Modern Graphics on the TI-99/4A
Image Conversion to the TMS9918A Mode 2

by Mike Brent aka Tursi

For quite a long time I've been working on a tool to convert modern images to a format the TI can render, specifically our Mode 2, also called Bitmap Mode. This article discusses some of the advances I made with my tool and how it works.

To understand this, I will delve briefly into how graphics work on the TMS9918, then how Mode 2 works. I will then cover some of the history of my tool and show where it has made it to so far.

VDP Theory

Except for text mode, which I will not cover here, the TMS9918 and TMS9918A Video Display Processor (VDP) always runs a graphics mode with 256 horizontal pixels, and 192 vertical pixels, plus some overscan area which is not user definable (except to set a global color). It is capable of generating 15 colors, which are fixed in the hardware to a predefined set (a 16th color, “transparent”, is used for video overlay which is not implemented in the 99/4A). It can also display 32 hardware sprites, which are independently positionable, single color objects. I do not cover sprites in this document except by occasional reference. My techniques do not use sprites today.

The standard graphics mode on the TI-99/4A is Mode 1, also called Graphics Mode. Mode 1 is a character based mode, meaning that it can display 32 characters across, and 24 characters down, for a total of 768 characters on the screen. These characters are each 8 pixels wide by 8 pixels tall, and any pattern fitting in this size may be defined. Finally, 256 characters may be defined at any one time. Normally, only 96 of these are defined by the system – these being the normal characters printable in the ASCII character set – for the most part the characters you can type on the keyboard. Although BASIC and Extended BASIC limit the characters you can define (in order to save video RAM for your program), the video chip can manage 256 unique patterns on this screen.

You can use this from TI BASIC. The CALL CHAR subprogram lets you change the definition of any of the characters that are legal for TI BASIC to use. CALL HCHAR and CALL VCHAR let you display them on the screen in any position – for that matter, so does PRINT, though PRINT is a little more limited in TI BASIC.

Mode 1 handles color by grouping these characters into sets of 8 – for every 8 characters, you can assign two colors. These are the foreground and background colors for that *color set*, and you can see this using the CALL COLOR subprogram.

This mode allows pretty decent graphics to be displayed, and most TI games use it. There are some limits, however. Because you have 768 screen positions and only 256 characters, some characters must by necessity by duplicated (normally this would be the blank spaces!) In addition, color is limited in two ways – first that each 8x8 pixel block may only have two colors through its entire area, and second that the color is associated with each character. (For instance, if you want a red 'A' and a green 'A', you must have two 'A's defined in your

character set, located in different color groups.)

This was the best available on the TMS9918, with the other color mode, Multicolor, limited to 4x4 blocks, giving an effective resolution of 64x48. However, with the TMS9918A, a new graphics mode, Mode 2, was added.

Mode 2, also called Bitmap mode, added the ability for the VDP to reference up to three character sets on a single display screen. It also greatly improved the color resolution over Mode 1, allowing every line of every character to have its own two colors assigned, and allowing every independent character it's own color mapping. Together, these two options remove the limitations of Mode 1, allowing the entire screen to contain unique bit patterns. (Using fewer than three character sets to take advantage of the color enhancements without defining extra characters is what is often referred to as “half-bitmap” mode.)

The one remaining limitation is that every row of every character, 8 pixels, may only have 2 colors assigned. Given the memory bandwidth, the designer felt this was the best tradeoff. While it is markedly better than the limitations of Mode 1, it does still introduce a complexity when dealing with the graphics mode.

Image Conversion

The point of converting an image is to reproduce, as closely as possible, an original image in a different format. Most people are probably familiar with resizing an image to produce a smaller version, in order to email or post on the web – this is one example.

Modern, “true color” images, typically represent every pixel as three bytes, one each for the relative amount of red, green, and blue in that pixel. This results in more than 16 million possible colors and brightnesses for each pixel. In addition, they typically present their images in increasingly large resolutions – 1600x1200 pixels come out of the average “2 megapixel” camera.

Our job here, is to take that image, reduce the 1600x1200 pixels down to 256x192, and reduce the 16 million colors down to our fixed set of 15, and still produce a recognizable image.

One problem that remains outstanding is that, in order to map an image to those fixed 15 colors, we need to define exactly what they are. But the TMS9918 does not output digital color, rather it outputs analog video which is, worse, NTSC encoded (on the US version). Colors in NTSC is a bit outside the scope of this document, but essentially they are based on when the signal occurs relative to a continuously varying, smoothly changing signal, and that the actual color displayed is a function of the internal setting of the VDP, the components used in the amplifier, the accuracy of the color timing signal, and the particular setting of the monitor that is viewing it. In my work so far, the colors have been based on the values in the datasheet, however, this doesn't mean they are perfect representations of what makes it out the rear of the console. They are a “best guess” for now, and it's one area that still needs some research.

Resizing

The first step, and the simplest, is to resize the image. There are several ways to reduce the resolution of the image. The simplest is usually called “Nearest-Neighbor”, and amounts to skipping every 'x' pixels. In our example here, we want to convert 1600 pixels across to 256 pixels. So we divide (1600/256), and then we take every 6th pixel (after rounding the result). This will result in an image that more or less resembles the original, but loses a lot of the detail. It will often look very stepped.

effect of nearest neighbor sampling on an image (1/6th)

The other way to resize, is to not take every sixth pixel, but to *average* those six pixels together to decide what to draw. In truth there are many algorithms that attempt to do this in more meaningful ways than a simple average, but for the purposes of this article, it's enough to understand that in the end, they take those pixels that would otherwise be discarded, and combine them to produce the result. In the end, the detail is preserved much better, and edges look smoother. At worst, it looks a little fuzzy, but usually much better than just skipping through the image.

effect of resizing with a sampling/averaging filter (1/6th)

My image converter provides a number of options for the resize – each has plusses and minuses. Some are sharper but trade off detail, some are fuzzier, some work more closely with the color. Generally the default will work, but the best filter may very per image.

Color Reduction

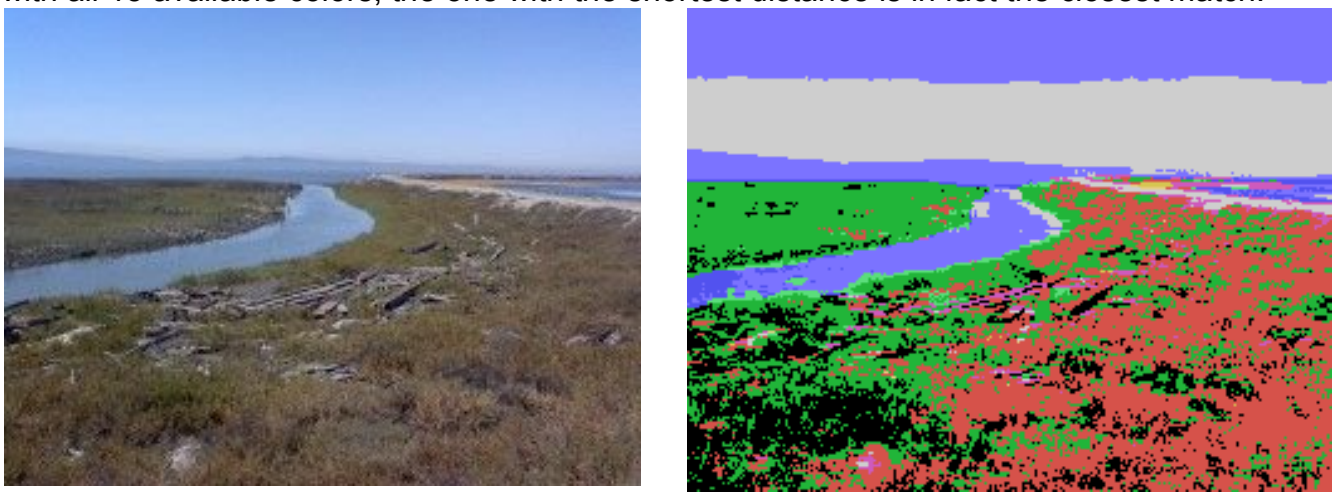
Once you have the image to the right size, you need to reduce the color count. We need to go from 16 million possible colors, down to only 15.

On systems where the palette can be chosen, this is often done by picking the 15 most popular colors, or colors that can be mixed to produce them, and then mapping the image. In our case, however, we don't get a choice – our colors were chosen for us by TI.

To remap a color, you want the closest representation of that original color that you can get.

In order for the computer to determine this, is it common to consider color in terms of an imaginary three-dimensional world, known as “color space”.

Simply put, color space maps the Red, Green, and Blue components of each pixel to the X, Y, and Z axes of an imaginary grid. In order to determine how similar a color is, then, you can simply reduce it to calculating the distance between two 3D points. If you compare a color with all 15 available colors, the one with the shortest distance is in fact the closest match.



Effect of a straight color reduction to the 9918A palette

Unfortunately, the palette that TI chose for us is better suited to rendering simple cartoon characters than complex scenes – it has few shades and many bright colors that rarely map directly to anything in real life.

Fortunately, color reduction is not a new problem, and a solution has been around for a very long time. It is called dithering (or error diffusion), and in fact it has been around as long as photos have needed to be printed (half-toning is a form of dithering that reduces an image to only two colors – black and white!) There are many approaches to dithering, because it is an incredibly subjective concept, but the basic idea is that by placing two colors very close together, your eye will mix them to a certain degree, allowing you to represent a third color. By varying the patterns, you can vary the amount of the mix and still preserve detail in the original image.

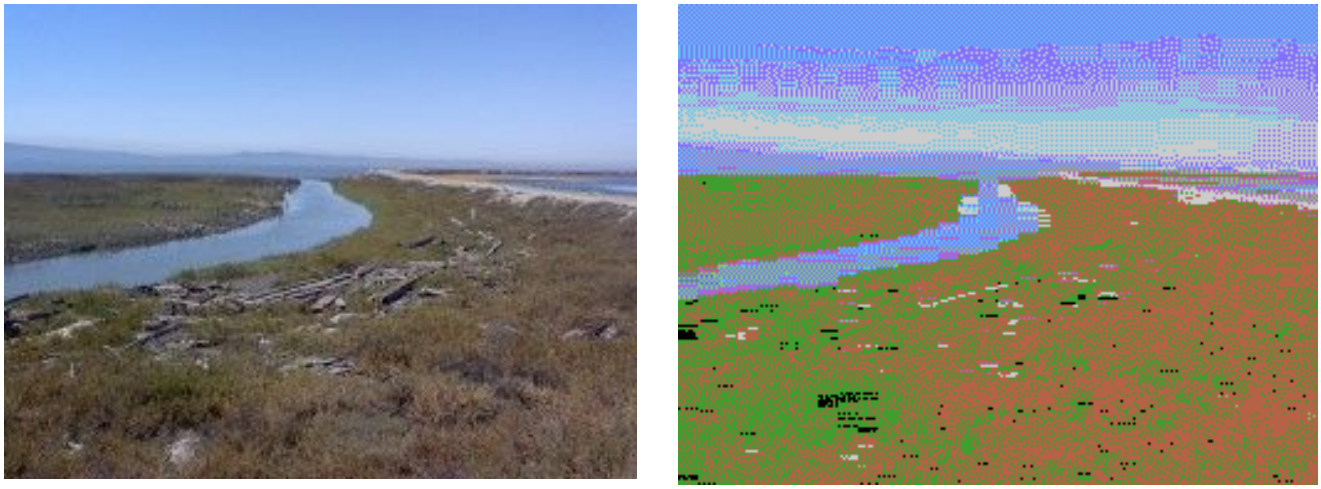
The dither algorithm that is probably the most popular in modern use is Floyd-Steinberg, first published in 1976. The idea behind Floyd-Steinberg is that after you chose your best match color for a pixel, you calculate the error – that is, how far your chosen color is from the actual color desired. You then divide this error up in a particular way to the neighboring pixels, such that they will attempt to compensate for the error. By controlling how much of the error is delivered to each pixel, you set up patterns that are (hopefully!) pleasing to the eye while still compensating for the color error.

However, we have a problem with implementing Floyd-Steinberg on the TI – it's a bit of a chicken-and-the-egg problem. Because of our color limitations, we need to calculate 8 horizontal pixels at a time. But Floyd-Steinberg requires you to distribute each pixel's error to the next pixel. We can't do this if we don't know what the error is yet! Worse, that entire group of 8 pixels is only allowed two colors. It seems like you have to pre-calculate the result before

you can calculate what the result will be!

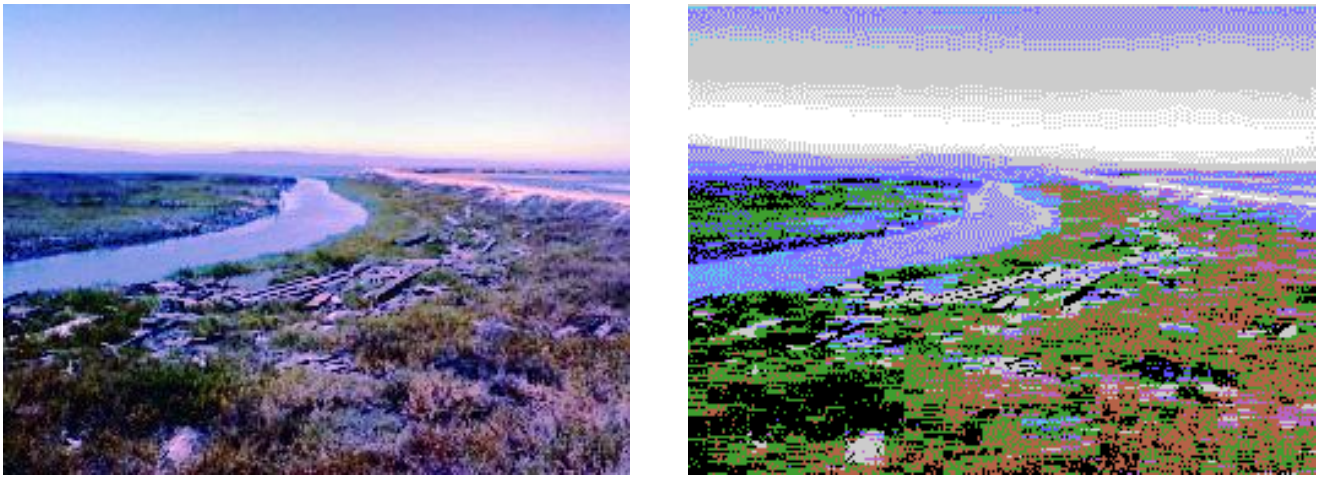
Brute Force to the Rescue

In my first attempt to solve this issue, I allowed the system to perform a normal dither of the original image. This gave me an image that converted quickly to the TI palette, but did not respect the limitation of Mode 2. I would then go through each block of 8 pixels, and if there were more than 2 colors there, I would count their usage, and remove the least used color until I finally had only two left, which I would then (again) remap the 8 pixels to. It was imperfect, but it was fast and it seemed to more or less work, given the chicken-and-the-egg problem I had with dithering.



Dither-then-remap approach

Unfortunately, there was a massive loss of contrast with this simple approach. To work around that, I added a “stretch histogram” option. What this did was to look at the colors used in the image, and effectively tweak the contrast to get the maximum range between the lightest and darkest colors in the image. This helped a lot, although it resulted in a color shift. At the time, however, I felt the results were as good as they were going to get.



Stretching the histogram

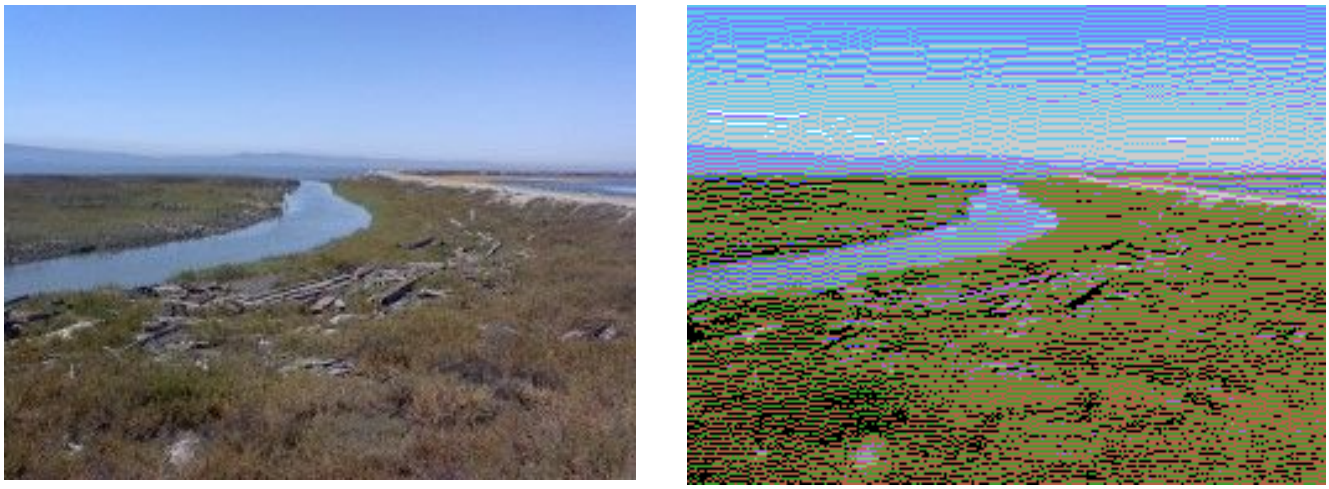
This was good, and for a while, I put it away, until I stumbled upon a converter for the MSX

computer, a machine that shares the same graphics chip. I was surprised by how much nicer their images were than mine, and so I went back to work.

I thought about the scope of the problem – the problem really comes down to deciding what two bytes go into each 8-pixel position – 8 bits of pattern, and 2 colors. So that's 65536 possibilities for every position. That's a big number, but.. it's not so bad to a modern computer. I further did the math for the number of positions – (256/8)*192 = 6144 positions. That means the maximum search to try every possible combination for every position on the screen is just over 400 million searches – a big number, but one that modern PCs can deal with.

And so I reworked the system to try every possibility – it basically set up every combination of pixels and colors, and compared them against the original image. If the new setup was closer than the old one, we remember it, otherwise, we discard it. Either way, go on to the next one. When we are done, the combination we remembered was the closest match to the original pattern. I coded this, and in fact it worked. It was slow – my machine took 40 seconds to convert one image – but it did produce better results than the original approach.

Unfortunately, I still didn't have a dither. Since I couldn't get the dither to work horizontally, I wondered what would happen if I just did the downward part of the dither, since that would be easy to implement. In fact, it helped some. Between the new search and the downward dither, images began to look a lot more like they should, and there was no longer any need for the histogram stretch.



Effect of brute force search and vertical-only dithering

Unfortunately, the vertical dithering led to a striping effect, and we had to choose what to do with the horizontal component of the error. If we threw it away, we had color loss and the image would still color shift. If we added it all downward, the color effect was better, but the image was a lot noisier and harder to look at.

Epiphanies, and Other Nighttime Disorders

This problem gnawed at me for some time, and I experimented with ways to induce flicker to increase the apparent number of colors, until suddenly I had a flash of inspiration and realized that I was coming at the problem backwards. Rather than picking a color, seeing how

well it matched the source, and then dithering the image, all I had to do was dither the image first!

This might seem confusing, especially with the brute force approach, but in fact it led itself very well to it. The brute force approach would build every possible combination of pixels for every comparison. All I had to do was assume the dither was taking effect while doing the comparison – that is, after measuring how far off the first pixel was, I added the difference to the ORIGINAL image's next pixel, representing what the dithered ideal would be IF that first pixel was accepted. The search itself was still blind, but the criteria was changed from matching the original image, to matching the ideal dithered image.

And this worked exceptionally well, producing wonderfully dithered images with good color matching and no more tricks needed at all.

Effect of dithering during the comparison stage

I was also able to improve speed substantially, by realizing that more than half the search space was duplicated. For example, if I test all combinations with a foreground color of red, and a background color of green, I don't need to test any combinations with a foreground color of green and a background color of red. Also, there was no point testing the 'transparent' color, it would never be needed. Also, if a block was pure black or pure white, I could quickly detect this and bypass the search completely. By doing this, I cut a huge amount of the search space out. By careful loop optimization, and adding multiple threads to handle the search, I got the processing time on my machine down to an average of 3 seconds.

For a long time, I considered this the best, but there was still one concept left in mind that I needed to look at – perceptual color matching. In theory, this takes advantage of the fact that our eyes have different levels of sensitivity to different colors. In particular, we see green the best, red the next best, and blue is actually a very distant third. Image formats like JPEG take advantage of this weakness – they actually reduce the color resolution to less than the actual image, relying on our eyes much better ability to see brightness and assume color.

Research into perceptual color reduction didn't help me much – they all focus on *choosing* the best colors, but we don't get that choice, so I couldn't see how to apply it, until I had a thought. What if I graded the value of each axis of the color match, such that green was most

important, red the next most so, and blue the least important? Would it help?

To choose the scaling, I used the standard values for converting a color image into greyscale. This standard maps green to 59%, red to 30% and blue to just 11% of the final value, so I scaled my RGB->XYZ axes on the color cube by multiplying by these values. This would lead to them being spaced out more for a very minimal extra cost in processing.

The answer is that for many images, the results ARE better, although I often find it difficult to see exactly why. The new images seem a little sharper, and the choices made by the program just seem *better*. This is likely due to the emphasis on green being right, which gives us most of our brightness sensing. The one downside is that dark areas tend to get a little blue creeping in. This is fair – the 9918 doesn't have a dark grey, and we've told the system that blue only has an 11% importance value, so when it wants a little extra shading, it sometimes finds dark blue is the closest match. Sometimes the effect is nonetheless pleasing, but that is why it can be disabled.

In the example below, compare especially fine details like the distant beach at top right to the sample image above without perceptual color matching, and see how the detail is much easier to make out in this version, also see how the sky dithering has less random noise.

Effect of perceptual color mapping

The End?

Image mapping is such a subjective thing that it's hard to say that it's ever “as good as it can be”, but I can certainly say that at this point, my knowledge of the concept is fully implemented in this program. One area that still needs improvement is the breaking up of regular patterns in dark or scenes with heavy gradient in them. Should I gain more knowledge in the future, it's likely I'll again attempt to apply it!

References (for more information!)

Convert9918 – my software, as described above
http://harmlesslion.com/software/convert

Dithering at Wikipedia

http://en.wikipedia.org/wiki/Dithering

Floyd Steinberg Dithering at Wikipedia
http://en.wikipedia.org/wiki/Floyd-Steinberg_dithering

Greyscale at Wikipedia
http://en.wikipedia.org/wiki/Greyscale

All about Mode 2 at Thierry Nouspikel's website
http://www.nouspikel.com/ti99

GameSX – Human Visual Acuity (aka Your Eyes Suck at Blue)
http://www.gamesx.com/misctech/visual.htm