图形瑰宝

第三册

david kirk 著

郭飞 译

2022年

Contents

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。

Algorithm 1: Line(x1,y1,x2,y2)

input : 由点(x1,y1)和点(x2,y2)构成的线段
output: 将线段绘制到位图上

```
// 从(x1, y1)到(x2, y2)在第一个八边形内画一条线；所有变量都是
   整数
dx ← x2 - x1
dy ← y2 - y1
e ← 2 * dy - dx
for i← 1,i⩽ dx, i← i+1 do
    WritePixel (x1,y1) // 在图上显示(x1,y1)
    while e⩾ 0 do
        y1 ← y1 + 1
        e ← e - 2 * dx
    x1 ← x1 + 1
    e ← e + 2 * dy
```

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x1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。x1和y1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，x1是目标线上的坐标，y1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。

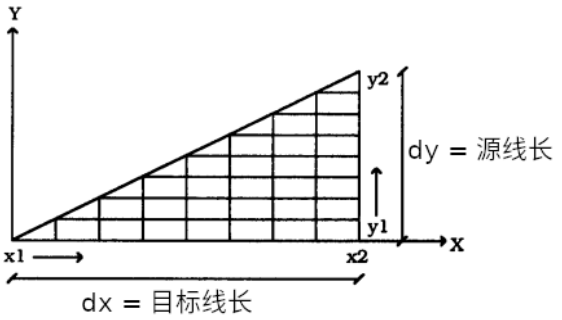

Figure 1.1.1: 源线与目标线

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$x2 < x1$和$y2 < y1$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。

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。

Algorithm 2: Stretch(x1, y1, x2, y2, yr, yw)

input :

output:

```
// 从(x1, y1)到(x2, y2)在第一个八边形内画一条线; 所有变量都是
   整数
// 将源线(y1   y2)延伸到目标线(x1   x2)。
// 源线和目标线都是水平的
// yr =从其中读取像素的水平线
// yw =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 ← abs (x2 - x1)
dy ← abs (y2 - y1)
sx ← sign (x2 - x1)
sy ← sign (y2 - y1)
e ← 2 * dy - dx
dx2 ← 2 * dx
dy ← 2 * dy
for i← 1,i⩽ dx, i← i+1 do
    color ← ReadPixel (x, y)
    SetColor (Color)
    WritePixel(x1, yw)
    while e⩾ 0 do
        y1 ← y1 + sy
        e ← e - dx2
    x1 ← x1 + 1
    e ← e + 2 * dy
```

- 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(image_width*image_height* (filter_width + filter_height)比简单的二维过滤O(image_width*image_height*filter_width*filter_height)要低得多。

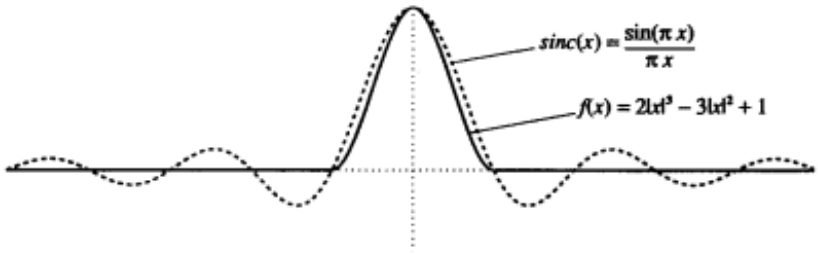

Figure 1.2.1: *sinc*示例

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 Schafer, 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。

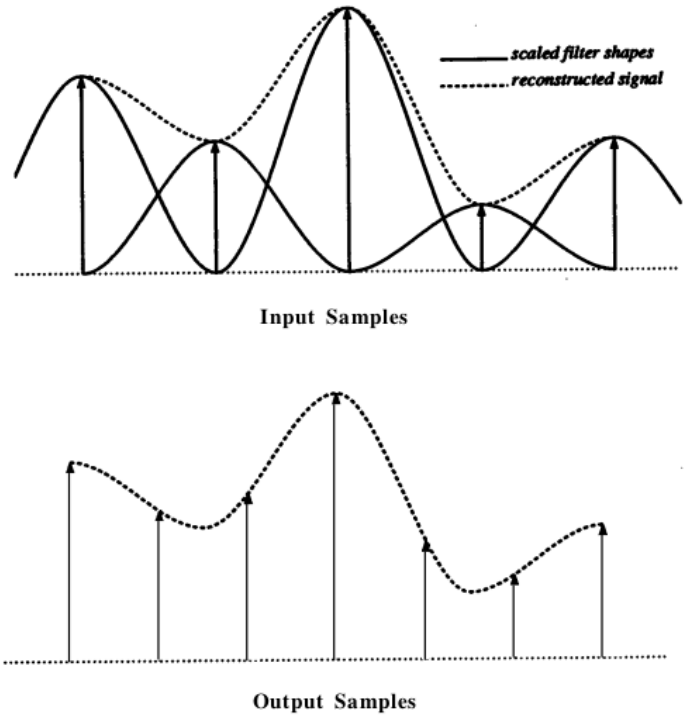

Figure 1.2.2: 滤波示例

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。

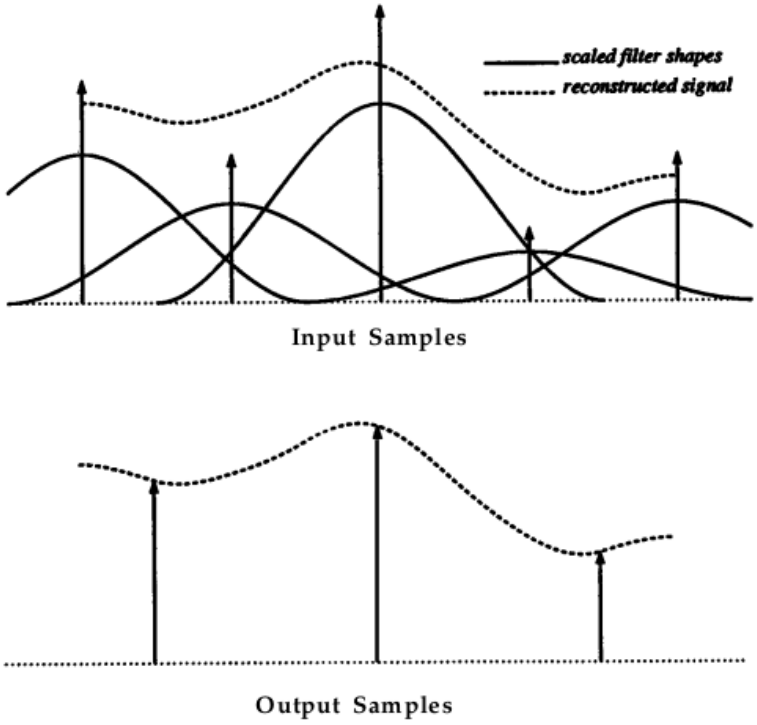

Figure 1.2.3: 滤波示例

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。

Algorithm 3: calculate_contributions(destination)

input :

output:

scale ← dst_size/src_size;

center ← destination/scale;

if *scale < 1.0* **then**
    width ← filter_width/scale;
    fscale ← 1.0/scale;
else width ← filter_width;

fscale ← 1.0;

;

left ← `floor` (center - width);

right ← `ceiling` (center + width);

for *source ← left,source ← source + 1, source ⩽ right* **do**
    weight ← `filter` ((center source)/fscale)/fscale;
    `add_contributor` (destination, source, weight);

Algorithm 4: scale_row(destination_row, source_row)

input :

output:

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