*"The Visual Computer: An International Journal of Computer Graphics", vol 10, no 4, pp 205-215, 1994.*

# Space-efficient Region Filling in Raster Graphics

Dominik Henrich

Institute for Real-Time Computer Systems and Robotics
University of Karlsruhe, Kaiserstrasse 12, D-76128 Karlsruhe, Germany
email: dhenrich@ira.uka.de

## Abstract

*This paper presents fill algorithms for boundary-defined regions in raster graphics. The algorithms require only a constant size working memory. The methods presented are based on the so-called "seed fill" algorithms using the internal connectivity of the region with a given inner point. Basic methods as well as additional heuristics for speeding up the algorithm are described and verified. For different classes of regions, the time complexity of the algorithms is compared using empirical results.*

Key words: seed filling, graphic processors, frame buffer operations, display algorithms, raster graphics

## 1 Introduction

In computer graphics the problem of region filling occurs mainly with interactive systems. For example, the user of a drawing application clicks with the mouse into a region of the drawing and the application is asked to shade this region instantly. Because of the high demands on speed in this area, more and more solutions with special graphic processors are considered. Since the chip area of the coprocessor is limited, algorithms without or with only little working memory provided locally by the coprocessor are of particular interest. Here, we describe and verify basic algorithms and additional heuristics which meet this constraint.

The regions to be filled can be described in different ways. For instance, the region can be given by a polygon with the edges defining the boundary of the region (Little and Heuft 1979, Brassel and Fegeas 1979). Another possibility is to define a region as a group of adjacent, connected pixels. If the values of the pixels defining the region are all the same, then the region is *interior-defined*. Otherwise the boundary pixels have the same value and the region is *boundary-defined* (Foley and van Dam 1990). In both cases, the values of all other pixels are random. In this paper we focus on boundary-defined regions, but interior-defined regions work analogously.

Considering regions defined by pixels, most of the fill algorithms have a seed point as input. The seed point is a pixel known to lie in the interior of the region. The so-called *seeding algorithms* start at the seed point and proceed with other neighbouring pixels according to some strategy. Only pixels in the interior of the region can be changed to the filling value. Thus, these pixels are "connected" to the seed, because they can be reached them from the seed point without leaving the region's interior.

With respect to the use of memory, the known filling algorithms can be subdivided into two groups. In one group, an extra "working plane", in addition to the frame buffer, is used for filling. After filling, it is copied to the frame buffer itself. In between, a pattern mapping may be applied (Ackland and Weste 1981, Dunlavey 1983, Tang and Lien 1988). In the other group,

an additional data structure is used. In most cases, it is a stack which stores several inner pixels if the region is split up while filling. In the following steps, each of these inner pixels serves as a new seed point for filling a separate part of the region (Newman and Sproull 1973, Liebermann 1978, Smith 1979, Shani 1980, Pavlidis 1982). In general, the methods of both groups need working memory in addition to the frame buffer. The amount of working memory used depends on the size of the region (increases with region complexity) and thus cannot be constant.

In this paper, we describe algorithms which need little additional memory, which is of constant size, and thus can be reserved in advance. To comply with this severe restriction, the connectivity of the inner pixels must be stored in a different way than in the former algorithms. For this purpose, it is necessary to use the frame buffer itself. Instead of storing the information explicitly in an extra working memory, it is kept implicitly in the existing frame buffer. This is possible, because each inner pixel has the property of being connected with the seed point. Thus, the value of a pixel may be changed, provided that all the other inner pixels remain connected. Roughly speaking, the global filling strategy is: *move around in the region and fill it in such a manner that the region remains connected*. With this strategy, no memory is needed for storing region parts because the algorithm does not divide the region into parts.

Following Fishkin and Barsky (1984), a filling algorithm can be logically decomposed into four components: a *start procedure*, which initialises the algorithm; a *set procedure*, which changes the value of a pixel; a *propagation method*, which determines the next point to be considered; and an *inside procedure*, which computes whether a pixel is in the region and should be filled. To implement our global filling strategy and to solve the problems arising from it, some of these components are discussed in Section 2. Using different instances of the components, we formulate algorithms for different region types. Basic methods for simple and general regions are discussed and improved by heuristics in Section 3. To evaluate the performance of the algorithms, they are compared using experiments with regions of different complexity. The results of these experiments are shown in Section 4.

## 2 Basics

For the above mentioned global fill strategy, this section concentrates on two basic procedures and introduces some theorems. The first procedure executes movement within the region. Different rules for selecting the next direction control the moves. The second procedure computes whether a pixel can be changed without losing necessary information. It is a local decision which is sufficient in most cases. The last subsection distinguishes two different types of regions. They lead us to the two basic fill algorithms of the next section. We begin with a more precise specification of the problem, followed by a description of the data structures used.

### 2.1 Problem Definition

For a precise definition of filling with constant working memory, some terminology has to be introduced first. In the following, we use a 2-dimensional **bitmatrix** as an abstract model of the frame buffer. It represents the relevant information of the display by orthogonally arranged elements. These picture elements (pixels) of the bitmatrix can take two alternative states, **set** or **free**. For example, black and white could be modelled as such states with a monochrome display – other colour pairs are applicable with multicoloured displays. Additionally, the graphic coprocessor, which stores and executes the fill algorithm code, has random access to the bitmatrix. It can read or write the state of a pixel in constant time using the procedures *ReadPixel* and *WritePixel*.

The pixel neighbourhood is defined as follows ( cf. Pavlidis (1982) and Rosenfeld, Kak (1982)).

Definition 1:

Two distinct pixels of a bitmatrix are called **4-adjacent** (4-neighbours) if they have a common edge in the raster of the bitmatrix (see Fig.1a), and **8-adjacent** (8-neighbours) if they have a common edge or a common vertex (see Fig.1b).


*Fig.1: In (a) all the 4-neighbours, and in (b) all the 8-neighbours of the free pixel* p *have the state "set".*

A 4-adjacent neighbour can be reached by any of the one-pixel-moves: up, down, left, right. For an 8-adjacent neighbour horizontal, vertical and diagonal moves are allowed. Thus, a pixel has four 4-adjacent and eight 8-adjacent neighbours. Note that all 4-neighbours are also 8-neighbours. Now the region filling problem can be specified.

Problem: *Region filling with constant working memory*

*Input*: a connected, finite region consisting of free, 4-adjacent pixels of a bitmatrix in a frame buffer, with a boundary contour of 8-adjacent pixels, which are set. Outside the region, the pixel states are random. With this region a seed point is given which marks its interior. Additionally, the available working memory is limited by a given constant, independent of the region size.

*Output*: a change of the bitmatrix so that all the inner pixels of the region are in the state "set".

The contour of the region can consist of several non-connected parts, i. e. the region can have holes not belonging to the interior.

### 2.2 Data Structures

Following the general seeding algorithms, the seed point is the only inner pixel known in the beginning. Starting with it, further inner pixels are identified by a so-called **cursor**. During the filling process, the cursor keeps the coordinates of one inner pixel of the region. Because a 4-adjacent, free neighbour also lies in the interior of the region, the cursor can be shifted onto it without a loss of information or without leaving the region. By stepwise shifting, the cursor can scan the whole region. There are different strategies for moving the cursor within the region, which are discussed in Section 2.3.

Another basic data structure is the **window**. It is used to reduce the number of frame buffer accesses. For most filling operations, a 3x3-section of the frame buffer provides sufficient information. Therefore, a window stores the states of the 8-neighbours around the cursor in local memory. Fig.2 shows the window with coordinates of the 8-neighbours relative to the cursor. This window is the minimum information needed to decide how the cursor's pixel influences the topology of the region. It is a local view of the fill algorithms for the region. To get a more global view, further investigations are necessary, cf. Subsection 3.2.

| | | |
|---|---|---|
| h (-1,1) | a (0,1) | b (1,1) |
| g (-1,0) | Cursor (0,0) | c (1,0) |
| f (-1,-1) | e (0,1) | d (1,-1) |

*Fig.2: The cursor and its window for storing the states of the 8-adjacent neighbours (here with Boolean variables* a *to* h *and relative coordinates).*

## 2.3 Cursor Movement

To fill the region successively, the cursor has to move around and reach the parts not yet filled. Therefore, it is shifted from the current pixel onto one of the neighbours. To prefent the cursor from leaving the 4-adjacent region, the only moves allowed are to its 4-neighbours. Hence, the possible directions are: up, down, left, right. After a move, the next direction is selected depending on the window and the last move.

All of the algorithms described in Section 3 require that the cursor always reach the free pixels. In addition to this, with the general filling algorithm, the cursor has to return to its start position to break cycles. For moving strategies "most right" or "most left" both requirements are accomplished. When moving "most right", one chooses the next free pixel in counterclockwise order beginning at the right. To precisely describe this strategy ("most left" works similarly) the following definition is introduced and illustrated in Fig.3.

Definition 2:

A sequence of three free pixels *(p, q, r)* is called **right-connected** if

(1) *p, q* are 4-adjacent and *q, r* are 4-adjacent and

(2) *r* is the next counterclockwise 4-adjacent free neighbour of *q* after *p*.


*Fig.3: Some examples for right-connected free pixels* (p, q, r) *and the corresponding moving directions.*

With the "most right" strategy, the priority of directions relative to the last direction is right, straight, left and back. Thus, if the last move was from pixel *p* to pixel *q*, then this strategy selects a direction to the pixel *r*, so that the triple *(p, q, r)* is right-connected. In total, the cursor walks along a path with only right-connected triples of pixels.

The cursor's movement with the strategy "most right" is implemented by two procedures. The procedure *WalkRight(w,d,stop)* chooses the direction which points to the right-connected pixel. It depends on the pixel states in the window *w* and the last direction, *d*, moved. If all the neighbours of the cursor are set, then the Boolean variable *stop* returns the value "true". Otherwise, *d* stores the new direction of movement. The second procedure *Shift(p,w,d)* displaces the cursor *p* one pixel in the new direction *d* and updates the window *w*. For that purpose, three read operations by *ReadPixel* are necessary, because now the 3x3-window is

placed on three new pixels with unknown values. Altogether, alternate calls of these two procedures make the cursor walk around the interior of the region on a well specified path.

### 2.4 Setting Pixels

While the region is being filled, the current pixel stored by the cursor is investigated. If it does not connect two parts of the region, then it can be changed to the state "set". Otherwise it has to stay free so that it does not destroy the connectivity. Whether or not a destruction is possible is indicated by its eight neighbours, as we will see at the end of this subsection. At first, we have to define some properties concerning the connectivity. The characteristic property of two interior pixels is that they are connected by a sort of link. Such a link is set up by a path of 4-adjacent pixels from one end to the other. This property is illustrated in Fig.4 and formulated precisely in the following definition (cf. Rosenfeld and Kak 1982).

Definition 3:

Two inner pixels *p, q* are called **4-connected** if a sequence of pixels $S = (s_1, ..., s_n)$ exist with:

(1) $s_1 = p$ and $s_n = q$,

(2) for all $i = 1, ..., n$-1 holds: $s_i$, $s_{i+1}$ is 4-adjacent or $s_i = s_{i+1}$ and

(3) for all $i = 1, ..., n$ holds: $s_i$ inner pixel.

The sequence of pixels *S* is called the **4-path** between *p* and *q*.

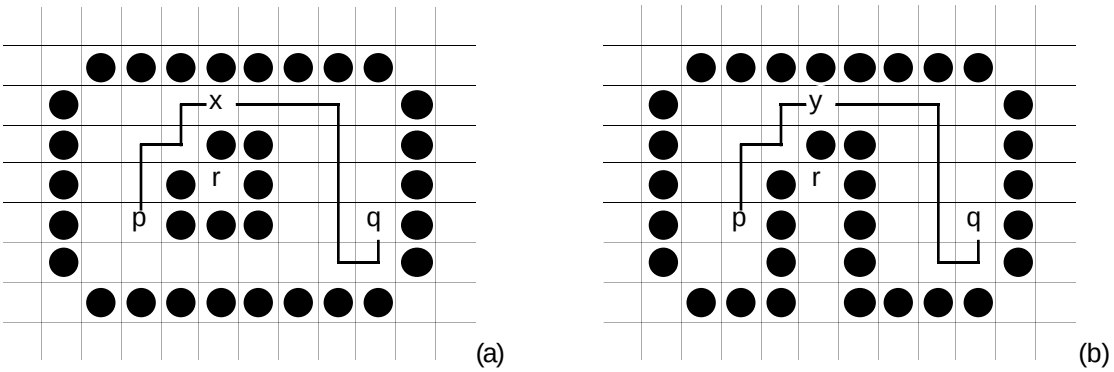


*Fig.4: The pixels* p, q *are 4-connected, because there exists a 4-path (pixels on the line) connecting them, but the pixels* p, r *or* r, q *are not 4-connected. The two pixels* x *and* y *are critical, but only* y *actually divides the region into two parts.*

For our global filling strategy (see Section 1), filling a dividing pixel will cut the cursor off from an unfilled area. With the property "4-connected" to hand, it is straight forward to formulate whether a certain pixel actually divides the region into parts or not. This is done in the next two definitions.

Definition 4:

An inner pixel *p* is called **dividing** for a region *R* if there exist pixels *q, r* such, that *p* is contained in all 4-paths of *R* between *q, r*. An inner pixel *p* is called **critical** if it is dividing for the section *W* of the region consisting of *p* and its free 8-neighbours (window).

A dividing pixel is the only connection between two (or more) sections of a region. Every path from one part to the other leads via this pixel. This is a global property – in contrast to "critical". If a pixel is critical, then we only know something about the pixel values in the window. But the property "critical" is the precondition for a pixel dividing the region into parts. See pixels *x* and *y* in Fig.4. These two properties are related as follows:

Theorem 1:

If pixel *p* is not critical, then *p* is not dividing.

*Proof*: Let *p* be an inner pixel which is not critical. Then *p* can not divide the 8-neighbours of *p*. Thus, for all pairs *q, r* of 8-neighbours, there exists a 4-path without *p*. Assume that *p* is dividing. Then there exist inner pixels *s, t* so that all 4-paths between *s, t* contain *p*. Let $(w_1, ..., w_n)$ be such a 4-path with $w_1=s$, $w_n=t$ and $w_i=p$ for one *i*. According to the precondition, there exists a 4-path without *p* between $w_{i-1}$, $w_{i+1}$. Let $(v_1, ..., v_m)$ be such a 4-path. Then $(w_1, ..., w_{i-1}=v_1, ..., v_m=w_{i+1}, ..., w_n)$ is a 4-path not containing *p*. Thus *p* is not dividing which, contradicts the assumption at the beginning. ◊

Hence, by inspecting the direct neighbours of a pixel we can decide whether it is safe to set it. Of course, there are cases where pixels are critical but not dividing. But these cannot be recognised by inspecting only the eight neighbours. More extensive investigations are necessary in this case, cf. Subsection 3.2.

Based on the eight neighbours of a pixel, a Boolean function *Critical(w)* is defined. It determines whether this pixel is critical or not. The states of the eight neighbours in the window *w* are used as parameters. With eight parameters, the value range has a size of $2^8 = 256$ values. By viewing all cases in a Karnough-Veitch Diagram (KV Diagram, Fig.5) a minimal conjunctive expression can be calculated. The value "true" (1) corresponds to the state "set" of the pixel. With a 0 in the KV Diagram, the cursor's pixel can be set because it is not critical for the 8-neighbours and therefore, according to Theorem 1, not dividing. For example, the left window of Fig.6a has the variable values *a*=1, *b*=1, *c*=1, *d*=0, *e*=1, *f*=1, *g*=0, *h*=1. It corresponds to the entry in the 12th row and the 14th column of the KV Diagram. In this field, the KV Diagram has the value "false" (0). With that, the cursor pixel has the property "non-critical". See Fig.6 for further examples.

| a | c | e | g | h | b | d | f | h | h | f | h | h |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 0 0 1 0 | 1 1 1 0 | 1 1 1 1 | 1 1 1 0 |
| | | | | | 0 0 0 0 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 |
| | | | g | | 0 0 0 0 | 0 0 0 0 | 1 1 1 1 | 1 1 1 1 |
| | | e | | | 0 1 1 0 | 0 1 1 0 | 1 1 1 1 | 1 1 1 1 |
| | | | | | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 |
| | | | g | | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
| | | | | | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 |
| | c | | | | 0 1 1 1 | 1 1 1 0 | 0 1 1 1 | 1 1 1 0 |
| | | | | | 0 0 1 1 | 1 1 0 0 | 0 0 1 1 | 1 1 0 0 |
| | | | g | | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
| | | | | | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
| a | | e | | | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
| | | | | | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 1 1 |
| | | | g | | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 |
| | | | | | 0 0 0 0 | 1 1 1 1 | 1 1 1 1 | 0 0 0 0 |
| | | | | | 0 0 1 1 | 1 1 1 1 | 1 1 1 1 | 1 1 0 0 |

*Fig.5: KV Diagram for computing whether the pixel of the cursor is critical. Variables* a *to* h *are the states of the 8-adjacent neighbours from Fig.2.*


*Fig.6: Examples for window settings; (a):* p *is not critical; (b):* p *is critical (probably dividing the region into two, three and four parts).*

## 2.5 Region Types

Before we describe the fill algorithms themselves, we have to specify for which types of regions they are applicable. At least two types of regions can be distinguished by their topological properties. When regarding discrete regions, these properties can be characterised similar to those of continuous point sets (Cullen 1968). First, a relationship between two neighbouring paths is defined.

Definition 5:

Two 4-paths $P = (p_1, ..., p_n)$ and $Q = (q_1, ..., q_m)$ are called **4-adjacent** if

(1) $p_1 = q_1$ and $p_n = q_m$,

(2) for all $p_i$ there exists a $q_j$ with $p_i$, $q_j$ 4-adjacent or $p_i = q_j$, and

(3) for all $q_j$ there exists a $p_i$ with $p_i$, $q_j$ 4-adjacent or $p_i = q_j$.

With two 4-adjacent paths, for each pixel of both paths there exists a corresponding pixel in the other path that is a 4-neighbour or identical with the first pixel. Thus, given an arbitrary 4-path, we obtain a second, 4-adjacent path by replacing some pixels by their 4-neighbours and/or by adding some 4-neighbours. Thereby, the resulting pixel sequence still represents a 4-path. Repeating this substitution several times with the second path, we obtain two 4-homotop paths. Thus, the 4-homotop paths are the last paths of a sequence of 4-adjacent paths. See Fig.7 and the following definition.

Definition 6:

Two 4-paths $P = (p_1, ..., p_n)$ and $Q = (q_1, ..., q_m)$ are called **4-homotop** if there exists a sequence of 4-paths $K_1, ..., K_k$ with:

(1) $K_1 = P$ and $K_k = Q$ and

(2) $K_i$, $K_{i+1}$ are 4-adjacent, for all $i = 1, ..., k$-1.

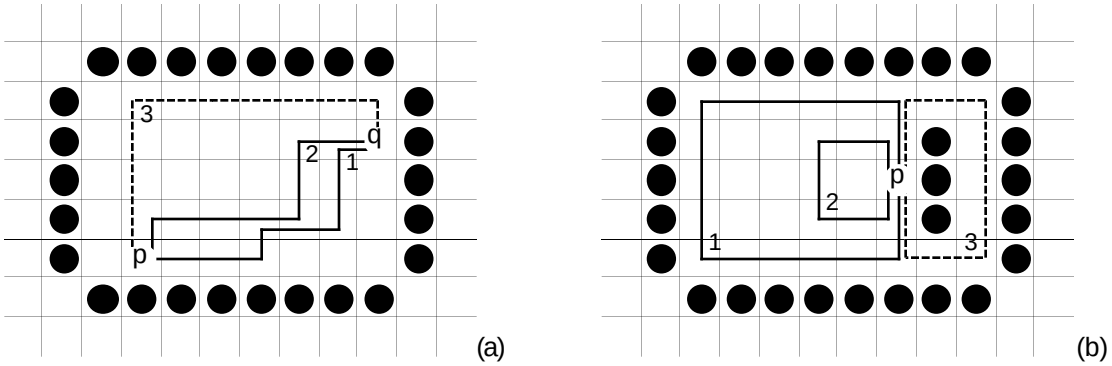


*Fig.7 (a): Path 1 and 2 from pixel* p *to* q *are 4-adjacent; path 3 is 4-homotop but not 4-adjacent to paths 1 or 2; (b): Paths 1 and 2 are 4-homotop and contractible to pixel* p. *Path 3 is neither 4-homotop to paths 1 or 2 nor contractible to* p.

A special case exists, if one of the two 4-homotop paths consists of only one pixel. Let this pixel be an element of the first path, then this path is possibly contractible to one of its own elements. See Definition 7 and the example in Fig.7b.

Definition 7:

A 4-path $P = (p_1, ..., p_n)$ with $p_1 = p_n$ (closed 4-path) is called **contractible** to the point $p_1$ if it is 4-homotop for the 4-path $(p_1)$.

Now, with these tools at hand, we can distinguish two fundamental types of regions.

Definition 8:

A region $R$ is called **simple** if every closed 4-path in $R$ is contractible to one pixel, and **cyclic** if the region is not simple.

Cyclic regions have holes which do not belong to the interior (e. g., the three pixels in Fig.7b). Closed 4-paths around those holes cannot be contracted, without leaving the interior. This fact is important for the termination of the first algorithm in the next section. Therefore, in the following theorem, we prove a relation between the kind of path used by the cursor and the type of region.

Theorem 2:

If $P = (p_1, ..., p_n)$ is a closed 4-path consisting of right-connected, critical pixels, then $P$ is not contractible to $p_1$.

*Proof*: Assume that $P$ is contractible to $p_1$. Let $T$ be a region with $P$ as contour, and a finite area as interior. Then all inner pixels of $T$ are not set. Now we have to show that there is at least one non-critical pixel on $P$.
Case 1: An inner pixel of $T$ does exist. Let $q$ be an inner pixel of $T$ with $p_i$ from $P$ as an 8-neighbour. Since all pixels of $P$ and $T$ are free, all 8-neighbours of $q$ are free. Thus, an 8-neighbour $p_j$ in $P$ of $q$ exists which is not critical, hence resulting in a contradiction.
Case 2: No inner pixel of $T$ exists. Then either a "return point" $p_i$ with $p_{i-1} = p_{i+1}$ or a square "return loop" $(p_{i-1}, p_i, p_{i+1}, p_{i+2})$ with $p_j$, $p_k$ 8-neighbours ($j$, $k$ in $\{i-1,...,i+2\}$) exists. Hence, $p_i$ is not critical, which contradicts the assumption. ◊

# 3 Fill Algorithms

In this section, different fill algorithms with constant working memory are presented. They use the basic procedures described in the previous section. The first algorithm applies only to simple regions, i. e. regions without holes. As a generalisation, the second algorithm operates on arbitrary regions. The next two fill methods are variations of the general algorithm. There, additional heuristics are used to speed up the algorithms.

## 3.1 Simple Regions

We start with the most primitive fill algorithm and take a seed point as input. With this algorithm it is assumed that the seed point lies in a simple, finite, 4-adjacent region. As initialisation, *InitFill(p,w,d,stop)* shifts the seed point $p$ and the window $w$ to a boundary of the region. Additionally, the direction of movement $d$ is set as if the cursor is just along the contour. After initialisation, the cursor $p$ moves and sets all non-critical pixels until no path is free, using the basic procedures *Critical*, *WalkRight* and *Shift* of Section 2. *Critical* determines if a pixel can be set without losing information. *WalkRight* and *Shift* make the cursor walk around in the region. The algorithm *SimpleFill*, here stated in Pascal, is the direct implementation of the global fill strategy mentioned in the introduction.

```
procedure SimpleFill (p: point);
(*INPUT:   point p within a simple, 4-adjacent region*)
(*OUTPUT:  region filled using limited working memory and no pattern*)
var  w     :window;                          (*3x3-section of frame buffer*)
     d     :direction;                       (*moving direction*)
     stop  :boolean;                         (*true, if no direction is free*)
begin
     InitFill(p, w, d, stop);                (*move to contour*)
     while not stop do begin
          if not Critical(w) then begin
               w[0,0] := true;               (*write to local memory*)
               WritePixel(p);                (*write to frame buffer*)
          end;
          WalkRight(w, d, stop);             (*choose new direction*)
          Shift(p, w, d);                    (*move cursor and update window*)
     end;
end;
```

In Fig.8a we see how *SimpleFill* works on a simple region, i. e., a region without holes. From the seed point, the initialisation moves the cursor to the right contour. There, the cursor turns upwards, just if it were moving along the contour and moves three steps filling non-critical pixels. At the top, the cursor turns most right and moves though a channel to the right part of the region. In this channel, no pixels can be set, because all of them are evaluated to be critical. In a sort of loop, the algorithm can fill the right part of the region and come back through the channel. Now these pixels are not critical anymore, and therefore, not dividing, so they are set. In the left part of the region, *SimpleFill* proceeds analogously: it sets pixels provided that they are not critical.

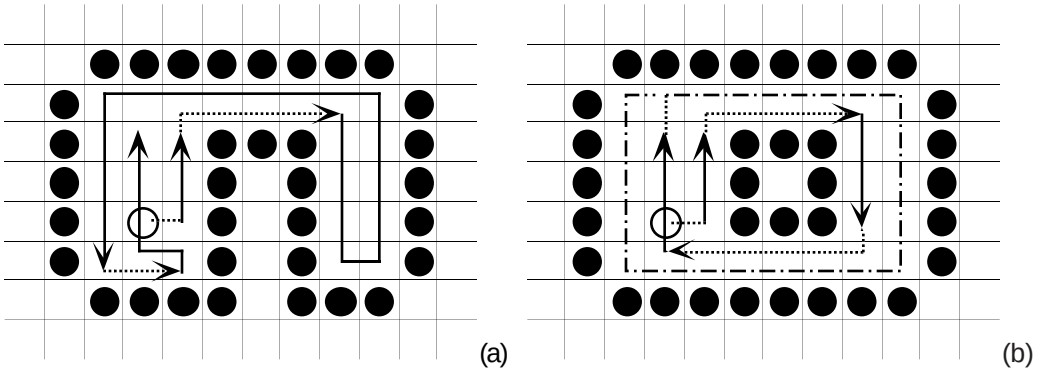


*Fig.8: Example for* SimpleFill *working on a simple region (a) and a cyclic region (b). The seed point is the outlined pixel. The arrows show the cursor's movement while setting pixels (normal lines) or leaving them unchanged (dotted lines). In (b) an endless loop is indicated by the dash-dotted line.*

Partial correctness: By definition, the seed point lies in the interior of the region. In *InitFill*, *WalkRight* and *Shift*, the cursor stays in the inner part. Thus, on the path only inner pixels can be processed. All pixels which are set have the property non-critical, which means, while pixels are set, all free inner pixels remain connected with the cursor (Theorem 1). If the algorithm terminates, i. e., the cursor is not in a cycle, no free inner pixel exists and the region is filled.

Termination: Because the region is finite, the algorithm must either terminate or the cursor remains in a cycle of critical pixels. Such a cycle exactly corresponds to a closed right-connected 4-path, cf. Definitions 2 and 3. According to Theorem 2, such a path is not contractible. With that, the region would be cyclic, so the algorithm terminates.

From the partial correctness and the termination theorems we have:

Theorem 3:

The algorithm *SimpleFill* fills simple regions correctly.

If the region is not simply connected, i. e., there are holes in the region, a problem arises with *SimpleFill*. The algorithm fills the region up to a 1-pixel-wide path. It describes one or more cycles on which the cursor moves without success. On this path, no pixels can be set because all of them are evaluated to be critical. The window itself does not allow a decision as to whether parts of the region can be cut off or not by setting critical pixels. See Fig.8b for example of such an endless loop. To break that cyclic path consisting of critical pixels, a more global point of view is necessary. This is discussed in the next subsection.

### 3.2 Cyclic Regions

The algorithm *SimpleFill* enters an endless loop if cycles exist in the region, i. e., the region is not simple. For an analysis of this behaviour, we concentrate on paths that are created by the fill algorithm. Using the strategy "most right", cf. Subsection 2.3, for the cursor's movement, the possible cyclic paths have the following characteristics:

Definition 9:

A sequence of inner pixels $(z_1, ..., z_n)$ are called **right-cycle** if

(1) for all $i = 1, ..., n\text{-}2$ holds: $(z_i, z_{i+1}, z_{i+2})$ right-connected,

(2) $(z_{n-1}, z_n, z_1)$ right-connected and

(3) for all $i = 1, ..., n$ holds: $z_1 \neq z_i$.

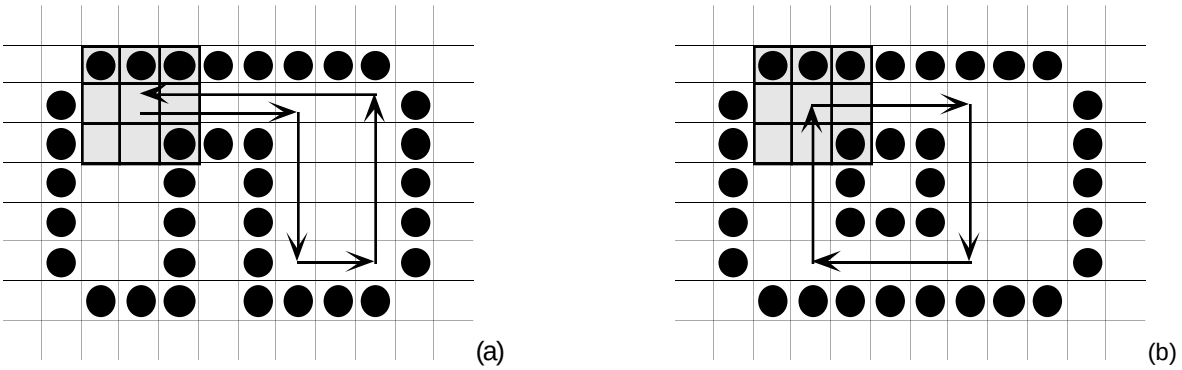


*Fig.9: Examples for right-cycles beginning at the same position with the window shaded grey; (a): non-true right-cycle; (b): true right-cycle.*

As can be seen in Fig.9, right-cycles can have very different shapes. They can be distinguished by whether they surround existing holes of the region or not. Our interest lies in right-cycles surrounding holes because only these are responsible for the problem of *SimpleFill*. A simple classification of right-cycles is obtained by exploiting the last degree of freedom in Definition 9. This freedom is the path's direction at the beginning and the end of the cycle. For example, in Fig.9a the beginning direction of the path is exactly reverse to the ending direction. In contrast, the path of Fig.9b shows directions orthogonal to each other. This is expressed in formal terms by:

Definition 10:

A right-cycle $(z_1, ..., z_n)$ is called **true** if $(z_n, z_1, z_2)$ are right-connected.

Before using this classification of right-cycles for fill algorithms, we have to prove its applicability. Given an arbitrary cursor position marking a critical pixel, then, according to the

following theorem, the type of right-cycle can be related to the importance of the questionable pixel for the region.

Theorem 4:

If $(p_1, ..., p_n)$ is a true right-cycle, then $p_1$ is not dividing for $p_n$, $p_2$.

*Proof*: If $p_1$ is not critical, then this theorem directly results from Theorem 1. Otherwise, let $(p_1, ..., p_n)$ be a true right-cycle and $p_1$ be critical. Assume that $p_1$ is dividing $p_n$, $p_2$. This is equivalent to the fact that there exist inner pixels $q$, $r$ such that $p_1$ belongs to all 4-paths between $q$, $r$. Let $(s_1, ..., s_m)$ be such a 4-path with:

$$s_1=q,\ s_{i-1}=p_2,\ s_i=p_1,\ s_{i+1}=p_n,\ s_m=r,\ p_n \neq p_2 \text{ and for all } j \neq i \text{ holds } s_j \neq p_1;$$

Thus $W := (q=s_1, ..., s_{i-1}=p_2, p_3, ..., p_{n-1}, p_n=s_{i+1}, .., s_m=r)$ is 4-connected. This insures that $p_1$ is not in $W$. Thus $p_1$ does not divide $p_n$, $p_2$, contradicting the assumption. ◊

Applying Theorem 4, the first pixel of a true right-cycle does not divide two parts of the region. Thus, by global investigation of the region, it is possible to break the problematic cycles of *SimpleFill*. The check whether a (critical) pixel is an element of a true right-cycle is performed by the function *RightCycle(p,w,d)*, cf. the appendix. It takes the cursor *p*, the window *w*, and the current direction *d* as input. At the present position of *p* a second cursor *p'* starts and walks "most right", until it returns to the start point *p*. Thus, the second cursor describes a right-cycle. Then, the starting direction is compared with the return direction. If the latter proceeds by turning "most right" to the former, then we have found a true right-cycle. Respectively, *RightCycle* returns the Boolean value. During this test, the first cursor *p* and its current direction *d* are left unchanged.

With the general fill algorithm, every critical pixel can be an element of a true right-cycle. This must be checked at the first critical pixel. If the cycle is not true, then the pixel divides one or more subregions and must not be set. With a true right-cycle, the pixel does not divide the region in the current direction. But it can divide the region in other directions. Therefore, it is only assumed as set for the next steps. In summary, the following algorithm is obtained (the differences from *SimpleFill* are underlined):

```
procedure CycleFill (p: point);
(*INPUT:   pixel p within a 4-adjacent region*)
(*OUTPUT:  region filled using limited working memory and no pattern*)
var  w     :window;                          (*3x3-section of frame buffer*)
     d     :direction;                       (*moving direction*)
     stop  :boolean;                         (*true, if no direction is free*)
begin
     InitFill(p, w, d, stop);                (*move to contour*)
     while not stop do begin
          if not Critical(w) then begin
               WritePixel(p);                (*write to frame buffer*)
               w[0,0] := true;               (*write to local memory*)
          end  else
               if RightCycle(p, w, d) then
                    w[0,0] := true;          (*assume cursor to be set*)
          WalkRight(w, d, stop);             (*choose new direction*)
          Shift(p, w, d);                    (*move cursor and update window*)
     end;
end;
```

In Fig.10 we see how *CycleFill* works on a region with one hole. As with *SimpleFill* in Subsection 3.1, the cursor moves upwards and sets three pixels after initialisation. There, the algorithm evaluates *p* to be the first critical pixel. Instead of ignoring this fact, like *SimpleFill*, the algorithm starts a second cursor by *RightCycle* moving around the hole. When the second cursor returns to *p*, the starting and returning direction are compared. If the latter proceeds by

turning "most right" to the former, then *RightCycle* evaluates to "true". Following Theorem 4, *p* is not dividing for this true right-cycle and *p* can be assumed as set. With this, the rest of the processed pixels are not critical anymore and can be set.

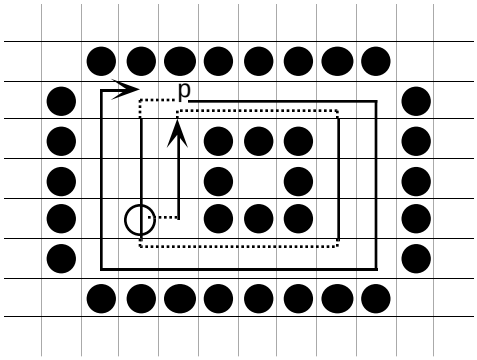


*Fig.10: An example for* CycleFill *working on a cyclic region. The outlined pixel marks the seed point. The arrows show where the cursor moves along setting pixels (normal lines) or leaving them unchanged (dotted lines). Pixel* p *indicates the start and the end of a true right-cycle.*

Partial correctness: Analogous to *SimpleFill*.

Termination: The region is finite, and at every essential step pixels are set: If a pixel is not critical, it is set, following Theorem 1. Otherwise, it is critical and checked for a right-cycle. If it is a true right-cycle, there exists at least one non-critical pixel in it, if the preceding is assumed as set (Theorem 4). If it is not a true cycle, the cursor moves into the part of the region (the current direction was not changed by *RightCycle*) and can either set pixels or break true right-cycles.

From the partial correctness and the termination theorems, it follows:

Theorem 5:

The algorithm *CycleFill* fills cyclic regions correctly.

### 3.3 Changing Direction

In this and the next subsection, two heuristics are presented for speeding up the general algorithm. They are pure supplements and do not affect the correctness or the constraint of constant working memory.

The general algorithm *CycleFill* can show high running times in unfavourable cases. In these cases, the region already contains a lot of dividing pixels or they are generated by successive filling. Dividing pixels are time consuming because they cannot be set and the cursor traverses long distances in order to break cycles. If the direction of the normal cursor movement and that for breaking cycles are the same, then the normal cursor and the auxiliary cursor in *RightCycle* can lengthen their own paths. The normal cursor sets pixels in certain parts of the region – due to this, dividing pixels are generated, too. Then, the auxiliary cursor uses the same region parts trying to break cycles. This job now needs more effort because of the increased number of dividing pixels.

To avoid this disadvantage, a heuristic states: "To break cycles, walk most left, otherwise most right". By replacing the procedure *RightCycle* by the procedure *LeftCycle*, the heuristic can easily be built into *CycleFill*. The procedure *LeftCycle* works the same as *RightCycle* except in

the reverse direction, cf. the appendix. Now the cycles are broken counterclockwise, and the region is filled clockwise. The resulting algorithm is called *LeftRightFill*.

### 3.4 Additional Memory

For the following modification, *MemoryFill*, of the algorithm *CycleFill*, another heuristic is used. It says "Save the divided parts of the region" or "Use multiple cursors". This corresponds to the methods with unlimited memory space. But here the space can be arbitrarily limited! If no memory can be allocated anymore, the algorithm keeps on running like *CycleFill*. This is a combination of a traditional seed fill algorithm with unlimited memory space and *CycleFill*.

With *MemoryFill*, one cursor starts filling the region. If it reaches a critical pixel, then intervention is necessary. Instead of breaking possibly existing cycles by the procedure *RightCycle*, a new cursor is installed to process the rest. The old cursor stops. By means of this, the critical pixel can be set to break further potential cycles. In this case, parts can also be divided, but because the old cursor is still known, they are correctly filled later on.

More and more new cursors are installed until either the current part of the region is filled or no memory space can be allocated. In the first case, one can delete the current cursor and access those still to be processed. In the second case, one checks whether all of the stored cursors are actually still used. If this is not true, then one of them can be reused. Otherwise, one must proceed with the current cursor like *CycleFill*.

The correctness of this heuristic can be seen easily: *MemoryFill* runs several *CycleFill*s in turn. Each cursor of the procedure *CycleFill* works independently. Therefore, each cursor works correctly by itself. A mutual influence is not possible because the essential steps of the single cursors are computed completely separately.

## 4 Experimental Results

In this section, the efficiency of the presented algorithms is examined. As given by the problem definition, the time complexity is measured depending on the most time-consuming operations. For common von-Neumann-computers, these are reading and writing to the frame buffer. Reading a pixel means getting its state from the frame buffer. Writing a pixel sets its state to either "set" or "free". Both operations are assumed to have an equal and constant cost. Furthermore, the costs for accessing the local working memory can be neglected. Compared with frame buffer operations, these accesses are very fast. Additionally, the algorithms need only a constant amount of accesses to the local memory per frame buffer operation.

To compare the behaviour of the algorithms, different regions have been generated. Overlapping lines, squares and circles were randomly placed on an interactive display (bitmatrix). From the resulting intersections, different regions arise. The corresponding seed points of the regions were selected using a mouse. Similar to polygons, there are three kinds of regions: convex, non-convex and cyclic regions (sorted by increasing complexity). Altogether, about one hundred different regions equally chosen among these three types were tested.

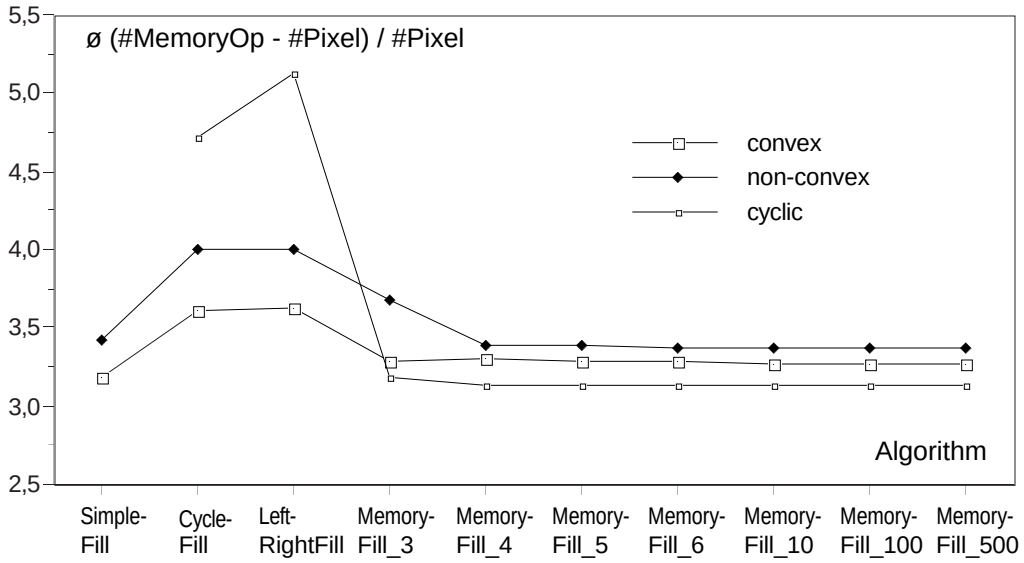


*Fig.11: The experimental results show the efficiency of the different algorithms sorted by the use of memory.*

In Fig.11, the algorithms presented are sorted by the amount of available working memory needed for the cursor along the horizontal axis. The number after the *MemoryFill* algorithm indicates the number of available cursors, i. e., *MemoryFill_n* uses up to *n* cursors *(MemoryFill_1 = SimpleFill* and *MemoryFill_2 = CycleFill)*. For each region class, the average overhead of memory operations relative to the region size (inner pixels) is mapped. This measure is independent of the region size and enables a direct comparison of the algorithms. Generally, to set a pixel at least once, the window must be shifted onto it. One shift costs three reading operations, i. e., the overhead cannot become smaller than three operations per inner pixel.

For cyclic regions, *SimpleFill* has no value because the algorithm would not terminate. The heuristic of *LeftRightFill*, applied to different regions, yields no advantages compared with *CycleFill*. The efficiency of the other heuristic, *MemoryFill*, approximates the optimum with a mere two additional memory addresses, i. e., by increasing memory space, the average results cannot be improved. This holds even in very complex cases, e. g., regions with 100 holes. Comparing the region classes, notice that cyclic regions can be filled faster than convex or non-convex regions at the optimum. This verifies the assumption in Subsection 3.3, that it is not the cycles but a lot of dividing pixels that are time consuming.

## 5 Conclusion

In summary, we solved the problem of region filling with constant working memory in bitmatrices. Several seeding algorithms were presented and verified. We proved that the basic algorithm, CycleFill, fills general, boundary-defined, 4-adjacent regions correctly.

The algorithm *MemoryFill* is shown to be the most efficient heuristic. With little additional memory, the efficiency converges to the optimal overhead of three reading operations per one writing operation. Some algorithms described previously have less overhead in the best case (e. g. *TintFill* (Smith 1979) has overhead Two) but none of them work with a constant working memory.

The presented algorithms can easily be realised in an interactive graphic device with an extra coprocessor for frame buffer operations. Using the most efficient heuristic *MemoryFill*, one

has to determine the maximum memory capacity of the coprocessor available for filling. The more memory that can be used, the more efficient is the heuristic. Thus, the code of *MemoryFill* is down-loaded with the corresponding number of cursors declared in the code.

An interesting generalisation of the region fill problem is filling patterns. The pattern is given by a Boolean function which specifies whether a pixel should be set or not, depending on its coordinates. Thus a region can, for example, be striped periodically or dotted randomly. Because the presented algorithms do not use an extra "working plane" in addition to the frame buffer, no pattern mapping can be applied while copying the filled region into the frame buffer. Further research on filling with constant working memory should aim at algorithms for filling patterns.

## Appendix

Here, the source code of the function *RightCycle* (cf. Section 3.2) is given in Pascal notation. The other subroutines mentioned in the paper such as *InitFill*, *Critical*, *WalkRight*, *Shift*, *ReadPixel* and *WritePixel* are realised in a straitforward fashion following the description of former sections. The function *LeftCycle* (cf. Section 3.3) is obtained through replacing the procedure *WalkRight* by *WalkLeft* in the *RightCycle*.

```
function RightCycle(p :point, w :window, d :direction):boolean;
(*INPUT:   pixel p within a 4-adjacent region, window w of p,*)
(*         current moving direction d *)
(*OUTPUT:  true, if p in true right-cycle; false, otherwise*)
var
     p_old      :point;                    (*starting point of cycle*)
     d_old      :direction;                (*starting direction*)
     stop       :boolean;                  (*dummy*)
begin
     WalkRight(w, d, stop);
     d_old := d; p_old := p;               (*store current values*)
     Shift(p, w, d);                       (*move cursor and update window*)
     while (p <> p_old) do begin           (*move in right-cycle*)
          if not Critical(w) then begin
               WritePixel(p);              (*write to frame buffer*)
               w[0, 0] := true;            (*write to local memory*)
          end;
          WalkRight(w, d, stop);           (*choose new direction*)
          Shift(p, w, d);                  (*move cursor and update window*)
     end;                                  (*right-cycle closed*)
     WalkRight(w, d, stop);
     if (d_old = d) then                   (*check directions*)
          RightCycle := true
     else
          RightCycle := false;
end;
```


## Acknowledgements

For the friendly support on the submitted paper, I would like to thank very much Prof. Heinrich Müller. He also indicated the original problem. Additionally, I would like to thank Alf-Christian Achilles, Dietmar Kappaey, Peter Kneisel, Volker Vogelgesang, and the anonymous reviewers for proofreading and their valuable comments.


## References


Ackland BD Weste NH (1981) The edge flag algorithm - A fill method for raster scan displays. IEEE Transactions on Computers 30(1): 41-48.

Brassel KE Fegeas R (1979) An algorithm for shading of regions on vector display devices. Computer Graphics 13(2): 126-133.

Cullen HF (1968) Introduction to general topology. Heath and Company, Boston.

Dunlavey MR (1983) Efficient polygon-filling algorithms for raster displays. ACM Transactions on Graphics 2(4): 264-275.

Fishkin KP Barsky BA (1984) A family of new algorithms for soft filling. Computer Graphics 18(3): 235-244.

Foley JD van Dam A Feiner SK Hughes JF (1990) Computer graphics: Principles and practice. 2.ed. Addison-Wesley: 979-986.

Liebermann H (1978) How to color in a coloring book. Computer Graphics 12(3): 111-116.

Little WD Heuft R (1979) An area shading graphics display system. IEEE Transactions on Computers 28(7): 528-531.

Newman WM Sproull RF (1973) Principles of interactive computer graphics. McGraw-Hill 2. Edition: 253.

Pavlidis T (1982) Algorithms for graphics and image processing. Computer Science Press: 167-193.

Rosenfeld A Kak AC (1982) Digital picture processing. 2. Edition, Academic Press.

Shani U (1980) Filling regions in binary raster images - A graph-theoretical approach. Computer Graphics 14(3): 321-327.

Smith AR (1979) Tint fill. Computer Graphics 13(2): 276-283.

Tang GY Lien B (1988) Region filling with the use of the discrete Green Theorem. Computer Vision, Graphics and Image Processing 42: 297-305.