RGB/YUV Pixel Conversion

I recently received an email from Mike Perry thoroughly explaining this whole issue. For the definitive answer, please look here. If you want my original posting, here it is...

Conversion back and forward between RGB and YUV formats is a topic that often causes confusion - are we dealing with gamma corrected RGB values? Which luminance/chrominance colour space are we actually dealing with ? Why can't someone just tell me how to do the conversion without giving me 5 chapters of theory first!

To cut a long story short, here are the formulae that I have used to do the conversions for PC video applications (well, OK, these are not exactly the formulae I used but I did generate a bunch of lookup tables from these). The colour space in question is actually YC_bC_r and not YUV, which a video purist will tell you is, in fact, the colour scheme employed in PAL TV systems and is somewhat different (NTSC TVs use YIQ which is different again). Why the PC video fraternity adopted the term YUV is a mystery but I strongly suspect that it has something to do with not having to type subscripts.

The following 2 sets of formulae are taken from information from Keith Jack's excellent book "Video Demystified" (ISBN 1-878707-09-4).

RGB to YUV Conversion

```
Y  =      (0.257 * R) + (0.504 * G) + (0.098 * B) + 16

Cr = V =  (0.439 * R) - (0.368 * G) - (0.071 * B) + 128

Cb = U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128
```

YUV to RGB Conversion

```
B = 1.164(Y - 16)                   + 2.018(U - 128)

G = 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128)

R = 1.164(Y - 16) + 1.596(V - 128)
```

In both these cases, you have to clamp the output values to keep them in the [0-255] range. Rumour has it that the valid range is actually a subset of [0-255] (I've seen an RGB range of [16-235] mentioned) but clamping the values into [0-255] seems to produce acceptable results to me.

Further Information

Julien (surname unknown) suggests that there are problems with the above formulae and suggests the following instead:

```
Y = 0.299R + 0.587G + 0.114B

U'= (B-Y)*0.565
```

```
    V'= (R-Y)*0.713
```

with reciprocal versions:

```
    R = Y + 1.403V'

    G = Y - 0.344U' - 0.714V'

    B = Y + 1.770U'
```

Scott Scriven has recently sent me a C program he wrote to convert Quantel YUV files as generated by SGI's "ZapIt!" video capture software into RGB PPM files. You can get hold of the source by clicking here. Please contact Scott with any support questions on this program.

Intel actually has an extensive web site on the topic of colour space conversion and you can find it here.

Mike Perry's Definitive Answer to the Conversion Question

"I looked at your RGB/YUV page again and it still contains extremely "iffy" wording as if no-one knows what the heck is really the correct way to do conversion. There are two specifications, CCIR 656 and CCIR 601 which define standards for component video, and I'm sure other pages on your site refer to them. In any case, CCIR 601 defines the relationship between YCrCb and RGB values:

$Ey = 0.299R+0.587G+0.114B$
$Ecr = 0.713(R - Ey) = 0.500R-0.419G-0.081B$
$Ecb = 0.564(B - Ey) = -0.169R-0.331G+0.500B$

where Ey, R, G and B are in the range [0,1] and Ecr and Ecb are in the range [-0.5,0.5] (Equations corrected per input from Stephan Bourgeois and Gregory Smith below)

If you form a matrix with those coefficients and compute the inverse you get coefficients equivalent to what Julien suggested.

The reason your first set of coefficients is incorrect, and related to your clamping comment, is because the defined range for Y is [16,235] (220 steps) and the valid ranges for Cr and Cb are [16,239] (235 steps) These are normalized ranges and when the data is used, 16 is added to Y and 128 is added from Cr and Cb to de-normalize them. The reason for this is that control and colorburst information is stored in the components along with the luminence and chromiance data which is why the full range of [0,255] is not used. There is no "rumor" about this being correct. It is a defined standard for YCrCb video. I wouldn't be surprised if certain PC programs are using a variant where the entire range of [0,255] is used but technically it is not YCrCb. If you use YCrCb for video purposes, its probably going to be CCIR 601 compliant. I don't know why you would use YCrCb for anything else as many rendering operations are extremely expensive to do in the non-linear YCrCb color space relative to the linear RGB color space.

Most importantly, you should probably mention that there are more efficient ways of converting from YCrCb to RGB than just doing a straightforward implementation of the floating point matrix multiplication.

First, the input values YCrCb component values you are given may not even be valid within the YCrCb color space. This can occur if the components were obtained by converting from another color space (usually RGB). This places additional overhead to implement a pre-clamp phase and to also to possibly normalize the component values if they arent normalized already (subtract 16 from Y, subtract 128 from Cr and Cb, divide by 219 or 235 to obtain normalized ranges). You can reduce this overhead to a minimum by creating two 256-entry lookups which contain pre-converted-and-clamped results for input values in the range 0-255. You need one table for luminance and one table for chromiance.

Second, floating point operations are usually much slower than integer operations, at least when it comes to most C compilers which do very poor instruction scheduling. Speed might not be important if you are converting static data "offline," but floating point is a killer if you're trying to do software conversion of YCrCb frame buffers. I do my conversion using 2.14 fixed point representation stored in 32 bit integers. You can use any number of fraction bits that you want but my target system (the Nuon) has pixel pack and unpack operations which treat the values as 2.14. You certainly don't need any more than 2 integer bits as you only care about the range [-2.0,2.0]. You can precalculate the normalize-and-clamp lookup tables to contain fixed point values so that the first stage is consists solely of three lookups.

At that point you do the equivalent matrix multiply steps except that you use fixed point instead of floating point. This is gobs faster than using floating point operations. After the conversion you have obtained R, G, and B. You then need to do another series of clamps to ensure everything is in the range [0,1]. If you get your coefficients just right, you probably shouldn't need to do this, but its rather difficult to ensure that underflow and overflow wont occur. If a component is not clamped, you need to obtain the scaled value using a fixed point multiply, usually by 255 if you're trying to convert to 8-bit component values.

Also, its important to note that if you are only interested in luminance data (for the purposes of creating a greyscale image) then the RGB triple for YCrCb is simply (Ey,Ey,Ey) and you don't have to do the matrix calculation nor post-clamping and you only have to scale one value. Simply run Y through the luminance lookup table, scale to [0,255] and plug into R, G and B.

I am looking into the possibility of using overlay channels to bypass software conversion altogether, although my emulated system (Nuon) has two channels (main + overlay) with alpha blending capabilities so I would only be able to use this mechanism when only one screen is active and I would lose chrominance information unless hardware vendors start implementing unpacked YCrCb instead of YUY2. Another interesting method of doing YUV to RGB conversion for the purpose of displaying video is to use programmable pixel shaders. You can almost pull it off on a Geforce3 but the GeforceFX and Radeon 9700/9800 cards are advanced enough that you can pass 32 bit YCrCb textures to the video card and do all conversion calculations on the video card. Using multi-texturing you can even do main channel and overlay channel alpha blending in one pass. Given that converting a single 720x480 32-bit YCrCb frame buffer at 60 frames per second takes up about 25% of my xp2800+ CPU time, this would be a very cool thing indeed."

Further Input from Stephan Bourgeois:

In Mike Perry's Section, the conversion matrix should be :

Ey = 0.299R + 0.587G + 0.114B
Ecr = 0.713(R - Ey) = 0.500R - 0.419G - 0.081B
Ecb = 0.564(B - Er) = -0.169R - 0.331G + 0.500B *(Gregory Smith points out that Er here should read Ey - equations above were corrected)*

there is a minus sign missing in the R->Cb coefficient, and a digit 3 missing in G->Cb.

Also I disagree with Mike on the idea that Y is "normalized" by adding 16. If RGB=[255,255,255], the equations would give Y=255+16. Y should be clamped at 16 to prevent sub-black, the same way as it should be clamped at 235 to prevent white>100%. This can be done by the use of a 256 entry lookup table.

Avery Lee's JFIF Clarification

"The equations you have from "Video Demystified" are the ones to use most of the time on PCs, such as for MPEG-1 decoding and handing most (all?) of the formats on the YUV FOURCC page. However, there is one notable exception: JPEG. The JFIF file format that is commonly used with JPEG defines a YCbCr color space that is slightly different than usual, in that all components have the full [0,255] excursion rather than [16,235] and [16,240]. The equations given by "Julien" are for this variant and are very close to the ones in JFIF:

R = Y + 1.402 (Cr-128)

G = Y - 0.34414 (Cb-128) - 0.71414 (Cr-128)

B = Y + 1.772 (Cb-128)

It should be noted that some Motion JPEG codecs don't take this consideration into account and improperly exchange YCbCr values between the YUY2 and UYVY formats and their internal JPEG planes without performing the necessary scaling. This results in contrast shifts when using YUV formats."

Microsoft's Answer

MSDN also has an answer to this question. You can find it here.

Want some sample code?

Peter Bengtsson recently sent code for a sample program showing how to perform YCrCb to RGB conversion with a Fragment Shader (Pixel Shader) in OpenGL. He says that it is tested on Linux but should be usable on Windows with only minor rework. Please contact him directly (tesla at och dot nu) with questions or comments.

The Thorny Issue of HD Video

Just when you thought things were quietening down, Richard Salmon correctly points out that the YCrCb colour space used for HD video (ITU.BT-709) is actually different from SD (ITU.BT-601). He adds:

"Found your page, and I'm afraid, even with Mike Perry's explanation, which covers the problems of coding ranges and black/white offsets in the digital coding of TV signals etc very well, it is still lacking;

In addition to the YUV/RGB equations relating to Rec.601 (which are used for Standard Definition TV) there are another completely different set adopted for HDTV (ITU Rec.709). This change of equation was entirely pointless, but unfortunately we have to live with it, since it is the internationally agreed standard. You will find the information on this at:

http://www.dcs.ed.ac.uk/home/mxr/gfx/faqs/colorconv.faq"

Page was last modified on: October 09 2007 22:19:11.