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)

## 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 |
| 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; |
| sigma | float |  |  |
|  |  |  |  |

###### 使用方法

###### 性能

## 高斯模糊的基本原理

## 对象中使用的两种模糊算法

###### FIR算法的原理和实现

###### FIR算法的优化

###### IIR算法的原理和实现

###### IIR算法的优化

###### 默认混合策略

## 以后想要加入的扩展和增强

这一篇介绍了基本的高斯模糊的原理.

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

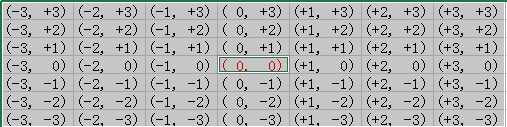
FIR (有限脉冲回应什么的), 两个维度同时计算:

可以看到高斯模糊本质上是对周围的点加权求和.

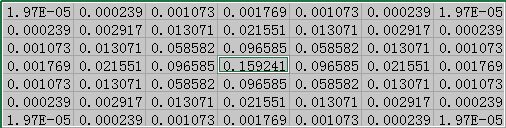
加权中”权”的矩阵叫做高斯核, 矩阵的半径叫做模糊半径, 如果我们按照正态分布来分配周边各点的权重的话(近的点权重大, 远的点权重小), 高斯函数中的方差sigma会影响权重矩阵(高斯核), 我们把这个sigma叫做模糊系数, sigma越大, 像素点受到周边像素的影响越大, 图片越模糊.

“周围的点”中的”周围”的范围由高斯半径决定, 半径为r, 需要计算的点的个数就是(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, 所以我们取3倍模糊系数作为模糊半径.

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(3\*2+1)^2 = 49个点来计算每一个像素点的颜色值:



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.



上图是模糊系数=1, 半径=3的高斯核矩阵.

这个计算量显然太大了, 而且半径为3的模糊效果也不过下图所示, 要增加模糊半径和模糊系数的计算量是平方的: O(w\*h\*(2\*r+1)\*(2\*r+1) ~ O (4\*w\*h\*r\*r), 其中w, h为图片大小.

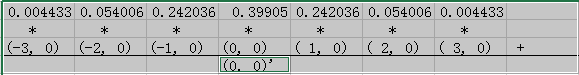


FIR, 两个维度叠加计算

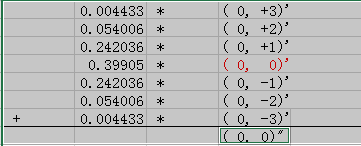
所幸高斯函数有个叫”线性可分特性”的特征, 二维矩阵变换可以通过两次一维矩阵变换来得到, 算法效率提高了一个数量级. 具体来讲就是: 在水平方向, 对每个像素点取左右半径的像素加权求和, 把结果在垂直方向对每个像素点取上下半径的像素加权求和.

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:



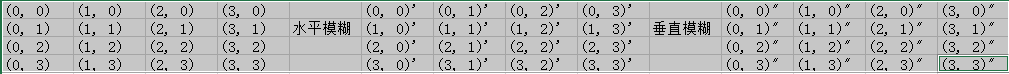
得到最终像素的颜色值.

这样算法复杂度为O(w\*h\*(2\*r+1)\*2) ~ O(4\*w\*h\*r). 半径对复杂度的影响是线性的, 没那么恶劣了.

在这里计算水平模糊时, 算法访问每一个像素点水平方向的邻居, 它们在内存中是相邻的;

在计算垂直方向的模糊时, 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, 这样不好.

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buffer.



伪代码实现, 没有写入边界条件. 实际实现中, 越界的点我用的边界点.

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];

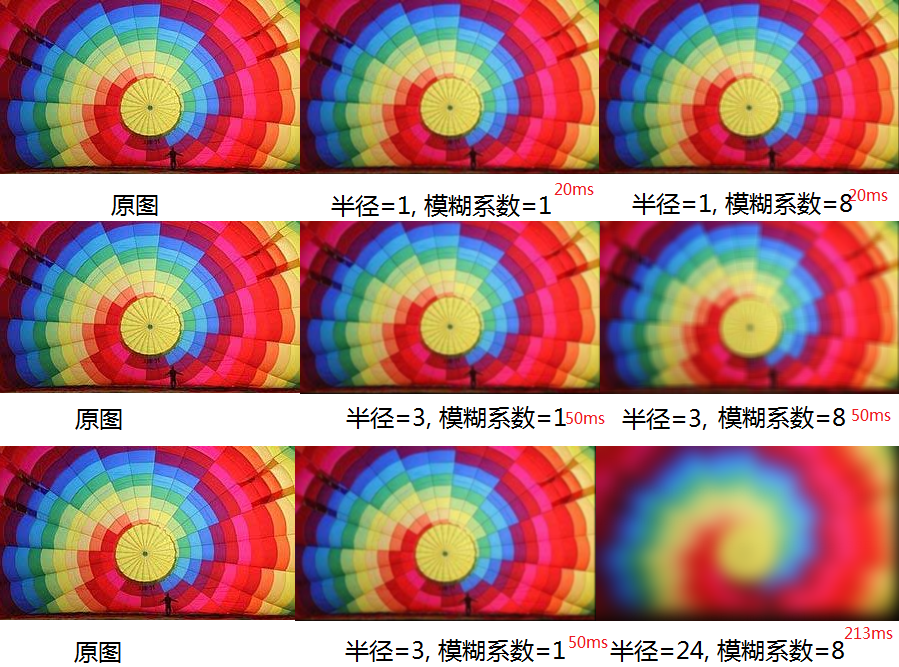
}

lpPixelBufferInitial[row][column] = localSum;

}

}

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. 是否有可能无视模糊系数对半径的影响, 始终让半径等于一个可以接受的值呢? 我们来做个比较看看, 下表第一行把模糊半径固定在1, 第二行把模糊半径固定在3, 第三行让模糊半径随着模糊系数的增长而增长, 表中红字标记的时间是对800x600的图片做模糊的时间:



可以看到固定模糊半径的效果差强人意, 本来应该很模糊的图片, 因为模糊半径不够而效果不明显. 但是模糊

所以请各位指点, 如上算法的改进方法以及指令集实现的改进方法.

附件GaussianBlurFIRImpl.cpp中OneDimentionRender方法是这种算法的c实现, OneDimentionRenderSSE改用asm指令实现循环体内部的算法, 即伪代码中绿色的部分. 在一步步把c实现改成指令集实现的过程中, 我发现把循环逻辑纳入asm段落没有明显的效率的提升, 所以没有继续把所有实现过程都用指令集来实现. 我用到了mmx的寄存器和指令来矫正越界索引, 访问图片内存和实现循环; 用到sse寄存器来计算颜色分量的加权求和, 并保存中间值. 因为

**我的问题在这里:**

1. 如果要牺牲精度换取效率, 把浮点数改用整形, xmm0寄存器改用mmx的话: 因为每一个像素都是周围像素加权求和的结果, 而且模糊系数越大, 要计算的像素越多, 浮点取整的误差也越多, 这样累积下来的误差能否接受? 高斯核的值都是0-1之间的浮点数, 如果想用位移来实现整形\*浮点的话, 具体怎么实现? (给个栗子就好)
2. 指令集配对的实现技巧
3. 越界的索引的饱和加法/减法. 对于前面r个点, 在计算当前点的前面r个点的时候会发生索引越界的情况, 有没有饱和减法可以用? 对于每一行最后r个点, 计算当前点后面r个点的时候会发生越界的情况, 有没有不用比较跳转的方法来保证索引不越界?
4. 用指令集实现循环和c循环效率区别大不?

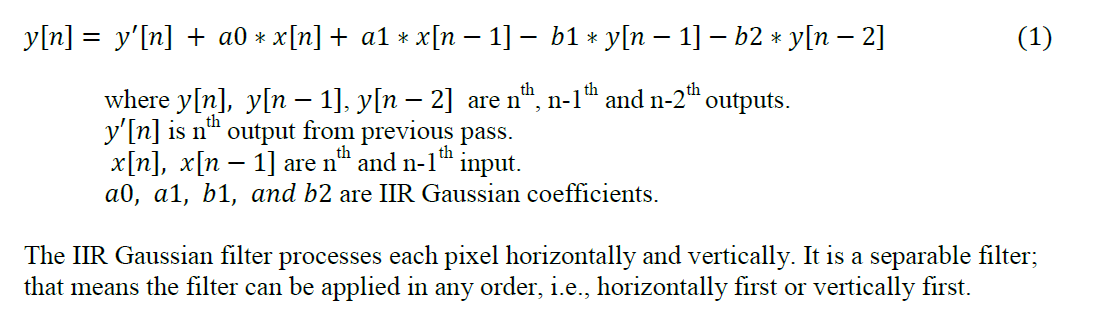
IIR: 摒弃模糊半径对复杂度的影响

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.