Filling Driven by Contour Marching

Gilles Mathieu

LISSE
Centre SIMADE, Ecole des Mines
158 cours Fauriel 42023 Saint-Etienne Cedex 2 France
tel 77 42 01 75 - fax 77 42 66 66
email mathieu@emse.fr

Abstract. A filling algorithm is proposed that works in a matrix of pixels. It proceeds by spans and is directed by contour marching, a contour being viewed as a circular list of linels. This technique has 3D applications: it is used in traversing the surfels graph of a voxelized object.

1 Introduction

Two-dimensional filling or coloring problems have received many solutions in the past. First methods were influenced by vector graphics technology and thus by continuous methods. They are inspired by *polygon scan-conversion* with ordered active edges list [FVFH90].
Among discrete methods, a second kind of approach aims at adapting the *parity check* technique to raster memory. Pavlidis has proposed various algorithms to solve what he calls undersampling problems (different boundary points being merged in a single pixel, diagonal contact between pixels) [Pav79].
An important family is the one of *connexity* methods. A *seed* pixel is supposed to be known. Depending on wether the transitive closure is done on pixels with the same value as the seed, or on those of value distinct from that of a boundary pixel, a distinction is made between *flood* and *boundary* techniques [FVFH90, p.979] (Hégron used the term *coloring*, in contrast to *filling* [Heg85, p.67]). Coloring better fits to an interactive use, because the automatic determination of the seed is tricky. The child's coloring book metaphor was first proposed by Lieberman [Lie78].
Most of the works consider *spans* which are limited by two boundary pixels and do not contain any pixel with the new color. Smith avoids redundant contiguous span explorations [Smi79]. Most of the methods also resort to a *stack* for the coloring fronts. At any time, the already colored part is limited by the object boundary, the current span and these fronts.
Shani emphasized that contour filling reduces to *graph traversal* [Sha80]. In his wake, Pavlidis unified parity check and filling using *boundary line adjacency graph* [Pav81]. Nodes in this graph of degree greater than 1 are decisive, because they correspond to vertical boundary extremums.
Finally, Tang et Lien analyse the Freeman code of the contour to fill [TL88].
The proposed filling method proceeds by spans. Its originality lies in the fact that it considers contours as *circular lists of linels* rather than pixels. Moreover, the will to *minimize pixel map accesses* has guided its elaboration. Finally, unlike the previous approaches, it is *recursive* and the processing of spans is *directed by contour marching.*

The next section gives the suitable definitions: some usual discrete topological notions, and definitions peculiar to our method. Section 3 is an informal and illustrated description of the algorithm. Section 4 discusses about complexity and advantages of our approach with respect to other techniques. Finally, the last section outlines the application of this filling algorithm to surface traversal of voxelized objects.

2 Definitions

2.1 Discrete topology

We review some of the definitions given by Kong and Rosenfeld in [KR89]. Let us consider a binary bidimensional matrix of *pixels*. Each pixel is associated with a *lattice* point, that is a point of the plane with integer coordinates. A lattice point associated with a pixel that has value 1 (resp. 0) is called *black* (resp. *white*).
A 3D (resp. 2D, 1D) unit cell is a cube (resp. a square, a segment) whose linear length is 1 and whose vertices are lattice points. A 0D unit cell is a single lattice point. In a 3D space, following the terminology adopted by Françon, we will respectively speak of *voxels*, *surfels*, *linels* and *pointels*, by analogy with *pixels*, unit cells in a 2D space [Fra95].

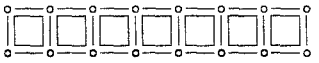


Fig. 1. Pixels (squares), linels (segments) and pointels (circles)

A *digital picture* is a quadruple $\mathcal{I} = (V, m, n, B)$ where $B \subseteq V = \mathbb{Z}^2$ is the black points set. $(m, n) = (8, 4)$ or $(4, 8)$. Two black points are said to be adjacent if they are m-adjacent. For any other pair of points n-adjacency is used.
A set S of black and/or white points in a digital picture is *connected* if S cannot be partitioned into two non adjacent subsets. A *component* of a set of black and/or white points S is a non-empty maximal connected subset of S (i.e. it is not adjacent to any other point in S). A component of the set of all black (resp. white) points of a digital picture is called a *black component* (resp. *white component*). The digital picture is supposed to be finite. So there is a unique infinite white component, called the *background*. Let X and Y be two sets of points in a digital picture (V, m, n, B), X being connected. Then X is said to *surround* Y if each point in Y is contained in a finite component of $V - X$. In 2D, a white component adjacent to a black component C and surrounded by it is called a *cavity* of C.
A black point is a *border point* if it is adjacent to one or more white points. Otherwise, it is called an *interior point*. The *border* (resp. *interior*) of a black component C of $\mathcal{I}$ is the set of every border (resp. interior) points of C.
We will use the classical notions of *xy-convexity* and *xy-concavity*. For a precise definition, we refer to [MF82] or to the automat given in [Mat95].

2.2 Span filling directed by contour marching

Let us consider a digital picture $\mathcal{I} = (V, 8, 4, B)$. Let us suppose that B has a unique component we call *object* O. This objet will possibly have holes: $V - B$ may have many components.

By *contour*, we mean the set of closed curves of border linels, that is circular lists of linels separating O from $\overline{O}$. Between O and the background stands an *exterior contour*. It is traced clockwise. Between O and the possible cavities stand *interior contours*. They are traversed counterclockwise. The fonction `march_contour()` will actually involve a contour marching algorithm [CM91]. But when the circular chaining of linels is built, it is enough to follow the links. A variable `current` contains a contour linel. It is initialized with an horizontal upper linel of a black pixel, taken in the first non empty scan line. Interior exploration is done by means of spans. A *span* is a maximal connected set of black points on a line of the picture. Thus spans are horizontal. They will systematically be explored from left to right. From a vertical left linel, `explore_span()` progresses inside the object and returns the first right linel encountered. Each span is caracterized by two vertical border linels which delimit it. In the sequel, we denote them by their two `begin` and `end` fields. Ends of the current span will be given by `current` and `end`. Taking into account only vertical border linels, we need fonctions `is_vertical()` and `is_left()`. Fonction `is_exterior()` is useful as well. Figure 2a shows the contour of an simple object. Figure 2b shows its two spans. Useful linels are numbered.

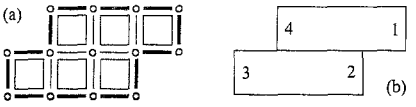


Fig. 2. Contour and spans

The algorithm uses a global table to mark linels. Initially, a linel `mark` is null, what can be tested by function `is_new()`. Function `color()` treats a span. It either assigns the color of the span pixels or it stores its ending linels for later use. In each case, the latter are marked such that `is_colored()` avoids coloring them again.
The method makes use of stacks and so has at its disposal the traditionnal primitives `create_stack()`, `empty_stack()`, `push()`, `top()` et `pop()`. Some stacks are sometimes accessed as queues, so operations `enqueue()`, `bottom()` and `dequeue()` have been added.

3 Description of the algorithm

We will present the algorithm in an informal way by showing its execution on some examples. They hare an increasing complexity (xy-convex object without cavity, object without cavity, object with cavities), so this allows an "incremental" understanding of the spirit of our method and of the data structures role. For a more formal approach, the reader is refered to the pseudo-code given in appendix. Throughout the text, i_j refers to instruction number j. The main loop presents six cases of iteration end. The first three ones work for a left current linel and the last three ones for a right current linel. Those cases are indicated as comments in the pseudo-code and in the last column in the tables showing the stacks states. In the text, l_i refers to the line of the table for which `current` is i. Significant elements of that line are in bold. Finally, in the figures, circled numbers give the order in which spans are colored.
The general methodology consists in marking each linel while marching along the contour, clockwise for the exterior contour, counterclockwise for an interior one. For a

vertical left met linel, a span exploration is carried out. For a vertical linel encountered again, or for a right linel, instead of exploring a span we retrieve it from one of the stacks `explored` (local) or `remainder` (global). Another global stack `linels` stores already encountered but not yet belonging to a colored span. The algorithm is called recursively for each discovery of an interior contour. The detection of a potential cavity happens when the exploration ends up at a never encountered linel. Confirmation is provided only farther on the current contour marching, so a last data structure is needed. Local variable `discovery` stores a span that may lead to a new contour. But let us explain more deeply the whole process by means of examples.

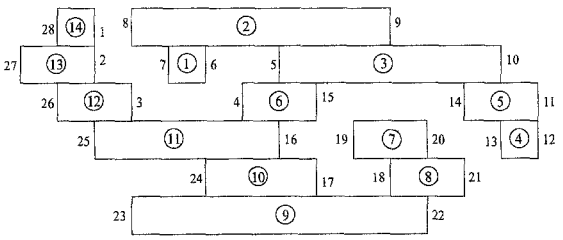
3.1 Xy-convex object without cavity

cur.	end	disc.	linels	explored	case
1	-	-	**1**	$\emptyset$	6
2	-	-	. **2**	$\emptyset$	6
3	**2**	-	1	$\emptyset$	1 : 3-2
4	**1**	-	$\emptyset$	$\emptyset$	1 : 4-1

Table 1. Execution for an xy-convex objet without cavity (see fig. 2)

The main loop consists in marching the whole exterieur contour (i_1, i_7, i_8, i_9), taking into account only vertical linels (i_{10}, i_{11}). Each new encountered linel is marked with the current contour index, here `n_contour` = 1 (i_7). According to the choices given above for the initial linel and the directions of marching, right linels come up first. They are pushed into the global stack `linels` (i_{31}, case 6, l_1 and l_2 table 1). After that, left linels are found (i_{12}). A span exploration is done from the beginning of each of them (i_{13}). It necessarily ends up at the linel on top of the stack (i_{15}). The latter is then popped and the span colored (i_{16}, case 1, l_3 and l_4 table 1). In this first coloring case, the left vertical linel of the span is always encountered after its right one.

3.2 Object without cavity



In the presence of xy-concavities, an exploration beginning at a left linel may now not end up at the linel on top of `linels` anymore (l_4 table 2 : top 3, end 15). This left

linel is then pushed into `linels` (i_{20}). A stack `explored`, local to the call, stores the span just explored (i_{26}, case 3). For a right linel, we examine the exploration stored on top of `explored`, by means of `already_explored()` (i_{29}, end between parentheses in the tables). If this span end do not correspond to the current linel, its `begin` is simply pushed into `linels` (i_{31}, case 6, l_{11} table 2). In the opposite case, the span is colored and the two stacks popped (i_{30}, case 5, l_9 table 2). In this second coloring case, the span left linel is always encountered before its right one.

cur.	end	disc.	linels	explored	case
1	-	-	1	∅	6
2	-	-	. 2	∅	6
3	-	-	. . **3**	∅	6
4	15	4-15	. . . 4	4-15	3
5	10	4-15	 5	4-15 5-10	3
6	(10)	4-15	 6	4-15 5-10	6
7	6	4-15	 5	4-15 5-10	1 : 7-6
8	9	4-15	 8	4-15 5-10 8-9	3
9	**(9)**	4-15	 5	4-15 5-10	5 : 8-9
10	(10)	4-15	. . . 4	4-15	5 : 5-10
11	**(15)**	4-15	 11	4-15	6
12	(15)	4-15	 12	4-15	6
13	12	4-15	 11	4-15	1 : 13-12
14	11	4-15	. . . 4	4-15	1 : 14-11
15	(15)	-	. . 3	∅	5 : 4-15
16	-	-	. . . 16	∅	6
17	-	-	 17	∅	6
18	21	18-21	 18	18-21	3
19	20	18-21	 19	18-21 19-20	3
20	(20)	18-21	 18	18-21	5 : 19-20
21	(21)	-	 17	∅	5 : 18-21
22	-	-	 22	∅	6
23	22	-	 17	∅	1 : 23-22
24	17	-	. . . 16	∅	1 : 24-17
25	16	-	. . 3	∅	1 : 25-16
26	3	-	. 2	∅	1 : 26-3
27	2	-	1	∅	1 : 27-2
28	1	-	∅	∅	1 : 28-1

Table 2. Execution for an objet without cavity

3.3 Object with cavities

When `explored` is empty, a span exploration ending up at a linel different from `linels` top possibly means the discovery of a cavity. So this span is stored in a variable `discovery` (i_{21}, i_{23}, i_{25}), local to each recursive call. Sometimes, it is a false alarm: linel `discovery->end` belongs to the contour. It will be encountered later and `discovery` canceled (figure 3a, table 2). In other cases, we actually deal with a cavity (figure 3b, table 3). While `discovery` is not assigned, explored spans are pushed in global stack `remainder` (i_{26}, case 3, l_{14} table 3).

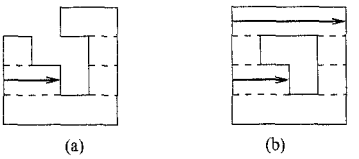


Fig. 3. Cavities : (a) false discovery (b) true one

The assignation of `discovery` is not sufficient to detect a cavity. Three situations are possible, as illustrated on figure 4. In the first one, the end of the exploration is different from the linel on top of `linels` and it has mark 1 of the exterior contour. Then it is certain that a cavity lies *under this last explored span* or *on the right of the current contour* (i_{17}, l_{14} table 3). In the example of figure 4a, span 2 confirms the presence under it of a cavity discovered with span 1. On figure 4d, span 8 confirms the presence on the right of the current contour of a cavity discovered with span 6.

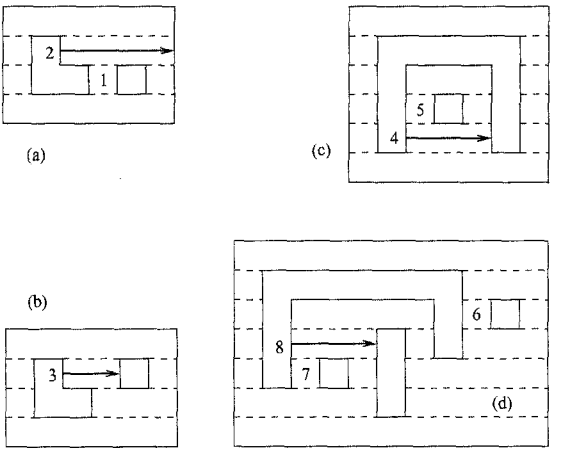
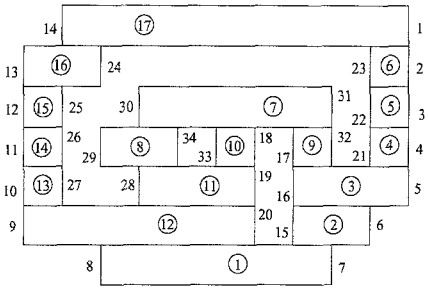


Fig. 4. Different cases for recursion

In the second situation, the current contour marching completes without any exploration having ended up at an exterior linel. The span `discovery` leads in this case to a cavity *on the right of the current contour* (i_8, i_9, l_{20} et l_{32} table 3). In the example of figure 4b, the discovery made with span 3 is confirmed only after completion of the contour marching.

Finally, in the last situation, the current interior contour marching reaches `discovery ->end`. Thus it is certain that a cavity lies *above the discovery span* (i_{27}). In that case, the span that actually leads to the cavity stands in `explored` after `discovery`. It is determined by fonction `actual_dec()`, that also transfers into `remainder` spans what is popped from `explored`. On figure 4c, span 4 has been considered as a discovery, but the true discovery span is 5.



cur.	end	disc	linels	explored	remainder	case
⋮	⋮	⋮	⋮	⋮	⋮	⋮
13	24	-	 13	9-20 10-27 11-26 12-25 13-24	∅	3
14	1	-	 13	9-20 10-27 11-26 12-25 13-24	**14-1**	2
15	6	-	 5	∅	14-1	1 : 15-6
16	5	-	. . . 4	∅	14-1	1 : 16-5
17	32	17-32	 17	17-32	14-1	3
18	(32)	17-32	 18	17-32	14-1	6
19	(32)	17-32	 19	17-32	14-1	6
20	(32)	17-32	 20	17-32	14-1	6
21	4	-	. . **3**	∅	14-1	1 : 21-4
22	3	-	. 2	∅	14-1	1 : 22-3
23	2	-	1	∅	14-1	1 : 23-2
24	-	-	. 24	∅	14-1	6
25	-	-	. . 25	∅	14-1	6
26	-	-	. . . 26	∅	14-1	6
27	-	-	 27	∅	14-1	6
28	19	-	 28	∅	14-1 28-19	3
29	34	29-34	 29	29-34	14-1 28-19	3
30	31	29-34	 30	29-34 30-31	14-1 28-19	3
31	(31)	29-34	 29	29-34	14-1 28-19	5 : 30-31
32	(34)	29-34	 32	29-34	14-1 28-19	6
33	18	-	. . . . . **33**	∅	14-1 28-19 33-18	3
34	-	-	 34	∅	14-1 28-19 33-18	6
(29)	**(34)**	-	 33	∅	14-1 28-19 33-18	1 : 29-34
32	-	-	 32	∅	14-1 28-19 33-18	6
(17)	(32)	-	 33	∅	14-1 28-19 33-18	1 : 17-32
18	-	-	 28	∅	14-1 **28-19**	5 : 33-18
19	-	-	 27	∅	14-1	5 : 28-19
20	-	-	 20	∅	14-1	6
(9)	**(20)**	-	 27	**10-27** 11-26 12-25 13-24	14-1	1 : 9-20
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 3. Execution for an objet with cavities

The detection of a cavity results in a recursive call of `filling()` on the new interior contour (i_{19} et i_{24}, case 2; 9; i_{28}, case 4). A call to fonction `step_back()` then manages poping `linels` up to `discovery->begin` included (i_2, i_3). After incrementation of the contour index, a recursion on `discovery->end` takes place (i_3, i_4, l_{21} et l_{33} table 3). Back from a recursive call, the current linel is restored to `discovery->begin` (i_5, `current` between parentheses in the tables, $l_{(29)}$ table 3).
Furthermore, after recursion (i_6), and while `explored` is not empty, no span exploration happens and the fonction `already_explored()` accesses to the bottom of `explored`, whilst `delist()` dequeues (i_{14}, i_{30}, $l_{(9)}$ table 3). When `explored` gets exhausted, `already_explored()` returns the top of `remainder`, whilst `delist()` pops it (second l_{18} table 3).

4 Discussion

In usual methods, and following the definition from [KR89], the border is a set of pixels. Following this border poses specific problems that Pavlidis collectively termed undersampling [Pav79]. We have considered a contour as a circular list of linels. Using a "border" in the common meaning of topology, undersampling problems have been disposed of.
In the general case, an exploration is saved in `explored` or `remainder` (i_{25}, i_{26}, case 3). A span which allows to claim that `discovery` actually leads to a new cavity is saved as well (i_{18}, i_{23}), except in the case when its end is an interior linel. For that kind of relatively uncommon configuration (see span 8 in figure 4d), the span will be explored twice. Most of the spans are visited only once and the redundancy of binary map accesses is minor. The number of accesses to each pixel is thus in $O(n)$. This number is in $O(3n)$ in Pavlidis's algorithm [Pav81]. In his approach, a span exploration systematically entails the one of the two neighboring spans, above and below, in order to determine the degree of this vertex in the line adjacency graph.
The number of recursive calls is proportional to the number of interior points in the naïve version of the coloring algorithm. In usual "span" methods, among whom the one of Pavlidis, the size of the stack is in ratio to the number of spans in the boundary. In the proposed algorithm, the number of recursive calls is he number of cavities in the object. So the size of the stack is not a problem anyway.
The fact that the method is driven by contour marching directly gives the number of cavities. The approach preserves more topological information than usual methods, the latters being driven by the waiting structure for spans. This is attractive when having 3D applications in mind.

5 Voxelized object surface traversal

Given a cut plane, surfels of a voxelized object can be divided into two sets: *slice surfels* and *ring surfels*. The former are parallel to the cut plane and shaded in figure 5. The latter take their name from the fact that in each slice, they can be described by circular lists.

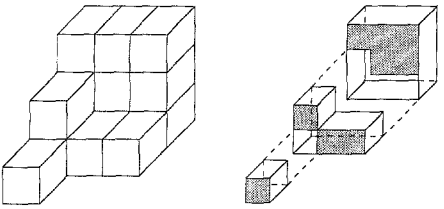


Fig. 5. Ring and slice surfels

Work is in progress to realize the *surface tracking* of a voxelized object according to a *ring/slice scheme.* It can be described informaly by the following procedure:

```
z = initial slice;
surfel = a ring surfel in slice z;
push( surfel, z, NEG );
push( surfel, z, POS );
while ( non empty stack )
    {
    ( surfel, z, direc ) = pop();
    if ( is_( surfel, z, direc ) = pop();
    ring = detect_contour( surfel );
    if ( is_already_treated( ring ) ) continue;
    z_next = ( direc == NEG ) ? z-1 : z+1;
    opp_direc = ( direc == NEG ) ? POS : NEG;
    if ( ring_type( ring ) == EXTERNAL )
        filling( ring );
    else
        next_ring( ring );
    for ( each new ring i )
        if ( trailer[ i ]->z == z )
            push( trailer[ i ]->surfel, z, opp_direc );
        else
            push( trailer[ i ]->surfel, z_next, direc );
    }
```

Fonction `detect_contour()` is a contour tracking that chains detected surfels as a ring. The filling algorithm is adapted for the slice surfels detection. Fonction `march_contour()` is reduced to tracing the ring just built, whilst `filling()` is enriched with the management of a `trailer` table, indexed by interior contours. Each of its elements is a trailer surfel for an incoming ring, coupled up with the corresponding slice value. The latter is either the current ring slice value, or the one of a contiguous slice, depending on wether the ring type (external or internal) reverses or not.
The surface tracking method that is traditionnally used is the one described by Artzy et al. [AFH81]. Though elegant, the algorithm does not offer any a priori knowledge of the order in which surfels will be detected. This poses problems, especially pagination

when accessing the 3D binary map [FHMU85]. It seems to be worthwhile to *load in memory* a few contiguous slices from the voxmap (we call it a "metaslice") and to treat it without having to access it again. Furthermore, if the detection progresses slice by slice, a *parallel implementation* can be taken under consideration, in which each processor should work on a different metaslice. Finally, this approach is also richer in topological information. It is indeed possible to build on the fly the *Reeb graph* of the object [SKK91].
More generally, this ring/slice scheme may be applied to every surfels graph traversal, for example the search for *local extremums* on the surface. It also directly leads to a *3D filling algorithm.*

6 Conclusion

We have proposed a 2D filling algorithm. It is free from undersampling problems because of its use of contours constituted of linels instead of pixels. The redundancy of binary map accesses is minor. The approach is recursive and the number of calls is only the number of cavities. The principle of a treatment of spans directed by contour marching directly provides the cavities. In that, it offers more topological information than usual techniques. The method finds an application in 3D voxelized objects surface traversal. It is in fact a basic procedure of algorithms for tracking, searching for extremum and filling of those objects surface.

The author wishes to thank the anonymous reviewers, D. Michelucci and B. Péroche for their constructive remarks on the preliminary version of this paper.

Appendix: pseudo-code

```
    filling( current )
    {
1   for ( explored = create_stack(), discovery = null,
    recurse = after_recursion = 0, n_contour = 1 ; ; )
        {
2       if ( recurse )
              {
3             step_back( discovery ); ++n_contour;
4             filling( march_contour( discovery->fin ) );
5             current = discovery->begin; discovery = null;
6             after_recursion = 1; recurse = 0;
              }
        else  {
7             current = march_contour( current ); mark( current ) = n_contour;
8             if ( contour completed )
                  {
9                 if ( discovery ) recurse = 1;    else return;
                  }
              }
10      if ( recurse || !is_vertical( current ) || is_colored( current ) )
11            continue;
12      if ( is_left( current ) )
              {
13            if ( !after_recursion )          end = explore_span( current );
14            else              { end = already_explored()->end; delist(); }
15            if ( end == top( linels ) )
                  {
16                color( current, end ); pop( linels );                    /* 1 */
                  }
17            else if ( cavity_under_or_right( discovery, end ) ) {
                  {
18                if ( is_exterior( end ) )   push( (current, end), remainder );
19                recurse = 1;                                             /* 2 */
                  }
              else{
20                push( current, linels );
21                if ( !discovery && is_new( end ) )
                      {
22                    if ( after_recursion )
                          {
23                        enqueue( discovery = (current, end), explored );
24                        recurse = 1;                                     /* 2 */
                          }
25                    else push( discovery = (current, end), explored );
                      }                                                  /* 3 */
26                else push( (current, end), discovery ? explored : remainder );
                  }
              }
27      else if ( cavite_above( discovery, current ) )
              {
28            actual_dec( discovery, explored ); recurse = 1;            /* 4 */
              }
29      else if ( span = already_explored() && span->end == current )
              {
30            color( span->begin, end ); pop( linels ); delist();        /* 5 */
              }
31      else push( current, linels );                                      /* 6 */
        }
    }
```

References

[AFH81] E. Artzy, G. Frieder, and G. Herman. The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm. *Computer graphics and image processing*, (15):1–24, 1981.

[CM91] J.M. Chassery and A. Montanvert. *Géométrie discrète en analyse d'images.* Hermes, 1991.

[FHMU85] G. Frieder, G. Herman, C. Meyer, and J. Udupa. Large software problems for small computers: an example from medical imaging. *IEEE Software*, (2):37–47, September 1985.

[Fra95] J. Françon. Topologie de khalimski-kovalevsky et algorithmique graphique. *Technique et science informatiques*, 14(10):1195–1219, 1995.

[FVFH90] Foley, VanDam, Feiner, and Hughes. *Computer graphics: principles and practice.* Addison Wesley, 2 edition, 1990.

[Heg85] G. Hegron. *Synthèse d'image: algorithmes élémentaires.* AFCET Dunod, 1985.

[KR89] T. Kong and A. Rosenfeld. Digital topology: introduction and survey. *Computer vision, graphics and image processing*, (48):357–393, 1989.

[Lie78] H. Lieberman. How to color in a coloring book. In *SIGGRAPH'78*, volume 12(3), pages 111–116. ACM, August 1978.

[Mat95] G. Mathieu. Points significatifs dans un contour discret : caractérisation des cavités et dilatation. In *5th Discrete geometry for computer imagery*, pages 249–258, 1995.

[MF82] D. Montuno and A. Fournier. Finding the x-y convex hull of a set of x-y polygons. Technical Report CSRG-148, University of Toronto, November 1982.

[Pav79] T. Pavlidis. Filling algorithms for raster graphics. *Computer graphics and image processing*, 10:126–141, 1979.

[Pav81] T. Pavlidis. Contour filling in raster graphics. In *Proceedings SIGGRAPH'81*, volume 15(3), pages 29–36. ACM, aug 1981.

[Sha80] U. Shani. Filling regions in binary raster images: a graph-theoretic approach. In *SIGGRAPH'80*, volume 14, pages 321–327. ACM, 1980.

[SKK91] Y. Shinagawa, T. Kunii, and Y. Kergosien. Surface coding based on morse theory. *IEEE computer graphics and applications*, 11(5):66–78, September 1991.

[Smi79] A. Smith. Tint fill. In *SIGGRAPH'79*, volume 13(2), pages 276–283. ACM, 1979.

[TL88] G. Tang and B. Lien. Region filling with the use of the discrete green theorem. *Computer vision, graphics and image processing*, (42):297–305, 1988.