VCB-Studio教程18 YUV与RGB的互转(1)

本教程旨在讲述avs中RGB相关的知识。

1. avs中视频的类型

我们在avs中，可以载入和编辑视频，但是视频格式各不相同，所以avs中需要对它们进行归类。avs中原生支持 YUV 8bit的视频和RGB8bit的视频，分别有以下类型：

YV12， 即YUV420p8，UV的分辨率是Y的

YV16， 即YUV422p8，UV的分辨率是Y的

YV24， 即YUV444p8，UV的分辨率和Y完全相同。

YUY2， 即YUV422p8另一种组织格式。

Y8， 单独记录黑白视频，只有Y通道。这玩意很适合拆分YUV/RGB平面的时候使用。

RGB24，RGB各8个bit。

RGB32，在RGB24的基础上加上8bit的alpha（透明度）通道。

这些是avs原生支持的视频格式。avs中的原生滤镜，比如Spline36Resize，就支持对这些格式进行处理。处理完的结果自然也是原生格式：

LWLibavVideoSource(“00000.m2ts”,threads=1) #YV12

Spline36Resize(1280,720) #YV12

类似Fraps这类，编码的avi是以RGB组织的，读入的时候就会是RGB24:

AVISource(“fraps.avi”) #RGB24

Spline36Resize(1280,720) # avs自带的resizer都支持RGB下做重采样

ConvertToYV12(matrix=”Rec.709”) #转为YV12

当然，你也可以先转为YV12再做resize:

AVISource(“fraps.avi”) #RGB24

ConvertToYV12(matrix=”Rec.709”) #转为YV12

Spline36Resize(1280,720)

对于RGB源，除非成品为了兼容性因素必须做成YUV420，一般都推荐保留YUV444. 那么可以这么写：

AVISource(“fraps.avi”) #RGB24

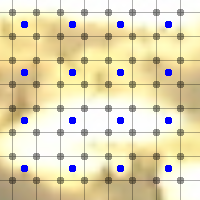
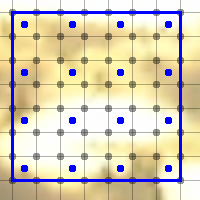
Spline36Resize(1280,720) # avs自带的resizer都支持RGB下做重采样

ConvertToYV24(matrix=”Rec.709”) #转为YV24

x264编码参数中加入 --input-csp “i444” --output-csp “i444”，表示使用YUV 4:4:4编码。x265只需要指定--input-csp "i444"；因为x265不会自己做转换，给什么出什么。

2. YUV模型中Chroma的那些事儿： Chroma Placement (cplace)

我们在科普中讲过，YUV模型中，因为chroma subsampling的使用，绝大多数情况下Chroma是被缩水的。最常见的就是缩水成:

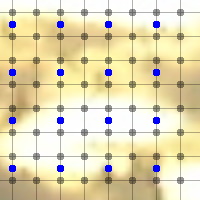
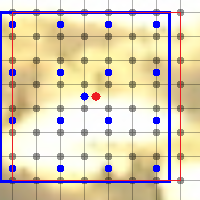
 

如图所示，图中每个黑点代表一个Y，每个蓝点代表一对UV。一共是4个Y对应1对UV。

图上的这种表示方法，每对UV的采样位置，正好在4个Y的中央，横纵都是如此。这种对齐方式，叫做中央对齐(center align)。这种对齐的特点是，将chroma放大到2x2倍之后，图像的中心和Y的中心是重合的（见右图，右图是将UV放大到Y的尺寸后，图像范围）

这种采样位置被称为 MPEG 1 Placement

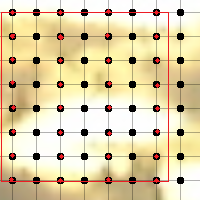
问题是，MPEG2时代，规则就变了。UV的位置，从上下看，依旧是在中间，但是从左右看，UV的位置靠左（即上下保持center align，左右改为left align）：

如果直接把UV放大到Y的尺度（我们常用的resizer都是center align的，这意味着图像放大缩小后，图像的中心点是对齐的），会发现图像相对于Y是有错位的。放大后的错位量是向左溢出了0.5个像素，右边亏缺0.5像素。图像中细红框表示Y覆盖的范围，粗蓝框表示UV放大后覆盖的范围。

两个矩形，中心点的距离是0.5像素（蓝点和红点间的距离，以luma的尺度算。如果按照chroma的尺度是0.25像素）

这是YUV 420的表示方法。YUV 422的表示方法如下(为了显眼，用红色点表示UV位置)：



可以发现，YUV422是纵向上保留全高度的UV，横向宽度上只有1/2. 横向上依旧是左靠齐（如果是MPEG1的中靠齐，那么红点的位置就是在相邻两个黑点的中点上）

YUV422依旧有着left align下，upscale后中心不对其的问题，而且也是向左溢出0.5个像素。

MPEG2的左对齐优势在于，处理插补的时候，可以保留一半插补另一半。以上图YUV422的示意图为例，如果要将UV分辨率拉升到Y的分辨率，所有红点保留，然后设法插补出剩下的一半（就是图中空白黑点所在的位置），就完成了拉升。

3. 利用Resizer做YUV 不同subsampling之间的转换

给你一个YV24的视频，如何帮它转化为YV12呢？

如果是8bit和8bit互转，只需要用avs自带的ConvertTo命令：

ConvertToYV12()

同理，还有ConvertToYV24, ConvertToYV16这种。

几个参数设置：

ChromaInplacement: 输入的cplace, 默认”MPEG2”, 可选”MPEG1”

ChromaOutplacement: 输出的cplace, 默认”MPEG2”, 可选”MPEG1”

chromaresample：chroma 放大缩小算法。默认”bicubic”,可选类似”lanczos”,”spline36”等。

比如一个YV24的视频，我们想转换为cplace为MPEG2的YV12, 用spline36做UV的downscale:

ConvertToYV12(chromaresample=”spline36”, ChromaOutplacement=”MPEG2”)

注意ChromaOutplacement=”MPEG2”可以省略，因为是默认值

如果丢给你的是16bit的视频呢？

这就要求我们自己写转换了。

首先介绍YUV平面拉出来的几个工具：

UtoY8()/UtoY(), 把U平面拉出来做成一个Y8/YV12。如果输出YV12，生成的clip UV都是空的。分辨率是实际U平面的分辨率。比如一个1080p的YUV420视频，UtoY8得到的是一个960x540 的Y8。

VtoY8()/VtoY (), 同上。

ConvertToY8(), 就是把UV平面都丢了，只保留Y平面。

YtoUV(A,B,C) 生成一个全新的YUV视频，Y平面是C的Y，U平面是A的Y，V平面是B的Y。常用于把处理完毕的YUV平面合并。AB的Y平面必须分辨率相同（因为一个视频的UV平面肯定是等分辨率），合并后，根据Y平面和UV平面的分辨率决定是YV12/16/24.

现在，假设我们手上有个1080p stacked 16bit YV24的视频,叫做src16，我们想把它转换为 stacked 16bit YV12, MPEG1格式:

U = src16.UtoY8 (). Dither\_resize16(960,540) #U平面单独拿出来放一个Y8，然后downscale

V = src16.VtoY8(). Dither\_resize16(960,540) #V平面一样。产生的Y8平面依旧是stacked 16bit

YtoUV(U,V,src16) #Y平面直接从src16的Y拿就行了。

vs的写法是：

U = core.std.ShufflePlanes(src16,1,vs.GRAY)

U = core.fmtc.resample(U,960,540)

V = …

res = core.std.ShufflePlanes([src16,U,V],[0,0,0],vs.YUV)

MPEG1的cplace是中心对齐的，而一般的resizer默认也是中心对齐的，所以可以直接downscale，没有问题。最后一步，因为将一对960x540的UV和1920x1080的Y对齐，系统自动判断为YV12。

vs还可以用自带的Resize（用的zimg library）:

res = core.resize.Spline36(src16, format=vs.YUV420P16, chromaloc=center)

注意chromaloc=center不能省略，因为默认是left，即按照MPEG2的来

如果我们要将一个cplace为MPEG1的YV12转换为YV24，方法也是一样：

U = src16.UtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4)

V = src16.VtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4)

YtoUV(U,V,src16) # UV用softcubic 60转换到1080p分辨率，然后和Y合并。

用vs自带的你得这么写：

res = core.resize.Bicubic(src16, format=vs.YUV444P16, chromaloc\_in=center, filter\_param\_a\_uv=0.6, filter\_param\_b\_uv=0.4)

注意输出的chromaloc是不需要指定的，YUV444不存在cplace一说。

如果给你的是MPEG2的YV12呢？

我们提到过，MPEG2的cplace，直接将UV平面拉升，会有向左0.5像素的溢出。

所以拉升之后，还需要将视频左边切割掉0.5的像素，同时右边增补0.5的像素。

一种做法是，resizer之后再接一个resizer做修正：

U = src16.UtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4)

U = U. Dither\_resize16(1920,1080,src\_left=0.5) #左边切割掉0.5像素，同时右边填补0.5像素以保持1080p的分辨率

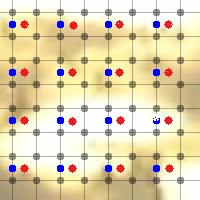
V = src16.VtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4)

V = V. Dither\_resize16(1920,1080,src\_left=0.5)

YtoUV(U,V,src16)

但是这种做法并非最好的。理由是我们做了一次resizer，又做了一次shift。重复计算，对精度损失和效率没有好处。能不能拉升之前就做shift呢？答案是肯定的：

我们只要把采样点位于蓝点处的UV，在红点处重采样就可以，或者说，原来UV的数值是蓝点处的UV，我们要让UV数值变成红点处的UV。



这个重采样的操作，相当于将UV平面左边，切掉一块宽度为红蓝点距离的图像。红蓝点距离，相对于Y的范围来说，是1/2像素，但是对于UV平面来讲，这个距离是相邻两个UV间距离的1/4, 即0.25个像素。换言之，upscale之前，我们要把UV平面左边切掉0.25个像素，右边插补0.25像素：

U = src16.UtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4, src\_left=0.25)

V = src16.VtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4, src\_left=0.25)

YtoUV(U,V,src16)

如果拉升后修复,src\_left=0.5; 如果拉升之前处理, src\_left=0.25.

一般实际操作中，我们选择拉升之前就做处理。这样不但简单高效，而且有个好处：

如果你需要对Y也进行拉升的时候，比如720p的YUV420拉升成1080p的YUV444，UV拉升到1080p后需要做的shift就不止0.5，但是如果拉升前做，shift还是0.25:

U = src16.UtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4, src\_left=0.25)

V = src16.VtoY8(). Dither\_resize16(1920,1080,kernel=”bicubic”,a1=0.6,a2=0.4, src\_left=0.25)

Y = src16.Dither\_resize16nr(1920,1080,kernel=”lanczos”,taps=4)

YtoUV(U,V,Y)

vs下的做法如下：

U = core.std.ShufflePlanes(src16,1,vs.GRAY)

U = core.fmtc.resample(1920,1080,kernel="bicubic",a1=0.6,a2=0.4,sx=0.25)

V = …

res = core.std.ShufflePlanes([src16,U,V],[0,0,0],vs.YUV)

或者vs自带的Resize也可以（用的zimg library）:

res = core.resize.Bicubic(src16, format=vs.YUV444P16, filter\_param\_a\_uv=0.6, filter\_param\_b\_uv=0.4)

这时候chromaloc可以省略。默认就是按照输出是MPEG2的来。

同理，将一个YUV444的视频，转为YUV420 MPEG2的视频：

先故意把UV平面左边插补出0.5像素的溢出，右边删了0.5像素；

再降低成1/4分辨率：

U = src16.UtoY8(). Dither\_resize16(960,540, src\_left=-0.5)

V = src16.VtoY8().Dither\_resize16(960,540, src\_left=-0.5)

YtoUV(U,V,src16)

vs的做法如下：

U = core.std.ShufflePlanes(src16,1,vs.GRAY).fmtc.resample(960,540,sx=-0.5)

V = …

res = core.std.ShufflePlanes([src16,U,V],[0,0,0],vs.YUV)

res = core.resize.Spline36(src16, format=vs.YUV420P16)

UV平面的拉升，一般用softcubic(烂源)、nnedi3(好源)、Bicubic（较为均衡），降低则用Catmull-Rom/spline36比较好。

4. YUV转RGB时候的两大参数：matrix，range。

YUV444的格式，就可以和RGB互转了。因为每个像素，YUV三个数值，经过计算，可以得到同样RGB三个数值；反之，RGB数值通过逆运算，也能得到YUV444的格式。这种计算方法背后的规定就是matrix和range。

首先说matrix，这玩意规定了大体上运算应该怎么进行。比如说最初的BT601 matrix（了解一下就好，不用记住的）

\begin{align}
  Y'  &=&  16 &+& \frac{ 65.738 \cdot R'_D}{256} &+& \frac{129.057 \cdot G'_D}{256} &+& \frac{ 25.064 \cdot B'_D}{256}\\
  C_B &=& 128 &-& \frac{ 37.945 \cdot R'_D}{256} &-& \frac{ 74.494 \cdot G'_D}{256} &+& \frac{112.439 \cdot B'_D}{256}\\
  C_R &=& 128 &+& \frac{112.439 \cdot R'_D}{256} &-& \frac{ 94.154 \cdot G'_D}{256} &-& \frac{ 18.285 \cdot B'_D}{256}
\end{align}

有RGB数据，你就可以用这个公式转换为YCbCr。

反过来：

\begin{align}
  R'_D &=& \frac{298.082 \cdot Y'}{256} &&&+& \frac{408.583 \cdot C_R}{256} &-& 222.921\\
  G'_D &=& \frac{298.082 \cdot Y'}{256} &-& \frac{100.291 \cdot C_B}{256} &-& \frac{208.120 \cdot C_R}{256} &+& 135.576\\
  B'_D &=& \frac{298.082 \cdot Y'}{256} &+& \frac{516.412 \cdot C_B}{256} &&&-& 276.836
\end{align}

这些运算基本上是线性的，所以相当于做了矩阵乘法。matrix就是定义这种矩阵的关键字。常见的有BT601（常用于标清），BT709（常用于高清，也是目前720p/1080p蓝光制作时候我们用的最多的），BT2020（4K时代的标准），YCgCo，等等。

另一个概念叫做range，range是定义YUV数据的范围。以8bit为例：8bit下数据范围是0~255，但是YUV表示的时候并不一定使用了所有的范围。tvrange下，Y的有效范围是16~235,16表示最低亮度，235表示最高亮度；UV的范围是240。跟tvrange相对的是pcrange，永远是0为最低，255为最高。

range的引入使得同一个值可能有不同的含义。以黑白图像为例，比如Y=16，在tvrange下就是最黑的值；但是如果把它看做pcrange，那么它还是有一点灰的。同理，Y=235在tvrange下就是白，而在pcrange下就会是接近白的亮灰色。

一张tvrange的图片被误认为是pcrange，图像的对比度会被压缩，就是颜色没有那么鲜艳；反之，一张pcrange的图被当做是tvrange，图像的对比度会被拉大，颜色似乎变得很鲜艳，然而极亮和极暗的地方会有细节损失（因为0~15,236~255的有效段位数据被丢弃了）

range的概念只存在于YUV下，默认一般是tvrange。RGB下没有range；永远是0代表最低值，255代表最高值。

avs中，convertto这个工具就可以实现YUV和RGB的互转，同时照顾到matrix和range：

AVISource(“fraps.avi”)

ConvertToYV12(matrix=”Rec709”)

就是将RGB的avi转换为BT709 tvrange的YV12

matrix可以有以下值：

Rec601： BT601 tvrange

Rec709： BT709 tvrange

PC.601：BT601 pcrange

PC.709：BT701 pcrange

编码的时候，x264可以指定输入的matrix和range，默认的matrix按照分辨率来，默认的range则是tv。用--matrix和--range指定。

--matrix 可选”BT470bg”/”smpte170m”, 或者”BT709”。前两个是BT601的别名；

--range 一般留空（自动选），否则可以指定 “tv”或者”pc”

所以如果你转为pcrange BT601 YUV444 8bit:

AVISource(“fraps.avi”)

ConvertToYV24(matrix=”PC.601”)

编码的时候记得在misc中指定：--matrix BT470bg --range pc --input-csp “i444” --output-csp ”i444”

同理，从YUV转换到RGB的时候，也需要指定这些参数。比如说JPEG图像的YUV数据格式：

YUV420/422/444都有可能

8bit

BT601 PCrange

MPEG1的cplace

如果你拿到一个JPEG的YUV数据（可以用JPEGSource输入），转换成RGB:

JPEGSource(“xxx.jpg”)

ConvertToRGB24(matrix=”PC.601”, ChromaInPlacement=”MPEG1”)

在现在视频规范下，如果一个视频不指定matrix，编码器应该默认按照分辨率来算：

如果 长>1024 或者 宽>576，则使用BT709；否则使用BT601。

不过其实并不是所有渲染器都这么照做的……不按照分辨率自动设置matrix，甚至设置了matrix也不读取的比比皆是。

5. avs中16bit RGB的伪装

我们在前面的教程中讲过，YUV格式的10bit和16bit，是靠interleaved/stacked 来hack成高bitdepth。YV12是这样，YV16和YV24也是如此：

AVISource(“fraps.avi”) #RGB24

Spline36Resize(1280,720) # avs自带的resizer都支持RGB下做重采样

ConvertToYV24(matrix=”Rec.709”) #转为YV24

U16() # 转为YUV444 stacked 16bit

Dither\_out() #转为YUV444 interleaved 16bit

avs中，也有对RGB的16bit hack。16bit的RGB称为RGB48，就是每个像素占用48个bit。主要的方法有两种： RGB48Y，和RGB48YV12

RGB48Y比较好理解，以一个分辨率为1080p的RGB48Y：

它把每一帧RGB拆成3帧Y8，按顺序分别表示R,G,B；

每个Y8分辨率为1920x2160，是一个stacked 16bit的Y8。以R对应的平面为例：MSB部分（就是上半部分）表示前8bit的R数值；LSB部分（下半部分）表示后8bit的R数值。

RGB48Y这种表示方法的优势在于，它相当于stacked 16bit的RGB，并且伪装成了YUV格式。那么就可以用Dither\_resize16这种只适用于stacked 16bit YUV的滤镜：

JPEGSource(“xxx.jpg”)

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

# 用nnedi3将一张jpeg图像高精度转换为RGB48Y

Dither\_resize16(1280,720)

对图像在RGB下做16bit的resize

RGB48YV12则是另一种伪装：它把RGB48的信息伪装成YV12。我们假设一个2x2的图像：

RGB48模式下，它需要储存4个R，4个G，4个B，每个各16bit，总共是3\*4\*16=192bit；

一个4x4的YV12：

4\*4个Y，每个8bit，总共是4\*4\*8=128bit

2\*2对UV，每个8bit，总共是 2\*2\*2\*8=64bit

总共是128+64=192bit

所以，横纵都是2倍分辨率的YV12视频，正好可以塞下RGB48的信息。只不过这些信息本身作为YV12来看杂乱无章了。

RGB48YV12多用于输出给ImageMagick等绘图工具。实际在avs中一般使用RGB48Y。

16bit RGB的知识我们会在后续教程中继续详解。

6. 高精度的RGB和YUV互转

avs自带的ConvertTo命令，虽然简单方便，功能也基本上齐全，但是毕竟精度低。YUV和RGB的互转，需要大量的计算，我们还是希望能够有原生高精度的工具，支持16bit的输入输出。Dither Tools里面提供了这个工具：

Dither\_convert\_yuv\_to\_rgb()

Dither\_convert\_rgb\_to\_yuv()

从名称看，不难理解它们分别是YUV转RGB，和RGB转YUV的工具。

Dither\_convert\_yuv\_to\_rgb() 接收YUV格式的输入，输出RGB。参数如下：

matrix，可选”601”, ”709”, “2020”, “YCgCo”。如果不输入，自动选择601或者709。视频高度>600则选择709。

tv\_range，true/false. 默认true表示是tvrange。

cplace，可选”MPEG1”和”MPEG2”

chromak，chroma upscaling的kernel。和dither\_resize16的kernel类似。默认bicubic

fh, fv, taps, a1, a2, a3，也是从Dither\_resize16中继承而来。

noring，true/false，是否使用non-ringing 算法。推荐搭配lanczos/spline等使用。

lsb\_in，true/false. 指定输入的YUV是否是stacked 16bit。

mode, ampo, ampn, staticnoise 这些是dither（抖动相关）。下一个章节单独讲述这些。

output，输出的格式。可选”RGB32”(带个空的alpha通道), “RGB24”, “RGB48YV12”,”RGB48Y”。

Dither\_convert\_rgb\_to\_yuv()接受RGB格式，转为YUV。输入是一个RGB24或者RGB32，也可以接受RGB48Y。输入RGB48Y的时候，需要三个参数指定：

# RGB48Y

R = SelectEvery(3,0) #每3帧中取第1帧，为红色通道

G = SelectEvery(3,1) #每3帧中取第2帧，为绿色通道

B = SelectEvery(3,2) #每3帧中取第3帧，为蓝色通道

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

matrix，同上。

tv\_range，同上。表示输出的YUV视频的range

cplace，同上

chromak，chroma downscaling的kernel。和dither\_resize16的kernel类似。默认bicubic

fh, fv, taps, a1, a2, a3，也是从Dither\_resize16中继承而来。

noring，true/false，是否使用non-ringing 算法。一般downscaling chroma就不用了

lsb\_in，true/false. 指定输出的YUV是否是stacked 16bit。

mode, ampo, ampn, staticnoise 这些是dither（抖动相关）。下一个章节单独讲述这些。

output，输出的格式。可选”YV12”,“YV16”, “YV24”,”Y8”。

比如Fraps的视频，我们把它转为YUV444p16的高精度视频，并且丢给x264编码：

AVISource(“fraps.avi”) #RGB24

Dither\_convert\_rgb\_to\_yuv(matrix=”709”, tv\_range=true, output=”YV24”, lsb=true)

Dither\_out() #把yuv444p16 stacked 转为 yuv444p16 interleaved.

x264中编码设置：

--input-csp “i444” --output-csp “i444” --input-depth 16 --matrix “BT709”

x265中省略--output-csp "i444"，建议加上--cbqpoffs 5 --crqpoffs 5

vs中类似的存在是mvf.ToYUV和mvf.ToRGB。对照着doc（源码）很容易看懂用法，非常类似：

res=mvf.ToRGB(JPEGYUV, full=True, cplace="MPEG1", matrix="601", depth=8)，就是把JPEG的YUV，转为8bit RGB。

src16 = mvf.ToYUV(RGBimg, css="420",depth=16)

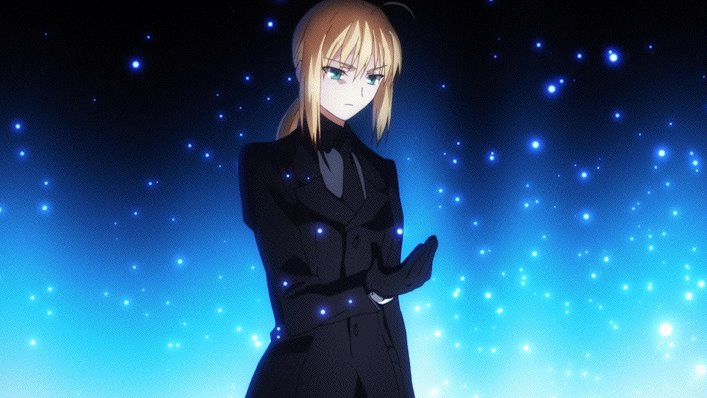
则是把一个RGB, 按照分辨率自适应选取matrix，转为YUV420P16, tvrange的YUV。

7. Dither (抖动)

图像算法中，当做高精度->低精度转换的时候，四舍五入往往不是最佳的选择。比如下图是播放的时候输出4bit的RGB，采用四舍五入：

很可怕吧？低bit下色带问题表现的非常明显

但是如果借助抖动算法，同样是4bit RGB，效果就完全不一样：



抖动算法通过增加抖动噪点，来达到次精度级别的过渡效果。多数dither算法还有误差均摊，将取整后误差值均摊到四周的噪点，来降低图像取整的偏差。对于YUV 8bit这种容易出现精度问题的，调节Dither 参数尤为重要。Dither Tools里面，很多涉及转换的，例如DitherPost(), Down10(), Dither\_convert\_xxx(),都有给你设置抖动的参数：

mode： 抖动的算法。可选：

-1： 不做抖动，四舍五入

0： 8bit ordered dither+噪点。ordered dither是非常常用的算法之一，它产生的噪点规律不易被有损压缩破坏，所以在视频和图像处理中广泛使用。这个我们后续再说。

1: 1bit抖动

2：2bit抖动，轻微

3：2bit抖动，中等

4：2bit抖动，较强

5：2bit抖动，很强

6：fs dither 误差均摊，非常均衡的算法。也一般是默认值。

7：Stucki dither 误差均摊，看上去很锐利，对细微线条保护的较好。

8：Atkinson dither 误差均摊，噪点规律很独特，但是平面处很干净。

一般需要特别处理YUV 8bit的时候选0，否则选6（默认）

ampo：ordered dither或者其他误差均摊算法的强度，这个值越高，ordered dither噪点的规律就越明显，或者其他误差均摊的算法的噪点规律也越突出。范围0~10.0

ampn: 噪点的强度。在执行抖动算法前，还可以加上一层随机噪点，来强化噪点的力度，并且让抖动噪点的分布更均匀。ampn就是控制这个力度的。范围0~4.0，默认0，表示不加入随机噪点。

staticnoise：如果加上随机噪点(ampn控制)，是否加入静态噪点。加入静态噪点可以在压制的时候省一些码率（因为噪点的时间复杂度降低），但是容易对观看产生负面效果——画面在动，噪点不动，看上去很像屏幕上一层灰……默认false，表示不用静态噪点。

转RGB24/YUV 10bit, 绝大多数情况下，默认的参数就已经足够好了。

8. 其他一些值得注意的知识点

如果图像处理中，你拿到的是原生RGB的图像，或者你某一步必须经过RGB处理，再转回YUV编码的时候，优先选择YUV444，保留全部的chroma信息。

比如说原图（左），注意右上角极红/极蓝的logo，和右下角进度条：

上图右是把它转为YUV420p16，然后再转回RGB24，用non-ringing lanczos 4（非常锐利的算法）做chroma upscaling。不难看出，类似上方logo，和下方文字、边框等地方，颜色损失非常明显。这是chroma subsampling必然的代价——UV平面的高频信息会遭到毁灭性的打击。这在原生RGB源，尤其是游戏视频中非常常见。

保留YUV422则好一点（下图左），保留YUV444则最佳（下图右）

所以，除非兼容性问题，尽量保持较高的chroma分辨率。

x26x输入可以选--input-csp（输入的sampling） 和 --output-csp（编码输出的sampling, x265不可指定）。分别如下：

“i420”，默认，YUV420

“i422”, YUV422

“i444”, YUV444

如果输入输出不一样，x264会自己做转换。但是效果不是很好，所以不建议丢给x264转换。

如果--output-csp 是 i444，x264会将chroma-qp-offset加5。这是为了让chroma编码的更烂一点，以节省码率。事实上我们说过，chroma的重要性低于luma，所以这么做是有利于压缩率的。

宁可让x264编码的烂一点，也不要暴力的直接缩减成422甚至420，因为x264编码的损失要小于暴力的缩小再放大。同理，x265也建议手动调整offset

x26x编码中可以用--chroma-loc来设定chroma placement，但是很多播放器根本不认。所以丢给x26x的东西永远保持MPEG2的cplace吧。就算是MPEG1的，你自己给转成MPEG2就好了。只需要把UV平面分离后，用src\_left=-0.25在UV平面左侧插补出0.25像素的空间，或者说向右shift 0.25像素。关于Chroma shift的修正，我们下一章节再说。