A Ghost in a Snowstorm

James F. Blinn

Microsoft Research

There's a storm cloud growing on the horizon of the digital convergence between computer graphics and television/graphic arts. Computer graphics and image processing assume that pixel values are linearly related to light intensity. A typical video or paint image, however, encodes intensity nonlinearly. Most image manipulation software doesn't take this into account and just does arithmetic on the pixel values as though they were linearly related to light intensity. This is obviously wrong. The question is, how wrong, and for what pixel values is the problem worst and best?

Let's review the two basic concepts.

Compositing

A full-color pixel has four components: a transparency value, alpha, and three color primaries that are implicitly already multiplied by the alpha value:

$$\mathbf{F} = [F_{\text{red}} \;\; F_{\text{green}} \;\; F_{\text{blue}} \;\; F_{\text{alpha}}]$$

The most useful image compositing operation, called *over*, simulates one partially transparent pixel in front of another. The *over* operation of a foreground **F** in front of a background **B** is defined as

$$\mathbf{F} \; over \; \mathbf{B} \equiv \mathbf{F} + (1 - F_{\text{alpha}})\mathbf{B}$$

Since each pixel component is treated identically, I will simplify by only dealing with the calculation of one component. I'll call a generic component of **F**, f; a generic component of **B**, b; and the alpha component of **F**, a. Our basic calculation is then

$$(f, a) \; over \; b \equiv f + (1 - a)b$$

The parameters a and b can have values from 0 to 1. Since f is premultiplied by a, we have $f \le a$.

Some applications have occasion to composite images with non-premultiplied colors. To distinguish this case, I'll call the generic *non*-premultiplied color component g, with $g = f/a$. We then have another basic calculation:

$$(g, a) \; over \; b \equiv ag + (1 - a)b$$

All of these parameters—f, g, a, and b—are assumed linearly related to light intensity.

Display gamma

Light output from CRT display electronics is not linearly related to input voltage. The light intensity I as a function of the voltage V from the digital-analog converters (DACs) on the display card is approximately $I = V^{\gamma}$. This means that an 8-bit pixel value sent directly to a DAC does not encode intensity linearly.

Many people who generate images with painting programs just pick colors that look good in the final image, little knowing that they are implicitly generating pixel values with the gamma of their personal monitor burned into them. And different monitors have different gamma values, explaining why an image created on one system might not look good on another system. There are, however, standards for explicitly specifying the translation from nonlinearly-encoded pixel values to a linear light-intensity space. For example, digital video is typically encoded with a gamma value of about 2.2. On the computer side, an evolving nonlinear encoding standard called sRGB uses a slightly different function, but also approximates a gamma of 2.2 (www.color.org/contrib/sRGB.html).

This nonlinear encoding actually has an advantage in that it approximately models the perceptual space of the eye. When a nonlinearly encoded pixel is quantized into the typical 8-bit byte, the jumps between pixel values are roughly equal perceptually.

Linear/nonlinear notation

For notational convenience, I will use tildes to identify stored pixel values that are nonlinearly related to light intensity: $\tilde{f}$, $\tilde{a}$, $\tilde{b}$. This nonlinear encoding doesn't change the range of values of a pixel; all values, tilded or not, range in value from 0 to 1.

I will write the function that transforms a stored pixel value to a linear intensity value as L, for "Linearize." We thus have

$$f = L(\tilde{f})$$

I will name the inverse of this function N, for "Nonlinearize," so

$$\tilde{f} = N(f)$$

Here are a few identities:

$L(0) = 0; L(1) = 1$
$N(0) = 0; N(1) = 1$

The following relation is precisely true only for pure power functions, but is close enough for others typically in use:

$$L(\tilde{a}\tilde{b}) = L(\tilde{a})L(\tilde{b})$$
$$N(ab) = N(a)N(b)$$

Converting into and out of linear space was the subject of my earlier column "Dirty Pixels" (*IEEE CG&A*, July 1989).

The good, the bad, the ugly

Let's now state the problem in algebraic terms, using the above notation. Given linearly encoded input pixel values, calculating the linearly encoded result is simple:

$$r = f + (1-a)b$$

This, for example, is the calculation I optimized in my earlier column "Fugue for MMX" (*IEEE CG&A*, March/April 1997, p. 88).

On the other hand, given nonlinearly encoded input pixel values, to get a nonlinearly encoded result the calculation should be

$$\tilde{r} = N\left(L(\tilde{f}) + \left(1 - L(\tilde{a})\right)L(\tilde{b})\right)$$

This is basically the technique I presented in "Image Compositing: Practice" (*IEEE CG&A*, Nov. 1994, p. 78). It's slower, but still accurate.

On the third hand, most image manipulation programs don't do this. This might either be because programmers are unaware of the problem, or because they are scared of the necessary arithmetic. They simply pretend the problem isn't there and calculate on the nonlinear pixel values as though they were linear. This means that they approximate the correct value with

$$\tilde{r}_{approx} = \tilde{f} + (1-\tilde{a})\tilde{b}$$

So how bad is this? Let's see.

Scenarios

Before getting into error analysis, I want to generalize a bit. Actually, several varieties of situations can occur depending on the encoding format of the new foreground image. I will discuss four possibilities:

- $(\tilde{f}, \tilde{a})$ *over* $\tilde{b}$. Both foreground color and transparency components are nonlinear. This is the example shown above.
- $(\tilde{f},\ a)$ *over* $\tilde{b}$. The alpha component is coded linearly. This might seem to make sense; since alpha represents the geometric coverage of a pixel, a display device-dependent alpha might seem wrong. As it turns out, this technique has many problems.
- $(f,\ a)$ *over* $\tilde{b}$. Linear intensity and linear alpha. The foreground pixel might come from some algorithmic calculations such as antialiased lines or a 3D rendering system.
- $(\tilde{g},\ a)$ *over* $\tilde{b}$. Nonlinear non-premultiplied foreground image. Both images might come from a scanner or paint program, while the alpha value comes from some algorithmically generated blending matte.

In each case, the correct result comes from linearizing any nonlinear input values, doing the calculation, and then re-nonlinearizing the result.

Error analysis

We will want to know three things about each approximation:

- When is it correct?
- When is it the most incorrect?
- When are these inaccuracies most visible?

When correct? The approximation is correct whenever

$$E \equiv \tilde{r}_{approx} - \tilde{r} = 0$$

You can usually just look at the formulas for $\tilde{r}_{approx}$ and $\tilde{r}$ and guess at some parameter ranges where the error is zero, usually at the extreme values of the parameters. A more general technique, though, is to solve for the surface defined mathematically by $E=0$. This can be made less messy by approximating the L function with $L(\tilde{x}) \approx \tilde{x}^2$. The resulting equation will be more algebraically tractable and can give us hints at where the true error function is zero. I'll use this approximation to the L function a lot, so I'll give it the name "G2".

Most incorrect? We can usually guess that the maximum error will be on a face or edge of the parameter space. This can reduce the calculation from a three-parameter ordeal to a one- or two- parameter cakewalk. When I solve for these, I will give them the names E_+ for the most positive error and E_- for the most negative error.

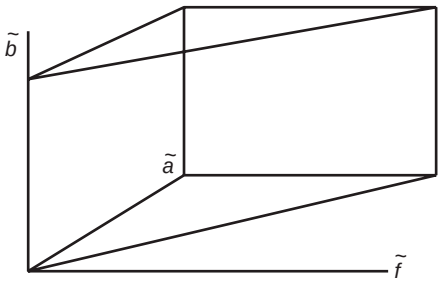
Most visible? The worst possible situation for an approximation is when a transparent color is composited over a background of the same color. Ideally the result should be the original background color. Mathematically this situation is where

$$f + (1-a)b = b$$

You can rearrange this to get

$$f/a = b \text{ or } f = ab$$

This makes sense intuitively, since f/a is the color of the foreground pixel with the transparency divided out. Any



1 Parameter space for scenario 1.

errors in the approximation on the surface $f = ab$ will be the most visible. I'll denote errors on this surface as E_v.

Scenario 1: $(\tilde{f},\ \tilde{a})$ *over* $\tilde{b}$

The calculation, as stated above, is

$$\tilde{r} = N\Big(L\big(\tilde{f}\big) + \big(1 - L\big(\tilde{a}\big)\big)L\big(\tilde{b}\big)\Big)$$
$$\tilde{r}_{approx} = \tilde{f} + \big(1 - \tilde{a}\big)\tilde{b}$$

The possible values of $\tilde{f}$ and $\tilde{a}$ fill a triangle in the $\tilde{f}\,\tilde{a}$ plane. The possible values of $\tilde{b}$ extrude this triangle into the third dimension. See Figure 1.

Where is the approximation correct? Well, plug $\tilde{a}=1$ into $\tilde{r}$ and $\tilde{r}_{approx}$ and you see that they simplify down to the value $\tilde{f}$. This is not too surprising; if the foreground pixel is opaque, it completely replaces the background pixel with no error, even with the approximation.

Plug $\tilde{b}=0$ into the expressions and they also simplify to $\tilde{f}$. In other words, when compositing over a black background, the approximation always gets the right answer. So far so good.

To find the general case for zero error we do the following trick. Equate $L(\tilde{r}) = L(\tilde{r}_{approx})$ to get:

$$L\big(\tilde{f}\big) + \big(1 - L\big(\tilde{a}\big)\big)L\big(\tilde{b}\big) = L\Big(\tilde{f} + \big(1 - \tilde{a}\big)\tilde{b}\Big)$$

Now use the approximation that $L(\tilde{x}) \approx \tilde{x}^2$ to get

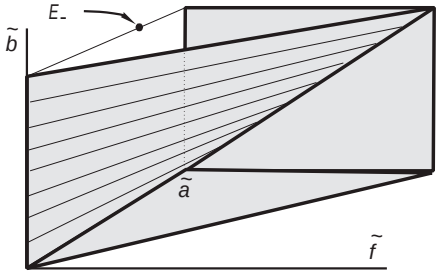
$$\tilde{f}^2 + \big(1 - \tilde{a}^2\big)\tilde{b}^2 = \Big(\tilde{f} + \big(1 - \tilde{a}\big)\tilde{b}\Big)^2$$

A bunch of algebraic canceling and factoring, which I won't bore you with, gives us the equation of the general surface in $(\tilde{f},\ \tilde{a},\ \tilde{b})$ space where this approximate error is zero.

$$\tilde{b}\big(1 - \tilde{a}\big)\big(\tilde{f} - \tilde{a}\tilde{b}\big) = 0$$

The first two factors are the conditions we knew already. The third, amazingly enough, is just our maximum-visibility surface. In fact, plugging it back into the original error formula shows that it is accurate even for general L functions. In other words, $\tilde{r} = \tilde{r}_{approx}$ in just the visual situations where errors would be most apparent, so

$$E_v = E\big(\tilde{a}\tilde{b},\ \tilde{a},\ \tilde{b}\big) = 0$$



2 Zero-error surfaces for scenario 1.

Now let's try to find where the error is *worst* over the space of possible values of $(\tilde{f},\tilde{a},\ \tilde{b})$. In Figure 2 I've shaded in the three surfaces where the approximation is exact: $\tilde{b}=0$ (the triangular floor), $\tilde{a}=1$ (the back wall), and $\tilde{f} = \tilde{a}\,\tilde{b}$ (the twisting surface that sweeps from the $\tilde{a}$ axis to the $\tilde{f}=\tilde{a}$ edge at $\tilde{b}=1$). This latter surface divides the parameter volume into two regions: one where the approximation is too high, and one where it's too low. The location of maximum errors will be on the walls farthest from this dividing surface. The largest negative error will be on the $\tilde{f}=0$ wall; the largest positive error will be on the $\tilde{f}=\tilde{a}$ wall. Let's find these.

The largest negative error will be on the surface

$$\begin{aligned} E\big(0,\ \tilde{a},\ \tilde{b}\big) &= \Big(\big(1 - \tilde{a}\big)\tilde{b}\Big) - N\Big(\big(1 - L\big(\tilde{a}\big)\big)L\big(\tilde{b}\big)\Big) \\ &= \big(1 - \tilde{a}\big)\tilde{b} - N\big(1 - L\big(\tilde{a}\big)\big)\tilde{b} \\ &= \tilde{b}\Big(\big(1 - \tilde{a}\big) - N\big(1 - L\big(\tilde{a}\big)\big)\Big) \end{aligned}$$

The error is proportional to $\tilde{b}$, so it is largest where $\tilde{b}=1$. We can approximate the worst possible value for $\tilde{a}$ along this parameter edge by using G2. This gives us

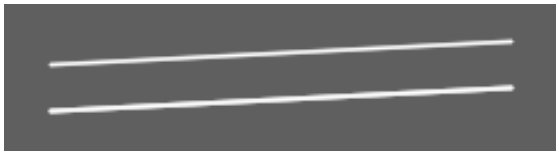
$$E\big(0,\ \tilde{a},\ 1\big) \equiv 1 - \tilde{a} - \big(1 - \tilde{a}^2\big)^{1/2}$$

Setting the derivative of this to zero and solving for $\tilde{a}$ gives us $\tilde{a} = \sqrt{1/2}$, with the error value itself being

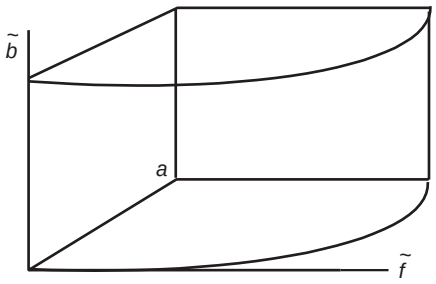
$$E_- \equiv E\big(0,\ \sqrt{1/2},\ 1\big) = 1 - \sqrt{2} \approx -.414$$

This is a pretty big error out of a total parameter range of 0 to 1! If these values are scaled to an 8-bit byte, the

3 Correct (top) and approximate (bottom).



4 Correct (top) and approximate (bottom).



5 Parameter space for scenario 2.

correct r is 180, while the approximate calculation gives 75, for an error of –105.

Visually, this means that placing transparent black over white would result in a much darker value than it should. How bad could that be? Just looking at the resulting gray square, you wouldn't have any reference for what the correct value was. What's the worst possible visual effect this can produce?

One place where accurate tonal reproduction is important is in antialiasing. Figure 3 shows an antialiased black line drawn with correct and approximate arithmetic. The approximate line is darker, appears too wide, and the antialiasing has been messed up. I won't vouch for the transfer function of the printing of this figure, but the approximate line looks mighty ropey on my monitor. (Look on my Web site for digital versions of the images to see for yourself how bad they look.)

Next, the largest positive error of our approximation occurs on the front surface of the parameter space, where $\tilde{f} = \tilde{a}$. This represents various transparencies of white. The location of maximum error is harder to solve for explicitly, but with a little numerical experimentation I found the maximum problem at

$$E_+ \equiv E(.3, .3, .37) \approx .559 - .463 = .096$$

The visual interpretation of this is that transparent white on a gray background will look too bright. Figure 4 shows a white line on a gray background. It's not as ropey as Figure 3, since the maximum error is much smaller.

Scenario 2: $\tilde{f}$, a over $\tilde{b}$

This is just like scenario 1, but with the alpha component encoded linearly. I originally thought this was a good idea, advocating it in my earlier article "Image Compositing: Practice," but the more I've played with it, the less I like it. The parameter space appears in Figure 5. The $\tilde{f} = \tilde{a}$ surface has become the curved surface $\tilde{f} = N(a)$.

The calculation is

$$\tilde{r} = N\left(L\left(\tilde{f}\right) + \left(1-a\right)L\left(\tilde{b}\right)\right)$$
$$\tilde{r}_{approx} = \tilde{f} + \left(1-a\right)\tilde{b}$$

Let's see what effects this coding has on the approximation error.

We again find the zero error surfaces by equating $\tilde{r} = \tilde{r}_{approx}$, applying L to both sides and using G2. Apply some algebra, and the surfaces of zero error reveal themselves:

$$0 = \tilde{b}\left(1-a\right)\left(a\tilde{b} - 2\tilde{f}\right)$$

Once again, we get no error for compositing over a black background ($\tilde{b} = 0$) or wherever the foreground object is opaque ($\tilde{a} = 1$). The third error-free surface term presents a problem, however. It shows that we no longer have the nice property of zero error where $f/a = b$. This means that it is possible to apply one transparent color over the same colored background and get a visible seam. We can find the worst case situation by substituting $\tilde{f} = \tilde{a}\tilde{b} = N(a)\tilde{b}$ into the error calculation to get

$$E\left(N(a)\tilde{b},\, a,\, \tilde{b}\right)$$
$$= \left(N(a)\tilde{b} + \left(1-a\right)\tilde{b}\right) - N\left(aL\left(\tilde{b}\right) + \left(1-a\right)L\left(\tilde{b}\right)\right)$$
$$= \left(N(a)\tilde{b} + \tilde{b} - a\tilde{b}\right) - \tilde{b}$$

So

$$E_v = \tilde{b}\left(N(a) - a\right)$$

The maximum error on this $\tilde{f} = \tilde{a}\tilde{b}$ surface is at $\tilde{b} = 1,\ a \approx 1/4$. The visual interpretation of these parameter values is that we are painting a one-quarter-transparent white over opaque white. The correct answer is, of course, white: $\tilde{r} = 1$. The approximation, however, gives a value of $\tilde{r}_{approx} = 5/4$. Wow! Not only is it too big, but it overflows the range 0 to 1. Thus, the "ghost in a snowstorm" (transparent white over white) would be visible using the approximate calculation. Even opaque white over white causes problems if the edges are antialiased. The alpha of the pixels range from 0 outside the edge to 1 inside it. The error in the approximation, as calculated above, is zero for $a = 0$ and for $a = 1$ and maximal in between. The inside of the ghost is invisible, and the outside of the ghost is invisible. Just the edges will show up. Figure 6 shows a (linear) ghost—I've drawn it as light gray over light gray to avoid the overflow problem. I have also made the line thicker and

slightly darker than the background so that it shows up for comparison purposes.

What other errors are there? The maximum negative error over the entire parameter space is still along the $\tilde{b}=1, \tilde{f}=0$ edge where

$$E(0, a, 1) = (1-a) - N(1-a)$$

Using G2 we can approximate the worst case at

$$E_{-} \equiv E(0, 1/4, 1) = -.116$$

Thus, a black antialiased line over a white background will look too dark and ropey, though not so much so as with scenario 1. See Figure 7.

Scenario 3: (f, a) *over* $\tilde{b}$

This common scenario occurs when a program generates the foreground image and alpha in linear space by some algorithmic process such as a 3D renderer. This is what happens when we try to overlay computer graphics on top of digital video, for example. The calculations are

$$\tilde{r} = N\left(f + (1-a)L\left(\tilde{b}\right)\right)$$
$$\tilde{r}_{approx} = f + (1-a)\tilde{b}$$

Our technique of finding the general zero-error surface generates a second-order expression that isn't factorable into three simple surfaces. Instead of pursuing this, let's look at a few key locations in parameter space to get a idea of the error shape.

Checking the edges of the parameter space, we find that the error is zero along the five edges:

$$\left(f, a, \tilde{b}\right) = \left(0, 0, \tilde{b}\right); \left(0, 1, \tilde{b}\right); \left(1, 1, \tilde{b}\right); \left(f, f, 1\right); \left(0, a, 0\right)$$

I've drawn these edges darkened in Figure 8.

Looking at the bottom face, $\tilde{b}=0$, we discover that

$$E(f, a, 0) = f - N(f)$$

Use G2 to find an approximate maximum value of

$$E_{-} \equiv E(1/4, a, 0) = -1/4$$

Note that this maximum error region is a line in parameter space rather than at just one point.

Drawing a white line over a black background will generate parameter values along the $f=a, \tilde{b}=0$ edge that pass across this region and show a maximal problem. The error as a function of a will be

$$E(a, a, 0) = a - N(a)$$

See Figure 9.

Next, on the top face of the parameter space, $\tilde{b}=1$, we have

$$E(f, a, 1) = (f - a + 1) - N(f - a + 1)$$

Solving for the maximum error gives

$$E_{-} \equiv E(f, f + 3/4, 1) = -1/4$$

Again, this maximum error region is a line in parameter space.

An antialiased black line over a white background generates parameter values that pass across this maximum error region. The error here is

$$E(0, a, 1) = (1-a) - N(1-a)$$

This is the same as the line in Figure 7.

Finally, let's look at the error on the maximum visibility surface $f = ab = aL(\tilde{b})$. Plug this into the error definition:

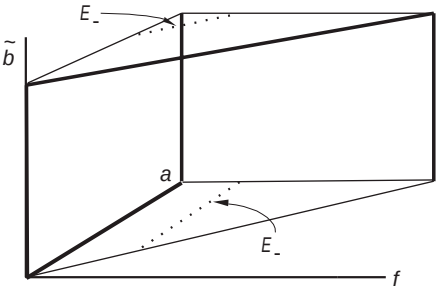
$$E_v = E\left(aL\left(\tilde{b}\right), a, \tilde{b}\right) = a\left(L\left(\tilde{b}\right) - \tilde{b}\right)$$

Thus the error will be maximum at approximately

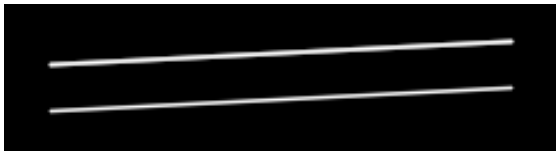
$$E_{v-} \equiv E(1/4, 1, 1/2) = -1/4$$

6 Linear ghost. Correct (top, invisible) and approximate (bottom).

7 Correct (top) and approximate (bottom).

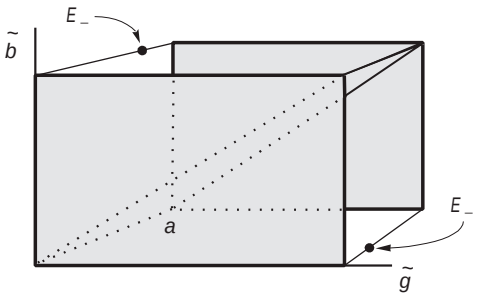


8 Zero error edges and maximal error lines for scenario 3.



9 Correct (top) and approximate (bottom).

10 Gray ghost. Correct (top, invisible) and approximate (bottom).



11 Parameter space for scenario 4 with zero-error surfaces shaded and maximum error points identified.

Here again we have the "ghost in a snowstorm" effect by compositing dark gray over dark gray. See Figure 10. Here, though, the error is worst when $a = 1$, so the ghost appears as a darker body rather than as an outline.

Scenario 4: $(\tilde{g}, a)$ *over* $\tilde{b}$

This scenario occurs when we blend two nonlinearly encoded images, such as painted or scanned images, with a linearly encoded matte. The calculation is

$$\tilde{r} = N\left(aL\left(\tilde{g}\right) + (1-a)L\left(\tilde{b}\right)\right)$$
$$\tilde{r}_{approx} = a\tilde{g} + (1-a)\tilde{b}$$

The error is zero at $a = 0$, $a = 1$, and $\tilde{g} = \tilde{b}$. The latter is also the maximum visibility surface for this scenario, so we get no ghost effects:

$$E_v = 0$$

These zero error surfaces are colored gray in Figure 11.

The error is always negative wherever it is not zero. The locations of maximal error are along the edges shown in the figure:

$$E(1, a, 0) = a - N(a)$$

and

$$E(0, a, 1) = (1-a) - N(1-a)$$

Applying G2, we find these approximately at

$$E_{-} \equiv E(1, 1/4, 0) = -1/4$$
$$E_{-} \equiv E(0, 3/4, 1) = -1/4$$

Once again, the worst case is with white over black, or with black over white. The lines look the same as with Figure 7 and Figure 9.

What do we do about this?

I see three ways to deal with this situation, none of them ideal:

1. Live with error as a real-time approximation to the correct image.

The problem with this is that many image editing programs that do this approximation are not really real-time applications; the image is ultimately destined for a printer somewhere. These programs can very well afford to take the time to generate the correct image. Another problem with this cop-out in real-time systems is that moving images will look optimally bad because the antialiasing is defeated.

2. Encode all pixels linearly and perform calculations in this space. Then, for display, translate to the appropriate nonlinear display space by using a hardware look-up table. The non-CRT displays of the future will require linear inputs, so no look-up table will be necessary.

My objection to this, at least in the 8-bit arena, has to do with quantization error. A look-up table that translates 8-bit pixels into 8-bit DAC inputs through such a nonlinear table will, of necessity, map several pixel values at the low end into one output DAC value. As mentioned in "Dirty Pixels," for a gamma of 2, only 194 distinct values will appear in the table. To preserve the same resolution as you have in nonlinear 8-bit encoding when you go to linear encoding, you need a 17-bit number (but 16 bits might be good enough). This is probably too much for a hardware look-up table. Another problem with this technique is that an extensive library of nonlinearly encoded images (both stills and video) already exists.

3. Linearize pixel values before doing calculation and unlinearize after you finish.

This is the slow but correct approach. The search for various ways to speed up these conversions could be a fruitful area for future research. In "Dirty Pixels" I sped up the calculation as much as I could by using a table look-up for L and a binary search for N. Another approach might be to use higher-order approximating functions that are more accurate but that stop short of doing exponential calculations. ■