这篇文档介绍了Bolt引擎的扩展元对象高斯模糊对象(GaussianBlurObject). 以下是内容梗概:

[对象的属性, 使用方法和性能数据](#_对象的属性,_使用方法和性能数据)

[属性](#_属性)

[使用方法](#_使用方法)

[性能](#_性能)

[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算法的原理和实现](#_IIR算法的原理和实现)

[IIR算法的优化](#_IIR算法的优化)

[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)

论恒定性能算法的可能性

## 对象的属性, 使用方法和性能数据

扩展对象名称: GaussianBlurObject

继承自: LayoutObject

###### 属性

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 默认值 | 说明 | 读写方法 |
| gaussiantype | string | Default  可选值: Default, IIR, FIR | 对象使用的模糊算法.  IIR的性能不随模糊系数和模糊半径变化;  FIR的性能随着模糊系数和半径递增;  Default根据模糊系数择优选择FIR或IIR; | SetGaussianType(string)  string GetGaussianType() |
| sigma | float | 1 | 模糊系数. 系数越大越模糊 | SetSigma(float)  float GetSigma() |
| radius | int | 3 | 模糊半径.  在FIR模式下, 建议取模糊系数的3倍取整, 性能随半径递增;  在IIR模式下, 这个属性没有意义. | SetRadius(int)  int GetRadius() |

###### 使用方法

扩展元对象像内置元对象一样使用. 可以指定的属性包括LayoutObject的所有属性和上文介绍的3个属性. 高斯模糊对象没有自定义事件, 如常使用LayoutObject的所有事件.

属性gaussiantype指定了高斯模糊对象使用的模糊算法. 对使用者来说, 需要注意以下三点:

* 在使用FIR算法时, 需要同时指定sigma和radius. radius=3\*sigma时可以达到预期的模糊效果, 与radius > 3\*sigma的效果差异微弱. 使用者当然有让radius < 3\*sigma的自由, ]可能面临模糊系数设置的很大, 但是模糊效果甚微的窘境.

由于FIR算法的计算量随模糊半径线性增长, **使用radius = 3\*sigma当然是性价比最高的策略**. 从下面的对比图左上和右上两组文字对比中可以看出, 模糊半径设大了没有用, 徒增CPU消耗; 从左下和右下两组文字对比中看出, 模糊半径设小了模糊效果不好.



图1 – 模糊半径, 模糊系数, 对模糊效果的影响

* 在使用IIR算法时, 算法和模糊效果都与模糊半径无关, 使用者只需按需设定模糊系数sigma即可.
* FIR算法在模糊系数小的时候效率高, IIR算法在模糊系数大的时候效率比FIR高. 所以高斯模糊对象提供”Default”的模糊类型混用两种算法. 在模糊系数变化的时候, 自动将模糊半径设为模糊系数的3倍; 在模糊半径小于等于1的时候, 使用模糊半径等于3倍模糊系数的FIR算法; 在模糊半径大于1的时候使用与半径无关的IIR算法以达到最优性能.
* 剪裁敏感. 高斯模糊对象是剪裁敏感的, 所以需要放在剪裁敏感的层对象LayerObject中使用. 这意味着同一层中的任何对象发生更新, 高斯模糊对象都会跟着更新.

###### 性能

Bolt对图像实时性能要求是800x600的图片20ms内处理完. 下面是目前提供的两种算法的性能数据:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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 | | |

可以看到使用IIR算法性能较稳定, 使用Default模式结合两种算法可以达到性能要求.

## 高斯模糊的基本原理

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>

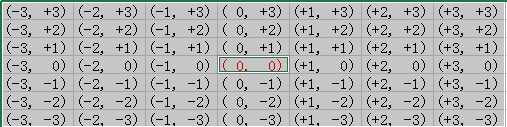
可以看到高斯模糊就是, 把图像中每一个像素的颜色值替换成它周围像素颜色值的加权求和.

我们把加权中”权”的矩阵叫做高斯核, 如果我们按照正态分布来分配周边各点的权重的话(近的点权重大, 远的点权重小), 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, 正态分布的”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, 周围的点的个数就是(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.

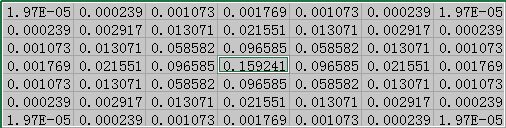
正态分布是连续的概念, 而在实际应用中我们需要离散化的高斯核, 而且权重和要等于1来保证图像明度不变. 为此先取半径内距中心点距离为整数的正态分布函数值, 得到离散的正态分布矩阵, 再对矩阵元素归一化处理使其和为1. 用G(x, y)来表示给定方差的二维正态分布函数, 取高斯半径为1, 最终得到的高斯核矩阵形如:

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

高斯模糊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.

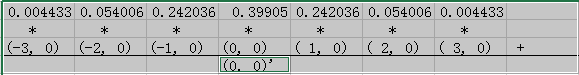


(模糊半径为3的模糊效果不过尔尔, 计算量已经相当于普通图像处理过49遍了!)

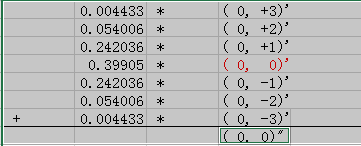
所幸高斯函数有个叫”线性可分特性”的特征, 二维矩阵变换可以通过两次一维矩阵变换来得到, 算法效率提高了一个数量级. 具体来讲就是: 在水平方向, 对每个像素点取左右半径的像素加权求和, 对于水平方向的处理结果, 在垂直方向对每个像素点取上下半径的像素加权求和.

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

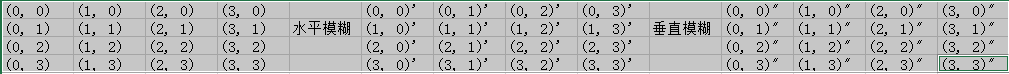
* 跨越扫描线访问内存

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算法的原理和实现

我们看到即使两个维度叠加计算, 模糊半径对算法复杂度的影响仍然是线性的. 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, 简单地 i’ = a\*i + b\*(i-1)’ + c (i+1)可达不到这样的效果, 更何况我们复用了前面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; 而从前往后, 和从后往前各自计算一遍, 就可以把之前像素们和之后像素们对当前像素的影响都囊括在内了. 这中间的证明过程, 和系数的推导, 在下面这些论文中有详细的论证.

[1] Rachid Deriche – “Recursively implementing the Gaussian and its derivatives”, 1993.

[2] Lucas J. van Vliet, Ian T. Young and Piet W. Verbeek – “Recursive Gaussian derivative Filters”, 1998

[3] Dave Hale, “Recursive Gaussian Filters”, CWP-546

[4] Luis Alvarez, Rachid Deriche, Francisco Santana – “Recursivity and PDE’s in Image Processing”, 2000

这种算法被称作无限脉冲回应滤波器(IIR, Infinite Impulse Response), Intel在下面给出了一个使用AVX指令集的实现, 我相信它的实现性能更好, 因为AVX可以一次处理8个浮点数, 两行一起算. 但是不是所有的处理器都支持AVX指令集, 而且

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

###### IIR算法的优化

###### 默认混合策略

## 以后想要加入的扩展和增强

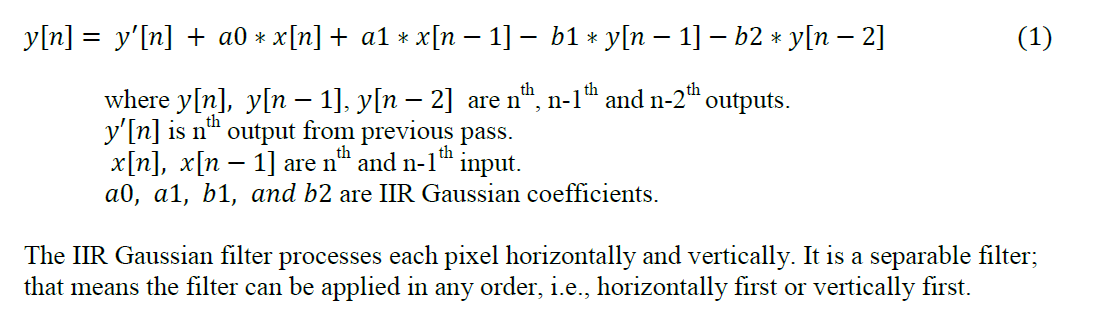
IIR: 摒弃模糊半径对复杂度的影响

我”参考”的源地址在这里: <http://software.intel.com/en-us/articles/iir-gaussian-blur-filter-implementation-using-intel-advanced-vector-extensions>

这个之前没有见过所以觉得新鲜,下面是我的理解, 请尽情吐槽:

在FIR高斯模糊实现中我们看到, 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, 简单地 i’ = a\*i + b\*(i-1)’ + c (i+1)可达不到这样的效果, 更何况我们复用了前面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(因为还米有算到). 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; 而从前往后, 和从后往前各自计算一遍, 就可以把之前像素们和之后像素们对当前像素的影响都囊括在内了. 这中间的证明过程, 和系数的推导, 在上面那篇文章的引用1和引用2中有, 我没有看, 也不会证, 就是觉得这个结论很酷, 要让它被应用.

算法的实现很简单, 按照下面这个公式计算每一个像素, 水平方向从左往右, 从右往左, 垂直方向从上往下, 从下往上各计算一次. 每次的输出是下一次的输入.



系数a0, a1, b1, b2的计算, 是从附带的源码里扒来的.

一点优化:

水平计算的结果, 写入转置矩阵中, 使得垂直计算时在内存中的访问是连续的.

对于多核cpu, 多线程同时计算;

这个算法在附件GaussianBlurIIRImpl.cpp的DericheIIRRenderSSEIntrinsics中有使用sse的intrinsics方法实现. 和Intel AVX的实现差不多, 只不过它是一次计算两行, 我是一次计算一行.