Image Compositing Fundamentals

Technical Memo 4

Alvy Ray Smith
August 15, 1995

Abstract

This is a short introduction to the efficient calculation of image compositions. Some of the techniques shown here are not well known, and should be. In particular, we will explain the difference between premultiplied alpha and not[1]. These two related notions are often confused, or not even understood. We shall show that premultiplied alpha is more efficient, yields more elegant formulas, and occurs commonly in practice. We shall show that the non-premultiplied alpha formulation is not closed on **over**, the fundamental image compositing operator—as usually defined. Most importantly, the notion of premultiplied alpha leads directly to the notion of *image object*, or *sprite*—a shaped image with partial transparencies.

The Basic Model

There are two ways to think of the alpha of a pixel. As is usual in computer graphics, one interpretation comes from the geometry half of the world and the other from the imaging half. Geometers think of "pixels" as geometrical areas intersected by geometrical objects[2]. For them, alpha is the percentage *coverage* of a pixel by a geometrical object. Imagers think of pixels as point samples of a continuum. For them, alpha is the *opacity* at each sample. In the end, it is the imaging model that dominates, because a geometric picture must be reduced to point samples to display—it must be rendered. Thus, during rendering coverage is always converted to opacity, and all geometry is lost.

The Porter-Duff matting algebra [PorterDuff84] that underlies what we present here is based on a model that is easiest to understand by alternating between the two conceptions[3].

The elementary imaging operation that we wish to elaborate is called, in [PorterDuff84], the **over** operator. It captures the notion of compositing image J

---

[1] These are called *associated* and *unassociated* alpha as well. I can never remember which is which so don't use the terms.

[2] A little square is a very common model for the "pixel". I place this term in quotes to remind us that this is not a pixel (a sample) but a model for possible geometric contributions to the final sample. The last thing I want to promulgate is the notion that a pixel is a little square.

[3] The Porter-Duff paper is an excellent example of why the little square model for contributions to a pixel has become confused, in the geometry-based computer graphics world, with the pixel itself. All illustrations in that paper use the little square model. A unit circle could have been used equally effectively, however—or any other unit area region.

over image I, where either I or J or both may be partially transparent. For ease, we will think of images I and J as being rectangular, the same size, and each having four channels—three for RGB color and one for alpha (ie, opacity).

Think of the following geometrical model: A "pixel" is an area α percent covered by an opaque geometrical object with color A. Thus the amount of color contributed by that area is αA. That is, we average the color over the pixel and come up with a single new color representing the entire area—the color αA is a point sample.

Now think of another opaque geometrical object with color B added to the original "pixel" area. Disregard for a moment the other geometrical object there. Assume that the new geometrical object has coverage of the "pixel" equal to β. So the pixel is contributing color βB due to this object. This again is a point sample representing the color of the second object.

But now we use the geometry model to conceptually combine the contributions of the two objects in the "pixel" area. The second object is allowing only $(1-\beta)$ percent of the pixel area to be transparent to any objects behind it. We simply ignore the actual geometry of the two objects at this point and assume that, in general, the pixel is allowing $(1-\beta)$ times the color from behind, αA, to show. This is added to the color due to the top object βB. So the total color of object with color B over object with color A is $\beta B + (1-\beta)\alpha A$.

Notice that this result could be completely wrong if the geometry of the second object exactly coincided with that of the first. The bottom color would not contribute at all to the final color in this special case. So the model we are using is an approximation for the general case of combining two images where we no longer have any idea of how the alpha at a point was determined. In an image there is no way to tell whether a point sample with a partial opacity comes from a partially transparent surface or from an opaque surface partially occluding the area represented by the point sample.

Premultiplied Alpha

The formula we have just derived from basic principles is this: For composite color C obtained by placing a pixel with color B and alpha β over a pixel with color A and alpha α:

$$C = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A$$

Notice how many multiplies this formula implies—three[4] at each pixel for each color component. Considering that this formula is extremely basic to computer graphics and that multiplies are expensive[5], the early researchers at Lucasfilm and Pixar observed that this formula could be reduced to one multiply per pixel per component if the alphas were *premultiplied* times the color of an image.

[4] Two, actually, with a little rearrangement: $T = \alpha A$, $C = \beta(B - T) + T$.

[5] They were especially expensive then. Now we would just like to avoid extra steps.

That is, if the color channels of image I contained, not color A, but weighted color αA, and similarly for image J, then the formula above reduces to

$$C' = B' + (1-\beta)A' = B' + A' - \beta A'$$

where the primes indicate colors have been premultiplied by their corresponding alphas. The images are said to have *premultiplied alphas.* Of course, it is the color channels that are different, not the alpha channels, despite this terminology.

There is a subtlety here that will cause trouble if not identified. We have called the resulting color here C' as if it were different from the color C computed above in the non-premultiplied alpha case. *But it isn't!* It is the same computation, where entities on the right have been abbreviated because of premultiplication. We will return to this problem later.

Images with premultiplied alphas have been used for many years very successfully by Lucasfilm, Pixar, and Altamira in hundreds of thousands, if not millions, of images. The TIFF image storage format is aware, as of version 6.0, of premultiplied alphas.

Composite Alpha

We have given the formulas above for the color channels in a composite of two partially transparent images. What is the composite alpha channel formula? Notice that it will be the same for both cases, since premultiplication only applies to the color channels.

The same model as used above for composite color can be used for composite alpha. The average opacity of the "pixel" partially covered by the first geometric object is β, and that for the second geometric object is α. But the geometry of the model allows only $(1-\beta)$ of the lower light filter to be effective. So the composite alpha is

$$\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$$

in either case, premultiplied or not.

An Elegant Formulation and a Flawed One

Let's collect together the results from above.

Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$C' = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A$$
$$\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$$

Compositing Formula for **over**, Colors Premultiplied by Alpha:

$$C' = B' + (1-\beta)A' = B' + A' - \beta A'$$

In the latter case, we need only one formula to represent the color channels *and* the alpha channel, a more elegant formulation certainly than the former case that requires a formula for the color channels different from that for the alpha channel.

Now we will see why the former case is flawed.

The "Second-Composition" Problem

You may have noticed that this time I used C' for the left side of the non-premultipled colors case, since it has already been observed that C and C' are the same color in either formulation. Recall that the prime indicates a color that has been premultiplied by its alpha. So you see the problem: The first formulation maps non-premultiplied colors into premultiplied colors. The second maps premultiplied colors into premultiplied colors. In other words, the usual definition of **over** for non-premultiplied images is not closed on **over**, a problem we will fix below.

This problem is called the "second-composition" problem because it shows up in second (or subsequent) compositions using results from first (or earlier) compositions. Let image K be the result obtained above for J **over** I. Suppose we want to perform a second non-premultiplied composition of L **over** K, where image L has color D and alpha δ. In order to use the formula above we need the non-premultiplied color of K and its alpha γ. The non-premultipled color of K is C' divided by γ. So the *second-composition* formula is

2nd-Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$E' = \delta D + (1-\delta)\gamma(C'/\gamma) = \delta D + (1-\delta)C' = \delta D + C' - \delta C'$$

The alpha channel calculation is as before, and the premultiplied case works as before. That is, there is no second-composition problem for the premultiplied case—another example of its relative elegance.

But the non-premultiplied case is a mess. One either has to divide through by the new alpha at each pixel in order to use the original (first-composition) formulas, or one has to carry around a mixture of premultiplied and nonpremultiplied information to use the simpler second-composition formulas.

So here are the cleanest formulations for the two cases, where we do not clutter up our minds with two different models during the course of a series of compositions and where there is no need for the confusing first- and second-composition distinction—ie, these formulations are closed on **over**[6]:

Closed Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$\gamma = \beta + (1-\beta)\alpha = \beta + \alpha - \alpha\beta$$
$$C' = \beta B + (1-\beta)\alpha A = \beta B + \alpha A - \beta\alpha A$$
$$C = C'/\gamma$$

Closed Compositing Formula for **over**, Colors Premultiplied by Alpha:

$$C' = B' + (1-\beta)A' = B' + A' - \beta A'$$

[6] I just discovered (November 5, 1996) in a correspondence with Marc Levoy that he and Bruce Wallace came up with an equivalent formulation for the nonpremultiplied case in [Wallace81], p. 257.

Many practitioners are unaware of the second-composition problem because they often only do one composition—eg, as the last stage of a 3D rendering project: all the objects are rendered as sprites[7], then they are composited, and never used again. Or more importantly, composites of them are never used for future composites. It is the modern world of cheap memory that has made it possible and common to recomposite a set of sprites many times and to use composites of composites frequently.

Non-Premultiplication Problems

The analysis above looks pretty bad for the non-premultiplied case, but let's look at it more closely. The bad step is the divide by alpha to return a non-premultiplied color. In the typical case of integer colors and integer alphas, this leads to inaccuracy. And it is not even possible if alpha is 0. But we can sometimes avoid this divide and / or loss of information.

A new alpha of 0 at a composite pixel means (1) that, in the coverage model, neither image I nor J was present at that pixel, or (2) that, in the opacity model, both colors are known but both opacities are 0, or (3) a mixture of these. In case (1), we know that there is no defined color so could store an indication of this in the color channels. In case (2), both colors have to be summarized somehow as one color—e.,g. an equal mixture of the two is stored. There *must* be a loss of color mixing information here. In case (3), one image is not present and we know the color of the other, so there is no problem.

In the non-0 alpha case, if we have a geometric model of the contributions to a pixel, then we can compute, in the reals, what the mixture of colors at the pixel should be—as opposed to using the integer divide technique above. If all we have is an opacity model at the pixel, then again there must be a loss of color mixing information.

This analysis shows that we can improve the non-premultiplied case but not completely. But, of course, there is no such problem if one is guaranteed to use a sprite for compositing exactly once. More carefully, there is no problem if one is guaranteed to never use the results of a composition for future compositions.

Premultiplication Problems

Is there *anything* wrong with the premultiplied alpha case? Well, yes there is. There are times when one wants the full non-premultiplied color of a pixel. This requires a divide by the corresponding alpha, hence the problems with integer divide and loss of information mentioned above.

So what to do? It seems clear that for reusable sprite objects—particularly reusuable composites of them—the premultiplied case is superior, except for the problem just mentioned. How often does it occur? And how substantial is it when it does occur?

[7] I am loosening the terminology here, temporarily, to extend "sprite-hood" to non-premultiplied images with an alpha.

My experience in the graphic arts use of sprites—the Altamira Composer image compositing application, for example—is that the error introduced by the occasional need to divide out alpha is typically so minor as to be unnoticed. In fact, no user has ever noticed it in Altamira Composer to my knowledge. The divide by zero problem never occurs because, by definition, a clear pixel (alpha and all color components equal to 0) does not "exist" so is ignored.

I am reminded of a division of the geometry-based computer graphics world into what is usually called CAD and, say, CGI. The distinction is that CAD requires *accurate* geometry because it is being used by architects and engineers. CGI is only required to look good. Accuracy can be, and often is, sacrificed in CGI to get a satisfactory look quickly.

The point is that there is a similar division of the sampling-based side of the computer picturing world, based on user type or market. Clearly, accuracy is very important to such users of images as medical doctors and astronomers. But use of images in the graphics arts is much more forgiving. Here, again, the result must be pleasing rather than accurate.

Some Useful Approximations

We derive now some very useful integer approximations for the implied floating point operations in the formulas above. These apply in the case of the very common 8-bit channel—eg, 24-bit color image plus 8-bit alpha.

The integer approximations below are derived from the geometric series

$$a + ar + ar^2 + ar^3 + \ldots = a / (1-r)$$

for $|r| < 1$. We apply the series this way: Let $r = 1/256$. Notice that

$$t / 255 \equiv (t / 256) / (1 - r).$$

Thus, given two numbers a and b, each on [0, 255] and with product t on [0, 255^2], we get $t/255$ on [0, 255]—as desired—by using the first two terms of the geometric series:

$$(t>>8) + (t>>16) + (t>>24) + \ldots$$

Notice that

$$(t>>8) + (t>>16) \equiv ((t>>8) + t) >> 8 \equiv ((t<<8) + t) >> 16.$$

This is captured by the INT_MULT() definition below which assumes a and b are each on [0,255], t is an **int** temporary variable that holds the product $a*b$, which is returned on [0,255], as if one of a or b were a fraction on [0,1] used to weight the other—eg, as an alpha. In the style of the language C:

```
#define INT_MULT(a,b,t)    ( (t) = (a) * (b), ( ( ( (t)>>8 ) + (t) ) >>8 ) )
```

We now use the INT_MULT() function to define other useful approximations. (We will present an even better macro for it below.)

Classic linear interpolation—or *lerp*—as it is affectionately called in computer graphics—is defined in floating point below. It is read "lerp p to q by alpha a". a is assumed to lie on [0, 1]. Note that $a \equiv 0$ implies p; $a \equiv 1$ implies q.

```
#define FLOAT_LERP(p, q, a)    ( (a) * ( (q) - (p) ) + (p) )
```

In this integer version *t* is an **int** temporary variable:

#define INT_LERP(p, q, a, t) ((p) + INT_MULT(a, ((q) - (p)), t))

Premultiplied lerp assumes *q* has been premultiplied by *a.*

#define FLOAT_PRELERP(p, q, a) ((p) + (q) - (a) * (p))

In this integer version *t* is an **int** temporary variable:

#define INT_PRELERP(p, q, a, t) ((p) + (q) - INT_MULT(a, p, t))

So our formulas for composition (with ‘ (prime) consistently representing premultiplication) become, in the 8-bits per channel case:

8-Bit Compositing Formulas for **over**, Colors Not Premultiplied by Alpha:

$$C' = \text{INT\_LERP}(\text{INT\_MULT}(A, \alpha, t_0), B, \beta, t_1)$$
$$\gamma = \text{INT\_PRELERP}(\alpha, \beta, \beta, t)$$
$$C = C' / \gamma$$

8-Bit Compositing Formula for **over**, Colors Premultiplied by Alpha:

$$C' = \text{INT\_PRELERP}(A', B', \beta, t)$$

Caution! The approximations above must be used with care. In particular, the case $\alpha = 1$ is a problem. Note that INT_MULT(255, 255, *t*) is 254, not 255. Also note that INT_PRELERP(255, 255, 255, *t*) is 256, which is even worse. One-bit errors at other than the high or low end of the range are often tolerable, but not at the extremes. In practice, this is not usually a problem. A typical software loop looks for the special cases of $\alpha = 0$ and $\alpha = 1$, and skips the interpolation computation there. These two cases are so common in imaging that this technique saves much computation. We see from the note above that **it is *important* to check for the a = 1 case and avoid the approximation in that case**.

The INT_MULT macro above suffers from 1-bit errors, in about half of the cases. There are better approximations if one does not mind absorbing a little more cost, if special casing is undesirable, or if hardware implementation is the goal. One pointed out to me by Microsoft colleague John Snyder is to use three terms in the power series approximation: (t>>8) + (t>>16) + (t>>24). This loses no bits, but requires 32-bit arithmetic as written.

Another, pointed out by friend and longtime colleague Jim Blinn at Cal Tech, is to use roundoff in the approximation, rather than truncation: ((*t*>>8) + *t* + 0x80)>>8. This is good, but it still suffers from 1-bit errors in a few cases (24 to be exact). Jim determined that rounding *t* before shifting got rid of even these errors. So the best macro is this:

#define INT_MULT(a,b,t) ((t) = (a) * (b) + 0x80, ((((t)>>8) + (t))>>8))

at a cost of one additional add. It has no 1-bit errors and can be performed in 16-bit arithmetic[8]. See [Blinn94a, Blinn94b] for Jim's arguments in support of pre-multiplied alpha.

Image Objects or Sprites

The most important result of using premultiplied alphas is conceptual—the conceptual change from

> **Old Notion:** An image is a *rectangular* array of pixels. The alpha channel, if any, tells how each pixel is to be treated. Each image pixel has a color that may be masked on or off (or partially on) by the corresponding alpha channel pixel.

to

> **New Notion:** An image is a shaped array of pixels with partial transparencies. The alpha channel is intrinsic. The image pixels at transparent pixels (alpha zero) simply do not conceptually exist.

This new notion is captured in the **sprite** object (or image object, as I formerly called it).

I argue strongly for Microsoft (and wider) adoption of and promulgation of the premultiplied alpha concept that led us to the notion of sprite, and for the elegance of its formulation. It has problems but they are far fewer than those for the alternative.

Acknowledgement

I keep learning about the subtleties of images and alphas by writing papers such as this. The second-composition problem was never clear to me before. I owe John Bradstreet—former colleague of mine at Pixar and currently in Microsoft's Consumer Imaging group—thanks for a probing and ultimately inspiring question, raised by an early draft of this paper. Thanks also to Jim Kajiya for his comments on closure, and to Jim Blinn and John Snyder on improved lerp approximations.

References

[Blinn94a] Blinn, James F., *Jim Blinn's Corner: Compositing Part 1: Theory*, **IEEE Computer Graphics & Applications**, Sep 1994, 83-87.

[Blinn94b] Blinn, James F., *Jim Blinn's Corner: Compositing Part 2: Practice*, **IEEE Computer Graphics & Applications**, Nov 1994, 78-82.

[PorterDuff84] Porter, Thomas, and Duff, Tom, *Compositing Digital Images*, **Computer Graphics**, Vol 18, No 3, Jul 1984, 253-259. SIGGRAPH'84 Conference Proceedings.

[Wallace81] Wallace, Bruce A., *Merging and Transformation of Raster Images for Cartoon Animation*, **Computer Graphics**, Vol 15, No 3, Aug 1981, 253-262. SIGGRAPH'81 Conference Proceedings.

[8] And can be realized in one register in six Intel instructions! Here they are (also by Jim Blinn and independently by Microsoft's David Jones): mov al,a; mul b; add ax,0x80; add al,ah; adc ah,0; mov r,ah.