NTSC video

From Nesdev wiki

Unlike many other game consoles, the NES does not generate RGB or YUV and then encode that to composite. Instead, it generates **NTSC video** directly in the composite domain, which leads to interesting artifacts.

NTSC Master clock is 21.47727273 MHz and each PPU pixel lasts four clocks; PAL master clock is 26.6017125 MHz, and each PPU pixel lasts five clocks. $xy refers to a palette color in the range $00 to $3F.

Contents

Scanline Timing

Values in PPU pixels (341 total per scanline).

Rendering scanlines (n=240):

name	start	duration	row	notes
short sync	280	25	0-239	
black (back porch)	305	4	0-239	
colorburst	309	15	0-239	
black (the rest of back porch)	324	5	0-239	
pulse (background color in grayscale)	329	1	0-239	
left border (background color)	330	15	0-239	
active	4	256	0-239	if background rendering is disabled, border will be rendered instead
right border (background color)	260	11	0-239	
black (front porch)	271	9	0-239	

Post-render scanlines (n=2):

name	start	duration	row
short sync	280	25	240-241
black (back porch)	305	4	240-241
colorburst	309	15	240-241
black (the rest of back porch)	324	5	240-241
pulse (background color in grayscale)	329	1	240-241
bottom border (background color)	330	282	240-241
black (front porch)	271	9	240-241

Post-render blanking scanlines (n=3):

name	start	duration	row
short sync	280	25	242-244
black (back porch)	305	4	242-244
colorburst	309	15	242-244
black	324	297	242-244

Vertical sync scanlines (n=3):

name	start	duration	row
long sync	280	318	245-247
black (sync separator)	257	23	245-247

Pre-render blanking scanlines (n=14):

name	start	duration	row	notes
short sync	280	25	248-261	
black (back porch)	305	4	248-261	
colorburst	309	15	248-261	14 columns on end of row 261 for odd frames, if either background or sprite rendering is enabled
black	324	297	248-261	

For a total of 262 scanlines.

This video timing is non-standard. In standard NTSC, a scanline is 227.5 subcarrier cycles long (equivalent to 341.25 NES pixels), and each field is 262 or 263 lines tall with the vertical sync pulse in one frame half a scanline late to make the TV draw the other field of the interlaced frame half a scanline up. The NES and most other pre-Dreamcast consoles draw the fields on top of each other, resulting in a non-standard low-definition "progressive" or "double struck" video mode sometimes called 240p (http://junkerhq.net/xrgb/index.php/240p_video). Some high-definition displays and upscalers cannot handle 240p video, instead introducing artifacts that make the video appear as if it were interlaced. Artemio Urbana's 240p test suite (http://junkerhq.net/xrgb/index.php/240p_test_suite), which has been ported to NES (http://forums.nesdev.com/viewtopic.php?p=157634#p157634) by Damian Yerrick, contains a set of test patterns to diagnose problems with decoding 240p composite video.

Note that emulators usually crop the top and bottom 8 lines from the picture, as most televisions will hide at least part of the picture in a similar way. See: Overscan

Brightness Levels

$xE/$xF output the same voltage as $1D. $x1-$xC output a square wave alternating between levels for $xD and $x0. Colors $20 and $30 are exactly the same.

When grayscale is active, all colors between $x1-$xD are treated as $x0. Notably this behavior extends to the first pixel of the border color, which acts as a sync pulse on every visible scanline.

Initial measurement

Voltage levels used by the PPU are as follows - absolute, relative to synch, and normalized between black level and white:

Type	Absolute	Relative	Normalized
Synch	0.781	0.000	-0.359
Colorburst L	1.000	0.218	-0.208
Colorburst H	1.712	0.931	0.286
Color 0D	1.131	0.350	-0.117
Color 1D (black)	1.300	0.518	0.000
Color 2D	1.743	0.962	0.308
Color 3D	2.331	1.550	0.715
Color 00	1.875	1.090	0.397
Color 10	2.287	1.500	0.681
Color 20	2.743	1.960	1.000
Color 30	2.743	1.960	1.000

These levels don't quite match the standard levels. Ideally, the composite signal is 1000 microvolts from peak to peak (1.0 Vp-p) when loaded with a 75 Ω output impedance. (Unloaded levels may be twice that amount, which may explain the levels seen above.) Levels are commonly measured in units called IRE.[1][2]

Terminated measurement

Standard video (not NES) looks like this:

Type	IRE level	Voltage (mV)
Peak white	120	
White	100	714
Colorburst H	20	143
Black	7.5	53.6
Blanking	0	0
Colorburst L	-20	-143
Sync	-40	-286

The following measurements by lidnariq (http://forums.nesdev.com/viewtopic.php?p=159266#p159266) into a properly terminated (75 Ω) TV have about 10 mV of noise and 4 mV of quantization error, which implies an error of ±2 IRE:

Signal	Potential	IRE
SYNC	48 mV	-37 IRE
CBL	148 mV	-23 IRE
0D	228 mV	-12 IRE
1D	312 mV	≡ 0 IRE
CBH	524 mV	30 IRE
2D	552 mV	34 IRE
00	616 mV	43 IRE
10	840 mV	74 IRE
3D	880 mV	80 IRE
20	1100 mV	110 IRE
0Dem	192 mV	-17 IRE
1Dem	256 mV	-8 IRE
2Dem	448 mV	19 IRE
00em	500 mV	26 IRE
10em	676 mV	51 IRE
3Dem	712 mV	56 IRE
20em	896 mV	82 IRE

Unlike PAL, US NTSC is supposed to have a "setup", a difference between blanking and black level. Japanese NTSC does not make this distinction.

Color Phases

The color generator is clocked by the rising *and* falling edges of the ~21.48 MHz clock, resulting in an effective ~42.95 MHz clock rate. There are 12 color square waves, spaced at regular phases. Each runs at the ~3.58 MHz colorburst rate. On NTSC, color $xY uses the wave shown in row Y from the table. NTSC color burst (pure shade -U) uses color phase 8 (with voltages listed above); PAL color burst is believed to alternate between 6 (-U+V) and 9 (-U-V), so hue is rotated by 15° from NTSC. PAL alternates the broadcast sign of the V component, so on PAL every odd scanline will use the appropriate opposite phase—e.g. phases 5-C are respectively replaced with C-5.

Color Tint Bits

There are three color modulation channels controlled by the top three bits of $2001. Each channel uses one of the color square waves (see above diagram) and enables attenuation of the video signal when the color square wave is high. A single attenuator is shared by all channels. It is active for 6 out of 12 half-clocks if one bit is set, 10 half-clocks if two bits are set, or all 12 if all three bits are set.

$2001	Active phase	Complement
Bit 7	Color 8	Color 2 (blue)
Bit 6	Color 4	Color A (green)
Bit 5	Color C	Color 6 (red)

When signal attenuation is enabled by one or more of the channels and the current pixel is a color other than $xE/$xF (black), the signal is attenuated as follows (calculations given for both relative and absolute values as shown in the voltage table above):

```
relative = relative * 0.746

normalized = normalized * 0.746 - 0.0912
```

For example, when $2001 bit 6 is true, the attenuator will be active during the phases of color 4. This means the attenuator is not active during its complement (color A), and the screen appears to have a tint of color A, which is green.

Note that on the Dendy and PAL NES, the green and red bits swap meaning.

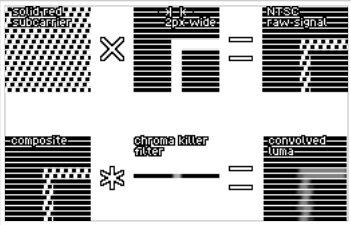
A test performed on NTSC NES (http://forums.nesdev.com/viewtopic.php?p=160669#p160669) shows that while emphasis affects color $1D, it does *not* affect colors $0E, $0F, $1E, $1F, $2E, $2F, $3E, and $3F.

Example Waveform

This waveform steps through various grays and then stops on a color.

```
1.0               +--+
0.9               |  |
0.8               |  |
0.7            +--+  | +-+ +-+
0.6            |     | | | | |
0.5            |     | | | | |
0.4         +--+     | | | | |
0.3      +--+        | | | | |
0.2      |           | | | | |
0.1      |           | | | | |
0.0 . +--+ . . . . . +-+ +-+ + . .
-0.1 --+
     0D 0F 2D 00 10 30   11
```

The PPU's shortcut method of NTSC modulation often produces artifacts in which vertical lines appear slightly ragged, as the chroma spills over into luma.



Generation and demodulation of a red corner

Emulating in C++ code

For efficient, ready to use implementations, see Libraries below. The following is an illustrative example:

Calculating the momentary NTSC signal level can be done as follows in C++:

```
// pixel = Pixel color (9-bit) given as input. Bitmask format: "eeellcccc".
// phase = Signal phase (0..11). It is a variable that increases by 8 each pixel.
float NTSCsignal(int pixel, int phase)
{
    // Voltage levels, relative to synch voltage
    static const float black=.518f, white=1.962f, attenuation=.746f,
        levels[8] = {.350f, .518f, .962f,1.550f,  // Signal low
                     1.094f,1.506f,1.962f,1.962f}; // Signal high

    // Decode the NES color.
    int color = (pixel & 0x0F);    // 0..15 "cccc"
    int level = (pixel >> 4) & 3;  // 0..3  "ll"
    int emphasis = (pixel >> 6);   // 0..7  "eee"
    if(color > 13) { level = 1;  } // For colors 14..15, level 1 is forced.

    // The square wave for this color alternates between these two voltages:
    float low  = levels[0 + level];
    float high = levels[4 + level];
    if(color == 0) { low = high; } // For color 0, only high level is emitted
    if(color > 12) { high = low; } // For colors 13..15, only low level is emitted

    // Generate the square wave
    auto InColorPhase = [=](int color) { return (color + phase) % 12 < 6; }; // Inline function
    float signal = InColorPhase(color) ? high : low;

    // When de-emphasis bits are set, some parts of the signal are attenuated:
    if( ((emphasis & 1) && InColorPhase(0))
    ||  ((emphasis & 2) && InColorPhase(4))
    ||  ((emphasis & 4) && InColorPhase(8)) ) signal = signal * attenuation;

    return signal;
}
```

The process of generating NTSC signal for a single pixel can be simulated with the following C++ code:

```
void RenderNTSCpixel(unsigned x, int pixel, int PPU_cycle_counter)
{
    int phase = PPU_cycle_counter * 8;
    for(int p=0; p<8; ++p) // Each pixel produces distinct 8 samples of NTSC signal.
    {
        float signal = NTSCsignal(pixel, phase + p); // Calculated as above
        // Optionally apply some lowpass-filtering to the signal here.
        // Optionally normalize the signal to 0..1 range:
        static const float black=.518f, white=1.962f;
        signal = (signal-black) / (white-black);
        // Save the signal for this pixel.
        signal_levels[ x*8 + p ] = signal;
    }
}
```

It is important to note that while the NES only generates eight (8) samples of NTSC signal per pixel, the wavelength for chroma is 12 samples long. This means that the colors of adjacent pixels get mandatorily mixed up to some degree. For the same reason, narrow black&white details can be interpreted as colors.

Because the scanline length is uneven (341*8 is not an even multiple of 12), the color mixing shifts a little each scanline. This appears visually as a sawtooth effect at the edges of colors at high resolution. The sawtooth cycles every 3 scanlines.

Because also the frame length is uneven (neither 262*341*8 nor (262*341-1)*8 is an even multiple of 12), the color mixing also changes a little every frame. When rendering is normally enabled, the screen is alternatingly 89342 and 89341 cycles long. The combination of these (89342+89341)*8 *is* an even multiple of 12, which means that the artifact pattern cycles every 2 frames. The pattern of cycling can be changed by disabling rendering during the end of the pre-render scanline; it forces the screen length to 89342 cycles, even if would be 89341 otherwise.

The process of decoding NTSC signal (convert it into RGB) is subject to a lot of study, and there are many patents and different techniques for it. A simple method suitable for emulation is covered below. It is not accurate, because in reality the chroma is blurred much more than is done here (the region of signal sampled for I and Q is wider than 12 samples), and the filter used here is a simple box FIR filter rather than an IIR filter, but it already produces a quite authentic looking picture. In addition, the border region (total of 26 pixels of background color around the 256-pixel scanline) is not sampled.

```
    float signal_levels[256*8] = {...}; // Eight signal levels for each pixel, normalized to 0..1 range.
Calculated as above.

    unsigned Width; // Input: Screen width. Can be not only 256, but anything up to 2048.
    float phase;    // Input: This should the value that was PPU_cycle_counter * 8 + 3.9
                    // at the BEGINNING of this scanline. It should be modulo 12.
                    // It can additionally include a floating-point hue offset.
    for(unsigned x = 0; x < Width; ++x)
    {
        // Determine the region of scanline signal to sample. Take 12 samples.
        int center = x * (256*8) / Width + 0;
        int begin = center - 6; if(begin < 0)     begin = 0;
        int end   = center + 6; if(end   > 256*8) end   = 256*8;
        float y = 0.f, i = 0.f, q = 0.f; // Calculate the color in YIQ.
        for(int p = begin; p < end; ++p) // Collect and accumulate samples
        {
            float level = signal_levels[p] / 12.f;
            y  =  y + level;
            i  =  i + level * cos( M_PI * (phase+p) / 6 );
            q  =  q + level * sin( M_PI * (phase+p) / 6 );
        }
        render_pixel(y,i,q); // Send the YIQ color for rendering.
    }
```

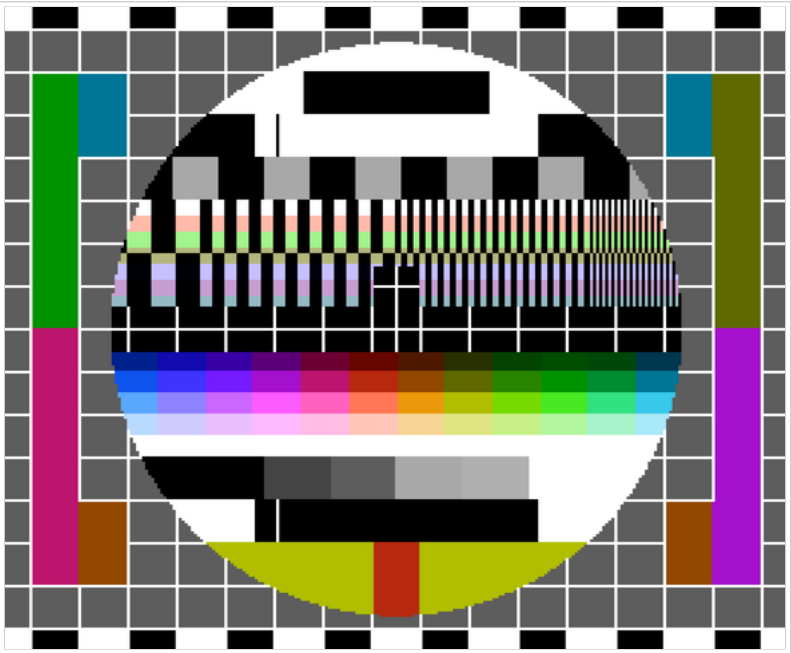
The NTSC decoder here produces pixels in YIQ color space.

If you want more saturated colors, just multiply `i` and `q` with a factor of your choosing, such as 1.7. If you want brighter colors, just multiply `y`, `i` and `q` with a factor of your choosing, such as 1.1. If you want to adjust the hue, just add or subtract a value from/to `phase`. If you want to see so called chroma dots, change the begin and end in such manner that you collect a number of samples that is not divisible with 12. If you want to blur the video horizontally, change the begin and end in such manner that the samples are collected from a wider region.

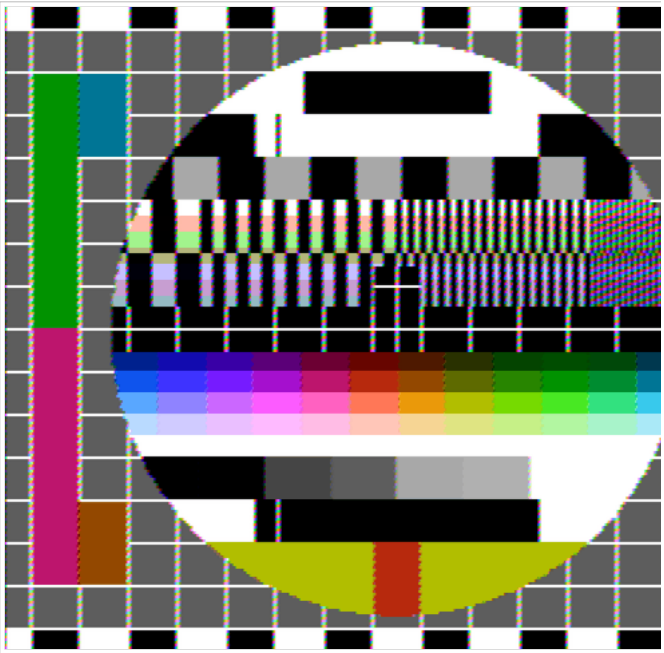
The YIQ colors can be converted into sRGB colors with the following formula, using the FCC-sanctioned YIQ-to-RGB conversion matrix. This produces a value that can be saved to e.g. framebuffer:

```
    float gamma = 2.0f; // Assumed display gamma
    auto gammafix = [=](float f) { return f <= 0.f ? 0.f : pow(f, 2.2f / gamma); };
    auto clamp    = [](int v) { return v>255 ? 255 : v; };
    unsigned rgb =
        0x10000*clamp(255.95 * gammafix(y +  0.946882f*i +  0.623557f*q))
      + 0x00100*clamp(255.95 * gammafix(y + -0.274788f*i + -0.635691f*q))
      + 0x00001*clamp(255.95 * gammafix(y + -1.108545f*i +  1.709007f*q));
```

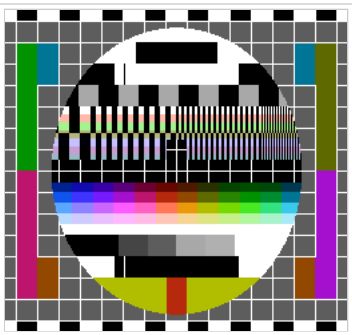
The two images below illustrate the NTSC artifacts. In the left side image, 12 samples of NTSC signal were generated for each NES pixel, and each display pixel was separately rendered by decoding that 12-sample signal. In the right side image, 8 samples of NTSC signal were generated for each NES pixel, and each display pixel was rendered by decoding 12 samples of NTSC signal from the corresponding location within the scanline.



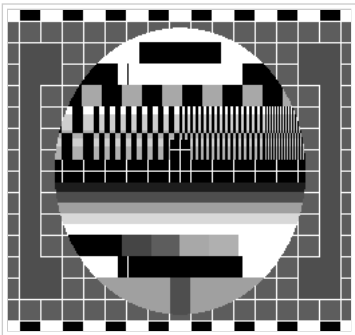
Per-pixel rendering: 12 samples of NTSC signal per input pixel; the same 12 samples are decoded for each output pixel



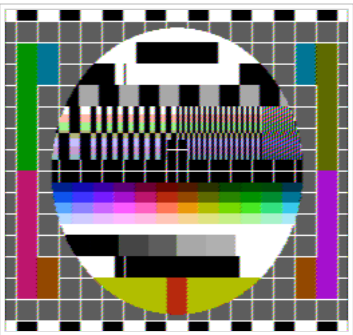
Per-scanline rendering: 8 samples of NTSC signal per input pixel; 12 samples are deco output pixel



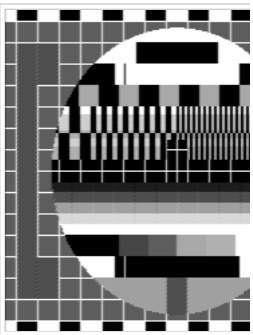
Same as above, but rendered at 256x240 without upscaling



Same in grayscale (zero saturation). This illustrates well how the different color values have exactly the same luminosity; only the chroma phase differs.



Same as above, but rendered at 256x240 rather than at 2048x240 and then downscaled



Same in grayscale

The source code of the program that generated both images can be read at [1] (http://bisqwit.iki.fi/jutut/kuvat/programming_examples/nesemu1/ntsc_test.phps). Note that even though the image resembles the well-known Philips PM5544 test card, it is not the same; the exact same colors could not be reproduced with NES colors. In addition, some parts were changed to better test NES features. For example, the backgrounds for the "station ID" regions (the black rectangles at the top and at the bottom inside the circle) are generated using the various blacks within the NES palette.

Later, Bisqwit made a generic integer-based decoder in C++ (http://forums.nesdev.com/viewtopic.php?p=172329#p172329). This takes a signal at 12 times color burst and can be used to emulate other systems that use shortcuts when generating NTSC video, such as Apple II (where *every* color in HGR is an artifact color) and Atari 7800 (whose game *Tower Toppler* seriously exploits artifact colors).

Libraries

- blargg's nes_ntsc library (http://slack.net/~ant/libs/ntsc.html)
- blargg's NTSC demo windows executable (http://forums.nesdev.com/viewtopic.php?f=21&t=11947)
- Forum thread (http://forums.nesdev.com/viewtopic.php?f=2&t=14338): New NTSC decoder with integer-only math (short C++ code) - by Bisqwit

See also

- PAL video

References

1. ↑ Tutorial 1184: Understanding Analog Video Signals (https://www.maximintegrated.com/en/app-notes/index.mvp/id/1184)
2. ↑ Analog Video 101 (http://www.ni.com/white-paper/4750/en/)