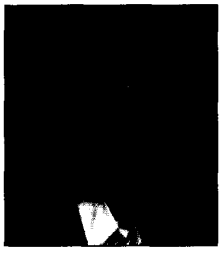
Compositing, Part I: Theory

James F. Blinn, *California Institute of Technology*



Associating a pixel's color with its opacity is the basis for a compositing function that is simple, elegant, and general. But there are more reasons than mere prettiness to store pixels this way.

My currently favorite journalistic quote comes from a magazine called *Morph's Outpost on the Digital Frontier*. They refer to the operation of avoiding jaggies as "anti-aliening." Either this was a typo or they thought of the jaggies as aliens. This got me thinking about ways to get rid of these creatures —the offspring of 3D geometry and raster displays.

One of the most important anti-aliening tools in computer graphics comes from a generalization of the simple act of storing a pixel into a frame buffer. Several people simultaneously discovered the usefulness of this operation, so it goes by several names: matting, image compositing, alpha blending, overlaying, or lerping. It was most completely codified in a paper by Porter and Duff,[1] where they call it the "over" operator. In this column I'm going to show a new way to derive Porter and Duff's "over" operator and describe some implementation details that I've found useful. In a later column I'll go into some of the subtleties of how this operator works with integer pixel arithmetic.

The basic idea

The simplest form of compositing goes as follows. Say we want to overlay a foreground image on some background image. The foreground image only covers a part of the background; pixels inside the foreground shape will completely replace the corresponding background pixels, and pixels outside the shape leave the background pixels intact.

If we want anti-aliened edges, though, things are a bit more complicated. Pixels on the edge of the shape only partially cover the background pixels. If the shape is to be properly anti-aliened, we must blend the foreground color, F, and background color, B, according to the fraction α. This value represents the percentage of the pixel covered by color F. The standard way to calculate this is to find the geometric area covered by F. This implements a simple box filter for anti-aliening. More accurate filters can be used, but I'll stick to the box for now.

Now let's get down to algebra. F and B are each three-element vectors representing the red, green, and blue components of a pixel. Ordinary vector algebra applies. The new color in the frame buffer is

$$B_{new} = (1 - \alpha)\, B_{old} + \alpha F$$

which can be more efficiently calculated as

$$B_{new} = B_{old} + \alpha(F - B_{old})$$

You can actually use the value of α for a variety of things. In addition to its anti-aliening function, it can represent transparent objects or establish a global fade amount. For this reason, the α value also goes by various names: coverage amount, opacity, or simply alpha. You can also think of it as 1 minus the transparency of the pixel. I'm going to call it opacity for now. If it's 0, the new pixel is transparent and does not affect the frame buffer. If it's 1, the new pixel is opaque and completely replaces the current frame buffer color.

Next, suppose that we want to layer another object on top of our image. We just blend in the new object's color, which I'll call G, on top of our current background image using its opacity β,

$$B_{newnew} = (1 - \beta)B_{new} + \beta G$$

We can keep on plastering stuff on top of our image until we are happy. This is the essence of 2-1/2D rendering, also known as the painter's algorithm or temporal priority.

For most rendering purposes I've been able to provide this as the only necessary accessing operation into the frame buffer. But it's not quite general enough.

Associativity

There is another intriguing generalization here. Both F and G have an opacity, but B doesn't. Does it even mean anything to composite into a pixel that already has an opacity? Yes. Consider the following scenario. Suppose we have the images F and G, but haven't yet decided what to use for a background. Let's see if we can merge F and G into one image, H, that we can store away and later overlay on B to get the same result. If we denote the compositing operation with the symbol &, what we want is

$$(B \,\&\, F) \,\&\, G = B \,\&\, (F \,\&\, G)$$

In other words we want to make compositing *associative*.

How can we define H = F & G to make this work out? We want to calculate a new pixel color H and opacity γ in terms of colors F and G and their own opacities α and β. Plug in the definitions:

0272-17-16/94/$04.00 © 1994 IEEE

$$(1-\beta)\left((1-\alpha)B+\alpha F\right)+\beta G=(1-\gamma)B+\gamma H$$

Rearrange the left side to get

$$(1-\alpha)(1-\beta)B+\left(\alpha(1-\beta)F+\beta G\right)=(1-\gamma)B+\gamma H$$

Since we want this to work for arbitrary backgrounds B, we can split this into two equations by equating the B coefficients and the non-B coefficients:

$$(1-\alpha)(1-\beta)=1-\gamma$$
$$\alpha(1-\beta)F+\beta G=\gamma H$$

The first of these gives us

$$\gamma=\alpha+\beta-\alpha\beta$$

The second equation gives us

$$H=\left(\alpha(1-\beta)F+\beta G\right)/\gamma$$

With a little fiddling this turns into

$$H=(1-\beta/\gamma)F+(\beta/\gamma)G$$

This gives us a definition for how to composite two colors, each of which has its own opacity.

Let's play with this a bit. If we composite G over a totally opaque color F, what is the result? Plug $\alpha = 1$ into the above and we get

$$\gamma=1$$
$$H=(1-\beta)F+\beta G$$

In other words, our more general compositing operation boils down to the basic one if we assume our background has its own opacity value, which happens to be 1.

Now let's try overlaying a completely opaque color G on F. Plug in $\beta = 1$ with an arbitrary α and we discover

$$\gamma=1$$
$$H=G$$

independent of α, as we expect.

Another form of association

The above definition of H is a bit complicated. Fortunately, there is a better way. One of the key insights in the Porter and Duff paper is that F shows up in the compositing formula only when multiplied by α, and G appears only when multiplied by β. Why not simply represent the pixel with the colors already premultiplied by their opacity? This representation is usually referred to as having the opacity *associated* with the color. I'll write (for the time being) an associated pixel color with a tilde over it. We have

$$\tilde{F}=\alpha F$$
$$\tilde{G}=\beta G$$
$$\tilde{H}=\gamma H$$

An associated color is just a regular color composited onto black—that is, if you displayed it directly by itself, you would get the correct anti-aliened image. (Is the joke worn out now? OK, I'll use the real word again.) Note that if the opacity equals 1, an associated color is the same as an unassociated color.

Using these definitions in the general compositing function and doing a bit of algebraic fiddling we get

$$\tilde{H}=(1-\beta)\tilde{F}+\tilde{G}$$
$$\gamma=(1-\beta)\alpha+\beta$$

This is a bit less arithmetic than our earlier definition, but what makes it particularly pretty is that we are now doing exactly the same arithmetic on the opacity components of a pixel as we are doing on the (associated) color components. This is simple, elegant, and general.

More reasons to associate

There are more reasons to store associated pixel colors than mere prettiness of the compositing formula. For one thing, some intensity calculation algorithms directly generate associated pixel colors. Additionally, we must use associated colors for any filtering or interpolation operations. Let's see why.

Antialiasing by subsampling

One typical way to do antialiasing is by subsampling. You calculate an image using point sampling at, say, four times your final resolution in x and y, and then downsample to get your final result. There are still aliases, but you have pushed them up into higher frequencies.

How does this work with our scheme here? You can consider each final pixel as broken into a 4×4 grid of subpixel cells, each containing a color and an opacity flag. Initialize these all to 0. Then, whenever your renderer writes a color to a subpixel cell, have it also set the opacity flag to 1. After rendering, sum up the 16 opacity flags within the pixel and call the result N. The net opacity for the pixel is $N/16$. Next, sum up the 16 color cells in the pixel. The average color of the pixel is this sum divided by N, the number of cells colored. But the *associated* color is even more simply calculated as the color sum divided by 16, $(\text{sum}/N) * (N/16) = \text{sum}/16$. You can then composite this associated color using the calculated opacity, $N/16$. In other words, the net associated pixel color and opacity is the sum of the subpixels divided by 16.

This works even better if your renderer is scan-line oriented—that is, it visits each pixel once in order left to right, top to bottom. You don't need individual subpixel cells. Just accumulate the color and opacity into a single pixel cell and divide by 16. In practice, I implement this with a scan-line buffer of pixel cells of length equal to the output picture. During scan-line processing, each high-resolution pixel gener-

ated simply adds its value to cell number $x/4$. Then, every four scan lines, I purge this buffer by dividing its contents by 16 and compositing it with the background using the associated compositing formula. Then I zero the buffer in preparation for the next four scan lines.

Clouds

The cloud simulation I used for Saturn's rings[2] generates a pixel's brightness as a product of the color of a cloud particle times the probability of a particle being both present in the pixel and illuminated. We can now recognize this as an associated color. The Saturn cloud simulation also generates a transparency value based on probabilities of blocking particles. The compositing operator I described for the simulation[2] is just the associated composition operator, but I didn't recognize it as such at first. Originally, I actually divided the color by the opacity before passing it to an unassociated compositing routine. Live and learn.

Filtering

Suppose we want to filter an image that has opacities at each pixel. Do we filter the unassociated colors F (this was my first thought), or do we filter the associated colors $\bar{F}$? To find out, consider the following thought experiment.

Let's downsample a scan line by a factor of two in the x direction by simply averaging successive pairs of pixels. Then let's overlay the result on an opaque background B. We want to arrange things so that downsampling and overlaying generate the same color as overlaying and downsampling. Let's follow the adventures of a typical pixel pair F (with opacity α) and G (with opacity β). Note that F and G are side by side here, not on top of each other as in our earlier examples.

First, try overlaying and then downsampling. Overlay (F, α) on B, getting $\alpha F + (1 - \alpha)B$. Overlay (G, β) on B, getting $\alpha G + (1 - \beta)B$. These two pixels are now opaque. Now downsample by averaging these results. The color will be

$$\frac{\alpha}{2}F + \frac{\beta}{2}G + \frac{2-\alpha-\beta}{2}B$$

As long as you composite first, it actually doesn't matter if you do it associated or unassociated.

Next, let's do this in the other order: downsampling first, then overlaying. Downsampling the unassociated colors and opacity, we get

$$\text{color} = (F + G)/2;\ \text{opacity} = (\alpha + \beta)/2$$

Now overlay this on B using the unassociated color compositing function to get

$$\left(1 - \frac{\alpha+\beta}{2}\right)B + \left(\frac{\alpha+\beta}{2}\right)\frac{F+G}{2} = \frac{\alpha+\beta}{4}F + \frac{\alpha+\beta}{4}G + \frac{2-\alpha-\beta}{2}B$$

—the wrong answer.

Now let's do this with associated colors. Downsample αF and βG:

$$\text{color} = (\alpha F + \beta G)/2;\ \text{opacity} = (\alpha + \beta)/2$$

Now overlay this on B using the associated compositing function to get

$$\left(1 - \frac{\alpha+\beta}{2}\right)B + \frac{\alpha F + \beta G}{2} = \frac{\alpha}{2}F + \frac{\beta}{2}G + \frac{2-\alpha-\beta}{2}B$$

—the right answer.

To reiterate, downsampling and, in fact, *all* filtering operations should operate on arrays of associated pixel colors as well as, of course, on the array of opacity values.

Interpolation

Here's another example. Suppose we are doing Gouraud interpolation across a polygon. Each vertex has a color, and we do the standard interpolation of vertex colors to get the colors inside the polygon. Now, what if the vertices have opacities as well? We simply interpolate them in a similar manner. But should we interpolate unassociated colors or associated colors? (I'll bet you can guess.)

Actually, this might seem a little open to interpretation. After all, Gouraud interpolation is itself an approximation of a more accurate curved-surface-shading function. Who's to say what the correct interpolation amount is? Well, consider the following: Interpolation is another form of filtering. Suppose we wanted to expand an image two times by interpolating between each pixel pair. We would again like this to look the same if we interpolated and then overlayed on a background or if we overlayed first and interpolated second.

Going back to polygons, we might have a scan line with the colors (F, α) on one end and (G, β) on the other. We want the inside colors to look the same when overlayed onto a background. We want to interpolate and then overlay over B, and we want to make this the same as overlaying and then interpolating.

You can do the algebra yourself. Does it look familiar? It's just the same as the filtering example, leading us to the conclusion that Gouraud interpolation should also be done on associated pixel colors.

Computer notation

Each pixel has a red, green, and blue color and an opacity α. Since we like associated colors so much, we will represent a pixel by the quadruple:

$$(\alpha F_{red}, \alpha F_{green}, \alpha F_{blue}, \alpha)$$

This looks suspiciously like homogeneous coordinates. I've tried real hard, but for the life of me I can't figure out any use for this observation.

I'm going to start talking about computation, so let's take a moment to switch from the vector algebra mathematical notation above to a more C-like computer data structure representation. We can represent the pixel color as a four element vector `F[i]`, with `F[0]` holding the opacity and `F[1]...F[3]` holding the colors. Or we could define a pixel data structure containing the fields `(r,g,b,a)`. We will assume the colors to be associated so that `F.r` = αF_{red} and so on.

The opacity value is more conventionally called alpha, which is why I've called that field `a`. I'll call it alpha from now on since we no longer have to worry about having a separate Greek letter for the opacity value of different pixels.

Marching in place

I want to reformulate this a bit to return to our original operation of overlaying a foreground on a background stored in a frame buffer. Here, $\tilde{H}$ and $\tilde{F}$ from the compositing formula are the same storage locations: the background `B`. $\tilde{G}$ is the new foreground image, which I'll call `F`. The in-place formulations of the operation become

```
for(i=0; i<4; i++)
  B[i] = B[i]*(1-F[0]) + F[i];
```

We could go really nuts and define a set of overloaded operators for four-dimensional array arithmetic. Compositing could then look something like

```
B = B*(1-F[0]) + F;
```

I'm not sure I would recommend this after seeing the sort of code most C++ compilers come up with for such things. Instead I'll splurge on source code and write explicitly

```
transparency = 1-F.a;
B.r = B.r*transparency + F.r;
B.g = B.g*transparency + F.g;
B.b = B.b*transparency + F.b;
B.a = B.a*transparency + F.a;
```

Again, for most rendering applications, this new formulation is the only access method you need to the frame buffer. I've resisted the temptation (barely) to overload a pixel class operator to do this.

Examples

To motivate our next bit of gimcrackery, let's look at a few examples of the types of things we might do with this.

Example 1

- Load an opaque background into the frame buffer, one with `B.a==1` everywhere.
- Run a rendering program that overlays its results into the frame buffer. Pixels inside the object (which is most of them) all have `F.a==1`. Only the ones on the edges have `F.a!=1`.
- Run another rendering program that similarly overlays its results into the frame buffer. Part of the new object overlaps the first one, and part of it sticks out onto the background. Note that a rendering program doesn't need to know what's under it. The overlay operation does the correct thing.
- Run another rendering program that places algorithmically defined shapes. These might be lines, spots, smears, and so forth, with a constant color across them but whose shapes are "sculpted" by the alpha channel.
- Save the resultant image to a file.

Example 2

- Load the frame buffer with the transparent color (0,0,0,0).
- Render all sorts of stuff as in the previous example.
- Save the resultant image to a file. This rectangular image will have `B.a==0` for pixels that never got "hit" by any renderers (which might be a substantial portion of the image).

Example 3

- Load an opaque background into the frame buffer.
- Overlay the image saved from Example 2 (there is was the partially transparent background image `B`; it now becomes the foreground image `F`). Only those pixels with `F.a!=0` will affect the background.

Example 4

I'll just mention here an entirely different way to do this that I don't happen to use. It involves the classic engineering trick of reversing the order of loops. The above examples performed the operations in the order

```
for (each overlayed image)
  for (each pixel in image)
    composite with background
```

Some systems reverse this order and do

```
for (each pixel in output)
  for (each overlayed image)
    composite all together
```

This latter allows some speedup by optimizing the (possibly complicated) algebraic expression generated by all the pixel arithmetic for one pixel.

Short cuts

Examples 1 through 3 show that a lot of compositing arithmetic can wind up being done with alpha values of 0 and 1. This motivates us to look for a few shortcuts in our implementation. Since the arithmetic is identical for the four fields of a pixel, I'll write calculations just once using the symbol `F.i` (where `i` stands for `r`, `g`, `b`, or `a`). Table 1 shows possible special cases.

You might think that the case of `F.a==0` necessarily implies that all the color fields `F.i` must be 0. After all, we are

Table 1. New value of B.i composited with associated pixel F.i, where B.i = B.i*(1-F.a)+F.i.	
F.a==0	B.i = B.i+F.i
F.a==1	B.i = F.i
F.a==anything else	B.i = B.i*(1-F.a)+F.i

Table 2. New value of B.a where B.a = B.a*(1-F.a)+F.a.	
B.a==0	B.a = F.a
B.a==1	Unchanged
B.a==anything else	B.a = B.a*(1-F.a)+F.a

Table 3. Porter and Duff operators in C notation.	
F over B	B.i*(1-F.a) + F.i
F in B	F.i*B.a Color of B unused
F out B	F.i*(1-B.a) Color of B unused
F atop B	F.i*B.a + B.i*(1-F.a)
F xor B	F.i*(1-B.a) + B.i*(1-F.a)
F plus B	F.i + B.i

dealing with associated colors here. The first case would then just leave B.i unchanged. It turns out, however, that there are some reasons not to always make this assumption. I'll go more deeply into this in a later column. Anyway, here's the optimized algorithm:

```
if      (F.a==0) B.i += F.i;
else if (F.a==1) B.i  = F.i;
else             B.i  = B.i*(1-F.a)+F.i;
```

We can do a little better with the a field in the latter case. If B.a==1, we can skip the calculation of a new B.a since it will stay 1 no matter what F.a is. Table 2 shows the possible values of B.a and what the calculations look like with redundant arithmetic and stores removed.

To take advantage of this, let's first note that the most likely case is B.a=1. We therefore make that the fastest test. Now turn this into an algorithm:

```
if (B.a!=1){
  if(B.a==0) B.a = F.a;
  else       B.a = B.a*(1-F.a)+F.a; }
```

There is, of course, a trade-off here between the number of tests you make versus the time to just go ahead and do the general arithmetic. On some pipelined machines, it's probably better to avoid conditional jump instructions, do the arithmetic always, and keep the pipe full. Still, I've found the above to be worthwhile in my situation. In general, I think it's nicest to provide one general-purpose, easy-to-remember function, then test for special cases inside the function. The special-case testing is usually negligible compared to the arithmetic you save by detecting the special case.

Other operators

Just for completeness, Table 3 lists how Porter and Duff's other operators look in my notation and what the algebra boils down to. Of these, the "over" operator is by far the most often used. This leads us to wonder the following: How about storing the value (1-F.a) instead of F.a in a pixel? This value would be the transparency rather than the opacity of the pixel. Let's call this field t. Transparencies combine by simple multiplication. Why? Remember, in our original derivation of γ, we had

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

and 1 minus opacity is transparency. The associated-color compositing operation would then be

```
B.i = B.i*F.t + F.i; // for i=r,g,b
B.t = B.t*F.t;
```

This formulation is, of course, even less arithmetic than before. The only trouble is that most industry standard file formats and compositing programs deal with alpha values as meaning opacity. You would have to translate your image by storing (1-B.t) when you output your image to a file. Since the transparency-based formulation only saves one add and one subtract per pixel, it might not be worth it.

A teaser

In a future column I'll talk about how number representations for pixels and round-off error affect compositing calculations. And I'll froth about the phenomenal stupidity of compilers and why parts of this calculation should still be done in assembly language. ❑

References

1. T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics* (Proc. Siggraph), Vol. 18, No. 3, July 1984, pp. 253-259.
2. J. Blinn, "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces," *Computer Graphics* (Proc. Siggraph), Vol. 16, No. 3, July 1982, pp. 21-29.