(note: the skull image that appears below was borrowed *from The Art and Science of Digital Compositing* by Ron Brinkmann. Should these notes ever be published, it will of course be replaced!)

## Blending

As we've learned, the following algorithm allows us to make *localized* changes to the colors in a digital image:

- Assign a value S to every pixel in the image. Assign S = 1 to the pixels we want to change, assign S = 0 to pixels we want to keep the same as the original (or "beginning" image). Assign an intermediate value of S (say, 0.5) to those pixels that we want to be a blend between the beginning and the altered image.
- Apply one or more of our mathematical color-changing operators to our original image. This creates a new, temporary image we'll call P, the processed image.
- Using the different values of S, blend between P and the beginning image B to get a new final image F.

Here it is mathematically:

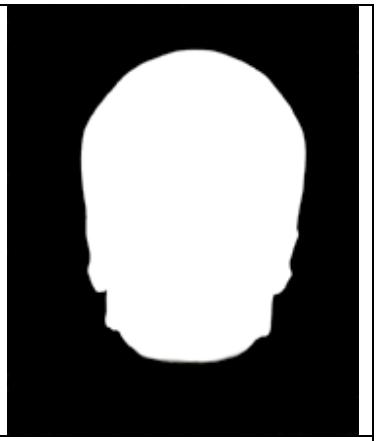
$F = S * A + (1 - S) * B$

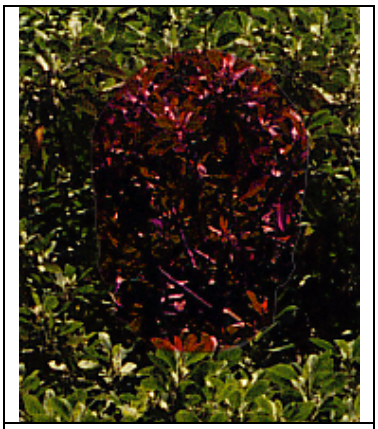
Checking the accuracy of this equation is easy. Where S = 1, the first term evaluates to A and the second term always evaluates to zero, no matter what values are stored in the original image B. Where S = 0, the first term always evaluates to zero and the second term always evaluates to B. If S = .5, F is an equally-weighted mix of half of A and half of B. This looks good!

Visual example (you may want to look at the PDF online-- these are color images):


Beginning image


Altered image


Blending parameter S

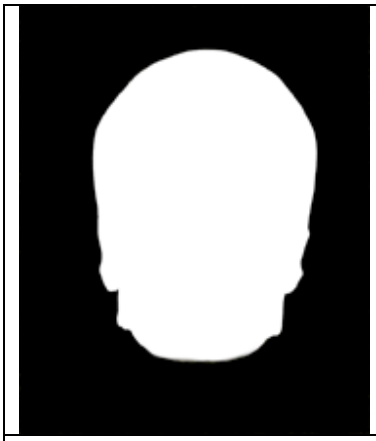


Final image

Up to this point, the image-manipulating operators we've explored have all been operators that take one image as input and return one image as output (recall multiplication, addition, etc). Interestingly enough, the blending equation we just derived can itself be considered an image operator too. One that requires three input images to function.

## Blending Two Completely Different Images

For the moment, forget that the beginning and the altered image have anything to do with each other. They really don't have to: we've put no constraints on the kinds of image-altering operations you can perform on B to get A. So imagine that A is an *entirely different image* than B and consider what the blending equation gives us with the same beginning image that we had above.



Blending image S

"Altered" image A

New Final image F

All of a sudden this looks like something different is happening. What is it? Close (or not so close) comparison of the S image with the new A image shows a high-level of correspondence between the features in each. What I mean by this is that where the skull in A has edges is exactly where the S image has edges. This is of course by design: this example is meant to demonstrate an important feature of the blending operator when the S image features are correlated or "aligned" with those in the A image.

The new final image looks a lot like a picture of a skull that was placed over a bunch of leaves. In fact, if we consider this a little more closely, we can see that the S image is acting as an *opacity* value for A such that when it's blended with B it appears to have been placed *over* B.

## Opacity or Alpha or Matte

This kind of thing happens so frequently in digital imaging that people decided it might be useful to have a fourth value stored with each pixel. Along with the R, G, B triplet for color, images can have an optional opacity or *alpha* value (often called an image's *matte*) that allows for easy compositing using the blending operator. Alpha, just like S, ranges from 0 to 1. 0 is fully transparent, 1 is fully opaque. This changes the blending equation to something more specific, called the "over" operator:

A over B = alpha * A + (1 - alpha) * B

This is just the blending operator with alpha substituted for S. The reason it merits a new operator name is because now alpha is understood to *be a part of image* A. In other words, it isn't a separate image S anymore.

## Premultiplication and Headaches

Now, as unanimous as the decision was to append an alpha value to each pixel value, there are two schools of thought on how to go about doing it.

The first approach is quite simple: Just tack on the alpha value without touching the R, G, or B values. This certainly seems like a good idea. That is, until you imagine a pixel with an alpha value of 0. Alpha of 0 means that the pixel is *completely transparent*. In other words, it is see through. So then does it make any sense for the pixel to have any R, G, or B value other than 0? If you composite this pixel over another one (see the "over" equation), the alpha * A term will always evaluate to 0, no matter what the RGB values are in A. In other words, it seems odd to store color values that mean nothing due to an image's transparency.

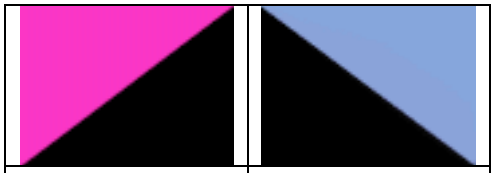
The second approach is to *pre-multiply* the RGB values in the image by the alpha value, thereby "baking" the transparency effect into the colors themselves. Pixels with alpha of 0 will also have RGB equal to 0. Pixels with alpha of 1 will have the original RGB values. An image stored this way is called a pre-multiplied image. It's the most common form, in fact.

The real truth here is that the decision was based on computation speed. A lot of computer math has to be done when evaluating the over equation (lots of "multiplies and adds" in computer-speak). Pre-multiplying an image's RGB by its alpha saves much of this work (the first term of the over equation no longer has a multiplication part if it's been baked into the image). Of course, computers are fast enough these days that we probably wouldn't notice the extra time hit. Ironically, we now have to spend more time teaching and learning about pre-multiplication (and debugging problems related to it)

than the computers would ever have had to spend compositing the un-pre-multiplied images!

## Other Prepositions

Consider the following two images that have alpha channels (opaque where there's color, transparent where there's black):

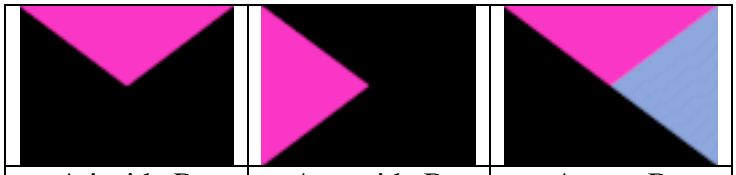


A | B

We just spent a fair amount of time considering the over operator, so the following composite should make perfect sense:

A over B

It's fun (and ultimately useful) to explore what I call the other *preposition* operators. See if you can infer what each of the operators do based on their names and the resultant images:



A inside B | A outside B | A atop B

Here are the answers:

A inside B is computed by taking the R, G, B, α values of A and multiplying them by the α value of B. Why is it called inside?

A outside B is computed by taking the R, G, B, α values of A and multiplying them by the inverted matte image of B (aka 1 - B's α).

A atop B can be thought of as a two-step composite: (A inside B) over B. I use parenthesis here to denote the order of operations.

What might "A under B" look like? (answer: B over A)

Looking at the examples above, it appears that A inside B combined in some way with A outside B should yield A (look at the images yourselves). Let's see if we can figure out how to combine them.

Notation break: now that we are using multiple images, I'm going to denote the R, G, B, and $\alpha$ channels of an image with a subscript to show which image they come from. For instance, the red channel of image B is $R_B$. I will also now consider the image name (A, for instance) to mean ALL channels of image A (namely, R, G, B, and $\alpha$).

Using this new notation, let's re-write the equations for inside and outside.

A inside B = $\alpha_B * A$
A outside B = $(1 - \alpha_B) * A$

Distributing the multiplication in A outside B makes it pretty clear how we can get A back:

A outside B = $A - \alpha_B * A$

So simply adding A inside B to A outside B will give us A back, and thus the following equality:

A = (A inside B) plus (A outside B)

## Matte Lines

Let's try another one. It looks as if A plus (B outside A) is the same as A over B. Take a look and see if you agree with this guess. Let's check it mathematically:

| | | | |
|---|---|---|---|
| B outside A | = | $(1 - \alpha_A) * B$ | |
| A plus (B outside A) | = | $A + (1 - \alpha_A) * B$ | (equation 1) |
| A over B | = | $\alpha_A * A + (1 - \alpha_A) * B$ | (equation 2) |

Aha! There's a little difference. Look at the first terms. In equation 1, we add in the full value of image A. In equation 2, however, we add in A multiplied by its own matte.

In other words, what we thought might be an equality *by inspection* was in fact NOT. Anywhere that A's matte doesn't equal 1 (namely, on the edges) there will be slight differences between the two composites. It may or may not be noticeable in this case, however, I can assure you that some day in your travels you will discover a *matte line*.

In short, a matte line is an unwanted artifact that you will notice on the matte border of an image. Matte lines result from many different things, but most frequently are due to sloppy use of alpha channels and pre-multiplied images. If you notice an artifact like this in your work, you may be able to solve it through careful examination of your compositing operations like we've just done, above.