A Complete and Efficient Implementation of Covered Windows

Brad A. Myers
University of Toronto

The Sapphire window manager supports many important features, including flexible window refreshing, full-functionality subwindows, and optimized raster-op, that are not supported in most comparable systems.

Sapphire, the Screen Allocation Package Providing Helpful Icons and Rectangular Environments, is a window manager[1] for the Accent operating system[2] running on the PERQ personal workstation. It has been used at Carnegie-Mellon University and by PERQ Systems Corp. and its customers. Like the window managers for many other personal workstations and intelligent terminals, Sapphire supports the covered window paradigm, in which windows are rectangular and can overlap arbitarily like pieces of paper on a desk (sometimes called the desktop metaphor).

In this paradigm, a window may be on top of another window, just as one piece of paper may be on top of another piece of paper. The window that is behind is covered by the window on top and the parts under the top window do not show through (see Figure 1).

There is a total ordering imposed on all the windows where windows higher in the order cover windows that are lower. This is often called 2½ dimensions, since the rectangles are thought to be layered in another dimension (z) pointing out of the screen. Windows that do not interact are still ordered. The top window is not covered by any windows. The window that is most covered is said to be on the bottom.

The advantages of the covered window paradigm are that (1) there can be several large windows that would not all fit on the screen if they were required to be side by side, (2) the user can make more efficient use of the screen, and (3) the metaphor is more familiar to users. The covered window paradigm is used by a large number of existing window managers.[1,3-11]

The disadvantages are that it is more complex and expensive for the software, it is more difficult to provide programs that automatically manage windows, and users may waste time rearranging windows.[12] Window managers that do *not* support covered windows only allow the windows to be side by side, often stacked in columns (these are called tiled window managers), and are growing in popularity.[13,14]

One of the major goals of Sapphire is to provide to users and application programs a wide range of functions so it would not unduly limit how application programs could use the screen or interact with users. The window size can be changed easily, and both characters and full bitmap graphic operations are supported. Sapphire supports full covered windows, and, unlike some older window managers, lets windows be updated while they are covered. Windows in Sapphire can also be partially or fully off the screen in any direction.

It can also divide a window into subwindows. An application might provide different areas for various types of displays while keeping them together in the same parent window so they can be manipulated as a unit. Subwindows also help the system and user organize the screen using context.[15] Subwindows can overlap and operate with full functionality. In fact,

0018-9162/86/0900-0057$01.00 © 1986 IEEE

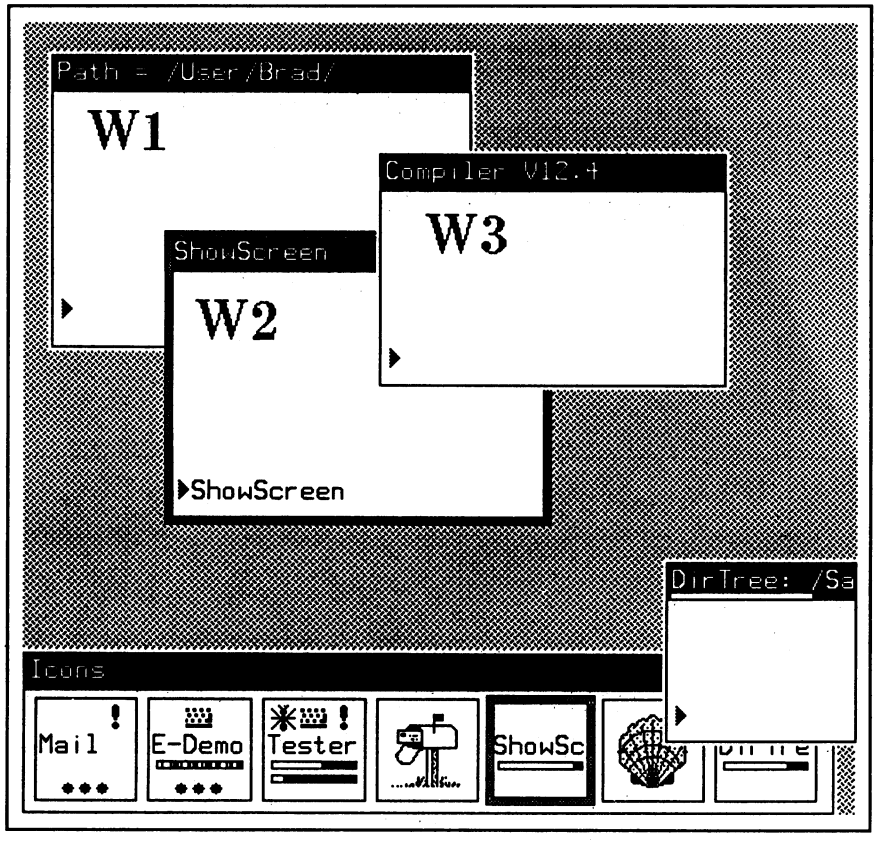


Figure 1. A portion of an actual Sapphire screen. Window *W3* is covering window *W2*, and both windows *W3* and *W2* are covering *W1*. *W1* is on the bottom and *W3* is on the top. *W2* has a gray border to show that it is accepting user input (typing). The window in the lower part of the screen contains a number of icons showing process and window state information.[1]

since top-level windows are actually subwindows of the full screen window, all windows are subwindows. Other systems such as SunWindows[7] limit the functionality of subwindows. The user interface to Sapphire, which includes a novel use of icons, windows with title lines, progress indicators,[16] and a simple but powerful user interface using the puck and keyboard, is described elsewhere.[1]

Like most other window managers, Sapphire is implemented entirely in software. A hardware implementation of the covered window paradigm exists,[4] but is probably too expensive to be practical in the near future.

Background

Sapphire runs on a raster (also called bitmap) display, which means that an area of memory holds the picture shown on the screen. Each point on the screen (pixel) corresponds to one bit in memory (so each pixel can be either white or black). For example, to set a spot on the screen to black, the corresponding bit in memory is set to 1.

The primary method for doing graphics on the PERQ's bitmap screen is the raster-op (sometimes called bitblt).[17] To avoid confusion this article uses "hardware raster-op" for the low-level function that actually moves bits, since it is implemented with special hardware on the PERQ. (The functionality of the hardware raster-op may actually be implemented by some combination of hardware, microcode, and software on other computers.) "Window raster-op" refers to the high-level function that works with covered windows.

Hardware raster-op moves an arbitrary bit rectangle from one part of memory to another. It also combines the source picture with the destination picture using a number of different Boolean functions, such as AND, OR, XOR, and NOT. The source and destination rectangles may overlap arbitrarily or may even coincide. Since the screen bitmap is stored in main memory on the PERQ, the hardware raster-op can be used with both the screen memory and off-screen memory, and can perform transfers between the two. The hardware raster-op on the PERQ operates at full memory speed no matter what the bit orientations of the source and destination rectangles are, so no special considerations need be made for the memory alignment used to hold bitmaps (as is done in Blit[5]).

Window raster-op can be used to implement many other graphics operations. For example, to display text, a window raster-op is performed for each letter to move it from a font table (containing a picture for each character) to the appropriate place on the screen. In addition, window raster-op is used to implement changing a window's size, position, and covering.

Handling window refresh

When a window is partially covered and then becomes uncovered, the parts that were previously hidden must be displayed. This can be the responsibility of the window manager (and thus hide the need to refresh from the application) or it can be the responsibility of the application (and thus free the window manager from having to remember the picture). The method for handling refresh is the main distinguishing difference among the implementations of covered window managers. If it is the responsibility of the window manager, the window manager can save either the picture *contained* in the windows or the picture that the window *covers* (that are under the window). Interlisp-D[11] saves the picture under every window, and Smalltalk[8] and others save the picture underneath for pop-up menus, but virtually all other systems—including Sapphire—save the contents of windows instead. Saving contents and areas underneath in the same window manager is very difficult. No matter how the window manager implements the saving internally, the procedure is hidden from programs using the windows.

If the window manager saves the picture under a window, the full bitmap must be saved. However, if the contents of a window are saved, there are several implementation possibilities, including saving

- the entire picture for each window in a separate off-screen buffer,

- only the covered portions offscreen,
- a general-purpose display list describing what was displayed, and
- combinations, such as bitmaps for pictures and characters for text.

The first approach keeps a full shadow bitmap for every window in a separate off-screen buffer. Graphics operations are done to this buffer, and any visible portions are copied to the screen. This approach is fairly simple to implement and very popular. It is used by SunWindows.[7] The main problem is that it takes a lot of extra memory to hold all the bitmaps, some of which is redundant, and it takes extra time for displaying because graphics in visible parts must be drawn twice (once to shadow memory and once to the screen).

The second approach tries to alleviate these problems by saving only in off-screen buffers the covered portions. The graphics operators must be changed so a single operation will work partially on the screen and partially in an off-screen buffer. This clearly makes the graphics operations more complex. There may be some operations provided by the hardware, such as filled polygons or circles, that cannot work separately on different parts of the picture. Saving the covered portions would be inappropriate in this case. Blit and Sapphire can use this approach since their graphics are limited mostly to rasterops.

The third approach calls for a general display list mechanism, such as that used on calligraphic (vector) screens, but this is inappropriate for a bitmap screen because saving a description of the picture may easily take more memory than the picture itself, especially for complex images. Refreshing from a display list also takes more time than refreshing from a stored picture, and it may be difficult to handle raster-selective erasures with display lists.

The fourth approach calls for a mixed style such as a limited form of display list when appropriate. Because many windows contain only text, an obvious optimization is for those windows to save only the characters displayed in them rather than the bitmaps for the characters. This will typically save more than an order of magnitude of storage even on a one-bit-deep screen (nine by 13 characters take 117 bits versus seven bits for the ASCII code). In systems where this has been implemented, however, such as the ICL PNX window manager,[4] many restrictions typically apply (for example, only one font per window, which must have a fixed width). Also, this might be of limited application since many text-only applications, such as sophisticated text editors, use graphics as part of their user interfaces (such as MacWrite for the Macintosh).[10]

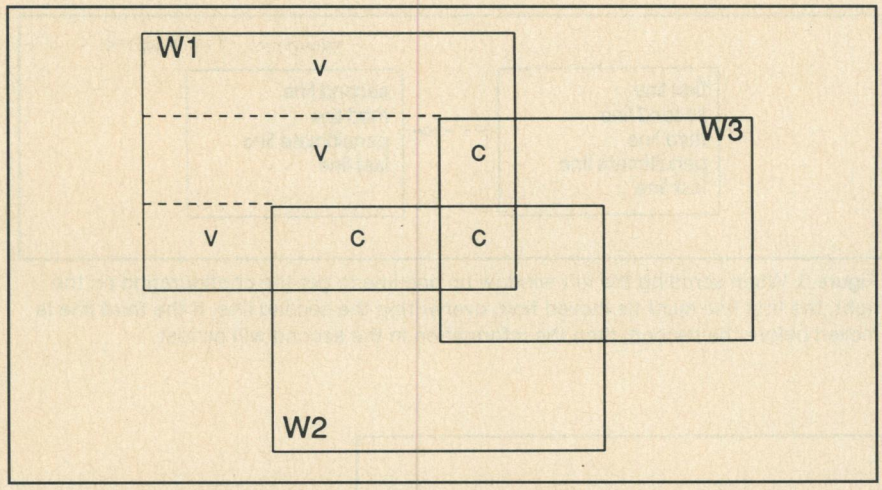


Figure 2. The rectangles that would be produced for window *W1* in Figure 1 based on the windows *W2* and *W3*. The rectangles marked *V* are visible, and those marked *C* are covered.

If an application handles window refresh, it must remember the contents of its window and regenerate the picture on demand. This is the approach in the Apple Macintosh[10] and IRIS Mex[6] window managers. Many programs, such as text and graphics editors, must save the contents of the window anyway, so it is easy for them to regenerate the screen picture when necessary. On the other hand, some applications may find it difficult to manage window refresh. The application program must be prepared to handle asynchronous refresh requests that may occur at any time, including while the program is modifying the picture. This may adversely affect the program's structure and the ease of porting programs written for nonwindow systems.

To provide maximum flexibility—and still conserve memory whenever possible—Sapphire implements two methods, allowing the programmer the choice of automatic refresh by saving only covered portions off screen or of application-handled refresh. Providing application-handled refresh was partially based on the observation that although Blit uses the memory-saving techniques of the covered-portions approach, some of its programs—such as the editor and debugger—use a different technique for covered subwindows to conserve memory.

Providing automatic refresh lets normal windows be used as the temporary buffers often needed in interactive graphics (for example, to hold a series of pictures being transferred to the screen one by one for an animation). The program simply creates a window that has automatic refresh (so it has backup memory), positions the window so it is entirely off screen, and then stores the picture in the window. Since the window is off screen, it will not be visible to users—but all the graphics operations will work normally, and the application can copy the contents of the window to the screen whenever desired. The window manager will also allocate and manage the temporary buffers automatically.

Raster-op for covered windows

Most computers provide little hardware support for the covered window paradigm. Therefore, to display only the portions of a window visible on the screen (and hide covered parts), window managers implementing the covered window paradigm in software must calculate which portions of windows are covered. This is especially true if the window manager allows graphics to appear in a window while the window is covered. (The Interlisp-D[11] and Smalltalk[8] window managers only allow graphics to be output to uncovered windows. Interlisp-D automatically brings a window to the top before doing output graphics, which causes a great deal of flashing when multiple windows are being used. Most modern window managers support updates to any window at any time.)

The window intersection information is usually precalculated and stored as a list of

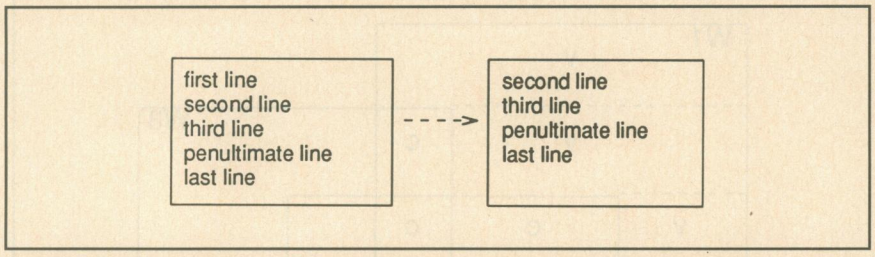


Figure 3. When scrolling the left window up one line to get the configuration on the right, the first line must be moved first, overwriting the second line. If the third line is moved before the second, then the information in the second will be lost.

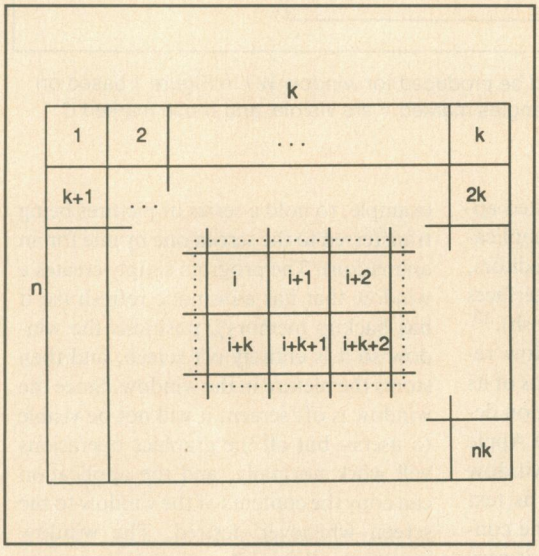


Figure 4. Typical bitmap organization. The first pixel is in the upper left and there are k pixels across and n down. The dotted rectangle in the center is to be shifted either starting with pixel i or pixel $i+k+2$. When moving the rectangle towards the upper left, for example, the pixel order will be $i, i+1, i+2, i+k, i+k+1, i+k+2$.

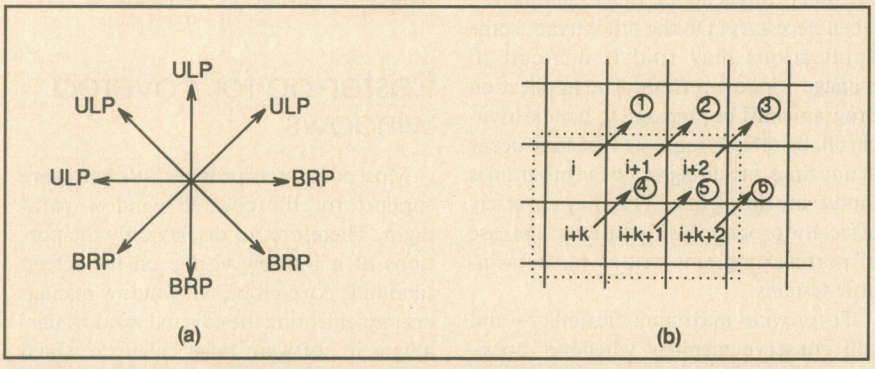


Figure 5. (a) To move the rectangle in Figure 4 in any direction requires only two orders for the pixels, starting from the upper left pixel (ULP) or from the bottom right pixel (BRP). **(b)** Starting from the upper left pixel *(i)* is correct when shifting a rectangle to the upper right, while starting from the bottom right pixel $(i+k+2)$ would be incorrect since pixels $i+1$ and $i+2$ would be overwritten before they are moved.

rectangles for each window, marked as visible (which means they appear on the screen) or covered (which means they are covered by other windows and not visible on the screen). Figure 2 shows the set of rectangles that would be generated for window *W1* in Figure 1.

When using window raster-op to transfer a picture to a new position that overlaps the old one, the order in which the bits are moved is important. It is important to note that the ordering problems discussed in this section are less relevant for window managers that use full window shadow bitmaps. For example, when scrolling up a window, the top must be moved before the bottom or else the bottom part will cover portions not yet transferred (see Figure 3). Window raster-ops of this form are used when scrolling text up or down in an editor or when panning a picture around in a graphics program.

This problem is analogous to shifting the elements in a conventional array, where the shift must be done from the correct end to avoid losing information. The hardware raster-op, like the conventional array shift, needs only two directions to work correctly: top to bottom and bottom to top (see Figure 4). When moving a rectangle up or to the left, the hardware raster-op will first move the upper-leftmost bit, then the bit to its right, and so on as shown in Figure 5. When moving down or to the right, raster-op must start with the last bit.

The covered-window raster-op applies the hardware raster-op to each rectangle (as in Figure 6) as a unit. It therefore must deal with the processing order of the rectangles. In this case, four different orders are needed.

Imagine a picture covering the three rectangles in Figure 6. When moving the picture to the upper right, the order for the rectangles must be *B, C, A*—or else some necessary portion will be overwritten. When going to the lower left, the order is *A, C, B,* which is the reverse of the order when going to the upper right. When going to the upper left, however, the order is *A, B, C*—which does not have any natural

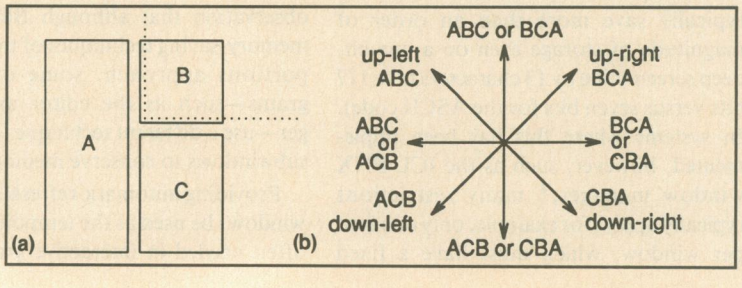


Figure 6. (a) If the rectangles need to be moved to a new position overlapping the old position, the moving order is important. **(b)** The correct order for the eight directions. The orders for up, down, left, and right can be either of the orders on the neighboring diagonals. Rectangle *B* has no other rectangles in its upper right quadrant, so it can be moved to the upper right without interfering with other rectangles.

correspondence to the upper right/lower left order. The lower right order is the reverse of the upper left order: *C, B, A*. The order is always important, no matter whether the rectangles are stored in contiguous memory (all on the screen) or if they are each stored in separate buffers.

The reason that two orders do not suffice for covered windows is that the objects to be moved (rectangles in this case) are larger than the distance they can be moved, so the new positions partially overlap parts of the old positions. Simply making all the rectangles be squares of the same size would not solve the problem, as Figure 7 shows.

In a more formal analysis, when going to the upper right, the rectangle whose upper right quadrant does not overlap with any other rectangles must first be found (see Figure 6). This rectangle can be moved in any direction in the quadrant without affecting any other rectangles. Since there are a finite number of nonoverlapping rectangles, there will always be such a rectangle. After this first rectangle is moved, it is eliminated from consideration and the next maximal rectangle is found. A similar operation is performed to move in the other four directions.

Some window managers (such as Blit) sort the rectangles when the window raster-op is performed. To avoid this overhead, Sapphire uses a technique that generates the rectangles in the correct order. Because graphic operations are done much more frequently than reconfiguring the rectangles (which is done only when windows are created, deleted, or modified), it is more efficient to generate the rectangles in sorted order, as Sapphire does.

The rectangles in Sapphire are stored in a quadruply linked list, one thread through the rectangles for each of the four orders. The window raster-op then follows the correct thread based on the direction of the window raster-op transfer. Each rectangle of the source is compared with each rectangle of the destination in the correct order, and any overlapping parts are transferred. It does not matter whether the source and destination windows are the same or different. Figure 8 gives an outline of the code for covered window raster-op.

The window raster-op clearly takes $O(n^2)$ time, where n is the number of rectangles. The maximum number of rectangles in a window is a constant multiple of the number of windows, so n can be considered to be the number of windows or the number of rectangles in any one window.

The $O(n^2)$ complexity for window raster-op cannot be improved in the worst case, since the number of overlapping rectangles can be $O(n^2)$ because every rectangle in the source area may intersect with every rectangle in the destination area. This will happen, for example, when the source rectangles are all horizontal and the destination's are all vertical, as shown by areas *A* and *B* in Figure 9. It is therefore not possible to write an algorithm that has less than quadratic complexity in the worst case.

The general case can be made faster, however, by trying to limit the number of rectangles compared on average. For ex-

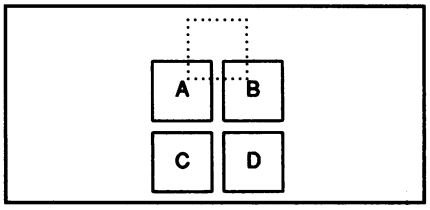


Figure 7. Even if all the rectangles are squares of the same size, four orders are needed if the new positions can overlap the old positions. *B* must be moved before *A* can move into the dotted position, and *A* must be moved before *B* can move there.

```
Procedure WindowRasterOp(RasterFunction, SourceCoords, Destination-
Coords)
  Calculate RasterOp direction from source and destination coordinates.
  FOR dr: = EACH destination rectangle is in order DO
    IF dr is not covered OR
      IF (dr is covered AND has backup memory) THEN
        dtmp: = Clip destination coords to dr
        IF anything left inside dtmp THEN
          FOR sr: = EACH source rectangle in order DO
            stmp1: = compute corresponding place in source for dtmp
            stmp2: = clip stmp1 to bounds of sr
            IF anything is left inside stmp2 THEN
              IF sr is covered THEN {no picture in source for this area}
                save stmp2 on a list for update by the application
              ELSE
                do hardware RasterOp on stmp2
          get next source rectangle in the current order.
    get next destination rectangle in the current order.
  END Procedure WindowRasterOp
```

Figure 8. Outline of the code for covered window raster-op in pseudo-Pascal.

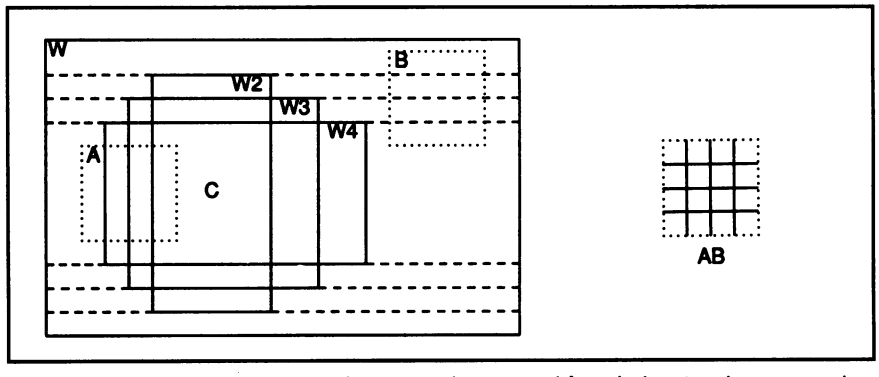


Figure 9. The dashed lines show the rectangles created for window *W* when covered by windows *W2*, *W3*, and *W4*. When transferring area *A* to area *B*, there are $O(n^2)$ rectangles to move, where n is the number of rectangles. Area *C* is covered by $O(n)$ rectangles, one for each window.

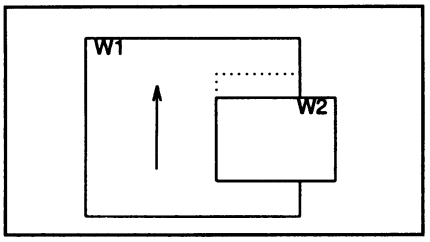


Figure 10. When window *W1* is scrolled up, the picture appearing from behind window *W2* needs to be displayed. If there is no backup memory for *W1*, the application must regenerate the picture for the dotted rectangle.

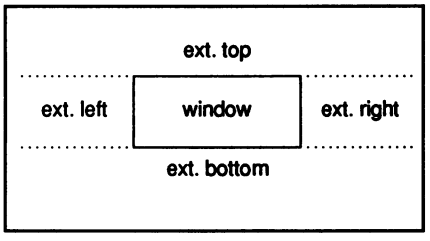


Figure 11. Special exterior rectangles surrounding a window. They extend to ± ∞ (± max_integer).

ample, the entire destination and source areas for the window raster-op can be compared to their respective rectangle lists to see which rectangles could be affected by the particular window raster-op (this is akin to the bounding box test often used to optimize other graphics operations). The comparison takes $O(n)$ time and generally will probably save a lot of time, since many window raster-ops affect only one source and one destination rectangle.

Many other optimizations are possible. For example, window raster-ops in uncovered windows (which have only one rectangle) can be performed directly with only a clip to the windows' borders.

The general hardware raster-op is often used to erase or invert a single rectangle. In this case, the source and destination rectangles are identical. For example, a rectangle can be set to white (0) by XORing it with itself. When the rectangles are the same, a much more efficient algorithm is used in Sapphire that goes through the list of rectangles only once and therefore has $O(n)$ complexity.

When performing a covered window raster-op, there are two cases where there may not be a picture available in the source. First, the source of the window raster-op might be a window that is covered and does not have backup memory (and so uses application-handled refresh). An example of this is scrolling a text window where some text comes out from behind another window (see Figure 10). The other case occurs when the specified source is outside a window. Some systems may flag this latter window raster-op as an error, but Sapphire allows it for consistency because the destination for graphics can extend outside the window. Sapphire surrounds every window with four special rectangles (see Figure 11) for the exterior of the window that extends to ± ∞ (implemented as ± max_integer) away from the window. This makes it possible to avoid a special case test and extra boundary clipping in the window raster-op code.

Whenever parts of the source are not available, the rectangles in the destination

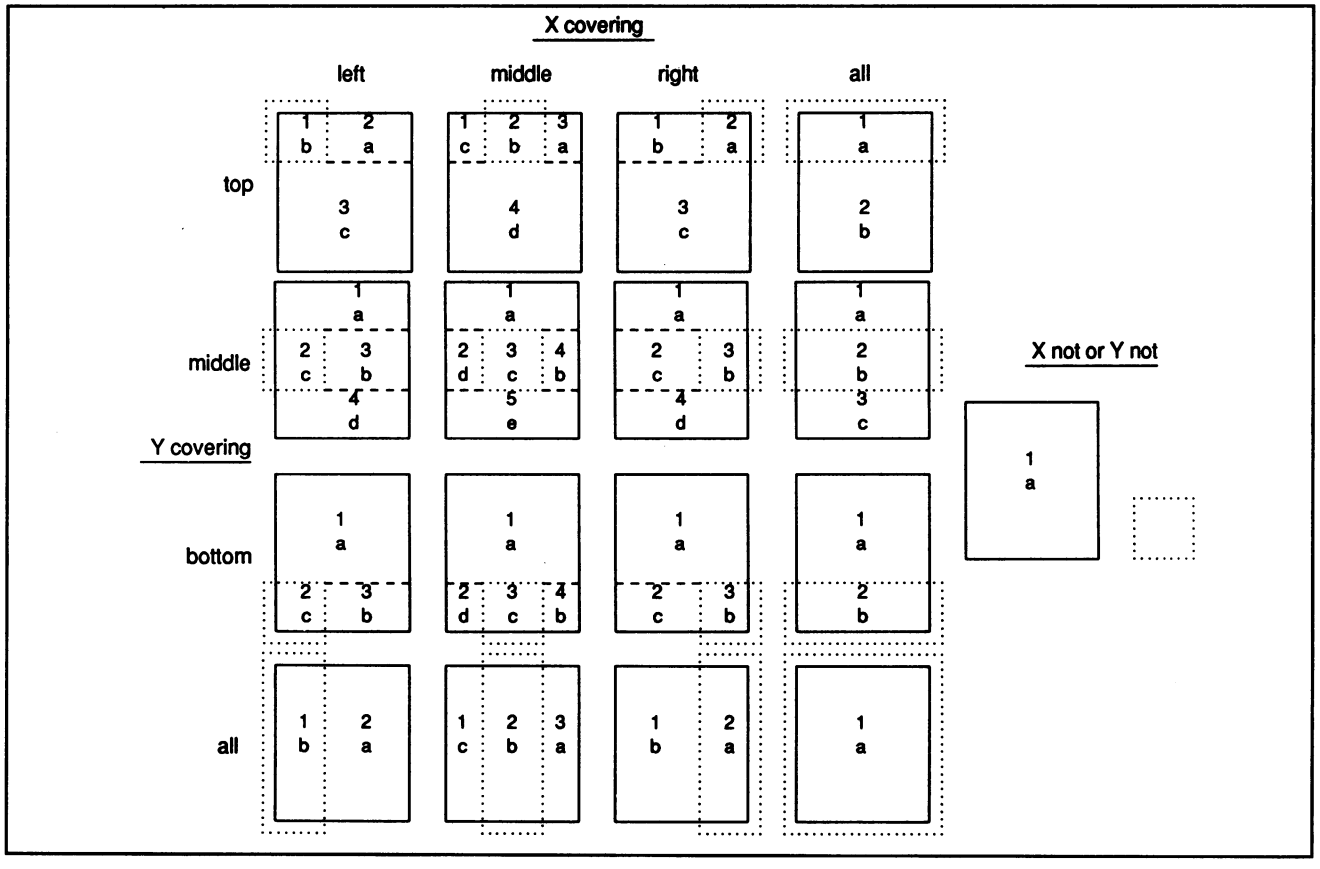


Figure 12. The 17 two rectangles can interact. The numbers show the up left ordering, and the letters show the up right ordering. The down right ordering is the reverse of the up left, and the down left is the reverse of the up right. The solid rectangle is to be divided based on the dotted window. If the dotted window covers the solid one, the rectangle inside both the dotted and solid rectangles is covered. If the dotted window is the parent of the solid one, the area of the solid rectangle outside of the dotted area is covered.

for those parts are saved on a list and later passed to the appropriate application program for regeneration using an operating system exception mechanism. The application is told exactly what portions need to be regenerated so it does not waste time drawing intact parts (although it can if it wants to).

Rectangle intersection algorithm

The coordinate system for each window in Sapphire has 0,0 at the upper left corner; x grows to the right and y grows down. The bounds of the window are therefore given by the lower right corner of the window and are inclusive. For graphics, however, coordinates can be used that are outside of the window (negative to the upper left or greater than the bounds for the lower right) and the picture is simply clipped to the inside of the window.

Sapphire's rectangle intersection algorithm is fairly simple. It is based on the straightforward enumeration of all the ways two rectangles can intersect. In each x and y direction, the rectangles can be covered in one of five ways. For x, they are left covered, middle covered, right covered, all covered, and not covered. The y values are similar. It turns out that if the rectangles do not intersect in either the x or the y direction, the rectangles do not overlap. There are thus 17 different possibilities ($4 \times 4 + 1$). These are shown in Figure 12.

The rectangles are optimized for maximal horizontal extent. This direction was chosen to minimize the number of rectangles crossed in typical text operations (which are usually horizontal). The only difficulty in implementing the subdivision is to avoid fence-post errors. (Fence-post errors are ± 1 errors. The name comes from the problem of determining how many posts are needed to fence a yard that is 10 feet long if one is placed every foot.) Each case splits the rectangle into one to five new rectangles, some covered and some visible. These rectangles are added to the four different rectangle lists in the correct order.

To demonstrate that this preserves the ordering, it must be proved that dividing one rectangle into a set of rectangles that replace the old rectangle in the list preserves the overall order. Imagine that rectangle B in Figure 6 is to be divided. Since all the new rectangles will be entirely enclosed in the area of B, they all still must follow A and precede C when going to the upper left. Therefore, if the new rectangles that replace B are in the correct order with respect to each other, they will be in the correct place with respect to A and C, and therefore with respect to the entire list.

The procedure that implements the subdividing algorithm traverses each list from back to front and adds each new rectangle after the current rectangle, so newly added rectangles will not be investigated during the pass of the algorithm that creates them. Figure 13 gives an outline of the pro-

```
Procedure WindowIntersect(W,RL, b: (wantOutside, wantInside) )
    {intersect window W with the current rectangle list RL. b is wantOutside
    when W covers RL, b is wantInside when W is parent of RL}
    {Rectangles in RL are screen coordinates}
    {RL is changed to have the new rectangle list}
    convert W's coordinates to screen coordinates {so it can be compared to
    RL}
    FOR current := EACH rectangle in RL starting with firstDownRight DO
      IF current is covered THEN {ignore since don't need to subdivide
      covered rectangles}
      ELSE
        Calculate intersection between current and W in X and Y directions
        (as in Figure 12)
        IF either X OR Y not intersected THEN
          Change Current rectangle coveredness to be NOT wantOutside
          {doesn't intersect}
        ELSE
          CASE x interaction OF
            xleftCov: CASE y interaction OF
                  ytopCov: {first create rectangles and add to the upLeft,
                  downRight lists}
                      t2 := AddUpLeft(current, 2a, b = wantInside)
                        {t2 becomes upLeft of current. Last argument to
                        AddUpLeft is coveredness of the new rectangle}
                      t3 := AddUpLeft(t2, 3c, b = wantInside)
                        {next, change size and position of t1 to be 1b; last
                        arg is coveredness}
                      t1 := Change(current, 1b, b = wantOutside)
                        {now add new rectangles to the upRight,
                        downLeft lists}
                      AddDownLeft(t1, t2) {t2 becomes downLeft of t1}
                      AddUpRight(t1, t3) {t3 becomes upRight of t1}
                  ymiddleCov:
                      t2 := AddUpLeft(current, 2c, b = wantOutside)
                      t3 := AddUpLeft(t2, 3b, b = wantInside)
                      t4 = AddUpLeft(t3, 4d, b = wantInside)
                      t1 := Change(current, 1a, b = wantInside)
                      AddDownLeft(t1, t3) {t3 becomes downLeft of t1}
                      AddDownLeft(t3, t2)
                      AddDownLeft(t2,t4)
                  ybottomCov:...
                  yallCov:...
                  END CASE on y
            xrightCov: CASE y interaction OF
            ...
            END CASE on x
        current := next downRight from current
    END Procedure WindowIntersect
```

Figure 13. Outline of the code to do rectangle subdivision. There are four cases for x in the outer case statement. Each branch of the case has four cases for y, making 16 cases. The 17th case is the special test for not intersecting performed before the cases. In each branch, the numbers (like 2a, 3c) correspond to the rectangle label in Figure 11. All 16 case branches are similar to the ones shown.

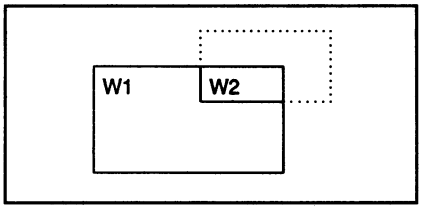


Figure 14. *W2* is a subwindow of *W1*, so it is clipped to the boundary of *W1*. The part of *W2* that is outside of *W1* is marked covered (and are not visible). The part of *W1* that is underneath *W2* is covered.

```
Procedure OuterLoopIntersectWindows(W)
    {Calculates the rectangle list for window W.
    The list is stored in the local variable RL.}
  RL := rectangle for W converted to screen coordinates {start rectangle list
    with one rectangle for the entire inside of W}
    IF W is offscreen THEN
    Set RL to be covered {optimization}
  ELSE
    {* first, clip to the inside of parent and its parent, etc. *}
    t := W's parent window
    WHILE t <> NIL DO
      Call WindowIntersect(t, RL, wantInside) {get the inside of t}
      t := t's parent window {go up the window hierarchy}
    {* next, remove areas covered by my immediate children *}
    FOR t := EACH immediate child window of W DO
      Call WindowIntersect(t, RL, wantOutside) {get the outside of t}
      t := next child of W
    {* finally, check all other windows that might cover W *}
    t := W
    WHILE t's parent <> NIL DO
      FOR t2 := EACH sibling window of t that is higher priority (more
          towards the top) than t DO
        Call WindowIntersect(t2, RL, wantOutside) {outside}
        t2 := next sibling
      t := t's parent window {go up the window hierarchy}
  {clean up}
  convert RL to be in W's coordinate system
  add the exterior rectangles
  set W's rectangle list to be RL
END Procedure Outer LoopIntersectWindows
```

Figure 15. Outline of the code for the main loop for calculating window's coveredness. The procedure WindowIntersect is outlined in Figure 13.

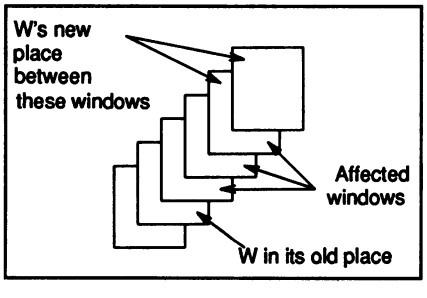


Figure 16. Bringing window *W* more forward (towards the top) can only affect certain windows.

cedure. The algorithm has quadratic complexity, but this is inherent in the problem since, in the worst case, the number of rectangles that can be created in one window intersecting with n other windows is n^2, as shown in Figure 9. Since this algorithm will be run on all n windows, the total complexity is $O(n^3)$.

This is also inherent in the problem. Since each covered portion of the screen will be represented by a rectangle for each window on it (the area marked *C* in Figure 8 is represented by n rectangles, one for each window covering that area), the total number of rectangles for all windows is $O(n^3)$. There are some heuristics available, however, to improve the general case. These include trying to limit the number of windows processed (for example, by ignoring windows that are totally off screen) and trying to run the algorithm as few times as possible.

Sapphire implements subwindows. Subwindows can overlap within their parent and may extend outside the parent window, but any parts outside are clipped and not visible, as Figure 14 shows. Because the screen is represented as the parent of all other windows, it is trivial to let windows extend partially or totally off screen. This also makes it easy to change the screen size (the PERQ can be configured with various screen sizes), and some of the memory normally used for the screen can be allocated to other applications by reducing the screen window size. The extension to the subdivision algorithm to handle subwindows is very simple (see Figure 13). When clipping a subwindow to its parent, the identical algorithm is used—except areas inside the parent are visible and areas outside are covered, which is the reverse of the case for overlapping windows presented above. Subwindows can recursively contain their own subwindows to an arbitrary depth.

Because subwindows cover the parent window, the immediate subwindows of a window affect that window's rectangles (see Figure 14). Subwindows of sibling windows do not have to be investigated, however, because they are entirely enclosed within the sibling window. An outline for the outer loop for window subdivision is shown in Figure 15. All the calculations are done in screen coordinates to make it easier to compare different windows.

As the final step of the algorithm, the rectangles are converted into the window's coordinate system. As part of this step, the rectangles are associated with the memory that will hold their picture: some rectangles correspond to windows on the screen and others correspond to off-screen memory holding covered portions of the picture. If the window does not have off-screen memory (so the application must handle refresh), the rectangles for covered parts are marked as having no memory to

hold the associated picture. Output to these portions will simply disappear (as shown in Figure 8). Also in this step, the four special rectangles for the exterior of the window (Figure 11) are added to the four rectangle lists in the correct order.

Manipulation operations

Top, bottom, create, and delete. Making a window less covered (towards the top) requires two steps. First, any portions of the other windows that become covered must be stored in their backup memory. Then portions of the window brought forward that become uncovered must be regenerated. Most systems only allow windows to be brought to the top (so they are not covered by any windows) or sent to the bottom. With window raster-op, Sapphire lets windows be moved to any position in the covering order. In the following description, the window being changed is called *W*.

First, Sapphire checks those windows closer to the front (less covered) than *W*'s old place and less covered than *W*'s new place (see Figure 16). Only these windows can be affected by the change. For each of these windows, Sapphire checks to see if its entire area intersects with the entire area for *W*. If so, all its subwindows are checked, since some may also be affected. For each affected window, the old rectangle list is saved and the new list is generated for the window with the new covering.

To do the actual update, Sapphire simply calls window raster-op to transfer from the old rectangle list to the new one. The standard window raster-op call (which takes two windows as parameters) is used by having a dummy window to which the old rectangle list is assigned. If the affected window had backup memory, the window raster-op will automatically move any newly covered portions to the backup memory.

After each affected window is updated, a similar operation is performed for *W* itself. Here, however, the window raster-op will copy from backup memory any portions of *W* that become *un*covered. If *W* did not have backup memory, the window raster-op will inform the appropriate application program to regenerate the correct portions of *W*. Creating a new window is similar to moving it from the bottom (the old rectangle list is empty).

Making a window more covered uses the same technique—but in reverse. In this case, the windows more covered than *W*'s old position and less covered than *W*'s new position are checked to see if any portions must be regenerated. The actual regeneration is done in the same manner as for top, except that *W* itself is handled before the other affected windows, since any portions that become covered in *W* must be saved to backup memory before the screen is overwritten by the newly uncovered windows.

Deleting a window works the same as sending it to the bottom.

Move and grow. Move and grow are much more complicated than top and bottom since as much of the original picture as possible should be retained without using extra temporary buffers, and subwindows should move with the parent window (so the subwindows remain in the same relative places within the parent window). Another requirement is that windows be movable even while being covered by other windows in the source or destination. Sapphire therefore transfers as much of the old picture as possible to the new location and only requires the application to refresh what is absolutely necessary. Because the screen contains the only copy of the picture for the visible parts of the window, no screen picture should be destroyed before it is moved.

Having subwindows also complicates the modify operation. Clearly, if the window is made bigger, the application will need to adjust the picture (and possibly the subwindows) to fill in the new areas, so it must notified. In other cases, however, the application can remain uninvolved. Move and grow are implemented by one procedure, called modify.

Again we call the window being modified "W." The modify procedure is a three-step process (see Figure 17). First, all areas covered by *W*'s new position must be copied into their backup memory (if any). The algorithm for implementing this is similar to that used when *W* is brought to the top. Second, the picture for *W* must be moved to its new position. Third, any windows that become exposed must be regenerated. The algorithm for the last step is similar to that used to send *W* to the bottom. The calculations for the first step must take into account *W*'s old position as well as its new position, so portions of *W*'s screen picture are not overwritten. Therefore, some windows may be updated twice if they are affected by both *W*'s new and old positions.

Step 2, where *W* itself is modified, turns

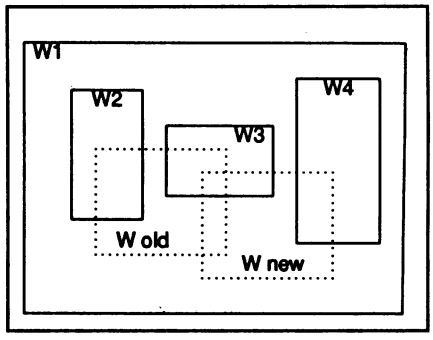


Figure 17. Modifying a window is a three step process. First, windows affected by *W*'s new place (*W1*, *W3*, and *W4*) have their newly covered portions stored to backup memory. Second, *W*'s picture is moved to the new place. Third, windows that need to be refreshed because portions are no longer covered by *W* (*W1*, *W2*, and *W3*) are regenerated. Note that *W1* and *W3* are affected twice.

out to be surprisingly difficult. *W* may be covered in different ways at its source and destination, and its new position may overlap its own old position (see Figure 17). A further problem is that the screen picture for the entire window must be moved as a unit—including the pictures in any subwindows. Neither the window nor its subwindows can be moved independently since they may overlap at the source and destination (see Figure 18). Therefore, to move the window itself, the following three steps are performed:

First, any portions of the window and all of its subwindows that will be covered in the destination position are saved into backup memory with a special version of the covered window raster-op that only transfers the covered parts of the destination and ignores the uncovered parts.

Second, the screen picture from the old location is transferred to the new location. To do this, Sapphire must calculate the covered rectangle sets for the old and new locations, ignoring all of *W*'s subwindows. The new rectangle lists are needed because *W* may be covered by other windows (not subwindows of *W*) at both the source and destination, as shown in Figure 18. Any backup memory that *W* might have is ignored here.

Third, any parts of the picture for *W* or its subwindows that become exposed are transferred to the screen from backup memory or redrawn by the application program (if there is no backup memory).

When changing a window's size, it is often inappropriate merely to adjust the subwindows' size proportionally to the parent's, since some subwindows may

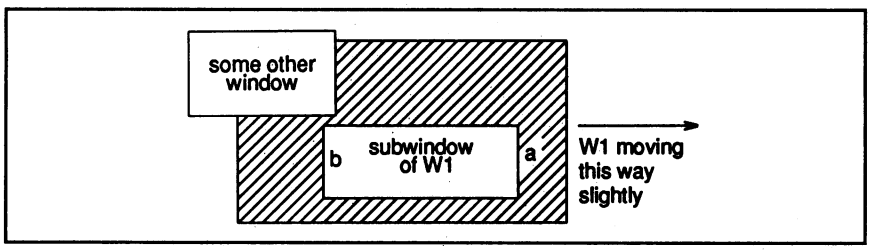


Figure 18. When *W1* is moved, its subwindow *W2* moves with it so that it remains in the same relative place inside *W1*. The screen picture for *W1* and *W2* cannot be moved separately to the new position. Imagine that *W1* is to be moved slightly to the right. If the subwindow is moved first inside *W1*, the part of *W1* marked *a* will be erased. If the uncovered part of *W1* (gray area) is moved first, the portion of *W2* marked *b* will be erased. Therefore, the entire picture *(W1* including *W2)* must be moved as a unit, but window *W3* must still not be affected.

have special size constraints. For example, a header subwindow may be exactly one text line high, irrespective of the window size. Therefore, Sapphire leaves all subwindows the same size and in the same relative position with respect to the origin of the parent window, and notifies the appropriate application program of the window's size change so it can reconfigure the subwindows (if necessary).

Although window modification is fairly complex, it does maintain the most information possible and provides a great deal of flexibility. When a window has no subwindows, many of the above steps are omitted. Because windows are moved rarely and only at a user's command, the modify procedure has proved acceptable.

Efficiency comparison

The covered window paradigm has been around for a fairly long time. It was developed at the Xerox Palo Alto (Calif.) Research Center in the Smalltalk[8] and Interlisp[11] environments. These and other early implementations of covered windows typically only allowed applications to update a window if it was not covered. The Blit window manager,[5] which does provide output to covered windows, was probably the first covered window system to have its implementation openly published. Sapphire implements a superset of Blit's features.

For example, the Blit implementation only provides automatic refresh with saved bitmaps, while Sapphire allows application-handled or automatic refresh. Also, Blit does not support subwindows or changing a window's size. (The change-size operation on Blit is implemented by deleting the window and recreating it, and therefore losing the contents of the window.) There are many other window managers supporting covered windows today,[1,3-11] but their algorithms are usually proprietary, so efficiency comparisons are difficult.

Sapphire was influenced by many of these systems; but the algorithms described here are original.

For most operations, Sapphire and Blit have the same complexity. For example, window raster-op in both is $O(n^2)$ (where n is the number of windows). In the rectangle intersection algorithm, Sapphire's complexity is the same as for Blit ($O(n^3)$ for all rectangles for all windows). Sapphire may create fewer rectangles because covered rectangles are not subdivided as they are in Blit, but this saving may be overridden in practice by the four exterior rectangles. The WHIM window manager[3] essentially uses the Blit algorithm, but does not subdivide covered rectangles, so it will have fewer rectangles.

When doing a window raster-op, Blit must sort the rectangles. In Sapphire, the rectangles are already sorted. The disadvantage of Sapphire's technique, however, is that the full intersection algorithm must be run whenever windows are manipulated, while Blit must only transfer some rectangles from one window to another. (Of course, this is done only when windows are moved, created, deleted, grown, or reduced.) Sapphire has been optimized, however, for the case where a window simply becomes more covered (sent to the bottom). Here, the existing rectangle set is simply intersected with the new window. This is much faster than recreating the rectangle list from scratch and is about as efficient as the Blit technique, while preserving the rectangle order.

The major time-critical part of Sapphire is the window raster-op itself. Some measurements show that, in many operations, the covered window raster-op overhead over the hardware raster-op is quite substantial. This is partially due to the large number of rectangles for typical windows. The average number of rectangles for covered windows has been measured as eight (which includes the four exterior rectangles) while some windows, such as the full-screen window, often have more than 60 rectangles.

The Sapphire optimizations have increased the window raster-op efficiency by a factor of two on average and a factor of 10 in certain special cases. Window raster-ops in uncovered windows bypass the expensive algorithm altogether and therefore perform almost at the hardware raster-op's speed. Of course, the window raster-op overhead could be vastly reduced by coding it in microcode. If the covered window raster-op is made faster, it will not only help applications but also Sapphire itself, since window raster-op performs all window manipulations.

Another area of efficiency is memory usage. In the Accent operating system supporting Sapphire, it is very expensive to allocate memory for pictures. Therefore, whenever a window is created with backup memory, enough memory is allocated for the entire window, even though only parts of the window may be covered. This will clearly waste a lot of memory. During updates, the appropriate parts of the memory buffer are addressed for the covered portions of the window. (The rest of the buffer is left unused.)

The window intersection algorithm allows memory allocation and deallocation that can be added easily if the operating system makes this feasible. When a window's covering changes, however, two pieces of memory would be allocated during the update (as in WHIM[3]). The old memory would be released after the update was completed. On Blit, only as much memory as needed is allocated, and an XOR swap transfers pictures from one buffer to another. Sapphire could use this technique if the temporary extra memory usage was a problem, but this takes three hardware raster-ops instead of the two now used.

The theoretical efficiency of a window manager is not nearly as interesting as the measured performance of actual procedures. Unfortunately, many of the optimizations applied to Sapphire have proved ineffective because a much larger proportion of the time is actually spent in operating system interprocess communication. The Accent[2] operating system uses message passing with separate address spaces for separate processes, and Sap-

phire is implemented as a separate process from all applications for protection and structured design.

Unfortunately, the time to send a message to Sapphire, along with the required process swapping, swamps the time to perform the actual graphics. Therefore, the most worthwhile optimization is allowing applications to do graphics directly to uncovered windows and thereby eliminate the message to Sapphire. To provide the necessary protection and synchronization, however, this optimization has required that more of the window manager be implemented in the operating system kernel. Unfortunately, the current design does not allow applications to do raster-ops to covered windows without a message to Sapphire, since Sapphire stores the rectangle lists in its private address space.

The algorithms performing covered window operations in Sapphire have many advantages over other algorithms. These include application-handled or automatic refresh, moves, and size changes of windows, support for subwindows, and generally more efficient window raster-ops because the rectangle lists are always kept sorted. Some of this flexibility does not appear to be required in practice. For example, windows virtually never change coveredness except to the top or bottom, and it is rare to change a window's size or position while it is covered by other windows. Also, applications rarely create subwindows that overlap.

More investigation needs to be done on what techniques and facilities are important in practice and on efficient ways of implementing them. Also, as graphics hardware supports more sophisticated operations, such as color and 3D transformations, as in the IRIS,[6] methods for windowing these must be investigated. It will also be useful to gather statistics on typical window manager use and efficiencies to evaluate the trade-offs between simple shadow bitmaps, complex clipping algorithms as described here, and application-handled refresh.

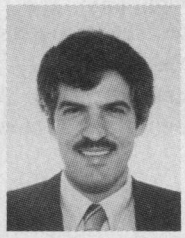
The Sapphire implementation demonstrates that full-functionality covered windows can be provided while saving only the covered portions off screen. Whether these algorithms will be appropriate depends on the particular circumstances, but the general principles and complexity results will continue to be important. □

Acknowledgments

Help with the algorithms came from Stoney Ballard, John Strait, and Dave Golub of PERQ Systems Corp. Amy Butler and Dave Golub have been largely responsible for maintaining Sapphire. Thanks also to Alain Fournier of the University of Toronto for help in evaluating the algorithms' complexity and to Rob Pike of AT&T Bell Laboratory for checking the Blit information. For help and support with this article, I thank my wife, Bernita Myers, and William Buxton, Brian Rosen, Joyce Swaney, Ron Baecker, Eugene Fiume, and many others at the University of Toronto and PERQ Systems Corp.

References

1. B. A. Myers, "The User Interface for Sapphire," *Computer Graphics and Applications,* Vol. 4, No. 12, Dec. 1984, pp. 13-23.
2. R. Rashid and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proc. Eighth Symp. Operating Syst. Princ.,* Asilomar, Calif., Dec. 1981, pp. 64-75.
3. M. J. Goodfellow, "WHIM, The Window Handler and Input Manager," *Proc. First Int'l Conf. Computer Workstations,* San Jose, Calif., Nov. 1985, pp. 12-21.
4. *ICL PERQ Guide to PNX,* International Computers Ltd., Reading, England RG3 1NR, UK.
5. R. Pike, "Graphics in Overlapping Bitmap Layers," *ACM Trans. Graphics,* Vol. 2, No. 2, April 1983, pp. 135-160.
6. R. Rhodes, P. Haeberli, and K. Hickman, "Mex – A Window Manager for the IRIS," *Usenix Summer Conf. Proc.,* Portland, Ore., June 1985, pp. 381-392.
7. *SunWindows Programmers' Guide,* Sun Microsystems, Inc., Jan. 1984.
8. L. Tesler, "The Smalltalk Environment," *Byte,* Aug. 1981, pp. 90-147.
9. A. J. Wilkes et al., "The Rainbow Workstation," *The Computer Journal,* Vol. 27, No. 2, 1984.
10. G. Williams, "The Apple Macintosh Computer," *Byte,* Feb. 1984, pp. 30-54.
11. *Interlisp Reference Manual,* Xerox Corp., Pasadena, Calif., Oct. 1983.
12. S. A. Bly and J. K. Rosenberg, "A Comparison of Tiled and Overlapping Windows," *Proc. SIGCHI 86: Human Factors in Comp. Systs.,* Boston, Apr. 1986, pp. 101-106.
13. *User's Manual for Release 1 of the Information Technology Center Prototype Workstation,* Information Technology Center, Carnegie-Mellon Univ., Pittsburgh, 1984.
14. W. Teitelman, "A Tour Through Cedar," *Software,* Vol. 1, No. 2, Apr. 1984.
15. L. Bannon et al., "Evaluation and Analysis of Users' Activity Organization," *Proc. SIGCHI 83: Human Factors in Comp. Syst.,* Boston, Dec. 1983, pp. 54-57.
16. B. A. Myers, "The Importance of Percent-Done Progress Indications for Computer-Human Interfaces," *Proc. SIGCHI 85: Human Factors in Comp. Systs.,* San Francisco, Apr. 1985.
17. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics, Second Edition,* McGraw-Hill, New York, 1979, pp. 261-265.



Brad A. Myers is a PhD candidate in computer science at the University of Toronto and an occasional consultant. From 1980 until 1983, he worked at PERQ Systems Corp., where he designed and implemented the Sapphire window manager and many PERQ demonstrations for the SIGGraph equipment exhibition. His research interests include User Interface Management Systems, or UIMSs, user interfaces, interaction techniques, window management, programming environments, debugging, and graphics.

Myers received the MS and BS degrees from the Massachusetts Institute of Technology, during which time he was a research intern at Xerox PARC.

Myer's address is Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto, Toronto, Ont., M5S 1A4, Canada.