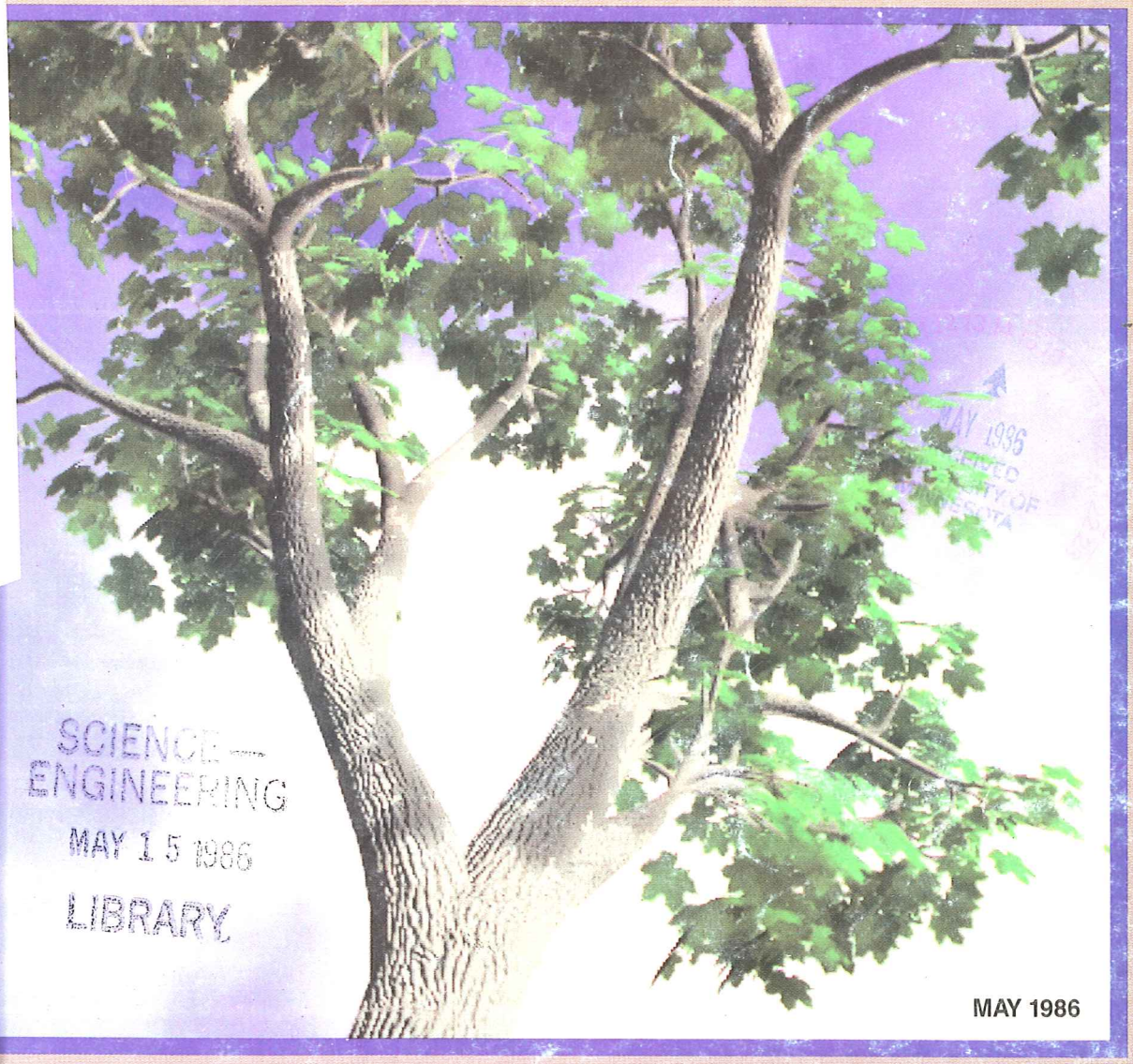
IEEE

Computer Graphics and applications



MAY 1986

1st INTERNATIONAL CONFERENCE ON COMPUTER WORKSTATIONS

- Productivity in the Creative Process
- WHIM–Window Handler and Input Manager
- Constraint-Based Tiled Windows
- Bridge from Full to Reduced Function
- Intelligent GKS Workstation
- Workstation Research Project

IEEE COMPUTER SOCIETY

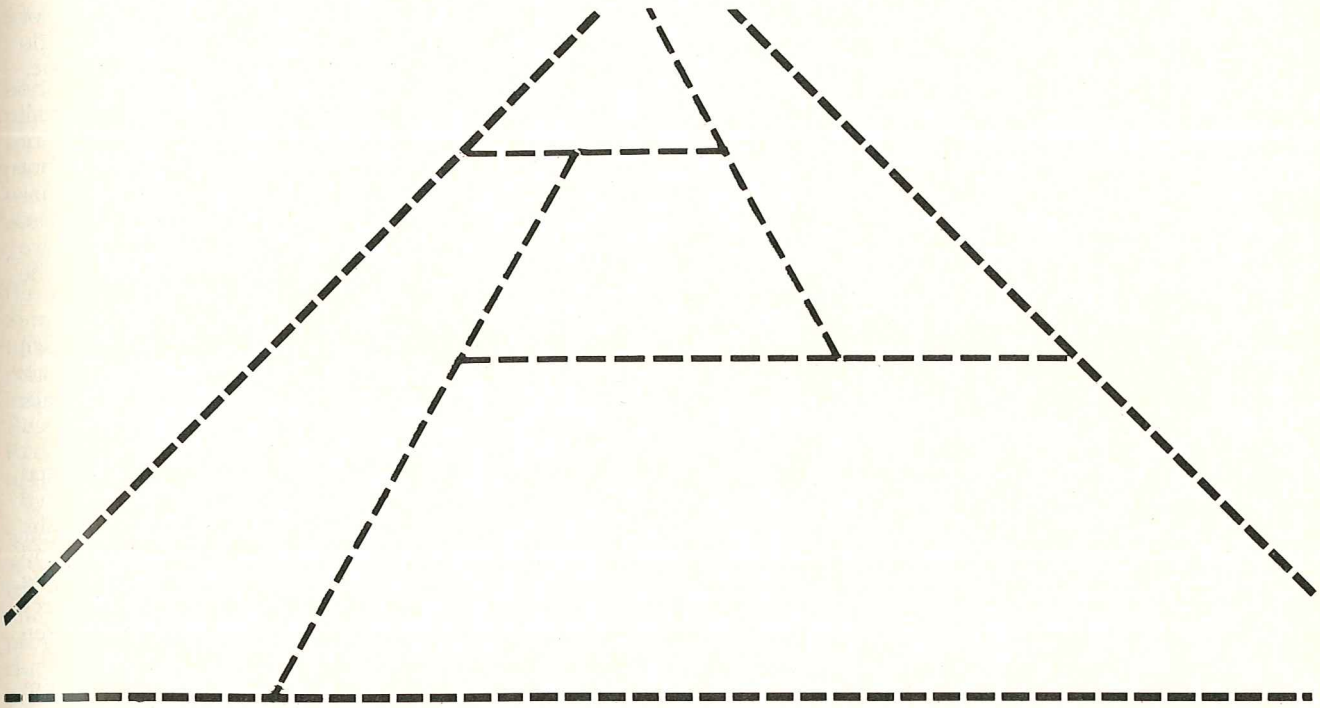
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Constraint-Based Tiled Windows

Ellis S. Cohen
Siemens Research & Technology Laboratories

Edward T. Smith
Carnegie-Mellon University

Lee A. Iverson
Siemens Research & Technology Laboratories



Typical computer workstations employ window managers for creating, destroying, and arranging windows on the screen. Window managers generally follow either a desktop metaphor, allowing windows to overlap each other like sheets of paper on a desk, or they use a tiling model, arranging each window with a specific size and location that avoids overlap. Desktop models allow for the most layout freedom, but can be frustrating to use when dealing with a large number of windows that must all be visible at once. Tiling models guarantee that each window will be completely visible on the screen, but thus far have provided relatively poor mechanisms for controlling layout decisions.

This article describes work in tiled window management featuring a constraint-based layout mechanism. With it the user can specify the appearance of individual windows and constrain relationships between windows, thus exercising necessary control over the tiling process. We discuss our constraint model and then detail an implementation approach that would make use of those constraints.

Window managers generally follow either a "desktop" metaphor, allowing windows to overlap each other like sheets of paper piled up on a physical desk, or they use a "tiling" model, arranging windows so that no overlap occurs.

Desktop models allow for the most freedom in arranging windows. Each time a window is created, the user typically is queried for its size and location. At any time, the user can rearrange the windows, changing the size, location, and/or degree of overlap. But users of desktop models pay a price for this flexibility, especially when large numbers of windows need to be accommodated. The desk top can become messy, forcing the user to constantly locate and rearrange windows.

Tiling offers a potential solution to the problem of messy desk tops. Tiling is especially useful for applications that generate large numbers of small, short-lived windows. The application specifies the constraints on its windows in a constraint language (though the user can modify the windows and constraints dynamically). Overlapping windows may be more useful above the application level, for systems with small screens, or for applica-

0272-1716/86/0500-0035$01.00 © 1986 IEEE

tions that use few windows over a long period of time. There is no denying that overlapped windows have their benefits, and that tiling presents problems of its own. In this article we will try to show that the benefits of overlapped windows can be provided by tiled systems (though the techniques may be quite different), and that the use of constraints solves most of the problems associated with tiled systems.

One approach to building a tiled system allows (and forces) the user to control the tiling. Each readjustment of window sizes must be done explicitly. This approach is useful and safe, but can require enough work of the user to become frustrating.

Another approach automatically retiles the screen, based upon the user's actions. This approach has a serious problem: It can do the wrong thing, requiring the user to readjust the layout manually. Fortunately, this problem is not inherent in automatic retiling. With the aid of constraints, the user rarely should have to fiddle with the layout, because the constraints can ensure that the right windows are on the screen in the right place with the right size.

Many users find tiled window systems quite frustrating at first, but later prefer them to window-overlaying systems.

Constraints are the key to making tiling work, and a large part of this article discusses the kinds of constraints that are needed. Briefly, the constraints we discuss specify

- which windows must be present on the screen
- their size and location
- adjacency and alignment requirements
- limits on the ways in which the system may automatically change the layout
- particular organizations of windows on the screen

Since window managers are real-time systems that must respond immediately to changes on the screen, not much time can be spent solving the constraint problem. Constraint systems that *work*—and don't take an exponential amount of time in the process—use domain-specific knowledge to reduce the work needed to satisfy the constraints. This article describes in some detail how we propose to use such knowledge in a tiled-window implementation.

We will describe tiling systems as if they use the entire screen. However, it is reasonable to expect—especially on large screens—that tiling will be applied *within* a window, by an application to its subwindows. We are not going to insist that tiling replace overlaying. There are good reasons for using both mechanisms.

Our system takes ideas from both architecture layout systems[1,2] and VLSI layout systems.[3–5] First a rule-based component is established to adjust the configuration of the windows. Then algorithms related to those used for VLSI compaction are used for determining window sizes once the configuration has been decided.

Why tile?

Tiling is not universally recognized as a good method. If the user is provided with a quick and easy mechanism for changing the size and location of his windows in an overlapping desktop environment, why would he want a tiling mechanism? The reasons for having a tiling mechanism are somewhat more convincing when there are a lot of windows on the screen, but the arguments remain valid even when the number of windows is small.

Managing windows. Whenever there are more than a few windows on the screen, managing them for effective viewing can be a burden for the user. Users end up spending much of their time manipulating the windows and window structure rather than working with the contents of the windows. Tiling offers the option of automatic control of windows, benefitting users who do not need to arrange and rearrange windows all the time.

Finding hidden windows. The desktop metaphor is all too real an analogy. Any of us whose desks get messy are familiar with the problem of locating a paper that is not right on top. The same situation arises when some windows are hiding under others. In a tiled window system, windows for which there is no room on the screen are automatically closed and iconized so they can easily be located.

Wasted screen space. Partially overlaid windows are all too often simply wasting space. Since tiling systems work by *filling* the screen with windows, screen space is always going to be allocated to some useful purpose.

Automatic placement. When a new window is opened in an overlaying desktop system, the user is usually asked to use the mouse to indicate the size and location of the window. This is flexible, but can become boring and annoying. Some systems provide the option of not asking and always put the same window in the same place, or try to put it where there is some semblance of open screen space.

The most convincing evidence for tiling comes from users. Many people who have used tiled window systems find them quite frustrating at first. The Andrew System* occasionally retiles automatically in unexpected ways (especially for novice users).[6] Quite a few people have found it difficult and unpleasant to use at first, and few would claim that it represents a really good tiling system. Nonetheless, those who persevered have learned to tame it, and many now prefer it to the competing candidate, Sun's window-overlaying system.

Even more convincing evidence comes from former users of the Cedar system[7–9] who were initially reluctant to use tiling, but came to like it. Some who have returned to overlapping windows are now annoyed at having to spend so much time moving and reshaping windows to make the screen appear tiled.

Tiling models

A system can provide different ways to tile a screen and different degrees of automatic retiling. In this section we look at a number of tiled window systems that already exist. In particular, it should help readers new to tiled

*We are referring to the initial version of the Andrew System. The latest version of the system uses a tiling system similar to the Cedar system.

windows to identify the strengths and weaknesses of such systems, understand their user interface requirements, and recognize how and where constraints can help.

Tiling organizations. Systems have been built to support four different kinds of tiling organizations.

Single column. Windows are lined up vertically in a single column. Most EMACS-like editors fall into this category, as does the virtual terminal manager of the RIG system.[10]

Multicolumn. The screen is divided into multiple columns, and windows are lined up vertically in each column. This model is used by Xerox PARC's Cedar System[11] (with a limit of two columns) and the Microsoft Windows System[9] (with a limit of 12 columns).

Hierarchical. The screen is divided either vertically or horizontally into partitions, and each partition is then recursively divided into subpartitions. Windows correspond to the final set of undivided partitions. Each partition corresponds to a node in a hierarchy, with windows at the leaves. The CMU/ITC Andrew System[6] uses this window organization.

Nonhierarchical. Any rectangular tiling can appear in this system. We have built such a system—RTL/RTL, or Siemens Research and Technology Laboratories/Rectangular Tiled Layout—on top of PERQ Systems' Sapphire Window System.[12] The system we describe in this article and are now building—RTL/CRTL, or Siemens Research and Technology Laboratories/*Constrained* Rectangular Tiled Layout—will also use this window organization.

Single-column systems are generally adequate for text windows on standard 80-column terminals. With wider screens or graphics multiple columns often become useful. Multicolumn layouts are certainly an improvement and are quite adequate in the majority of cases. They provide a clear, understandable layout and user interface. But there are cases where columns are not the best solution.

Figure 1 shows two cases where hierarchical layouts would be more useful than multicolumn layouts. In each case the screen is more or less divided into two columns. However, in the first case window E needs to be very wide (perhaps because it contains a strip graph for music or speech analysis). In a pure multicolumn system A and D would have to be in one column, giving them unnecessary additional width while reducing their height.

In the second case windows B and C contain related information and need to be placed side by side. If they were forced into the two major columns in the layout (i.e., C would stay in the right column with D while B and A would be placed in the same column), space would be wasted by making B and C each wider, and valuable height might be taken away from window A by placing B in the same column with it. Though these have not tended to be major problems for most Cedar users, they do suggest that there may be some advantages to a more general layout scheme.

Our choice, then, is between a hierarchical and a nonhierarchical system. There are two ways in which these differ. First, there are some inherently nonhierarchical window layouts. Figure 2 is one such layout. Where windows have minimum size constraints, a nonhierarchical layout can sometimes accommodate more windows than a hierarchical one.

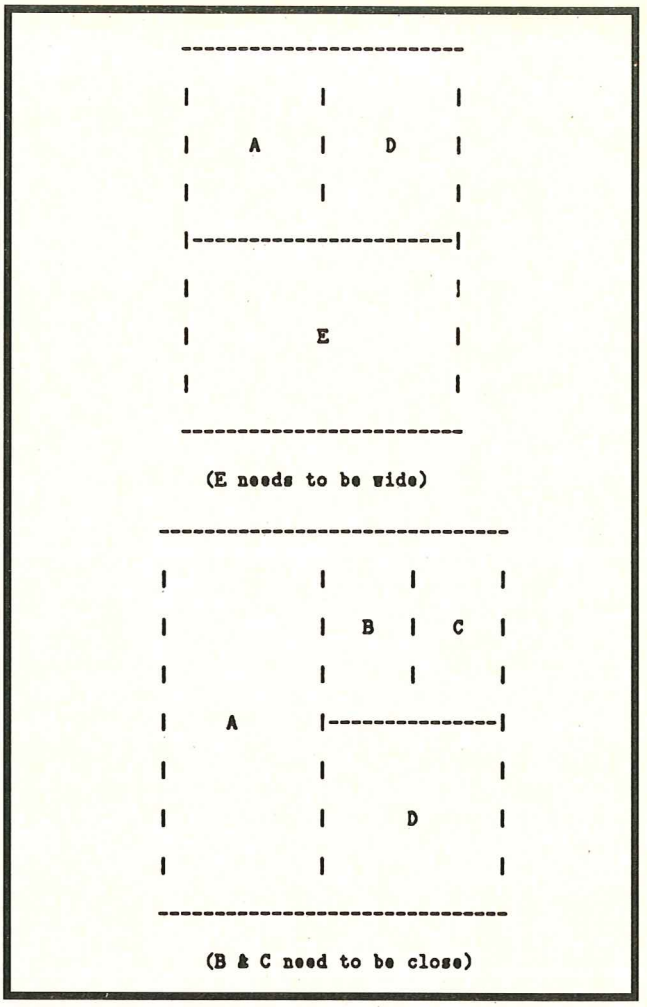


Figure 1. A hierarchical layout.

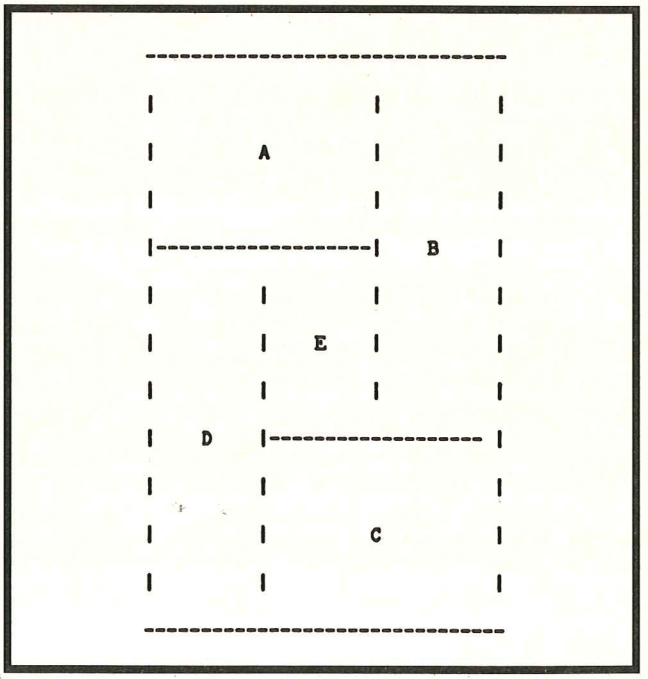


Figure 2. An inherently nonhierarchical layout.

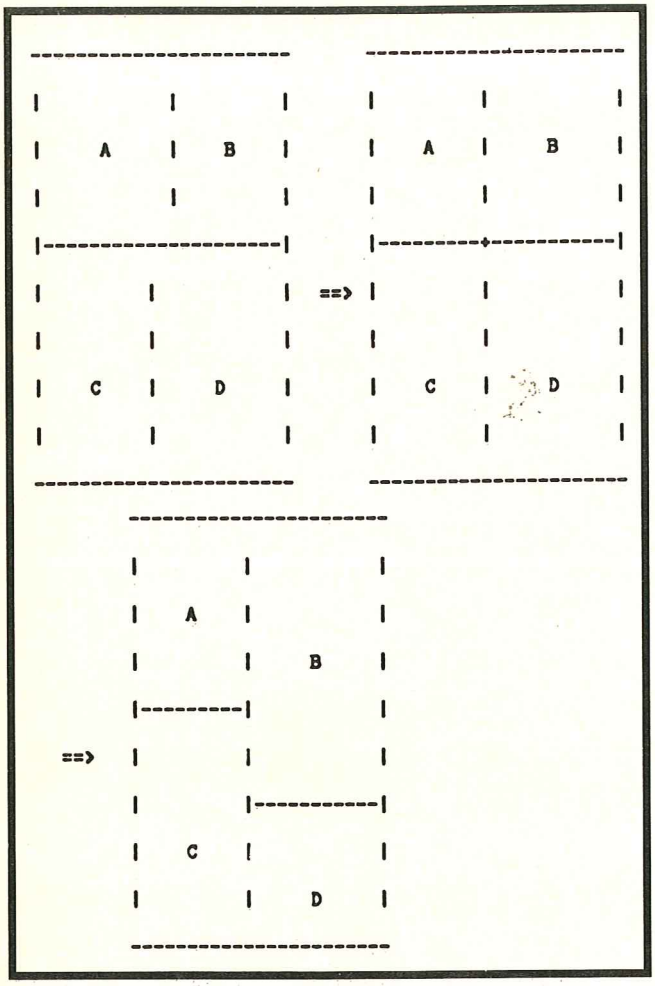


Figure 3. A layout change that alters the hierarchy.

In addition, certain layout changes that ought to be very easy tend to be difficult to make in hierarchical systems.

For example, in Figure 3 it should be possible to first widen B, lining its left edge up with D's, and then to lengthen it, shrinking D in the process. This is very easy to do in our nonhierarchical RTL/RTL system, but almost impossible to do in Andrew, which is hierarchical. The reason is that the horizontal line separating A and B from C and D defines the top level in Andrew's hierarchy. An attempt in Andrew to pull B down would require changing the hierarchy completely, pulling down A as well.

In RTL/RTL this is not a problem. When the user first tries to widen B and align its edge with that of D, the system provides a *gravity* feature to help. When the left edge of B gets close to the edge of D (currently within 20 pixels), the system lines them up, and the internal representation (a variant of the corner-stitch representation used in the Magic VLSI Layout System[5]) is modified accordingly. In pulling down B in RTL/RTL, only B is enlarged (although if while pulling down that edge, the cursor is moved into C, A's edge will begin to be pulled down along with B's).

Manual versus automatic tiling. The appearance of the screen in a tiled window can be manually controlled by the user, or can be automatically controlled by the system.

Resizing and locality. When a window is resized, will space be given to or taken away from the adjacent window (a manual, local effect), or can other windows potentially change their size and location (an automatic, global effect)?

Typically, to resize a window along one dimension, the user grabs a window edge with a mouse and then moves the mouse, pulling the window edge along. As the mouse moves, some indication of the new border is indicated: either a dotted line or cross-hatching. Resizing is finalized when the mouse button is lifted.

In Andrew and RTL/RTL effects are manual and local. As the window edge is moved, one window is enlarged and the adjacent window is shrunk. Other windows are not affected. This reduces repainting costs, which can be time-consuming for low-end workstations.

Gosling's EMACS uses an interesting compromise: It is as local as possible. As a window is enlarged, it bites into the adjacent window until the adjacent window is only one line high, and then that window is "plowed" along, biting into the next adjacent window.

In Cedar and RTL/CRTL, effects are automatic and global. When a window is enlarged, all the remaining windows are shrunk by a proportional amount.

Global changes make the most sense when there are minimum size constraints that prevent windows from being shrunk below a useful size. Both Cedar and RTL/CRTL have minimum size constraints. In both systems extra space (the space left over after satisfying each window's size constraints) is prorated among the windows, although in RTL/CRTL options may control the degree of prorating.

Window placement. When a window is opened, can or must the user manually indicate where it goes? Or is the decision made automatically, and if so, how?

Systems differ widely in the way they deal with an Open. RTL/RTL is the most explicit. After specifying a window to open, the user places it in a crack between two tiles and then pulls on the edge to make space for it. The new window is effectively enlarged from a starting dimension of zero.

Both Cedar and Microsoft Windows support automatic placement, placing the new window at the bottom of a column. Cedar takes space away from the other windows proportionately; Microsoft Windows splits the bottom one.

Microsoft Windows also supports two manual placement techniques. In one the window is simply dropped in the crack, and space for the window is taken away equally from the windows on both sides of it. Microsoft Windows also allows the user to drop the window on top of an existing window, and the old and new windows then change places.

Andrew supports both manual and automatic placement. For automatic placement, it finds a window large enough to be split so that even after space for the new window is taken away from it, it is still larger than its size constraints specify. In manual placement the specified location is used to find a location in the window hierarchy, which determines the actual placement.

In RTL/CRTL we will support both manual and automatic placement. For automatic placement, we find the best place to put the window—a region where existing windows can give up space to the new window and still satisfy size constraints.

Manual placement in RTL/CRTL can be either exact (the window is placed exactly where the user specified) or approximate (it may be moved slightly to accommodate surrounding windows). In either case, we prorate space for the remaining windows, taking enough space away from each to accommodate the new one.

Closing windows. Must windows be closed explicitly (by clicking on a close "gadget" or selecting a menu item), or can a window be closed automatically as a result of enlarging or opening another window?

Enlarging or opening a new window must take space from other windows. If the system imposes minimum size constraints, and there is no space available, then either the system must not allow the window to be resized or opened, or some other window must be closed automatically.

In systems where resizing has local effects, enlarging a window means taking a bite out of an adjacent window. When the adjacent window becomes too small, either enlargement is prohibited (as in Gosling Emacs when there is no room left to plow), or the adjacent window is automatically closed, as in Andrew and RTL/RTL. Andrew also may close windows when another window is opened, and it appears to be hard to tell in advance which windows will be closed.

Neither Cedar nor Microsoft Windows automatically closes windows. Cedar will not allow a window to be enlarged if it would force all the other windows in the column to be shrunk below their minimum. Cedar is less conservative about opening windows. It will open a new window even if it means shrinking the remaining windows below their minimums. On the other hand, Microsoft Windows will not open the window if there is no space available for it.

RTL/CRTL has a variety of options for treating automatic closure. If explicit closure is required, then minimum size constraints will be violated. Alternatively, limits will be placed on enlargement, and windows that are too large will not be allowed to be opened until the user closes other windows.

If the user chooses to permit automatic window closure in RTL/CRTL, then the windows to be closed may be required to be local so that only windows adjacent to the opened window will be closed. Alternatively, windows will be closed on the basis of age and priority.

Window management techniques

When working with overlaying window systems, users develop various personal styles for shaping and positioning windows to make the most use of the screen space available. As a result, the features of overlaid windows that we have previously described as disadvantages have become useful properties to many users. The tiling system can often provide the same effects, though in different and sometimes more elegant ways.

Peeking. Users can "peek" at what is going on in a window by overlaying all but a sliver of it. In the desktop model, a user can move windows relatively easily to obtain just the sliver or slivers desired. In a tiled window system, as long as the sliver is rectangular, the window can simply be shrunk to the sliver size. However, there are a number of difficulties.

1. **Multiple slivers.** Sometimes the user wants to peek at multiple slivers—on both the bottom and the left side, for example. There is no easy solution to this, but this does not appear to be a major problem.

2. **Virtual mapping.** In a desktop system the sliver shown is just the part of the window that remains visible. But when the window is shrunk, it is not clear whether it should show the left, right, or middle of the original image. The simplest answer would be to think of the image as fixed on the screen, with the movement of the window edges determining what is covered up. Alternatively, the window can determine the direction of shrinkage (or expansion). If the user desires a different sliver, the window must provide scrolling.

The best solution is of course a semantic one. The window menu could include one or more summarizing commands. Each such command would shrink the window and replace its contents with some sort of summary. To provide complete support, the process controlling the window should update the summary in a way corresponding to how it updates the full contents. This is really just a generalization of the "smart icon" approach used in systems such as Cedar[11] and Sapphire,[12] except that the summary window appears in the main region of the screen rather than in an icon area.

Retained size. In a desktop system the user adjusts the size of a window to one appropriate for its use. It can always be overlaid by other windows. When it needs to be used again, it is brought to the top where it has its original size. In a tiled system, the window may be shrunk to its minimum size or even iconized when it is not in use, and needs to be explicitly resized when opened.

There are two closely related solutions. Both depend upon retaining the current size.

1. The user or application can easily set the minimum size to the current size. Even if the window is iconized, it will have this size as a minimum when it is reconstituted.
2. In addition to the ordinary minimum window size, a window has a *use size* which is set whenever the window is explicitly resized. We can opt to make the window at least this large if it is being used. The window may shrink (or be iconized) if it is not in use. However, when the mouse is buttoned within the window (or when it is reconstituted), it will be enlarged to the use size.

By employing use sizes and setting the minimum width to sliver sizes, an effect very similar to overlaying can be achieved. In Figure 4 window A is being used, while B has been allowed to shrink and appears as if it were partly overlaid by A. When the mouse is clicked in window B, B expands to its use size. A is allowed to shrink and appears as if it had been partly overlaid by B.

After A has shrunk, the same contents may be mapped to the upper part of A, giving the same effect as overlaying. Or, the contents may be shifted or even summarized. This choice is a property of the window, and is aimed at ensuring that the most useful information always remains visible.

Window families. Users often create a stack of windows, with one of the windows in the family on top. Desktop models trivially allow for such windows to be shown as an overlapped stack of windows, each offset by

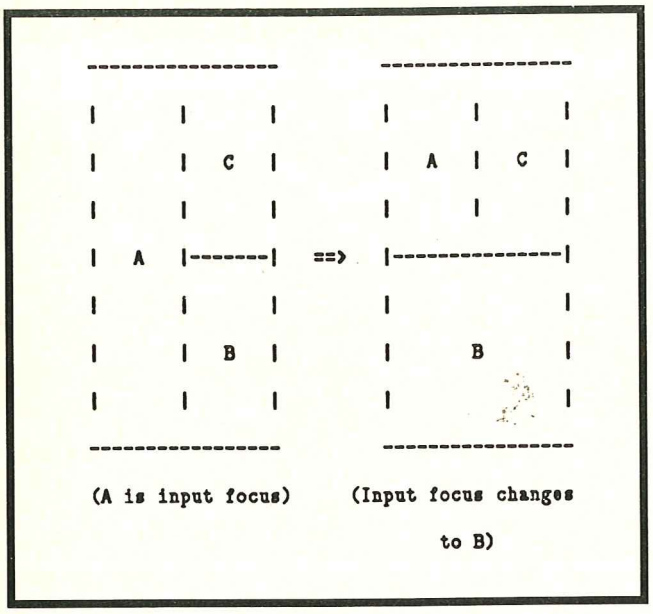


Figure 4. Overlay effects with tiled windows.

the desired amount so some portion of each window is visible.

This functionality can be supported through the use-size mechanism discussed above. The window currently in use can be shown full size while the remaining windows in the stack are shown as adjacent slivers.

Another approach provides *family* windows. A family window displays a single window of the family (the one on top in the desktop model). In addition to showing the header of the top window, the family window has its own header containing index tabs for the other windows. To switch to a different window, the tab for that window is selected and the contents of the selected window replace the current window. Naturally it is possible to move additional windows into a family, and to remove a window from a family and display it separately on the screen.

Stability. In a desktop system windows don't change their size or location unless the user moves them, even when other windows are being created or destroyed. Since tiling systems can reconfigure the entire screen at each window creation or destruction, all windows on the screen can be shifting constantly. This can result in a much more confusing display for the user, even overriding the advantage of using screen space efficiently.

To deal with this problem, a tiled window system must explicitly address stability. In a later section we introduce two types of stability constraints. One ensures that the layout of the windows changes minimally, and the other retains the size and location of as many windows as possible.

Repainting. When the size or location of a window changes, repainting the window is necessary. In a tiling system many windows may change as a side effect of creating, destroying, or resizing a single window. This necessitates a great deal of repainting, and can be especially costly for programs that paint their windows differently depending on the size available.

This problem can be avoided by fixing, or appearing to fix, the window size or location.

1. The size or location of a window can be locked so that resizing will only occur at the user's command. A less absolute approach is to set appearance stability (see the section on stability below) very high so that resizing or relocation occurs only as a last resort.
2. A virtual image can be interposed between the application and the screen to hide resizing from the application. When the actual window is smaller than the virtual image, the image is clipped (in a manner specified by the window's properties). When the actual window is larger, the virtual image is displayed inside the window surrounded by a blank mat.[6]

Temporary enlargement. Users sometimes want to temporarily enlarge a window, work with it for a while, and then return to the original configuration. In the desktop model the window is enlarged, but the configuration of the windows beneath it remains unchanged. When the window is shrunk, the original configuration is again visible. A pure tiling system would be forced to shrink or iconize all other windows.

There are a variety of solutions to this problem. They are each quite different and place different requirements on the overall window manager.

1. If a window is enlarged (especially if it is enlarged substantially), other windows are very likely to be iconized. When room is once again available on the screen (perhaps by shrinking the enlarged window), certain iconized windows (in particular, those forced off the screen) can be automatically reconstituted. Each icon retains the size and location of its corresponding window when last displayed. By having the system place iconized windows back in their previous location when reconstituted after the enlarged window is shrunk, the resulting configuration is likely to be the same before and after.
2. Perhaps the simplest solution is to allow any single tiled window to be enlarged to the size of the screen. Deleting it brings back the original tiling.
3. Another option is to support separate desk tops (as in SmallTalk, Cedar, and RIG VTMS), each one the size of the screen, and allow a window to be in several desk tops at once. Rather than being temporarily enlarged, the window is explicitly put into a new or existing desk top along with any other windows needed when working with it. In this other desk top, the window can be made as large as necessary. Switching back to the original desk top returns to the previous configuration. Desk tops can be built very easily if the system provides the ability to take a "snapshot" of a configuration (remembering the size and location of each window on the screen) and save it as a desk top.
4. A fourth option is to integrate tiling with overlaying rather than having a pure tiled system. In its simplest form this could mean having two layers: a layer on the bottom holding only tiled windows, and another layer above it in which windows are overlaid. In more general window systems, tiled and overlaid windows can be arbitrarily nested inside one another.

Constraints

In the RTL/CRTL system we are now building, constraints are used to control the layout of the system. Constraints can be specified in several ways. A constraint language can be used by an application to specify a

default window layout. Constraints can also be specified dynamically so that the layout can be adjusted as execution of the application proceeds. In addition the user implicitly asserts constraints when resizing or moving a window.

Constraints have priorities, and the system attempts to satisfy the constraints in priority order. Constraints asserted by a user action (such as explicitly moving or resizing a window) initially have very high priorities to ensure that the window remains as the user indicated.

Constraints specify

- which windows must be present on the screen
- their size and location
- adjacency and alignment requirements
- limits on the ways in which the system may change the layout
- particular organizations of windows on the screen

Presence constraints. In a tiled window system not all windows may be able to fit on the screen. The remainder are not displayed and may be iconized in a special icon area. To help the system decide which windows should be displayed, a variety of *presence* constraints can be specified.

Window priorities. The simplest form of presence constraint specifies the priority of a window. When a visible window is explicitly enlarged, or a new window is opened, there may no longer be space for all the remaining windows. The window with the lowest priority is usually closed and iconized.

The priority of a window can be thought of as constant: Some windows are inherently more important than others. However, no matter how inherently unimportant, a window currently in use must remain on the screen. In effect, window priorities must be based on an aging scheme. On use, a window attains the maximum priority. When it is not in use, its priority is reduced to the level of its inherent worth. There are two things the system might allow the user to specify here: aging and what constitutes use.

The aging function would generally be linear, in which case the user would specify the rate down to some specific minimum value. Use probably means input, so that aging should start just after a window has lost the input focus. Use might also include output, so that the user might want the window to remain present as long as output is being directed to it.

Required presence. Often the presence of one window requires the presence of another. Joint windows can be most easily arranged by making them subwindows of a larger windows. SmallTalk's browsers[13] are a prime example of this. This approach is certainly adequate, but independent windows have some advantages. One is that an unnecessary window can easily be deleted. This may not be so easy if it is a subwindow of a larger window.

Another advantage is that independent windows are not forced to be adjacent. Adjacency is desired for related windows, but presence may be more important. There could be cases where a window might be forced off the screen to satisfy an adjacency requirement. An application may want to place all windows of one kind at one end of the screen, and windows of another kind at the opposite end. For each window of the first kind present, there would be a requirement that a corresponding window of the other kind be present. In this case, required presence is desired, but adjacency is not desired at all.

Consequently, we provide *required presence* constraints. These constraints have priorities which interact with window priorities.

Automatic reconstitution. When a window is explicitly shrunk or closed, more space becomes available on the screen. Iconized windows can be reopened, perhaps automatically. Automatically reopening windows can sometimes be disconcerting, especially if the user no longer wants to see those windows. The priority of iconized windows may not in itself be a good indicator for determining which windows to automatically reconstitute. It may be desirable to have a global "aggressiveness" setting the user can set to control how aggressively iconized windows can be reconstituted.

Layout constraints. Window layout is controlled by a group of layout constraints that constrain the appearance of an individual window, or that establish geometric relationships among windows.

Appearance constraints. These determine the appearance (size and location) of an individual window. *Size* constraints constrain the height, width, aspect ratio, or area of a window. *Location* constraints constrain the location of a window's edge, corner, or center.

Association constraints. These specify relations between windows. *Adjacency* constraints specify that two windows must be adjacent, and can additionally constrain their degree of adjacency (how much of their borders must be in common). *Alignment* constraints specify that edges of two windows must be aligned, or constrain the distance between them.

Priorities and inconsistency. Constraints may be inconsistent. In some constraint-satisfaction systems, inconsistency will simply halt the system. Smarter systems continue by arbitrarily breaking constraints. The IDEAL system,[14] a picture-specification system, even allows the user to tag constraints that may be broken.

We go a step further. Every constraint may have a priority associated with it. These priorities may be fixed, they may be tied to the priorities of the windows they constrain, or they may be aged in some independent way.

Constraints are satisfied in priority order. Any number of low-priority constraints may be left unsatisfied to satisfy a higher priority constraint. Even if a high-priority constraint cannot be satisfied, an attempt should still be made to satisfy lower priority constraints.

Layout constraints interact with presence constraints. If the priority of a layout constraint involving a window is higher than the priority of the window itself, the layout requirement is more important than the presence of the window itself. If the layout constraint cannot be satisfied, the window is iconized.

Stability constraints. A small enlargement of a single window could cause the constraints of adjacent windows (typically minimum width or height constraints) to be violated. The subsequent reconfiguration could conceivably shuffle the entire layout to best satisfy the constraints. Such behavior would almost guarantee that users will avoid window tiling systems. Stability constraints prevent gross reshuffling and allow the tiled-window model to be usable.

Global stability. Unlike other constraints, stability is primarily global, determining the appearance of the screen as a whole. Nonetheless, individual windows may participate in global stability to a greater or lesser degree.

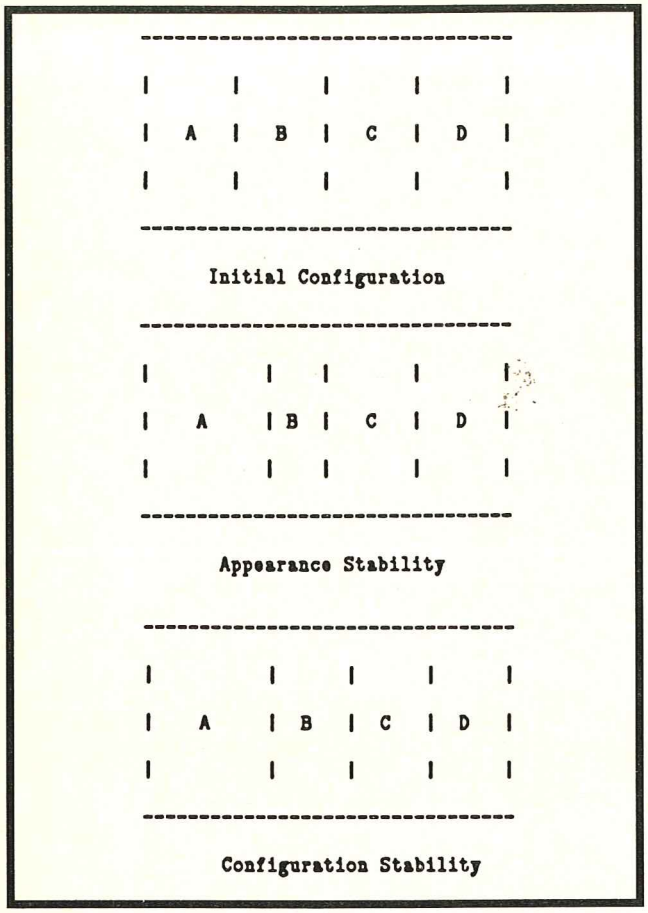


Figure 5. Configuration vs. appearance stability.

1. *Configuration* stability requires that the adjacency relationships between windows remain the same (as much as possible) even though the size and locations of the windows may change. The priority determines how many adjacency relationships can be changed. (For more details see the description of realignment in the adjustment model in a subsequent section of this article.)
2. *Appearance* stability (possibly divided into size stability and location stability). This requires that the appearance (the size and/or location) of as many windows as possible remains the same. The priority determines how much the appearance must stay the same. In essence, constraints whose priority is lower than that of global appearance stability may be ignored to maintain the appearance.
3. *Presence* stability can be used to control which windows may be closed when other windows are enlarged or opened. By setting the priority of presence stability low, we allow all but high-priority windows to be closed automatically. By setting presence stability at its maximum priority, no automatic closure of windows will occur. Other constraints (generally size constraints) will be violated instead.

Some of the differences between these can be illustrated in Figure 5. Imagine that the screen is initially laid out as a row of four windows all having exactly the same size constraints. The user then explicitly enlarges window A. Configuration stability requires only that the adjacency relationships between the windows be retained. In this case it is easy as long as B, C, and D can all be made smaller. Since they all have the same size constraints, they are shrunk the same amount. If they could not be made this narrow, then the one with the lowest presence priority would be deleted.

When appearance stability is made very important, the system tries to retain the size and/or location of as many windows as possible. In this example C and D retain their size and location while B is greatly reduced in size, even though B, C, and D all have the same size constraints.

Windows and the constraint language. Windows are tiled within a desk top. In the simplest system there will be exactly one of these at a time, and it will be the size of the screen. In a more general system an entire tiled desk top may be a window that is itself part of a parent desk top (either tiled or untiled).

A tiled desk top has a set of constraints associated with it that control tiling within the desk top. These constraints may be described by a *constraint language.* This article does not include the details of that language (it is still in flux), but the language is similar in spirit to those used by others to specify floor and VLSI layout (see Roach in particular[15]).

A *specification* in the constraint language is used to initialize the constraint set of a newly defined desk top. This is typically done when an application starts up. However, the constraints may be changed dynamically by an application or by the user.

Explicitly resizing a window asserts a size constraint, and explicitly moving a window asserts a location constraint. In addition, through a menu associated with each window, the user can lock the presence, size, or location of a window. This can have the effect of asserting at high priority the corresponding constraints. If two windows are placed so that they are aligned or adjacent, they can be constrained to remain this way.

Windows may be organized in a hierarchy of window classes. In addition applications may dynamically associate attributes with windows. Constraints may be written so that they refer to windows in a particular class and/or with a particular attribute.

Binary relations may be defined between windows, and an application can dynamically include or remove a pair of windows. A constraint can be limited to window pairs that take part in a relation. For example, in a debugging situation, a stack frame window may be related to its corresponding source code window, and a constraint may require such pairs of windows be adjacent when present.

Finally, there is one language feature we believe to be important, but which seems guaranteed to impose cost penalties on any tiling algorithm: support for *constraint alternatives.* An example of this is a constraint that requires a window to be either at the top left or bottom right of the screen. We can minimize the cost of allowing alternate groups of constraints if each constraint in the first group has a higher priority than any constraint in the second group (and so on if there are more than two groups). Since we try to satisfy constraints in priority order, this ensures that the group of constraints is tried only if a preceding group cannot first be satisfied.

Reconfiguration. Constraints may change in many ways. Applications or the user can add or change them,

and their priorities can also be changed explicitly as a result of aging. Each change could have an effect on the layout, but a continuously changing layout would not please the user. The layout ought to change only at the following times:

1. when the user or application directs the windows to be reconfigured
2. when a window is explicitly moved, resized, opened, or closed
3. when the input focus is changed to a window, and when a value or priority of that window's size constraints has been changed in a way that would enlarge it along some dimension

An implementation: The adjustment model of RTL/CRTL

The problem of window tiling is very closely related to the architectural floor layout problem,[1,16,17] and to VLSI layout problems when treated as floor layouts.[5,18,19] The floor layout problem specifies a number of rooms; minimum length, width, and area constraints on each room; adjacency requirements between rooms; and the dimensions of the floor in which the rooms are to be placed. There are systems that solve floor layout problems with modest efficiency, but none run in anything like the response time that is our goal: well under a second with a reasonable number of windows and straightforward repainting. Two major differences between the window tiling problem and the floor layout problem permit retiling to be done quickly.

1. **Stability.** Stability limits the extent of layout changes. So far, we have viewed this as a human engineering feature to prevent user confusion and frustration. It also limits the search space for solutions so that relatively cheap heuristics can be used.
2. **Presence.** Not all windows need be present. Usually, of course, not all windows can be present. The highest priority windows are displayed, while the rest are iconized. It would be best if as many windows as possible could be fit on the screen. However, it is acceptable if a fast heuristic approach puts up one less window than could be squeezed in by an exhaustive search.

Consequently, our implementation is based on an adjustment model, which reconfigures by making local changes, or *adjustments*, to the existing layout. By making only local changes, the adjustment approach is inherently geared toward maintaining configuration stability. Adjustment is divided into four separate phases.

1. **Prorating.** When constraints change, typically due to the resizing of a window, we try to maintain essentially the same configuration by adjusting the sizes of the other windows.
2. **Realignment.** If a constraint cannot be satisfied even after prorating, a local change (possibly including the closing of a window) is made to the configuration aimed at satisfying the broken constraint.
3. **Placement.** This determines the best place for a window to be opened. Placement may also be used to find a new location for a window which has been forcibly closed during realignment, and as a first step in reconfiguring the screen around a window which the user has explicitly relocated.
4. **Initial partitioning.** When the screen changes in a major way, the windows are partitioned hierarchically from scratch with a fast but approximate technique. The other phases are then used to produce the resulting configuration.

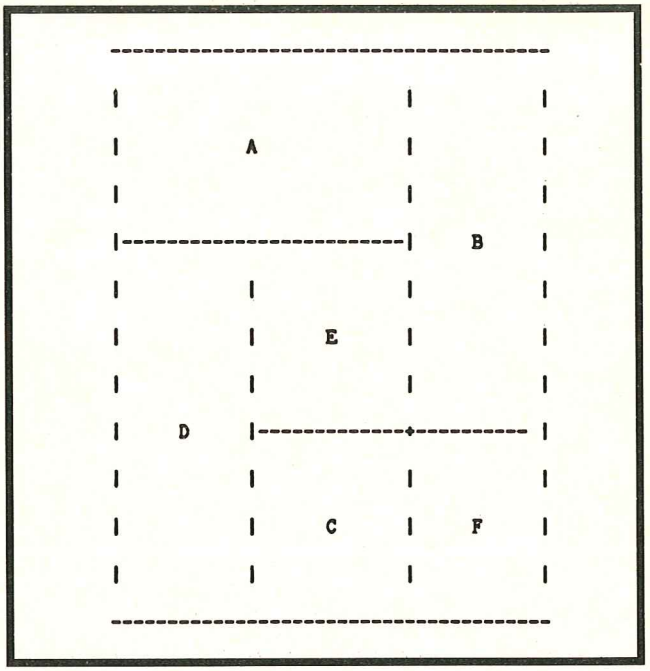


Figure 6. Configuration before widening A.

Prorating. When a window is resized, other windows must be resized to make room for it. Prorating is similar in some ways to VLSI compaction. As in the Earl VLSI Layout System,[3,4] we deal with the horizontal and vertical dimensions separately.

We find the most constrained path along a particular dimension and allocate space to the windows along that path. Ignoring those windows, we again find the most constrained path and allocate space along it, iterating until the size of each window in that dimension is determined.

We can solve for horizontal and vertical dimensions separately because, for a given configuration, almost all the constraints operate in one dimension only. Only aspect and area constraints operate in two dimensions, and these do require special treatment. (Because our constraints can be represented algebraically, we considered solving both dimensions simultaneously by using constraint networks such as those in Magritte[20] and ThingLab.[21] However, the use of priorities and alternatives complicates these techniques.)

Realignment. If prorating alone cannot resatisfy constraints, realignment attempts to move a window out of the way. In Figure 6, B may be required to have a certain width. The widening of A makes it impossible for that constraint to be satisfied in the current configuration. As a result, realignment tries to move B out of the way, as demonstrated in Figure 7, so that A can be made wider. Prorating is then performed once again.

Realignment may not be possible due to additional constraints. For example, if B is required to be at the top right of the screen, there is generally no reason to try to push it down out of the way. This depends upon priorities, of course. If B's presence constraint were at a higher

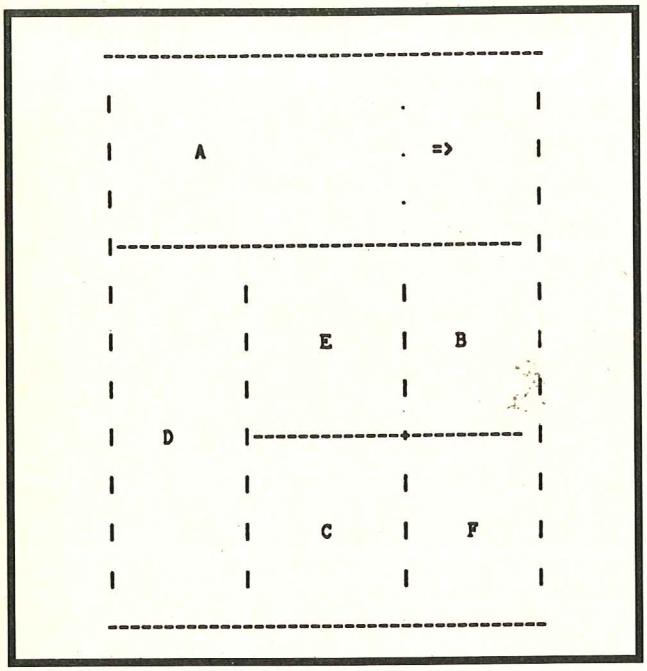


Figure 7. Squeezing B to make room for A.

priority than its requirement to be at the top of the screen, realignment would be tried. If its presence were at a lower priority, B would be closed instead. This would leave a hole which would probably be filled at the next prorating—in this case, E or F would expand into it.

A realignment or a move (closing and placement) may be conditionally tried even if it would break a new constraint. In this example, suppose that B is *not* required to be at the top right of the screen, and so could be squeezed down to make room for A. It could be that both B and F have high-priority minimum-height constraints, and cannot both fit below A. There are two possibilities:

1. Realignment can be tried again, this time moving F out of the way of B by squeezing C and F next to one another. It should be clear that this could be continued for quite a few iterations. The extent of iterated realignments and moves depends upon the level of configuration stability that has been set.
2. If additional realignment would exceed the level of configuration stability, either B or F would be closed and iconized.

Placement. Placement decides where to put a window when it is opened, when it is to be moved after being forcibly closed during realignment, or when the user has explicitly relocated it. Constraints (in priority order) that relate the window to others already on the screen are used to determine where to place it. Initially it will have either zero height or width. Prorating is then used to adjust its dimensions. Additional realignment and more prorating can then follow. We plan to use a rule-based system to determine the initial placement of a window. Some example rules follow:

1. If it must be adjacent to the left of some window, place it there with zero width and height equal to the height of the window.
2. If it must be adjacent to two windows at right angles to each other, place it in the corner along the edge of one of them.
3. If it must be adjacent to two currently adjacent windows, place it along the edge between them.
4. If it must be adjacent to two windows separated by one or more windows, and if there is an edge which connects the two windows, then place the new window along the edge, (and if there is more than one edge, use additional constraints to pick one).
5. If it must be located at a particular position (for example, when the user has explicitly opened or moved it there), place it along the existing window edge whose center is nearest to the required center of the window to be placed.

Often there will be more than one possible placement site. If a new window is required to be adjacent to one already on the screen, it can be placed on any side of it. In the absence of additional constraints, information retained from the previous prorating is used to make the decision. It would make more sense to place it above or below the window if there were greater slack for squeezing horizontally than vertically. Constraints on the windows to be squeezed out of the way in each case can also be used.

Even with placement realignments it may not be possible to satisfy all constraints, and in that case we simply place the window where it satisfies the highest priority constraints. We can now close the existing window that is responsible for the highest priority constraint broken by the addition of the new window, and apply placement to it. Depending upon the level of configuration stability, this can be continued as long as necessary.

Initial partitioning. The adjustment model works best when the changes to the screen are relatively minor. Major changes can occur for a number of reasons.

1. putting up the initial configuration
2. automatically reconstituting a number of windows when closing or shrinking a large window
3. significantly enlarging a window or opening a large window, forcing windows already on the screen to be closed.

When major changes occur we first use a separate *initial partitioning* phase. Initial partitioning produces a hierarchical layout approximately satisfying the constraints. Prorating and realignment are then applied to produce the final configuration.

When it appears that a major change will occur, we first approximately determine the windows to be displayed when we are finished. If we err, it is because we assume that slightly more, rather than fewer, windows will end up being displayed. (Any extra windows can easily be deleted at the end during the realignment phase.)

To initially partition, the screen is divided into two parts and the windows to be tiled are allocated between them. Each part is then divided into two subparts, and this continues until each part has one and only one window in it. This creates a strict hierarchical division of the screen.

At each cut, a number of decisions must be made:

- the cut direction (whether to split the partition horizontally or vertically)
- what size to make each partition

- how to allocate the windows between the two partitions
- which cut to make next

In making these decisions, the greatest emphasis is on satisfying adjacency and alignment constraints. If configuration stability has high priority, we also try to maintain existing adjacencies and alignments. If appearance stability has high priority, we assign windows to partitions containing their original location. High-priority location constraints also serve to place a window in the partition containing the specified location.

Size constraints are primarily used to determine partition size, but even here they are less important than adjacency and alignment. We can take this approach because prorating will be able to adjust window sizes after initial partitioning is completed.

Final remarks

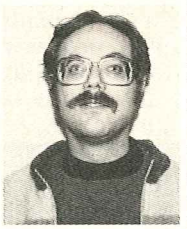
In this article we have described RTL/CRTL, a constraint-based nonhierarchical tiled window system we are now building at Siemens RTL. The prototype system, RTL/RTL (built on top of the Sapphire Window System on the PERQ), was built in the summer of 1984. An early version of the RTL/CRTL system is planned for release in the spring of 1986. Running on the Sun and MicroVAX workstations, it performs prorating based on size constraints, and also supports the manual tiling features of RTL/RTL. ■

Acknowledgments

We are grateful to Dave Emery for his useful comments and careful reading of this article, and to Guillermo Irisarri for emphasizing the importance of supporting features that are natural to overlapping windows. Keith Lantz provided many helpful comments in reviewing an earlier version of the article.

References

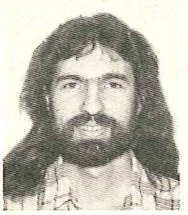
1. C.M. Eastman, "Preliminary Report on a System for General Space Planning," *Comm. ACM*, Vol. 15, No. 2, Feb. 1972.
2. C.E. Pfefferkorn, "A Heuristic Problem Solving Design System for Equipment or Furniture Layouts," *Comm. ACM*, Vol. 18, No. 5, May 1975.
3. C. Kingsley, *Earl: A Language for Integrated Circuit Design*, PhD dissertation, California Institute of Technology, Pasadena, 1981.
4. C. Kingsley, "A Hierarchical Error-Tolerant Compactor," *Proc. 21st Design Automation Conf.*, June 1984.
5. J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor, "Magic: A VLSI Layout System," *Proc. 21st Design Automation Conf.*, June 1984.
6. *User's Manual for Release 1 of the Information Technology Center Prototype Workstation*, Information Technology Center, Carnegie-Mellon University, Pittsburgh, Pa., 1984.
7. D.C. Smith, C. Irby, R. Kimball, and B. Verplank, "Designing the Star User Interface," *Byte*, Vol. 7, No. 4, Apr. 1982.
8. R. Levin, private communication, summer 1985.
9. S. McGregor, private communication, summer 1985.
10. K. Lantz and R. Rashid, "Virtual Terminal Management in a Multiple Process Environment," *Proc. 7th ACM Symposium on Operating Systems Principles*, Dec. 1979.
11. W. Teitelman, "A Tour Through Cedar," *IEEE Software*, Vol. 1, No. 2, Apr. 1984.
12. B. Myers, "The User Interface for Sapphire," *IEEE CG&A*, Vol. 4, No. 12, Dec. 1984.
13. L. Tesler, "The SmallTalk Environment," *Byte*, Vol. 6, No. 8, Aug. 1981.
14. C.J. Van Wyk, "A Graphics Typesetting Language," *Proc. ACM Sigplan Symposium on Text Manipulation*, June 1981.
15. J.A. Roach, "The Rectangle Placement Language," *Proc. 21st Design Automation Conf.*, June 1984.
16. J. Grason, "A Dual Linear Graph Representation for Space-Filling Location Problems of the Floor Plan Type," in *Emerging Methods of Environmental Design and Planning*, G.J. Moore, ed., The MIT Press, Cambridge, Mass., 1970.
17. P. Galle, "An Algorithm for Exhaustive Generation of Building Floor Plans," *Comm. ACM*, Vol. 24, No. 12, Dec. 1981.
18. K. Kozminski and E. Kinnen, "An Algorithm for Finding the Rectangular Dual of a Planar Graph for Use in Area Planning for VLSI Integrated Circuits," *Proc. 21st Design Automation Conf.*, June 1984.
19. S.M. Leinwand and T.T. Lai, "An Algorithm for Building Rectangular Floor Plans," *Proc. 21st Design Automation Conf.*, June 1984.
20. J. Gosling, *Algebraic Constraints*, PhD dissertation, Carnegie-Mellon University, Pittsburgh, Pa., May 1983.
21. A. Borning, *THINGLAB—A Constraint Oriented Simulation Laboratory*, PhD dissertation, Stanford University, Stanford, Calif., July 1979.



Ellis S. Cohen is a senior research scientist at Siemens Research and Technology Laboratories. His research interests include window managers, user interfaces, programming languages and environments, and distributed systems. Prior to joining Siemens, he taught and did research at Brandeis University, Carnegie-Mellon University, and the University of Newcastle upon Tyne. At Carnegie-Mellon, he was part of the Hydra project. He has also worked at a day care center, been a venereal disease counselor, and run a salad restaurant.

Cohen received a PhD in computer science from Carnegie-Mellon in 1976. He is a member of the IEEE and the ACM.

Cohen can be contacted at Siemens Research & Technology Laboratories, 105 College Rd. East, Princeton, NJ 08540; (609) 734-6524.



Edward T. Smith is a research associate in the Computer Science Department at Carnegie-Mellon University. His teaching and research interests are in network operating systems, graphics, user interface management systems, and window managers. He received a BS in computer science from the University of Oregon in 1977 and an MS and a PhD in computer science from the University of Rochester in 1979 and 1982, respectively.

Smith can be contacted at the Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA 15213.

Lee A. Iverson works in the Computer Vision and Robotics Laboratory at McGill University, where he studies computer models for visual motion processing. His work in prototyping RTL/RTL was done on an internship with Siemens during the summer of 1984. He received a BSE (magna cum laude) in electrical engineering and computer science from Princeton University in 1984. He is now working towards a PhD at McGill.

Iverson can be contacted at the Computer Vision and Graphics Laboratory, Dept. of Electrical Engineering, McGill University, Montreal, PQ, Canada.