[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。

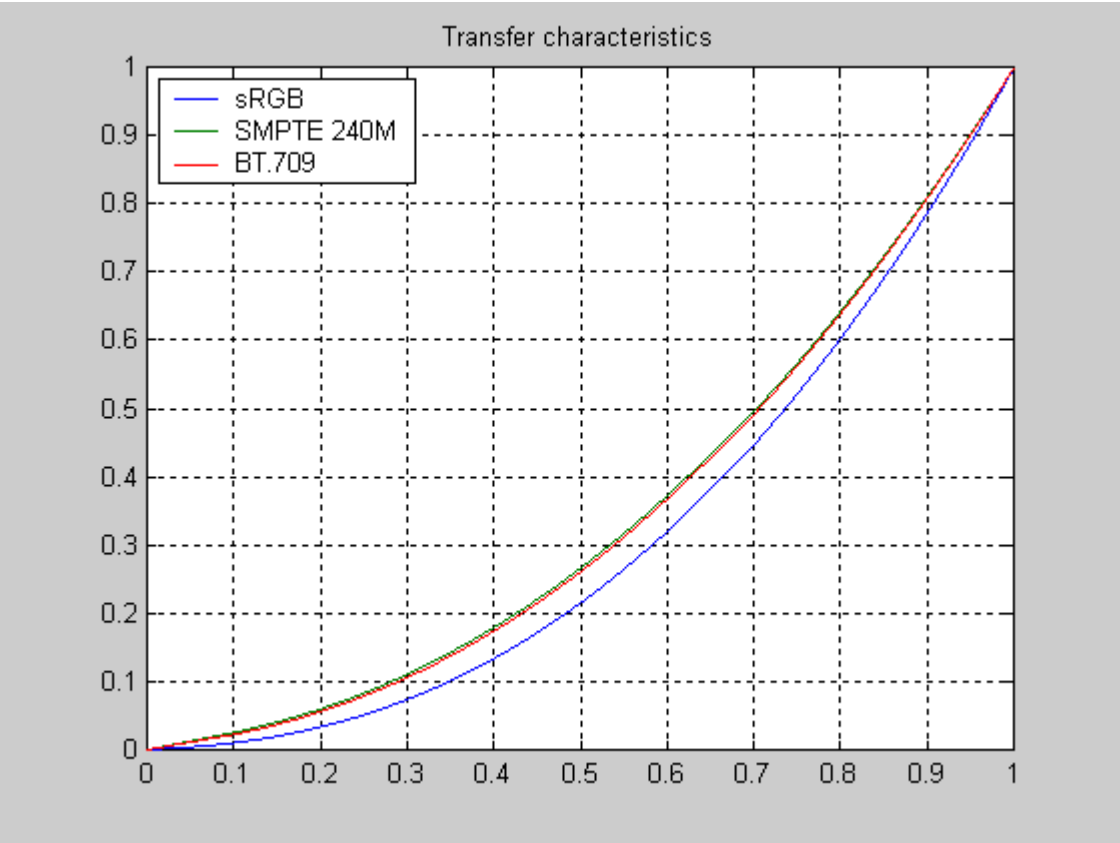
4. Gamma-aware resize 的基础知识

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 $f(x)=x^2$，$\frac{f(0)+f(2)}{2}=\frac{0+4}{2}=2$，$f(\frac{0+2}{2})=f(1)=1\neq 2$。
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。