

# 图形瑰宝

第三册

david kirk 著

郭飞 译

2022年



# 目录

<b>第 1 章 图像处理</b>	<b>5</b>
1.1 位图快速拉伸	6
1.2 一般的滤波图像缩放	10
1.3 位图缩放操作的优化	17
1.4 一个简单的减色滤镜	19
1.5 从抽样数据得到的紧凑等高线	22
1.6 从位图生成等值线	27
1.7 合成黑白位图	31
1.8 计算机动画的 $2\frac{1}{2}D$ 景深模拟	32
1.9 复合区域的快速边界生成器	35
<b>第 2 章 数值和编程技术</b>	<b>41</b>
2.1 IEEE 快速平方根	42
2.2 一个简单的快速内存分配器	42
2.3 滚动球	44
2.4 区间运算	52
2.5 快速生成循环序列	57
2.6 一个通用的像素选择机制	66
2.7 经扭曲的非均匀随机点集	69
2.8 II.7 NONUNIFORM RANDOM POINTS VIA WARPING	72
2.9 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND	73
2.10 Introduction	73
2.11 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND	74
2.12 Tensor Product	74

2.13 Wedge Product . . . . . 75

2.14 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND 75

2.15 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND 76

2.16 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND 77

2.17 Acknowledgment . . . . . 78

# 第 1 章 | 图像处理

本章中的所有算法都涉及对图像或像素的二维数组执行的操作。通常，图形程序员可能想要更改图像的大小、颜色或其他功能。前三个算法描述了在各种环境下拉伸或缩放图像的技术。第一条强调速度，而第二条强调质量。第四个算法描述了一种使用简化的颜色集显示全彩色图像的方法。

在某些情况下，将多个图像的特征组合起来是很有用的。第七个算法将我们现在熟悉的图像组合代数应用于黑白位图或1位图像。第八个算法讨论了如何有选择地模糊两幅图像，同时结合它们，以模拟相机的光圈景深效果。

有时，期望的结果不是另一个图像，而是图像中某些特征的另一种表示。第五、第六和第九算法描述了从图像中提取区域边界信息的技术。

## 1.1 位图快速拉伸

Tomas Möller

Lund Institute of Technology Hoganas, Sweden

### 介绍

这里提出了一个整数算法，用于将位图的任意水平线或垂直线拉伸到任何其他任意直线上。该算法可用于要求接近实时性或实时性的绘图和绘图程序。应用程序区域的例子包括扩大和缩小位图的矩形区域，以及将矩形区域包装到圆形区域上。

### 算法

程序本身非常简单，大多数计算机图形程序员可能都熟悉它所基于的Bresenham划线算法(1965)。事实上，它可以基于任何画线算法；然而，Bresenham被选中，因为它是基于整数的，并且在计算机图形社区中非常普遍。对于那些不熟悉Bresenham算法的人来说，伪代码用于在第一个八边形中绘制线段。

上面的伪代码也适用于第二个八分位数，但是在这种情况下，行不会是连续的，因为 $x_1$ 总是递增1。这非常适合算法。

让我们回到拉伸算法的解释上。 $x_1$ 和 $y_1$ 不能被解释为二维直线上的一对坐标，它们必须被解释为一维坐标。 $dx$ 必须解释为目标线的长度， $dy$ 是源线的长度。使用这些解释， $x_1$ 是目标线上的坐标， $y_1$ 是源线上的坐标。对于目标线上的每个像素，从源线上选择一个像素。这些像素以统一的方式选择。参见图1.1.1。

如果 $dx$ 大于 $dy$ ，那么目标线比源线长。因此，在绘制到目的线上时，源线将被放大。另一方面，如果 $dy$ 大于 $dx$ ，源线就会减小。如果 $dx = dy$ ，算法会得到与源相同的直线。下面是伪代码中完整的stretcher算法，重写后能够处理 $x_2 < x_1$ 和 $y_2 < y_1$ 的行。

如果 $x$ 等于0，那么符号函数不需要返回0，因为 $dx$ 或 $dy$ 都等于0，这意味着一条长度为1的直线。由于该算法只使用整数运算，而不使用乘法或除法，因此非常高效和快速。

这个小程序的另一个有趣之处是，它可以用来生成几种不同形状的位图。下面列出了一些可以用来渲染的东西。

**算法 1:** Line( $x_1, y_1, x_2, y_2$ )**input** : 由点( $x_1, y_1$ )和点( $x_2, y_2$ )构成的线段**output:** 将线段绘制到位图上

// 从( $x_1, y_1$ )到( $x_2, y_2$ )在第一个八边形内画一条线; 所有变量都是整数

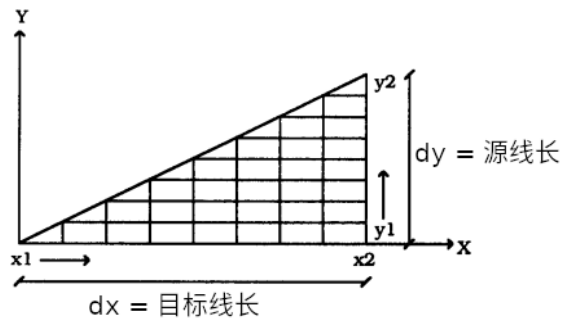
 $dx \leftarrow x_2 - x_1$  $dy \leftarrow y_2 - y_1$  $e \leftarrow 2 * dy - dx$ **for**  $i \leftarrow 1, i \leq dx, i \leftarrow i + 1$  **do**    WritePixel ( $x_1, y_1$ ) // 在图上显示( $x_1, y_1$ )    **while**  $e \geq 0$  **do**         $y_1 \leftarrow y_1 + 1$          $e \leftarrow e - 2 * dx$      $x_1 \leftarrow x_1 + 1$      $e \leftarrow e + 2 * dy$ 

图 1.1.1: 源线与目标线

---

**算法 2:** Stretch( $x_1, y_1, x_2, y_2, y_r, y_w$ )

---

**input :****output:**

// 从( $x_1, y_1$ )到( $x_2, y_2$ )在第一个八边形内画一条线; 所有变量都是  
整数

// 将源线( $y_1 \quad y_2$ )延伸到目标线( $x_1 \quad x_2$ )。

// 源线和目标线都是水平的

//  $y_r$  =从其中读取像素的水平线

//  $y_w$  =要写入像素的水平线

// ReadPixel( $x, y$ )返回像素( $x, y$ )处的颜色

// WritePixel( $x, y$ ) 用当前颜色写入( $x, y$ )处的像素

// SetColor(Color) 设置当前写入的颜色为Color

$dx \leftarrow \text{abs}(x_2 - x_1)$

$dy \leftarrow \text{abs}(y_2 - y_1)$

$sx \leftarrow \text{sign}(x_2 - x_1)$

$sy \leftarrow \text{sign}(y_2 - y_1)$

$e \leftarrow 2 * dy - dx$

$dx2 \leftarrow 2 * dx$

$dy \leftarrow 2 * dy$

**for**  $i \leftarrow 1, i \leq dx, i \leftarrow i + 1$  **do**

$\text{color} \leftarrow \text{ReadPixel}(x, y)$

    SetColor (Color)

    WritePixel( $x_1, y_w$ )

**while**  $e \geq 0$  **do**

$y_1 \leftarrow y_1 + sy$

$e \leftarrow e - dx2$

$x_1 \leftarrow x_1 + 1$

$e \leftarrow e + 2 * dy$

---



## 一些项目使用位图扩展器

- 包裹在圆形或椭圆形区域上的矩形图片。关于绕圈，请参阅附录中的源代码。
- 放大和缩小位图的矩形部分。参见附录中的源代码。
- 将位图的矩形部分绕平行梯形包绕。例如，一个绕x或y轴旋转，然后进行透视转换的矩形可以用作目标形状。

## 进一步的工作

为了改进算法，也许可以添加一个抗锯齿例程。

See also G1, 147; G1, 166; G3, A.2.

## 1.2 一般的滤波图像缩放

Dale Schumacher  
St. Paul, Minnesota

栅格图像可以看作是连续二维函数 $f(x, y)$ 的样本的矩形网格。这些样本被假定为连续函数在给定样本点的精确值。理想的光栅图像缩放程序包括重建原始的连续函数，然后以不同的速率重新采样该函数(Pratt, 1991;Foley *etal.*, 1990 )。采样率越高(采样越靠近)，采样就越多，图像也就越大。采样率越低(采样间隔越远)产生的样本越少，因此图像越小。幸运的是，我们不需要真正地重建整个连续函数，而只是确定重建函数在与新样本对应的点上的值，这是一个更容易的任务(Smith, 1981)。仔细选择过滤器，这个重采样过程可以分两步进行，首先水平地拉伸或缩小图像，然后垂直地拉伸或缩小(反之亦然)，可能有不同的比例因子。双通道方法的运行时成本 $O(\text{image\_width}*\text{image\_height}*(\text{filter\_width} + \text{filter\_height}))$ 比简单的二维过滤 $O(\text{image\_width}*\text{image\_height}*\text{filter\_width}*\text{filter\_height})$ 要低得多。

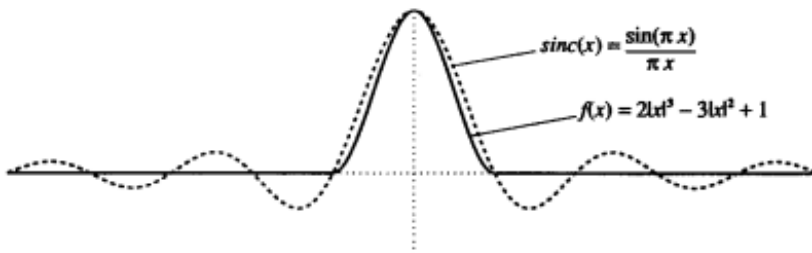


图 1.2.1: *sinc*示例

放大图像的过程有很多名字，包括放大、拉伸、缩放、插值和上采样。我将把这个过程称为放大。缩小图像的过程也有很多名字，包括缩小、缩小、按比例缩小、抽取和下采样。我将把这个过程称为简化。这些过程将在一维而不是二维中进行解释，因为缩放是在每个轴上独立进行的。

在放大过程中，我们通过应用滤波函数来确定每个源像素对每个目标像素的贡献。采样理论表明，*sinc*函数 $f(x) = \sin(x)/x$ 是理想的重构函数;然而，我们有一个有限的样本集，并且需要一个具有有限支持的过滤器(即过滤器非零的区

域)。我在这个例子中使用的过滤器是一个三次函数， $f(x) = 2|x|^3 - 3|x|^2 + 1$ ，从- 1到+1，当单独应用时，它覆盖了每个样本的单位体积。图1比较了这些过滤函数。重采样滤波器的设计是一个没完没了的争论的来源，超出了这个宝石的范围，但在许多其他作品中讨论(Pratt, 1991; Turkowski, 1990; Mitchell, 1988; Smith, 1982; Oppenheim and Schaffer, 1975; Rabiner and Gold, 1975)。为了应用滤镜，我们将滤镜函数的副本放在每个源像素的中心，并缩放到该像素的高度。对于每个目标像素，我们计算源图像中相应的位置。我们将这一点上加权滤波函数的值相加，以确定目标像素的值。图1.2.2说明了这个过程。

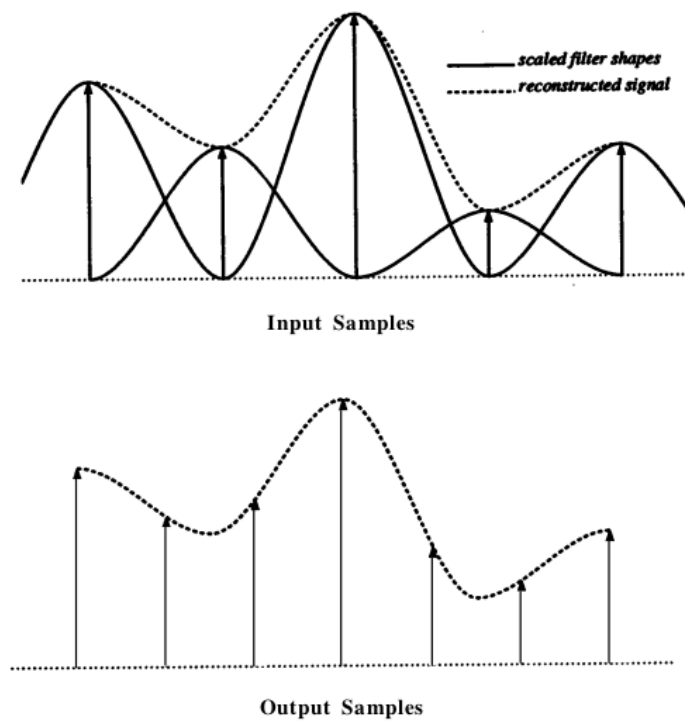


图 1.2.2: 滤波示例

在缩小过程中，过程是相似的，但不完全相同，因为我们必须考虑频率混叠。采样理论将奈奎斯特频率定义为能够正确捕获连续源信号中所有频率成分的采样率。奈奎斯特频率是源信号中最高频率分量频率的两倍。任何频率成分高于采样率的一半将被不正确地采样，并将被混叠到一个更低的频率。

因此，重构信号将只包含采样率的一半或更少的频率成分。在放大过程中，我们拉伸重构信号，降低其分量频率。然而，在缩小过程中，我们正在缩小重构

信号，提高其分量频率，并可能超过我们新的采样率的奈奎斯特频率。为了创建合适的样本，我们必须消除重采样奈奎斯特频率以上的所有频率成分。这可以通过图像缩小因子拉伸滤波函数来实现。此外，由于每个源像素处的滤波器更宽，和将按比例更大，并应除以相同的因子进行补偿。图1.2.3说明了这个过程。

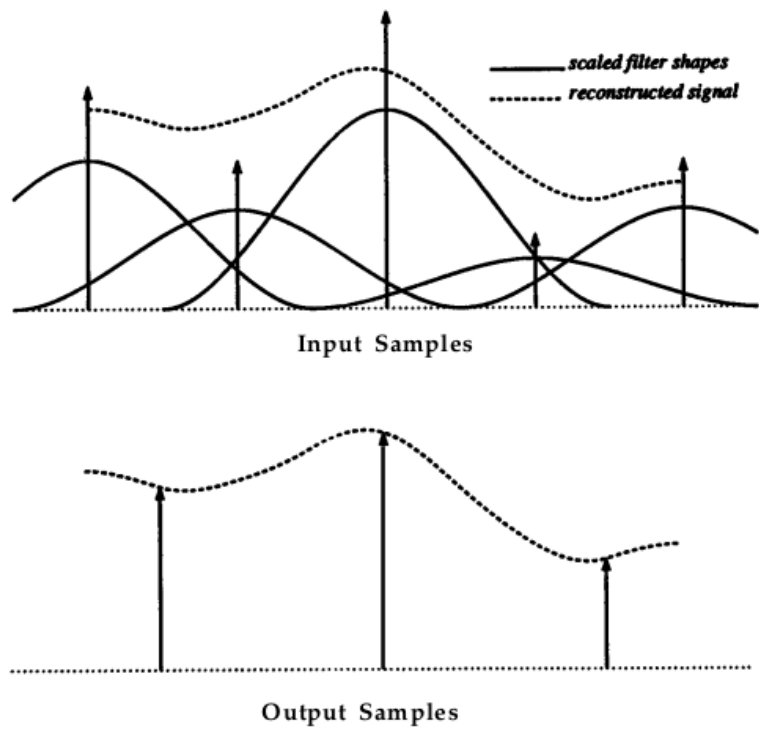


图 1.2.3: 滤波示例

到目前为止，我们只考虑了一维情况。我们将其扩展到典型的光栅图像的二维情况，首先水平缩放，然后垂直缩放。这里将不再演示缩放最小目标轴的进一步优化。滤波操作会导致大量的计算，所以我们尽可能多地预先计算。每个行(或列)的缩放过程是相同的。过滤器的位置和面积是固定的;这样，我们就可以预先计算出每个目标像素的贡献者和相应的滤波器权值。计算目标像素贡献者的伪代码如3。

在计算出贡献之后，目标图像的所有行(或列)都可以使用相同的预计算的过滤器值进行处理。下面的伪代码4显示了这些值用于扩展单个目标行的应用程序。

然后将相同的过程应用于图像的列—首先根据垂直比例因子(可能与水平比

---

**算法 3:** calculate\_contributions(destination)

---

**input :****output:**

```

scale  $\leftarrow$  dst_size/src_size;
center  $\leftarrow$  destination/scale;
if scale < 1.0 then
    width  $\leftarrow$  filter_width/scale;
    fscale  $\leftarrow$  1.0/scale;
else width  $\leftarrow$  filter_width;
fscale  $\leftarrow$  1.0;
;
left  $\leftarrow$  floor (center - width);
right  $\leftarrow$  ceiling (center + width);
for source  $\leftarrow$  left, source  $\leftarrow$  source + 1, source  $\leq$  right do
    weight  $\leftarrow$  filter ((center - source)/fscale)/fscale;
    add_contributor (destination, source, weight);

```

---



---

**算法 4:** scale\_row(destination\_row, source\_row)

---

**输入:****输出:**

```

for i  $\leftarrow$  0, i  $\leftarrow$  i + 1, i < dst_size do
    v  $\leftarrow$  0;
    for j  $\leftarrow$  0, j  $\leftarrow$  j + 1, j < contributors[i] do
        s  $\leftarrow$  contributor[i][j];
        w  $\leftarrow$  weight_value[i][j];
        v  $\leftarrow$  v + (source_row[s]*w);
    destination_row[i]  $\leftarrow$  v;

```

---

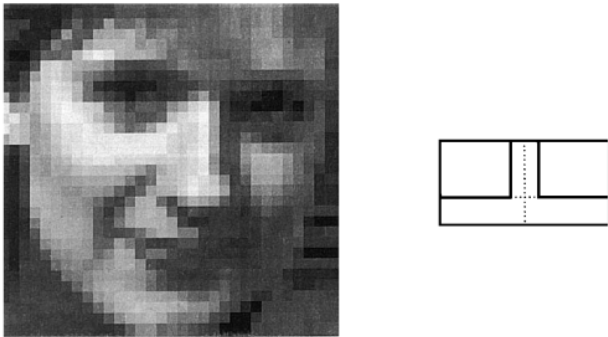


图 1.2.4: 滤波示例

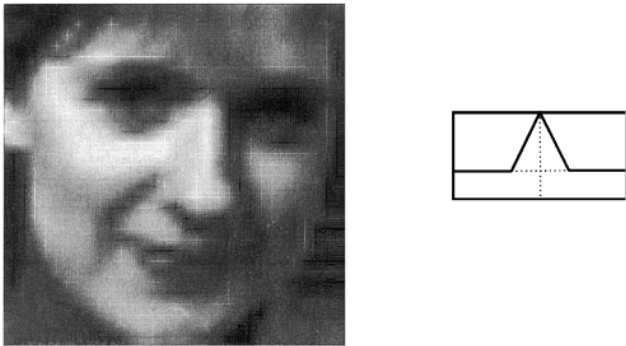


图 1.2.5: 滤波示例



图 1.2.6: 滤波示例



图 1.2.7: 滤波示例



图 1.2.8: 滤波示例

例因子不同)预计算过滤器的贡献, 然后处理从中间(水平比例)图像到最终目标图像的列。

在附录中提供的源代码中, 给出了许多过滤器函数, 可以很容易地添加新的函数。`zoom()`函数以所需过滤器的名称和过滤器支持作为参数。图1.2.4到图1.2.8显示了不同滤波器对样本图像的影响, 以及每个滤波器函数的脉冲响应图。

样本图像在两个方向上都按比例放大了12倍。图1.2.4显示了一个框过滤器, 它相当于直接复制像素值, 因为它显示了相当多的平铺或“锯齿”。图1.2.5显示了三角形或Bartlett滤波器, 相对于方框来说是一个相当大的改进, 计算仍然简单, 但仍然有明显的过渡线。图1.2.6显示了一个三次b样条, 它没有产生尖锐的过渡, 但是它的宽度导致了过度的模糊。三角形和b样条函数分别通过盒形滤波器与自身卷积1次和3次来计算。图1.2.7显示了Lanczos 3滤波器, 一个在 $-3$ 到 $+3$ 范围外衰减为零的 $\text{sinc}$ 函数, 它显示了在完全 $\text{sinc}$ 函数时出现的过度的“振铃”效应。图1.2.8显示了Mitchell滤波器( $B = \frac{1}{3}, C = \frac{1}{3}$ ), 一个没有急剧过渡的三次函数, 以及“振铃(ringing)”和“模糊(blurring)”效果之间的一个很好的折衷。

See also G1, 147; G1, 166; G3, A.1.



## 1.3 位图缩放操作的优化

Dale Schumacher

St. Paul Minnesota

这一节描述了一系列位图缩放操作的优化。我们没有给出一般的缩放算法，而是利用了几个特定于应用程序的限制，这些限制允许显著减少执行时间：每像素图像的位、已知的源和目标位图大小、以及位压缩的水平光栅存储和显示格式。示例应用程序是在典型的视频监视器上显示传真位图图像。

我们首先假设在内存中有源FAX，未压缩，存储为8位字节，每个字节的高阶位表示沿水平行的一组8个字节中最左边的像素。此外，在选择示例缩放因子时，我们假设源FAX的分辨率在两个方向上都是200点每英寸。如果数据是常用的 $200 \times 100$  dpi格式，我们可以通过复制每个扫描线使其为 $200 \times 200$  dpi，这是我们在解压阶段经常可以处理的任务。最初，我们将假设数据存储为与显示所用的位值匹配的白色和黑色位值。下面将讨论一种反演0位和1位含义的好方法。最后，我们假设目标位图的格式与源位图相同。

由于我们的示例图像分辨率高于您的典型视频监视器，我们将只考虑缩小图像的情况，而不是放大它。同样，我们用8来表示比例因子， $\frac{7}{8} = 87.5\%$ ， $\frac{6}{8} = 75\%$ ， $\frac{5}{8} = 62.5\%$ ， $\frac{4}{8} = 50\%$ ， $\frac{3}{8} = 37.5\%$ ， $\frac{2}{8} = 25\%$ 。一般算法工作如下：从源图像中取一条扫描线。对于每个字节，使用字节值作为查找表的索引，该查找表给出给定输入字节的简化位。将派生的输出位移位到累加器中。每个输入字节加到累加器的位数是基于比例因子的（例如，如果我们减少到 $\frac{5}{8}$ 比例，我们为每个8位输入生成5位输出）。当累加器中至少有8位时，我们从累加器中删除最左边的8位，并将它们作为输出字节写入目标扫描行。扫描线末端剩余的任何位都会被移到相应的位置并输出。许多源扫描线可以完全跳过，再次基于比例因子（例如，在 $\frac{5}{8}$ 的比例下，我们每8个扫描线中只处理5个，跳过3个）。

既然已经理解了基本的算法，我们可以讨论一些有用的变化和改进过程。该算法的核心是约简查找表。如果我们需要反转最终图像中的黑白图像，一种方法是反转查找表中存储的位。然后，它将映射到1111b，而不是00000000b映射到00000b。这本质上给了我们在缩放期间免费的光度反演。类似地，通过仔细创建查找表，我们可以解决另一个问题，同样是免费的。如果我们缩小到 $\frac{3}{8}$ 的比例，我们将在每8位中寻找3个来输出。图1.3.1a显示了这样做的最简单的方法。

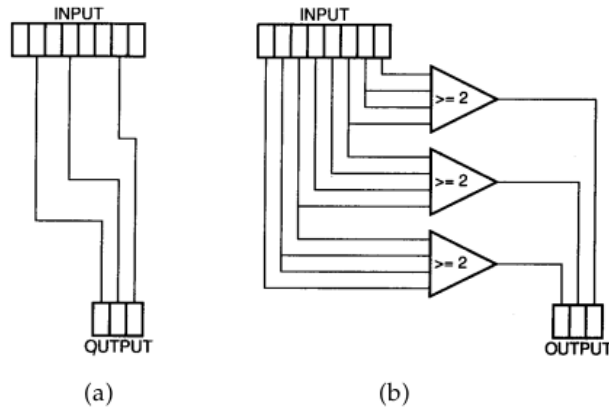


图 1.3.1: 缩放操作

一个更好的方法是模拟一种对源比特进行滤波或加权平均的形式，如图1.3.1b所示。由于查找表可以在编译时创建，所以使用更复杂的算法创建表的计算成本与运行时性能无关。为了进行适当的过滤缩放，我们真的应该跨相邻扫描线和跨字节边界应用过滤器。由于这些操作会带来很高的运行时成本，并且以有限的方式应用过滤，我们可以在没有额外成本的情况下展示改进，所以我们做的是便宜的事情。即使在这些限制条件下，使用滤波也比直接对输入进行子采样要好，如图1.3.1a所示。

您喜欢的任何类型的传递函数都可以以同样的方式应用，在8位跨度的限制内，并且仅以黑白作为输入值。您甚至可以做一些事情，如反转位的顺序，这可以与不同的存储顺序一起使用，以使图像从左到右翻转，或将其旋转180度(以防有人将图像倒过来输入扫描仪)。将表扩展到16位，这将占用128Kb的内存，而不是8位表使用的256b，从而提供了更大的灵活性。使用16位时，您可以使用更大的跨度，并且可以选择16倍而不是8倍的比例因数，这可能会使您的视频显示尺寸更匹配。这些技术以及附录中给出的示例代码只是简单的构建块。检查您自己的应用程序的约束，以找到更多应用这些原则和提高代码性能的方法。

See also G1, 147; G1, 166; G2, 57; G2, 84.

## 1.4 一个简单的减色滤镜

Dennis Bragg  
Graphics Software Inc.  
Bullard, Texas

### 介绍

提出了一个简单的滤波器，将一个24位的彩色光栅图像减少到15位有效位，并消除了可见的颜色步进问题。所得到的图像可以直接显示在16位帧缓冲器上，或用作颜色量化方法的输入，以进一步减少图像中的颜色数量。

光栅图像通常存储为一个24位像素数组，8位分配给每个红、绿、蓝(RGB)组件。每个RGB组件包含256种可能的强度级别。图1(见彩色插入图)是一个24位图像，使用了2215种不同的颜色。注意彩色球平滑连续的阴影。

不幸的是，能够显示24位彩色图像的帧缓冲区并不总是可用的。使用8位像素作为256色彩色地图索引的彩色显示器被广泛使用。颜色量化方法(Gervautz和Purgathofer, 1990)常用于减少24位图像中使用的颜色数量，以便在8位设备上准确显示。

每像素可以显示16位颜色的帧缓冲器(每个RGB组件5位，加上一个属性位)也变得越来越便宜。在16位帧缓冲区上显示24位图像的典型解决方案是屏蔽每个RGB组件的三个最不重要的位。这种方法将每种颜色的256个强度级别减少到只有32个级别。

一个发生在减色平滑阴影图像的问题是颜色步进。在原始24位图像中，亮度从暗到亮连续变化的区域，在16位或8位帧缓冲区中显示时，通常会出现明显的亮度级步骤。在第2页(见彩色插入图)中，第1页的图像使用Gervautz和Purgathofer的颜色量化方法减少到256色。由于可供选择的颜色数量有限，请注意踏在球上的颜色。

该宝石通过一个加权随机量来改变每个像素RGB组件的强度级别，从而解决了颜色步进问题。方差量的加权方式这种方法的结果图像中任何像素局部区域的平均值非常接近源图像的实际24位颜色。

结果图像每像素包含15个有效的颜色位，每个RGB组件包含5个有效的颜色位。图像可以直接显示在16位帧缓冲区上，或用作颜色量化方法的输入，以进一步减少颜色的数量。得到的图像有一些“颗粒”的外观，但比可见的颜色梯度要

少得多。

## 滤波器

过滤器分别考虑每个像素的RGB组件。将一个分量的256个强度等级划分为32个相等的区域。每个区域覆盖8个强度等级。第一个区域的强度等级为0，下一个区域的强度等级为8，以此类推。

RGB组件的强度将被设置为这些区域之一。如果将组件设置为最接近的强度级别，得到的图像仍然会显示颜色步进。相反，强度除以8(或模量)的余数被确定。这给出了一个从0到7的数字。生成一个范围为0到8的随机数，并与余数进行比较。如果余数小于或等于随机数，则分量强度增加8。这具有以一种随机的方式改变组件的效果，但偏重于最接近的强度水平。

接下来，根据用户提供的噪声水平，将一些随机噪声添加到组件强度中。噪声的添加消除了任何残留的色彩踏步，否则可能是显而易见的。最后，组件较低的3位被屏蔽，将每像素的有效位数减少到15。

这个过程产生的RGB组件与原始的24位组件有很大的不同。然而，图像任意局部区域的像素分量的平均强度与原始图像的平均强度非常接近。在第三张图中(见彩色插入图)，首先对原始的24位图像进行滤波处理，然后采用与图2相同的方法将图像压缩到每像素8位。

## 实现

这个过滤器是用函数`rgbvary()`实现的。该函数需要四个参数:一个由待处理像素的RGB组件组成的三个字符数组(RGB)，一个指定所需噪声级别的整数(`noise_level`)，以及像素的 $x$ 和 $y$ 位置( $x$ 和 $y$ )。

该函数返回源RGB数组中修改后的RGB组件。噪音等级可以从0(无噪音)到8(吵闹!)2级的噪音在实践中效果很好。

像素的 $x$ 和 $y$ 位置由两个宏(`jitterx`和`jittery`)使用，它们生成随机数。抖动宏基于GRAPHICS GEMS中的抖动函数(Cychosz, 1990)。使用抖动的优点是它总是在特定的 $x, y$ 位置上以相同的幅度变化一个像素。当你在动画中减少几帧的色彩时，这是很重要的。使用标准的随机数生成器将在动画播放时产生“雪花”效果。`jitter`函数消除了这个问题。

在调用`rgbvary()`以初始化`jitter`宏使用的查找表之前，必须调用函数`jitter_init()`。这个过程使用标准的C函数`rand()`来填写表格。

总结

一种滤波器被提出，以减少24位图像为15有效位每像素。该程序消除了颜色步进的问题，但代价是外观略有颗粒。生成的图像可以直接显示在16位帧缓冲上，或用作颜色量化方法的输入，以进一步减少颜色。



1.4 Plate 1. Original 24-bit color image.



1.4 Plate 2. 256 color image after standard color quantization.



1.4 Plate 3. 256 color image after processing with `rgbvary()` and standard color quantization.

图 1.4.1: 颜色滤波

## 1.5 从抽样数据得到的紧凑等高线

Dennis Bragg  
Graphics Software Inc.  
Bullard, Texas

### 问题

包括医学成像、地震学和气象学在内的许多领域的的数据，都是在一个大立方体网格的顶点上采集的一组测量数据。在这些领域中，从数据立方体生成可视化表示的技术非常重要。许多常见的可视化技术将数据值视为连续函数 $F$ 的样本函数值，并对某些 $c$ 生成 $F(x, y, z) = c$ 的分段平面近似，即函数的等值线。最初的《图形瑰宝》之一，“从采样数据定义表面”，调查了几种最著名的从数据立方体生成等高线的技术(Hall, 1990)。

在本文中，我们将对该类型的所有技术进行增强。这种增强减少了任何等高线近似的元素数量，并改善了元素的形状。第一个改进通常将表示的大小减少约50%，允许更快的重新显示并减少内存需求。第二种方法通过避免在许多照明模型中造成不希望看到的阴影影响的狭窄元素，从而得到质量更好的图片。

### 基于立方体的轮廓

一些作者提出了大致类似的方法，可以从立方体数据网格中创建可视化的等高线。这些方法分别处理每个立方体上的数据，并利用沿立方体边缘的线性插值来计算位于等高线上的点集合。在Lorenson和Cline的Marching Cubes算法(Lorenson和Cline, 1987)中，这些交点通过表查找来连接成边和三角形，表查找的基础是定义立方体顶点的值 $F(x, y, z) - c$ 的符号。

不幸的是，这种方法并不能保证轮廓的连续性，因为共享一张带有混合符号的面的相邻立方体可以以不同的方式划分(Durst, 1988)。其他人提出了另一种方法，通过对模糊人脸中心的函数进行采样来消除这种情况的歧义(Wyvil等人, 1986)。我们称之为这样的方法，通过沿着三次网格边缘的线性插值计算出轮廓的顶点，基于边缘的插值方法。

基于边缘的插值方法的另一个问题是，它们产生的表面网格可能非常不规则，即使是简单的三元数据。这些不规则性包括微小的三角形(当轮廓通过立方

体网格的顶点时产生)和狭窄的三角形(当轮廓通过网格的边缘时产生)。根据我们的经验，在某些曲面网格中，这种三角形可以占到三角形的50%。这些形状糟糕的元素通常会降低渲染算法和有限元分析应用于网格的性能，而对近似的总体精度贡献很小。

紧凑立方体

这个章节的贡献是一种通用的技术，用于从基于边的插值中消除近简并三角形的问题。该技术背后的想法很简单:当网格的一个顶点靠近表面时，将网格“弯曲”一点，这样顶点就位于表面上。小三角形坍缩成点，小三角形坍缩成边，只剩下形状良好的大三角形。本文的其余部分概述了这一理念的实现;有更详细的解释(Moore和Warren, 1991)。

对数据立方体应用任何基于边的插值算法，并在此过程中，记录沿着立方体边缘生成的每个顶点，该顶点附近的立方体网格的点。我们称这个顶点为它最近的网格点的卫星。如果顶点位于一条边的中点，则可以使用这条边的任意端点，只要共享这条边的所有其他立方体使用同一个端点。当算法的这一阶段完成后，就得到了等高线的三角剖分S和一个离三角剖分的每个顶点最近的网格点。

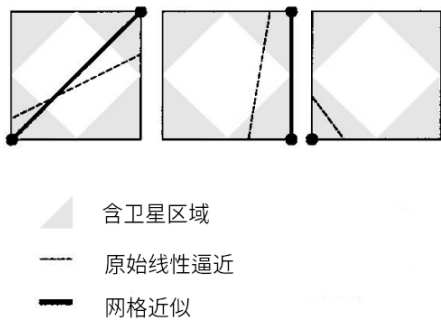


图 1.5.1: 二维案例表紧凑型立方体

为了得到一个新的、更小的等值线近似值，请采用以下步骤(算法5 ):

该方法的第一步是定义新网格连接点的拓扑结构。在S中，一个特定网格点的所有卫星被合并成结果网格中的单个顶点。因此，当一个网格点被“切掉”时产生的小三角形被折叠到网格点上。当两个顶点非常接近同一个网格点时产生的窄三角形被折叠成一条边。图1以两个维度说明了这一点。从这个角度来看，如果原曲面网格是连续的，那么算法第一步生成的网格也必须是连续的。

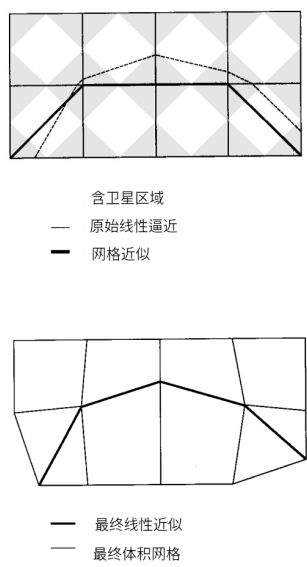


图 1.5.2: 一个二维立方体的例子

---

**算法 5:** 等值线近似值

---

```
for each triangle  $T$  in  $S$  do
    if the vertices of  $T$  are satellites of distinct gridpoints then
        | produce a triangle connecting the gridpoints;
    else
        |  $T$  collapses to a vertex or edge so ignore it;
    for each gridpoint  $g$  of the new triangulation do
        | displace  $g$  to the average position of its satellites;
```

---



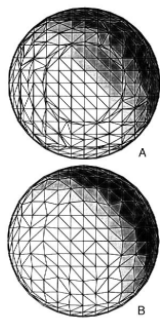


图 1.5.3: 球面的两个近似值

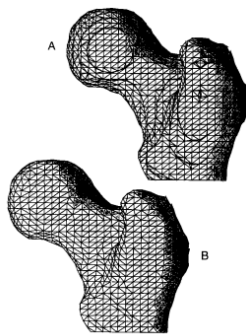


图 1.5.4: 两个股骨头的近似图

在第二步中，网格的顶点被平移到原始等高线上或附近。由于每个新的顶点位置被选择为位于原始轮廓上一小簇点的平均位置，新的近似通常只与原始轮廓稍微偏离。

图1.5.2 说明了这种方法应用于二维网格。上面的部分说明了第一步的结果。下面的部分说明了第二步的输出。图形上部的短边已折叠成下部的顶点。

实际上，这种方法效果很好，将三角形的数量减少了40%到60%。图1.5.3 显示了一个球体由游行立方体(a)和相同的球体后紧凑的立方体(B)的应用。图1.5.4显示了一个人类的股骨,最初作为CT数据,为波状外形的游行立方体(a)和紧凑的立方体(B)。在每个例子中,三角形的数量减少了使用紧凑的数据集,其余三角形的形状也得到了明显的改善。

正如这里所描述的，紧凑立方体产生的轮廓可能有几个不受欢迎的特征。首先，最终轮廓的边界可能不在定义的立体网格的边界上。第二，通过公共网格点附近的轮廓的两个不相交的薄片可以在该网格点融合。Moore和Warren(1991)描述了对Compact Cubes的简单修改，解决了这些问题。

See also G1, 552; G1, 558; G2, 202.

# 1.6 从位图生成等值线

Tim Feldman  
Island Graphics Corporation  
San Rafael, California

本节提出了一种算法，该算法遵循采样数据数组中的轮廓边缘。它使用Freeman链编码生成一个向量列表，描述轮廓的轮廓。

假设您有一个已采样或“数字化”成灰度像素矩形阵列的地形图。不同的像素值对应不同的地形高程。该算法可用于生成以等高线表示地形高程的“地形图”。附录中给出了一个遵循一条轮廓线的示例程序(*contour.c*)。

该算法能够处理包含单个样本点的等高线、围绕不同海拔区域的等高线、不形成闭合曲线的等高线、以及形成交叉曲线、形成环路的等高线。在所有情况下，它遵循轮廓的最外层边缘。给定阵列中高程轮廓的初始点，该算法找到轮廓的边缘。然后顺时针方向沿着这条边走，直到它回到起点。沿着每个方向向量描述了从路径上的一个像素到路径上的下一个像素的方向。路径上的所有像素都是近邻。因此，向量可以被认为是传统二维向量的方向部分，其长度部分总是等于一个像素。这样的向量列表被称为“弗里曼链”，以其创始人Herbert Freeman命名(Freeman, 1961)。图1.6.1 显示了定义从一个像素到其邻居路径上的八个可能方向的值。本例中使用的像素阵列如图1.6.2a所示;图1.6.2b显示了示例程序的输出。算法从样本 $x = 3, y = 2$ 开始，寻找高程 = 200的等高线边缘并跟踪。

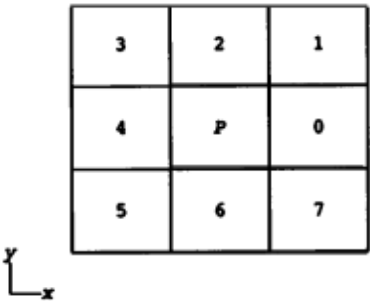


图 1.6.1: 从点P到它的八个邻居的方向向量。

该算法的核心在于知道如何以顺时针方向沿着轮廓的方式选择邻居。测试

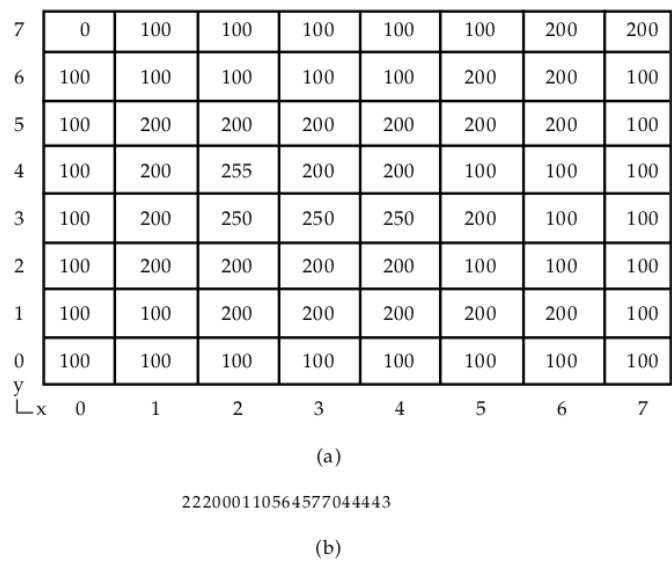


图 1.6.2: a)标高数据采样样例。  
b)绘制高程 = 200等高线轮廓的22个点向量。

图1.6.2a，并想象您从 $x = 1, y = 2$ 的采样点出发，沿着高程= 200的等高线边缘行走。为了顺时针移动轮廓线，你的第一个移动应该是 $x = 1, y = 3$ 处的样本。你走在自己轮廓的边缘;在你的左边是危险的悬崖，它向下坠落到一个较低的高度。当你沿着轮廓走的时候，注意你的头部是变化的，但是悬崖总是在你的左边。在选择下一步的时候，你总是试着向前走，向左走，而不是走下悬崖。如果你不能向前移动到你的左边，你试着顺时针方向:直走。如果不行，你试着顺时针方向向前，向右，等等。如果你发现自己在一个海角上，你顺时针方向的方向会让你掉头，重新走一段路。最终，你将完全沿着轮廓移动，然后回到你的起点。

算法的工作原理是一样的。*build()*过程构建围绕轮廓边缘移动时所采取的方向的Freeman链。*build()*调用*neighbor()*过程来获取路径上的下一个邻居。*neighbor()*反过来调用*probe()*来查找该邻居。最低级的过程是*in\_cont()*，它只是简单地测试给定的样本是否在轮廓中。请注意，采样数据的整个数组不需要立即放入内存;*in\_cont()*可以修改为随机访问脱机存储，如果需要的话。

*neighbor()*中的*last\_dir*变量维护*neighbor()*的方向感。检查图3，看看邻居()过程是如何实现上面描述的“尝试向前移动并向左移动”的步骤的。假设你从样本a到达样本P，那么*last\_dir*是2，而样本C总是在等高线之外，所以第

一个探测的邻居是D。从P到D的方向是3; $new\_dir \leftarrow last\_dir + 1$ 。

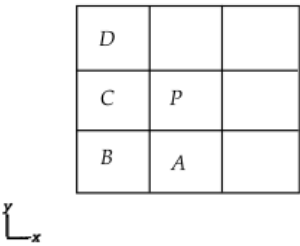


图 1.6.3: 从P点通过A或B点移动到D点

现在假设您从b到达了P,  $last\_dir$ 是1,C仍然在等高线之外, D仍然是要探测的第一个邻居。P到D的方向仍然是3; $new\_dir \leftarrow last\_dir + 2$ 。

请注意, 当 $last\_dir$ 等于0、2、4或6时, 到达P点的情况都是全等的(它们只是被旋转了90度)。类似地,  $last\_dir = 1、3、5$ 和7的情况都相等。因此,  $neighbor()$ 使用的简单规则是将 $new\_dir$ 设置为 $last\_dir + 1$ (如果 $last\_dir$ 为偶数), 或将其设置为+ 2(如果 $last\_dir$ 为奇数)。当然,  $new\_dir$ 的范围必须保持在0到7之间, 因此加法对8取模。

E的取值范围从0到7, 所以加法对8取模。唯一剩下的微妙之处是如何正确地选择第一个移动, 以便以顺时针方向围绕轮廓开始。这是很容易实现的:算法, 当给定一个起点在轮廓的任何地方, 在轮廓向左移动, 直到它遇到边缘。这保证了路径从边缘开始。它还保证了初始排列如图3所示:路径从样本P开始, C处的邻居不在等高线中, 并且 $new\_dir$ 的值应为3。这意味着 $last\_dir$ 的初始值应该是1。算法在 $build()$ 过程中设置它。

这个示例程序是为了演示Freeman链背后的思想而编写的, 但它的存储效率并不高。链表中的每个成员占用一个整数和一个指针。但正如Freeman在他最初的工作中指出的, 每个方向向量编码只需要三个比特。使用三位的实现将使用一个大的内存块而不是链表, 并且将有将方向向量链打包到块中的过程, 并按顺序提取它们。为了确定应该为保存轮廓中所有方向向量的块分配多少内存, 将使用轮廓跟踪算法的简化版本。它会沿着轮廓线计算描述完整路径所需的方向向量的数量, 但它不会存储方向向量。一旦确定了矢量的数量, 就会分配内存, 调用主算法重描轮廓, 并将方向矢量打包到内存块中。

前面的方法比示例程序更有效, 但它用速度换取了内存空间。第三种方法仍

然允许轮廓具有任意长度，同时有效地使用内存空间，同时保持良好的速度。它通过消除预扫描步骤来实现这一点。对于使用大数据集或不保存在内存中的数据集的实现来说，这是一个重要的考虑因素。方法是使用一个简单的链表，就像示例程序一样。但是，列表中的每个成员都有一个分配的内存块，而不是一个整数。这个块包含许多方向向量，每一个都被压缩成3个比特。更多的数据块将被分配，并根据需要链接到列表中，因为轮廓是遵循的。包装和提取带菌者将需要程序，而且必须保持额外的管理资料，以便使一切都处于控制之下。这种技术为链表中的指针使用了少量空间，但仍然比示例程序的内存效率高得多。这种方法的权衡是在实际编程中经常遇到的问题：为了在保持速度的同时节省存储空间，会增加程序的复杂性。

最后，一些实现可能根本不需要在内存中保存轮廓的表示；它们可以简单地将方向向量写入顺序访问磁盘文件或某些输出设备或并发进程。在这种情况下，示例程序的`build()`过程将被修改。

See also G3, A.5.

## 1.7 合成黑白位图

David Salesin  
Cornell University  
Ithaca, New York

and

Ronen Barzel  
California Institute of Technology  
Pasadena, California

### 介绍

典型的位图编码黑白像素。添加一个辅助位图允许我们表示透明的像素。这种两位表示对于非矩形或有孔的黑白图像很有用。它还为组合位图提供了一组更丰富的操作。我们用布尔值对 $(\alpha, \beta)$ 编码三个可能的像素值，如下所示：

$\alpha$	$\beta$	Meaning
1	0	Black
1	1	vvnite
0	0	Transparent
0	1	Undefined

### Compositing Bitmaps

我们可以使用合成操作 $R \leftarrow P \text{ op } Q$  将两个像素 $P = (P_\alpha, P_\beta)$  和 $Q = (Q_\alpha, Q_\beta)$  组合成一个新的像素 $R = (R_\alpha, R_\beta)$ ，如表一所示。

该表是将全彩合成代数(Porter and Duff, 1984)简化为两个位域(Salesin and Barzel, 1986)。注意，表中 $R_\alpha$ 和 $R_\beta$ 的方程现在是布尔公式:and被写成乘法，OR写成加法，XOR写成 $\oplus$ 。布尔操作可以使用一系列标准的“bitblt”操作一次对整个位图执行。根据操作的不同，所需的bitblt总数从2到4不等。

表中的数字描述了操作对两个示例位图P和q的影响。在这些数字中，使用灰色调表示结果中的透明区域。

over运算符用于将非矩形黑白位图置于现有图像之上——它非常适合绘制游标。in和out操作符允许一个位图作为另一个位图的哑光——例如，如果P是一个“砖”纹理，而Q是一个“建筑”，那么P in Q就会用砖对建筑进行贴片。如果一个位图同时充当另一个位图的哑光和背景，那么atop操作符很有用——例如，它允许将一小块纹理增量地绘制到现有位图上。

See also G1, 210.

Operation		Figure	$R_\alpha$	$R_\beta$ Description
clear		0	0	Result is transparent
P		$P_\alpha$	$P_\beta$	P only
Q		$Q_\alpha$	$Q_\beta$	Q only
P over Q		$P_\alpha + Q_\alpha$	$P_\beta + \overline{P}_\alpha Q_\beta$	P occludes background Q
P in Q		$P_\alpha Q_\alpha$	$Q_\alpha P_\beta$	$Q_\alpha$ acts as a matte for P; P shows only where Q is opaque
P out Q		$P_\alpha \overline{Q}_\alpha$	$\overline{Q}_\alpha P_\beta$	$\overline{Q}_\alpha$ acts as a matte for P; P shows only where Q is transparent
P atop Q		$Q_\alpha$	$Q_\alpha P_\beta + \overline{P}_\alpha Q_\beta$	$(P \text{ in } Q) \cup (Q \text{ out } P)$ ; Q is both background and matte for P
P xor Q		$P_\alpha \oplus Q_\alpha$	$\overline{Q}_\alpha P_\beta + \overline{P}_\alpha Q_\beta$	$(P \text{ out } Q) \cup (Q \text{ out } P)$ ; P and Q mutually exclude each other

表 1.7.1: 位图合成操作

1.8 计算机动画的 $2\frac{1}{2}D$ 景深模拟

Cary Scofield

Hewlett-Packard Company

Chelmsford, Massachusetts

介绍

景深被定义为在光学透镜系统中包围焦平面的区域，在该区域内物体保持确定的焦质量。长期以来，摄影师和电影摄影师一直利用相机镜头和光圈的这一方面，将观众的注意力从感兴趣的领域以外的区域引导到图像的特定部分。正因为如此，在计算机动画系统中加入景深效果是非常有利的。

这个gem描述了一个 $2\frac{1}{2}$ -D的景深算法，用于模拟计算机生成的动画中的焦点变化。这种特殊的算法实际上独立于任何隐藏面去除技术。我们根据深度将3d场景分层为独立渲染的不相交对象组。生成的图像经过滤波模拟景深，然后通过合成后处理重新组合成单幅图像。当逐渐改变滤镜应用于动画序列的连续帧时，其效果是将观看者的注意力从场景的一个深度面拉到另一个深度面。



## 相关工作

以前模拟景深的尝试涉及使用针孔相机模型(Potmesil和Chakravarty, 1982)。然而, 该算法没有考虑到透镜表面提供了连续的环境不同视图这一事实。分布射线追踪(Cook et al., 1984)克服了这一缺陷, 但将该技术嵌入到渲染过程中。我们的算法可以被认为是这两种方法的折中: 虽然我们不整合阴影和景深, 但我们确实以一种受限的方式结合了表面可见性和景深。

## 算法

我们的方法本质上是一个三个阶段的过程:

### 1. 隐藏表面去除阶段:

在第一个阶段中, 可以手动或自动地将对象分层或聚类到前景和背景组中。每个簇或组被单独渲染成它自己的图像, 每个像素的alpha通道有一个不透明度掩码(见图1, 颜色插入)。这种不透明度遮罩最终在第三阶段发挥作用, 但它可以在第二阶段进行修改。

### 2. 滤波器后处理阶段:

该阶段使用类似于指数低通滤波器的卷积掩模来模糊各种图像, 以模拟景深效果。由于我们的算法与模拟透镜的焦距无关, 这允许我们自由地操纵图像的模糊程度。然而, 我们的目的是创造一个现实的效果, 所以我们仔细选择模糊程度。还必须小心避免“渐晕”(Perlin, 1985), 如果过滤算法不补偿图像边界之外的未知区域, 就会出现这种现象。这是通过重新计算加权滤波器, 每当卷积掩模的任何部分被图像边界剪辑。

### 3. 合成阶段:

最后, 在这一阶段, 我们遵循波特和达夫(1984)、达夫(1985)和麦克斯和勒纳(1985)建立的算法。第一阶段中不透明度蒙版的重要性在这里发挥了作用, 因为它允许我们在将前景图像叠加到背景图像时避免混淆工件。

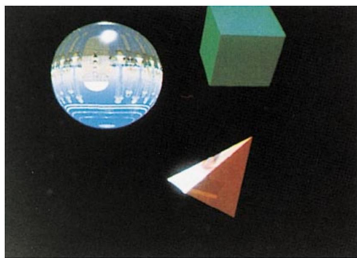
## 焦距变化模拟

正如在介绍中所述, 电影摄影师长期以来一直使用焦点的变化来推动或拉观众的注意力从场景的一个部分到另一个部分(例如, 从前景到背景)。这种“聚焦

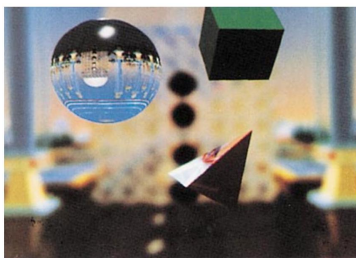
”是未经过滤的前景对象(“聚焦中”)与模糊的背景(“超出”焦平面)叠加的结果。因此,将观众的注意力从前景拉到背景,相当于在一系列帧中,将镜头的“焦平面”沿着光轴从前景逐渐转移到背景。因为我们不使用镜头和光圈相机模型,我们必须从一帧到一帧修改模糊滤镜的形状。假设发生焦点变化模拟的帧数是先验的,我们可以很容易地对滤波器的“模糊度”进行插值,范围从一个很小的值到一个能给我们期望的最大图像模糊度的大小。这是为前景对象做的。对于相应的背景图像,我们使用相同的插值值系列,只是顺序相反。当过滤过程完成后,两个独立的前景和背景图像流在最后一个阶段被拼接在一起,形成拍摄的最终图像帧。图b、c和d(见颜色插入)分别是由这个过程产生的动画序列的第一帧、中间帧和最后帧的例子。作为一个边注,这一过程所获得的结果与塞尔动画中偶尔使用的多平面相机系统所获得的结果非常相似(Thomas和Johnston, 1981)。

## 致谢和历史记录

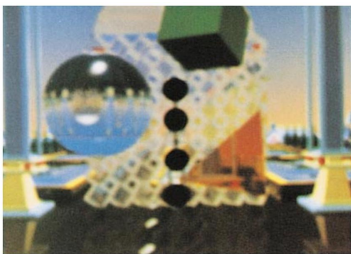
这本书是几年前与詹姆斯·阿尔沃(James Alvo)合作撰写的一篇更详细但从未发表过的论文的浓缩。作为一个历史笔记,在这个宝石中描述的焦点变化模拟被用于两个阿波罗计算机射线追踪动画公司中的第一个(即“Quest”)。



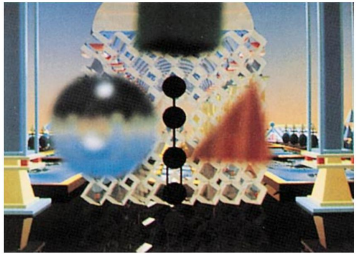
(a) 由它们自己渲染的场景中的一簇物体。



(b) 前景和背景物体的合成图像。前景是“聚焦”。



(c) 合成图像。前景和背景对象模糊化。



(d) 合成图像。动画序列的最后一帧，背景对象“聚焦”。

# 1.9 复合区域的快速边界生成器

Eric Furman

General Dynamics, Electronics Division  
San Diego, California

## 问题

在计算机图形学的许多应用中，寻找由封闭边界定义的多区域的轮廓是一个常见的问题。通过求一组圆的包络线可以确定一组雷达站点的二维覆盖，如图1所示。另一个例子是土地利用和城市规划中若干分区的大纲。一般来说，这类应用程序都是将兴趣区域的联合可视化。需要一种算法来找到这些区域的复合边界或包络线。换句话说，我们希望显示一组二维区域的轮廓。每个区域在二维上都是一个封闭边界。常见的区域原语是圆、多边形和椭圆，但任何其他封闭区域轮廓都可以使用所描述的技术很好地工作。在这个gem中，我们将使用圆作为我们的基本示例区域。图1到图3显示了该算法的步骤。圆集合的轮廓有许多短的连接弧或扇形。

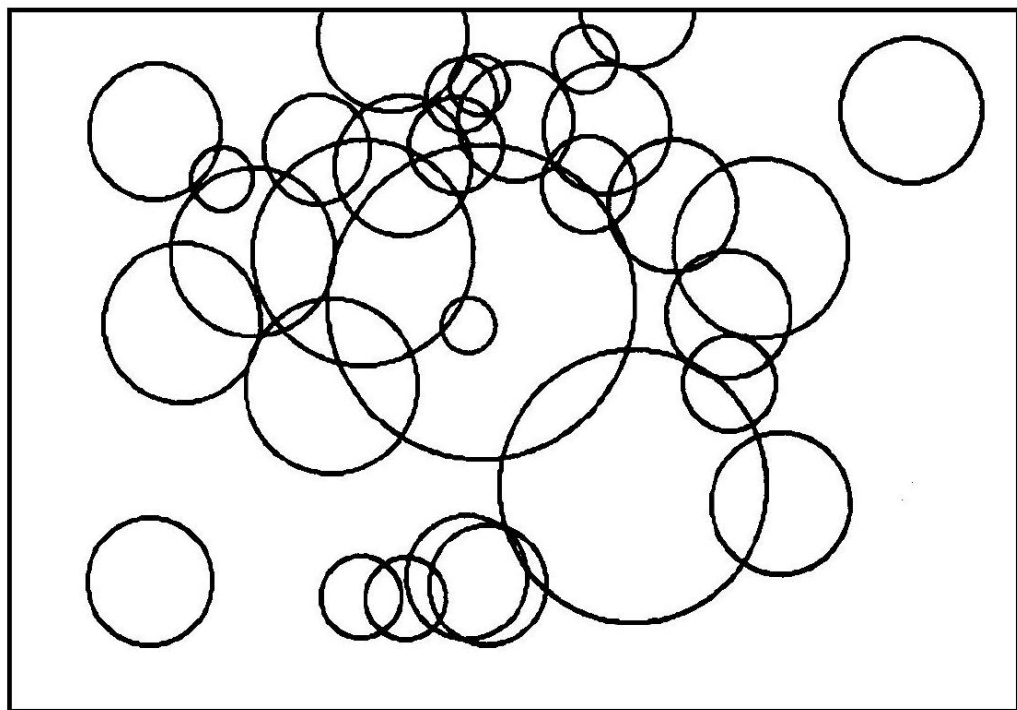


图 1.9.1: A set of circles shown in a frame buffer.

### 其他方法

针对多雷达站点问题的几种解决方案都采用了直接分析的方法。他们通过一系列雷达范围圆进行工作，通过将每个新圆与以前的交叉点生成的弧相交，从而创建一组弧(Bowyer and Woodwark, 1983)。不相交的圆必须带着，并与每个新圆相交。对每个新圆进行内部/外部测试，并修改弧集以删除一些弧段并添加新的弧。不幸的是，随着圆列表的增长，生成边界的时间随着圆数量的平方而增加。已经实现了一些改进，例如完全丢弃包含在其他圆中的圆，并使用边界框测试来限制更昂贵的圆/弧相交测试的数量。

Figure 2. The circles of Fig. 1 after filling.

这种直接的分析方法适合于创建分辨率远高于典型帧缓冲区所能显示的区域边界数据集。然而，可视化一组区域的包络可以更快地完成。

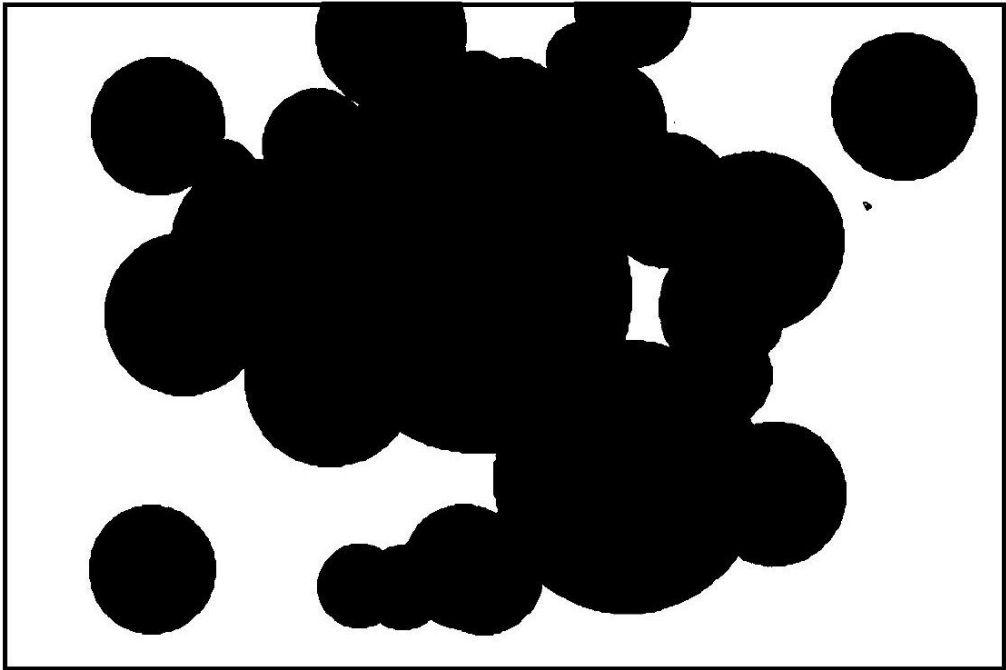


图 1.9.2: The circles of Fig. 1 after filling.

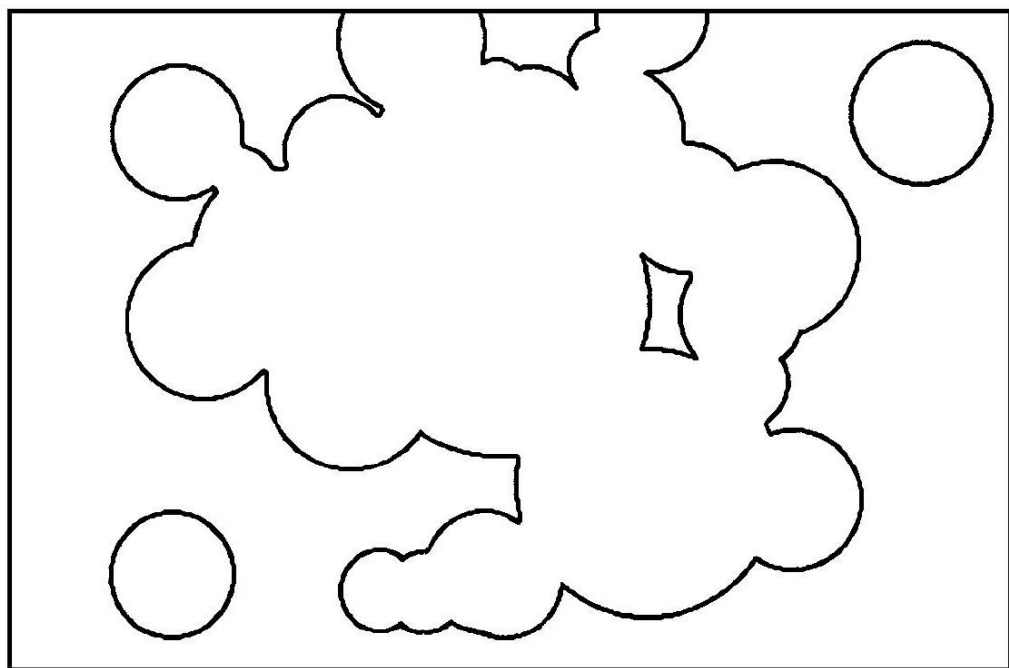


图 1.9.3: Scallops. The envelope of Fig. 1 circles.

### 快速生成边界

一种混合了计算机图形学和图像处理技术的快速生成区域边界的方法。该算法由两个基本步骤组成，用于帧缓冲区的有限分辨率。首先，对于每个封闭原语，绘制填充区域并将其剪切到帧缓冲区限制。尽管我们希望只显示许多闭合对象的信封，但我们同时绘制和填充每个对象。通常绘图和填充可以通过精心的算法构造在同一个步骤中完成。其次，在整个帧缓冲区上应用侵蚀运算符来删除所有填充的内部点。没有填充的对象，侵蚀不会使它们减少到复合边界。这个过程如图1-3所示，每个圆都是独立绘制和填充的。

该算法的执行时间是所有区域的绘图和填充时间加上单道侵蚀操作的时间。在我们的示例中，这是绘制/填充所有圆所需的时间加上侵蚀它们所需的时间。当绘制单个重叠对象时，不填充先前由另一个区域填充的像素似乎可以节省一些时间。不幸的是，测试确定一个区域是否已经被填满通常比仅仅将数据写入帧缓冲区要花费更长的时间。所需的时间与所有区域的填充面积成线性增长，对于直接分析方法的平方增长率有相当大的改进。

附录II中用于生成填充圆的示例C代码改变了计算机图形文本中给出的通常的中点圆生成算法，以创建填充圆(Foley et al., 1990)。通过调用`raster_fill`函数来填充圆，该函数直接实现了使用`fill_circle`函数中的二阶偏差生成中点圆的算法。

将填充区域侵蚀到它们的边界轮廓，查看帧缓冲区中每个像素及其最近的四个连接邻居。一个填充的像素值或颜色将被擦除为侵蚀值，只有当它的所有四个连接的邻居像素也被填充相同的值。对背景值的侵蚀将只留下以原始填充颜色绘制的边界像素。然而，侵蚀到不同的颜色将留下轮廓在原始填充值和填充内部用新的侵蚀值。在这个测试过程中使用了三个光栅缓冲区，以避免在以后计算其他像素时替换帧缓冲区中的像素，或者需要从一个帧缓冲区渲染到另一个帧缓冲区。在附录II的C代码中，栅格缓冲区向帧缓冲区添加了一个单像素的边界。用对象的原始像素填充值填充这个边界将留下一个开放的边缘，在这个边界被剪切在帧缓冲区的边缘。当边界设置为背景值(在代码中为零)时，在发生帧边缘裁剪的地方绘制一个封闭边缘包络线。这个过程是图像处理的二值边缘检测(Gonzalez和Wintz, 1987)在计算机图形学中的一个相当简单的应用。

### 注意事项

尽管这种技术简单快速，像大多数帧缓冲算法一样，它只能精确到最近的像素。解析解的精度将受到计算所用的算术精度的限制。

对于圆，可以通过生成一个固定圆的栅格跨度宽度表来提高速度。直径应该至少是帧缓冲区最大尺寸宽度的两倍，以保持在Nyquist区间内。然后，该表可用于任何所需圆大小的比例宽度查找。表查找消除了进一步的圆生成，但仍然必须执行剪切和填充操作。对于任何直接可伸缩和未旋转的区域，也可以获得类似的表好处。

为了帮助用户验证函数的正确实现，C代码包含一个简单的程序来测试填充和侵蚀例程。一个很小的伪帧缓冲区使用了简单的`"printf"`语句在ASCII中显示的左上角。





## 第 2 章 | 数值和编程技术

计算机图形学领域充满了复杂的数学，图形程序通常充满了计算密集型的操作。简化计算的技术和技巧或有用的近似总是受欢迎的。本节包含的Gems为那些喜欢“关注细节”的程序员增加了技巧。

第一个Gem描述了IEEE标准平方根运算的快速近似，并改进了前一个Gem中提出的技术。第二个Gem描述了围绕众所周知的UNIX(tm)内存分配器“malloc()”放置的包装器，以提高其可用性和可预测性。第三个Gem解释了如何考虑由轨道球控制的3-D旋转，并提供了背后的群论数学。第四个Gem简要介绍了区间算法以及如何在计算机图形学中使用它。

第五Gem讨论了与常用的两个、三个或更多数字的循环排列技术相关的效率问题。第六Gem讨论了如何选择颜色来突出显示或选择图像特征，并提出了一个类比魔方的空间!第7个Gem处理的是生成具有各种分布的随机点集，均匀的和其他的。这些技术对于分布射线追踪和其他蒙特卡罗方法非常有用。最后两个Gems采用了二维和三维空间中经常使用的一些概念，并将它们扩展到高维空间中。

## 2.1 IEEE 快速平方根

Steve Hill

University of Kent

Canterbury, Kent, United Kingdom

这个gem是Paul Lalonde和Robert Dawson在Graphics Gems i中提出的快速平方根算法的重新实现。

在我的实现中，我添加了一个额外的例程，它允许将平方根表转储为C源代码。该文件可以单独编译，以消除在运行时创建表的必要性。

新的例程使用IEEE双精度浮点格式。我包含了许多有用的`#defines`，以使程序更易于访问。注意，在某些体系结构中，单词的顺序是颠倒的。常数`MOST_SIG_OFFSET`可以设置为1或0，以允许这一事实。

表的大小可以通过改变常量`SQRT_TAB`大小来调整。它一定是4的幂。恒定的`MANT_SHIFTS`必须相应地进行调整——如果将表的大小增加四倍，那么从`3MANT_SHIFTS`中减去2。

See also G1, 403; G1, 424; G2, 387.

## 2.2 一个简单的快速内存分配器

Steve Hill

University of Kent

Canterbury, Kent, United Kingdom

这个Gem描述了一个简单的内存分配包，可以用来代替传统的`malloc()`库函数。该包维护一个内存块链表，以顺序的方式从其中分配内存。如果一个块用完，则从下一个块分配内存。如果下一个块是`NULL`，则使用`malloc()`分配一个新块。

我们把内存块的列表称为池。一个池可以被全部释放，也可以被重置。在前一种情况下，使用库函数`free()`将分配给池的所有内存返回给系统。在后一种情况下，不会释放任何内存，但会重置池的高水位标记。这允许在一个操作中丢弃池中分配的所有数据，几乎没有任何开销。然后池中的内存就可以重用了，不需要重新分配。

这个包允许程序员创建多个池，并在它们之间切换。

该方案的一些优点是：

- 内存分配很快。
- 数据可能具有更大的局部性。
- 我们不再需要每个数据结构都有一个免费的例程。
- 重置池非常简单。这可能会取代对`free()`库例程的许多调用。
- 内存泄漏的可能性较小。

主要的缺点是：

- 单个结构不能被释放。这可能会导致更大的项目驻留。

该软件包已成功用于射线追踪程序。使用了两个池。第一个池保存在读取模型文件时创建的永久数据。第二个池用于在呈现过程中创建的临时数据。在计算完每个像素后重置该池。

该一揽子方案的合并产生了三个重大影响。首先，程序运行得更快。虽然速度不是特别快，但是程序的大部分时间都花在计算十字路口，而不是分配内存上。其次，许多操作的代码变得更简单。这是因为消除了对释放内存的调用。最后，所有的空间泄漏都被根除了。这个程序由许多人共同开发，在某些情况下，对适当的内存分配函数的调用被忘记了。使用包消除了对这些调用的需要；因此，空间泄漏也被消除了。

## 2.3 滚动球

Andrew J. Hanson

Indiana University

Bloomington, Indiana

交互式图形系统通常需要允许用户使用常用的二维输入设备(如鼠标)在三维空间中自由旋转图形对象的技术。实现这一目标受到一个事实的阻碍,即从输入设备的两个参数到定向的三个参数空间没有单一的自然映射。

在这里,我们介绍了鼠标驱动三维方向控制的滚动球方法,以及它在其他科学可视化问题上的一些有趣的扩展。这种技术利用连续的二维运动(以球在平桌上滚动而不滑动为模型)来达到任意的三维方向。与其他各种方法不同,滚动球方法只有一个状态,并且完全与上下文无关:可以关闭鼠标光标,忽略移动的历史或演进状态,但仍然确切地知道下一个增量鼠标移动将会产生什么效果。对于从直接操作印象中获益的应用程序来说,这个属性非常有吸引力。

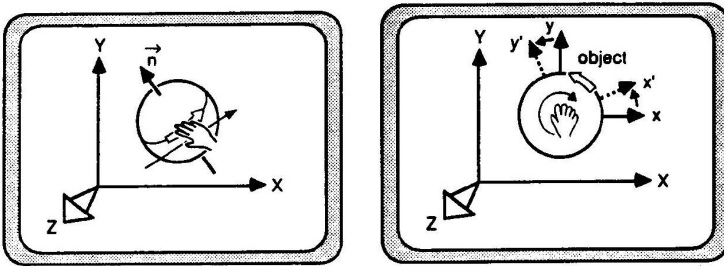
很明显,鼠标可以控制围绕两个轴(图1中的 $x$ 和 $y$ 方向)的旋转。令人惊讶的是,滚动球还自然地包含了诱导屏幕垂直方向(图1中的 $z$ 轴)的顺时针和逆时针旋转的能力。根据空间旋转的群体理论的一个基本但违反直觉的性质,在小的顺时针圆周上移动滚动球控制器必然会产生小的逆时针旋转,反之亦然。这就解释了为什么一个明显不可能的三度转动自由度确实可以通过一个上下文无关的二自由度输入设备产生。

下面给出的滚动球算法的数学形式实际上是Chen等人(1988)在广泛调查方向控制方法中所研究的算法的一部分;我们的方法以Chen等人(1988)没有处理过的方式利用和扩展了算法的属性。滚动球应该被理解为一种新颖的、上下文无关的方法,它利用了通常以上下文相关的方式使用的已知旋转算法。

下面的处理包括两个主要部分。第一部分介绍了如何使用滚动球法进行三维定位控制,以及如何在交互式图形系统中实现它。第二部分描述了如何将滚动球方法扩展到科学可视化中极为重要的其他转换组;滚动球法被看作是一个迷人的工具,在它自己的权利可视化变换组的性质。

### 使用方法

为了理解这个方法的基本原理,假设一个球放在桌子上,在你手掌的水平下



(a) 由它们自己渲染的场景中的一簇物体。 (b) 前景和背景物体的合成图像。前景是“聚焦”。

图 2.3.1: 滚动球算法中使用的两种基本技术用于实现图形对象的任意空间旋转。a)用鼠标按黑色箭头指示的方向移动手，使物体绕位于屏幕平面上的轴 $\vec{n}$ 旋转，即使赤道沿空心箭头方向旋转。b)小圈移动手，使物体绕屏幕平面法线旋转，方向与手的运动方向相反，再次旋转赤道沿空心箭头方向。

方。

球绕任何与桌面平行的轴旋转时，都是通过手在垂直于轴的方向上水平移动，从而使球绕该轴旋转。注意，这个类的任何单一运动都不会产生围绕垂直轴的旋转，垂直于手掌的轴。

然而，如果你把你的手平放在一个球的顶部，并在水平的小圆圈中移动你的手，球实际上会绕着垂直轴向相反的方向旋转。

控制空间方向的滚动球算法是通过将要旋转的图形对象的方向作为球本身的方向来实现的，同时使用鼠标(或类似的二维输入设备)来模拟手掌的动作。

通过执行鼠标(或手)的指示动作，可以使用滚动球算法在显示的图形对象上实现以下效果:

- 绕水平屏幕线或 $x$ -轴旋转，是通过相对于查看器向前或向后移动鼠标来实现的。
- 通过向左或向右移动鼠标，可以旋转垂直屏幕线或 $y$ 轴。
- 绕屏幕平面上的对角线旋转，我们用向量 $\vec{n}$ 表示其方向，通过垂直移动鼠标到 $\vec{n}$ 来实现，就像手掌在绕轴 $\vec{n}$ 旋转圆柱或球一样。
- 小的顺时针旋转垂直于屏幕，或 $z$ 轴，通过移动鼠标在小的，逆时针的圆圈。更明显的旋转是通过使用更大的圆周运动来实现的。

- 小的逆时针旋转，垂直于屏幕是通过移动鼠标在小的，顺时针的圆圈。
- 围绕屏幕垂直方向的大旋转是通过向任何方向旋转对象 $90^\circ$ 来实现的，围绕原始屏幕垂直轴旋转所需的量(现在位于屏幕平面上)，然后旋转 $90^\circ$ 来恢复原始屏幕垂直轴的方向。这个动作本质上是一个大的长方形运动，与绕屏幕垂直旋转的小圆形运动形成对比。

图1总结了两个最基本的动作，绕屏幕平面上的轴 $\vec{n}$ 旋转和绕屏幕垂直轴旋转。

输入设备光标的位置与滚动球算法无关，在旋转操作期间，它通常对用户是不可见的。计算只需要先前和当前设备位置之间的差异，因此通常需要在每次移动后将鼠标弯曲到屏幕中央，以防止它离开交互窗口。因此，该方法是真正的上下文无关的，非常适合强调直接操作的用户界面。

## 实现

滚动球算法是通过取一个给定的增量输入设备运动来定义一个在右屏幕坐标系中包含组件 $(dx, dy)$ 的向量来实现的。右旋轴 $\vec{n}$ 被定义为位于屏幕平面上的垂直于输入设备运动的以下单位向量：

$$n_x = \frac{-dy}{dr}, \quad n_y = \frac{+dx}{dr}, \quad n_z = 0 \quad (2.3.1)$$

这里我们定义了输入设备的位移 $dr = (dx^2 + dy^2)^{1/2}$ 。

其次，引入了该算法的单一自由参数——有效滚动半径 $R$ ，它决定了旋转角度对位移 $dR$ 的灵敏度；如果 $dr$ 只有几个像素，那么大约100的 $r$ 值是合适的。我们选择旋转角度为 $\theta = \arctan(dr/R) \approx (dr/R)$ ，使

$$\begin{aligned} \cos \theta &= \frac{R}{(R^2 + dr^2)^{1/2}} \\ \sin \theta &= \frac{dr}{(R^2 + dr^2)^{1/2}} \end{aligned} \quad (2.3.2)$$

绕轴 $\vec{n}$ 旋转一个角度 $\theta$ 的矩阵的一般形式是，其中 $\vec{n} \cdot \vec{n} = 1$ ，(参见M. Pique在Graphics Gems I, p. 446 [Glassner, 1990]中的“矩阵技术”)：

$$\begin{vmatrix} \cos \theta + (n_x)^2 (1 - \cos \theta) & n_y n_x (1 - \cos \theta) - n_z \sin \theta & n_x n_z (1 - \cos \theta) + n_y \sin \theta \\ n_y n_x (1 - \cos \theta) + n_z \sin \theta & \cos \theta + (n_y)^2 (1 - \cos \theta) & n_y n_z (1 - \cos \theta) - n_x \sin \theta \\ n_z n_x (1 - \cos \theta) - n_y \sin \theta & n_z n_y (1 - \cos \theta) + n_x \sin \theta & \cos \theta + (n_z)^2 (1 - \cos \theta) \end{vmatrix} \quad (2.3.3)$$

将Eq.(2.3.1)中 $\vec{n}$ 的值代入Eq.(2.3.3)，得到滚动球旋转矩阵

$$\begin{vmatrix} \cos \theta + (dy/dr)^2 (1 - \cos \theta) & -(dx/dr)(dy/dr)(1 - \cos \theta) & +(dx/dr) \sin \theta \\ -(dx/dr)(dy/dr)(1 - \cos \theta) & \cos \theta + (dy/dr)^2 (1 - \cos \theta) & +(dy/dr) \sin \theta \\ -(dx/dr) \sin \theta & -(dy/dr) \sin \theta & \cos \theta \end{vmatrix} \quad (2.3.4)$$

其中三角函数的值由式(2.3.2)给出。我们观察到:

- 在应用式(2.3.4)之前，必须将所有向量转换到所需的旋转中心。
- 旋转必须在一个单独的步骤中执行，如Eq.(2.3.4)。执行旋转作为一个序列，例如，首先绕 $x$ -轴，然后绕 $y$ 轴，将给出一个完全不同的结果(尽管，由于微妙的原因，差异可能几乎是不可观察的)。
- 更改 $\vec{n}$ 的整体符号将产生围绕对象的视点旋转，而不是视图内的对象旋转。小的顺时针的手运动将产生小的顺时针旋转的视点，但在视图中心的对象将继续逆时针旋转。这一现象源于群理论对身体固定旋转和空间固定旋转描述的符号差异(Whittaker, 1944)。

### 滚球法的推广

当我们分析滚动球法的群论背景时，各种相关的应用立即浮现出来。在这里，我们总结了普通旋转所涉及的基本群论，以及几个易于实现的扩展。这些技术对于许多科学可视化应用程序都很有用，包括建立关于一般群体的直觉。如果读者对群论没有兴趣，只是想知道如何实现和使用算法，就不必再读下去了。

## 无穷小旋转的群论

涉及滚动球行为的基本群论(Edmonds, 1957)可以总结如下:如果我们定义 $L_i, i = \{x, y, z\}$ 为旋转群 $O(3)$ 的无穷小产生子, 具有正旋转的右手约定, 那么我们就有了对易关系

$$[L_y, L_z] = -L_x, \quad (2.3.5)$$

$$[L_z, L_x] = -L_y, \quad (2.3.6)$$

$$[L_x, L_y] = -L_z, \quad (2.3.7)$$

其中我们使用了定义 $[A, B] = AB - BA$ 。这些无穷小生成器可以表示为矩阵, 也可以表示为这种形式的微分算子

$$L_x = y \frac{\partial}{\partial z} - z \frac{\partial}{\partial y}$$

和它的循环排列。方程式中的负号。(5-7)不是任意的, 而是由我们的惯例决定的,  $L_i$ 使用右手规则绕第 $i$ 轴旋转一个向量。这个负号直接导致了观测到的反旋转, 是旋转群性质的必然结果。

## 四元数旋转, $2 \times 2$ 矩阵, 和 $SU(2)$ 旋量

滚动球变换用于定义四元数旋转(例如, 参见Shoemake, 1985, 或p - g的“使用四元数”)。Maillot在Graphics Gems I, 第498页[Glassner, 1990])甚至比普通的空间旋转更自然。这是因为四元数公式等价于组 $SU(2)$ 的更标准的 $2 \times 2$ 矩阵表示法,  $SU(2)$ 是通常旋转组 $O(3)$ 的双重覆盖(Edmonds, 1957)。(尽管这两个基团对应完全不同的拓扑空间, 但它们被滚动的球利用的无穷小性质是相同的。)

为了利用滚动球进行 $SU(2)$ 转动, 我们用

$$U = I_2 \cos \frac{\theta}{2} - i \vec{n} \cdot \vec{\sigma} \sin \frac{\theta}{2}, \quad (2.3.8)$$

其中 $I_2$ 是 $2 \times 2$ 单位矩阵,  $\vec{\sigma}$ 表示 $SU(2)$ 服从循环关系 $\sigma_x^2 = 1, \sigma_x \sigma_y = i \sigma_z$ 的 $2 \times 2$ 矩阵基。这相当于一个基于四元数的转换, 其中 $(c, u) = (\cos(\theta/2), \vec{n} \sin(\theta/2))$ 。注意, 与完整矩阵Eq.(3)相比, Eq.(8)合并滚动球的基本参数要简单得多。



更改 $\vec{n}$ 的整体符号将产生围绕对象的视点旋转，而不是视图内的对象旋转。

矩阵Eq.(8)的元素可以根据需要从Eq.(3)中计算一个普通的矢量旋转矩阵，也可以直接用作 $2 \times 2$ 矩阵来旋转旋量(Edmonds, 1957)，这是旋转群所能作用的最基本对象。

### 欧几里得四维

在四维欧氏空间中， $O(4)$ 群有6个转动自由度，而不是由于 $O(3)$ 而在三维空间中存在的3个。

6个 $O(4)$ 旋转算子 $L_{\mu\nu}$ ,  $\mu, \nu = \{1, 2, 3, 4\}$ ,  $L_{\mu\nu} = -L_{\nu\mu}$ , 可以通过定义以下组合分解为 $O(3) \times O(3)$ :

$$L_i^\pm = \frac{1}{2} \left( \frac{1}{2} \epsilon_{ijk} L_{jk} \pm L_{4i} \right). \quad (2.3.9)$$

这里 $\epsilon_{ijk}$ 是三维中的完全反对称张量，我们使用重复罗马指标从1到3求和的惯例。每一个组合都服从独立的 $O(3)$ 变换关系，

$$[L_i^\pm, L_j^\pm] = -\epsilon_{ijk} L_k^\pm, \quad [L_i^\pm, L_j^\mp] = 0, \quad (2.3.10)$$

因此可以使用 $O(3)$ 滚动球算法单独控制。以这种方式产生的旋转可以写成

$$R^\pm = I_4 \cos \theta + \vec{n} \cdot \vec{L}^\pm \sin \theta, \quad (2.3.11)$$

其中

$$\vec{n} \cdot \vec{L}^\pm = \begin{vmatrix} 0 & -n_z & n_y & \mp n_x \\ n_z & 0 & -n_x & \mp n_y \\ -n_y & n_x & 0 & \mp n_z \\ \pm n_x & \pm n_y & \pm n_z & 0 \end{vmatrix},$$

单位向量 $\vec{n}$ 通常由式(1)定义。因此，我们可以使用滚动球的两个副本来操纵四维方向的所有自由度，一个为 $L_1^+$ ，一个为 $L_1^-$ 。

另一种方法也适用于N维欧氏空间中的旋转，它是将群 $O(4)$ (或N中的 $O(N)$ )分解为 $O(3)$ 子群，并将每个子群视为独立的滚动球变换。

## 洛伦兹变换

研究高速物理系统必须使用时空洛伦兹变换，而不是欧几里得旋转。洛伦兹变换将空间和时间混合在一起，保留了一个具有一个负分量的闵可夫斯基空间二次型。纯粹的速度变化，或“加速”，在形式上类似于用双曲函数取代三角函数的旋转。速度  $\vec{v} = \hat{v} \tanh \xi$  通过矩阵转换向量  $(\vec{x}, t)$

$$\begin{pmatrix} \delta_{ij} + \hat{v}_i \hat{v}_j (\cosh \xi - 1) & \hat{v}_j \sinh \xi \\ \hat{v}_i \sinh \xi & \cosh \xi \end{pmatrix}. \quad (2.3.12)$$

为了实现  $O(2, 1)$  洛伦兹转换(它保留了  $\text{diag}(1, 1, -1)$  的形式)，我们将鼠标运动解释为“洛伦兹滚动球”在鼠标移动方向上的微小速度变化。(我们也可以研究物理时空的变换群  $O(3, 1)$ ；不幸的是，导致式(10)的论点的类比需要引入复向量。)

$O(2, 1)$  转换的无穷小生成器是 boost 操作符

$$B_x = t \frac{\partial}{\partial x} + x \frac{\partial}{\partial t}, \quad B_y = t \frac{\partial}{\partial y} + y \frac{\partial}{\partial t},$$

和操作符

$$L = x \frac{\partial}{\partial y} - y \frac{\partial}{\partial x}$$

在  $x - y$  面上产生旋转。boost 算子在旋转下变换为普通向量， $[L, B_x] = -B_y$ ,  $[L, B_y] = +B_x$ ，而它们的相互交换产生的旋转符号与类似的  $O(3)$  算子， $[B_x, B_y] = +L$  相反。

我们将鼠标输入与  $O(2, 1)$  转换 Eq.(12) 关联起来，将 Eq.(1) 替换为

$$\hat{v}_x = \frac{+dx}{dr}, \quad \hat{v}_y = \frac{+dy}{dr} \quad (2.3.13)$$

选择 boost 参数为  $\xi = \tanh^{-1}(dr/s) \approx (dr/s)$ ，其中  $s$  是一个合适的缩放因子，确保  $(dr/s) < 1$ 。

然后我们发现，在小的顺时针方向移动输入设备会产生顺时针方向坐标框架的空间部分的旋转，这与标准  $O(3)$  旋转的结果相反！这种效应被称为托马斯进动，使滚动球技术成为洛伦兹变换的一种非常自然的技术。

## 总结

总之，滚动球技术提供了一种在交互式图形系统中使用二维输入设备控制三个转动自由度的方法，这种方法不依赖于输入设备的状态、位置或历史。由于算法丰富的群论起源，许多相关的科学可视化应用自然而然地出现了。要想熟练使用该技术，用户需要付出一些努力。但是，一旦掌握了这种方法，就会提供与情境无关的、探索性的方向调整，强烈支持直接操作的感觉。

## 鸣谢

这项工作的一部分得到了美国国家科学基金会的资助。是- 8511751。

See also G1, 462 .

## 2.4 区间运算

Jon Rokne

The University of Calgary

Calgary, Alberta, Canada

在计算机图形学中，离散化问题发生在两个不同的领域：

- 计算的最终输出是用有限分辨率的设备显示或打印的二维图像，这会造成诸如混叠等不良影响。
- 确定位置、强度和颜色所需的计算在通用计算设备或专用图形计算机中进行。在任何一种情况下，存储实数的基本方法都是所谓的浮点表示。这种表示为存储浮点数分配固定数目的二进制位数，浮点数可以是输入或输出量，也可以是中间计算的结果。

我们将讨论一种工具，区间分析，它使用有保证的上界和下界来估计和控制数值计算中可能出现的数值误差，特别是在计算机图形问题中出现的数值误差。区间算术是一门涉及面很广的学科，我们只涉及到一些基本思想。然而，我们注意到区间算术和分析已经导致了基于包含和收缩的新技术的发展，这些技术适用于计算机图形学中的一些问题。

我们首先给出一个可能发生在图形应用程序中的问题的例子。一个基本例程可能包括确定两条线(为了简单起见，在 $E^2$ 中)是否平行(也就是说，它们是否有一个有限的交集)。如果它们相交，计算交点。让线条保持原样

$$0.000100x + 1.00y = 1.00 \quad (2.4.1)$$

$$1.00x + 1.00y = 2.00 \quad (2.4.2)$$

假设这个算法是一个三位数四舍五入的算法。四舍五入到五位数的真正解是 $x = 1.0001$ 和 $y = 0.99990$ ，而使用Forsythe和Moler(1967)中描述的过程，三位数的算术给出 $y = 1.00$ 和 $x = 0.00$ 。在本文中，使用不同的计算顺序安排也得到了其他错误的结果。从这个例子可以看出，这样的计算可能有很大的误差，以致于一个应该在区域内的交集实际上可能被计算在区域外。

另一个例子是区域填充，其中区域的连通性依赖于一个计算，该计算可能以与交叉计算相同的方式充满错误。

这样的错误很难防范，最终它们会在计算机生成的场景中产生不受欢迎的工件，并且很难在大型程序中跟踪和纠正。

其中许多误差可以使用区间算法自动控制。它使程序能够给出项目 $p$ 和集合 $P$ 的三个答案之一。

1.  $p$ 肯定在 $P$ 中。
2.  $p$ 肯定不在 $P$ 中。
3. 在执行的计算和可用的精度中，不可能告诉 $p \in P$ 或 $p \notin P$ ，也就是说，结果是不确定的。

根据前面的例子，我们可以说直线相交，它们不相交，或者它们是否相交是不确定的。类似地，我们可以声明一个域是连接的，它是没有连接的，或者域是否连接是不确定的。在每一种情况下，可以将决策程序内置于程序中以处理这三种情况。

区间运算有着悠久的历史;然而，它的现代使用源于摩尔(1966)的《区间分析》一书的出版。后来，有大量的出版物专门讨论这个问题。Garloff出版了参考书目(1985,1987)，举行了一些会议，最近出版了一份新的苏联期刊，《区间计算》(新技术研究所，1991)，完全致力于区间分析。

区间算术定义如下:设  $I = \{A : A = [a, b], a \cdot b \in \mathbf{R}\}$  为实紧区间集，设  $A, B \in I$ 。然后区间算术运算定义为

$$A * B = \{\alpha * \beta : \alpha \in A, \beta \in B\},$$

其中  $*$   $\in \{+, -, \cdot, /\}$  (注意  $/$  在  $0 \in B$  时未定义)，即  $A * B$  的区间结果包含所有可能的点结果  $\alpha * \beta$ ，其中  $\alpha$  和  $\beta$  是实数，使得  $\alpha \in A$  和  $\beta \in B$  和  $*$  是基本的算术运算之一。

这个定义是由下面的论证驱动的。给定两个区间 $A$ 和 $B$ ，我们知道它们分别包含精确的值 $x$ 和 $y$ 。然后定义保证 $x * y \in A * B$ 中用于上述任何操作，即使我们不知道 $x$ 和 $y$ 的确切值。

这个定义在实际计算中不是很方便。令 $A = [a, b]$ ，且 $B = [c, d]$ ，可得其等价于

$$\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d], \\
[a, b] - [c, d] &= [a - d, b - c], \\
[a, b] \cdot [c, d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)], \\
[a, b]/[c, d] &= [a, b][1/d, 1/c] \text{ if } 0 \sim [c, d],
\end{aligned} \tag{2.4.3}$$

这意味着在 $*$   $\in \{+, -, \cdot, /\}$ 中的每个区间操作都被简化为真正的操作和比较。

区间算术的一个重要性质是

$$\begin{aligned}
A, B, C, D \in \mathcal{A} \subseteq B, C \subseteq D, \Rightarrow A_{\square} * CC_{\square} \subseteq B_{\square} * D \\
\text{for } * \in \{+, -, \cdot, /\}
\end{aligned} \tag{2.4.4}$$

如果定义了操作。换句话说，如果 $A$ 和 $C$ 分别是 $B$ 和 $D$ 的子集，那么 $A * C$ 是用于任何基本算术操作的 $B * D$ 的子集。因此，在计算的任何阶段引入的错误，如浮点错误或输入错误，都可以被解释。由于式(4)的重要性，它被称为区间运算的包含等度。

如果定义了操作。换句话说，如果 $A$ 和 $C$ 分别是 $B$ 和 $D$ 的子集，那么 $A * C$ 是用于任何基本算术操作的 $B * D$ 的子集。因此，在计算的任何阶段引入的错误，如浮点错误或输入错误，都可以被解释。由于式(4)的重要性，它被称为区间运算的包含等度。

这样做的一个结果是，任何可编程的实计算都可以使用操作之间的自然对应关系嵌入到区间计算中，因此，如果 $x \in X \in I$ 中，则 $f(x) \in f(X)$ 中，其中 $f(X)$ 被解释为 $f(.x)$ ， $x$ 替换为 $X$ ，操作替换为区间操作。

间隔算术的另一个重要原理是，它可以在浮点计算机上实现，这样得到的间隔包含使用等式进行的真实间隔计算的结果。(3)和定向四舍五入。有几种软件系统可用于此目的，如PASCAL-SC (Bohlender et al., 1981)。实现只需要注意间隔端点的每个计算都是从间隔的内部向外舍入。

区间算术也有一些缺点:

- 减法和除法不是加法和乘法的逆运算。
- 分配律不成立。只有子分配律 $a(B + C) \subseteq AB + AC$ ,  $a, B, C \in I$ 是有效的。

- 区间算术操作比相应的实际操作大约耗时3倍(尽管某些问题的区间算术实现可能比相应的实际版本运行得更快;参见suffn and Fackerell, 1991)。

作为使用区间计算的一个简单例子, 我们使用三位区间算术来考虑上面给出的相交线问题。根据克拉默法则, 我们得到

$$x = \left| \begin{array}{cc} 1.00 & 0.000100 \\ 2.00 & 1.00 \end{array} \right| / \left| \begin{array}{cc} 0.000100 & 1.00 \\ 1.00 & 1.00 \end{array} \right|$$

和

$$y = \left| \begin{array}{cc} 0.000100 & 1.00 \\ 1.00 & 2.00 \end{array} \right| / \left| \begin{array}{cc} 0.000100 & 1.00 \\ 1.00 & 1.00 \end{array} \right|.$$

利用区间算法得到

$$x \in X = [0.980, 1.03]$$

和

$$y \in Y = [0.989, 1.02],$$

在每种情况下都包含精确解。这个计算与Forsythe和Moler(1967)引用的计算相比, 在实际算术中对应更稳定的计算。如果这个特定的计算序列是在区间算术中执行的, 那么间隔会更大, 但在所有情况下, 准确的结果都包含在结果的区间中。

区间算术的一个有趣的特点是, 它可以用来开发新的算法, 而不是简单地扩展实际算术中的算法。其中一个例子是摩尔(1966)首先提出的区间牛顿法。假设给定 $F(x)$ , 并假设我们想要找到在给定区间 $X_0$ 中 $F(\xi) = 0$ 的点 $\xi$ 。然后定义区间牛顿法为迭代

$$X_{n+1} = m(X_n) - F(X_n) / F'(m(X_n)), \quad n = 0, 1, \dots,$$

其中 $m([a; b]) = ((a + b)/2)$ 和 $F'(X)$ 是 $F$ 的导数的区间求值。这个方法有一些有趣的性质。

1. 如果 $F$ 的0  $\xi$ 存在于 $X_0$ 中, 则对于所有 $n$ ,  $\xi \in X_n$ 中;见Moore(1966)。这意味着初始 $X_0$ 中的所有零都保留在后续的间隔中。
2. 如果 $X_n$ 对于某个 $n$ 是空的, 那么 $F$ 在 $X$ 中没有零(Moore, 1966)。

方程解的区间迭代的进一步性质可以在neuaier(1990)中找到。

该方法可应用于计算机图形学中的射线跟踪。隐式曲面和射线之间的交点计算会导致一个问题，即找到函数 $F(x) = 0$ 的一个(最小)根或所有根(见Hanrahan, 1989)。通过使用区间算术技术，可以保证结果包含在生成的区间中，避免渲染过程中的异常(参见Kalra和Barr, 1989，关于该问题的讨论)。

在Mudur和Koparkar(1984)以及summern和Fackerell(1991)中，我们发现了在隐式表面绘制中使用区间算法，在轮廓绘制算法和面度估计中使用区间算法的进一步讨论，其中它与细分技术相结合，以改善结果。

See also G2, 394.



## 2.5 快速生成循环序列

Alan W. Paeth

NeuralWare Incorporated

Pittsburgh, Pennsylvania

自由运行的内部循环通常需要一系列重复每个 $N$ 步骤的值或条件。例如，高速 $z$ -缓冲区绘制技术(Booth et al., 1986)必须以三个周期执行缓冲区交换和内务管理。当 $N$ 不是2的幂时，直接检查寄存器的低位可能不会被用来形成 $N$ 的计数模。类似地，一个快速的二维 $N$ -gon生成器需要循环生成 $N$ 的值序列，顶点 $N$ 与顶点0相同。这个Gem考虑了 $N < 8$ 的紧凑方法，它使用的机器指令不超过三条，也不超过三个寄存器。不使用条件逻辑，使得该技术非常适合手工编码。

### $N = 2$ (Review)

我们熟悉的双重“toggle”在true和false之间交替：

```
condition := not (condition);    True in alternating cases
if (condition) ...
```

(2.5.1)

类似地，值的双重“循环”是一个简单的交替。当两个值都是预先确定的，一条指令和一个整数寄存器就足够了：

```
register  $a := v1$ ;           initialize
constant  $c := v1 + v2$ ;
repeat                               cycle
     $a := a - c$ ;            $a : [v1 \ v2 \dots]$ 
```

例如，Wirth(1973)通过使用(2.2)来加速质数筛选，生成序列 $\left[ 2 \ 4 \ 2 \ 4 \ \dots \right]$ ，该序列表示不能被3除的连续奇数之间的距离(Knuth, 1981)。用逻辑异或操作重写(2.2)的算术，可以得到一种著名的专利方法，以交替的方式反转帧缓冲区的像素。以算术形式，重新推导出适合于灰度帧缓冲区的像素反演方案(Newman和Sproull, 1979)。

通常情况下，循环序列只在运行时指定。对于 $N = 2$ ，循环是交换，很容易在三个算术或逻辑操作中完成，而不需要借助第三个持有寄存器，分别在Paeth(1990)和Wyvill(1990)之前的Gems中描述过。

### N = 3 (扩展)

成对交换技术不能优雅地扩展:序列[a, b, c]的循环排列通过交换(例如)位置(1,2)和(2,3)的元素需要6条指令和3个寄存器。第一性原理循环队列方法需要 $N + 1$ 个寄存器和 $N + 1$ 个赋值。后者虽然简单明了，但仍然超出了序言中规定的指令和寄存器的限制:

```

r1 := r1 xor r2;   r2 := r2 xor r1;           rx := r1;   r1 := r2;
r1 := r1 xor r2;   r2 := r2 xor r3; <versus>  r2 := r3;   r3 := rx;
r3 := r3 xor r2;   r2 := r2 xor r3
    
```

通常，如(2.1)所示，只需要值来触发1-in- $N$ 事件。对于 $N = 3$ ，两个寄存器和两行就足够了。每个寄存器指令都是紧凑的双运算形式 $\ll rx = rx \text{ op } ry \gg$

```

register r1 := 0;           Three fold trigger
register r2 := 1;
repeat                                cycle:
    r1 := r1 xor r2;   r1 : [1  1  0  ...]
    r2 := r2 xor r1;   r2 : [0  1  1  ...]
    
```

这就产生了所示的三级 $(r1, r2)$ 列集。当一个寄存器为零时，就会触发。在假设硬件条件代码是由前面的逻辑操作设置的情况下，对 $r2$ 的测试简化了操作。测试的阶段可以通过在初始化 $(r1, r2)$ 时替换第三列以外的列来调整。触发2-3次定义了互补集：非零的测试被替换。三个相位不同的1-in-3测试可以同时进行，形成一个循环的开关。

```

if r = 0 then ...   1-in-3, phase = 0
if r1 = r2 then ... 1-in-3, phase = 1
if r1 = 0 then ...  1-in-3, phase = 2
    
```

条件和相关块可以嵌入到异或操作中，以利用隐式条件代码感知。也就是说，隐式定义模数为3的计数器的两条异或线不必相邻。

三个寄存器中三个变量的循环排列可以在最少的指令数(3条)内完成。推导过程并不明显，并与后面描述的三重算术情形相关。

---

---

```
register int a=c1; // Three fold cycle
register int b=c1 xor c2 ;
constant int c=c1 xor c2 xor c3 ;
repeat // cycle:
a=a xor b; //  a:  [c1 c2 c3...]
b=b xor c ;
b=b xor a ;
```

---

逻辑**xor**的使用是有价值的，因为元素可以是整数、指针和浮点数的混合序列;算术运算不允许这样做。注意，最后两行都更新了b中的值，而寄存器c从未被写入。这暗示了另一条路线:

$$b := b \text{ xor } (a \text{ xor } c)$$

其中c等于一个预先确定的编译时间常数。但是，该值在运行时通常必须占用第三个寄存器。(请参阅c代码中产生循环[1,2,3]的双寄存器变体。)当只需要触发时，修复 $c = 0$ 将省略循环的中线，重新构造 $N = 3$ 触发情况。另外，这两行可以看作是熟悉的异或交换代码的前两行。因为后者的最后一行与第一行匹配，所以三次通过两行异或代码在两个寄存器上产生的顺序动作与两次通过三行代码产生的顺序动作相同(这是通过指令分组在式(3.1a)中提出的)。两者都定义了恢复双交换:对两个元素进行不少于6步的标识操作。因此，两行代码形成长度为3的循环，同时生成序列 $[r1, r1 \oplus r2, r2]$ 。

$$N = 3, 4, 6$$

值得注意的是，长度不超过 $N = 6$ 的循环只需要两个寄存器。显然，没有足够的存储空间来交换所有元素，否则 $N + 1$ \*寄存器和 $N + 1$ \*赋值就足够了。相反，目标是派生一组值(在一个或两个寄存器上)，其中所有生成的值都是不同

的。因此，寄存器必须“计数”，并与第一性原理代码相竞争，例如快速六边形绘制例程：

```
Xval = X_Value_Table[(i := (i + 1 mod 6))];
Yval = Y_Value_Table[i];
```

在这里，模数是一个主要的费用：它的成本与整数除法相当。另一种第一性原理方法使用条件逻辑重新启动一个递减的计数器，在现代流水线硬件上产生很大的分支惩罚，而小N则使情况变得更糟。

在(6.2)中可以看到，二维六边形的顶点生成可以使用这样的六重循环：

```
register x = y = 1;

repeat

    Xval = X_coord[x := x+y] ;

    Yval = Y_coord[y := y+not(x)]
```

其中**not**(x)是位反转，即**not**(x) = x **xor**(-1)在两个互补硬件下。长度为7的数组需要有合适的偏移量，如配套的C代码所示。

## N = 6 推导

非均匀旋转可以用函数组合来模拟。即 $F(F \cdots (F([x])) \cdots) = [x]$ 在不少于N步骤。例如，线性分数函数 $F(x) = [2x - 1] / [x + 1]$ 得到 $F^6(x) = x$  (Yale, 1975)。这种形式可以直接等同于 $2 \times 2$ 矩阵的代数(Birkhoff and MacLane, 1965);为了便于推导，优先处理前者。

两个寄存器的值可以用整数点阵上的点 $[x, y]$ 表示，每个寄存器一个坐标。作为一个(列)向量 $v$ ，函数 $F$ 是一个 $2 \times 2$ 矩阵，它的前乘是 $v$ 。对于给定的N， $F$ 必须确定为： $F^N v = I$ 。当 $F$ 是一个剪切矩阵时，旋转可以在三个剪切中实现(Paeth, 1986)，每个剪切只需要一个赋值语句(第182页)。当非对角矩阵元素为 $\{\pm 1\}$ 时，不会发生乘法运算，一条机器指令就足够了。全有理形式也产生旋转，但圆周点的集合不是单位根(在复笛卡尔平面上单位圆内刻的 $n$ 边形的顶点)。唯一的解决方案是四次旋转。这种分解是：

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

第一个和第三个矩阵的重新分组(第192页)允许两行旋转, 在提供隐式乘2的机器上很有用:

$$\begin{aligned} x &:= x + y; & x &:= x + (2^*y); \\ y &:= y - x; & < \text{or} > & y := y = x; \\ x &:= x + y; \end{aligned} \quad (4.1)$$

这种顺序形式比紧凑的 $\{x = -y, y = x\}$ 形式稍微昂贵一些, 当寄存器值可以同时重赋时, 就像在硬件中一样。3和6的旋转可以通过寻找符号形式的X和Y剪切的乘积的特征值并将它们与单位根相等来形成, 从而确定两个离轴元素的值。在其最紧凑的形式中, 这将产生二次方程 $z = \frac{1}{2}(m \pm \sqrt{m^2 - 4})$ , 它可以表示 $N = \{1, 2, 3, 4, 6\}$ 的单位根 $z^N = 1$ , 其值分别为 $m = \{2, 2, 1, 1, 0, 1\}$ 。使用MAPLE对矩阵方程 $(XY)^N = I$ 的符号矩阵元素的解, 不揭示与下面通解明显不同的实值积分形式:

$$\left( \begin{bmatrix} 1 & 0 \\ m \mp 2 & \pm 1 \end{bmatrix} \begin{bmatrix} \mp 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \pm 1 & 1 \\ \pm m - 2 & m \mp 1 \end{bmatrix} \right)^N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

$$(m, N) = \{(-1, 3), (0, 4), (1, 6)\}$$

因此, 使用单位元件可以进行三倍和六倍的旋转。考虑到 $\cos 60^\circ$ 的不合理性, 这是意料之外的。简化的双剪切旋转形式的变形已成为固定顶点到积分位置的优点。注意 $N = \{3, 4, 6\}$ 的三个非平凡解枚举了平铺平面的 $N$ 边形集合(图1a, 1b)。

我们对所有具有小乘数的三指令、三寄存器剪子进行了自动检查。没有发现新的 $N$ 的解决方案, 而且大多数形式没有明显的区别。双寄存器的 $N = 3$ 的形式被用xor重写, 导致(3.3)。 $N = 6$ 的双寄存器形式被认为可以容纳一个额外的常数, 它抵消了 $x$ 中的六边形, 如图1b所示, 下面用代数法表示:

$$\begin{aligned} a &:= a + b; & a &: \begin{bmatrix} a & a + b & b + c & -a + 2c & 2c - a - b & c - b \end{bmatrix} \\ b &:= b - a + c; & b &: \begin{bmatrix} b & c - a & c - a - b & -b & a - c & a + b - c \end{bmatrix} \end{aligned} \quad (6.1)$$

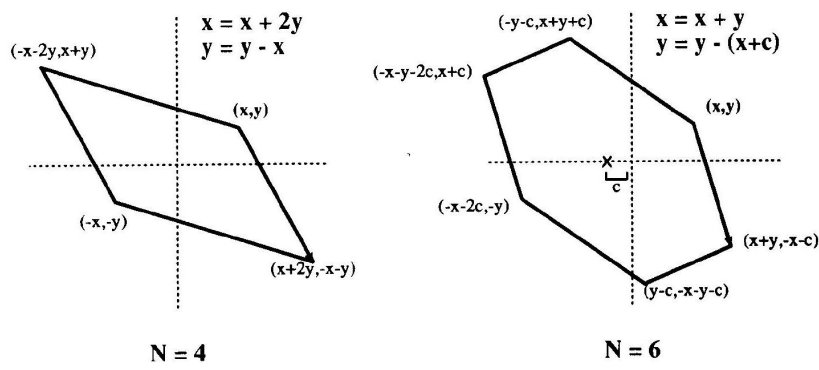


图 2.5.1

当 $c = 0$ 时，寄存器 $a$ 中的值滞后于寄存器 $b$ 中的值两步，提示存在一个 $(\cos t \text{fi} \sin t)$ 生成器，除了 $90^\circ$ 正交变成 $120^\circ$ 相位偏移。通过 $c \neq 0$ ，实现了一系列不同值的生成，达到了最初的目标。设置 $c = -1$ 允许使用逻辑补体- $(a + 1)$ 项的隐式形式(Paeth和Schilling, 1991)，给出：

$$a = b = 1; \quad \text{six-fold fixed-value cycle}$$
$$\text{repeat}$$
$$a := a + b; \quad a : \begin{bmatrix} 1 & 2 & 0 & -3 & -4 & -2 \end{bmatrix}$$
$$b := b + \text{not}(a); \quad b : \begin{bmatrix} 1 & -2 & -3 & -1 & 2 & 3 \end{bmatrix}$$

(6.2)

其中， $c$ 偏移量横向位移六边形的中心，消除了中心反转的对称性。这有助于实现不同的价值。对于 $a > 0$ ， $b > 0$ 和 $2c > a + b$ ， $a$ 中的序列始终为正。

N = 6 (触发)

使用六重形式可以任意触发。上述算法的不同值允许并发1-in- N触发器具有任何相位偏移。然而，带有非相邻触发器的2-in-N和3-in- N形式存在更大的困难:它们可能不能通过用不等式替换相等测试来创建。这是图形凸性的结果:在几何术语中，像 $y > 4$ 这样的测试表示值的水平半空间。多边形与平面的交点将其分割为两个相邻顶点的不同边界集。目标是一个简单的触发器，它不需要像上面引用的Gem那样使用寄存器内位测试。

6个状态允许64个触发模式，其中第 $i$ 个位置的 “\*” ( “.” )表示第 $i$ 个状态的

触发器解除。互补模式的消除使这个数字减半。包含重复触发器的模式，如“\*\*\*...”和“\*.\*.\*”可以分解为超循环或子循环并被消除。左边是三个原型模式，分别设置了一个、两个或三个比特。测试使用(6.2)中的六倍旋转变量，隐含( $c = -1$ )和起始值 $a = b = 0$ :

(6.3)

a:    0    0  -1  -2  -2  -1

b:    0  -1  -1    0  +1  +1

a = 0 AND b = 0: \*    .    .    .    .    .

a = 0:    \*    .    \*    .    .    .

a = 0 OR b = 0: \*    \*    .    \*    .    .

**xor**的广泛使用提出了类似于在整数**mod**1上生成伪随机数(RPN)的方法(参见Morton, 1990)。传统的移位和携带测试逻辑硬件可以直接“连接”到三个**xor**寄存器指令，具有排列形式，给出

```
repeat    cycle by seven
b := b xor a;
c := c xor b;
a := a xor b
```

这将产生如下所示的值表。

A	B	C
a	b	c
a ⊕ b	b ⊕ c	a ⊕ b ⊕ c
a ⊕ c	a	b
c	a ⊕ b	b ⊕ c
a ⊕ b ⊕ c	a ⊕ c	a
b	c	a ⊕ b
b ⊕ c	a ⊕ b ⊕ c	a ⊕ c

这里A列领先B两步，同样B领先C，但C领先A三步。每一列取三个变量中所有N - 1可能的**xor**排列，省略了禁止的零状态。这并不限制零元素的周期性生

成, 可以通过将 $\{a, b, c\}$ 的任意(而不是全部)设为零, 或者将两个寄存器中的初始值相等来形成, 因为 $M \text{ xor } M = 0$ 。

使用四个寄存器( $r = 4$ )表示 $2^4 - 1 = 15$ 状态。由于 $r$ 是偶数,  $N$ 是因子 $(2^2 + 1)(2^2 - 1)$ 的复合。这揭示了 $N = 5$ 的子循环, 将小 $N$ 的表舍入。然而, 这种方法比旅法(5个变量, 1个临时寄存器, 6个分配)只显示了边际增益, 因此没有进行研究。对于那些倾向于大 $N$ 的人, 可以使用因子来组成更大的循环: 相对素数长度的并发循环在若干步等于它们长度的乘积(GCM)后重新同步。

对于最后一个个位数值,  $N = 9$ 仍然很困难, 因为它既不是质数, 也不是无平方的合数。(11, 13)的下一个质数不属于 $2^m - 1$ 梅森形式。根据费马定理, 它们(和任何质数 $p$ )都是 $2^{p-1}$ 的因数, 这里 $2^{10} - 1$ 和 $2^{12} - 1$ 。由于这意味着寄存器的数量至少与 $\text{xor}$ 方法的周期长度呈线性增长, 因此brigade方法由于简单而获胜。尽管到目前为止所探索的所有方法的实际限制都是 $N < 8$ , 但更奇特和更复杂的方法是可能的, 并且可以通过暴力手段进行检查。其中之一如下所示。

## N = 24

作为最后一个例子, 代码

```

register a = 4;
register b = 3;
repeat
    a := a - b;
    a := a bit-and 15; explicit limit on register a
    b := b xor a;

```

提供了一种循环模24的方法。将寄存器 $a$ 的域限制为16个值必然会引入值的多样性。所选的初始值将 $a$ 和 $b$ 限制在域 $[1..]$ 并进一步保证它们永远不会同时相等。

这段代码的价值是分别使用条件测试 $(b = 1) \Delta (a = 4) \Delta (b = 7)$ 和 $(b = 12)$ 形成并行的24:1、12:1、8:1和6:1的速率除法。这些测试被选择, 因此在任何步骤中最多一个为真, 通过将 $\{1, 2, 3, 4\}$  -in-24测试通过触发位的ring组合, 允许速率乘法(最多10-in-24)。注意, 只有3 / 24的比率显示出轻微的不均匀性:



a :	1	15	2	3	7	12	5	3	2	15	3	4	9	7	2	11	15	12	13	1	12	7	11	4
b :	2	13	15	12	11	7	2	1	3	12	15	11	2	5	7	12	3	15	2	9	11	12	7	3

---

b=1 :	*																							
a=4 :											*											*		
b=7 :					*								*									*		
b=12 :			*				*					*						*					*	

总结

给出了循环生成任意值和布尔状态的方法。对 $N = \{2, 3, 4, 6, 7\}$ 的情况进行了详细的处理。附录中提供的大量c代码变体为图形程序员的技巧包提供了一组有用的补充。

See also G1, 436.

## 2.6 一个通用的像素选择机制

Alan W. Paeth

NeuralWare Incorporated

Pittsburgh, Pennsylvania

反转帧缓冲区像素的颜色是突出显示区域的常用方法。一个有用的反转函数提供了视觉上不同的颜色对。在较新的硬件上，查找表(映射像素的外观)是通过窗口输入的，引入了空间依赖性。这给“最佳”函数的设计带来了负担。这个宝石提供了一个简单的先验解决方案，可以保证视觉上不同的颜色对，尽管算法仍然不知道它们的最终外观。典型的用途是创建屏幕范围、窗口不变的工具，例如用于显示“快照”的系统游标或选择矩形。

一个有用的反求函数 $F$ 在像素 $p$ 上满足两个代数条件： $F(F(p)) = p$ 和 $F(p) \mu p$ 。第一个条件保证函数是它自己的逆函数。第二个关键是保证任何颜色对中的两个元素“不几乎相等”，使它们在视觉上不同。对于一位像素，互补(位切换)是明显的解决方案。在更高的精度下，所有位的(ones)补变成了一个算术运算： $F(p) = \text{not}(p) = -1 - p$ 在2的补运算下(Paeth, 1991)。对于 $0 \leq c < 1$ ，这已经被推广(Newman和Sproull, 1979)为 $F_c(p) = \text{frac}(c - p)$ 。这不能满足第二个条件：对于参数 $c$ ，值 $c/2$ 的像素映射到它自己。几何上，单位区间已经显示(通过 $c$ )，并映射到原始区间，从而引入一个静止点。

在调色板系统中使用的解决方案(Higgins和Booth, 1986)回到了逻辑运算作。给定一个二进制整数，它定义了沿区间的离散位置，仅最上面的位的位互补交换区间的上下半部分，而没有任何镜像。任何颜色对中的像素现在都被间隔距离的一半取代，以保证不同的颜色。对于颜色映射的像素(作为索引)，一对中的元素在映射函数的域中被移除得很远，如果颜色映射定义了单调函数，则产生的颜色在范围内也被移除——这是一种常见的情况。某些非线性非笛卡尔色图(Paeth, 1991)在这个函数下也能很好地工作，并支持简单的几何解释。

现在可以通过做一些简单的假设来构造泛型函数。典型帧缓冲器上单色通道的像素精度为1位、4位或8位。操作

```
macro bwpixflip (x)  x := x bit-xor 133      hex 85
```

在不知道使用精度的情况下，对最上面的位进行补位。当底层像素精度较低时，

切换高阶位无关紧要，或者被硬件写掩码的动作压制。相反，在高精度上的操作，像素将补充额外的低精度位，但这些被充分去除会产生很大的后果。

对于RGB像素，十六进制85的三个副本确保在三个相邻通道上操作。这还在第12位引入了一个切换，在硬件上提供了扩展的单色精度或颜色表索引。通用的颜色反转函数是

```
macro pixelflip(x)  x := x bit-xor 8750469      hex 858585
```

操作的三次使用将沿着每个颜色轴交换一半的单位间隔。从几何上看，这表示在单位颜色立方体中围绕中层灰色中心点( $\frac{1}{2}, \frac{1}{2}, \frac{1}{2}$ )的八个子立方体的变换。在非魔方的方式下，每个立方体的朝向被保留(图2.6.1a)。在第一性原理“xor - 1”的情况下(未显示)，八个立方体的附加中心反转反转了整个固体，并且在中间灰色位置重新引入了不希望出现的静止点。

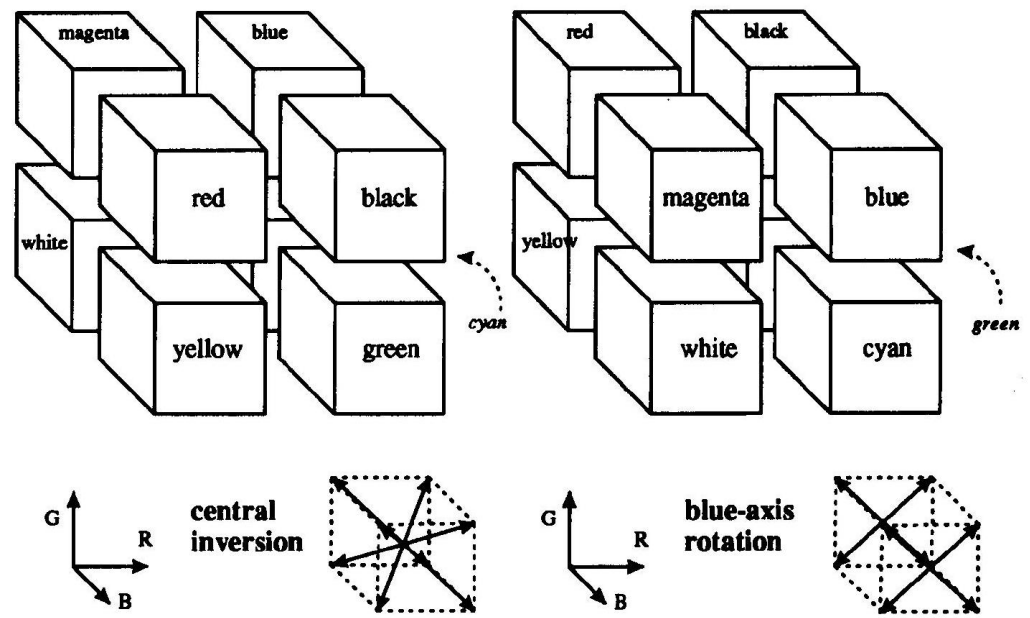


图 2.6.1:

最后，不补充蓝色通道通常是有利的。当蓝色占据最上面的像素位(如Adage/Ikonas或SGI/Iris)时，定义红色和绿色通道的下16位的互补仍然发生;所有单色和查找情况(像素精度从不超过16位)也隐含地涵盖。另一个通用宏是

```
macro pixelflip2(x)  x := x bit-xor 34181 hex 8585
```

对于24位颜色，保留蓝色意味着子立方体不再通过中心反转交换(图1a)，而是以“摩天轮”方式围绕蓝色轴旋转半圈(1b)。这就产生了一对对立的颜色(红、绿)，人类的视觉系统对它们的反应非常灵敏，再加上一对(蓝、白)、(青色、品红)和(黄、黑)。备用宏支持在反转中使用短的16位整数。

See also G1, 215; G1, 219; G1, 233; G1, 249.

## 2.7 经扭曲的非均匀随机点集

Peter Shirley

Indiana University

Bloomington, Indiana

我们经常希望在单位正方形上生成随机或伪随机点集，用于分布射线跟踪等应用。有几种方法可以做到这一点，例如抖动和泊松盘采样。这些方法给出了一组在单位正方形上合理分布的 $N$ 点： $(u_1, v_1)$ 到 $(u_N, v_N)$ 。

有时，我们的采样空间可能不是方形的(例如，圆形透镜)或可能不是均匀的(例如，以像素为中心的滤波函数)。如果我们能写一个数学变换，将我们的等分布点 $(u_i, v_i)$ 作为输入，并以期望的密度在期望的采样空间中输出一组点，那就太好了。例如，要对一个相机镜头进行采样，变换将取 $(u_i, v_i)$ 并输出 $(r_i, \theta_i)$ ，使新点在镜头的圆盘上近似均匀分布。

结果表明，这种变换函数在蒙特卡罗积分领域是众所周知的。表2.7.1给出了对计算机图形学有用的几种转换的表格。本文其余部分将讨论生成这种转换的方法。请注意，其中一些转换可以简化为简单密度。例如，要生成余弦分布的方向，使用Phong密度和 $n = 1$ 。要在单位半球上生成点，使用单位球密度上的扇区，带 $\theta_1 = 0, \theta_2 = \pi/2, \varphi_1 = 0$ 和 $\varphi_2 = \pi$ 。

对于蒙特卡罗方法，我们必须经常根据某个概率密度函数生成随机点，或根据某个方向概率密度生成随机射线。在本节中，将描述一种一维和二维随机变量的方法。本文的讨论紧跟Shreider(1966)的讨论。

如果密度是在区间 $x \in [a, b]$ 上定义的一维 $f(x)$ ，那么我们可以从一组均匀随机数 $\xi_i$ 中生成具有密度 $f$ 的随机数 $\alpha_i$ ，其中 $\xi_i \in [0, 1]$ 。为此，我们需要概率分布函数 $F(x)$ ：

$$F(x) = \int_a^x f(x') d\mu(x') \quad (2.7.1)$$

要得到 $\alpha_i$ ，只需转换 $\xi_i$ ：

$$\alpha_i = F^{-1}(\xi_i),$$

其中 $F^{-1}$ 是 $F$ 的倒数。如果 $F$ 不是解析可逆的，那么数值方法就足够了，因为所有有效的概率分布函数都存在一个逆。

表 2.7.1: (符号 $u, v$ 和 $w$ 表示范围超过 $[0,1]$ 的均匀分布随机变量的实例。)

Target Space	Density	Domain	Transformation
Radius $R$ disk	$p(r, \theta) = \frac{1}{\pi R^2}$	$\theta \in [0, 2\pi]$ $r \in [0, R]$	$\theta = 2\pi u$ $r = R\sqrt{v}$
Sector of radius $R$ disk	$p(r, \theta) = \frac{2}{(\theta_2 - \theta_1)(r_2^2 - r_1^2)}$	$\theta \in [\theta_1, \theta_2]$ $r \in [r_1, r_2]$	$\theta = \theta_1 + u(\theta_2 - \theta_1)$ $r = \sqrt{r_1^2 + v(r_2^2 - r_1^2)}$
Phong density exponent $n$	$p(\theta, \phi) = \frac{n+1}{2\pi} \cos^n \theta$	$\theta \in [0, \frac{\pi}{2}]$ $\phi \in [0, 2\pi]$	$\theta = \arccos((1-u)^{1/(n+1)})$ $\phi = 2\pi v$
Separated triangle filter	$p(x, y)(1 -  x )(1 -  y )$	$x \in [-1, 1]$ $y \in [-1, 1]$	$x = \begin{cases} 1 - \sqrt{2(1-u)} & \text{if } u \geq 0.5 \\ -1 + \sqrt{2u} & \text{if } u < 0.5 \end{cases}$ $y = \begin{cases} 1 - \sqrt{2(1-v)} & \text{if } v \geq 0.5 \\ -1 + \sqrt{2v} & \text{if } v < 0.5 \end{cases}$
Triangle with vertices $a_0, a_1, a_2$	$p(a) = \frac{1}{\text{area}}$	$s \in [0, 1]$ $t \in [0, 1-s]$	$s = 1 - \sqrt{1-u}$ $t = (1-s)v$ $a = a_0 + s(a_1 - a_0) + t(a_2 - a_0)$
Surface of unit sphere	$p(\theta, \phi) = \frac{1}{4\pi}$	$\theta \in [0, \pi]$ $\phi \in [0, 2\pi]$	$\theta = \arccos(1-2u)$ $\phi = 2\pi v$
Sector on surface of unit sphere	$p(\theta, \phi) = \frac{1}{(\phi_2 - \phi_1)(\cos \theta_1 - \cos \theta_2)}$	$\theta \in [\theta_1, \theta_2]$ $\phi \in [\phi_1, \phi_2]$	$\theta = \arccos[\cos \theta_1 + u(\cos \theta_2 - \cos \theta_1)]$ $\phi = \phi_1 + v(\phi_2 - \phi_1)$
Interior of radius $R$ sphere	$p = \frac{3}{4\pi R^3}$	$\theta \in [0, \pi]$ $\phi \in [0, 2\pi]$ $R \in [0, R]$	$\theta = \arccos(1-2u)$ $\phi = 2\pi v$ $r = w^{1/3}R$

如果我们有一个二维密度 $(x, y)$ 定义在 $[a, b: c, d]$ 上, 那么我们需要二维分布函数:

If we have a two-dimensional density  $(x, y)$  defined on  $[a, b: c, d]$ , then we need the two-dimensional distribution function:

$$F(x, y) = \int_c^y \int_a^{x'} f(x', y') d\mu(x', y').$$

We first choose an  $x_i$  using the marginal distribution  $F(x, d)$ , and then choose  $y_i$  according to  $F(x_i, y) / F(x_i, d)$ . If  $F(x, y)$  is separable (expressable as  $g(x)h(y)$ ), then the one-dimensional techniques can be used on each dimension.

As an example, to choose reflected ray directions for zonal calculations or distributed ray tracing, we can think of the problem as choosing points on the unit sphere or hemisphere (since each ray direction can be expressed as a point on the sphere). For example, suppose that we want to choose rays according to the density

$$p(\theta, \varphi) = \frac{n+1}{2\pi} \cos^n \theta,$$

where  $n$  is a Phong-like exponent;  $\theta$  is the angle from the surface normal and  $\theta \in [0, \pi/2]$  (is on the upper hemisphere); and  $\varphi$  is the azimuthal angle ( $\varphi \in [0, 2\pi]$ ). The distribution function is

$$P(\theta, \varphi) = \int_0^\varphi \int_0^\theta p(\theta, \varphi') \sin \theta d\theta d\varphi'.$$

The  $\sin \theta$  term arises because  $d\omega = \sin \theta d\theta d\varphi$  on the sphere. When the marginal densities are found,  $p$  (as expected) is separable, and we find that a  $(r_1, r_2)$  pair of uniform random numbers can be transformed to a direction by

$$(\theta, \varphi) = \left( \arccos \left( (1 - r_1)^{1/(n+1)} \right), 2\pi r_2 \right).$$

## 2.8 II.7 NONUNIFORM RANDOM POINTS VIA WARPING

Typically, we want a directional  $(\theta, \varphi)$  pair to be with respect to some unit vector  $\mathbf{y}$  (as opposed to the  $z$  axis). To do this we can first convert the angles to a unit vector  $\mathbf{a}$ :

$$\mathbf{a} = (\cos \varphi \sin \theta, \sin \varphi \sin \theta, \cos \theta).$$

We can then transform  $\mathbf{a}$  to be an  $\mathbf{a}'$  with respect to  $\psi$  by multiplying  $\mathbf{a}$  by a rotation matrix  $\mathbf{R}$  ( $\mathbf{a}' = \mathbf{R}\mathbf{a}$ ). This rotation matrix is simple to write down:

$$\mathbf{R} = \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix}$$

where  $\mathbf{u} = (u_x, u_y, u_z)$ ,  $\mathbf{v} = (v_x, v_y, v_z)$ ,  $\mathbf{w} = (w_x, w_y, w_z)$ , form a basis (an orthonormal set of unit vectors where  $\mathbf{u} = \mathbf{v} \times \mathbf{w}$ ,  $\mathbf{v} = \mathbf{w} \times \mathbf{u}$ , and  $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ ) with the constraint that  $\mathbf{w}$  is aligned with  $\psi$ :

$$\mathbf{w} = \frac{\psi}{|\psi|}.$$

To get  $\mathbf{u}$  and  $\mathbf{v}$ , we need to find a vector  $\mathbf{t}$  that is not colinear with  $\mathbf{w}$ . To do this simply set  $\mathbf{t}$  equal to  $\mathbf{w}$  and change the smallest magnitude component of  $\mathbf{t}$  to one. The  $\mathbf{u}$  and  $\mathbf{v}$  follow easily:

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{|\mathbf{t} \times \mathbf{w}|},$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

This family of techniques is very useful for many sampling applications. Unfortunately, some sampling spaces (e.g., the surface of a dodecahedron) are not naturally dealt with using the methods in this gem. Special purpose or, as a last resort, rejection techniques are then called for.


See also G1, 438.





## II.8

### 2.9 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND



Ronald N. Goldman  
*Rice University*  
*Houston, Texas*

#### 2.10 Introduction

Cross product is one of the gods' great gifts to mankind. It has many applications in mathematics, physics, engineering, and, of course, computer graphics. Normal vectors, rotations, curl, angular momentum, torque, and magnetic fields all make use of the cross product.

Given two linearly independent vectors  $u$  and  $v$  in three dimensions, their cross product is the vector  $u \times v$  perpendicular to the plane of  $u$  and  $v$ , oriented according to the right-hand rule, with length equal to  $|u||v|\sin\theta$ , where  $\theta$  is the angle between  $u$  and  $v$ . In rectangular coordinates, the cross product can be computed from the simple determinant formula

$$u \times v = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix}.$$

Equivalently,

$$u \times v = (u_2v_3 - u_3v_2, u_3v_1 - u_1v_3, u_1v_2 - u_2v_1).$$

At first glance, cross product seems to be an artifact of three dimensions. In three dimensions the normal direction to the plane determined by two vectors

is unique up to sign, but in four dimensions there are a whole plane of vectors normal to any given plane. Thus, it is unclear how to define the cross product of two vectors in four dimensions. What then

## 2.11 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND

is the analogue of the cross product in four dimensions and beyond? The goal of this gem is to answer this question.

### 2.12 Tensor Product

There is a way to look at the cross product that is more instructive than the standard definition and that generalizes readily to four dimensions and beyond. To understand this approach, we need to begin with the notion of the tensor product of two vectors  $u, v$ .

The tensor product  $u \otimes v$  is defined to be the square matrix

$$u \otimes v = u^t * v,$$

where the superscript  $t$  denotes transpose and  $*$  denotes matrix multiplication. Equivalently,

$$(u \otimes v)_{ij} = (u_i, v_j).$$

Notice that for any vector  $w$ ,

$$w(u \otimes v) = (w \cdot u)v.$$

Thus, the tensor product is closely related to the dot product.

Like dot product, the tensor product makes sense for two vectors of arbitrary dimension. Indeed, the tensor product shares many of the algebraic properties of the dot product. However, unlike the dot product, the tensor product is not commutative. That is, in general,

$$u \otimes v \neq v \otimes u \quad \text{because } u_i v_j \neq u_j v_i$$

Applications of the tensor product of two vectors to computer graphics are given in Goldman (1990, 1991).

## 2.13 Wedge Product

The wedge product of two vectors  $u$  and  $v$  measures the noncommutativity of their tensor product. Thus, the wedge product  $u \wedge v$  is the square

## 2.14 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND

matrix defined by

$$u \wedge v = u \otimes v - v \otimes u.$$

Equivalently,

$$(u \wedge v)_{ij} = (u_i v_j - u_j v_i).$$

Like the tensor product, the wedge product is defined for two vectors of arbitrary dimension. Notice, too, that the wedge product shares many properties with the cross product. For example, it is easy to verify directly from the definition of the wedge product as the difference of two tensor products that:

$$u \wedge u = 0,$$

$$u \wedge v = -v \wedge u$$

(anticommutative),

$$u * (v \wedge w) \neq (u \wedge v) * w^t$$

(nonassociative),

$$u \wedge cv = c(u \wedge v) = (cu) \wedge v,$$

$$u \wedge (v + w) = u \wedge v + u \wedge w$$

(distributive),

$$\mathbf{u} * (\mathbf{v} \wedge \mathbf{w}) + \mathbf{v} * (\mathbf{w} \wedge \mathbf{u}) + \mathbf{w} * (\mathbf{u} \wedge \mathbf{v}) = 0$$

(Jacobi identity),

$$\mathbf{r} * (\mathbf{u} \wedge \mathbf{v}) * \mathbf{s}^t = (\mathbf{r} \cdot \mathbf{u})(\mathbf{s} \cdot \mathbf{v}) - (\mathbf{r} \cdot \mathbf{v})(\mathbf{s} \cdot \mathbf{u})$$

(Lagrange identity).

The wedge product also shares some other important properties with the cross product. The defining characteristics of the cross product are captured by the formulas

$$\mathbf{u} \cdot (\mathbf{u} \times \mathbf{v}) = \mathbf{v} \cdot (\mathbf{u} \times \mathbf{v}) = 0,$$

$$|\mathbf{u} \times \mathbf{v}| = |\mathbf{u}|^2 |\mathbf{v}|^2 \sin^2 \Theta.$$

By the Lagrange identity, the wedge product satisfies the analogous

## 2.15 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND

identities:

$$\mathbf{u} * (\mathbf{u} \wedge \mathbf{v}) * \mathbf{u}^t = \mathbf{v} * (\mathbf{u} \wedge \mathbf{v}) * \mathbf{v}^t = 0,$$

$$\mathbf{u} * (\mathbf{u} \wedge \mathbf{v}) * \mathbf{v}^t = (\mathbf{u} \cdot \mathbf{u})(\mathbf{v} \cdot \mathbf{v}) - (\mathbf{u} \cdot \mathbf{v})^2 = |\mathbf{u}|^2 |\mathbf{v}|^2 \sin^2 \Theta$$

A variant of the last identity can be generated by defining the norm of a matrix  $M$  to be

$$|M|^2 = \frac{1}{2} \sum_{ij} (M_{ij})^2.$$

Then by direct computation it is easy to verify that

$$|\mathbf{u} \wedge \mathbf{v}|^2 = (\mathbf{u} \cdot \mathbf{u})(\mathbf{v} \cdot \mathbf{v}) - (\mathbf{u} \cdot \mathbf{v})^2 = |\mathbf{u}|^2 |\mathbf{v}|^2 \sin^2 \Theta.$$

In addition, the cross product identity

$$(\mathbf{u} \times \mathbf{v}) \times \mathbf{w} = (\mathbf{w} \cdot \mathbf{u})\mathbf{v} - (\mathbf{w} \cdot \mathbf{v})\mathbf{u}$$

has the wedge product analogue

$$\mathbf{w} \cdot (\mathbf{u} \wedge \mathbf{v}) = (\mathbf{w} \cdot \mathbf{u})\mathbf{v} - (\mathbf{w} \cdot \mathbf{v})\mathbf{u}.$$

The cross product can be used to test for vectors perpendicular to the plane of  $\mathbf{u}$  and  $\mathbf{v}$  because

$$\mathbf{w} \times (\mathbf{u} \times \mathbf{v}) = 0 \Leftrightarrow \mathbf{w} \perp \mathbf{u}, \mathbf{v}.$$

Similarly, the wedge product recognizes vectors perpendicular to the plane determined by  $\mathbf{u}$  and  $\mathbf{v}$  because by (1),

$$\mathbf{w} * (\mathbf{u} \wedge \mathbf{v}) = 0 \Leftrightarrow (\mathbf{w} \cdot \mathbf{u}) = (\mathbf{w} \cdot \mathbf{v}) = 0 \Leftrightarrow \mathbf{w} \perp \mathbf{u}, \mathbf{v}.$$

Moreover, in three dimensions,

$$u \wedge v = \begin{vmatrix} 0 & u_1 v_2 - u_2 v_1 & u_1 v_3 - u_3 v_1 \\ u_2 v_1 - u_1 v_2 & 0 & u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 & u_3 v_2 - u_2 v_3 & 0 \end{vmatrix}$$

## 2.16 II.8 CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND

Thus, in three dimensions the entries of the wedge product matrix  $u \wedge v$  are, up to sign, the same as the components of the cross product vector  $\mathbf{u} \times \mathbf{v}$ . This observation explains why wedge product and cross product share so many algebraic properties.

In three dimensions we are really very lucky. The matrix  $u \wedge v$  is antisymmetric so, up to sign, it has only three unique entries. This property allows us to identify the matrix  $u \wedge v$  with the vector  $\mathbf{u} \times \mathbf{v}$ . Nevertheless, there is something very peculiar about the vector  $\mathbf{u} \times \mathbf{v}$ . If  $\mathbf{u}$  and  $\mathbf{v}$  are orthogonal unit vectors, then the vectors  $\mathbf{u}, \mathbf{v}, \mathbf{u} \times \mathbf{v}$  form a right-handed coordinate system. But if  $\mathbf{M}$  is the linear transformation that mirrors vectors in the  $\mathbf{u}, \mathbf{v}$  plane, then  $\{\mathbf{u} \cdot \mathbf{M}, \mathbf{v} \cdot \mathbf{M}, (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{M}\} = \{\mathbf{u}, \mathbf{v}, -\mathbf{u} \times \mathbf{v}\}$  forms a left-handed coordinate system. Thus,  $(\mathbf{u} \cdot \mathbf{M}) \times (\mathbf{v} \cdot \mathbf{M}) \neq (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{M}$ , so  $\mathbf{u} \times \mathbf{v}$  does not really transform as a vector. This anomaly

should alert us to the fact that cross product is not really a true vector. In fact, cross product transforms more like a tensor than a vector.

In higher dimensions we are not nearly so lucky. For example, in four dimensions the antisymmetric matrix  $u \wedge v$  has, up to sign, six, not four, distinct entries. Thus, the matrix  $u \wedge v$  cannot be identified with a four-dimensional vector. In  $n$  dimensions, the antisymmetric matrix  $u \wedge v$  has  $n(n-1)/2$  unique entries. But  $n(n-1)/2 \neq n$  unless  $n = 0, 3$ . Thus, only in three dimensions can we identify the wedge product of two vectors with a vector of the same dimension. In general, the wedge product is an antisymmetric 2-tensor. This antisymmetric tensor shares many of the important algebraic properties of the cross product, and thus it is a natural generalization of the cross product to four dimensions and beyond.

## 2.17 Acknowledgment

I would like to thank Joe Warren for pointing out that the formula for the length of the cross product  $u \times v$  has a direct analogue in the formula for the norm of the wedge product  $u \wedge v$ .

## II.9

# FACE-CONNECTED LINE SEGMENT GENERATION IN AN $n$ -DIMENSIONAL SPACE

Didier Badouel and Charles A. Wüthrich  
University of Toronto  
Toronto, Ontario, Canada

In the early days of Computer Graphics, straight line rasterization was developed to render segments onto the raster plane. Later, three-dimensional segment discretization had to be developed to keep track of the path of a ray in the object space. These algorithms generate a connected sequence that represents the segment in the discrete space; moreover, they define a path in which the directions are uniformly distributed. An extension to higher-dimensional spaces is suited for

applications ranging from line generation in a time-space coordinate system to the incremental generation of a discrete simultaneous linear interpolation of any number of variables.

This gem presents an algorithm that generates a face-connected line segment in discrete  $n$ -dimensional spaces. In two dimensions, the algorithm introduced below coincides with any classical 4-connected straight line drawing algorithm. Among all discrete segments joining two points, this algorithm produces one in which the directions are uniformly distributed. A definition of uniform distribution is given below.

Consider an  $n$ -dimensional lattice, or hyperlattice, i.e., the set of all points  $P = (p_0, p_1, \dots, p_{n-1})$  of  $\mathbf{Z}^n$  : Neighbourhood relations can be defined between the Voronoi hypervoxel associated with each point of the hyperlattice. In fact, only voxels having a hyperface in common, i.e., corresponding to hyperlattice points having  $n - 1$  coordinates in common, will be considered here as neighbours. In a two-dimensional lattice, such neighbourhood relation is the well-known 4 - connection, while in the three-dimensional space it leads to 6-connection. The neighbourhood relations among the hyperlattice points introduce a rasterization procedure for curves into the hyperlattice: A rasterization of a curve is in fact a path of neighbouring lattice points.

Consider two hyperlattice points  $P = (p_0, p_1, \dots, p_{n-1})$  and  $Q = (q_0, q_1, \dots, q_{n-1})$ . Let  $n_i = |q_i - p_i|$  Then a face-connected shortest path between  $P$  and  $Q$  requires  $m = \sum n_i$  steps. The hyperline points are the points of coordinates  $x_i = (q_i - p_i)t + p_i$ , with  $t \in [0, 1]$ . The parameter  $t$  introduces an ordering on the points of the straight line. Consider the straight line points  $P_{i,h_i}$  obtained for  $t = h_i/n_i$ , where  $h_j = 1, \dots, n_i$ , and order them in increasing order of their corresponding parameter value. Whenever an ambiguity occurs, and two parameter values  $h_j/n_i$  and  $h_j/n_j$  coincide, an arbitrary order has to be chosen. In other words, the segment  $PQ$  is subdivided into  $n_i$  parts for each dimension  $i$ , and the points obtained on the straight line segment are ordered by increasing values of the parameter  $t$ . When two subdivision points coincide, the one corresponding to the smaller dimension is considered to precede the other one.

The resulting set is a finite ordered set of the segment points  $P_{i,h_i}$ , which can be renamed as  $A_0, A_1, \dots, A_{m-1}$ . Consider the finite path built by taking the sequence of directions  $\{a_k\}_{k=0, \dots, m-1}$ , such that each direction  $a_k$  corresponds to the point  $A_k = P_{a_k, 1}$ , for some  $l$ . Such a path is said to be uniformly distributed with respect to the directions that constitute it. It is clear that in such a path the occurrences of the different directions that have to appear in it are as evenly spaced as possible in the chain. Moreover, if we follow the previously defined path from the point  $P$ , the point  $Q$  shall be reached.

Whenever the hyperface-connected rasterization onto the  $n$ -dimensional hyperlattice of a straight line segment joining two hyperlattice points is computed, the result is a hyperface-connected path joining the two points. This path is uniformly distributed among all directions. A simplified version of the routing algorithm can be therefore summarized as follows. For each direction  $i$ , an integer counter  $d_i$  is used. In order to generate the straight line between the two points  $P$  and  $Q$ , the values of  $n_i$  are computed. Their least common multiple  $l = \text{LCM}(n_i)$  is evaluated,<sup>1</sup> and the values of  $n_i'' = l/n_i$  are computed. To obtain only integer

<sup>1</sup> In fact, either a low-complexity method in  $O(n \log k)$  based on a table lookup or a simple common multiple can be used here. computations, the values of  $n_i' = 2n''$  are used. The cells  $d_i$  are initialized to the value  $n_i'/2$ . This initialization has to be made, otherwise the path generated corresponds to another rasterization scheme. At each step,  $n_i'$  is added to the  $d_i$  with the smallest value, and the  $i$ th signed direction is generated. The generation procedure is repeated until all  $d_i$  have reached the value  $2l + n_i''$ . which is equivalent to  $\forall i, d_i \geq 2l$ .