGNMENT,1);
glReadPixels(r.left,r.bott, nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
return 0;
}
//<<<<<<<<<<<<<<<setPixel >>>>>>>>>>>>>>
void setPixel(int x, int y, mRGB color)
{
if(x>=0 && x <nCols && y >=0 && y<nRows)
pixel[nCols * y+x]=color;
}
//<<<<<<<<<<<<<<<<<getPixel >>>>>>>>>>>
mRGB getPixel(int x, int y)
{
mRGB bad(255,255,255);
assert(x >=0 && x<nCols);
assert(y >=0 && y<nRows);
return pixel[nCols * y+x];
}
};                                         //RGBpixmap 类结束
#endif
```

```
//RGBpixmap.cpp-读取 BMP 文件的例程
#include "RGBpixmap.h"
typedef unsigned short ushort;
typedef unsigned long ulong;
fstream inf;                          //为了方便,在这个文件中是全局的
//<<<<<<<<<<<<<<<<<<<<<<getShort >>>>>>>>>>>>>>>>>>>>>
ushort getShort()                     //辅助函数
{ //使用低位在前存储方式整型的 BMP 格式
  //获取一个低位在前存贮方式的 2 字节整数
  char ic;
  ushort ip;
  inf.get(ic); ip=ic;                 //第一个字节是低位
  inf.get(ic);  ip |= ((ushort)ic <<8); //或在高字节①
  return ip;
  }
//<<<<<<<<<<<<<<<<<<<<<<getLong >>>>>>>>>>>>>>>>>>>>>
ulong getLong()                       //辅助函数
{                                     //使用低位在前存储方式整型的 BMP 格式
                                      //获取一个低位在前存贮方式的 4 字节整数
  ulong ip=0;
  char ic=0;
  unsigned char uc=ic;
  inf.get(ic); uc=ic; ip=uc;
  inf.get(ic); uc=ic; ip |= ((ulong)uc <<8);
  inf.get(ic); uc=ic; ip |= ((ulong)uc <<16);
  inf.get(ic); uc=ic; ip |= ((ulong)uc <<24);
  return ip;
  }
//<<<<<<<<<<<<<<<<<<<<RGBPixmap:: readBmpFile>>>>>>>>>>>>>>
int RGBpixmap:: readBMPFile(string fname)
{ //从未压缩的 BMP 文件中读入 mRGB 图像到内存中
//失败返回 0,成功返回 1
inf.open(fname.c_str(), ios::in|ios::binary);   //读二进制字符串
if(!inf){ cout <<" can't open file: " <<fname <<endl; return 0;}
int k, row, col, numPadBytes, nBytesInRow;
//读入文件头信息
char ch1, ch2;
inf.get(ch1); inf.get(ch2);           //类型: 总是'BM'
ulong fileSize=getLong();
ushort reserved1=getShort();          //总是 0
ushort reserved2=getShort();          //总是 0
ulong offBits =getLong();             //到图像数据的偏移量-不可靠
ulong headerSize=     getLong();      //总是 40
```

① 原文是"or in high order byte",此段代码是把高字节赋值给 ip 的高位。

```
ulong numCols =getLong();                    //图像中的列数
ulong numRows=getLong();                     //图像中的行数
ushort planes=     getShort();               //总是 1
ushort bitsPerPixel=   getShort();           //8 或 24; 这里允许是 24
ulong compression=   getLong();              //对未压缩的图片必须是 0
ulong imageSize=getLong();                   //图像的总字节数
ulong xPels=   getLong();                    //总是 0
ulong yPels=   getLong();                    //总是 0
ulong numLUTentries=getLong();               //对 8 位是 256,否则为 0
ulong impColors=getLong();                   //总是 0
if(bitsPerPixel !=24)
{ //错误-必须是 24 位的未压缩图像
cout <<"not a 24 bit/pixelimage, or is compressed! \n";
inf.close(); return 0;
}
//给每行加入若干字节以保证总字节数是 4 的倍数
//对 3* numCols 上舍入到 4 的倍数
nBytesInRow= ((3* numCols+3)/4) * 4;
numPadBytes=nBytesInRow-3* numCols;          //需要这么多
nRows=numRows;                               //设置类的数据成员
nCols=numCols;
pixel=new mRGB[nRows * nCols];               //为数组申请空间
if(!pixel) return 0;                         //内存溢出!
long count=0;
char dum;
for(row=0; row<nRows; row++)                 //读入像素值
{
for(col=0; col<nCols; col++)
{
char r,g,b;
inf.get(b); inf.get(g); inf.get(r);          //读入字节
pixel[count].r=r;                            //设置颜色
pixel[count].g=g;
pixel[count++].b=b;
}
   for(k=0; k<numPadBytes ; k++)             //跳过行尾的扩充字节
inf >>dum;
}
inf.close(); return 1;                       //成功
}
```

A3.1.2 场景和支持类

```
//SDL.h
```

```
//简单支持类的定义
#fndef _SDL
#define _SDL
#include <string>
#include <iostream>
#include <fstream>
#include <strstream>
using namespace std;
#include <windows.h>
#include <assert.h>
#include <math.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//如果要在场景中加入像素映射域,应该包括 RGBpixmap
#include "RGBpixmap.h"
//@@@@@@@@@@@@@@@@Point3类@@@@@@@@@@@@@@@@
class Point3{
public:
float x,y,z;
void set(float dx, float dy, float dz){x=dx; y=dy; z=dz;}
void set(Point3& p){x=p.x; y=p.y; z=p.z;}
Point3(float xx, float yy, float zz){x=xx; y=yy; z=zz;}
Point3(){x=y=z=0;}
void build4tuple(float v[])
{//用当前颜色建立 4元组:考虑齐次性 v[3]=1
v[0]=x; v[1]=y; v[2]=z; v[3]=1.0f;
}
};
//@@@@@@@@@@@@@@@@@@Vector3类 @@@@@@@@@@@@@@@@@
class Vector3{
public:
float x,y,z;
void set(float dx, float dy, float dz){ x=dx; y=dy; z=dz;}
void set(Vector3& v){ x=v.x; y=v.y; z=v.z;}
void flip(){x=-x; y=-y; z=-z;}          //翻转该向量
void setDiff(Point3& a, Point3& b)      //设为 a-b
{ x=a.x-b.x; y=a.y-b.y; z=a.z-b.z;}
void normalize();                       //调整该向量到单位长度
Vector3(float xx, float yy, float zz){x=xx; y=yy; z=zz;}
Vector3(Vector3& v){x=v.x; y=v.y; z=v.z;}
Vector3(){x=y=z=0;}                     //默认构造函数
Vector3 cross(Vector3 b);               //返回和 b 叉乘的结果
float dot(Vector3 b);                   //返回和 b 点积的结果
};
```

```
//@@@@@@@@@@@@@@@@@@Color3类 @@@@@@@@@@@@@@@@@
class Color3 {                                  //存储红、绿、蓝三元组
public:
float red, green, blue;
Color3(){red=green=blue=0;}
Color3(float r, float g, float b){red=r; green=g; blue=b;}
Color3(Color3& c){red=c.red; green=c.green; blue=c.blue;}
void set(float r, float g, float b){red=r; green=g; blue=b;}
void set(Color3& c)
      {red=c.red; green=c.green; blue=c.blue;}
void add(float r, float g, float b)
      {red+=r; green+=g; blue+=b;}
void add(Color3& src, Color3& refl);
void add(Color3& colr);
void build4tuple(float v[]);
};
//@@@@@@@@@@@@@@@@@@light类 @@@@@@@@@@@@@@@@@@
class Light{                                    //用于光源颜色位置的连接表
public:
Point3 pos;
Color3 color;
Light * next;
void setPosition(Point3 p){pos.set(p);}
void setColor(Color3 c){color.set(c);}
Light(){next=NULL;}
};
//@@@@@@@@@@@@@@@@@@Affine4类@@@@@@@@@@@@@@@@@
class Affine4 {                                 //管理齐次仿射变换
//包括逆变换
//和保存他们的栈
//被 Scence 类用来读取 SDL 文件
public:
float m[16];                                    //保存 4×4 的矩阵
Affine4();
void setIdentityMatrix();
void set(Affine4 a);
void preMult(Affine4 n);
void postMult(Affine4 n);
};                                              //end of Affine4 class
//@@@@@@@@@@AffineNode类 @@@@@@@@@@@
class AffineNode{
//被 Scence 类用来读取 SDL 文件
public:
Affine4 * affn;
Affine4 * invAffn;
```

```
AffineNode * next;
AffineNode()
{
next=NULL;
affn=new Affine4;                    //新申请一个 affine 并赋初值为单位矩阵
invAffn=new Affine4;                 //对逆变换也做相同的事情
}
~AffineNode()                        //析构函数
{
delete affn;
delete invAffn;
}
};
//@@@@@@@@@@@@@@@@AffineStack 类 @@@@@@@@@@@@
class AffineStack{
//被 Scence 类用来读取 SDL 文件
public:
AffineNode * tos;
AffineStack()                        //默认构造函数,把单位阵放在栈顶
{
tos=new AffineNode;                  //包含单位阵的节点
tos->next=NULL;
}
void dup();
void setIdentity();                  //设置栈顶元素为单位矩阵
void popAndDrop();
void releaseAffines();               //弹出并删除所有剩余元素
void rotate(float angle, Vector3 u);
void scale(float sx, float sy, float sz);
void translate(Vector3 d);
};                                   //AffineStack 类结束
//@@@@@@@@@@@@@@@@Material 类 @@@@@@@@@@@@@@
class Material{
public:
Color3 ambient, diffuse, specular, emissive;
int numParams;                       //用于纹理
float params[10];                    //用于纹理
int textureType;                     //0 代表无,负数代表固体,正数代表图片
float specularExponent, reflectivity, transparency, speedOfLight;
float specularFraction, surfaceRoughness;
void setDefault();
void set(Material& m);
};                                   //Material 类结束
//@@@@@@@@@@@@@@@@@GeomObj 类 @@@@@@@@@@@@@@@@@
class GeomObj{
```

```
public:
//IntRect scrnExtnt;
GeomObj * next;
GeomObj(): next(NULL){}
virtual void loadStuff(){}
virtual void drawOpenGL(){}
virtual void tellMaterialsGL(){}
};
//@@@@@@@@@@@@@@@@@@Boolean @@@@@@@@@@@@@@@@@@@
class Boolean: public GeomObj{
public:
GeomObj * left, * right;
Boolean():left(NULL),right(NULL){}
virtual void drawOpenGL()
{ //只绘制它的孩子
if(left)left->drawOpenGL();
if(right)right->drawOpenGL();
}
};
//@@@@@@@@@@@@@@@@@@UnionBool @@@@@@@@@@@@@@@@@@
class UnionBool: public Boolean{
public:
UnionBool(){Boolean();}                    //构造函数
};
//@@@@@@@@@@@@@@@@@@IntersectionBool @@@@@@@@@@@@@@@@@
class IntersectionBool: public Boolean{
public:
IntersectionBool(){Boolean();}
};
//@@@@@@@@@@@@@@@@@@DifferenceBool @@@@@@@@@@@@@@@@@@
class DifferenceBool: public Boolean{
public:
DifferenceBool(){Boolean();}
};
//@@@@@@@@@@@@@@@@@@Shape @@@@@@@@@@@@@@@@@@@
class Shape: public GeomObj{
public:
Material mtrl;
Affine4 transf,invTransf;
//virtual Color3 texture(HitInfo& h, int whichTexture);
Shape(){mtrl.textureType=0; mtrl.numParams=0;}
void setMaterial(Material& mt){mtrl.set(mt);}
void tellMaterialsGL();
};                                         //结束: Shape类
//@$@$@$@$@$@$@$@$@Cube类 $@$@$@$@$@$@$@$@$@$@$@
```

```
class Cube: public Shape{
public:
Cube(){}
void drawOpenGL()
{
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);                    //调用仿射
glEnable(GL_NORMALIZE);
if(doEdges) glutWireCube(2.0);
if(doFaces) glutSolidCube(2.0);             //一个以-1到1为顶点的立方体
glPopMatrix();
}
};
//@$@$@$@$@$@$@$@$Sphere类 @$@$@$@$@$@$@$@$@$@$@$@$@
class Sphere: public Shape{
public:
void drawOpenGL()
{
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
glutSolidSphere(1.0,20,20);
glPopMatrix();
}
Sphere() { }
};
//@$@$@$@$@$@$@$@$@$@TaperedCylinder类 @$@$@$@$@$@$@$@$@$@$
class TaperedCylinder: public Shape{
public:
float smallRadius;
TaperedCylinder(){}
void drawOpenGL();
};
//@$@$@$@$@$@$@$@$@$@Square类 @$@$@$@$@$@$@$@$@$@$
class Square: public Shape{
public:
Square(){}
void drawOpenGL() {                         //尚未实现};
};
//@$@$@$@$@$@$@$@$@$@Plane类 @$@$@$@$@$@$@$@$@$@$
class Plane: public Shape{
public:
Plane() {}
void drawOpenGL() {                         //尚未实现};
};
//##################class VertexID #################
```

```
//用于定义一个 Mesh
class VertexID{public: int vertIndex, normIndex;};
//##################class FACE ###############
//用于定义一个 Mesh
class Face{
public:
int nVerts;
VertexID * vert;                    //顶点数组和法线索引
Face(){ nVerts=0; vert=NULL;}
~Face(){delete[] vert; nVerts=0;}
};
//@$@$@$@$@$@$@$@$@Mesh 类 @$@$@$@$@$@$@$@$@$
class Mesh: public Shape{
private:
int numVerts, numNorms, numFaces;
Point3 * pt;                        //点数组
Vector3 * norm;                     //法线数组
Face * face;                        //面数组
int lastVertUsed;
int lastNormUsed;
int lastFaceUsed;
public:
void readMesh(string fname);
void writeMesh(char* fname);
void printMesh();
void drawMesh();
void drawEdges();
void freeMesh();
int isEmpty();
void makeEmpty();
Mesh();
virtual void drawOpenGL();
Mesh(string fname);
Vector3 newell4(int indx[]);
string meshFileName;                //保存这个 Mesh 的文件名
};                                  //Mesh 类结束
//@$@$@$@$@$@$@$@$@Torus class @$@$@$@$@$@$@$@$@$
class Torus: public Shape{
public:
void drawOpenGL(){
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
//if(doFaces) glutSolidTorus(0.2,1.0,10,12);
glPopMatrix();
}
```

```
};
//@$@$@$@$@$@$@$@$@$@Teapot class @$@$@$@$@$@$@$@$@$@$
class Teapot: public Shape{
public:
void drawOpenGL(){ tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
glutSolidTeapot(1.0); glPopMatrix();}
};
//@@@@@@@@@@@@@@DefUnit & DefUnitStack类 @@@@@@@@@@@@@@
//被 Scence 类用来读取 SDL 文件
class DefUnit {
//由 Steve Morin 开发
public:
string name, stuff;
DefUnit(string n, string s) {stuff=s;name=n;}
};
class DefUnitStack {
public:
DefUnitStack() { stack=NULL;}
void push(string n, string s);
void print();
int search(string s);
string contents(string s);
void release();
private:
struct D4S {
DefUnit * current;
struct D4S * next;
} d4s;
D4S * stack;
};                                    //DefUnitStack 类结束
//++++++++++++++TokenType ++++++++++++++
enum mTokenType {IDENT, LIGHT, ROTATE, TRANSLATE, SCALE, PUSH, POP, IDENTITYAFFINE,
GLOBALAMBIENT, BACKGROUND, MINREFLECTIVITY, MINTRANSPARENCY, MAXRECURSIONDEPTH, CUBE,
SPHERE, TORUS, PLANE, SQUARE, CYLINDER, CONE, TAPEREDCYLINDER, TETRAHEDRON, OCTAHEDRON,
DODECAHEDRON, ICOSAHEDRON, BUCKYBALL, TEAPOT, DIAMOND, UNION, INTERSECTION, DIFFERENCEa,
MAKEPIXMAP, MESH, DEFAULTMATERIALS, AMBIENT, DIFFUSE, SPECULAR, SPECULARFRACTION,
SURFACEROUGHNESS, EMISSIVE, SPECULAREXPONENT, SPEEDOFLIGHT, TRANSPARENCY, REFLECTIVITY,
PARAMETERS, TEXTURE, LFTCURLY, RGHTCURLY, DEF, USE, T_NULL, F_EOF, UNKNOWN };
//@@@@@@@@@@@@@@@@Scene 类 @@@@@@@@@@@@@@@@@@
class Scene{
public:
Light * light;                        //光源列表附在这里
GeomObj * obj;                        //物体列表附在这里
Color3 background, ambient;
```

```
int maxRecursionDepth;
//要有下面的纹理数据域就必须#include RGBpixmap
RGBpixmap pixmap[8];                    //附带像素映射的列表
float minReflectivity, minTransparency;
bool isInShadow(Ray& f);                //用于光线跟踪：留给读者实现
Scene():light(NULL),obj(NULL),tail(NULL)     //默认构造函数
{
currMtrl.setDefault();
background.set(0,0,0.6f);
ambient.set(0.1f,0.1f,0.1f);
minReflectivity=0.5;
minTransparency=0.5;
maxRecursionDepth=3;
}
Scene(string fname) {Scene(); read(fname); }
void freeScene();
void makeLightsOpenGL() {/* 尚未实现 */}
void drawSceneOpenGL();
bool read(string fname);
GeomObj * getObject();
private:
//用于读取场景的私有成员
int line;
int nextline;
ifstream  * file_in;
strstream * f_in;
strstream temp_fin;
DefUnitStack * def_stack;
GeomObj * tail;                         //物体列表尾指针
AffineStack affStk;                     //仿射栈
Material currMtrl;
string nexttoken(void);
float getFloat();
bool isidentifier(string keyword);
void cleanUp();
TokenType whichtoken(string keyword);
};                                      //Scene.h 结束
#endif
//SDL.cpp
//SDL.h 中类的支持代码
#include "SDL.h"
//Vector3 的方法
Vector3 Vector3:: cross(Vector3 b)      //返回和 b 叉乘的结果
{
   Vector3 c(y*b.z-z*b.y, z*b.x-x*b.z, x*b.y-y*b.x);
```

```
  return c;
}
float Vector3::dot(Vector3 b)                    //返回和 b 点乘的结果
{return x*b.x+y*b.y+z*b.z;}
void Vector3::normalize()                        //把该向量调整到单位长度
{
double sizeSq=x*x+y*y+z*z;
if(sizeSq<0.0000001)
{
cerr<<"\nnormalize() sees vector (0,0,0)!";
return;                                          //对零向量不做任何操作
}
float scaleFactor=1.0/(float)sqrt(sizeSq);
x*=scaleFactor; y*=scaleFactor; z*=scaleFactor;
}
//Color3 类的方法
void Color3::add(Color3& src, Color3& refl)
{ //加上原颜色和反射系数的乘积
red+=src.red*refl.red;
green+=src.green*refl.green;
blue+=src.blue*refl.blue;
}
void Color3::add(Color3& colr)
{ //把 colr 加到当前颜色
red+=colr.red ; green+=colr.green; blue+=colr.blue;
}
void Color3::build4tuple(float v[])
{//用当前颜色建立 4 元组：考虑齐次性 v[3]=1
v[0]=red; v[1]=green; v[2]=blue; v[3]=1.0f;
}
//Affine4 类的方法
Affine4::Affine4(){                              //建立单位变换
m[0]=m[5]=m[10]=m[15]=1.0;
m[1]=m[2]=m[3]=m[4]=0.0;
m[6]=m[7]=m[8]=m[9]=0.0;
m[11]=m[12]=m[13]=m[14]=0.0;
}
void Affine4::setIdentityMatrix(){               //建立单位变换
m[0]=m[5]=m[10]=m[15]=1.0;
m[1]=m[2]=m[3]=m[4]=0.0;
m[6]=m[7]=m[8]=m[9]=0.0;
m[11]=m[12]=m[13]=m[14]=0.0;
}
void Affine4::set(Affine4 a)                     //把当前矩阵设置为 a
{
```

```
for(int i=0; i<16; i++)
m[i]=a.m[i];
}
//<<<<<<<<<<<<preMult >>>>>>>>>>>>
void preMult(Affine4 n){                    //当前矩阵左乘以 n①
float sum;
Affine4 tmp;
tmp.set(* this);                            //tmp 副本
for(int c=0; c<4; c++)                      //form this=tmp * n
for(int r=0; r <4 ; r++)
{
sum=0;
for(int k=0; k<4; k++)
sum +=tmp.m[4 * k+r] *  n.m[4 * c+k];
m[4 * c+r]=sum;
}//end of for loops
}
//<<<<<<<<<<<<postMult >>>>>>>>>>>>
void postMult(Affine4 n){                   //当前矩阵右乘以 n
float sum;
Affine4 tmp;
tmp.set(* this);                            //tmp 副本
for(int c=0; c<4; c++)                      //form this=tmp * n
for(int r=0; r <4 ; r++)
{
sum=0;
for(int k=0; k<4; k++)
sum +=tmp.m[4 * k+r] *  n.m[4 * c+k];
m[4 * c+r]=sum;
}                                           //for 循环结束
}
//AffineStack 类的方法
void AffineStack::dup()
{
AffineNode *  tmp=new AffineNode;
tmp->affn=new Affine4(* (tos->affn));
tmp->invAffn=new Affine4(* (tos->invAffn));
tmp->next=tos;
tos=tmp;
}
void AffineStack::setIdentity()             //设置栈顶元素为单位阵
{
assert(tos !=NULL);
```

① 原书中此处为"postmultiplies this with n",根据后面代码可以看出此处应该是"premultiplies this with n"

```
tos->affn->setIdentityMatrix();
tos->invAffn->setIdentityMatrix();
}
void AffineStack::popAndDrop()
{
if(tos==NULL) return;                    //什么也不做
AffineNode * tmp=tos;
tos=tos->next;
delete tmp;                              //会调用析构函数来删除矩阵
}
void AffineStack::releaseAffines()
{                                        //弹出并释放所有剩余元素
while(tos) popAndDrop();
}
void AffineStack::rotate(float angle, Vector3 u)
{
Affine4 rm;                              //建立单位矩阵
Affine4 invRm;
u.normalize();                           //把旋转轴变为单位长度
float ang=angle*3.14159265/180;          //弧度变角度
float c=cos(ang), s=sin(ang);
float mc=1.0-c;
                                         //填充 3×3 的左上三角阵
rm.m[0]=c+mc*u.x*u.x;
rm.m[1]=mc*u.x*u.y+s*u.z;
rm.m[2]=mc*u.x*u.z-s*u.y;
rm.m[4]=mc*u.y*u.x-s*u.z;
rm.m[5]=c+mc*u.y*u.y;
rm.m[6]=mc*u.y*u.z+s*u.x;
rm.m[8]=mc*u.z*u.x+s*u.y;
rm.m[9]=mc*u.z*u.y-s*u.x;
rm.m[10]=c+mc*u.z*u.z;
//对于逆矩阵也一样,只是 s 的符号改变了
invRm.m[0]=c+mc*u.x*u.x;
invRm.m[1]=mc*u.x*u.y-s*u.z;
invRm.m[2]=mc*u.x*u.z+s*u.y;
invRm.m[4]=mc*u.y*u.x+s*u.z;
invRm.m[5]=c+mc*u.y*u.y;
invRm.m[6]=mc*u.y*u.z-s*u.x;
invRm.m[8]=mc*u.z*u.x-s*u.y;
invRm.m[9]=mc*u.z*u.y+s*u.x;
invRm.m[10]=c+mc*u.z*u.z;
tos->affn->postMult(rm);
tos->invAffn->preMult(invRm);
}
```

```
void AffineStack::scale(float sx, float sy, float sz)
{                                          //用比例因子右乘栈顶元素
#define sEps 0.00001
Affine4 scl;                               //建立一个单位阵
Affine4 invScl;
scl.m[0] =sx;
scl.m[5] =sy;
scl.m[10]=sz;                              //调整为一个缩放矩阵
if(fabs(sx)<sEps || fabs(sy)<sEps || fabs(sz)<
{
cerr <<"degenerate scaling transformation!\n"; );
}
invScl.m[0] =1/sx; invScl.m[5] =1/sy; m[10]=1/sz;
tos->affn->postMult(scl);                  //
tos->invAffn->preMult(invScl);
}
void AffineStack::translate(Vector3 d)
{
Affine4 tr;                                //建立一个单位阵
Affine4 invTr;
tr.m[12]=d.x; tr.m[13]=d.y;tr.m[14]=d.z;
invTr.m[12]=-d.x;invTr.m[13]=-d.y; [14]=-d.z;
tos->affn->postMult(tr);
tos->invAffn->preMult(invTr);
}
//Material 类的方法
void Material::setDefault(){
textureType=0;                             //无用
numParams=0;
reflectivity=transparency=0.0;
speedOfLight=specularExponent=1.0;
specularFraction=0.0;
surfaceRoughness=1.0;
ambient.set(0.1f,0.1f,0.1f);
diffuse.set(0.8f,0.8f,0.8f);
specular.set(0,0,0);
emissive.set(0,0,0);
}
void Material::set(Material& m)
{
textureType=m.textureType;
numParams=m.numParams;
for(int i=0; i<numParams; i++) params[i]=m.params[i];
transparency=m.transparency;
speedOfLight=m.speedOfLight;
```

```
reflectivity=m.reflectivity;
specularExponent=m.specularExponent;
specularFraction=m.specularFraction;
surfaceRoughness=m.surfaceRoughness;
ambient.set(m.ambient);
diffuse.set(m.diffuse);
specular.set(m.specular);
emissive.set(m.emissive);
}
//Shape 类的方法
void Shape::tellMaterialsGL()
{
float amb[4],diff[4],spec[4], emiss[4];
float zero[]={0,0,0,1};
mtrl.ambient.build4tuple(amb);          //填充数组
mtrl.diffuse.build4tuple(diff);
mtrl.specular.build4tuple(spec);
mtrl.emissive.build4tuple(emiss);
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_AMBIENT,amb);
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_DIFFUSE,diff);
if(specularOn)
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_SPECULAR,spec);
else
glMaterialfv(GL_FRONT,GL_SPECULAR,zero);
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_EMISSION,emiss);
glMaterialf (GL_FRONT/* _AND_BACK* /,GL_SHININESS,mtrl.specularExponent);
}
//Mesh 类的方法
Mesh ::Mesh(){
numVerts=numFaces=numNorms=0;
pt=NULL; norm=NULL; face=NULL;
lastVertUsed=lastNormUsed=lastFaceUsed=-1;
}
void Mesh::freeMesh()
{                                       //释放被该 mesh 占用的内存
delete [] pt;                           //释放整个顶点列表
delete [] norm;
for(int f=0; f<numFaces; f++)
delete[] face[f].vert;                  //删除面的 vert[]数组
delete [] face;
}
int Mesh::isEmpty()
{
return (numVerts==0) || (numFaces==0) || (numNorms==0);
}
```

```
void Mesh::makeEmpty()
{
numVerts=numFaces=numNorms=0;
}
void Mesh::drawOpenGL()
{
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
if(doFaces) drawMesh();
if(doEdges) drawEdges();
glPopMatrix();
}
Mesh::Mesh(string fname){                    //读入该文件,建立 Mesh
numVerts=numFaces=numNorms=0;
pt=NULL; norm =NULL; face=NULL;
lastVertUsed=lastNormUsed=lastFaceUsed=-1;
readMesh(fname);
}
Vector3 Mesh::newell4(int indx[])
{ /* 返回规范化的法线,使其和顶点 pt[indx[0]],…,pt[indx[3]]相对。例如: indx[]中包含四个
指向顶点列表的索引,这些顶点用于计算 Newell。 */
Vector3 m;
for(int i=0; i<4 ; i++)
{
int next=(i==3) ? 0: i+1;                    //哪一个是下一个索引?
int f=indx[i], n=indx[next];                 //表示这一对的名字
m.x += (pt[f].y-pt[n].y) * (pt[f].z+pt[n].z);
m.y += (pt[f].z-pt[n].z) * (pt[f].x+pt[n].x);
m.z += (pt[f].x-pt[n].x) * (pt[f].y+pt[n].y);
}
m.normalize();
return m;
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<readMesh >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void Mesh:: readMesh(string fname)
{
  fstream inStream;
inStream.open(fname.c_str(), ios::in);  //打开操作需要一个类 C 的字符串
if(inStream.fail() || inStream.eof())
  {
cout <<"can't open file or eof: " <<fname <<endl;
makeEmpty();return;
  }
inStream >>numVerts >>numNorms >>numFaces;
//建立顶点、法线和面的数组
```

```
pt=new Point3[numVerts]; assert(pt !=NULL);
norm=new Vector3[numNorms]; assert(norm !=NULL);
face=new Face[numFaces]; assert(face !=NULL);
for(int i=0; i<numVerts; i++)                    //读入顶点
inStream >>pt[i].x >>pt[i].y >>pt[i].z;
for(int ii=0; ii<numNorms; ii++)                 //读入法线
inStream >>norm[ii].x >>norm[ii].y >>norm[ii].z;
for(int f=0; f<numFaces; f++)                    //读入面数据
{
inStream >>face[f].nVerts;
int n=face[f].nVerts;
face[f].vert=new VertexID[n]; assert(face[f].vert !=NULL);
for(int k=0; k<n; k++)                           //读入表示该面的顶点索引
inStream >>face[f].vert[k].vertIndex;
for(int kk=0; kk<n; kk++)                        //读入表示该面的法线索引
inStream >>face[f].vert[kk].normIndex;
}
inStream.close();
}                                                //readMesh 结束
//<<<<<<<<<<<<<<<<<<<<<<<<drawMesh >>>>>>>>>>>>>>>>>>>>>>
void Mesh :: drawMesh()
{                                                //用 OpenGL 绘制该网格中的每个面：绘制每个多边形
if(isEmpty()) return;                            //网格为空
for(int f=0; f<numFaces; f++)
{
int n=face[f].nVerts;
glBegin(GL_POLYGON);
for(int v=0; v<n; v++)
{
int in=face[f].vert[v].normIndex;
assert(in >=0 && in<numNorms);
glNormal3f(norm[in].x, norm[in].y, norm[in].z);
int iv=face[f].vert[v].vertIndex; assert(iv >=0 && iv<numVerts);
glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
}
glEnd();
}
glFlush();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<write mesh>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void Mesh:: writeMesh(char * fname)
{                                                //把网格对象写入一个新的具有第 6 章格式的文件当中
if(numVerts ==0 || numNorms ==0 || numFaces ==0) return;     //空
fstream outStream(fname, ios :: out);    //打开输出流
if(outStream.fail()) {cout <<"can't make new file: " <<fname <<endl; return;}
```

```
outStream <<numVerts <<" " <<numNorms <<" " <<numFaces <<"\n";
//写顶点和顶点法线列表
for(int i=0; i<numVerts; i++)
outStream <<pt[i].x <<" " <<pt[i].y <<" " <<pt[i].z <<"\n";
for(int ii=0; ii<numNorms; ii++)
outStream<<norm[ii].x <<" " <<norm[ii].y <<" " <<norm[ii].z <<"\n";
//写面数据
for(int f=0; f<numFaces; f++)
{
int n=face[f].nVerts;
outStream <<n <<"\n";
for(int v=0; v<n; v++)                    //写该面上的顶点索引
outStream <<face[f].vert[v].vertIndex <<" ";outStream <<"\n";
for(int k=0; k<n; k++)                    //写该面的法线索引
outStream <<face[f].vert[k].normIndex <<" "; outStream <<"\n";
}
outStream.close();
}
//Scene 类的方法
//<<<<<<<<<methods >>>>>>>>>>>
string Scene::nexttoken(void)             //##########nexttoken()
{
char c;
string token;
int lastchar=1;
if (!f_in) {return(token); }
if (f_in->eof()) {return(token);}
while (f_in->get(c))
{
if (f_in->eof()) {
return(token);
}
switch (c) {
case '\n': nextline +=1;
case ' ' :
case '\t':
case '\a':
case '\b':
case '\v':
case '\f':
case '\r': {
if ( lastchar ==0 ) {return(token);}break; }
case '{': {
token=c; return(token); break;}
case '}': {
```

```
token=c;
return(token);
break; }
case '!': {
while ( c != '\n' && f_in->get(c)) {
}
nextline++; break;}
default: {
token=token+c;
lastchar=0;

if ((f_in->peek() == '{') ||
(f_in->peek() == '}') ) {
if ( lastchar == 0 ) {
return(token);
} else {
f_in->get(c);
token=c;
return(token);
}
}
line=nextline;
}
}
}
return(" ");
}
//<<<<<<<<getFloat >>>>>>>>>>>>
float Scene::getFloat()                    //############getFloat()
{
strstream tmp;
float number;
string str=nexttoken();
tmp <<str;
if(!(tmp >>number))
{
cerr <<"Line " <<line <<": error getting float" <<endl;
exit(-1);
}
else
{
char t;
if ( (tmp >>t ) )
{
cerr <<"Line " <<line <<": bum chars in number" <<endl;
```

```
exit(-1);
}
}
return number;
}
//<<<<<<<<<isidentifier >>>>>>>>>>>
bool Scene::isidentifier(string keyword) {//########isidentifier
string temp=keyword;
if (!isalpha(temp[0])) return(false);
for (int count=1; count<temp.length(); count++) {
if ((!isalnum(temp[count]))&& (temp[count]!='.')) return(false);
}
return(true);
}
//<<<<<<<<<cleanUp >>>>>>>>>>>
void Scene::cleanUp()                    //#########cleanUp
{                                        //分析文件后释放材料
affStk.releaseAffines();                 //删除仿射栈
def_stack->release();
delete def_stack;                        //释放 DefUnitStack 内存
}
//<<<<<<<<<freeScene >>>>>>>>>>>
void Scene::freeScene()
{                                        //释放物体和光列表
GeomObj * p=obj;
while(p)
{
GeomObj * q=p;
p=p->next;
delete q;
}
Light * q=light;
while(q)
{
Light * r=q;
q=q->next;
delete r;
}
}
//<<<<<<<<<<<<<<whichtoken >>>>>>>>>>>>>>
TokenType Scene::whichtoken(string keyword)
{
string temp=keyword;
if ( temp =="light" )            return (LIGHT);
if ( temp =="rotate" )           return (ROTATE);
```

```
if ( temp =="translate" )          return (TRANSLATE);
if ( temp =="scale")               return (SCALE);
if ( temp =="push")            return (PUSH);
if ( temp =="pop"  return (POP);
if ( temp =="identityAffine")     return (IDENTITYAFFINE);
if ( temp =="cube")               return (CUBE);
if ( temp =="sphere")             return (SPHERE);
if ( temp =="torus")              return (TORUS);
if ( temp =="plane")              return (PLANE);
if ( temp =="square")             return (SQUARE);
if ( temp =="cylinder")           return (CYLINDER);
if ( temp =="taperedCylinder")    return (TAPEREDCYLINDER);
if ( temp =="cone")               return (CONE);
if ( temp =="tetrahedron")        return (TETRAHEDRON);
if ( temp =="octahedron")         return (OCTAHEDRON);
if ( temp =="dodecahedron")       return (DODECAHEDRON);
if ( temp =="icosahedron")        return (ICOSAHEDRON);
if ( temp =="buckyball")          return (BUCKYBALL);
if ( temp =="diamond")            return (DIAMOND);
if ( temp =="teapot")   return (TEAPOT);
if ( temp =="union")              return (UNION);
if ( temp =="intersection")       return (INTERSECTION);
if ( temp =="difference")         return (DIFFERENCEa);
if ( temp =="mesh")               return (MESH);
if ( temp =="makePixmap")         return (MAKEPIXMAP);
if ( temp =="defaultMaterials")   return (DEFAULTMATERIALS);
if ( temp =="ambient")            return (AMBIENT);
if ( temp =="diffuse")            return (DIFFUSE);
if ( temp =="specular")           return (SPECULAR);
if ( temp =="specularFraction")   return (SPECULARFRACTION);
if ( temp =="surfaceRoughness")   return (SURFACEROUGHNESS);
if ( temp =="emissive")           return (EMISSIVE);
if ( temp =="specularExponent")   return (SPECULAREXPONENT);
if ( temp =="speedOfLight")       return (SPEEDOFLIGHT);
if ( temp =="transparency")       return (TRANSPARENCY);
if ( temp =="reflectivity")       return (REFLECTIVITY);
if ( temp =="parameters")         return (PARAMETERS);
if ( temp =="texture")  return (TEXTURE);
if ( temp =="globalAmbient")  return (GLOBALAMBIENT);
if ( temp =="minReflectivity")  return (MINREFLECTIVITY);
if ( temp =="minTransparency")  return (MINTRANSPARENCY);
if ( temp =="maxRecursionDepth") return (MAXRECURSIONDEPTH);
if ( temp =="background")         return (BACKGROUND);
if ( temp =="{")                  return (LFTCURLY);
if ( temp =="}")                  return (RGHTCURLY);
```

```
if ( temp =="def")                    return (DEF);
if ( temp =="use")                    return (USE);
if ( temp =="  " )                    return (T_NULL);
if ( isidentifier(temp) )             return (IDENT);
cout <<temp <<":" <<temp.length() <<endl;
return(UNKNOWN);
}                                          //whichtoken 结束
//<<<<<<<<<<<<<<drawSceneOpenGL>>>>>>>>>>>>>>>>>>>>
void Scene::drawSceneOpenGL()
{//绘制物体列表中的每一个物体
for(GeomObj * p=obj; p; p=p->next)
p->drawOpenGL();                           //绘制它
}
//<<<<<<<<<<<<<<<<<Scene :: read>>>>>>>>>>>>>>>>>>
bool Scene:: read(string fname)            //如果成功返回 true,否则返回 false
{
file_in=new ifstream(fname.c_str());
if(!(* file_in))
{
cout <<"I can't find or open file: " <<fname <<endl;
return false;
}
f_in=new strstream();
line=nextline=1;
def_stack=new DefUnitStack();
char ch;
freeScene();                               //删除所有以前的场景

//初始化所有用于读的数据
obj=tail=NULL;
light=NULL;
affStk.tos=new AffineNode;
affStk.tos->next=NULL;

while (file_in->get(ch)) {* f_in <<ch;}  //读整个文件
while(1)                                   //读文件,收集物体直到文件结束或者发生错误
{
GeomObj * shp=getObject();                 //获取下一个形状
if(!shp) break;                            //没有物体:错误或者文件结束
shp->next=NULL;                            //为了安全
if(obj ==NULL){ obj=tail=shp;}             //目前是空列表
else{tail->next=shp; tail=shp;}            //给队列中加入新物体
}
file_in->close();
cleanUp();                                 //删除临时列表等
```

```
return true;
}                                          //read()结束
//<<<<<<<<<<<<<<<Scene :: getObject >>>>>>>>>>>>>>>>
GeomObj * Scene :: getObject()
{                                          //从 f_in 流中读入标记,(Scene 类的数据成员),
//建立光源,获取材质,进行变换
//直到找到新的物体
//如果发生任何错误或文件技术则返回 NULL
string s;
GeomObj * newShape;
TokenType typ;
while ((typ=(whichtoken( s=nexttoken() ))) !=T_NULL)
{
if(typ ==UNION || typ ==INTERSECTION || typ ==DIFFERENCEa)
{
switch(typ)
{
case UNION:newShape=new UnionBool();break;
case INTERSECTION:newShape=new IntersectionBool();break;
case DIFFERENCEa:newShape=new DifferenceBool();break;
}                                          //小 switch 结束
GeomObj * p=newShape;
p=getObject();                             //获取左孩子
if(!p) return NULL;                        //错误! 应该总是得到一个物体
((Boolean * )newShape)->left =p;           //把它链接起来
p=getObject();//获取右孩子
if(!p) return NULL;
((Boolean * )newShape)->right=p;           //把它链接起来
return newShape;
}//if(typ ==UNION etc….结束
switch(typ)
{
case LIGHT: {
Point3 p;
Color3 c;
p.x=getFloat(); p.y=getFloat();p.z=getFloat();
c.red=getFloat(); c.green=getFloat();c.blue=getFloat();
Light * l=new Light;
l->setPosition(p);
l->setColor(c);
l->next=light;                             //把它放入列表
light=l; break;}
case ROTATE: {
float angle;
Vector3 u;
```

```
angle=getFloat(); u.x=getFloat();
u.y=getFloat(); u.z=getFloat();
affStk.rotate(angle,u);break;}
case TRANSLATE: {
Vector3 d;
d.x=getFloat(); d.y=getFloat(); d.z=getFloat();
affStk.translate(d);break;}
case SCALE: {
float sx, sy, sz;
sx=getFloat(); sy=getFloat(); sz=getFloat();
affStk.scale(sx, sy, sz);break;}
case PUSH: affStk.dup(); break;
case POP:  affStk.popAndDrop(); break;
case IDENTITYAFFINE: affStk.setIdentity();break;
case AMBIENT: {
float dr, dg, db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.ambient.set(dr,dg,db); break;}
case DIFFUSE: {
float dr,dg,db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.diffuse.set(dr,dg,db); break;}
case SPECULAR:{
float dr,dg,db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.specular.set(dr,dg,db); break;}
case EMISSIVE: {
float dr,dg,db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.emissive.set(dr,dg,db); break;}
case PARAMETERS: {                          //获取 numParams 参数列表
currMtrl.numParams=(int)getFloat();
for(int i=0; i<currMtrl.numParams; i++)
currMtrl.params[i]=getFloat();
break;}
case SPECULARFRACTION: currMtrl.specularFraction=getFloat(); break;
case SURFACEROUGHNESS: currMtrl.surfaceRoughness=getFloat(); break;
case TEXTURE: {                             //获取类型,0 表示无
currMtrl.textureType=getFloat();}
break;
case DEFAULTMATERIALS: currMtrl.setDefault();break;
case SPEEDOFLIGHT: currMtrl.speedOfLight=getFloat(); break;
case SPECULAREXPONENT: currMtrl.specularExponent=getFloat(); break;
case TRANSPARENCY:currMtrl.transparency=getFloat(); break;
case REFLECTIVITY: currMtrl.reflectivity=getFloat(); break;
```

```
case GLOBALAMBIENT:
ambient.red=getFloat(); ambient.green=getFloat();
ambient.blue=getFloat(); break;
case BACKGROUND:
background.red=getFloat();
background.green=getFloat();
background.blue=getFloat();break;
case MINREFLECTIVITY: minReflectivity=getFloat(); break;
case MINTRANSPARENCY:minTransparency=getFloat(); break;
case MAXRECURSIONDEPTH: maxRecursionDepth=getFloat(); break;
case MAKEPIXMAP: {                          //为像素映射获取 BMP 文件名
  int which=getFloat();                     //像素映射数组中该像素映射的索引
  if(which<0 || which >7){cout <<"\nbad index of RGBpixmap! \n";}
  string fname=nexttoken();                 //获取网格文件名
  cout <<"I got fname=" <<fname <<endl;
  if(!pixmap[which].readBMPFile(fname))
  {//读入 BMP 文件到该像素映射
cout <<" \ncan't read that RGBpixmap file! \n";
return NULL;
  }
  break;}//case: MAKEPIXMAP 结束
case T_NULL: break;                         //空标记代表文件结束
case DEF: {
string name, temp, lb, rb;
int l=line;
string inp;
name=nexttoken();
if ( whichtoken(name) !=IDENT ) {
cout <<"Error:  Identifier expected." <<endl;
return NULL;
}
if ( def_stack->search(name) ) {
cout <<line <<": " <<name;
cout <<": attempt to redefine. " <<endl;
return NULL;
}
lb=nexttoken();
if ( whichtoken(lb) !=LFTCURLY ) {
cout <<"Error: { expected." <<endl;
return NULL;
}
while ( whichtoken( temp=nexttoken()) !=RGHTCURLY ) {
cout <<temp <<endl;
inp=inp+temp+" ";
if (!f_in) {
```

```
cout <<"Error: end of file detected." <<endl;
return NULL;
}
}
//字符串内容压栈
def_stack->push(name, inp);
break;}                                    //end of case: DEF
case USE: {
string name;
name=nexttoken();
if ( whichtoken(name) !=IDENT ) {
cout <<line <<": " <<name;
cout <<": identifier expected.";
return NULL;
}
if (!def_stack->search(name) ) {
cout <<line <<": " <<name;
cout <<": not defined.";
return NULL;
}
cout <<def_stack->contents(name) <<endl;
strstream * temp_fin=new strstream;
* temp_fin <<def_stack->contents(name) <<" ";
* temp_fin <<f_in->rdbuf();
delete (f_in);
f_in=temp_fin;
break; }                                   //case: USE 结束
default:  {                                //Shapes 的内 switch
switch(typ)
{
case CUBE:newShape=new Cube;break;
case SPHERE:newShape=new Sphere;break;
case TETRAHEDRON:newShape=new Mesh("tetra.3vn");break;
case TORUS:newShape=new Torus;break;
case PLANE:newShape=new Plane;break;
case SQUARE:newShape=new Square;break;
case TAPEREDCYLINDER:newShape=new TaperedCylinder;
((TaperedCylinder * )newShape)->smallRadius=getFloat(); break;
case CONE:newShape=new TaperedCylinder;
((TaperedCylinder * )newShape)->smallRadius=0; break;
case CYLINDER:newShape=new TaperedCylinder;
((TaperedCylinder * )newShape)->smallRadius=1; break;
case OCTAHEDRON:newShape=new Mesh("octa.3vn");break;
case DODECAHEDRON:newShape=new Mesh("dodeca.3vn"); break;
case ICOSAHEDRON:newShape=new Mesh("icosa.3vn"); break;
```

```
case BUCKYBALL:newShape=new Mesh("bucky.3vn");break;
case DIAMOND:newShape=new Mesh("diamond.3vn"); break;
case TEAPOT:newShape=new Teapot; break;
case MESH: {                                    //获取该网格的文件名(带扩展名)
string fname=nexttoken();                       //获取网格文件名
newShape=new Mesh(fname); break;
}//case: MESH结束
   default: {
cerr <<"Line " <<nextline <<": unknown keyword " <<s <<endl;
return NULL;
}
}                                               //内 switch结束
//对所有形状的常规操作
((Shape * )newShape)->mtrl.set(currMtrl);
//加载变换及其逆变换
((Shape * )newShape)->transf.set(* (affStk.tos->affn));
((Shape * )newShape)->invTransf.set(* (affStk.tos->invAffn));
return newShape;
}//default: block结束
    }                                           //外 switch结束
  }                                             //while结束
  return NULL;
}                                               //getObject结束
//<<<<<<<<<<<<<<<<<<<<<<<<<DefUnitStack方法>>>>>>>>>>>>>>>>>>>>>>>>>>
void DefUnitStack::push(string n, string s) {
D4S * temp_d4s=new D4S;
temp_d4s->current=new DefUnit(n, s);
temp_d4s->next=stack;
stack=temp_d4s;
}
void DefUnitStack::print() {
D4S * temp=stack;
string t;
while (temp) {
cout <<temp->current->name <<":" ;
cout <<temp->current->stuff <<endl;
temp=temp->next;
}
}
int DefUnitStack::search(string s) {
D4S * temp=stack;
while (temp) {
if ( temp->current->name ==s ) {
return(1);
}
```

```
temp=temp->next;
}
return(0);
}
string DefUnitStack::contents(string s) {
D4S * temp=stack;
while (temp) {
if (temp->current->name ==s ) {
return(temp->current->stuff);
}
temp=temp->next;
}
return(NULL);
}
void DefUnitStack::release()
{
while(stack)
{
D4S *  tmp=stack;                          //获取它
//cerr <<"releasing def_stack item: "<<tmp->current->name<<endl;
stack=stack->next;                         //增加指针
delete tmp->current;                       //释放两个字符串
delete tmp;                                //释放节点
}
stack=NULL;
}
//SDL.cpp 结束
```

噪音类

```
//Noise.h
//用来产生基于按 Peachey/Perlin 方式噪声框架的伪随机噪声场的 Noise 类
#include <assert.h>

class Noise{
public:
Noise()                                    //构造噪声对象
{
int i;
index=new unsigned char[256]; assert(index);
for(i=0; i<256; i++) index[i]=i;           //用索引填充数组
for(i=0; i<256; i++)                       //shuffle it①
{
```

① 指像洗牌一样随机的产生噪声。

```
int which=rand() %256;                    //在数组中选择随机的位置
unsigned char tmp=index[which];           //交换它们
index[which]=index[i];
index[i]=tmp;
}
noiseTable=new float[256]; assert(noiseTable);
for(i=0; i<256; i++) noiseTable[i]=rand()/32767.99;
}                                         //构造函数结束
float noise(float scale, Point3& p)
{                                         //线性插值的栅格噪声
#define lerp(f, A, B) A+f* (B-A)
float d[2][2][2];
Point3 pp;
pp.x=p.x* scale+10000;                    //偏移量避免负值
pp.y=p.y* scale+10000;
pp.z=p.z* scale+10000;
long ix=(long)pp.x; long iy=(long)pp.y; long iz=(long)pp.z;
float tx,ty,tz, x0,x1,x2,x3, y0,y1;
tx=pp.x-ix; ty=pp.y-iy; tz=pp.z-iz;  //小数部分
float mtx=1.0-tx, mty=1.0-ty, mtz=1.0-tz;

for(int k=0; k<=1; k++)                   //获取 8 个栅格点的噪声
for(int j=0; j<=1; j++)
for(int i=0; i<=1; i++)
d[k][j][i]=latticeNoise(ix+i, iy+j,iz+k);

x0=lerp(tx, d[0][0][0],d[0][0][1]);
x1=lerp(tx, d[0][1][0],d[0][1][1]);
x2=lerp(tx, d[1][0][0],d[1][0][1]);
x3=lerp(tx, d[1][1][0],d[1][1][1]);
y0=lerp(ty, x0, x1);
y1=lerp(ty, x2, x3);
return lerp(tz, y0, y1);
}
float turbulence(float s, Point3& p)
{
float val=noise(s, p)/2 +
noise(s* 2, p)/4 +
noise(s* 4, p)/8 +
noise(s* 8, p)/16;
return val;
}
float marble(float strength,Point3& p)
{
float turbul=turbulence(10, p);
```

```
float val=sin(6*p.z+strength*turbul);
return mySpline(val);
}
float
private:
float* noiseTable;                    //噪声值数组
unsigned char* index;                 //伪随机索引
float mySpline(float x)               //用于大理石
{
if(x<-0.4) return 0.15+2.857*SQR(x+0.75);
else if(x<0.4) return 0.95-2.8125*SQR(x);
else return 0.26+2.666*SQR(x-0.7);
}
float latticeNoise(int i, int j, int k)
{                                     //返回整数格点的 PR 噪声值
#define PERM(x) index[(x) & 255]
#define INDEX(ix, iy, iz) PERM( (ix)+PERM((iy)+PERM(iz)) )
return noiseTable[INDEX(i,j,k)];
}
};                                    //噪声类结束
```

A3.1.3 一些用于光线跟踪的类

```
//@@@@@@@@@@@@@@@@PointCluster 类 @@@@@@@@@@@@@@@@
class PointCluster{
public:                               //保存一个形状轮廓点的数组
int num;
Point3* pt;
PointCluster() {num=0; pt=NULL;}
PointCluster(int n)                   //建立 n 个点的串
{
pt=new Point3[n]; assert(pt);
num=n;
  }
};
//@@@@@@@@@@@@@@@@SphereInfo @@@@@@@@@@@@@@@@
class SphereInfo{                     //保存球的中心和半径
public:
Point3 center;
float radSq;
void set(float x, float y, float z, float rsq)
{
center.set(x,y,z);
radSq=rsq;
```

```
}
};
//@@@@@@@@@@@@@@@@@@@@Cuboid @@@@@@@@@@@@@@@@@@@
class Cuboid{                                   //保存立方体的 6 个边界值
public:
float left, top, right, bott, front, back;
void set(float l, float t, float r, float b, float f, float bk)
{
left=l; top=t; right=r; bott=b; front=f; back=bk;
}
void set(Cuboid& c)
{
left=c.left;top=c.top; right=c.right; bott=c.bott;
front=c.front; back=c.back;
}
};

//@@@@@@@@@@@@@@@@@@@Ray @@@@@@@@@@@@@@@@@@@
class Ray{
public:
Point3 start;
Vector3 dir;
int recurseLevel;
int row, col;                                   //用于表示屏幕范围
int numInside;                                  //列表中物体的数目
GeomObj * inside[10];                           //物体指针数组
Ray(){start.set(0,0,0); dir.set(0,0,0); numInside=0;}
Ray(Point3 origin);                             //构造函数：设置光线的起点
Ray(Point3& origin, Vector3& direction)
{ start.set(origin); dir.set(direction); numInside=0;}
void setStart(Point3& p){start.set(p);}
void setDir(float x, float y, float z)
{dir.x=x; dir.y=y; dir.z=z;}
void setRayDirection(Light * L);                //用于阴影触角
void setRayDirection(Vector3& dir);             //用于产生的光线
int isInShadow();
void makeGenericRay(GeomObj * p, Ray& gr);
};
//@@@@@@@@@@@@@@@@@@HitInfo @@@@@@@@@@@@@@@@@@
class HitInfo {                                 //一个表面的每次命中的数据
  public:
double hitTime;                                 //命中时间
GeomObj * hitObject;                            //命中的物体
int surface;                                    //那个面被命中
int isEntering;                                 //光线进入物体了吗？
```

```
Point3 hitPoint;                        //命中点
Vector3 hitNormal;                      //命中点处的法线
HitInfo()
{
hitObject=NULL; hitTime=-1000; surface=0; isEntering=0;
}
void set(HitInfo& h)
{
hitTime=h.hitTime; hitObject=h.hitObject;
surface=h.surface; isEntering=h.isEntering;
hitPoint.set(h.hitPoint); hitNormal.set(h.hitNormal);
}
};

//@@@@@@@@@@@@@@@@Intersection @@@@@@@@@@@@@@@@
class Intersection{                     //保存命中列表
public:
#define maxNumHits 8
int numHits;                            //命中次数
HitInfo hit[maxNumHits];                //命中列表;
Intersection(){numHits=0;}              //默认构造函数
void set(Intersection& intr)
{                                       //拷贝相交信息
numHits=intr.numHits;
for(int i=0; i<maxNumHits; i++)
hit[i].set(intr.hit[i]);
}
};
```

A3.1.4 SDL：场景描述语言

场景描述语言是一种用清晰通俗的方式描述几何物体和形状的非常简单的语言。Scene 类中的 read()方法可以处理这种语言。我们首先简单介绍一下 Scene 类的基础，然后描述 SDL 并讨论其用法。

要读取一个用 SDL 写的场景文件(例如 mySence. dat)，我们首先得定义一个全局的场景对象(例如 scn)，然后对它调用 read()方法。将某个待处理的 SDL 文件名给该方法：

```
Scene scn;                      // 建立一个场景对象
scn.read("myScene.dat");        //读 SDL 文件,建立场景
```

读入并解释 myScene. dat 文件，并建立物体列表。同时创建光源列表。这些列表可以分别通过 scn. obj 和 scn. light 域得到。第 5 章的 drawOpenGL()方法在用 OpenGL 程序渲染这些场景的时候用到这些列表。第 12 章介绍的 shade()也可以用这些列表通过光线跟踪渲染场景。

A3.1.5 Scene 类

Scene 类中的一个对象有多个域来描述场景的特性。主要的元素是一个在场景中的几何形状的列表。obj 域是一个指向列表中第一个元素的指针。为了绘制列表中的所有物体，我们只需简单遍历整个列表，并告知每个物体绘制自己。

```
for(GeomObj * p=scn.obj; p!=NULL; p=p->next)
p->drawOpenGL();
```

列表中的物体属于 Shape 中的一种或者另一种类型(球，立方体，十二面体，等等)，而且每一种 Shape 类型都知道如何绘制它自己。这里用到了多态：所有的 Shape 类型都继承自基本类型 GeomObj(几何物体"geometric object"的缩写)，所以任何 Shape 类型都能放在 GeomObj 指针的列表中。

```
Light * light;                  //光源列表
GeomObj * obj;                  //物体列表
Color3 background;              //背景颜色
Color3 ambient;                 //全局环境色
```

另外还有 3 个域用于光线跟踪(见第 12 章)：maxRecursionDepth、minShinyness 和 minTransparency。

A3.2 SDL 语法

SDL 区分大小写，但形式灵活：多个泛空格符(空格、Tab、换行、换页等)等于一个空格。注释以"!"开头持续到行尾。SDL 中的关键字用来指定不同的仿射变换、几何物体、光源和场景属性(如背景颜色)。

创建几何对象

简单的说明物体的类型就可以创建物体并将其放入物体列表中。例如，cube 添加一个立方体对象到物体列表中，而 sphere 添加一个球体对象。

其他几何对象包括(参考 SDL.h 获得完全列表) torus、plane、square、cylinder、cone、tetrahedron、dodecahedron、icosahedron、buckyball、diamond 和 teapot。当这些对象类型中的任何一个在文件中被指定的时候，相应的物体就被添加到物体列表(尾部)了。

有些几何物体需要一个或多个参数。参数是浮点数或者文件名。参数被直接放在物体名后面，例如：

```
taperedCylinder .312 !生成小端半径为 0.312 的圆台
mesh pawn.3vn !生成网格
```

第一个例子创建一个用参数定义其确切形状的圆台对象；第二个例子创建了一个顶点和面列表在文件 pawn.3vn 中描述的网格对象(本书的网站上有许多的 3vn 格式样例文件，

包括十二面体、菱形和 buckyball 等样例)。

管理仿射变换

每个物体被创建的时候都会存储一个仿射变换。(该变换的逆也同时被存储;逆变换在光线跟踪时有用。)和物体一起创建的那个变换是当前变换(CT),既当时生效的变换。SDL 文件中的各种关键字都修改 CT。例如:

```
identityAffine
```

将恒等变换(由一个 4×4 的单位针给出)放在 CT 中。这是当 read()开始解析 SDL 文件时的初始变换。

SDL 使用词语 scale、rotate 和 translate 以及每个词后面跟的一个合适的参数来改变 CT。这同 OpenGL 使用 glScalef()、glRatef()和 glTranslatef()改变模型视图或投影矩阵是相似的。

特别地,每个词用相应的变换右乘以 CT 并把乘积放回 CT,如:

```
CT=CT * Trans
```

这里的 Trans 是一个表示新变换的 4×4 矩阵。

三个动词如下:

```
scale <sx><sy><sz>
rotate <ang><ux><uy><uz>
translate <dx><dy><dz>
```

scale 有 3 个分别表示 x、y 和 z 方向上缩放比例的浮点型参数。rotate 有 4 个参数:旋转角度(以度为单位)和旋转轴在 x、y 及 z 方向上的分量。(从 u 向原点望去,ang 的正值产生关于 u 轴的 CCW 旋转。)translate 有 3 个参数:变换所用向量的 x、y 和 z 分量。

例如,命令

```
scale 2 1.3 -5.33
translate 4 -5 6
rotate 45 0 1 0
```

分别产生如下矩阵 $Sc=\begin{bmatrix}2&0&0&0\\0&1.3&0&0\\0&0&-5.33&0\\0&0&0&1\end{bmatrix}$, $Tr\begin{bmatrix}1&0&0&4\\0&1&0&-5\\0&0&1&6\\0&0&0&1\end{bmatrix}$ 和 $Rot\begin{bmatrix}0.707&0&0.707&0\\0&1&0&0\\-0.707&0&0.707&0\\0&0&0&1\end{bmatrix}$,每个命令把其对应的矩阵右乘以 CT 并把结果放回 CT。下面有一个完整的例子。

仿射变换栈

CT 实际上是矩阵栈的栈顶矩阵。利用字 push 和 pop 来操作该栈:

- push 创建 CT 的一个副本并将其压入栈中(使栈顶的两个元素相同)。
- pop 从栈中弹出 CT 并丢弃它。刚好在栈顶元素下面的矩阵变成 CT。如果栈顶元素下面没有元素,pop 不做任何操作。

管理材料属性

在创建每个物体的同时创建的材料属性记录,称为当前材料(CM)。记录包含如下域:

- 四个颜色域,每个域都包含(r,g,b)三元组:ambient、diffuse、specular 和 emissive。
- 六个标量域(用于光线跟踪):specularExponent、specular Fraction、surfaceRoughness、speedOfLight、transparency 和 reflectivity。
- 描述物体纹理性质的三个域:textureType 指明哪种纹理被用于物体,numParams 为物体定义的参数数目,params[]是一个有 10 个变量值的数组。textureType 编码为 0 代表空,-1 代表棋盘,-2 代表木材,-3 代表大理石。代码 makePixmap 2 filename.bmp 读入一个 BMP 文件并把它赋给纹理 2。2 这个 textureType 后面跟着一个图像文件名,它建立了该图的一个像素映射。

对 CM 的改变可以通过下面的 SDL 字控制,每个后面跟了一个或多个浮点数:

```
ambient <r><g><b>
diffuse <r><g><b>
specular <r><g><b>
emissive <r><g><b>
specularExponent <value>
specularFraction <value>
surfaceRoughness <value>
speedOfLight <value>
transparency <value>
reflectivity <value>
textureType <value>
parameters <value><value><value>p
```

关键字 parameters 后面跟着一个指定参数个数的数字和参数值列表。例如,要把 4.5、6 和-12 放进随后定义的物体的 params[]数组中,可以用

```
parameters 3 5 4.5 6 -12
```

CM 中最初包含了如下默认值:

```
ambient=<0.1><0.1><0.1>
diffuse=<0.8><0.8><0.8>
specular=<0><0><0>
emissive=<0><0><0>
specularExponent 1
specularFraction 0
surfaceRoughness 1.0
speedOfLight=1
transparency=0
```

```
reflectivity=0
textureType 0
```

关键字 defaultMaterials 可以被用来恢复 CM 为这些默认值。以下是和光源、全局属性、布尔对象和像素映射有关的关键字：

光源

```
light <x><y><z><r><g><b>!在<x, y, z>处放置一个颜色为<r, g, b>的光源
```

指定全局场景属性

```
globalAmbient <r><g><b>!指定全局泛光颜色为<r, g, b>
minReflectivity <value>
minTransparency <value>
maxRecursionDepth <value>
background <r><g><b>
```

下面是一个简单的 SDL 文件：

```
!myScene1.dat -f.s.hill
!有几个简单的发光物体
globalAmbient .4 .2 .3
light 0 10 0 1 1 1 !在(0,10,0)处的白光
background 0 0 .5
ambient .2 .2 .2
diffuse .8 .7 .6
emissive .8 0 0 !物体放出红色
cube !在原点处放置一个普通的立方体
emissive 0 1 0
!在 (2,0,0)处放置一个发光的椭圆
push translate 2 0 0 rotate 45 0 0 1 scale .5 .5 2 sphere pop
push translate -2 0 0 cone pop !也 (放置)一个圆锥在 (-2,0,0)
```

布尔对象

SDL 也支持指定布尔对象，使用的关键字为：

```
union
intersection
difference
```

这些词中的每个都可以创建一个指定类型的布尔对象并将它放在对象列表中。在 SDL 文件中，这样的每个对象都必须跟有两个对象(其中的每个可能是几何或者布尔对象)，它们被放在最初创建的对象的左孩子和右孩子的位置。下面的 SDL 代码描述了一个包含两个布尔对象的场景。第一个对象是由一个四面体和巴克球进行并运算然后再和一个立方体交得到的。

```
light 1 2 3 0.2 0.3 0.4
intersection
cube !交叉点的左孩子
union !交叉点的右孩子
diffuse .2 .5 .7 tetrahedronn !并节点的左孩子
rotate 180 3 4 5 buckyball !并节点的右孩子
difference !另一个布尔对象
tetrahedron intersection !差节点的左孩子
plane union !交叉节点的右孩子
dodecahedron torus !并节点的左孩子
```

动词

```
makePixmap <value><fname>
```

定义了一个用于纹理绘制的像素映射。第一个参数是用来识别像素图的整型。第二个是包含图像纹理的 BMP 文件名。例如，如下的 SDL 代码：

```
makePixmap 5 clouds.bmp
makePixmap 5 stone.bmp
texture 5
parameters 4 0 1 1 0
push scale 3 4 2 cube pop
texture 2
parameters 3 .2 .8 2
sphere
```

创建了两个纹理。然后它把 5 号(跟着 4 个参数)纹理和一个缩放了的立方体关联，2 号(跟着 3 个参数)和一个球关联。

A3.3 SDL 中的宏

为了方便，关键字 def 允许你把许多 SDL 命令组合到一个宏中，并给其一个名字。组成宏的 SDL 命令被括号包起来。例如：

```
def red{ambient 1 0 0 diffuse 1 0 0}
```

把名为 red 的宏和上述语句联系在一起。如果在 SDL 文件后面的部分用到了命令

```
use red
```

那么语句 ambient 1 0 0 diffuse 1 0 0 就会被放在该处，就像在文件的这个地方键入一样。宏可以保存键入语句并允许定义被重用。例如，一个"L"形栈可以用如下代码定义：

```
def Lstack
{
push cube translate 2 0 0 cube
```

```
translate -2 2 0 cube
translate 0 2 0 cube pop
}
```

以后这些宏可以被用来为场景放入多个不同的 L 形,实现这个功能的代码如下:

```
use Lstack
push translate 3 2 4 scale .5 .5 .5 use Lstack pop
push translate -3 -4 -4 scale .5 .5 .5 use Lstack pop
```

A3.4 扩展 SDL

可以直接给 SDL 添加关键字。我们通过两个例子予以说明如何实现这个功能。

示例 A3.4.1 给场景添加一个属性

如果你想给 Scene 类添加新的域 fogThickness,来描述场景中雾的薄厚。为了控制放入该域中的值,需要在 SDL 语言中添加关键词,例如 fogginess。SDL 文件中的语句 fogginess 0.5 就可以把 fogThickness 的值变为 0.5。为了完成以上内容,需要进行如下变动。

sdl.h 中的改动

1. 给 Scene 类添加 float fogThickness 域。
2. 将 FOGTHICKNESS 添加到 TokenType 枚举表中的任何位置。

sdl.cpp 中的改动

1. 在函数 whichtoken 中加入一行: if (temp == "fogginess") return (FOGTHICKNESS)。

2. 在函数 getObject 中添加一行: case FOGTHICKNESS: fogThickness = getFloat (); break。

当然,如何用这个新的域的问题得由程序员自己去研究了。例如,在开发光线跟踪器时,程序员可能给 Scene: : Shade( )中添加一些代码,如: if(fogThickness > 0) do something。

示例 A3.4.2 给场景添加一个属性

如果你想在可能出现在场景的物体集合中加入扇形饼。这个扇形饼将是位于 xy 平面一个薄圆盘的一部分。扇形由 0 度角开始(沿着 x 轴),连续逆时针(从(0, 0, 1)向原点看)旋转到以角度来度量的角 sweep。因此,如果 sweep 是 180,扇形将是一个处于正 y 轴空间的半圆,而如果 sweep 是 360 度,该扇形将是一个完整的圆。扩展 SDL 语言,使得它能够识别带有一个和 sweep 角相关的参数的 pieSlice 关键字,如

```
pieSlice 90
```

文件 sdl.h 中的改动

1. 定义个含有保存 sweep 角域的类 PieSlice，通过下述代码实现：

```
class PieSlice : public Shape {
public:
float sweep;
etc.
}
```

2. 为该类编写适当的方法，如 drawOpenGL()。
3. 在 TokenType 枚举列表中的任意位置加入 PIESLICE 项。

文件 sdl.cpp 中的改动

1. 在函数 whichtoken()中添加一行

```
if (temp==="pieSlice") return (PIESLICE);
```

2. 在函数 getObject()中添加如下行

```
case PIESLICE;
newShape=new PieSlice;
((PieSlice*) newShape)->angle=getFloat(); break;
```

附录 4

分形和曼德布洛特集合

A4.1 引　　言

计算机特别擅长做重复性工作：它们可以毫无抱怨地不断重复做一件事情。另外，现代计算机在计算方面有很高的精确度，从而产生一些算法，能够有效地把客观物体放大到很大程度，可以更近距离的观察。

在这个附录中，我们从很多方面更进一步扩展，发掘计算机图形学的潜力，展现计算机图形学可能遇到的一些问题。针对无穷的范围，检测三种方法：无穷小，放大到更深层的细节或者可以给一个图像添加更细微的细节；无穷大，检测的图案可以通过某些方法重新产生出更大的图案；无穷多次，研究一个进程理论上无限重复时会发生什么。

通过前面的章节，我们可以看出计算机图形学如何产生自然界中不存在或者曾经存在的事物的图片。我们现在研究的这些内容确实是这样的，而计算机图形学提供了很强大的工具去研究这些内容。这里更进一步，我们突破任何计算机产生图片的固有限制：存在一个有限的解法，有限的大小，可以在有限的时间内完成。因此，我们所作的图片仅仅是被研究生物的近似：图片的观察者把它作为基本的真实物体的样子的一种暗示。

A4.2 分形和自相似

我们想找到一些方法，使得我们接近无穷——更精细的无穷小——在某种组织方式下。这些方法是某种特征的递归，现代计算机能够很有效地做好这一点。递归通常会使得很复杂的集合问题变得很简单。在其他事情中，它使得一个分解或者精细的轮廓更加细小，达到理论上的无限。递归算法构造的形状不仅美妙迷人，在科学和工程中也有很大的应用。

在这里我们描述的很多曲线和图片，都有一个尤其重要的属性：它们是自相似的。直观上理解，在任何尺度下，他们表现形式都是相同的：不管把曲线的图片放大多少倍，它仍然具有相同层次的细节。一些曲线是**严格自相似**的，由此可以得出，如果一个区域被放大很多倍后，放大后的图和原图仍然是很像的(除了可能的旋转或者移动)。剩下的是统计意义下**自相似**的，曲线中的摆动和无规则在平均意义上是一样的，不管图片被放大多少倍。

自然界提供了一些模仿统计意义下自相似的例子。很典型的例子就是海岸线。从卫星上看，由于海湾、水湾和半岛等，海岸线有某种意义上险峻的层次。当一个人飞近一点看的时候，很多细节就显现出来了。一个海湾呈现某种险峻性是以前所看不到的。进一步放大，海滩上单个的大石头和波动，以类似险峻性的形式呈现在眼前。再进一步放大，小石头和小

鹅卵石看起来产生了同样的险峻性。继续这个过程，直到通过显微镜看到沙子的单个的颗粒。其他自然现象，也同样呈现了自相似的特性，比如树的枝条，海绵的表面，人行道上的裂缝，动物的血管系统等。云也具有大概的自相似，并且提供了更有趣的例子。当乘飞机飞行的时候，很难判断一个云有多大：它是小而近呢，还是大而远呢？

在20世纪70年代，耶鲁大学的本瑟姆·曼德布洛特(Benoit Mandelbrot)（那个时候他在IBM研究中心）一起广泛地研究自相似的性质(比如文章【Mandelbrot83】)。他把自相似曲线的各种形式称为不规则的图形。线是一维的，平面是二维的，但是有些自然现象介于两者之间。比如，定义一条曲线在一个有限的矩形内无限长：他们的维数介于1和2之间。

曼德布洛特和其他人的工作，关于自然界中不规则事物，在数学上和计算机图形学上产生了大量的调查研究结果，在世界许多研究中心同样很令人兴奋。

A4.3 曼德布洛特集合

图形学提供了很强大的工具来研究一些很有趣的集合。一些人认为曼德布洛特集合是数学上最复杂的对象。基于简单的几个很令人惊讶的定义，它的结构居然这么丰富，借助计算机图形学来表示的时候能够得到漂亮到令人吃惊的图片。

曼德布洛特集合是从分析的一个分支**迭代理论**中发展起来的。迭代理论研究的是无止境的反复调用某个函数时会产生什么现象。相关集合Julia set的很多关键的结果，在本世纪早期被Gaston Julia(1893～1978)和Pierre Fatou(1978～1929)发现(当然，并没有借助计算机)。他们的想法潜伏了很多年，直到他们再次被想起，并且被本瑟姆·曼德布洛特在20世纪70年代的时候扩充。作为他研究的一部分，他用计算机图形学做非常重要的实验，依靠很多线条激发了一些猜想，使得能够进一步分析和发现。

在这些发现的背后，有很多很有意思的故事，(见【Mandelbrot83，Peitgen88】)，同时附带了一些漂亮的图片，产生的方法我们下面会讲到。

A4.3.1 曼德布洛特集合和迭代函数系统

图A4.1给出了曼德布洛特集合的一个直观的例子。彩色图片可以在Plate37得到。它的内部是黑色的，看起来是由心形线组成，旁边还有很多瘤状圆。事实上，它的边界十分复杂，在边界的某个地方特写，不断放大，就可以看出它的复杂性。

理论上，可以永远重复地放大——边界是无限复杂的，事实上它就是个不规则曲线！图像上的每一个点都是渐变的，或者根据在重复函数系统上的实验结果给它标记颜色。关心的重复函数系统如图A4.2。它使用的是一个相当简单的函数，如等式A4.1，其中c是常数。

$$f(z) = z^2 + c \tag{A4.1}$$

这个系统产生的输出是输入的平方与c的和。我们认为这个过程从初始值0开始，因此这个系统产生了一个值的序列或者轨迹(见第2章)，如等式。(仔细检查)这个轨迹的只决定于一个因素：c的值。

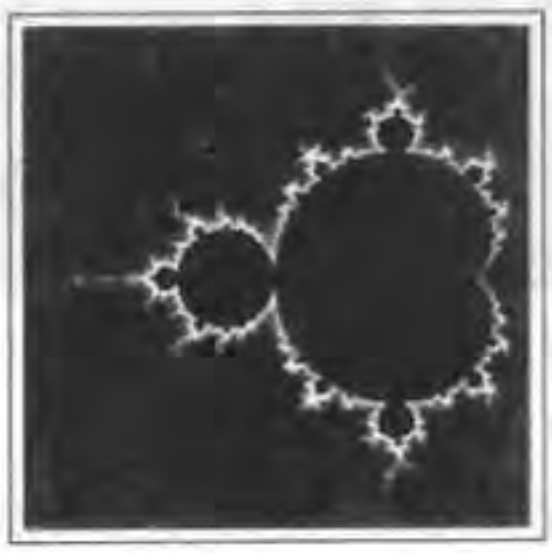


图 A4.1 曼德布洛特集合

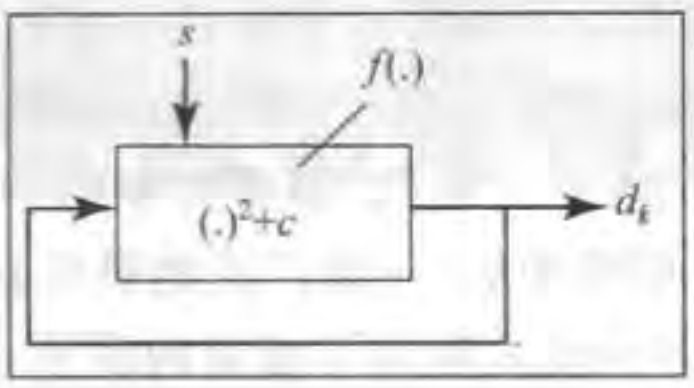


图 A4.2 曼德布洛特集合的迭代函数习题

$$
\begin{aligned}
d_1 &= c \\
d_2 &= (c)^2 + c \\
d_3 &= ((c)^2 + c)^2 + c \\
d_4 &= (((c)^2 + c)^2 + c
\end{aligned}
\tag{A4.2}
$$

注意到给定一个 c,经过一次迭代 0 的轨迹变成了 c 的轨迹。不管是从 0 开始,还是从 c 开始,我们得到的是同样的轨迹。c 的轨迹就是曼德布洛特集合主要研究的对象。最基本的问题是:给定一个 c 的值,当 k 不断变大的时候,轨迹上的点 d_k 如何表现?尤其是,当这个轨迹仍然是有限的时候(轨迹上所有的点和 0 的距离都是有限的)或者它会剧增(接近无限)吗?下面我们定义更准确一点,轨迹上仍然有限的点属于其相应的曼德布洛特集合,而剧增的点不属于这个集合。

例 A4.1 c 取不同值的时候,0 的轨迹

这里我们考虑 c 取不同值的时候的轨迹:

令 $c=-1$。这个轨迹是 0,0,0,0,0,0,0,……,无限重复。然而,这个轨迹是有限的,并且基于 $c=-1$ 的值属于曼德布洛特集合。

令 $c=1$。这个轨迹是 0,1,2,5,26,677,458330,……剧增,因此基于 $c=1$ 的值不属于曼德布洛特集合。

令 $c=-1.3$,这个轨迹是 0,-1.3,……,大约经过 50 步,可以得到一个循环的序列:-1.148665,0.019430,-1.299622,0.389018,不断重复。

我们尝试着把这些结果画出来,看看 c 取值在哪个范围的时候可以得到有限的轨迹。图 A4.3 给出了第一步尝试。它说明了选择其他 c 的值时候得到结果。比如,我们从 0 开始增加 c,存在有限轨迹的最大值是什么?

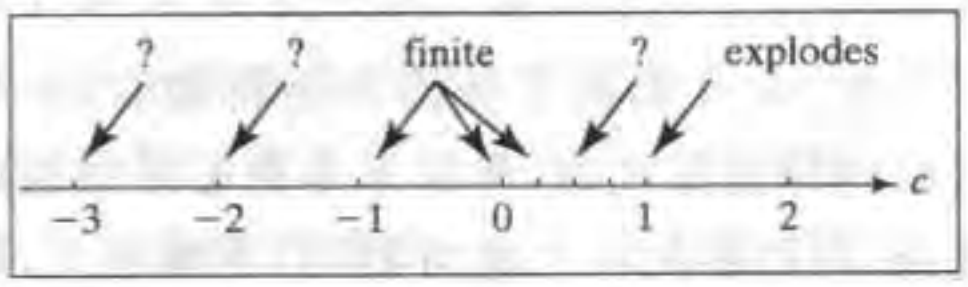


图 A4.3 对于 c 的每个值,0 的轨迹如何

这个系统已经展示了有趣的动态过程。当 c 取复数的时候,这个问题变得更加复杂。复数算术常使用在应用函数中。从算术到复数平面是曼德布洛特集合在理论上很重要的贡献。曼德布洛特集合在复数平面上存在。回想一下,图形学里我们用一个图表来表示复数:一个复数 $z=x+yi$,表示成在点(x,y)处。

图 A4.2 所示的系统能够很好的表现复数,就和实数一样。现在 c 的值是个复数,每次迭代,我们要把上次的结果平方,然后加上 c。回想平方复数 $z=x+yi$,可以得到一个新的复数,如等式(A4.3)。

$$(x+yi)^2=(x^2-y^2)+(2xy)i \tag{A4.3}$$

这个系统的固定点的注释

研究曼德布洛特集合时，检测系统 $f(.)=(.)^2+c$ 的不动点是很有用的。轨迹的行为很大程度上决定于这些不动点，这是因为，那些复数 z 影射到自己的时候 $z^2+c=z$。可以把这个式子转化成二次方程 $z^2-z+c=0$。系统的不动点就是该方程的两个根，表示为：

$$p_+,p_-=\frac{1}{2}\pm\sqrt{\frac{1}{4}-c} \tag{A4.4}$$

注意，我们处理的是复数情况，因此求 $1/4-c$ 的平方根没有问题，即使它是负数或者复数。附录 2 中给出了如何编程计算平方根。它们是系统的两个不动点。对于先前的例子，我们可以得到两个不动点 $p_+=1.2499323-0.33667i$，$p_-=-0.249227+0.33667i$。如果一个轨迹到达了某个不动点 p，它就会一直停留在那里。两个不动点位置上是对称的，它们距离 $1/2+0i$ 的距离是相同的。

可以更深层次地理解不动点，认为它们是引力中心(attracting)或者挡板(repelling)。严格地讲，如果一个轨迹离某个不动点 p 很近的话，轨迹上的下一个点必然满足下面的条件：

- 如果 p 是引力中心，则距离 p 更近。
- 如果 p 是挡板，则距离 p 更远。

如果一个轨迹距离某个引力中心不动点近的话，它会被"吸"进去；而对一个挡板不动点来说，会越来越远。很难表现出一个不动点是引力中心，只有它和源点的距离在 1/2 以内时，这个时候这些点在以源点为中心以 1/2 为半径的圆内。

图 A4.4 显示了不同 c 取值时的轨迹的例子，给曼德布洛特集合添加粗略的解释，使看起来更明白些。每个轨迹从各自的 c 开始，表示为一个点(我们可以从 0 开始画每条轨迹，但是这样的话，会导致整个图片被画满)。

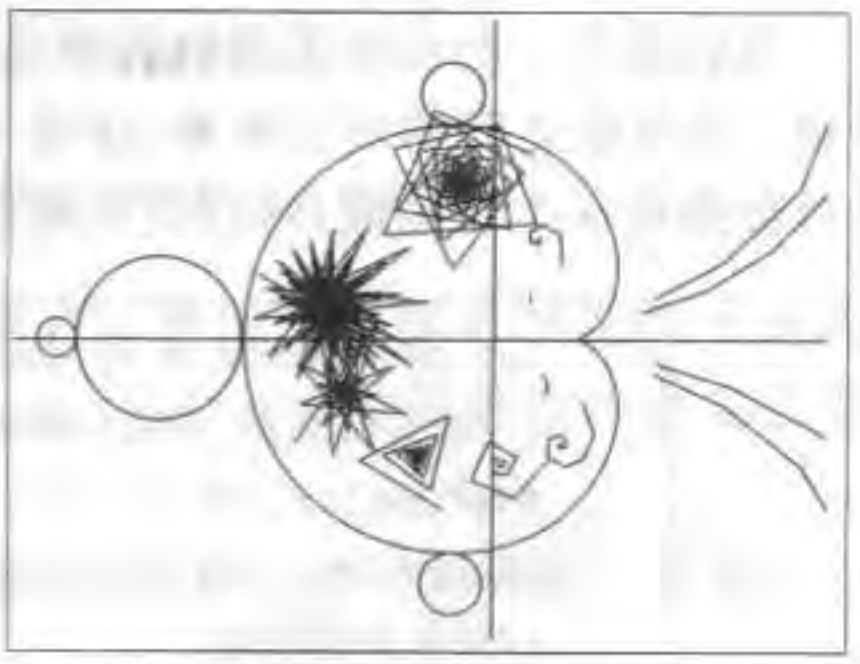


图 A4.4　不同 c 取值时的轨迹的例子

认真注意，每个点表示了是用在系统 $f(.)=(.)^2+c$ 中的相关 c 的值是重复的，和该轨迹起始点相同。

如果 c 选择的是不在曼德布洛特集合(称为集合 M)的元素轨迹的结果就会剧增。两个剧增的轨迹在途中表现出来了。如果选择的 c 恰好在 m 的边界上的话，该轨迹会变化很强烈，最后接近无穷。这些轨迹可能会使得图片混乱，因此没有画出来。

如果 c 的值在 M 内部，相应的曲线可以做很多事情。对于某些 c，它可能会很快到达一个不动点或者螺旋式的到达(当 c 比较接近 M 的边界时，这个速度非常慢)。另一方面，没有一个不动点是引力中心的话，这个轨迹一点都不会收缩到某个固定的值。取而代之的是，它会进入一个环形的序列(通常是周期性的)，在某个挡板不动点的周围。举个例子，c 取值的轨迹在 M 的顶部或者底部的小圆圈中的，周期是 3。

A4.3.2　如何判断点 c 是否属于曼德布洛特集合

现在转向应用，用于两个目的：

1. 更深入地研究曼德布洛特集合的特性。

2. 创造曼德布洛特集合炫目的全彩色图片。

给出 M 中的一个复数 c。路线是根据式检测轨迹上数字 d_k 的大小。当 k 增加的时候，$|d_k|$ 的值可能会剧增(c 不属于集合 M)或者不会(c 属于集合 M)。来自复数分析的一个理论说明，如果 $|d_k|$ 在某个时刻超过 2，那么轨迹在某个点的时候一定会剧增。使得 $|d_k|$ 大于 2 的重复的次数称为是该轨迹的 **dwell**，轨迹在某个区域停留的时间可能会上升。

但是如果 c 属于集合 M，轨迹必然存在一个有限的 **dwell**，如果没有无限重复我们就不知道这一点。我们最多可以设定重复最多次数的上限 Num。一个很典型的值就是 $Num=100$。如果重复了 Num 次后，$|d_k|$ 还没有超过 2，我们就认为它不可能超过了，认为 c 属于 M。可以得出刚好在 M 边界外的 c 的取值的那些轨迹存在一个很大的 $dwell$，如果这些 $dwell$ 超过了 Num，我们就会错误地把他们判定为在 M 内部。(属于 M 集合 c 的取值曾经被错误地解释过吗?)基于很小 Num 图画可以表现出曼德布洛特集合是相当巨大的。

图 A4.5 给出了如果压缩这些计算的程序 dwell()。对于给定的 $c=c_x+c_yi$，它的返回值是当 $|d_k|$ 大于 2 所需要的重复的次数，或者如果经过 Num 次重复没有超过 2，就返回 Num。为了简单，它为函数 dwell()定义了参数的数据类型。

每一次重复，和当前 d_k 相关的数对 (d_x,d_y)，先被平方然后加到数对 (c_x,c_y) 中得到一个 d 的值。$|d_k|^2$ 的值保存在 fsq 中，并且和 4 进行比较，这个和，用 $|d_k|$ 和 2 比较是等价的，而且还不用每次都求平方根。函数 dwell()在描绘曼德布洛特集合中起很重要的作用。

A4.3.3 描绘曼德布洛特集合

我们想在一个光栅图形设备中显示 M。为了实现这个，必须建立起显示器中的每个像素和 c 的取值之间的对应关系，还有 c 的值和其响应的 dwell。根据 dwell 是否有限或者它是否会到达某个有限值，给它设定颜色。

```
int dwell(double cx,double cy)
{//返回真正的 dwell 和 Num 中较小的一个
#define Num 100                    //想得到更好的图片就要增加该值
double tmp, dx=cx,dy=cy,fsq=cx * cx+cy * cy;
//开始计算轨迹
for(int count=0;count<=Num && fsq<=4;count++)
{
tmp=dx;                            //保存旧的实部
dx=dx * dx-dy * dy+cx;             //新的实部
dy=2.0 * tmp * dy+cy;              //新的虚部
fsq=dx * dx+dy * dy;               //所用的迭代次数
}
return count;
}
```

图 A4.5 估算 dwell

曼德布洛特集合的这个简单的图片，把在 M 内部的点设定为黑色(dwell 达到最大值)，在 M 外部的点设定为白色(dwell 比最大值小)。如果颜色和 M 外部的点联系起来的话，图片会更加吸引眼球。这些点的 dwell 比最大值小。根据 dwell 的大小不同，我们可以分别设定不同的颜色。dwell 小的值代表"hot"点，这些点的轨迹很快就剧增了，因此它们被标记为鲜艳的红色或者黄色。dwell 大的值代表轨迹"逃逸"的比较缓慢，这些点应该被标记为绿色或者蓝色。图 A4.6 使用五种颜色显示了一个可能的颜色设定的方式。使用的颜色越多，图片会更加漂亮。这些颜色的选择也是很仔细的，以达到最终的图片具有最好的感觉。想了解更多颜色的版本请参看相关站点。

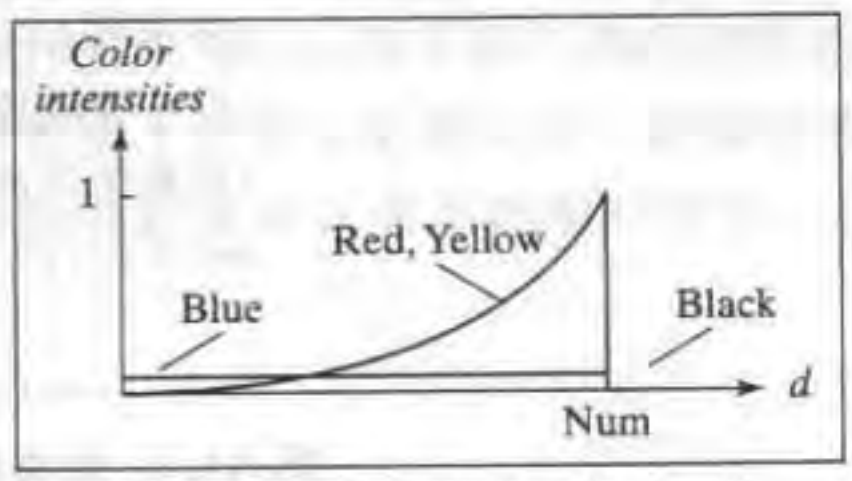


图 A4.6　根据轨迹上的 dwell 来赋予颜色值

还有一个遗留的问题是对于一个特殊的 c 的取值，如何与某个像素联系。图 4.7 给出了一个简单的方法。使用者指定期望的图片在屏幕上的大小，使用数据如下：

行的数目(如 rows=80)

列的数目(如 cols=120)

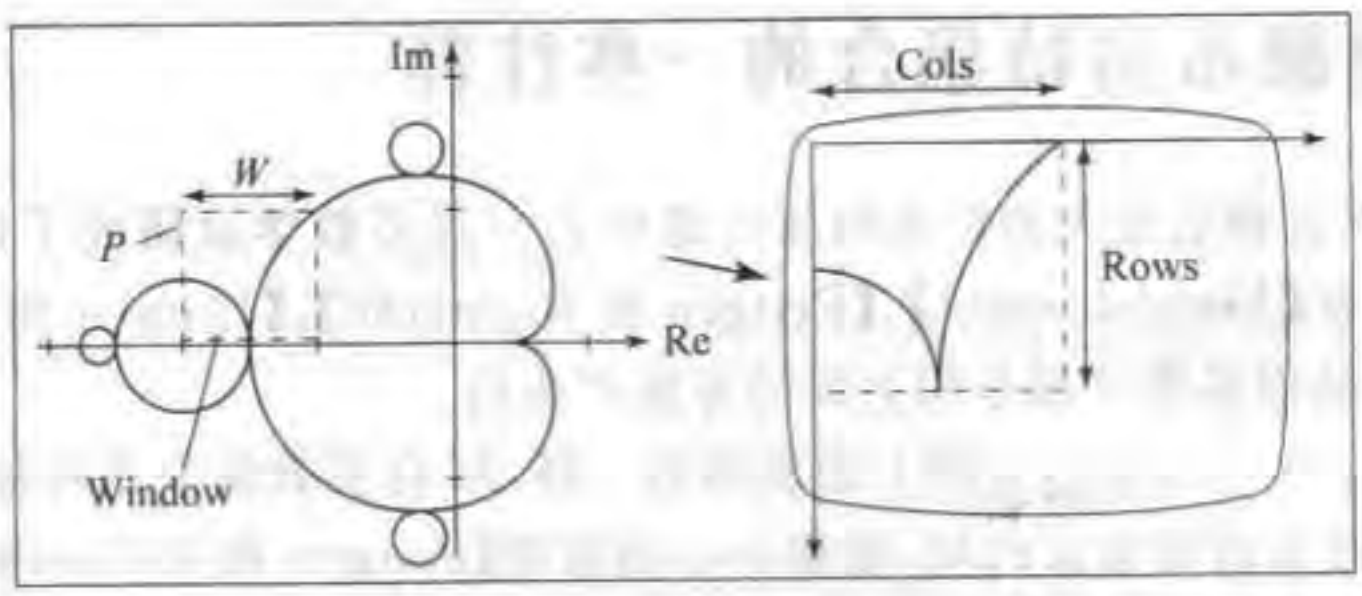


图 A4.7　建立 M 是你干的一个窗口以及点和像素之间的联系

这个决定了图像的纵横比：$R=rows/cols$。使用者还要选择要显示的复数平面的端口：与原图像有相同纵横比的矩形区域。要做到这一点，用户只需要简单地指定左上角 P 以及宽度 w(如：$P=-1+0.5i$ 或者 $P=(-1,5)$，$W=0.5$)。

计算机完成剩下的事情。矩形的高度被设定为需要的纵横比需要的高度。和图像中建议的方式一样，我们在左上角显示图像。

现在第 i,j 个像素的中心对对应的复数值 $c=c_x+c_yi$ 是多少？结合梯度，可以用等式(A4.5)表示。

$$c_{ij}=\left(P_x+\frac{i+\frac{1}{2}}{cols}W,\quad P_y-\frac{j+\frac{1}{2}}{rows}W\right) \tag{A4.5}$$

其中 $i=0,\cdots,cols-1$，$j=0,\cdots,rows-1$。

曼德布洛特集合所选择的区域被逐像素的画出来。对于每个像素，相关的值 c 被传给 dwell()，然后根据计算出来的 dwell 可以选择合适的颜色。该像素也就被设定为这个颜色。图 A4.8 给出了该算法的伪代码。

```
for(j=0;j<rows;j++)
for(i=0;i<cols;i++)
{
<根据式 A4.5 找到响应的 c 的值>
<估计该轨迹的 dwell>
<根据估计出来的 dwell 找到相应的颜色>
setPixel(j,i,color);
}
```

图 A4.8 绘制曼德布洛特集合一个区域的伪代码

一个实际的例子是研究曼德布洛特集合的特写镜头(close-up views),数字需要以高精度方式被记录和操作。需要双精度(或更高精度)的算术操作。同时,当与集合的边界比较近的时候,像上面讨论的一样,应该使用大一点的 Num。因此,当放大 M 的区域的某个边界时,每一副图片的计算次数会增加。但是,适当大小的图片,都能够在合理的时间内被微型计算机构造出来,这个时间是值得期待的。

A4.3.4 曼德布洛特集合的一些注释

曼德布洛特集合和它强大的扩展的属性是什么?很多数学家研究了这个问题,也得到很多有趣的事实(见【Mandelbrot83】、【Peitgen 和 Richter86】、【Peitgen 和 Saupe88】有很多观点)。这本书包括很多图片都是用上面的方法产生的。

1. M 中的点 $c=-1$,$c=-2$ 同上面提到的一样,M 在实数轴上是对称的。
2. 最大的曼德布洛特集合的块,被很小心的检测【Peitgen 和 Richter86】:
   - M 的块中心的边界是个心形线,用参数表示如下:

     $$c.x = 0.25 + 0.5(1 - \cos(t))\cos(t)$$
     $$c.y = 0.5(1 - \cos(t))\sin(t)$$

     对不动点来说,心形线内部的轨迹是引力中心,从 $z^2+c=z$ 的的两个根可以看出。
   - 心形线左边的圆半径为 1/4,中心在 $c=-1$,在圆内的轨迹以 2 为周期。
   - 它左边小一点的圆,半径为 0.0607,中心在 $c=-1.3107$,圆内的轨迹以 4 为周期。
   - 在心形线上面和下面的圆半径都为 0.0954,中心是 $c=-0.1226\pm0.7449i$。在圆内的点多周期是 3。
3. 集合的边界是看起来最有意思的地方,而且事实上他是个不规则曲线。对某个区域不断进行放大,使用越来越小的窗口时,一些新的细节就会出现。图 A4.9 表现了以 $c=0.7469+0.1073i$ 为中心的区域的 3 种连续的放大效果。

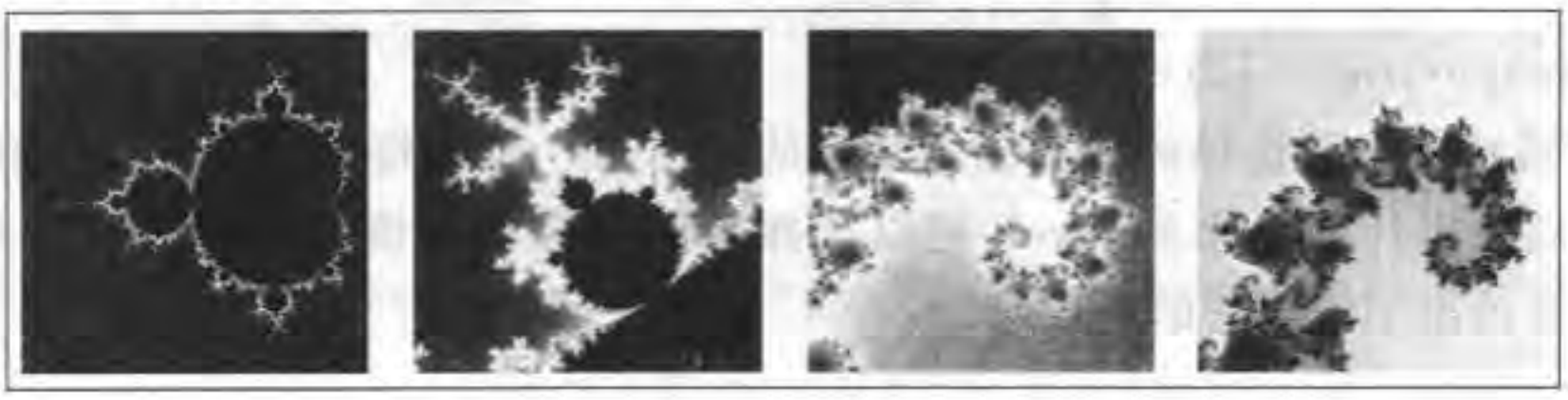


图 A4.9 在一个区域内缩放

序列中最后一幅图片使用的窗口，左下角是（－0.74758，0.10671），右上角是（－0.74624，0.10779）。每次放大时候，都可以看见一些新的细节。比如，一个缩放图中的一个单个的黑点，在下一次放大的时候变成了一个整个的新的瘤状的图像（“……一个小跳蚤拥有更小的跳蚤为它的战利品……；因此是个无穷的过程”）。最后，没有两个微小的缩影是一样的。另外一个被康奈尔大学的 John H. Hubbard 证明了的惊奇的事实是：曼德布洛特集合是连通的【Dewdney88】：在平面上即使是很微小的看起来毫无关系的瘤状结构，在父集合中总是存在很细小的点的序列把它们联系起来。

附录 5

相对性和海龟绘图

概述

在我们的工具包中加入一两个工具，将会获得明显的优势，能使一些画图任务变得更加容易。一般来说，固定一个点称为当前点(currPos)，而描述其他点时采用和当前点的相对位置，这样比较方便。因此，我们在写函数时，函数的参数表示点的变化：程序员指定绘制点必须沿着每个坐标轴走多远才能到达下一个目标点。

A5.1 编写 moveRel()和 lineRel()函数

下面，介绍两个新的程序：moveRel()和 lineRel()。函数 moveRel()较简单：将当前顶点移动(dx,dy)。函数 lineRel(float dx,float dy)实现同样的功能，但是它首先画一条从当前顶点到新的顶点的直线，然后再更新当前顶点。图 A5.1 展示了这两个函数。

```
moveRel(float dx,float dy){
currPos.set( currPos.x+dx , currPos.y+dy);
}
lineRel(float dx ,float dy) {
lineTo(currPos.x+dx, currPos.y+dy);
}
```

图 A5.1 函数 moveRel()和 lineRel()

例 A5.1 数组标记

在绘制的时候，不同形状的标记可以被放在许多点上以增加强调。图 A5.2 表明使用五角星(如图 A5.19 所描述)标记以突出线形图中的数据点。

因为同样的图要在许多不同的点绘制，简单地用 drawMarker ()会很方便，并且在 curPos 处绘制它。这样，图 A5.2 中的线形图能够使用下面的伪代码得到：

```
moveTo(first data point);
drawMarker();                    //在这里画一个标记
for(each remaining data point)
```

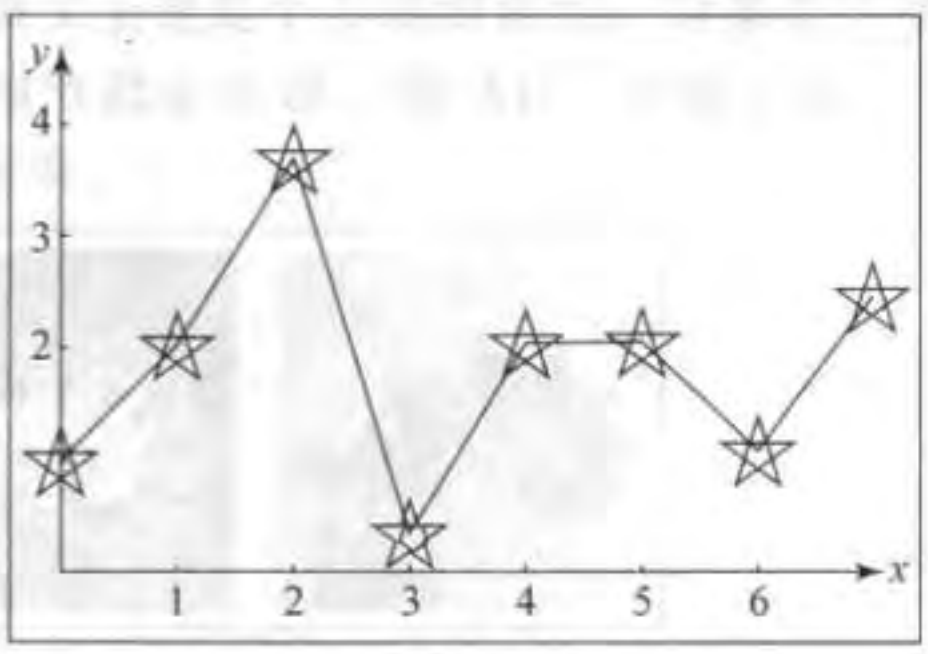


图 A5.2 为强调点放置五角星

```
{
lintTo(the next point);    //画出下一条线段
drawMarker();              //在 CP 画标记
}
```

图 A5.3 表示了一个箭头形状的标记，它是用图 A5.4 中的程序画的。这个箭头的最顶端定点就是 currPos，这样就能确定其位置。为了方便，函数使用 4 个参数 f、h、t 和 w，函数 arrow()只调用了 lineRel()，而且没用绝对坐标。应该注意的是，虽然 currPos 随着画图的进行而改变，绘图结束后 currPos 回到了它的初始位置。因此，这个函数没有产生副作用（在绘制操作自身范围之内）。

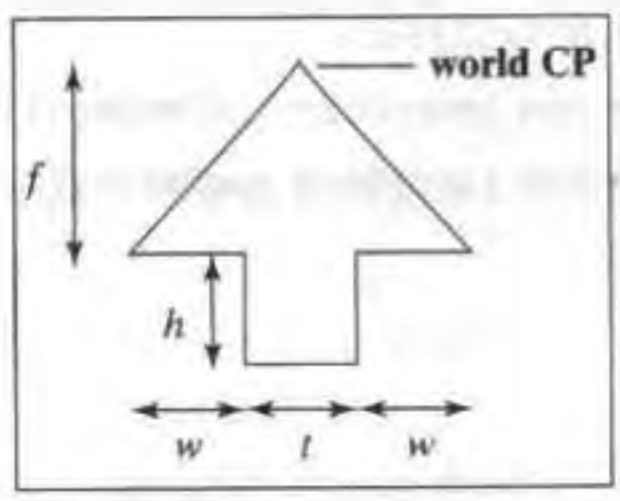


图 5.3 一个箭头模型

```
void arrow(float f, float h, float t, float w)
{
lineRel(-w -t/2, -f);      //往左下的边
lineRel(w, 0);             //往右
lineRel(0, -h);            //往下
lineRel(t, 0);             //王右
lineRel(0, h);             //往上
lineRel(w, 0);             //往右
lineRel(-w -t/2, f);       //往左上，回到最初顶点
}
```

图 A5.4 使用相对画图画一个箭头

A5.2 海龟绘图法

我们增加的另一个工具非常的方便。想法是一个海龟在纸上移动，在它自己的身后留下了一个看似线段的足迹。上述的海龟类似于绘图仪中的一支铅笔。它不仅用 currPos 保存了当前所处的位置，而且保存了当前的朝向。这是海龟绘图法（turtle graphics）的一种形式，是图形学中的一种自然的编程方法[①]。海龟被放在 currPos，朝向一个被称为当前方向的特定方向。currDir 表示从正 x 轴开始逆时针旋转的角度数。

添加函数功能来控制海龟是很简单的。首先添加三个函数：

(1) turnTo(double angle)。这个函数将 turtle 转向给定的角度，其实现如下：

```
turnTo(double angle){
currDir=angle;          //转向一个确定的角度
}
```

(2) turn(double angle)。这个函数将 turtle 逆时针旋转 angle 度，其实现如下：

```
turn(double angle){
```

① MIT 的 Seymour Papert 曾介绍过，是 LOGO 语言的一部分，可以教孩子们编程。参见[Abel81]。

```
currDir+=angle;
}
```

我们使用负的角度表示一个往右的旋转。注意,旋转是一个相对方向变化,我们不指定方向,只是指定方向的一个改变量。这个简单的函数能够对用海龟绘制复杂图形提供强大的功能。

(3) forward(float dist, int isVisible)。forward()稍微复杂一些:海龟从当前点 currPos 沿着当前方向 currDir 划一条长为 dist 的线段。注意,currDir 就像是一个静态变量一样,和背景颜色、画图颜色等变量类似。如果 isVisible 非 0,画一条可见线段,否则不画任何东西。

图 A5.5 表明,海龟在 x 方向移动 dist * cos(pai * currDir/180),在 y 方向移动 dist * sin(pai * currDir/180),所以 forward()的实现如图 A5.6 所示。

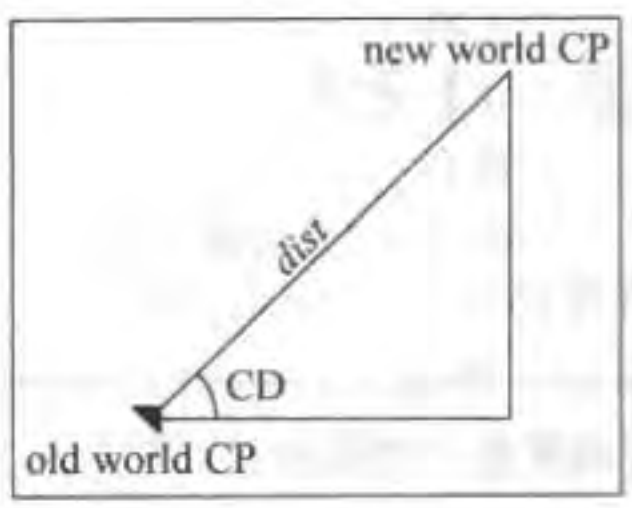


图 A5.5 forward()的功能

```
forward(float dist, bool visible){
const double radPerDeg=0.017453393f;
double x=currPos.x+(dist*cos(currDir*radPerDeg));
double y=currPos.y+(dist*sin(currDir*radPerDeg));
if(visible)
    lineTo(x, y);
else
    moveTo(x, y);
}
```

图 A5.6 forward()的实现

正如下面将要看到的,海龟绘图法让利用简单图形绘制复杂图形的工作变得很简单。

例 A5.2 画一个单位正方形

为了遍历一个单位正方形,只需画四条边,其中相邻两条边之间转动角度为 90°(如图 A5.7 所示),其实现如下面的伪代码。如果想让海龟结束的方向和开始的一致,则在最后加入 turn(90)语句:

```
forward(1,1);        //画一条边
trun(90);            //转第一个角度
forward(1,1);        //画下一条边
turn(90);            //转第二个角度
forward(1,1);        //画下一条边
turn(90);            //转第三个角度
forward(1,1);        //画最后一条边
turn(90);            //恢复到最初的角度
```

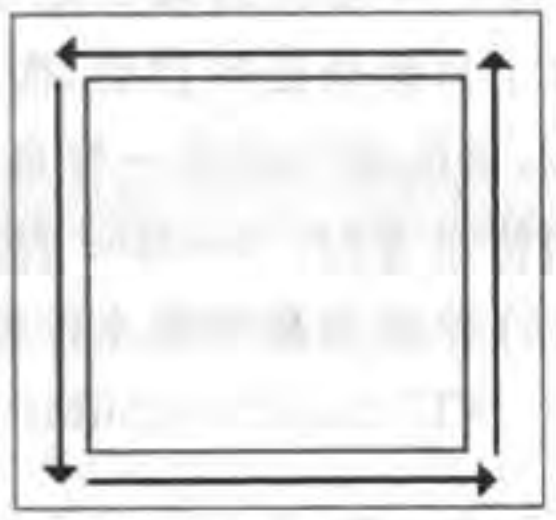


图 A5.7 画一个单位正方形

例 A5.3 建立一个以钩子为基本图案的图形

图 A5.8(a)中由三段组成的基本图案钩子能使用下面的代码绘制:

```
forward(3 * L, 1);     //L是长边的
turn(90);
forward(L, 1);
turn(90);
forward(L,1);
turn(90);
```

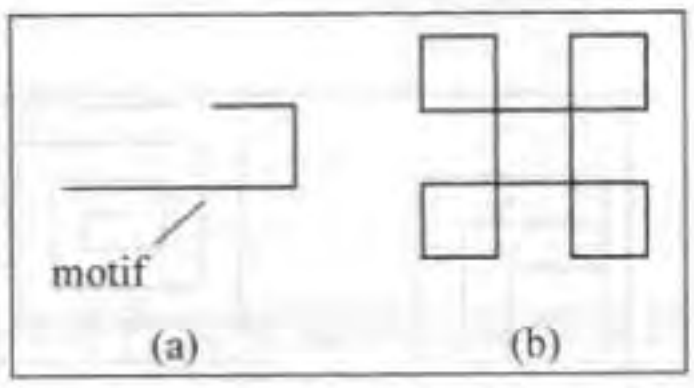


图 A5.8 通过几个海龟的运动来建立一个图形

假设一个函数 hook()包含了上面的代码，则图 5.8(b)中的图形能够用 4 个重复的 hook()画出来。该图能够通过改变初始的 currPos 和 currDir 进行平移和旋转。

例 A5.4 多线螺旋

大量被称为多线螺旋的图形能够很容易通过海龟绘图法产生。一个多线螺旋是一个每条连续线段都比它在某个方向相邻的前一条要大(或小)一个确定的值的多线。其框架类似于：

```
for(<一定数量的循环> )
{
    forward(length,1);        //沿当前方向画一条线
    trun(angle);              //转动 angle 角度
    length+=increment;        //增大线的长度
}
```

每画一条线，线的长度和方向逐步增加。如果增量为 0，则图形既不增长也不收缩。图 A5.9 展示了一些多线螺旋。

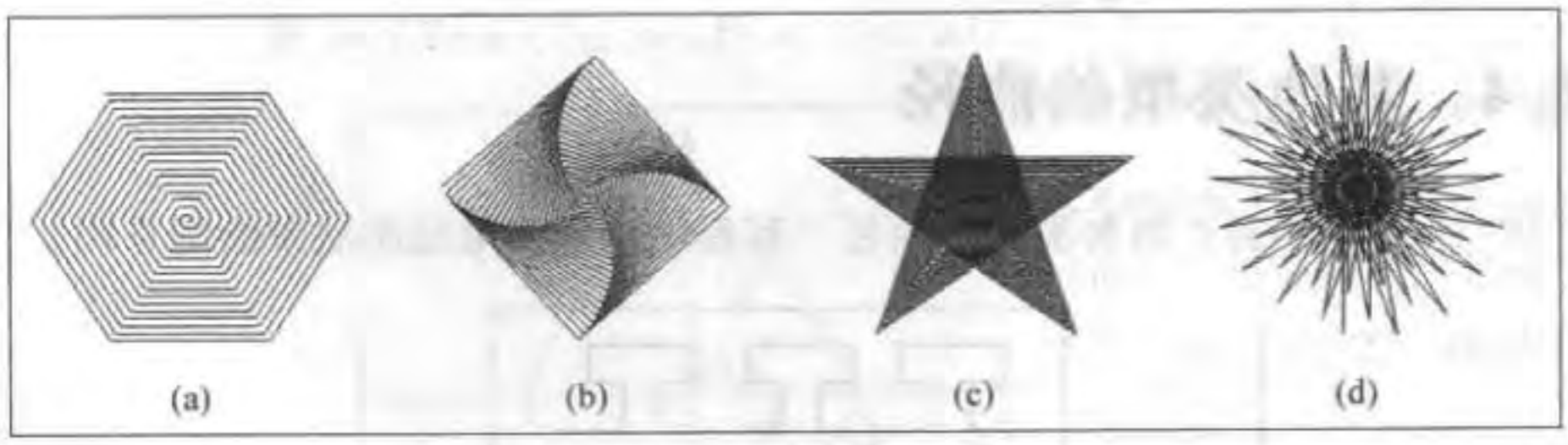


图 A5.9 多线螺旋的一些例子，角度为：
(a) 60；(b) 89.5；(c) −144；(d) 170

练习

练习 A5.2.1 绘制海龟图形

给出用海龟移动来绘制图 A5.10 所示的三个图形的程序。

练习 A5.2.2 绘制一个有名的标记

写一段程序让一个海龟来绘制图 A5.11 所示标记的轮廓(不需要填充多边形)。

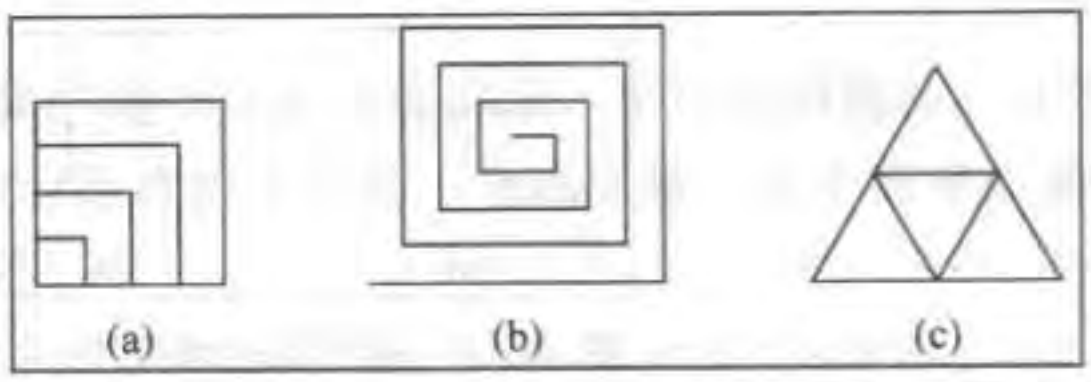


图 A5.10 另外一些简单的海龟图形

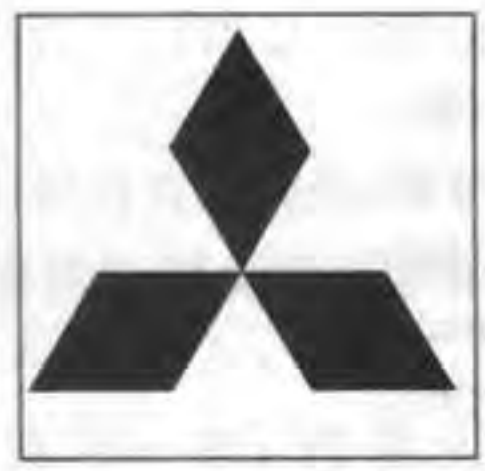


图 A5.11 一个有名的标记

A5.2.3 绘制曲径

曲径是类似于图 A5.12(a)所示的一种模式,经常是由连续的线绕某种路径形成。曲径常见于希腊的花瓶、中国盘子或者各个国家的地砖。这个曲径的基本图案如图 A5.12(b)所示。每个基本图案绘制之后,海龟转向以便准备绘制下一个基本图案。

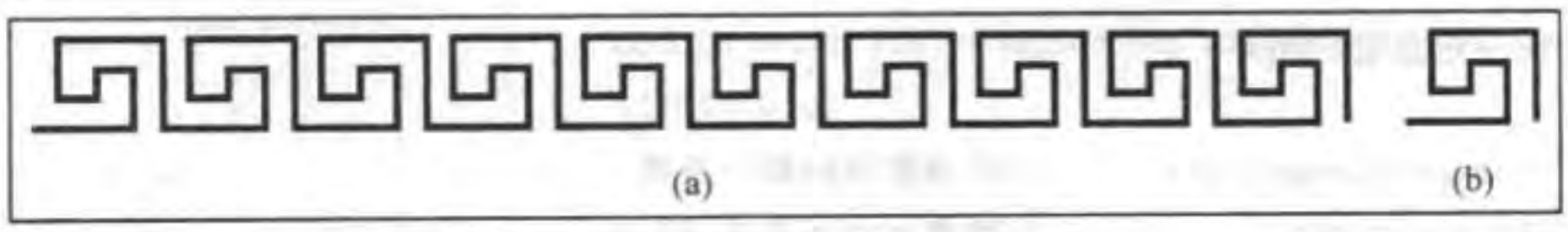


图 A5.12 一个曲径的例子

写一个程序来绘制这个基本图案,再写一个程序来绘制这个曲径(如果手头上有图形软件包可以支持控制线的宽度来画细线的话,曲径就很有吸引力)。正如后面的练习所述,可以设计出各种各样更为复杂的曲径。

A5.2.4 其他类型的曲径

图 A5.13 显示了另外两种类型的曲径。写程序利用海龟绘图法来绘制它们。

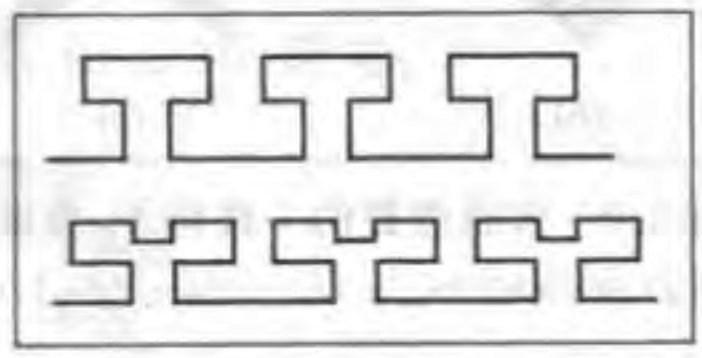


图 A5.13 额外的曲径图

A5.3 基于正多边形的图

正多边形构成了大量计算机图形学中常见的重要类型的形状,需要高效的方法来绘制它们。本节将介绍怎样实现上述功能并介绍怎样创建由正多边形演变的图形。

A5.3.1　正多边形

首先回顾一下正多边形的定义。

定义：如果一个多边形是简单多边形，它的所有边长度相等并且相邻的边有相等的夹角，它是正多边形。

正如第 1 章所述，如果一个多边形的任意两边不相互交叉（更精确的说，只有邻接的边可以接触，而且只能在它们的公共端点接触），则该多边形是简单的。我们给具有 n 条边的多边形起名为 n-gon。常见的例子有 3-gon（等边三角形）、4-gon（正方形）、5-gon（正五边形）和 8-gon（正八边形）等。图 A5.14 演示了各种例子。如果一个 n－gon 的边数很大，这个多边形看起来就像圆。事实上这就是后续将要介绍的绘制圆的一种方法。

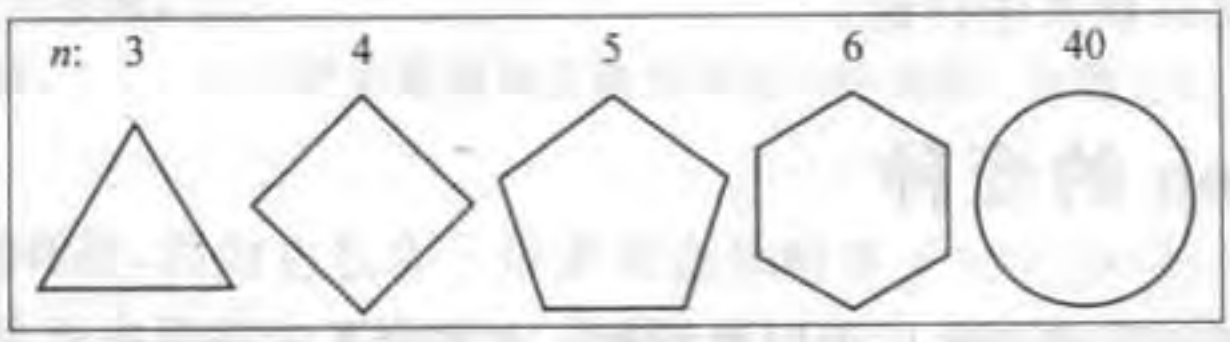


图 A5.14　n-gon 的例子

n-gon 的顶点在一个圆上，就是所谓的 n-gon 的母圆，这些顶点的位置很容易计算。图 A5.15 演示了一个正六边形的实例，其中每个顶点都均匀间隔 60°分布在圆上。半径为 R 的母圆中心在原点，第一个顶点 P_0 被放在正 x 轴上。其他顶点按如下方法分布，$P_i = R\cos(i-a), R\sin(i-a))$，$i=1,\cdots,5$，其中 a 是 $2\pi/6$ 弧度。类似地，普通 n-gon 的顶点位于

$$P_i = (R\cos(2\pi i/n), R\sin(2\pi i/n)), i = 0,\cdots,n-1$$

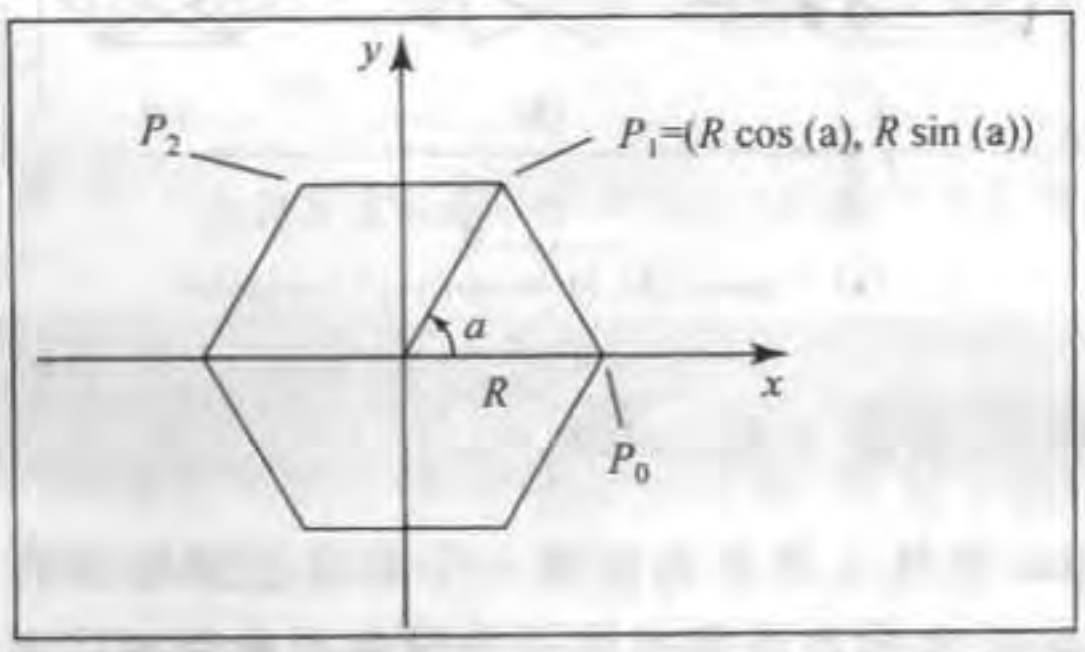


图 A5.15　计算 6-gon 的顶点

修改这个 n-gon 很简单。要想把它的中心调到 (cx, cy)，需要给 x 和 y 坐标分别加上 cx 和 cy。要想用比例因子 S 来缩放它，只需用 S 乘以 R。要旋转角度 A，只需给 sin ()和 cos ()的参数加上 A。实现几何变换更一般的方法在第 6 章中讨论了。正如在书中多次看到的那样，绘制 n-gon 的实现很直接。

例 A5.5 一个海龟-驱动[1]的 n-gon

用海龟绘图法绘制 n—gon 也很简单。图 A5.16 显示了怎样去绘制一个正六边形。海龟的最初位置和方向用小三角来指示。海龟简单地前行 6 次,每次移动之前逆时针转 60°。

```
for (i=0; i<6; i++)
{
    forward(L, 1);
    turn(60);
}
```

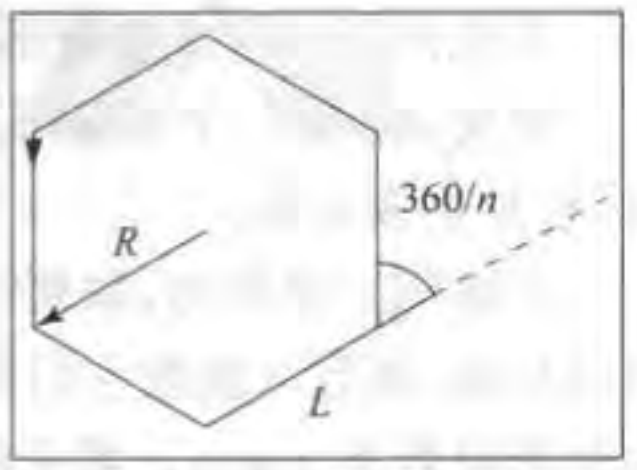


图 A5.16 绘制六边形

一个顶点位于初始 curPos,该操作保持 currPos 和 currDir 不变。绘制一般的 n—gon 及其几个变种的方法在联系中讨论。

A5.3.2 n-gon 的变种

基于 n-gon 顶点的有趣变种也可以被绘制。如图 A5.17 所示,n—gon 顶点可以被以多种方式连接以形成多种图案。图 A5.17(a)显示了通过连接邻接的顶点来绘制一个标准的 n-gon,而图 A5.17(b)则显示了通过连接每隔一个的点来形成 stellation(或星形图)。图 A5.17(c)显示了通过连接每个顶点和其他所有顶点形成 rosette。然后讨论 rosette。其他图案将在练习中描述。

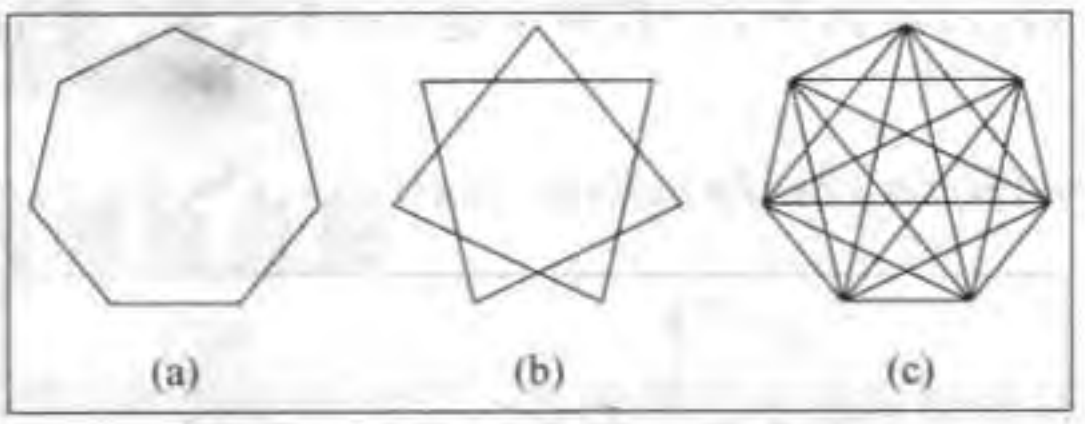


图 A5.17 一个 7-gon 及其后代

(a) 7-gon; (b) stellation; (c) 7-rosette

例 A5.6 Rosette 和黄金 5-Rosette

rosette 是一个 n-gon 中每个顶点和相隔一个的点连接形成的。图 A5.18 显示了 5-、11-,和 17-rosettes。rosette 经常被计算机图形学设备用来做测试图案。它有秩序的形状很容易揭示任何失真,而且设备的分辨率也可以通过观察遇到顶点时线条间的拥挤和模糊程度来确定。

Rosette 很容易被绘制:简单地将每个顶点和其他顶点相连。伪代码形式类似于

```
void Rosette(int N, float radius)
{
```

[1] Turtle-Driven 指利用上文所述的海龟绘图法。

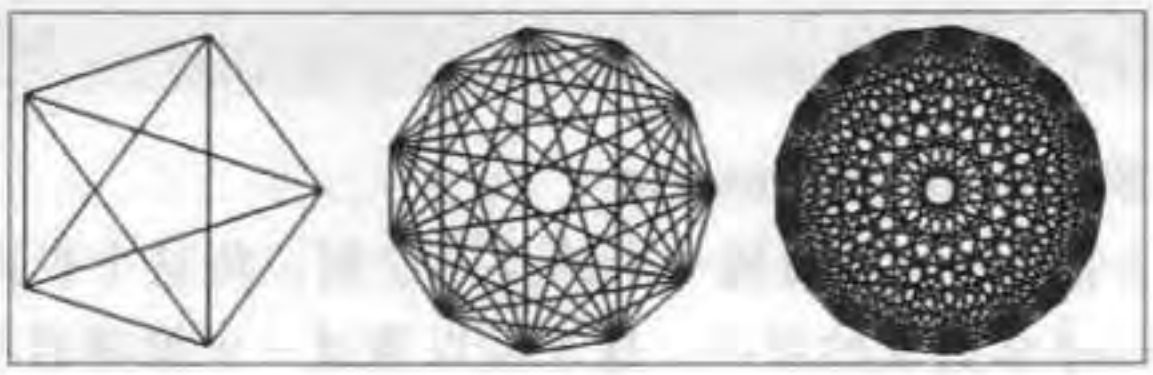


图 A5.18 5、11 和 17-rosettes

```
Point2 pt[< 针对最大的 rosette 的足够大的数> ];
< 生成图 3.43 中类似的顶点 pt[0], … pt[N-1]>
for(int i=0; i<N; i++)
for(int j= i+1; j<N; j++)
{
<forward(正确的距离, 1)>
turn(正确的角度);     // 正确的距离和正确的角度一样对每一步都相同
}
```

5-rosette 特别有趣，因为它包含了很多黄金分割率 ϕ 的实例（回顾第 2 章）。图 A5.19(a)显示了一个由外五边形和内五角形组成的 5-rosette。希腊人看到了这张图里的神秘意义。它的分隔有有趣的意义：每段都比下一个更小的长 ϕ 倍（见练习）。同样，由于五角星的边形成了内五边形，五边形的无穷回归是可能的，如图 A5.19(b)所示。

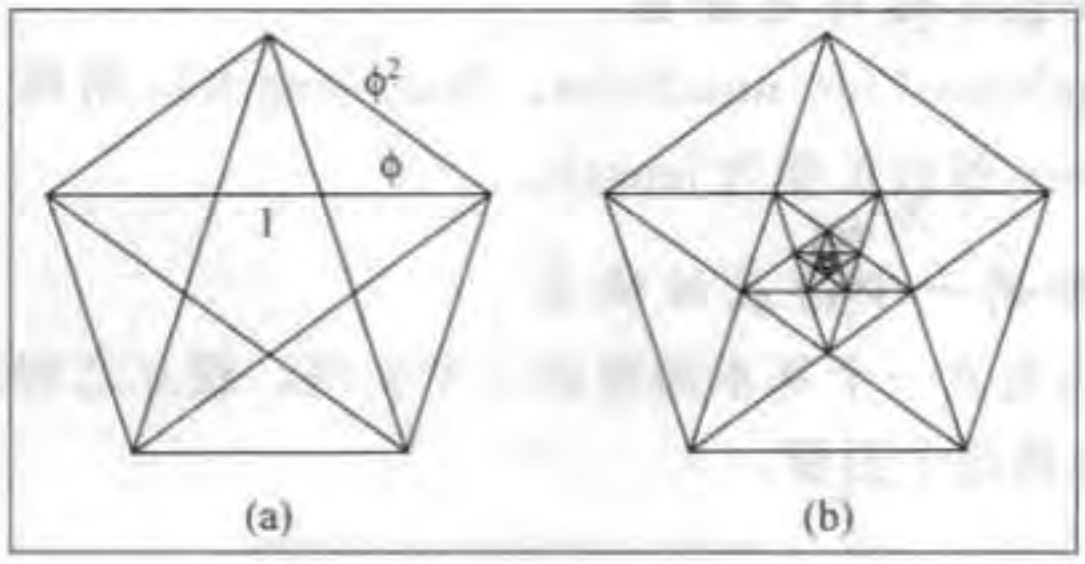


图 A5.19 5-rosette 和无限回归——五边形和五角星

例 A5.7 基于两个共心 n-gon 的图形

图 A5.20 显示了一些建立于两个同心母圆的形状，对某个因子 f 外半径 R 和内半径 fR。每个图用了一个半径在内外半径间相互交替的 n-gon 的一个变种。图 A5.20(a)和图 A5.20(b)两部分演示了基于 6-gon 和 10-gon 的熟悉的公司商标。图 A5.20(c)基于 14-gon，图 A5.20(d)显示的表现了内圆。

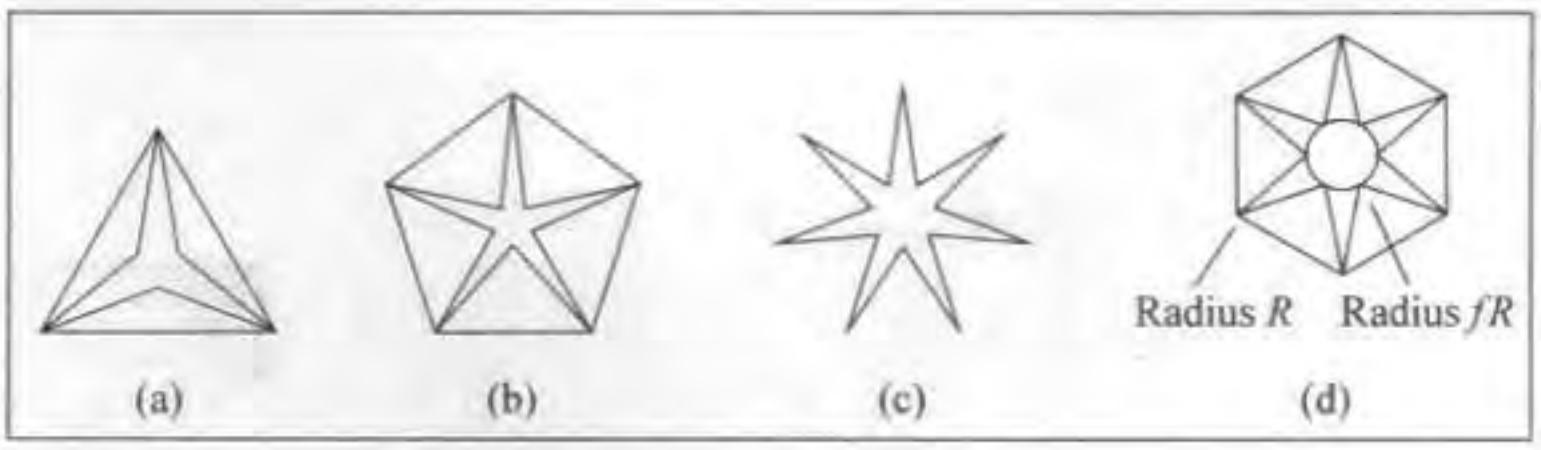


图 A5.20 熟悉的 n-grams 和 n-gons

练习

练习 A5.3.1 Stellation 和 rosette

五角星通过遍历 5-gon 并连接每隔一个的点来绘制。将这个扩展到任意的奇数 n-gon 并开发绘制所谓 stellated 多边形的程序。这个可以通过一个简单的初始化 moveTo()后面仅跟 lineTo()来完成吗(也就是说,不能抬笔)? n 是偶数又如何?

练习 A5.3.2 N-rosette 有多少条边?

证明基于 N-gon 和 N-rosette 的 rosette 有 $N(N-1)/2$ 条边。这和 N 个人坐成一圈每个人都和其他所有人一起敲玻璃的时候一个人所能听到的敲击声数目一样。

练习 A5.3.3 星形的几何形状

证明 5-rosette 上的每段长度都和下一个更小的满足黄金分隔率。解决这个问题的一个办法是证明五角星的角是具有内角弧度为 $\pi/5$ 的黄金角。证明 $2*\cos(\pi/5)=\phi$ 且 $2*\cos(2\pi/5)=1/\phi$。另外一个方法是仅用五角星中的两组相似角及 ϕ 所满足的关系 $\phi^3=2\phi+1$。

练习 A5.3.4 用相关的移动和绘制来画星

写一个程序,只用相关的移动和绘制来画中心在 currPos 的五角星。

练习 A5.3.5 n-gon 的海龟绘图

写一个程序 turtleNgon(int numSides, float length),用海龟绘图法来画一个有 numSides 个边的 n-gon,每边长度为 length。

练习 A5.3.6 绘制一个潜在的标志

图 A5.21 所示标志包含一个基本图案的三个实例。相互之间旋转了一定的角度。写一个程序用海龟绘图法画这个图形。

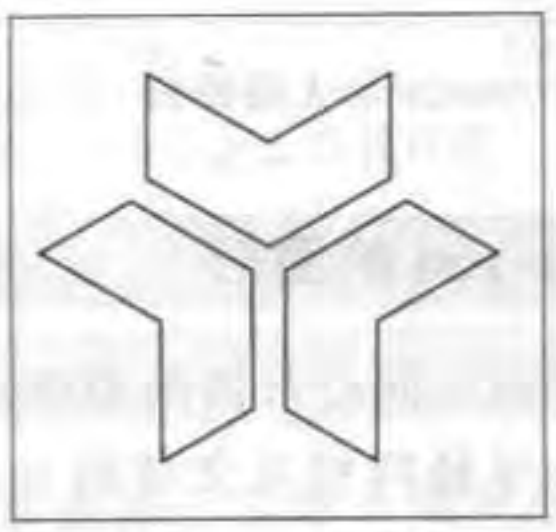


图 A5.21 一个标志的设计

COMPUTER GRAPHICS USING OpenGL

计算机图形学

（OpenGL版）（第3版）

计算机图形学是利用计算机研究图形的表示、生成、处理和显示的一门重要的计算机学科分支，而且是计算机科学中最活跃的分支之一。近年来，随着计算机及互联网技术的迅速发展，计算机图形学正越来越深入我们的生活，它在工业建模、视频处理、游戏制作、影视特技、生物信息和医药医疗等各行各业都有着及其重要的作用。可以说，计算机图形学的应用无所不在，且日益广泛；已经成为计算机科学技术与其他应用学科之间沟通的桥梁，成为许多计算机从业人员的必备素质之一。

本书是一本国外很有影响的教材，为许多国外著名大学所采用。本书通过最能代表技术发展状况的示例综合介绍了计算机图形学方面的原理和技术，本书对每个概念都进行了详细介绍，阐述了其背后的数学原理，并给出了用OpenGL实现的代码以及实现结果展示。新版本还为读者提供了计算机图形学领域的最前沿信息。本书是计算机图形学课程的很好教材，也是计算机图形学专业人员的很好参考书。

本书特点

* 本书配套的网站提供了书中使用到的代码、彩色插图、完整程序以及与本书密切相关的参考素材。
* 以C++语言作为程序设计语言。
* 深入介绍3D图形学及其数学知识，使学生可以创建真实的3D图形。
* 每章后面都给出了实际的案例研究。
* 清晰地介绍了计算机图形学的原理和技术。
* 大量真实而逼真的示例。
* 大量实践练习（每章大概有30多道）。

PEARSON
Prentice Hall

http://www.pearsoned.com

ISBN 978-7-302-18697-7

9 787302 186977 >

定价：99.00元