Andrew Glassner's Notebook

http://www.research.microsoft.com/research/graphics/glassner/

Going the Distance

Andrew Glassner

Microsoft Corporation

Quick, what's the fastest way to get from one point to another?

A straight line, right?

Well, it depends on what you mean by "straight."

One of the fun aspects of non-Euclidean geometries is discovering how familiar shapes change under new rules. The new rules mean we have to think about distances differently than in the world Euclid described. For example, suppose you want to travel from Paris to New York. In practical terms, you can't take the straight-line path—that choice would require you to drill a tunnel through the Earth. Instead, you might take a boat or a plane, both of which travel in curved paths over the Earth's surface. Compared to the tunnel, the plane takes a longer path and you'll need more fuel, but at least your wings won't get ripped off.

In this month's column, we'll take a look at a couple of simple 2D geometries that obey different distance rules than those that Euclid described.

The Euclidean world

We'll start with the familiar world of 2D Euclidean geometry. The distance d_E between two points A and B is given by the familiar formula

$$d_E(A,B) = \sqrt{(A_x - B_X)^2 + (A_y - B_y)^2}$$

Let's put this formula into action. Suppose we have a circle of radius r, centered at point M. Points on the circle are those where the value of the function C are 0:

$$C(P,M,r) = d_E(P,M)^2 - r^2$$

Figure 1 shows a region of the plane with this function plotted. In all of the figures in this column, the values in the plotted domain are scaled to place black at the minimum value and white at the maximum. The yellow curve indicates where the function has a value of 0. Of course, it's a circle.

More interesting than the circle is the blob. There are lots of blob functions; I like the one developed by Wyvill, McPheeters, and Wyvill. It's a circularly symmetric shape parameterized by the distance r from the center and the size R of the blob. The blob equation B is given by

$$B(A,B,R) = \begin{cases} 1 - \dfrac{\alpha(22-\alpha(17-4\alpha))}{9} & \text{if } 0 \le \alpha \le 1 \\ 0 & \text{otherwise} \end{cases}$$

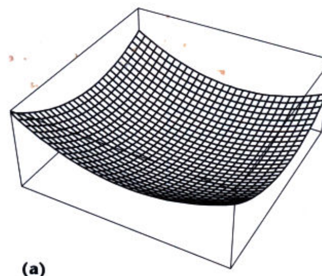
where

$$\alpha = \left(\frac{d_E(A,B)}{R}\right)^2$$

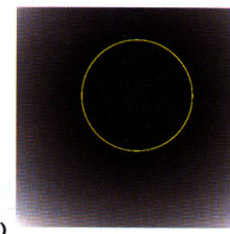
This is plotted in Figure 2.

Figure 3 shows three blobs in the plane. Where blobs overlap, their values are simply summed up. The yellow line here indicates the curve where the value is 0.5.

Another interesting function of distance demands your attention following your recent wedding (congratulations!). The question is where to live so that you



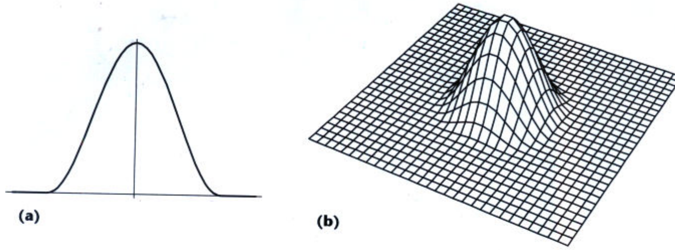
(a)



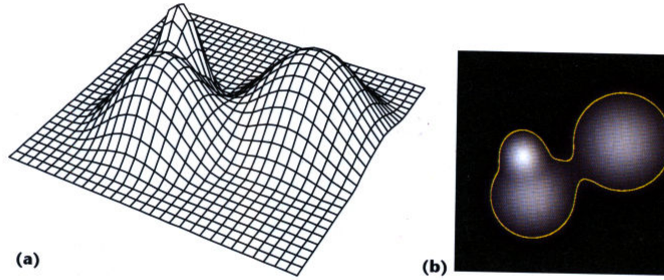
(b)

1 (a) A height-field plot of the circle function, centered at (0.1, 0.2) with radius 0.5. (b) Heights are mapped to shades of gray. The yellow curve is that set of points with the value 0.0.

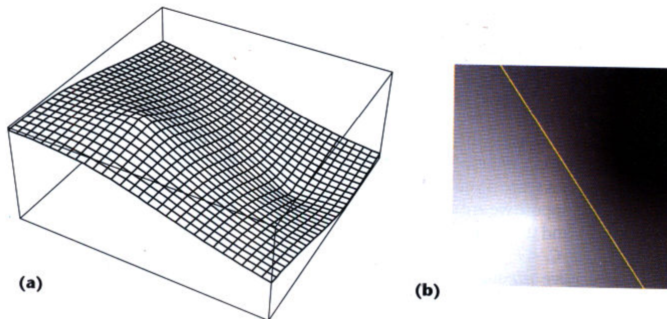
0272-1716/97/$10.00 © 1997 IEEE



2 (a) The blob function $B(r, 0.5)$. (b) A 3D plot of $B(r, 0.5)$ using the Euclidean distance metric d_E to find the value r between a point on the plane and the blob's center at the origin.



3 (a) Three blob functions. The centers are at (0.5, 0.2), (–0.3, –0.3), and (–0.4, 0.1), with radii 0.7, 0.6, and 0.3 respectively. (b) The grayscale version of the three blobs. The yellow line is an isocontour at a height of 0.5.



4 (a) The equal-distance function. One workplace is at (0.6, 0.2) and the other at (–0.3, –0.4). (b) The grayscale version. The yellow line is the set of points equally distant from the two workplaces.

5 Two points *A* and *B*, and a taxicab's route between them. The taxi needs to cover a distance of seven blocks, though the Euclidean distance is only five.

and your spouse travel the same distance to your respective workplaces. If your job is at point *A*, and your spouse works at point *B*, then your house at *P* can be anyplace where $d_E(P, A) = d_E(P, B)$. If we plot

$$H(P, A, B) = d_E(P, A) - d_E(P, B)$$

then we're once again looking for points *P* where the value of *H* is zero, shown in Figure 4 by the yellow line.

Our three formulas each depend on d_E to give us the distance between two points. Thanks, Euclid.

Taxicab geometry

You can't get there from here.

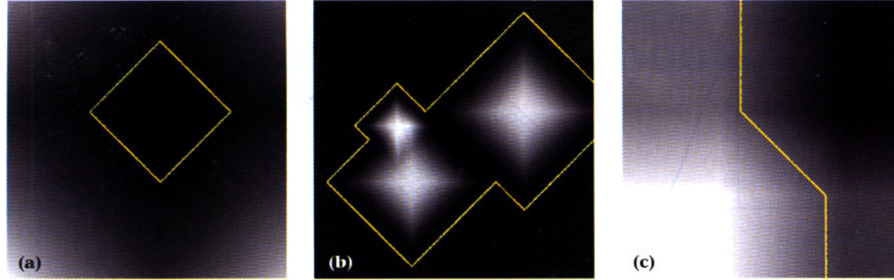
Well, you can, but you have to take a cab.

Suppose that you live in a city laid out on a grid, like midtown Manhattan. For simplicity, we'll assume it's a perfect square grid. If you want to take a cab from point *A* to point *B*, how far do you have to travel?

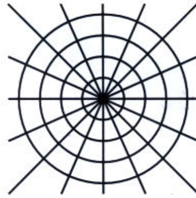
Figure 5 shows the situation. A little thought reveals that the taxicab distance $d_T(A, B)$ is given by

$$d_T(A, B) = |A_x - B_x| + |A_y - B_y|$$

Simply put, you need to go horizontally and then ver-



6 The three test functions using the taxicab metric: (a) the circle, (b) the three blobs, and (c) the two-workplace distance.



7 A polar grid for measuring distances at the North Pole.

tically. Strictly speaking, this is the shortest taxicab distance. As anyone visiting an unfamiliar city knows, a cab can take a very circuitous path from one place to another, and you can pay for many more miles than were actually required. But conceptually, the distance d_T is all you need to cover.

Let's revisit our three functions from the previous section, using d_T rather than d_E. Figure 6 shows the same functions with this new measure. The circle has become a diamond, as in Figure 6a. This makes sense if you think about it in terms of the form of d_T. Suppose we are sitting on a point P where $d_T(P, C, r) = 0$, and the center of the "circle" is to our northwest; that is, $C_x < P_x$ and $P_y < C_y$. Then as we move left, we have to move an equal amount down to keep the function at 0. This equal trade-off keeps us moving in a straight line. Similarly, we'd expect the blob function to also turn into diamonds, and Figure 6b shows that they do.

Figure 6c is a little more interesting. Some of the points equidistant from the two workplaces still lie along a line between the two, but then the line turns vertical. As we move along those vertical segments, we move the same taxicab distance from both workplaces.

Polar bear geometry

Last night I shot a polar bear in my pajamas. How a bear got into my pajamas, I'll never know (thanks, Groucho). These guys live up north, where instead of a square grid we can plot the landscape in a latitude-longitude format, as in Figure 7.

Up here at the North Pole, there are lots of ways to compute distance. Commonly, we first convert a point P from Cartesian (x, y) format to polar (r, θ) format, representing the radius from the pole and angle made with respect to a particular line. This rectangular-to-polar conversion is simply

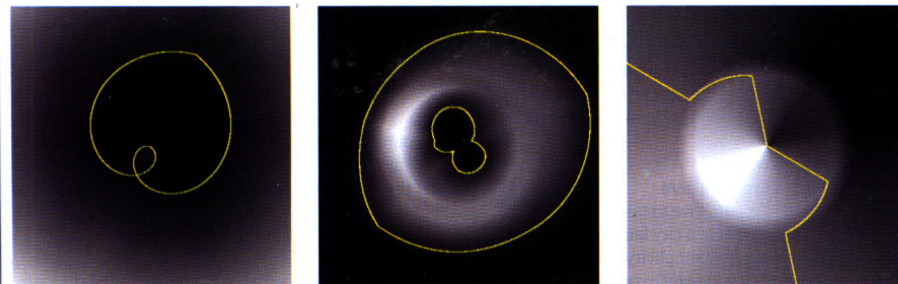
$$A_r = \sqrt{A_x^2 + A_y^2}$$
$$A_\theta = \tan^{-1}(A_y / A_x)$$

You can prove to yourself that using these values you can compute a distance d_N that is the same as d_E:

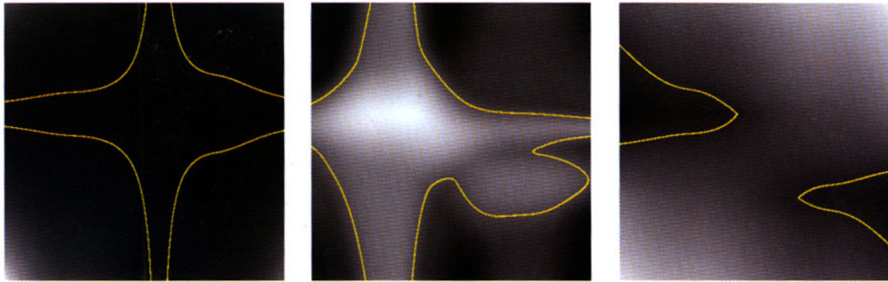
$$d_N(A,B) = A_r^2 + B_r^2 - 2A_r B_r \cos(A_\theta - B_\theta)$$

That's nice, but because $d_N = d_E$, if we plot our three functions again they'll look just as the same for the Euclidean measure.

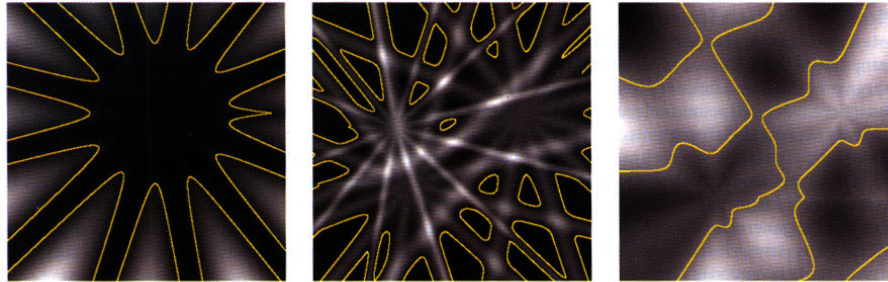
So let's cook up something that's a little different, but still interesting. Just like the taxicab distance, we can simply add up the difference in the radii and the difference in the angles. To make the images look interesting with the same equations that we used above, I arbitrarily decided to scale the angle measure down by dividing by 2 . One trick with the angles is that we want the difference between 5 degrees and 355 degrees to



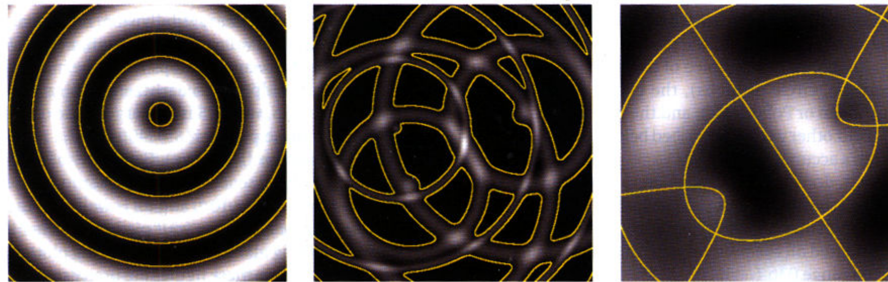
8 The three test functions using the modified polar metric d_P.



9 The three test functions using the arc-length metric d_A with the sine function.



10 The three test functions using the star metric d_S.



11 The three test functions using the ring metric d_R.

be 10 degrees, not 350. The cosine operator did that for us in the definition of d_N; we can do that procedurally for our polar-bear distance d_P:

$$d_P(A,B) = |A_r - B_r| + \frac{\mathrm{Min}(|A_\theta - B_\theta|, 2\pi - |A_\theta - B_\theta|)}{2\pi}$$

This is how far you'd have to travel under a spoke-and-ring type of monorail system. Figure 8 shows our three functions under the polar-bear measure of distance.

Other metrics

It's easy to change the definition of the distance function to try out other metrics. Here are a few fun metrics that I cooked up. The first one involved mapping the horizontal interval (–1, 1) to the interval (0, 2π) and then taking the sine of that value. The metric is the arc length of the fragment of sine curve between the X components of the two points, scaled by their vertical distance. In symbols, using $s(f, x0, x1)$ to represent the arc length of function f between arguments $x0$ and $x1$, the arc length metric d_A is

$$d_A(f, A, B) = s(f, A_r, B_r)\ |A_y - B_y|$$

Our three functions plotted with this metric using $f(x) = \sin(x)$ are shown in Figure 9.

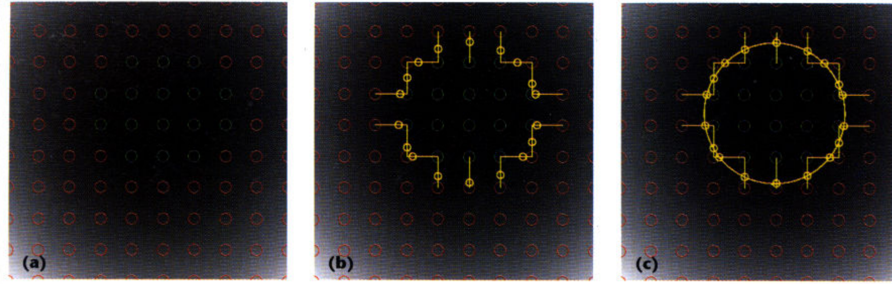
The star metric d_S has a sort of polar-ish feel:

$$d_S(A,B) = d_E(A,B)\cos\left(2\pi \tan^{-1}\left(\frac{A_y - B_y}{A_x - B_x}\right)\right)$$

The results are shown in Figure 10. Finally, the ring metric d_R takes the Euclidean distance and maps it into the sine curve:

$$d_R(A, B) = \sin(2\pi\, d_E(A, B))$$

The ring-metric images appear in Figure 11.



12 (a) The coarse resampling grid for the circle function measured with the Euclidean metric. Red circles are positive values and green are negative. (b) Lines join resampling points of different sign, and the small circle indicates the starting point for the curve along that line. (c) The grid, starting lines, and the followed curve.

Plotting implicit functions

I created the yellow curves in this column using a program that plots contours of implicit functions. Basically, you give the program a function $f(x, y)$ and a level value a, and it finds that set of points (or locus) where $f(x, y) = a$.

Plenty of commercial programs out there will do this job for you, but not everybody has one. I didn't when I felt like playing around with these curves. Happily, it's both easy and fun to make your own. In this section I'll give you the general approach that I followed, which does the job pretty well. It only took me a couple of hours to write the first version in Basic; it took about the same amount of time to write a much spiffier second version in C (the one I used for all the figures in this column).

I found it useful to have four runtime controls: the size of the graph (all of the plots in this article are 400 by 400 points), the choice of the function to plot (for example, circle or blobs), the choice of metric (like Euclidean or Polar-Bear), and the threshold a (usually 0). Four other values, described below, control the algorithm's accuracy.

First, draw the grayscale picture. To do this, I evaluated all the points in the image and kept a record of the minimum and maximum values of the function. Then I evaluated all the points again, this time scaling them to the range 0 to 1, which I mapped into grayscale values 0 to 255. If you have lots of memory and a slow processor, you could save the points in an array and then scale them in place instead of recomputing them all. I just drew the scaled points into the screen and forgot about them.

I use a very simple strategy to draw the level curve. I begin by searching the function to find points on the curve. I look around each point and follow the curve as far as I can in both directions. Then I search for another point. This way if there are several disconnected curves or segments, I can pick up each one in turn, at the risk of going over some of them more than once.

The first step in finding a curve to follow is to find a point on the curve. If we're plotting a function $z = f(x, y)$, then we're looking for points P where $f(P) = 0$. If we find two nearby points A and B such that $f(A) < 0$ and $f(B) > 0$, then we can search the line AB for points where the function goes through zero. I find these points by scanning a coarse grid on the domain. The density of this grid needs to match the high-frequency content of the function being plotted; smooth functions can be sampled loosely, while wiggly ones needed a denser mesh. In this article, I sampled Figures 1, 3, 4, and 6 with a mesh 10 samples on a side; the other figures used a 30-by-30 grid. Figure 12a shows the coarse grid.

Next, I use this grid to search for points with different signs. For each point on the grid (with the exception of the top row and rightmost column) I compare the sign of the point with the sign of the point to its right and the one above. If the signs are the same, I move on. If they differ, I find a point on the curve between them, follow it, and draw the curve, then return to test the next pair of points.

Figure 12b shows the grid marked with lines that join points of different sign.

I trap the curve with binary subdivision, recursively halving the input interval to always contain a point P where $f(P) = 0$. The recursion stops when the interval is too short or the midpoint is nearly zero. I use a minimum length of $1/kS$, where S is the largest side of the display grid (in these pictures, $S = 400$), and a tolerance ε (so I stop when $|f(P)| < \varepsilon$). For plenty of functions you can fly by the seat of your pants, setting these around $k = 4$ and $\varepsilon = 1 \times 10^{-6}$, and all is well. You can be less conservative and draw your pictures quicker if you know something about the function being plotted. Most of the functions in these figures are pretty smooth, and these values worked fine. These two values k and ε are the first two of four numbers that control the algorithm's precision.

Returning to the job of drawing the curve, the binary searcher returns a starting point S on or near the curve. Hey, one point! Now, if we only had a second point, we could draw a line (Euclid really did have this all figured out).

I assume that if we stand on the point S and look around, we would see two branches of the curve leaving the point, in opposite directions. Of course, they could turn around really fast, but when we face one branch, the other is at our back. This assumption can fail—for example, if S is a cusp. I'll get back to this later. But usually it is true. To keep the bookkeeping simple, I handle the two branches independently.

To find a branch, I search a circle around S, looking for points on the curve. This circle is specified by the other two numbers that set the accuracy of the algorithm: the radius r of the circle and the number of samples n taken around it. The radius of the circle controls the length of the little line segments that make up the

curve. Adding more steps around the circle lets us follow wigglier curves.

So I sample the function at n points on the circle and look for adjacent points of different sign. Figure 13 shows a curve passing through S. The two-branch assumption above says that we'll find two pairs of adjacent points with different signs. Each of these pairs surrounds a point on one of the branches.

I take the two pairs in turn, following (and drawing) the curve passing through one and then repeating the process for the other. If the curve is a circle (as in Figure 1), I'll end up following the curve twice, once clockwise and once counterclockwise. It wastes some time, but there's no other harm done.

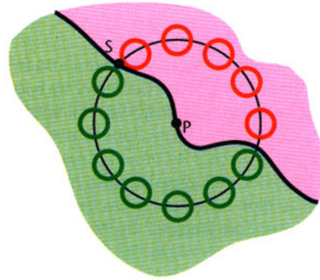
Given a point S and one pair of points on the circle, I again hand the pair of points to the binary subdivision routine. It gives me back a new point P on the line between the points and on the curve. That's the second point! I immediately draw the segment SP with a thick yellow line.

Now it's time to follow the curve. I can repeat exactly the same circle-searching procedure around the point P, finding two pairs of points that contain the curve. One of those pairs contains the branch we just came in on, where the other pair holds the new branch. We can try to determine either the old branch or the new one. We can find the old branch by finding the one that contains S; then the other branch is the new one. That works fine, but we can make the algorithm a bit more robust without any more cost by doing this a bit differently.

Suppose that the curve crosses over itself at P, like at the middle of a figure eight. Then four branches will come out of P. If we find the one containing S, we still don't know which of the remaining three to take. But suppose we use the pair furthest from S. Then we'll always head out in the direction opposite the way we entered, and as long as the two intersecting lines aren't nearly parallel, we'll follow each one just fine. So to find a new point, I search around P for pairs that contain the curve, then pick the pair farthest from S. To test for a pair's distance from S, I find the distance from S to the pair's midpoint. To determine this distance I use the conventional Euclidean metric, though I don't bother with the square root. (This is a standard trick, which works because I am only looking for the biggest distance—the square-root function doesn't change that.)

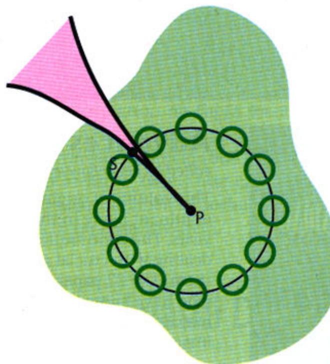
So I draw another thick yellow line from P to the new point. The new point becomes P, the old point becomes S, and I repeat the procedure, pushing forward one circle-radius at a time. When I'm done following the first branch, I follow the other branch, then return to the coarse grid to find another starting point.

Four criteria determine when I'm done following a branch. First, if the branch goes off-screen, I stop, figuring that even if the curve comes back on-screen, I'll catch that new piece from another starting point. Second, if the branch closes itself, I stop. To determine that, I save the very first point S. As I follow the curve, I check the distance of each new point against this original point. If I get within a pixel, I join the gap and stop following the curve. This test only cuts in after I've already drawn 10 segments, so I don't accidentally stop as soon as I've started. Third, I have an arbitrary upper limit on the number of steps I take on a branch; I used 5,000 for this column. This guarantees the program doesn't get stuck in an infinite loop. This could arise, for example, between two narrow cusps—the tracker could just Ping-Pong between them forever unless otherwise stopped. Finally, I stop if for any reason I can't find a new point to move to.



13 Pushing forward. The previous point is marked *S*; the current point is *P*. Two pairs of points on the circle around *P* trap the curve, giving two points on the curve. We would choose the one on the right, since it's farthest from *S*.

That's it! It's a bit wasteful (since most branches are drawn a few times), but it's robust enough for playing around with the sorts of functions in this column. This simple algorithm also runs pretty quickly—all the grayscale figures were created by my first-version program in interpreted Basic in less than 15 seconds each on a Pentium 90-MHz machine. The compiled C code ran much faster.



14 A cusp at *P* results in both arcs leaving the circle through the same pair of points.

Not exactly bulletproof

This algorithm is hardly bulletproof. I mentioned earlier that cusps could be a problem. In fact, any tight U-turn poses a problem for this algorithm, whether it's pointed or just a sharp turn. Suppose that we're at a point P on (or near) a cusp. As we search the circle around P, we find that there aren't any pairs of points with differing sign. Figure 14 shows the problem.

In my implementation, this caused the program to stop because of criterion 4 from above—there simply wasn't anywhere to go. You can actually handle this problem with another subdivision step. Find the pair of

Further Reading

You'll find an elementary introduction to taxicab metrics in *Taxicab Geometry* by Eugene F. Krause (Dover Publications, 1986). If you're looking for interesting functions to play with, a great starting place is *A Catalog of Special Plane Curves* by J. Dennis Lawrence (also from Dover, 1972). The blob function comes from "Data Structures for Soft Objects" by Wyvill, McPheeters, and Wyvill, in *The Visual Computer*, Vol. 2, No. 4, April 1986. You can find some discussion of contour tracking in *Numerical Continuation Methods, an Introduction* by E. Allgower and K. Georg (Springer-Verlag, 1990), and "Automatic Contour Map" by G. Cottafava and G. Le Moli (*CACM*, Vol. 12, No. 7, 1969). One way of avoiding the problems of the coarse sampling grid is to use interval analysis, as discussed in "Interval Analysis for Computer Graphics" by John Snyder in *Computer Graphics* (Proc. Siggraph, Vol. 26, No. 2, July 1992).

points that includes *S* and subdivide that interval. You'll need to follow all the branches of the tree, but sooner or later you should find a point that has a different sign from the others, and this will give you a pair of intervals. Pick the one that doesn't contain *S*, and that will contain the new branch.

Lots of other improvements could make this a more general algorithm. One of the pleasures of hacking up your own routine is that you can design it to work perfectly well for your particular needs, but making that specialized tool more general, more efficient, and so on is also fun. A few starting places for making this program better follow.

First, the coarse sampling grid can miss small features. This is a standard sampling problem, but because you draw the grayscale version first, you can use information from that to help guide a smarter search strategy. Second, big flat basins are something of a problem; if a whole big region has the value 0, the tracking routine can wander around erratically. Third, you could run some numerical analysis routines on the function being plotted to gather initial estimates and values for the four accuracy controls described above.

Different measures of distance can lead toward all kinds of non-Euclidean geometries. I find the connection between the math of the space and the aesthetics of the imagery captivating. Playing around with different metrics and functions is a lot of fun, and after a while I find my intuition gets pretty good at predicting what's going to happen. Try it for yourself and see. ■