带阴影的三角形

在上一章中，我们开发了一种算法来绘制一个用纯色填充的三角形。本章的目标是绘制一个*带阴影的*三角形，即一个填充了颜色渐变的三角形。

定义我们的问题

我们想用一种颜色的不同*色调*填充三角形。它将如图 8-1 所示。

我们需要一个更正式的定义来定义我们试图绘制的内容。为此，我们将分配一个实际值h到每个顶点，表示顶点处颜色的强度。h在$[0.0, 1.0]$范围，其中0.0表示可能最暗的阴影（即黑色）和1.0表示尽可能亮的阴影（即原始颜色，而不是白色！）

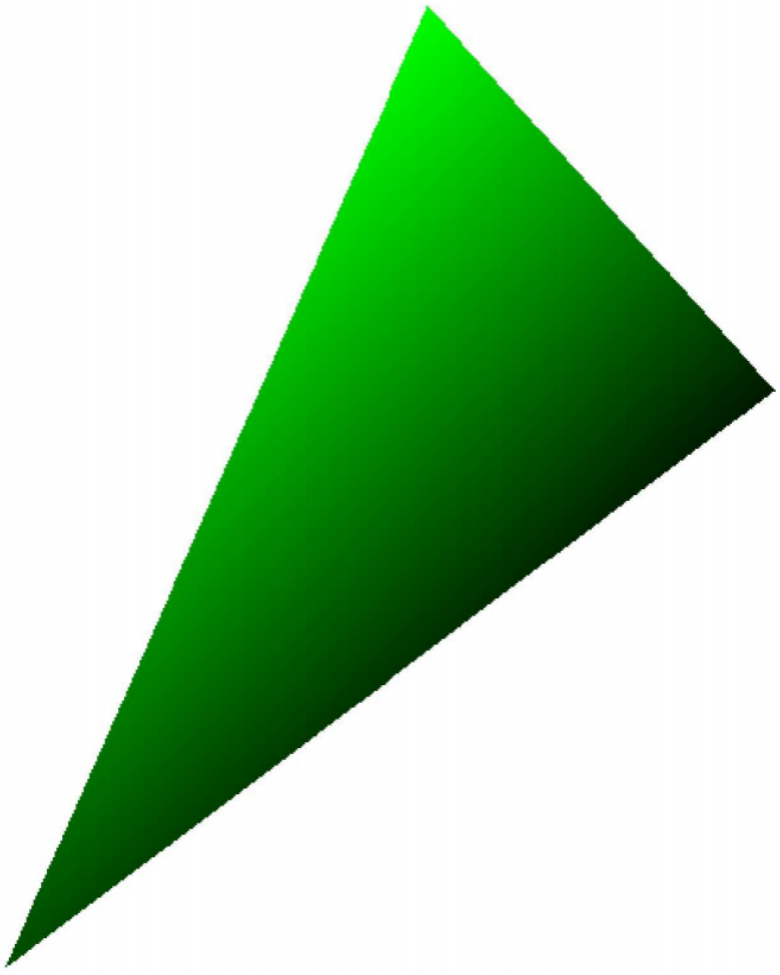

图 8-1：带阴影的三角形

在给定三角形基色的情况下计算像素的确切色调C以及该像素处的强度h，我们将按通道乘法：$C_h = (R_C \cdot h,\ G_C \cdot h,\ B_C \cdot h) = 0.0$产生纯黑色，$h = 1.0$产生原始颜色$C$和$h = 0.5$产生的颜色是原始颜色的一半亮度。

计算边缘着色

为了绘制一个阴影三角形，我们需要做的就是计算一个值h对于三角形的每个像素，计算相应的颜色阴影，并绘制像素。容易！

然而，在这一点上，我们只知道h对于三角形顶点，因为我们选择了它们。我们如何计算值h对于三角形的其余部分？

让我们从三角形的边缘开始。考虑边缘AB，我们知道h_A和h_B，在M线会发生什么，在AB的中点，由于我们希望强度从A到B平滑地变化，值h_M必须介于两者之间h_A和h_B.因为M在AB的中间，为什么不选择h_M在中间h_A和h_B——也就是说，他们的平均水平？

更正式地说，我们有一个函数$h = f(P)$给出每一分P强度值h;我们知道它的价值一个和B,$h\ (A) = h_A$和$h\ (B) = h_B$。我们希望这个函数是平滑的。因为我们对其他一无所知$h = f(P)$，我们可以选择任何与我们*所知*兼容的函数，例如线性函数(图 8-2)。

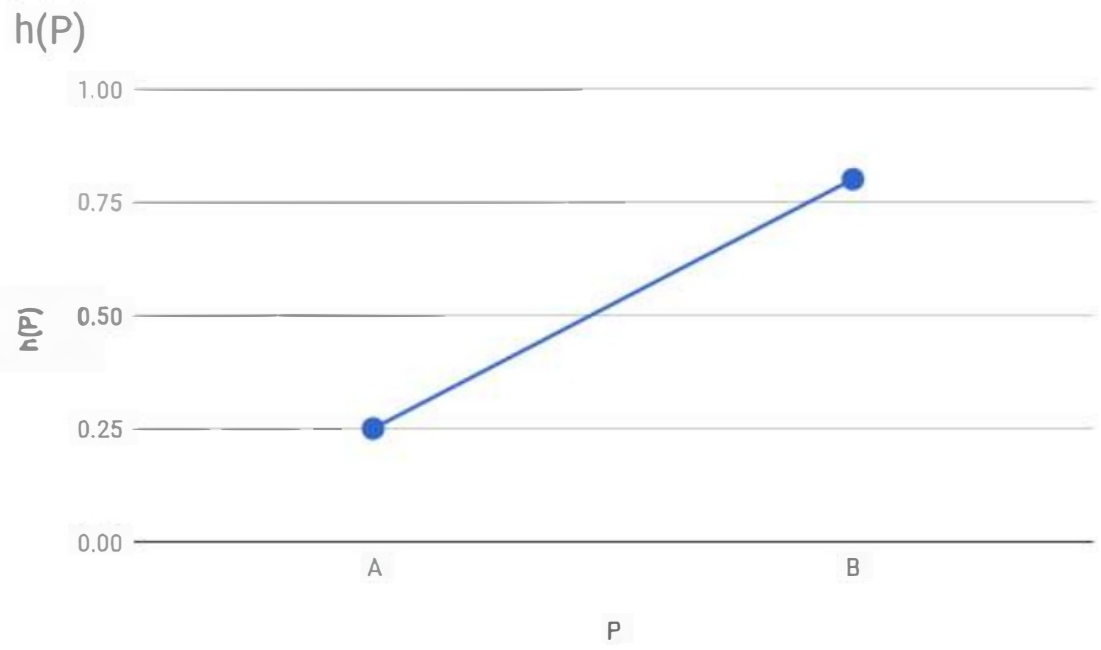

图 8-2：线性函数 h（P），与我们所知道的 h（A）和 h（B 兼容）

这与上一章的情况非常相似：我们有一个线性函数$x = f(y)$，我们知道这个函数在三角形顶点处的值，并且我们想计算x沿着它的两侧。我们可以计算以下值h沿着三角形的边以非常相似的方式，使用 y 作为自变量（我们知道的值），使用 h 作为因变量（我们想要的值）：`Interpolate`

```
x01 = Interpolate(y0, x0, y1, x1)
h01 = Interpolate(y0, h0, y1, h1)

x12 = Interpolate(y1, x1, y2, x2)
h12 = Interpolate(y1, h1, y2, h2)

x02 = Interpolate(y0, x0, y2, x2)
h02 = Interpolate(y0, h0, y2, h2)
```

接下来，我们连接了x“短”边的数组，然后确定哪个是和是，哪个是。同样，我们可以在这里为x02x012x_leftx_right$h$向量。

但是，我们将始终使用x值来确定哪一侧为左侧，哪一侧为右侧，以及h价值观只会“随波逐流”。x和h是屏幕上实际点的属性，因此我们无法自由混合搭配左侧和右侧值。

我们可以按如下方式编码：

```
// Concatenate the short sides
remove_last(x01)
x012 = x01 + x12

remove_last(h01)
h012 = h01 + h12

// Determine which is left and which is right
m = floor(x012.length / 2)
if x02[m] < x012[m] {
    x_left = x02
    h_left = h02

    x_right = x012
    h_right = h012
} else {
    x_left = x012
    h_left = h012

    x_right = x02
    h_right = h02
}
```

这与上一章（示例 7-1）中代码的相关部分非常相似，不同之处在于每次我们对向量执行某些操作时，我们都会对相应的向量执行相同的操作。xh

计算内部遮阳

最后一步是绘制实际的水平段。对于每个段，我们知道和，如上一章;现在我们也知道和.但是这一次我们不能只是从左到右迭代并用基色绘制每个像素：我们需要计算一个x_leftx_righth_lefth_right$h$对于线段的每个*像素*。

同样，我们可以假设h随 线性变化x，并用于计算这些值。在这种情况下，自变量为Interpolate$x$，它从值变为我们正在着色的特定水平段的值;因变量为x_leftx_right$h$，以及该段的 和 are 和 的相应值：x_leftx_righth_lefth_right

```
x_left_this_y = x_left[y - y0]
h_left_this_y = h_left[y - y0]

x_right_this_y = x_right[y - y0]
```

```
h_right_this_y = h_right[y - y0]

h_segment = Interpolate(x_left_this_y, h_left_this_y,
                        x_right_this_y, h_right_this_y)
```

或者，以更紧凑的方式表示：

```
h_segment = Interpolate(x_left[y - y0], h_left[y - y0],
                        x_right[y - y0], h_right[y - y0])
```

现在只需计算每个像素的颜色并绘制它！示例 8-1 显示了 的完整伪代码。
DrawShadedTriangle

```
DrawShadedTriangle (P0, P1, P2, color) {
    ❶// Sort the points so that y0 <= y1 <= y2
     if y1 < y0 { swap(P1, P0) }
     if y2 < y0 { swap(P2, P0) }
     if y2 < y1 { swap(P2, P1) }

     // Compute the x coordinates and h values of the triangle edges
     x01 = Interpolate(y0, x0, y1, x1)
     h01 = Interpolate(y0, h0, y1, h1)

     x12 = Interpolate(y1, x1, y2, x2)
     h12 = Interpolate(y1, h1, y2, h2)

     x02 = Interpolate(y0, x0, y2, x2)
     h02 = Interpolate(y0, h0, y2, h2)

     // Concatenate the short sides
     remove_last(x01)
     x012 = x01 + x12

     remove_last(h01)
     h012 = h01 + h12

     // Determine which is left and which is right
     m = floor(x012.length / 2)
     if x02[m] < x012[m] {
         x_left = x02
         h_left = h02

         x_right = x012
         h_right = h012
     } else {
         x_left = x012
         h_left = h012

         x_right = x02
         h_right = h02
     }

     // Draw the horizontal segments
    ❷for y = y0 to y2 {
         x_l = x_left[y - y0]
         x_r = x_right[y - y0]

        ❸h_segment = Interpolate(x_l, h_left[y - y0], x_r, h_right[y - y0])
         for x = x_l to x_r {
            ❹shaded_color = color * h_segment[x - x_l]
             canvas.PutPixel(x, y, shaded_color)
         }
     }
}
```

示例 8-1：绘制阴影三角形的函数

此函数的伪代码与上一章中开发的函数的伪代码非常相似（示例 7-1）。在水平段循环 ❷ 之前，我们操作x向量和h矢量以类似的方式，如上所述。在循环中，我们有一个额外的调用 ❸ 来计算`Interpolate`h当前水平线段中每个像素的值。最后，在内部循环中，我们使用h计算每个像素的颜色❹。

请注意，我们像以前一样对三角形顶点进行排序❶。但是，我们现在考虑这些顶点及其属性，例如强度值h，成为一个不可分割的整体;也就是说，交换两个顶点的坐标也必须交换其属性。

源代码和现场演示>>

总结

在本章中，我们扩展了上一章中开发的三角形绘制代码，以支持平滑着色的三角形。请注意，我们仍然可以使用它来绘制单色三角形，方法是使用 1.0 作为h对于所有三个顶点。

这个算法背后的想法实际上比看起来更通用。事实上h是强度值对算法的“形状”没有影响;我们只在最后，当我们要调用 .这意味着我们可以使用此算法来计算三角形顶点的任何*属性*的值，对于三角形的每个像素，只要我们假设该值在屏幕上线性变化。`PutPixel`

在接下来的章节中，我们确实会使用这个算法来改善三角形的视觉外观。出于这个原因，在继续之前，最好确保您真正了解此算法。

然而，在下一章中，我们走了一个小弯路。掌握了在 2D 画布上绘制三角形后，我们将注意力转向第三维度。