How To Color In A Coloring Book

Henry Lieberman

Artificial Intelligence Laboratory
Massachusetts Institute of Technology

CR Category: 8.2 (Computer Graphics)
Keywords: Shading (or filling, flooding) closed curves raster graphics

Abstract

Children's coloring books contain line drawings which a child can fill in with a crayon to produce colored pictures. Two dimensional colored areas can be produced on a raster display by an analogous method. After drawing a closed curve with line drawing commands, the graphics system can fill the area bordered by the curve. This paper presents an algorithm for filling in areas of any size or shape. The area may be filled with any color, texture, or "wallpaper" pattern. The algorithm is simple, flexible and efficient, optimized to take advantage of the the memory organization of most current raster graphics systems.

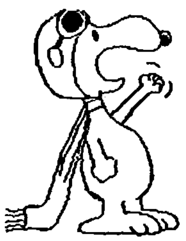
Figure [1]

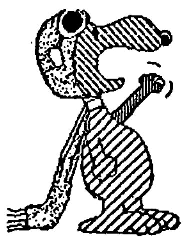
Figure [2]

Here's a simple example of the shading program at work. We start out with a line drawing of Snoopy, similar to the kind of drawing that might appear in a child's coloring book.

We tell the system which area we want to fill by picking a point with the graphics cursor. A ***shading pattern*** tells the system how to fill the area. In this example, Snoopy's body is filled with diagonal lines, his hat and scarf with random points, his arm with vertical lines.

1. Introduction

For years, the predominance of vector displays for computer graphics has limited computer generated pictures to drawings consisting solely of lines and points. Now, the development of raster displays allows pictures containing two dimensional solid and textured areas. What are good ways of describing regions of various sizes, shapes and colors to the computer? This paper explores one answer to this question.

The computer's display screen can be treated like a *child's coloring book*. The coloring book supplies line drawings indicating the borders of areas to be filled in The child takes a crayon, picks a place in the interior of an area, and fills it in by continually expanding the colored area until it hits the boundary lines between areas.

Most currently available computer graphics systems provide a wide range of facilities for making line drawings. These can be used to draw the outlines of regions to be filled in. The graphics system can be extended with a primitive called SHADE (FLOOD in some systems), which is given an interior point, and fills in the area. A "crayon" of any color may be chosen, or textures, shading or wallpaper patterns may be used to fill the area in instead of a solid color. (See Figure [1] - Figure [2].)

This provides a conceptually simple and flexible means of creating and manipulating two dimensional areas. The program doesn't have to know the size or shape of the region in advance. Since it searches the screen to find the extent of the area, it is not tied to a particular representation for the region's boundary. The outline of the region may be produced by a line drawing program, hand drawn by an artist using a tablet or other graphic input device, or taken from a photograph or television picture. This is a popular operation in "paint" programs, which allow a user to input line drawings on a tablet, outlining areas, and have the computer *paint by number*, filling in each area with a paint of a chosen color.

Here's another way of thinking about the problem: Consider the task of building a robot vacuum cleaner. The robot should be able to vacuum a room of any size or shape, containing furniture or other obstacles. It should make sure to clean every spot on the floor, without getting stuck by hitting the walls or furniture. It must also avoid getting into an infinite loop cleaning the same spot over and over again, and must be able to tell when it is finished. Since computer vision systems are expensive and difficult to build, and the robot should be simple and cheap, a further restriction will be imposed that the robot will be *blind* - it won't have a TV camera to watch for obstacles. Instead, it will be equipped with touch sensors, so it will be able to detect when it hits something or brushes up against it. It can only tell whether there's an obstacle adjacent to it in front or on the side. Given these constraints, the procedure presented in this paper will enable the robot vacuum cleaner to efficiently get the job done.

2. Description of the Algorithm

The shading process starts out with a point known to be in the interior of the figure to be shaded. (Of course, the process should not depend upon which initial point is chosen.) The procedure determines the extent of the figure by continually examining points adjacent to known interior points. If a point adjacent to an interior point is the same color as the interior, it can also be considered an interior point. If a color change is noted, then that point is part of the boundary of the region.

In what order shall the points on the screen be examined? This choice is crucial to the efficiency of the resulting procedure. One possibility is to spiral out from the initial point, looking at points at ever increasing distance from the initial point. Another might be to have some procedure for following the boundary of a figure.

Since the screen memory in most raster graphics systems is organized into words, where the memory representing several horizontally adjacent points occupies a single word, the scan for the boundary of the figure should be made *horizontally* as much as possible. It is often the case that several points which are horizontally adjacent may be read or written with a single memory operation, whereas manipulation of points on separate lines may require a separate memory operation for each point. Further, the choice of using horizontal scans minimizes the amount of state information the procedure must maintain to keep track of what parts of the figure it has already covered. The result of scanning horizontally can be represented as the left and right X values and one Y value, rather than a list of points chosen in some arbitrary order.

The procedure, then, will be built up out of *horizontal scans.* Each scan starts at an interior point, looking to the left until a color change indicating the boundary of the figure has been reached, then looking to the right until the scan hits the boundary. This will yield a set of horizontally adjacent points known to lie entirely within the boundary of the figure. Those points can be shaded by applying the shading pattern.

Let's consider what the shading procedure must do wnen it is moving from one line to the next. The procedure must now scan *vertically*, picking a new interior point adjacent to the previously shaded line on the line above or below, and performing a horizontal scan from that point.

If there are no such points, if all points adjacent to the previously shaded line are boundary points, then it is an indication that the top or bottom of the region has been reached. The boundary then completely encloses the area shaded, and the job has been completed.

A simple figure such as a square or a circle could be shaded by starting two vertical scans from the initial point. After performing a horizontal scan, the procedure would move up to the next line, continuing until the top of the figure was reached. It would return to the initial point, then shade downward to the bottom of the figure. However, this simple method would fail to work for more complex figures

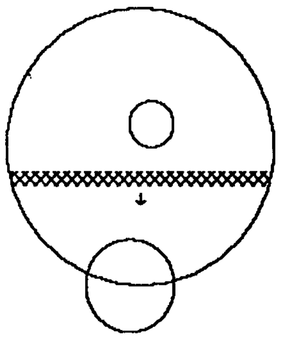
Figure [3]

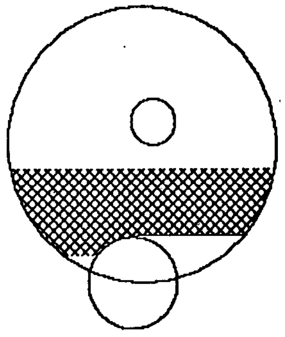
Figure [4]

The next group of figures shows some of the control structure of the shading process as it fills a region. The picture consists of a large circle, with a hole in the middle and an overlapping circle at the bottom. The procedure must, of course, avoid shading inside the hole, and must shade around the overlapping circle.
The procedure scans horizontally on each line, to the left and right until the boundary of the area is reached, then displays the shading pattern. It then moves down to the next line and continues.

Eventually, it encounters the top of the overlapping circle. It continues down the left side of the circle, but remembers that it has to return later and complete the right side. It saves an ***imaginary boundary line*** from the top of the circle to the right side of the area on an ***agenda*** list. The imaginary boundary line allows the procedure to pretend that it has already completed the area to the right of the small circle.

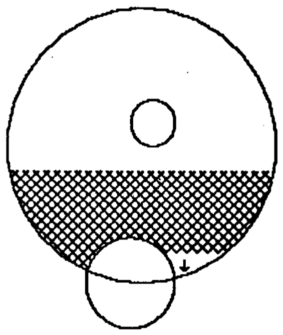
Figure [5]

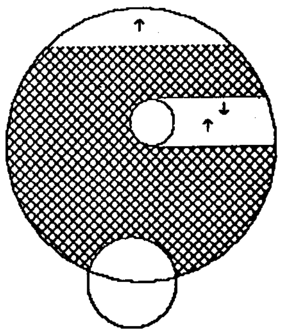
Figure [6]

Upon completing the area to the left of the small circle, the procedure returns to the agenda list to see if anything more remains to be completed. It finds that both the right side of the circle and the area above the initial point need to be shaded. It picks the one going downward, as it always prefers to continue in the same direction as the last vertical scan.

Now, the only task left is filling the area above the initial point. The procedure proceeds upward, to the left of the hole. Two imaginary boundary lines are now placed on the agenda, one going upward from the bottom of the hole, one going downward from the top. At this point, the program can't tell that they would both attempt to cover the same area!

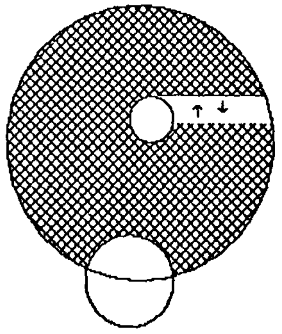
Figure [7]

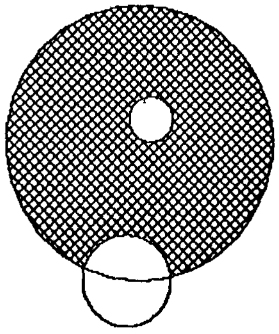
Figure [8]

When the top of the large circle is reached, it starts shading upward from the bottom of the hole. After finishing one horizontal line, it always checks to see if it hits an imaginary boundary line before moving up or down to the next line. Thus, the procedure will hit the line from the top of the hole. When that happens, the vertical scan is stopped, and the line with which it collided is also removed.
Now, all of the figure has been shaded. There are no more lines left on the agenda, so the procedure stops.

containing *concave* portions. If we tried to shade a U-shaped figure starting from the bottom, the procedure would go up one branch, and ignore the other one!

After shading a line, the procedure must then cover *all* interior points adjacent to that line. Instead of stopping after one horizontal scan, it must examine all points adjacent to the shaded line, possibly yielding several more lines to be shaded, rather than just one. Further, after shading the new line, all points on the old line adjacent to the newly shaded line must be examined as well. If the new line is longer, there may be interior points adjacent to the new line, but separated horizontally from the old line by boundary points. The new interior points on the old line should start another vertical scan going in the opposite direction from the scan that discovered them, so these are called *U-turns*. The new points adjacent to the previously shaded line initiate a new scan in the same direction, and are called *S-turns*.

If the system allows *parallel processing*, a new vertical scanning process can be started for each new line found. On a serial machine, one of the lines must be chosen to continue the current vertical scanning process (by convention, we choose the *leftmost* line). The program then remembers the other lines on an *agenda* list, as a reminder that it must do another vertical scan. At the conclusion of the current scan, the agenda is checked to see if anything more remains to be done, and the program loops until the agenda has emptied.

There's just one more problem to worry about: What if the figure has a *hole* in it? As described so far, the procedure would just keep circling around and around the hole forever!

The process must therefore have some method for keeping track of what areas of the screen it has already covered, and stop whenever it retraces its footsteps. If we were limiting the program to shading in an area with a solid color, there would be no problem, as previously shaded areas would provide a barrier which would stop any other scan running into them. Allowing arbitrary patterns means that the procedure cannot use the screen to record such information. Expensive solutions such as keeping an auxiliary array to record which points have been shaded are out of the question.

If we think about all the vertical scanning processes running in parallel, the agenda represents a list of *leading edges* of the scans as they proceed to shade more and more of the figure. The shaded area grows by shading new lines adjacent to previously shaded areas. We can recognize that the area shaded is always completely enclosed by the boundary of the figure, the current line being shaded and the lines on the agenda. Thus, it is sufficient merely to check if the scan is colliding with a line on the agenda to prevent it from shading an area already covered.

The lines on the agenda serve as *imaginary boundary lines* and stop the progress of the search as do the real boundary lines of the figure. Once the process collides with an imaginary boundary line, the imaginary boundary is removed from the agenda, so that it will not start another scan itself.

To minimize searching and comparisons, it is convenient to keep the agenda in the form of two lists, one of scans to be performed in the upward direction and one for downward scans. Each list is kept ordered by Y value, upward scans from top to bottom, downward from bottom to top.

The top level procedure, then, starts out with an agenda containing an upward scan and a downward scan starting from the initial point. Each of the vertical scans might result in other scans being added to the agenda, in both directions. The top level procedure does all the scans in one direction at a time, then reverses direction and chooses scans from the other list. Each time, the procedure chooses the highest scan if it is going upward (or lowest one if it is proceeding downward). This insures that it is only necessary to check against the list of scans going in the opposite direction. That list is ordered in the direction of the current scan so only the remainder of that list which lies beyond the current point need by checked. When both lists have been exhausted, the region has been completely shaded, and the procedure is finished.

3. Proof of the Shading Procedure

Can we be sure that the shading procedure actually performs as intended? What follows is an informal sketch of a proof that the procedure outlined above has the desired result.

First, it is necessary to check that the shading procedure shades *only* points which lie in the interior of the figure. This is true since the procedure starts out at a point known to be in the interior of the figure, and at each step, only shades points which are not boundary points, and which are adjacent to points already known to be interior points.

□ **Interior points**
▩ **Boundary points**
⬚ **Imaginary boundary points**
▨ **Shaded points**
→ **Scan in direction of arrow**

Figure [9]

The next group of illustrations will show the operation of the shading procedure in more detail, down to the level of moving from point to point on the screen. Individual points (or *pixels*) will be represented by one of five different types of boxes. Points in the interior of the figure will be shown as blank boxes. Boundary points, those of a different color surrounding the figure will be shown with cross-hatching. The points will start out as either interior points or boundary points. As the shading procedure passes over a point, it will either shade the point (which will then appear as a box with diagonal lines), or consider it as an *imaginary boundary* point (shown with dots). Imaginary boundary points are placed on an *agenda* list to be shaded later. The current point being scanned is indicated by a box with an arrow pointing in the direction of the scan.

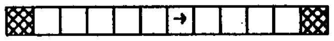

Figure [10]

We start shading a line of interior points, with two boundary points at either end. The scan starts rightward from the first point.

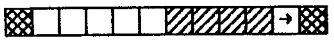

Figure [11]

Points are shaded until the boundary point is reached.

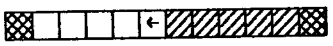

Figure [12]

The scan then proceeds leftward from the initial point until the left boundary is reached.

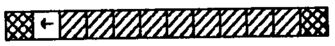

Figure [13]

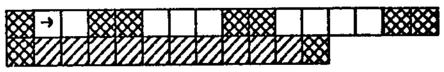

Figure [14]

We will assume that the scan is proceeding vertically upward. A new point on the line above is chosen, the leftmost interior point adjacent to the previously shaded line.

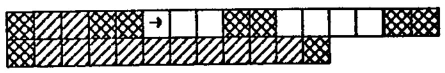

Figure [15]

The horizontal scan continues shading to the left until a boundary point is reached. Now, the scan continues past the boundary, looking at points adjacent to the previously shaded line, since these are also part of the interior of the area.

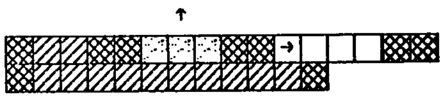

Figure [16]

The procedure will defer shading these points until later. A notation is made on the *agenda* list that a new scanning process must be initiated shading upward from those points. This is called an *S-turn*, since the new process will shade in the same vertical direction as the one which discovered it. The points are marked as *imaginary boundary* points. Any other scan which tries to shade past them will be stopped, since the area below has already been shaded. They stop the progress of the shading procedure as do the real boundary lines of the area.

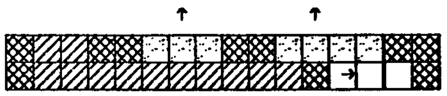

Figure [17]

Another S-turn has been found. Now, the procedure shades points on the lower line which are adjacent to shaded points on the upper line, as these are also part of the interior. These start a new vertical scan going downward. Since these reverse the vertical direction of the scan which discovered them, they are called *U-turns*.

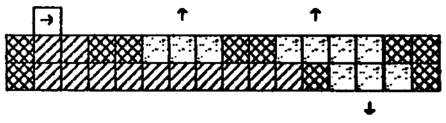

Figure [18]

A third line is now started, beginning from the leftmost interior point adjacent to the second line shaded.

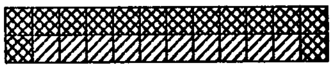

Figure [19]

Two situations can terminate a vertical scan. If, after shading a line, every point adjacent to the line is found to be a boundary point, then the top or bottom of the figure has been reached and the vertical scan stops.

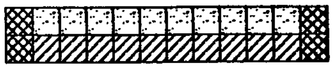

Figure [20]

If every point adjacent to the shaded line is an imaginary boundary point, this indicates that the region on the other side of the imaginary boundary has already been shaded earlier in the process. This situation arises when shading around holes in the area. There's nothing left for the vertical scan to do, so it stops. The imaginary boundary is also removed from the agenda, since it would normally start a new vertical scan downward, but this is no longer necessary.

The procedure must be verified to shade *all* of the interior points. Suppose there's some point which has been missed by the shading procedure but is still inside the region. It must be adjacent to some point which has been shaded. We will show that this cannot occur, that if a point has been shaded, every interior point adjacent to it must also have been shaded. Each point, P, has four neighbors in the raster grid, one on each side horizontally, one on the line above and one below. Since each horizontal scan starts out on a leftmost interior point and proceeds rightward, the horizontal neighbors of P must be shaded if P has been. Consider the neighbor of P on the line above, its *upstairs neighbor*. If there is no boundary point between the start of the horizontal scan on that line and P's upstairs neighbor, then the horizontal scan will cover it. But if there is, the part of the shading procedure which detects turns will detect this situation and put a new scan on the agenda which will cover P's upstairs neighbor (in this case an S-turn, if we take the vertical scan to be going upward). A similar argument will show that the downstairs neighbor of P will be covered by a U-turn.

The procedure never retraces a point which it has already shaded. This is true because the following property remains *invariant* during the execution of the program. At all times, the area shaded by the procedure is completely bounded by the boundary of the figure, the imaginary boundary lines on the agenda, and the line currently being scanned. The procedure is designed so that it *stops* whenever the scan hits the boundary of the figure, or an imaginary boundary line, so the scan can never break through a real or imaginary boundary to retrace part of the area it has already covered.

Finally, the procedure must be shown to terminate. Since it continually shades more and more points, never retracing a point that it has already shaded, and the number of interior points is finite, the procedure eventually stops.

4. Shading Patterns

How does the shading procedure decide what to display on the screen to fill the area once it's discovered where the boundary lies? The set of directions for filling the area, the *shading pattern*, is itself a procedure, to afford flexibility in choosing different methods of shading. The system supplies a small predefined set of patterns, which are usually sufficient for common uses of the shading feature to distinguish visually between several neighboring regions, like countries on a map. These patterns include solid colors, horizontal, vertical and diagonal lines, and crosshatching. An interesting *texture* effect is created by turning on *randomly* chosen points within the area.

Saved pictures, containing any mixture of lines, points, text, or other areas, may also be used as shading patterns. The pictures are normally stored by the system as bit arrays or using run length encoding (called *windows* in [1]). If the region to be filled is larger than the saved picture, the picture is repeated "wallpaper" style as necessary. If the region is smaller than the boundary, it is *clipped* against the boundary of the figure. The ability to clip a picture against an arbitrary boundary is an operation which is often useful in its own right.

Alternatively, the user may provide his or her own function to do any specialized computation. For example, it might be desired to check the distance of a point from a *light source* before deciding on the brightness of the point. The shading pattern function accepts coordinates on the screen, and is responsible for updating the display. Rather than apply the shading function once for each point, it is called to shade a whole line instead, to take advantage of the fact that accessing points horizontally is especially efficient. A further area of experimentation would be to create higher level means of specifying shading patterns, which would not be so closely tied to the coordinates of the place being shaded.

5. Implementation

The shading program is part of the *TV Turtle*, a graphics system for raster displays developed by the author and described in [1] and [2]. It is implemented in MacLisp [5] on the PDP 10, and can be used either from Lisp or our Lisp implementation of Logo.

Implementations of a shading procedure for raster displays have also been developed at the MIT Architecture Machine Group, shown in [3], Xerox PARC [6], and several others, but details of the operation of these systems have not been previously published in the literature to my knowledge. For previous work concerned with the somewhat different problem of shading a figure whose boundary is given by a vector display list, (which also forms the basis for many hidden surface algorithms) see [4].

We now present a sketch of an implementation of the algorithm. (Names of important functions and variables are capitalized.)

```
Define SHADE:
* The arguments to SHADE are:
 * The ORIGIN, an interior point to start
   shading from, and
 * A SHADING-PATTERN to fill the area.
* Let INTERIOR-COLOR be the color of the ORIGIN.
* Set the VERTICAL-DIRECTION to be UP.
* INITIALIZE-THE-AGENDA.
* Repeat until the agenda is empty in both
  directions:
  Vertical Shading Loop:
    * If    * The agenda in the current
              VERTICAL-DIRECTION is empty,
      then  * Switch directions.
    * Choose a vertical scan from the agenda.
    * SHADE-VERTICALLY in the
      current VERTICAL-DIRECTION.
  End Vertical Shading Loop.


Define INITIALIZE-THE-AGENDA:
* Enter a scan starting UPward from the ORIGIN.
* Enter a scan starting DOWNward from the ORIGIN.
```

```
Define SHADE-VERTICALLY:
* The arguments to SHADE-VERTICALLY are:
 * A VERTICAL-DIRECTION, either UP or DOWN, and
 * A starting POINT from which to begin shading.
* Repeat until
 * either  * All points adjacent to the
             previously shaded line are boundary
             points,
   or      * All such points are contained in
             lines on the agenda.
 Horizontal Shading Loop:
  * SHADE-HORIZONTALLY from the POINT.
  * LOOK-FOR-TURNS.
  * Move the POINT up or down,
    in the current VERTICAL-DIRECTION.
 End Horizontal Shading Loop.

Define SHADE-HORIZONTALLY:
* Start at a POINT, known as an interior point.
* Look at successive points to the LEFT
  until a color change occurs, indicating
  the boundary has been reached.
* Look RIGHT until the boundary is encountered.
* Remember the points where the boundary was
  found as LEFT-BOUNDARY and RIGHT-BOUNDARY.
* Fill in the SHADING-PATTERN on the line
  between the boundaries.


Define LOOK-FOR-TURNS:
* LOOK-FOR-S-TURNS.
* LOOK-FOR-U-TURNS.

Define LOOK-FOR-S-TURNS:
* If     * There are interior points on the
           new line adjacent to the previously
           shaded line,
  then   * Add these as to the agenda a new
           imaginary boundary line, starting
           a scan going in the current
           VERTICAL-DIRECTION.

Define LOOK-FOR-U-TURNS:
* If     * There are interior points on the
           previous line adjacent to the
           newly shaded line,
  then   * Add them to the agenda as an
           imaginary boundary line going in the
           opposite direction to the current
           VERTICAL-DIRECTION.
```

Acknowledgments

I would like to thank Ken Kahn for many stimulating discussions and helpful ideas. Ken has had a major influence on my thinking about graphics, and as a user of my graphics system, has put some of my ideas to a practical test. Hal Abelson made insightful comments on an earlier draft of this paper and discovered an important bug in a previous version of the algorithm. Carl Hewitt, Bruce Edwards, Ron Lebel, Gerhard Fischer, Andy Di Sessa and others at the MIT AI Lab and MIT Architecture Machine Group also deserve thanks.

Bibliography

[1] Lieberman, H., The TV Turtle: A Logo Graphics System for Raster Displays, ACM SigGraph/SigPlan Graphics Languages Symposium, April 1976

[2] Goldstein, I., Lieberman, H., Bochner, H., Miller, M., LLogo: An Implementation of Logo in Lisp, Logo memo 11, MIT Artificial Intelligence Lab, March 1975

[3] Kahn, K., Lieberman, H., Computer Animation: Snow White's Dream Machine, Technology Review, October 1977

[4] Reynolds, C., A Multiprocess Approach to Computer Animation, MIT Master's Thesis, August 1975

[5] Moon, D. A., MacLisp Reference Manual, MIT Laboratory for Computer Science (formerly Project Mac)

[6] Learning Research Group, Personal Dynamic Media, Xerox Palo Alto Research Center technical report

[7] Negroponte, N., Raster Scan Approaches to Computer Graphics, Computer Graphics, Vol. 2, No. 3