D

Appendix D
PROOF'S RENDERING ALGORITHMS

This appendix describes the algorithms Proof uses to render geometry (lines, arcs, etc.), text, and slide files (.BMP, .PCX, and .RTF) as pixels (dots) on the screen. Although this is a complex topic, most Proof users will find it unnecessary to read this appendix in detail, because Proof was designed to comply with the "Law of Least Astonishment." In most cases, you will find that Proof works the way you expect it to work; your intuition will serve you well. In some situations, however, there may be no intuitive rules to follow. For example, what happens if an Object moves across a Bar or a Message at the same time the Bar or Message is being updated? This appendix answers such questions.

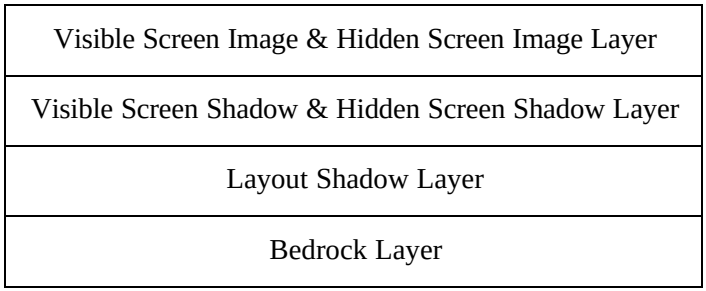
Rendering Considerations

The following considerations will be discussed throughout this appendix:

- **Logical Layers** - Proof maintains a number of logical layers beneath the topmost layer visible on the screen. In certain situations, changes made in a given layer require restoring pixels from a lower layer.
- **Color Priorities** - Colors have priorities. In many, but not all situations, higher priority colors take precedence over lower priority colors.
- **Order of Writing -** The order in which items are drawn is important. When two separate write operations attempt to alter the same pixel, and there are no priority rules to resolve the conflict, the write operation that takes place *last* will determine the color of the pixel.
- **Collision Detection** - Many, but not all, forms of collisions among Objects, Bars, Plots, and other layout elements are detected and handled automatically. Some rare forms of collisions, e.g., overlapping Bars, are not detected automatically.

Logical Layers Maintained by Proof

Proof maintains four logical layers:



The top layer includes the image seen on the screen, and a hidden image. Proof applies all screen updates to the hidden image and flips the roles of the visible and hidden images when the next vertical retrace takes place. A vertical retrace is a pause that takes place periodically as the video hardware repaints (refreshes) the screen. The hardware paints the screen from left-to-right and top-to-bottom. When it has painted the lower right corner of the screen, the hardware pauses to reposition its electron beam to the top left corner of the screen. A hardware signal is raised, indicating that a vertical retrace is taking place. During the vertical retrace interval, flipping of the visible and hidden images can be performed seamlessly. Proof bases its timing on the frequency of vertical retrace signals. For most video hardware, vertical retraces take place at least 60-70 times per second.

Note that because Proof maintains two images at all times, every change shown on the screen must be drawn twice, once for each image. When an animation is running, Proof must keep track of such things as “old odd color” and “old even color.” Even simple color changes require a 2-cycle update. Such details are handled internally by Proof; you don’t have to worry about them.

The second layer is a pair of “shadow” copies of the screen images in the top layer. We will note their role here, but throughout the rest of this appendix, we will rarely distinguish between screen images and their shadow copies. Shadow copies are used for such things as restoring the screen when a dialog box is moved or deleted. For example, when you draw a line in Draw Mode, a line-drawing dialog box is shown on the screen. Suppose you draw a line, all or part of which ends up under the dialog box. When you’re done drawing the line, and the dialog box is removed from the screen, the line instantly appears in its proper position. This is made possible by drawing the line in the shadow image and restoring the portion of the screen obscured by the dialog box from the shadow image. In other words, the line is drawn in the shadow image, which is “under” the dialog box.

The third layer is a shadow copy of the complete layout. Moving Objects are excluded from this

layer. When an Object moves, the area it previously covered is restored from the layout shadow, subject to color priority rules.

The bottom (bedrock) layer is a copy of the layout which excludes Layout Objects. If a Layout Object is moved, the area it previously covered is restored from the bedrock layer. In some applications, it is convenient to pre-position equipment by creating Layout Objects in Draw Mode. If a Layout Object is moved, it is automatically converted from a Layout Object to an "ordinary" (movable) Object.

In the remainder of this appendix, we will present additional details on the roles of the logical layers described above.

How a Layout is Drawn

A layout is drawn as follows:

1. The Proof window is cleared to the backdrop color.
2. If a the current view has multiple windows, borders are drawn around each window.
3. If a grid is required, one is drawn. Dots and text comprising a grid are drawn with low priority. Grid dots and text overwrite only Backdrop-colored pixels. If the active backdrop color is a dark color, grid dots and text are drawn in white. If the active backdrop color is a light color, grid dots and text are drawn in black.
4. All Lines and Arcs are drawn, ignoring color priorities. Thus, the order in which Lines and Arcs appear in a layout file makes a difference. For example, if a red Arc is tangent to a white Line, there will be at least one overlapping pixel. The layout element drawn last determines the color of the overlapping pixels. When you edit a layout file, each time you select a layout element for editing, that element is placed at the end of the entire list of layout elements. The original order of elements in a layout file will change with almost every editing session. Thus, depending on the order in which layout elements appear in a layout file is risky.
5. Fills are drawn. Fills must be drawn after all Lines and Arcs, because Lines and Arcs comprise the boundaries of Fills.
6. Layout Text, Message, Bar, and Plot elements are drawn. These elements are drawn after Fills have been drawn, because they may be superimposed over a filled region. For example, one might place white Text in a blue (filled) box. The rules for rendering Text, Message, Bar, and Plot elements are presented in individual sections later in this appendix. In Draw Mode and Class Mode, a Message's prototype text is drawn. In other modes, the current text of Message is drawn. In Draw Mode, Bars are drawn at there fullest extent. In other modes, Bars are drawn at their current sizes. In Draw Mode, Plot

data is suppressed; only a Plot's axes and labels are drawn.

7. The current image of the layout is copied to the bedrock layer.
8. Layout Objects are drawn. The rules by which Objects are drawn are presented later in this appendix.
9. The current image of the layout is copied to the layout shadow.
10. The layout is copied to the hidden screen image.
11. The hidden image is made visible, and the newly visible image is copied to the hidden image. A variety of special effect are available for making an image visible, e.g., fade, dissolve, etc.

The Screen Update Cycle

This section describes the order in which screen update operations are performed as an animation runs. The details of how each kind of animation component is drawn are specified in subsequent sections. As an animation proceeds, each screen update performs the following operations:

- All updates of Messages are applied.
- All updates of Bars are applied. For each updated Bar, Proof keeps track of the region of the screen that was affected.
- All updates of Plots are applied.
- All moving Objects are erased, keeping track of the approximate screen region affected by the erasure of each Object.
- All stationary non-Layout Objects that occupy a screen region affected by erasure of active Objects and/or Bars are also erased.
- All moving objects and those stationary Objects that were affected by erasures are redrawn, except for Objects that are scheduled for destruction, which implies permanent erasure. Object destruction requires a 2-cycle update.

How Ordinary Objects Are Drawn

By "ordinary" Objects, we mean Objects which are dynamically created by means of the "create" trace stream command. Layout Objects, discussed below, are created and positioned in Draw Mode. The rules for drawing Layout Objects are slightly different.

Ordinary Objects are drawn in strict accordance with color priorities. Each pixel of an ordinary

Object is compared with the current screen pixel it may overwrite. If the object's pixel has a higher color number than the current pixel, the Object's pixel overwrites the current pixel. Although this comparison may sound time-consuming, the use of MMX hardware instructions makes it easy to do up to eight such comparisons simultaneously.

Erasure of ordinary Objects also entails pixel-by-pixel color comparisons. Each pixel of an ordinary Object which is being erased is compared with the corresponding pixel in the current screen image. A comparison is necessary, because (1) some pixels of an Object may never have been written, because their colors' priorities were too low, and (2) some pixels may have been overwritten by other Objects with even higher priority colors. Each pixel which matches its corresponding current screen pixel is rewritten from the layout shadow layer. (The layout shadow contains "what's underneath the object.")

At any given point in time, every ordinary Object either active or idle. All moving Objects are active; however, in certain circumstances, stationary objects can also be active. For example, if the color of an Object is changed, the Object is made active for at least two update cycles, one for each of the screen images Proof maintains. Idle Objects are stationary objects that have not been changed for at least two update cycles. The distinction between active and idle Objects is important and was discussed in the Screen Update Cycle section above.

How Layout Objects Are Drawn

Layout Objects are created and positioned in Draw Mode. In the discussion of logical layers maintained by Proof, we stated that Layout Objects are written into the layout shadow layer. Therefore, if a layout Object is changed in any way, its image in the layout shadow layer must be updated. This is accomplished by restoring the layout object's pixels from the corresponding pixels of the bedrock layer. Erasures require color comparisons, since another Object may have overwritten a Layout Object with colors of higher priority.

Changing a Layout Object's color or changing the Class on which a Layout Object is based are treated as "simple" changes. For simple changes, the Layout Object continues to be treated as a Layout Object, with its up-to-date image stored in the layout shadow layer. All other changes, e.g., placing a Layout Object on a Path, require transformation of the Layout Object into an ordinary Object. In the latter case, Layout Objects are *permanently* erased from the layout shadow layer.

Both types of changes to Layout Objects require, at a minimum, a 2-cycle update. For simple changes, a Layout Object becomes active for exactly two cycles. A Layout Object that becomes active, either temporarily or permanently, is written and erased according to the rules described above for ordinary Objects.

How Text and Message Elements Are Drawn

Text and Message layout elements do not follow color priority rules. Text elements are written to the bedrock layer, the layout shadow, and the visible screen images when a layout is loaded. Message elements are written to the visible screen images and to the layout shadow layer. The first time a Message is written, it is written to the visible screen images (in a 2-cycle update) and to the layout shadow layer. After the first time, a Message is first erased by rewriting the old Message string in the BGcolor specified when the Message was created.

Proof does not maintain collision detection information for Messages. Other animation elements, e.g., Objects, Bars, Plots, and other Messages, are overwritten by Message updates. You should design your layout so that stationary Objects, Bars, and Plots do not come into contact with Messages and Messages do not come into contact with one another. Collisions between moving Objects and Messages are handled by Proof, but only because Proof updates all moving Objects in every screen update cycle, after any Message updates have been made.

How Bars Are Drawn

When a Bar is drawn, other than the very first time, the old extent is erased by rewriting the bar in the BGcolor that was specified when the Bar was defined in Draw Mode. (The very first time a bar is written, there's no old Bar to erase.) Bars are drawn and erased using a variant of color priority rules. When drawing Bar pixels, only those pixels are written for which the Bar's color number is (1) greater than the current color of the pixel or (2) equal to the Bar's BGcolor. When erasing Bar pixels, only those pixels are written for which the Bar's BGcolor number is (1) greater than the current color of the pixel or (2) equal to the Bar's color. In other words, priorities are observed, except that BGolor values can overwrite normal color values (erase operations) and normal color values can overwrite Bgcolor values (draw operations).

Proof keeps track of the regions of the screen affected by Bar updates, in order to detect collisions between Bars and Objects. Collisions between Bars and Objects (both moving and stationary Objects) are handled automatically by Proof. Collisions between Bars and Messages and Plots are not handled automatically.

How Plots Are Drawn

Plots do not follow color priority rules. Plot axes and labels are written to the bedrock layer, the layout shadow, and the visible screen images when a layout is loaded. Lines added to a Plot via the "plot" trace stream command are written to the visible screen images and to the layout shadow layer.

Lines in a Plot can be assigned an optional ID number. If a given ID number is reused, the previous line with that ID number is erased by restoring from the layout shadow the pixels it occupies.

Proof does not maintain collision detection information for Plots. Other animation elements, e.g., Objects, Bars, Messages, and other Plots, are overwritten by Plot updates. You should design your layout so that stationary Objects and Bars do not come into contact with Plots and Plots do not come into contact with other Plots. Collisions between moving Objects and Plots are handled automatically by Proof, but only because Proof updates all moving Objects in every screen update cycle, after any Plot updates have been made.

How Fills Are Drawn

The drawing of Fills is a notable exception to the manner in which Proof handles drawing operations. Virtually all other drawing operations are based on computed transformations of geometry into pixels. The only algebraic transformation made when a Fill is drawn is the mapping of a Fill's seed point (x, y) location into a pixel. The remaining processing is entirely pixel-based.

Fills are drawn after lines and arcs, because lines and arcs comprise the boundaries of Fills. Proof draws Fills without regard for color priorities. The first step in drawing a Fill is to sample the pixel located at the Fill's seed point. If the seed pixel's color is already equal to the Fill's color, or if the seed pixel is located off-screen, no further processing takes place. Otherwise, a search fans out in all directions from the seed point. Each pixel encountered whose color is equal to the original color of the seed pixel is changed to the Fill's color. If there is no closed boundary around a Fill, the Fill will leak, filling the entire screen.

It is possible to specify multiple seed points for the same Fill. In some circumstances, this may be necessary. For example, if your animation is superimposed over a map of the world, and the map has blue-filled oceans, you will need to specify many seed points for the ocean fill. If you used only a single seed point, if a pan or zoom viewing operation caused the seed point to be located off-screen, the oceans would not be filled; they would remain drawn in the animation's backdrop color.

When an animation is viewed from a greatly zoomed-out perspective, animation elements can become quite small. If the seed point for a fill is very near an arc or line comprising the border of the Fill, in a zoomed out view, the seed pixel may actually overlap a pixel of the line or arc. Therefore, seed points for Fills should be specified away from lines and arc comprising the boundaries of the filled region.

Conversion of Objects into Bit Maps

Objects are comprised of Proof Lines, Arcs, Text, Messages, and/or Fills. For each Object created in an animation, the elements comprising its Object Class are converted into a bit map representation. Proof moves Objects across the screen using very efficient bit map operations, rather than redrawing their geometry for every change of position.

Individual Objects of a Class can be rotated and scaled, and they can be drawn in a color different

from that (or those) of their Class. Therefore, for each Class, Proof maintains a list of bitmaps, sorted by scale, rotation, and color. Whenever a new bit map is required for an Object, the list is searched to see whether an appropriate bit map already exists. Thus, many Objects can share the same bit map. If an Object Class contains Messages, each Object created from the Class has its own private list of bitmaps, again sorted by scale, rotation, and color.

The rules by which Objects are converted to bit map form are nearly identical to the rules by which lines, arcs, text, messages, and fills are drawn in a layout. There are only two differences. First, the destination is a bit map rather than the screen. Second, the possibility of "double rounding" exists. When an Object is converted to a bit map, the positions of its components are rounded to the nearest pixel. When an Object is actually placed on the screen, its hot point ((0,0) coordinate) position is rounded to the nearest pixel. The sum of an Object's rounded position and the rounded position of an element within the Object's bit map can end up being "off" by one pixel. Unfortunately, there's no way around this problem, since it is infeasible from an efficiency standpoint to redraw an entire Object from its Class description every time the Object is repositioned.