**1 前言**

        自然界的颜色千变万化，为了给颜色一个量化的衡量标准，就需要建立色彩空间模型来描述各种各样的颜色，由于人对色彩的感知是一个复杂的生理和心理联合作用 的过程，所以在不同的应用领域中为了更好更准确的满足各自的需求，就出现了各种各样的色彩空间模型来量化的描述颜色。我们比较常接触到的就包括 RGB / CMYK / YIQ / YUV / HSI等等。

        对于数字电子多媒体领域来说，我们经常接触到的色彩空间的概念，主要是RGB , YUV这两种（实际上，这两种体系包含了许多种具体的颜色表达方式和模型，如sRGB, Adobe RGB, YUV422, YUV420 …）, RGB是按三基色加光系统的原理来描述颜色，而YUV则是按照 亮度，色差的原理来描述颜色。

        即使只是RGB YUV这两大类色彩空间，所涉及到的知识也是十分丰富复杂的，自知不具备足够的相关专业知识，所以本文主要针对工程领域的应用及算法进行讨论。

**2 YUV相关色彩空间模型**

**2.1 YUV 与 YIQ YcrCb**

        对于YUV模型，实际上很多时候，我们是把它和YIQ / YCrCb模型混为一谈的。

        实际上,YUV模型用于PAL制式的电视系统，Y表示亮度，UV并非任何单词的缩写。

        YIQ模型与YUV模型类似，用于NTSC制式的电视系统。YIQ颜色空间中的I和Q分量相当于将YUV空间中的UV分量做了一个33度的旋转。

        YCbCr颜色空间是由YUV颜色空间派生的一种颜色空间，主要用于数字电视系统中。从RGB到YCbCr的转换中，输入、输出都是8位二进制格式。

        三者与RGB的转换方程如下：

        RGB -> YUV：

        实际上也就是：

*Y=0.30R+0.59G+0.11B ， U=0.493(B－Y) ， V=0.877(R－Y)*

        RGB -> YIQ：

        RGB -> YCrCb：

        从公式中，我们关键要理解的一点是，UV / CbCr信号实际上就是蓝色差信号和红色差信号，进而言之，实际上一定程度上间接的代表了蓝色和红色的强度，理解这一点对于我们理解各种颜色变换处理的过程会有很大的帮助。

        我们在数字电子多媒体领域所谈到的YUV格式，实际上准确的说，是以YcrCb色彩空间模型为基础的具有多种存储格式的一类颜色模型的家族（包括 YUV444 / YUV422 / YUV420 / YUV420P等等）。并不是传统意义上用于PAL制模拟电视的YUV模型。这些YUV模型的区别主要在于UV数据的采样方式和存储方式，这里就不详述。

        而在Camera Sensor中，最常用的YUV模型是 YUV422格式，因为它采用4个字节描述两个像素，能和RGB565模型比较好的兼容。有利于Camera Sensor和Camera controller的软硬件接口设计。

**3 YUV2RGB快速算法分析**

        这里指的YUV实际是YcrCb了 8  ) YUV2RGB的转换公式本身是很简单的，但是牵涉到浮点运算，所以，如果要实现快速算法，算法结构本身没什么好研究的了，主要是采用整型运算或者查表来加快计算速度。  
首先可以推导得到转换公式为：

*R = Y + 1.4075 \*（V-128）  
        G = Y – 0.3455 \*（U –128） – 0.7169 \*（V –128）  
        B = Y + 1.779 \*（U – 128）*

**3.1 整型算法**

       要用整型运算代替浮点运算，当然是要用移位的办法了，我们可以很容易得到下列算法：

*u = YUVdata[UPOS] - 128;  
        v = YUVdata[VPOS] - 128;*

*rdif = v + ((v \* 103) >> 8);  
        invgdif = ((u \* 88) >> 8) +((v \* 183) >> 8);  
        bdif = u +( (u\*198) >> 8);*

*r = YUVdata[YPOS] + rdif;  
        g = YUVdata[YPOS] - invgdif;  
        b = YUVdata[YPOS] + bdif;*

为了防止出现溢出，还需要判错计算的结果是否在0-255范围内，做类似下面的判断。

*if (r>255)  
            r=255;  
        if (r<0)  
            r=0;*

        要从RGB24转换成RGB565数据还要做移位和或运算：

*RGBdata[1] =( (r & 0xF8)  | ( g >> 5) );  
        RGBdata[0] =( ((g & 0x1C) << 3) | ( b >> 3) );*

**3.2 部分查表法**

        查表法首先可以想到的就是用查表替代上述整型算法中的乘法运算。

*rdif = fac\_1\_4075[u];  
        invgdif = fac\_m\_0\_3455[u] + fac\_m\_0\_7169[v];  
        bdif = fac\_1\_779[u];*

        这里一共需要4个1维数组，下标从0开始到255，表格共占用约1K的内存空间。uv可以不需要做减128的操作了。在事先计算对应的数组元素的值的时候计算在内就好了。

        对于每个像素，部分查表法用查表替代了2次减法运算和4次乘法运算，4次移位运算。但是，依然需要多次加法运算和6次比较运算和可能存在的赋值操作，相对第一种方法运算速度提高并不明显。

**3.3 完全查表法**

        那么是否可以由YUV直接查表得到对应的RGB值呢？乍一看似乎不太可能，以最复杂的G的运算为例，因为G与YUV三者都相关，所以类似 G=YUV2G[Y][U][V]这样的算法，一个三维下标尺寸都为256的数组就需要占用2的24次方约16兆空间，绝对是没法接受的。所以目前多数都 是采用部分查表法。

        但是，如果我们仔细分析就可以发现，对于G我们实际上完全没有必要采用三维数组，因为Y只与UV运算的结果相关，与UV的个体无关，所以我们可以采用二次查表的方法将G的运算简化为对两个二维数组的查表操作，如下：

*G = yig2g\_table[ y ][ uv2ig\_table[ u ][ v ] ]；*

        而RB本身就只和YU或YV相关，所以这样我们一共需要4个8\*8的二维表格，需要占用4乘2的16次方共256K内存。基本可以接受。但是对于手机这样的嵌入式运用来说，还是略有些大了。

        进一步分析，我们可以看到，因为在手机等嵌入式运用上我们最终是要把数据转换成RGB565格式送到LCD屏上显示的，所以，对于RGB三分量来说，我们 根本不需要8bit这么高的精度，为了简单和运算的统一起见，对每个分量我们其实只需要高6bit的数据就足够了，所以我们可以进一步把表格改为4个 6\*6的二维表格，这样一共只需要占用16K内存！在计算表格元素值的时候还可以把最终的溢出判断也事先做完。最后的算法如下：

*y = (YUVdata[Y1POS] >> 2);  
        u = (YUVdata[UPOS] >> 2);  
        v = (YUVdata[VPOS] >> 2);  
  
        r = yv2r\_table[ y ][ v ];  
        g = yig2g\_table[ y ][ uv2ig\_table[ u ][ v ] ];  
        b = yu2b\_table[ y ][ u ];  
   
        RGBdata[1] =( (r & 0xF8)  | ( g >> 5) );  
        RGBdata[0] =( ((g & 0x1C) << 3) | ( b >> 3) );*

        这样相对部分查表法，我们增加了3次移位运算，而进一步减少了4次加法运算和6次比较赋值操作。

        在计算表格元素数值的时候，要考虑舍入和偏移等因数使得计算的中间结果满足数组下标非负的要求，需要一定的技巧。

        采用完全查表法，相对于第一种算法，最终运算速度可以有比较明显的提高，具体性能能提高多少，要看所在平台的CPU运算速度和内存存取速度的相对比例。内 存存取速度越快，用查表法带来的性能改善越明显。在我的PC上测试的结果性能大约能提高35%。而在某ARM平台上测试只提高了约15%。

**3.4 进一步的思考**

        实际上，上述算法：

*RGBdata[1] =( (r & 0xF8)  | ( g >> 5) );  
        RGBdata[0] =( ((g & 0x1C) << 3) | ( b >> 3) );*

        中的 (r & 0xF8) 和 ( b >> 3) 等运算也完全可以在表格中事先计算出来。另外，YU / YV的取值实际上不可能覆盖满6\*6的范围，中间有些点是永远取不到的无输入，RB的运算也可以考虑用5\*5的表格。这些都可能进一步提高运算的速度，减 小表格的尺寸。

        另外，在嵌入式运用中，如果可能尽量将表格放在高速内存如SRAM中应该比放在SDRAM中更加能发挥查表法的优势。

**4 RGB2YUV ?**

        目前觉得这个是没法将3维表格的查表运算化简为2维表格的查表运算了。只能用部分查表法替代其中的乘法运算。

        另外，多数情况下，我们需要的还是YUV2RGB的转换，因为从Sensor得到的数据通常我们会用YUV数据，此外JPG和MPEG实际上也是基于YUV格式编码的，所以要显示解码后的数据需要的也是YUV2RGB的运算 8 ）运气运气

1. **1.  static void ccvt\_420p\_rgb565(int width, int height, const unsigned char \*src, \_\_u16 \*dst)**
2. **2.  {**
3. **3.  int line, col, linewidth;**
4. **4.  int y, u, v, yy, vr, ug, vg, ub;**
5. **5.  int r, g, b;**
6. **6.  const unsigned char \*py, \*pu, \*pv;**
7. **7.**
8. **8.  linewidth = width >> 1;**
9. **9.  py = src;**
10. **10. pu = py + (width \* height);**
11. **11. pv = pu + (width \* height) / 4;**
12. **12.**
13. **13. y = \*py++;**
14. **14. yy = y << 8;**
15. **15. u = \*pu - 128;**
16. **16. ug =   88 \* u;**
17. **17. ub = 454 \* u;**
18. **18. v = \*pv - 128;**
19. **19. vg = 183 \* v;**
20. **20. vr = 359 \* v;**
21. **21.**
22. **22. for (line = 0; line < height; line++) {**
23. **23.    for (col = 0; col < width; col++) {**
24. **24.     r = (yy +      vr) >> 8;**
25. **25.     g = (yy - ug - vg) >> 8;**
26. **26.     b = (yy + ub     ) >> 8;**
27. **27.**
28. **28.     if (r < 0)   r = 0;**
29. **29.     if (r > 255) r = 255;**
30. **30.     if (g < 0)   g = 0;**
31. **31.     if (g > 255) g = 255;**
32. **32.     if (b < 0)   b = 0;**
33. **33.     if (b > 255) b = 255;**
34. **34.    \*dst++ = (((\_\_u16)r>>3)<<11) | (((\_\_u16)g>>2)<<5) | (((\_\_u16)b>>3)<<0);**
35. **35.**
36. **36.     y = \*py++;**
37. **37.     yy = y << 8;**
38. **38.     if (col & 1) {**
39. **39.      pu++;**
40. **40.      pv++;**
41. **41.**
42. **42.      u = \*pu - 128;**
43. **43.      ug =   88 \* u;**
44. **44.      ub = 454 \* u;**
45. **45.      v = \*pv - 128;**
46. **46.      vg = 183 \* v;**
47. **47.      vr = 359 \* v;**
48. **48.     }**
49. **49.    } /\* ..for col \*/**
50. **50.    if ((line & 1) == 0) { // even line: rewind**
51. **51.     pu -= linewidth;**
52. **52.     pv -= linewidth;**
53. **53.    }**
54. **54. } /\* ..for line \*/**
55. **55. }**

**YUV422 ---->>RGB565**

1. **#define YCbCrtoR(Y,Cb,Cr) (1000\*Y + 1371\*(Cr-128))/1000**
2. **#define YCbCrtoG(Y,Cb,Cr) (1000\*Y - 336\*(Cb-128) - 698\*(Cr-128))/1000**
3. **#define YCbCrtoB(Y,Cb,Cr) (1000\*Y + 1732\*(Cb-128))/1000**
4. **#define min(x1, x2)   (((x1)<(x2))?(x1):(x2))**
6. **\_\_u32 Conv\_YCbCr\_Rgb(\_\_u8 y0, \_\_u8 y1, \_\_u8 cb0, \_\_u8 cr0)**
7. **{**
8. **// bit order is**
9. **// YCbCr = [Cr0 Y1 Cb0 Y0], RGB=[R1,G1,B1,R0,G0,B0].**
11. **int r0, g0, b0, r1, g1, b1;**
12. **\_\_u16 rgb0, rgb1;**
13. **\_\_u32 rgb;**
15. **#if 1 // 4 frames/s @192MHz, 12MHz ; 6 frames/s @450MHz, 12MHz**
16. **r0 = YCbCrtoR(y0, cb0, cr0);**
17. **g0 = YCbCrtoG(y0, cb0, cr0);**
18. **b0 = YCbCrtoB(y0, cb0, cr0);**
19. **r1 = YCbCrtoR(y1, cb0, cr0);**
20. **g1 = YCbCrtoG(y1, cb0, cr0);**
21. **b1 = YCbCrtoB(y1, cb0, cr0);**
22. **#endif**
24. **if (r0>255 ) r0 = 255;**
25. **if (r0<0) r0 = 0;**
26. **if (g0>255 ) g0 = 255;**
27. **if (g0<0) g0 = 0;**
28. **if (b0>255 ) b0 = 255;**
29. **if (b0<0) b0 = 0;**
31. **if (r1>255 ) r1 = 255;**
32. **if (r1<0) r1 = 0;**
33. **if (g1>255 ) g1 = 255;**
34. **if (g1<0) g1 = 0;**
35. **if (b1>255 ) b1 = 255;**
36. **if (b1<0) b1 = 0;**
38. **// 5:6:5 16bit format**
39. **rgb0 = (((\_\_u16)r0>>3)<<11) | (((\_\_u16)g0>>2)<<5) | (((\_\_u16)b0>>3)<<0); //RGB565.**
40. **rgb1 = (((\_\_u16)r1>>3)<<11) | (((\_\_u16)g1>>2)<<5) | (((\_\_u16)b1>>3)<<0); //RGB565.**
42. **rgb = (rgb1<<16) | rgb0;**
44. **return(rgb);**
45. **}**

在整个视频行业中，定义了很多 YUV 格式，我以UYVY格式标准来说明，4:2:2 格式UYVY每像素占16 位，UYVY字节顺序如下图：

(图3 UYVY字节顺序)

其中第一个字节为U0，每二个字节为Y0，依次排列如下：

[U0，Y0，V0，Y1] [U1，Y2，V1，Y3] [U2，Y4，V2，Y5] ……

经过仔细分析，我们要实现RGB转YUV格式的话，一个像素的RGB占用三个节，而UYVY每像素占用两个字节，在演示中直接把UYVY字节信息保存到\*.pal格式中（这是我自己写来测试用的^\_^），\*.pal格式字节顺序是先保存上场像素，接着保存下场像素，如果是720x576的一张图像转换为YUV格式并保存的话，文件大小应该是829,440字节(720\*576\*2)。您可以执行本文附带的程序 (功能菜单->转换并写入YUV两场) 查看转换过程。

关于YUV转换为RGB公式，我直接使用一篇文章提供的公式，经过思考，我发觉要想实现准确无误的把YUV还原为原有的RGB图像很难实现，因为我从UYVY的字节顺序来分析没有找到反变换的方法（您找到了记得告诉我哟： [liyingjiang@21cn.com](mailto:liyingjiang@21cn.com) ），例如我做了一个简单的分析，假设有六个像素的UYVY格式，要把这12个字节的UYVY要转换回18个字节的RGB，分析如下：

12个字节的UYVY排列方式：

[U0 Y0 V0 Y1] [U1 Y2 V1 Y3] [U2 Y4 V2 Y5]

完全转换为18个字节的RGB所需的UYVY字节排列如下：

[Y0 U0 V0] [Y1 U1 V1] [Y2 U2 V2] [Y3 U3 V3] [Y4 U4 V4] [Y5 U5 V5]

我们可以看到，12个字节的UYVY无法实现，缺少U3 V3 U4 V4。于是我抛开准确无误地把UYVY转换回RGB的想法，直接使用最近的UV来执行转换，结果发觉转换回来的RGB图像用肉眼根本分辩不出原有RGB图像与反变换回来的RGB图像差别，您可以执行本文附带的程序 (功能菜单->读取YUV并显示) 查看效果，下面是反变换公式和代码的实现：

// 反变换公式

 R= 1.0Y + 0 +1.402(V-128)

 G= 1.0Y - 0.34413 (U-128)-0.71414(V-128)

 B= 1.0Y + 1.772 (U-128)+0

代码实现：

void CRGB2YUVView::YUV2RGB(byte \*pRGB, byte \*pYUV)

{

    byte y, u, v;

    y = \*pYUV; pYUV++;

    u = \*pYUV; pYUV++;

    v = \*pYUV;

    \*pRGB = static\_cast<byte>(1.0\*y + 8 + 1.402\*(v-128));    pRGB++;                 // r

    \*pRGB = static\_cast<byte>(1.0\*y - 0.34413\*(u-128) - 0.71414\*(v-128)); pRGB++;   // g

    \*pRGB = static\_cast<byte>(1.0\*y + 1.772\*(u-128) + 0);                            // b

}

// 读取PAL文件转换为RGB并显示

void CRGB2YUVView::OnReadPAL()

{

    // TODO: Add your command handler code here

    CDC \*pDC = GetDC();

    CRect rect;

    CBrush brush(RGB(128,128,128));

    GetClientRect(&rect);

    pDC->FillRect(&rect, &brush);

    // PAL 720x576 : 中国的电视标准为PAL制

    int CurrentXRes = 720;

    int CurrentYRes = 576;

    int size        = CurrentXRes \* CurrentYRes;

    // 分配内存

    byte \*Video\_Field0 = (byte\*)malloc(CurrentXRes\*CurrentYRes);

    byte \*Video\_Field1 = (byte\*)malloc(CurrentXRes\*CurrentYRes);

    // 保存内存指针

    byte \*Video\_Field0\_ = Video\_Field0;

    byte \*Video\_Field1\_ = Video\_Field1;

    // 初始化内存

    ZeroMemory(Video\_Field0, CurrentXRes\*CurrentYRes);

    ZeroMemory(Video\_Field1, CurrentXRes\*CurrentYRes);

    byte yuv\_y0, yuv\_u0, yuv\_v0; // yuv\_v1; // {y0, u0, v0, v1};

    byte r, g, b;

    byte bufRGB[3]; // 临时保存{R,G,B}

    byte bufYUV[3]; // 临时保存{Y,U,V}

    // 初始化数组空间

    memset(bufRGB,0, sizeof(byte)\*3);

    memset(bufYUV,0, sizeof(byte)\*3);

    char strFileName[MAX\_PATH]="720bmp.pal";

    // 分配图片像素内存

    RGBTRIPLE \*rgb;

    rgb = new RGBTRIPLE[CurrentXRes\*CurrentYRes];

    memset(rgb,0, sizeof(RGBTRIPLE)\*CurrentXRes\*CurrentYRes); // 初始化内存空间

    CFile\* f;

    f = new CFile();

    f->Open(strFileName, CFile::modeRead);

    f->SeekToBegin();

    f->Read(Video\_Field0, CurrentXRes\*CurrentYRes);

    f->Read(Video\_Field1, CurrentXRes\*CurrentYRes);

    // 上场 (1,3,5,7...行)

    for ( int i = CurrentYRes-1; i>=0; i--) {

        for ( int j = 0; j<CurrentXRes; j++) {

            if(!(i%2)==0)

            {

                // UYVY标准 [U0 Y0 V0 Y1] [U1 Y2 V1 Y3] [U2 Y4 V2 Y5] 每像素点两个字节，[内]为四个字节

                if ((j%2)==0)

                {

                    yuv\_u0 = \*Video\_Field0;

                    Video\_Field0++;

                }

                else

                {

                    yuv\_v0 = \*Video\_Field0;

                    Video\_Field0++;

                }

                yuv\_y0 = \*Video\_Field0;

                Video\_Field0++;

                bufYUV[0] = yuv\_y0; // Y

                bufYUV[1] = yuv\_u0; // U

                bufYUV[2] = yuv\_v0; // V

                // RGB转换为YUV

                YUV2RGB(bufRGB,bufYUV);

                r = bufRGB[0];   // y

                g = bufRGB[1];   // u

                b = bufRGB[2];   // v

                if (r>255) r=255; if (r<0) r=0;

                if (g>255) g=255; if (g<0) g=0;

                if (b>255) b=255; if (b<0) b=0;

                for (int k=0; k<1000; k++) ; //延时

                // 视图中显示

                pDC->SetPixel(j, CurrentYRes-1-i, RGB(r, g, b));

            }// end if i%2

        }

    }

    // 下场 (2,4,6,8...行)

    for ( int i\_ = CurrentYRes-1; i\_>=0; i\_--) {

        for ( int j\_ = 0; j\_<CurrentXRes; j\_++) {

            if((i\_%2)==0)

            {

                // UYVY标准 [U0 Y0 V0 Y1] [U1 Y2 V1 Y3] [U2 Y4 V2 Y5] 每像素点两个字节，[内]为四个字节

                if ((j\_%2)==0)

                {

                    yuv\_u0 = \*Video\_Field1;

                    Video\_Field1++;

                }

                else

                {

                    yuv\_v0 = \*Video\_Field1;

                    Video\_Field1++;

                }

                yuv\_y0 = \*Video\_Field1;

                Video\_Field1++;

                bufYUV[0] = yuv\_y0; // Y

                bufYUV[1] = yuv\_u0; // U

                bufYUV[2] = yuv\_v0; // V

                // RGB转换为YUV

                YUV2RGB(bufRGB,bufYUV);

                r = bufRGB[0];   // y

                g = bufRGB[1];   // u

                b = bufRGB[2];   // v

                if (r>255) r=255; if (r<0) r=0;

                if (g>255) g=255; if (g<0) g=0;

                if (b>255) b=255; if (b<0) b=0;

                for (int k=0; k<1000; k++) ; //延时

                // 视图中显示

                pDC->SetPixel(j\_, CurrentYRes-1-i\_, RGB(r, g, b));

            }

        }

    }

    // 提示完成

    char buffer[80];

    sprintf(buffer,"完成读取PAL文件：%s ", strFileName);

    MessageBox(buffer, "提示信息", MB\_OK | MB\_ICONINFORMATION);

    // 关闭PAL电视场文件

    f->Close();

    // 释放内存

    free( Video\_Field0\_ );

    free( Video\_Field1\_ );

    delete f;

    delete rgb;

}

<http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx>

Gary Sullivan and Stephen Estrop  
Microsoft Digital Media Division

April 2002

Updated August 2003

Applies To:  
   Microsoft® Windows®, Microsoft DirectShow®

**Summary:** This article describes the 8-bit YUV formats that are recommended for video rendering in the Microsoft Windows operating system. This article presents techniques for converting between YUV and RGB formats, and also provides techniques for upsampling YUV formats. This article is intended for anyone working with YUV video decoding or rendering in Windows. (13 printed pages)

#### Introduction

Numerous YUV formats are defined throughout the video industry. This article identifies the 8-bit YUV formats that are recommended for video rendering in the Microsoft® Windows® operating system. Decoder vendors and display vendors are encouraged to support the formats described in this article. This article does not address other uses of YUV color, such as still photography.

The formats described in this article all use 8 bits per pixel location to encode the Y channel (also called the luma channel) and use 8 bits per sample to encode each U or V chroma sample. However, most YUV formats use fewer than 24 bits per pixel on average, because they contain fewer samples of U and V than of Y. This article does not cover YUV formats with 10-bit and 12-bit Y channels.

**Note**   For the purposes of this article, the term U is equivalent to Cb, and the term V is equivalent to Cr.

This article covers the following topics:

* [Identifying YUV Formats in DirectShow](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_identifying_yuv_formats_in_directshow) — Explains how to describe Microsoft DirectShow® YUV format types.
* [YUV Sampling](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_yuvsampling) — Describes the most common YUV sampling techniques.
* [Surface Definitions](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_surface_definitions) — Describes the recommended YUV formats.
* [Color Space and Chroma Sampling Rate Conversions](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_colorspaceconversions) — Provides guidelines for converting between YUV and RGB formats, and for converting between different YUV formats.
* [Additional Information](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_additionalinformation) Provides additional information.

#### Identifying YUV Formats in DirectShow

Each of the YUV formats described in this article has an assigned FOURCC code. A FOURCC code is a 32-bit unsigned integer that is created by concatenating four ASCII characters.

There are various C/C++ macros that make it easier to declare FOURCC values in source code. For example, the **MAKEFOURCC** macro is declared in Mmsystem.h, and the **FCC** macro is declared in Aviriff.h. Use them as follows:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_84516714-2fa1-4e9c-8ea7-f4495975fecf');" \o "Copy to clipboard.)

DWORD fccYUY2 = MAKEFOURCC('Y','U','Y','2');

DWORD fccYUY2 = FCC('YUY2');

You can also declare a FOURCC code directly as a character literal simply by reversing the order of the characters. For example:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_cd0d5499-2093-4125-bbd7-6e4ea02b5364');)

DWORD fccYUY2 = '2YUY'; // Declares the FOURCC 'YUY2'

Reversing the order is necessary because the Windows operating system uses a little-endian architecture. 'Y' = 0x59, 'U' = 0x55, and '2' = 0x32, so '2YUY' is 0x32595559.

In DirectShow, formats are identified by a major-type globally unique identifier (GUID) and a subtype GUID. The major type for computer video formats is always MEDIATYPE\_Video. The subtype can be constructed by mapping the FOURCC code to a GUID, as follows:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_e70f655e-58d9-49a1-85bb-8dd6036afcd8');)

XXXXXXXX-0000-0010-8000-00AA00389B71

where *XXXXXXXX* is the FOURCC code. Thus, the subtype GUID for YUY2 is:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_30a2510d-9a4b-41f2-8320-355a9ee2c64b');)

32595559-0000-0010-8000-00AA00389B71

Many of these GUIDs are defined already in the header file Uuids.h. For example, the YUY2 subtype is defined as MEDIASUBTYPE\_YUY2. The DirectShow base class library also provides a helper class, **FOURCCMap**, which can be used to convert FOURCC codes into GUID values. The **FOURCCMap** constructor takes a FOURCC code as an input parameter. You can then cast the **FOURCCMap** object to the corresponding GUID:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_0c534d13-b097-4555-bf0d-b5ebd1a69015');)

FOURCCMap fccMap(FCC('YUY2'));

GUID g1 = (GUID)fccMap;

// Equivalent:

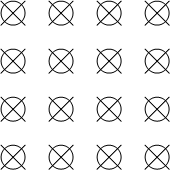
GUID g2 = (GUID)FOURCCMap(FCC('YUY2'));

#### YUV Sampling

One of the advantages of YUV is that the chroma channels can have a lower sampling rate than the Y channel without a dramatic degradation of the perceptual quality. A notation called the *A:B:C* notation is used to describe how often U and V are sampled relative to Y:

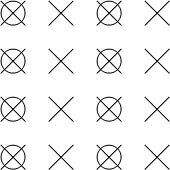
* 4:4:4 means no downsampling of the chroma channels.
* 4:2:2 means 2:1 horizontal downsampling, with no vertical downsampling. Every scan line contains four Y samples for every two U or V samples.
* 4:2:0 means 2:1 horizontal downsampling, with 2:1 vertical downsampling.
* 4:1:1 means 4:1 horizontal downsampling, with no vertical downsampling. Every scan line contains four Y samples for every U or V sample. 4:1:1 sampling is less common than other formats, and is not discussed in detail in this article.

Figure 1 shows the sampling grid used in 4:4:4 pictures. Luma samples are represented by a cross, and chroma samples are represented by a circle.



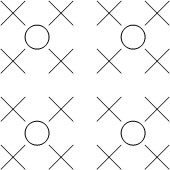
**Figure 1. YUV 4:4:4 sample positions**

The dominant form of 4:2:2 sampling is defined in ITU-R Recommendation BT.601. Figure 2 shows the sampling grid defined by this standard.

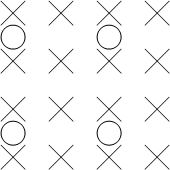


**Figure 2. YUV 4:2:2 sample positions**

There are two common variants of 4:2:0 sampling. One of these is used in MPEG-2 video, and the other is used in MPEG-1 and in ITU-T recommendations H.261 and H.263. Figure 3 shows the sampling grid used in the MPEG-1 scheme, and Figure 4 shows the sampling grid used in the MPEG-2 scheme.



**Figure 3. YUV 4:2:0 sample positions (MPEG-1 scheme)**



**Figure 4. YUV 4:2:0 sample positions (MPEG-2 scheme)**

Compared with the MPEG-1 scheme, it is simpler to convert between the MPEG-2 scheme and the sampling grids defined for 4:2:2 and 4:4:4 formats. For this reason, the MPEG-2 scheme is preferred in Windows, and should be considered the default interpretation of 4:2:0 formats.

#### Surface Definitions

This section describes the 8-bit YUV formats that are recommended for video rendering. These fall into several categories:

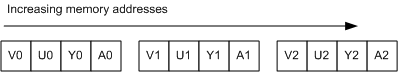
* [4:4:4 Formats, 32 Bits per Pixel](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_444formats_32bitsperpixel)
* [4:2:2 Formats, 16 Bits per Pixel](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_422formats_16bitsperpixel)
* [4:2:0 Formats, 16 Bits per Pixel](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_420formats_16bitsperpixel)
* [4:2:0 Formats, 12 Bits per Pixel](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_420formats_12bitsperpixel)

First, you should be aware of the following concepts in order to understand what follows:

* **Surface origin**. For the YUV formats described in this article, the origin (0,0) is always the upper-left corner of the surface.
* **Stride**. The *stride* of a surface, sometimes called the pitch, is the width of the surface in bytes. Given a surface origin at the upper-left corner, the stride is always positive.
* **Alignment**. The alignment of a surface is at the discretion of the graphics display driver. The surface must always be DWORD aligned, that is, individual lines within the surface are guaranteed to originate on a 32-bit (DWORD) boundary. The alignment can be larger than 32 bits, however, depending on the needs of the hardware.
* **Packed format versus planar format**. YUV formats are divided into *packed* formats and *planar* formats. In a packed format, the Y, U, and V components are stored in a single array. Pixels are organized into groups of macropixels, whose layout depends on the format. In a planar format, the Y, U, and V components are stored as three separate planes.

##### 4:4:4 Formats, 32 Bits per Pixel

A single 4:4:4 format is recommended, with the FOURCC code AYUV. This is a packed format, where each pixel is encoded as four consecutive bytes, arranged in the following sequence.



**Figure 5. AYUV memory layout**

The bytes marked A contain values for alpha.

##### 4:2:2 Formats, 16 Bits per Pixel

Two 4:2:2 formats are supported, with the following FOURCC codes:

* YUY2
* UYVY

Both are packed formats, where each macropixel is two pixels encoded as four consecutive bytes. This results in horizontal downsampling of the chroma by a factor of two.

##### YUY2

In YUY2 format, the data can be treated as an array of unsigned **char** values, where the first byte contains the first Y sample, the second byte contains the first U (Cb) sample, the third byte contains the second Y sample, and the fourth byte contains the first V (Cr) sample, as shown in Figure 6.

Figure 6. YUY2 memory layout image 

**Figure 6. YUY2 memory layout**

If the image is addressed as an array of two little-endian **WORD** values, the first **WORD** contains Y0 in the least significant bits (LSBs) and U in the most significant bits (MSBs). The second **WORD** contains Y1 in the LSBs and V in the MSBs.

YUY2 is the preferred 4:2:2 pixel format for Microsoft DirectX® Video Acceleration (DirectX VA). It is expected to be an intermediate-term requirement for DirectX VA accelerators supporting 4:2:2 video.

##### UYVY

This format is the same as YUY2, except the byte order is reversed — that is, the chroma and luma bytes are flipped (Figure 7). If the image is addressed as an array of two little-endian **WORD** values, the first **WORD** contains U in the LSBs and Y0 in the MSBs, and the second **WORD** contains V in the LSBs and Y1 in the MSBs.

Figure 7. UYVY memory layout

**Figure 7. UYVY memory layout**

##### 4:2:0 Formats, 16 Bits per Pixel

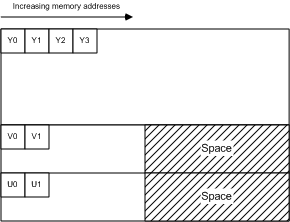
Two 4:2:0 16-bits per pixel formats are recommended, with the following FOURCC codes:

* IMC1
* IMC3

Both FOURCC codes are planar formats. The chroma channels are subsampled by a factor of two in both the horizontal and vertical dimensions.

##### IMC1

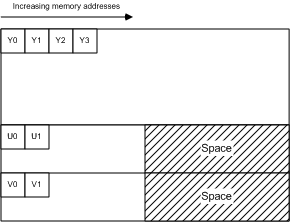
All of the Y samples appear first in memory as an array of unsigned **char** values. This is followed by all of the V (Cr) samples, and then all of the U (Cb) samples. The V and U planes have the same stride as the Y plane, resulting in unused areas of memory, as shown in Figure 8.



**Figure 8. IMC1 memory layout**

##### IMC3

This format is identical to IMC1, except the U and V planes are swapped:



**Figure 9. IMC3 memory layout**

##### 4:2:0 Formats, 12 Bits per Pixel

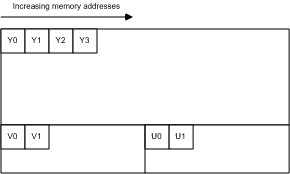
Four 4:2:0 12-bpp formats are recommended, with the following FOURCC codes:

* IMC2
* IMC4
* YV12
* NV12

In all of these formats, the chroma channels are subsampled by a factor of two in both the horizontal and vertical dimensions.

##### IMC2

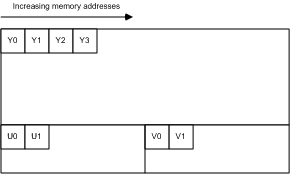
This format is the same as IMC1 except that the V (Cr) and U (Cb) lines are interleaved at half-stride boundaries. In other words, each full-stride line in the chroma area starts with a line of V samples, followed by a line of U samples that begins at the next half-stride boundary (Figure 10). This layout makes more efficient use of address space than IMC1. It cuts the chroma address space in half, and thus the total address space by 25 percent. Among 4:2:0 formats, IMC2 is the second-most preferred format, after NV12.



**Figure 10. IMC2 memory layout**

##### IMC4

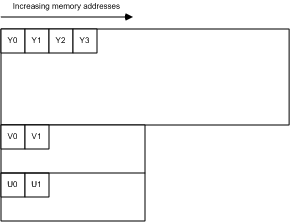
This format is identical to IMC2, except the U (Cb) and V (Cr) lines are swapped:



**Figure 11. IMC4 memory layout**

##### YV12

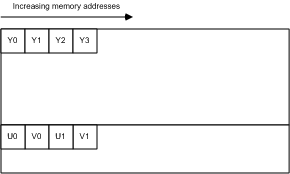
All of the Y samples appear first in memory as an array of unsigned **char** values. This array is followed immediately by all of the V (Cr) samples. The stride of the V plane is half the stride of the Y plane, and the V plane contains half as many lines as the Y plane. The V plane is followed immediately by all of the U (Cb) samples, with the same stride and number of lines as the V plane (Figure 12).



**Figure 12. YV12 memory layout**

##### NV12

All of the Y samples are found first in memory as an array of unsigned **char** values with an even number of lines. The Y plane is followed immediately by an array of unsigned **char** values that contains packed U (Cb) and V (Cr) samples, as shown in Figure 13. When the combined U-V array is addressed as an array of little-endian **WORD** values, the LSBs contain the U values, and the MSBs contain the V values. NV12 is the preferred 4:2:0 pixel format for DirectX VA. It is expected to be an intermediate-term requirement for DirectX VA accelerators supporting 4:2:0 video.



**Figure 13. NV12 memory layout**

#### Color Space and Chroma Sampling Rate Conversions

This section provides guidelines for converting between YUV and RGB, and for converting between some different YUV formats. We consider two RGB encoding schemes in this section: *8-bit computer RGB*, also known as sRGB or "full-scale" RGB, and *studio video RGB*, or "RGB with head-room and toe-room." These are defined as follows:

* Computer RGB uses 8 bits for each sample of red, green, and blue. Black is represented by R = G = B = 0, and white is represented by R = G = B = 255.
* Studio video RGB uses some number of bits N for each sample of red, green, and blue, where N is 8 or more. Studio video RGB uses a different scaling factor than computer RGB, and it has an offset. Black is represented by R = G = B = 16\*2N-8, and white is represented by R = G = B = 235\*2N-8. However, actual values may fall outside this range.

Studio video RGB is the preferred RGB definition for video in Windows, while computer RGB is the preferred RGB definition for non-video applications. In either form of RGB, the chromaticity coordinates are as specified in ITU-R BT.709 for the definition of the RGB color primaries. The (x,y) coordinates of R, G, and B are (0.64, 0.33), (0.30, 0.60), and (0.15, 0.06), respectively. Reference white is D65 with coordinates (0.3127, 0.3290). Nominal gamma is 1/0.45 (approximately 2.2), with precise gamma defined in detail in ITU-R BT.709.

**Conversion between RGB and 4:4:4 YUV**

We first describe conversion between RGB and 4:4:4 YUV. To convert 4:2:0 or 4:2:2 YUV to RGB, we recommend converting the YUV data to 4:4:4 YUV, and then converting from 4:4:4 YUV to RGB. The AYUV format, which is a 4:4:4 format, uses 8 bits each for the Y, U, and V samples. YUV can also be defined using more than 8 bits per sample for some applications.

Two dominant YUV conversions from RGB have been defined for digital video. Both are based on the specification known as ITU-R Recommendation BT.709. The first conversion is the older YUV form defined for 50-Hz use in BT.709. It is the same as the relation specified in ITU-R Recommendation BT.601, also known by its older name, CCIR 601. It should be considered the preferred YUV format for standard-definition TV resolution (720 x 576) and lower-resolution video. It is characterized by the values of two constants *Kr* and *Kb*:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_fc7163c9-d2a9-4dc4-a521-3a9b84064788');)

Kr = 0.299

Kb = 0.114

The second conversion is the newer YUV form defined for 60-Hz use in BT.709, and should be considered the preferred format for video resolutions above SDTV. It is characterized by different values for these two constants:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_c599cc80-443b-4faf-ac98-09104dc00232');)

Kr = 0.2126

Kb = 0.0722

Conversion from RGB to YUV is defined by starting with the following:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_f1c6275d-d159-40b6-a547-a88719e0a5e6');)

L = Kr \* R + Kb \* B + (1 – Kr – Kb) \* G

The YUV values are then obtained as follows:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_f0c31448-d18b-45af-856d-b56688398662');)

Y = floor(2^(M-8) \* (219\*(L–Z)/S + 16) + 0.5)

U = clip3(0, 2^M-1, floor(2^(M-8) \* (112\*(B-L) / ((1-Kb)\*S) + 128) + 0.5))

V = clip3(0, 2^M-1, floor(2^(M-8) \* (112\*(R-L) / ((1-Kr)\*S) + 128) + 0.5))

where

* M is the number of bits per YUV sample (M >= 8).
* Z is the black-level variable. For computer RGB, Z equals 0. For studio video RGB, Z equals 16\*2N-8, where N is the number of bits per RGB sample (N >= 8).
* S is the scaling variable. For computer RGB, S equals 255. For studio video RGB, S equals 219\*2N-8.

The function floor(x) returns the largest integer greater than or equal to *x*. The function clip3(x, y, z) is defined as follows:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_3c53ae7a-a773-48a6-98a1-2535f89732ea');)

clip3(x, y, z) = ((z < x) ? x : ((z > y) ? y : z))

The Y sample represents brightness, and the U and V samples represent the color deviations toward blue and red, respectively. The nominal range for Y is 16\*2M-8 to 235\*2M-8. Black is represented as 16\*2M-8, and white is represented as 235\*2M-8. The nominal range for U and V are 16\*2M-8 to 240\*2M-8, with the value 128\*2M-8 representing neutral chroma. However, actual values may fall outside these ranges.

For input data in the form of studio video RGB, the clip operation is necessary to keep the U and V values within the range 0 to 2M-1. If the input is computer RGB, the clip operation is not required, because the conversion formula cannot produce values outside of this range.

These are the exact formulas without approximation. Everything that follows in this document is derived from these formulas.

* [Example: Converting RGB888 to YUV 4:4:4](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_1)
* [Example: Converting 8-bit YUV to RGB888](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_2)
* [Converting 4:2:0 YUV to 4:2:2 YUV](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_3)
* [Converting 4:2:2 YUV to 4:4:4 YUV](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_4)
* [Converting 4:2:0 YUV to 4:4:4 YUV](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx#yuvformats_5)

##### Example: Converting RGB888 to YUV 4:4:4

In the case of computer RGB input and 8-bit BT.601 YUV output, we believe that the formulas given in the previous section can be reasonably approximated by the following:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_ceaa22f1-5860-42d6-a15e-8558e2864389');)

Y = ( ( 66 \* R + 129 \* G + 25 \* B + 128) >> 8) + 16

U = ( ( -38 \* R - 74 \* G + 112 \* B + 128) >> 8) + 128

V = ( ( 112 \* R - 94 \* G - 18 \* B + 128) >> 8) + 128

These formulas produce 8-bit results using coefficients that require no more than 8 bits of (unsigned) precision. Intermediate results will require up to 16 bits of precision.

##### Example: Converting 8-bit YUV to RGB888

From the original RGB-to-YUV formulas, one can derive the following relationships for the 8-bit BT.601 definition of YUV:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_3c080ac3-dcaa-4321-9e41-3af3927d308d');)

Y = round( 0.256788 \* R + 0.504129 \* G + 0.097906 \* B) + 16

U = round(-0.148223 \* R - 0.290993 \* G + 0.439216 \* B) + 128

V = round( 0.439216 \* R - 0.367788 \* G - 0.071427 \* B) + 128

Therefore, given:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_0251bc0e-e915-4aa1-b066-2ed7e02ef394');)

C = Y - 16

D = U - 128

E = V - 128

the formulas to convert YUV to computer RGB can be derived as follows:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_8c96d712-5f5e-4958-af58-1e2d0d7f915d');)

R = clip( round( 1.164383 \* C + 1.596027 \* E ) )

G = clip( round( 1.164383 \* C - (0.391762 \* D) - (0.812968 \* E) ) )

B = clip( round( 1.164383 \* C + 2.017232 \* D ) )

where clip() denotes clipping to a range of [0..255]. These formulas can be reasonably approximated by the following:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_bddad81a-1d83-436d-a8f8-88d9cfabfa64');)

R = clip(( 298 \* C + 409 \* E + 128) >> 8)

G = clip(( 298 \* C - 100 \* D - 208 \* E + 128) >> 8)

B = clip(( 298 \* C + 516 \* D + 128) >> 8)

These formulas use some coefficients that require more than 8 bits of precision to produce each 8-bit result, and intermediate results will require more than 16 bits of precision.

##### Converting 4:2:0 YUV to 4:2:2 YUV

Converting 4:2:0 YUV to 4:2:2 YUV requires vertical upconversion by a factor of two. This section describes an example method for performing the upconversion. The method assumes that the video pictures are progressive scan.

**Note**The 4:2:0 to 4:2:2 interlaced scan conversion process presents atypical problems and is difficult to implement. This article does not address the issue of converting interlaced scan from 4:2:0 to 4:2:2.

Let each vertical line of input chroma samples be an array Cin[] that ranges from 0 to N - 1. The corresponding vertical line on the output image will be an array Cout[] that ranges from 0 to 2N - 1. To convert each vertical line, perform the following process:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_bd79f622-f74f-4988-89dc-fde81ac888ce');)

Cout[0] = Cin[0];

Cout[1] = clip((9 \* (Cin[0] + Cin[1]) – (Cin[0] + Cin[2]) + 8) >> 4);

Cout[2] = Cin[1];

Cout[3] = clip((9 \* (Cin[1] + Cin[2]) - (Cin[0] + Cin[3]) + 8) >> 4);

Cout[4] = Cin[2]

Cout[5] = clip((9 \* (Cin[2] + Cin[3]) - (Cin[1] + Cin[4]) + 8) >> 4);

...

Cout[2\*i] = Cin[i]

Cout[2\*i+1] = clip((9 \* (Cin[i] + Cin[i+1]) - (Cin[i-1] + Cin[i+2]) + 8) >> 4);

...

Cout[2\*N-3] = clip((9 \* (Cin[N-2] + Cin[N-1]) - (Cin[N-3] + Cin[N-1]) + 8) >> 4);

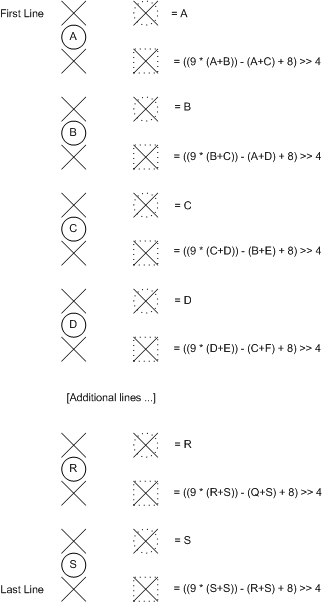
Cout[2\*N-2] = Cin[N-1];

Cout[2\*N-1] = clip((9 \* (Cin[N-1] + Cin[N-1]) - (Cin[N-2] + Cin[N-1]) + 8) >> 4);

where clip() denotes clipping to a range of [0..255].

**Note**The equations for handling the edges can be mathematically simplified. They are shown in this form to illustrate the clamping effect at the edges of the picture.

In effect, this method calculates each missing value by interpolating the curve over the four adjacent pixels, weighted toward the values of the two nearest pixels (Figure 14). The specific interpolation method used in this example generates missing samples at half-integer positions using a well-known method called Catmull-Rom interpolation, also known as cubic convolution interpolation.



**Figure 14. 4:2:0 to 4:2:2 upsampling**

In signal processing terms, the vertical upconversion should ideally include a phase shift compensation to account for the half-pixel vertical offset (relative to the output 4:2:2 sampling grid) between the locations of the 4:2:0 sample lines and the location of every other 4:2:2 sample line. However, introducing this offset would increase the amount of processing required to generate the samples, and make it impossible to reconstruct the original 4:2:0 samples from the upsampled 4:2:2 image. It would also make it impossible to decode video directly into 4:2:2 surfaces and then use those surfaces as reference pictures for decoding subsequent pictures in the stream. Therefore, the method provided here does not take into account the precise vertical alignment of the samples. Doing so is probably not visually harmful at reasonably high picture resolutions.

If you start with 4:2:0 video that uses the sampling grid defined in H.261, H.263, or MPEG-1 video, the phase of the output 4:2:2 chroma samples will also be shifted by a half-pixel *horizontal* offset relative to the spacing on the luma sampling grid (a quarter-pixel offset relative to the spacing of the 4:2:2 chroma sampling grid). However, the MPEG-2 form of 4:2:0 video is probably more commonly used on PCs and does not suffer from this problem. Moreover, the distinction is probably not visually harmful at reasonably high picture resolutions. Trying to correct for this problem would create the same sort of problems discussed for the vertical phase offset.

##### Converting 4:2:2 YUV to 4:4:4 YUV

Converting 4:2:2 YUV to 4:4:4 YUV requires horizontal upconversion by a factor of two. The method described previously for vertical upconversion can also be applied to horizontal upconversion. For MPEG-2 and ITU-R BT.601 video, this method will produce samples with the correct phase alignment.

##### Converting 4:2:0 YUV to 4:4:4 YUV

To convert 4:2:0 YUV to 4:4:4 YUV, you can simply follow the two methods described previously. Convert the 4:2:0 image to 4:2:2, and then convert the 4:2:2 image to 4:4:4. You can also switch the order of the two upconversion processes, as the order of operation does not really matter to the visual quality of the result.

#### Additional Information

To learn more about Microsoft DirectShow, see the [DirectShow SDK Documentation](http://go.microsoft.com/fwlink/?linkid=19689).

#include "convert.h"

// Conversion from RGB to YUV420

int RGB2YUV\_YR[256], RGB2YUV\_YG[256], RGB2YUV\_YB[256];

int RGB2YUV\_UR[256], RGB2YUV\_UG[256], RGB2YUV\_UBVR[256];

int RGB2YUV\_VG[256], RGB2YUV\_VB[256];

// Conversion from YUV420 to RGB24

static long int crv\_tab[256];

static long int cbu\_tab[256];

static long int cgu\_tab[256];

static long int cgv\_tab[256];

static long int tab\_76309[256];

static unsigned char clp[1024]; //for clip in CCIR601

//

// Table used for RGB to YUV420 conversion

//

void InitLookupTable()

{

int i;

for (i = 0; i < 256; i++) RGB2YUV\_YR[i] = (float)65.481 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_YG[i] = (float)128.553 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_YB[i] = (float)24.966 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_UR[i] = (float)37.797 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_UG[i] = (float)74.203 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_VG[i] = (float)93.786 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_VB[i] = (float)18.214 \* (i<<8);

for (i = 0; i < 256; i++) RGB2YUV\_UBVR[i] = (float)112 \* (i<<8);

}

//

// Convert from RGB24 to YUV420

//

int ConvertRGB2YUV(int w,int h,unsigned char \*bmp,unsigned int \*yuv)

{

unsigned int \*u,\*v,\*y,\*uu,\*vv;

unsigned int \*pu1,\*pu2,\*pu3,\*pu4;

unsigned int \*pv1,\*pv2,\*pv3,\*pv4;

unsigned char \*r,\*g,\*b;

int i,j;

uu=new unsigned int[w\*h];

vv=new unsigned int[w\*h];

if(uu==NULL || vv==NULL)

return 0;

y=yuv;

u=uu;

v=vv;

// Get r,g,b pointers from bmp image data....

r=bmp;

g=bmp+1;

b=bmp+2;

//Get YUV values for rgb values...

for(i=0;i<h;i++)

{

for(j=0;j<w;j++)

{

\*y++=( RGB2YUV\_YR[\*r] +RGB2YUV\_YG[\*g]+RGB2YUV\_YB[\*b]+1048576)>>16;

\*u++=(-RGB2YUV\_UR[\*r] -RGB2YUV\_UG[\*g]+RGB2YUV\_UBVR[\*b]+8388608)>>16;

\*v++=( RGB2YUV\_UBVR[\*r]-RGB2YUV\_VG[\*g]-RGB2YUV\_VB[\*b]+8388608)>>16;

r+=3;

g+=3;

b+=3;

}

}

// Now sample the U & V to obtain YUV 4:2:0 format

// Sampling mechanism...

/\* @ -> Y

# -> U or V

@ @ @ @

# #

@ @ @ @

@ @ @ @

# #

@ @ @ @

\*/

// Get the right pointers...

u=yuv+w\*h;

v=u+(w\*h)/4;

// For U

pu1=uu;

pu2=pu1+1;

pu3=pu1+w;

pu4=pu3+1;

// For V

pv1=vv;

pv2=pv1+1;

pv3=pv1+w;

pv4=pv3+1;

// Do sampling....

for(i=0;i<h;i+=2)

{

for(j=0;j<w;j+=2)

{

\*u++=(\*pu1+\*pu2+\*pu3+\*pu4)>>2;

\*v++=(\*pv1+\*pv2+\*pv3+\*pv4)>>2;

pu1+=2;

pu2+=2;

pu3+=2;

pu4+=2;

pv1+=2;

pv2+=2;

pv3+=2;

pv4+=2;

}

pu1+=w;

pu2+=w;

pu3+=w;

pu4+=w;

pv1+=w;

pv2+=w;

pv3+=w;

pv4+=w;

}

delete uu;

delete vv;

return 1;

}

//

//Initialize conversion table for YUV420 to RGB

//

void InitConvertTable()

{

long int crv,cbu,cgu,cgv;

int i,ind;

crv = 104597; cbu = 132201; /\* fra matrise i global.h \*/

cgu = 25675; cgv = 53279;

for (i = 0; i < 256; i++) {

crv\_tab[i] = (i-128) \* crv;

cbu\_tab[i] = (i-128) \* cbu;

cgu\_tab[i] = (i-128) \* cgu;

cgv\_tab[i] = (i-128) \* cgv;

tab\_76309[i] = 76309\*(i-16);

}

for (i=0; i<384; i++)

clp[i] =0;

ind=384;

for (i=0;i<256; i++)

clp[ind++]=i;

ind=640;

for (i=0;i<384;i++)

clp[ind++]=255;

}

//

// Convert from YUV420 to RGB24

//

void ConvertYUV2RGB(unsigned char \*src0,unsigned char \*src1,unsigned char \*src2,unsigned char \*dst\_ori,

int width,int height)

{

int y1,y2,u,v;

unsigned char \*py1,\*py2;

int i,j, c1, c2, c3, c4;

unsigned char \*d1, \*d2;

py1=src0;

py2=py1+width;

d1=dst\_ori;

d2=d1+3\*width;

for (j = 0; j < height; j += 2) {

for (i = 0; i < width; i += 2) {

u = \*src1++;

v = \*src2++;

c1 = crv\_tab[v];

c2 = cgu\_tab[u];

c3 = cgv\_tab[v];

c4 = cbu\_tab[u];

//up-left

y1 = tab\_76309[\*py1++];

\*d1++ = clp[384+((y1 + c1)>>16)];

\*d1++ = clp[384+((y1 - c2 - c3)>>16)];

\*d1++ = clp[384+((y1 + c4)>>16)];

//down-left

y2 = tab\_76309[\*py2++];

\*d2++ = clp[384+((y2 + c1)>>16)];

\*d2++ = clp[384+((y2 - c2 - c3)>>16)];

\*d2++ = clp[384+((y2 + c4)>>16)];

//up-right

y1 = tab\_76309[\*py1++];

\*d1++ = clp[384+((y1 + c1)>>16)];

\*d1++ = clp[384+((y1 - c2 - c3)>>16)];

\*d1++ = clp[384+((y1 + c4)>>16)];

//down-right

y2 = tab\_76309[\*py2++];

\*d2++ = clp[384+((y2 + c1)>>16)];

\*d2++ = clp[384+((y2 - c2 - c3)>>16)];

\*d2++ = clp[384+((y2 + c4)>>16)];

}

d1 += 3\*width;

d2 += 3\*width;

py1+= width;

py2+= width;

}

}

上面是CPP函数定义，以下是头文件  
分别实现YUV420TORGB24和RGB24TOYUV420的转换  
#include<stdio.h>  
// Conversion from RGB24 to YUV420  
void InitLookupTable();  
int ConvertRGB2YUV(int w,int h,unsigned char \*rgbdata,unsigned int \*yuv);  
// Conversion from YUV420 to RGB24  
void InitConvertTable();  
void ConvertYUV2RGB(unsigned char \*src0,unsigned char \*src1,unsigned char \*src2,unsigned char \*dst\_ori,  
int width,int height);

YUV是指亮度参量和色度参量分开表示的像素格式，而这样分开的好处就是不但可以避免相互干扰，还可以降低色度的采样率而不会对图像质量影响太大。YUV是一个比较笼统地说法，针对它的具体排列方式，可以分为很多种具体的格式。转载一篇对yuv格式解释的比较清楚地文章，也可以直接参考微软的那篇文章。

对于YUV格式，比较原始的讲解是MPEG-2 VIDEO部分的解释，当然后来微软有一个比较经典的解释，中文的大多是翻译这篇文章的。文章来源：[http://msdn.microsoft.com/en-us/library/aa904813(VS.80).aspx](http://msdn.microsoft.com/en-us/library/aa904813%28VS.80%29.aspx)

这里转载有人已经翻译过的，个人认为已经翻译的很不错了，遂放弃翻译。

[http://hondrif82q.spaces.live.com/blog/cns!776E82726DE60562!177.entry](http://hondrif82q.spaces.live.com/blog/cns%21776E82726DE60562%21177.entry)

[http://hondrif82q.spaces.live.com/blog/cns!776E82726DE60562!178.entry](http://hondrif82q.spaces.live.com/blog/cns%21776E82726DE60562%21178.entry)

#### YUV格式解析1（播放器——project2）

根据板卡api设计实现yuv420格式的视频播放器

打开\*.mp4;\*.264类型的文件，实现其播放。

使用的视频格式是YUV420格式

YUV格式通常有两大类：打包（packed）格式和平面（planar）格式。前者将YUV分量存放在同一个数组中，通常是几个相邻的像素组成一个宏像素（macro-pixel）；而后者使用三个数组分开存放YUV三个分量，就像是一个三维平面一样。表2.3中的YUY2到Y211都是打包格式，而IF09到YVU9都是平面格式。（注意：在介绍各种具体格式时，YUV各分量都会带有下标，如Y0、U0、V0表示第一个像素的YUV分量，Y1、U1、V1表示第二个像素的YUV分量，以此类推。）

MEDIASUBTYPE\_YUY2 **YUY2**格式，以4:2:2方式打包

MEDIASUBTYPE\_YUYV **YUYV**格式（实际格式与YUY2相同）

MEDIASUBTYPE\_YVYU **YVYU**格式，以4:2:2方式打包

MEDIASUBTYPE\_UYVY **UYVY**格式，以4:2:2方式打包

MEDIASUBTYPE\_AYUV 带Alpha通道的4:4:4 **YUV**格式

MEDIASUBTYPE\_Y41P **Y41P**格式，以4:1:1方式打包

MEDIASUBTYPE\_Y411 **Y411**格式（实际格式与Y41P相同）

 MEDIASUBTYPE\_Y211 **Y211**格式

MEDIASUBTYPE\_IF09 IF09格式

MEDIASUBTYPE\_IYUV IYUV格式

MEDIASUBTYPE\_YV12 YV12格式

 MEDIASUBTYPE\_YVU9 YVU9格式

               表2.3

## YUV 采样

YUV 的优点之一是，色度频道的采样率可比 Y 频道低，同时不会明显降低视觉质量。有一种表示法可用来描述 U 和 V 与 Y 的采样频率比例，这个表示法称为 A:B:C 表示法：

|  |  |
| --- | --- |
| • | 4:4:4 表示色度频道没有下采样。 |
| • | 4:2:2 表示 2:1 的水平下采样，没有垂直下采样。对于每两个 U 样例或 V 样例，每个扫描行都包含四个 Y 样例。 |
| • | 4:2:0 表示 2:1 的水平下采样，2:1 的垂直下采样。 |
| • | 4:1:1 表示 4:1 的水平下采样，没有垂直下采样。对于每个 U 样例或 V 样例，每个扫描行都包含四个 Y 样例。与其他格式相比，4:1:1 采样不太常用，本文不对其进行详细讨论。 |

图 1 显示了 4:4:4 图片中使用的采样网格。灯光样例用叉来表示，色度样例则用圈表示。

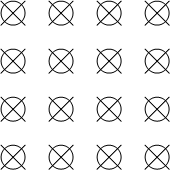


图 1. YUV 4:4:4 样例位置

4:2:2 采样的这种主要形式在 ITU-R Recommendation BT.601 中进行了定义。图 2 显示了此标准定义的采样网格。

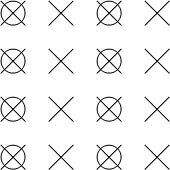


图 2. YUV 4:2:2 样例位置

4:2:0 采样有两种常见的变化形式。其中一种形式用于 MPEG-2 视频，另一种形式用于 MPEG-1 以及 ITU-T recommendations H.261 和 H.263。图 3 显示了 MPEG-1 方案中使用的采样网格，图 4 显示了 MPEG-2 方案中使用的采样网格。

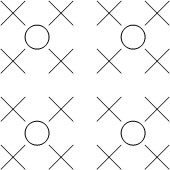


图 3. YUV 4:2:0 样例位置（MPEG-1 方案）

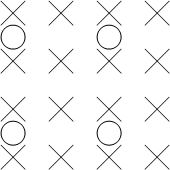


图 4. YUV 4:2:0 样例位置（MPEG-2 方案）

与 MPEG-1 方案相比，在 MPEG-2 方案与为 4:2:2 和 4:4:4 格式定义的采样网格之间进行转换更简单一些。因此，在 Windows 中首选 MPEG-2 方案，应该考虑将其作为 4:2:0 格式的默认转换方案。

## 表面定义

本节讲述推荐用于视频呈现的 8 位 YUV 格式。这些格式可以分为几个类别：

|  |  |
| --- | --- |
| • | 4:4:4 格式，每像素 32 位 |
| • | 4:2:2 格式，每像素 16 位 |
| • | 4:2:0 格式，每像素 16 位 |
| • | 4:2:0 格式，每像素 12 位 |

首先，您应该理解下列概念，这样才能理解接下来的内容：

|  |  |
| --- | --- |
| • | **表面原点**。对于本文讲述的 YUV 格式，原点 (0,0) 总是位于表面的左上角。 |
| • | **跨距**。表面的跨距，有时也称为间距，指的是表面的宽度，以字节数表示。对于一个表面原点位于左上角的表面来说，跨距总是正数。 |
| • | **对齐**。表面的对齐是根据图形显示驱动程序的不同而定的。表面始终应该 DWORD 对齐，就是说，表面中的各个行肯定都是从 32 位 (DWORD) 边界开始的。对齐可以大于 32 位，但具体取决于硬件的需求。 |
| • | **打包格式与平面格式**。YUV 格式可以分为打包 格式和平面 格式。在打包格式中，Y、U 和 V 组件存储在一个数组中。像素被组织到了一些巨像素组中，巨像素组的布局取决于格式。在平面格式中，Y、U 和 V 组件作为三个单独的平面进行存储。 |

**4:4:4 格式，每像素 32 位**

推荐一个 4:4:4 格式，FOURCC 码为 AYUV。这是一个打包格式，其中每个像素都被编码为四个连续字节，其组织顺序如下所示。

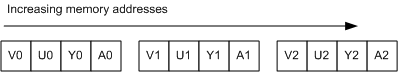


图 5. AYUV 内存布局

标记了 A 的字节包含 alpha 的值。

**4:2:2 格式，每像素 16 位**

支持两个 4:2:2 格式，FOURCC 码如下：

|  |  |
| --- | --- |
| • | YUY2 |
| • | UYVY |

两个都是打包格式，其中每个巨像素都是编码为四个连续字节的两个像素。这样会使得色度水平下采样乘以系数 2。

**YUY2**

在 YUY2 格式中，数据可被视为一个不带正负号的 **char** 值组成的数组，其中第一个字节包含第一个 Y 样例，第二个字节包含第一个 U (Cb) 样例，第三个字节包含第二个 Y 样例，第四个字节包含第一个 V (Cr) 样例，如图 6 所示。

http://www.microsoft.com/china/MSDN/library/enterprisedevelopment/softwaredev/art/yuvformats02.gif

图 6. YUY2 内存布局

如果该图像被看作由两个 little-endian **WORD** 值组成的数组，则第一个 **WORD** 在最低有效位 (LSB) 中包含 Y0，在最高有效位 (MSB) 中包含 U。第二个 **WORD** 在 LSB 中包含 Y1，在 MSB 中包含 V。

YUY2 是用于 Microsoft DirectX® Video Acceleration (DirectX VA) 的首选 4:2:2 像素格式。预期它会成为支持 4:2:2 视频的 DirectX VA 加速器的中期要求。

**UYVY**

此格式与 YUY2 相同，只是字节顺序是与之相反的 — 就是说，色度字节和灯光字节是翻转的（图 7）。如果该图像被看作由两个 little-endian **WORD** 值组成的数组，则第一个 **WORD** 在 LSB 中包含 U，在 MSB 中包含 Y0，第二个 **WORD** 在 LSB 中包含 V，在 MSB 中包含 Y1。

http://www.microsoft.com/china/MSDN/library/enterprisedevelopment/softwaredev/art/yuvformats03.gif

图 7. UYVY 内存布局

**4:2:0 格式，每像素 16 位**

推荐两个 4:2:0 每像素 16 位格式，FOURCC 码如下：

|  |  |
| --- | --- |
| • | IMC1 |
| • | IMC3 |

两个 FOURCC 码都是平面格式。色度频道在水平方向和垂直方向上都要以系数 2 来进行再次采样。

**IMC1**

所有 Y 样例都会作为不带正负号的 **char** 值组成的数组首先显示在内存中。后面跟着所有 V (Cr) 样例，然后是所有 U (Cb) 样例。V 和 U 平面与 Y 平面具有相同的跨距，从而生成如图 8 所示的内存的未使用区域。

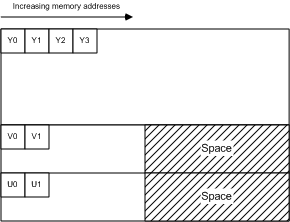


图 8. IMC1 内存布局

**IMC3**

此格式与 IMC1 相同，只是 U 和 V 平面进行了交换：

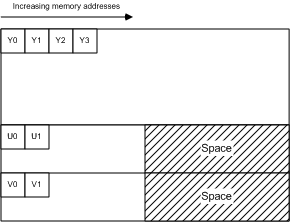


图 9. IMC3 内存布局

**4:2:0 格式，每像素 12 位**

推荐四个 4:2:0 每像素 12 位格式，FOURCC 码如下：

|  |  |
| --- | --- |
| • | IMC2 |
| • | IMC4 |
| • | YV12 |
| • | NV12 |

在所有这些格式中，色度频道在水平方向和垂直方向上都要以系数 2 来进行再次采样。

**IMC2**

此格式与 IMC1 相同，只是 V (Cr) 和 U (Cb) 行在半跨距边界处进行了交错。换句话说，就是色度区域中的每个完整跨距行都以一行 V 样例开始，然后是一行在下一个半跨距边界处开始的 U 样例（图 10）。此布局与 IMC1 相比，能够更加高效地利用地址空间。它的色度地址空间缩小了一半，因此整体地址空间缩小了 25%。在各个 4:2:0 格式中，IMC2 是第二首选格式，排在 NV12 之后。

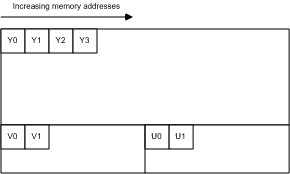


图 10. IMC2 内存布局

**IMC4**

此格式与 IMC2 相同，只是 U (Cb) 和 V (Cr) 行进行了交换：

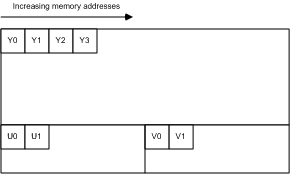


图 11. IMC4 内存布局

**YV12**

所有 Y 样例都会作为不带正负号的 **char** 值组成的数组首先显示在内存中。此数组后面紧接着所有 V (Cr) 样例。V 平面的跨距为 Y 平面跨距的一半，V 平面包含的行为 Y 平面包含行的一半。V 平面后面紧接着所有 U (Cb) 样例，它的跨距和行数与 V 平面相同（图 12）。

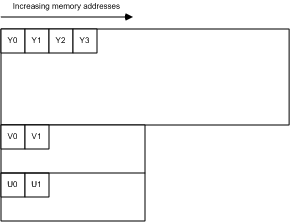
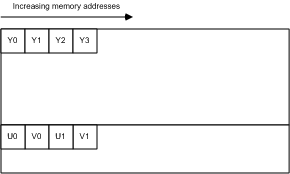


图 12. YV12 内存布局

**NV12**

所有 Y 样例都会作为由不带正负号的 **char** 值组成的数组首先显示在内存中，并且行数为偶数。Y 平面后面紧接着一个由不带正负号的 **char** 值组成的数组，其中包含了打包的 U (Cb) 和 V (Cr) 样例，如图 13 所示。当组合的 U-V 数组被视为一个由 little-endian **WORD** 值组成的数组时，LSB 包含 U 值，MSB 包含 V 值。NV12 是用于 DirectX VA 的首选 4:2:0 像素格式。预期它会成为支持 4:2:0 视频的 DirectX VA 加速器的中期要求。



#### YUV格式解析2

又确认了一下H264的视频格式——H264支持4：2：0的连续或隔行视频的编码和解码

YUV（亦称YCrCb）是被欧洲电视系统所采用的一种颜色编码方法（属于PAL）。YUV主要用于优化彩色视频信号的传输，使其向后兼容老式黑白电视。与RGB视频信号传输相比，它最大的优点在于只需占用极少的带宽（RGB要求三个独立的视频信号同时传输）。其中“Y”表示明亮度（Luminance或Luma），也就是灰阶值；而“U”和“V”表示的则是色度（Chrominance或Chroma），作用是描述影像色彩及饱和度，用于指定像素的颜色。“亮度”是通过RGB输入信号来创建的，方法是将RGB信号的特定部分叠加到一起。“色度”则定义了颜色的两个方面—色调与饱和度，分别用Cr和CB来表示。其中，Cr反映了GB输入信号红色部分与RGB信号亮度值之间的差异。而CB反映的是RGB输入信号蓝色部分与RGB信号亮度值之同的差异。

补充一下场的概念——

场的概念不是从DV才开始有的，电视系统已经有了（当然，DV和电视的关系大家都知道）归根结底还是扫描的问题，具体到PAL制式是：   
每秒25帧，每帧两场，扫描线（包括电视机的电子束）自上而下先扫描一场，然后再自上而下扫描第二场   
之所以引入场的概念，我的理解是主要为了在有限的带宽和成本内使画面运动更加平滑和消除闪烁感。   
这两个场的扫描线是一条一条互相间隔开的，比如说对于一个帧来讲，最上面一条线编号为0，紧挨着的是1，再下来是2，3，4，5，6。。。。那么第一场也许是0，2，4，6；也许是1，3，5，7——这就是隔行扫描   
在逐行扫描模式下，就是扫描线按照0，1，2，3，4，5的顺序依次扫描，很明显，这时候就不存在场的概念了。

下面区分一下YUV和YCbCr

YUV色彩模型来源于RGB模型，

该模型的特点是将亮度和色度分离开，从而适合于图像处理领域。

应用：模拟领域

Y'= 0.299\*R' + 0.587\*G' + 0.114\*B'

U'= -0.147\*R' - 0.289\*G' + 0.436\*B' = 0.492\*(B'- Y')

V'= 0.615\*R' - 0.515\*G' - 0.100\*B' = 0.877\*(R'- Y')

R' = Y' + 1.140\*V'

G' = Y' - 0.394\*U' - 0.581\*V'

B' = Y' + 2.032\*U'

YCbCr模型来源于YUV模型。YCbCr是 YUV 颜色空间的偏移版本.

应用：数字视频，ITU-R BT.601建议

Y’ = 0.257\*R' + 0.504\*G' + 0.098\*B' + 16

Cb' = -0.148\*R' - 0.291\*G' + 0.439\*B' + 128

Cr' = 0.439\*R' - 0.368\*G' - 0.071\*B' + 128

R' = 1.164\*(Y’-16) + 1.596\*(Cr'-128)

G' = 1.164\*(Y’-16) - 0.813\*(Cr'-128) - 0.392\*(Cb'-128)

B' = 1.164\*(Y’-16) + 2.017\*(Cb'-128)

PS: 上面各个符号都带了一撇，表示该符号在原值基础上进行了伽马校正,伽马校正有助于弥补在抗锯齿的过程中，线性分配伽马值所带来的细节损失，使图像细节更加丰富。在没有采用伽马校正的情况下，暗部细节不容易显现出来，而采用了这一图像增强技术以后，图像的层次更加明晰了。

所以说H264里面的YUV应属于YCbCr.

下面再仔细谈谈YUV格式, YUV格式通常有两大类：打包（packed）格式和平面（planar）格式。前者将YUV分量存放在同一个数组中，通常是几个相邻的像素组成一个宏像素（macro-pixel）；而后者使用三个数组分开存放YUV三个分量，就像是一个三维平面一样。

我们常说得YUV420属于planar格式的YUV, 颜色比例如下:

Y0U0V0             Y1                 Y2U2V2                      Y3

Y4                 Y5                 Y6                          Y7

Y8U8V8             Y9                 Y10U10V10                   Y11

Y12                Y13                Y14                         Y15

其他格式YUV可以点这里查看详细内容, 而在YUV文件中YUV420又是怎么存储的呢? 在常见H264测试的YUV序列中,例如CIF图像大小的YUV序列(352\*288),在文件开始并没有文件头,直接就是YUV数据,先存第一帧的Y信息,长度为352\*288个byte, 然后是第一帧U信息长度是352\*288/4个byte, 最后是第一帧的V信息,长度是352\*288/4个byte, 因此可以算出第一帧数据总长度是352\*288\*1.5,即152064个byte, 如果这个序列是300帧的话, 那么序列总长度即为152064\*300=44550KB,这也就是为什么常见的300帧CIF序列总是44M的原因.

4:4:4采样就是说三种元素Y,Cb,Cr有同样的分辨率,这样的话,在每一个像素点上都对这三种元素进行采样.数字4是指在水平方向上对于各种元素的采样率,比如说,每四个亮度采样点就有四个Cb的Cr采样值.4:4:4采样完整地保留了所有的信息值.4:2:2采样中(有时记为YUY2),色度元素在纵向与亮度值有同样的分辨率,而在横向则是亮度分辨率的一半(4:2:2表示每四个亮度值就有两个Cb和Cr采样.)4:2:2视频用来构造高品质的视频彩色信号.

在流行的4:2:0采样格式中(常记为**YV12**)Cb和Cr在水平和垂直方向上有Y分辨率的一半.4:2:0有些不同，因为它并不是指在实际采样中使用4:2:0，而是在编码史中定义这种编码方法是用来区别于4:4:4和4:2:2方法的).4:2:0采样被广泛地应用于消费应用中，比如视频会议，数字电视和DVD存储中。因为每个颜色差别元素中包含了四分之一的Y采样元素量，那么4:2:0YCbCr视频需要刚好4: 4:4或RGB视频中采样量的一半。

4:2:0采样有时被描述是一个"每像素12位"的方法。这么说的原因可以从对四个像素的采样中看出. 使用4:4:4采样，一共要进行12次采样，对每一个Y,Cb和Cr，就需要12\*8=96位，平均下来要96/4=24位。使用4:2:0就需要6\*8 =48位，平均每个像素48/4=12位。

在一个4:2:0隔行扫描的视频序列中，对应于一个完整的视频帧的Y,Cb,Cr采样分配到两个场中。可以得到，隔行扫描的总采样数跟渐进式扫描中使用的采样数目是相同的。

对比一下：

Y41P（和Y411）（packed格式）格式为每个像素保留Y分量，而UV分量在水平方向上每4个像素采样一次。一个宏像素为12个字节，实际表示8个像素。图像数据中YUV分量排列顺序如下： U0 Y0 V0 Y1 U4 Y2 V4 Y3 Y4 Y5 Y6 Y8 …

IYUV格式（planar）为每个像素都提取Y分量，而在UV分量的提取时，首先将图像分成若干个2 x 2的宏块，然后每个宏块提取一个U分量和一个V分量。YV12格式与IYUV类似，但仍然是平面模式。

YUV411、YUV420格式多见于DV数据中，前者用于NTSC制，后者用于PAL制。YUV411为每个像素都提取Y分量，而UV分量在水平方向上每4个像素采样一次。YUV420并非V分量采样为0，而是跟YUV411相比，在水平方向上提高一倍色差采样频率，在垂直方向上以U/V间隔的方式减小一半色差采样，如下图所示。

（好像显示不出来突下图像）

各种格式的具体使用位数的需求（使用4:2:0采样，对于每个元素用8个位大小表示)：

格式： Sub-QCIF 亮度分辨率： 128\*96  每帧使用的位: 147456  
格式： QCIF  亮度分辨率： 176\*144  每帧使用的位: 304128  
格式： CIF  亮度分辨率： 352\*288  每帧使用的位: 1216512  
格式：  4CIF  亮度分辨率： 704\*576  每帧使用的位: 4866048