# Jim Blinn's Corner

## Composting, Part 2: Practice

**James F. Blinn**

***California Institute of Technology***

At this year's Siggraph, Chris Wedge told the story of an audience member who once asked him about "image composting." I like this comment on the value of our industry even better than the "anti-aliening" I used in Part 1 of this discussion.

Well, anyway. Enough of this levity. This time I want to discuss the practice of image composting and in particular the Porter-Duff "over" operator. Last time we derived the operator for compositing a foreground pixel F "over" a background pixel B as

$$B(1-F\alpha)+F$$

where each pixel is a vector with four components: red, green, blue, and alpha (the coverage or opacity amount), and standard vector algebra applies.

I've found it most useful to provide "over" as an in-place operator; you have an image stored in a frame buffer and want to lay another image on top of it. In other words, the result of "F over B" replaces B.

### C notation

In translating this vector notation to C, I'll write the components of a pixel using class member notation: (`F.r, F.g, F.b, F.a`). Code to implement the compositing operation is then

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

For most rendering operations this is the only access method to the frame buffer that you will need.

In what follows I'm going to complexify this a bit, so let me make life a bit easier and define an abbreviation. Since I will always treat the red, green, and blue components in the same way, I'll just give a single statement for color components, using the generic class member `.c`. Any such statement actually expands into three statements. I will sometimes do different arithmetic for the `.a` member, so I'll keep it separate. Using this abbreviation scheme, the above code looks like

```
t = 1 - F.a;
B.c = B.c*t + F.c;
B.a = B.a*t + F.a;
```

### Short cuts

In "Compositing, Part 1," I talked about some shortcuts to avoid unnecessary arithmetic whenever `F.a` and/or `B.a` are zero or one. The result is the code

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

Now this is, of course, more complicated than simply doing the arithmetic straightforwardly as in the first code fragment. It might seem barely worthwhile, but in Part 1 I showed that the various cases tested are actually pretty common. They occur, for example, when you overlay or render an opaque object into the frame buffer. When this happens, you have `F.a==1` almost everywhere in the object except at the antialiased edges. With the special-case testing, you can simply store those pixels with no arithmetic at all. In this column, I describe even better reasons why this economization is worthwhile.

## Pixel representations

So far I haven't said anything about how to represent the numerical quantities in a pixel. Well, here's where we get real. I will represent the red, green, blue, and alpha components of a pixel in three ways.

### Floating point

This is the easiest method and the one that most complex rendering algorithms generate. You don't have to worry about overflow or underflow except in extreme cases. Arithmetic with floating-point numbers is slow—but at least they take up lots of memory!

### 16-bit scaled integers (`short int`)

Here, we scale the floating-point value by the quantity 16,384 and round to the nearest integer. Thus the 16-bit value 0 represents the floating-point value 0.0, and the 16-bit value 16,384 represents the floating-point value 1.0. (I described this system in my column,

"Dirty Pixels," *IEEE CG&A*, July 1989). Note that this choice of scale factor does not use all the possible precision of a 16-bit number; only about a quarter of the possible 16-bit values cover the range from 0.0 to 1.0. I chose this range so that the scale factor is a power of 2 (good for divisions, see below) and so that there is an explicit representation of the number 1.0. Using this scheme, the possible 16-bit integer values actually represent the floating-point quantities from −2.0 to +1.9999. Giving the pixels this "head room" avoids overflow and underflow problems when doing filtering with negative lobed filters (see "Return of the Jaggy," *IEEE CG&A*, March 1989).

### *8-bit bytes (`unsigned char`)*

These are the values that we actually store in the frame buffer and that the refresh hardware uses to display the image. Typical display hardware does not translate these values directly into intensity, however. Instead, the displayed brightness, D, is a power of the byte value, I:

$$D = (I / 255.)^{\gamma}$$

with $\gamma$ typically being around 2 or 2.2. For this reason, you must calculate the inverse of this power function when you convert a desired intensity to a byte value. (I also described this process in "Dirty Pixels.")

In addition to being needed for hardware display, 8-bit bytes are what you typically store if you save an image in a file (although not all image manipulation programs expect the pixels to be gamma corrected).

The alpha component of the pixel, however, doesn't need to be gamma corrected. For this component I scale the desired alpha by 255 and round to the nearest integer. This uses the whole range of byte values. The byte value 0 represents 0.0 and the byte value 255 represents 1.0.

## Conversions

One of the reasons to look for economizations in the compositing calculations is that conversion between these forms is not free (timewise). If we can skip a conversion because we detect that its result is going to be multiplied by zero, we win even more than if we only avoid the multiplication.

Let me just quickly list the conversions necessary. Following the conventions in "Dirty Pixels," I will use the variable `D` to represent a floating point value, `J` to represent a 16-bit integer, and `I` to represent an 8-bit byte value.

### *float to 16-bit*

```
J = int(D*16384. +.5);
```

### *16-bit to float*

```
D = J/16384.;
```

### *float to 8-bit*

The arithmetic is different for the color components and the alpha component because the color component is gamma corrected.

```
I.c  int(pow(D.c,1./gamma)*255.+.5);
I.a = int(    D.a          *255.+.5);
```

Strictly speaking, the type coercion should be `(unsigned char)`, but `(int)` works and fits the statement onto one line.

### *8-bit to float*

```
D.c = pow(I.c/255.,gamma);
D.a = I.a/255.;
```

### *8-bit to 16-bit*

We define the result of this conversion by converting 8-bit to float and then float to 16-bit. In practice this is too slow, since you must calculate a power. But, since there are only 256 possible 8-bit patterns, you can precalculate all the possible values and do conversion by table lookup. I will define a routine for such a conversion:

```
J.c = cv16c(I.c);
```

The alpha value, on the other hand, is not gamma corrected, so we can convert it by direct scaling:

```
J.a = (I.a*16384 + 127)/255;
```

Again, it is usually faster to put this into a table. I will define a routine for this:

```
J.a = cv16a(I.a);
```

### *16-bit to 8-bit*

Again, we define the desired result by converting 16-bit to float and then float to 8-bit. In "Dirty Pixels" I described how to do this relatively quickly by using a simplified binary search through a table. I'll hide this in the function:

```
I.c = cv8c(J.c);
```

Since the alpha component is encoded linearly here too, we can convert directly:

```
I.a = (J.a*255 + 8192)/16384;
```

and define the function to do this:

```
I.a = cv8a(J.a);
```

### *Unassociated float to associated 16-bit*

On some occasions, we will be given an unassociated floating point pixel to process (one whose color components haven't been multiplied by alpha). As a happy bonus, we can combine the association operation (multiplying color by alpha) with the scaling to 16-bit integers. We first get the 16-bit value for alpha, then use that to scale the color components:

```
J.a = int(16384.*D.a + .5);
J.c = int(J.a*D.c + .5);
```

But this does have a potential round-off problem since we are multiplying D.c by an approximate version of D.a instead of the exact amount. For example, starting with (D.c,D.a)=(.5,.2) we have J.a=3277. Converting J.c the hard but correct way gives us

```
J.c=int(.5*.2*16384.+.5)=int(1638.9)
   =1638
```

Converting J.c our quicker way gives us

```
J.c = int(3277*.5 + .5) = 1639
```

This might be close enough if you are really worried about speed, but maybe the bonus isn't so happy after all. This just shows us that we have to be careful about compounding the results of rounding operations (cue the foreshadowing music).

## Pixel arithmetic

In a nutshell, I do all pixel arithmetic in 16-bit form; floating-point arithmetic is too slow and 8-bit representations are not linear. Some rendering programs generate results in floating point, but I convert the numbers to 16-bit as soon as possible. 16-bit arithmetic is plenty accurate for image compositing and filtering. If you need more accuracy, say for Gouraud interpolation across a polygon, you can easily tack 16 bits of zeros on the right to convert it to a 32-bit fraction.

Now let's review how to do arithmetic with 16-bit scaled integers. Happily, addition and subtraction are the same as normal addition. Only multiplication is tricky. Since each factor has a built-in scale factor, the result winds up scaled twice. We must divide out the extra scale factor and round the result to get

```
(J1*J2 +8192)/16384
```

Here's where our choice of scale factor pays off. The product of the two 16-bit numbers is a 32-bit number, and you can do the division by simply shifting this 32-bit result right by 14 bits.

*At least you could if compilers were built correctly.*

Unfortunately, I have yet to see a compiler that realizes that the product of two 16-bit numbers is a 32-bit number. The hardware knows it. We know it. But the four different compilers I looked at take this perfectly natural function and make it into something dirty: They do the multiply and throw away the high-order 16 bits. To circumvent this you have to convert the two 16-bit numbers to two 32-bit numbers and do a 32-bit multiply (often calling a subroutine), giving a 64-bit result. Then, since the compiler throws away the top 32 bits of that resultant product, you're in business.

I mean, I'm not one to spend inordinate amounts of time shaving bits, but it *hurts* me to see this. I have been forced to code the scaled 16-bit multiplication operation in inline assembly language. I've currently put it into a subroutine, but apparently I need to put it in a C macro to get real inline code. (Generally speaking, C macros have been obsoleted by the C++ concept of inline functions. However, in the real world, it seems that inline functions cannot contain assembly language instructions.) The routine does a simple 16-bit multiply, adds the rounding bias, and shifts the result down. To emphasize what's going on, I'll explicitly call the routine M instead of overloading the multiplication operator in what follows.

```
J = M(J1,J2);
```

One other thing to note about M: It also works properly if one of the arguments is an 8-bit linearly scaled value (padded on the left with zeros), producing an 8-bit result. Why? The first parameter is really a fraction whose "actual" value is J1/16384. M then effectively multiplies the other parameter (however it may be scaled) by this fraction.

## What we want

My "over" operation supports two services: overlaying 16-bit pixels into the frame buffer, and overlaying 8-bit pixels into the frame buffer. The frame buffer itself is always 8-bit pixels. To provide some orthogonality of functionality, the "over" routines operate on arrays of pixel values (typically a scan line's worth). Reading and writing to the actual frame buffer is done elsewhere. I won't explicitly show the array loop below, but it's there. It's always best to make your subroutines process data in big globs to minimize the overhead of subroutine calls.

## Things that cost time

In the code at the beginning of this article many seemingly simple operations contained, hidden within them, a lot of conversions between types. Let's review them, roughly in order of nastiness.

| | |
|---|---|
| cv8c(J) | binary search in table |
| cv8a(J) | multiply, 14 bit shift |
| cv16c(I) | table lookup |
| cv16a(I) | table lookup |

In the code below I'll explicitly call these conversion routines to make it easier to see where time goes. Assignments happen between like types, so no hidden conversions are performed. Also, I'll use the variable names BJ and FJ to hold the 16-bit form of the pixels, and BI and FI to hold the 8-bit form. Reference to simply B or F means the floating-point version.

## 16-bit over 8-bit

This is the workhorse routine. Generally, the results of any rendering or image-processing calculation are in 16-bit form. Here is the raw code (with no special case testing) to composite the pixel FJ over the pixel BI:

```
BJ.c = cv16c(BI.c);
BJ.c = M(BJ.c,16384-FJ.a) + FJ.c;
BI.c = cv8c(BJ.c);

BJ.a = cv16a(BI.a);
BJ.a = M(BJ.a,16384-FJ.a) + FJ.a;
BI.a = cv8a(BJ.a);
```

### Alpha calculation

Let's first see how we can improve on the calculation of B.a. The formula is

```
B.a = B.a(1-F.a) + F.a
```

The two main special cases

```
if   (F.a==0) {/* B.a unchanged*/ }
else if (F.a==1) B.a=1;
```

are taken care of in the general shortcut scheme at the beginning of this article. For more general values of F.a we actually have to do some work.

We can avoid a conversion by recalling that our M routine can multiply a 16-bit fractional number by an 8-bit number.

```
BI.a =M(BI.a,16384-FJ.a) + cv8a(FJ.a);
```

The only problem with this is that it gets the wrong answer much of the time; it's sometimes off by one. The correct answer is defined by converting both BI.a and FJ.a to float, doing the calculation, then converting that floating result to 8-bit. In fact, the brute-force calculation that we are trying to improve on often gets the wrong answer too. The basic reason is that both schemes contain the sum of two or more rounded calculations (either explicitly or hidden inside the M routine). If each term of the sum was, say, 2.4, then rounding first and summing would give 4. Summing and then rounding would give 5.

To get the right answer, and incidentally make the code faster, we can rewrite the mathematical definition of B.a as

```
B.a = B.a + F.a*(1-B.a)
```

We turn this into code again by using the fact that the M routine can multiply the 8-bit version of (1-B.a) by the fractional value represented by F.a:

```
BI.a += M(FJ.a,255-BI.a);
```

This formulation only rounds once, and I've verified that it always gets the "right" answer as defined above. Faster and more correct—who could ask for anything more?

Now, to get the fastest calculation of the new BI.a we include some tests for trivial case values of B.a. The most likely case is B.a=1; the next most likely case is B.a=0.

```
if(BI.a != 255) {
     if(BI.a ==0)
       BI.a = M(FJ.a,255);
     else
       BI.a+= M(FJ.a,255-BI.a);}
```

This is how we will calculate BI.a in both the 16-bit and 8-bit routines below.

### Superluminous pixels

There is another issue to discuss that relates to associated pixels with alpha=0. In an associated pixel, the color components have been multiplied by alpha. Therefore, if alpha=0, all the color components should be 0. When compositing such a pixel over a background, you should be able to skip pixels with F.a==0. The very first special case test can then read

```
if (F.a==0) {}
```

There are two situations where this can cause problems. The first is the possible use of special pixel values for unusual lighting effects. For example an associated pixel with values (.2, .2, .2, 0) has effectively some light but no "coverage." Applying it to a background pixel just adds something to the color components without changing the background pixel's coverage. This can be useful, but I haven't had occasion to use it. I will therefore, for illustration purposes in this code, not expect superluminous pixels. For the nonce, I'm going to assume that if F.a==0, then all other elements of F are zero.

The second problem will only hit us when we overlay 8-bit pixels into the frame buffer, so I'll talk about it later.

### Final 16-bit code

Note that in the following code I've also tossed in a test to skip some arithmetic if the color of B is black (another common occurrence):

```
if(FJ.a==0)
   {} // leave B unchanged
 else if (FJ.a==16384){
   BI.c = cv8c(FJ.c);
   BI.a = 255;}
 else {
   if (BI.c==0)
     BJ.c = FJ.c;
   else {
     BJ.c = cv16c(BI.c);
     BJ.c = M(BJ.c,16384-FJ.a)+FJ.c;}
   BI.c = cv8c(BJ.c);
   BI.a (calculated as above)}
```

## 8-bit over 8-bit

We need this second routine when compositing a stored image over the 8-bit frame buffer. It calculates FI over BI. The naive approach is to convert all 8-bit quantities to 16-bit, do the arithmetic, then convert 16-bit to 8-bit. Here it is, with no special case testing, but using the improved calculation of BI.a:

```
FJ.c = cv16c(FI.c);
FJ.a = cv16a(FI.a);
BJ.c = cv16c(BI.c);
BJ.c = M(16384-FJ.a,BJ.c) + FJ.c;
BI.c = cv8c(BJ.c);
BI.a+= M(FJ.a,255-BI.a);
```

This cries out for special-case testing.

### Alpha calculation

Since we have the two alphas encoded linearly into 8 bits, we could do the arithmetic directly. Since they are scaled by 255, the formula

```
B.a = B.a*(1-F.a) + F.a;
```

translates into

```
BI.a=(BI.a*(255-FI.a)+127)/255 +FI.a;
```

We have to divide by 255. Ick. We are actually better off converting the `FI.a` to 16-bit form (it's only a table lookup), then using the same code as in the 16-bit algorithm.

### Superluminous pixels and round off error

Here's where we have to face the "other" problem that occurs with `F.a==0`. It comes from the fact that, for 8-bit pixels, color components are gamma corrected but the alpha components are not. Thus if you convert the very dim, very transparent 16-bit pixel (1, 1, 1, 16) to 8-bit values you get (2, 2, 2, 0). The alpha component has rounded to zero even though the color components haven't.

In this situation we cannot simply ignore pixels with `F.a==0`. Consider the following scenario:

1. Clear the frame buffer to black:
   `BI=(0,0,0,255).`
2. Overlay `FJ=(1,1,1,16)` giving
   `BJ=(1,1,1,16384)`
3. Convert to 8 bits: `BI=(2,2,2,255).`

We want this result to match the scenario

1. Clear the frame buffer to transparent:
   `BI=(0,0,0,0).`
2. Overlay `FJ=(1,1,1,16)` giving`J=(1,1,1,16).`
3. Convert to 8 bits: `BI=(2,2,2,0).`
4. Store this in a file.
5. Clear the frame buffer to black: `BI=(0,0,0,255)`
6. Overlay the file pixel: `FI=(2,2,2,0)`

If we then use the code

```
if (F.a==0) B.c += F.c
```

(with appropriate conversions) we will get the desired result:

```
BI = (2,2,2,255)
```

### Final 8-bit code

We won't give up completely on skipping transparent foreground pixels. The first test in the code below is a test of all 32 bits of `FI` being zero. This handles overlaying an image with substantial transparent regions. You can often do this test in one instruction by leaning on the compiler a little.

In fact, I actually scan the `F` array to see if any pixels at all are nontransparent. This allows me to skip whole scan lines of transparent foreground image without even reading the frame buffer. I won't show this part explicitly.

Anyway, the code:

```
if(FI==0) skip this mess;
if (FI.a==0){
   FJ.c = cv16c(FI.c);
   BJ.c = cv16c(BI.c);
   BI.c = cv8c(FJ.c+BJ.c);}
 else if(FI.a==255){
   BI.c = FI.c;
   BI.a = 255;}
 else{
   FJ.c = cv16c(FI.c);
   FJ.a = cv16a(FI.a);
   BJ.c = cv16c(BI.c);
   BJ.c = FJ.c + M(16384-FJ.a,BJ.c);
   BI.c = cv8(BJ.c);
   BI.a (calculated as above)
```

Note that if a pixel of the foreground image is opaque (that is, if `F.a==1`) no conversion or arithmetic of any kind is necessary.

## Another 8-bit possibility

Suppose we had encoded the rgb values linearly into 8-bit bytes in the same way that we did alpha. We could write the overlaying code to operate directly on 8-bit numbers and wouldn't need to convert up and down to the 16-bit representation. Wouldn't that be a lot faster? Well, not nearly as much as you might think. When scaling to 8 bits we must use the factor 255. We don't have the bits to waste to make the factor a power of 2 as we did with 16-bit scaling. Therefore, we would have to divide by 255 after multiplying two scaled-by-255 byte values together. The overhead for this starts getting comparable to what we currently incur with gamma-corrected bytes.

## Summary

Wow. Life sure can get complicated. I still have a nagging feeling that all this special-case testing might not be worth it. But no, all the cases I added to the code were motivated by actual profiling runs showing that time was being wasted in these routines. And consider the situations that fall into the special cases: opaque regions of foreground, transparent regions of foreground, opaque backgrounds, and black backgrounds. These are all pretty common situations.

This is all part of the craft of programming—trading off special-case speed with complex code against general-case slowness with simple code. Actually, in this situation, we have the best of both worlds. There is just one general-purpose routine for each data type to learn. The special cases are automatically detected internally, so you don't have to worry about a speed penalty. We get the generality of being able to overlay translucent objects onto transparent backgrounds with virtually no overhead if you don't happen to be using that generality.

What could be finer? ■