Andrew Glassner's Notebook

http://www.glassner.com

Fill 'Er Up!

Andrew Glassner

One of the most important and useful ideas in 2D computer graphics is the *fill*. Simple in concept but powerful in practice, filling is so basic a technique that there's a fill tool in just about every 2D drawing system sold today. Probably one of the first important uses of fills was to color in 2D cartoon characters, as shown in Figure 1. But that's just the tip of the fill iceberg. In this column I'll talk about some fill techniques that I've cooked up over time to solve different jobs.

Before we get started, a little history will help set the stage. It's hard to know who wrote the first fill routine, but it may have been Dick Shoup, who implemented a fill technique for his Superpaint system in the mid 1970s. The next big step was Alvy Ray Smith's 24-bit flood fill algorithm, which also included tint fill, discussed below. Change came fast after that point, culminating in the sophisticated filling tools we have today.

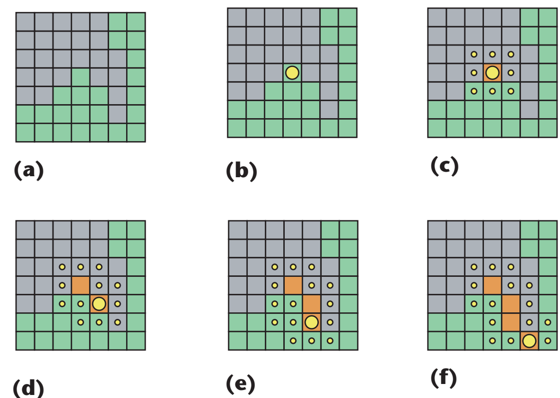
The no-frill fill

The most basic fill algorithm is the *8-neighbor fill*. You start it off by providing a starting pixel (called the *seed*) and a *fill color*. The algorithm first looks at the color under the seed—let's call this the *target color*. The idea is to spread out from the seed, finding all contiguous pixels that share the target color, and rewriting them with the fill color. The 8-neighbor fill recolors the seed and then looks at the eight pixels that surround the seed, as in Figure 2. Each pixel falls into one of three categories, depending on its color. If it's the target color, I call it a *target pixel*. If it's already the fill color, I call it a *filled pixel*. Every other pixel blocks the spread of the fill, so I call it a *blocking pixel*.

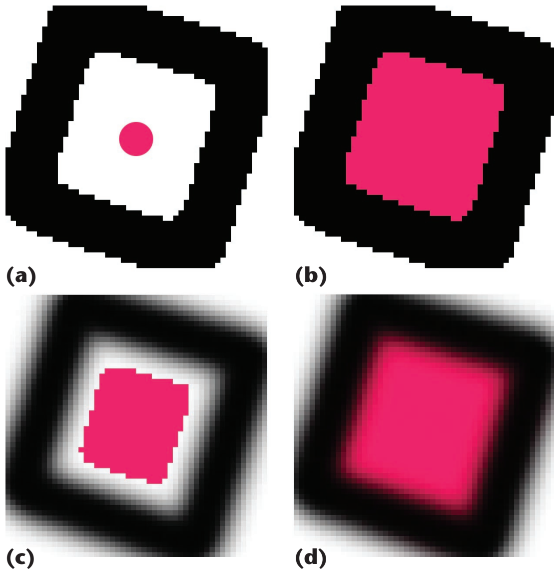
Any time the 8-neighbor fill finds a target color pixel, the algorithm recolors it and then examines its eight neighbors. You can write a recursive program to implement this algorithm. In my opinion, it's actually one of the nicest examples of recursion in computer graphics. The recursive algorithm is short, simple, and easy to program. Unfortunately, it's also very slow when compared to other approaches.

The basic fill has a number of important variations. The most important is the *tint fill*. The algorithm I just described replaces target pixels with the fill color, and that's fine if you're filling in something with hard edges, as in Figures 3a and 3b. But take a look at Figure 3c, where the black lines have been drawn with an antialiased stroke. If we only fill pixels with the target color, we get a little halo around the

1 The classic flood-fill algorithm for cartoon cels. (a) A line drawing. (b) The color scheme indicated by dots. (c) The result of filling each region.

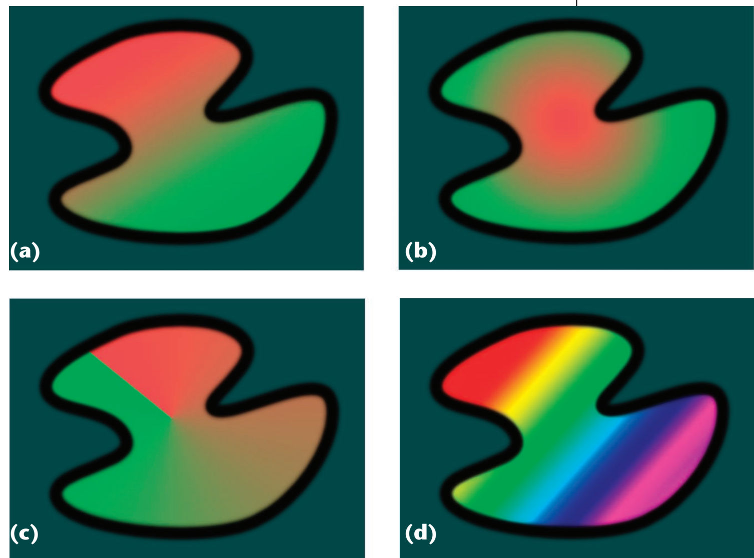


2 A few steps in the 8-neighbor fill algorithm. (a) The starting image. (b) The initial seed. (c) Changing the color under the seed pixel, and identifying the eight neighbors. (d) In this implementation, the algorithm first tries to move to the southeast. It finds another pixel of the same color as the original seed, so it recolors that pixel and marks the new neighbors for later examination. (e) To the southeast is a blocked pixel, so the algorithm looks to the south and moves there. (f) Moving southeast is now available again.



3 Flood fills. (a) The original region and the fill color. (b) The result. (c) The result when the black lines have been antialiased: only fully white pixels are filled. (d) Tint fill blends into the black lines, respecting the antialiasing.

4 A combination of tint fill and pattern fill.



5 Gradient fills. (a) Linear. (b) Circular. (c) Radial. (d) Multicolor.

lines, which looks bad. What we really want is Figure 3d, where the fill algorithm knows that in this case black is special, and that colors between the target color and black should be replaced by a similarly tinted blend between the fill color and black. A robust implementation will tint-fill as long as it's changing pixels that get blacker and blacker, but stops either when they get to solid black (in the case of a thick line), or when they start lightening up again (in the case of a thin line).

Fill farther

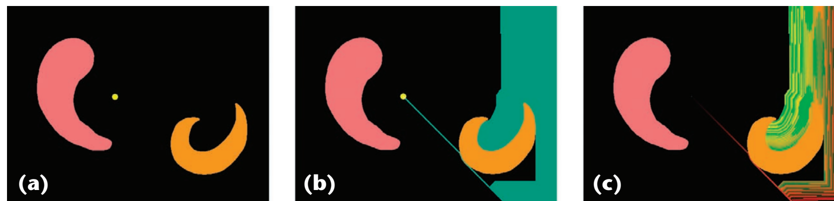
Not too long after the invention of the 8-neighbor fill came the *pattern fill.* Rather than simply replacing the target color with the seed color, why not replace it with a piece of a repeating pattern, as in Figure 4? Suppose you define a rectangular block of pixels as a source pattern. Then nestle the lower left corner of the pattern rectangle into the lower left corner of the image, and stamp out new copies of the pattern to the right and above until the image is covered. Of course, this is all conceptual. But when you're filling, rather than replacing a pixel with the fill color, replace it with the color that would have come from this rubber-stamped pattern.

Fill patterns can vary—the pattern can have its origin anywhere, including the seed point. The replications of the rectangle can include flipping it over at each edge to improve continuity. And you don't need to use a rectangle. Any shape that can tile the plane will do the trick, from triangles and squares to Escher's chameleons.

A few patterns are algorithmic by nature, so it makes sense to do them procedurally. The *gradient fill* is the simplest of these. You begin by defining two points. The system imagines drawing a line between them, and then determines the fill color by finding a perpendicular line back to the input and using a color interpolated from the endpoints. Figure 5 shows the idea. Beyond the endpoints, people typically either *clip* (that is, everything beyond one end of the line is the color at the end of the line), or continue to *interpolate* (which can create some pretty strange colors).

Other gradient fills include the *radial gradient,* where two colors are interpolated in circles from the seed to an outer radius, and the *angle gradient*, where the colors change as though painted around a clock. These are also illustrated in Figure 5. These are only simple examples, and it's fun to cook up other procedural ways to paint color within a fill, such as by using noise, clouds, or checkerboards.

An important variation on all of these colored fills is that you can use more than two colors. For example, in the gradient fill you can define several different colors along the length of the line, and the system will move from one to the next as it fills, as shown in Figure 5d.



6 A test image for our filling algorithms. (a) Two objects and a black background. The yellow disk marks the seed. (b) This 8-neighbor fill starts by heading to the southeast. The result after 30,000 steps. (c) The steps color-coded by their serial numbers. The pixels get greener as the fill goes on.

Most 2D painting programs provide some tools for creating and applying gradient fills of different types that involve several colors.

You can really cut loose with new kinds of things to put into filled regions. For example, you could imagine doing a z-buffer fill. Suppose you render a scene that includes a ground plane going off into the distance. In addition to the picture, you save the z-buffer. Now create a little tile of a flower, point to the ground plane in the image, and fill it with the flower using a z-buffer fill. The algorithm uses the depth at each point to scale the flower and adjust its colors to account for fog. The algorithm could hop from pixel to pixel in the fill area randomly. Each time it draws a flower, it uses the z-buffer to determine visibility, and then it updates the z-buffer to account for the new object. After a while you stop the fill, and you've got a field of flowers.

Of course, at this point we're starting to move from pure fills to image-processing applications constrained to a given region of the image plane. I like the fact that these ideas blend so nicely from one to the next.

Fill disclosure

The job of the filling algorithm can be broken into two pieces—determining the pixels to be affected and then affecting them. These two steps can be repeated and alternated. Since, as we just saw, the second phase can get arbitrarily complex, let's focus on some variations on the job of identifying pixels to be processed.

In the following sections, I'm going to talk about a variety of fill algorithms and run some of them simultaneously. Because the idea of running multiple fills in parallel may seem a little strange, I'll describe how I do it. To run these experiments, I built a fill-algorithm playground. The playground rules are these: each fill is defined by a seed location, fill color, and fill type. Some types of fills also have additional parameters, such as a preferred direction or width. I read a text file that describes the fills for that experiment, build an internal list, and then run through the list over and over.

On each pass through the list, I take the first fill and run it for one step. Typically the routine will determine some pixels to fill, change the color of some or all of them, and then return. Sometimes it also creates or updates some internal state, such as a list of pixels already filled or the size of an expanding region. When that routine returns, I call the next one in line. When I reach the end of the list I come around again, repeating until none of the fills change any pixels.

This means that when there's only one fill, it runs like we're used to. But when there's more than one, the fills effectively race against each other. Whoever gets to a given pixel first gets to color it, and others subsequently see that new color. So I tend to write the routines in a way that doesn't look ahead much. Each time a fill runs, it finds its pixels, modifies them, updates its internal state, and returns. So although they never contradict each other's view of the pixel data, the fills take turns and can get in each other's way, which leads to some interesting results.

8-neighbor fill

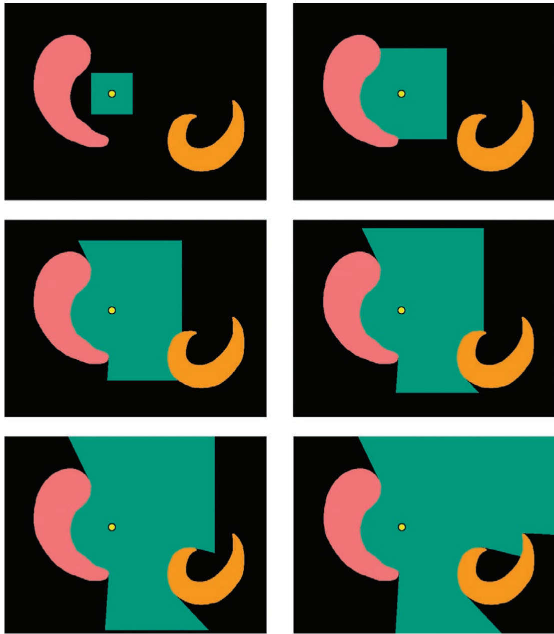
In a naive implementation, the 8-way neighbor fill I previously described spreads out from the seed haphazardly. Figure 6a shows an image we want to fill and its starting seed. Figure 6b shows a typical example of the process stopped in mid-action, after about 30,000 steps. Figure 6c shows this intermediate result in a color-coded form, where the RGB color of each pixel is its 24-bit serial number in order of being filled in. The important part to look at is the green channel—the brighter the green, the higher the serial number.

In this naive implementation, the first direction to be searched was the bottom right. You can see that the program made a beeline to the southeast from the seed, since at each point it checked the bottom right first and made that the top priority for the next pass. When it hit the picture's edge, it started searching in other directions.

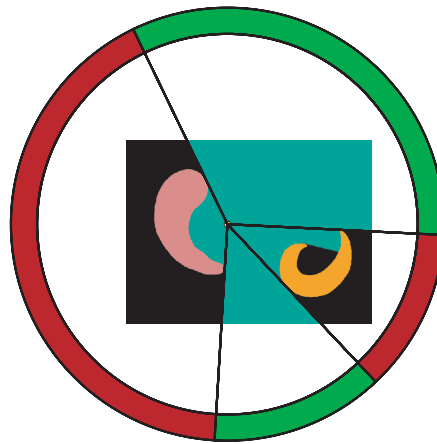
I wondered about this motion and how it might interact with itself during multiple fills. Figure 7a shows the same starting image and three seeds. Figure 7b shows the result after about 30,000 steps (as I mentioned before, the fills take turns executing, one step at a time). Notice that they all dived toward the lower right corner, and then as they spread out, interfered with each other.

Why did the algorithm take 30,000 steps to fill in only a few thousand pixels? Think about the fact that pixels that get pushed onto the stack may not get examined for a while. By the time a pixel pops, more often than not the algorithm has already handled it. Like I said, this is a naive implementation. Coming up with a more efficient approach isn't too difficult, so I'll leave that to you as an exercise.

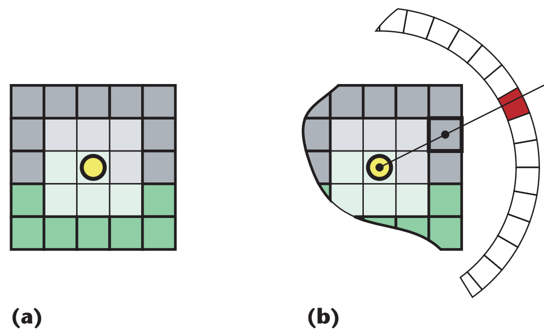
7 Running a trio of 8-neighbor fills. (a) The seed points. (b) The result after 30,000 steps.



8 Progress of the square visibility fill. In this time-lapse series of images, we can see the square expanding. At each step, it only fills in pixels that can be "seen" by the seed.



9 The visibility ring is a data structure centered on the seed point. A green cell indicates that pixels in that direction are still visible as the fill expands; a red cell indicates directions that cells are blocked. This shows the final result after the fill is complete.



(a) (b)

10 The ring is broken up into many cells. (a) The square of pixels currently being examined. (b) When a blocking pixel is located, I conceptually draw a line from the center of the seed through the center of the blocker to determine which ring cell to mark as blocked.

Square visibility fill

Let's try another fill geometry, motivated by a problem I was trying to solve for a research project. I call it a *visibility fill*. Suppose you place a 2D lightbulb in the scene and turn it on. It fills everything it can see. Anything of a color other than the target color acts as a shadow.

We could do this by drawing expanding circles (and I'll get to that) but it's even faster to draw expanding squares. The end result is the same if you're doing only a single fill, but of course if two or more fills interact, we'll see the effect of the square geometry.

Figure 8 shows an example on our test image, as the fill progresses from the seed.

Let's see how to make an efficient implementation. Before starting the fill I create a big ring around the image and chop it up into little pieces. Each piece represents a small solid angle. If that piece holds a 1, then that range of angles is visible from the seed; otherwise it's not. Figure 9 shows the idea. The ring starts out as all 1s.

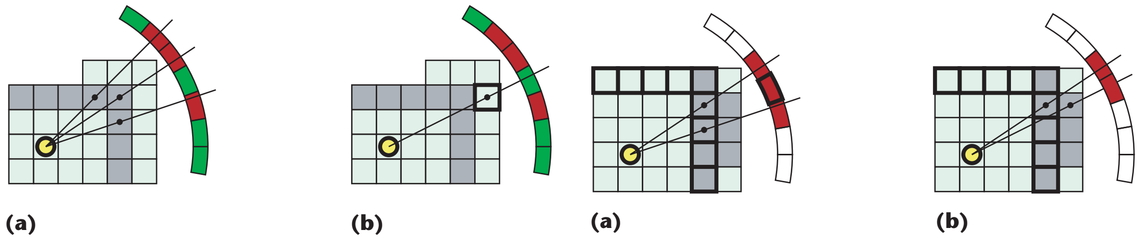
Each time I enter the fill algorithm, I draw a square of the current side length, which starts at 1—that's why I call this the *square visibility fill*. Figure 10a shows an example. The first thing I do is start at one corner and walk clockwise around the square, looking for shadow casters—that is, pixels of any color other than the target color. For each shadow caster I find, I find its associated ring cell, and set it to 0, as in Figure 10b. To find the appropriate cell on the ring, I conceptually draw a line from the center of the seed through the center of the blocker and extend that until it hits the ring.

Once I've gone all the way around looking for blockers, I start over again at the same corner and walk around the ring a second time. Every pixel currently set to the target color is a candidate for being filled in. To determine if it should be filled, I check its visibility. Just as for the blockers, I conceptually draw a line out to the ring and check the status of the visibility cell, as in Figure 10b. If it's a 1, then the pixel is visible, so I change the pixel's color to the new color; if it's a 0 I skip it. Then I move on to the next pixel around the square.

When I've finished going around the square for the second time, I bump the radius by one (so the next time around I'm looking at the next larger square), and return.

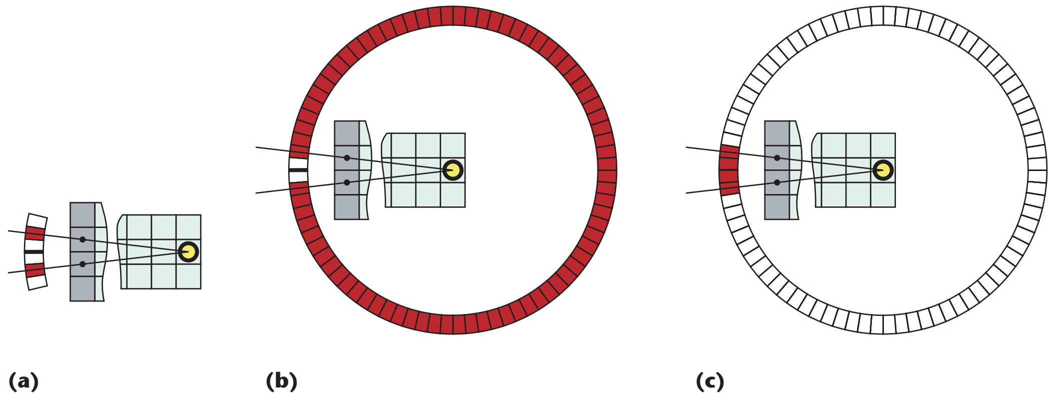
There are a few important "gotchas" to getting this to work right. First, make sure that the ring has enough cells—too many is better than too few. To compute the appropriate number of cells, I add the width of the image to its height and multiply by ten. Multiplying by four would probably be enough, but since more is better, overkill must be terrific.

Second, accurately mark all the blocked angles. Of course I don't actually draw the lines I talked about



11 A failure mode when the ring elements are marked on a per-pixel basis. (a) Three neighboring pixels clearly block this whole region of visibility, but they don't project to continuous visibility cells. (b) A pixel that should be marked as invisible gets misclassified as visible because its cell has not been turned off.

12 Solving the continuity problem. The pixels in heavy outline are part of the square being processed for this seed. (a) Process neighboring pairs of blocking pixels, and fill in all the cells spanned by each pair. (b) Process outward neighbors as well.



13 The crossover problem. (a) A pair of neighboring pixels spans the discontinuity in the arctangent function. (b) If we fill in span from the low angle to the high one, we go around the wrong way. (c) The correct solution.

above through pixel centers and out to the ring. The "ring" itself is nothing but an array of 0s and 1s of the size previously mentioned. To get the index into the ring for a given pixel, I find the angle formed by the line from the seed pixel to the blocker pixel by using the atan2 math library call. The atan2 function is great because it takes the signs of x and y and resolves the correct quadrant for the angle. On my system, atan2 returns a value from $-\pi$ to π. So to get the right cell, I compute the angle, add π, multiply by the number of cells, divide by 2π, and round to the nearest integer.

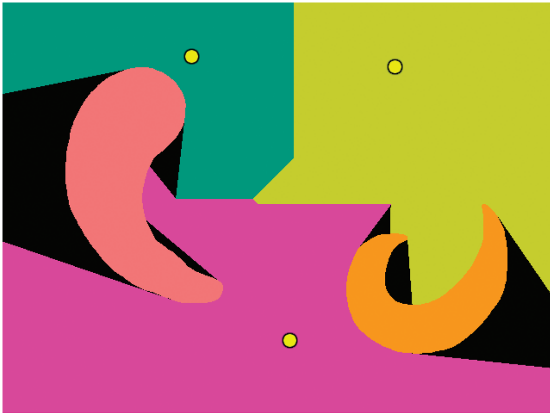
But there's a problem with this approach, shown in Figure 11. When you're near the seed, two adjacent pixels might map to ring entries far from one another. That means that pixels in between these blockers would be unmarked, erroneously identifying that region as visible. Later pixels would be misidentified, as shown in Figure 11b. It seems to me that the easiest solution is to always look at pairs of pixels around the square, rather than single pixels, and mark all the angles between them.

Okay, no problem. I start at the corner and walk around the square, looking for a blocking pixel. Suppose I find one at index n. Then I check square entry $n+1$. If entry $n+1$ is not a blocker, then I just turn off the visibility for the cell corresponding to n, as in Figure 11a. But if $n+1$ is also a blocker, I find the visibility cell for it, and turn off all the visibility cells between the two, as in Figure 12. Then I move to cell $n+1$, check it (and its neighbor, if necessary), and so on around the square. Whew, I'm done!

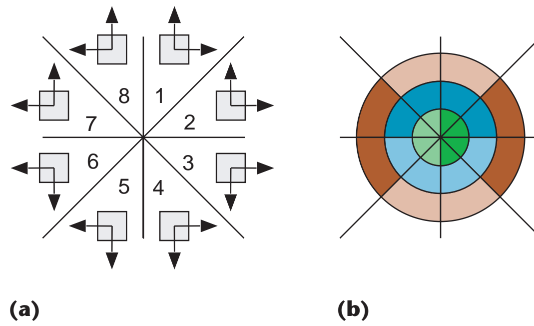
Um, not quite. Remember that atan2 function? Figure 13 shows a problem. The first blocker has an angle θ_1 only a bit smaller than $-\pi$, and the second θ_2 is a little less than $+\pi$. If the array has 80 entries, then these might map to index entries $i_1 = 1$ and $i_2 = 78$, respectively. If we always fill in visibility cells from the smaller angle to the larger, which often is the right thing to do, then in this case we'd go from 1 to 78, as in Figure 13b, which is wrong. The correct answer is in Figure 13c, where I filled in the entries 78-79 and then 0-1. What's the general rule for deciding between these cases? Let's assume that $\theta_1 < \theta_2$. If you think about it a while, you'll come to this conclusion: if $|\theta_2 - \theta_1| > \pi$, then you want to mark as blocked the two ranges $[\theta_2, 2\pi]$ and $[0, \theta_1]$. Otherwise, you want the range $[\theta_1, \theta_2]$. In terms of cells, if there are R visibility cells, then

```
i1 = round(R(θ1 + π)/2π)
i2 = round(R(θ2 + π)/2π)
if (i2 - i1 < R/2)
then mark_range[i1, i2]
else {
  mark_range[i2,R]
  mark_range[0, i1]
}
```

That does the trick. Now our squares expand without leaving behind any holes in the visibility ring.



14 **Running three simultaneous box-visibility fills. Notice that the fills interfere with each other and that their boundaries are all on 45-degree lines.**



15 **Finding neighbors for a given blocker during a circular visibility fill. Given a seed pixel at the center of the graph at (sx,sy) and a blocker at (bx,by), we compute dx = bx – sx and dy = by – sy. (a) The octant in which the vector (dx, dy) lands finds its two neighbors in the directions shown. (b) To determine the octant, compare dx and dy. The color codes break down the eight choices into three tests. Dark green cells show where dx > 0. Dark blue cells show where dy > 0. Dark red cells show where |dx| > |dy|. Each of the eight choices in Figure 15a corresponds to one of the eight values that characterize these three tests. The eight octants can be simplified to four quadrants.**

It's important to remember that we normally need to check two pairs for each blocked pixel. One couples the pixel with the next one around the square. The other check links the square to its counterpart in the next larger square. Corners of the square are special. They have two neighbors in the current square and two more in the next larger one. Figure 12 shows both of these cases.

What if we run a few square visibility fills simultaneously? Figure 14 shows the result. As you might guess, the lines that join neighboring regions are all along 45-degree increments.

Circular visibility fill

In the last section I faked the expanding ring of light around the seed as an expanding square. Why not use a real expanding circle?

The reason for using a square is that it's easy to satisfy two criteria: we get to make sure we hit every pixel as we move outward from the seed, and we don't hit any pixels twice. So we have completeness and efficiency.

Circles are a little trickier. Growing a set of nested circles that never share a pixel and never miss a pixel is probably possible, but the squares were so much easier that I did them first. Then I wondered what circles would look like, so I threw caution to the wind, copied the expanding-square code, and hacked it to do circles.

The hack works this way. I track the side length of the expanding square, as before. I scan all the pixels inside the square and compute their distance to the seed. If that distance is less than or equal to the current radius—but more than 90 percent of that radius—then I check the pixel for filling. Obviously, this isn't as efficient as the expanding square for a variety of reasons.

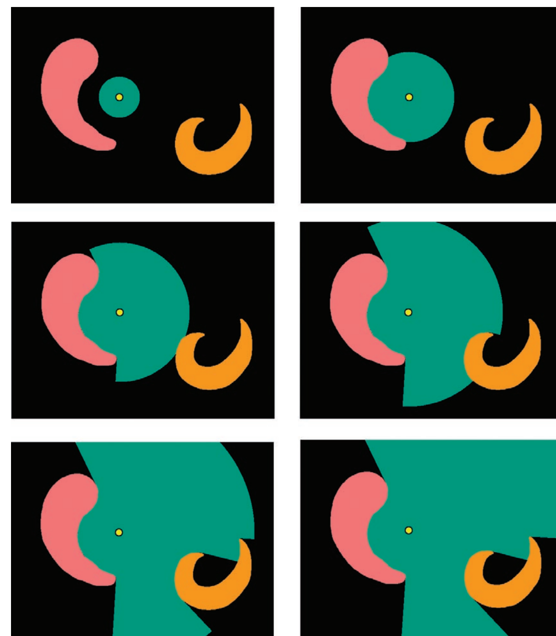
Before I forget, though, I need to point out three details.

First, I don't really compute the distance—I compute the squared distance. When I enter the routine for a given side length of the square, I compute the squared radius of the circle and the squared radius of the 90 percent circle. Then I compare the squared radius of the pixels to these distances. It saves me a square root at every pixel.

Second, finding the two neighbors to handle spans of angles is a little trickier. Because we're scanning left to right and top to bottom, the next pixel to be examined may not be a good choice of neighbor. So I break up the region around the seed into octants, as in Figure 15. In the figure, I indicate the test that identifies which quadrant I'm in and then show which way to move to get the neighbors.

Third, there's no magical reason for picking the 90 percent inner circle. I wanted something that would guarantee that as the circles moved outward, they overlapped enough that I never missed a pixel. I thought 90 percent was a good guess, and it's worked fine in all my tests.

Figure 16 shows the circular versions of Figure 9.



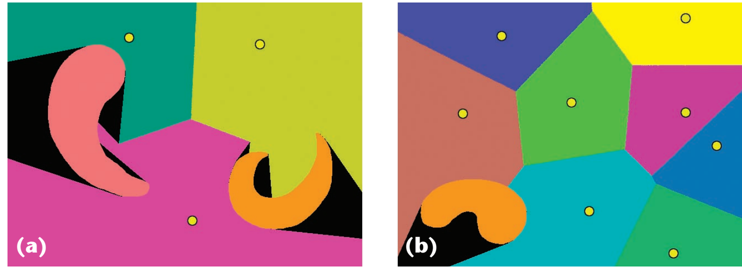
16 **Progress of the circular visibility fill. Note that the final result is the same as for the box.**

Figure 17a shows the result of three simultaneous circular visibility fills. One interesting thing to observe is that the lines between regions are now perpendicular to the line between the respective seeds, and pass through that line's midpoint. In other words, running this fill algorithm in parallel on all the seeds is a way to compute the Voronoi diagram for those seeds. It's accurate to the pixel level but obviously no better than that. Figure 17b shows a more complex example.

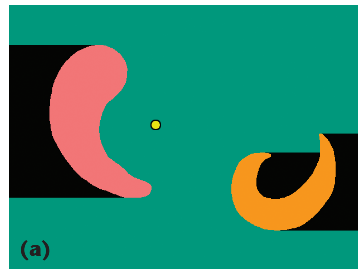
Line sweep fill

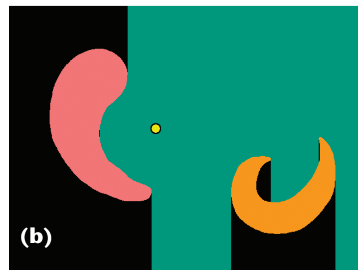
I call another useful fill algorithm the *line sweep*. The idea is to pick a direction, which I'll call the *primary direction*. Starting on the seed, looking forward along the primary direction, search out for pixels to your left and right. Find and replace them as long as they're the target color. When you reach a pixel that isn't the target color, stop. When both sides have been blocked, set a distance counter to 1 and return.

The next time we enter the fill routine, we repeat the process, moving forward from the seed a distance set by the distance counter before

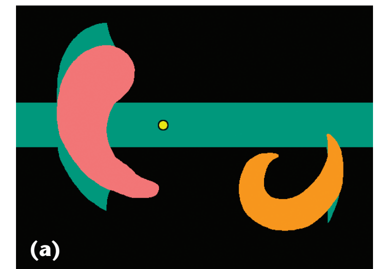


17 Running multiple simultaneous circular visibility fills. Note that the boundaries between regions are perpendicular to the line joining the seeds points. (a) Three fills. (b) Eight fills in a scene with a single orange blob. The result is a pixel-accurate Voronoi diagram.





18 Line sweep fills. (a) A horizontal fill. (b) A vertical fill. (c) Three simultaneous horizontal fills. (d) Three simultaneous vertical fills.



19 River fills. (a) A horizontal fill. (b) A vertical fill. (c) Three simultaneous horizontal fills. Note the interferences between the green and yellow fills past the point where they collide. In the bottom part of the image, the green fill gets under the pink crescent first, then the purple fill, and then the yellow one. (d) Three simultaneous vertical fills.

filling left and right. Then we spread out left and right, filling target pixels as we go. In this way we extend the fill to the edges of the image. Figure 18 shows an example of the line sweep fill in operation.

River fill

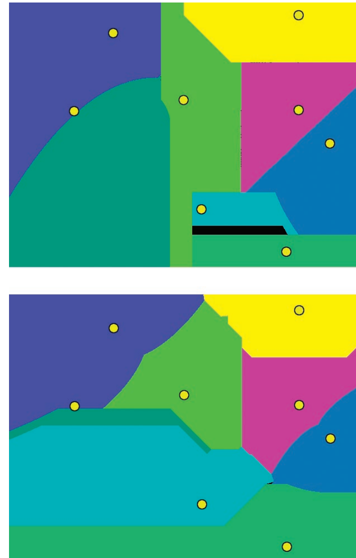
There's a nice little variation on the line sweep fill that I thought of when writing the code. Suppose that during a line sweep fill, when you hit a blocking pixel, you don't stop. Instead, you remember how many pixels you've filled in so far, and you keep heading outward, looking for more target pixels. Each time you find one, you fill it and increment the count of changed pixels. You only stop when you've run off the edge of the image or you've changed a predetermined number of pixels on that pass.

If you do this in a blank image, you'll get a thick band. But if while filling in the ribbon you hit an object, the fill seems to split, with one or both pieces going around the object in order to continue. For mild interruptions, it can look like the ribbon is a river of water meandering around rocks and other objects, so I call this a *river fill.* Figure 19 shows some river fills in our test case images.

Wrapping up

I've shown what happens when two or more fills of the same type are both executing on the same image. What happens if you mix fill types? The results are usually difficult to predict. Figure 20 gives a couple of examples. I don't see any immediate applications for mixing fill types in this way, but that doesn't mean there aren't any!

Before leaving this topic, I'd like to point out an interesting theoretical connection between filling and cellular automata, or CA. The idea of CA is that every pixel in the image is running its own independent little program, which usually combines its internal state with the state of its neighbors to compute a new internal state. The most famous CA is probably the game of Life. You can certainly imagine casting many fill algorithms in terms of CA, and I was planning to write this column in just those terms. But before I stated programming, I remembered that Alvy Ray Smith, one of the inventors of the fill technique, had also done early research in CA. I asked him about this connection, and it turns out that his very first fill algorithm (in about 1976) was written as a cellular automata simulation. It was incredibly slow, so he turned to scanline techniques and assembly language



20 A visual cacophony of eight mixed fills of different, randomly assigned types.

for huge speedups. Computers are fast enough now that I was able to use C++ for this column, but I too stayed clear of a pure CA approach. I think it would be a lot of fun to try out, though.

In this column I've focused on techniques for identifying fill pixels. As I said at the start, what you do with those pixels once you've found them is wide open. One fun approach is to shuffle around the pixels themselves—for example, in the visibility fills, you could rotate all the pixels around the square or circle by an amount based on the distance from the seed, creating a swirling drain effect.

Playing with fill patterns is easy once you've got a little testbed in place. I found myself doodling little fill patterns on restaurant napkins and sometimes seeing possibilities in manhole covers, Venetian blinds, and other objects in the everyday world. This is a ripe area for exploration and playing. Have fun! ■

Acknowledgments

Thanks to Steven Drucker and Alvy Ray Smith for filling me in on the history of this topic.

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