Tint Fill

Alvy Ray Smith
Computer Graphics Lab
New York Institute of Technology
Old Westbury, NY 11568

Published in: *SIGGRAPH 79 Conference Proceedings*, Aug 1979, 276-282. Also Technical Memo No 6, Computer Graphics Lab, New York Institute of Technology, Jul 1978, and issued as tutorial notes at SIGGRAPHs 78, 80-82. The document was reentered by Alvy Ray Smith and his son Sam in Microsoft Word on 10 Oct 2000. Spelling and punctuation are generally preserved, but trivially minor spelling errors are corrected. Otherwise additions or changes made to the original are noted inside square brackets. The following note accompanies the original document:

Abstract

To fill a connected area of a digital image is to change the color of all and only those pixels in the area. Fill algorithms for areas defined by sharp boundaries (e.g., a white area surrounded by a black curve) have been implemented at several color computer graphics installations. This paper presents an algorithm for the more difficult problem of filling areas with shaded boundaries (e.g., a white area surrounded by a curve consisting of several shades of gray). These images may arise from digitizing photographs or line drawings with a scanning video camera, or they may be generated by programs which produce antialiased line segments or dekink black-and-white images. When an area in such an image is to be filled with a new color, it is desirable to have the fill algorithm understand the shaded edges and maintain the shading with shades of the new color instead of the old. The tint fill algorithm presented here accomplishes this task. Its name arises from its ability to change only the tint (hue and saturation) of a pixel, leaving the value (blackness) unchanged. Although the algorithm was motivated by and is written in terms of color, it has a more general interpretation, which is also presented.

Key words: fill, flood, tint, gradient, hue, saturation, value, color, matte.

CR categories: 8.2, 3.41.

Introduction

We reinvent the wheel in the first half of this paper by solving the following simply stated problem:

> Given a connected set A of points on the 2-dimensional integer grid, all of the same color c, and bounded by points, none of color c,

> and given a color c' $\neq$ c, find an algorithm for changing all and only the points of A from color c to c'.

We shall call an algorithm which solves this problem a fill algorithm. Fill algorithms have been presented several times before in various guises (e.g., [1,2,3,6]). We present one here not only to document it thoroughly but also to serve as a basis for generalization, in the second half of the paper, to a more sophisticated algorithm to be called the tint fill algorithm. It will be easier to explain tint filling after (simple) filling is described, so a detailed definition is postponed until then. The filling described here is independent of information external to the integer grid, so is not to be confused with the rendering of an external data set [5].

Our bias is toward color computer graphics as evidenced by use of the word "color" in the problem statement above. By the color of a point we simply mean a mapping of the point into a set C, which we choose here to call the set of colors. Hence the intuitive model has each point represented by a small square centered on the point and painted some color. C might be the set M={0,1,…,255} where 0 is interpreted as black, 255 as white, and all other "colors" as grays. Or C might be the set MxMxM where each triple is interpreted as the red, green, and blue (RGB) primary components of colors. A rectangular subset of these squares forms a picture, and each square is therefore called a pixel[1], from picture element. Changing the color of an area, a connected set of pixels, in a picture is called filling the area.

By connected we shall mean 4-connected in the sense of Rosenfield [7]. Two pixels are 4-connected if they share exactly an edge, 8-connected if they share at most an edge or at least a corner. For I the integers, a set A of points in IxI is 4-connected if and only if for any two points P and P' in A there is a subset B of points in A, B={P_0,P_1,…,P_n}, such that P_i is 4-connected to P_{i+1} for $0 \le i < n$, P_0=P, and P_n=P'. B is said to be a 4-connected path in A.

The boundary of a 4-connected set A of points in IxI is the set B of all points 4-connected to points in A but not in A. Thus the boundary of a 4-connected set is only 8-connected (and vice versa). We shall describe the filling of 4-connected areas and only briefly mention the filling of 8-connected areas.

For presentation of the fill algorithm it is convenient to think of a picture as stored in a digital memory where a location has address (x,y) and the value stored there is c. Such special purpose memories do exist and are called frame buffers. It is the presence of several of these frame buffers at NYIT (New York Institute of Technology) which was the motivation for a fill program and hence for this paper.

More specifically, a frame buffer is a random access digital computer memory designed to hold two-dimensional information, where its contents is continually displayed on a standard color video monitor. At NYIT a typical frame buffer has 243K bytes of memory (3x243K bytes for RGB frame buffers) arranged to display 486 lines of 512 pixels each. Thus there are 8 bits of storage for each pixel (24 bits for RGB), and the contents is called a pvalue (pixel value). Thirty

[1] [I now eschew the use of the word pixel to mean a little colored square, but the "little square" model is useful here.]

times a second, the video circuitry of a frame buffer displays the memory, in standard video interleaved scanline order, by doing a table lookup on each successive pvalue along a scanline. The three values returned from the table directly control the three gun voltages of the monitor. Hence the table is called a colormap. Because of this indirection, the association of a given pvalue with a color—and hence the tint and value of this color [8]—is completely arbitrary to within the gamut, or range, of colors which the video monitor can display.

There are many ways to create arbitrarily shaped areas in a frame buffer. Two common ways at NYIT are by handpainting [9] or by automatic entering of animated cartoon characters [10]. Using the fill algorithm, every pvalue in such an area may be changed automatically by first selecting the new pvalue and then indicating any point in the area to be filled. Because of the table lookup described above, changing every pvalue in an area is equivalent to changing the color of the area.

The fill algorithm described below assumes a user has selected an area to fill with a new pvalue he has selected. He passes the area information to the algorithm by specifying only a seedpoint to it. This point—i.e., an (x,y) pair—might be selected by typing at a keyboard or by pointing with the stylus of a tablet, for example. The area to be filled is, of course, the set of points 4-connected to the seedpoint and of the same pvalue as that originally held by the pixel there.

First a basic fill algorithm is presented. Then it is made faster by a simple observation. The extension of fill to RGB, or 24-bit, frame buffers is discussed briefly. Finally a more sophisticated fill algorithm is presented which can fill areas bounded by antirastered, or “smooth”, edges (see below). This algorithm, called tint fill, is used extensively at NYIT for coloring animated cartoon characters entered into a frame buffer by digitizing the output of a scanning video camera.

Conventions

In the sequel there is a set of routines which define the fill algorithm and variations on it. We will use the following conventions: A variable in a routine is assumed global to all subroutines called from it. The symbols $right, $left, $top, and $bottom represent the maximum and minimum values which x and y may assume in a given frame buffer. For example, for the frame buffers at NYIT $left=0, $right=511, $bottom=0, and $top=485. The data type pvalue holds the information contained in one pixel in a given frame buffer. Finally, upper case names are procedure names and underlined terms in procedure statements are assumed to be reserved words of the language used.

Basic Fill Algorithm

All the algorithms to be presented are scanline oriented. That is, each algorithm tries to fill along a scanline before it changes y, the vertical coordinate. The basic notion is that FILL fills all pixels 4-connected to the seedpoint on the first scanline. Then it looks at the scanline above and below for points 4-connected to the scanline segment just filled (and of the same color as the seedpoint pixel before it was filled). A number of these points sufficient to guarantee connectivity go on a stack maintained by FILL. When the scans are finished, a point is popped from the stack and this becomes a new seedpoint. Only one point per scanline segment to be filled need be pushed. Fig. 3 shows the filling of five scanline segments. The black dots are points pushed onto the stack. The variables lx and rx used in the formal algorithm statement below are the left and right x coordinates of a scanline segment.

```
procedure BASICFILL (seedx,seedy,newpv);
    integer seedx,seedy;
    pvalue newpv;
begin
    integer x,y,lx,rx;
    pvalue new,old;

    x:=seedx; y:=seedy;
    new:=newpv; old:=GET;
    if old=new then return;
    PUSH;
    while STACKNOTEMPTY do begin
        POP;
        if GET=new then continue;
        FILLINE;
        SCANHI; SCANLO;
    end
end
```

The utility subroutine GET returns the pvalue stored at frame buffer location (x,y). Its complement SET sets frame buffer location (x,y) to pvalue new.

The utility subroutine PUSH pushes x and y onto the stack. POP pops the top two locations from the stack into x and y. STACKNOTEMPTY returns true if the stack is not empty or false if it is but does not alter the stack. We will assume, until a later section, an infinite stack.

Utility SAVEX saves the current value of x in temporary storage. It is retrieved by RESTOREX. SAVEXY and RESTOREXY serve the same purpose for (x,y).

The other subroutines are defined by these procedures.

```
procedure FILLINE;
begin FILLRIGHT; FILLEFT; end

procedure FILLRIGHT;
begin
    SAVEX;
    while GET=old and x≤$right do begin
        SET; x:=x+l;
    end
    rx:=x-l; RESTOREX;
end

procedure FILLEFT;
begin
    SAVEX; x:=x-l;
    while GET=old and x≥$left do begin
        SET; x:=x-l;
    end
    lx:=x+l; RESTOREX;
end
```

The shadow of a scanline segment is the set of pixels just under (or just above) the pixels in the segment. A scanline segment of length n pixels has two shadows (except, of course, one that lies in line $top or $bottom), each of length n. SCANLO (or SCANHI) below stacks only one point from each scanline segment just filled with FILLINE. This point is the leftmost point in each scanline segment, or subset of a scanline segment, in the shadow.

```
procedure SCANHI;
begin
    if y+l>$top return;
    SAVEXY; x:=lx; y:=y+l;
    while x≤rx do begin
        while GET≠old and x≤rx do x:=x+l;
        if x>rx then break;
        PUSH;
        while GET=old and x≤rx do x:=x+l;
    end
    RESTOREXY;
end

procedure SCANLO;
begin
    if y-l<$bottom return;
    SAVEXY; x:=lx, y:=y-l;
    while x≤rx do begin
        while GET≠old and x≤rx do x:=x+l;
        if x>rx then break;
        PUSH;
        while GET=old and x≤rx do x:=x+l;
    end
    RESTOREXY;
end
```

Improved Fill Algorithm

The speed of the fill algorithm above can be improved by noticing that when neighboring scanline segments are being filled, one of the procedure calls to SCANHI or SCANLO may be redundant. For example, consider filled scanline segments b and c where c falls completely in the shadow below b, and c was filled after b. There is no need to scan along the scanline above c for new seedpoints. In general, there can be no new seedpoints in a scanline segment already filled unless that segment includes pixels more than distance one outside the shadow (on either end) of the segment just filled. Thus the improved fill algorithm below checks for the cases when it can skip one of the scans above or below. The criteria are summarized in the two new subroutines HINEIGHBOR and LONEIGHBOR.

```
procedure FILL (seedx,seedy,newpv);
    integer seedx,seedy;
    pvalue newpv;
begin
    integer x,y,lx,rx,yref,lxref,rxref;
    pvalue new,old;
    x:=seedx; y:=seedy;
    new:=newpv; old:=GET;
    if old=new then return;
    yref:=y; PUSH;
    while STACKNOTEMPTY do begin
        POP;
        if GET=new then continue;
        FILLINE;
        if HINEIGHBOR then SCANHI
            else if LONEIGHBOR then SCANLO
            else begin
                SCANHI; SCANLO;
            end;
        yref:=y; lxref:=lx; rxref:=rx;
    end
end

procedure HINEIGHBOR;
begin
    if y=yref+l and lx≥lxref-l and rx≤rxref+l then return true else return false;
end
```

Replace yref+l in HINEIGHBOR with yref-l to get LONEIGHBOR.

Variations on Fill

Programs realizing several variations on the simple fill algorithm just presented have been written and run at NYIT. This summary does not include tint fill which is more than a simple variation and will be presented subsequently. The most important variation is the extension of the 8-bit fill program to the 24-bit version. In this case the algorithm is not changed. Only the definition of the data type pvalue changes. The 24-bit pvalues are equal if and only if the red fields are equal, the green fields are equal, and the blue fields are equal.

Another variation is the so-called "boundary fill" algorithm. Here an area is assumed to be surrounded by an 8-connected curve of pixels all of the same pvalue, called the boundary pvalue. The algorithm is designed to fill the area enclosed by this boundary regardless of what colors the enclosed pixels have (see Fig. 2). The basic test in the fill algorithm becomes in this case

```
    if GET ≠ boundarypvalue then SET
```

instead of

```
    if GET=old then SET
```

(see FILLINE above). This boundary fill algorithm has the restriction that the color used to fill the area must not already exist at any pixel in the area (unless the filling color is that of the boundary pvalue). The GET=new test after POP may fail otherwise. An easy solution is to boundary fill with a reserved color then refill with simple fill and the desired color.

Another variation is called "texture fill". Instead of filling an area of constant color with a single new color, it is filled with a pattern either algorithmically generated or contained in another frame buffer. In this case the restriction is that no color in the pattern can be the original color of the area being filled.

The texture fill program fell into disuse at NYIT after 1975. Regular fill with a matte color and then a one-pass copy of one frame buffer contents to another based on the matte so generated is computationally cheap and fast and does not suffer from the restriction of texture fill. The extra frame buffer is not needed if the texture is algorithmically generated. Another approach is presented in [3] which uses more computation instead of more memory to avoid the problems of texture fill.

Another variation, which has not been implemented at NYIT, is the fill of 8-connected areas instead of 4-connected areas. All that is required to adapt BASICFILL to the 8-connected case is a more elaborate neighborhood check in the routines SCANHI and SCANLO.

Tint Fill

Line drawings digitized into a frame buffer from pen or pencil drawings via a scanning video camera have the appearance of white areas enclosed by "shaded" curves. That is, the curves are not sharp black lines but are composed of many shades of gray. They tend to be black or almost black near the centerline of the curve and shade to white near its edges. When one of these areas is to be filled, it is desirable to have the fill algorithm understand the shaded edges and maintain the shading with shades of the new color instead of the old. The tint fill algorithm was created to accomplish this task.

Another increasingly common practice in computer graphics also creates areas with shaded edges. This is the process of antialiasing edges. A two-color digital approximation to a straight line in a frame buffer has the appearance of stairsteps. This is called the "jaggies" or "aliasing". A human can be fooled into seeing a smooth rendering of a line by techniques called antialiasing. One of these techniques consists of laying down a ramp of gray shades (assume a black-and-white line with jaggies for this explanation) along each stairstep. All of the antialiasing techniques introduce shades into the edges. Here too the tint fill algorithm is useful.

The basic notion of the tint fill algorithm below is this. Suppose white is high and black is low. Then a scanned-in image or an antirastered line drawing may be thought of as a physical terrain where the white areas have high elevation and the black lines are valleys of low elevation. The shadings of the lines are represented in this terrain by the slopes of the valley walls. To fill a pixel with tint fill means to change the hue and saturation, i.e., the tint, of the pixel color only, not its value, or blackness [8]. Tint fill fills along a scanline under the rule that it can never go uphill. It can fill along level ground or downhill only. A scanline segment for tint fill consists of all the pixels proceeding from the seedpoint and right (and left), which have the same tint as the seedpoint and a value which is either the same or less than the pixel just left (right). Thus a scanline segment is a section of a hill or mesa. The shadows of a scanline segment are as before: all the pixels just below (above) the pixels in the scanline segment. Three scanline segments are tint filled in Fig. 4.

A formal statement of the problem in the style used for simple fill is:

> Given a 4-connected set A of points on the 2-dimensional integer grid, all of the same tint t, and a distinguished point P in A of value v, such that the value of any other point P' in A is a monotonically decreasing function of distance from P along at least one 4-connected path of points in A from P to P', given a tint $t' \neq t$, find an algorithm for changing all and only the points of A from tint t to t'.

The TINTFILL procedure below resembles the FILL procedure above in organization, with TFILLINE, TSCANLO, and TSCANHI corresponding respectively to FILLINE, SCANLO, and SCANHI. We continue to use the terrain analogy to indicate how the shadows of a scanline segment are scanned for tint fill. Two points P and P' on the same scanline will be said to be monotonically connected if the values for all the pixels from P to P', inclusively, monotonically increase with x or monotonically decrease with x. Briefly, we stack the hilltops which are in the shadow, or at least the highest point on a hill which falls in the shadow. These points are later popped from the stack and used as new seed-points. Specifically, a point in a shadow is pushed if:

1) Its tint is that of the original seedpoint before filling, and
2) its value is less than or equal the value of the pixel above (below for TSCANHI), and
3) there is no monotonically connected pixel to the right of higher value that also satisfies 1) and 2) and is in the shadow, and
4) there is no monotonically connected pixel to the left of higher value that has already been stacked.

Fig. 1 illustrates which pixels would be stacked under these rules. They are indicated by upper case letters.

The algorithm below assumes the user supplies the coordinates of the initial seedpoint and the desired new tint. We will employ two new data types, tint and value, to hold color tint and color value. For the purposes of this presentation, we need not be more specific about them. In the current implementation of tint fill at NYIT for 8-bit frame buffers, both of these data types contain four bits. Special colormaps are assumed which map the low four bits of a pvalue into the value and the high four bits into the tint of the color of the pvalue. For RGB frame buffers, tint and value are algorithmically computed [8] from the RGB primary components.

```
procedure TINTFILL (seedx,seedy,newtint);
    integer seedx,seedy;
    tint newtint;
begin
    integer x,y,lx,rx,yref,lxref,rxref;
    tint newt,oldt;
    value oldv;

    x:=seedx; y:=seedy;
    newt:=newtint; oldt:=GETT; oldv:=GETV;
    if oldt=newt then return;
    yref:=y; PUSH;
    while STACKNOTEMPTY do begin
        POP;
        if GETT=newt then continue;
        TFILLINE;
        if HINEIGHBOR then TSCANHI
            else if LONEIGHBOR then TSCANLO
            else begin
                TSCANHI; TSCANLO;
            end;
        yref:=y; lxref:=lx; rxref:=rx;
    end
end
```

These procedures assume utility subroutines GETT, GETV, and SETT as well as the stack maintenance utilities used above. GETT and GETV return the tint and value of the pvalue at current location (x,y). SETT changes the pvalue at current location (x,y) to one with a tint of newt.

```
procedure TFILLINE;
begin
    value lastv;

    lastv:=oldv; TFILLRIGHT;
    lastv:=GETV; TFILLEFT;
end

procedure TFILLRIGHT;
begin
    SAVEX;
    while GETT=oldt and GETV≤lastv and x≤$right
    do begin
        SETT; lastv:=GETV; x:=x+l;
    end
    rx:=x-l; RESTOREX;
end

procedure TFILLEFT;
begin
    SAVEX; x:=x-l;
    while GETT=oldt and GETV≤lastv and x≥$left
    do begin
        SETT; lastv:=GETV; x:=x-l;
    end
    lx:=x+l;
    RESTOREX;
end

procedure TSCANHI;
begin
    if y+l>$top return;
    SAVEXY;
    x:=lx; y:=y+l;
    while x<rx do begin
```

```
        while GETT≠oldt and x≤rx do x:=x+l;
        if x>rx break;
        TPUSHHI;
        x:=x+l;
    end
    RESTOREXY;
end

procedure TSCANLO;
begin
    if y-l<$bottom return;
    SAVEXY;
    x:=lx; y:=y-l;
    while x<rx do begin
        while GETT≠oldt and x≤rx do x:=x+l;
        if x>rx break;
        TPUSHLO;
        x:=x+l;
    end
    RESTOREXY;
end
```

The following two routines are the most complex of this presentation. They are designed to stack all points in the shadow of the current scanline segment which meet the four criteria listed above. It should be noted that correct tint filling would occur if only criteria 1) and 2) were satisfied. The set of points satisfying only these two criteria is the largest set of points that could be stacked for correct operation of the algorithm. The set of points satisfying all four criteria is the smallest set of points that must be stacked for correct filling. For ease of presentation, the routines specified below stack a set of points falling between these two extremes. Fig. 1 shows the set of points actually stacked. The upper case letters indicate points in the smallest set, which must necessarily be stacked, and the lower case letters denote the additional points stacked by these routines.

In the following routines, v is the color value of the current pixel, and vup (vdn) is the value of the pixel just above (below). Four flags are used: Vover is true only when a shadow pixel has a value larger than that of the corresponding pixel in the scanline above (below). Oldvover is the state of vover at the location considered just previously to the current location. Uphill is true only when, proceeding left to right along the shadow, the color values increase monotonically. Stackready true implies points are to be pushed onto the stack. False implies replacement of the top of the stack. In words, the highest points on hillsides which satisfy 1) and 2) are stacked.

Three more utility routines are assumed. These are GETVDN and GETVUP which return the value of the pixel immediately below, respectively above, the current pixel. Routine RETOP replaces the top element on the stack with the current location.

```
procedure TPUSHHI
begin
    integer vover,oldvover,uphill,stackready;
    value v,vdn;

    comment initialize flags;
    v:=GETV; vdn:=GETVDN;
    vover:=if v>vdn then true else false;
    if x=rx and not vover then begin
        PUSH; return;
    end
    stackready:=true;
    uphill:=false;

    comment stack the first valid point;
    if not vover then begin
        stackready:=false;
        PUSH;
    end
    else while x≤rx do begin
        x:=x+l;
        if x>rx or GETT≠oldt then return;
        GETDNFLAGS;
        if not vover then begin
            PUSH;
            stackready:=false;
            break;
        end
    end

    comment main loop;
    while x≤rx do begin
        x:=x+l;
        if x>rx or GETT≠oldt then return;
        GETDNFLAGS;
        if uphill and not vover then begin
            if not stackready then RETOP;
            else begin
                PUSH; stackready:=false;
            end
```

```
                end
                else if not uphill then begin
                        if not vover and oldvover then PUSH;
                        stackready:=true;
                end
        end
end
```

GETDNFLAGS is the following flag maintenance routine:

```
procedure GETDNFLAGS
begin
        vlast:=v; v:=GETV;
        vdn:=GETVDN; oldover:=vover;
        vover:=if v>vdn then true else false;
        if v ≠ vlast then begin
                uphill:=if v>vlast then true else false;
        end
end
```

Procedure TPUSHLO is the same as TPUSHHI above if vdn is replaced everywhere with vup, GETVDN is replaced everywhere with GETVUP, and GETDNFLAGS is replaced everywhere with GETUPFLAGS. GETUPFLAGS is GETDNFLAGS under the same set of replacements used to obtain TPUSHHI.

Historical Note

The earliest version of the fill algorithm as stated here of which I am aware is that of Ken Knowlton [1] in 1964. A less sophisticated algorithm was the foundation of a fill program implemented on the “tricolor cartograph” in 1969 [2] and of a similar program implemented by Joan Miller on a 3-bit frame buffer at Bell Labs in 1969-70 [4]. These two programs could fill only convex simply-connected areas. The first complete frame buffer program was apparently that implemented by Patrick Baudelaire and Dick Shoup at Xerox Palo Alto Research Center in 1973. Programs realizing the fill, tint fill, and boundary fill algorithms have been in service at NYIT since 1975. A similar program, called “flood” however [3,6], was implemented at the Massachusetts Institute of Technology shortly thereafter. More recently, Marc Levoy programmed an 8-bit fill at Cornell University. A 24-bit, or RGB, version of fill and the tint fill program at NYIT were certainly the first of these varieties with only tint fill being a truly novel program. The RGB versions of fill anf tint fill were implemented in 1977 and 1978, respectively, at NYIT. Most recently, Marc Levoy has implemented a 10-bit version of tint fill at Cornell under the name “gradient flooding”.

Implementation Notes

The actual implementation of any of the algorithms of this paper has to deal with several practical matters glossed over by the presentation thus far. The stack, for example, is unfortunately finite. It is possible to make pictures the filling of which, even with careful stack usage described, requires a stack to grow so large as to exceed memory size. Since great speed is highly desirable in a fill program, elaborate stack checks are undesirable. Fortunately, it has been our experience at NYIT that pictures which require such immense stacks are pathological cases, created for the sole purpose of exceeding the limits and of little interest artistically.

Since speed is so important, all fill programs at NYIT have been implemented as in-line assembly code, subroutine calls being too costly. The much used 8-bit simple fill was implemented to fill two pixels (a word) at a time instead of just one (a byte) to exploit the speed advantage of the Digital Equipment Corporation PDP11 family of processors and the Evans and Sutherland frame buffers in word mode. Although the connectivity checks are quite a bit more elaborate in word mode, the resulting speedup was worth the coding trouble. The tint fill program was speeded up substantially by checking for the case where it must fill large full-value areas and doing the highly optimized simple fill in that event.

Concluding Remarks

Tint fill is called by that name because of its original use at NYIT. However, tint and value are only one possible interpretation of the two quantities checked by the algorithm. Hue and brightness or hue and saturation are two other color-related interpretations. But any two quantities, such that one is changed if the other “only goes downhill” from the initial seedpoint, form another interpretation. For example, the contents of one frame buffer A is matted into another frame buffer B just for those points in a third frame buffer C with absolute numeric contents (weights) which only go downhill (decrease arithmetically) from an initial seedpoint in C. The matting is done as a weighted combination of the corresponding pixels in frame buffers A and B, the weight being the pvalue in C at the same (x,y) location. A less colorful name for the algorithm, derived from its terrain model explanation, would be “hill fill”.

As another example, consider an area of color1 in an RGB frame buffer surrounded by an antirastered boundary of color2. We desire to fill the area with color3 while maintaining the antialiasing at the edges. The hill fill algorithm can be used under this interpretation: “Color1ness” is changed to “color3ness” if “not-color2ness” only goes downhill from the initial seedpoint. In fact, an RGB tint fill results if color2 is taken to be black.

Acknowledgement

Ed Catmull helped me a great deal with the fill and tint fill algorithms. In particular, he emphasized the scanline orientation of algorithms and suggested the “never go uphill” concept behind tint fill. Special thanks is due Dr. Alexander

Schure, president of NYIT, for his strong support of graphics research and for the Computer Graphics Lab, his creation.

References

1. Kenneth C. Knowlton, “The Beflix Movie Language”, in Proceedings of the Spring Joint Computer Conference, 1964. (See also, Kenneth C. Knowlton and Lorinda L. Cherry, “Fortran IV Beflix”, in Proceedings of the UAIDE Annual Convention, San Diego, 1969.)
2. W. J. Kubitz and W.J. Poppelbaum, “The Tricolor Cartograph: A Display System with Automatic Coloring Capabilities”, in Information Display, November / December, 1969, pp. 76-79.
3. Henry Lieberman, “How to Color in a Coloring Book”, in Proceedings of the Fifth Annual Conference on Computer Graphics and Interactive Techniques (Siggraph 78) August 21-25, 1978, pp. 111-116.
4. Joan E. Miller, personal communication, Bell Labs, Murray Hill, N.J., July 1978.
5. Theodosios Pavlidis, “Filling Algorithms for Raster Graphics”, in Proceedings of the Fifth Annual Conference on Computer Graphics and Interactive Techniques (Siggraph 78), August 21-25, 1978, pp. 161-166.
6. Craig Reynolds, “Filling Polygons”, in Architecture Machinations, Department of Architecture, Massachusetts Institute of Technology, Room 9-518, May 3, 1977.
7. Azriel Rosenfeld, “Connectivity in Digital Pictures”, in JACM 17:146-160, January 1970.
8. Alvy Ray Smith, “Color Gamut Transform Pairs”, in Proceedings of the Fifth Annual Conference on Computer Graphics and Interactive Techniques (Siggraph 78), August 21-25, 1978, pp. 12-19.
9. Alvy Ray Smith, “Paint”, Technical Memo No. 7, Computer Graphics Lab, NYIT, Old Westbury, NY 11568, July 1978.
10. Garland Stern, “SoftCel—An Application of Raster Scan Graphics to Conventional Cel Animation”, in these Proceedings.

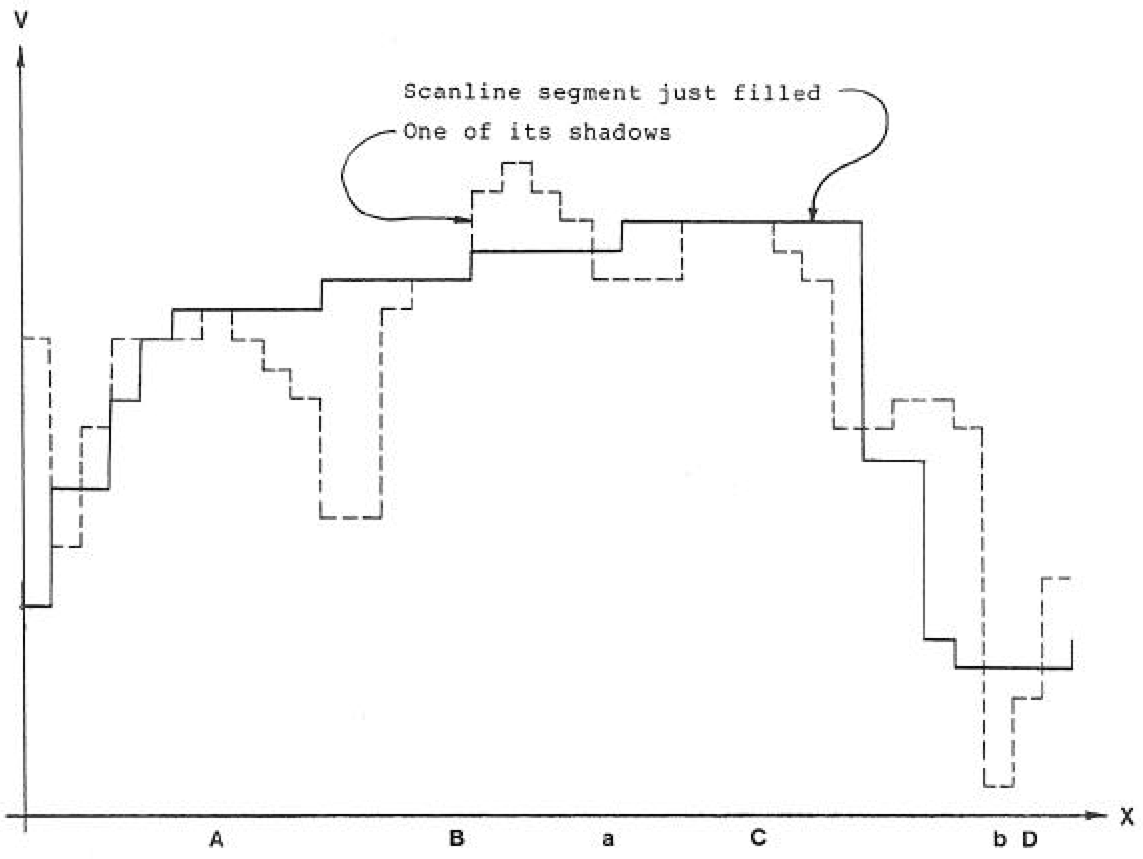

Fig. 1. Only Points A, B, C, D, a, and b of the shadow would be stacked in tint fill.

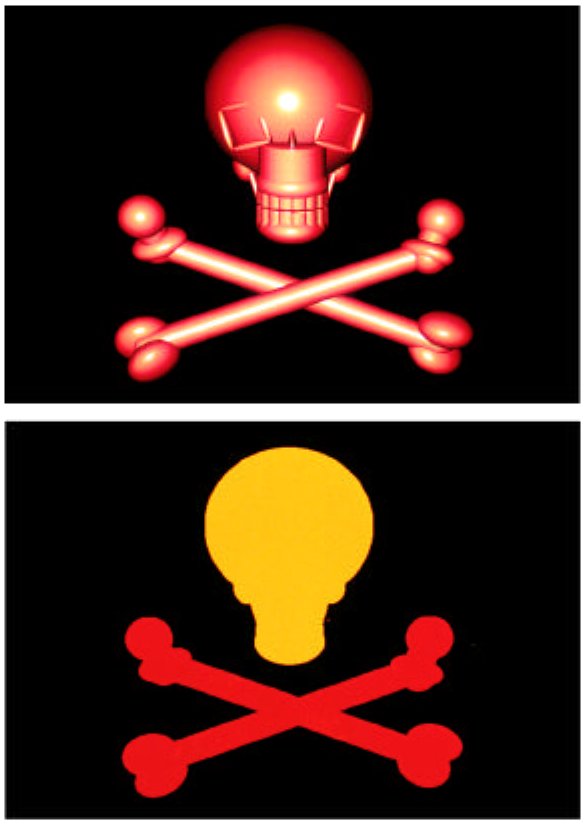

Fig. 2. Before and after two uses of boundary fill.

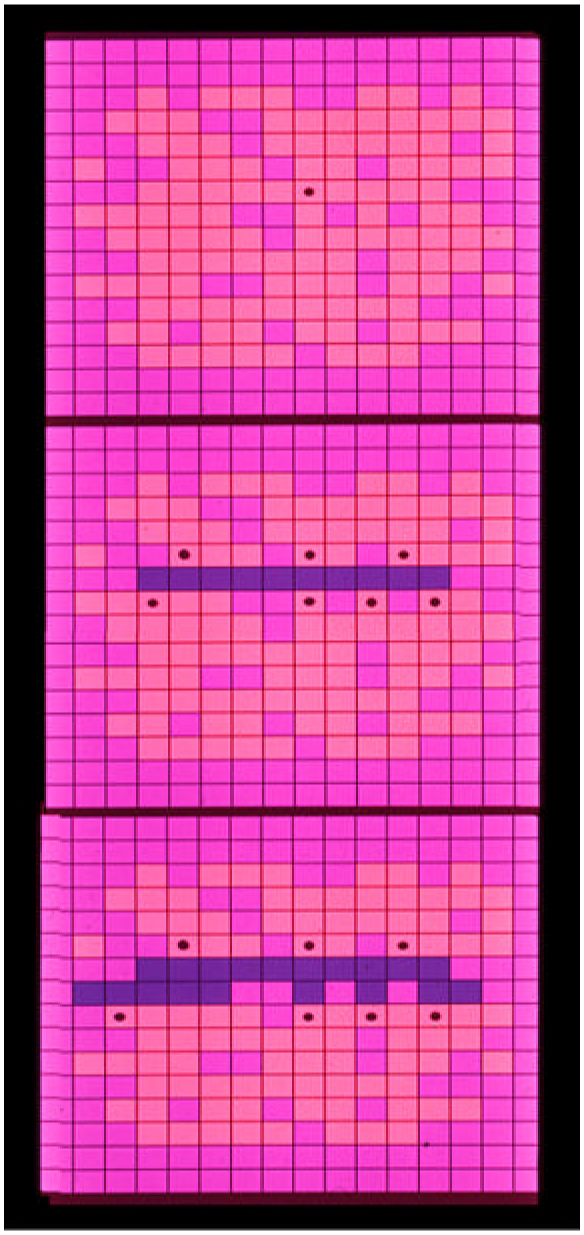

Fig. 3. First few steps of a simple fill. Black dotted points are stacked.

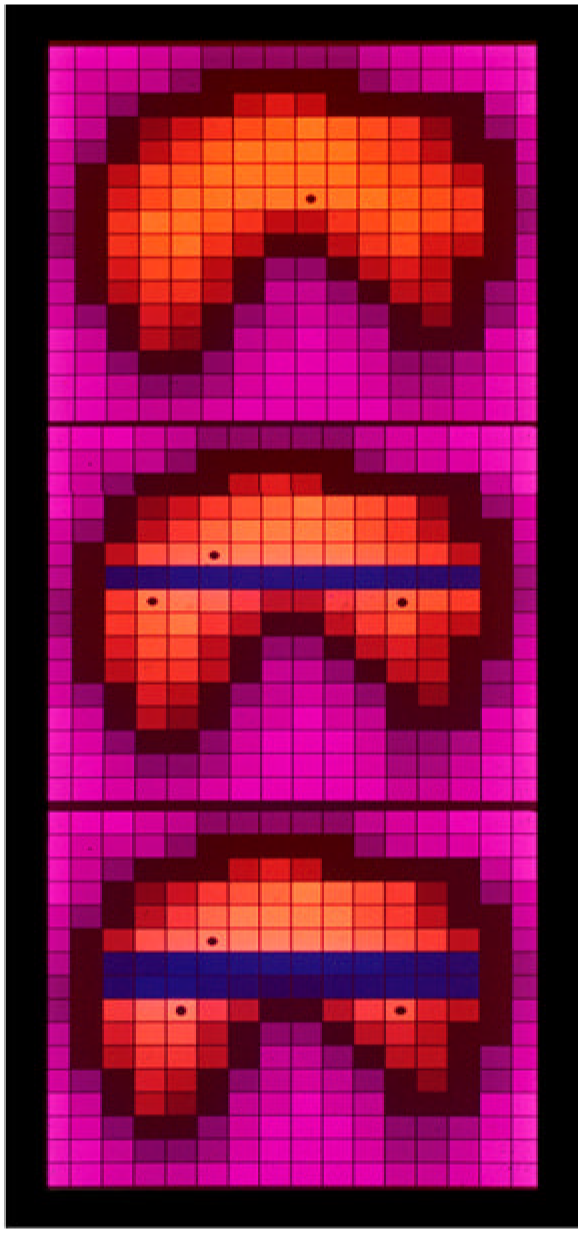

Fig. 4. First few steps of a tint fill. Black dotted points are stacked.