AN ANALYSIS AND ALGORITHM FOR FILLING PROPAGATION

Kenneth P. Fishkin

Brian A. Barsky

Berkeley Computer Graphics Laboratory
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720
U.S.A.

Abstract

Fill algorithms are a common graphics utility used to change the colour of regions in the frame buffer. The *propagation algorithm* is a key component of filling algorithms. The problem of propagation within a fill algorithm is presented and defined, and the difficulties of formalization and comparison are discussed. The previous algorithms are presented and analyzed under a new comparison metric whose validity is confirmed by run-time tests. A new algorithm is developed, and is shown to have better average and worst case behaviour than the others.

Résumé

Les *algorithmes de remplissage* sont souvent utiles en infographie. Ils servent à produire des changements chromatiques de certaines régions dans la mémoire d'image. La *propagation du remplissage* est un aspect important de ces algorithmes. Dans cet article, nous commençons par définir le problème de la propagation. Nous discutons ensuite des difficultés inhérentes à la formalisation de cette notion, ainsi qu'à la comparaison d'algorithmes servant à résoudre ce problème. Nous proposons une métrique simple permettant d'effectuer de telles comparaisons. La validité de cette métrique est confirmée par des tests empiriques. Les algorithmes précédemment proposés sont alors présentés et analysés à la lumière de cette métrique. Un nouvel algorithme est finalement développé, et nous montrons que ce nouvel algorithme se comporte mieux que les précédents en moyenne comme en pire cas.

This work was supported in part by the Semiconductor Research Corporation under grant number 82-11-008, the National Science Foundation under grant number ECS-8204381, and the State of California under a Microelectronics Innovation and Computer Research Opportunities grant.

1. Introduction and problem definition

A *region* is a group of connected pixels in a frame buffer consisting of a set of connected *spans*. A span is a horizontal row of pixels that are all within the region. Furthermore, a span is the *largest* such row; the pixels horizontally adjacent to the span are assumed to be *outside* the region. The region may be *4-connected* ("Manhattan geometry") in which case spans may only be connected vertically, or *8-connected*, in which case spans may be connected across diagonals. We focus on 4-connected filling, as have others. We show in Section that most 4-connected propagation algorithms can be trivially changed into an 8-connected algorithm, if need be.

Every pixel in the region possesses some property P (for example, possessing a certain colour). The region is delimited by a set of *boundary pixels* that do not have P. We assume a Boolean function INSIDE, which returns **true** if and only if the pixel has P. We wish to perform a certain SET operation exactly once upon each pixel in the region (for example, writing a certain colour into the pixel).

Application of the INSIDE function across the frame buffer creates a Boolean matrix. The *propagation algorithm* explores this matrix. Given a seed pixel known to be inside the region, the propagation algorithm finds the region's boundaries. The propagation algorithm calls the SET procedure (exactly once) on each connected INSIDE pixel, where the SET and INSIDE procedures can be varied to achieve different effects (see Fishkin[1] for examples).

The set of pixels that are in the region can be defined inductively:

1) The seed pixel is INSIDE the region.
2) A pixel is in the region if and only if it is INSIDE, and connected to another pixel that is in the region.

Another formalism is to consider each span in the region as a vertex in a graph, and to connect two such vertices with an edge if and only if those two spans are connected. In this case, the propagation algorithm is essentially a graph-traversal algorithm, which finds a connected component from a single vertex. We will show that the propagation algorithms can be easily classified according to which of these two formalisms they use.

The composite process of finding and SETting the pixels in the region is known as *filling*. In this paper, we focus on the propagation method, to the exclusion of the other parts. We assume only that INSIDE and SET exist, and may be called as necessary.

Once a span has been found, the algorithm must explore outwards from it. As per Smith,[7] we define a *shadow* of a span to be some set of pixels connected to the span that are to be explored by the algorithm. A shadow has a key property: any span that lies (wholly or partially) in a shadow is connected to the region. The propagation process consists of pushing shadows from known spans onto a stack, and then later finding the set of spans that contact that shadow, which we term the *spanset*. Figure 1.1 summarizes our notation, and our representational convention. A span may cast as few as one or as many as three shadows.

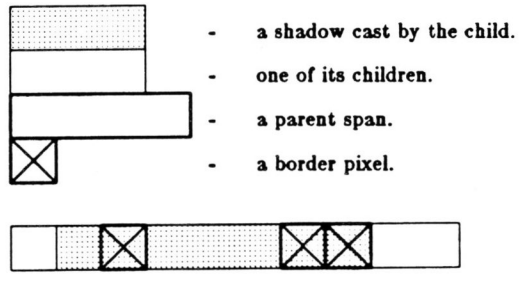

Figure 1.1.
Basic terms and figures to represent them

When a span is created, it occupies a certain topological relationship to its parent span. There are three possible cases.

First, it is possible that the child span does not extend beyond the parent span by more than one pixel on either end (Figure 1.2). Lieberman[3] terms these *S-turns*.

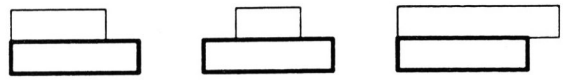

Figure 1.2.
An S-turn: the child span does not overlap the parent by more than one pixel on either end

Secondly, the child could extend beyond the parent span on one end, but not on the other (Figure 1.3). Lieberman[3] terms these *U-turns*.

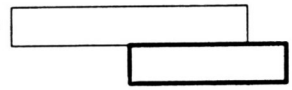

Figure 1.3.
A U-turn: the child partially overlaps the parent

Finally, the child could extend beyond the parent span on both ends (Figure 1.4). We term these *W-turns*.

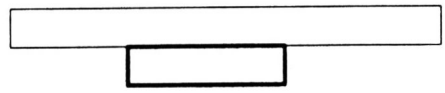

Figure 1.4.
A W-turn: the child wholly overlaps the parent

1.1. 8-connected propagation

Since a span is added to the region if and only if it contacts a shadow, it is easy to convert 4-connected propagation algorithms into 8-connected algorithms. When a shadow is pushed, it consists precisely of those pixels that could extend the region from the parent span. Therefore, if we simply extend the borders of a shadow by one pixel in each direction when we push it on the stack, the 4-connected algorithm becomes 8-connected.

2. Comparing algorithms

There are a number of fast propagation algorithms[2,3,4,6,8,7] extant. Three factors motivate our decision for re-examination:

1) The algorithms are required to solve large problems in real-time.

2) Recently, filling algorithms have been presented[1] that require much more expensive computation on a per-pixel basis.

3) A desire for formalism.

Unfortunately, it is very difficult to compare propagation algorithms, for the following reasons:

1) *Topology.* Algorithm performance depends upon the topology of the region, which is not known in advance.

2) *Seed.* Within a certain topology, performance is also a function of the starting point.

3) *Calls* to the other components. The algorithm makes repeated calls to INSIDE and SET, whose expenses vary with the particular type of filling.[1]

4) *Machine dependence.* The cost and power of the instruction set depend on the particular target machine.

2.1. Comparison metrics

We will compare algorithms based on two metrics:

1) *Space.* Regions may be cyclic. The propagation algorithm should not "rediscover" pixels that have already been filled, since this could lead to infinite looping. Some algorithms keep one bit per pixel as a "visited" bit. Other algorithms don't need this bit; they avoid infinite looping via internal data structures.

2) *Exploration behaviour.* As mentioned previously, a span may have three different topological relationships to its parent. The main difference between the algorithms lies in their differing behaviour in the different cases; they may push different shadows onto the stack and push them in different orders.

We measure the efficiency of the exploration by ρ, the average number of reads per pixel over the region; that is, the total number of pixel reads performed by the propagation algorithm divided by the number of pixels in the region. In addition to those pixels inside the region, those pixels adjacent to the region ("boundary pixels") are also read at least once. We will not count these pixel reads in our comparison, for three reasons: the ratio of boundary to interior pixels is usually very low, the analysis is greatly simplified if they are neglected, and memory requirements can be reduced if the interior ρ achieves 1.0, regardless of the boundary reads.

Since filling is a linear-time problem, we will be comparing constant-factor reductions in ρ. An extra pixel read represents not only a wasted frame buffer access, but indicates wasted control logic. Our contention is that this single number ρ (within the context of the previous assumptions) measures the efficiency of a propagation algorithm; our run-time tests confirm the strength of this metric.

2.2. Other criteria

The algorithms were compared on three other criteria, which tests showed to be non-crucial. We list them for the sake of completeness.

1) *Computational cost, for a given ρ.*

2) *Stack area.*

3) *Instruction set* needed to implement the algorithm. Each of the algorithms can be implemented with only assignments, negations, tests, increments, and decrements.

Table 5.2 shows that the main algorithms were all approximately the same when evaluated on these criteria; we will not mention them for the rest of the paper.

3. The algorithms

There are two schools of propagation algorithms, corresponding to the two formalisms mentioned in Section 1. Considering the region as a connected graph gives rise to algorithms that are *global* and *vertex-based.* These algorithms consider each span as a vertex, and connect two vertices if and only if their corresponding spans are connected.

Second, if the region is considered solely as a Boolean matrix, the algorithms engendered are *local* and *pixel-based.* They pay little or no attention to graph-theoretic properties, considering solely the topology of the current span.

A Taxonomy of Filling Algorithms		
Author	Year	Class
Lieberman	1978	graph
Smith	1979	pixel
Shani	1980	graph
Pavlidis	1981	graph
Smith	1982	pixel
Levoy	1982	pixel
new	1985	pixel

Table 3.1.

3.1. Graph-oriented algorithms

3.1.1. Lieberman

The first published propagation algorithm is that of Lieberman.[3] This graph-oriented algorithm keeps two sorted lists consisting of unexplored edges (shadows) leading up and down, respectively.

This algorithm avoids cycling by referring to the lists of unexplored edges. Intuitively, the unexplored edges represent the border of the current region; if exploration contacts these edges, they represent an "imaginary boundary", and the exploration will retreat. This requires that the list be searched on every pixel (since a vertex (span) may cross more than one edge), that pushes perform insertion into a sorted list, and that edges on the stack be modified *in situ* if spans contact them. Therefore, the algorithm's behaviour depends heavily on the "bushiness" of the region, the density of the region's graph.

Lieberman's algorithm is mainly of historical interest. Shani[6] shows that it is not always correct, and it can be quite slow. However, it does contain five important ideas used by later algorithms:

1) The treatment of the region as a graph, with one vertex per span, and edges between connected spans.
2) Recognition of U-turns.
3) Storage of parental information in a data element.
4) The use of stacked shadows to represent a imaginary boundary enclosing the region.
5) Noting that regions with holes represent the worst case.

3.1.2. Shani

Shani[6] avoids cycling by *explicitly* drawing the imaginary boundaries mentioned above. Boundary lines are drawn temporarily, and then erased.

The algorithm traverses the region's graph, but *only* pursuing edges which go in a certain direction (upwards, say). When no such edges remain, the algorithm reverses direction; only downwards edges are pursued, until *they* are exhausted. This series of back-and-forth waves continues until no unexplored edges remain.

Newly discovered edges are pursued if they lead in the current direction, and *blocked* if they lead in the opposite direction. An edge is blocked by drawing a physical barrier along that edge, on the side of the discovering vertex.

If a vertex is explored, and there is a blocked edge preventing further exploration, then a cycle has been found; the blocked edge is removed from the stack and the current exploring process is terminated. This blocked edge prevented the exploring process from re-entering the blocked vertex, which would have caused an infinite loop.

When all upwards edges have been pursued, the direction is reversed. Any previously blocked edges in the current direction are re-drawn (unblocked) and then pursued. In this manner, all downwards edges are pursued, and all upwards edges are blocked for the next upwards sweep.

Shani uses a *deque*-like structure for his main data structure. Unexplored edges leading in the current direction are pushed on the top of the deque, and blocked edges are pushed on the bottom of the deque. This technique ensures that direction will be changed only when all edges in the current direction have been exhausted. This deque-like structure combines the two sorted lists of Lieberman, by pushing onto the two different ends. This deque-like structure is not a "pure" deque, which would only allow removal from the ends. When a cycle is discovered, the blocking edge is removed from the structure, wherever it may be.

This algorithm is qualitatively different from the pixel-based algorithms, and cannot be compared solely on the basis of ρ. First, the algorithm doesn't need a bit per pixel, a decided advantage. However, the algorithm pays for this by (1) blocking edges, and (2) removing blocked edges that are found to form a cycle. This latter step requires that the deque be searched after each span, to see if that span was claimed by a blocked edge. Depending on the region, this step can be very expensive.

Shani never explicitly describes the behaviour of his algorithm when confronted with U and W turns. However, his paper contains a figure that shows the algorithm performing optimally on a U-turn, and his algorithm requires that non-cycle-causing spans not be revisited. For these two reasons, our implementation of his algorithm uses both U and W turn optimization.

The algorithm visits every non-cycle-causing span once, and every potentially cycle-causing span twice. This leads to a worst-case ρ of 1.5, but this is extremely rare; our test regions had an average-case ρ of 1.09. This is almost identical to (but slightly higher than) the ρ of our algorithm, the only other one that optimizes both U and W turns. This is because Shani's algorithm only revisits pixels that *could* form a cycle, and ours only revisits those that *do* form a cycle.

This algorithm, unfortunately, is not extendible to 8-connected propagation. When a blocked edge is found, the blocking line must be drawn on the side of the blocking vertex, not the blocked vertex. Otherwise, a span could (wholly or partially) traverse any number of blocking edges; the deque would have to be consulted not on every span, but every *pixel*, and shadows have to be modified in-place.

Drawing the line on the side of the blocking vertex is only correct in 4-connected propagation; it is only in this case that we are guaranteed that the blocking edge overlays a set of pixels in the blocking vertex that are all inside the region. For example, consider a to-be-blocked span that lies diagonally and one pixel away from the blocking span. In this case, no pixels in the blocking span will both (1) block the cycle and (2) correctly re-discover the blocked vertex when direction is changed.

3.1.3. Pavlidis

Pavlidis' algorithm[4] notes that the graph formed by the region is implicitly defined by a graph that defines the *border* of the region. Furthermore, this border graph is usually much sparser than the interior graph. His algorithm, then, explores not the interior graph but rather the border graph.

His algorithm uses a stack as the main data structure, containing the address of a border span, and its direction with respect to its parent.

If the interior and border have I and B pixels, respectively, Pavlidis' algorithm makes $I + 3B$ pixel visits, for a ρ of $(I+2B)/I$. In the worst case, the ratio of B to I can be arbitrarily large; the algorithm has worst-case ρ approaching infinity.

We did not implement Pavlidis' algorithm for testing because of two disadvantages:

1) The algorithm assumes that the border of the region has a distinct colour, distinct even from the surrounding background. If this is not the case (e.g. when the picture has only two colors), then the algorithms' behavior is not defined.
2) Pavlidis' algorithm is unique among the graph-oriented algorithms in reserving a bit per pixel.

3.2. Pixel-based algorithms

3.2.1. Smith0

Smith has published two propagation algorithms.[7,8] His first,[7] which we will term "Smith0", is the only algorithm that keeps no parental information of any kind on the stack.

The endpoints of the shadow and the values of the parent span are kept as program variables rather than as stack data. This means that when the algorithm switches direction the algorithm has no recourse to parental information.

This algorithm detects S-turns by these program variables (except for immediately after a change of direction), but does not detect U or W-turns. Therefore, the algorithm will always push one shadow continuing in the same direction, and will push one shadow in the opposite direction of the same size as the child in the case of a U or W turn, (see Figure 3.1).

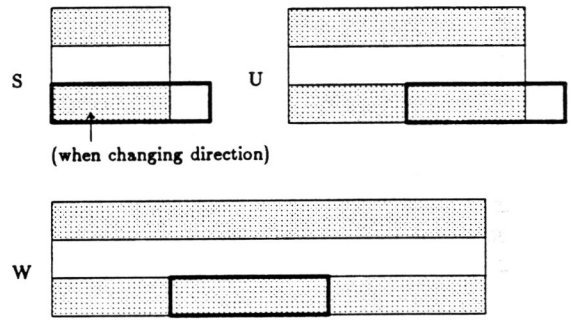

Figure 3.1.
The Smith0 algorithm acting upon S (left), U (right), and W (bottom) turns

On both U and W turns, the algorithm will read pixels at least twice; those pixels that are in the parent span and also in shadow. This can also happen on S turns, when the algorithm changes direction. Since spans are pushed in both directions, this algorithm can achieve a worst-case ρ of 3, as noted by Pavlidis.[4]

3.2.2. Smith

At the SIGGRAPH '82 2-D Animation tutorial, Smith presented an improvement on his first algorithm.[8] It is this algorithm that we will refer to as "Smith's algorithm" for the rest of the paper.

Smith's algorithm keeps the endpoints and y coordinate of the span explicitly on the stack, and therefore avoids the S-turn anomaly noted above after a switch of direction.

However, the algorithm still performs the same in the case of a U or W turn; two shadows are pushed, of the same size of the child, in either direction (see Figure 3.2).

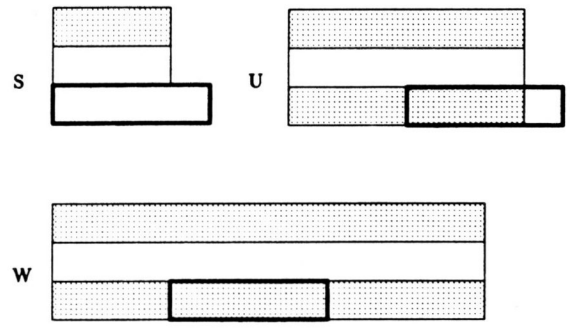

Figure 3.2.
Smith's algorithm acting upon S, U, and W turns

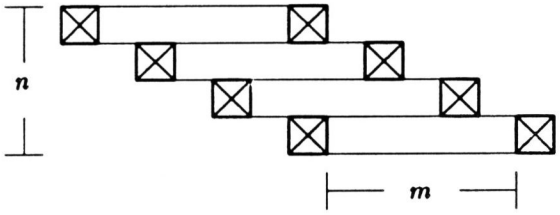

Figure 3.3.
Worst case for Smith's algorithm

The worst case is a region that consists exclusively of children that barely exceed their parents on one end, as shown in Figure 3.3. In this case, we have n rows of m pixels each. Each row is a U-turn with relation to its parent. On the top and bottom rows, pixels will be read only once. On the other rows, $m-2$ pixels will be re-read by each of the adjoining spans. This leads to a worst-case ρ of

$$\rho = \frac{2m+(n-2)[3(m-2)+2]}{mn} = 3-\frac{4}{n}-\frac{4}{m}+\frac{8}{mn}$$

As m and n approach infinity, this approaches a worst-case ρ of 3. In our tests, the algorithm had an average-case ρ of 2.02.

3.2.3. Levoy

At the same tutorial, Levoy presented a propagation algorithm[2] which, though similar to Smith's, makes more use of the parental information.

The endpoints of the shadow, when popped, are compared to the endpoints of the span that pushed it. At this time, S and U turns are detected. If the shadow represents the downward side of an S turn, it is discarded, and if it represents the downward side of a U turn its endpoints are shaved.

Levoy's algorithm delays stack pushes as long as possible; as shown in Table 5.2, it tends to have the lowest stack heights of any algorithm.

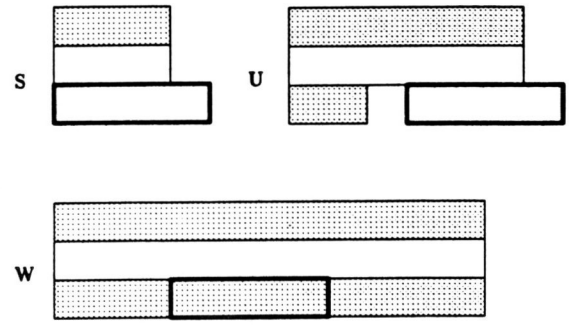

Figure 3.4.
Levoy's algorithm acting on S, U, and W turns

The behaviour of Levoy's algorithm is summarized in Figure 3.4. Since it does not detect W turns, the worst case arises when the region consists entirely of W turns, with edges as small as possible (Figure 3.5). If the seed point is at the apex of the triangle, then the algorithm will read every pixel twice except for those in the top row. If the triangle has n rows, it consists of $2n^2-n$ pixels, $4n-3$ of which are in the top row. Then

$$\rho = \frac{2(2n^2-n)-(4n-3)}{2n^2-n}$$

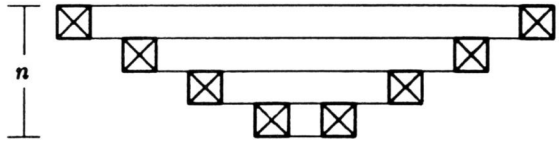

Figure 3.5.
Worst case for Levoy's algorithm

As n approaches infinity, ρ approaches a worst-case of 2. In our tests, Levoy's algorithm had an average ρ of 1.53. Levoy's algorithm shows substantially better worst and average case behaviour than Smith's, solely due to the detection of U turns.

3.2.4. The new algorithm

This section presents a new algorithm for fill propagation. Like Levoy's and Shani's, it keeps full parental information on the stack. Only the direction that the parent came from is kept, rather than its y value; this simplifies the logic considerably.

The algorithm checks each span against its parent for the S, U, or W configuration. The shadows pushed are the largest set of pixels that could extend the region and *do not* contact the parent span. The algorithm's performance is most easily shown visually by Figure 3.6.

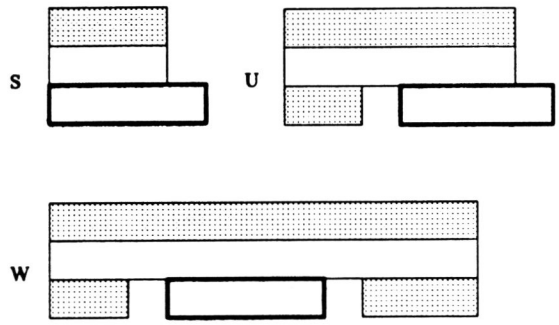

Figure 3.6.
Our algorithm acting on S, U, and W turns

In any serially-implemented recursive algorithm, recursive calls must be evaluated in some order. In the previous algorithms, the order is arbitrary. Our algorithm uses the well-known heuristic of "do the smallest piece first".[5] In practice, shadows pushed in the opposite direction for U and W turns are almost always very small, corresponding to local bumps in the region; therefore, we always push those spans last.

The new algorithm is the only one with a non-arbitrary stacking order. Otherwise, it is very similar to Levoy's, except that ours detects W-turns. Our tests show that these two minor improvements reduce run-time by roughly 25% on our test data.

The new algorithm is also very similar to Shani's, except that ours needs a bit/pixel but does not require a data structure search, reducing run-time by roughly 75% on our test data.

Even though our algorithm pushes as many as three shadows per span, as opposed to the two of most other algorithms, our stack height is virtually identical to theirs, due to this simple heuristic. Of course, there could be narrow spans leading into arbitrarily complex regions, but this rarely happens in practice.

If the region has a hole, the algorithm will re-discover some of those pixels that formed the cycle, as shown in Figure 3.7.

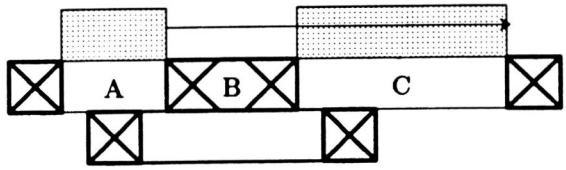

Figure 3.7.
Our worst case; a hole (region B)

If the widths of the left span, hole, and right span are A, B, and C, respectively, the child of the left span will bleed over on top of the hole and revisit the pixels which are the responsibility of the right span. This results in C pixels being read twice.

The region has A+B+C pixels in the top row, A+C in the middle row, and at least B+2 in the bottom row, giving the region 2A + 2B + 2C + 2 pixels. All pixels are visited once except for the C cycle-forming pixels on the top right, which are visited twice. Therefore, the worst-case ρ is

$$\rho = \frac{2A+2B+3C+2}{2A+2B+2C+2}$$

When C approaches infinity and A and B are minimized, ρ approaches its worst-case value of 1.5.

3.2.4.1. Behaviour on simply connected regions

In this section, we show that the algorithm achieves ρ of 1.0 on regions without holes (simply connected regions).

We need only consider two shadows whose spansets overlap. Those shadows must lie on the same line, by the definition of spanset. Suppose, then, that we have some overlap of spansets. There are four possible cases, depending on whether or not the shadows overlap and whether or not the parent spans are on the same line. We only prove the result for one case: the others follow similarly.

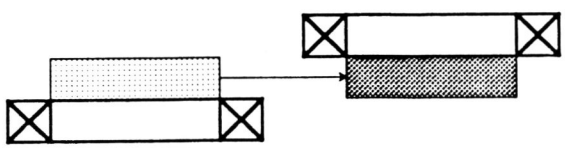

Figure 3.8.

Case 1: The parent spans lie on different lines, and the shadows don't overlap (Figure 3.8).

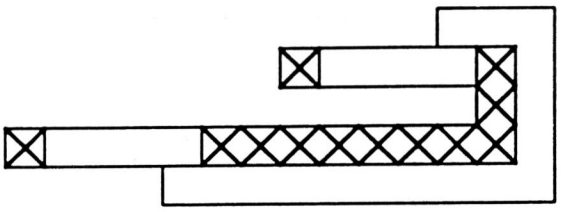

Figure 3.9.

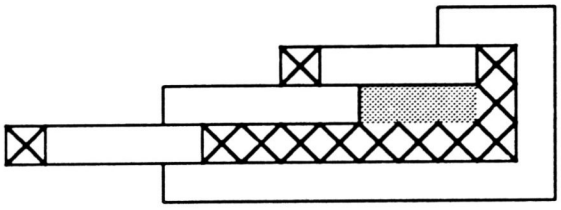

Figure 3.10.

At any point in the algorithm, the set of pixels that have been SET is 4-connected. Furthermore, we defined the shadows for our algorithm such that they don't overlap any previous parents. Therefore, the parent spans must be 4-connected by a path that does not touch either of the shadows (Figure 3.9). Then, we fill in the current spanset, a spanset which, by assumption, contacts both parent spans. This forces a hole either to the right of the upper span, or to the right of the lower span (Figure 3.10). Again, there are other symmetrical cases.

The algorithm achieves the optimal ρ if the arbitrary region turns out to be simply connected. If we know *in advance* that the region is simply connected, is our algorithm optimal?

Unfortunately, no. Consider any region whose border has area proportional to the square root of the area of the entire region (a square, for example). If the border of the region is traversed, and the region is known not to have holes, then the entire interior must be entirely within the region (see Figure 3.11). Since we only visit the border, this process gives us a ρ of $O(\sqrt{n})$, where n is the area of the region.

Therefore, although our algorithm achieves the lower bound for a subset of the possible regions, it is not necessarily optimal if we know in advance that we are dealing with an element of that subset.

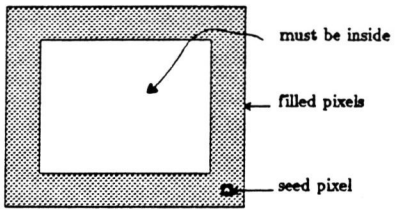

Figure 3.11.
After $\sqrt{n}$ visits, we can fill n pixels

4. An analogy

All the algorithms explore (either explicitly or implicitly) the graph defined by the topology of the region.

Both Smith's and Levoy's algorithms proceed in some fixed direction until reaching a dead end, then back up to the last node where a choice was possible, and pursue that. This recalls *depth*-first graph traversal.

Our algorithm always changes direction if possible. Branches are filled before the main trunk, as in *breadth*-first graph traversal.

Shani's algorithm proceeds in one direction as long as there are unexplored arcs anywhere in the region in that direction, repeating the process in the other direction, This exploration by back-and-forth waves recalls network flow or *spanning tree* algorithms.

5. Comparison of the three algorithms

Four of the algorithms were implemented in C under 4.2 BSD UNIX on a VAX-11/750, using an Adage/Ikonas RDS3000 frame buffer. The **gprof** command provided run-time profiling. The "CPU time" row in Table 5.2 represents the relative total CPU time to fill each region, using *boundary fills*'[7] INSIDE and SET procedures.

One of the 21 regions was a grid. This region represents a maximally dense graph (the greatest possible number of holes), and was created specifically to demonstrate worst-case behaviour. Each of the algorithms fared the worst by far on this region, especially Shani's; the list of blocked spans was so long that his algorithm's behaviour was extremely poor. Without this region, Shani's algorithm performed quite well; it's relative CPU time dropped from 2.17 times Smith's to 0.79 times.

An Overall Comparison of the Filling Algorithms							
	Lieberman	Smith0	Shani	Pavlidis	Smith	Levoy	new
year	1978	1979	1980	1981	1982	1982	1985
taxonomy	graph	pixel	graph	graph	pixel	pixel	pixel
data structure	sorted list	stack	deque	stack	stack	stack	stack
element width	2 int	2 int	5 int, 3 Bool.	2 int 1 Bool.	3 int	5 int	5 int, 1 Bool.
space requirement?	No	Yes	No	Yes	Yes	Yes	Yes
detects S turns?	Yes	usually	Yes	No	Yes	Yes	Yes
detects U turns?	Yes	No	Yes	No	No	Yes	Yes
detects W turns?	No	No	Yes	No	No	No	Yes
gradient fills?	No	Yes	No	No	Yes	Yes	No
8-connected?	Yes	Yes	No	Yes	Yes	Yes	Yes
pushes/span	1-2	1-2	1-3	NA	1-2	1-2	1-3
ρ, worst case	2	3	1.5	∞	3	2	1.5
av. ρ, test cases	--	--	1.09	--	2.02	1.55	1.05
relative CPU time, test cases	--	--	217.2	--	100.0	83.5	60.2

Table 5.1.

Average per-region statistics, 4 algorithms, 21 regions				
	Algorithm			
	Shani	Smith	Levoy	new
pushes	821.3	724.9	618.0	780.0
max height, stack/deque	198.1	192.3	98.7	102.7
max area (bytes), stack/deque*	2575.3	1153.8	987.0	1129.7
ρ	1.09	2.02	1.53	1.05
incr	25.18	2.11	1.99	1.20
decr	23.07	0.24	0.30	0.20
negate	0.01	0.00	1.05	0.04
test	94.69	2.20	4.23	3.76
assign	1.22	0.48	0.76	0.27
CPU time	217.2	100.0	83.5	60.2

Table 5.2.

* assuming 2 bytes/integer, 1 byte/Boolean

6. Conclusion

A number of algorithms to fill regions were presented and compared under a new metric. A new algorithm was developed with better average and worst-case behaviour under this metric. The appendix gives code to implement the algorithm.

7. Acknowledgements

The authors wish to thank Marc Levoy of the University of North Carolina at Chapel Hill for his many helpful comments and discussions, and the reviewer for suggesting re-examination of the graph-oriented algorithms.

References

1. Kenneth P. Fishkin and Brian A. Barsky, "A Family of New Algorithms for Soft Filling," pp. 235-244 in *SIGGRAPH '84 Conference Proceedings*, ACM,Minneapolis(July 23-27, 1984). Extended abstract in *Proceedings of Graphics Interface '84*, Ottawa (28 May - 1 June 1984), pp. 181-185.
2. Marc S. Levoy, *Area Flooding Algorithms*, Report, Hanna-Barbera Productions (June, 1981). Presented at SIGGRAPH '82 2-D Animation Tutorial.
3. Henry Lieberman, "How To Color in a Coloring Book," pp. 111-116 in *SIGGRAPH '78 Conference Proceedings*, ACM,Atlanta(1978).
4. Pavlidis, Theo, "Contour Filling in Raster Graphics," pp. 29-36 in *SIGGRAPH '81 Conference Proceedings*, ACM,Dallas
5. Robert Sedgewick, *Algorithms*, Addison-Wesley.
6. Uri Shani, "Filling Regions in Binary Raster Images: A Graph-Theoretic Approach," pp. 321-327 in *SIGGRAPH '80 Conference Proceedings*, ACM,Seattle(July, 1980).
7. Alvy Ray Smith, "Tint Fill," pp. 276-283 in *SIGGRAPH '79 Conference Proceedings*, ACM,Chicago(August, 1979). Also Technical Memo No. 6, New York Institute of Technology.
8. Alvy Ray Smith, *Fill Tutorial Notes*, Report No. 40, LucasFilm (April 27, 1982). Presented at SIGGRAPH '82 2-D Animation Tutorial.

Appendix

Here we list C source for the key portion of our algorithm, with pseudo-code provided for the other parts.

```
struct {
int   MyLx, MyRx;    /* endpoints of this shadow*/
int   DadLx, DadRx;  /* and parent span     */
int   Myy;           /* my shadows y coord  */
TWO_VAL Mydir;       /* only holds values of +1 or -1 */
} Stack[STACK_HEIGHT];
macro PUSH(a,b,c,dd,e,f)  {
   push a shadow from (a) to (b), inclusive,
   on line (e), going in direction (f)
   from the parent span of of [(c)..(dd)].
}

/* pop top of stack into local variables */
macro POP()   {
   pop the top shadow into local variables
   lx,rx,y,direction, DadLx, DadRx
}

/* stack a shadow on the stack. The current span is
[lx..rx] on line y, the parent is [DadLx..DadRx],
and the current direction is (direction) */
macro STACK(direction,DadLx,DadRx,lx,rx,y)    {
/* store the *shoulders* of the span,
to simplify testing 3,6 lines down */
   pushrx = rx + 1; pushlx = lx - 1;
   PUSH(lx,rx,pushlx,pushrx,y+direction,direction);
   /* U turn to the right */
   if (rx > DadRx)
      PUSH(DadRx + 1,rx,pushlx,pushrx,
         y-direction,-direction);
   /* U turn to the left */
   /* W turn handled implicitly */
   if (lx < DadLx) /* U turn to the left;
      PUSH(lx,DadLx - 1,pushlx,pushrx,
         y-direction,-direction);
}



/* fill a region, with seed at (seedx,seedy) */
Fill(seedx,seedy)
int   seedx,seedy;
{
int   x,y;
int   lx,rx,DadLx,DadRx;
int   pushlx,pushrx;
int   direction;
BOOLEANWasIn;     /* are the pixels from [lx..x) in a run?  */
(*Start) ();

find the span containing the seed point.
suppose it goes from (lx) to (rx), inclusive.
PUSH(lx,rx,lx,rx,seedy+1,1);
PUSH(lx,rx,lx,rx,seedy-1,-1);

while (tos >= 0) {
   POP();
   if ( (y < Top) OR (y > Bottom))
      continue;
   x = lx + 1;
   if (WasIn = (*Inside)(lx,y)) {
      (*Set)(lx,y); lx = lx - 1;
      while ((*Inside)(lx,y) AND (lx >= Left)) {
         (*Set)(lx,y); lx = lx - 1;
      }
      lx = lx + 1;
   }
   /* now looking at pixel (x).
   if (WasIn), then am inside a run of pixels from [lx..x)
   else, lx is meaningless     */
   while (x <= Right) {
      if (WasIn) {
         if ((*Inside)(x,y)) {
            (*Set)(x,y);
         } else {
            /* just found the end of a run */
            STACK(direction,DadLx,DadRx,lx,(x-1),y);
            WasIn = FALSE;
         }
      } else {
         if (x > rx) break;
         if ((*Inside)(x,y)) {
            (*Set)(x,y);
            /* just found the start of a run */
            WasIn = TRUE; lx = x;
         }
      }
      x = x + 1;
   }
   if (WasIn) {
      /* hit an edge while inside a run */
      STACK(direction,DadLx,DadRx,lx,(x-1),y);
   }
}
}
```