这篇文档介绍了Bolt引擎的扩展元对象高斯模糊对象(GaussianBlurObject)在渲染时使用的两种算法的原理, 实现和优化.

[高斯模糊的基本原理](#_高斯模糊的基本原理)

[对象中使用的两种模糊算法](#_对象中使用的两种模糊算法)

[FIR算法的原理和实现](#_FIR算法的原理和实现)

[FIR算法的优化](#_FIR算法的优化)

[IIR算法的原理, 实现和优化](#_IIR算法的原理,_实现和优化)

[默认混合策略](#_默认混合策略)

[以后想要加入的扩展和增强](#_以后想要加入的扩展和增强)

[参考文献](#_参考文献)

## 高斯模糊的基本原理

图像的模糊就是, 把图像中每一个像素的颜色值替换成它周围像素颜色值的加权求和, 这样每一个像素都受到周边像素的影响, 界限变得模糊.

高斯模糊就是对图像使用高斯滤波器滤波.

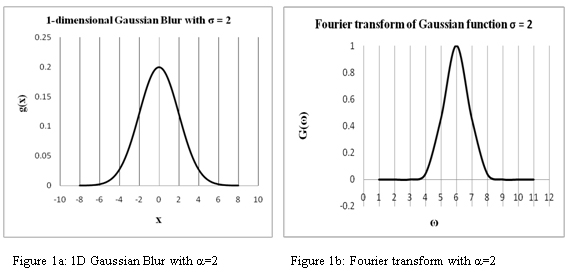
高斯滤波器广泛应用于去噪, 模糊和边界探测. 这是一种减弱图片中高频噪音(可以理解为把图片中清晰的线条减弱)的低通滤波器. 一维高斯函数定义为:

http://software.intel.com/sites/default/files/article/181134/29355.jpg

其中http://software.intel.com/sites/default/files/article/181134/29356.jpg 是高斯分布的标准方差.

高斯函数的傅里叶变换也是高斯函数:

http://software.intel.com/sites/default/files/article/181134/29357.jpg



二维高斯函数定义为:

http://software.intel.com/sites/default/files/article/181134/29359.jpg

这一篇通俗易懂地介绍了高斯模糊的原理.

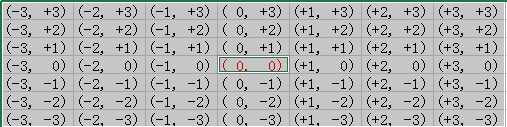
<http://www.ruanyifeng.com/blog/2012/11/gaussian_blur.html>

高斯模糊按照高斯函数来分配周边各点的权重 (近的点权重大, 远的点权重小), 我们把加权中”权”的矩阵叫做高斯核, 高斯函数中的方差sigma会影响权重矩阵(高斯核), 我们把这个sigma叫做模糊系数, 模糊系数越大, 高斯函数的”山峰”越平坦, 像素点受到周边像素的影响越大, 图片越模糊.

我们用高斯半径来确定“周围的点”中的”周围”的范围, 半径为r, 周围的点的个数是(2\*r+1)\*(2\*r+1) ~ 4\*r\*r, 显然半径越大, 计算量越大. 因为3倍模糊系数之外的点对当前像素的影响权重很小(从图1中可以看出), 所以一般取3倍模糊系数作为模糊半径.

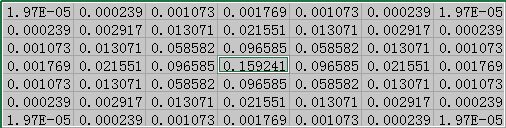
高斯函数是连续的, 而在实际应用中我们需要离散化的高斯核, 而且权重和要等于1来保证图像明度不变. 为此先取半径内距中心点距离为整数的高斯函数值, 得到离散的高斯矩阵, 再对矩阵元素归一化处理使其和为1. 用G(x, y)来表示给定方差的二维高斯函数, 取高斯半径为1, 最终得到的高斯核矩阵形如:

举个例子, 模糊系数是1, 那么模糊半径取3比较合适, 我们需要计算每个像素周围(3\*2+1)^2 = 49个点来计算模糊后的颜色值:



上图(0, 0)是当前要计算的点, 要用到包括它一共49个点的颜色.

模糊系数为1, 模糊半径为3的权重矩阵, 就是方差为1的二维高斯分布离散化成7x7的二维矩阵



上图是模糊系数=1, 半径=3的高斯核矩阵.

对于每一个像素, 我们需要把它周围的颜色值乘以对应的矩阵元素加在一起, 作为这个像素模糊后的颜色值. 这就是基本的高斯模糊算法: 计算高斯核矩阵, 加权求和. 矩阵只需要计算一次, 对每一个像素的周边像素加权求和占据了计算的大部分, 这也是各种高斯模糊算法力争优化的部分.

## 对象中使用的两种模糊算法

###### FIR算法的原理和实现

如上的基本原理中介绍的模糊算法, 也叫做有限脉冲回应滤波器(Finite Impulse Response). 对于长宽为height, width, 模糊半径为r的高斯模糊图像, 需要做乘法的次数为(2\*r+1)\*(2\*r+1)\*width\*height ~ O(4\*width\*height\*r\*r), 这还没有把每个像素分别需要对三个颜色通道分别计算考虑进去. 模糊半径有可能随着模糊系数取的很大, 即使使用SIMD指令集加速, 这样平方增长的计算量也不可取.

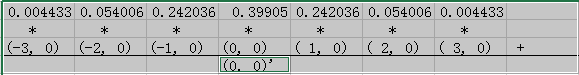


(模糊半径为3的模糊效果不过尔尔, 计算量已经相当于普通图像处理过49遍了!)

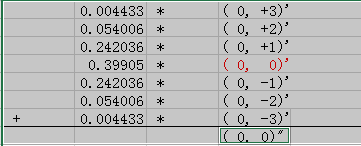
所幸高斯函数有个叫”线性可分特性”的特征, 二维矩阵变换可以通过两次一维矩阵变换来得到, 算法效率提高了一个数量级. 具体来讲就是: 在水平方向, 对每个像素点取左右半径的像素加权求和, 对于水平方向的处理结果, 在垂直方向对每个像素点取上下半径的像素加权求和.

还是以模糊系数=1, 模糊半径=3举个例子:

先计算水平方向, 半径是3的一维高斯核矩阵是一个长度7的向量, 对每个像素取左右各三个像素加权求和



得到的(0, 0)’ 是中间变量, 以(0, 0)’ 为输入, 对每个像素取上下各三个像素加权求和:



得到最终像素的颜色值.

用公式来表达的话, 先计算一维正态分布离散化成一个长度为7的数组, 用F(x)来表示方差为1 的正态分布函数, 这个数组是:

,

用P(i, j)来表示当前计算的模糊像素(x,y)的周边像素(x+i, y+j)的颜色值, 则先在水平方向上计算(也叫滤波)

计算结果放在临时内存中. 对所有计算结果在垂直方向上进行滤波计算, P(i, j)’表示临时内存中的模糊像素(x, y)’的周边像素(x+i, y+j)’.

把计算结果(x,y)’’写回原图缓存地址中, 模糊计算结束.

对于边界条件, 使用的处理方法是越界的像素当做边界像素来处理. 这样做的弊端是边界像素权重变大. 其他的算法实现有把另一侧的像素镜面映射到边界以外的.

总结一下, 一维的FIR算法就是: 计算一维高斯核, 水平方向对每个点周边点加权求和, 结果放在临时内存中, 垂直方向对临时内存中的每个点周边点加权求和, 结果写回原图缓存. Bolt的扩展元对象高斯模糊对象属性gaussiantype=”FIR”时, 用的就是这种算法. 可以看到它的计算量是随着模糊半径, 也就是模糊系数线性增长的, 计算量是 width \* height \* (2\*r+1) \*2 ~ O(4\*width\*height\*r)

###### FIR算法的优化

无论算法最后优化成什么样, 使用SIMD指令集加速是一定的. 编译器对C++代码的优化已经做得很好, 但是没有到对向量操作的程度, 这对需要分别对三个颜色通道(也许还有Alpha通道吧?)计算的图像处理来说是很重要的. 我们首先要做的, 是在使用指令集加速之前, 尽可能优化C++实现的算法.

* 边界条件处理

在需要模糊的像素接近边界(和边界距离小于模糊半径)时, 会访问到越界的点. 最初的处理方法是每一个要计算的索引和边界比较, 越界了就取边界值, 需要和上下界各比较一次, 而循环内部的分支是很影响性能的, 即使分支预测也会出现两次错误的预测( < 0 为true变false, 大于width-1从false变true). 所以应当尽可能去掉分支.

对于扫描线左边radius个像素, 会越界的范围是[i-radius, i – (i-1)], 这些点要取低边界

对于扫描线中间(width – 2\*radius)的一段, 是不可能越界的;

对于扫描线右边radius个像素, 会越界的范围是[i+(width-i), i+radius], 这些点要取上边界

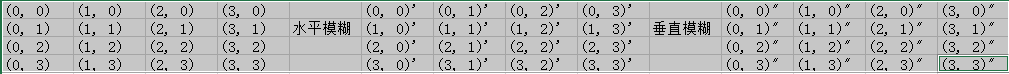
* 跨越扫描线访问内存

图像在内存中的缓存是水平方向存储的, 即水平方向相邻的像素在内存中也是相邻的, 这是我们访问[i-radius, i+radius]这些像素都不会在内存中跳跃太远;

如果我们把水平模糊的结果按照原来的顺序放在临时内存中, 算法访问每一个像素点上下方向的邻居, 它们在内存中动辄跳跃bmp.ScanLineLength(原图片宽)那么多个像素, 而且要跳跃2\*r次, 每个像素! 这样对性能有影响.

所以水平模糊的计算结果, 我们对矩阵转置一下放在临时内存中, 使得在垂直方向的模糊计算时, 它们在内存中是相邻的. 垂直模糊的计算结果, 我们直接写入原图像位图缓存.

这样做在把水平模糊的计算结果写入临时内存时, 我们需要跳跃图片高长度的内存访问临时内存; 在把垂直模糊的计算结果写回原图像位图缓存时, 我们需要跳跃位图扫描线(scanLineLength)长度的内存访问原图像位图缓存. 长距离的内存跳跃会发生(2\*Width\*Height)次. 比不做转置的跳跃次数要少而且和模糊半径无关.



转置矩阵的伪代码实现:

input: lpPixelBufferInitial[height][width] (原图buffer首地址)

input: width, height (原图宽, 高)

input: weight[2\*r+1] (一维高斯核, 半径为r)

局部变量: lpLocalBuffer[width][height] (暂存水平模糊结果)

for (int row = 0; row < height; row++)

{

for (int column = 0; column < width; column++)

{

float\_4 localSum = 0; // 一个四维向量表示一个像素点argb分量

for (int j = -r; j <= +r; j++)

{

localSum += lpPixelBufferInitial[row][column + j] \* weight[j + r];

}

lpLocalBuffer[column][row] = localSum;

}

}

for (int column = 0; column < width; column++)

{

for (int row = 0; row < height; row++)

{

float\_4 localSum = 0;

for (int j = -r; j <= +r; j++)

{

localSum += lpLocalBuffer[column][row+j]\*weight[j+r];

}

lpPixelBufferInitial[row][column] = localSum;

}

}

* 利用多核CPU

对位图水平/垂直线分别做模糊还有一个好处就是, 线和线之间的计算是独立的(同一个方向计算时), 这样我们就可以充分利用CPU, 同时对多条线进行处理.

优化的具体实现就是看看CPU有几个处理器, 有几个就开几个线程, 同时计算.

* 丢失精度的处理

做了以上优化之后, 着手进行指令集优化. 使用128位浮点向量指令集SSE直接对浮点向量进行操作的效果不尽人意. 遂转向MMX指令集. MMX指令集只能处理整形向量, 所以要先对算法进行丢失精度处理. 把浮点高斯核round成加起来等于256的整形向量, 和颜色值相乘后再右移8位(相当于除以256), 这样中间的计算过程可以简化成16位的整形向量乘法和加法, 使用64位寄存器MMX同时计算三个颜色通道顺带上Alpha.

简单来说就是, 在对高斯核归一化的时候, 让它的和为256, 且每一项都是整数;

使用64位整形向量寄存器进行水平和垂直方向的模糊;

写回原位图缓存时, 把结果的每个颜色分量和Alpha分量右移8位.

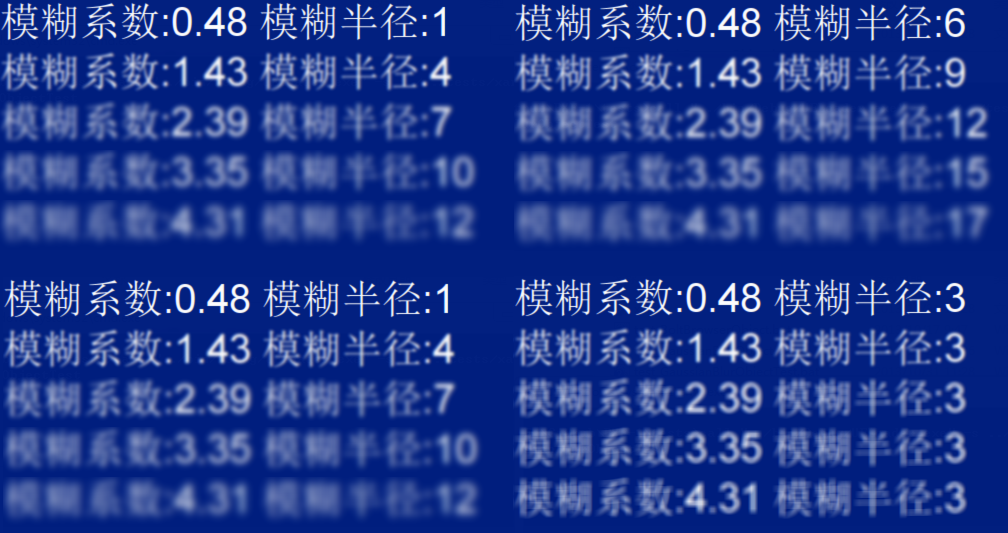
最终性能:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 模糊系数= 0.33  模糊半径= 1 | 模糊系数=1  模糊半径=3 | 模糊系数=8  模糊半径=24 |
| FIR算法, MMX 实现 | <15 ms | < 15 ms | <50 ms |

###### IIR算法的原理, 实现和优化

* IIR高斯滤波器的介绍

因为对每个输出像素要做的计算随着http://software.intel.com/sites/default/files/article/181134/29356.jpg按比例增长, 高斯滤波器的性能随着半径增长越来越差. 但是我们在图一的对比中已经看到了, 固定模糊半径的后果就是模糊效果差强人意. 我们要做的是寻找一个计算量和模糊半径无关的算法. 初步想来这是不可能的, 要计算某个像素的模糊值就非用到它前后半径那么多个像素原来的值不可, 所以我们要留着而且访问每个像素前面r(r是模糊半径)个像素. 而实际上我们计算第i个像素的时候, 它会包含之前r个像素的信息, 这些信息已经在计算第i-1个像素的时候用过了, 我们有没有可能复用这些信息, 只利用新的i-1和旧的i计算新的i呢? 看起来好像很难, 因为高斯核矩阵相邻的系数之间并不是线性的关系, 简单的线性计算 i’ = a\*i + b\*(i-1)’ + c (i+1)可达不到这样的效果, 更何况即便我们复用了前面r个点在i-1中的信息, 也没办法预支后面r个点在i上的影响(因为还没有算到). 但是利用之前两个像素的信息, 和一些高斯函数的数学特性, 我们可以得到前面所有像素对当前像素的影响, 最起码可以拟合这种影响; 而把一行像素从左往右, 和从右往左各自计算一遍, 就可以把当前像素左边的和右边的像素们对当前像素的影响都囊括在内了.

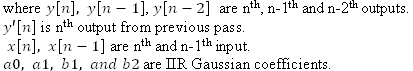


图一: 右下是固定模糊半径的后果

IIR高斯滤波器(Infinite Impulse Response)及其衍生算法[[1][2][3][4]](#_参考文献)通过递归地求解和方差http://software.intel.com/sites/default/files/article/181134/29356.jpg(即模糊系数)无关的差分方程, 做到了这一点.

高斯模糊扩展元对象的IIR算法中使用的方程形如:

http://software.intel.com/sites/default/files/article/181134/29367.jpg



(关于y’[n], 在从左往右计算时是0, 就是不加, 在从右往左计算时是从左往右的计算结果)

IIR高斯滤波器在水平和垂直方向上按行或者按列计算每一个像素. 它是可分离滤波器, 即水平方向处理和垂直方向处理的顺序对结果没有影响.

Intel在[这里](http://software.intel.com/en-us/articles/iir-gaussian-blur-filter-implementation-using-intel-advanced-vector-extensions)给出了一个使用AVX指令集的实现,它的实现性能更好, 因为AVX一次处理8个浮点数, 每个线程里可以两行一起算. 但是不是所有的处理器都支持AVX指令集, 也不是所有编译器都支持伪指令函数, 所以我们使用SSE指令集来对IIR算法进行优化.

* 高斯模糊对象的IIR滤波器实现

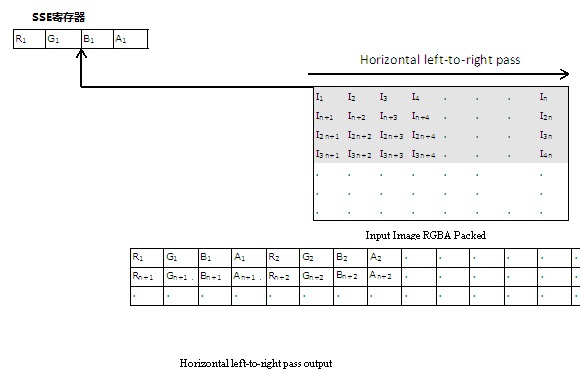
IIR高斯模糊滤波器通过两个有序通道对每个像素应用两次等式(1):

* 水平通道: 对输入图像的每一行从左到右, 然后从右到左进行计算. 从左往右计算的输出结果被加到从右往左计算的输出结果中.
* 垂直通道: 通常垂直通道对水平通道的输出结果先从上往下, 再从下往上计算每个像素. 按列访问输入图像会带来大量缓存块从而影响滤波器性能. 为避免如此, 水平通道的输出要先做转置再写入输出缓存中. 这样垂直通道的操作就和水平通道的差不多了, (对内存的访问)先从左往右计算, 再从右往左计算. 垂直通道的输出在写回模糊图像缓存之前再做一次转置以恢复图像方向.

扩展元对象在实现绘制时拿到的XL\_BITMAP是32位RGBA像素的. 因为取整的误差在每个通道上是逐像素的累计的(每一个像素的计算都会用到前面那个像素的计算结果), 会影响图像质量, 所以选择使用SSE寄存器操作32位单精度浮点数, 没有中间的压缩和解压步骤(每个通道的计算结果作为浮点数直接放在缓存中, 而不是压缩成4个字节的像素), IIR滤波器可以生成高质量的模糊图像. 32位的RGBA像素还免去了在寄存器中对齐的麻烦. 如果XL\_BITMAP有可能是无Alpha通道的24位像素或者16位压缩像素, 要先通过shuffle和shift来保证寄存器内对齐.

* 水平通道

128位SSE寄存器一次可以处理四个单精度浮点数, 正好容纳RGBA四个分量. 高斯模糊对象的IIR算法每次读入一个像素(32位), 使用pmovzxbd从原图像内存解压到128位SSE寄存器中, 使用cvtdq2ps转化为单精度浮点数.



因为需要频繁第访问高斯模糊系数, 所以放在寄存器中. SSE没有广播指令, 使用movups加载IIR高斯模糊系数并用shufps把系数放在寄存器的每一位. 然后使用向量乘法和加法指令对每一个像素进行计算.

每一行水平通道从左向右计算的输出结果作为单精度浮点数使用存储指令直接存储在一小条临时内存中(图像宽\*4\*float size), 根据图像大小, 这条临时缓存有可能放在CPU缓存中, 这样从右往左计算时访问起来就很快了.

pmovzxbd xmm0, [esi];

cvtdq2ps xmm0, xmm0; xmm0: id[i]

mulps xmm0, xmm1; xmm0: id[i] \* a0;

mulps xmm2, xmm3; xmm2: id[i-1] \* a1;

addps xmm0, xmm2; xmm0: id[i] \* a0 + id[i-1] \* a1;

movups xmm2, xmm4; xmm2 = oTemp[i-1]

mulps xmm2, xmm5; xmm2 = oTemp[i-1] \* b1;

subps xmm0, xmm2; xmm0: id[i] \* a0 + id[i-1]\*a1 - oTemp[i-1]\*b1;

movups xmm2, xmm6; xmm2 = oTemp[i-2];

mulps xmm2, xmm7; xmm2 = oTemp[i-1] \* b2;

subps xmm0, xmm2; xmm0: result: id[i] \* a0 + id[i-1]\*a1 - oTemp[i-1]\*b1 - oTemp[i-2]\*b2;

movups [edx], xmm0; oTemp = result;

pmovzxbd xmm2, [esi];

cvtdq2ps xmm2, xmm2; xmm2 = id[i]; for next round: xmm2 = id[i-1];

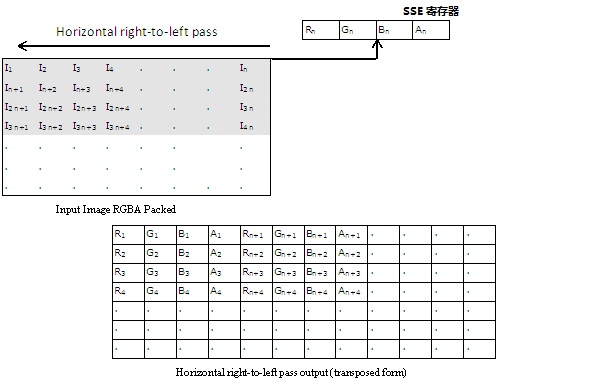
movups xmm6, xmm4; xmm6 = oTemp[i-1]; for next round: xmm6 = oTemp[i-2];

movups xmm4, xmm0; xmm4 = oTemp[i]; for next round: xmm4 = oTemp[i-1];

add esi, 0x04;

add edx, 0x10;

水平通道从右往左的计算差不多. 使用SSE指令解压缩一个像素, 加载从左往右计算的输出, 加上这一次计算的结果. 最终的输出存储在和图片一样大小的临时缓存中, 作为垂直通道的输出. 计算结果在写入输出缓存之前要做一次转置, 直接写入浮点数, 不压缩成4字节像素值.



loop\_body\_right\_left\_start:

movups current\_in, xmm0; currIn = xmm0;

mulps xmm0, xmm1; xmm0 = id[i+1] \* a0;

mulps xmm2, xmm3; xmm2 = id[i+2] \* a1;

addps xmm0, xmm2; xmm0 = id[i+1] \* a0 + id[i+2] \* a1;

movups xmm2, xmm4; xmm2 = od[i+1];

mulps xmm2, xmm5; xmm2 = od[i+1] \* b1;

subps xmm0, xmm2; xmm0 = id[i+1] \* a0 + id[i+2] \* a1 - od[i+1] \* b1;

movups xmm2, xmm6; xmm2 = od[i+2];

mulps xmm2, xmm7; xmm2 = od[i+2] \* b2;

subps xmm0, xmm2; xmm0 = id[i+1] \* a0 + id[i+2] \* a1 - od[i+1] \* b1 - od[i+2] \* b2;

movups xmm6, xmm4; xmm6 = xmm4 = od[i+1] ->next round->od[i+2];

movups xmm4, xmm0; xmm4 = xmm0 = od[i] -> next round->od[i+1];

movups xmm2, [EDX];

addps xmm0, xmm2; xmm0 = oTemp[i] + id[i+1] \* a0 + id[i+2] \* a1 - od[i+1] \* b1 - od[i+2] \* b2;

movups [edi], xmm0; od[i] = xmm0;

pmovzxbd xmm0, [esi];

cvtdq2ps xmm0, xmm0; xmm0: id[i] ->next round -> id[i+1]

movups xmm2, current\_in; xmm2 = currIn = id[i+1] ->next round->id[i+2];

sub esi, 0x04;

sub edi, EAX;

sub EDX, 0x10;

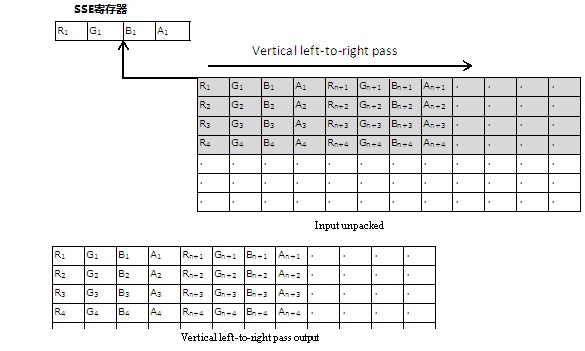
loop\_body\_right\_left\_end:

扩展元对象在绘制时得到的XL\_BitMap在每一行是连续的, 但是垂直方向上并不保证每行的内存是连续的. 所以在访问下一行时需注意要跳过bmp.scanLineLength那么长的内存, 而不是图像宽度那么长的内存.

* 垂直通道

经过转置, 垂直通道的计算和水平通道的计算是类似的. 把图像大小的临时缓存, 即水平通道的输出作为输入, 把图像输出目标作为输出.

垂直通道先加载IIR高斯模糊系数到SSE寄存器中. 从左往右(经过转置, 垂直方向从上往下的内存现在是从左往右排列的)每次计算一个像素, 从临时内存加载到寄存器中. 然后计算, 输出.



垂直通道从左往右计算的结果放在一小条临时缓存里. 作为从右往左计算的输入.

loop\_top\_down\_start:

movups xmm0, [esi]; xmm0 : currIn;

mulps xmm0, xmm1;

mulps xmm2, xmm3;

addps xmm0, xmm2; xmm0 : currIn \* a0 + prevIn \* a1;

movups xmm2, xmm4;

mulps xmm2, xmm5;

mulps xmm6, xmm7;

addps xmm2, xmm6; xmm2 : prevOut \* b1 + prev2Out \* b2;

subps xmm0, xmm2; xmm0 : currIn \* a0 + prevIn \* a1 - (prevOut \* b1 + prev2Out \* b2);

movups xmm6, xmm4; prev2Out = prevOut;

movups xmm4, xmm0; prevOut = result;

movups xmm2, [esi]; prevIn = currentIn;

movups [edx], xmm0;

add esi, 0x10;

add edx, 0x10;

loop\_top\_down\_end:

垂直方向从右往左计算和从左往右是类似的. 从每一行最后一个像素开始计算到第一个像素. 先加载之前一次的输入然后加到这一次的计算结果上. 最终的输出结果压缩成32位RGBA形式写到目标位图缓存中. 在扩展元对象的实现中, 即写到原位图缓存中. 要注意写回原图像缓存的时候每一行的间隔是bmp.scanLineLength而不是图像宽度.

loop\_bottom\_up\_start:

movups current\_in, xmm0;

mulps xmm0, xmm1;

mulps xmm2, xmm3;

addps xmm0, xmm2;

movups xmm2, xmm4;

mulps xmm2, xmm5;

mulps xmm6, xmm7;

addps xmm2, xmm6;

subps xmm0, xmm2;

movups xmm6, xmm4;

movups xmm4, xmm0;

movups xmm2, [edx];

addps xmm0, xmm2;

CVTTPS2DQ xmm0, xmm0;

packusdw xmm0, xmm0;

PACKUSWB xmm0, xmm0;

pextrd ebx, xmm0, 0;

mov [edi], ebx;

movups xmm0, [esi];

movups xmm2, current\_in;

sub esi, 0x10;

sub edi, eax;

sub edx, 0x10;

loop\_bottom\_up\_end:

使用SSE指令集实现IIR算法的性能和模糊半径无关, 因为模糊半径只在计算模糊系数的时候用一次, 在后面对图像的遍历中不再需要. 对800x600的图像, 做一次模糊的时间可以控制在20ms以下.

###### 默认混合策略 Default

以上我们介绍了在高斯模糊对象中指定gaussiantype分别为FIR和IIR时对象渲染是怎样实现的. 通过性能对比可以看到在模糊半径比较小的时候FIR更快, 模糊半径大的时候IIR更快. 而使用FIR算法时需要使用者来设定模糊半径=3\*模糊系数. 所以对高斯模糊对象增加了默认的模糊类型, 根据不同的模糊系数使用最优的模糊算法, 并且在使用FIR时硬性把模糊系数设为模糊系数\*3.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 模糊系数= 0.33  模糊半径= 1 | 模糊系数=1  模糊半径=3 | 模糊系数=8  模糊半径=24 |
| FIR算法, MMX 实现 | <15 ms | < 15 ms | <50 ms |
| IIR 算法, SSE实现 | < 20 ms | | |

## 以后想要加入的扩展和增强

* 对于计算能力更差的CPU, 使用MMX指令集实现IIR算法, 在丢失精度处理时为了避免累积误差, 考虑舍弃小数部分时做锯齿状矫正, 一个像素+1, 下一个像素-1之类的.
* FIR的MMX指令集实现在计算高斯核的时候, 因为要归一化到8位的整数, 在模糊系数很小的时候会出现(0, 256, 0)这样的高斯核, 这和没做模糊没有区别, 这种情况应该矫正成(1, 254, 1); 在高斯半径很大的时候(虽然我们不鼓励使用者这个时候还用FIR算法), 高斯核会变成(一堆0, 一堆1, 一堆0)的样子, 这时候在现有实现中会把两边的0去掉, 把半径改成1的个数(128), 但是其实已经不在是高斯模糊, 而是平滑模糊了(取周围的平均数).

## 参考文献

[1] Rachid Deriche – “[Recursively implementing the Gaussian and its derivatives](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.5904)”, 1993. [2] Lucas J. van Vliet, Ian T. Young and Piet W. Verbeek – “[Recursive Gaussian derivative Filters](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.1458&rep=rep1&type=pdf)”, 1998

[3] Dave Hale, “[Recursive Gaussian Filters](http://www.cwp.mines.edu/Meetings/Project06/cwp546.pdf)”, CWP-546

[4] Luis Alvarez, Rachid Deriche, Francisco Santana – “[Recursivity and PDE’s in Image Processing](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.5804)”, 2000

[5]<http://software.intel.com/en-us/articles/iir-gaussian-blur-filter-implementation-using-intel-advanced-vector-extensions>