**[VCB-Studio][教程25]Resizer(2)**

1. RGB48Y的相关操作

RGB48Y是avs中常用的高精度RGB的表示方法。这里我们详细讲述一下关于它的一些处理：

YUV->RGB48:

可以使用Dither\_convert\_yuv\_to\_rgb(output=”RGB48Y”), 也可以使用nnedi3\_resize16(output=”RGB48Y”)。其实使用nnedi3\_resize16就是将UV放大到Y的分辨率之后，再用Dither\_convert\_yuv\_to\_rgb转换，只不过使用了更适合好源的UV放大算法。

RGB48->YUV:

使用Dither\_convert\_rgb\_to\_yuv

R=SelectEvery(3,0)

G=SelectEvery(3,1)

B=SelectEvery(3,2)

Dither\_convert\_rgb\_to\_yuv(R,G,B)

RGB24->RGB48Y

思路就是先将RGB三个通道分别拆成3个Y8，然后再转为stacked 16bit：

src = AVISource(“fraps.avi”) #RGB24输入

R=src.ShowRed(“Y8”) #将红色通道提取，并且转换为Y8格式

G=src.ShowGreen(“Y8”)

B=src.ShowBlue(“Y8”)

Interleave(R,G,B) #创建一个新视频，每3帧分别从R G B中取

U16(tvrange=false) #将新视频转换为16bit。注意RGB都相当于是pcrange，所以设定tvrange=false

RGB48Y->RGB24

思路是先将RGB48每个通道由stacked 16bit转为8bit，再将3帧Y8合并成一帧RGB24：

R=src16.SelectEvery(3,0).Down10(8,tvrange=false)

#注意这里不能用DitherPost()来做16->8. DitherPost只适用于tvrange的视频；RGB这种pcrange的视频必须用Down10(8,tvrange=false)来做16->8.

G=src16. SelectEvery(3,1).Down10(8,tvrange=false)

B=src16. SelectEvery(3,2).Down10(8,tvrange=false)

MergeRGB(R,G,B) #MergeRGB是将输入的3个clip的Y拿出来，作为RGB通道的分量，来生成一个RGB的视频。默认生成的是RGB32（带个空的alpha），如果一定要输出RGB24需要MergeRGB(R,G,B,pixel\_type=”RGB24”)

2. 用Resizer修正Chroma shift和RGB shift

Chroma shift，是指Chroma平面和Y平面不重合，出现规律性、一致性的偏移。通常在线条、极蓝、极红处较为明显。比如说原图（YUV420的源）：



将Chroma向左偏移4个像素(相对于Luma；如果是单看Chroma平面则是2个像素)：



可以看出，在颜色剧烈变化的边缘，出现大量单边亏欠，另一边溢出，这种就是典型的Chroma shift。UV平面相对Y平面是不重叠的。

更多时候，chroma shift的问题绝不会这么明显。实际压片中碰到的比较明显的，比如电磁炮S的OVA:



甚至，最常见的Chroma shift是由于官方制作的时候，Chroma实际的placement是MPEG1（比如说，RGB的母带，转成YUV444的YUV格式，然后降低420的时候，用了中心对齐的resizer。导致的后果就是UV的中心跟Y是重合的，即MPEG1的cplace。但是压片的时候标注的又是按照MPEG2的规范来解码，造成了Chroma相对Luma有0.5像素的偏移）。

典型的例子包括Little Buster系列，注意头发那边：



point-resize到4x4倍大小，应该可以看出线条左边有颜色亏缺：



Chroma shift的解决方式就是用resizer给shift回去。不过在那之前，我们再来详细回顾一下Resizer做shift的一些细节：

Resize(src\_left=x,src\_top=y,src\_width=a,src\_height=b)

x,y,a,b都可以是小数。

如果x或者y>0:

意味着左边切掉x像素，上边切掉y像素。表现的后果是视频向左平移x像素，向上平移y像素

如果x或者y<0:

意味着左边插补|x|像素，上边插补|y|像素。表现的后果是视频向右平移|x|像素，向下平移|y|像素。

（PS: x和y不一定需要同时>0或者<0）。比如x>0,y<0，就是向左平移x像素，向下平移|y|像素。）

如果a或者b>0:

在x和y设定下，做完裁剪或者插补的基础上，取片源长宽值为a和b的一块矩形，来做resize

如果a或者b<0:

在x和y设定下，做完裁剪或者插补的基础上，切掉片源右边|a|像素，下边|b|像素，来做resize

在resizer中src\_left、src\_top实际的作用不是切割，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。

其中，如果片源是YUV格式的，切割或者增补的时候，UV平面会相对调整。比如对一个YUV420的视频做src\_left=0.5, 意味着Y平面左移0.5像素，UV平面左移0.25像素，因为尺度只有Y的一半。

如果一个实际为MPEG1 cplace的视频，我们想要去修正它，让它成为标准的MPEG2 cplace的视频，方法就是将chroma右移0.5像素(相对于Y)，或者说将Chroma右移0.25像素（相对于UV）。这个方向跟之前MPEG2的视频做chroma upscale是正好相反的：MPEG2的视频做Chroma upscale，第一步是先左移UV变成实际上的MPEG1视频。

我们依旧可以把UV分离出来，再修正:

src16=LWLibavVideoSource(“00000.m2ts”,threads=1,format=”yuv420p16”,stacked=true)

U=src16.UtoY.Dither\_resize16(960,540,src\_left=-0.25)

V=src16.VtoY.Dither\_resize16(960,640,src\_left=-0.25)

YtoUV(U,V,src16)

然而，使用Dither Tools，我们可以设置对哪些平面做处理。Dither Tools绝大多数函数都有Y,U,V三个参数,可以设置为整数：

<=0, 表示该平面直接每个像素统一填数。填的数字是你设定的数的绝对值。

1，表示不做处理，输出毫无意义的垃圾数据

2，不做处理，输出输入的数据。

3. 处理（默认就是3，表示做处理）

所以我们可以让Dither\_resize16只处理UV，不处理Y：

LWLibavVideoSource(“00000.m2ts”,threads=1,format=”yuv420p16”,stacked=true)

Dither\_resize16(1920,1080,src\_left=-0.5,Y=2)

Y=2表明直接输出源的Y平面。因为这次我们针对的是1920x1080的视频，所以移动的尺度是0.5个像素。Resizer在处理只有一半宽度的UV平面的时候，自动将它减半为0.25。

如果使用vs，写法还更简单：

fix = core.fmtc.resample(src16, 1920, 1080, sx=[0,-0.5,-0.5])

其他的一些Chroma shift则不一定这么标准。修复的时候，一般只能慢慢去实验src\_left和src\_top两个值（有时候UV还得分开处理，而不能一个Resize搞定），直到画面看上去基本没有什么问题为止。

除了chroma shift，还有RGB Shift的问题。RGB shift跟Chroma shift很类似，只不过不同步是产生在RGB平面上的而已，表现为颜色溢出往往是红、绿、蓝色有偏移和溢出。处理思路是先将图像转换为RGB48Y，在RGB48Y下做修复，最后转换回YUV。

通常情况下，你可能很难分辨究竟应该往上下左右哪个方向移动，这时候可以在avspmod中输出RGB48Y或者RGB24三个通道被拆开后的图像，通过前后帧切换，你就很容易判断出，绿色通道比起红色通道，往哪个方向偏了，这种信息。方向判断对了，下面只需要耐心的调试偏移值就可以了。

在官网科普文评论中，我演示RGB shift的时候，故意将一张图像红色平面向左上移动2个像素，蓝色平面向右下移动2个像素。我们现在来还原下当时操作用的avs:

JPEGSource(“xxx.jpg”) #读入jpg为YUV-8bit, MPEG1, BT601 pcrange数据

nnedi3\_resize16(output=”RGB48Y”,matrix=”601”,tv\_range=false, cplace=”MPEG1”) #转为RGB48Y

R=SelectEvery(3,0).Dither\_resize16(480,648,src\_left=2,src\_top=2).Down10(8,tvrange=false)

G=SelectEvery(3,1).Down10(8,tvrange=false)

B=SelectEvery(3,2). Dither\_resize16(480,648,src\_left=-2,src\_top=-2).Down10(8,tvrange=false)

#将三个平面单独提取出来，并且对R和B用Dither\_resize16做shift，然后降低到8bit

MergeRGB(R,G,B)

#将三个通道合并为RGB24，然后截图保存

如果是视频处理，修复shift后，不要用Down10转为8bit，而是用Dither\_convert\_rgb\_to\_yuv将RGB48Y转换为YUV数据，并且最好是YUV 4:4:4格式，然后用于压制。

3. avs自带Resizer的Chroma shift问题

我们提到过，MPEG2的chroma placement下，Y和UV的中心有着0.5像素的偏差（这个数值是按照Y的尺度算，下同。如果按照UV的尺度算就要缩小1/2），MPEG1的chroma placement下不存在这个偏差，Y和UV的中心是重叠的。

avs自带的resizer，都是中心对齐的。如果一个YUV420的视频被放大到两倍，有什么后果呢？

MPEG1的cplace下，没问题，原本重合的，通过中心对齐的放大缩小后依旧是对齐的；

MPEG2的cplace下，这个偏差也被放大到两倍，Y和UV的中心差了1.0像素。

问题是，缩放完毕的图像，也是按照MPEG2的标准来的。它的Y和UV中心差应该依旧保持0.5像素，而不是变化到1.0像素！

这就产生了Chroma shift。产生的原因是，MPEG2下的UV相对于Y是靠左对齐，而用的resizer是中心对齐。

假设缩放后，放大的倍数为r. 如果r<1则说明是缩小：

缩放后，Y和UV中心的差距为0.5r

那么产生的chroma shift就是0.5r-0.5。正数表示UV相对于Y往左偏；负数表示往右偏。

一般放大缩小，产生的shift值在0.2~0.4这个数量级，不是特别明显，也一般不用特别的去修正；

如果你想要避免，就用Dither\_resize16/nnedi3\_resize16/Resize8(<https://www.nmm-hd.org/newbbs/viewtopic.php?f=7&t=1323>) 这些滤镜。它们会处理这个问题，使得出来的结果没有chroma shift。

vs自带的Resizer滤镜不存在这些问题，事实上，vs自带的滤镜允许你指定cplace。

4. Gamma-aware resize的基础知识

RGB模型下RGB三个通道，和YUV模型下的Y通道，都是经过了Gamma曲线压缩，而不是线性的。

怎么理解这句话呢？以8bit pcrange下亮度Y为例（UV假设都是中值，使得Y从小到大变化，图像从黑变到灰，再变到白）：

它的取值是0~255，0代表黑，255代表白。

有一种颜色叫做标准灰，它是黑白按照50%的比例混合而来的。也就是说，标准灰的亮度是黑和白的平均。

如果图像的实际亮度，跟数值是线性关系，那么我们应该可以得到标准灰的Y值：(0+255)/2≈128。

但是实际上，标准灰的Y值是188。

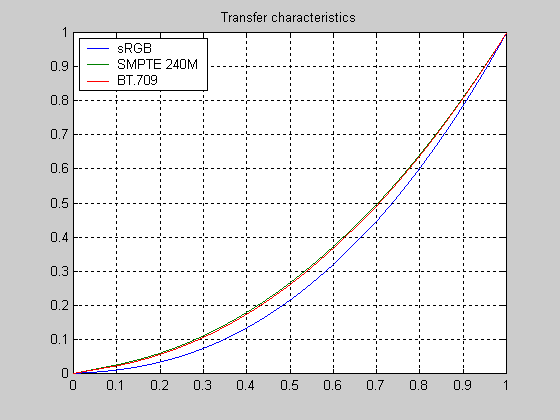
什么意思呢？从黑到标准灰，Y的范围是0~188

从标准灰到白，Y的范围是188~255

亮度较低的颜色区间，被赋予了更多的取值空间；相反，亮度较高的区域，取值空间少。这就是Gamma compression的作用。

更多的取值空间，意味着颜色的表示可以越精细，因为Y每增加/减少1，暗场的亮度变化是小于亮场的。这是符合人眼的视觉特性的（对暗部的亮度变化比亮部敏感）。我们经常说视频里的暗场有什么什么问题，实际上gamma compression在数字信号时代就是为了更好地保留暗场信息、增加暗场的精度而存在的。

下图横轴为Y/RGB的信号大小，纵轴为亮度，可以很明显的看出，在各种transfer characteristics(常见有sRGB， BT.709等——没错，也用这个名称。BT.709严格来说是一套标准，包括color primaries, transfer characteristics和matrix。我们之前说的matrix其实只是BT709/601标准的一部分)的规范下，信号数据和亮度都不是线性关系，而是一个凹形的曲线：



Resizer在做resize的时候，简单的说，如果要把两个像素，分别是0和255，downscale成一个，你可以想象出来的结果应该差不多是128，至少Bilinear这种线性的算法会这么做。Lanczos等多taps的还需要参照它们周边的像素，但是出来的结果也是八九不离十，就是两个像素的平均。

问题是，黑和白结合在一起，应该是标准灰，但是这样出来的128比标准灰来的暗。

为啥，原因是，Resizer的插值基本上是线性的，而Y的值和实际亮度之间的关系不是线性的。

好比一个非线性函数,, 。

线性的操作，不适合用在本身是非线性上的东西。函数值的平均数，不能通过变量的平均数得到；同理，亮度的平均数，不能通过Y值的平均数对应。

那么造成的影响是什么呢？就是图像的高频信息，经过常规resizer downscale后，亮度/对比度会下降。Gamma-aware resize就是在resize的时候，修正这个问题。做法是，将图像转换到线性光下（Y，R，G，B和其对应的亮度、红色、绿色、蓝色呈线性关系），再做resize，然后转回Gamma压缩的效果。

一般来说，只推荐在downscale时候做gamma-aware resize。upscale的时候不用。这跟madVR推荐设置也是一个道理——downscale设置scale in linear light(线性光下做resize，其实就是gamma-aware resize)， upscale设置non-ringing。

下页我们通过一张实际的图进行对比（原图和对应的帖子可以看这里：<http://forum.doom9.org/showthread.php?p=1484392>）

这是原图:



我们把它缩小到1/4大小，用常规resizer spline36:

是不是感觉灯光黯淡了很多？

如果使用Gamma-aware spline36 resize: 

灯光的亮度就和原图非常匹配了。

如何在常规gamma压缩后的视频，和线性光下的视频做转换呢？Dither tools提供了两个工具：

Dither\_y\_gamma\_to\_linear()

Dither\_y\_linear\_to\_gamma()

参数如下：

tv\_range\_in：true/false, 输入的clip是不是tv\_range。

tv\_range\_out：true/false, 输出的clip是不是tv\_range。

curve：你要以什么matrix的曲线来调整。可选”709”, “601”, “sRGB”, “2020”

严格意义上，Dither\_y\_gamma\_to\_linear() 需要选择源端的transfer characteristics，而Dither\_y\_linear\_to\_gamma()则是选择显示器端的transfer characteristics。但是实际上你可以不用管那么多——统一选709就行了，因为那么多选项虽然看上去不同实际效果没啥区别，一般场合下不需要精确的那么细……

这两个函数都是只接受stacked 16bit的数据(stacked YUV 16bit/RGB48Y)。因为Gamma压缩转换，对精度要求极高。8bit的运算会产生很严重的精度不足现象。

于是我们从最简单的开始：一个1080p的原盘，我们用gamma-aware的resize来调整成720p:

LWLibavVideoSource(“00000.m2ts”,threads=1)

U16()

Dither\_y\_gamma\_to\_linear(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

Dither\_resize16(1280,720)

Dither\_y\_linear\_to\_gamma(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

输出的结果就是gamma-aware resize后的stack 16bit 720p。

在vs中，写法是：

gray = core.std.ShufflePlanes(src16, 0, colorfamily=vs.GRAY)

gray = core.fmtc.transfer(gray,transs="709",transd="linear")

gray = core.fmtc.resample(gray,1280,720)

gray = core.fmtc.transfer(gray,transs="linear",transd="709")

UV = core.fmtc.resample(src16,1280,720)

down = core.std.ShufflePlanes([gray,UV],[0,1,2], vs.YUV)

一般来说，如果涉及到RGB，Gamma-aware过程在RGB下做更好（因为日常的YUV模型其实并不能完美抽象出亮度，导致YUV下操作有很小的偏差）。比如一个JPEG图像，我们把它读入，downscale为1280x720的RGB输出（以下脚本可以作为高画质jpeg缩放avs脚本）:

JPEGSource(“xxx.jpg”)

nnedi3\_resize16(output=”RGB48Y”,nns=4,matrix=”601”,tv\_range=false,cplace=”MPEG1”)

Dither\_y\_gamma\_to\_linear(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

Dither\_resize16(1280,720)

Dither\_y\_linear\_to\_gamma(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

#这一步出来的结果是downscale后的RGB48Y。下面只需要转为RGB24

Down10(8,tvrange=false)

MergeRGB(SelectEvery(3,0), SelectEvery(3,1), SelectEvery(3,2))

vs写法：

a="trial.jpg"

src8 = core.lsmas.LWLibavSource(a)

src16 = mvf.ToRGB(src8,cplace="MPEG1",full=False,matrix="601",depth=16)

down = core.fmtc.transfer(src16,transs="601",transd="linear",fulls=True, fulld=True)

down = core.fmtc.resample(down,1280,720)

down = core.fmtc.transfer(down,transs="linear",transd="srgb",fulls=True, fulld=True)

res = mvf.Depth(down,depth=8)

如果输入是RGB24的png/bmp，我们也可以类似写一个高质量缩放脚本：

ImageSource(“xxx.png”)

Interleave(ShowRed(“Y8”), ShowGreen(“Y8”), ShowBlue(“Y8”)).U16(tvrange=false)

Dither\_y\_gamma\_to\_linear(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

Dither\_resize16(1280,720)

Dither\_y\_linear\_to\_gamma(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

Down10(8,tvrange=false)

MergeRGB(SelectEvery(3,0), SelectEvery(3,1), SelectEvery(3,2))

比起常规工具，这个脚本的优势有：高精度，好算法，gamma-aware

5. 用avs模仿madVR的缩放流程

排除各种image-doubling，deband，smooth之类的处理，madVR的缩放流程可以这么理解：

1. 以高精度，将Chroma 用算法A 拉升成Luma的分辨率，算法A可以在Scaling Algorithms - Chroma Upscaling中设置。

2. 将Luma和Chroma 转换为高精度RGB。matrix之类的从视频自带信息中获取。

3. 如果播放分辨率低于源分辨率，对RGB用算法B做downscale，否则用算法C做upscale。算法B和C可以在Scaling Algorithms - Image Downscaling/Upscaling中设置。

4. 将高精度RGB抖动输出。

我们现在不妨模拟一下madVR的过程：

我们输入一个1080p 10bit MKV的视频，解码后为YUV420p10，BT709，TV range, MPEG2的chroma placement；

算法A为softcubic 60；

算法B为Catmull-Rom + Scale in Linear Light；

算法C为 non-ringing Lanczos taps=4；

我们需要在1366x768的笔记本屏幕上播放；

RGB抖动输出，不妨认为效果跟Down10()默认相似。

avs中，虽然有现成的工具可以整合模拟（Dither\_srgb\_display），但是我们还是想用基础滤镜来分步模拟一下：

src16 = LWLibavVideoSource(“xxx.mkv”,format=”yuv420p16”,stacked=true) #尽管是YUV420p10，我们读入就把它转为YUV420p16

Dither\_convert\_yuv\_to\_rgb(lsb\_in=true,a1=0.6,a2=0.4,output=”RGB48Y”) #用softcubic 60做chroma upscale，并转为16bit RGB。注意其他参数的默认值就和设定值是一致的，比如matrix=“709”

Dither\_y\_gamma\_to\_linear(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

Dither\_resize16(1366,768)

Dither\_y\_linear\_to\_gamma(tv\_range\_in=false, tv\_range\_out=false, curve=”709”)

#以上是对RGB48Y做gamma-aware resize

Down10(8,tvrange=false)

MergeRGB(SelectEvery(3,0), SelectEvery(3,1), SelectEvery(3,2))

换个情况：

我们输入一个720p 8bit mp4的视频，解码后为YUV420p8，BT709，TV range, MPEG2的chroma placement；

算法A为nnedi3 nns=4(n=256)；

算法B为Catmull-Rom + Scale in Linear Light；

算法C为 non-ringing Lanczos taps=4；

我们需要在1366x768的笔记本屏幕上播放；

RGB抖动输出，不妨认为效果跟Down10()默认相似。

src16 = LWLibavVideoSource(“xxx.mkv”,format=”yuv420p16”,stacked=true) nnedi3\_resize16(lsb\_in=true,nns=4,output=”RGB48Y”) #用nnedi3\_resize16做chroma upscale，并转为16bit RGB。注意其他参数的默认值就和设定值是一致的，比如matrix=“709”

Dither\_resize16nr(1366,768,kernel=”lanczos”,taps=4)

#以上是对RGB48Y做nonringing-lanczos 4。注意我们没有设置gamma-aware resize

Down10(8,tvrange=false)

MergeRGB(SelectEvery(3,0), SelectEvery(3,1), SelectEvery(3,2))

6. 用Dither\_resize16做逆向upscale

有时候，当片源是劣质upscale而来，你又能判断出upscale的算法，或者退而求其次，观察它upscale的副效果，你是可以做逆向的upscale的（原则上说，是模拟逆向）。这个只需要在Dither\_resize16中设置invks=true就可以。

Point-Resize非常好判断；它的逆向算法就是Point-Resize, 不需要借助Dither\_resize16;

Bilinear算法upscale后，锯齿很多，片子锐度很低，几乎没有什么ringing/haloing。这时候可以Dither\_resize16(1280,720,kernel=”bilinear”,invks=true)

Bicubic算法upscale后，锯齿偏多，片子锐度中等，有少许ringing/haloing。这时候可以Dither\_resize16(1280,720,kernel=”bicubic”,invks=true)

Lanczos/Spline算法upscale后，锯齿中等，片子锐度较高，有较多ringing。这时候可以Dither\_resize16(1280,720,kernel=”spline36”,invks=true)

这样降低到低分辨率之后，再接aa/dering等后续操作，效果要好于直接用常规resizer 缩分辨率。