Available online at www.sciencedirect.com
SCIENCE DIRECT®

Computers & Graphics 29 (2005) 441–450

COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Technical section

Note: An algorithm for contour-based region filling

Marius C. Codrea*, Olli S. Nevalainen

Turku Centre for Computer Science and Department of Information Technology, University of Turku, Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland

Abstract

A linear-time algorithm for filling regions defined by closed contours in raster format is proposed. The algorithm relies on a single pass contour labeling and the actual filling is done in a scan-line manner, visiting the interior pixels only once. The interior endpoints of the scan-lines are discovered solely on the basis of the labeled contour. Despite its simplicity and low computation cost, the proposed method fills-in arbitrary shapes correctly, with no obvious exceptions. Possible special cases may arise from the 8-connectivity, inner contour detection. However, in extensive tests on complicated images, the proposed labeling scheme has given correct results.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Raster region filling; Linear-time algorithm; Contour labeling; Scan-line method

1. Introduction

The region filling is an important problem with many practical applications in computer graphics and image processing. Two broad categories of region-filling algorithms can be identified depending on the domain of the graphics they operate in: *raster filling* and *vector filling*. A series of raster region-filling algorithms were presented by Pavlidis [1] in 1979. Efficient filling algorithms for vector display graphics also have a long history [2,3]. In raster graphics the images are plain matrices (*bit maps*) that store the color of the pixels. Vector graphics handles geometric information about lines, curves or other shapes called *vectors*. However, in the end, vector graphics produces raster representations of the images for visualization or printing. This can be done accurately at various scales and sizes, unlike in the case of raster graphics. Another advantage of vector representation comes from the reduced amount of storage space required.

Although there exists some terminological confusion in the field, region-filling algorithms are generally classified into *seed filling* (also referred to as *connectivity filling* or *region growing*) and *edge filling* (*scan-line* or *parity check filling*) [4]. Several authors also refer to the seed-filling algorithms as *flood filling*. However, flood filling (according to e.g., [5,6]), addresses actually a slightly different problem, explained in the following paragraph. As a rule of thumb, edge-filling algorithms are faster and require less or no additional working memory compared to seed-filling algorithms. However, many edge-filling algorithms fail to correctly handle complex objects while seed filling is, in principle, more robust. Alternatively, new encoding schemes like bin-code represented contours [7], chain-code flags [8] or quad-trees [9] and algorithms for filling areas have been investigated. These improved image representations aim at fast image manipulation, processing and display.

*Corresponding author. Tel.: +358 2333 8668.
E-mail address: codrea@it.utu.fi (M.C. Codrea).

0097-8493/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2005.03.005

Seed-filling algorithms need an interior starting point (the seed), a propagation procedure and the color of the boundary of the object of interest (OOI). *Flood filling* also requires a seed point and a propagation procedure, yet, the aim is to (re)color a region that can be bordered by several other areas of different colors. The result of the two procedures may therefore be different. Assume that another region of a different color (both from the color of the OOI and its contour), lies entirely in the interior of the OOI. A flood filling algorithm preserves the color of the overlapping region whereas seed filling does not.

The initialization of these algorithms (the choice of the seed point) can be done either automatically or user-guided. For complex OOIs, multiple starting points may be required and their automatic detection can be very difficult. This is one of the main reasons why, this family of algorithms is more suitable for interactive region filling. The propagation can be based on pixel-adjacency, where the filling procedure moves from a current pixel to all immediate neighbors in a certain order or line-adjacency, where the interior of a region is regarded as a set of adjacent horizontal line segments. In both cases, the naive implementation of the algorithm uses recursive search for connected pixels and stores the frontier pixels (i.e., the pixels that still have unprocessed neighbors) in a memory stack [6]. However, several space and computational efficient algorithms have been proposed. Two dedicated stacks that alternate between the active or passive state corresponding to the direction of inspection (up/down) were used by Burtsev and Kuzmin [5]. Queue data structures are used instead of stacks and iterative filling is performed following a centrifugal pattern in the algorithms suggested by Albert and Slaaf [10]. Strategies that require only a constant size working memory and heuristics for speeding up the basic methods are described in [11].

Edge filling. The underlying idea of the edge-filling algorithms originates from the problem of filling polygons, which can be found in virtually every Computer Graphics textbook (e.g., [6]). The core of the algorithm is the detection of the spans of the scan line lying inside the polygon, based on the *odd-parity rule*. A point is *interior* if any half-line originating from the point in question crosses the polygon edges an odd number of times. Otherwise, the point is *exterior* to the polygon area. It is known that correct edge intersection count, and implicitly the interiority decision, is heavily dependent on the topological distribution of the polygon vertices. Concavities, self-crossing edges, parallel edges to the scan direction are examples of situations requiring additional analysis and computation. In such cases, the polygons can be decomposed into simpler shapes, usually triangles or trapezoids, that are further processed individually [2,12]. This decomposition can be computationally demanding though and therefore find limited application. Another method for finding the interior regions of polygons is the so-called *nonzero winding number rule*. This method is also based on counting the intersections of polygon edges and similar problems as with odd-parity rule arise. We therefore omit its discussion here [6]. A particular form of polygon filling using parity checks is described in [13]. Depending on the type of the polygon edges (left or right), interior border pixels are identified during the integral linear interpolation (discretization) of the edges. These border pixels are subsequently used as endpoints of the scan-lines.

Most studies on region filling omit a formal or explicit formulation of the target of the proposed algorithms (that is, the definition or description of the interior/exterior regions of digital objects). This aspect is important because, for example, in addition to the aforementioned comparison of seed filling and flood filling, the odd-parity and the nonzero winding number rules may lead to different results when applied to complex polygons [6]. This lack of clear formulation of the filling problem might occur because the discrete topology research offers a complicated and yet not fully agreed basis upon the theoretical background (see, for example, [14–16] or more recent studies [17,18] and the references therein). The key concept of interest here is the *Jordan theorem* which states that, in the continuous case, a *simple*, *closed curve* separates the space into two disjoint connected sets. In image processing or computer graphics, these sets are called *interior* and *exterior.* The equivalent discrete Jordan theorem is thanks to Khalimsky [17]. Nevertheless, the definition of a digital, simple, closed curve itself is very restrictive [16]. Because the formal definition requires additional notations, we only mention that the points of a simple curve must have exactly two neighbors and be traversed only once. It is therefore obvious that for real life applications the concept may not be suitable. A more relaxed, graph-theoretical-based definition of the so-called *closed quasi-curves* was introduced by Malgouyres [19] and extended by Thürmer [18]. Such a curve allows the border pixels to have more than two neighbors and yet have the Jordan property for certain topological configurations. However, the definition of contours and connectivity relations (neighborhoods) is not a trivial problem at all [15]. We return to this issues in Section 2, after introducing the necessary notation.

The algorithm proposed in this paper fills regions given their contour in *raster form*, expressed either by the explicit pixels of the OOI, the contour pixels or the 8-connected chain-code. We show that a single contour traversal enables a consistent labeling of the border pixels such that the endpoints of the interior pixels on each line are accurately defined. The algorithm operates without a need for investigation of exceptional shape scenarios like self-contacting borders or corner points (cf., for example [1]). The method will be described, starting the contour tracing of the OOI and then the methods of labeling and region filling. Appendix A of the paper provides the complete C code of the algorithm. The labeling of the contour requires the investigation of the 3×3 neighborhood of the pixel in focus. We use static memory allocation for the two matrices (the image itself and the labeled contour matrix) because the offset variables used to locate the entries in dynamically allocated arrays would obscure the flow of the algorithm. Possible optimized implementation is discussed in the concluding section of the paper.

The motivation of the present work arose from an image processing application. The task was to estimate the area of arbitrary OOIs, operation that could not be performed unless by counting the interior pixels. The OOIs were obtained by threshold segmentation, applied to images with edges of enhanced contrast. Thus, in most cases, the contour of the segmented OOIs was accurate but they contained "holes" that had to be filled in order to obtain correct area estimation.

2. The algorithm

2.1. Contour tracing and labeling

We chose the 8-*connectivity* for border tracing. Let $P(i,j)$ be a pixel of the image area. The set of neighboring pixels

$$N_8(P) = \{(i+1,j),(i-1,j),(i,j+1),(i,j-1),\\ (i+1,j+1),(i+1,j-1),\\ (i-1,j+1),(i-1,j+1)\}$$

is called the 8-*neighborhood* of $P(i,j)$ (see Fig. 1). Two pixels P and Q are 8-*adjacent* if Q is in the set $N_8(P)$. Two pixels are 8-*connected* if there exists a path (a sequence of 8-adjacent points) joining them. Similar definitions are used for 4-adjacency and connectivity (Fig. 1, only the even directions). A *closed curve* is a path that starts and ends at a given point.

There are basically two known types of contour tracing techniques: *edge-following* and *border-following* [16]. The difference between the two approaches is rather conceptual than technical, as both produce the same sequence of contour elements for simple, closed curves. The border-following algorithms are faster but slightly more complicated. In edge-following, the contour consist of pairs of adjacent pixels (one in the OOI and the other not) that form "edges" and the tracing is performed following a "hand on the wall" procedure (e.g., counterclockwise) around the object. In the border-following case, the elements of the contour are the pixels themselves. However, according to Kovalevski [15] the only consistent contour definition which also produces zero-thickness discrete curves is the 6-adjacency rule (Fig. 1, without directions 1 and 5). Contours can then be presented as pairs of pixels, called *pixedges*, one in the region and the other not. An algorithm for contour tracing, using a particular encoding scheme is given in [15]. Region filling can be done for simple, closed contours using the odd-parity rule, between the vertical pixedges.

In the following we do not attempt to give a formal definition of a discrete contour but rather a clear statement of the type of "interiority" and implicitly the type of region filling we address. This is illustrated by the sample images in Appendix B. By convention, pixels are considered unit squares in $\mathbb{Z}^2$. We regard the contour of a digital object as a series of line segments representing edges of the border pixels. We use the informal, self explanatory terms "right" (R), "top" (T), "left" (L) and "bottom" (B) to refer these edges. Assuming that the edges lie at an arbitrary small

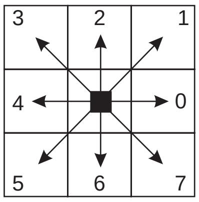


Fig. 1. Conventional numbering of the neighbors and contour search directions, 8-*adjacency*.

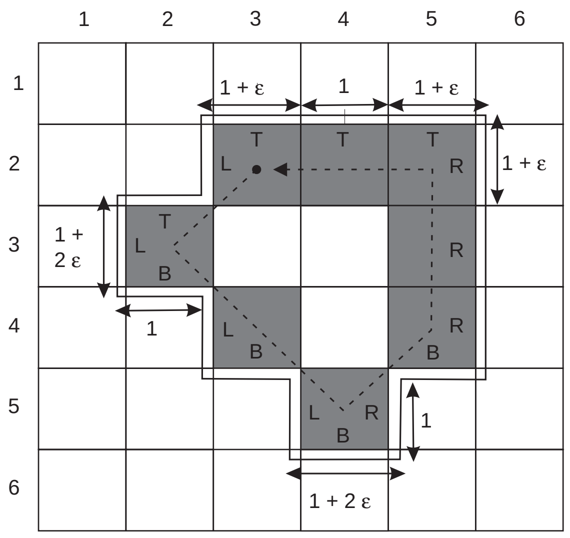


Fig. 2. Labeling the boundary pixels on the basis of the traversing direction, as shown by the dashed line. The contour consists of the horizontal and vertical line segments by the border pixels. The segments are at a small distance (ε) from the pixels and their length is given for few sample pixels.

distance (ε) from the actual pixel, towards the corresponding direction (R, T, L or B) and have lengths of 1, $1+\varepsilon$ or $1+2\varepsilon$ as exemplified in Fig. 2, we argue that such a contour has no self-contact or self-intersections, regardless of the shape of the OOI.

Without redefining here all the concepts, we remind the reader that the Jordan theorem applied to simple, closed curves relies on the fact that 8-connected contours separate the space into two 4-connected sets and vice versa [16,18]. Two points in a compact set are said to be path-connected if there exists a path, joining them, that lies entirely in the set, without crossing the boundary. In a compact set, every pair of points are path-connected. A simple counter example is the set of points of two triangles that share a vertex. Obviously, the proposed contour coding scheme leads to a convenient definition of the connectivity of the interior pixels but, as such, the contour of the object and the contour of its complement do not coincide in the discrete space. Recall that the goal here is to define the target objects and the behavior of the filling algorithm and not to explore topological issues. Upon the proposed definition of the contour, the interior of an OOI becomes a compact set of points in the continuous space. Even diagonal lines of single-pixel width consist of points that can be joined by a continuous path that does not intersect the contour.

We next show how the contour can be translated into a series of multiple state labeled boundary pixels, algorithmically detectable. More precisely, there are in total 15 states, corresponding to the number of combinations of the four labels (R, T, L, B). In fact, for the filling purpose, it suffices to consider only the L and R labels that define the spans for the scan line. The T and B labels can serve a similar, column-wise filling. Although not used in filling, for completeness of the contour description, we include the labeling rules of T and B also. The intuition behind the proposed labeling scheme is that after traversing the left-hand side of the object, producing L labels, the boundary tracing must go upwards in order to reach the starting point, producing R labels.

The inner contour tracing procedure starts at the top, left-most boundary pixel P_0 of the OOI (the pixel at location $(2,3)$ in Fig. 2). The search direction is initialized to $dir = 7$ (Fig. 1). The next contour point is located by anticlockwise scanning the 8-neighborhood of the current pixel starting at the pixel positioned in the direction:

$$(dir+7) \bmod 8 \text{ for even } dir \text{ values and}$$
$$(dir+6) \bmod 8 \text{ for odd } dir \text{ values,}$$

until the initial border pixel (P_0) is met again. The direction dir is updated before searching for the next contour pixel. A sample sequence of contour traversal is given along with the labeling decisions in Section 2.3 after the description of the algorithm.

The question whether the tracing always returns to P_0 or for arbitrary OOIs is notoriously difficult. To the best of our knowledge, there is no proof available but, intuitively, it should hold. However, if the starting point P_0 has neighboring pixels in direction 5 and 0 or 7 (Fig. 1), and there is no other path connecting these pixels, except through P_0, the algorithm does not completely scan the OOI. Such a configuration is exemplified in Fig. 5b where P_0 is at $(2,4)$. The contour tracing follows the part of the OOI connected through the pixel at position 5 (in this example, only the pixel at location $(3,3)$), returns to P_0 and stops before traversing the other pixels of the OOI. To correct for this undesired behavior, either a pixel can be always added at position 2 of P_0 and removed after contour traversal or allow the tracing to visit P_0 more than twice and test for complete labeling of the neighboring pixels in directions 0 and 7. We chose the first solution because it seems easier to implement. Also, obviously, a single pixel OOI causes the algorithm to enter an infinite loop.

We use in the following the terminology of the C code (Appendix A) where, for short, we denote the "current search direction" by *curr_dir*, the "previous search direction" by *prev_dir*, the "current point/pixel" by *cp* and "next pixel" by *np*. The labels of the contour pixels are encoded by integers such that their binary representation allows logical "OR" operator to assign labels and bit shift operators to test for the presence of labels. We let nonborder or unlabeled pixels be $0 = 0000_2$, $R = 1 = 0001_2$, $T = 2 = 0010_2$, $L = 4 = 0100_2$ and $B = 8 = 1000_2$.

The proposed labeling scheme is intuitive and relies on the fact that the traversal direction reveals what part of the object is scanned. Labeling is done by performing two operations for each boundary pixel: *single direction-based labeling* and *double direction-based labeling*. The distinction of these operations is only conceptual as they are performed at the same time, during a single contour traversal. Though, this distinction facilitates the explanation of the algorithm. In the first labeling operation, the *curr_dir* is used to mark both the *cp* and the *np* as regular edges of a certain orientation (horizontal, vertical or diagonal). In double direction labeling, both *curr_dir* and *prev_dir* are used to mark the *cp*, addressing the turn-points of the contour. Obviously, at each step of the traversal, certain labels are assigned to the boundary pixels and the complete set of labels is obtained after the contour following has terminated. This is because some pixels may be visited more than once during tracing (e.g., the pixel $(6,7)$ in Fig. 5, which is traversed three times) and therefore their state changes. Note that some labels may be assigned more than once (e.g., T label of pixel $(5,9)$ in Fig. 5).

2.1.1. Single direction-based labeling

A visualization of the labeling rules is given in Fig. 3 and explained below with example pixels from Fig. 5.

If *curr_dir* is

- 0, both *cp* and *np* are labeled *B*. Horizontal traversal, from left to right (e.g., *cp* = (9, 3) and *np* = (9, 4)).
- 1, *cp* becomes *R* and *np* *B* (e.g., (8, 11) and (7, 12)).
- 2, both *cp* and *np* are labeled *R*. Vertical traversal, upwards (e.g., (3, 5) and (2, 5)).
- 3, *cp* becomes *T* and *np* *R* (e.g., (7, 11) and (6, 10)).
- 4, both *cp* and *np* are labeled *T*. Horizontal traversal, from right to left (e.g., (5, 3) and (5, 2)).
- 5, *cp* becomes *L* and *np* is labeled *T* (e.g., (6, 7) and (7, 6)).
- 6, both *cp* and *np* are labeled *L*. Vertical traversal, downwards (e.g., (6, 2) and (7, 2)).
- 7, *cp* becomes *B* and *np* *L* (e.g., (4, 5) and (5, 6)).

2.1.2. Double direction-based labeling

Certain turn-points of the contour need additional labels than those provided by the single direction-based labeling. Their proper setting is achieved on the basis of traversal direction change from *prev_dir* to *curr_dir* (Fig. 4). The headings of the rows and columns are the directions of the previous and current traversal and the entries are the labeling decisions for the *cp* only. There are in total $8 \times 8 = 64$ theoretically possible direction changes. But, according to the counterclockwise traversal, 12 combinations of these are not valid turns (as indicated in Fig. 4 by "-"). In 32 cases, marked by "*", no update of the labels is needed because already the single direction-based labeling leads to correct and complete results. These facts are easy to check using the labeling rules in Fig. 3. Obvious examples of pairs of direction change that need extra labeling are the opposite directions (0–4, 1–5, 2–6 and 3–7), in either sense. We remark that the pairs of changes 1–5 and 3–7 even require two new labels to be assigned. As a nontrivial example, a turn into the search direction from 7 to 1 signifies a protuberance of one pixel of the object, downwards (e.g., the pixel (9, 10) in Fig. 5). This means that the pixel should be labeled also *B*, in addition to the *L* label gained at the previous step (when, coming from (8, 9), the traversal direction was 7 and it was the "*np*") and in addition to the *R* label, just assigned, according to the current single direction-based labeling (*curr_dir* = 1).

Previous search direction \ Current search direction	0	1	2	3	4	5	6	7
0	*	*	*	R	R	—	—	*
1	*	*	*	R	R	TR	—	*
2	—	*	*	*	*	T	T	—
3	—	*	*	*	*	T	T	LT
4	L	—	—	*	*	*	*	L
5	L	LB	—	*	*	*	*	L
6	*	B	B	—	—	*	*	*
7	*	B	B	RB	—	*	*	*

— Impossible direction change
* No label update required

Fig. 4. Labeling of turn-points according to the change in traversal direction, as known from *prev_dir* and *curr_dir*. Note that only the current pixel (*cp*) is subject to the labeling decisions.

2.2. Filling algorithm

The filling algorithm is straightforward: it scans the OOI, line by line, filling between pairs of $L-R$ pixels.

2.3. Sample operation of the labeling algorithm

For an example of the search direction change and labeling during contour traversal let us refer to the object in Fig. 5. Recall that the traversal proceeds counter-clockwise (Fig. 1). Let us suppose that the current point is the pixel (5, 6). It is found as the *np* when coming from the pixel (4, 5), hence in direction *curr_dir* = 7. Its label is therefore updated with *L*. Save *prev_dir* = *curr_dir* and set *cp* = *np*. Because the

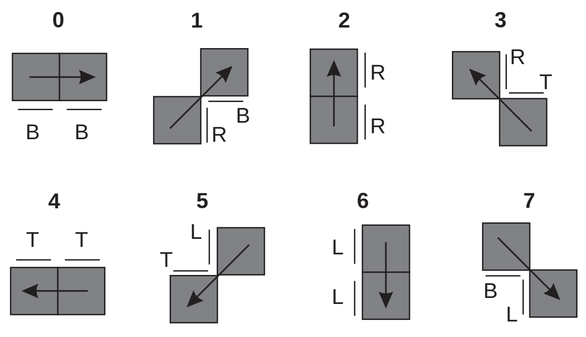


Fig. 3. Single direction (*curr_dir*) based labeling rules. Both the current pixel (*cp*) and the next pixel (*np*) are labeled in each case.

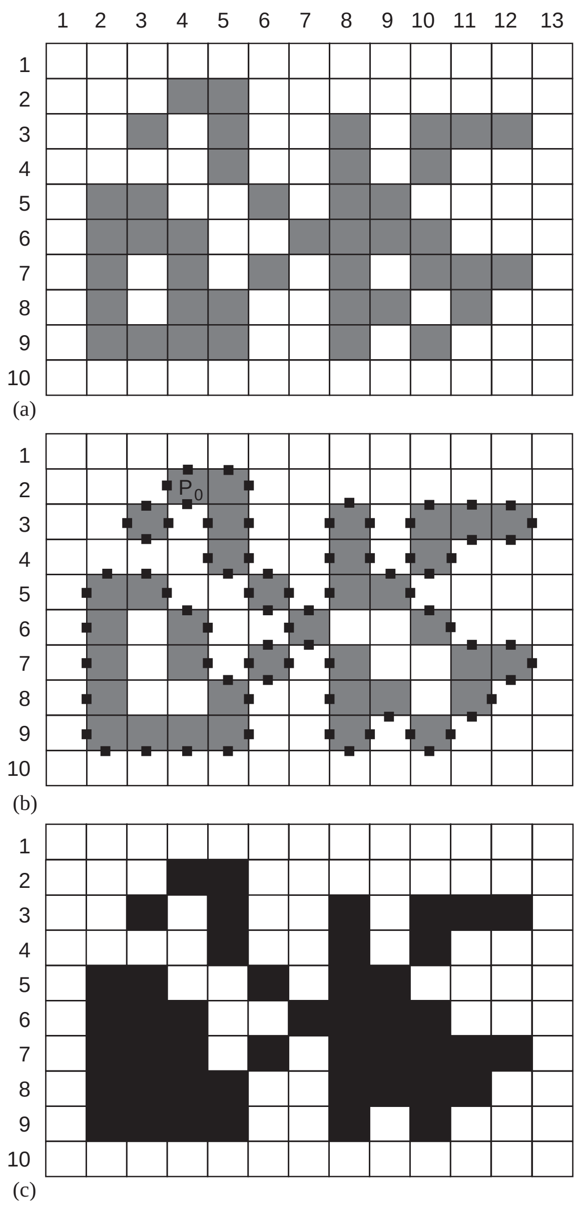


Fig. 5. An object consisting of a concave main body (with holes) and several other branches (a), its labeled inner contour (b) and the result of the filling (c).

curr_dir is odd, the direction is updated using the rule *curr_dir* $= (curr\_dir + 6) \bmod 8 = (7+6) \bmod 8 = 5$. Therefore, the search for the next boundary pixel starts at the location $(6, 5)$ which is not a boundary point. The search direction is incremented (modulo 8): *curr_dir* $= (curr\_dir + 1) \bmod 8 = (5+1) \bmod 8 = 6$. No contour pixel is found at $(6, 6)$ either. The direction is incremented again: *curr_dir* $= (6+1) \bmod 8 = 7$ and the next contour pixel is found $np = (6, 7)$. Applying the single direction-based labeling, the *cp* $(5, 6)$ is labeled B and *np*, L (see Fig. 3). Because *prev_dir* = *curr_dir* = 7, no double direction-based labeling is needed (see Fig. 4). Again, save *prev_dir* = *curr_dir* and set *cp* = *np*. The *curr_dir* is updated with the odd direction rule and the *np* is located at $(7, 6)$. Because *curr_dir* = 5, *cp* $(6, 7)$ is labeled (again) L and *np*, T. The process continues with similar direction updates and labeling decisions until P_0 is met again. Recall the discussion about P_0 of this example OOI given in the beginning of the section.

3. Conclusion and discussion

A linear-time algorithm for filling regions with closed contours in raster format was presented. The only overload of the method is an auxiliary matrix of equal size to OOI for storing the labels (e.g., a four bit depth matrix). However, recall that only two labels (L and R) are needed for filling and therefore a two bit depth matrix would be enough. Also, the exclusion of the T and B labeling rules: the algorithm. Moreover, a linked list can be used to store the labeled contour instead of a matrix. In this manner, the extra space required by the algorithm becomes linear to the length of the perimeter of the OOI instead of the area of its surrounding box. However, during contour traversal and labeling, multiple search operations in the list must then be performed in order to locate, update or insert the neighbors of a given contour pixel. This means that such an optimization of the storage space is paid in the form of increased processing time and its benefit should be evaluated against the kind of the desired system and the complexity of the target OOIs.

The strength of the proposed algorithm is that it is not prone to classical cases of exceptional topological configurations that require additional investigation and thus, computation [1,2,12]. However, the operation of the algorithm depends critically on the 8-connected contour tracing. While its correctness has not been theoretically proved and remains intuitive, it seems to be hard to find special cases where the tracing algorithm would fail. We include in Appendix B four OOIs to exemplify both the results of the filling and the complexity of our test images.

The proposed method can also be used for coloring OOIs with a number of other overlapping OOIs that need to be preserved. The algorithm can handle a hierarchy of overlapping OOIs provided that the (inner) contour detection and tracing is performed on the basis of the color of the objects. No pair of objects can then share any boundary pixels, which means that the region-filling algorithm handles them individually.

Appendix A. Contour labeling and region-filling algorithms

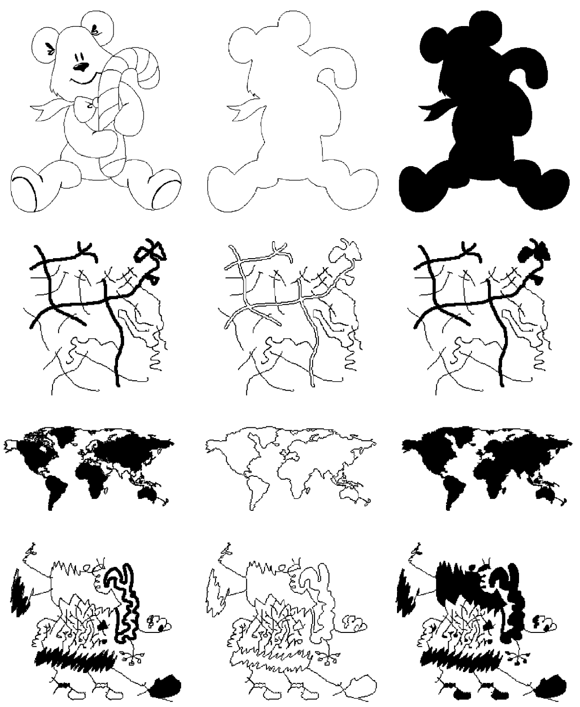
The input of the algorithm is an image (restricted here to 256 × 256 pixels, IM) containing a single OOI of a known color (*old_color*). The algorithm fills the interior of the OOI with a *new_color*. The additional label matrix (LM), of equal size to the initial image, contains at the end the labeled contour of the OOI.

```
# define R 1
# define T 2
# define L 4
# define B 8
# define xMax 256
# define yMax 256

struct {
    int x, y;
  } p0,cp,np, direction[8] = {{0, 1}, {−1, 1}, {−1, 0},{−1, −1},
  {0, −1}, {1, −1}, {1, 0}, {1, 1}};
byte IM[xMax][yMax]; //initial image, containing the OOI
byte LM[xMax][yMax]; // the label matrix
int old_color, new_color; // old and new OOI colour - somehow initialized!
int top_P0; // stores the original color of the pixel at top of P0
byte curr_dir, prev_dir; // current and previous search direction
byte found_next_pixel; // flag
byte stop_fill; // flag used for filling between L–R border points
byte sw_p0 = 1;// flag for the contour starting point P0
int i, j; //index variables
old_color = 0; new_color = 128; //for example

//- - contour tracing part- - - - - -
i = j = 0;
sw_p0 = 1;
while (sw_p0) //search for first boundary pixel P0.
  {
    if (IM[i][j] == old_color)
        sw_p0 = 0;
      else
        if (j<yMax)
          j++;
        else
          if (i<xMax)
            { i++; j = 0;}
    } // end while
p0.x = i−1; p0.y = j; // P0 is taken on top of the actual P0,
    // to correct for the case discussed in Fig. 5b
top_P0 = IM[p0.x][p0.y];
IM[p0.x][p0.y] = old_color;

// - - - start traversing the contour - - -
cp.x =  p0.x; cp.y =  p0.y; // set current pixel to P0
np.x =  p0.x+1; np.y =  p0.y; // next pixel
curr_dir = 7; //initialize search direction

while ( np.x! = p0.x || np.y! = p0.y ) // search until returns to P0
    {
    found_next_pixel = 1;
    prev_dir = curr_dir; // store previous direction
    curr_dir  = (curr_dir+6+((curr_dir+1)%2))%8;
        // update search direction: if odd, add 6; if even, add 7
```

```
   do{ //do
     np.x = cp.x+direction[curr_dir].x; // new point coordinates
     np.y = cp.y+direction[curr_dir].y;

     if ( IM[np.x][np.y] = = old_color ) //next contour point found
       {
         found_next_pixel = 0;

 //- - - contour labeling - - -
         if ( curr_dir == 0 )
           { LM[cp.x][cp.y] = LM[cp.x][cp.y] | B;
           LM[np.x][np.y] = LM[np.x][np.y] | B;

           if ( prev_dir == 4 || prev_dir == 5 )
             LM[cp.x][cp.y] = LM[cp.x][cp.y] | L;
           } // - - - 0
         else
         if ( curr_dir == 1 )
           { LM[cp.x][cp.y] = LM[cp.x][cp.y] | R;
           LM[np.x] [np.y] = LM[np.x][np.y] | B;

           if ( prev_dir == 6 || prev_dir == 7 )
             LM[cp.x][cp.y] = LM[cp.x][cp.y] | B;
           if ( prev_dir  == 5 )
             LM [cp.x] [cp.y] = LM [cp.x] [cp.y] | (L+B);
           } // - - - 1
         else
         if ( curr_dir == 2 )
           { LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | R;
           LM[np.x] [np.y] = LM[np.x] [np.y] | R;

           if ( prev_dir == 6 || prev_dir == 7)
             LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | B;
           } // - - - 2
         else
         if ( curr_dir == 3 )
           { LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | T;
           LM[np.x] [np.y] = LM[np.x] [np.y] | R;

           if ( prev_dir == 0 || prev_dir == 1 )
             LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | R;
           if ( prev_dir == 7 )
             LM [cp.x] [cp.y] = LM [cp.x] [cp.y] | (R+B);
           } // - - - 3
         else
         if ( curr_dir == 4)
           { LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | T;
           LM[np.x] [np.y] = LM[np.x] [np.y] | T;

           if ( prev_dir == 0 || prev_dir == 1 )
             LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | R;
           } // - - - 4
         else
         if ( curr_dir == 5 )
           { LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | L;
           LM[np.x] [np.y] = LM[np.x] [np.y] | T;

           if ( prev_dir == 1 )
             LM [cp.x] [cp.y] = LM [cp.x] [cp.y] | (T+R);
```

```
          if ( prev_dir == 2 || prev_dir == 3 )
            LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | T ;
            } // - - - 5
          else
          if ( curr_dir == 6 )
            { LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | L;
            LM [np.x] [np.y] = LM [np.x] [np.y] | L;

          if ( prev_dir == 2 || prev_dir == 3 )
            LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | T ;
            } // - - - 6
          else
          if ( curr_dir == 7 )
            { LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | B;
            LM[np.x] [np.y] = LM[np.x] [np.y] | L;

            if ( prev_dir == 3 )
              LM [cp.x] [cp.y] = LM [cp.x] [cp.y] | (L+T);
            if ( prev_dir == 4|| prev_dir == 5 )
              LM[cp.x] [cp.y] = LM[cp.x] [cp.y] | L;
            } // - - - 7
        cp.x = np.x; cp.y = np.y; //update the current pixel
      } // end if next contour point found
    else
      curr_dir = (curr_dir+1)%8;
  } while ((found_next_pixel)); // do
}// end main while – until returns to P0

IM[p0.x] [p0.y] = top_P0;//restore the modified pixel in the original image
LM[p0.x] [p0.y] = 0; // delete the extra point on top of the actual P0
LM[p0.x+1] [p0.y] = LM[p0.x+1] [p0.y] |T; // label the top edge of P0

// - - - the actual filling - - - -
for (i = 0; i<xMax ; i++) // scan all lines
  { j = 0;
   do
     { if (LM[i][j])
         IM[i][j] = new_color; //color the border pixels themselves

       if (( (LM[i][j]>>2)%2 ) && (!( LM[i][j]%2 )) )
         { // if it is an L point but not R, start filling
            stop = 1;
            while ( stop )
              { j++;
                if ( LM[i] [j] %2 )
                  stop = 0; // stop at the first R point
                IM[i] [j] = new_color;
              }
          }
          else
              j++;
  } while (j<yMax); // end while j
} // end for i, scan all lines
```

Appendix B. Four sample OOIs, their contours and the filling results



References

[1] Pavlidis T. Filling algorithms for raster graphics. Computer Graphics and Image Processing 1979;10:126–41.

[2] Brassel KE, Fegeas R. An algorithm for shading of regions on vector display devices. In: Proceedings of the sixth annual conference on computer graphics and interactive techniques, Chicago, United States, 1979. p. 126–33.

[3] Lane JM, Magedson R, Rarick M. An algorithm for filling regions on graphics display devices. ACM Transactions on Graphics 1983;2(3):192–6.

[4] Pavlidis T. Algorithms for graphics and image processing. Berlin: Springer; 1982.

[5] Burtsev SV, Kuzmin YP. An efficient flood-filling algorithm. Computers & Graphics 1993;17(5):549–61.

[6] Hearn D, Baker MP. Computer graphics, C version, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall; 1997.

[7] Tsai YH, Chung KL. Region-filling algorithm on bincode-based contour and its implementation. Computers & Graphics 2000;24(4):529–37.

[8] Lejun S, Hao Z. A new contour fill algorithm for outlined character image generation. Computers & Graphics 1995;19(4):551–6.

[9] Gargantini I, Atkinson HH. Linear quadtrees: a blocking technique for contour filling. Pattern Recognition 1984; 17(3):285–93.

[10] Albert TA, Slaaf DW. A rapid regional filling technique for complex binary images. Computers & Graphics 1995; 19(4):541–9.

[11] Henrich D. Space-efficient region filling in raster graphics. Visual Computer 1994;10(4):205–15.

[12] Schachter B. Decomposition of polygons into convex sets. IEEE Transactions on Computers 1978;27(11):1078–82.

[13] Yao C, Rokne J. Applying rounding-up integral linear interpolation to the scan-conversion of filled polygons. Computer Graphics Forum 1997;16(2):101–6.

[14] Kong TY, Roscoe AW, Rosenfeld A. Concepts of digital topology. Topology and its Applications 1992;46(3): 219–62.

[15] Kovalevski VA. Discrete topology and contour definition. Pattern Recognition Letters 1984;2:281–8.

[16] Rosenfeld A. Connectivity in digital pictures. Journal of the Association for Computing Machinery 1970;17(1): 146–60.

[17] Khalimsky E, Kopperman R, Meyer PR. Computer graphics and connected topologies on finite ordered sets. Topology and its Applications 1990;36(1):1–17.

[18] Thürmer G. Closed curves in n-dimensional discrete space. Graphical Models 2003;65(1–3):43–60.

[19] Malgouyres R. Graphs generalizing closed curves with linear construction of the Hamiltonian cycles—parameterization of discretized curves. Theoretical Computer Science 1995;143:189–249.