Compositing and Matting

Robert Bridson

December 2, 2011

As mentioned at the beginning, the final frames in a film are usually assembled from many different images. Obviously if computer generated effects are to be blended with real-world plates, there has to be at least two images being combined — but even in a purely computer generated shot the rendering will be split into several component images which are only combined at the very end. This is useful from the standpoint of balancing the load — different images can be rendered on different machines, and even managed by a different team of artists — but is even more important from the standpoint of control and artistic direction. If one element of a shot doesn't look quite right, it can be re-rendered hopefully without the expense of re-rendering all the rest of the scene. Final decisions about how bright or dark or blurred or colour balanced each element of the decision can also be deferred to relatively cheap image-processing operations as late as possible in production.

This chapter first deals with the basic operations used to combine, or *composite*, images together. The second part tackles a related and much trickier problem of bringing parts of real-world images into the compositing, separating the foreground from the background.

1 Compositing Operations

1.1 One Image Over Another

Let's begin with the most common and simplest compositing operation, placing one image over another, in particular thinking of the first image as supplying a foreground which should supplant the background given by the second image. A particularly good model to have in mind is of images painted on clear sheets of plastic, and the composited result coming from stacking the sheets on top of each other — in fact, this is exactly how a lot of traditional animation with multiple layers is done.

At the most basic level, this operation requires us to make a decision at every point in the image whether the final colour should come from the foreground image or from the background image. We encode this decision for every pixel in an extra channel (besides the usual red, green, and blue) of the foreground image, usually called *alpha*. When $\alpha = 0$, the foreground image is perfectly transparent and the background can show through; when $\alpha = 1$ the foreground image is opaque and overwrites the background.

If the alpha channel only has a single bit, on or off, it can't handle the cases where we would want the final pixel to be a blend of both the foreground and background colours. This occurs when the foreground image is light fog or smoke, for example, and should allow some fraction of the light through from the background. Much more importantly, at the silhouette edges of objects in the foreground, some degree of blending is required to avoid ugly aliasing — as we have dealt with in primary rendering via supersampling and averaging.

We therefore allow the alpha channel to take fractional values between zero and one, which raises the question of how to do the blending in this case. Thinking of alpha as encoding the "opacity" of the foreground pixel, it's reasonable to say the foreground allows $(1-\alpha)$ of the background colour through and contributes α of its own colour:

$$\begin{aligned} R_{\text{final}} &= \alpha R_{\text{fore}} + (1-\alpha) R_{\text{back}} \\ G_{\text{final}} &= \alpha G_{\text{fore}} + (1-\alpha) G_{\text{back}} \\ B_{\text{final}} &= \alpha B_{\text{fore}} + (1-\alpha) B_{\text{back}}. \end{aligned} \tag{1}$$

Here the subscripts *final*, textitfore and *back* refer to colour channels (for a single pixel) in the final, foreground, and background images respectively. As an abbreviation we will sometimes combine the colour channels into a single RGB vector $\vec{C}$:

$$\vec{C}_{\text{final}} = \alpha \vec{C}_{\text{fore}} + (1-\alpha) \vec{C}_{\text{back}}. \tag{2}$$

Another way to think about this, especially in the context of antialiasing edges, is that the alpha channel represents the fraction of the pixel area which the foreground occupies. With α of the final pixel's area being the foreground colour and the remaining $(1-\alpha)$ of the final pixel's area being the background colour, the final pixel's average colour works out to the expression in equation (2).

1.2 Premultiplied Alpha

In the case where $\alpha = 0$, the foreground image essentially doesn't exist: it contributes nothing to the final colour. It's a little odd in this case to still have arbitrary RGB values. Additionally we will almost always only look at a foreground image via a compositing operation with equation (2), where the RGB values of the foreground are multipled by the alpha value. For these reasons, and a matter of mathematical convenience later, it is very common to actually store the RGB values already scaled by alpha: this is called *premultiplied alpha*.

With a premultiplied alpha image, the stored RGB values for a pixel with $\alpha = 0$ must also be zero, and the compositing equation simplifies a little:

$$\vec{C}_{\text{final}} = \vec{C}_{\text{fore}} + (1-\alpha)\vec{C}_{\text{back}}. \tag{3}$$

This makes it clear that the premultiplied RGB represents the actual contribution the image makes to the final pixel colour, not just the colour of its fraction of the pixel.

It almost goes without saying that it's critically important to be aware whether an image has premultiplied alpha or not — if strange overly bright or dark pixels appear around the edges of an object in the final composited image, most likely confusion over premultiplied alpha is to blame.

1.3 Compositing Many Images Over One Another

What happens if we have a whole stack of images to composite together? Obviously we need an alpha channel for each of them, apart from the furthest background. Armed with the above, we could start with the "furthest" background and composite the next furthest over it to get a combined background for the next, and so on. However, there may be cases where we at first don't know the furthest background and wish to do this in another order. There we hit a snag, because we don't yet have a formula to produce a new alpha channel when compositing two images.

We'll work this out with premultiplied alpha. One way to work out what the result of compositing three images together would be, starting from the background, and then demand *associativity*: the result should be the same if the nearest two images are composited first, and then the result composited over the background. Symbolically we can write associativity as

$$I_1 \text{ over } (I_2 \text{ over } I_3) = (I_1 \text{ over } I_2) \text{ over } I_3, \tag{4}$$

for images I_1, I_2, and I_3. Using equation (3), the left hand side at any pixel gives:

$$\vec{C}_{\text{final}} = \vec{C}_1 + (1-\alpha_1)\big[\vec{C}_2 + (1-\alpha_2)\vec{C}_3\big]. \tag{5}$$

Using α_{12} to denote the unknown alpha channel from I_1 over I_2, the right hand side gives:

$$\vec{C}_{\text{final}} = \big[\vec{C}_1 + (1-\alpha_1)\vec{C}_2\big] + (1-\alpha_{12})\vec{C}_3. \tag{6}$$

Setting these two formulas equal gives us:

$$\begin{aligned} (1-\alpha_1)(1-\alpha_2) &= (1-\alpha_{12}) \\ \Rightarrow \quad \alpha_{12} &= \alpha_1 + (1-\alpha_1)\alpha_2. \end{aligned} \tag{7}$$

This has the delicious property of being exactly the same form as equation (3) only for alpha instead of colour.

An alternative derivation, which provides greater insight into the underlying assumptions, is to take a half-geometric, half-probabilistic viewpoint on alpha. Again think of alpha as indicating the fraction of the pixel's area covered by the image — or equivalently, the probability that a random point chosen uniformly from the pixel will be from the image. When we stack an image over another, both with $\alpha < 1$, the final result isn't actually well-posed. For example, if both alpha values were one half, the top image could completely obscure the bottom image (because they occupy exactly the same half of the pixel) or the bottom image could show up in full (because it occupies exactly the other half of the pixel), or anything in between. To make it a well-posed problem, we need an additional assumption on how the two images relate to each other.

The most natural assumption, from a probabilistic perspective, is that the two images are *independent*: if you pick a point at random inside a pixel, the chance it hits the second image is identical whether or not you know it hits (or doesn't) the first image. Geometrically this could be modeled as the first image dividing the pixel along the x axis at fraction α_1 and the second image dividing the pixel along the y axis at fraction α_2 — but this mental picture doesn't extend beyond two images!

From the assumption of independent probabilities, we can work out α_{12}.A random point chosen uniformly from the pixel has α_1 chance of hitting the first image, and $(1 - \alpha_1)$ chance of not hitting it. If it doesn't hit, it still has an α_2 chance of hitting the second image and $(1 - \alpha_2)$ chance of not hitting it. The chance of hitting either the first or the second is therefore $\alpha_1 + (1 - \alpha_1)\alpha_2$ as before.

1.4 Rendering Alpha

We have so far assumed the existence of the alpha channel without discussing where it comes from. Estimating it directly from a raw RGB image, e.g. real footage, is difficult — see the next section "Matting". However, for synthetic images generated by a renderer, creating an alpha channel is almost trivial.

For rendering opaque surfaces, every sample the renderer sets a colour for should get $\alpha = 1$, and the other "empty" samples should be left at $\alpha = 0$. If there are multiple samples per pixel for anti-aliasing, with a weighted average over samples producing the final RGB colour for a pixel, the same weighted average between the alpha values of the samples should be used for the pixel's alpha. This very naturally fits the probabilistic or geometric view of alpha: we're estimating the fraction of samples that would hit an object versus nothing.

Rendering partially transparent objects is trickier. In a raytracer, alpha can be accumulated along the rays along with colours; in a Reyes-type algorithm (that uses a so-called "A-buffer") a compositing operation must be performed at every sample to combine semi-transparent surfaces. OpenGL doesn't easily support rendering partially transparent objects in the first place (without requiring triangles sorted by depth or similar restrictions).

1.5 Differential Rendering

With matchmove and the *over* compositing operation, and environment mapping or a simpler estimation of the lighting in a real scene, it's possible to convincingly place rendered objects in real footage — with one exception. A real object in a photograph usually affects more than just the pixels where it is directly visible: the light it reflects and, more importantly, the shadows it casts are visible in other parts of the image. A synthetic object has to approximate these effects too. It's not immediately clear how to arrange for a synthetic object to cast a shadow on a real object in an image. Rendering algorithms for computing shadows only work for the synthetic objects being rendered.

In a landmark paper [Deb98], Debevec solved this problem by introducing *differential rendering* (and in the same paper also introduced high dynamic range environment maps for accurate lighting, i.e. using images that properly encode actual brightness rather than being clipped to the narrow 8-bit colour channel range of typical computer images, and many other big steps forward). The core insight is that the difference between a rendered image with shadows and without shadows encodes just the effect of the shadows themselves — and that can be composited into real photos.

Differential rendering requires a "proxy" or rough approximation of the real objects in the scene, or at least the surfaces where shadows from synthetic objects will fall. The proxy need not be extremely accurate — it won't be seen in the final image — but it has to match the geometry of the real scene closely enough to allow the synthetic cast shadow to look realistic. There are computer vision methods for automatically reconstructing 3D geometry from photographs, but often a quick approximation created manually by an artist in modeling software will suffice; in either case establishing the world space location of the proxy can be made part of the matchmove process. The surface colours of the proxy can be even more approximate — perhaps arrived at by back-projecting the real photo onto the proxy geometry, or even just using a uniform average colour from the real photo. Again, the proxy won't appear in the final image, but the more accurate it is the more accurately we can compute the effect of the synthetic shadows on the real scene.

The rendering proceeds in three passes as follows:

1 Render the synthetic object on its own, as you would normally, producing image I_1.

2 Render the proxy on its own (without the object you want to composite in), producing image I_2.

3 Render the synthetic object and proxy together, including the shadows of the synthetic object on the proxy, producing image I_3.

The compositing steps are then:

4 Composite I_1 over I_2 to get I_4: this is an image of the synthetic object on top of the proxy without shadows.

5 Compute the pure shadow image I_5 as the difference $I_4 - I_3$ in the

RGB channels: this encodes how much light is lost in each pixel of the proxy due to the synthetic object's shadow.

6 Subtract the shadow image I_5 from the real photo, darkening it where the synthetic object should cast its shadow, to get I_6.

7 Composite the synthetic object I_1 over the shadowed real photo I_6 to get the final image I_7.

1.6 Faking Motion Blur and Depth-of-Field

We earlier discussed how motion blur and defocus from depth-of-field can be accurately approximated using supersampling in time and position on the aperture. However, this isn't always the most desirable approach:

- avoiding visible noise when the blur is extensive requires a large number of samples per pixel;
- there might not be adequate supersampling support in the renderer (e.g. OpenGL);
- it doesn't provide any means to adjust the effect after 3D rendering despite it mostly looking like a 2D image-space phenomenon.

This last point, control over the amount of blur, is a big deal despite our earlier arguments that motion blur and defocus are critical elements to get physically "right" for convincing renders. Like every aspect of graphics, what is physically correct (at least, under some limited mathematical model of the scenario) isn't necessarily the best answer particularly when other approximations are in play, and especially taking into account artistic intentions. Physical correctness is a good starting point, but not the end goal.

In the case of motion blur, the physically correct motion blur might make it too hard to see the most important details of a character's performance and thus should be reduced — perhaps only for one part of the image (say the character's face). For an exciting action scene the director might want a jerky, strobe-like effect as if filmed with a very short shutter time, but would want to tweak exactly how jerky it looks only at the end of production once all elements are in place. Extreme additional blur might be called for on just one element to give a stylistic impression of incredible speed.

Depth-of-field is commonly exploited with real cameras to direct the viewer's attention. Even a purely static scene shot with an unmoving camera can tell a dynamic story if the focus purposefully shifts from one object to another. Directors sometimes want the same freedom to adjust depth-of-field as late in the process as possible, perhaps in non-physical ways.

A common approach, therefore, is to save some extra data for each image to guide an image-space blur during compositing to approximate the effect of motion blur and defocus blur.

For motion blur, in the (fragment) shader we estimate the "displacement" (velocity integrated over the frame time) of the object at that point projected into screen space — essentially how many pixels it is expected to move horizontally and vertically in the duration of the frame. We calculate each pixel in the final motion-blurred image as an average of raw pixels that could move through it during the frame — or alternately put, a raw pixel at (i, j) with displacement $(\Delta i, \Delta j)$ would contribute to the whole line segment between (i, j) and $(i + \Delta i, j + \Delta j)$. If the motion blur is to be increased or decreased, the $(\Delta i, \Delta j)$ displacement can be scaled up or down.

For depth-of-field, we simply save the Z-buffer along with the image. The Z-buffer allows us to compute how far a pixel is from the desired focus plane, which in turn controls the radius of a blur kernel (which has the shape of the camera aperture, perhaps just a circle). This is just like motion blur except instead of contributing to averages along a line segment, the pixel will contribute to averages within a certain radius.

1.7 Other Compositing Operations

As hinted at above there are many other ways to combine images. Adding, subtracting, multiplying, blurring, or any other operation can be used. Extra channels can be especially useful — for example Z-buffers can guide a smarter over operation between two interlaced images (where one image is on top of the other depending on their relative Z values). The Z-buffer is also useful in faking stereo 3D images, translating objects for the left versus right eye depending on depth. Some shading or lighting operations can even be deferred to the compositing stage if the per-pixel inputs are saved (like normals, texture coordinates, etc.).