AREA FLOODING ALGORITHMS

Marc Levoy
Hanna-Barbera Productions

June, 1981

1.0 INTRODUCTION

If the animator's task is considered to represent the largest share of the labor required to complete an animated film, the cel painters's job runs a close second. Typical large production studios employ more than one hundred people in each of these two departments. The replacement of xeroxing and cel painting by optical scanning and computer-assisted cel coloring boast improvements of between 5:1 and 10:1 in the time required to color a single cel, making a significant reduction in the labor force required to produce an animated film.

The heart of any computer-assisted cel coloring system (occasionally referred to in the literature as a "scan and paint" system) is a soft area flooding algorithm (occasionally referred to as a "tint filling" algorithm). The basic algorithm for flooding contiguously colored areas in pixel arrays is well known [3,4]. Area flooders depend by definition on the placement of a seed point at some pixel in a digital frame buffer. This seed pixel is usually selected by a human operator, although some automatic key-frame interpolation systems attempt to place seeds in each frame of an animated sequence without manual intervention [1]. Most area flooders in the literature will replace the color of each pixel encountered with a desired new color, and will stop flooding upon reaching any color other than the color found at the seed pixel. This technique yields jagged region boundaries, and drawings colored in this manner exhibit alaising. The so-called soft area flooders, on the other hand, are designed to operate on anti-aliased region boundaries, replacing only the "color" of the pixels being flooded while leaving the "shade" intact. Drawings colored with this technique have a soft, pleasing appearance, and are acceptable for use in production studio animation.

The paper by Alvy Ray Smith [3] (reproduced in this tutorial) gives an excellent presentation of both hard and soft area flooding. While many variations on the particular algorithm offered by Smith are conceivable, all area flooders have certain characteristics in common which may be summarized as follows.

1. They are recursive in the sense that they flood a limited set of pixels according to some rule of proliferation, storing the positions of pixels encountered along the way that bear further examination, then retrieving these positions for use as seeds in a new invocation of the algorithm.

2. They are non-deterministic in the sense that the sequencing of the flooder through a contiguously colored region is ordered by the particular topology encountered along the way, rather than by some arbitrary rule such as top-downward or center-outward.

3. They are localized in the sense that the only information available at each instant are the colors and positions of pixels that have already been encountered. No global information is available during the flooding operation, such as the existence of local intensity minima (which signify edges) or the colors in the centers of adjacent regions (which are required for rgb flooding).

Even within this framework, there is much room for variation and optimization in both the basic search algorithm and in its implementation. This paper will explore a few alternatives for flooding algorithms. It will also consider improvements in the implementation of existing algorithms.

Unfortunately, evaluating the performance of a flooding algorithm is not as simple as letting it loose on a sample region and recording the elapsed time. Depending on the hardware configuration for which the flooder has been designed, several other issues may be pertinent as summarized below:

1. Some algorithms trade computational expense for memory requirements, either by performing pre-processing on the entire image to be flooded, or by retaining additional information about individual pixels as they are flooded. If these flooders must operate in a multi-user environment, how do processor time costs compare to memory usage costs for various approaches? How much memory must be allocated for each flooder?

2. Some algorithms operate in virtual frame buffers contained in the host processor, writing only modified pixels to the real frame buffer. If these flooders are written for a virtual-memory environment, what are the paging loads associated with different approaches? What are the ratios between processor time and elapsed time?

3. Some algorithms operate on single pixels, while others operate on entire scanlines at once. How suitable are the various algorithms to implementation on different frame buffers? Does the interface between the host processor and the image storage mandate a particular approach? How much time is lost in accessing the colors of pixels being considered for flooding? For changing the colors of pixels that have been flooded?

In the remainder of this paper, numerous alternatives to the standard area flooder will be examined and their performance and overall desirability evaluated in terms of some of the issues listed above. Unfortunately, space prohibits the inclusion of code for each variation discussed. Therefore, the differences between each algorithm and those presented in the Smith paper will be discussed in narrative form. A single algorithm, the favorite of this author, will be shown in code form near the end of the paper.

2.0 FLOODERS THAT USE ALTERNATIVE PROLIFERATION PATTERNS

The particular algorithm being used to determine which pixel is processed next gives rise to what can be loosely termed the proliferation pattern. In all but the very fastest area flooders, this progression is easily visible on the frame buffer display as the seed color spreads through the region being flooded. The proliferation pattern for a certain area flooding algorithm is independent from what is often called the proliferation rule, and the distinction between these two is crucial to understanding the discussion which follows. The test performed on any pixel adjacent to the current pixel, whose success or failure will determine whether or not that pixel is stored for use as a future seed position, is called the proliferation rule. Smith has suggested several such rules, including filling all pixels until the color changes (basic fill), filling all pixels until a certain color is encountered (boundary fill), and filling until the shade begins to climb uphill (tint fill). The proliferation rule determines which pixels in a given environment will be flooded by placing a seed of a given color in a given position; the proliferation pattern determines the order in which these pixels will be flooded.

With this distinction made clear, several alternative proliferation patterns can now be examined, together with their effect on performance and overall desirability.

2.1 Stack Algorithms Versus Queue Algorithms

The progress of a standard area flooder through a contiguously colored region is familiar to all who have seen one operate. Since the processing of any one scanline usually results in the generation of at least two new seed positions, the order in which these seeds are retrieved from storage controls the order in which additional scanlines will be processed. The Smith algorithm, like most area flooders, uses a simple push-down stack to store the positions of pixels to be processed at the completion of the current scanline. As a result, the seed most recently generated dictates the scanline that will be considered by the next invocation of the flooder. Figure 1 illustrates the sequence of scanlines that will be processed by a stack-oriented flooding algorithm operating in a sample region.

One characteristic of this approach is that it maintains a high degree of locality, working in only one portion of the sample region at a time and moving consistently in one direction until it enounters a change in the boundary geometry. For implementations in which locality is important, this is a significant advantage. If, for example, the processor is interfaced to the frame buffer by a scheme in which only one narrow strip of the image memory is accessable at a time, such a proliferation pattern is highly desirable. On the other hand, small pockets are often left unflooded as the algorithm speeds through a given portion of the region, as shown in Figure 1b. These pockets are reflected by unprocessed seed points in the stack which must be picked up on the way back as the flooder unwinds its stack, as shown in the figure 1c.

One of the first variants on the standard flooder that comes to mind is to substitute a first-in first-out queue for the push-down stack. In this case, the first seed generated during the processing of the current scanline becomes the first to be retrieved from storage. Figure 2 illustrates the sequence of scanlines that will be processed by a queue-oriented flooding algorithm operating in the same sample region as that used in Figure 1. The flooding is seen to progress simultaneously in as many directions as there are "arms" of the region being flooded.

This approach implies that the flooder may be working in several areas of the sample region at one time, but will expand into and fill all pockets as it progresses through an area, as shown in the figure. This insures that once an area has been passed, no further processing on scanlines in that vicinity will be necessary. This is a terrible algorithm for implementation with the strip mapping processor/frame buffer interface cited above, since more than one area is active at a time. If, however, the processor can map to several strips of image data at once, or if a virtual-memory processor with generalized LRU (Least Recently Used) paging is utilized, this algorithm may actually offer improved performance over the stack algorithm.

The twists and turns taken by any flooder are highly dependent on the detailed geometry of the region boundaries. In the final analysis, comparisons between stack and queue approaches using various sample regions has suggested that the relative merit of each approach depends to a great extent on the conditioning of the input data. If the lines have been drawn smoothly and the digitizing system is producing consistent and artefact-free anti-aliasing (in the case of optically scanned soft edges), the stack algorithm triumphs. If the data is poorly conditioned, containing broad or noisy gray-scale transitions such as might be observed in medical images, queue algorithms are preferrable.

2.2 Scanline Algorithms Versus Pixel Algorithms

Scanline-oriented algorithms are by far the most popular in existence, but they are hardly the easiest to design. A class of students presented with the problem of writing an area flooder will most likely come up with a concentric, or diamond flooder. In these algorithms, the unit of work is a single pixel rather than a scanline. After flooding the seed pixel, all pixels surrounding the current pixel (either 4-connected to the current pixel, or 8-connected, the general algorithm remains the same) are considered as possible seed positions for the next invocation according to the current rule of proliferation. When all such pixels have been considered, the current invocation terminates and the stored seeds are retrieved and processed.

This approach can be associated with either a stack or queue implementation, each yielding different orders of proliferation. The stack version is shown in Figure 3 and the queue version in Figure 4. Both exhibit absolutely terrible locality as the figures demonstrate. This is clearly not the best algorithm to use in a virtual or mapped frame buffer environment. The size of the stack or queue is also enormous, reflecting the pixel-oriented nature of the algorithm. This incurs both a memory cost and a processing cost, since the additional seed points must be stored and retrieved.

On the bright side, this algorithm is extremely simple to program and therefore lends itself very well to hardware implementation. For systems that boast an efficient means of random access to the entire image memory at once, this approach might be the optimum solution. There is in fact at least one micro-processor based turnkey cel animation system available on the market that uses a concentric proliferation rule in its hard-edge area flooder. The simplicity of this approach also suggests that an implementation utilizing multiple parallel micro-processors could be designed, wherein all processors receive seed pixels from a single stack and utilize occasional checks of the image memory itself to avoid duplicating the work being done by the other processors.

2.3 Effect Of The Display System Architecture

In comparing the overall desirability of these alternative proliferation patterns, the importance of the display system architecture becomes evident. Particularly worthy of study is the interface between the processor executing the flooding algorithm, whether an external mini-computer or a micro-processor contained within the display system, and the image memory containing the pixels to be flooded. To conclude this section, a list of the typical frame buffer interface architectures is offered, ordered according to this author's opinion as to their suitability for area flooding algorithms:

1. Direct refresh of processor memory. The area flooder runs in a processor whose address space includes the entire image memory, which is being scanned directly and continuously onto the display screen. The maintenance of locality is unimportant and no transfers of pixel information to or from a separate image memory are required. This is probably the ideal environment for an area flooder.

2. X,Y,C register on a frame buffer interface. The area flooder runs in a host processor and can access random pixels in the frame buffer by depositing a desired X and Y position into registers, then reading or writing the color from a third register. The maintenance of locality is relatively unimportant, although most frame buffers containing this type of interface exhibit a reduced cost if pixels are accessed in row or column sequence. This environment is a good second choice.

3. Strip or block mapping from processor address space into image memory. Either in horizontal strips or rectangular blocks, the host processor maps the address space of its resident area flooding program into image memory. Within a certain range, the preservation of locality is unimportant. When flooding proliferation excceeds this range, the mapping must be changed accordingly. This constraint makes this type of interface only a third choice, well behind the first two.

4. Channelled I/O on entire scanline segments. The area flooder runs in a host processor, but can only transfer entire scanlines or portions of scanlines to the frame buffer using channelled or directed I/O. The overhead required to examine the value of a single random pixel is so high as to be nearly prohibitive. For this reason, this architecture does not lend itself very well to area flooders, even those exhibiting good locality.

For the sake of completeness, there is one final architecture to consider. There are raster display devices on the market whose image storage mechanism is based on the principle of run-length encoding of color information. In such a device, segments of color that are continuous along scanlines are represented by single entries in a list, resulting in a substantial reduction of the storage requirements for the entire image. Unfortunately, it is difficult if not impossible to break apart and recombine color runs not accessed in sequence along a scanline, a necessary capability for even the standard scanline area flooder. It therefore seems reasonable to conclude that although an area flooder could undoubtedly be written for a run-length encoded frame buffer, it is not really the best use of either these display devices or the time of the programmer.

If the reader has found the interface of his or her frame buffer down at the bottom of this list, there is an alternative scheme for area flooding that might offer some salvation, the use of a virtual frame buffer. In this approach, the area flooder runs in a high-level language in a host processor, reading and writing pixel values to an image array contained entirely within the memory of the host processor. Those pixels which have been modified by the flooding process are copied at intervals from the image array in the host to the frame buffer image memory. Since the virtual frame buffer contains an exact copy of the image being flooded, no transfer in the other direction is ever necessary. The preservation of locality is fairly important, since random accesses will incur paging costs, but the paging is handled by the operating system. The advantage of this solution is that it is fairly independent of both the architecture of the frame buffer and of the design of the interface between the host processor and the frame buffer image memory. This therefore becomes the fall-back approach for systems in which the frame buffer interface is non-optimal, such as in the case of channelled I/O.

A further discussion of this problem, and hardware considerations for painting and area flooding programs in general, is available in [2] (reproduced in this tutorial).

3.0 IMPROVEMENTS TO THE STANDARD AREA FLOODING ALGORITHM

The code given by Smith for the standard scanline area flooder is extremely compact. Particularly elegant is the scan for "children" (pixel positions to be stacked for use as future seed points) above and below the current scanline. It is possible, however, to improve the performance of these procedures, and by implication the performance of the entire flooder, at the cost of introducing a measure of additional complexity into the code executed by the procedures.

3.1 Improvements To Hard-edge Flooders

The first improvement that can be made in the scanning procedures requires no additional information and can be easily added to the existing algorithm. The presence of the parental variables (labelled "yref", "lxref", "rxref" in the Smith algorithm) allows the scan procedures to avoid re-processing the parent scanline if the current scanline lies entirely within the shadow of the parent. If the current scanline lies only partially in the shadow of the parent, it is still possible to avoid testing those pixels which do in fact lie in this shadow. The introduction of a few well placed min-max functions to establish new do-loop termini for the scanning process will implement these optimizations and the performance of the scanning procedures is improved correspondingly. The algorithm shown at the end of this section incorporates these modifications.

The key to achieving the next increment of improvement lies in placing these parental variables on the stack along with seed points, making them available to the flooder as those seed points are unstacked and processed. In order to understand the importance of retaining parental information, it is helpful to consider the flooding situation diagrammed in Figure 5. The standard area flooder begins by processing the scanline containing the initial seed point, scanline 1 in Figure 5a. Two pixels are placed on the stack, one above and one below this scanline. After following the lower branch of the region to its terminus, scanlines 2 and 3 in Figure 5b, the only remaining pixel on the stack is the first pixel to have been placed there. After processing the scanline containing that pixel, scanline 4 in Figure 5c, the two lines above and below that scanline are considered for possible stacking, scanlines 5 and 1 in Figure 5d. Scanline 5 is new and needs to be stacked. Scanline 1 has already been flooded and is at least as large as scanline 4. It should therefore be ignored. Unfortunately, the standard area flooder has retained parental information for only the most recent scanline processed, scanline 3, and is therefore helpless to prevent itself from re-scanning scanline 1. The solution to this problem is to stack the necessary parental information along with the seed positions. Specifically, those variables labelled "yref", "lxref", "rxref" in the Smith algorithm are placed into storage along with the "x" and "y" of the proposed seed point. In this manner, the HINEIGHBOR and LONEIGHBOR procedures may be of greater service to the FILL procedure, preventing the re-scanning of the parent scanline whenever possible.

The number of times that the situation described above occurs is not great since the flooder is usually moving in only one direction at a time. In almost all cases, the single set of parental information retained by the standard flooder is sufficient to avoid re-scanning. The stacked parental values are only useful in this context when the flooder changes direction, as shown in the figure.

On the other hand, the knowledge that this parental information is available during the processing of subsequent scanlines allows for a significant savings in another area. Specifically, the retension of parental information allows the SCANHI and SCANLO procedures to stop after stacking a single seed point. The sample region in Figure 6 illustrates this concept. The standard area flooder, after flooding scanline 1 in Figure 6a, begins to scan its upper and lower neighbors. The standard SCANHI procedure recognizes that scanline 2 is the parent of scanline 1 and stops immediately. The standard SCANLO procedure will find and stack two pixels, one for each of the arms of scanline 3, as shown in Figures 6a and 6b. In order to find the second pixel, however, the procedure was forced to search the entire shadow of scanline 1. Over the course of flooding an entire region, this implies a pixel processing cost proportional to double the number of pixels to be flooded. The reason that SCANLO must scan the entire shadow is that the FILL procedure, when it begins to process scanline 3, is forced to stop as soon as it encounters a boundary pixel and is not allowed to probe any further. If SCANLO did not stack both pixels, the second arm would never be discovered and flooded. If, however, FILL knew its parent scanline, scanline 1 in Figure 6c, and could verify that its parent extended much further rightward than the encountered boundary, it could stack the seed pixel in scanline 4 of Figure 6c and then begin to probe beyond the blocking pixels. It would then discover the second arm, flood it, and re-invoke the scanning procedures to find stack-worthy pixels in scanline 4 of this second arm, as shown in Figure 6d. All of this could be done without the benefit of the second pixel stacked by the SCANLO procedure in Figure 6b.

A cursory analysis of this modification suggests that it has merely shifted the smarts of the SCANHI/SCANLO procedures into the FILL procedure. The great savings in time comes in the fact that the scanning procedures were allowed to stop searching after stacking a single point, which is usually found after a search of only one or two pixels. If more scanning will be needed later, after encountering and penetrating a boundary, the FILL procedure will recognize this during processing which it must do anyway and can re-invoke the scanning procedures at that time. In the greatest majority of scanlines, no second arm will be discovered by the filling procedure, and the scanners will have processed only a few pixels. Therefore, the overall pixel processing cost becomes proportional only to the number of pixels in the region (plus a small constant), rather than to double that cost. The implementation of this technique is given by the algorithm below.

3.2 Improvements To Soft-edge Flooders

The retension of parental information becomes particularly important in the efficient implementation of a soft flooder (tint filler). In this case, the inability of the filling procedure in the standard algorithm to probe beyond where it would otherwise stop (upon encountering an uphill gradient) places the onus of insuring that all arms of a region are flooded entirely on the scanning procedures. These routines are then forced not only to process the entire shadow under the parent scanline, stacking multiple points as they proceed, but to unstack any point which is not optimal according to the gradient rule, replacing it with a better point if one if found during later scanning. If, alternatively, parental information is available, the filling routine assume the burden of finding additional arms during its normal progression across the current scanline. The scanning routines can then stop after finding and stacking the first valid point they encounter and the overall performance of the flooder is greatly improved.

As an illustration of the improvements which can be made to a soft-edge area flooder, the following algorithm is offered.

```
                    *
procedure SOFTFLOOD(seedx,seedy,newcolor); begin
   position seedx,seedy; color newcolor;
   position x,y,firstx,lx,rx;
   position parentlx,parentrx,parenty;
   color oldcolor; shade cents; oldcolor:=GETC(seedx,seedy);
   parentlx:=$left; parentrx:=seedx; parenty:=seedy;
   PUSH(seedx,seedy,parentlx,parentrx,parenty);

   while STACKNOTEMPTY do begin
       POP(parenty,parentrx,parentlx,y,x);
   a:  firstx:=x; FLOODLEFT; lx:=x; x:=firstx;
   b:  SETC(x,y,newcolor); FLOODRIGHT;
       if PROBERIGHT(x+1) then go to b;
       rx:=x; SCANHI; SCANLO;
       if PROBERIGHT($right) then go to a;
   end;
end;

procedure FLOOD{LEFT,RIGHT}; begin
    shade nexts; cents:=GETS(x,y);
    while x{>$left,<$right} do begin
        x:=x{-,+}1; nexts:=GETS(x,y);
        if nexts>cents then begin
            x:=x{+,-}1; break;
        end;
        cents:=nexts; SETC(x,y,newcolor);
    end;
end;

procedure SCAN{HI,LO}; begin
    position scanlx,scanrx,scanx; shade scans,parents;
    if y{<$top,>$bottom} then begin
        scanlx:=lx; scanrx:=rx;
        if y=parenty then begin          [{+,-}1]
            if parentlx<lx then scanlx:=parentrx+1;
            if parentrx>rx then scanrx:=parentlx-1;
        end;      [<]
        if scanrx>scanlx then return;
        else for scanx:=scanlx to scanrx do begin
            scans:=GETS(scanx,y{+,-}1; parents:=GETS(scanx,y);
            if parents>scans and GETC(x,y)=oldcolor then begin
                PUSH(x,y{+,-}1,lx,rx,y); return;
            end;  [scan]
        end;
    end;
end;

procedure PROBERIGHT(limitx); begin
   position limitx; shade nexts,parents;
   while x<parentrx do begin
        x:=x+1; nexts:=GETS(x,y); parents:=GETS(x,parenty);
        if parents>nexts then return true;
        if x=limitx then ~~break;~~   [begin
                                          x:=x-1; break;
                                       end;]
   end;
   ~~x:=x-1;~~ return false;
end;
```

* For other bounding rules,
replace:
nexts>cents
parents>scans
parents>nexts
with appropriate test,
such as:
nexts ≠ cents
or:
nexts = boundary shade
etc.

The reader will notice that procedure and variable names have been selected so as to be fairly similar to those in the Smith algorithm, facilitating comparison between the two approaches. In this algorithm, GETS and GETC (GETV and GETT in the Smith algorithm) require two input parameters and return either the shade bits (value) or the color bits (tint) of the pixel at the specified X and Y position. The SETC procedure (SETT in Smith) requires three input parameters and replaces the color bits (tint) of the pixel at the specified X and Y position with the specified color. The PUSH routine pushes the specified values on the stack in the order shown. The POP routine pops the stack into the specifed variables, also in the order shown. The curly bracket notation (e.g. FLOOD{LEFT,RIGHT}) indicates that two procedures are required, one containing the first token enclosed within the brackets and one containing the second token. This notation is used whenever two near-mirror-image procedures are required.

4.0 OPTIMIZING THE IMPLEMENTATION OF AN AREA FLOODER

The optimization of an area flooder is highly dependent on the hardware configuration for which it is implemented. This makes generalized suggestions for optimization difficult to propose. In light of this, it seems most instructive to assume a particular hardware configuration that seems in some sense typical and to demonstrate how the area flooding algorithm can be tailored for maximum efficiency within the opportunities and constraints posed by that particular configuration.

The configuration that will be used consists of the following components:

1. A virtual-memory host processor containing sufficient real memory to maintain a complete copy of the image being flooded without paging.

2. A generalized set of micro-coded hardware instructions available to the machine language programmer, such as instructions that will find the next instance of a particular bit pattern, fill a block of storage with a particular pattern, move a block of storage from one location in memory to another, pass a block of storage through a translation table, find the first set or cleared bit in a block of storage, and so on.

3. A colormapped 8-bit frame buffer interfaced to the host processor by a scanline-oriented channelled I/O data path.

4. A generalized set of hardware coloring instructions in the frame buffer, such as the drawing of rectangular boxes, vectors, text, and so on.

Some readers may smile and recognize the equipment being described, particularly if they own a similar configuration themselves. In any case, the configuration is not grossly atypical, and represents capabilities available in many state-of-the-art mini-computers and color display systems.

4.1 Optimizing Hard-edge Flooders

The key to designing an optimized area flooder is to reflect back on the algorithmic improvements suggested in the previous section. The ability of the scanning procedures to stop after finding the first stackable point implies that they will process only a few pixels at the left edge of the shadow of the current scanline rather than sailing clear across it. This reduces the per-pixel cost of processing each scanline, which in turn decreases the correlation between the size of a region and the time required to flood it. If some means could be found to reduce or eliminate the per-pixel costs associated with the flooding procedure itself, the cost of flooding would depend solely on the number of scanlines processed, not on the number of pixels.

The flooding procedure is really responsible for two separate tasks. The first is to find the limits of the current scanline according to the current rule of proliferation. The second task is to replace the color of each pixel within the found limits according to the current coloring scheme (filling, tint filling, texture filling, etc.) These two steps are logically independent, although the second must follow the first. They do not, however, have to be performed simultaneously.

Without belaboring this discussion, it becomes evident that the micro-coded machine instructions of the sample host processor, coupled with the hardware coloring instructions of the frame buffer, can serve to accomplish these two tasks very efficiently. Specifically, the boundary color can be found (in the example of boundary filling) by executing the single machine instruction that searches rightwards through storage for the particular bit pattern representing the boundary color. The scanline thus delimited can be flooded in the host processor image array by executing the single machine instruction that fills a block of storage with a given bit pattern, the seed color. The frame buffer can be made to reflect the flooded scanline by executing the single display instruction that draws a single-pixel high horizontal box extended the full length of the current scanline and composed of the seed color. The result of these optimizations is that the cost of flooding a region is dependent solely on the number of scanlines involved rather than on the number of pixels. This makes for a very fast flooder.

4.2 Optimizing Soft-edge Flooders

The problem of optimizing a soft flooder is somewhat more difficult. In this case, the proliferation rule (never go uphill) requires a comparison between each pixel and its neighbor, with different actions to be taken depending on the sign of the result. In lieu of a single micro-code instruction capable of performing this complex operation, an alternative procedure can be utilized. By subtracting each pixel in the image array from its neighbor (to the right, for example), and storing the sign of the result as a single bit in a separate array, a mask is generated that indicates the location of all uphill gradients in the selected direction. By computing this mask prior to the commencement of flooding, it is available for use by the flooding algorithm during the processing of each scanline. The flooding procedure may now locate the limits of the current scanline simply by executing the single machine instruction that searches the mask for the first set bit, which indicates an uphill climb. In combination with the fact that the seeds left by the improved scanning procedure are generally at the left edge of the floodable portion of a scanline, this means that a single machine instruction can again find the limits of the current scanline.

The coloring task in soft flooders is also more complicated that its counterpart in hard flooders. Rather than simply replacing the found color by the seed color, it is necessary to replace only the color bits, leaving the shade bits intact. In the host processor, this can be accomplished by the single machine instruction that passes a block of storage through a translation table that modifies only the color bits of each pixel value while leaving the shade bits unaltered. Reflecting this re-coloring in the frame buffer can be handled in a similar fashion. Many frame buffers offering hardware rectangular boxes also provide the capability to affect only selected bits of each pixel value rather than all bits. These bitplane selection options can be utilized to suppress modification of the shade bits during the writing of the single-pixel high horizontal box described previously.

The computation of the mask required to find the limits of the current scanline is not as simple a job as might be imagined. It would be highly inefficient to pre-process the entire drawing in this manner prior to flooding, since only a small proportion of the pixels are likely to be flooded. This is particularly true in cel animation, where the cartoon character occupies only a small fraction of the total area of the frame buffer. Fortunately, there are ways to lessen the cost of this pre-processing step. Most frame buffers that include the capability of optical scanning include some type of hardware image processor. It is a simple matter to produce a copy of each drawing during scanning offset by one pixel in the desired direction. This copy can then be subtracted from the original using the hardware image processor and the sign of the result stored in a separate bitplane. This becomes the gradient mask required by the optimized flooder. This operation is extremely fast, falls naturally into the drawing scanning sequence, and incurs no cost to the host processor. If such a hardware image processor were not available, it becomes somewhat questionable whether or not this particular optimization is still of value.

In all of the implementations discussed here, sequences of machine instructions have been replaced by single micro-coded instructions. It is important to realize that this optimization does not necessarily offer as great a speed improvement as might be imagined. The single machine instruction that moves a block of storage from one location in memory to another in this example must still establish a do loop, compute each source address, fetch each value, and so on. The fact that these operations are occurring in micro-code makes them faster, but not infinitely fast. In fact, some sequences of micro-coded instructions are better executed by a greater number of simpler instructions in sequence surrounded by a single do loop because the total looping overhead is reduced. This consideration is important to the user of any micro-coded instruction set, and should be given particularly careful attention by the implementer of an area flooder.

ACKNOWLEDGEMENT

I wish to acknowledge Theodore Crane of the Cornell Program of Computer Graphics, who first suggested to me the idea of retaining parental information on the stack.

REFERENCES

1. Levoy, Marc, "A color animation systēm based on the multiplane technique", Siggraph 1977 conference proceedings, July, 1977

2. Levoy, Marc, "Frame buffer configurations for paint programs", Siggraph 1980 Animation Graphics tutorial notes, July, 1980

3. Smith, Alvy Ray, "Tint fill", Siggraph 1979 conference proceedings, August, 1979

4. Stern, Garland, "SoftCel - An application of raster scan graphics to conventional cel animation", Siggraph 1980 conference proceedings, July, 1979

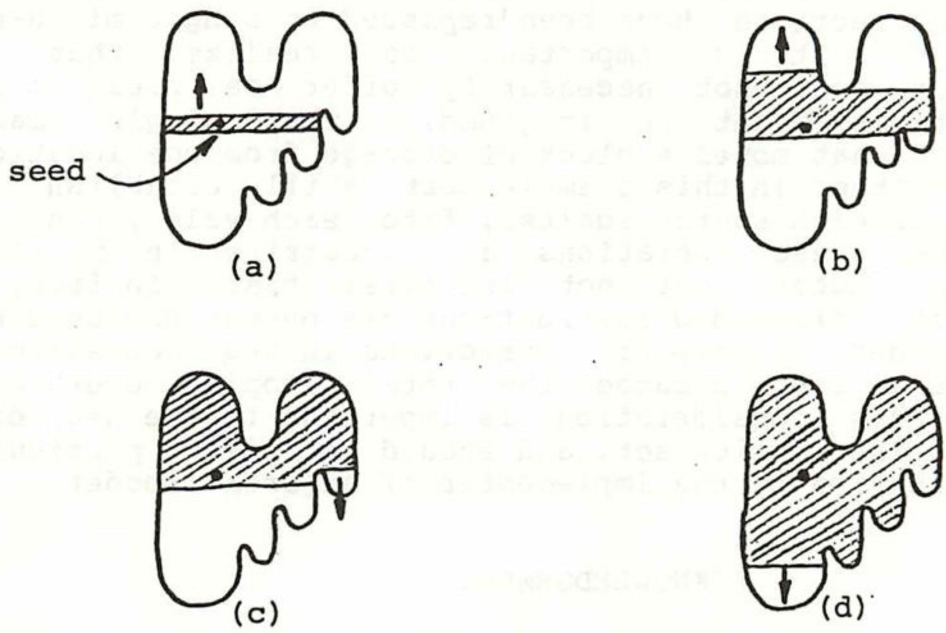

Figure 1: Scanline-oriented stack algorithm

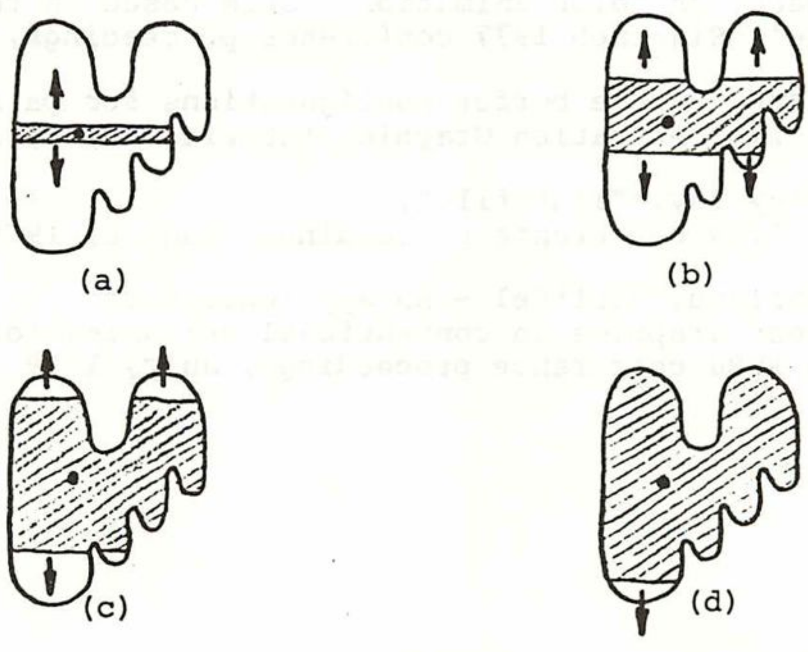

Figure 2: Scanline-oriented queue algorithm

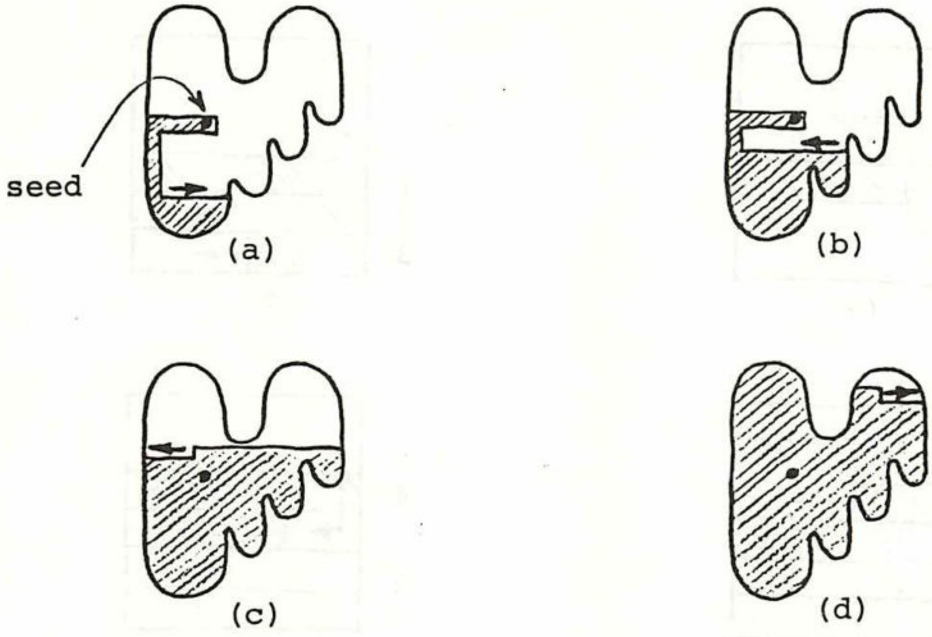

Figure 3: Pixel-oriented stack algorithm

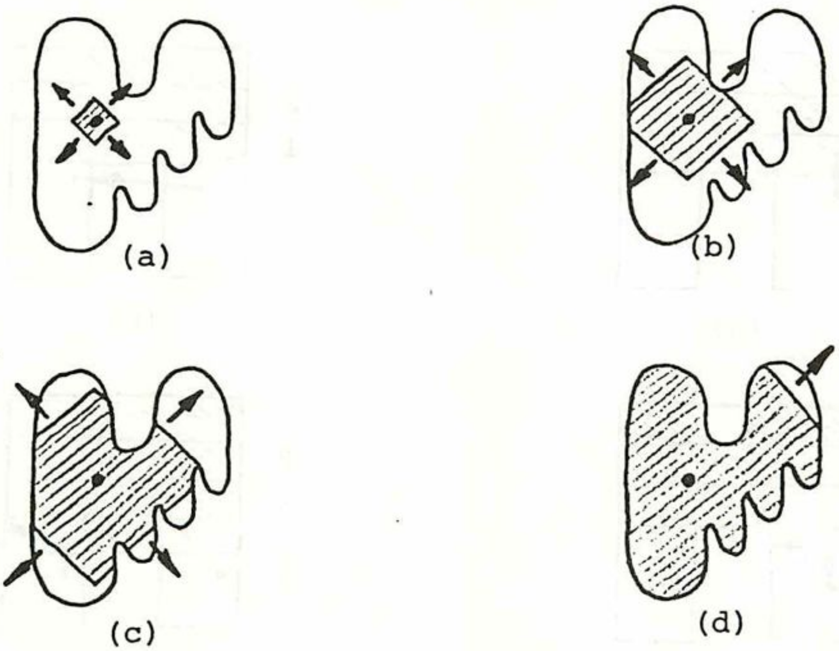

Figure 4: Pixel-oriented queue algorithm ("diamond")

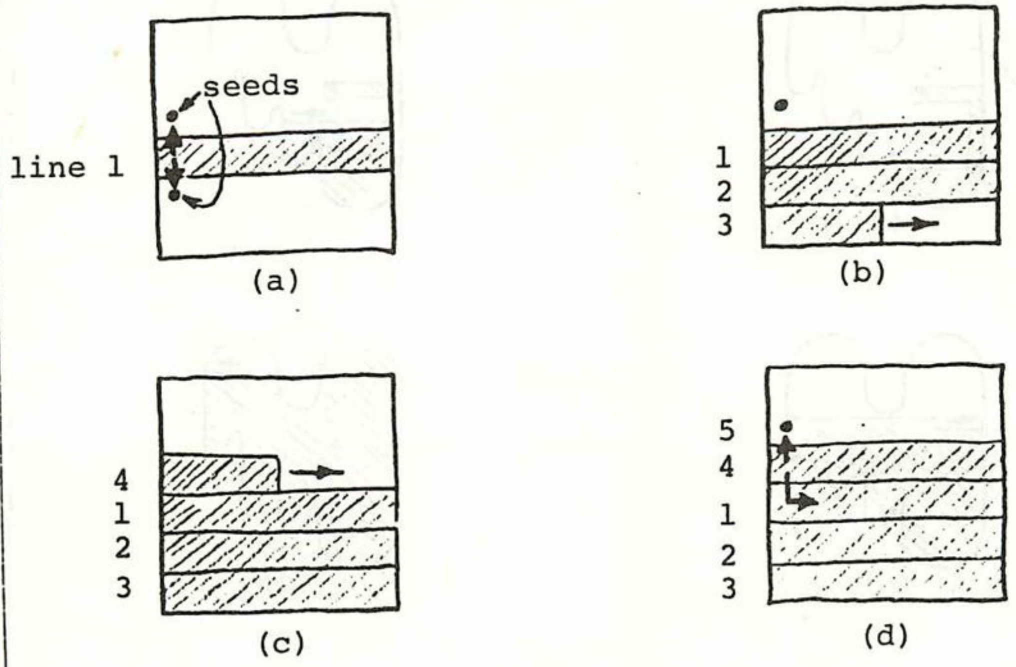

Figure 5: The need for retaining parental information

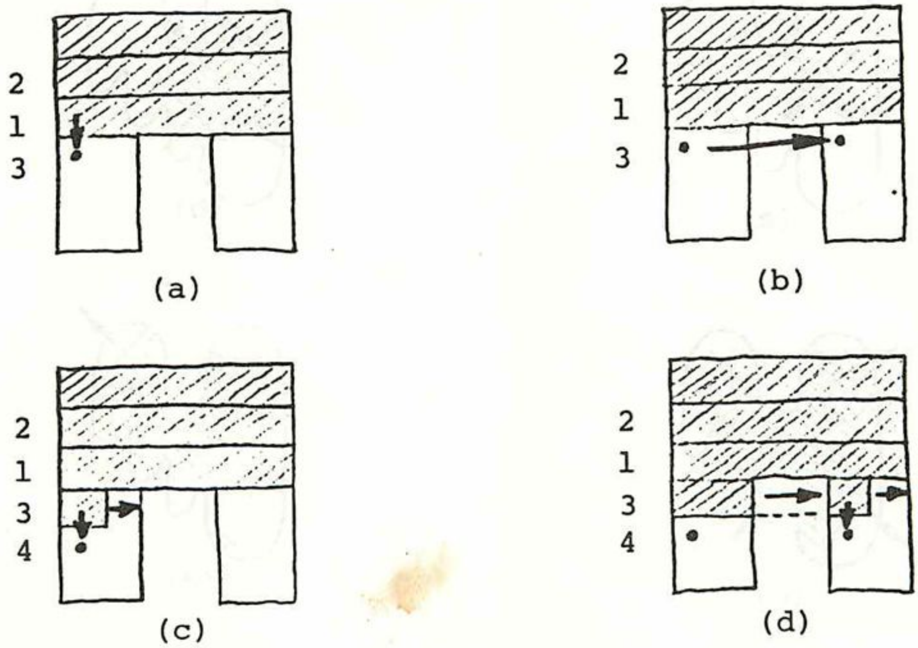

Figure 6: The advantage of parental information