双线性插值算法

图像的双线性插值放大算法中，目标图像中新创造的象素值，是由源图像位置在它附近的 2*2 区域 4 个邻近象素的值通过加权平均计算得出的。双线性内插值算法放大后的图像质量较高，不会出现像素值不连续的的情况。然而次算法具有低通滤波器的性质，使高频分量受损，所以可能会使图像轮廓在一定程度上变得模糊。

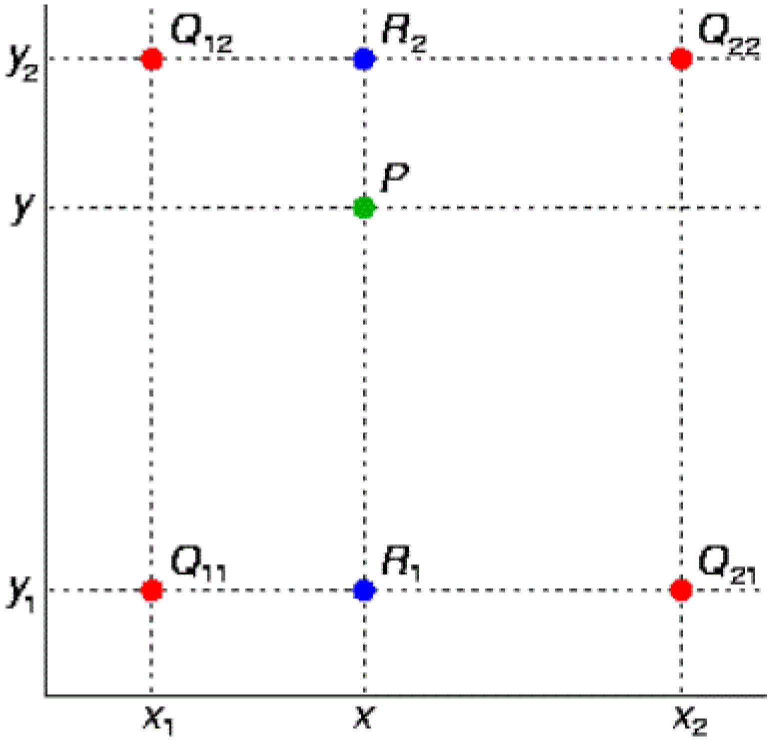


图 1

X 方向的线性插值

对于标准的双线性差值算法，X 方向的线性插值：

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \quad \text{where} \quad R_1 = (x, y_1),$$

[通用 1]

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \quad \text{where} \quad R_2 = (x, y_2).$$

[通用 2]

具体到我们所实现的算法中，我们使 Q11、Q12、Q21、Q22 为光栅上相邻的四点，即 P 只能落于这四点其中一点上。Δ_{col}是当前像素离像素所属区域原点的

水平距离，比如图 2，各种不同的颜色代表一个区域，区域原点为区域左上角的像素。

$$\delta(R_2) = \left(Color(Q_{22}) - Color(Q_{12})\right) \cdot \Delta_{col} + Color(Q_{12}) \cdot 256 \quad (1)$$

$$\delta(R_1) = \left(Color(Q_{21}) - Color(Q_{11})\right) \cdot \Delta_{col} + Color(Q_{11}) \cdot 256 \quad (2)$$

其中：$\Delta_{col} = (DestColNumber \cdot ((SrcWidth \ll 8)/DestWidth))\&255$，Color(X)表示点 X 的颜色，具体算法使用的是 24 位真彩色格式。

Y 方向的线性插值

做完 X 方向的插值后再做 Y 方向的插值，对于一般情况，有：

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2).$$

[通用 3]

而我们的具体算法中，Y 方向的线性插值方法如(3)所示。Δ_{row}是当前像素离像素所属区域原点的垂直距离，比如图 2，各种不同的颜色代表一个区域，区域原点为区域左上角的像素。

$$Color(P) = \left(\delta(R_2) \cdot 256 + \left(\delta(R_2) - \delta(R_1)\right) \cdot \Delta_{row}\right) \gg 16 \quad (3)$$

其中：$\Delta_{row} = (DestRowNumber \cdot ((SrcHeight \ll 8)/DestHeight))\&255$，由于前面为了便于计算左移了 16 位，因此最后需要右移 16 位保持匹配。

算法描述

```
for (目标图像第一行的像素++)
{
    // 源图像上 Q12, Q22, Q11, Q21 的选取见下一节
    获取源图像 Q12, Q22, Q11, Q21 的颜色;

    // X 方向的插值
    δ(R2) = (Color(Q22) - Color(Q12)) * Δcol + Color(Q12) * 256;
    δ(R1) = (Color(Q21) - Color(Q11)) * Δcol + Color(Q11) * 256;

    // 保存 δ (R1)到一个临时数组，因为下一行的δ (R2)等于这一行的δ (R1)
    temp[i++] = δ(R1);
```

```
        // Y 方向的插值
        Color(P) = (δ(R2) * 256 + (δ(R2) - δ(R1)) *Δ_row ) >> 16;

        将 P 输出到目标位图中。
}

for (目标图像第二行到最末行)
{
        for (行上的像素++)
        {
                // 源图像上 Q12, Q22, Q11, Q21 的选取见下一节
                获取源图像 Q12, Q22, Q11, Q21 的颜色;

                // X 方向的插值
                δ (R2) = temp[i++]; // 下一行的δ (R2)等于上一行的δ (R1)
                δ(R1) = (Color(Q21) - Color(Q11)) *Δ_col + Color(Q11) * 256;

                // 保存 δ (R1)到一个临时数组，因为下一行的δ (R2)等于这一行的δ (R1)
                temp[i++] = δ(R1);

                // Y 方向的插值
                Color(P) = (δ(R2) * 256 + (δ(R2) - δ(R1)) * Δ_row ) >> 16;

                将 P 输出到目标位图中。
        }
}
```

算法中 Q12, Q22, Q11, Q21 的选取

我们以放大两倍为例，说明选取 Q12, Q22, Q11, Q21 的过程。源图像 3*3 区域放大为目标区域 6*6 区域。设以下为目标图像：

A	A	B	B		
A	A	B	B		
		C	C		
		C	C		
				D	D
				D	D

图 2

目标图像 A 像素区域对应的 Q21，Q22，Q11，Q12，以红色区域为原点向右下方扩展的 2*2 区域。

Q21	Q22	
Q11	Q12	

图 3

目标图像 B 像素区域对应的 Q21，Q22，Q11，Q12，以蓝色区域为原点向右下方扩展的 2*2 区域。

	Q21	Q22
	Q11	Q12

图 4

目标图像 C 像素区域对应的 Q21，Q22，Q11，Q12，以绿色区域为原点向右下方扩展的 2*2 区域。

	Q21	Q22
	Q11	Q12

图 5

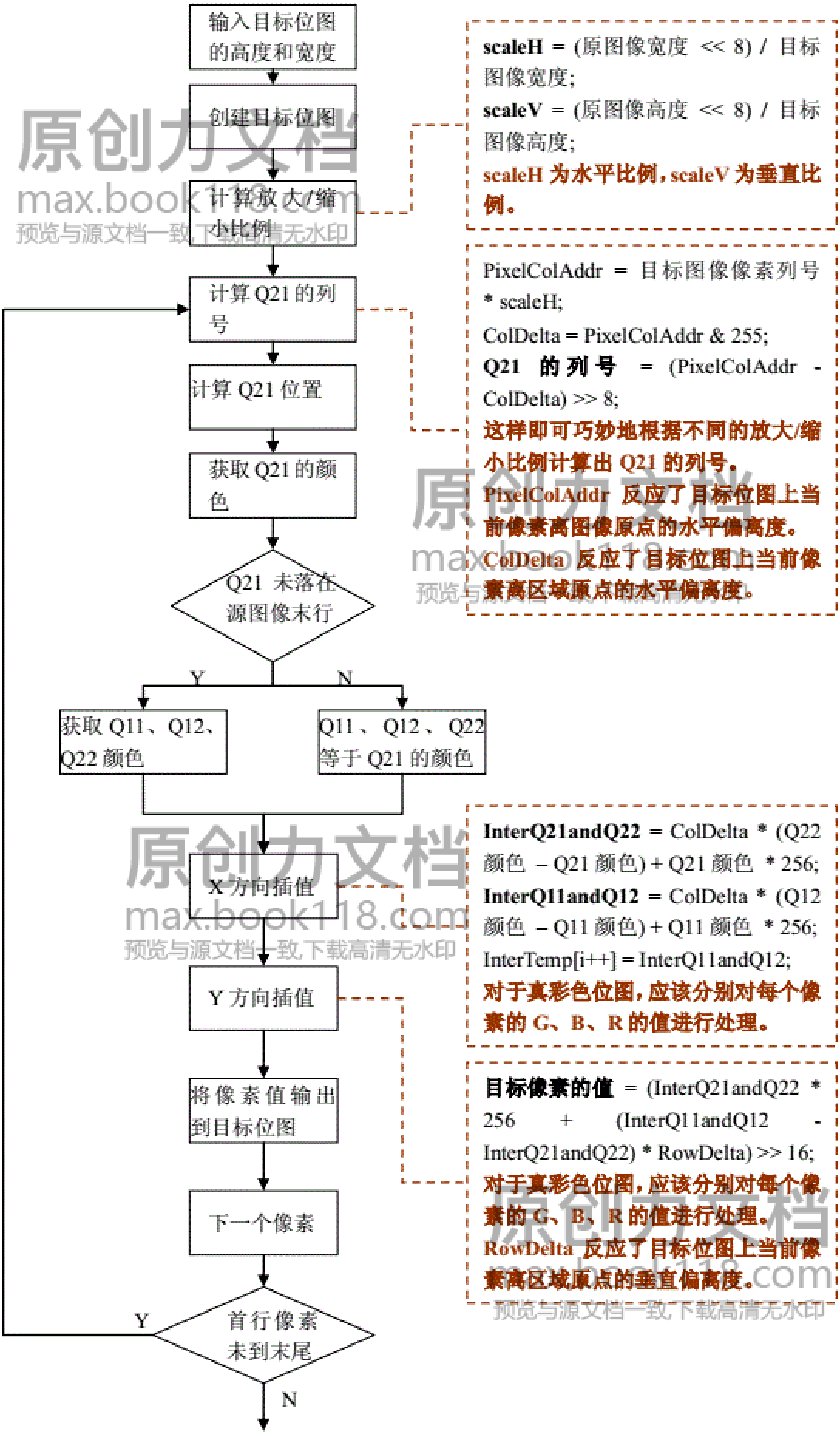
目标图像 D 像素区域对应的 Q21，Q22，Q11，Q12，目标图像处于最后两行的边界情况，将 Q21，Q22，Q11，Q12 这四个点的值设为一样。

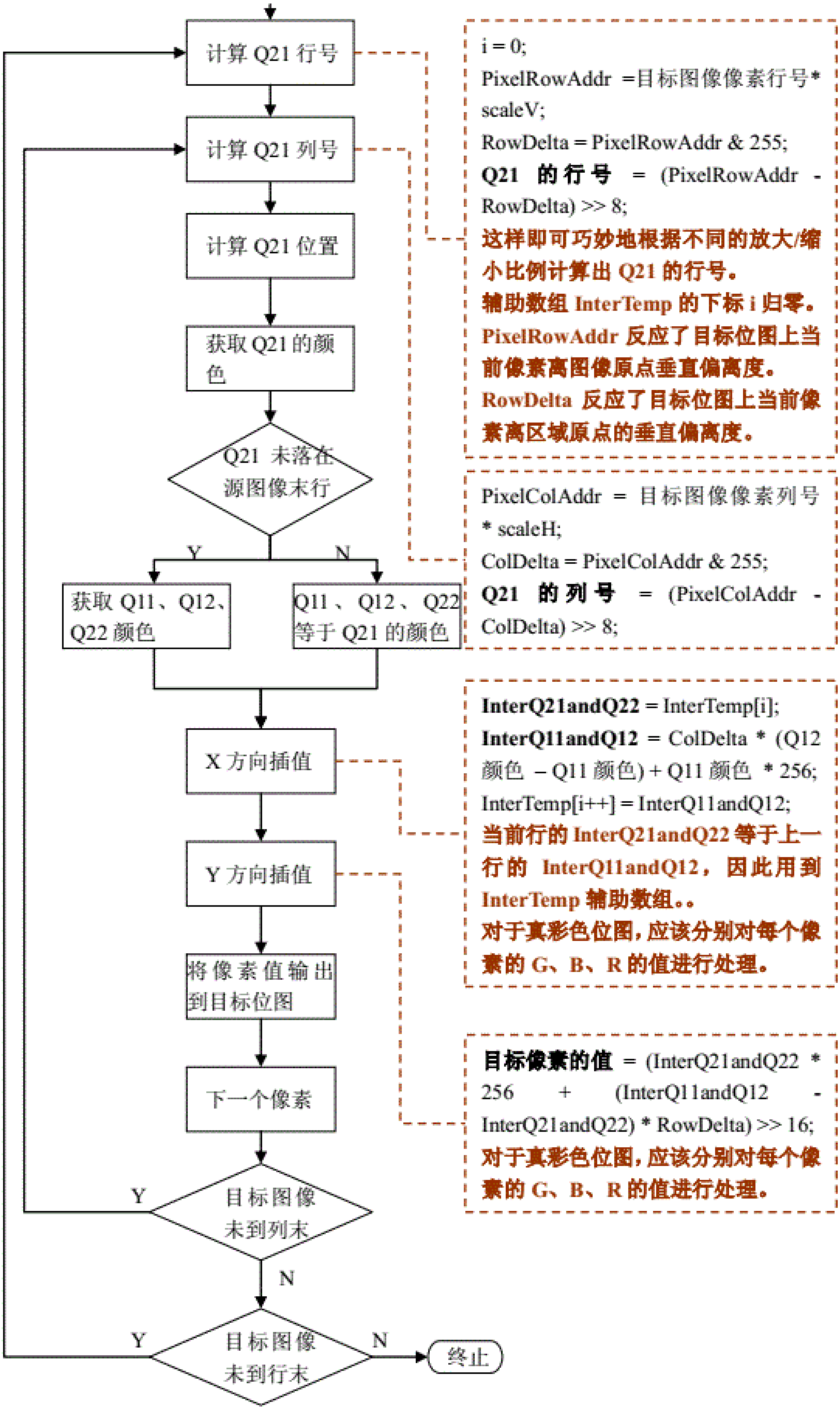
		Q11=Q12=Q22=Q21

图 6

程序流程图

流程图右边虚线框中为相关过程的注解。


输入目标位图的高度和宽度
创建目标位图
计算放大/缩小比例
计算 Q21 的列号
计算 Q21 位置
获取 Q21 的颜色
Q21 未落在源图像末行
Y
N
获取 Q11、Q12、Q22 颜色
Q11、Q12、Q22 等于 Q21 的颜色
X 方向插值
Y 方向插值
将像素值输出到目标位图
下一个像素
首行像素未到末尾
Y
N
scaleH = (原图像宽度 << 8) / 目标图像宽度;
scaleV = (原图像高度 << 8) / 目标图像高度;
scaleH 为水平比例，scaleV 为垂直比例。
PixelColAddr = 目标图像像素列号 * scaleH;
ColDelta = PixelColAddr & 255;
Q21 的列号 = (PixelColAddr - ColDelta) >> 8;
这样即可巧妙地根据不同的放大/缩小比例计算出 Q21 的列号。
PixelColAddr 反应了目标位图上当前像素离图像原点的水平偏离度。
ColDelta 反应了目标位图上当前像素离区域原点的水平偏离度。
InterQ21andQ22 = ColDelta * (Q22 颜色 – Q21 颜色) + Q21 颜色 * 256;
InterQ11andQ12 = ColDelta * (Q12 颜色 – Q11 颜色) + Q11 颜色 * 256;
InterTemp[i++] = InterQ11andQ12;
对于真彩色位图，应该分别对每个像素的 G、B、R 的值进行处理。
目标像素的值 = (InterQ21andQ22 * 256 + (InterQ11andQ12 - InterQ21andQ22) * RowDelta) >> 16;
对于真彩色位图，应该分别对每个像素的 G、B、R 的值进行处理。
RowDelta 反应了目标位图上当前像素离区域原点的垂直偏离度。



计算 Q21 行号
计算 Q21 列号
计算 Q21 位置
获取 Q21 的颜色
Q21 未落在源图像末行
Y
N
获取 Q11、Q12、Q22 颜色
Q11、Q12、Q22 等于 Q21 的颜色
X 方向插值
Y 方向插值
将像素值输出到目标位图
下一个像素
目标图像未到列末
Y
N
目标图像未到行末
Y
N
终止
i = 0;
PixelRowAddr =目标图像像素行号* scaleV;
RowDelta = PixelRowAddr & 255;
Q21 的行号 = (PixelRowAddr - RowDelta) >> 8;
这样即可巧妙地根据不同的放大/缩小比例计算出 Q21 的行号。
辅助数组 InterTemp 的下标 i 归零。
PixelRowAddr 反应了目标位图上当前像素离图像原点垂直偏离度。
RowDelta 反应了目标位图上当前像素离区域原点的垂直偏离度。
PixelColAddr = 目标图像像素列号 * scaleH;
ColDelta = PixelColAddr & 255;
Q21 的列号 = (PixelColAddr - ColDelta) >> 8;
InterQ21andQ22 = InterTemp[i];
InterQ11andQ12 = ColDelta * (Q12 颜色 – Q11 颜色) + Q11 颜色 * 256;
InterTemp[i++] = InterQ11andQ12;
当前行的 InterQ21andQ22 等于上一行的 InterQ11andQ12，因此用到 InterTemp 辅助数组。。
对于真彩色位图，应该分别对每个像素的 G、B、R 的值进行处理。
目标像素的值 = (InterQ21andQ22 * 256 + (InterQ11andQ12 - InterQ21andQ22) * RowDelta) >> 16;
对于真彩色位图，应该分别对每个像素的 G、B、R 的值进行处理。

双线性插值图像放大/缩小算法

```
void ResizeWorkingBitmap(tWorkBMP *a, tWorkBMP *b, WORD bx, WORD by)
{
    unsigned int PtAR = 0, PtBR = 0, PtCR = 0, PtDR = 0, PixelValueR = 0;
    unsigned int PtAG = 0, PtBG = 0, PtCG = 0, PtDG = 0, PixelValueG = 0;
    unsigned int PtAB = 0, PtBB = 0, PtCB = 0, PtDB = 0, PixelValueB = 0;
    register unsigned SpixelColNum = 0, SpixelRowNum = 0, DestCol = 0, DestRow = 0;
    unsigned int SpixelColAddr = 0, SpixelRowAddr = 0;
    unsigned int ColDelta = 0, RowDelta = 0, scaleV = 0, scaleH = 0;
    unsigned int ContribAandBR = 0, ContribCandDR = 0;
    unsigned int ContribAandBG = 0, ContribCandDG = 0;
    unsigned int ContribAandBB = 0, ContribCandDB = 0;
    unsigned int ContribTem[2048 * 3];// Max width is 2048
    int i = 0;

    CreateWorkingBitmap(bx, by, b);

    // Calculation of zoom proportion
    scaleH = (a->x << 8) / bx;
    scaleV = (a->y << 8) / by;

    // First line of destination image
    for (DestCol = 0; DestCol < bx; DestCol++)
    {
        // Horizontal distance between origin of image and current pixel
        SpixelColAddr = DestCol * scaleH;
        // Horizontal distance between A and current pixel
        ColDelta = SpixelColAddr & 255;
        // Column number of A
        SpixelColNum = (SpixelColAddr - ColDelta) >> 8;

        // Get color of A
        PtAB = a->b[3 * SpixelRowNum * a->x + 3 * SpixelColNum];
        PtAG = a->b[3 * SpixelRowNum * a->x + 3 * SpixelColNum + 1];
        PtAR = a->b[3 * SpixelRowNum * a->x + 3 * SpixelColNum + 2];

        // Get color of B, C, D
        if ((SpixelColNum + 1) < a->x)
        {
            PtBB = a->b[3 * SpixelRowNum * a->x + 3 * (SpixelColNum + 1)];
            PtBG = a->b[3 * SpixelRowNum * a->x + 3 * (SpixelColNum+1) + 1];
            PtBR = a->b[3 * SpixelRowNum * a->x + 3 * (SpixelColNum+1) + 2];
```

```
            PtCB = a->b[3 * (SpixelRowNum+1) * a->x + 3 * SpixelColNum];
            PtCG = a->b[3 * (SpixelRowNum+1) * a->x + 3 * SpixelColNum + 1];
            PtCR = a->b[3 * (SpixelRowNum+1) * a->x + 3 * SpixelColNum + 2];

            PtDB = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * (SpixelColNum + 1)];
            PtDG = a->b[3 * (SpixelRowNum+1) * a->x + 3 * (SpixelColNum+1) + 1];
            PtDR = a->b[3 * (SpixelRowNum+1) * a->x + 3 * (SpixelColNum+1) + 2];
        }
        else
        {
            PtBB = PtCB = PtDB = PtAB;
            PtBG = PtCG = PtDG = PtAG;
            PtBR = PtCR = PtDR = PtAR;
        }

        // X-direction interpolation of blue
        ContribAandBB = ColDelta * (PtBB - PtAB) + PtAB * 256;
        ContribCandDB = ColDelta * (PtDB - PtCB) + PtCB * 256;
        ContribTem[i++] = ContribCandDB;
        // Y-direction interpolation of blue
        PixelValueB = (ContribAandBB * 256 + (ContribCandDB - ContribAandBB) * RowDelta) >>
16;

        // X-direction interpolation of green
        ContribAandBG = ColDelta * (PtBG - PtAG) + PtAG * 256;
        ContribCandDG = ColDelta * (PtDG - PtCG) + PtCG * 256;
        ContribTem[i++] = ContribCandDG;
        // Y-direction interpolation of green
        PixelValueG = (ContribAandBG * 256 + (ContribCandDG - ContribAandBG) * RowDelta) >>
16;

        // X-direction interpolation of red
        ContribAandBR = ColDelta * (PtBR - PtAR) + PtAR * 256;
        ContribCandDR = ColDelta * (PtDR - PtCR) + PtCR * 256;
        ContribTem[i++] = ContribCandDR;
        // Y-direction interpolation of red
        PixelValueR = (ContribAandBR * 256 + (ContribCandDR - ContribAandBR) * RowDelta) >>
16;

        // Output current pixel to destination image
        b->b[3 * DestRow * bx + 3 * DestCol] = PixelValueB;
        b->b[3 * DestRow * bx + 3 * DestCol + 1] = PixelValueG;
        b->b[3 * DestRow * bx + 3 * DestCol + 2] = PixelValueR;
    }
```

```
    // Other line of destination image
    for (DestRow = 1; DestRow < by; DestRow++)
    {
        i = 0;
        // Vertical distance between origin of image and current pixel
        SpixelRowAddr = DestRow * scaleV;
        // Vertical distance between A and current pixel
        RowDelta = SpixelRowAddr & 255;
        // Row number of A
        SpixelRowNum = (SpixelRowAddr - RowDelta) >> 8;

        for (DestCol = 0; DestCol < bx; DestCol++)
        {
            SpixelColAddr = DestCol * scaleH;
            ColDelta = SpixelColAddr & 255;
            SpixelColNum = (SpixelColAddr - ColDelta) >> 8;
            PtAB = a->b[3 * SpixelRowNum * a->x + 3 * SpixelColNum];
            PtAG = a->b[3 * SpixelRowNum * a->x + 3 * SpixelColNum + 1];
            PtAR = a->b[3 * SpixelRowNum * a->x + 3 * SpixelColNum + 2];

            if (((SpixelColNum+1)<a->x)&&((SpixelRowNum+1)<a->y))
            {
                PtCB = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * SpixelColNum];
                PtCG = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * SpixelColNum + 1];
                PtCR = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * SpixelColNum + 2];

                PtDB = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * (SpixelColNum + 1)];
                PtDG = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * (SpixelColNum+1) + 1];
                PtDR = a->b[3 * (SpixelRowNum + 1) * a->x + 3 * (SpixelColNum+1) + 2];
            }
            else
            {
                PtBB = PtCB = PtDB = PtAB;
                PtBG = PtCG = PtDG = PtAG;
                PtBR = PtCR = PtDR = PtAR;
            }

            ContribAandBB = ContribTem[i];
            ContribCandDB = ColDelta * (PtDB - PtCB) + PtCB * 256;
            ContribTem[i++] = ContribCandDB;
            PixelValueB = (ContribAandBB * 256 + (ContribCandDB - ContribAandBB) * RowDelta) >>
16;
```

```
                ContribAandBG = ContribTem[i];
                ContribCandDG = ColDelta * (PtDG - PtCG) + PtCG * 256;
                ContribTem[i++] = ContribCandDG;
                PixelValueG = (ContribAandBG * 256 + (ContribCandDG - ContribAandBG) * RowDelta) >>
16;

                ContribAandBR = ContribTem[i];
                ContribCandDR = ColDelta*(PtDR-PtCR)+PtCR*256;
                ContribTem[i++] = ContribCandDR;
                PixelValueR = (ContribAandBR * 256 + (ContribCandDR - ContribAandBR) * RowDelta) >>
16;

                b->b[3 * DestRow * bx + 3 * DestCol] = PixelValueB;
                b->b[3 * DestRow * bx + 3 * DestCol + 1] = PixelValueG;
                b->b[3 * DestRow * bx + 3 * DestCol + 2] = PixelValueR;
           }
      }
}
```