VCB-Studio教程13 Resizer (1)

本教程旨在讲述avs和vs中Resizer的基础应用。

1. 图像放大缩小中不可避免的三种artifacts: blurring(模糊), aliasing(锯齿)和ringing/haloing(振铃/晕轮)

为了保证最佳效果，请在100%下观看doc版本。

这是原图：，我们现在把它放大到2.6倍

典型的blurring效果（guassian p=15）：



典型的aliasing效果(point):



典型的ringing/haloing效果(lanczos8)：



原图（高sharpness/锐利度）



一个好的缩放算法，就是在尽可能锐利的前提下，尽量控制blurring, aliasing和ringing/haloing。特别是在放大的时候这一点尤为重要。而某些blurry的算法，比如softcubic/guass，因为blurring可以掩盖瑕疵的特性，有一些特殊用途（比如可以用来放大chroma）。

在缩小图像的时候，往往区别不是很明显。

2. Bilinear——最广泛使用的缩放算法。

Bilinear又称为双线性。BilinearResize(1280,720)是它在avs中的用法。效果图如下：



它是简单高效，效果不差的算法，并且很容易在硬件层面上实现，所以被广泛应用。其特点是不出ringing/haloing，也不blurry，但是锐利度很低，锯齿多。不适合用来放大，但是在一些低码率的编码中，如果需要缩小图像，Bilinear是很好的选择。因为它出来的线条锐利度不高，比较节省码率。

3. Bicubic——最广泛使用的高质量缩放算法。

Bicubic又称为双立方。它比Bilinear来的复杂，效果也要略好。avs中使用方法为BicubicResize(1280,720,b=0,c=0.5)

b和c是Bilinear的两个参数，一般认为，b代表blurry的强度，c代表sharpness的强度。默认值是b=c=1/3=0.3333…, 这两个值是Mitchell和Netravali测试总结的，最适合人眼的值。所以b=c=1/3的BicubicResize又被称为标准双立方，或者米切尔算法。

设置b和c的时候，为了保证图像缩放后的准确性，一般建议b<0.4，同时b+2c=1。因为不推荐b为负数，所以常规搭配中，b=0, c=0.5是最锐利的组合。这个组合被称为Catmull-Rom，是一个很好的高质量downscale算法。

当c>0.6的时候，图像会被刻意锐化，并且表现出ringing/haloing，虽然在目视效果上，锐利度的提高往往会抵消ringing/haloing带来的副效果。这时候一般依旧是搭配b=0. 例如BicubicResize(b=0,c=0.75). 很多播放器会使用这种算法，并且用-c来代表锐利度。比如双立方（锐利度=-0.6）就表示BicubicResize(b=0,c=0.6).

你也可以刻意拉大b和c来看看有啥奇妙的效果，比如可以试试b=0, c=-5

下图是标准双立方缩放后的值：



avs的算法中，常用taps这个概念来代表确定一个点的值，需要用到多少个它的临近点。Bilinear算是 1 taps，而Bicubic则是2taps。

无论是Bilinear还是Bicubic，缩放的时候，确定一个像素的值，只用到原图中2x2=4像素点。在缩小时候这不是问题，当放大的时候就明显的出现锐度不够的问题。于是我们继续介绍一些常用的多taps的resizer。

Didée给出过另一组适合缩小的b和c组合：b=-0.5,c=0.25。Bicubic通过灵活设置b和c，可以做出2-taps的resizer能做出的一切效果。mawen1250曾经推荐用Catmull-Rom来处理较好源的Chroma upscaling.

4. Lanczos——适合放大的缩放算法。

Lanczos算法是一个广谱，偏放大的算法。avs中用法是LanczosResize(1920,1080)或者Lanczos4Resize(1920,1080)。后者表示用4taps Lanczos，锐度更高，锯齿更少，ringing也更多。

实际使用的时候，Lanczos做放大一般搭配non-ringing算法。这个我们后续再说；下图是non-ringing Lanczos 4的拉大效果：



5. Spline——最广谱的缩放算法。

Spline系列的设计理念是接近Lanczos的锐利度，同时控制ringing等问题。Spline系列也有taps的区别，Spline36Resize是3tpas，适合缩小，Spline64Resize是4taps，适合放大。

Spline系列适合比例不大的时候放大缩小的使用。一般放大的时候也建议搭配non-ringing算法。下图是non-ringing spline 64的效果：



6. nnedi3\_resize16——最好的事实放大算法。

nnedi3是一个反交错滤镜，基于nnedi插值放大算法。

在之前的教程中我们讲过，反交错关键就是将高度切了一半的场景给拉回去，nnedi3最基础的功能，是将一个图像纵向拉伸到两倍。

那么既然能拉伸两倍，就能通过多次拉伸，拉伸到4倍，8倍……

另外，如果你把图像旋转90°，然后拉伸，然后再转回去，你还可以用它实现横向拉伸。

所以nnedi3的功能就是，将一个图像横向拉伸2^a倍，纵向拉伸2^b倍。

nnedi3是一个不会产生、相反还会消除aliasing的算法。在需要对付锯齿的地方，比如反交错/抗锯齿操作，nnedi3是一个常用滤镜。

nnedi3会产生一点点的ringing，非常少，少到可以忽略不计。

这个算法在拉伸时候，对线条的效果，对ringing/aliasing的控制，完胜lanc/spline/jinc这些常规算法。

nnedi3拉伸后的图像锐利度很高。

现在avs版的nnedi3似乎只接受8bit输入，只能输出8bit。因此在非线条部分，精度不如已经在16bit/32bit下实现的算法。比较容易导致的问题就是色带，那是典型精度不足的症状。好在这种问题不常见，不用过分担心。

nnedi3\_resize16的拉升原理是，用nnedi3来处理线条部分保证效果，用高精度resizer来处理非线条部分以提升精度。

nnedi3用来放大到任何一个分辨率的原理：先用nnedi3拉到一个分辨率，再用常规resizer做downscale/upscale：

比如我们想将一个720p拉伸到1080p:

1、用nnedi3将1280x720的分辨率拉升到2560x1440

2、用常规downscale算法缩到1920x1080

在不允许用nnedi3拉伸4倍的前提下，将一个848x480的480p拉伸到1080p:

1、用nnedi3将848x480拉升到1696x960

2、用常规upscale算法拉升到1920x1080

avs中的使用方法：

# 处理到某一步骤是stacked 16bit

nnedi3\_resize16(3840,2160,nns=4,lsb\_in=true,lsb=true)

#处理完后依旧是stacked 16bit

这是nnedi3\_resize16放大后的结果：



7. 如何保证放大观看的过程中没有干扰——PointResize

point-resize中文名叫做临近采样，说白了直接找原图中最对应位置的像素点糊上去。后果前面也给了，aliasing逆天。但是point-resize最适合需要放大后仔细观看图片的场合：它可以完美的把一个像素放大成4个/9个/16个…像素，在这个过程中，没有计算精度的问题。

PointResize只适合整数倍的放大，而且是为了仔细检查图像像素级别瑕疵的放大。

这是用point-resize放大到2x的后果。可以看到原图本身就带有次像素级别的锯齿（这在高分辨率素材做downscale时候很常见）：



8. 用blurring换ringing的算法：softcubic/Gaussian

softcubic 是bicubic的变种，满足b+c=1且b>=0.5

softcubic 50就是BicubicResize(b=0.5,c=0.5)

softcubic 75 则是BicubicResize(b=0.75,c=0.25)，以此类推。softcubic 100则是最模糊的。

GaussResize则是另一种保证不出ringing的算法。使用方法是GuassResize(1920,1080,p=30)

p=1~100,表示锐利度。一般保证很blurry的效果用p=15~30， p很高的时候则接近Bilinear的效果。

这是用GausResize(p=30)拉出来的效果：



softcubic经常被用于chroma upscaling，而Gaussian Resize则用于缩放的时候non-ringing的实现。

9. avs中Resizer的16bit实现——Dither\_resize16

Dither\_resize16是Dither\_tools里提供的函数，可以在16bit下做高精度的resize处理。这对图像中的低频区域非常有帮助，可以避免因为resize精度不足产生的各种问题。

Dither\_resize16只接受stacked 16bit输入，输出也是stacked 16bit。

它通过kernel来指定算法(默认Spline)，taps来指定tap数量（默认3），a1/a2来指定其余参数（kernel是bicubic时候默认1/3 1/3）

比如Bilinear算法：Dither\_resize16(1280,720,kernel="bilinear")

Catmull-Rom算法：Dither\_resize16(1280,720,kernel="bicubic",a1=0,a2=0.5)

spline36算法： Dither\_resize16(1280,720) #全默认就好

Lanczos 4算法：Dither\_resize16(1920,1080,kernel="lancsoz",taps=4)

Gaussian 算法： Dither\_resize16(1920,1080,kernel="gauss",a1=50)

一般来说，resize应该永远在16bit下处理，所以除了某些特殊场合，一概建议在16bit下做resize

10. vs中Resizer的实现

vs自带的Resizer是基于zimg的。可供选择的种类有Bilinear，Bicubic，Point，Spline16，Spline36，Lanczos。通过filter\_param\_a, filter\_param\_b来调整bicubic下b和c，以及Lanczos的taps。

doc：<http://www.vapoursynth.com/doc/functions/resize.html>

vs自带的resizer输出精度和输入精度一致。

除了自带的resizer，还可以用fmtc.resample。它几乎是avs中dither\_resize16的复刻版。fmtc.resample的运算精度和输出精度永远是16bit整数或者32bit浮点（取决于输入是整数和浮点）。所以类似：

src8 = core.lsmas.LWLibavSource("00000.m2ts")

down16 = core.fmtc.resample(src8,1280,720)

down16出来就是16bit整数。

11. Resizer的non-ringing用法

用Lanczos/Spline做拉升的时候，经常建议搭配non-ringing算法。如果是avs内置的resize，可以这么写：

Lanczos4Resize(1920,1080).repair(GaussResize(1920,1080,p=100),1)

就是在后面加一个.repair(GaussResize(xxxx,yyyy,p=100),1)

vs中则必须依赖fmtc：

upscaled=core.fmtc.resample(src\_720, 1920, 1080, kernel="lanczos",taps=4)

upscaled=core.rgvs.Repair(upscaled, core.fmtc.resample(src\_720, 1920, 1080, kernel="gauss",a1=100),1)

如果使用avs中的dither\_resize16，则更简单：Dither\_resize16nr()…

加一个nr就好（nr=non-ringing）

12. 修改Dither\_convert\_yuv\_to\_rgb()中chroma upscaling算法

Dither\_convert\_yuv\_to\_rgb()默认使用的是Bicubic算法。你可以在chromak参数中调节算法，taps/a1/a2中调节参数。

比如对画质不是非常好的源，使用softcubic 60做chroma upscaling：

Dither\_convert\_yuv\_to\_rgb(a1=0.6,a2=0.4)

对画质非常好的源，使用non-ringing Spline64/Lanczos4做chroma upscaling：

Dither\_convert\_yuv\_to\_rgb(chromak="spline",taps=4,noring=true)

也可以用Catmull-Rom:

Dither\_convert\_yuv\_to\_rgb(a1=0,a2=0.5)

或者干脆用nnedi3\_resize16来处理chroma并转为RGB24:

nnedi3\_resize16(output=”RGB24”,nns=4)

13. 不同Resizer背后的数学原理

见 <http://svn.int64.org/viewvc/int64/resamplehq/doc/kernels.html>

以下我简单的翻一下：

为了让你能更好的了解并选择resize算法，这个页面把许多常用算法图表化。没有所谓的“最好算法”，既然这些算法被人研究，那么它们都是有用的，只是适用的场合和使用者的偏好各不一样罢了。

先从一个通用的算法开始，比如Bicubic或者Lanczos，如果你觉得它看上去不够好，和其他的算法比较一下，然后找出更符合你需要和口味的算法。

X轴代表了理想取样点（比如说，按照公式计算，resizer后的图像应该对应原图(1.75,1)的位置，那么(1.75,1)就是理想采样点）和实际取样点（但是实际原图中只有(1,1)和(2,1)两个像素，没有小数坐标，那么这两个点就是实际取样点）间的距离(分别为0.75和0.25)。0.0这个位置表示resize的时候最理想样点正好落在原图某个像素上，1.0表示理想取样点和原图中有的像素，在缩放后的图像里面偏差了1个像素。以此类推。

Y轴代表了在不同距离段采取怎样的加权。注意了有些算法有负数权值。

注意：看图需要HTML5，IE6这种滚粗。

以下是我个人一些观察：

Blurring发生在曲线总体有个正数的偏差。你看下Bilinear，如果任何算法的权值给的比Bilinear高，那么几乎必然需要在某个地方有负数的权值来抵消掉。softcubic100就不去做这个抵消，所以这玩意糊你一脸。

sharpness，锐利度，跟采样点数量和权值的绝对值总和（就是上上下下曲线围成的总面积）呈正相关。Bilinear就是最不sharp的，尽管它不算blurry。

如果有负数权值，一定会引起ringing效果。负数权值越多越严重，比如Lanczos 4

13. Resizer中的裁剪和位移

所有resizer中还有4个参数： src\_left, src\_top, src\_width, src\_height

这四个参数的用法和效果和crop是基本一致的，可以用来在resize前做裁剪。不同的是，resizer中并非是物理裁剪，而是重采样过程中的边界值设定问题。所以resizer中可以设置小数，而crop中必须是整数。

src\_width, src\_height可以是正数，表示保留视频的宽度和高度，如果是负数则表示切割掉原视频右边和下边若干像素。但是src\_left和src\_top则并非如此，这两个数字是resize的时候，左侧和上侧的偏移值，可正可负。这两个可以用来做非整数像素的视频平移。

自己找个1080p的视频，试一下这两个avs的效果：

Crop(960,540,0,0)

Spline36Resize(1920,1080)

和Spline36Resize(1920,1080,src\_left=960,src\_top=540)

Resizer这四个参数，和Crop有着微妙的区别，建议多在avspmod中尝试和对比，通过实际操作掌握resizer这些参数的设置规律。

src\_left是正数的时候，表示对视频向左偏移。因为视频左边被切掉了一块，导致的后果是整个视频必须向左移动来填补缺失的部分。

src\_left是负数的时候，表示对视频向右偏移。因为视频左边多出来一块空白（resizer会合理插值），导致的后果是整个视频必须向右移动来放下这块东西。

src\_top是正数的时候，表示向上偏移，反之是向下偏移。

在resizer中src\_left、src\_top实际的作用不是切割或者插值，而是指定重采样时，原图的起始位置。比如src\_left=1, src\_top=-1就意味着，把原图(1, -1)这个位置作为resize时的原点(0, 0)。因此，我们能够设定小数的位置作为原点，并且那些被“切掉”的部分依然会对resize结果产生影响。