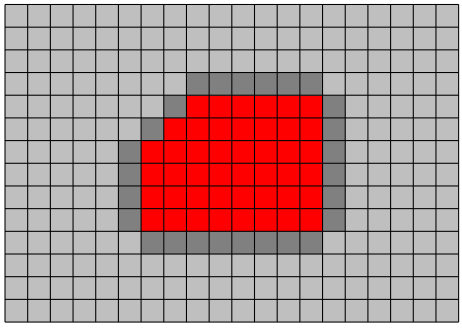
An Efficient and Versatile Flood Fill Algorithm for Raster Scan Displays

by C. Bond, 2011
www.crbond.com



Contents

List of Figures

1 Background

There are several types of *flood fill* algorithms[1] discussed in graphics literature. The problem to be solved involves identification of contiguous pixels on a raster display and re-coloring them according to certain chosen constraints. This paper addresses a method for filling a region under the *4-way* connectivity rule, which is illustrated in Fig. (1).

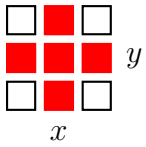


Figure 1: Template for 4-way Connectivity

This rule holds that a pixel, such as the center pixel at (x, y) in the figure, should be filled if one or more of its four adjacent vertical or horizontal neighbors is filled. Diagonal pixels are not considered connected in this scheme. If diagonal neighbors ***are*** considered connected, an *8-way* connectivity results. The practical difference is this: if a diagonal line on a raster display can serve as a barrier separating a filled region from an unfilled one, 4-way connectivity is implied. Fig. (2) shows an example.

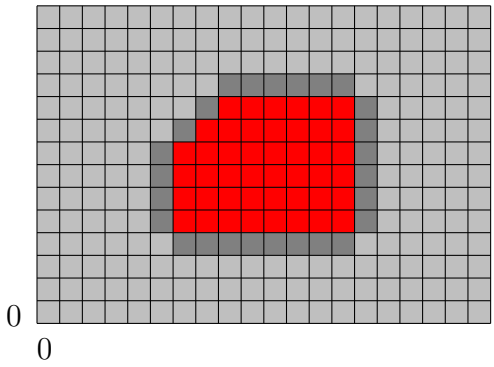


Figure 2: Filled Surface with Diagonal Boundaries

[1]Sometimes referred to as the painter's algorithm or the *seed fill* algorithm.

2 Preliminary Considerations

2.1 Objectives

The development of the method described in this paper was guided by a specific set of requirements. Briefly, these are:

Correctness It was a top level requirement that the method correctly executes the flood fill under all conditions. It must fill every pixel which is reachable from the starting point and not overwrite any other pixel. This can be a very demanding requirement in complex work spaces. For example, spiralled geometries or regions containing text must be handled correctly.

Efficiency This has two aspects. One, the implementation should make modest use of system resources. Some fill algorithms are recursive at the pixel level and may require prohibitive stack space. Two, each filled pixel should be visited only once. This requirement has a benefit beyond performance. Namely, if each filled pixel is only visited once, it becomes possible to fill with arbitrary colors such as transferring an image from another bitmap into the filled region. This may not be possible with methods which revisit pixels. Border pixels may be visited from either side, but that does not affect any of the fill methods described here, except the re-bordering option discussed later.

Note that elegance was *not* a driving requirement. Some published flood fill algorithms are quite elegant, but may be very resource intensive or slow.

2.2 Overview

The chosen method for filling consists of finding line segments which contain fillable pixels with termination end points. It lends itself to very efficient code, particularly in assembler language, and only requires each tested and filled pixel to be visited once. Since a completed line segment is processed in each call to the scan and fill routine, the segments are also only visited once.

We will use a stack[2] to hold pending operations, but the method is less stack intensive than pixel by pixel methods.

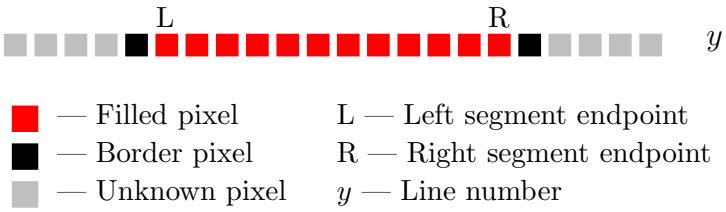


Figure 3: Scanline Legend for Current Segment

Since this paper relies heavily on graphic representations of the implementation details, a standard for illustrating data types is needed. Such a standard is suggested by the legend in Fig. (3). As the exposition continues, additional identifiers will be defined, but for the moment, this filled segment of a scanline shows the plan.

3 Architecture of Implementation

In order to make best use of system resources, the author suggests that the low level scan and fill routines be implemented in assembler language. There are several reasons for this, but the primary motivation is to take advantage of proximity relations in memory accesses. The bitmap upon which the filling operations are performed is an array of contiguous memory locations and higher level languages do not always streamline consecutive accesses to adjacent locations.

The line scan and fill routines can be called by an efficient higher level language with little performance penalty. For ease of coding, debugging and maintenance, C++ is a good choice.

Hence, we divide the flood fill algorithm into the following sections:

[2]Actually, two stacks are used, as will be explained later.

Section	Purpose	Language
1	Set parameters and options	C++
2	Logical decisions and stack management	C++
3	Scan and fill routines	assembler

4 Basic Algorithm

There are two standard forms of the flood fill algorithm. One is to fill a region of a specified color with a replacement color. This method is commonly used to color a background or to create one for overlayed graphics or text. This form only colors pixels of the selected color with a new color, stopping at every pixel which differs from the selected one.

A second form is to color every pixel in regions stopping at a selected border color. This form is used to wipe out all text or graphics within a specified boundary.

Both types are common, and both can be implemented with the method presented. In addition, several uncommon or novel fill strategies will be given.

4.1 Top Level Calling Routine

The highest level call must provide the following information: 1) the type of fill requested, 2) the starting pixel in the bitmap, 3) the fill parameters. For the simplest cases the type of fill will be *fill-surface*, or *fill-to-border*.

In the event that a fill-surface is requested, the parameters include the color of the surface to fill and the replacement color. Of course, these should be different. The lower level routines will test the starting pixel to determine whether it has the surface color, and will return with no action if is not.

For a fill-to-border, the parameters include the fill color and the border color. They must be different. Here the lower level routines will test the starting pixel to determine whether it is the border color, and will return to the caller

with no action if it is.

In either case, the other parameters include a pointer to the bitmap, the coordinates of the starting pixel and the number of rows and columns in the bitmap. A typical calling convention in C++ might look like this:

```
void flood_fill(Byte *bmp[],int xc, int yc, int rows,
                int cols, short bcolr, short fcolr, int FOPT);
```

where an 8-bit bitmap is specified with starting pixel (x, y), number of rows, $rows$ and number of columns, $cols$. Generally, $rows - 1$ is the last row and $cols - 1$ is the last column in the bitmap because of 0-based indexing in C++. These values are required to limit the scanning operations so no runaway memory searching takes place. The parameter *bcolr* is the surface or border color, and $fcolr$ is the fill color. $FOPT$ is the parameter which specifies the type of fill requested, and is used to distinguish between *bcolr* as a background or border color.

It might be desirable to return a value to the calling routine which can signal an error condition or failure to fill. Also, an additional color may be required for specialized fill operations. Further, if the fill involves transferring pixels from another image an additional bitmap pointer may be needed. Other ways of managing the calls are also possible. For example, a structure could be passed which contains options, auxiliary colors, pointers, etc.

4.2 Low Level Routines

It is preferable to discuss the lowest level operations before the mid-level C++ routines are explained. This is because there are a number of special conditions which may occur and which require handling at higher levels. In some cases, further scan and fill requirements are uncovered which are used to guide the stack handler.

The scan and fill procedure is given a line *descriptor* which identifies the line segment to be processed. The descriptor consists of three integers: the line number (y-coordinate), the left end of the search region and the right

end. The latter two integers are x-coordinates which we will identify as L and R. The limits are those of an adjacent line with y-coordinate one more or one less than the current line. The endpoints of any segment found are returned to the caller for further processing as $Lnew$ and $Rnew$. The special case which arises when no fillable pixels are found is flagged by returning a value of $Lnew$ which is greater than $Rnew$. Otherwise, $Lnew \leq Rnew$. $Lnew = Rnew$ when a single pixel is filled.

The pixels are examined starting from the left end L and any fillable pixels contiguous in the region are filled until termination points on the left ($Lnew$) and right ($Rnew$) are reached. As will be seen, the rightmost termination point may stop the search for candidate pixels before reaching $R - 1$, and this is one of the conditions which must be handled.

Note that there may be other fillable segments along the line, before or after the current segment boundaries, but that these are not visible to the routine at this point.

It's easiest to understand the various cases which can occur by examining graphic illustrations. These will be shown and briefly discussed in the following paragraphs. We will consider the various situations which might arise when we have filled a segment and are examining an upper neighbor for fillable pixels.

Figure 4: Pixel Properties for *no fill* Conditions on $y + 1$

In the figure, the lowest line, y, contains a previously filled segment. The line above, $y + 1$, has just been scanned for fillable pixels. If all possible adjacent pixels have been tested and no fillable ones have been found, the region covered is a boundary. In this case, $Lnew > Rnew$ is returned to the caller. It doesn't really matter what specific values are chosen, only that the inequality is satisfied.

A *simple* fill is one which does not uncover additional candidate search regions. For simple fills, $Lnew > L - 2$ and $R - 2 < Rnew < R + 2$.

Figure 5: Pixel Properties for Longest Simple Fill on $y + 1$

Since only one filled segment is completed during any call to the scan and fill routine, complicated geometries may require additional calls with the same line number, but differing L and R values.

For example, a fill which terminates on the right before reaching pixel $R-1$, leaves critical pixels untested. This case is illustrated in Fig. (6). It is detected in the stack handler by checking whether $Rnew < R - 1$.

Figure 6: Possibly Incomplete Fill on Line $y + 1$

Here the *magenta* pixels remain untested during the current scan and fill, and a descriptor for this segment must be saved on the stack for future processing.

Figure 7: Possibly Incomplete Fill on Line y and $y + 1$.

In Fig. (7) we find a case where candidate pixels are uncovered in the current line as well as the previous one. The *cyan* pixels were untested and a descriptor must be saved. This situation is what motivated the creation of an "up" stack and a "down" stack to account for the direction of adjacent line.[3] For our example, untested regions in the current line (which is higher than the previous line) will be examined later and if any fillable pixels are found, they will be filled and a descriptor will be stored on the *up*-stack. Untested

[3]It is also possible to extend the line descriptor to include a direction flag and use a single stack.

regions on the previous line may include fillable pixels, in which case they will be filled and further processing will continue downward.

Figure 8: Possibly Incomplete Fill on Line y.

In this case, we need to generate a descriptor for a segment on the previous line. The untested pixels are shown in *tan*. It is possible for uncovered segments to exist at either end of the previous line, so one or two descriptors may be placed on the *down*-stack by the stack handler.

Pseudo-code for Scan and Fill The following code-like algorithm describes the operation of a *scan and fill* module for a fill-surface flood fill. It is very concise (and hard to read), but at least it is fairly language independent. It uses C/C++ increment and decrement operators and bracket array notation to indicate position along a line.

The program should be entered with the following parameters:

```
p = pointer to bitmap,
y = current line number,
L = left end of previous segment,
R = right end of previous segment,
Lnew = pointer to storage for left end,
Rnew = pointer to storage for right end,
bcolr = background (surface color),
fcolr = fill color,
xmax = number of columns-1,
ymax = number rows-1,
idx = local variable or register for index.
```

The program is shown in Fig. (9). The operation of this routine is straightforward. Namely to find and fill the first fillable segment in the region passed to it, and return the endpoints in *Lnew* and *Rnew* if one is found, or return

```
 1) idx = L.                              Initialize index
 2) if y[idx] <> bcolr goto 9),           Test starting pixel
 3) y[idx] = fcolr.                       Fill it
 4) do scan_and_fill y[idx--] until       Continue to left end
     y[idx] <> bcolr or idx = 0.
 5) Lnew = idx, idx = L.                  Left found, start right
 6) if idx = xmax goto 8).                If right end, we're done
 7) do scan_and_fill y[idx++] until       Continue to right end
     y[idx] <> bcolor or idx = xmax.
 8) Rnew = idx, return.                   Save and exit
 9) if idx = R goto 13).                  No pixels found
10) do scan_and_fill y[idx++] until       Continue to right
     y[idx] = bcolr or idx = R.
11) if idx = R goto 13).                  No pixels found
12) y[idx] = fcolr, i++, goto 5.          Fill it and continue
13) Rnew = idx, Lnew = idx+1, return.     No pixels, save and exit.
```

Figure 9: Scan and Fill Algorithm for Fill-Surface.

Lnew > *Rnew* otherwise. Nevertheless, simple as this is to state, it does involve considerable logic to execute all possible cases correctly.

4.3 Mid-level Routine

The mid-level module carries the burden of calling the scan and fill routines, identifying further pending operations and maintaining a stack. Much of the logic required to support viable and versatile flood fills occurs here.

There are several modules at this level and we will begin by discussing the stack and stack management routines. This is the best place to start because the stack handler is so tightly coupled to the scan and fill routines.

4.3.1 Stack Management

Several global variables are used to hold critical stack values. These are:

```
up_stack[STACKSZ] = stack used for upward scanning
dn_stack[STACKSZ] = stack used for downward scanning
ustkidx = running index into up_stack
dstkidx = running index into dn_stack
umax = maximum value used by ustkidx
dmax = maximum value used by dstkidx
xmax = number of columns-1
ymax = number of rows-1
lnew = left pixel location on current segment
rnew = right pixel location on current segment
L = left pixel location on previous segment
R = right pixel location on previous segment
y = line number of segment
FOPT = option for type of fill
```

Note that the descriptor values L, R and y are available to all routines in the module.

When the flood fill module is entered, the stacks must have memory allocated. L, R and y are given values returned by the scan and fill routines. The other values are initially set to zero.

During a flood fill run, three values are pushed or popped at a time: L, R, and y. These are descriptors for a segment identified for future processing.

When debugging, *umax* and *dmax* may be monitored to get good estimates of stack usage. A starting value of the STACKSZ for gaining confidence in the program should be about 2000. Note that there may be multiple segments along any line, especially when flooding text areas.

Pseudo-code for Stack Management The modules required support repetitive processing of line segments as well as simple pushing and popping.

Prototypes in C/C++ might look like this:

```
void push_dn(int lp,int rp,int yp);
void push_up(int lp,int rp,int yp);
int pop_stack();
void fill_up(int FOPT);
void fill_dn(int FOPT);
```

The operation of each module is described in the following paragraphs.

```
1) test for sufficient stack space.
2) if no more room goto 6.
3) push lp, push rp, push yp.
4) update ustkidx.
5) update umax.
6) exit.
```

Figure 10: Push Up Method

The `push_dn(lp,rp,yp)` routine is similar to `push_up(lp,rp,yp)`. It may be desirable to return a value to the calling program if there is insufficient stack space. This is a case where the designer may choose to simply abort further action silently or with error notification.

The `pop_stack()` method manages both stacks and returns a value indicating the direction, 1 for upward unstacking, -1 for downward unstacking and 0 for empty stacks. The `fill_up(fopt)` routine handles flood filling in the upward direction. It calls the specific scan and fill method given by the parameter *fopt*. The following shows how it works. Again, the corresponding `fill_dn(fopt)` method is similar. These are the routines which identify line segments that need further processing.

The *linefill(fopt)* routine is a simple switch statement which calls the desired scan and fill method and passes the appropriate parameters to it. On return, any fillable pixels have been processed and *lnew* and *rnew* are available.

```
1) if ustkidx > dstkidx goto 6)
2) if dstkidx = 0 return 0.
3) pop L, R and y from dn_stack.
4) update dstkidx.
5) return -1.
6) pop L, R and y from up_stack.
7) update ustkidx.
8) return 1.
```

Figure 11: Stack Pop Method

```
1) if y>ymax-1 return.
2) y++.
3) call linefill(fopt).
4) if lnew > rnew return.
5) if lnew < L-1 push_dn(lnew,L-2,y).
6) if rnew > R+1 push_dn(R+2,rnew,y) goto 8).
7) if rnew < R-1 push_up(rnew,R,y-1).
8) R = rnew, L = lnew, goto 1)
```

Figure 12: Fill Up Method

A considerable amount of logic has been implemented in the previous code blocks. All that remains is to code the highest level call, whose C/C++ prototype, as previously shown, looks like:

```
void flood_fill(Byte *bmp[], int xc, int yc, int rows,
                int cols, short bcolr, short fcolr, int FOPT);
```

```
 1) if parameters are invalid return.     no action.
 2) allocate stack memory.
 3) initialize globals,
     set lnew=rnew=xc,y=yc,
     ymax=rows-1,xmax=cols-1.
 4) call linefill(FOPT).                  kick start fill.
 5) if starting pixel invalid goto 10).
     else lnew and rnew are set.
 5) push_dn(lnew,rnew,y),                 save segment for down scan
     L=lnew,R=rnew,                        since we start scanning
     fill_up(FOPT).                        upward.
 5) dir = pop_stack().                    let stack handler control
 7) if dir = 0 goto 10)                     completion.
 8) if dir = 1 fill_up(FOPT),
     else fill_dn(FOPT).                  loop until done.
 9) goto 5)
10) deallocate memory and return.
```

Figure 13: Flood Fill Method

5 Variations

Several different flood fill routines are described in this section. Some of them can be combined to permit customizing more elaborate filling schemes.

5.1 Flood-Under

The *flood-under* variation permits the user to flood fill a region without erasing certain graphic objects. For example, a region containing text may have the background filled without changing any of the text characters.

This interesting option differs from ordinary flood filling around text in that it properly fills closed loops in the text, such as those in 'a', 'B', 'd', etc. Standard flood fills leave these regions unfilled.

Flood-under is accomplished by passing a color to the scan and fill routines which will be ignored. That is, as the line is scanned, this color will stay on top because the scan routine skips over each matching pixel. Scanning continues until suitable termination conditions occur. Skipped pixels are treated as if they were filled in so the region containing filled or skipped pixels form the complete segment.

5.2 Flood with Image Transfer

In this version pixels are filled by transferring pixels from another bitmap, perhaps an image of some kind. It requires passing a pointer to the other bitmap, which would normally have the same dimensions. Thus in assembler language one could create an index used as a pixel pointer which facilitates moving a pixel from one bitmap to the other.

Recall that since each processed pixel is only tested once, the bitmap may consist of arbitrary colors, including border colors or skip colors. This is one strong point of the method outlined in this paper.

5.3 Flood with Gradient Fill

One method for creating gradients is to construct a 2^n-byte buffer a containing a range of color values this is passed to the scan and fill routines along with a mask consisting of n 1's. At each point to fill the mask is ANDed with the current x or y position to index into the color buffer for the current value.

Again, there are no restrictions on the colors used in the buffer.

5.4 Tiled Fill

Just as gradients can be formed in either direction, tiles can be transferred from small bitmaps. If the previous method using a mask is used, the tile should have dimensions that are powers of 2.

5.5 Re-Border

Re-bordering can be done by simply re-coloring any border found. Since each segment is completed in a single visit, the re-coloring will not interfere with other segment fills. But note that border pixels may be visited from either side, so this option is most easily implemented with the fill-surface method.

6 Conclusion

A comprehensive flood fill algorithm has been presented. Complete details for implementation in any computer language are given. A number of hitherto undocumented filling options are supported and described. Demonstrations are posted at:

http://www.crbond.com

APPENDIX

A Assembler Language Scan And Fill

For completeness in the exposition, the author includes an assembler language implementation of the fill-surface routine. Note that in this module the symbol *bcolor* has been replaced by *scolr*. This was done to simplify focus on the type of fill being performed.

The code was written for Borland TASM32, but is easily ported to other assemblers. The bitmap origin $(0, 0)$ is at the lower left corner.

```
        .586
        .model flat,C
        .code
        locals

        public fill_surf

;
;       "C" calling conventions:
;
;       void fill_surf(Byte *lptr[],int y,int xmax,int l,int r,
;               int *lnew,int *rnew,short fcolr,short scolr);
;
;       'fcolr' is the fill color. 'scolr' is the color of the
;       background surface to fill. Any other color is considered a
;       boundary color, and will terminate the spread of the filled line.
;
;       Inputs:
;
;           lptr    - pointer to 256-color bitmap
;           y       - row number for current scanline fill
;           xmax    - maximum 'x' value (bitmap width-1)
;           l       - leftmost pixel of adjacent filled line segment
;           r       - rightmost pixel of adjacent filled line segment
;           fcolr   - fill color (0-256)
```

```
;           scolr   - color of surface to fill (0-256)
;
;       Outputs:
;
;           lnew    - leftmost pixel of current filled line segment
;           rnew    - rightmost pixel of current filled line segment
;           ****    - the filled pixels, if any, on the current
;                       row of the bitmap
;
;       NOTES:
;           -   If no pixels on the current line can be filled, the
;               returned value of 'lnew' is greater than 'rnew'
;               (lnew > rnew).
;           -   'fcolr' must be different from 'scolr'.
;           -   The fill routine is bounded on the left by 0
;               and on the right by 'xmax'.
;
;   ---------------------------------------------------------------------
fill_surf proc
        arg lptr:ptr,y:dword,xmax:dword,l:dword,r:dword
        arg lnew:ptr,rnew:ptr,fcolr:word,scolr:word
        uses ebx,esi,edi

        mov esi,lptr ; esi points to bitmap
        mov ebx,y
        mov esi,[esi+4*ebx] ; esi now points to row 'y'
        mov ecx,l     ; ecx has offset to 'L'
        mov ax,fcolr
        mov dx,scolr

        cmp byte ptr [esi+ecx],dl       ; check starting pixel value...
        jne @@scanrt                    ; ...for direction of initial search.
        mov byte ptr [esi+ecx],al       ; replace color and extend to left
        cmp ecx,0                       ; check for lower bound for 'x'
        je @@foundlft
@@lp1:
        cmp byte ptr [esi+ecx-1],dl
        jne @@foundlft
        mov byte ptr [esi+ecx-1],al     ; fill left until left edge...
        dec ecx                         ; ...or color barrier.
        jne @@lp1
@@foundlft:
        mov edi,lnew         ; left boundary has been found
        mov [edi],ecx                   ; 'lnew' now has left edge of line
        mov ecx,l                       ; start search for right edge
        mov edi,maxx
        cmp ecx,edi
```

```
        je @@foundrt
@@lp2:
        cmp byte ptr [esi+ecx+1],dl
        jne @@foundrt
        mov byte ptr [esi+ecx+1],al     ; fill right until right edge...
        inc ecx                         ; ...or color barrier.
        cmp ecx,edi
        jne @@lp2
@@foundrt:
        mov edi,rnew
        mov [edi],ecx                   ; 'rnew' now has right edge of line
        ret
@@scanrt:
        mov edi,r                       ; entry point is not a left bound.
        cmp ecx,edi
        je @@noline
@@lp3:
        cmp byte ptr [esi+ecx+1],dl     ; skip over pixel until surface
        je @@foundlft2                  ; color or right edge is reached.
        inc ecx
        cmp ecx,edi
        jne @@lp3
@@noline:
        mov edi,rnew    ;update rnew    ; no valid pixels to recolor.
        mov [edi],ecx
        mov edi,lnew    ;update lnew
        inc ecx
        mov [edi],ecx
        ret
@@foundlft2:
        mov byte ptr [esi+ecx+1],al     ; color left edge, save
        inc ecx
        mov l,ecx                       ; update reference location to current
        jmp @@foundlft                  ; pixel and search for right edge
fill_surf endp
        end
```

The fill-to-border routine is the same, except that the pixels are tested against the border color and the opposite logic is applied.

B Proof of Performance

Here we present a proof that the flood fill algorithm described in this paper does not visit any filled pixel more than once.

Recall that the first segment filled is the only segment which was not filled under the control of *fill_up* or *fill_dn*. All other segments were either filled immediately following the fill of an adjacent segment or were stacked for future processing.

At any time, either up-filling or down-filling is in progress. We wish to prove that no filled segment will ever be visited more than once. This part of the proof applies to the fill-surface method, but similar arguments can be devised for the other methods described in this paper.

B.1 The Proof for Fill_Up

The *fill_up* routine at all times identifies candidate segments by testing the next higher line within the current line segment boundaries or, if the current segment is a border and non-fillable, by popping a new descriptor off the *up_stack*. When a border line is encountered and the up_stack is empty, the routine returns to the caller. Similar remarks apply to the *fill_dn* routine.

There are only three situations which can occur where descriptors are stacked. These have been identified earlier in the expositions as uncovered segments to the right of the current line, or beyond the ends of the previous line.

To simplify the proof, we will restrict the discussion to the fill-surface method and the fill_up routine. Similar arguments can be made for the fill-to-border method and identical ones to the fill_dn routine.

As previously noted, the fill_up routine operates until an upper border boundary is reached and the up_stack is empty. Keep this in mind as the discussion continues.

Fig. (14) suggests a situation which might be encountered when scanning and filling upward. Here, both possible untested regions on the previous line have been uncovered.

Figure 14: Untested Segments on y During Fill_Up.

We will prove that these segments cannot have been previously filled.

B.1.1 Case 1.

Line $y + 1$ is the empty segment which has just been filled. Suppose that the segment on the right (the *tan* pixels) in line y were already completely or partially filled. If this condition occurs, it would mean that we will be revisiting filled pixels during a flood fill operation. Now, the pixels would have had to have been filled during *fill_up* or *fill_dn*. But they could not have been filled during fill_up, because if any of the pixels had been filled, line $y + 1$ would have been also immediately filled in the next step, and therefore could not be part of the current scan and fill operation. In other words, if the uncovered segment had been already filled during fill_up, the segment in line $y + 1$ would have not been stacked, but would have been filled in contradiction to the assumption that we have just encountered an empty segment here and filled it.

Therefore, if the *tan* segment had been previously filled, it must have been filled during a fill_dn sequence. But this could not occur either. If those pixels had been filled during a fill_dn they would have been identified during the fill of line $y + 1$, and would have been filled during the fill_dn sequence, again in contradiction to the condition that we have only just now filled it during a fill_up sequence.

B.1.2 Case 2.

Now consider the untested segment on the left (*cyan* pixels). We argue that this segment could not have been filled during a fill_dn operation, because if they had been the current line, $y + 1$ would have also been filled, in which case it would not have been encountered during a fill_up. Don't forget that when fill_dn is in effect, it will continue until it encounters a border and its stack is empty. But these pixels could not have been filled during a fill_up either, because if they had been, the segment on line $y + 1$ would have immediately followed, contradicting the condition that we just encountered it empty.

B.1.3 Case 3.

Figure 15: Untested Pixels on Line $y + 1$ During Fill_Up

This last case deals with uncovered (*magenta*) pixels to the right of the current line, $y+1$, and adjacent to a previous fill. We now argue that pixels in that segment could not have been filled during a fill_dn because if they had been, the segment on line y would have been filled immediately during fill_dn and not stacked. This contradicts the requirement that we identified this segment during a fill_up on line y.

Finally, these pixels could not have been filled during a fill_up, because the only way they could been filled would be via the condition we are currently processing. That is, the only way they could have been identified for processing is the exact condition we are currently handling, which would have caused the current line to be filled and therefore been excluded from a current fill operation.

B.2 The Proof for Fill_Dn

By symmetry, the arguments presented for fill_up apply identically to fill_dn. Thus, it is not possible for any filled segment to appear more than once in any flood fill. This supports the contention that the method described in this paper can be used without error to fill any fillable surface with completely arbitrary color specifications.

B.3 Stack Management

Although we have shown that re-visiting pixels will not occur during the Fill_Up or Fill_Dn operations, it is possible for a stack pop to specify a segment which has already been partially or completely filled. In such cases, it is necessary to trim the segment or delete it.

Our method is to check the stack for partially or completely overlapping segments after each *scan and fill* operation. These are trimmed or deleted as appropriate. Stack checking for this is not particulaly time- consuming since only entries matching the current line number can have overlapping pixels. Generally there will be none, so scanning for matching line numbers is quick.

Many special cases are possible, and we have covered each case in a companion paper, available elsewhere on this website.[4]

B.4 Summary

We have proposed a proof of the contention that no filled pixels are ever visited more than once by the defined algorithm. The suggestion that in certain cases parts of a fillable segment could be encountered more than once lead to immediate contradictions.

[4]C. Bond, "A Flood Fill Algorithm for Digital Computer Displays." Note: The companion paper, along with this paper, amounts to a PhD dissertation and has taken far more time to adequately document than I originally intended. Hence, the companion paper has not been completed or posted yet. I am still working on it and will post it as soon as possible.