Tricks of the Trade

Andrew Glassner

Everybody has their own favorite bunch of programming tricks. A few years ago, I started the "Graphics Gems" series of books to bring together lots of these handy techniques. The *Journal of Graphics Tools* (http://www.acm.org/jgt) has continued that tradition. In this column, I thought I'd describe some little tricks of mine that are too small even for *JGT*.

Small can be beautiful. Most of these techniques are so useful that I've bundled them up into little libraries. Since they're not related in any deep way, I'll simply present them in alphabetical order.

The amazing expando-square

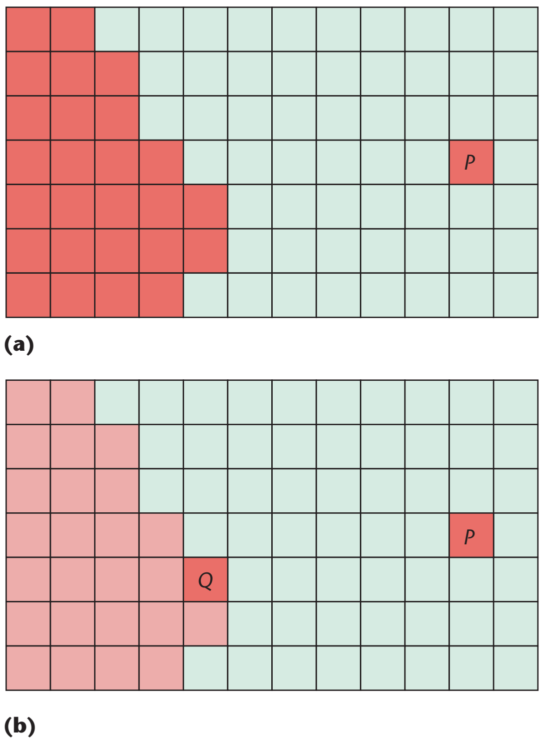
This trick is useful when working with raster images. Suppose that you have a pixel of interest, *P*, and you're looking for the nearest pixel *Q* in the image that satisfies some property. For example, you might want to find the nearest pixel *Q* in the image that has the same color as *P*.

The brute-force way to solve this problem is to scan the entire image pixel by pixel. For each test pixel *T*, we first see if it has the desired property. If so, we compute its distance to *P*. If this distance is the smallest distance of any pixel found that has passed the test so far (or if it's the first pixel that passes the test), we provisionally assign it to the result pixel *Q*. When we reach the end of the image, *Q* is left with the nearest pixel with the desired property.

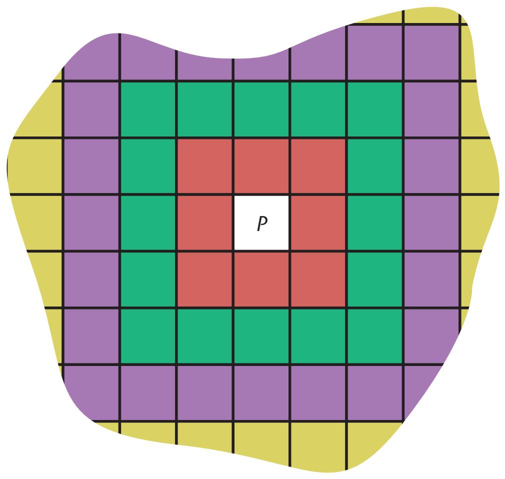
Technically, if there are several points at the same distance, then *Q* will hold only one of them. Which one is kept depends on the specific test used for *T*. If we only take *T* if it's strictly closer than the value already in *Q*, then *Q* will be left with the first point encountered at that distance.

This can be a slow procedure, particularly on big images. One way to speed it up is to realize that we don't actually need the distance from *P* to *T*, which requires a square root. Because the square root function is monotonic, we can just use the squared distance throughout.

This is faster, but it still can be pretty slow. And it can be frustrating when we look at the picture. Suppose we have something like Figure 1, where we have a little red pixel in a green field and a red blob nearby. In this case, we're looking for the red pixel nearest to *P*. It seems



1 (a) The red pixel on its own is *P*. We want to locate the nearest red pixel to *P*. (b) The nearest pixel *Q* is marked.



2 Searching around pixel *P*. The pixels belonging to a square of radius 1, 2, and 3 appear as red, purple, and yellow, respectively.

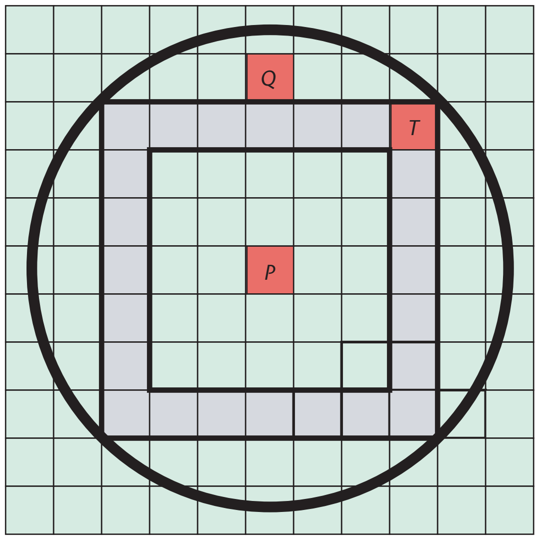
wasteful to scan the entire image when Q is so nearby.

Let's take a cue from my January/February 2001 column on filling and change our scanning process from a complete scan to an expanding square. Figure 2 shows the idea. We'll start with a square of radius 1 (equivalent to a side length of 3), which gives us the eight pixels around P. We scan these eight pixels and test them. If none of them pass our test, then we increment the radius to 2 (which gives us a side length of 5) and repeat, growing the square one pixel at a time until we find a pixel T that passes the test (and is thus assigned to Q). At the end of each square, we look to see if we have a pixel Q. If so, we stop right away, since any other pixels that pass the test will be farther away than what we have so far.

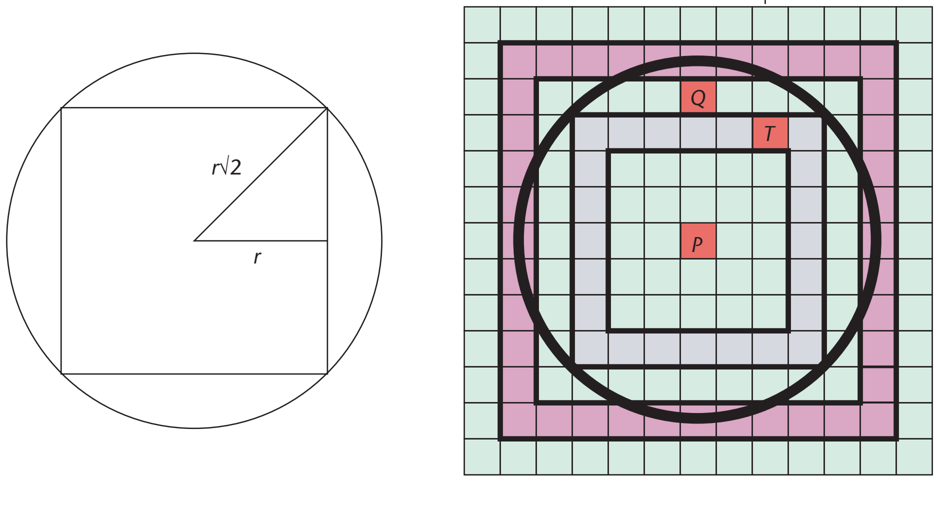
Well, almost. There's a gotcha here, shown in Figure 3, where the real pixel Q that we're looking for doesn't show up until after we've found a pixel T that will actually end up being farther away than Q. The problem, of course, is that we're expanding a square rather than a circle.

The problem is easily cured, using the idea in Figure 4. When a pixel T passes the test, we find the circumscribing circle around the current square and then find the square that encloses that circle. Then we continue scanning until we reach that larger square. If the current square has a radius r, then the limiting square has radius $r\sqrt{2}$.

This algorithm has a potential slowdown that the full-scan approach doesn't suffer from: the expanding square may fall off one or more sides of the image while it's still scanning, as in Figure 5. If we try to access a pixel that's off the bitmap, we can get an error (or bogus data, which can be just as bad). The easy but slow way to handle this is to test each pixel's coordinates before looking up its value, and only proceeding if the pixel is actually in the image. A faster approach is to run this test for each side of the square. If the entire left column is off the top of the image, for example, we can skip it entirely.



3 Failure of the square-searching technique. Pixel Q is the nearest red pixel to P, but we found T first. If we stop now, we'd have the wrong result.



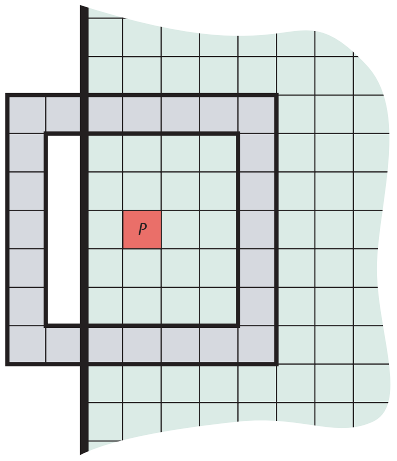
4 Fixing the square-search failure. When we find a pixel T, we find the radius of the circle that encloses the square that T belongs to (in gray) and continue searching squares until they exceed that radius. (a) The geometry. (b) When we search up to the new square (in purple), we find the nearer point Q.

Blobs

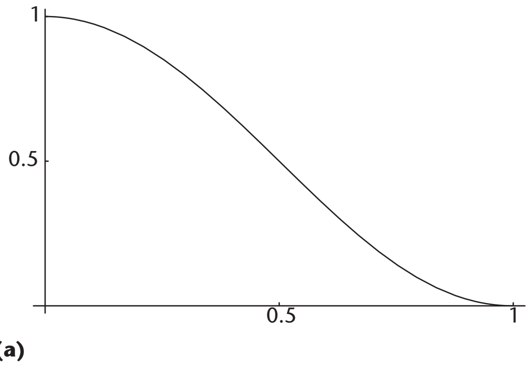
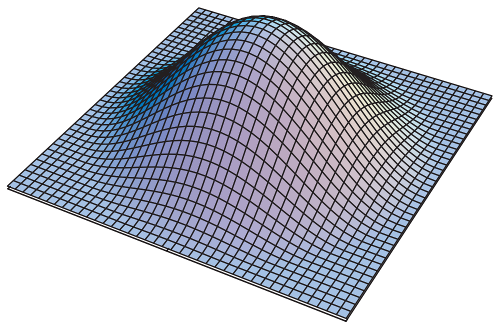
A nice blob function can be handy for all kinds of occasions. Blobs are radially symmetrical, smooth shapes—the sort of thing you'd get if you plopped a spoonful of pudding onto a dish.

It's conventional to build blobs defined in the interval [0, 1] and scaled so that they have a value of 1 at $r = 0$ and 0 at $r = 1$. The best blobs have a derivative near zero at both $r = 0$ and $r = 1$, so that they blend into their surroundings without a seam.

I like to use three blob functions. The first comes from Wyvill, McPheeters, and Wyvill. This is the blob shape I used in my January/February 1997 column:



5 Sometimes the searching square can partially fall off the image.

6 The Wyvill-McPheeters-Wyvill blob function. (a) The curve. (b) The radially symmetric blob.

$$b_W(r) = \begin{cases} \text{if } r<1 & 1-(22r^2+17r^4-4r^6)/9 \\ \text{else} & 0 \end{cases}$$

I plotted this function in Figure 6.

The derivative with respect to r is

$$\frac{db_W}{dr} = \frac{1}{9}(-44r+68r^3-24r^5)$$

Plugging in $r = 0$ and $r = 1$ confirms that b_W has a zero slope at both points.

By the way, it's more efficient to implement this blob using the form in which it was originally published:

$$b_W(\alpha) = 1 - [\alpha(22 - \alpha(17 - 4\alpha))]/9$$

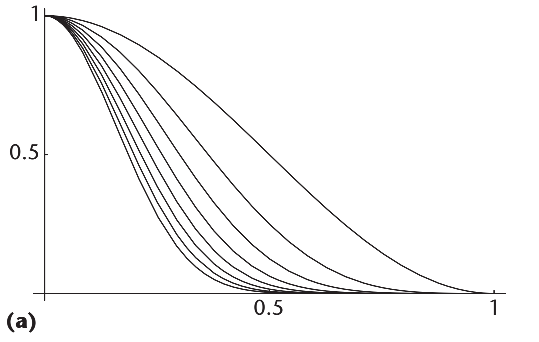
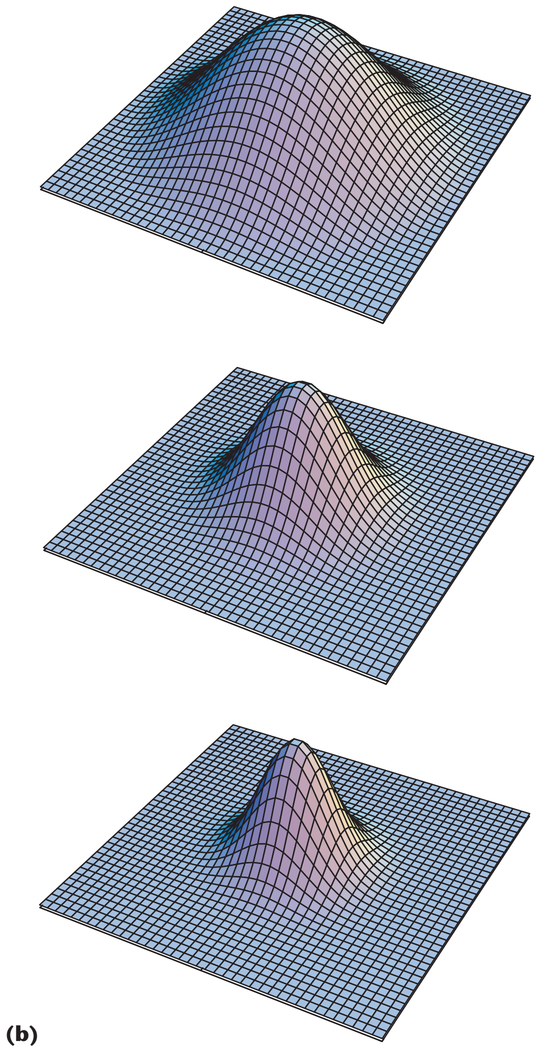
where $\alpha = r^2$.

This is a nice blob, but it doesn't have any tunable parameters. If you don't like the shape, there's not much you can do about it. We can make a tunable blob based on the trick that Phong shading uses to create a specular highlight out of a piece of the cosine curve. By cranking up the parameter p (usually an integer), you can squeeze together the blob and make it tighter:

$$b_P(r,p) = \begin{cases} \text{if } r<1 & (1+\cos^p(r\pi/2))/2 \\ \text{else} & 0 \end{cases}$$

I plotted this is in Figure 7 for some different values of p. Since the derivative of cosine is sine and $\sin(0) = \sin(\pi/2) = 0$, this blob is also flat at both ends.

These last two blobs share a close relationship: $b_W(r)$ is simply a very good polynomial approximation to $b_P(r, 1)$.

7 Phong blob function. (a) Plots for $p = 1$ (broadest) to $p = 8$ (narrowest). (b) The radially symmetric blob for $p = 1$, $p = 3$, and $p = 5$.

The Phong blob is useful and can be adjusted, but there's another version I like even more. The simple Gaussian blob is very nice and can be easily tuned:

$$b_G(r,p) = \begin{cases} \text{if } r<1 & (e^{-(rp)^2} - e^{-p^2})/(1-e^{-p^2}) \\ \text{else} & 0 \end{cases}$$

This is plotted in Figure 8. Like the other blobs, it has

$b_G(0) = 1$ and $b_G(1) = 0$. At $r = 0$, this has a zero derivative like the other blobs. At $r = 1$ the derivative is

$$\frac{db_G}{dr} = (-2p^2 e^{-p^2}) / (1 - e^{-p^2})$$

When $p = 1$, the derivative at $r = 1$ is about –1.2, which is pretty big and would make an obvious seam if we use this function for blending or filtering. Happily, the derivative falls off fast. By $p = 3$ the derivative at $r = 1$ is about –0.002, which is pretty darn close to zero. The derivative drops very quickly: at $p = 5$, it's down to less than 10^{-9}.

I like to use b_G for general filtering and blending work when I need to be able to tune the parameter, and I'm willing to use values of $p \geq 2$. For smaller values of p I use b_P. If I don't need a tunable shape, but just something smooth and blob-like, b_W is my choice.

Burp

Interpolation is an important and frequent problem in computer graphics when you've got a few data points, and you want to find some in-between values.

When you have two pieces of data, the simplest way to blend them is with linear interpolation (sometimes referred to as *lerp*). Lerp takes the two pieces of data (say p_0 and p_1), weights them, and adds up the result:

$$P = \alpha_0 p_0 + \alpha_1 p_1$$

The most common form of lerp is the *linear uniform lerp*, where the weights add up to 1. I call this a *lurp*. Using a single parameter α to control the blending, we can write the lurp P of p_0 and p_1 as a function L:

$$P = L(\alpha, p_0, p_1) = (1 - \alpha)\, p_0 + \alpha\, p_1$$

where $0 \leq \alpha \leq 1$. When $\alpha = 0$, $P = p_0$, and when $\alpha = 1$, $P = p_1$. For intermediate values of α, we get a smooth blend between the extremes.

If we have more than two objects to combine, we can perform a uniform, multipoint lurp:

$$P = \sum_{i=0}^{n-1} \alpha_i p_i \quad \text{where} \quad \sum_{i=0}^{n-1} \alpha_i = 1$$

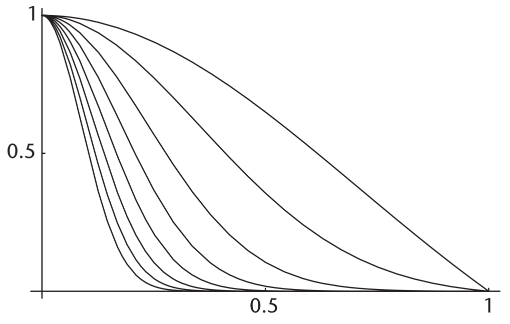
I'll assume we're given all the values p_i and the weights α_i; our goal is to use them to compute P.

To compute P, we need to scale each data element p_i by a factor α_i and then add up the results. Usually, these steps are both easy to do. For example, if we want to lurp a few RGB colors, we could just scale each component by the corresponding weight and then add up all the components.

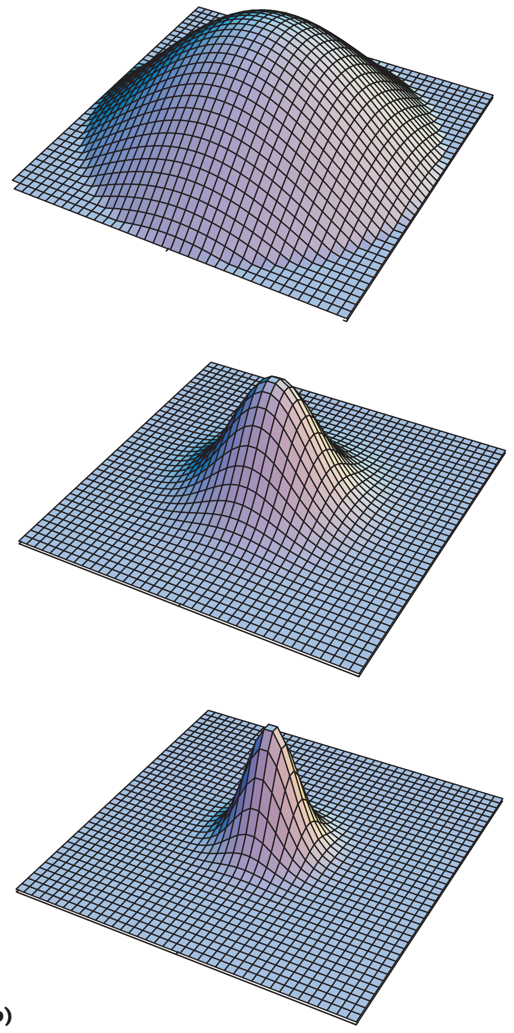
But sometimes these conditions are harder to meet. In my March/April 1997 column, I talked about the visual effects created by Phong shading due to its step of interpolating vector components and then normalizing the result. As an alternative, I presented an angular interpolation technique that rotated one vector into the other in equal steps.

That operation is inherently a lurp: it takes exactly two vectors and one interpolation running from 0 to 1, with our parameter α moving the result from one to the other.

Now suppose that we want to blend several vectors simultaneously. If we weight the components, there's no problem. But suppose we want to use the angle-interpolation formula? That works with only two vectors. Somehow we have to convert the earlier general n-

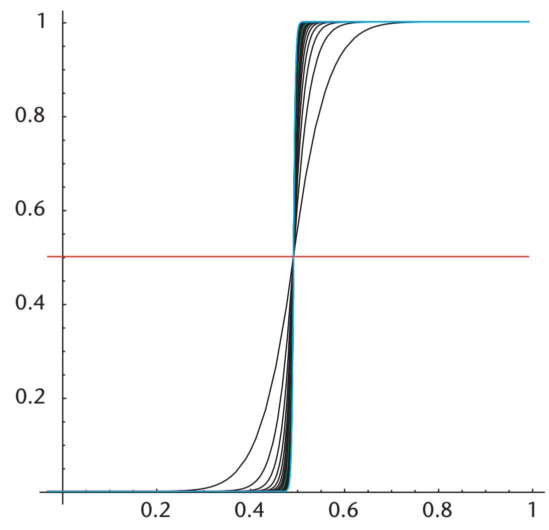


(a)

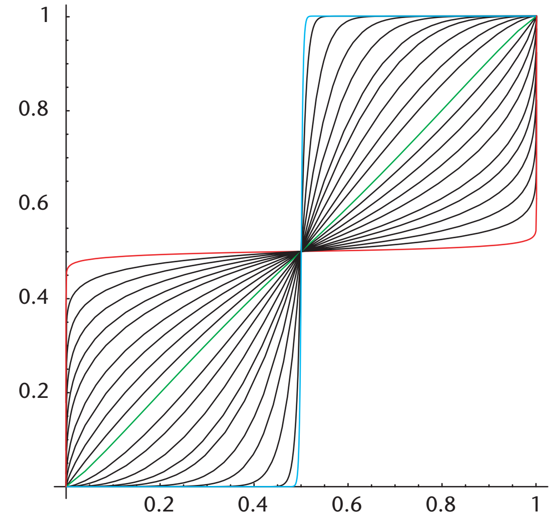


(b)

8 Gauss blob function. (a) Plots for $p = 1$ (broadest) to $p = 8$ (narrowest). (b) The radially symmetric blob for $p = 1$, $p = 3$, and $p = 5$.



(a)



(b)

9 The ease function. In both plots, $0 \le r \le 1$. (a) The old parameterization. The *p* values run from 0 to 100, in 20 equal steps of five. $p = 0$ is red, $p = 50$ is green, and $p = 100$ is blue. (b) The new parameterization. The values of *p* run from 0 to 1 in equal steps of 0.05. $p = 0$ is red, $p = 0.5$ is green, and $p = 1$ is blue.

element formula into a combination of lurps.

For this application, and others like it, we want a *bilinear uniform interpolation*, which I can't resist calling a *burp*.

Let's get to the general formula for a burp by looking at an example using three elements:

$$P = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 \text{ where } \sum_{i=0}^{2} \alpha_i = 1$$

I'm going to break this down into steps. In the first step, I'll write a two-element lurp of p_0 and everything else, which I'll bundle together into a composite, temporary result r_1. Then I can write

$$P = \frac{\alpha_0}{\alpha_0 + \alpha_1 + \alpha_2} p_0 + \frac{(\alpha_0 + \alpha_1 + \alpha_2) - \alpha_0}{\alpha_0 + \alpha_1 + \alpha_2} r_1$$

Since the weights α_i add up to 1, the first weight is simply α_0, which is what we want to apply to p_0. The remaining weights are combined together in the other weight and applied to r_1, which I haven't defined yet.

To find r_1, we apply the same idea again. Write r_1 as the lurp of p_1 and everything else, which we'll roll together and call r_2:

$$r_1 = \frac{\alpha_1}{\alpha_1 + \alpha_2} p_1 + \frac{(\alpha_1 + \alpha_2) - \alpha_1}{\alpha_1 + \alpha_2} r_2$$

And r_2 at this point is nothing more than p_2.

The trick here is to break down the big multipoint lerp into a series of two-point lurps, which by definition uses weights that summed to 1. If you expand all the weights in the last few equations and cancel out common terms, you'll find that in the final sum each data element p_i is weighted by α_i—no more and no less. Success!

So we've found that if we want to compute an interpolation of $n > 2$ data points, but we need to use a two-point uniform interpolation formula, then we can do so using burp.

Here's the summary for computing burp. We want to find $P = r_0$:

$$r_0 = \sum_{i=0}^{n-1} \alpha_i p_i \text{ where } \sum_{i=0}^{n-1} \alpha_i = 1$$

We'll start by computing r_{n-1} and then work our way back up to r_0, using our two-point lurp function L:

$$\begin{aligned} r_{n-1} &= p_{n-1} \\ r_k &= \frac{\alpha_k}{\beta_k} p_k + \frac{\beta_k - \alpha_k}{\beta_k} r_{k+1} = L\left(\frac{\alpha_k}{\beta_k}, r_{k+1}, p_k\right) \\ \beta_k &= \sum_{j=k}^{n-1} \alpha_j \end{aligned}$$

This is a handy little technique.

Easy does it

In the January/February 2000 issue, I presented a nice easing curve for making 3D Celtic knots. Ease curves are useful any time you want to smooth the transition from one value to another. The parameter of the ease curves lets you adjust the speed of the transition. Eases are useful in all sorts of applications. They're particularly handy in animation, where they help us smooth out the motion of objects when they start and stop.

The ease function I cooked up comes from two little helper functions. The ease function $e(r, p)$ is:

$$\begin{aligned} s(r) &= r^2(3 - 2r) \\ t(r,p) &= s((2r)^p)/2 \\ e(r,p) &= \begin{cases} \text{if } r < .5 & t(r,p) \\ \text{if } r < 1 & 1 - t(1-r,p) \\ \text{else} & 0 \end{cases} \end{aligned}$$

Like the blobs, the ease curve runs from $0 \le r \le 1$ and is controlled by the parameter p (in this case, a floating-point value). These equations are how I originally implemented and presented the ease curve. Figure 9a

Table 1 . Data for the piecewise mapping function $m(p)$ for ease curves.

Input x_i	0	0.125	0.25	0.375	0.5	0.625	0.75	0.8125	0.875	0.9375	1
Output y_i	0.01	0.08	0.18	0.35	0.6084	1	2	3.5	6	15	100

shows the family of curves it makes as the parameter p varies. In this figure, I varied p from 0 to 100 in steps of five—one half of the whole range of curves is completely missing. See Figure 9b for the shapes that the ease function is capable of creating.

Although I loved the curves produced by this easing function, I found that adjusting the parameter was always a kind of hit-and-miss affair because of the nonlinear nature of the control space. At small p values, a change of 0.1 can make a visible difference; for big values, a change of 10 or more doesn't seem to do much. Notice how in Figure 9a the squarish shapes belonging to large values of p dominate the range. The closest that this function comes to a diagonal line is at $p = 0.6084$. The range from [0, 0.6084] makes up the slow eases, while [0.6084, 100] makes up the fast ones.

I spent some time trying to find a smoother range for p. Nothing analytical seemed to work out very well. I eventually decided on a little piecewise function m that maps p from an input range [0, 1] to the range [0, 100] wanted by e and does it in a way that creates a roughly uniform spacing of the curves.

The function consists of 10 linear pieces. The first few pieces are each 0.125 unit wide; the last few are half that width to provide better control. Table 1 gives the endpoints for each segment. To use this data, first find the appropriate segment of the function. Suppose that the input p lies in the region $[x_j, x_{j+1}]$, which has values y_j and y_{j+1}, respectively. Then we can find the new value for p from

$$m(p)=y_j+(y_{j+1}-y_j)\frac{p-x_j}{x_{j+1}-x_j} \qquad (1)$$

Figure 9b shows a plot of $e(r, m(p))$. Here the value of p moves from 0 to 1 in equal steps of 0.05. The spacing is now roughly uniform. Specifying p in the range [0, 1] with equal steps makes it much easier to find the ease you're after.

Multipoint weighting

This topic is similar to burp but has a different set of goals.

Suppose that we have a set of points A_i located in space. (The space can be any dimension, as long as we can compute the distance $|AB|$ between two points A and B.) Each point has an associated value p_i. We don't know anything else about the points. Specifically, we don't have any kind of information on connectivity (that is, if they're joined up into polygons or something else) nor on what else is going on in the universe. There's just a bunch of points, each with a position and a value. Now we're given a new point P, and we want to find a value for that point by weighting and combining the input points:

$$P=\sum_{i=0}^{n-1} w_i p_i$$

for some weights w_i.

Recall that in the section on burp we already had the weights, and our goal was to figure out how to use them. Here, we want to find the weights.

We'll insist on getting only two things from our technique: one, whenever the point P is on top of one of the inputs, it has the value of the input, and two, the values of P vary smoothly. I can't say too much more because we don't know anything about the points themselves. For example, we might want to say that whenever P lies on the straight line between two points A and B, P is only made up of values from those two points. But what if some other point C also lies on that line and is in fact closer to P than A? There could be a lot of these little unexpected gotchas, so I'll stick with these two simple conditions.

I tried a lot of approaches to solving this problem, but they all left something to be desired. For example, I tried placing little blobs (like the ones in the blob section) on every input point and using the height of the blob at every pixel as the weight for the corresponding input. But if the blobs got far enough apart, that left big, empty areas in the image where all the blobs were essentially zero. The results were constant in those regions or else got very noisy as these tiny numbers fell below the ability of the computer to sensibly manipulate them.

Then I tried computing the distance of each input to every other input and combining those distances with the location of the input point P to compute relative weights for each pair of inputs. I then used those inputs to figure out a smooth set of individual weights.

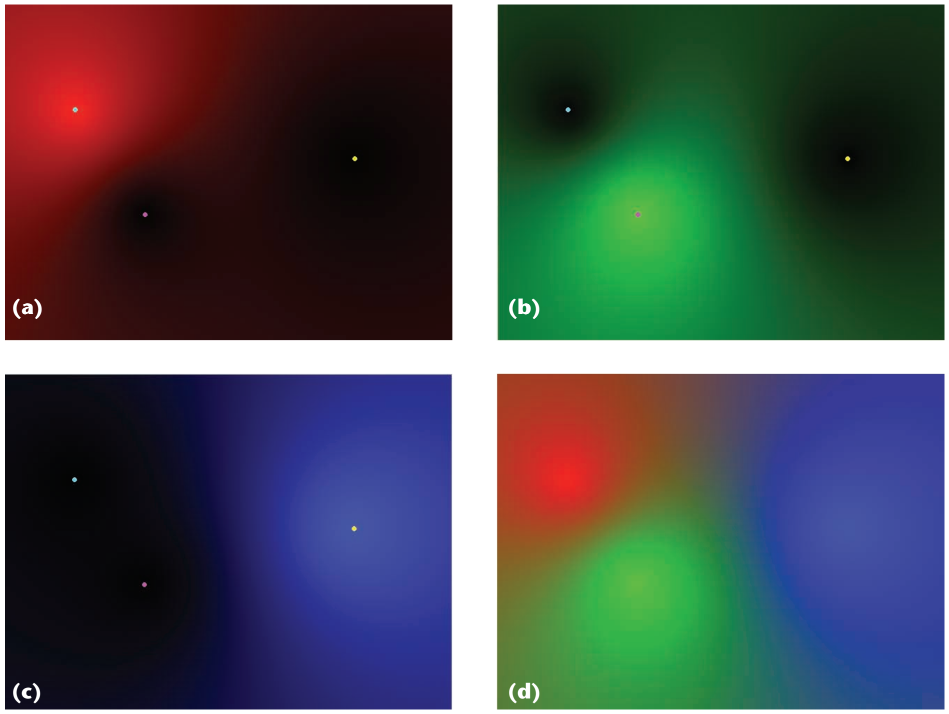
These approaches were all too complicated and produced results that had seams, edges, and other problems. Surely there was a technique that was simultaneously fast, easy, and without artifacts. One that would fill in the whole image with smoothly interpolated values, no matter whether the input points were close to one another, far, or distributed in uniform or nonuniform ways.

I kept plugging along and one evening I found a sweet little technique that works just fine for this problem. Let's begin by finding the distance d_i as the distance between P and each of the n input points A_i:

$$d_i = |P - A_i|$$

Now we'll normalize the distances, so that they sum to one. I'll call the normalized distances g_i:

$$g_i = d_i \Big/ \sum_{i=0}^{n-1} d_i$$



10 **Interpolating three points in the plane. In parts (a), (b), and (c), each weight has been assigned to a different color channel. Note how the value is a maximum right on top of the corresponding input point, and drops to zero at the other points. (d) The three channels overlaid.**

To make our formula, let's look at each weight independently. Suppose we want to compute w_0.

If $P = A_0$, then we want $w_0 = 1$. Since we're on top of A_0, the normalized distance $g_0 = 0$. So let's provisionally assign $w_0 = 1 - g_0$. Still thinking about weight w_0, let's now suppose that point P is sitting on top of input point 2, so $P = A_2$. Now we want w_0 to be zero, since only point 2 should contribute to the sum. Because we're on point 2, $g_2 = 0$. To send our weight w_0 to zero, we just need to multiply it by 0. Tacking this onto our value so far gives us $w_0 = (1 - g_0)g_2$. The same argument applies to all the other points, so if we have three points, then $w_0 = (1 - g_0)g_1g_2$. Similarly, we can apply the same argument to the other weights, finding that $w_1 = (1 - g_1)\, g_0\, g_2$ and $w_2 = (1 - g_2)g_0g_1$.

What happens in between? Well, the closer we are to a given point the stronger its weight, which is just what we want.

Here's the general recipe:

$$v_i = (1 - g_i)\prod_{\substack{j=0 \\ j \neq i}}^{n-1} g_j$$

To get our weights w_i, we normalize the v_i:

$$w_i = v_i \Big/ \sum_{i=0}^{n-1} v_i$$

That does the trick! We plug these weights into the formula for P at the start of the section, weighting the value p_i at each A_i by w_i and then add them up.

Because everything is continuous and smooth, we get just the results we want. On top of an input point, we get just the value for that point. When we're not on a point, the nearest points have the greatest influence and the others fade out slowly. The values of P are smooth and there are no blank spots where all the weights drop out, nor any flat spots where all the weights are essentially the same.

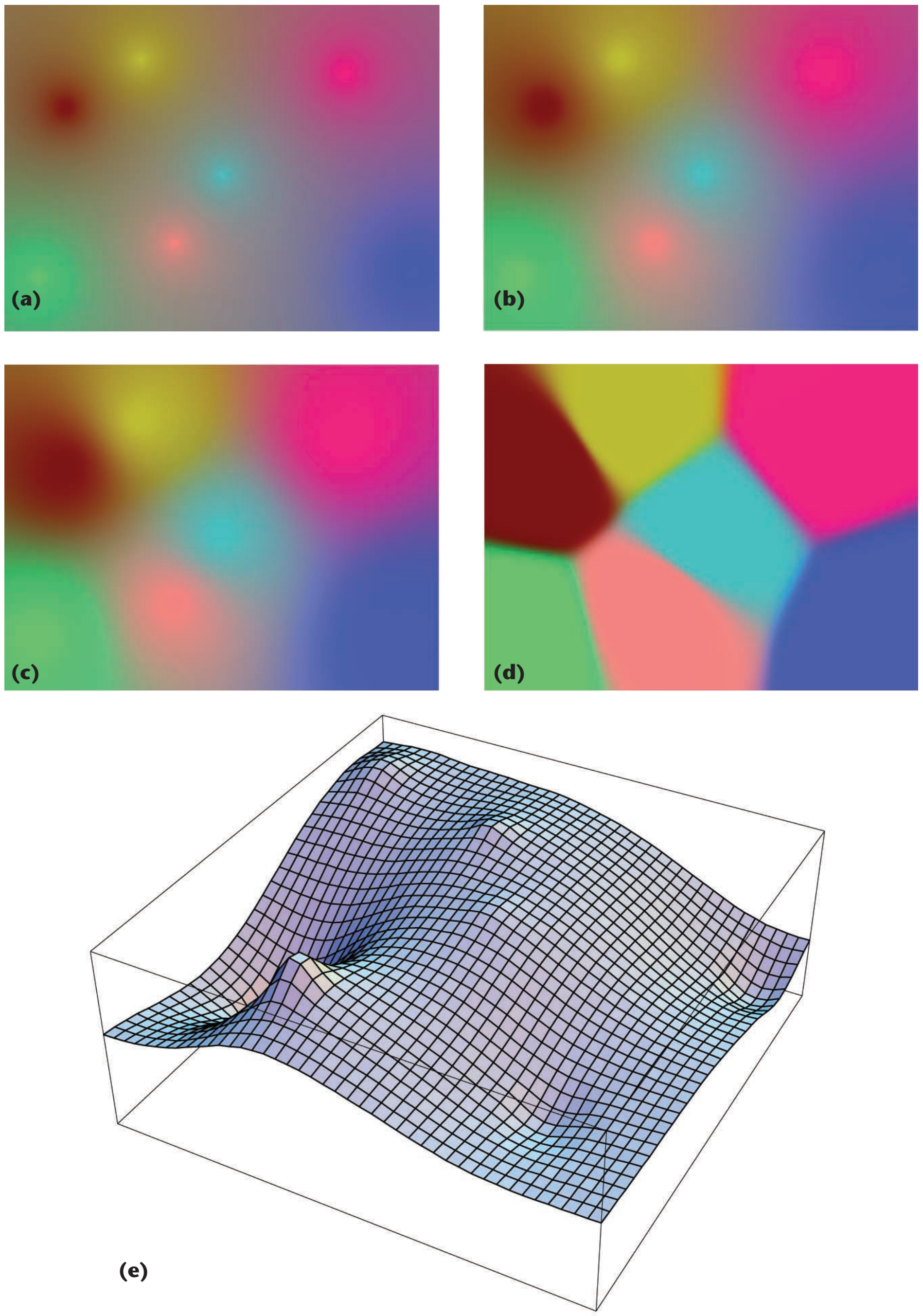
Figure 10 shows this formula applied to a three-point interpolation in the plane. Since there are only three points involved, I colored each weight with a different primary color. Figures 10a through 10c show the weights for each point separately, while Figure 10d shows them all overlaid. If you added up the color values at each pixel, you'd find they always sum to 1.

Figure 11 shows a more complicated example. In Figure 11, I've also applied the ease function from the last section to the weights just before using them; this also means I had to also re-normalize them:

$$t_i = e(w_i, m(p))$$
$$w_i' = t_i \Big/ \sum_{j=0}^{n-1} t_j$$

As the equation shows, I used the mapping function $m(p)$ described earlier to make it easier to choose good values for the ease curve.

Combining one technique with another is the kind of thing that becomes easy when all the routines are in a library. I find that when I'm working on an algorithm it's nice to experiment by combining techniques like this. And this is a good example of my claim earlier that the ease curve was useful in all sorts of places! Notice that in Figure 11d each blob spreads out until it hits another blob of stronger value. The result is a rough version of a Voronoi diagram for the colored points. Of course, pushing the blobs this far defeats our original purpose of creating a smoothly varying field, but it's fun to poke



11 **Multipoint weighting. (a) Some colors in the plane. (b) Easing the weight with *m*(.625). (c) Further easing with *m*(.6625). (d) A lot of easing with *m*(.8625). (e) A height plot for the grayscale intensity of Figure 11 b.**

around to see what happens when we push algorithms to their extremes.

Remember that this interpolation technique doesn't care how many dimensions it's working in, so you can interpolate 3D volume densities, 4D space–time events, or the genetic codes for 9D bumblebees as easily as colors or heights.

A final word

I didn't know about the techniques in this column before I cooked them up. Some of them seem so nice that they must have been discovered before, but I haven't seen them reported. If you're already familiar with some of these techniques, I'd love it if you could send me a note with a pointer to the publication where you saw it before.

There's nothing like figuring out an elegant solution to a problem. When I find one of these, I usually try to find a way to encapsulate it into a stand-alone function call or two, so I can put it into a library. All the procedures in this column are simple to write up and implement. I hope you enjoy using them as much as I have. ■

Readers may contact Glassner by email at andrew_glassner@yahoo.com.